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 3

Виконав студент ІП-13 Макарчук Лідія Олександрівна
(шифр, прізвище, ім'я, по батькові)

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

Київ 2023

Лабораторна робота №2

Варіант 3

Тема: Модульне тестування. Ознайомлення з засобами та практиками модульного тестування.

Мета: навчитися створювати модульні тести для вихідного коду розроблювального програмного забезпечення.

Постановка задачі

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.

3	Бінарне дерево	Додавання вузлів, обходи дерева, перевірка на наявність, пошук(видалення реалізовувати не обов'язково)	Збереження даних за допомогою динамічно зв'язаних вузлів

Виконання завдань

Код

//BinaryTreeBaseTests.cs

```
namespace MyBinaryTree.Tests.Base;

public abstract class BinaryTreeBaseTests
{
    protected static readonly int[] _intValues = new int[] { 3, 5, 2, -1, 4 };

    protected static readonly int[] _intInorder = new int[] { -1, 2, 3, 4, 5 };

    protected static readonly string[] _stringValues = new string[] { "Alice", "Bob",
"Oliver", "Andrew" };
    protected static readonly string[] _stringInorderDefaultComparer = new string[] {
"Alice", "Andrew", "Bob", "Oliver" };

    public static IEnumerable<object[]> GetDataForInitializing()
    {
        yield return new object[] { _intValues };
        yield return new object[] { _stringValues };
    }
    public static IEnumerable<object[]> GetValuesThatContainItem()
```

```
    {
        yield return new object[] { _intValues, -1 };
        yield return new object[] { _stringValues, "Alice" };
    }

    public static IEnumerable<object[]> GetValuesThatNotContainItem()
    {
        yield return new object[] { _intValues, 1 };
        yield return new object[] { _stringValues, "a1234556" };
    }
}
```

// BinaryTreeAddTests.cs

```
using MyBinaryTree.Tests.Base;
using Xunit;

namespace MyBinaryTree.Tests;

public class BinaryTreeAddTests : BinaryTreeBaseTests
{
    public static IEnumerable<object[]> GetDataWithInorderResult()
    {
        yield return new object[] { _intValues, _intInorder };
        yield return new object[] { _stringValues, _stringInorderDefaultComparer };
    }

    public static IEnumerable<object[]> GetTwoValuesWhereSecondIsBigger()
    {
        yield return new object[] { 1, 2 };
        yield return new object[] { "Alice", "Bob" };
    }

    public static IEnumerable<object[]> GetTwoValuesWhereSecondIsSmaller()
    {
        yield return new object[] { 2, 1 };
        yield return new object[] { "Bob", "Alice" };
    }

    [Fact]
    public void Add_WhenItemIsNull_ShouldThrow()
    {
        var tree = new BinaryTree<string>();

        var act = () => tree.Add(null!);

        Assert.Throws<ArgumentNullException>(act);
    }

    [Theory]
    [MemberData(nameof(GetTwoValuesWhereSecondIsBigger))]
    public void Add_Add2ItemsAndNextItemBiggerThanRoot_SecondItemShouldBeRightChild<T>(T
rootItem, T nextItem) where T : IComparable<T>
    {
        var tree = new BinaryTree<T>();
        tree.Add(rootItem);

        tree.Add(nextItem);

        Assert.NotNull(tree.Root?.Right);
        Assert.Equal(nextItem, tree.Root.Right.Value);
        Assert.Null(tree.Root?.Left);
    }

    [Theory]
    [MemberData(nameof(GetTwoValuesWhereSecondIsSmaller))]
    public void Add_Add2ItemsAndNextItemSmallerThanRoot_SecondItemShouldBeLeftChild<T>(T
rootItem, T nextItem) where T : IComparable<T>
```

```
    {
        var tree = new BinaryTree<T>();
        tree.Add(rootItem);

        tree.Add(nextItem);

        Assert.NotNull(tree.Root?.Left);
        Assert.Equal(nextItem, tree.Root.Left.Value);
        Assert.Null(tree.Root?.Right);
    }

    [Theory]
    [MemberData(nameof(GetDataForInitializing))]
    public void Add_WhenItemIsAlreadyInTree_ShouldThrow<T>(T[] items) where T :
IComparable<T>
    {
        var tree = new BinaryTree<T>(items);
        T itemThatAlreadyInTree = items[0];

        var act = () => tree.Add(itemThatAlreadyInTree);

        Assert.Throws<InvalidOperationException>(act);
    }

    [Theory]
    [MemberData(nameof(GetDataWithInorderResult))]
    public void Add_WhenTreeIsEmpty_ShouldAdd<T>(T[] items, T[] expectedInorder) where T
: IComparable<T>
    {
        var tree = new BinaryTree<T>();
        var expectedVerstion = expectedInorder.Length;

        foreach (var item in items)
        {
            tree.Add(item);
        }

        Assert.Multiple(
                () => Assert.True(tree.SequenceEqual(expectedInorder)),
                () => Assert.Equal(tree.Count, expectedInorder.Length),
                () => Assert.Equal(tree.Version, expectedVerstion)
        );
    }

    [Fact]
    public void Add_WhenCustomComparerIsUsedAndItemInTree_ShouldThrow()
    {
        var tree = new BinaryTree<string>(StringComparer.OrdinalIgnoreCase) { "Alice",
"Bob" };

        var act = () => tree.Add("alice");

        Assert.Throws<InvalidOperationException>(act);
    }
}
```

// BinaryTreeClearTests.cs

```
using MyBinaryTree.Tests.Base;
using Xunit;

namespace MyBinaryTree.Tests;

public class BinaryTreeClearTests : BinaryTreeBaseTests
{
    [Theory]
    [MemberData(nameof(GetDataForInitializing))]
    public void Clear_WhenTreeHasItems_TreeShouldBeEmpty<T>(T[] items) where T :
IComparable<T>
    {
```

```
        var tree = new BinaryTree<T>(items);
        var expectedVersion = tree.Version + 1;

        tree.Clear();

        Assert.Multiple(
            () => Assert.Empty(tree),
            () => Assert.Equal(expectedVersion, tree.Version)
        );
    }
}
```

// BinaryTreeContainsTests.cs

```
using MyBinaryTree.Tests.Base;
using Xunit;

namespace MyBinaryTree.Tests;

public class BinaryTreeContainsTests : BinaryTreeBaseTests
{

    [Theory]
    [MemberData(nameof(GetValuesThatContainItem))]
    public void Contains_WhenItemIsInTree_ShouldBeTrue<T>(T[] items, T itemInTree) where
T : IComparable<T>
    {
        var tree = new BinaryTree<T>(items);

        var contains = tree.Contains(itemInTree);

        Assert.True(contains);
    }

    [Theory]
    [MemberData(nameof(GetValuesThatNotContainItem))]
    public void Contains_WhenItemIsNotInTree_ShouldBeFalse<T>(T[] items, T
itemThatNotInTree) where T : IComparable<T>
    {
        var tree = new BinaryTree<T>(items);

        var contains = tree.Contains(itemThatNotInTree);

        Assert.False(contains);
    }

    [Fact]
    public void Contains_WhenItemIsNull_ShouldBeFalse()
    {
        var tree = new BinaryTree<string>() { "David", "Bob", "Alice" };

        var contains = tree.Contains(null!);

        Assert.False(contains);
    }

}
```

// BinaryTreeCopyToTests.cs

```
using MyBinaryTree.Tests.Base;
using Xunit;

namespace MyBinaryTree.Tests;

public class BinaryTreeCopyToTests : BinaryTreeBaseTests
{
    public static IEnumerable<object[]>
GetDataAndArraySizeAndStartPositionWithEnoughtSpace()
    {
        yield return new object[] { _intValues, 5, 0 };
```

```
        yield return new object[] { _intValues, 6, 1 };

        yield return new object[] { _stringValues, 8, 0 };
    }

    public static IEnumerable<object[]>
GetDataAndArraySizeAndStartPositionWithNotEnoughtSpace()
    {
        yield return new object[] { _intValues, 2, 0 };
        yield return new object[] { _intValues, 5, 1 };
        yield return new object[] { _intValues, 6, 3 };

        yield return new object[] { _stringValues, 2, 0 };
    }

    [Fact]
    public void CopyTo_WhenArrayIsNull_ShouldThrow()
    {
        var tree = new BinaryTree<int>() { 1, 2 };
        int arrayStartIndex = 0;

        var act = () => tree.CopyTo(null!, arrayStartIndex);

        Assert.Throws<ArgumentNullException>(act);
    }

    [Fact]
    public void CopyTo_WhenArrayStartIndexIsNegative_ShouldThrow()
    {
        var tree = new BinaryTree<int>() { 1, 2 };
        var array = new int[2];
        int arrayStartIndex = -1;

        var act = () => tree.CopyTo(array, arrayStartIndex);

        Assert.Throws<ArgumentOutOfRangeException>(act);
    }

    [Fact]
    public void CopyTo_WhenArrayStartIndexIsBiggerThanSize_ShouldThrow()
    {
        var tree = new BinaryTree<int>() { 1, 2 };
        var array = new int[2];
        int arrayStartIndex = 3;

        var act = () => tree.CopyTo(array, arrayStartIndex);

        Assert.Throws<ArgumentOutOfRangeException>(act);
    }

    [Theory]
    [MemberData(nameof(GetDataAndArraySizeAndStartPositionWithEnoughtSpace))]
    public void CopyTo_WhenEnoughSpace_ShouldCopy<T>(T[] items, int arraySize, int
startIndex) where T : IComparable<T>
    {
        var tree = new BinaryTree<T>(items);
        var array = new T[arraySize];

        tree.CopyTo(array, startIndex);

        foreach (var item in tree)
        {
            Assert.Equal(array[startIndex], item);
            startIndex++;
        }
    }

    [Theory]
    [MemberData(nameof(GetDataAndArraySizeAndStartPositionWithNotEnoughtSpace))]
```

```
    public void CopyTo_WhenNotEnoughSpace_ShouldThrow<T>(T[] items, int arraySize, int
startIndex) where T : IComparable<T>
    {
        var tree = new BinaryTree<T>(items);
        var array = new T[arraySize];

        var act = () => tree.CopyTo(array, startIndex);

        Assert.Throws<ArgumentOutOfRangeException>(act);
    }

    [Fact]
    public void CopyTo_WhenTreeIsEmpty_ArrayShoudBeEmpty()
    {
        var tree = new BinaryTree<string>();
        var array = new string[2];
        var startIndex = 0;

        tree.CopyTo(array, startIndex);

        Assert.True(array.SequenceEqual(new string[] { null!, null! }));
    }
}
```

// BinaryTreeEventsTests.cs

```
using Xunit;
using FakeItEasy;
using MyBinaryTree.Tests.Base;

namespace MyBinaryTree.Tests;

public class BinaryTreeEventsTests : BinaryTreeBaseTests
{
    [Theory]
    [MemberData(nameof(GetValuesThatNotContainItem))]
    public void Add_WhenTreeIsNotEmptyAndOneSubscriber_ShouldCallOneMethod<T>(T[] items,
T itemToAdd) where T : IComparable<T>
    {
        var tree = new BinaryTree<T>(items);
        var callback = A.Fake<EventHandler<BinaryTreeEventArgs<T>>>();
        tree.ItemAdded += callback;

        tree.Add(itemToAdd);

        A.CallTo(() => callback(A<object?>._, A<BinaryTreeEventArgs<T>>._))
            .MustHaveHappenedOnceExactly();
    }

    [Theory]
    [MemberData(nameof(GetValuesThatNotContainItem))]
    public void Add_WhenTreeIsNotEmptyAndTwoSubscribers_ShouldCallTwoMethods<T>(T[]
items, T itemToAdd) where T : IComparable<T>
    {
        var tree = new BinaryTree<T>(items);
        var callback = A.Fake<EventHandler<BinaryTreeEventArgs<T>>>();
        tree.ItemAdded += callback;
        tree.ItemAdded += callback;

        tree.Add(itemToAdd);

        A.CallTo(() => callback(A<object?>._, A<BinaryTreeEventArgs<T>>._))
            .MustHaveHappenedTwiceExactly();
    }

    [Theory]
    [MemberData(nameof(GetValuesThatNotContainItem))]
    public void Add_WhenTreeIsNotEmptyAndOneSubscriber_ArgsItemShouldBeValid<T>(T[]
items, T itemToAdd) where T : IComparable<T>
    {
```

```
        var tree = new BinaryTree<T>(items);
        var callback = A.Fake<EventHandler<BinaryTreeEventArgs<T>>>();
        tree.ItemAdded += callback;

        tree.Add(itemToAdd);

        A.CallTo(() => callback(A<object?>._, A<BinaryTreeEventArgs<T>>._))
            .WhenArgumentsMatch((object? _, BinaryTreeEventArgs<T> treeArgs) =>
treeArgs.Item.CompareTo(itemToAdd) == 0)
            .MustHaveHappened();
    }

    [Theory]
    [MemberData(nameof(GetValuesThatContainItem))]
    public void Remove_WhenTreeIsNotEmptyAndOneSubscriber_ShouldCallOneMethod<T>(T[]
items, T ItemRemoved) where T : IComparable<T>
    {
        var tree = new BinaryTree<T>(items);
        var callback = A.Fake<EventHandler<BinaryTreeEventArgs<T>>>();
        tree.ItemRemoved += callback;

        tree.Remove(ItemRemoved);

        A.CallTo(() => callback(A<object?>._, A<BinaryTreeEventArgs<T>>._))
            .MustHaveHappenedOnceExactly();
    }

    [Theory]
    [MemberData(nameof(GetValuesThatContainItem))]
    public void Remove_WhenTreeIsNotEmptyAndTwoSubscribers_ShouldCallTwoMethods<T>(T[]
items, T ItemRemoved) where T : IComparable<T>
    {
        var tree = new BinaryTree<T>(items);
        var callback = A.Fake<EventHandler<BinaryTreeEventArgs<T>>>();
        tree.ItemRemoved += callback;
        tree.ItemRemoved += callback;

        tree.Remove(ItemRemoved);

        A.CallTo(() => callback(A<object?>._, A<BinaryTreeEventArgs<T>>._))
            .MustHaveHappenedTwiceExactly();
    }

    [Theory]
    [MemberData(nameof(GetValuesThatContainItem))]
    public void Remove_WhenTreeIsNotEmptyAndOneSubscriber_ArgsItemShouldBeValid<T>(T[]
items, T ItemRemoved) where T : IComparable<T>
    {
        var tree = new BinaryTree<T>(items);
        var callback = A.Fake<EventHandler<BinaryTreeEventArgs<T>>>();
        tree.ItemRemoved += callback;

        tree.Remove(ItemRemoved);

        A.CallTo(() => callback(A<object?>._, A<BinaryTreeEventArgs<T>>._))
            .WhenArgumentsMatch((object? _, BinaryTreeEventArgs<T> treeArgs) =>
treeArgs.Item.CompareTo(ItemRemoved) == 0)
            .MustHaveHappened();
    }

    [Fact]
    public void Clear_WhenTreeIsNotEmptyAndOneSubscriber_ShouldCallOneMethod()
    {
        var tree = new BinaryTree<int>() { 1, 3, 2 };
        var callback = A.Fake<EventHandler>();
        tree.TreeCleared += callback;

        tree.Clear();
```

```
        A.CallTo(() => callback(A<object?>._, A<EventArgs>._))
            .MustHaveHappenedOnceExactly();
    }

    [Fact]
    public void Clear_WhenTreeIsNotEmptyAndTwoSubscribers_ShouldCallTwoMethods()
    {
        var tree = new BinaryTree<int>() { 1, 3, 2 };
        var callback = A.Fake<EventHandler>();
        tree.TreeCleared += callback;
        tree.TreeCleared += callback;

        tree.Clear();

        A.CallTo(() => callback(A<object?>._, A<EventArgs>._))
            .MustHaveHappenedTwiceExactly();
    }
}
```

// BinaryTreeOtherTests.cs

```
using MyBinaryTree.EnumeratorFactories;
using MyBinaryTree.Interfaces;
using Xunit;

namespace MyBinaryTree.Tests;

public class BinaryTreeOtherTests
{
    private static readonly int[] _initialData = { 4, 3, 1, 5, 2 };

    public static IEnumerable<object[]> GetEnumeratorFactoriesAndResultSequence()
    {
        yield return new object[] { new PreorderEnumeratorFactory<int>(), new
InorderEnumeratorFactory<int>(), _initialData, new int[] { 1, 2, 3, 4, 5 } };
        yield return new object[] { new InorderEnumeratorFactory<int>(), new
PreorderEnumeratorFactory<int>(), _initialData, new int[] { 4, 3, 1, 2, 5 } };
        yield return new object[] { new InorderEnumeratorFactory<int>(), new
PostorderEnumeratorFactory<int>(), _initialData, new int[] { 2, 1, 3, 5, 4 } };
    }

    [Theory]
    [MemberData(nameof(GetEnumeratorFactoriesAndResultSequence))]
    public void SetEnumeratorFactory_WhenTreeIsNotEmpty_ShouldReturnSpecifiedSequence<T>(
        IEnumeratorFactory<T> initialFactory,
        IEnumeratorFactory<T> factoryToSet,
        T[] items,
        T[] expectedSequence) where T : IComparable<T>
    {
        var tree = new BinaryTree<T>(initialFactory, items);

        tree.EnumeratorFactory = factoryToSet;

        Assert.True(tree.SequenceEqual(expectedSequence));
    }

    [Fact]
    public void SetEnumeratorFactory_WhenFactoryIsNull_ShouldThrow()
    {
        var tree = new BinaryTree<int>();

        var act = () => tree.EnumeratorFactory = null!;

        Assert.Throws<ArgumentNullException>(act);
    }
}
```

// BinaryTreeRemoveTests.cs

```
using MyBinaryTree.Tests.Base;
using Xunit;
```

```
namespace MyBinaryTree.Tests;

public class BinaryTreeRemoveTests : BinaryTreeBaseTests
{
    public static IEnumerable<object[]> GetDataAndDataToDeleteAndDataToLeftInorder()
    {
        yield return new object[] { _intValues, new int[] { 5, 4 }, new int[] { -1, 2, 3
} };
        yield return new object[] { _intValues, new int[] { 3, -1 }, new int[] { 2, 4, 5
} };
        yield return new object[] { _stringValues, new string[] { "Bob" }, new string[] {
"Alice", "Andrew", "Oliver" } };
    }

    [Fact]
    public void Remove_WhenItemIsNull_ShouldNotRemove()
    {
        var tree = new BinaryTree<string>() { "David", "Bob", "Alice" };
        var expectedCountAfterRemove = tree.Count;
        var expectedVersion = tree.Version;

        var removed = tree.Remove(null!);

        AssertNotRemoved<string>(tree, removed, expectedCountAfterRemove,
expectedVersion);
    }

    [Theory]
    [MemberData(nameof(GetValuesThatContainItem))]
    public void Remove_WhenItemIsInTree_ShouldRemove<T>(T[] items, T itemThatInTree)
where T : IComparable<T>
    {
        var tree = new BinaryTree<T>(items);
        var expectedCountAfterRemove = tree.Count - 1;
        var expectedVersion = tree.Version + 1;

        var removed = tree.Remove(itemThatInTree);

        AssertRemoved<T>(tree, removed, expectedCountAfterRemove, expectedVersion);
    }

    [Theory]
    [MemberData(nameof(GetValuesThatNotContainItem))]
    public void Remove_WhenItemIsNotInTree_ShouldNotRemove<T>(T[] items, T
itemThatNotInTree) where T : IComparable<T>
    {
        var tree = new BinaryTree<T>(items);
        var expectedCountAfterRemove = tree.Count;
        var expectedVersion = tree.Version;

        var removed = tree.Remove(itemThatNotInTree);

        AssertNotRemoved<T>(tree, removed, expectedCountAfterRemove, expectedVersion);
    }

    [Theory]
    [MemberData(nameof(GetDataForInitializing))]
    public void Remove_WhenRemoveAll_ShouldBeEmpty<T>(T[] items) where T : IComparable<T>
    {
        var tree = new BinaryTree<T>(items);

        foreach (var item in items)
        {
            tree.Remove(item);
        }

        Assert.Empty(tree);
    }
```

```
    [Theory]
    [MemberData(nameof(GetDataAndDataToDeleteAndDataToLeftInorder))]
    public void
Remove_WhenRemoveSpecifiedItems_ShouldContainOtherSpecifiedItemsInorder<T>(
        T[] initialData,
        T[] itemsToDelete,
        T[] itemsToLeftInorder) where T : IComparable<T>
    {
        var tree = new BinaryTree<T>(initialData);

        foreach (var item in itemsToDelete)
        {
            tree.Remove(item);
        }

        Assert.True(tree.SequenceEqual(itemsToLeftInorder));
    }

    [Fact]
    public void Remove_WhenTreeIsEmpty_ShouldNotRemove()
    {
        var tree = new BinaryTree<int>();
        int itemToRemove = 0;
        var expectedCountAfterRemove = tree.Count;
        var expectedVersion = tree.Version;

        var removed = tree.Remove(itemToRemove);

        AssertNotRemoved<int>(tree, removed, expectedCountAfterRemove, expectedVersion);
    }

    private static void AssertRemoved<T>(BinaryTree<T> tree, bool removed, int
expectedCountAfterRemove, int expectedVersion) where T : IComparable<T>
    {
        Assert.Multiple(
            () => Assert.True(removed),
            () => Assert.Equal(expectedCountAfterRemove, tree.Count),
            () => Assert.Equal(expectedVersion, tree.Version)
        );
    }

    private static void AssertNotRemoved<T>(BinaryTree<T> tree, bool removed, int
expectedCountAfterRemove, int expectedVersion) where T : IComparable<T>
    {
        Assert.Multiple(
            () => Assert.False(removed),
            () => Assert.Equal(expectedCountAfterRemove, tree.Count),
            () => Assert.Equal(expectedVersion, tree.Version)
        );
    }
}
```

// EnumeratorFactoriesTests.cs

```
using MyBinaryTree.EnumeratorFactories;
using MyBinaryTree.Interfaces;
using System.Collections;
using Xunit;

namespace MyBinaryTree.Tests.EnumeratorsTests;

public class EnumeratorFactoriesTests
{
    private static readonly int[] _initialData = { 4, 3, 1, 5, 2 };
    public static IEnumerable<object[]> GetEnumeratorFactories()
    {
        yield return new object[] { new InorderEnumeratorFactory<int>() };
        yield return new object[] { new PreorderEnumeratorFactory<int>() };
        yield return new object[] { new PostorderEnumeratorFactory<int>() };
```

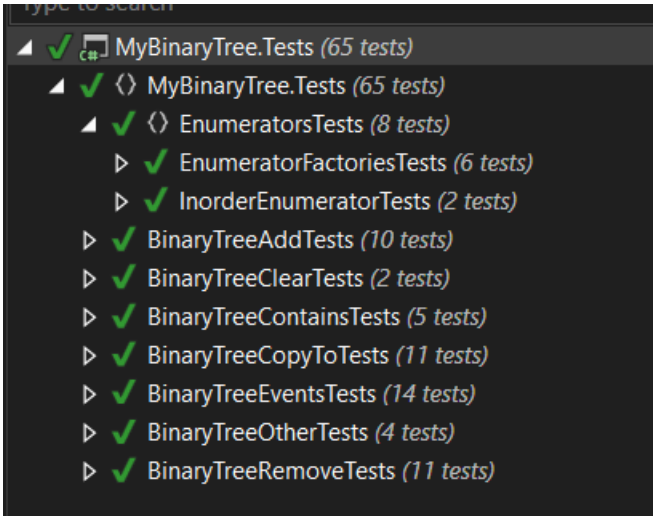
```
    }

    public static IEnumerable<object[]> GetEnumeratorFactoriesAndResultSequence()
    {
        yield return new object[] { new InorderEnumeratorFactory<int>(), _initialData,
new int[] { 1, 2, 3, 4, 5 } };
        yield return new object[] { new PreorderEnumeratorFactory<int>(), _initialData,
new int[] { 4, 3, 1, 2, 5 } };
        yield return new object[] { new PostorderEnumeratorFactory<int>(), _initialData,
new int[] { 2, 1, 3, 5, 4 } };
    }

    [Theory]
    [MemberData(nameof(GetEnumeratorFactories))]
    public void
MoveNext_WhenTreeIsNotEmpty_ShouldBeTrueWhileNodesAvailable(IEnumeratorFactory<int>
factory)
    {
        var tree = new BinaryTree<int>(factory) { 3, 1, 2 };
        int nodesNumber = 3;

        var enumerator = tree.GetEnumerator();

        for (int i = 0; i < nodesNumber; i++)
        {
            Assert.True(enumerator.MoveNext());
        }
        Assert.False(enumerator.MoveNext());
    }

    [Theory]
    [MemberData(nameof (GetEnumeratorFactoriesAndResultSequence))]
    public void CreateEnumerator_WhenTreeIsNotEmpty_ShouldReturnSpecifiedSequence(
        IEnumeratorFactory<int> factory,
        int[] items,
        int[] expectedSequence)
    {
        // arrange
        var tree = new BinaryTree<int>(factory, items);

        // act
        var enumerator = tree.GetEnumerator();

        // assert
        foreach (var item in expectedSequence)
        {
            enumerator.MoveNext();
            Assert.Equal(item, enumerator.Current);
            Assert.Equal(item, ((IEnumerator)enumerator).Current);
        }
    }
}
```

// InorderEnumeratorTests.cs

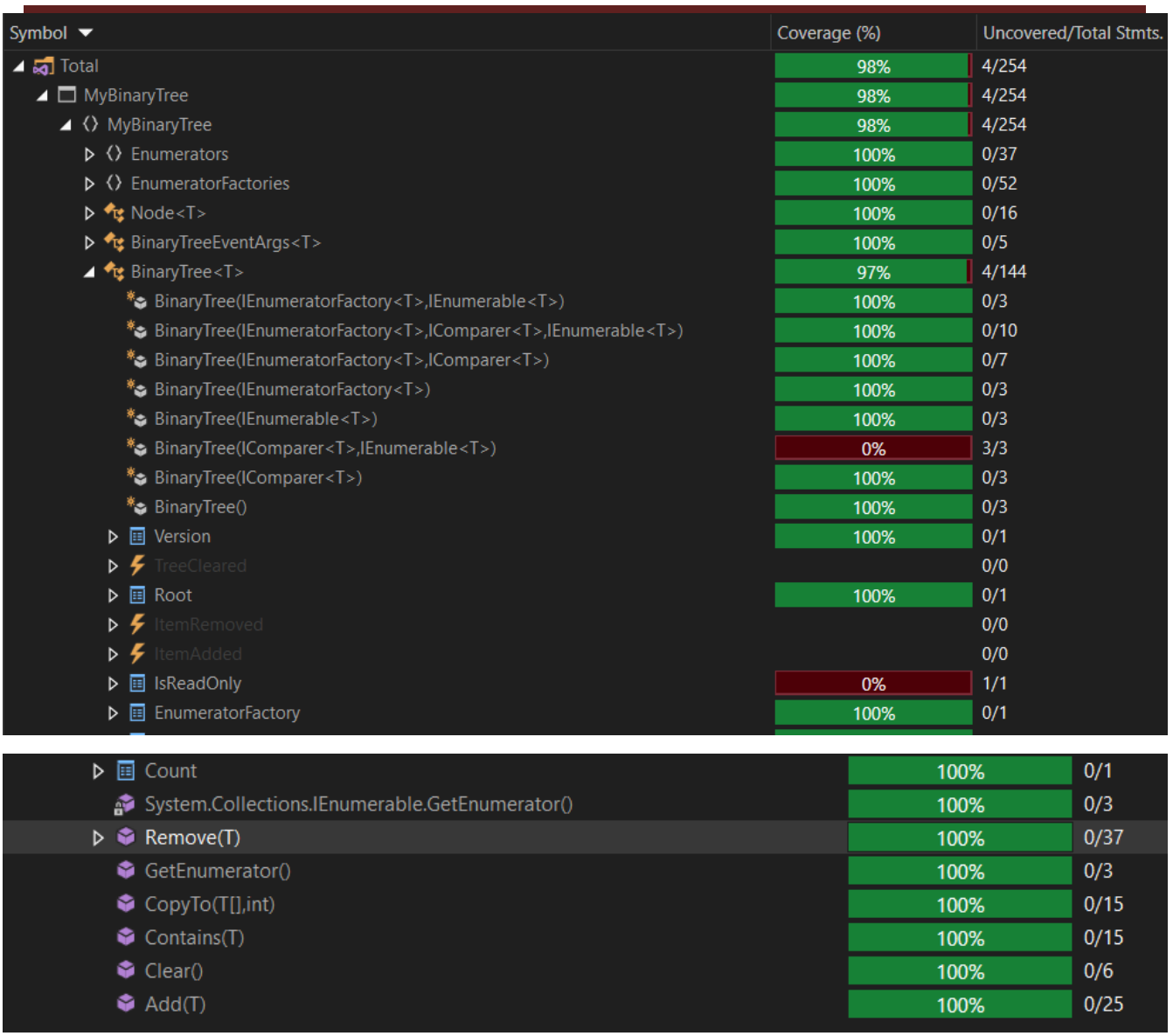
```
using Xunit;

namespace MyBinaryTree.Tests.EnumeratorsTests;

public class InorderEnumeratorTests
{
    [Fact]
    public void Reset_WhenTreeWasNotchanged_ShouldSetCurrentToRoot()
    {
        // arrange
        var tree = new BinaryTree<int>() { 2, 5, 3 };
        var root = tree.Root;
        var enumerator = tree.GetEnumerator();

        while (enumerator.MoveNext()) { }
```

```
        // act
        enumerator.Reset();

        // assert
        Assert.Equal(root!.Value, enumerator.Current);
    }

    [Fact]
    public void Reset_WhenTreeWasChanged_ShouldThrow()
    {
        // arrange
        var tree = new BinaryTree<int>() { 2, 5, 3 };
        var root = tree.Root;
        var enumerator = tree.GetEnumerator();

        while (enumerator.MoveNext()) { }
        tree.Add(-2);

        // act
        var act = () => enumerator.Reset();

        // assert
        Assert.Throws<InvalidOperationException>(act);
    }
}
```

Скріншот запуску модульних тестів:



Скріншоти покриття модульних тестів:



Symbol	Coverage (%)	Uncovered/Total Stmts.
Total	98%	4/254
MyBinaryTree	98%	4/254
MyBinaryTree	98%	4/254
Enumerators	100%	0/37
EnumeratorFactories	100%	0/52
Node<T>	100%	0/16
BinaryTreeEventArgs<T>	100%	0/5
BinaryTree<T>	97%	4/144
BinaryTree(IEnumeratorFactory<T>,IEnumerable<T>)	100%	0/3
BinaryTree(IEnumeratorFactory<T>,IComparer<T>,IEnumerable<T>)	100%	0/10
BinaryTree(IEnumeratorFactory<T>,IComparer<T>)	100%	0/7
BinaryTree(IEnumeratorFactory<T>)	100%	0/3
BinaryTree(IEnumerable<T>)	100%	0/3
BinaryTree(IComparer<T>,IEnumerable<T>)	0%	3/3
BinaryTree(IComparer<T>)	100%	0/3
BinaryTree()	100%	0/3
Version	100%	0/1
TreeCleared		0/0
Root	100%	0/1
ItemRemoved		0/0
ItemAdded		0/0
IsReadOnly	0%	1/1
EnumeratorFactory	100%	0/1
Count	100%	0/1
System.Collections.IEnumerable.GetEnumerator()	100%	0/3
Remove(T)	100%	0/37
GetEnumerator()	100%	0/3
CopyTo(T[],int)	100%	0/15
Contains(T)	100%	0/15
Clear()	100%	0/6
Add(T)	100%	0/25

Висновки:

Висновок: під час виконання лабораторної роботи я розробила модульні тести для функціоналу попередньо створеної колекції за допомогою xUnit та FakeItEasy. Після написання було досліджено ступінь покриття вихідного коду модульними тестами за допомогою dotCover. В результаті покриття коду тестами становить 98%.