Міністерство освіти і науки України
Національний технічний університет України «Київський політехнічний інститут імені Ігоря Сікорського»
Факультет інформатики та обчислювальної техніки

Кафедра інформатики та програмної інженерії

Звіт

з лабораторної роботи № 1 з дисципліни
«Сучасні технології розробки WEB-застосувань на платформі Microsoft.NET»

«Узагальнені типи (Generic) з підтримкою подій. Колекції»

Варіант 7

Виконала студентка ІП-12 Бондарчук Анастасія Олександрівна

Київ 2023

Завдання:

1. Розробити клас власної узагальненої колекції, використовуючи стандартні інтерфейси колекцій із бібліотек System.Collections та System.Collections.Generic. Стандартні колекції при розробці власної не застосовувати. Для колекції передбачити методи внесення даних будь-якого типу, видалення, пошуку та ін. (відповідно до типу колекції).

2. Додати до класу власної узагальненої колекції підтримку подій та обробку виключних ситуацій.

3. Опис класу колекції та всіх необхідних для роботи з колекцією типів зберегти у динамічній бібліотеці.

4. Створити консольний додаток, в якому продемонструвати використання розробленої власної колекції, підписку на події колекції.

Приклади виключних ситуацій: вихід за межи диапазону чи неприпустимий аргумент (індекс), відсутнє значення за ключем/індексом, несумісна зі станом об’єкту операція.

Приклади подій: очищення колекції, додавання, видалення елементу, потрапляння в початок\кінець.

7	Відсортований динамічний масив	Див. SortedList<T>	Збереження даних за допомогою динамічно зв’язаного списку або вектору

Виконання:

Node.cs:

```
namespace Lab1;

public class Node<T> where T : IComparable
{
    public Node(T data)
    {
        Data = data;
        Next = null;
    }
    public T Data { get; set;  }
    public Node<T>? Next { get; set; }
}
```

MySortedList.cs:

```
using System.Collections;

namespace Lab1;

public class MySortedList<T> : ICollection<T> where T : IComparable
{
    public delegate void EventHandler(MySortedList<T> sender);
    public event EventHandler? CountIncrease;
    public event EventHandler? CountDecrease;
    private Node<T>? _head;
    public int Count { get; set; }
    public bool IsReadOnly => false;

    public T[] GetValuesArray()
    {
        if (Count is 0) throw new Exception("SortedList is empty!");
        T[] array = new T[Count];
        Node<T>? current = _head;
        for (int i = 0; i < Count; i++)
        {
            array[i] = current!.Data;
            current = current.Next;
        }

        return array;
    }
    public List<T> GetValuesList()
    {
        if (Count is 0) throw new Exception("SortedList is empty!");
        List<T> list = new List<T>();
        Node<T> current = _head!;
        for (int i = 0; i < Count; i++)
        {
            list.Add(current.Data);
            current = current.Next!;
        }

        return list;
    }
    public T GetByIndex(int index)
    {
        if (Count is 0) throw new Exception("SortedList is empty!");
        if (index < 0) throw new IndexOutOfRangeException("Index must not be
```

```
less than 0!");
        if (index >= Count) throw new IndexOutOfRangeException("Index must
not be greater than SortedList Count!");
        if (index is 0) return _head!.Data;
        Node<T> current = _head!;
        for (int i = 0; i < index; i++)
        {
            current = current.Next!;
        }

        return current.Data;
    }
    public int GetIndexOf(T item)
    {
        if (Contains(item))
        {
            Node<T>? current = _head;
            for (int i = 0; i < Count; i++)
            {
                if (current!.Data.Equals(item)) return i;
                current = current.Next;
            }
        }
        throw new Exception("There is no element in the SortedList equal to
your value!");
    }
    public void Add(T item)
    {
        Node<T> node = new Node<T>(item);
        Node<T>? previous = FindPlace(node);
        if (previous is not null)
        {
            Node<T>? next = previous.Next;
            previous.Next = node;
            node.Next = next;
        }
        else
        {
            node.Next = _head;
            _head = node;

        }
        CountIncrease?.Invoke(this);
    }
    private Node<T>? FindPlace(Node<T> node)
    {
        if (Count is 0) return null;
        Node<T>? previous = null;
        Node<T>? current = _head;
        for (int i = 0; i < Count; i++)
        {
            if (node.Data.CompareTo(current!.Data) < 0)
            {
                return previous;
            }

            previous = current;
            current = current.Next;
        }

        return previous;
    }
    public void CopyTo(T[] array, int arrayIndex)
    {
```

```
        if (array.Length - arrayIndex < Count) throw new
IndexOutOfRangeException("SortedList contains more elements than Array might
contain!");
        if (arrayIndex < 0) throw new IndexOutOfRangeException("ArrayIndex
must not be less than 0!");
        if (array.Rank != 1) throw new ArgumentException("Array must be one-
dimensional!");
        if (Count is 0) return;
        Node<T>? current = _head;
        for (int i = 0; i < Count; i++)
        {
            array[i + arrayIndex] = current!.Data;
            current = current.Next;
        }
    }
    public void Clear()
    {
        _head = null;
        Count = 0;
    }
    public bool Contains(T item)
    {
        if (Count > 0)
        {
            Node<T>? current = _head;
            for (int i = 0; i < Count; i++)
            {
                if (current!.Data.Equals(item)) return true;
                current = current.Next;
            }
        }

        return false;
    }
    public bool Remove(T item)
    {
        if (!Contains(item)) return false;
        Node<T>? previous = null;
        Node<T>? current = _head;
        for (int i = 0; i < Count; i++)

        {
            if (current!.Data.Equals(item))
            {
                RemoveNode(current, previous);
                CountDecrease?.Invoke(this);
                return true;
            }
            previous = current;
            current = current.Next;
        }

        return false;
    }
    public bool RemoveAll(T item)
    {
        if (!Contains(item)) return false;
        while (Contains(item)) Remove(item);
        return false;
    }
    public bool RemoveByIndex(int index)
    {
        if (Count is 0) throw new Exception("SortedList is empty!");
        if (index < 0) throw new IndexOutOfRangeException("Index must not be
```

```
less than 0!");
        if (index >= Count) throw new IndexOutOfRangeException("Index must
not be greater than SortedList Count!");
        if (index is 0)
        {
            _head = _head!.Next;
            return true;
        }

        Node<T>? previous = null;
        Node<T>? current = _head;
        for (int i = 0; i < index; i++)
        {
            previous = current;
            current = current!.Next;
        }
        RemoveNode(current!, previous);
        CountDecrease?.Invoke(this);
        return true;
    }
    private void RemoveNode(Node<T> current, Node<T>? previous)
    {
        if (previous is not null)
        {
            previous.Next = current.Next;
        }
        else
        {
            _head = current.Next;
        }
    }
    public IEnumerator<T> GetEnumerator()
    {
        return new MyEnumerator(this);
    }
    private class MyEnumerator : IEnumerator<T>
    {
        public T Current => _current!.Data;
        object IEnumerator.Current => _current!.Data;

        private static Node<T>? _current;
        // private static Node<T>?  next;
        private readonly MySortedList<T> _list;
        private int _counter;

        public MyEnumerator(MySortedList<T> list)
        {
            _list = list;
            if (_list._head != null) _current = _list._head;
            _counter = 0;
        }

        public bool MoveNext()
        {
            if (_counter >= _list.Count - 1)
            {
                return false;
            }

            _current = _current!.Next;
            _counter++;
            // _next = _current!.Next;
            return true;
        }
```

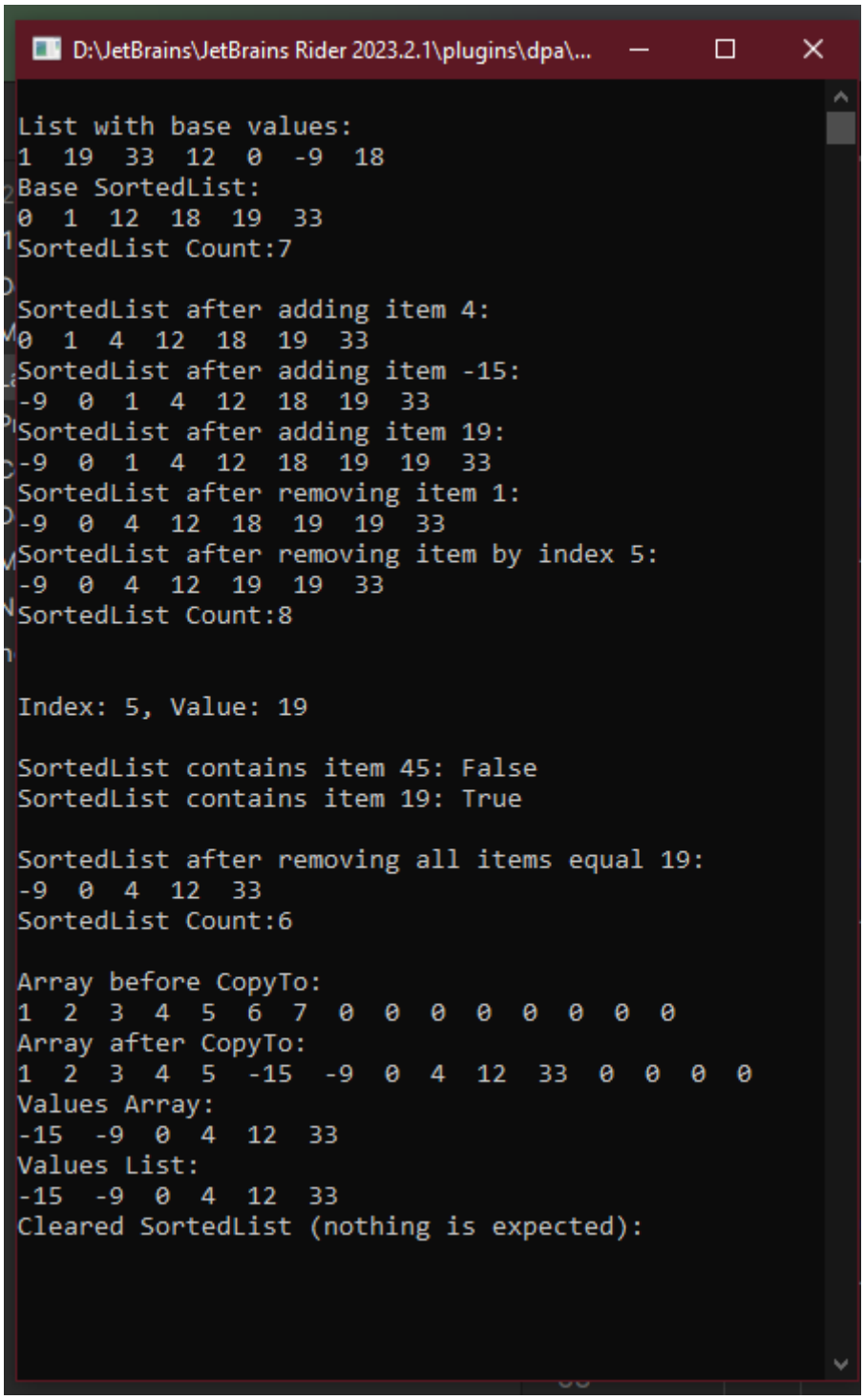
```
        public void Reset()
        {
            _current = _list._head;
        }


        public void Dispose() { }
    }
    IEnumerator IEnumerable.GetEnumerator()
    {
        return GetEnumerator();
    }

}
```

Program.cs:

```
namespace Lab1;
internal class Program
{
    static void Main(string[] args)
    {
        MySortedList<int> sortedList = new MySortedList<int>();

        sortedList.CountIncrease += sender => sender.Count++;
        sortedList.CountDecrease += sender => sender.Count--;

        List<int> list = new List<int> {1, 19, 33, 12, 0, -9, 18};

        Print(list, "List with base values:");
        foreach (var number in list) sortedList.Add(number);
        Print(sortedList, "Base SortedList:");
        Console.WriteLine($"\nSortedList Count:{sortedList.Count}");

        sortedList.Add(4);
        Print(sortedList, "SortedList after adding item 4:");
        sortedList.Add(-15);
        Print(sortedList, "SortedList after adding item -15:");
        sortedList.Add(19);
        Print(sortedList, "SortedList after adding item 19:");
        sortedList.Remove(1);
        Print(sortedList, "SortedList after removing item 1:");
        sortedList.RemoveByIndex(5);
        Print(sortedList, "SortedList after removing item by index 5:");
        Console.WriteLine($"\nSortedList Count:{sortedList.Count}");

        int index = sortedList.GetIndexOf(19);
        int value = sortedList.GetByIndex(index);
        Console.WriteLine($"\n\nIndex: {index}, Value: {value}");

        Console.WriteLine($"\nSortedList contains item 45:
{sortedList.Contains(45)}");
        Console.WriteLine($"SortedList contains item 19:
{sortedList.Contains(19)}");
        sortedList.RemoveAll(19);

        Print(sortedList, "SortedList after removing all items equal 19:");
        Console.WriteLine($"\nSortedList Count:{sortedList.Count}");
        int[] arr = new int[15];
        for (int i = 0; i < 7; i++) arr[i] = i + 1;
        Print(arr, "Array before CopyTo:");
```

```
        sortedList.CopyTo(arr, 5);
        Print(arr, "Array after CopyTo:");
        int[] valuesArray = sortedList.GetValuesArray();
        Print(valuesArray, "Values Array:");
        List<int> valuesList = sortedList.GetValuesList();
        Print(valuesList, "Values List:");
        sortedList.Clear();
        Print(sortedList, "Cleared SortedList (nothing is expected):");
        Console.ReadLine();
    }

    private static void Print(List<int> list, string message)
    {
        Console.WriteLine($"\n{message}");
        foreach (var item in list) Console.Write($"{item}  ");
    }

    private static void Print(int[] array, string message)
    {
        Console.WriteLine($"\n{message}");
        foreach (var item in array)  Console.Write($"{item}  ");
    }

    private static void Print(MySortedList<int> sortedList, string message)
    {
        Console.WriteLine($"\n{message}");
        foreach (var item in sortedList)  Console.Write($"{item}  ");
    }

}
```

Результати виконання:



Доступ до коду:

https://github.com/NastasaBondarchuck/Web_dotNet_Labs/tree/main/Lab1