Міністерство освіти і науки України

КПІ ім. Ігоря Сікорського

Факультет інформатики та обчислювальної техніки

Кафедра інформаційних систем та технологій

Лабораторна робота № 2

З дисципліни “Сучасні технології розробки WEB-застосувань на платформі Microsoft.Net ”

студентки ІІ курсу ФІОТ

групи ІК-12

Макарчук Ольги

Перевірив:
Бардін В.

Київ 2023

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

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

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.

Варіант 3:

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

Посилання на код GitHub:

https://github.com/olha-makarchuk/DotNet_lab1-2

Код

```
using DotNet_lab1_2;

namespace BinaryTreeCollectionTests
{
    public abstract class BinaryTreeTests
    {
        private static readonly int[] Numbers = { 4, 5, 1, 3, 2 };
        private static readonly string[] Letters = { "s", "ac", "ab", "p" };
        private static readonly TestObject<int>[] Objects = { new(4), new(6),
new(1), new(3), new(0) };


        public static IEnumerable<object[]> GetEmptyTreeTestData()
        {
            yield return new object[] { new BinaryTree<int>() };
            yield return new object[] { new BinaryTree<string>() };
            yield return new object[] { new BinaryTree<TestObject<int>>() };
        }

        public static IEnumerable<object[]> GetTreeData()
        {
            yield return new object[] { Numbers };
            yield return new object[] { Letters };
            yield return new object[] { Objects };
        }

        protected class TestObject<T> : IComparable<TestObject<T>>
        {
            public int Value { get; set; }

            public TestObject(int value)
            {
                Value = value;
            }

            public int CompareTo(TestObject<T> other)
            {
                return Value.CompareTo(other.Value);
            }
        }
    }
}

using DotNet_lab1_2;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace BinaryTreeCollectionTests
{
    public class Add_Tests: BinaryTreeTests
    {
        [Theory]
        [MemberData(nameof(GetTreeData))]
        public void Add_WhenElementsAreContained_ShouldConteined<T>(T[] values)
        {
            var tree = new BinaryTree<T>();

            foreach (var value in values)
            {
                tree.Add(value);
            }
            foreach (var value in values)
            {
                Assert.Contains(value, tree);
```

```
            }
        }

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

            tree.Add(1);
            tree.Add(2);
            var containerResult = tree.Contains(7);
            Assert.False(containerResult);
        }

        [Fact]
        public void Add_WhenAddNullElement_ShouldThrow()
        {
            var tree = new BinaryTree<string>();
            Assert.Throws<ArgumentNullException>(() => tree.Add(null));
        }

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

            Assert.Throws<ArgumentNullException>(() => tree.Contains(null));
        }

    }
}
```

```
using DotNet_lab1_2;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace BinaryTreeCollectionTests
{
    public class Clear_Tests: BinaryTreeTests
    {
        [Theory]
        [MemberData(nameof(GetTreeData))]
        public void Clear_WhenClearedTree_ShouldMakeTreeEmpty<T>(T[] values)
        {
            var tree = new BinaryTree<T>();
            foreach (var value in values)
            {
                tree.Add(value);
            }
            tree.Clear();

            Assert.Empty(tree);
        }
    }
}
```

```
using DotNet_lab1_2;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace BinaryTreeCollectionTests
{
```

```
    public class Compare_Tests
    {
        [Fact]
        public void
Compare_ThrowsArgumentException_WhenTDoesNotImplementIComparable()
        {
            var node = new Node<MyComparableClass>(new MyComparableClass(1));

            var exception = Assert.Throws<ArgumentException>(() => node.Compare(new
MyComparableClass(1), new MyComparableClass(2)));
        }


        public class MyComparableClass
        {
            public int Value { get; set; }

            public MyComparableClass(int value)
            {
                Value = value;
            }
        }
    }
}



using DotNet_lab1_2;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace BinaryTreeCollectionTests
{
    public class CopyTo_Tests : BinaryTreeTests
    {
        [Theory]
        [MemberData(nameof(GetTreeData))]
        public void CopyTo_WhenCopiedTreeToArray_ShouldContainsOfElements<T>(T[]
values)
        {
            var tree = new BinaryTree<T>();

            foreach (var value in values)
            {
                tree.Add(value);
            }

            T[] array = new T[tree.Count];
            int number = 0;
            tree.CopyTo(array, number);

            Assert.Equal(values.Length, array.Length);

            for (int i = 0; i < values.Length; i++)
            {
                Assert.Contains(values[i], array);
            }
        }

        [Fact]
        public void CopyTo_WhenCopyWithNegativeIndex_ShouldThrow()
        {
            var tree = new BinaryTree<int>();
            tree.Add(1);
            tree.Add(2);

            int[] array = new int[2];
```

```
            Assert.Throws<ArgumentOutOfRangeException>(() => tree.CopyTo(array, -
2));
        }

        [Fact]
        public void CopyTo_WhenCopyToArrayWhichHaveNotEnoughSpace_ShouldThrow()
        {
            var tree = new BinaryTree<int>();
            tree.Add(1);
            tree.Add(2);
            tree.Add(3);

            int[] array = new int[1];

            Assert.Throws<ArgumentException>(() => tree.CopyTo(array, 1));
        }
    }
}

using DotNet_lab1_2;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace BinaryTreeCollectionTests
{
    public class Events_Tests
    {
        [Fact]
        public void ItemAddedEvent_IsRaisedWhenElementIsAdded_ShouldBeTrue()
        {
            var binaryTree = new BinaryTree<int>();
            bool eventRaised = false;
            int addedValue = 42;

            binaryTree.ItemAdded += (sender, args) =>
            {
                eventRaised = true;
                Assert.Equal(addedValue, args.Value);
            };

            binaryTree.Add(addedValue);

            bool result = binaryTree.Contains(addedValue);

            Assert.True(eventRaised);
            Assert.True(result);
        }

        [Fact]
        public void ItemContainedEvent_IsRaisedWhenElementIsContained_ShouldBeTrue()
        {
            var binaryTree = new BinaryTree<int>();
            int containedValue = 42;
            binaryTree.Add(containedValue);
            bool eventRaised = false;

            binaryTree.ItemContained += (sender, args) =>
            {
                eventRaised = true;
                Assert.Equal(containedValue, args.Value);
            };

            bool result = binaryTree.Contains(containedValue);

            Assert.True(eventRaised);
            Assert.True(result);
```

```
        }


        [Fact]
        public void ItemRemovedEvent_IsRaisedWhenElementIsRemoved_ShouldBeTrue()
        {
            var binaryTree = new BinaryTree<int>();
            int removedValue = 42;
            binaryTree.Add(removedValue);
            bool eventRaised = false;

            binaryTree.ItemRemoved += (sender, args) =>
            {
                eventRaised = true;
                Assert.Equal(removedValue, args.Value);
            };

            bool result = binaryTree.Remove(removedValue);

            Assert.True(eventRaised);
            Assert.True(result);
        }

        [Fact]
        public void
ItemCleanedEvent_IsRaisedWhenTreeIsCleared_ShouldBeTrueAndEmpty()
        {
            var binaryTree = new BinaryTree<int>();
            int addedValue = 42;
            binaryTree.Add(addedValue);
            bool eventRaised = false;

            binaryTree.ItemCleaned += (sender, args) =>
            {
                eventRaised = true;
            };

            binaryTree.Clear();

            Assert.True(eventRaised);
            Assert.Empty(binaryTree);
        }
    }
}


using DotNet_lab1_2;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace BinaryTreeCollectionTests
{
    public class MyList_Tests: BinaryTreeTests
    {
        [Theory]
        [MemberData(nameof(GetTreeData))]
        public void MyList_WhenIndexerSetUpdatesElement_ShouldBeCorect<T>(T[]
values)
        {
            var myList = new MyList<T>();
            foreach (var item in values)
            {
                myList.Add(item);
            }
            myList[1] = myList[2];
            Assert.Equal(myList[1], myList[1]);
```

```
        }

        [Fact]
        public void MyList_WhenIndexerSetOutOfRange_ShouldThrow()
        {
            var myList = new MyList<int>();
            Assert.Throws<ArgumentOutOfRangeException>(() => myList[0] = 1);
        }

        [Fact]
        public void MyListAdd_WhenAddNullElement_ShouldThrow()
        {
            var myList = new MyList<string>();
            string nullItem = null;

            Assert.Throws<ArgumentNullException>(() => myList.Add(nullItem));
        }

        [Fact]
        public void MyListAddRange_WhenAddNullCollectionRange_ShouldThrow()
        {
            var myList = new MyList<string>();
            List<string> nullCollection = null;

            Assert.Throws<ArgumentNullException>(() =>
myList.AddRange(nullCollection));
        }

        [Fact]
        public void MyListEnumerator_WhenIndexerGetOutOfRange_ShouldThrow()
        {
            var myList = new MyList<int>();

            Assert.Throws<ArgumentOutOfRangeException>(() => myList[0]);
        }

        [Fact]
        public void MyListEnumerator_WhenResetResetsTheIndex_ShouldBeCorect()
        {
            var myList = new MyList<int> { 1, 2, 3 };
            var enumerator = myList.GetEnumerator();

            enumerator.MoveNext();
            enumerator.MoveNext();

            enumerator.Reset();

            enumerator.MoveNext();
            Assert.Equal(1, enumerator.Current);
        }

    }
}
using DotNet_lab1_2;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace BinaryTreeCollectionTests
{
    public class Remove_Tests: BinaryTreeTests
    {
        [Theory]
        [MemberData(nameof(GetTreeData))]
        public void Remove_WhenRemovedElementsAreNotContained_ShouldBeFalse<T>(T[]
values)
        {
            var tree = new BinaryTree<T>();
```

```
            foreach (var value in values)
            {
                tree.Add(value);
            }

            foreach (var value in values)
            {
                tree.Remove(value);
                Assert.False(tree.Contains(value));
            }
        }

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

            Assert.Throws<InvalidOperationException>(() => tree.Remove(5));
        }

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

            tree.Add(4);
            tree.Add(1);
            tree.Add(6);
            tree.Add(7);
            tree.Add(3);

            Assert.Throws<ArgumentException>(() => tree.Remove(5));
        }

        [Fact]
        public void
Remove_WhenRemoveElementAndElementReplaceWithMinValue_ShouldBeCorect()
        {
            BinaryTree<int> binaryTree = new BinaryTree<int>();

            binaryTree.Add(4);
            binaryTree.Add(2);
            binaryTree.Add(6);
            binaryTree.Add(1);
            binaryTree.Add(3);
            binaryTree.Add(5);

            int elementToRemove = 4;
            binaryTree.Remove(elementToRemove);

            var expectedValues = new List<int> { 1, 2, 3, 5, 6};
            var actualValues = binaryTree.Inorder().ToList();

            Assert.Equal(expectedValues, actualValues);
        }

        [Fact]
        public void Remove_WhenRemovesValue_ShouldBeFalse()
        {
            var node = new Node<int>(10);
            node.Add(5);
            node.Add(15);

            node.Remove(node, 15);

            bool containsRemovedValue = node.Contains(15);

            Assert.False(containsRemovedValue);
        }
```

```
    }
}
using DotNet_lab1_2;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace BinaryTreeCollectionTests
{
    public class Traversal_Tests: BinaryTreeTests
    {
        [Fact]
        public void Preorder_WhenThePreorderTraversalReturns_ShouldBeCorect()
        {
            var tree = new BinaryTree<int>();

            tree.Add(4);
            tree.Add(3);
            tree.Add(7);
            tree.Add(1);
            tree.Add(8);

            List<int> expected = new() { 4, 3, 1, 7, 8 };

            var actual = tree.Preorder();

            Assert.Equal(expected, actual);
        }

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

            tree.Add(4);
            tree.Add(3);
            tree.Add(7);
            tree.Add(1);
            tree.Add(8);

            List<int> expected = new() { 1, 3, 8, 7, 4 };
            var actual = tree.Postorder();
            Assert.Equal(expected, actual);
        }

        [Theory]
        [MemberData(nameof(GetTreeData))]
        public void Inorder_WhenTheInorderTraversalReturns_ShouldBeCorect<T>(T[]
values)
        {
            var tree = new BinaryTree<T>();

            foreach (var value in values)
            {
                tree.Add(value);
            }

            var expected = values.ToList();
            expected.Sort();

            var actual = tree.Inorder();

            Assert.Equal(expected, actual);
        }

        [Fact]
        public void
Preorder_WhenThePreorderTraversalReturnsForEmptyTree_ShouldBeEmpty()
```

```
        {
            var tree = new BinaryTree<int>();

            var result = tree.Preorder();

            Assert.Empty(result);
        }

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

            var result = tree.Postorder();

            Assert.Empty(result);
        }

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

            var result = tree.Inorder();

            Assert.Empty(result);
        }
    }
}
using DotNet_lab1_2;
using System;
using System.Xml.Linq;

namespace BinaryTreeCollectionTests
{
    public class TreeTests : BinaryTreeTests
    {
        [Theory]
        [MemberData(nameof(GetEmptyTreeTestData))]
        public void GetEnumerator_WhenEmptyTree_ShouldFalse<T>(BinaryTree<T> tree)
        {
            var enumerator = tree.GetEnumerator();

            var moveNextResult = enumerator.MoveNext();
            var current = enumerator.Current;

            Assert.False(moveNextResult);
            Assert.Equal(default, current);
        }


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

            tree.Add(1);
            tree.Add(2);
            tree.Add(3);
            tree.Add(4);
            tree.Add(5);
            tree.Add(6);

            tree.Balance();

            int[] action = new int [tree.Count()];
            tree.CopyTo(action, 0);
```

```
            int[] expected = { 3, 1, 2, 5, 4, 6};
            var a = action.Equals(action);
            Assert.True(action.Equals(action));
        }

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

            bool isReadOnly = binaryTree.IsReadOnly;

            Assert.False(isReadOnly);
        }
    }
}
```