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»

Виконав *ІС-13,Харчук А.В.* _____
(шифр, прізвище, ім'я, по батькові)

Перевірив *Бардін В.* _____
(прізвище, ім'я, по батькові)

Київ 2023

Варіант 5

Завдання

1	Стек	Див. Stack<T>	Збереження даних за допомогою динамічно зв’язаного списку

Код тестів

```
using System.Collections;
using System.Collections.Generic;

namespace CollectionRealisation;

public class StackTests
{
    public class PropertiesTests
    {
        [Fact]
        public void
Constructor_WithNegativeCapacity_ThrowsArgumentOutOfRangeException()
        {
            // Arrange
            int capacity = -1;

            // Act and Assert
            Assert.Throws<ArgumentOutOfRangeException>(() => new
MyStack<int>(capacity));
        }

        [Fact]
        public void Count_Property_Should_Return_Correct_Value()
        {
            var stack = new MyStack<int>();

            stack.Push(1);
            stack.Push(2);
            stack.Push(3);
```

```
            Assert.Equal(3, stack.Count);
        }
        [Fact]
        public void IsReadOnly_Property_Should_Always_Return_False()
        {
            var stack = new MyStack<int>();

            Assert.Equal(false, stack.IsReadOnly);
        }
        [Fact]
        public void IsSynchronized_Property_Should_Always_Return_False()
        {
            var stack = new MyStack<int>();

            Assert.Equal(false, stack.IsSynchronized);
        }
        [Fact]
        public void SyncRoot_Property_Should_Return_This_Instance()
        {
            var stack = new MyStack<int>();

            Assert.Equal(stack, stack.SyncRoot);
        }
        [Fact]
        public void ToString_Should_Return_Correct_String_Representation()
        {
            var stack = new MyStack<int>();
            stack.Push(1);
            stack.Push(2);

            string result = stack.ToString();

            Assert.Equal("Count = 2", result);
        }
    }
    public class ClearTests
    {
        [Fact]
        public void Clear_Should_Clear_Stack_And_Invoke_Cleared_Event()
        {
            var stack = new MyStack<int>();
```

```
            stack.Push(1);
            stack.Push(2);
            stack.Push(3);

            stack.Clear();

            Assert.Empty(stack);
        }

        [Fact]
        public void Clear_Should_Invoke_Cleared_Event()
        {
            var stack = new MyStack<int>();
            bool clearedEventInvoked = false;
            stack.Cleared += () => clearedEventInvoked = true;

            stack.Push(1);
            stack.Clear();

            Assert.True(clearedEventInvoked);
        }
    }

    public class ContainsTests
    {
        [Fact]
        public void Contains_Should_Return_True_When_Item_Exists()
        {
            var stack = new MyStack<int>();
            stack.Push(1);
            stack.Push(2);
            stack.Push(3);

            Assert.True(stack.Contains(2));
        }

        [Fact]
        public void Contains_Should_Return_False_When_Item_Does_Not_Exist()
        {
            var stack = new MyStack<int>();
            stack.Push(1);
            stack.Push(2);
```

```
            stack.Push(3);

            Assert.False(stack.Contains(4));
        }

        [Fact]
        public void Contains_Should_Return_False_For_Empty_Stack()
        {
            var stack = new MyStack<int>();

            Assert.False(stack.Contains(1));
        }
    }

    public class GetEnumeratorTests
    {
        [Fact]
        public void
GetEnumerator_Should_Return_Enumerator_For_Generic_Enumerable()
        {
            var stack = new MyStack<int>();
            stack.Push(1);
            stack.Push(2);
            stack.Push(3);

            IEnumerator<int> enumerator = stack.GetEnumerator();

            Assert.NotNull(enumerator);
        }

        [Fact]
        public void GetEnumerator_Should_Return_Non_Generic_Enumerator()
        {
            var stack = new MyStack<int>();
            stack.Push(1);
            stack.Push(2);
            stack.Push(3);

            IEnumerator<int> enumerator =
((IEnumerable<int>)stack).GetEnumerator();

            Assert.NotNull(enumerator);
```

```
    }

    [Fact]
    public void GetEnumerator_Should_Traverse_Stack_Elements()
    {
        var stack = new MyStack<int>();
        stack.Push(1);
        stack.Push(2);
        stack.Push(3);

        List<int> elements = new List<int>();
        foreach (int item in stack)
        {
            elements.Add(item);
        }

        Assert.Equal(new List<int> { 3, 2, 1 }, elements);
    }

}

public class PeekTests
{
    [Fact]
    public void Peek_Should_Return_Top_Element_Without_Removing_It()
    {
        var stack = new MyStack<int>();
        stack.Push(1);
        stack.Push(2);
        stack.Push(3);

        var peekedValue = stack.Peek();

        Assert.Equal(3, peekedValue);
        Assert.Equal(3, stack.Count);
    }

    [Fact]
    public void Peek_Should_Throw_InvalidOperationException_On_Empty_Stack()
    {
        var stack = new MyStack<int>();
```

```
        Assert.Throws<InvalidOperationException>(() => stack.Peek());
    }
}
public class PopTests
{
    [Fact]
    public void Pop_Should_Return_Top_Element_And_Remove_It()
    {
        var stack = new MyStack<int>();
        stack.Push(1);
        stack.Push(2);
        stack.Push(3);

        var poppedValue = stack.Pop();

        Assert.Equal(3, poppedValue);
        Assert.Equal(2, stack.Count);
    }

    [Fact]
    public void Pop_Should_Throw_InvalidOperationException_On_Empty_Stack()
    {
        var stack = new MyStack<int>();

        Assert.Throws<InvalidOperationException>(() => stack.Pop());
    }
}

public class PushTests
{
    [Fact]
    public void Push_Should_Add_Item_To_The_Top_Of_The_Stack()
    {
        var stack = new MyStack<int>();

        stack.Push(1);
        stack.Push(2);

        Assert.Equal(2, stack.Count);
        Assert.Equal(2, stack.Peek());
    }
```

```
    [Fact]
    public void Push_Should_Invoke_Pushed_Event()
    {
        var stack = new MyStack<int>();
        int pushedValue = 0;
        stack.Pushed += value => pushedValue = value;

        stack.Push(3);

        Assert.Equal(3, pushedValue);
    }
    [Fact]
    public void Push_Should_Add_Capacity()
    {
        var stack = new MyStack<int>();

        var defaultCapacity = MyStack<int>.DefaultCapacity;

        for (int i = 0; i < defaultCapacity; i++)
        {
            stack.Push(default);
        }
        stack.Push(1);

        Assert.Equal(stack.Count, defaultCapacity+1);
    }
}

public class CopyToTests
{

    [Fact]
    public void CopyTo_Should_Copy_Stack_Elements_To_Array()
    {
        var stack = new MyStack<int>();
        stack.Push(1);
        stack.Push(2);
        stack.Push(3);
        int[] array = new int[5];

        stack.CopyTo(array, 1);
```

```
            Assert.Equal(new int[] { 0, 3, 2, 1, 0 }, array);
        }

        [Fact]
        public void CopyTo_Should_Throw_ArgumentNullException_If_Array_Is_Null()
        {
            var stack = new MyStack<int>();
            int[] array = null!;

            Assert.Throws<ArgumentNullException>(() => stack.CopyTo(array!, 0));
        }

        [Fact]
        public void
CopyTo_Should_Throw_InvalidDataException_If_Array_Rank_Is_Not_1()
        {
            var stack = new MyStack<int>();
            int[,] array = new int[2, 2];

            Assert.Throws<InvalidDataException>(() => stack.CopyTo(array, 0));
        }

        [Fact]
        public void
CopyTo_Should_Throw_ArgumentException_If_Array_Type_Mismatch()
        {
            var stack = new MyStack<int>();
            double[] array = new double[5];

            Assert.Throws<ArgumentException>(() => stack.CopyTo(array, 0));
        }
        [Fact]
        public void CopyTo_Should_Copy_Empty_Stack_To_Array()
        {
            var stack = new MyStack<int>();
            int[] array = new int[5];

            stack.CopyTo(array, 1);

            Assert.Equal(new int[] { 0, 0, 0, 0, 0 }, array);
        }
    }
```

```
public class IEnumeratorTests
{
    [Fact]
    public void Current_Property_Returns_Current_Element()
    {
        var stack = new MyStack<int>();
        stack.Push(1);
        stack.Push(2);
        var enumerator = stack.GetEnumerator();
        enumerator.MoveNext();
        var current = enumerator.Current;

        Assert.Equal(2, current);
    }

    [Fact]
    public void IEnumerator_Current_Property_Returns_Current_Element()
    {
        var stack = new MyStack<int>();
        stack.Push(1);
        stack.Push(2);
        var enumerator = stack.GetEnumerator();
        enumerator.MoveNext();
        var current = (enumerator as IEnumerator)?.Current;

        Assert.Equal(2, current);
    }

    [Fact]
    public void MoveNext_Should_Move_To_Next_Element()
    {
        var stack = new MyStack<int>();
        stack.Push(1);
        stack.Push(2);
        var enumerator = stack.GetEnumerator();

        var result1 = enumerator.MoveNext();
        var result2 = enumerator.MoveNext();
        var result3 = enumerator.MoveNext();

        Assert.True(result1);
```

```
            Assert.True(result2);
            Assert.False(result3);
        }

        [Fact]
        public void Reset_Should_Reset_Enumerator()
        {
            var stack = new MyStack<int>();
            stack.Push(1);
            stack.Push(2);
            var enumerator = stack.GetEnumerator();
            enumerator.MoveNext();

            var firstCurrent = enumerator.Current;

            enumerator.Reset();

            enumerator.MoveNext();

            var secondCurrent = enumerator.Current;

            Assert.Equal(firstCurrent, secondCurrent);
        }
    }
}
```