Exercises: Implement Binary Search Tree

This document defines the **in-class exercises** assignments for the "Data Structures" course @ Software University.

Part I: Implement Insert, Contains and Search

The third part of this lab aims to implement a **binary search tree** (ordered binary tree).

Problem 1. Define the Node Data Structure

First, you will need a node to store the data of each tree and a pointer to the root of the tree.

```
public class BinarySearchTree<T> where T : IComparable<T>
{
    private Node root;

    private class Node
    {
        public Node(T value)...

        public T Value { get; set; }
        public Node Left { get; set; }
        public Node Right { get; set; }
    }
}
```

Problem 2. Iteration

Implement the **`EachInOrder`** method, so you have a way to iterate over tree nodes

```
public void EachInOrder(Action<T> action)
{
    // TODO
}
```

Problem 3. Implement Contains

We will consider the non-recursive implementation. Start at the root and compare the searched value with the roots value. If the searched value is less it could be in the left subtree, if it is greater it could be in the right subtree:

```
public bool Contains(T value)
{
    Node current = this.root;
    while (current != null)
    {
        if (value.CompareTo(current.Value) < 0)
        {
            
        }
        else if (value.CompareTo(current.Value) > 0)
        {
            
        }
        else
        {
            break;
        }
    }

    return current != null;
}
```

Problem 4. Implement Insert

Insert is very similar to Contains. Before traversing however, you should check if there are any elements in the tree. If not, set the root:

```
public void Insert(T value)
{
    if (this.root == null)
    {
        this.root = new Node(value);
        return;
    }
}
```

Next, traverse the tree, but hold reference to both current node and its parent

```
Node parent = null;
Node current = this.root;
while (current != null)
{
    //TODO: Find insertion node
}
```

Finally, insert the new node:

```
Node newNode = new Node(value);
if (value.CompareTo(parent.Value) < 0)
{
    parent.Left = newNode;
}
else
{
    parent.Right = newNode;
}
```

Problem 5. Implement Search

Search is very similar to Contains. First of all, it should return **`BinarySearchTree<T>`**.

```
public BinarySearchTree<T> Search(T item)
{
    throw new NotImplementedException();
}
```

You will start by finding the element:

```
Node current = this.root;
while (current != null)
{
    //TODO: Find the element
}
```

Now you only need to return the element you found:

```
return new BinarySearchTree<T>(current);
```

In order for that to work, we need to implement a new private **`constructor`**:

```
private BinarySearchTree(Node root)
{
    this.Copy(root);
}
```

Finally, you need to implement the **`Copy()`** method that will copy the elements in exactly the same way in which they exist in the parent tree (Pre-Order traversal):

```
private void Copy(Node node)
{
    if (node == null)
    {
        return;
    }

    this.Insert(node.Value);
    this.Copy(node.Left);
    this.Copy(node.Right);
}
```

Problem 6. Run Unit Tests

Run the unit tests and ensure that some of them work correctly:

✓ Contains_ExistingElement_ShouldReturnTrue	Success
✓ Contains_NonExistingElement_ShouldReturnFalse	Success
⛔ DeleteMin_Empty_Tree_Should_Work_Correctly	Failed
⛔ DeleteMin_One_Element_Should_Work_Correctly	Failed
⛔ Insert_Multiple_DeleteMin_Should_Work_Correctly	Failed
✓ Insert_Multiple_TraverseInOrder	Success
✓ Insert_Single_TraverseInOrder	Success
⛔ Range_ExistingElements_ShouldReturnCorrectCount	Failed
⛔ Range_ExistingElements_ShouldReturnCorrectElements	Failed
✓ Search_ExistingElement_ShouldReturnSubTree	Success
✓ Search_NonExistingElement_ShouldReturnEmptyTree	Success

In the next parts, we will implement the rest of the methods.

Part II: Implement DeleteMin and Range

Problem 7. Implement Delete Min

`DeleteMin` consists of three steps. Check if the root is null. If not, find the parent of the min element and set its left child to be the min's right child.

Check if root is not null

```
public void DeleteMin()
{
    if (this.root == null)
    {
        return;
    }
}
```

Find the min element's parent

```
Node parent = null;
Node min = this.root;
while (min.Left != null)
{
    [redacted]
}
```

If you found such a parent set its left child to be the min's right

```
if (parent == null)
{
    [redacted]
}
else
{
    [redacted]
}
```

Problem 8. Implement Range

Range will be implemented using a queue of **T**. First, let's implement the public **Range(T, T)** method:

```
public IEnumerable<T> Range(T startRange, T endRange)
{
    Queue<T> queue = new Queue<T>();

    this.Range(this.root, queue, startRange, endRange);

    return queue;
}
```

Now create a new method, that will take the **Node** and our queue as parameters:

```
private void Range(Node node, Queue<T> queue, T startRange, T endRange)
{
    if (node == null)
    {
        return;
    }

}
```

We need to verify that our node is not **null** and this will be the bottom of our recursion. Now, we need to find if our node is in the lower and higher range borders:

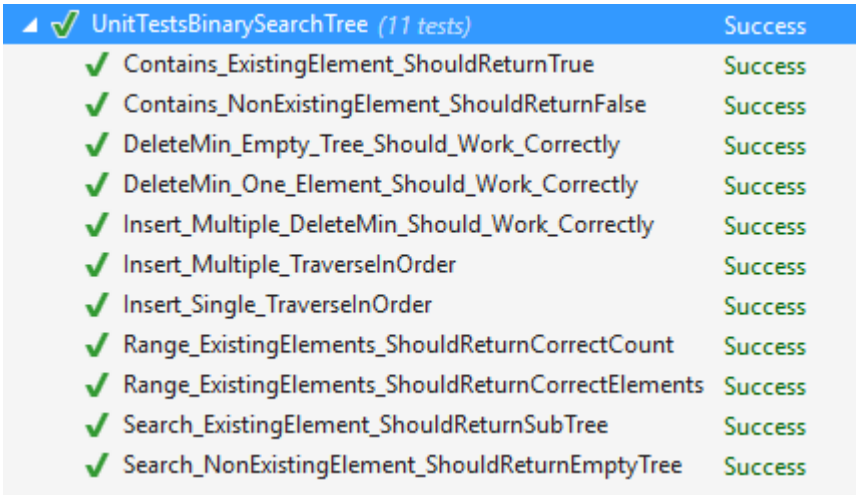
```
int nodeInLowerRange = startRange.CompareTo(node.Value);
int nodeInHigherRange = endRange.CompareTo(node.Value);
```

Finally, we have to implement **in-order** traversal for the items that are in the range. Recursively **go to the left** if our node value is **bigger** than the lower range, and **go to the right** if our node value is **smaller** than the higher range border. If we found the element between the recursive calls, we need to add it to the queue:

```
if (
{
    
}
if (nodeInLowerRange <= 0 && nodeInHigherRange >= 0)
{
    queue.Enqueue(node.Value);
}
if (
{
    
}
```

Problem 9. Run Unit Tests

Run the unit tests and ensure that all of them pass successfully.



That's it. You're ready to begin the exercise!