# Exercises: AVL Tree and Binary Heap

This document defines the **in-class exercises** assignments for the ["Data Structures" course @ Software University](https://softuni.bg/trainings/1147/Data-Structures-June-2015).

# Part I - Implement an AVL Tree

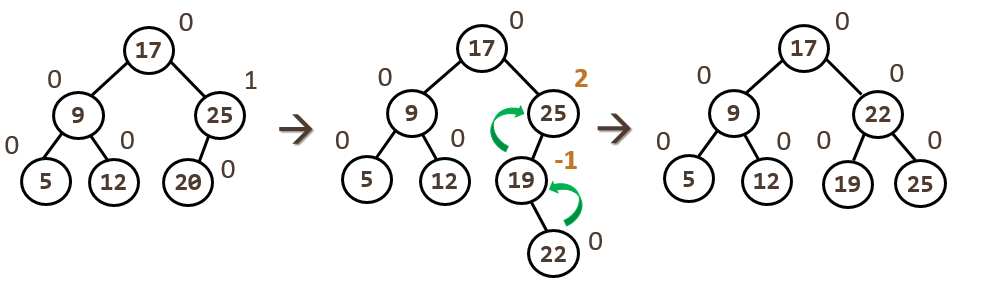
An [AVL tree](https://en.wikipedia.org/wiki/AVL_tree) is a balanced **binary search tree** (BST).

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

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

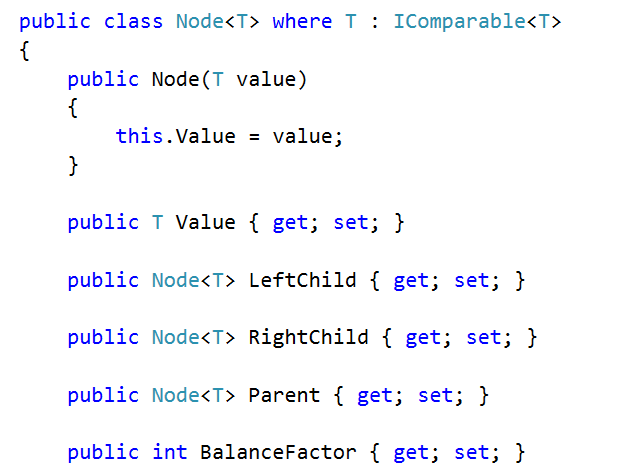
* When a nodeis inserted **its balance factor is 0** (because it's a leaf, it has no left or right subtrees). **Balance factors** are modified each time a subtree grows or decreases (when a node is **inserted** or **deleted**).
* An AVL tree allows 2 subtrees to have a **height difference at most 1**. 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** isdone 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.*



## Define a Node class

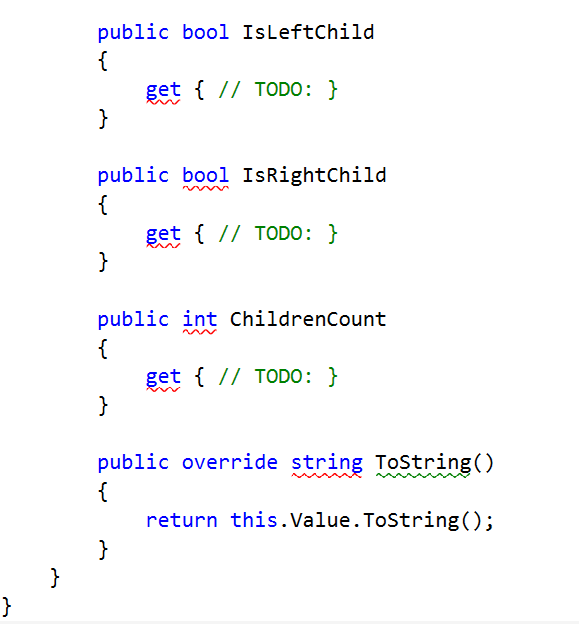
Let's define a node class for representing nodes in our tree. Like in any BST each node will hold its **value**, **left** and **right** children. In addition to that, it should also hold its **parent** + **balance factor** (used by the AVL tree).



## Add Helper Properties

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

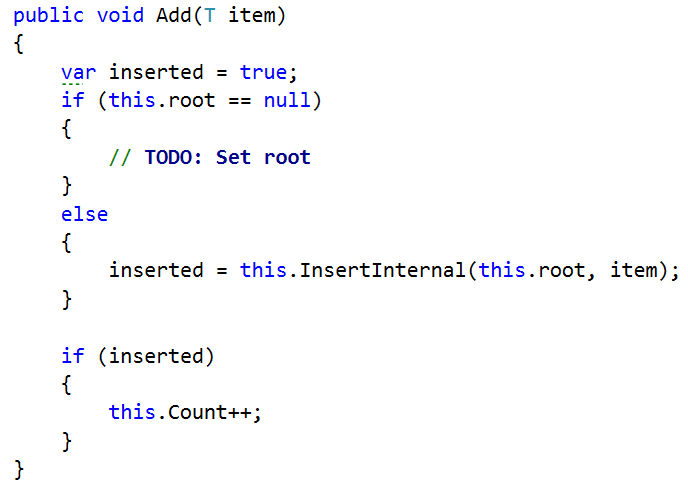
* **IsLeftChild** - returns whether the node is a left child of its parent (if there is no parent, returns false)
* **IsRightChild** - returns whether the node is a right child (if there is no parent, returns false)
* **ChildrenCount** - returns the count of all children
* **ToString()** - returns the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



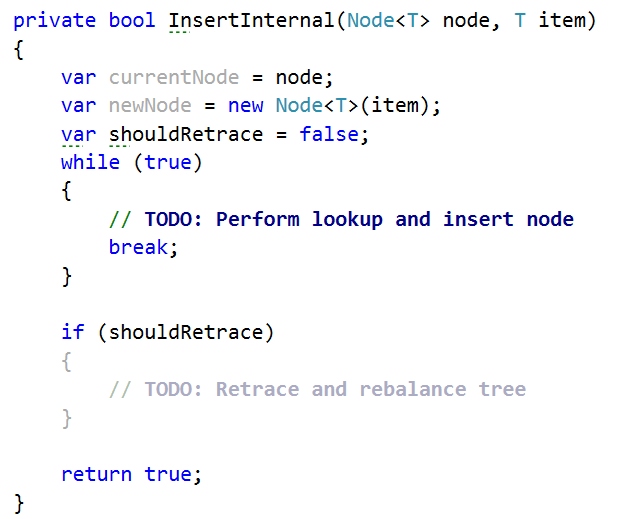
## Adding Elements to the Tree

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.



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**.

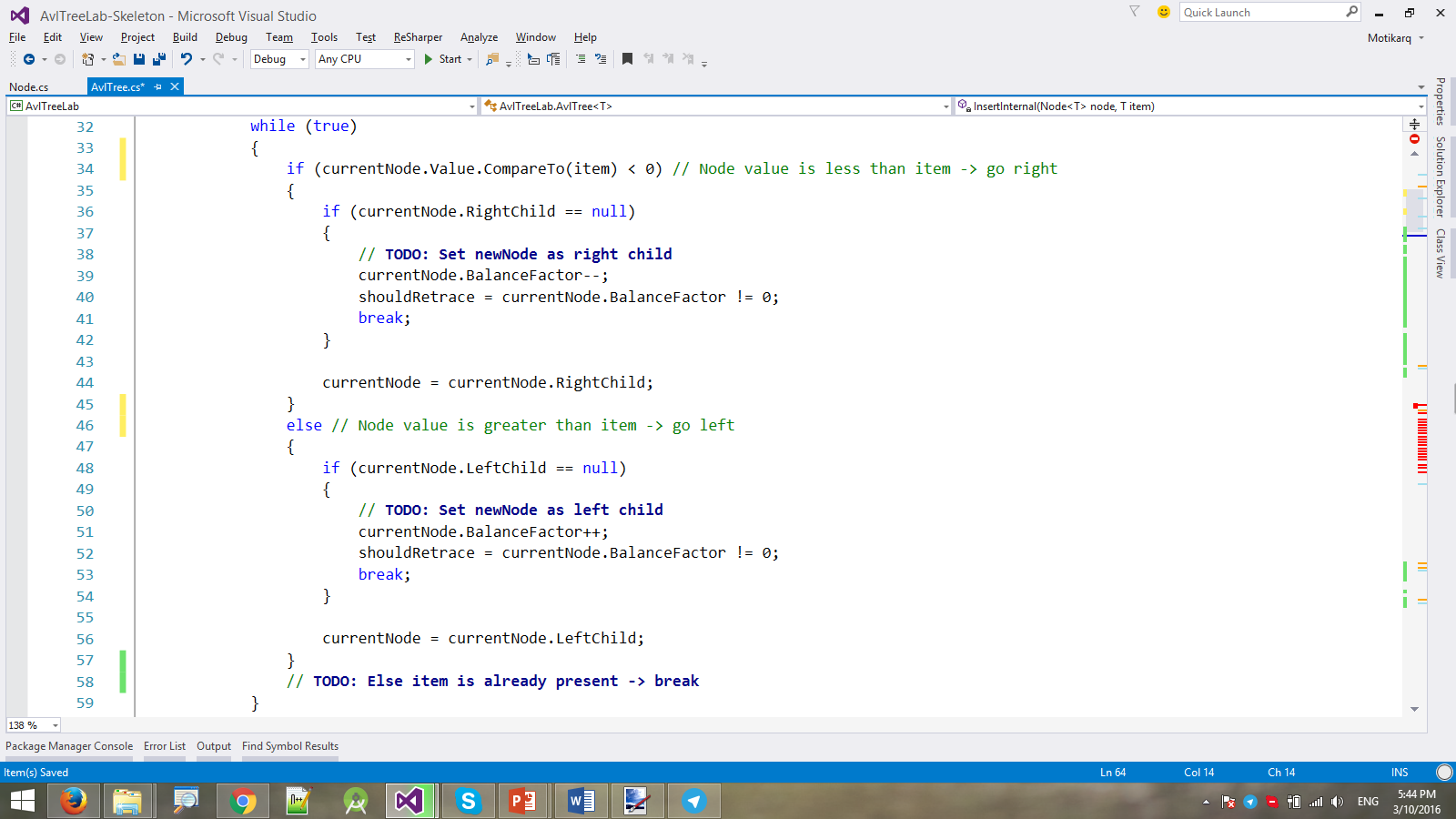


|  |  |
| --- | --- |
| It **starts from the root** and goes down the tree following the rules:   * **Left child** < current node * **Right child** > current node |  |

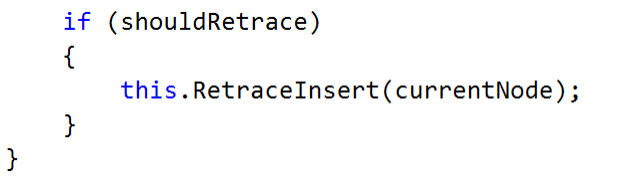
We go down the tree until we **find a leaf node** and **insert the new node** as either left or right child.

* 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**.

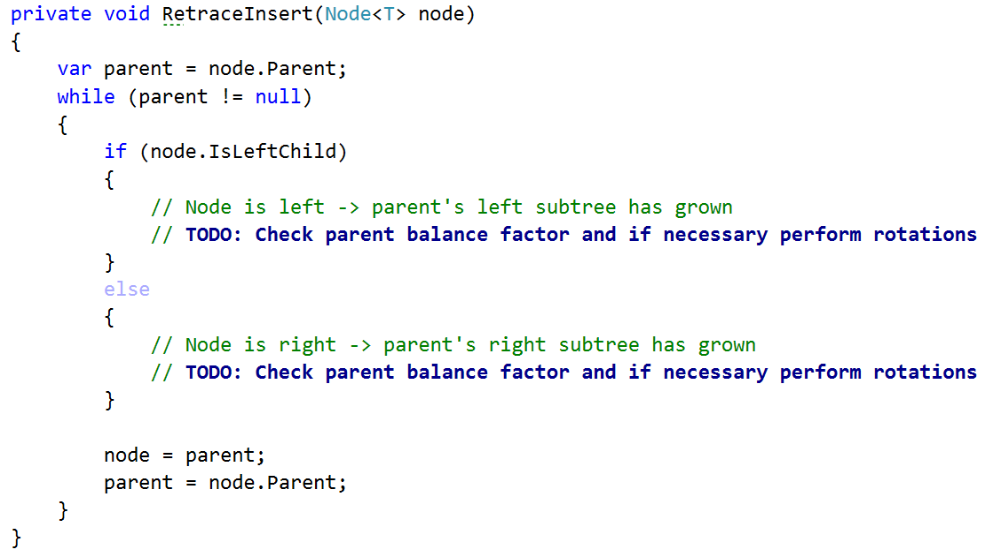
Before we break out of the loop, remember that **bool shouldRetrace** variable? We need to know if retracing (modifying the balance factors up to root) is necessary. When is it necessary? When the subtree holding the parent node grows (when its height increases). In other words, we **only retrace if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



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

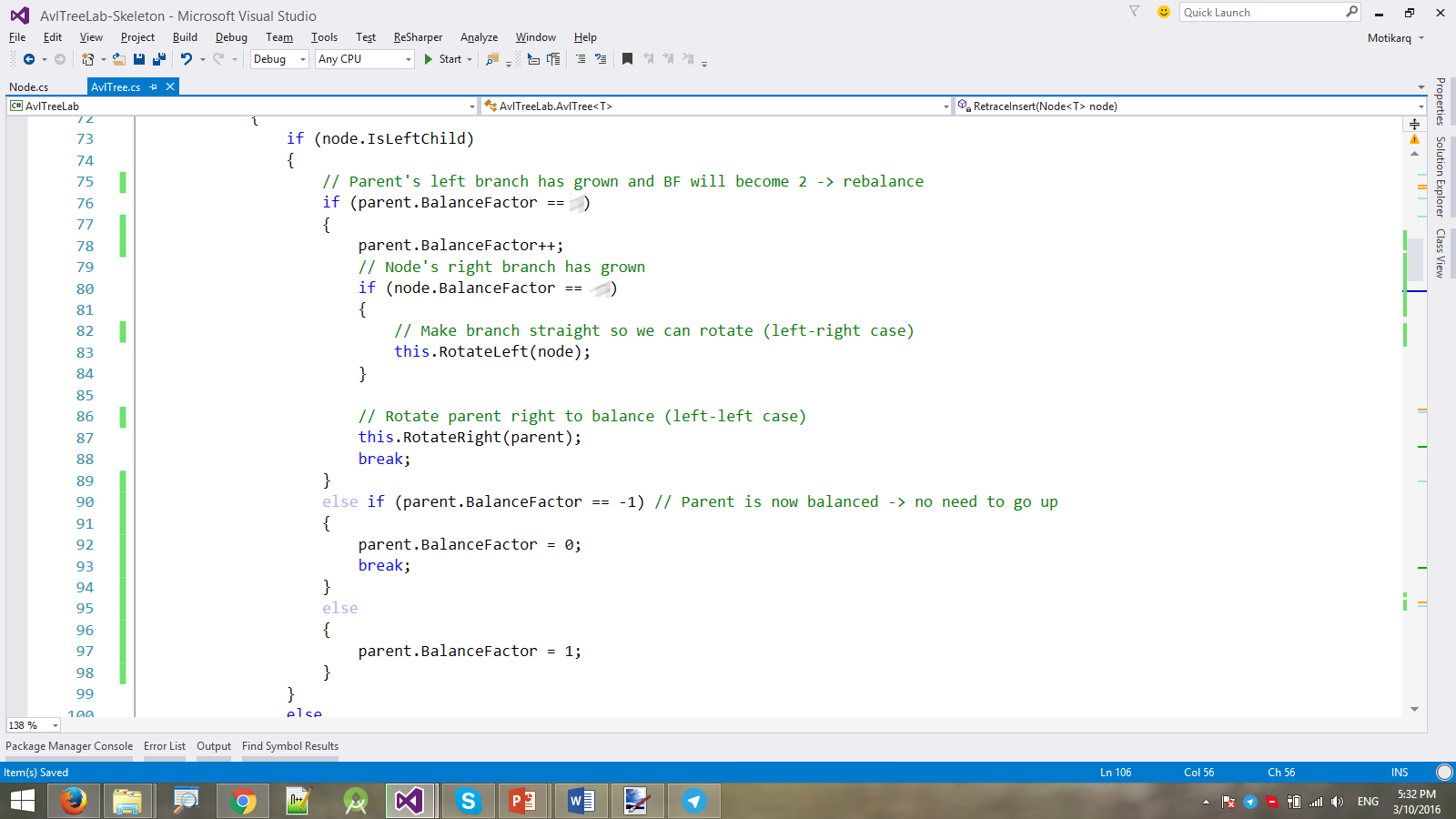


If so, we call method **RetraceInsert(Node<T> startNode)** and begin retracing from the new node's parent to the root.

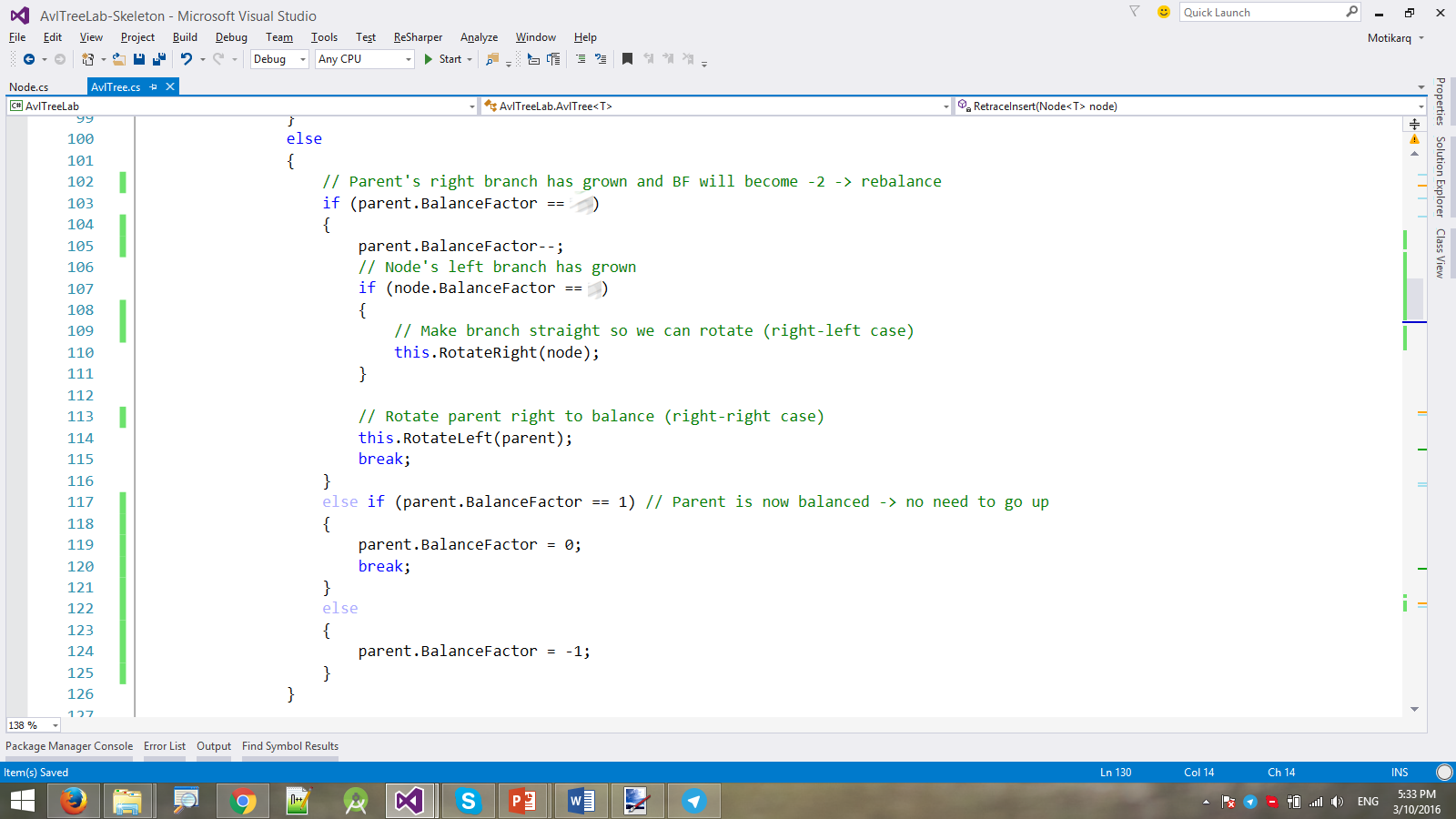


|  |  |
| --- | --- |
| The **retracing loop** "climbs up" back to the root and **modifies the balance factors**, depending on the **direction** we come from.   * If we come from **right** 🡪 reduce **parent BF** * If we come from **left** 🡪 increase **parent BF** | ***22*** *is inserted in the tree* |

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

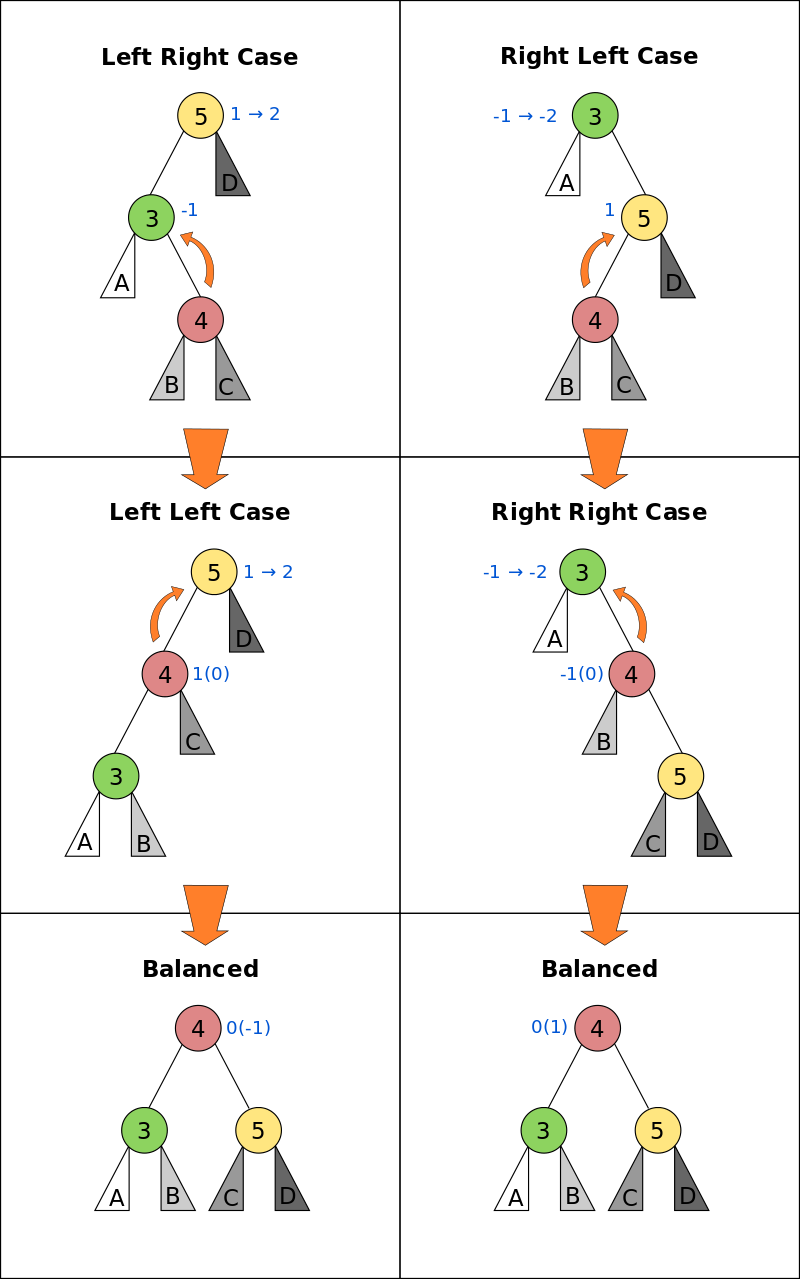


We do the same thing for when we come from right:

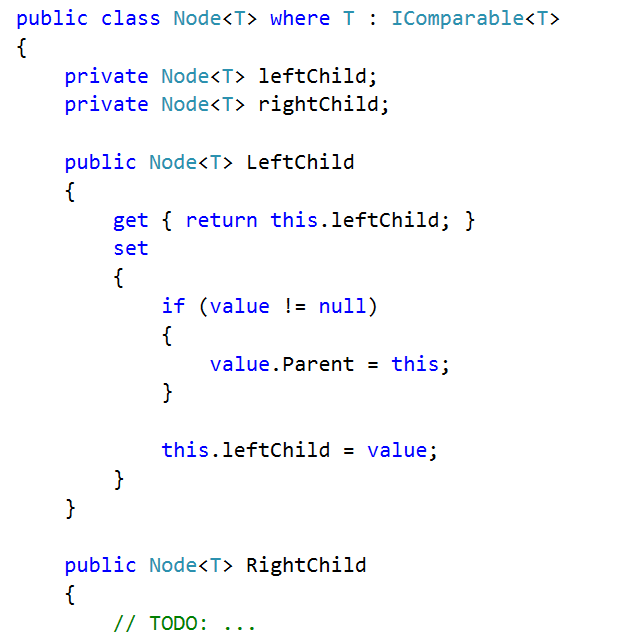


### Rotations

These are 4 different cases when rotating:



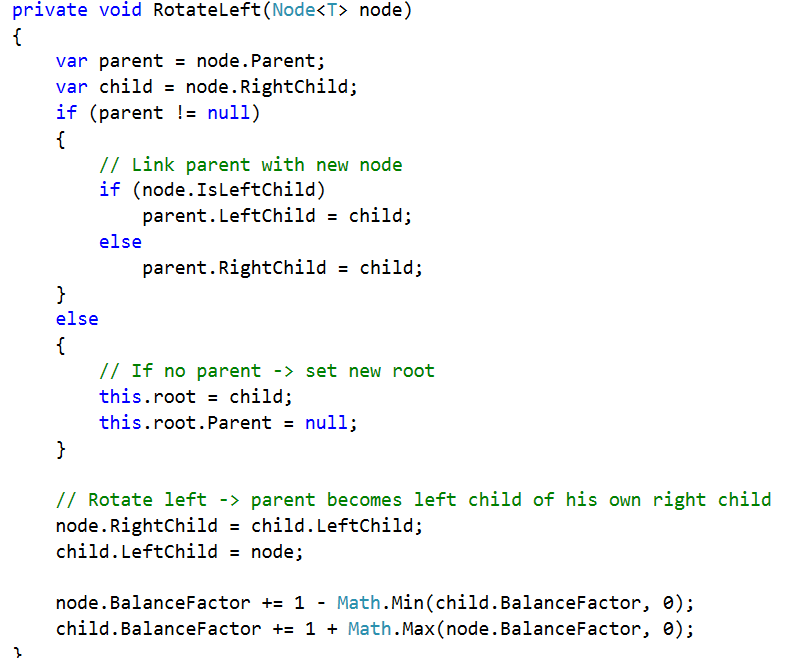
Add another helper method (property in this case). Adding a **node N** as left or right child to **node P**, automatically sets his parent link as well.



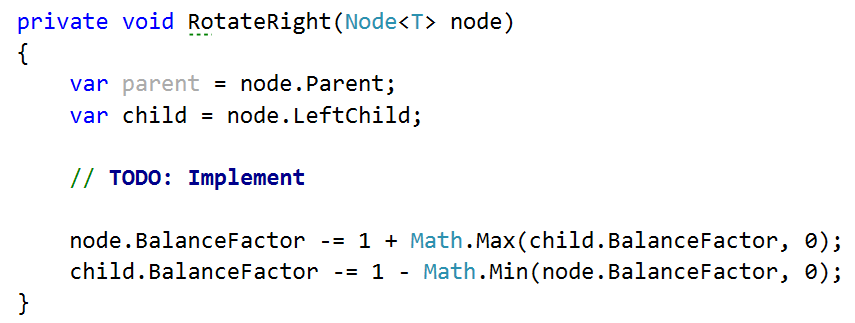
An AVL tree has 2 types of rotations - left and right as seen below:

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

The method performing the left rotation looks as follows:

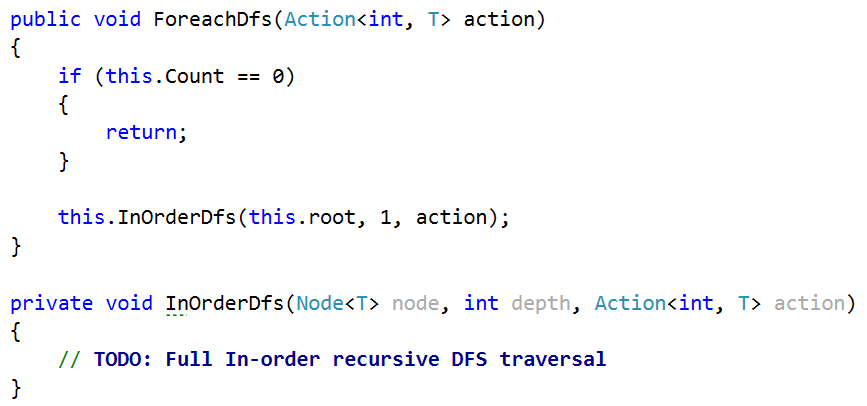


Likewise, implement the rotate right method. Use the illustrations above for reference.



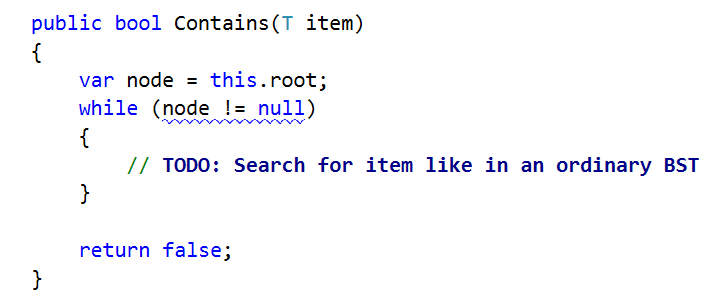
And voila! Your insertion into the AVL tree should be ready.

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.

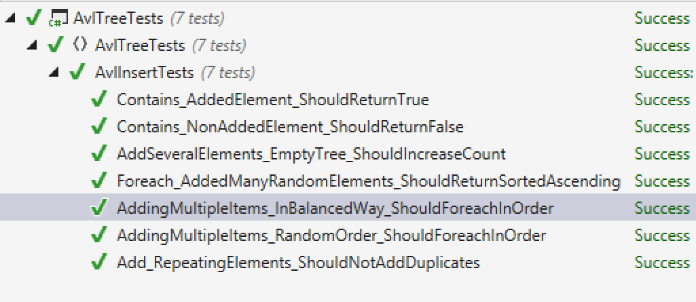


## Contains Method

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.



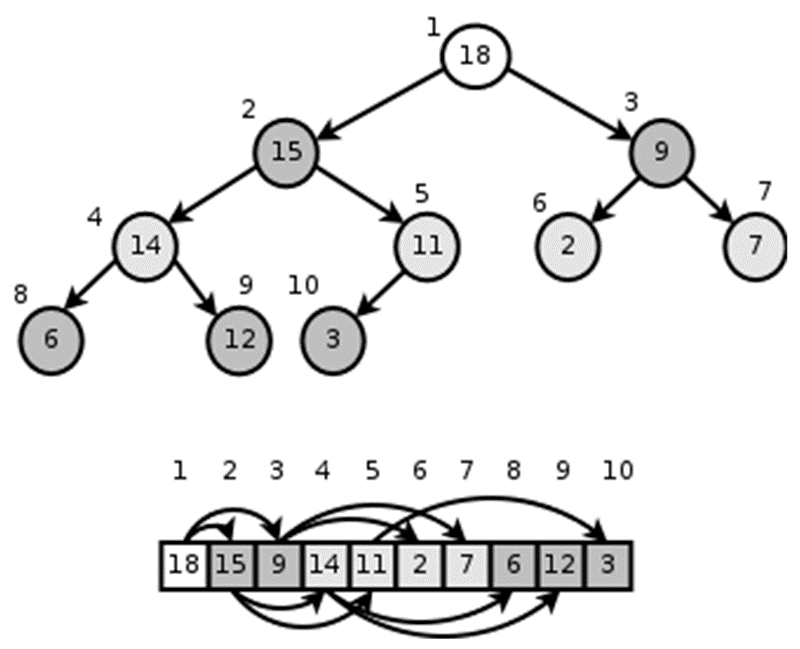
The unit tests should now all pass.



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.

## Learn about Binary Heap in Wikipedia

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

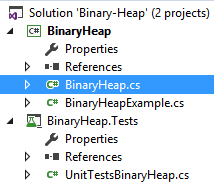
The typical **operations** over a binary heap are:

* 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!

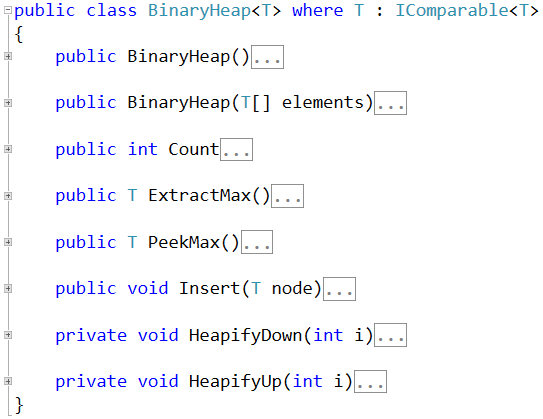
## BinaryHeap<T > – Project Skeleton

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:

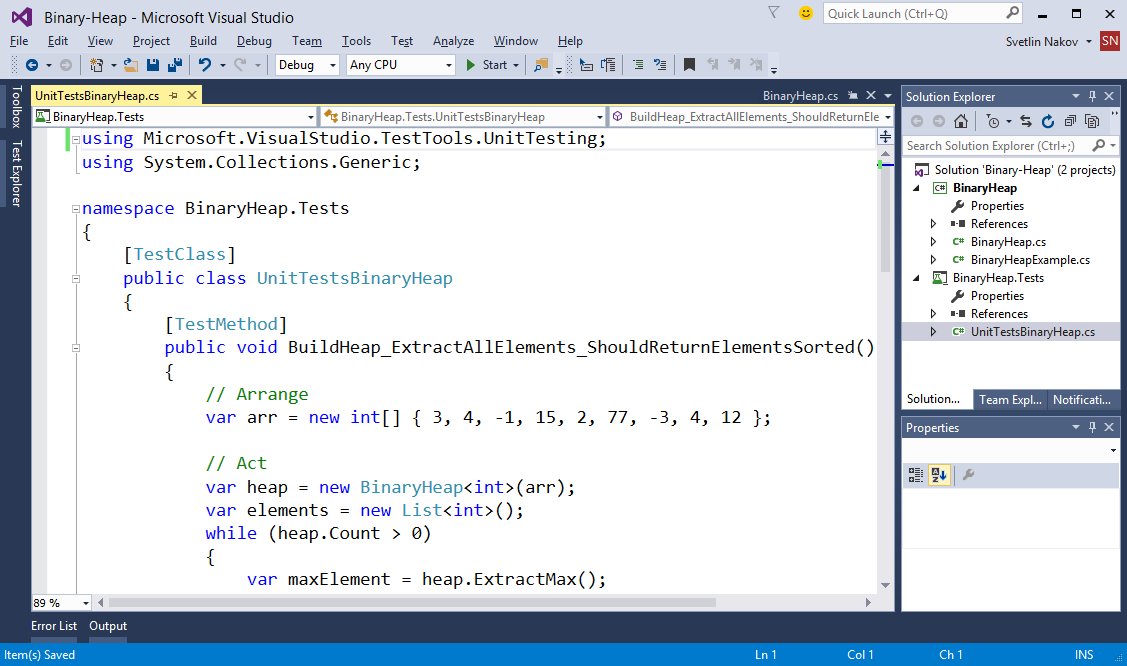


The project skeleton opens correctly in **Visual Studio 2013** but can be open in other Visual Studio versions as well and also can run in **SharpDevelop** and **Xamarin Studio**. Your goal is to implement the missing functionality in order to finish the project.

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:

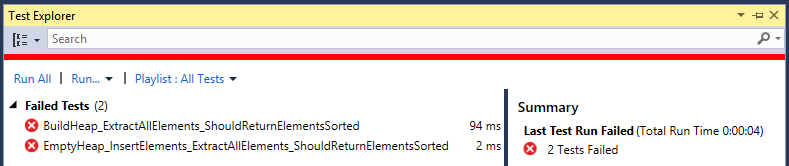


The project comes also with **unit tests** covering the functionality of the **binary heap** (see the class UnitTestBinaryHeap):



## 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:



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

## 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):

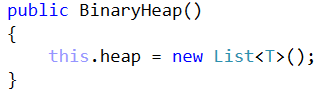


## Implement the Binary Heap Constructor

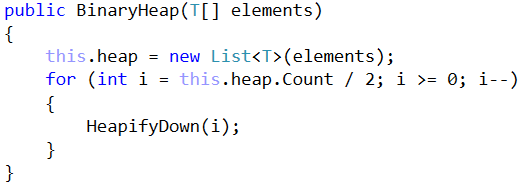
Now, let's implement the binary heap **constructor**. Its purpose is to allocate the internal array that will hold the binary heap elements (balanced binary tree). The binary heap constructor has two forms:

* 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:

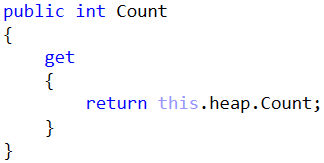


The **second constructor** is more complex:



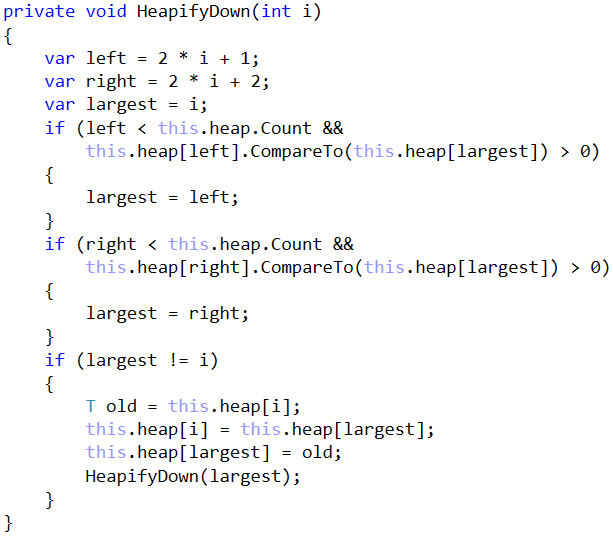
The above code first **allocates the internal list** to hold the binary heap elements, then **fills** the passed as argument elements in the internal list and then "***heapifies***" the elements. This means that each **element** becomes **less or equal to its parent**. This happens by moving up each 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:



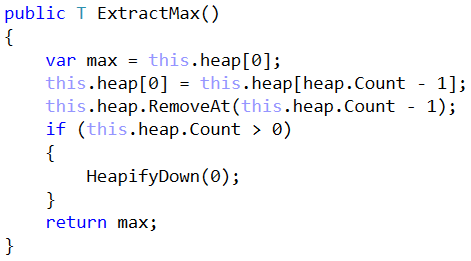
## Implement HeapifyDown(index) Method

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:



## Implement the ExtractMax() Method

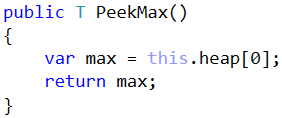
Now, we are ready to implement the most important method ExtractMax() which returns and removes the maximal element:



How it works? It works in thee steps:

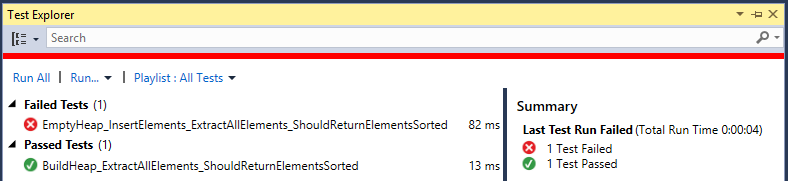
1. Takes as result the **maximal element** – the elements at **index 0** (the root node in the tree).
2. **Deletes the last element** from the internal list holding the heap elements and **moves it at position 0** (as root node).
3. Moves down the root node to **apply the "*heap property*"**, i.e. call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

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



## Run the Unit Tests

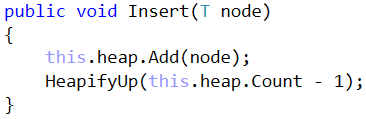
We have **partially implemented** the binary heap. It supports Build-Heap and Extract-Max operations. Let's run the tests. We can expect some of them to pass:



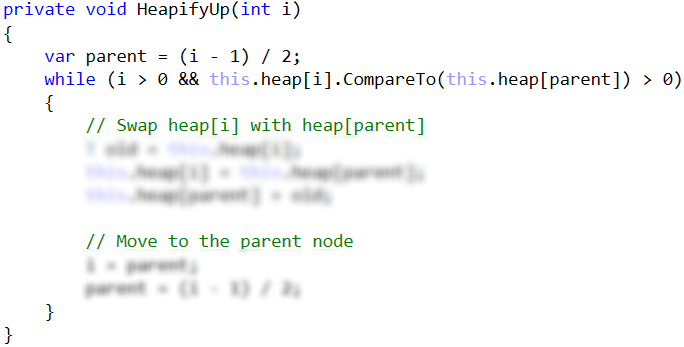
To have more tests passed, we need to implement the rest of the functionality. Let's continue.

## 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:



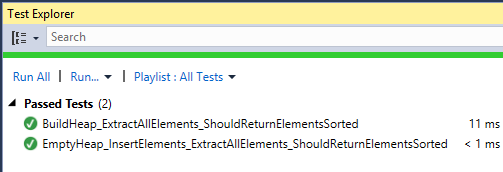
This method relies on the Heapify-Up operation. It starts from given index and **interchanges** the element at this **index** with its **parent** until the "*heap property*" becomes valid:



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

## Run the Unit Tests Again

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



**Congratulations!** You have implemented your binary heap.