**ФЕДЕРАЛЬНОЕ АГЕНТСТВО СВЯЗИ**

**ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ**

**«САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ТЕЛЕКОММУНИКАЦИЙ ИМ. ПРОФ. М.А. БОНЧ-БРУЕВИЧА»**

**(СПбГУТ)**

**Кафедра безопасности информационных систем**

**ОТЧЁТ**

по практической работе № 6 на тему:   
**«Разработка программ моделирования работы стека и очереди.»**

по дисциплине «Алгоритмы и структуры данных»

Выполнил: студент группы ИСТ-312, Кандиков М.В.

«14» Октября 2024 г. \_\_\_\_\_\_\_\_\_\_\_/М.В. Кандиков /

Принял: к.ф.-м.н., доцент, И.А. Моисеев

«15» Октября 2024 г. \_\_\_\_\_\_\_\_\_\_\_/ И.А. Моисеев /

**Содержание**

**1. Титульный лист 1**

**2. Содержание 2**

**3. Основная часть 3**

3.1. Цель работы 3

3.2. Теоретическая часть 3

3.2. Практическая часть 4

3.3. Выводы 5

**4. Приложение 6**

**Основная часть**

**Цель работы:**

Целью данной работы является изучение и практическая реализация основных структур данных: стек и очередь, а также понимание их внутренней работы на примерах. Работа направлена на закрепление навыков по созданию и применению линейных списков, а также на моделирование реальных процессов, таких как обработка данных в стеке и симуляция работы конвейера процессора с использованием очереди.

**Теоретическая часть:**  
 Стек — это структура данных, которая работает по принципу LIFO (Last In, First Out), что означает, что последним помещённый элемент извлекается первым. Основные операции для стека включают:

1. push: добавление элемента в стек.
2. pop: удаление верхнего элемента стека.
3. peek: просмотр верхнего элемента стека без его удаления.

Использование стека актуально для задач, где требуется "отложенное выполнение", например, при работе с обратным порядком данных (например, когда нужно выводить элементы в обратной последовательности).  
Очередь — это структура данных, работающая по принципу FIFO (First In, First Out), что означает, что первым добавленный элемент извлекается первым. Основные операции для очереди включают:

1. enqueue: добавление элемента в очередь.
2. dequeue: удаление первого элемента из очереди.

Очереди широко используются в моделировании различных процессов, например, в операционных системах при организации выполнения команд с разделением на стадии (конвейерная обработка).

**Практическая часть:**

**Задача 1. Реализация стека для обратного вывода слов предложения ВОДИТЬ ПО СЛОВАМ (ВЫВОДИТЬ В СТОЛБИК)**

Требуется создать стек и занести в него слова предложения:  
«Разработать алгоритм и программу умножения двух целочисленных матриц». Каждое слово предложения нужно помещать в стек в обратном порядке (с конца предложения). Далее из стека выполнить последовательный вывод слов в прямом порядке (по одному слову за раз).

**План выполнения задачи:**

* Занести в стек слова предложения в обратном порядке.
* Произвести вывод слов, сохраняя их последовательность.

Результат выполнения задачи:



Рисунок 1. Результат выполнения задачи №1.

**Задача 2.** Моделирование работы конвейера процессора с использованием очереди. ВЫВОД ПОСТРОЧНО

Требуется смоделировать работу процессора с конвейерной архитектурой, состоящего из пяти ступеней обработки команд (C1–C5). Каждая команда должна проходить через все пять стадий: загрузка, декодирование, получение операндов, выполнение, запись результата.  
Необходимо реализовать очередь, которая будет моделировать последовательную обработку 10 команд через конвейер.

**План выполнения задачи:**

* Реализовать модель конвейера на основе очереди, где каждая команда последовательно проходит через 5 ступеней обработки.
* Пропустить через конвейер 10 команд.
* После завершения обработки каждой команды выводить её на экран.

Результат выполнения задачи:

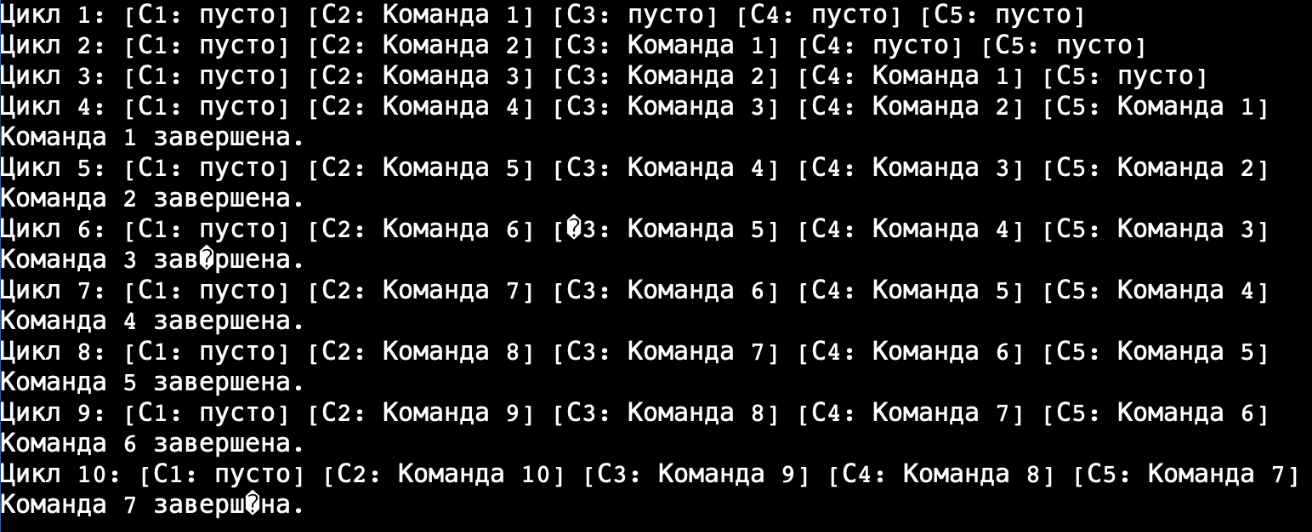


Рисунок 2. Результат для выполненя задачи №2.

**Выводы:**

Работа со стеком показала, как эффективно использовать эту структуру данных для обработки данных в обратном порядке. Мы реализовали стек, который позволил нам помещать слова предложения в обратном порядке, а затем выводить их в прямом порядке. Это продемонстрировало принцип LIFO (последний вошел — первый вышел), который часто используется в алгоритмах, связанных с парсингом, отменой операций и рекурсивными вызовами.

Вторая часть работы, посвященная очереди, показала ее полезность в моделировании конвейерной обработки команд. Мы использовали очередь для представления процесса, через который последовательно проходят команды. Это отражает принцип FIFO (первый вошел — первый вышел) и позволяет понять, как команды обрабатываются в порядке их поступления. Модель конвейера, реализованная с помощью очереди, иллюстрирует, как эффективно управлять последовательной обработкой данных, что важно для повышения производительности процессоров.

Анализируя эффективность структур данных, мы поняли, что стек и очередь имеют свои сильные стороны в зависимости от задач. Стек подходит для обработки данных в обратном порядке, тогда как очередь незаменима в задачах, где важен порядок выполнения. Оба инструмента имеют широкое применение в различных сферах программирования и помогают эффективно выбирать подходящие структуры данных для решения конкретных задач.

В целом, данная работа подтвердила, что выбор правильной структуры данных является ключевым моментом для оптимального решения задач. Стек и очередь, как базовые структуры, предоставляют различные возможности для работы с данными и являются важными инструментами в арсенале программиста для реализации эффективных алгоритмов.

**Приложение**

Задача №1.

#include <iostream>

#include <string>

#include <sstream>

using namespace std;

// Структура узла для стека

struct Node {

    string data;  // Слово

    Node\* next;   // Указатель на следующий узел

};

// Класс для стека

class Stack {

private:

    Node\* top;  // Указатель на верхний элемент стека

public:

    Stack() {

        top = nullptr;  // Инициализация пустого стека

    }

    // Функция для добавления элемента в стек

    void push(string word) {

        Node\* newNode = new Node();  // Создаем новый узел

        newNode->data = word;        // Записываем слово

        newNode->next = top;         // Связываем новый узел с текущим верхним элементом

        top = newNode;               // Новый узел становится верхним элементом

    }

    // Функция для удаления элемента из стека

    string pop() {

        if (top == nullptr) {

            return "";  // Если стек пуст, возвращаем пустую строку

        }

        string word = top->data;     // Получаем верхний элемент

        Node\* temp = top;            // Временный указатель для удаления узла

        top = top->next;             // Перемещаем указатель на следующий элемент

        delete temp;                 // Удаляем старый верхний узел

        return word;

    }

    // Функция для проверки, пуст ли стек

    bool isEmpty() {

        return top == nullptr;

    }

};

int main() {

    // Исходное предложение

    string sentence = "Разработать алгоритм и программу умножения двух целочисленных матриц.";

    // Создаем стек

    Stack stack;

    // Разбиваем предложение на слова

    stringstream ss(sentence);

    string word;

    while (ss >> word) {

        stack.push(word);  // Заносим каждое слово в стек

    }

    // Выводим слова в прямом порядке

    cout << "Предложение в прямом порядке:" << endl;

    while (!stack.isEmpty()) {

        cout << stack.pop() << " ";  // Извлекаем слова из стека

    }

    cout << endl;

    return 0;

}

Задача №2.

#include <iostream>

#include <queue>

#include <string>

using namespace std;

int main() {

    queue<string> pipelineStages[5];  // Очереди для каждой стадии C1, C2, C3, C4, C5

    // Создаем массив команд

    string commands[] = {"Команда 1", "Команда 2", "Команда 3", "Команда 4", "Команда 5",

                         "Команда 6", "Команда 7", "Команда 8", "Команда 9", "Команда 10"};

    int commandCount = 10;  // Количество команд

    // Процесс моделирования конвейера

    for (int cycle = 1; cycle <= commandCount + 4; ++cycle) {

        // Перемещаем команды по конвейеру

        for (int stage = 4; stage > 0; --stage) {

            if (!pipelineStages[stage-1].empty()) {

                string command = pipelineStages[stage-1].front();

                pipelineStages[stage-1].pop();

                pipelineStages[stage].push(command);

            }

        }

        // На первой стадии вызываем новую команду, если она существует

        if (cycle <= commandCount) {

            pipelineStages[0].push(commands[cycle - 1]);

        }

        // Если команда дошла до пятой стадии, выводим её

        if (!pipelineStages[4].empty()) {

            cout << pipelineStages[4].front() << " завершена на цикле " << cycle << endl;

            pipelineStages[4].pop();

        }

    }

    return 0;

}