Exercises: AVL Tree and Binary Heap

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

Part I - Implement an AVL Tree

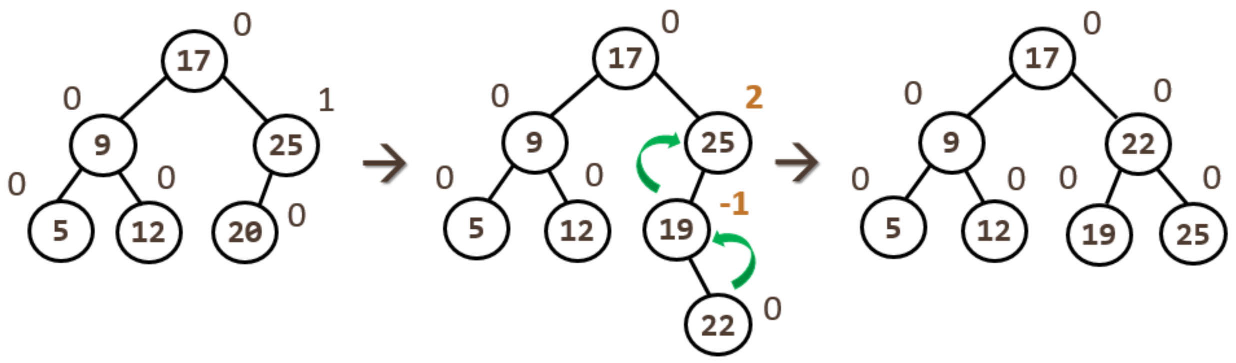
An AVL tree is a balanced **binary search tree** (BST).

- Each node holds a **balance factor (BF)**. It is calculated as follows:

Balance Factor Formula
Balance Factor = Left subtree heigh - Right subtree height

- When a node is inserted **its balance factor is 0** (because it's a leaf, it has no left or right subtrees). **Balance factors** are modified each time a subtree grows or decreases (when a node is **inserted** or **deleted**).
- An AVL tree allows 2 subtrees to have a **height difference at most 1**. Therefore, balance factors must be in the range **[-1, 1]**.
- If a balance factor becomes -2 or 2 it means a subtree has grown too much and tree must be rebalanced.
  - **BF == -2** means the **right subtree** has grown too much
  - **BF == 2** means the **left subtree** has grown too much
- **Rebalancing** is done by starting from the **inserted node** and going up to the **root**. If a node's BF becomes -2 or 2 → perform **rotations** (which we will discuss later) and stop.

*Example of inserting **22** to an AVL tree.*



Problem 1. Define a Node class

Let's define a node class for representing nodes in our tree. Like in any BST each node will hold its **value**, **left** and **right** children. In addition to that, it should also hold its **parent** + **balance factor** (used by the AVL tree).

```
public class Node<T> where T : IComparable<T>
{
    public Node(T value)
    {
        this.Value = value;
    }

    public T Value { get; set; }

    public Node<T> LeftChild { get; set; }

    public Node<T> RightChild { get; set; }

    public Node<T> Parent { get; set; }

    public int BalanceFactor { get; set; }
```

Problem 2. Add Helper Properties

Let's add a couple of **helper properties** which we will use later when implementing our AVL tree.

- **`IsLeftChild`** - returns whether the node is a left child of its parent (if there is no parent, returns false)
- **`IsRightChild`** - returns whether the node is a right child (if there is no parent, returns false)
- **`ChildrenCount`** - returns the count of all children
- **`ToString()`** - returns the node's value (this is helpful when debugging)

```
        public bool IsLeftChild
        {
            get { // TODO: }
        }

        public bool IsRightChild
        {
            get { // TODO: }
        }

        public int ChildrenCount
        {
            get { // TODO: }
        }

        public override string ToString()
        {
            return this.Value.ToString();
        }
    }
}
```

Problem 3. Adding Elements to the Tree

Let's implement our **`Add(T item)`** method.

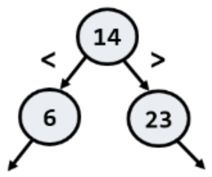
- If there is no root, we **create a new Node** and **set it as root**.
- Otherwise, we call another method to process the **insertion**.
- If the item already exists, we do not increase the count.

```
public void Add(T item)
{
    var inserted = true;
    if (this.root == null)
    {
        // TODO: Set root
    }
    else
    {
        inserted = this.InsertInternal(this.root, item);
    }

    if (inserted)
    {
        this.Count++;
    }
}
```

The **`InsertInternal(Node<T> node, T item)`** method performs a lookup (the **`while`**-loop) and determines where to insert the new node. If the insertion is successful, it should return **true**.

```
private bool InsertInternal(Node<T> node, T item)
{
    var currentNode = node;
    var newNode = new Node<T>(item);
    var shouldRetrace = false;
    while (true)
    {
        // TODO: Perform lookup and insert node
        break;
    }

    if (shouldRetrace)
    {
        // TODO: Retrace and rebalance tree
    }

    return true;
}
```

It **starts from the root** and goes down the tree following the rules:



- **Left child** < current node
- **Right child** > current node

We go down the tree until we **find a leaf node** and **insert the new node** as either left or right child.

- If we **inserted it as right child**, we **decrease the balance factor** of the parent node. Why? Because the parent's right subtree has grown.
- If we inserted it as **left child**, we, respectively, **increase the balance factor**.

Before we break out of the loop, remember that **`bool shouldRetrace`** variable? We need to know if retracing (modifying the balance factors up to root) is necessary. When is it necessary? When the subtree holding the parent node grows (when its height increases). In other words, we **only retrace if the subtree's height increases**.

```
while (true)
{
    if (currentNode.Value.CompareTo(item) < 0) // Node value is less than item -> go right
    {
        if (currentNode.RightChild == null)
        {
            // TODO: Set newNode as right child
            currentNode.BalanceFactor--;
            shouldRetrace = currentNode.BalanceFactor != 0;
            break;
        }

        currentNode = currentNode.RightChild;
    }
    else // Node value is greater than item -> go left
    {
        if (currentNode.LeftChild == null)
        {
            // TODO: Set newNode as left child
            currentNode.BalanceFactor++;
            shouldRetrace = currentNode.BalanceFactor != 0;
            break;
        }

        currentNode = currentNode.LeftChild;
    }
    // TODO: Else item is already present -> break
}
```

Once we've **successfully inserted the new node**, we only have to **check if retracing is needed**.

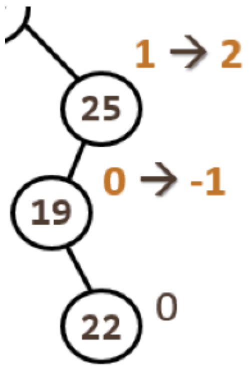
```
        if (shouldRetrace)
        {
            this.RetraceInsert(currentNode);
        }
    }
```

If so, we call method **`RetraceInsert(Node<T> startNode)`** and begin retracing from the new node's parent to the root.

```
private void RetraceInsert(Node<T> node)
{
    var parent = node.Parent;
    while (parent != null)
    {
        if (node.IsLeftChild)
        {
            // Node is left -> parent's left subtree has grown
            // TODO: Check parent balance factor and if necessary perform rotations
        }
        else
        {
            // Node is right -> parent's right subtree has grown
            // TODO: Check parent balance factor and if necessary perform rotations
        }

        node = parent;
        parent = node.Parent;
    }
}
```

The **retracing loop** "climbs up" back to the root and **modifies the balance factors**, depending on the **direction** we come from.

- If we come from **right** → reduce **parent BF**
- If we come from **left** → increase **parent BF**



***22** is inserted in the tree*

If a balance factor becomes **2** or **-2** -> the insertion has **unbalanced the tree** and we need to **rebalance it**.

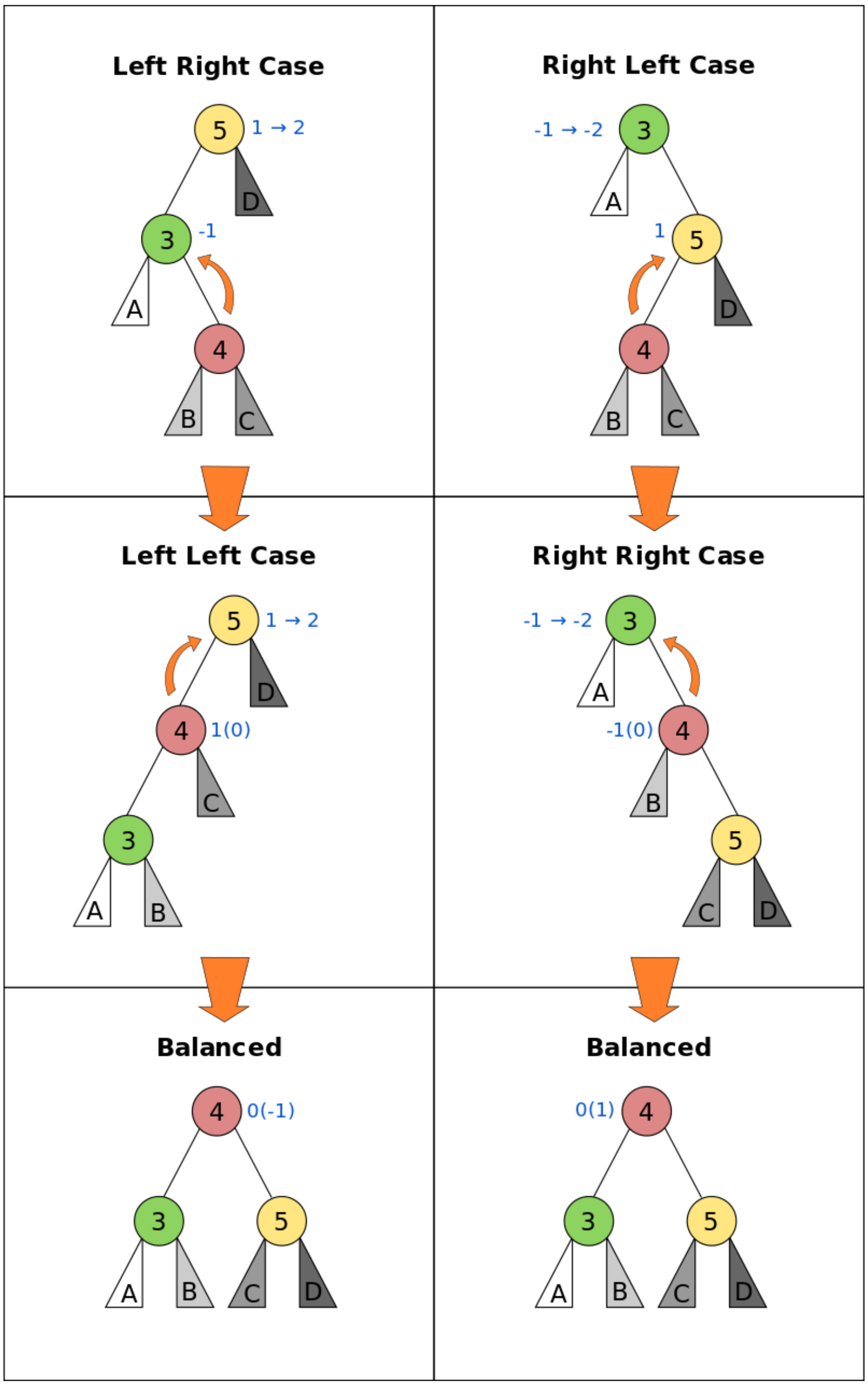
```
    if (node.IsLeftChild)
    {
        // Parent's left branch has grown and BF will become 2 -> rebalance
        if (parent.BalanceFactor == [illegible])
        {
            parent.BalanceFactor++;
            // Node's right branch has grown
            if (node.BalanceFactor == [illegible])
            {
                // Make branch straight so we can rotate (left-right case)
                this.RotateLeft(node);
            }

            // Rotate parent right to balance (left-left case)
            this.RotateRight(parent);
            break;
        }
        else if (parent.BalanceFactor == -1) // Parent is now balanced -> no need to go up
        {
            parent.BalanceFactor = 0;
            break;
        }
        else
        {
            parent.BalanceFactor = 1;
        }
    }
```

We do the same thing for when we come from right:

```
else
{
    // Parent's right branch has grown and BF will become -2 -> rebalance
    if (parent.BalanceFactor == [illegible])
    {
        parent.BalanceFactor--;
        // Node's left branch has grown
        if (node.BalanceFactor == [illegible])
        {
            // Make branch straight so we can rotate (right-left case)
            this.RotateRight(node);
        }

        // Rotate parent right to balance (right-right case)
        this.RotateLeft(parent);
        break;
    }
    else if (parent.BalanceFactor == 1) // Parent is now balanced -> no need to go up
    {
        parent.BalanceFactor = 0;
        break;
    }
    else
    {
        parent.BalanceFactor = -1;
    }
}
```

Rotations

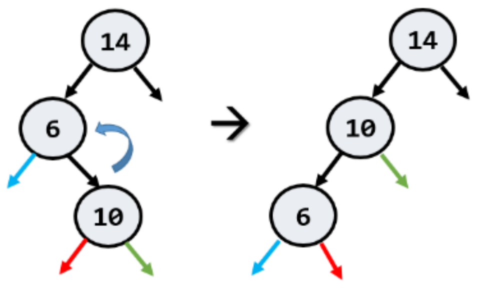
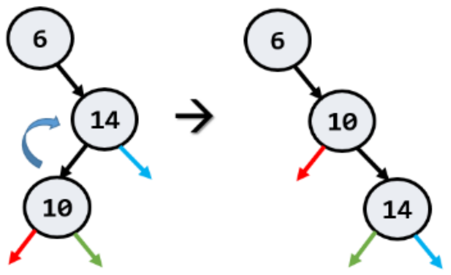
These are 4 different cases when rotating:



Add another helper method (property in this case). Adding a **node N** as left or right child to **node P**, automatically sets his parent link as well.

```
public class Node<T> where T : IComparable<T>
{
    private Node<T> leftChild;
    private Node<T> rightChild;

    public Node<T> LeftChild
    {
        get { return this.leftChild; }
        set
        {
            if (value != null)
            {
                value.Parent = this;
            }

            this.leftChild = value;
        }
    }

    public Node<T> RightChild
    {
        // TODO: ...
```

An AVL tree has 2 types of rotations - left and right as seen below:

Left rotation of node 6	**Right rotation of node 14**
	

The method performing the left rotation looks as follows:

```
private void RotateLeft(Node<T> node)
{
    var parent = node.Parent;
    var child = node.RightChild;
    if (parent != null)
    {
        // Link parent with new node
        if (node.IsLeftChild)
            parent.LeftChild = child;
        else
            parent.RightChild = child;
    }
    else
    {
        // If no parent -> set new root
        this.root = child;
        this.root.Parent = null;
    }

    // Rotate left -> parent becomes left child of his own right child
    node.RightChild = child.LeftChild;
    child.LeftChild = node;

    node.BalanceFactor += 1 - Math.Min(child.BalanceFactor, 0);
    child.BalanceFactor += 1 + Math.Max(node.BalanceFactor, 0);
}
```

Likewise, implement the rotate right method. Use the illustrations above for reference.

```
private void RotateRight(Node<T> node)
{
    var parent = node.Parent;
    var child = node.LeftChild;

    // TODO: Implement

    node.BalanceFactor -= 1 + Math.Max(child.BalanceFactor, 0);
    child.BalanceFactor -= 1 - Math.Min(node.BalanceFactor, 0);
}
```

And voila! Your insertion into the AVL tree should be ready.

Before we can test it, write a **`ForeachDfs(Action<int, T> action)`** method. It will be used by the unit tests for performing **in-order DFS traversal**. On each node visit it will call the given **action** and pass the depth of the node + its value.

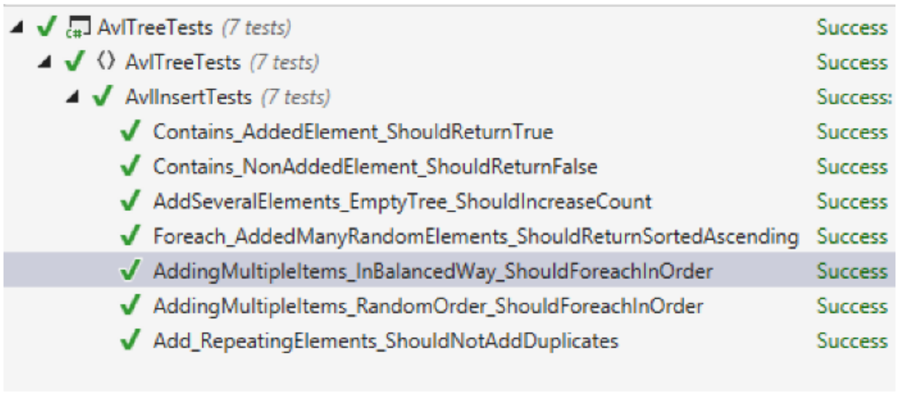
```
public void ForeachDfs(Action<int, T> action)
{
    if (this.Count == 0)
    {
        return;
    }

    this.InOrderDfs(this.root, 1, action);
}

private void InOrderDfs(Node<T> node, int depth, Action<int, T> action)
{
    // TODO: Full In-order recursive DFS traversal
}
```

Problem 4. Contains Method

Write a **`Contains(T item)`** method for finding if an item is within our tree. The lookup is the same as in any other binary search tree.

```
public bool Contains(T item)
{
    var node = this.root;
    while (node != null)
    {
        // TODO: Search for item like in an ordinary BST
    }

    return false;
}
```

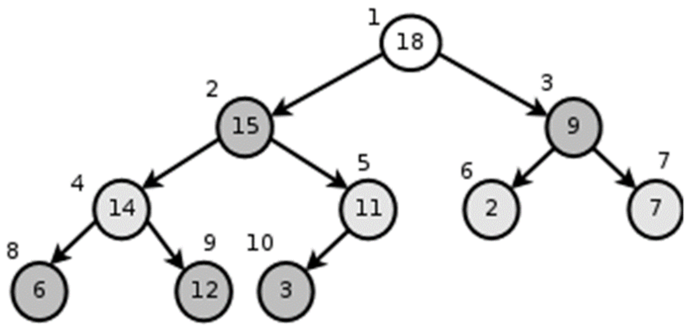
The unit tests should now all pass.

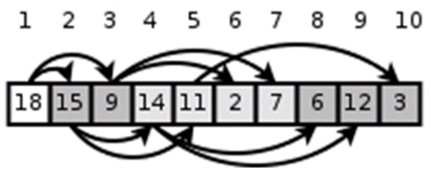

```
◢ ✓ AvlTreeTests (7 tests)                                              Success
  ◢ ✓ AvlTreeTests (7 tests)                                            Success
    ◢ ✓ AvlInsertTests (7 tests)                                        Success
        ✓ Contains_AddedElement_ShouldReturnTrue                        Success
        ✓ Contains_NonAddedElement_ShouldReturnFalse                    Success
        ✓ AddSeveralElements_EmptyTree_ShouldIncreaseCount              Success
        ✓ Foreach_AddedManyRandomElements_ShouldReturnSortedAscending   Success
        ✓ AddingMultipleItems_InBalancedWay_ShouldForeachInOrder        Success
        ✓ AddingMultipleItems_RandomOrder_ShouldForeachInOrder          Success
        ✓ Add_RepeatingElements_ShouldNotAddDuplicates                  Success
```

Congratulations! You have implemented your AVL tree with insertion and lookup!

Part II - Implement a Binary Heap

You have to implement a **binary heap**:





The binary heap holds its element in an **array**. Elements are numbered with **indexes** 0 … length-1. The array represents a perfectly **balanced binary tree**. Each node **i** may have children (left and right) and parent:

- **`parent(i) = (i - 1) / 2`**
- **`leftChild(i) = 2 * i + 1`**
- **`rightChild(i) = 2 * i + 2`**

Binary heaps always hold the "***heap property***":

- Each **node** is **smaller** or equal than its **parent** node.

We should **maintain the "*heap property*"** all the time during our work, so "**heapify up**" or "**heapify down**" should apply each time after we modify the heap. See the steps below to learn how to maintain it.

Problem 5. Learn about Binary Heap in Wikipedia

Before starting, get familiar with the concept of **binary heap**: https://en.wikipedia.org/wiki/Binary_heap.

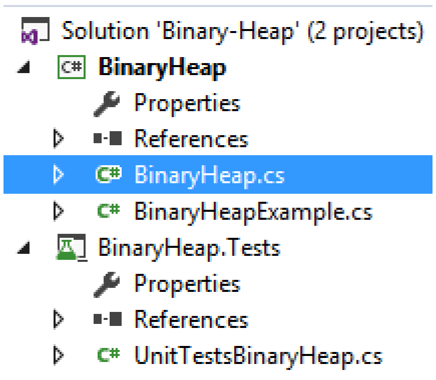
The typical **operations** over a binary heap are:

- **`Build-Max-Heap(arr)`** – builds a binary heap from array of unordered elements
- **`Heapify-Down(index)`** – apply the "*heap property*" down from given node
- **`Extract-Max()`** – extract (and remove) the max element from the heap.
- **`Insert(element)`** – inserts a new element in the heap (and maintains the "*heap property*")
- **`Heapify-Up(index)`** – apply the "*heap property*" up from given node
- **`Peek-Max()`** – finds the max element from the heap (without remove).

Let's start coding!

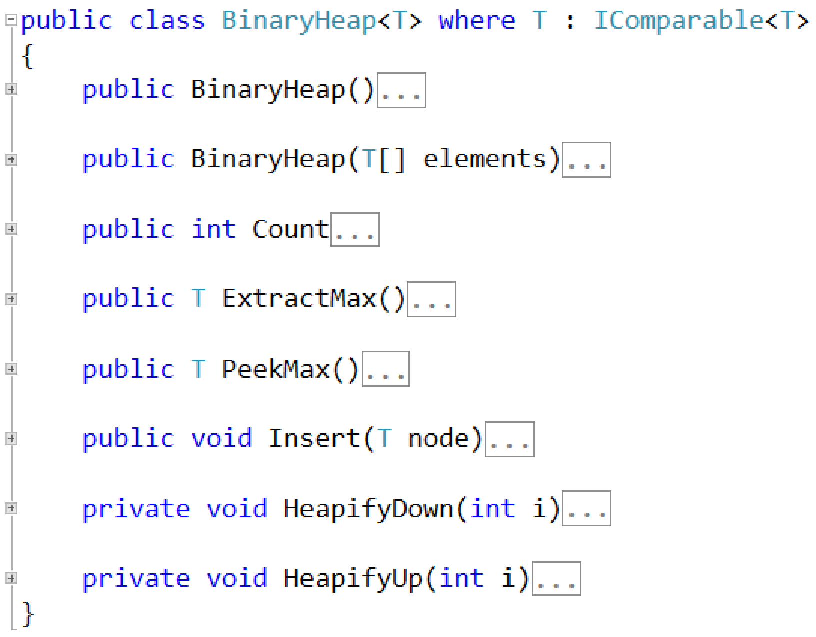
Problem 6. BinaryHeap<T > – Project Skeleton

You are given a **Visual Studio project skeleton** (unfinished project) holding the unfinished class **`BinaryHeap<T>`** and **unit tests** covering its functionality. The project holds the following assets:



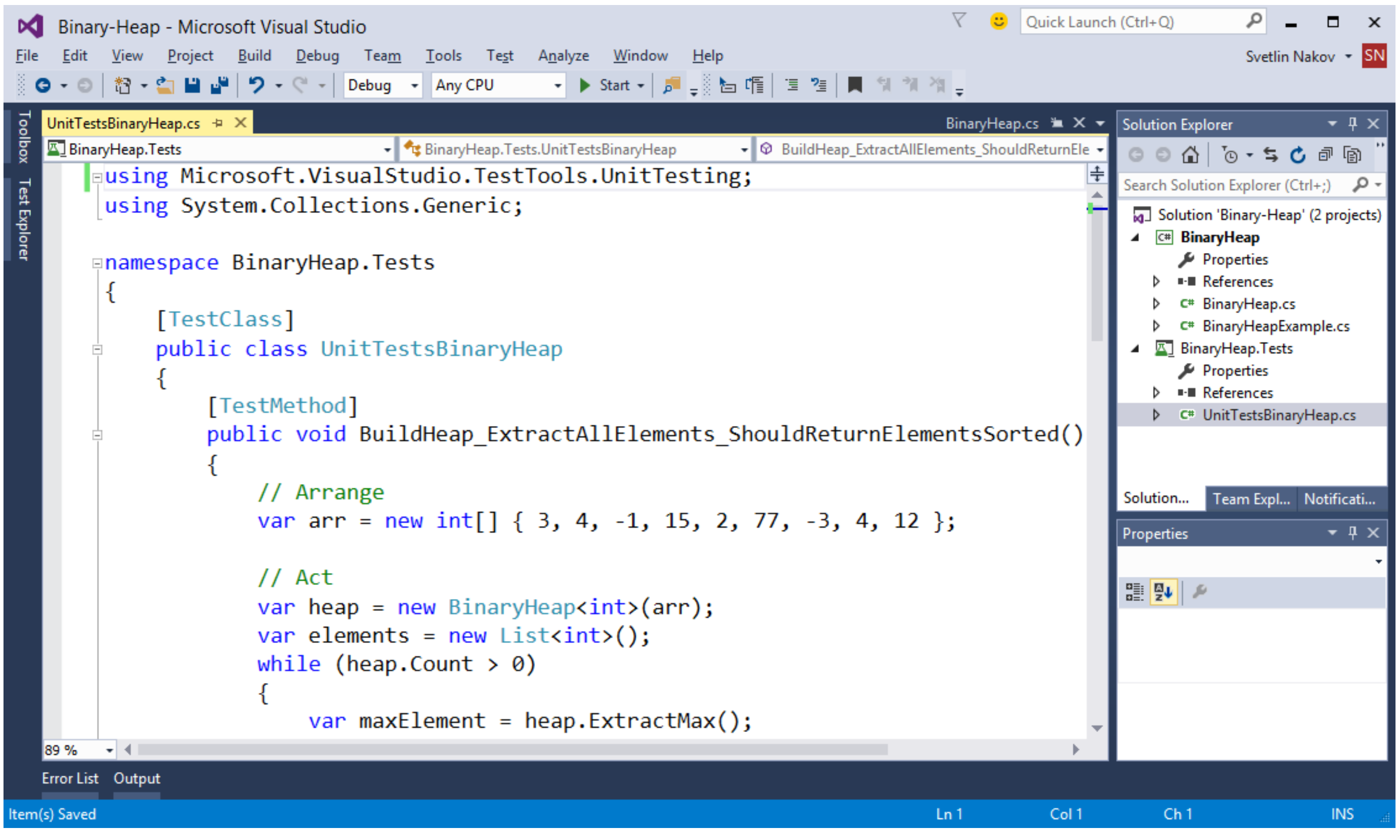
The project skeleton opens correctly in **Visual Studio 2013** but can be open in other Visual Studio versions as well and also can run in **SharpDevelop** and **Xamarin Studio**. Your goal is to implement the missing functionality in order to finish the project.

First, let's take a look at the **`BinaryHeap<T>`** class. It holds a **binary heap** of parameterized type **T**. You need to finish it:



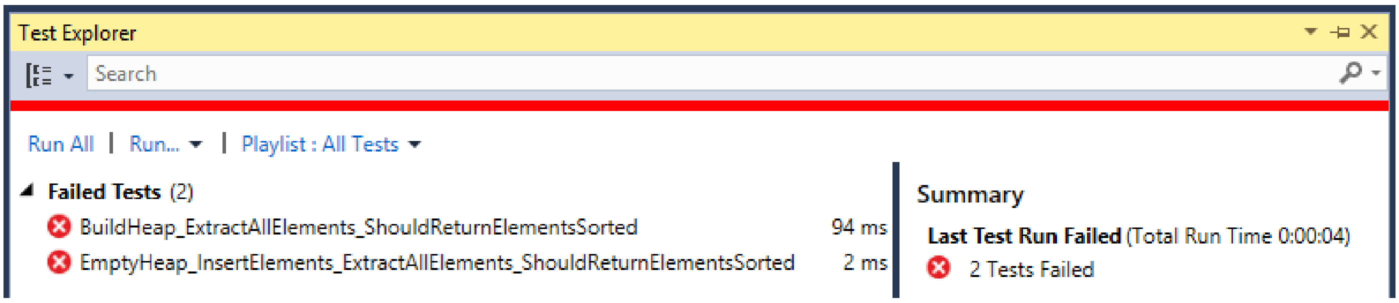
```
public class BinaryHeap<T> where T : IComparable<T>
{
    public BinaryHeap()...

    public BinaryHeap(T[] elements)...

    public int Count...

    public T ExtractMax()...

    public T PeekMax()...

    public void Insert(T node)...

    private void HeapifyDown(int i)...

    private void HeapifyUp(int i)...
}
```

The project comes also with **unit tests** covering the functionality of the **binary heap** (see the class **`UnitTestBinaryHeap`**):


```csharp
using Microsoft.VisualStudio.TestTools.UnitTesting;
using System.Collections.Generic;

namespace BinaryHeap.Tests
{
    [TestClass]
    public class UnitTestsBinaryHeap
    {
        [TestMethod]
        public void BuildHeap_ExtractAllElements_ShouldReturnElementsSorted()
        {
            // Arrange
            var arr = new int[] { 3, 4, -1, 15, 2, 77, -3, 4, 12 };

            // Act
            var heap = new BinaryHeap<int>(arr);
            var elements = new List<int>();
            while (heap.Count > 0)
            {
                var maxElement = heap.ExtractMax();
```

Problem 7. Run the Unit Tests to Ensure All of Them Initially Fail

Run the unit tests from the **`BinaryHeap.Tests`** project. All of them should fail:



```
Failed Tests (2)
  BuildHeap_ExtractAllElements_ShouldReturnElementsSorted                94 ms
  EmptyHeap_InsertElements_ExtractAllElements_ShouldReturnElementsSorted  2 ms

Summary
Last Test Run Failed (Total Run Time 0:00:04)
2 Tests Failed
```

This is quite normal. We have unit tests, but the code covered by these tests is missing. Let's write it.

Problem 8. Define the Binary Heap Internal Data

The **internal data** holding the binary heap elements is quite simple, it is just a **list** of elements (array that can grow):

```csharp
private List<T> heap;
```

Problem 9. Implement the Binary Heap Constructor

Now, let's implement the binary heap **constructor**. Its purpose is to allocate the internal array that will hold the binary heap elements (balanced binary tree). The binary heap constructor has two forms:

- Parameterless constructor – should allocate and empty binary heap
- Constructor with parameter **`array`** – converts existing array of elements to binary heap

The first **parameterless constructor** is quite simple:

```
public BinaryHeap()
{
    this.heap = new List<T>();
}
```

The **second constructor** is more complex:

```
public BinaryHeap(T[] elements)
{
    this.heap = new List<T>(elements);
    for (int i = this.heap.Count / 2; i >= 0; i--)
    {
        HeapifyDown(i);
    }
}
```

The above code first **allocates the internal list** to hold the binary heap elements, then **fills** the passed as argument elements in the internal list and then "***heapifies***" the elements. This means that each **element** becomes **less or equal to its parent**. This happens by moving up each element, which is bigger than its parent. See the implementation of **`HeapifyDown(index)`** method.

We implement also the **`Count`** property which it trivial and returns the number of elements in the heap:

```
public int Count
{
    get
    {
        return this.heap.Count;
    }
}
```

Problem 10. Implement HeapifyDown(index) Method

The **`HeapifyDown(index)`** method starts from given **index** and **reorders the element** from this index **down to its correct place**. The element is swapped with its biggest child element (if the "*heap property*" is not hold). This happens recursively again, and again until we reach a leaf node or the heap property is already hold. See the code below:

```
private void HeapifyDown(int i)
{
    var left = 2 * i + 1;
    var right = 2 * i + 2;
    var largest = i;
    if (left < this.heap.Count &&
        this.heap[left].CompareTo(this.heap[largest]) > 0)
    {
        largest = left;
    }
    if (right < this.heap.Count &&
        this.heap[right].CompareTo(this.heap[largest]) > 0)
    {
        largest = right;
    }
    if (largest != i)
    {
        T old = this.heap[i];
        this.heap[i] = this.heap[largest];
        this.heap[largest] = old;
        HeapifyDown(largest);
    }
}
```

Problem 11. Implement the ExtractMax() Method

Now, we are ready to implement the most important method **`ExtractMax()`** which returns and removes the maximal element:

```
public T ExtractMax()
{
    var max = this.heap[0];
    this.heap[0] = this.heap[heap.Count - 1];
    this.heap.RemoveAt(this.heap.Count - 1);
    if (this.heap.Count > 0)
    {
        HeapifyDown(0);
    }
    return max;
}
```

How it works? It works in thee steps:

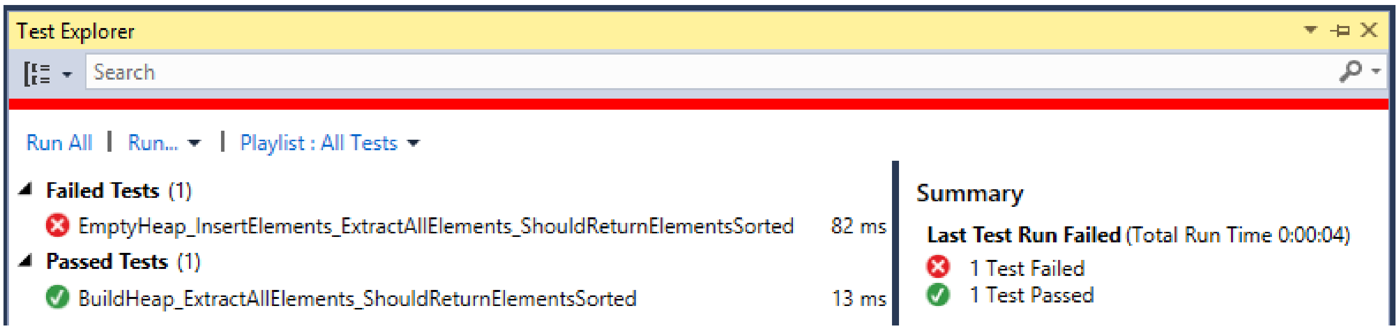
1. Takes as result the **maximal element** – the elements at **index 0** (the root node in the tree).
2. **Deletes the last element** from the internal list holding the heap elements and **moves it at position 0** (as root node).
3. Moves down the root node to **apply the "*heap property*"**, i.e. call **`Heapify-Down()`**.

We also implement the **Peek-Max** operation. It is trivial: just **return the root element** (from index 0):

```
public T PeekMax()
{
    var max = this.heap[0];
    return max;
}
```

Problem 12. Run the Unit Tests

We have **partially implemented** the binary heap. It supports **`Build-Heap`** and **`Extract-Max`** operations. Let's run the tests. We can expect some of them to pass:


Test Explorer

Search

Run All | Run... | Playlist : All Tests

Failed Tests (1)
EmptyHeap_InsertElements_ExtractAllElements_ShouldReturnElementsSorted 82 ms
Passed Tests (1)
BuildHeap_ExtractAllElements_ShouldReturnElementsSorted 13 ms

Summary
Last Test Run Failed (Total Run Time 0:00:04)
1 Test Failed
1 Test Passed

To have more tests passed, we need to implement the rest of the functionality. Let's continue.

Problem 13. Implement Insert(node) Method

Let's implement **inserting a new node**. It should append the new node at the **end of the internal list** holding the binary heap elements and **pull it up** until it finds its correct place in the heap:

```
public void Insert(T node)
{
    this.heap.Add(node);
    HeapifyUp(this.heap.Count - 1);
}
```

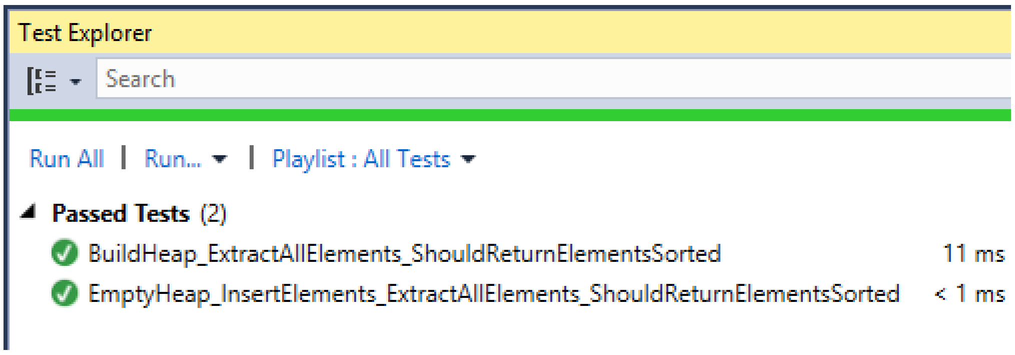
This method relies on the **`Heapify-Up`** operation. It starts from given index and **interchanges** the element at this **index** with its **parent** until the "*heap property*" becomes valid:

```
private void HeapifyUp(int i)
{
    var parent = (i - 1) / 2;
    while (i > 0 && this.heap[i].CompareTo(this.heap[parent]) > 0)
    {
        // Swap heap[i] with heap[parent]
        [illegible]
        [illegible]
        [illegible]

        // Move to the parent node
        [illegible]
        [illegible]
    }
}
```

The above code is **intentionally blurred**. Write it yourself!

Problem 14. Run the Unit Tests Again

Run the unit tests again to check whether the methods testing the "insert" functionality work as expected:



```
◢ Passed Tests (2)
  ✓ BuildHeap_ExtractAllElements_ShouldReturnElementsSorted                  11 ms
  ✓ EmptyHeap_InsertElements_ExtractAllElements_ShouldReturnElementsSorted  < 1 ms
```

Congratulations! You have implemented your binary heap.