Минобрнауки РФ

ФГБОУ ВО Тверской Государственный Технический Университет

Кафедра  “Программное Обеспечение”

Лабораторная работа N7

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.**

7. Очереди:

- "простая"

- кольцевая

- с приоритетами.

Методы класса "Очередь":

- добавление

- чтение

- проверка на пустоту.

# **Часть 2. Алгоритм решения.**

# Очередь – тип данных, реализующий схему FIFO – first-in-first-out – первый вошел, первый вышел.

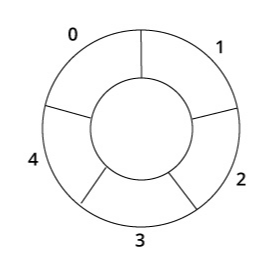
**Простая очередь** может быть реализована следующим образом:

Дан узел – класс, хранящий какую-то информацию и ссылку на следующий узел.

Хранится ссылка на первый и на последний узел в очереди (на последний хранить необязательно, но лучше хранить одну ссылку, которая почти ничего не весит, чем каждый раз при вставке полностью проходить очередь). При добавлении нового элемента, он устанавливается как следующий для последнего узла, ссылка на последний узел в очереди смещается на этот новый последний узел. Для чтения считывается значение первого узла, для удаления первого узла ссылке на первый узел присваивается ссылка на узел, связанный с первым как следующий.

Проверку на пустоту можно осуществить либо сравнивая ссылки с null, либо введя поле Count и сравнивая его с 0.

**Кольцевая очередь** – способ реализации очереди на основе массива. Массив представляется так, будто он закольцован, что позволяет избежать проблемы, когда элементы очереди «съежают» при реализации очереди на основе массива, без излишнего количества перемещений.



Тогда мы будем хранить в объекте значения первого и последнего индекса, и при добавлении элемента смещать последний, беря от получившегося значения остаток от деления на длину буффера, и на получившийся индекс добавлять элемент. Для чтения первого элемента действия те же, только происходят они со значением первого индекса. Также примем, что при отсутствии элементов в очереди, эти индексы будут равны -1 (то есть в изначальном состоянии и при удалении последнего элемента), тогда проверка на пустоту будет очевидна.

**Приоритетная очередь** – надстройка над обычной очередью, только за тем исключением, что элементы будут вставляться не на последнее место, а согласно их приоритету (от меньшего к большему), и в случае равенства приоритетов, элемент ставится после всех элементов с равным приоритетом.

Для ее реализации также нужна модифицированная версия узла, хранящая помимо значения его приоритет.

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

Составим сводную таблицу эффективности для очередей вышеописанных видов. На пересечении строк с операциями и столбцов с типом очередей будет сложность операции.

|  |  |  |  |
| --- | --- | --- | --- |
|  | Простая очередь | Кольцевая | Приоритетная |
| Вставка в конец | O(1) | O(1) | Отсутствует,  Вставка в принципе O(n) |
| Получение последнего | O(1) | O(1) | O(1) |
| Удаление последнего | O(1) | O(1) | O(1) |
| Проверка на пустоту | O(1) | O(1) | O(1) |
| Получение по критерию | О(n) (по значению) | О(n) (по значению) | О(1) (по верхнему приоритету)  O(n) (по значению) |
| Удаление по критерию | O(n) (по значению) | O(n) (по значению) | О(1) (по нижнему приоритету)  O(n) (по значению) |

# Обычная и кольцевая очереди одинаковы по сложности, ведь это лишь разные реализации одного и того же типа данных. Операции вставки, получения, удаления и проверки на пустоту требуют единичных действий с полями класса, что означает сложность О(1).

Для получения либо удаления по значению требуется последовательно перебирать элементы до необходимого, в среднем это n/2 сравнений, а значит сложность O(n).

Вставка в приоритетную очередь требует последовательного сравнения приоритетов элементов до подходящего, в среднем n/2, а значит сложность O(n). Зато работа с элементами с высшим приоритетам это О(1), ведь они по определению хранятся либо в конце либо в начале. Впрочем, при поиске элемента по значению все так же необходим последовательный перебор, сложностью О(n/2).

# **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>

/// Класс, отобрающий узел, хранящий значение, приоритет

/// и ссылку на следующий узел

/// </summary>

/// <typeparam name="T">Тип данных значение</typeparam>

/// <typeparam name="K">Тип данных приоритета</typeparam>

public class MyPriorityNode<T, K>

{

/// <summary>

/// Значение, хранимое в узле

/// </summary>

protected T Value;

/// <summary>

/// Приоритет, хранимы в узле

/// </summary>

protected K Priority;

/// <summary>

/// Ссылка на следующий узел

/// </summary>

protected MyPriorityNode<T, K> Next;

/// <summary>

/// Создает новый узел с данным значением

/// </summary>

public MyPriorityNode(T value, K priority)

{

Value = value;

Priority = priority;

Next = null;

}

/// <summary>

/// Создает пустой узел

/// </summary>

public MyPriorityNode()

{

Value = default(T);

Priority = default(K);

Next = null;

}

/// <summary>

/// Возвращает значение в текущем узле

/// </summary>

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

public T GetValue()

{

return this.Value;

}/// <summary>

/// Возвращает приоритет в текущем узле

/// </summary>

/// <returns>Приоритет в текущем узле</returns>

public K GetPriority()

{

return this.Priority;

}

/// <summary>

/// Возвращает ссылку на следующий элемент

/// </summary>

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

public MyPriorityNode<T, K> GetNext()

{

return this.Next;

}

/// <summary>

/// Назначает ссылку на следующий элемент

/// </summary>

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

public void SetNext(MyPriorityNode<T, K> newNode)

{

this.Next = newNode;

}

}

/// <summary>

/// Реализует структуру данных очередь

/// </summary>

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

public class MyQueue<T>

{

/// <summary>

/// Ссылка на первый элемент в очереди

/// </summary>

protected MyNode<T> first;

/// <summary>

/// Ссылка на последний элемент в очереди

/// </summary>

protected MyNode<T> last;

/// <summary>

/// Количество элементов о очереди

/// </summary>

protected int Length = 0;

/// <summary>

/// Создает новую пустую очередь

/// </summary>

public MyQueue()

{

last = first;

}

/// <summary>

/// Создает новую очередь с единственным элементом

/// </summary>

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

public MyQueue(T elem)

{

first = new MyNode<T>(elem);

last = first;

Length++;

}

/// <summary>

/// Возвращает количество элементов в очереди

/// </summary>

/// <returns>Количество элементов в очереди</returns>

public int GetLength()

{

return this.Length;

}

/// <summary>

/// Возвращает первый элемент в очереди

/// </summary>

/// <returns>Первый элемент в очереди</returns>

public T GetCurrent()

{

return this.first.GetValue();

}

/// <summary>

/// Вставляет элемент в начало очереди

/// </summary>

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

public virtual void InsertElem(T val)

{

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

if (Length == 0)

{

this.first = newNode;

Length++;

return;

}

if (Length == 1)

{

this.last = newNode;

Length++;

this.first.SetNext(last);

return;

}

this.last.SetNext(newNode);

this.last = newNode;

Length++;

}

/// <summary>

/// Возвращает первый элемент в очереди, удаляя его

/// </summary>

/// <returns>Первый элемент в очереди</returns>

public T PopElem()

{

MyNode<T> oldNode = this.first;

MyNode<T> newNode = this.first.GetNext();

this.first = newNode;

Length--;

return oldNode.GetValue();

}

/// <summary>

/// Проверяет очередь на пустоту

/// </summary>

/// <returns>

/// True: очередь пуста и не содержит элементов

/// False: все остальные случаи

/// </returns>

public bool IsEmpty()

{

return Length == 0;

}

}

/// <summary>

/// Реализует структуру данных приоритетная очередь

/// </summary>

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

public class MyPriorityQueue<T, K>

where K : IComparable<K>

{

/// <summary>

/// Ссылка на первый элемент в очереди

/// </summary>

protected MyPriorityNode<T, K> first;

/// <summary>

/// Ссылка на последний элемент в очереди

/// </summary>

protected MyPriorityNode<T, K> last;

/// <summary>

/// Количество элементов о очереди

/// </summary>

protected int Length = 0;

/// <summary>

/// Создает новую пустую приоритетную очередь, в которой элементы

/// распределяются по возрастанию

/// </summary>

public MyPriorityQueue() { }

/// <summary>

/// Создает новую приоритетную очередь с единственным элементом

/// </summary>

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

public MyPriorityQueue(T elem, K priority) {

MyPriorityNode<T, K> first = new(elem, priority);

this.first = first;

}

/// <summary>

/// Вставляет элемент в очередь

/// </summary>

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

public void InsertElem(T val, K priority)

{

MyPriorityNode<T, K> newNode = new MyPriorityNode<T, K>(val, priority);

if (Length == 0)

{

this.first = newNode;

Length++;

return;

}

if (Length == 1)

{

if (this.first.GetPriority().CompareTo(priority) <= 0)

{

this.last = newNode;

} else

{

this.last = first;

first = newNode;

}

first.SetNext(last);

Length++;

return;

}

Length++;

//insert in 1st place

if (this.first.GetPriority().CompareTo(priority) > 0)

{

MyPriorityNode<T, K> temp = first;

first = newNode;

first.SetNext(temp);

return;

}

//insert in midlle

MyPriorityNode<T, K> curNode = this.first;

while (curNode.GetNext() != null)

{

if (curNode.GetNext().GetPriority().CompareTo(priority) > 0)

{

MyPriorityNode<T, K> next = curNode.GetNext();

curNode.SetNext(newNode);

newNode.SetNext(next);

return;

}

curNode = curNode.GetNext();

}

//insert in last place

last.SetNext(newNode);

last = newNode;

}

/// <summary>

/// Возвращает количество элементов в очереди

/// </summary>

/// <returns>Количество элементов в очереди</returns>

public int GetLength()

{

return this.Length;

}

/// <summary>

/// Возвращает первый элемент в очереди

/// </summary>

/// <returns>Первый элемент в очереди</returns>

public T GetCurrent()

{

return this.first.GetValue();

}

/// <summary>

/// Возвращает первый элемент в очереди, удаляя его

/// </summary>

/// <returns>Первый элемент в очереди</returns>

public T PopElem()

{

MyPriorityNode<T, K> oldNode = this.first;

MyPriorityNode<T, K> newNode = this.first.GetNext();

this.first = newNode;

Length--;

return oldNode.GetValue();

}

/// <summary>

/// Проверяет очередь на пустоту

/// </summary>

/// <returns>

/// True: очередь пуста и не содержит элементов

/// False: все остальные случаи

/// </returns>

public bool IsEmpty()

{

return Length == 0;

}

}

/// <summary>

/// Реализует структуру данных кольцевая очередь

/// </summary>

/// <typeparam name="T">Тип данных, хранящийся в очереди</typeparam>

public class MyCyclingQueue<T>

{

/// <summary>

/// Индекс первого элемента в очереди

/// </summary>

private int first;

/// <summary>

/// Индекс последнего элемента в очереди

/// </summary>

private int last;

/// <summary>

/// Размер буффера

/// </summary>

private int size;

/// <summary>

/// Массив-буффер для очереди

/// </summary>

private T[] buffer;

/// <summary>

/// Создает кольцевую очередь с буффером указанного размера

/// </summary>

/// <param name="size">Размер буффера</param>

public MyCyclingQueue(int size)

{

this.size = size;

buffer = new T[size];

first = -1;

last = -1;

}

/// <summary>

/// Возвращает первый элемент очереди

/// </summary>

/// <returns>Первый элемент очереди</returns>

public T GetCurrent()

{

return buffer[first];

}

/// <summary>

/// Возвращает первый элемент очереди, удаляя его

/// </summary>

/// <returns>Первый элемент очереди</returns>

public T PopElem()

{

T elem = buffer[first];

if (first == last)

{

first = -1;

last = -1;

}

else

{

first = (first + 1) % size;

}

return elem;

}

/// <summary>

/// Вставляет элемент в конец очереди

/// </summary>

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

public void InsertElem(T elem)

{

last = (last + 1) % size;

if (IsEmpty())

first = 0;

buffer[last] = elem;

}

/// <summary>

/// Проверяет очередь на пустоту

/// </summary>

/// <returns>

/// True: очередь пуста и не содержит элементов

/// False: все остальные случаи

/// </returns>

public bool IsEmpty()

{

return first == -1;

}

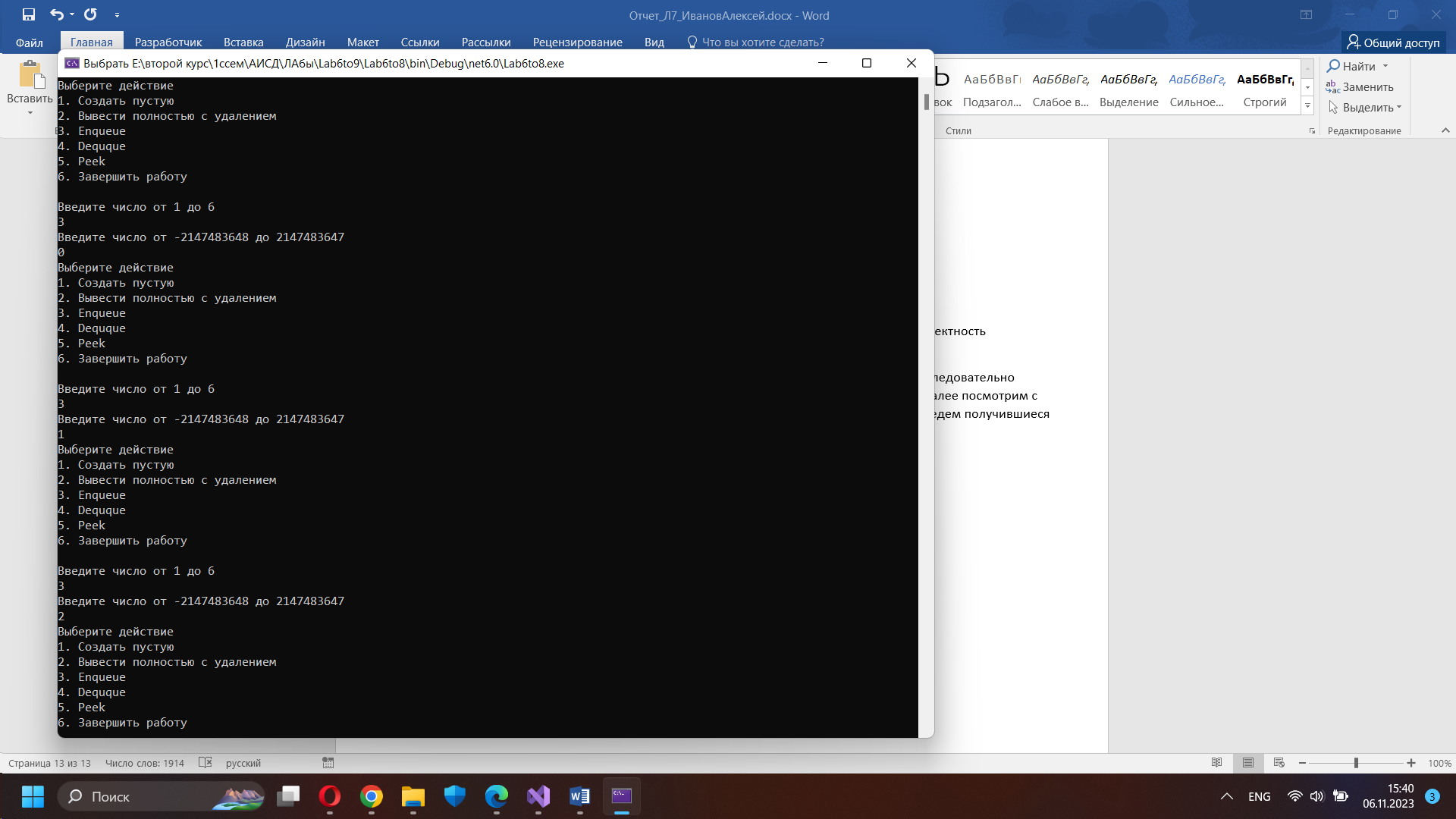
}

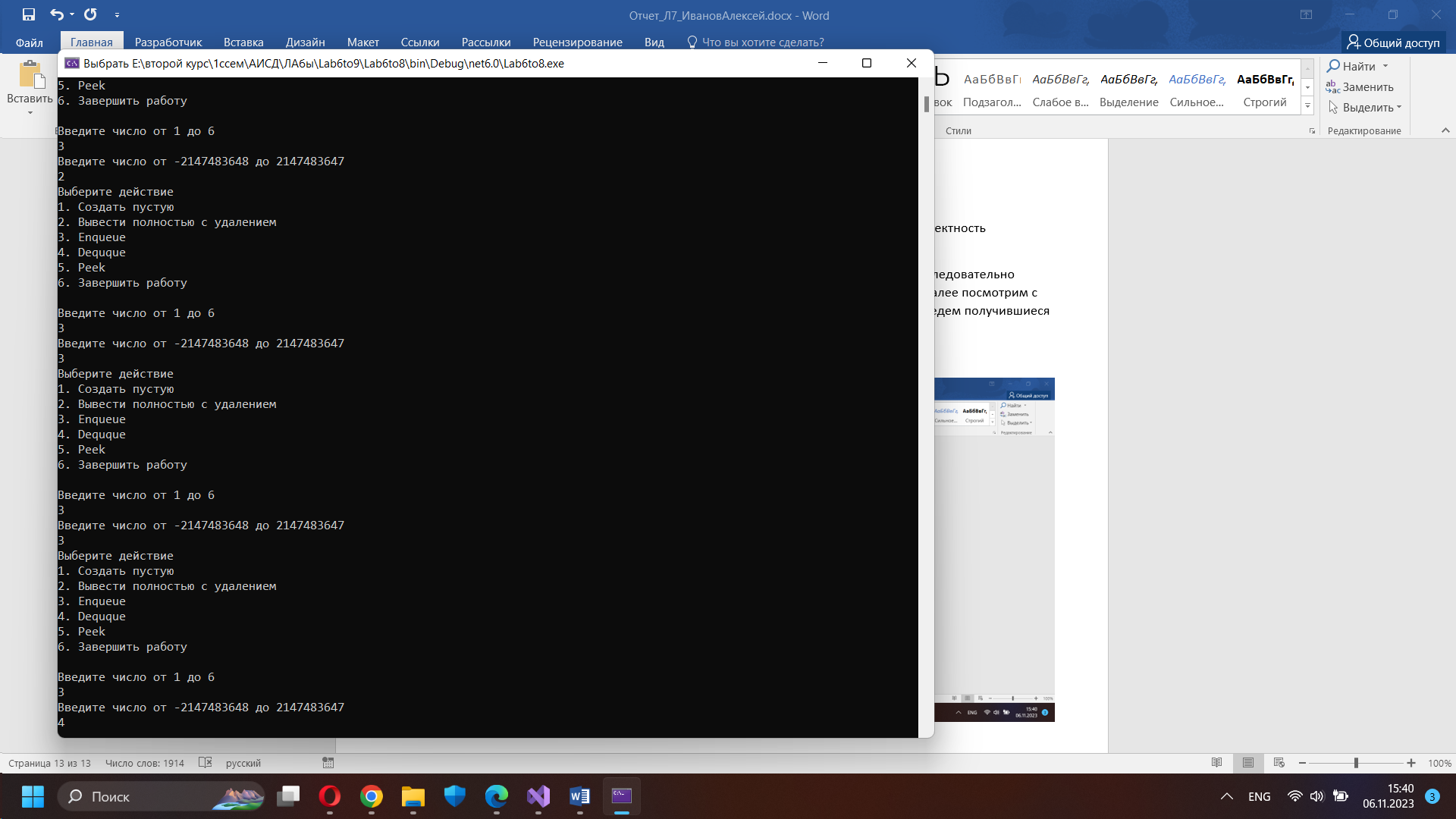
**Часть 5. Тестирование и результаты работы.**

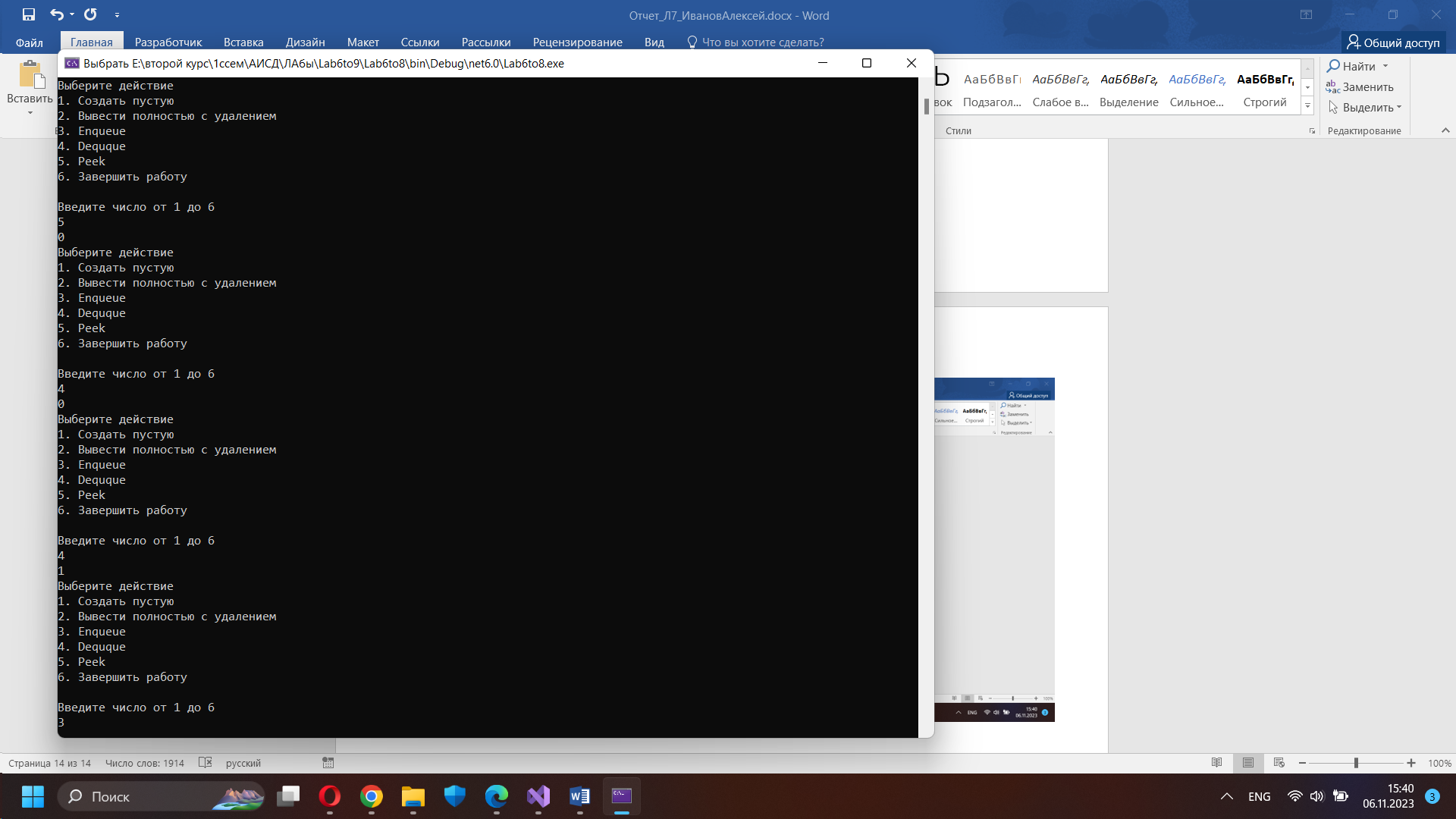
Для тестирования было разработано консольное приложение, проверяющее корректность алгоритмов.

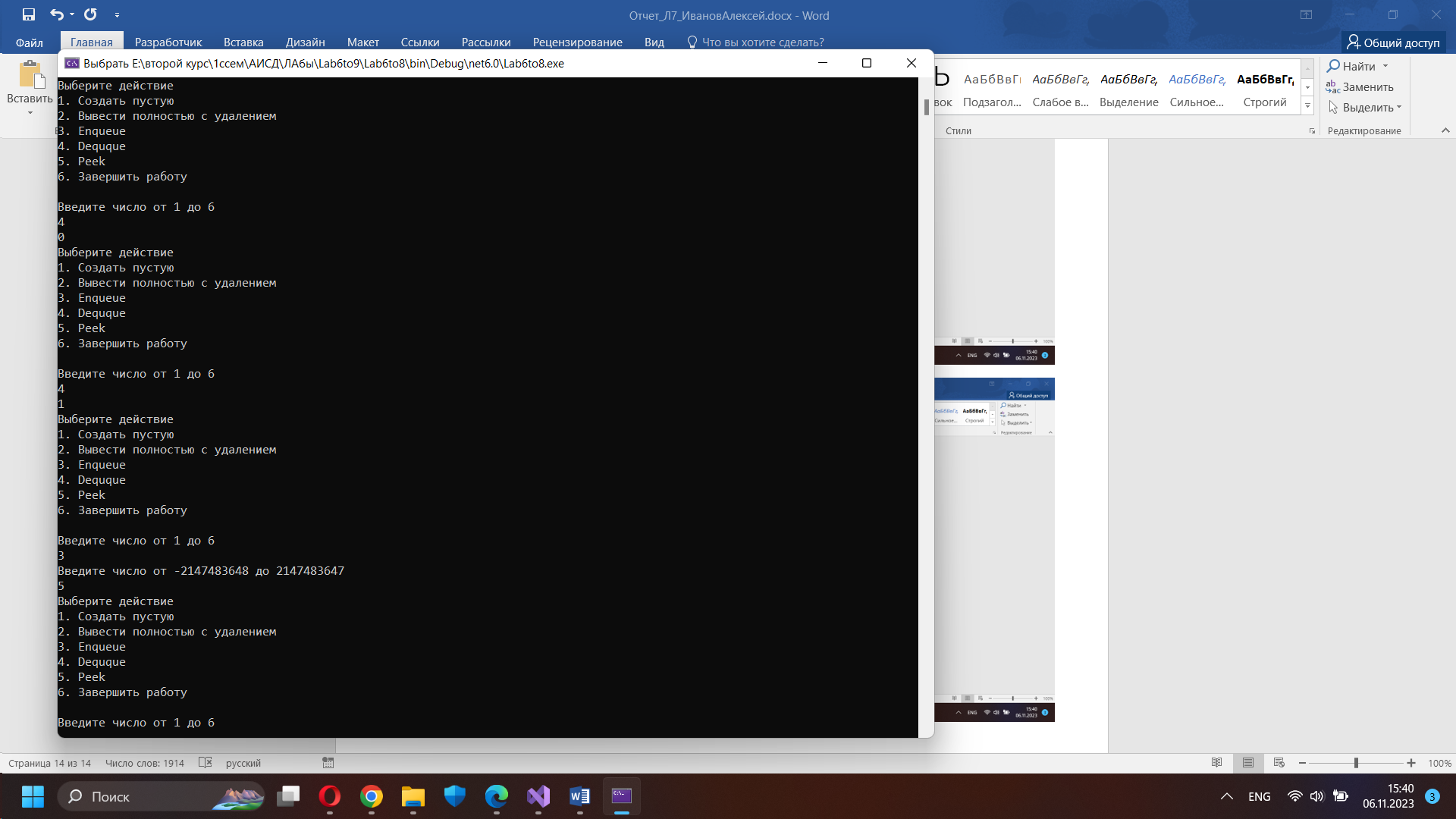
Рассмотрим кольцевую и обычную очереди. Для тестирования запишем в них последовательно элементы 0, 1, 2, 3, 3, 4. Далее посмотрим первый элемент(ожидается вывод 0), далее посмотрим с удалением первые 2(ожидается вывод 0 и 1), далее добавим элементы 5, 5 и выведем получившиеся очереди. Ожидается вывод 2, 3, 3, 4, 5, 5

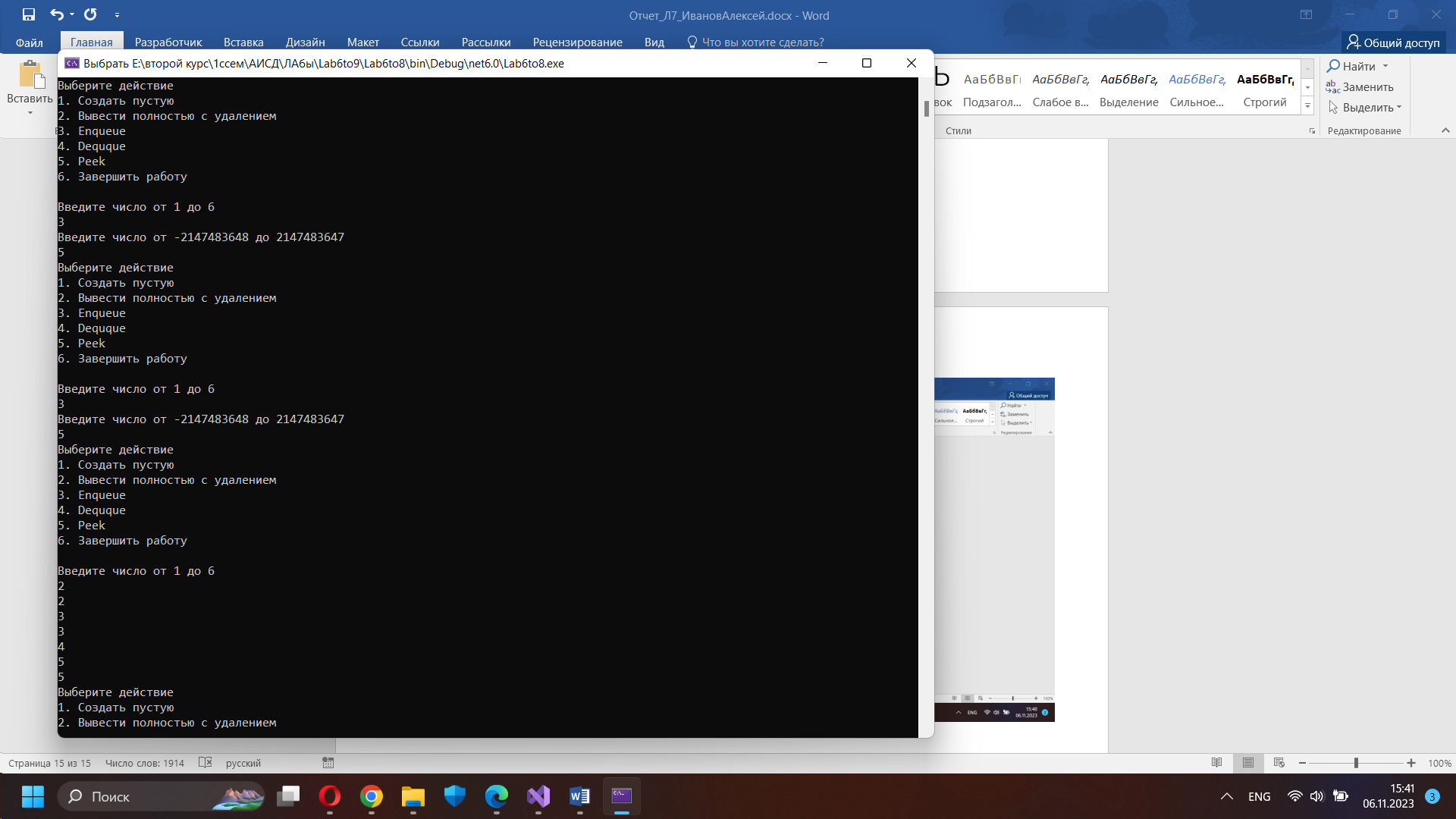
Для простой очереди:



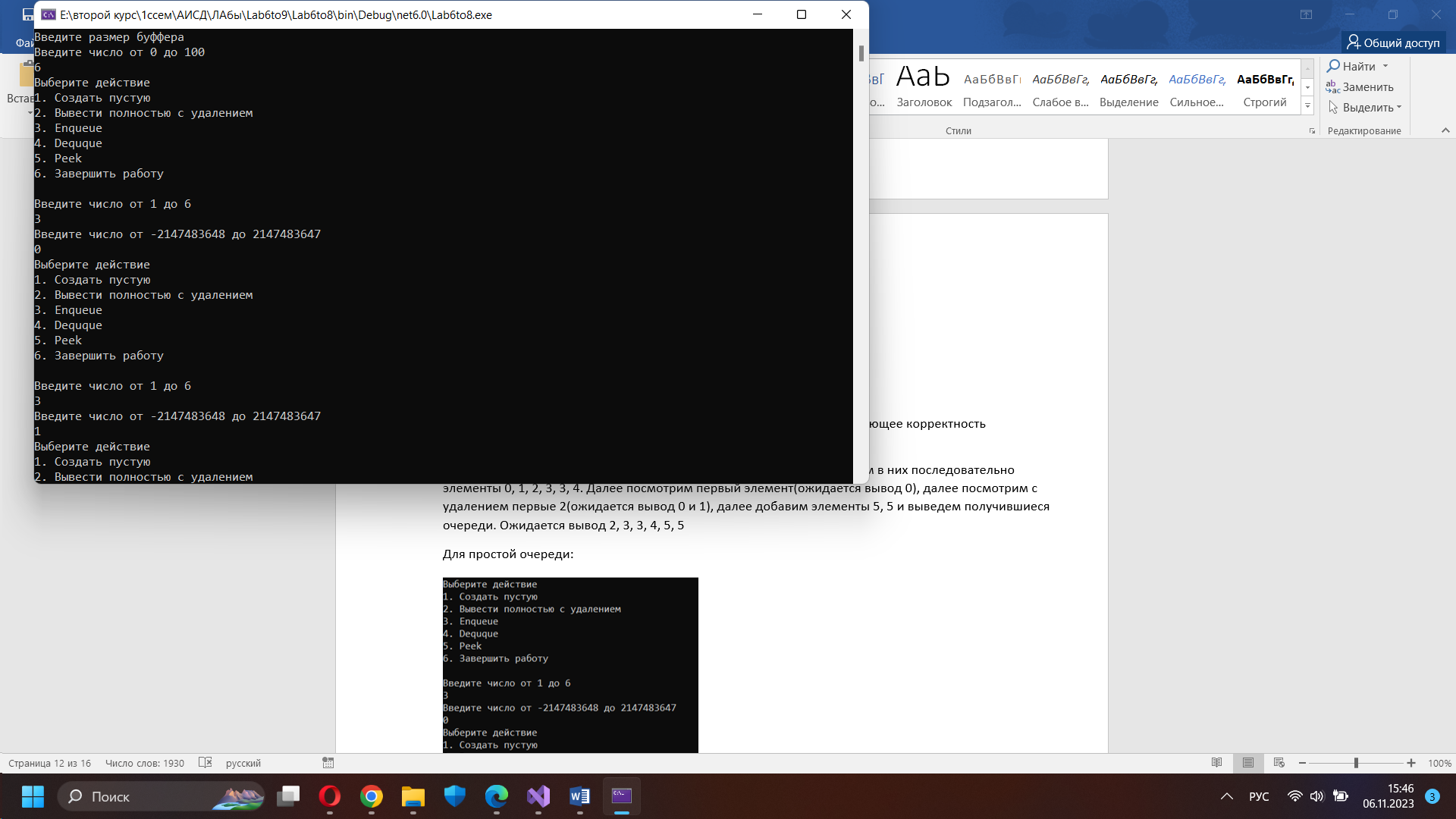


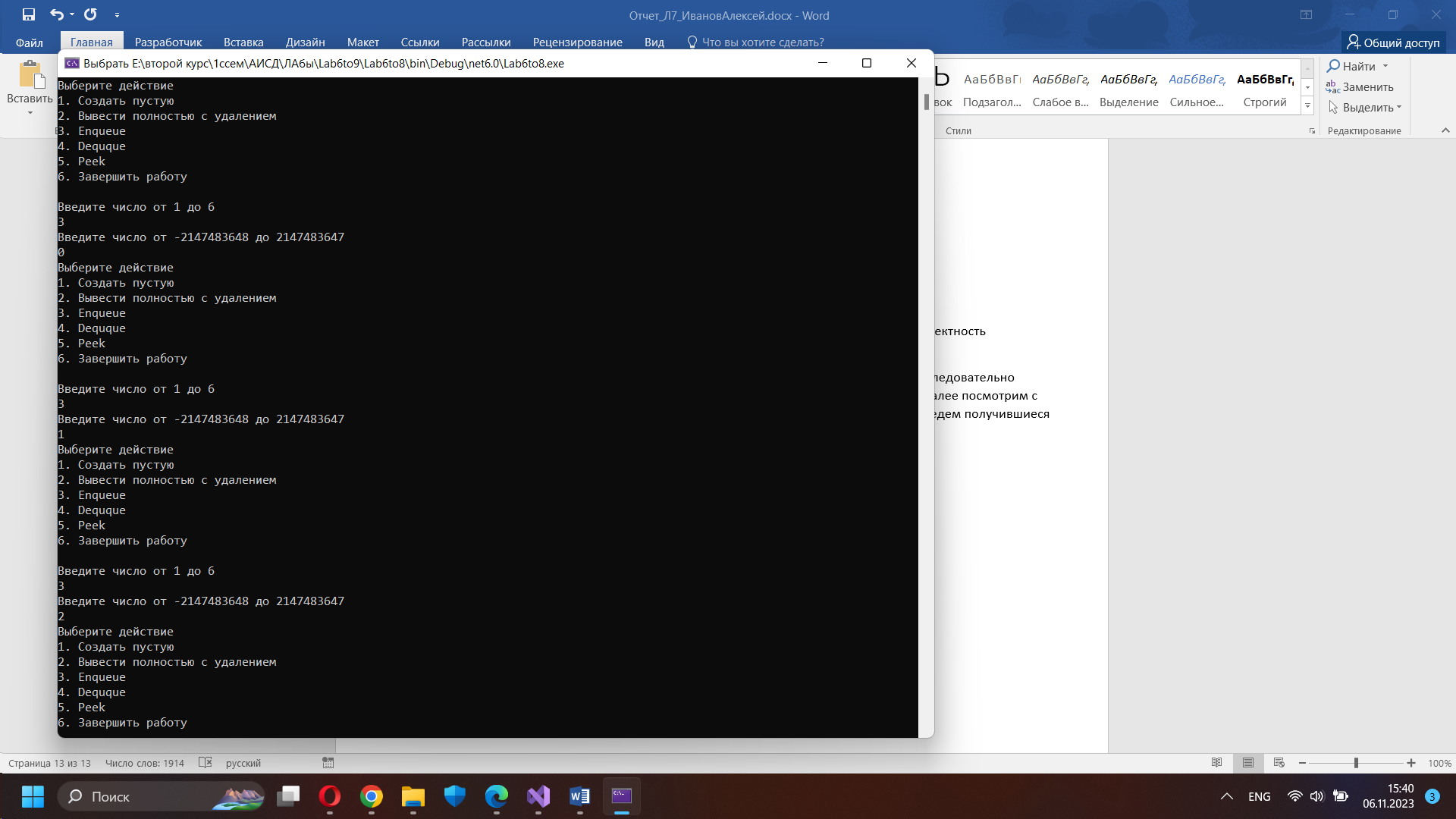


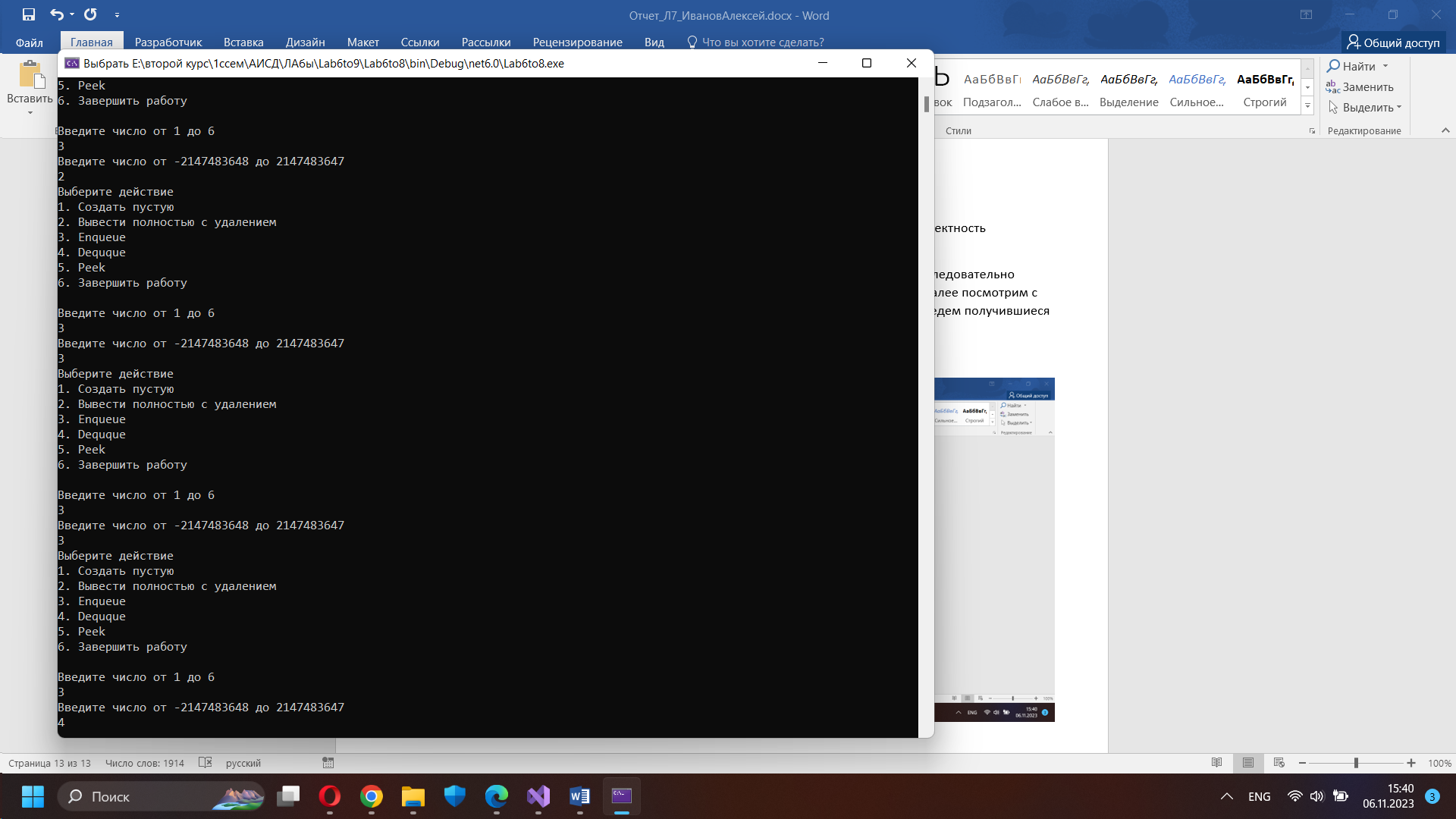


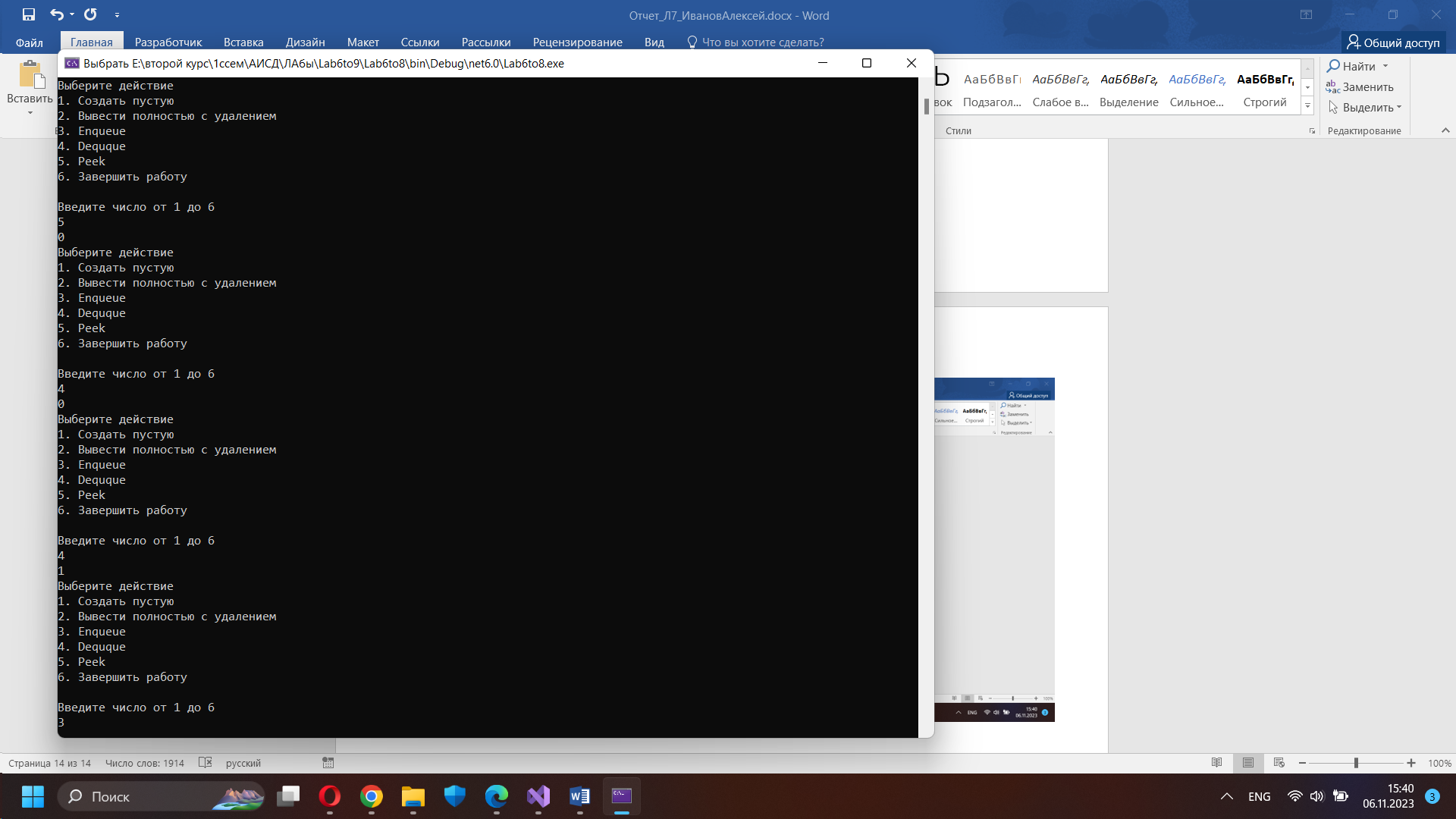


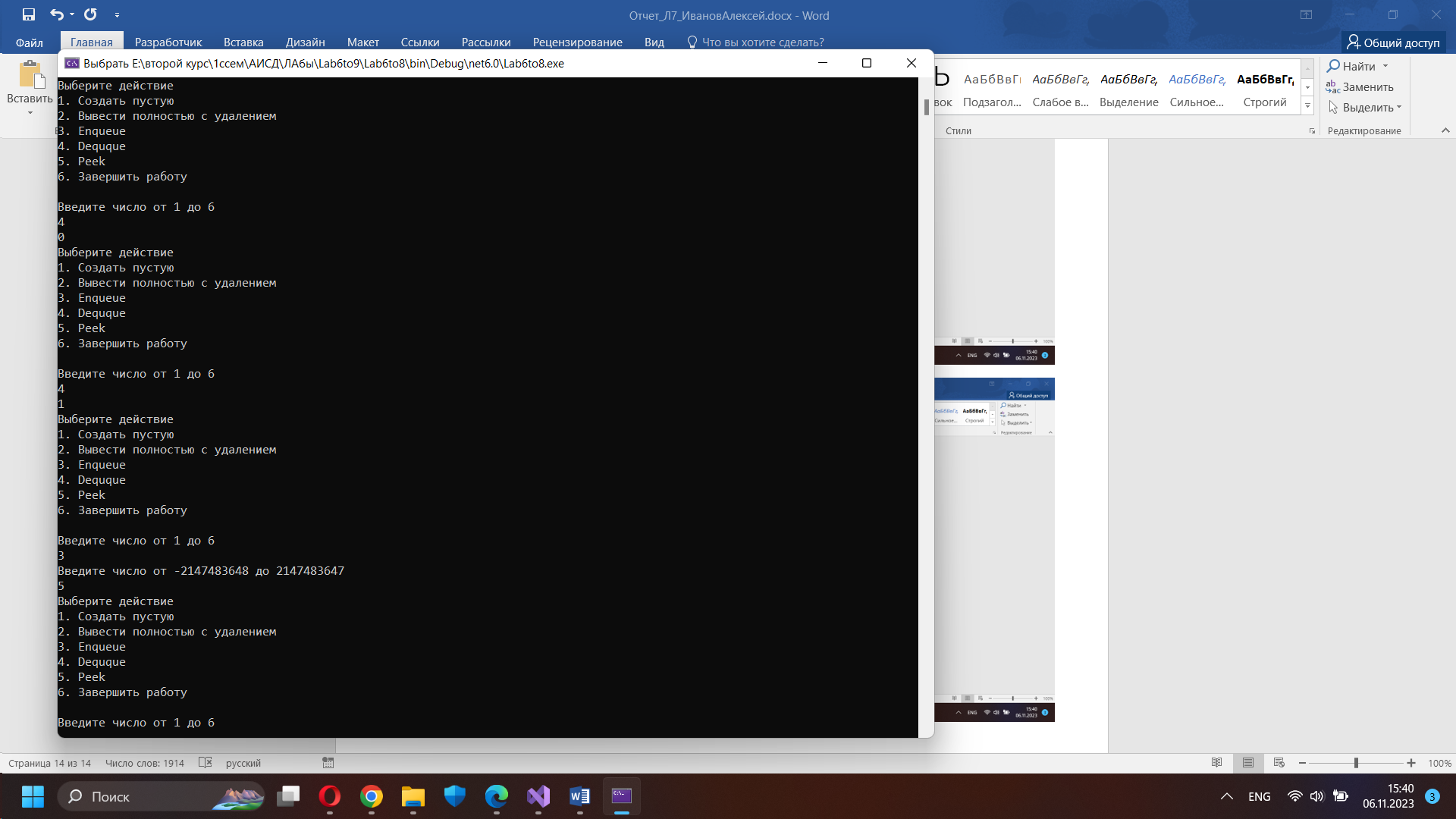
Для кольцевой очереди также определим размер буффера 6, чтобы проверить, правильно ли реализовано смещение по кругу.

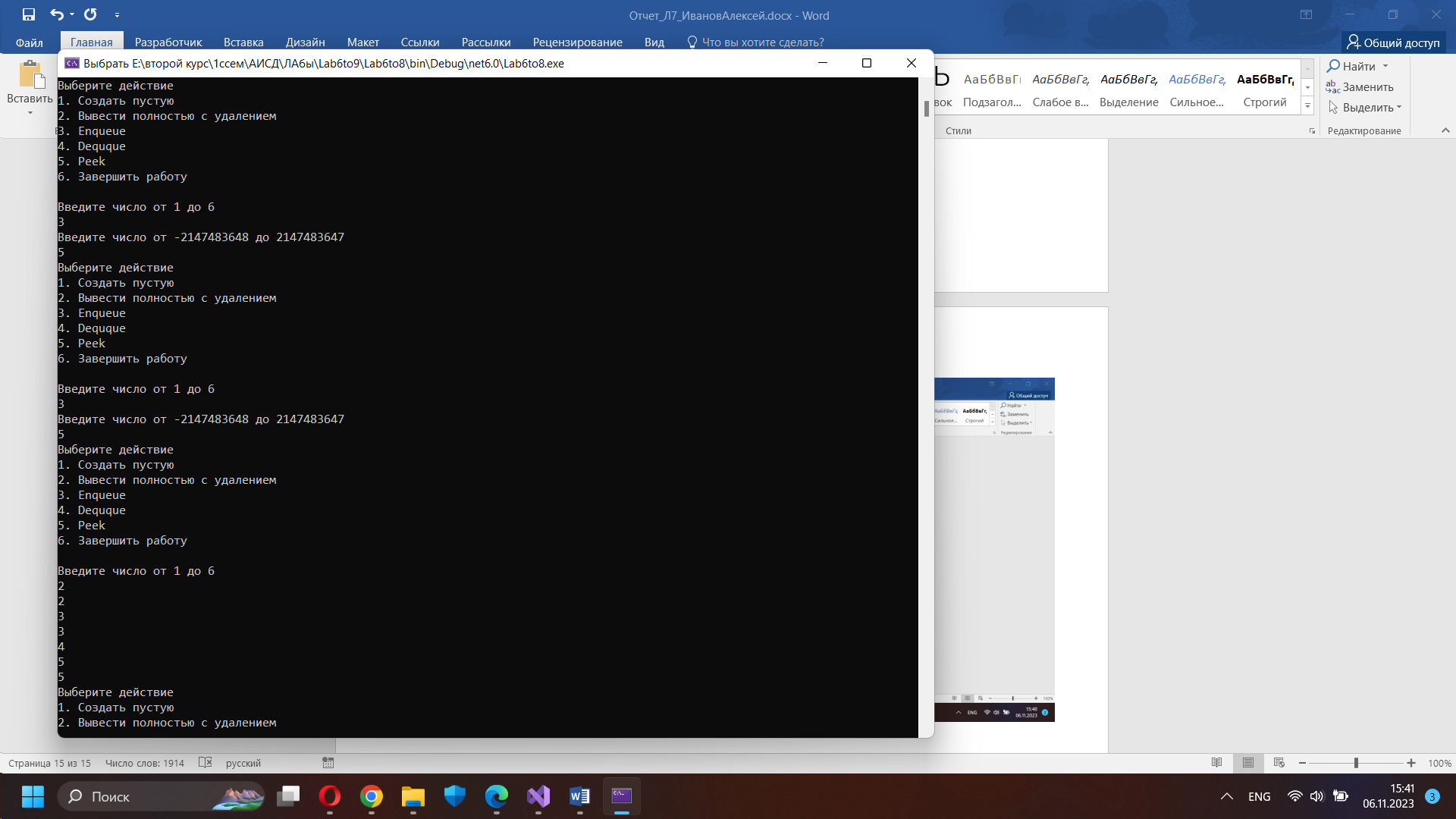






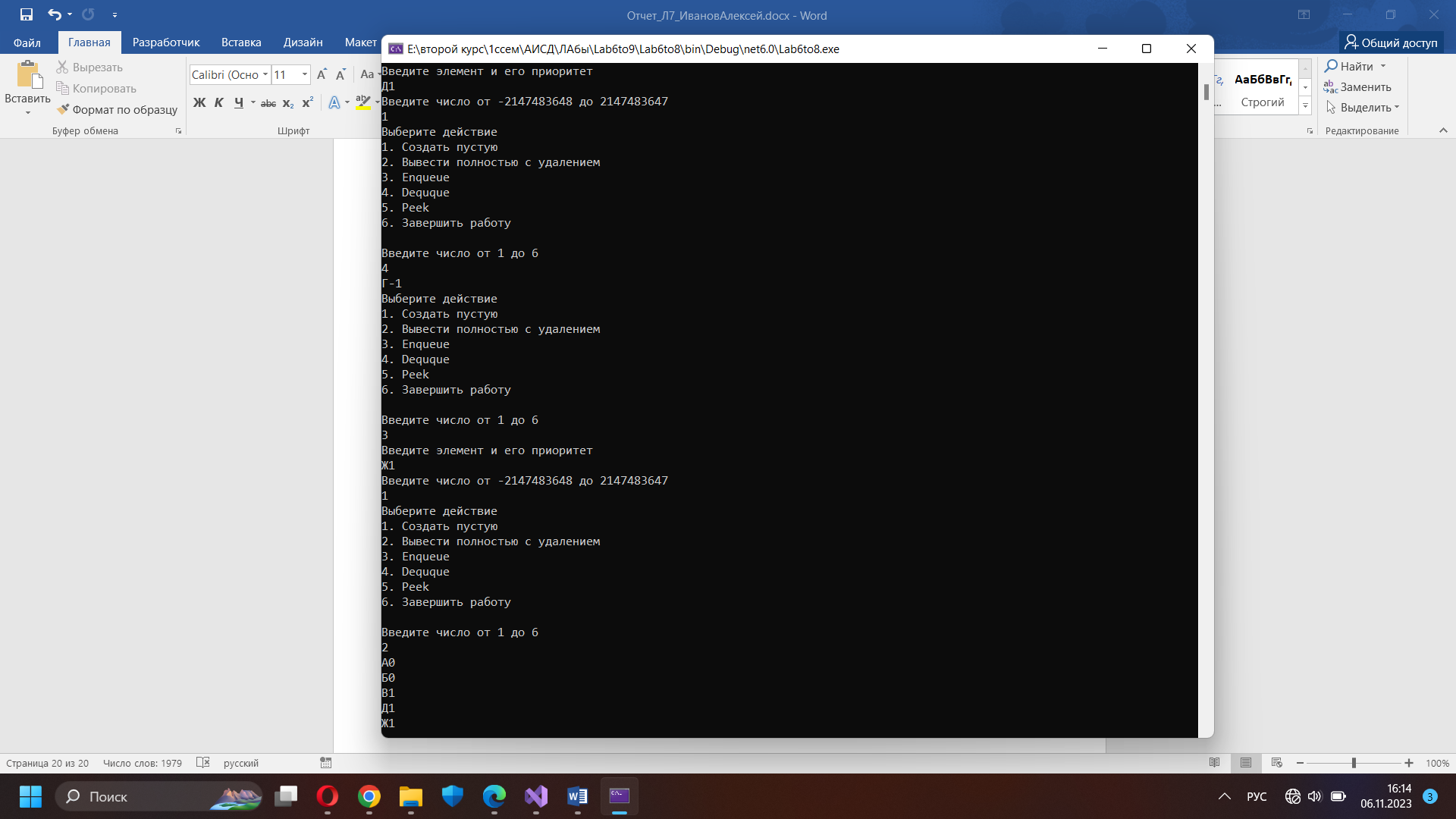
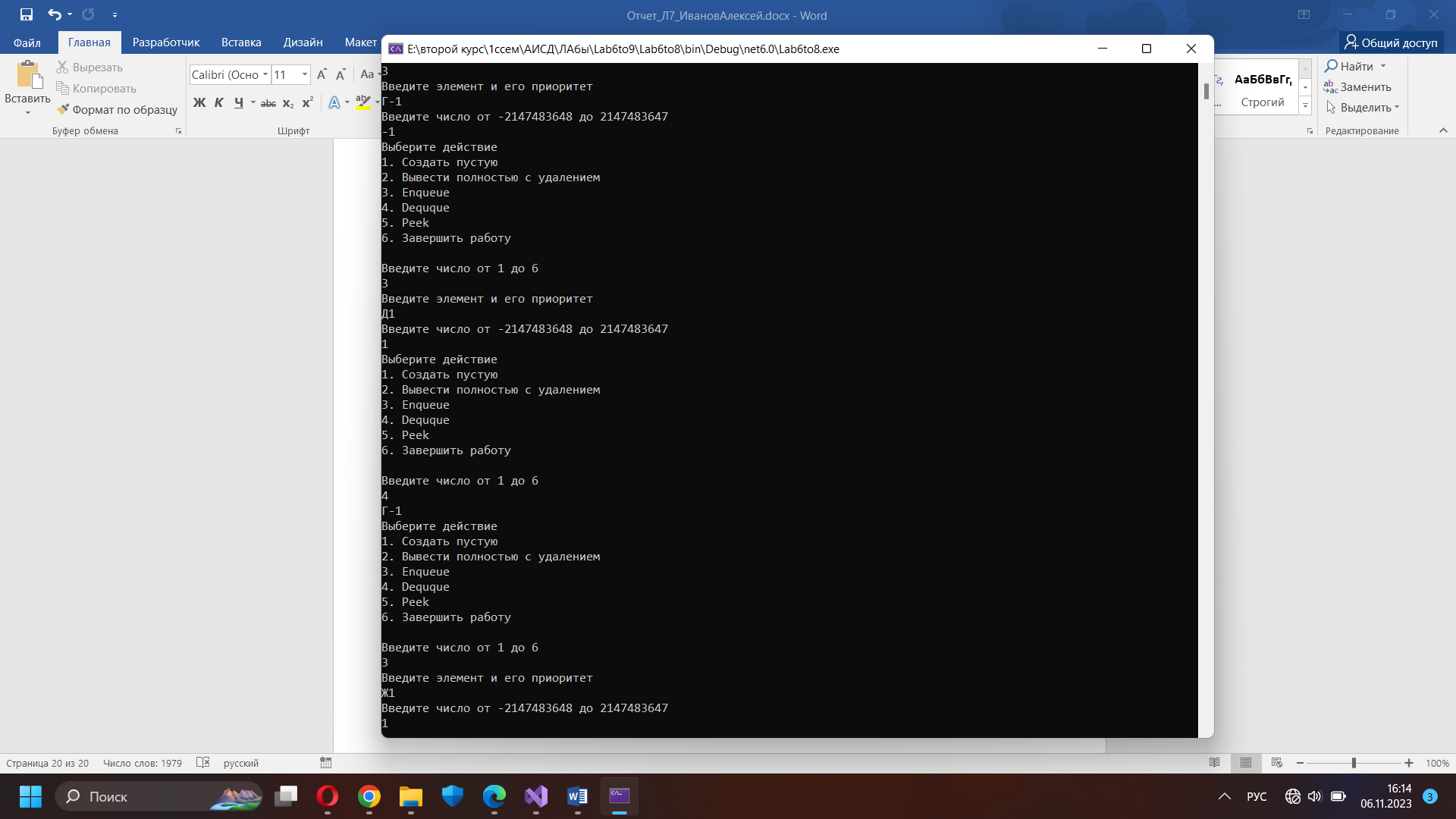
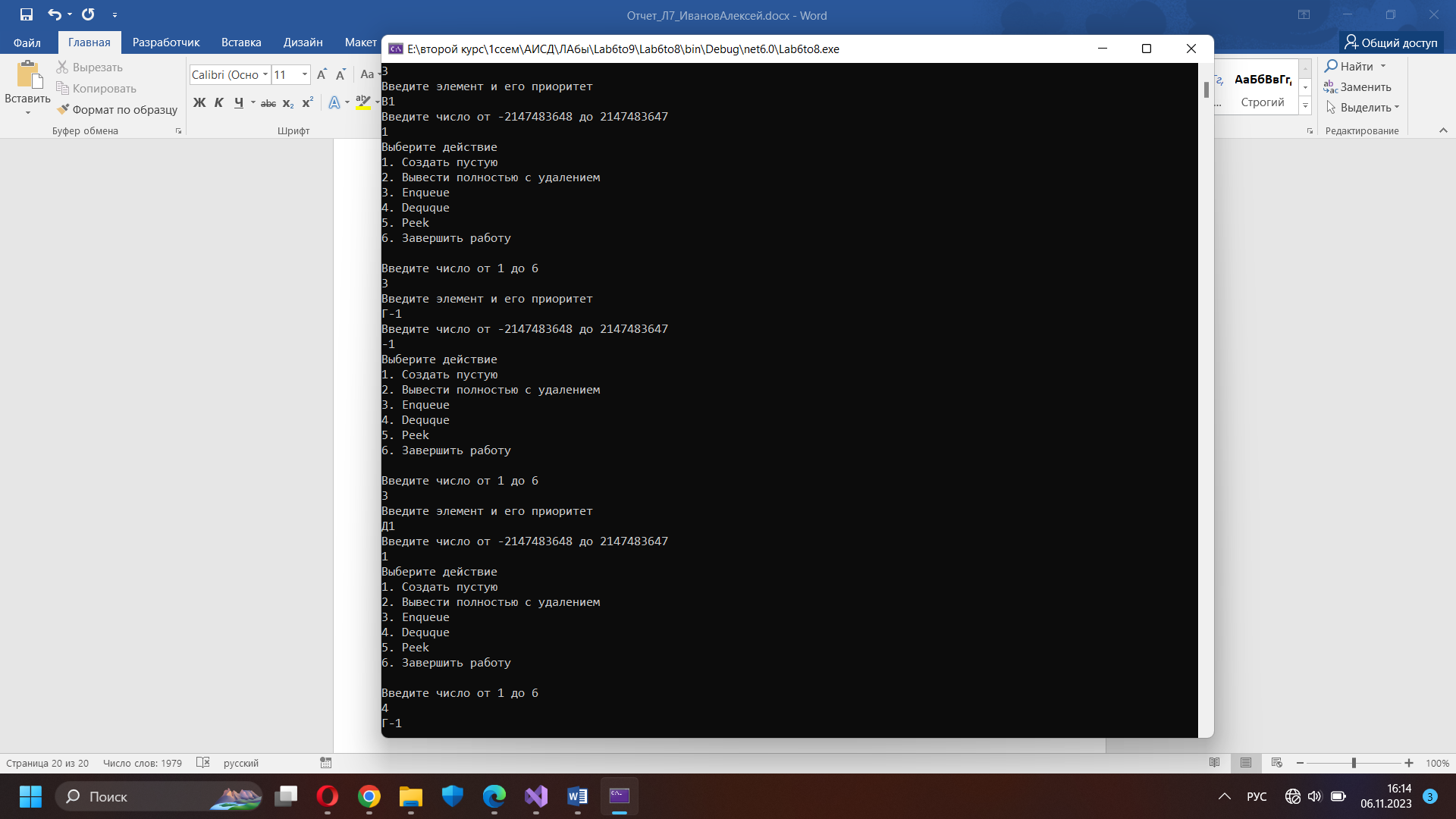
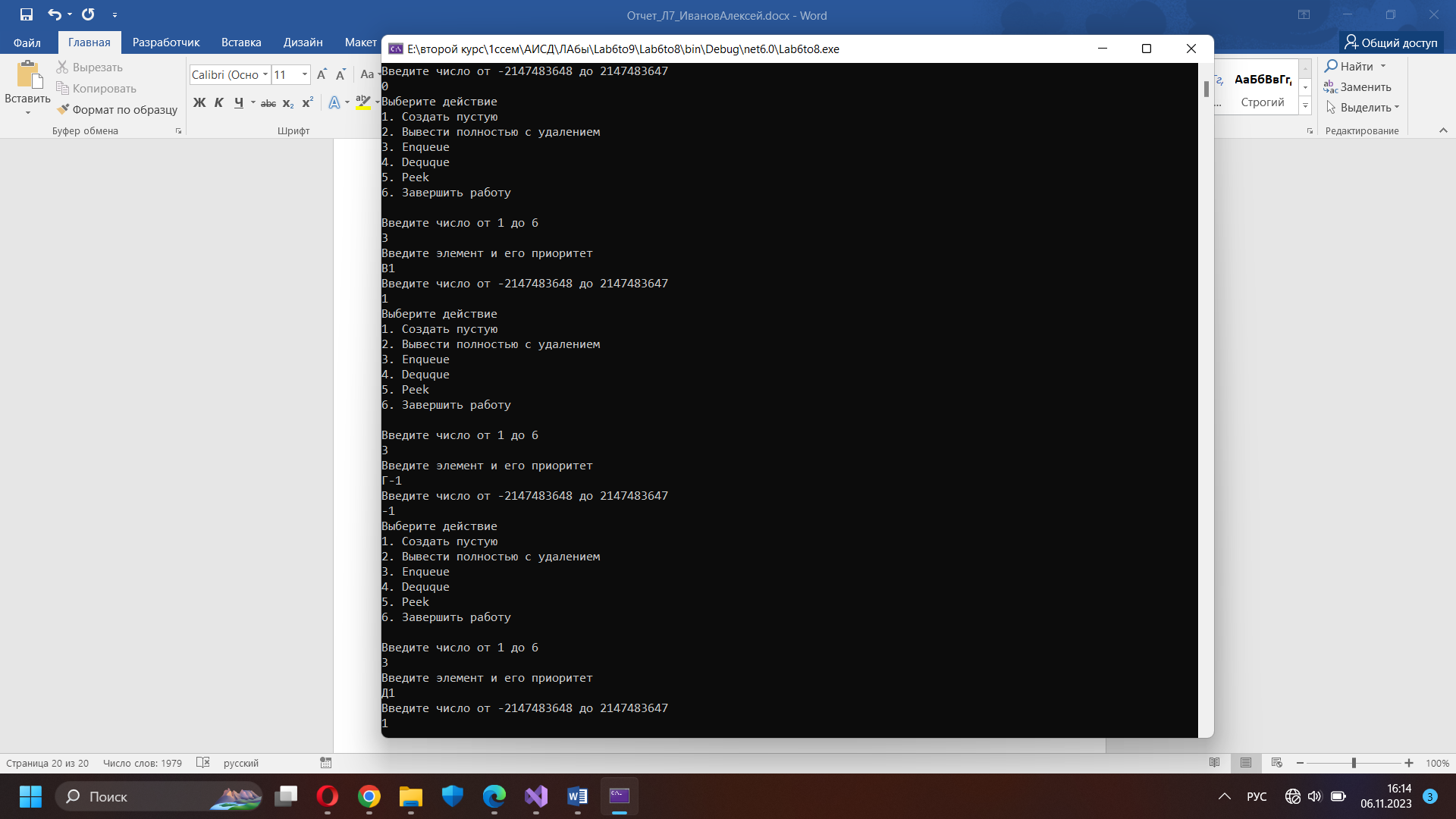
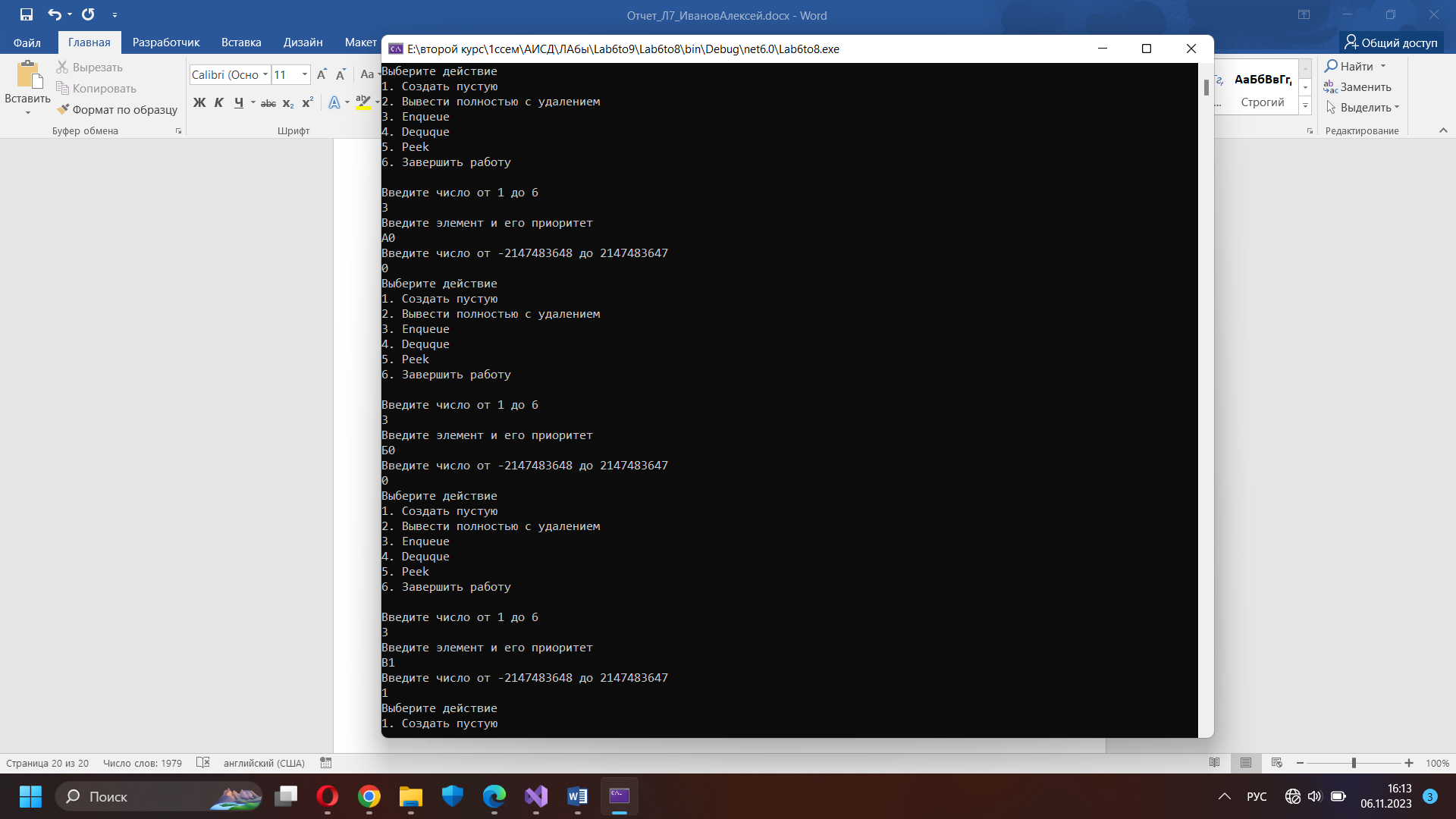






Как видно, все работает корректно.

Для приоритетной очереди будем вставлять элементы А0 с приоритетом 0, Б0 с 0, В1 с 1, Г-1 с -1, Д1 с 1. Просмотрим первый элемент с удалением (ожидается Г-1), далее вставим Ж1 с приоритетом 1, и выведем полностью (ожидается А0 Б0 В1 Д1 Ж1)



Программа работает корректно.