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

«Модульне тестування. Ознайомлення з засобами та практиками модульного тестування»

Варіант 7

Виконала студентка ІП-12 Бондарчук Анастасія Олександрівна

Київ 2023

Завдання:

1. Додати до проекту власної узагальненої колекції (застосувати виконану лабораторну роботу No1) проект модульних тестів, використовуючи певний фреймворк (Nunit, Xunit, тощо).

2. Розробити модульні тести для функціоналу колекції.

3. Дослідити ступінь покриття модульними тестами вихідного коду колекції, використовуючи, наприклад, засіб AxoCover.

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 MySortedList()
    {
        CountIncrease += sender => sender.Count++;
        CountDecrease += sender => sender.Count--;
    }

    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 (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 true;
    }
    public bool RemoveAll(T item)
    {
        if (!Contains(item)) return false;
        while (Contains(item)) Remove(item);
```

```
        return true;
    }
    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) _next = _list._head;
            _counter = 0;
        }
        public bool MoveNext()
        {
            if (_counter > _list.Count - 1)
            {
                return false;
            }

            _current = _next;
```

```
            _counter++;
            _next = _current!.Next;
            return true;
        }
        public void Reset()
        {
            _current = _list._head;
        }
        public void Dispose() { }
    }
    IEnumerator IEnumerable.GetEnumerator()
    {
        return GetEnumerator();
    }

}
```

Usings.cs:

```
global using NUnit.Framework;
```

MyCollectionAddTests.cs:

```
using Lab1;

namespace MyCollection.Tests;

[TestFixture]
public class MyCollectionAddTests
{
    [Test]
    public void Add_SortedList_ReturnSortedList()
    {
        Random rand = new Random();
        MySortedList<int> sortedList = new MySortedList<int>();
        List<int> list = new List<int>();

        for (int i = 0; i < 10; i++)
        {
            int item = rand.Next(-10, 11);
            list.Add(item);
            sortedList.Add(item);
        }
        list.Sort();

        Assert.That(sortedList.ToList(), Is.EqualTo(list));
    }
}
```

MyCollectionClearTests.cs:

```
using Lab1;

namespace MyCollection.Tests;

[TestFixture]
public class MyCollectionClearTests
{
    [Test]
```

```
    public void
Clear_NotEmptySortedList_ReturnContainsFalseAndCountEqualToZero()
    {
        MySortedList<int> sortedList = new MySortedList<int>();
        Random rand = new Random();

        for (int i = 0; i < 10; i++)
        {
            int item = 10 - i;
            sortedList.Add(item);
        }
        sortedList.Clear();

        Assert.That(sortedList.Count, Is.EqualTo(0));
        Assert.That(sortedList.Contains(rand.Next(1, 11)),
Is.EqualTo(false));
    }

    [Test]
    public void
Clear_EmptySortedList_ReturnContainsFalseAndCountEqualToZero()
    {
        MySortedList<int> sortedList = new MySortedList<int>();
        Random rand = new Random();
        sortedList.Clear();

        Assert.That(sortedList.Count, Is.EqualTo(0));
        Assert.That(sortedList.Contains(rand.Next(1, 11)),
Is.EqualTo(false));
    }
}
```

MyCollectionContainsTests.cs:

```
using Lab1;

namespace MyCollection.Tests;

[TestFixture]
public class MyCollectionContainsTests
{
    [Test]
    public void GetIndexOf_EmptySortedList_ReturnFalse()
    {
        MySortedList<int> sortedList = new MySortedList<int>();
        Random rand = new Random();

        Assert.That(sortedList.Contains(rand.Next(-10, 11)), Is.False);
    }

    [Test]
    public void
GetIndexOf_NotEmptySortedListWithoutSearchedValue_ReturnFalse()
    {
        MySortedList<int> sortedList = new MySortedList<int>();
        Random rand = new Random();

        for (int i = 0; i < 10; i++)
        {
            int item = rand.Next(-10, 11);
            sortedList.Add(item);
        }
```

```
        Assert.That(sortedList.Contains(rand.Next(11, 21)), Is.False);
    }

    [Test]
    public void GetIndexOf_NotEmptySortedListWithSearchedValue_ReturnTrue()
    {
        MySortedList<int> sortedList = new MySortedList<int>();
        Random rand = new Random();

        for (int i = 0; i < 10; i++)
        {
            int item = 10 - i;
            sortedList.Add(item);
        }

        Assert.That(sortedList.Contains(rand.Next(1, 11)), Is.True);
    }
}
```

MyCollectionCopyToTests.cs:

```
using Lab1;

namespace MyCollection.Tests;

[TestFixture]
public class MyCollectionCopyToTests
{
    [Test]
    public void
CopyTo_InsertDiapasonLessThanNotEmptySortedListCount_ReturnIndexOutOfRangeExc
eption()
    {
        MySortedList<int> sortedList = new MySortedList<int>();
        Random rand = new Random();
        int[] array = { 1, 2, 3, 4, 5};

        for (int i = 0; i < 10; i++)
        {
            int item = rand.Next(-10, 11);
            sortedList.Add(item);
        }

        Assert.Throws<IndexOutOfRangeException>(() =>
sortedList.CopyTo(array, rand.Next(0, 5)));
    }

    [Test]
    public void
CopyTo_IndexLessThanZeroWithNotEmptySortedList_ReturnIndexOutOfRangeException
()
    {
        MySortedList<int> sortedList = new MySortedList<int>();
        Random rand = new Random();
        int[] array = { 1, 2, 3, 4, 5};

        for (int i = 0; i < 10; i++)
        {
            int item = rand.Next(-10, 11);
            sortedList.Add(item);
        }
```

```
        Assert.Throws<IndexOutOfRangeException>(() =>
sortedList.CopyTo(array, rand.Next(-10, 0)));
    }

    [Test]
    public void CopyTo_EmptySortedList_ReturnArrayWithNoChanges()
    {
        MySortedList<int> sortedList = new MySortedList<int>();
        Random rand = new Random();
        int[] arrayBeforeCoping = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
        int[] arrayAfterCoping = new int[10];
        arrayBeforeCoping.CopyTo(arrayAfterCoping, 0);
        sortedList.CopyTo(arrayAfterCoping, rand.Next(0, 10));

        Assert.That(arrayAfterCoping, Is.EqualTo(arrayBeforeCoping));
    }

}
```

MyCollectionEventsTests.cs:

```
using Lab1;

namespace MyCollection.Tests;

[TestFixture]
public class MyCollectionEventsTests
{
    [Test]
    public void Events_CountIncrease_ReturnCountIncreasedByOne()
    {
        Random rand = new Random();
        MySortedList<int> sortedList = new MySortedList<int>();
        int count = sortedList.Count;
        sortedList.CountIncrease += delegate { count += 1; };

        sortedList.Add(rand.Next(-10, 11));

        Assert.That(count, Is.EqualTo(sortedList.Count));
    }

    [Test]
    public void Events_CountDecrease_ReturnCountDecreasedByOne()
    {
        Random rand = new Random();
        MySortedList<int> sortedList = new MySortedList<int>();

        int item = rand.Next(-10, 11);
        sortedList.Add(item);
        int count = sortedList.Count;
        sortedList.CountDecrease += delegate { count -= 1; };
        sortedList.Remove(item);

        Assert.That(count, Is.EqualTo(sortedList.Count));
    }
}
```

MyCollectionGetByIndexTests.cs:

```
using Lab1;

namespace MyCollection.Tests;

[TestFixture]
public class MyCollectionGetByIndexTests
{
    [Test]
    public void GetByIndex_EmptySortedList_ReturnException()
    {
        MySortedList<int> sortedList = new MySortedList<int>();
        Random rand = new Random();

        Assert.Throws<Exception>(() => sortedList.GetByIndex(rand.Next(0,
11)));
    }

    [Test]
    public void
GetByIndex_IndexLessThanZeroInNotEmptySortedList_ReturnIndexOutOfRangeExcepti
on()
    {
        MySortedList<int> sortedList = new MySortedList<int>();
        Random rand = new Random();

        for (int i = 0; i < 10; i++)
        {
            int item = rand.Next(-10, 11);
            sortedList.Add(item);
        }

        Assert.Throws<IndexOutOfRangeException>(() =>
sortedList.GetByIndex(rand.Next(-10, 0)));
    }

    [Test]
    public void
GetByIndex_IndexGreaterThanCountInNotEmptySortedList_ReturnIndexOutOfRangeExc
eption()
    {
        MySortedList<int> sortedList = new MySortedList<int>();
        Random rand = new Random();

        for (int i = 0; i < 10; i++)
        {
            int item = rand.Next(-10, 11);
            sortedList.Add(item);
        }

        Assert.Throws<IndexOutOfRangeException>(() =>
sortedList.GetByIndex(rand.Next(10, 20)));
    }

    [Test]
    public void
GetByIndex_IndexIsZeroInNotEmptySortedList_ReturnIndexOutOfRangeException()
    {
        MySortedList<int> sortedList = new MySortedList<int>();
        Random rand = new Random();
        List<int> list = new List<int>();

        for (int i = 0; i < 10; i++)
```

```
        {
            int item = rand.Next(-10, 11);
            list.Add(item);
            sortedList.Add(item);
        }
        list.Sort();

        Assert.That(sortedList.GetByIndex(0), Is.EqualTo(list[0]));
    }

    [Test]
    public void
GetByIndex_IndexIsCorrectInNotEmptySortedList_ReturnCurrentItem()
    {
        MySortedList<int> sortedList = new MySortedList<int>();
        Random rand = new Random();
        List<int> list = new List<int>();

        for (int i = 0; i < 10; i++)
        {
            int item = rand.Next(-10, 11);
            list.Add(item);
            sortedList.Add(item);
        }
        list.Sort();
        int index = rand.Next(1, 10);

        Assert.That(sortedList.GetByIndex(index), Is.EqualTo(list[index]));
    }
}
```

MyCollectionGetEnumeratorTests.cs:

```
using Lab1;

namespace MyCollection.Tests;

[TestFixture]
public class MyCollectionGetEnumeratorTests
{
    [Test]
    public void
GetEnumerator_EmptySortedList_ReturnMoveNextFalseCurrentDefault()
    {
        MySortedList<int> sortedList = new MySortedList<int>();

        using var enumerator = sortedList.GetEnumerator();
        var resultMoveNext = enumerator.MoveNext();

        Assert.That(resultMoveNext, Is.False);
    }

    [Test]
    public void GetEnumerator_NotEmptySortedList_ReturnMoveNextTrue()
    {
        Random rand = new Random();
        MySortedList<int> sortedList = new MySortedList<int>();
        List<int> list = new List<int>();

        for (int i = 0; i < 10; i++)
        {
            int item = rand.Next(-10, 11);
```

```
            sortedList.Add(item);
        }
        foreach (var item in sortedList) list.Add(item);

        Assert.That(sortedList.ToList(), Is.EqualTo(list));
    }

    [Test]
    public void GetEnumerator_ResetInNotEmptySortedList_ReturnHead()
    {
        Random rand = new Random();
        MySortedList<int> sortedList = new MySortedList<int>();

        sortedList.Add(rand.Next(-10, 11));
        using var enumerator = sortedList.GetEnumerator();
        enumerator.MoveNext();
        enumerator.Reset();

        Assert.That(sortedList.GetByIndex(0),
Is.EqualTo(enumerator.Current));
    }
}
```

MyCollectionGetIndexOfTests.cs:

```
using Lab1;

namespace MyCollection.Tests;

[TestFixture]
public class MyCollectionGetIndexOfTests
{
    [Test]
    public void GetIndexOf_NotContainsItem_ReturnException()
    {
        MySortedList<int> sortedList = new MySortedList<int>();
        Random rand = new Random();

        for (int i = 0; i < 10; i++)
        {
            int item = rand.Next(-10, 11);
            sortedList.Add(item);
        }

        Assert.Throws<Exception>(() => sortedList.GetIndexOf(rand.Next(11,
20)));
    }

    [Test]
    public void GetIndexOf_ContainsItem_ReturnItemIndex()
    {
        MySortedList<int> sortedList = new MySortedList<int>();
        Random rand = new Random();
        List<int> list = new List<int>();

        for (int i = 0; i < 10; i++)
        {
            int item = 10 - i;
            list.Add(item);
            sortedList.Add(item);
        }
        int searchedItem = rand.Next(1, 11);
```

```
        list.Sort();

        Assert.That(sortedList.GetIndexOf(searchedItem),
Is.EqualTo(list.IndexOf(searchedItem)));
    }

    [Test]
    public void GetIndexOf_EmptySortedList_ReturnException()
    {
        MySortedList<int> sortedList = new MySortedList<int>();
        Random rand = new Random();

        Assert.Throws<Exception>(() => sortedList.GetIndexOf(rand.Next(-10,
11)));
    }
}
```

MyCollectionGetValuesArrayTests.cs:

```
using Lab1;

namespace MyCollection.Tests;

[TestFixture]
public class MyCollectionGetValuesArrayTests
{
    [Test]
    public void
GetValuesArray_NotEmptySortedList_ReturnEqualToSortedListArray()
    {
        Random rand = new Random();
        MySortedList<int> sortedList = new MySortedList<int>();
        int[] array = new int[10];

        for (int i = 0; i < 10; i++)
        {
            int item = rand.Next(-10, 11);
            array[i] = item;
            sortedList.Add(item);
        }
        Array.Sort(array);
        var resultArray = sortedList.GetValuesArray();

        Assert.That(resultArray, Is.EqualTo(array));
    }

    [Test]
    public void GetValuesArray_EmptySortedList_ReturnException()
    {
        MySortedList<int> sortedList = new MySortedList<int>();

        Assert.Throws<Exception>(() => sortedList.GetValuesArray() ) ;
    }
}
```

MyCollectionGetValuesListTests.cs:

```
using Lab1;

namespace MyCollection.Tests;
```

```
[TestFixture]
public class MyCollectionGetValuesListTests
{
    [Test]
    public void GetValuesList_NotEmptySortedList_ReturnEmptyList()
    {
        Random rand = new Random();
        MySortedList<int> sortedList = new MySortedList<int>();
        List<int> list = new List<int>();

        for (int i = 0; i < 10; i++)
        {
            int item = rand.Next(-10, 11);
            list.Add(item);
            sortedList.Add(item);
        }
        list.Sort();
        var resultList = sortedList.GetValuesList();

        Assert.That(resultList, Is.EqualTo(list));
    }

    [Test]
    public void GetValuesList_EmptySortedList_ReturnException()
    {
        MySortedList<int> sortedList = new MySortedList<int>();

        Assert.Throws<Exception>(() => sortedList.GetValuesList() ) ;
    }
}
```

MyCollectionIsReadOnlyTests.cs:

```
using Lab1;

namespace MyCollection.Tests;

[TestFixture]
public class MyCollectionIsReadOnlyTests
{
    [Test]
    public void IsReadOnly_EverySortedList_ReturnFalse()
    {
        MySortedList<int> sortedList = new MySortedList<int>();

        Assert.That(sortedList.IsReadOnly, Is.EqualTo(false));
    }
}
```

MyCollectionRemoveAllTests.cs:

```
using Lab1;

namespace MyCollection.Tests;

[TestFixture]
public class MyCollectionRemoveAllTests
{
    [Test]
    public void RemoveAll_NotContainsValueToRemove_ReturnFalse()
    {
```

```
        MySortedList<int> sortedList = new MySortedList<int>();
        Random rand = new Random();

        for (int i = 0; i < 10; i++)
        {
            int item = rand.Next(-10, 11);
            sortedList.Add(item);
        }

        Assert.That(sortedList.RemoveAll(rand.Next(11, 21)),
Is.EqualTo(false));
    }

    [Test]
    public void
RemoveAll_ContainsValueToRemove_ReturnTrueAndSortedListWithoutAllMatchedValue
s()
    {
        MySortedList<int> sortedList = new MySortedList<int>();
        Random rand = new Random();
        List<int> list = new List<int>();

        for (int i = 0; i < 10; i++)
        {
            int item = 10 - i;
            list.Add(item);
            sortedList.Add(item);
        }
        list.Sort();
        int itemToRemove = rand.Next(1, 11);
        while (list.Contains(itemToRemove)) list.Remove(itemToRemove);

        Assert.That(sortedList.RemoveAll(itemToRemove), Is.EqualTo(true));
        Assert.That(sortedList.ToList(), Is.EqualTo(list));
    }
}
```

MyCollectionRemoveByIndexTests.cs:

```
using Lab1;

namespace MyCollection.Tests;

[TestFixture]
public class MyCollectionRemoveByIndexTests
{
    [Test]
    public void RemoveByIndex_EmptySortedList_ReturnException()
    {
        MySortedList<int> sortedList = new MySortedList<int>();
        Random rand = new Random();

        Assert.Throws<Exception>(() => sortedList.RemoveByIndex(rand.Next(0,
11)));
    }

    [Test]
    public void
RemoveByIndex_IndexLessThanZeroWithNotEmptySortedList_ReturnIndexOutOfRangeEx
ception()
    {
        MySortedList<int> sortedList = new MySortedList<int>();
```

```
        Random rand = new Random();

        for (int i = 0; i < 10; i++)
        {
            int item = rand.Next(-10, 11);
            sortedList.Add(item);
        }

        Assert.Throws<IndexOutOfRangeException>(() =>
sortedList.RemoveByIndex(rand.Next(-10, 0)));
    }

    [Test]
    public void
RemoveByIndex_IndexGreaterThanNotEmptySortedListCount_ReturnIndexOutOfRangeEx
ception()
    {
        MySortedList<int> sortedList = new MySortedList<int>();
        Random rand = new Random();

        for (int i = 0; i < 10; i++)
        {
            int item = rand.Next(-10, 11);
            sortedList.Add(item);
        }

        Assert.Throws<IndexOutOfRangeException>(() =>
sortedList.RemoveByIndex(rand.Next(10, 20)));
    }

    [Test]
    public void
RemoveByIndex_IndexGreaterThanZeroWithNotEmptySortedList_ReturnTrueAndSortedL
istWithoutValueByGivenIndex()
    {
        MySortedList<int> sortedList = new MySortedList<int>();
        Random rand = new Random();
        List<int> list = new List<int>();

        for (int i = 0; i < 10; i++)
        {
            int item = rand.Next(-10, 11);
            list.Add(item);
            sortedList.Add(item);
        }
        list.Sort();
        int indexToRemove = rand.Next(1, 10);
        list.RemoveAt(indexToRemove);

        Assert.That(sortedList.RemoveByIndex(indexToRemove),
Is.EqualTo(true));
        Assert.That(sortedList.ToList(), Is.EqualTo(list));
    }

    [Test]
    public void
RemoveByIndex_IndexIsZeroWithNotEmptySortedList_ReturnTrueAndSortedListWithou
tValueByGivenIndex()
    {
        MySortedList<int> sortedList = new MySortedList<int>();
        Random rand = new Random();

        for (int i = 0; i < 10; i++)
        {
```

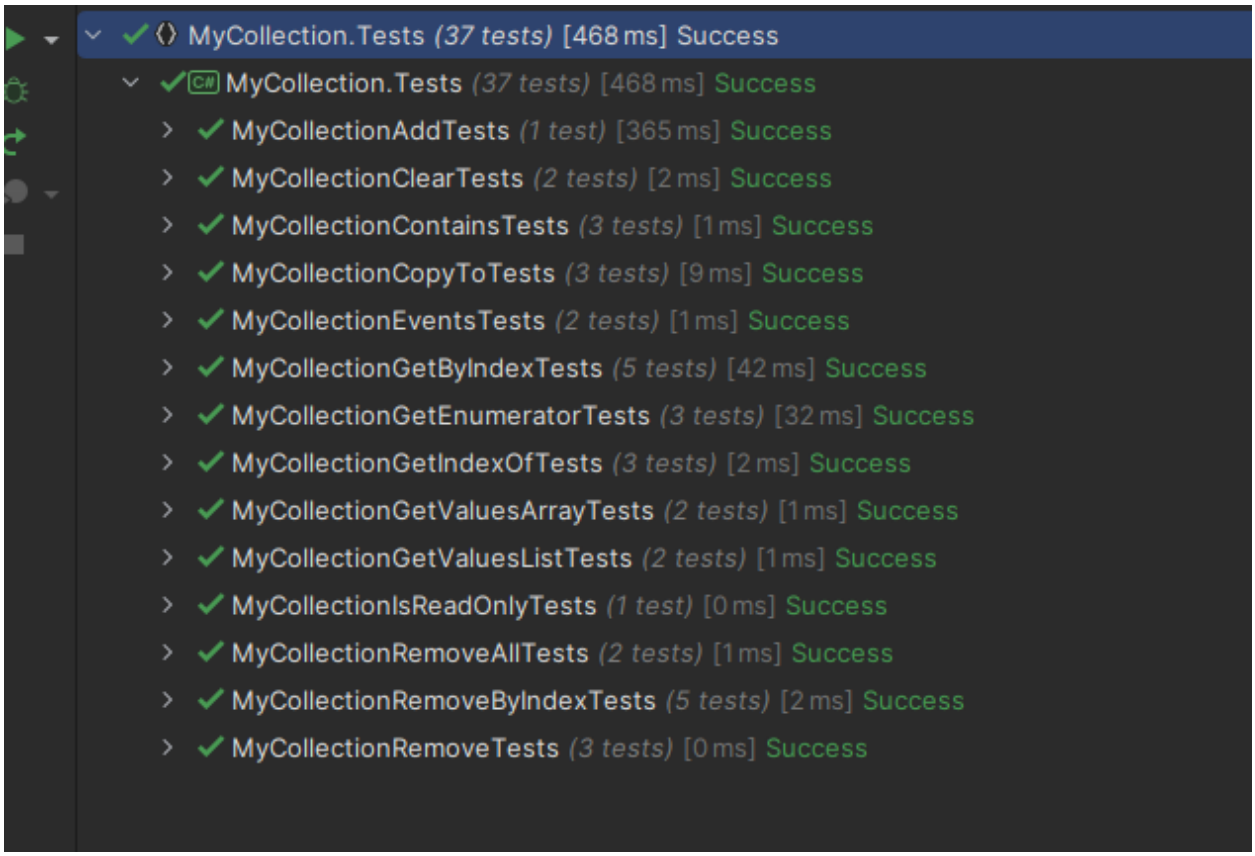
```
            int item = rand.Next(-10, 11);
            sortedList.Add(item);
        }
        int indexToRemove = 0;
        int headNext = sortedList.GetByIndex(1);

        Assert.That(sortedList.RemoveByIndex(indexToRemove),
Is.EqualTo(true));
        Assert.That(sortedList.GetByIndex(0), Is.EqualTo(headNext));
    }
}
```

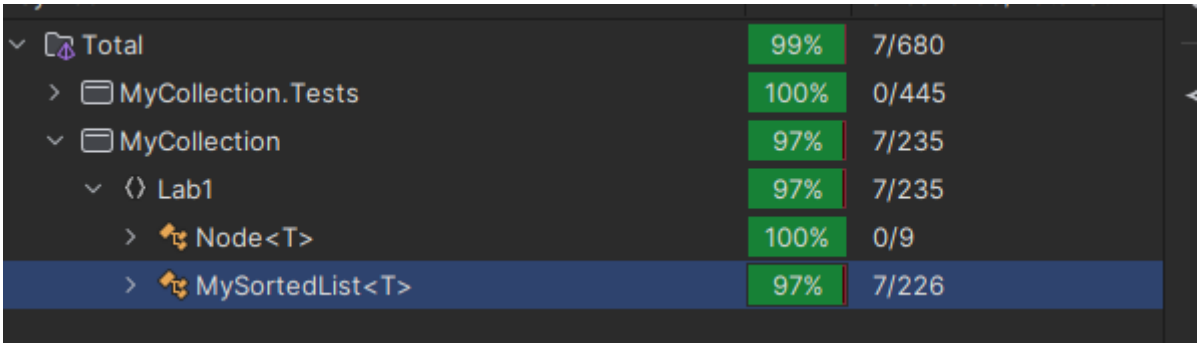
MyCollectionRemoveTests.cs:

```
using Lab1;

namespace MyCollection.Tests;

[TestFixture]
public class MyCollectionRemoveTests
{
    [Test]
    public void Remove_NotContainsValueToRemove_ReturnFalse()
    {
        MySortedList<int> sortedList = new MySortedList<int>();
        Random rand = new Random();

        for (int i = 0; i < 10; i++)
        {
            int item = rand.Next(-10, 11);
            sortedList.Add(item);
        }

        Assert.That(sortedList.Remove(rand.Next(11, 21)), Is.EqualTo(false));
    }
    [Test]
    public void
Remove_ContainsValueToRemove_ReturnTrueAndSortedListWithoutFirstMatchedValue(
)
    {
        MySortedList<int> sortedList = new MySortedList<int>();
        Random rand = new Random();
        List<int> list = new List<int>();

        for (int i = 0; i < 10; i++)
        {
            int item = 10 - i;
            list.Add(item);
            sortedList.Add(item);
        }
        list.Sort();
        int itemToRemove = rand.Next(1, 11);
        list.Remove(itemToRemove);

        Assert.That(sortedList.Remove(itemToRemove), Is.EqualTo(true));
        Assert.That(sortedList.ToList(), Is.EqualTo(list));
    }

    [Test]
    public void
Remove_ValueToRemoveIsHead_ReturnTrueAndNewHeadIsNextElement()
    {
```

```
        MySortedList<int> sortedList = new MySortedList<int>();

        for (int i = 0; i < 10; i++)
        {
            int item = 10 - i;
            sortedList.Add(item);
        }
        int itemToRemove = sortedList.GetByIndex(0);
        int headNext = sortedList.GetByIndex(1);

        Assert.That(sortedList.Remove(itemToRemove), Is.EqualTo(true));
        Assert.That(sortedList.GetByIndex(0), Is.EqualTo(headNext));
    }
}
```

Результати виконання:





Доступ до коду:

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