Минобрнауки РФ

ФГБОУ ВО Тверской Государственный Технический Университет

Кафедра  “Программное Обеспечение”

Лабораторная работа N6

По дисциплине “Структуры и алгоритмы обработки данных”

Выполнил: студент группы Б.ПИН.РИС-22.06

Иванов Алексей Михайлович

Проверил: Мальков Александр Анатольевич

г. Тверь, 2023

Оглавление

[**Часть 1. Постановка задачи.** 3](#_Toc145406338)

[**Часть 2. Алгоритм решения.** 3](#_Toc145406339)

[**Часть 3. Оценка временной эффективности в терминах О-функций.** 4](#_Toc145406340)

[**Часть 4. Программный код.** 5](#_Toc145406341)

[**Часть 5. Тестирование и результаты работы.** 9](#_Toc145406342)

# **Часть 1. Постановка задачи.**

Стек.

Методы класса "Стек":

- добавление

- чтение

- проверка на пустоту.

# **Часть 2. Алгоритм решения.**

# Стэк – тип данных, реализующий схему LIFO – Last-in-first-out – последний вошел первый вышел.

Для его реализации необходим класс, представляющий узел – то есть хранящий в себе значение поля и ссылку на следующий элемент. Тогда в объекте стэка нам нужно лишь хранить ссылку на верхний узел, и при добавлении ставить его как следующий для нового узла, а ссылку в объекте перемещать на новый узел. Для чтения нам нужно лишь вернуть значение верхнего узла, если это чтение с удалением, то еще дополнительно сместить ссылку в объекте на следующий узел. Проверка на пустоту - это лишь определение, является ли верхний узел стэка пустой ссылкой.

# **Часть 3. Оценка временной эффективности в терминах О-функций.**

Составим сводную таблицу эффективности операций.

|  |  |
| --- | --- |
|  | Эффективность (n длина массива) |
| Вставка | O(1) |
| Удаление верхнего элемента | O(1) |
| Получение верхнего элемента | O(1) |
| Проверка на пустоту | O(1)) |
| Поиск элемента по значению | до O(n) |
| Удаление элемента по значению | O(n) |

# Вставка, удаление, получение элемента и проверка на пустоту требуют лишь работы со ссылкой на верхний элемент, что выполняется на константное время.

А вот для поиска и удаления по значению требуется последовательно пройтись по всем элементам стэка. Это потребует в среднем n/2 операций сравнения, что приводит к сложности O(n).

# **Часть 4. Программный код.**

Итоговая библиотека классов имеет следующий вид:

/// <summary>

/// Класс, отображающий узел в связном списке

/// </summary>

/// <typeparam name="T">Тип данных, хранимый в узле</typeparam>

public class MyNode<T>

{

/// <summary>

/// Значение, хранимое в узле

/// </summary>

protected T Value;

/// <summary>

/// Ссылка на следующий узел

/// </summary>

protected MyNode<T> Next;

/// <summary>

/// Создает новый узел с данным значением

/// </summary>

/// <param name="value"></param>

public MyNode(T value)

{

Value = value;

Next = null;

}

/// <summary>

/// Создает пустой узел

/// </summary>

public MyNode()

{

Value = default(T);

Next = null;

}

/// <summary>

/// Возвращает значение в текущем узле

/// </summary>

/// <returns>Значение в текущем узле</returns>

public T GetValue()

{

return this.Value;

}

/// <summary>

/// Возвращает ссылку на следующий элемент

/// </summary>

/// <returns>Ссылка на следующий элемент</returns>

public MyNode<T> GetNext()

{

return this.Next;

}

/// <summary>

/// Назначает ссылку на следующий элемент

/// </summary>

/// <returns>Ссылка на следующий элемент</returns>

public void SetNext(MyNode<T> newNode)

{

this.Next = newNode;

}

/// <summary>

/// Реализует структуру данных стэк

/// </summary>

/// <typeparam name="T">Тип данных, хранимый в экземпляре стека</typeparam>

public class MyStack<T>

{

/// <summary>

/// Ссылка на узел, являющийся верхушкой стека

/// </summary>

private MyNode<T> Current;

/// <summary>

/// Количество элементов в стеке

/// </summary>

private int Length = 0;

/// <summary>

/// Создает новый пустой стэк

/// </summary>

public MyStack()

{

}

/// <summary>

/// Создает новый стэк с единственным элементом

/// </summary>

/// <param name="elem">Элемент</param>

public MyStack(T elem)

{

Current = new MyNode<T>(elem);

Length++;

}

/// <summary>

/// Возвращает количество элементов в стеке

/// </summary>

/// <returns>Количество элементов в стеке</returns>

public int GetLength()

{

return this.Length;

}

/// <summary>

/// Возвращает верхний элемент стека

/// </summary>

/// <returns>Верхний элемент стека</returns>

public T GetCurrent()

{

return this.Current.GetValue();

}

/// <summary>

/// Вставляет элемент на верхушку стека

/// </summary>

/// <param name="val">Вставляемый элемент</param>

public void InsertElem(T val)

{

MyNode<T> newNode = new MyNode<T>(val);

newNode.SetNext(Current);

Current = newNode;

Length++;

}

/// <summary>

/// Возвращает верхний элемент стека, удаляя его

/// </summary>

/// <returns>Верхний элемент стека</returns>

public T PopElem()

{

MyNode<T> oldNode = Current;

MyNode<T> newNode = Current.GetNext();

Current = newNode;

Length--;

return oldNode.GetValue();

}

/// <summary>

/// Проверяет стек на пустоту

/// </summary>

/// <returns>

/// True: стек пуст и не содержит элементов

/// False: все остальные случаи

/// </returns>

public bool IsEmpty()

{

return this.Current == null;

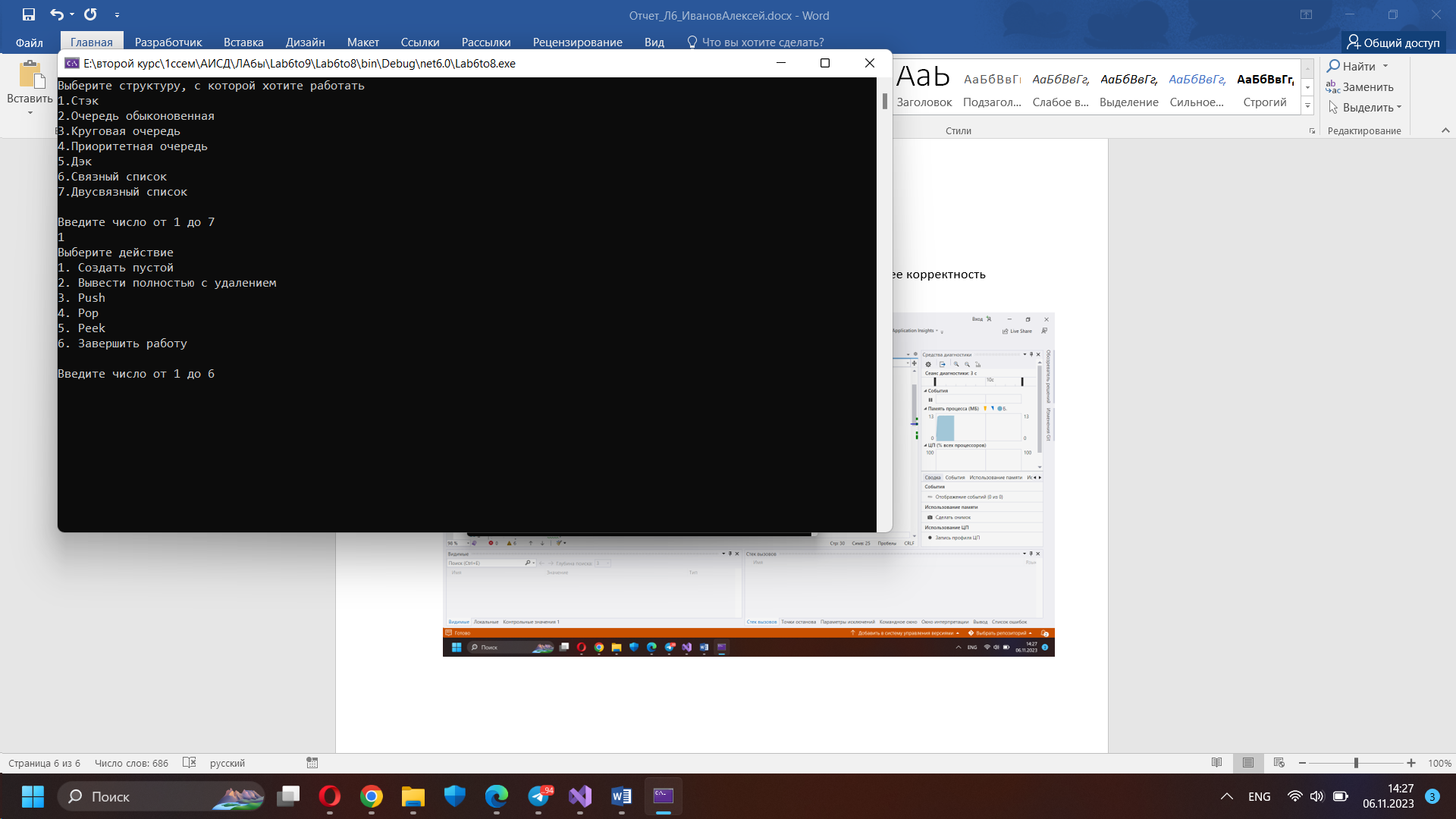
}

}

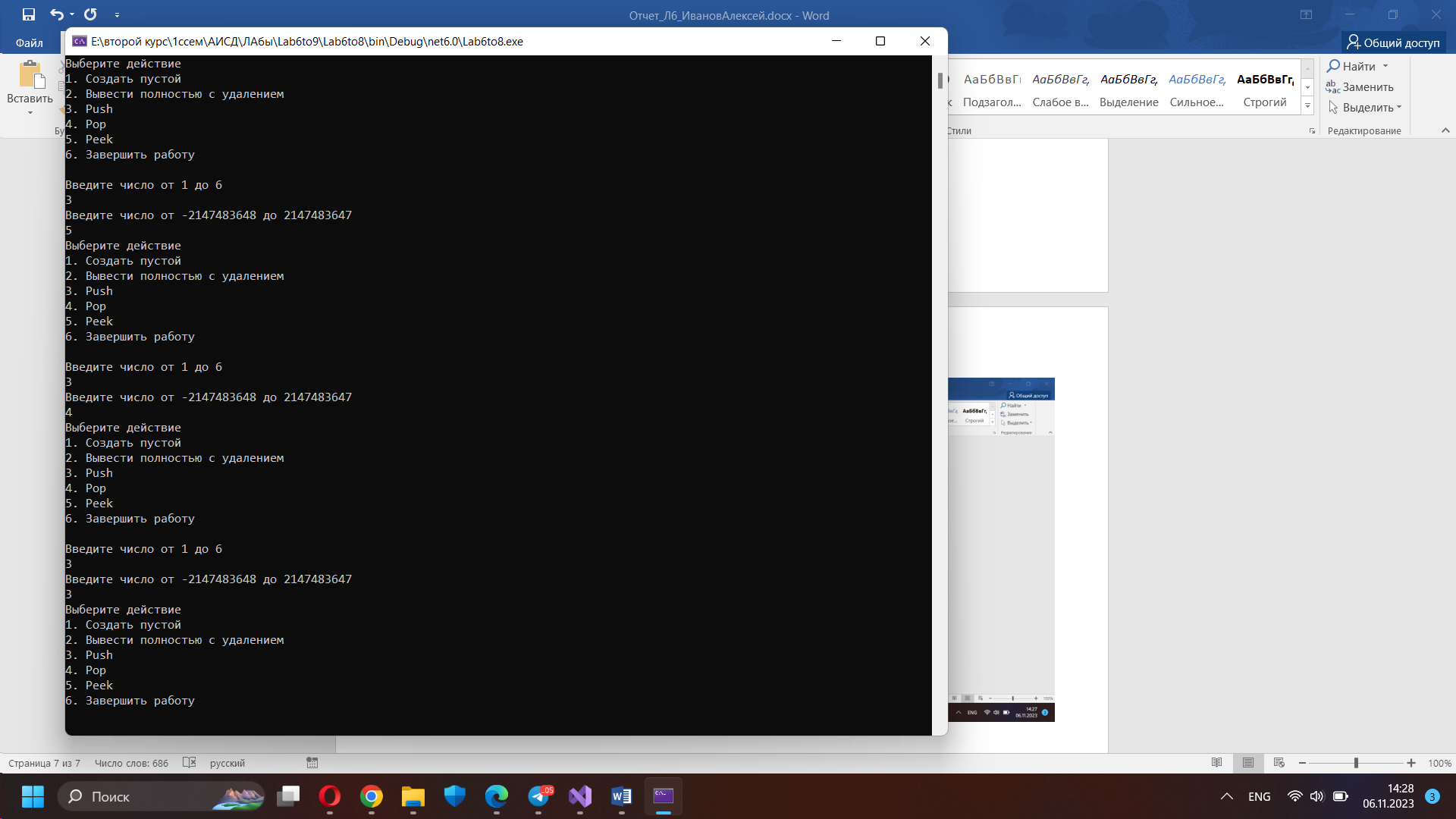
**Часть 5. Тестирование и результаты работы.**

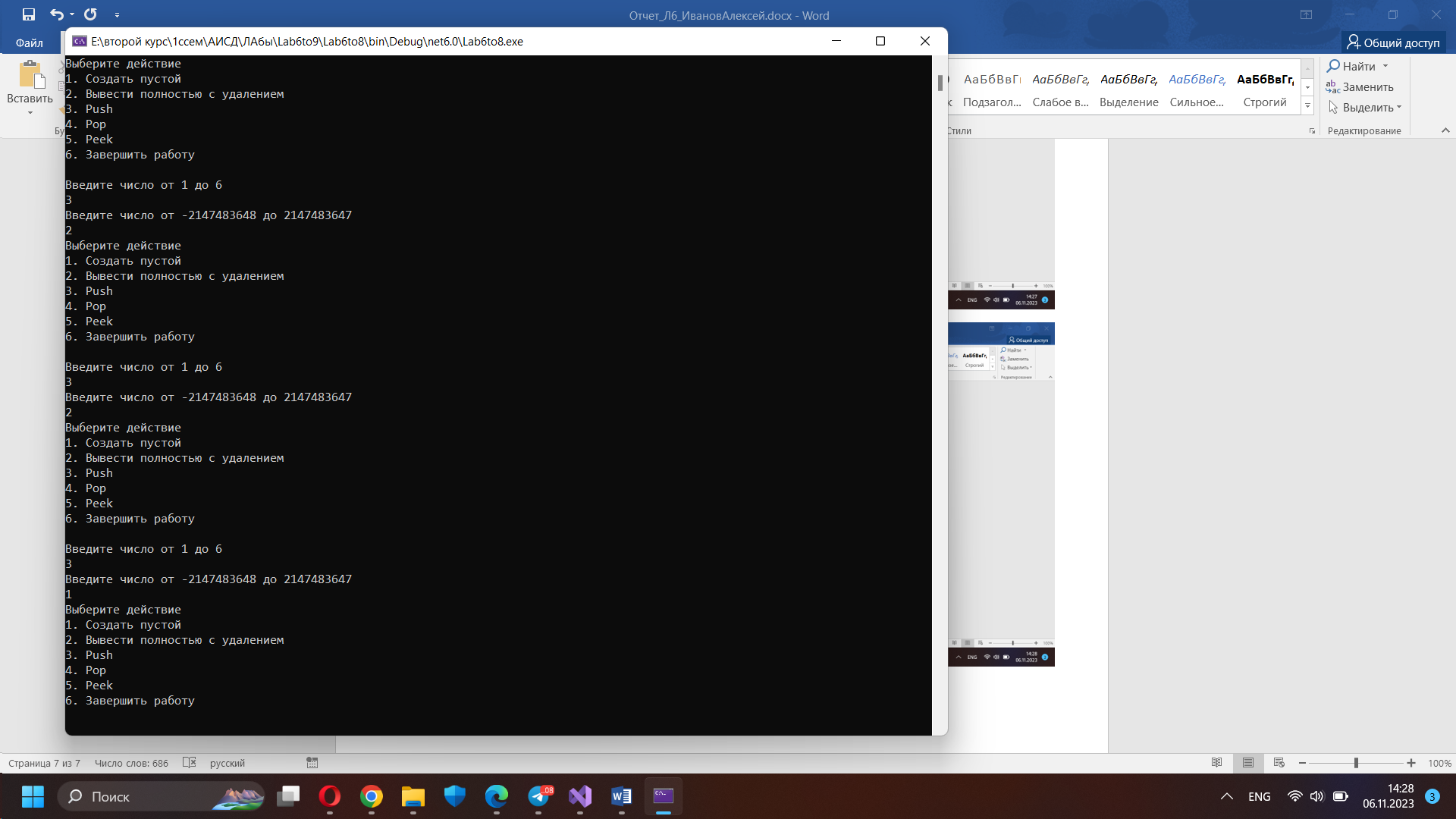
Для тестирования было разработано консольное приложение, проверяющее корректность алгоритмов.

Создадим пустой новый стэк.



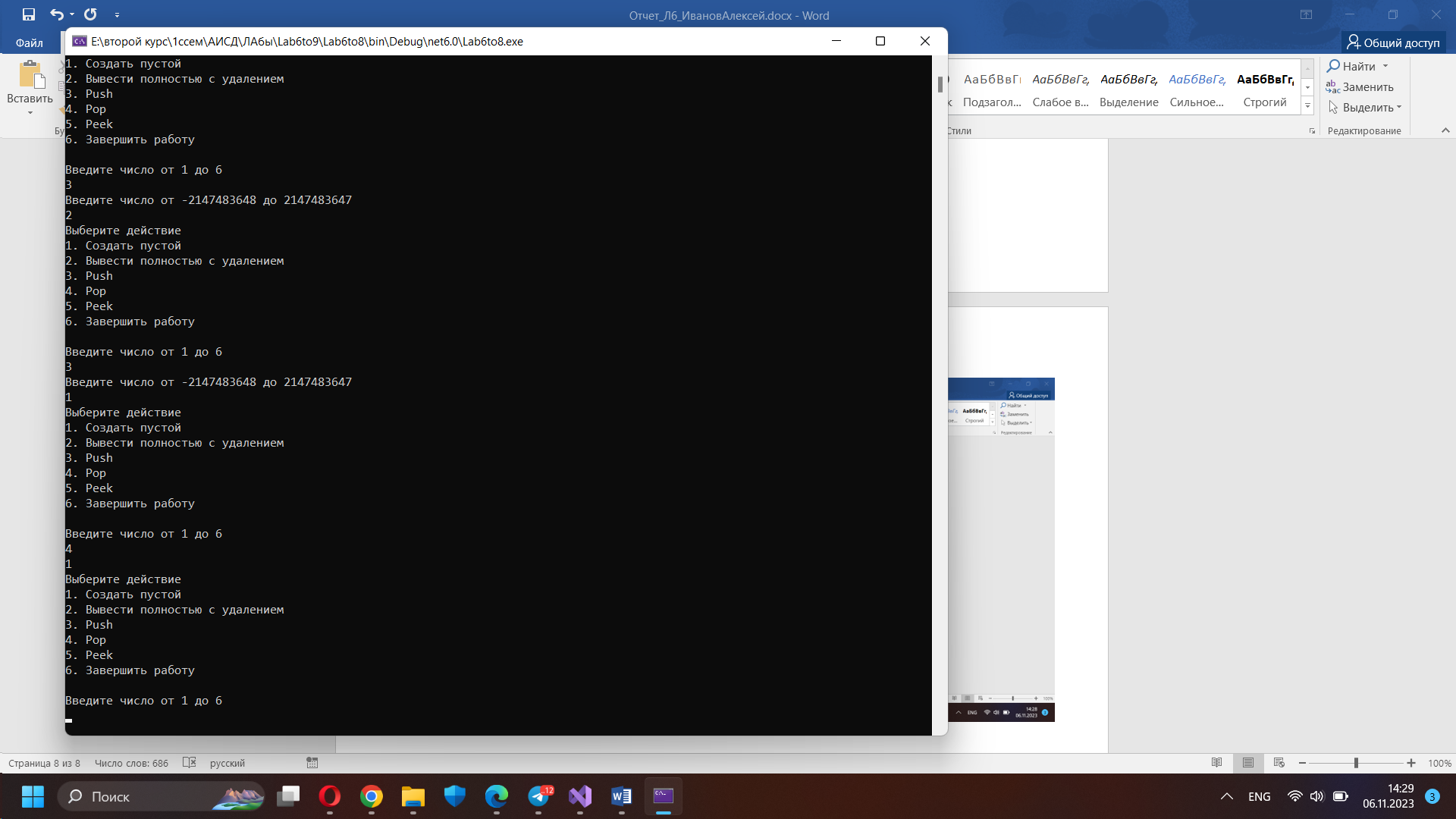
Последовательно будем вставлять вверх элементы 5, 4, 3, 2, 2, 1.



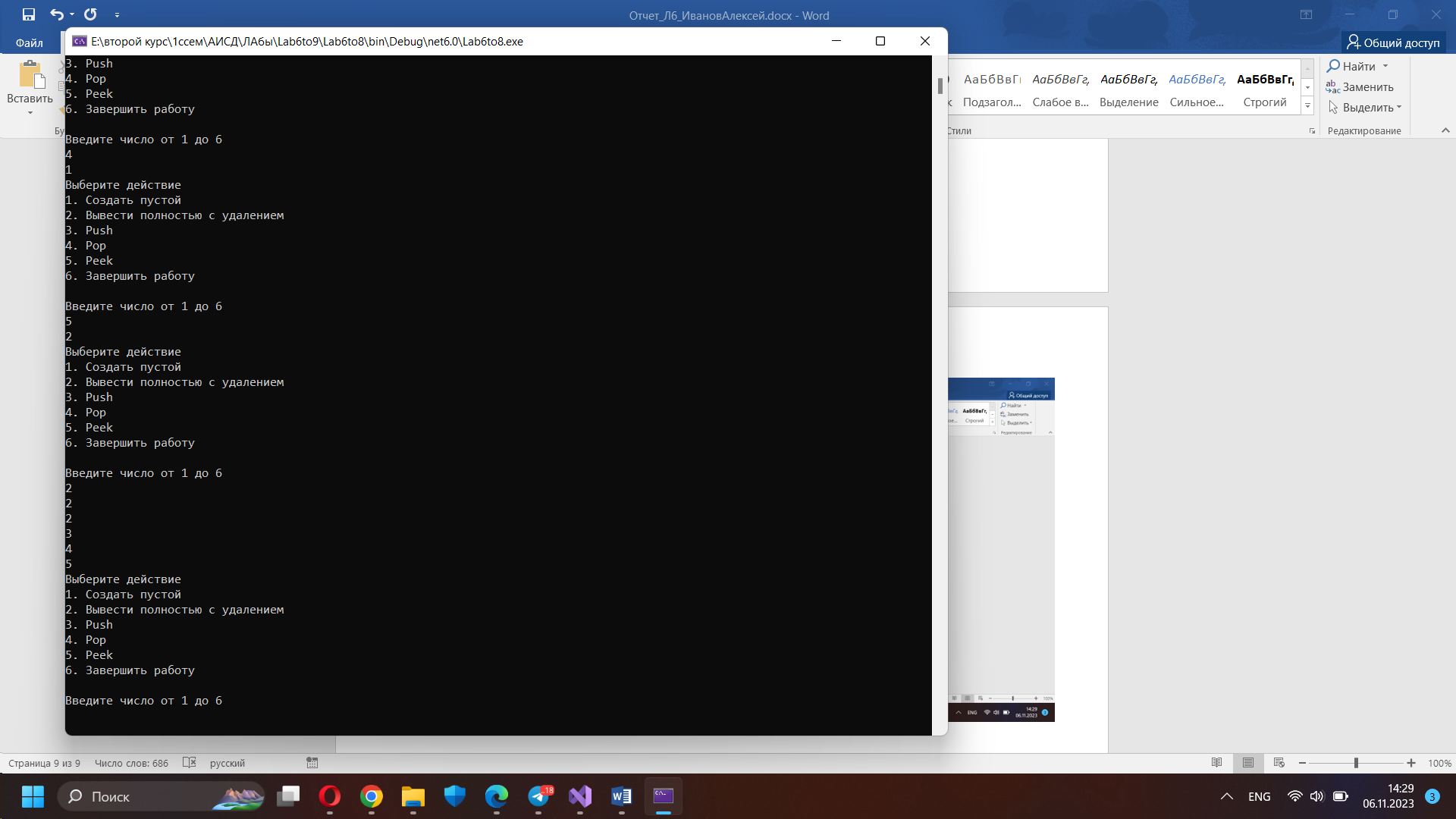


Удалим первый элемент, выводя его на экран. Ожидается вывод 1.

При этом ожидается, что в стэке останутся элементы 2, 2, 3, 4, 5.



Проверим чтение первого элемента. Должно быть показано число 2, при этом стэк не должен измениться.



Теперь осуществим полный вывод содержимого стэка с удалением. Ожидается вывод 2, 2, 3, 4, 5.

