Lab: Unit testing

This document defines the lab overview for the "C# OOP Advanced" course @ Software University. Please submit your solutions (source code) of all below described problems at the end of the course at softuni.bg.

Introduction

In the current lab piece we are going to extend the sorted data structure that we have a little bit and after that test if the whole functionality works correctly. If we work through the interface, in a later stage of our project, we may change the concrete implementation of the data structure, but the behavior will stay the same, so the only thing we will need to modify is the actual instantiation in the unit test class.

Problem 1. Modify the ISimpleOrderedBag and SimpleSortedList

There are two things that we will add in the interface. The first one is a Remove method that receives a <T> element and returns true or false - whether it has been removed or not.

The second thing is a property for the Capacity of the data structure. Here is how the interface should look after the modification:

```
public interface ISimpleOrderedBag<T> : IEnumerable<T> where T : IComparable<T>
{
    bool Remove(T element);

    int Capacity { get; }

    int Size { get; }

    void Add(T element);

    void AddAll(ICollection<T> elements);

    string JoinWith(string joiner);
}
```

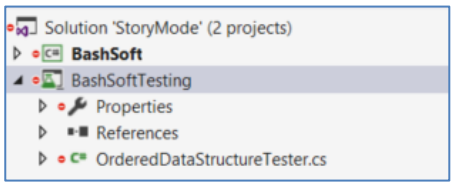
Now implement the missing functionality in the SimpleSortedList. Here is how it should look:

```
public int Capacity
{
    get { return this.innerCollection.Length; }
}
```

```
public bool Remove(T element)
{
    bool hasBeenRemoved = false;
    int indexOfRemovedElement = 0;
    for (int i = 0; i < this.Size; i++)
    {
        if (this.innerCollection[i].Equals(element))
        {
            indexOfRemovedElement = i;
            this.innerCollection[i] = default(T);
            hasBeenRemoved = true;
            break;
        }
    }

    if (hasBeenRemoved)
    {
        for (int i = indexOfRemovedElement; i < this.Size - 1; i++)
        {
            this.innerCollection[i] = this.innerCollection[i + 1];
        }

        this.innerCollection[this.size - 1] = default(T);
    }

    return hasBeenRemoved;
```

Problem 2. Writing Unit Tests

Start by creating a new Unit testing project in the current Solution. You can call it BashSoftTesting and rename the class inside it to OrderedDataStructureTester. Here is how your structure should look by now:



Now it's time to add a reference to the BashSoft in the current testing project.

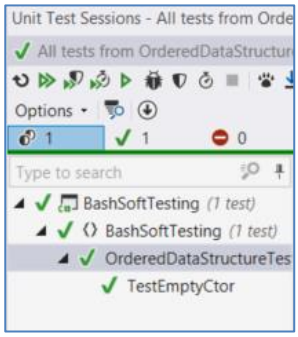
Right click references -> Add reference -> Select projects -> Select BashSoft -> Click OK

Make the simple sorted list public, so that you can see it in the testing app.

Let's first start by testing the constructors of the class. Name the first test method TestEmptyCtor and in it we will make a new instance of the SimpleSortedList and check the capacity and size values:

```
[TestClass]
public class OrderedDataStructureTester
{
    private ISimpleOrderedBag<string> names;

    [TestMethod]
    public void TestEmptyCtor()
    {
        this.names = new SimpleSortedList<string>();
        Assert.AreEqual(this.names.Capacity, 16);
        Assert.AreEqual(this.names.Size, 0);
    }
}
```

If you run the unit test everything should be according to plan and the result should be such:

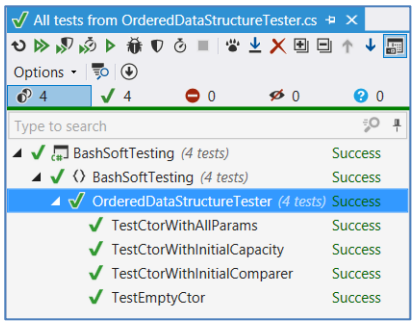


Now let's make methods that test the other 3 constructors:

Here are their checks:

```
[TestMethod]
public void TestCtorWithInitialCapacity()
{
    this.names = new SimpleSortedList<string>(20);
    Assert.AreEqual(this.names.Capacity, 20);
    Assert.AreEqual(this.names.Size, 0);
}

[TestMethod]
public void TestCtorWithAllParams()
{
    this.names = new SimpleSortedList<string>(StringComparer.OrdinalIgnoreCase, 30);
    Assert.AreEqual(this.names.Capacity, 30);
    Assert.AreEqual(this.names.Size, 0);
}

[TestMethod]
public void TestCtorWithInitialComparer()
{
    this.names = new SimpleSortedList<string>(StringComparer.OrdinalIgnoreCase);
    Assert.AreEqual(this.names.Capacity, 16);
    Assert.AreEqual(this.names.Size, 0);
}
```

Resulting in:



Now that we are done with testing the constructors, it's time to make a method that initializes an empty collection, to test the rest of the object behavior and it is initialized for each test:

```
[TestInitialize]
public void SetUp()
{
    this.names = new SimpleSortedList<string>();
}
```

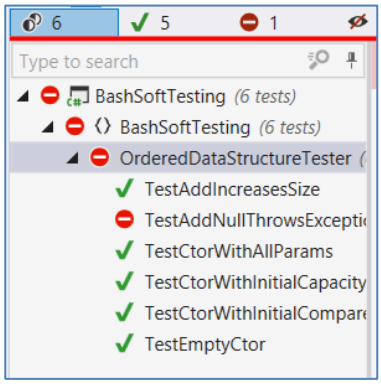
Since we are done with that, let's check that the add method works as expected and increases the size (count) of the collection:

```
[TestMethod]
public void TestAddIncreasesSize()
{
    this.names.Add("Nasko");
    Assert.AreEqual(1, this.names.Size);
}
```

Now we should start thinking like QAs and think what should happen if someone adds **null**. We think it makes sense to throw a new argument null exception, because that does not match the conditions for what values can be held in our data structure:

```
[TestMethod]
[ExpectedException(typeof(ArgumentNullException))]
public void TestAddNullThrowsException()
{
    this.names.Add(null);
}
```

Now let's run the unit tests and see the results:



If we look at the implementation of the add method we will see, that we do not have any check for null and so we do not throw argument null exception, anywhere. Let's implement the functionality now:

```
public void Add(T element)
{
    if (element == null)
    {
        throw new ArgumentNullException();
    }

    if (this.innerCollection.Length == this.size)
    {
        this.Resize();
    }
```

If your add method begins like the picture above, everything is great.

Now we've shown you how to make unit tests. Try doing unit tests for the following functionality on your own:

TestAddUnsortedDataIsHeldSorted – adds three strings ("Rosen", "Georgi", "Balkan") and checks by **iterating** the collection if it holds them in a sorted order: `"Balkan", "Georgi", "Rosen"`

TestAddingMoreThanInitialCapacity – adds 17 elements and checks whether the Size is 17 and Capacity is NOT 16.

TestAddingAllFromCollectionIncreasesSize – adds 2 elements to a List<string> and adds this list to the names using AddAll. Then check whether the size of the collection equals 2.

TestAddingAllFromNullThrowsException – adds a null value to the AddAll and expects it to throw an ArgumentNullException. Since this functionality was not considered in the implementation of the DS, we should now implement it.

TestAddAllKeepsSorted – adds a collection with a few elements and after that check if the elements in the SimpleSortedList are sorted.

TestRemoveValidElementDecreasesSize – adds an element, and removes the same element and then checks if the size has decreased. As you can see, this test should not pass, because we are not decreasing the size anywhere. Fix that and everything should be fine.

TestRemoveValidElementRemovesSelectedOne – adds two elements (ivan, nasko) and then removes "ivan" from the collection and asserts it is not there.

TestRemovingNullThrowsException – tries to remove null from the collection, which should result in an exception being thrown. Since we haven't thought of that earlier, it's now time to implement it.

TestJoinWithNull – adds a few elements and tries to join them with null joiner. This should throw an argument null exception, however the current implementation does not do so. Fix the implementation and the unit test should pass.

TestJoinWorksFine – adds a few elements and tries to join them with ", ". This should not result in a passing test. Let's see why:

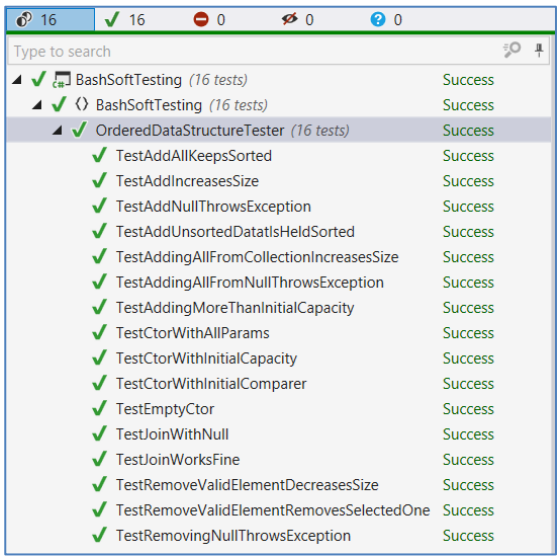
```
}

builder.Remove(builder.Length - 1, 1);
return builder.ToString();
```

As we can see, we've assumed that joiner will be with length 1 which is not a valid case. Let's fix these things.

```
builder.Remove(builder.Length - joiner.Length, joiner.Length);
return builder.ToString();
```

These are only the common unit tests. You may want to make some more using the other constructors, to check if there are any problems with the comparators.

Final result:



This is your last lab for the current course and for the whole C# Fundamentals Module. Hope you've found it interesting and helpful. You can add more functionality on your own, or extend and refactor it even further. You can even upload it to your GitHub profile as it could be a part of your portfolio.