# 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

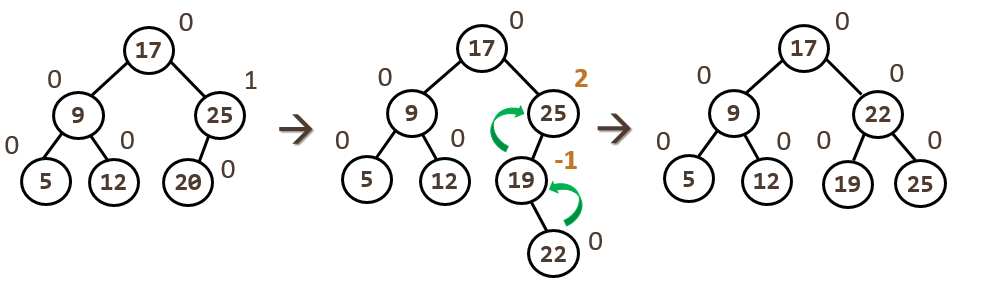
An [AVL tree](https://en.wikipedia.org/wiki/AVL_tree) is a balanced **binary search tree** (BST).

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

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

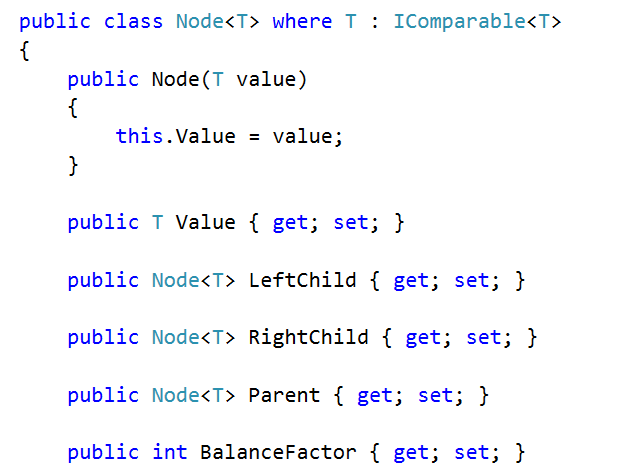
* When a nodeis inserted **its balance factor is 0** (because it's a leaf, it has no left or right subtrees). **Balance factors** are modified each time a subtree grows or decreases (when a node is **inserted** or **deleted**).
* An AVL tree allows 2 subtrees to have a **height difference at most 1**.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

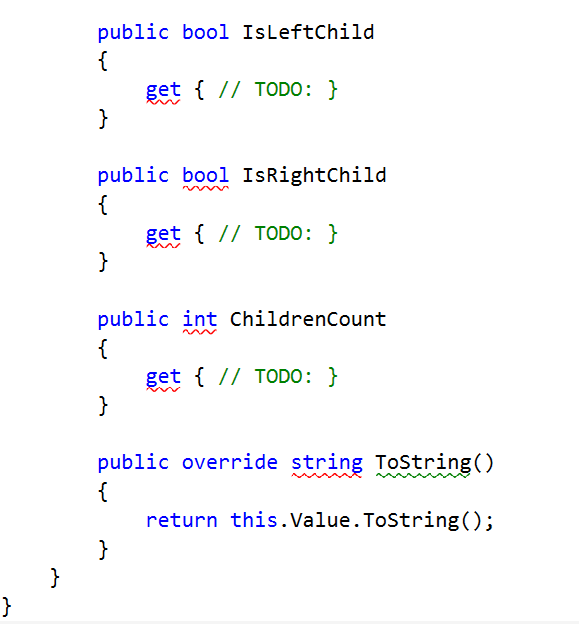
Let's define a node class for representing nodes in our tree. Like in any BST each node will hold its **value**, **left** and **right** children. In addition to that, it should also hold its **parent** + **balance factor** (used by the AVL tree).



## Add Helper Properties

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

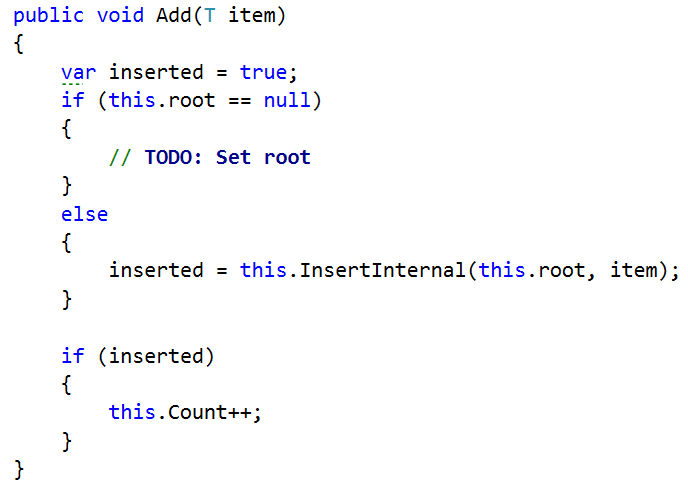
* **IsLeftChild** - returns whether the node is a left child of its parent (if there is no parent, returns false)
* **IsRightChild** - returns whether the node is a right child (if there is no parent, returns false)
* **ChildrenCount** - returns the count of all children
* **ToString()** - returns the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



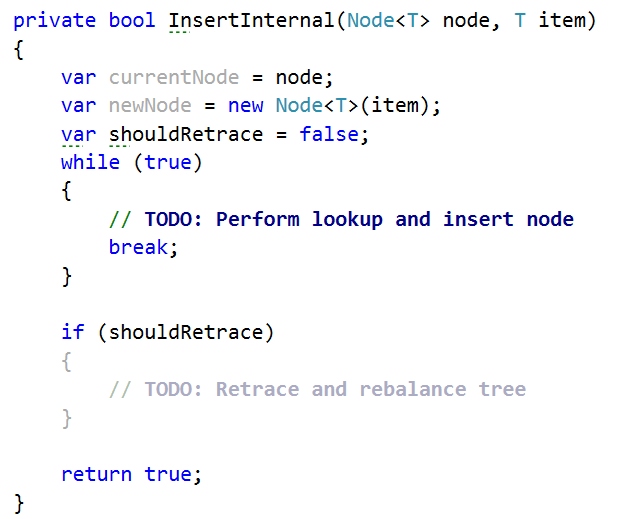
## Adding Elements to the Tree

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**.

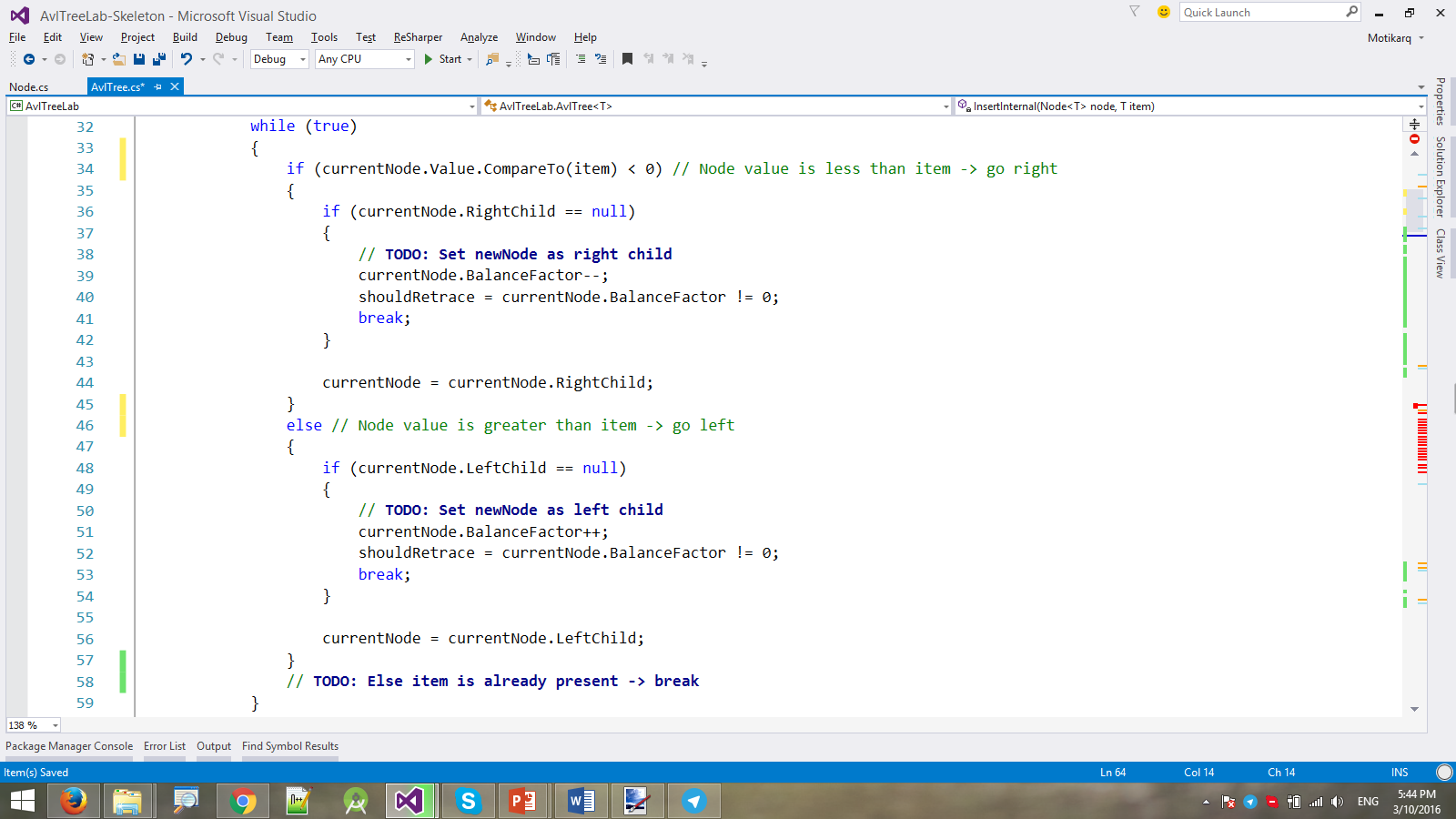


|  |  |
| --- | --- |
| It **starts from the root** and goes down the tree following the rules:   * **Left child** < current node * **Right child** > current node |  |

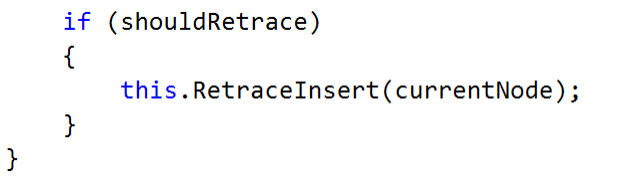
We go down the tree until we **find a leaf node** and **insert the new node** as either left or right child.

* 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**.

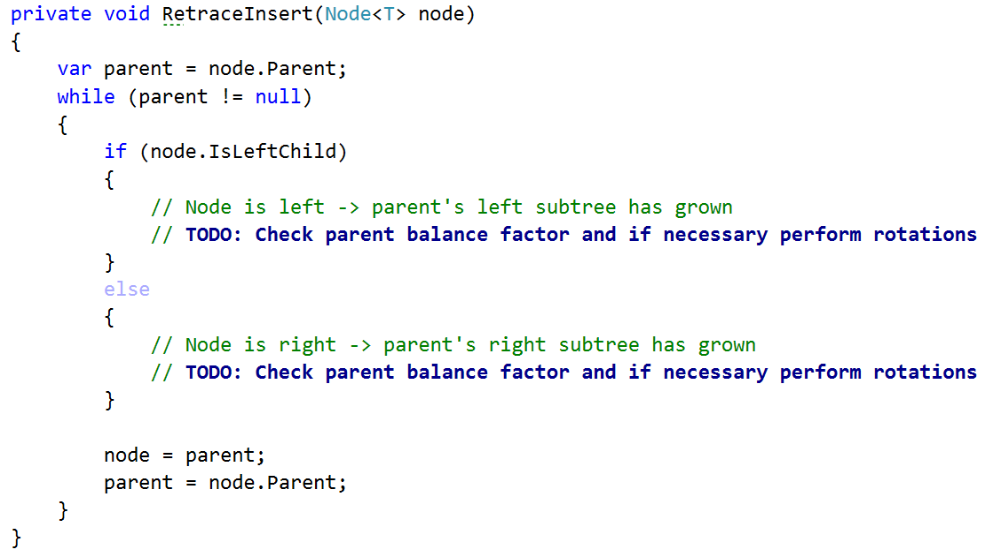
Before we break out of the loop, remember that **bool shouldRetrace** variable? We need to know if retracing (modifying the balance factors up to root) is necessary. When is it necessary? When the subtree holding the parent node grows (when its height increases). In other words, we **only retrace if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



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

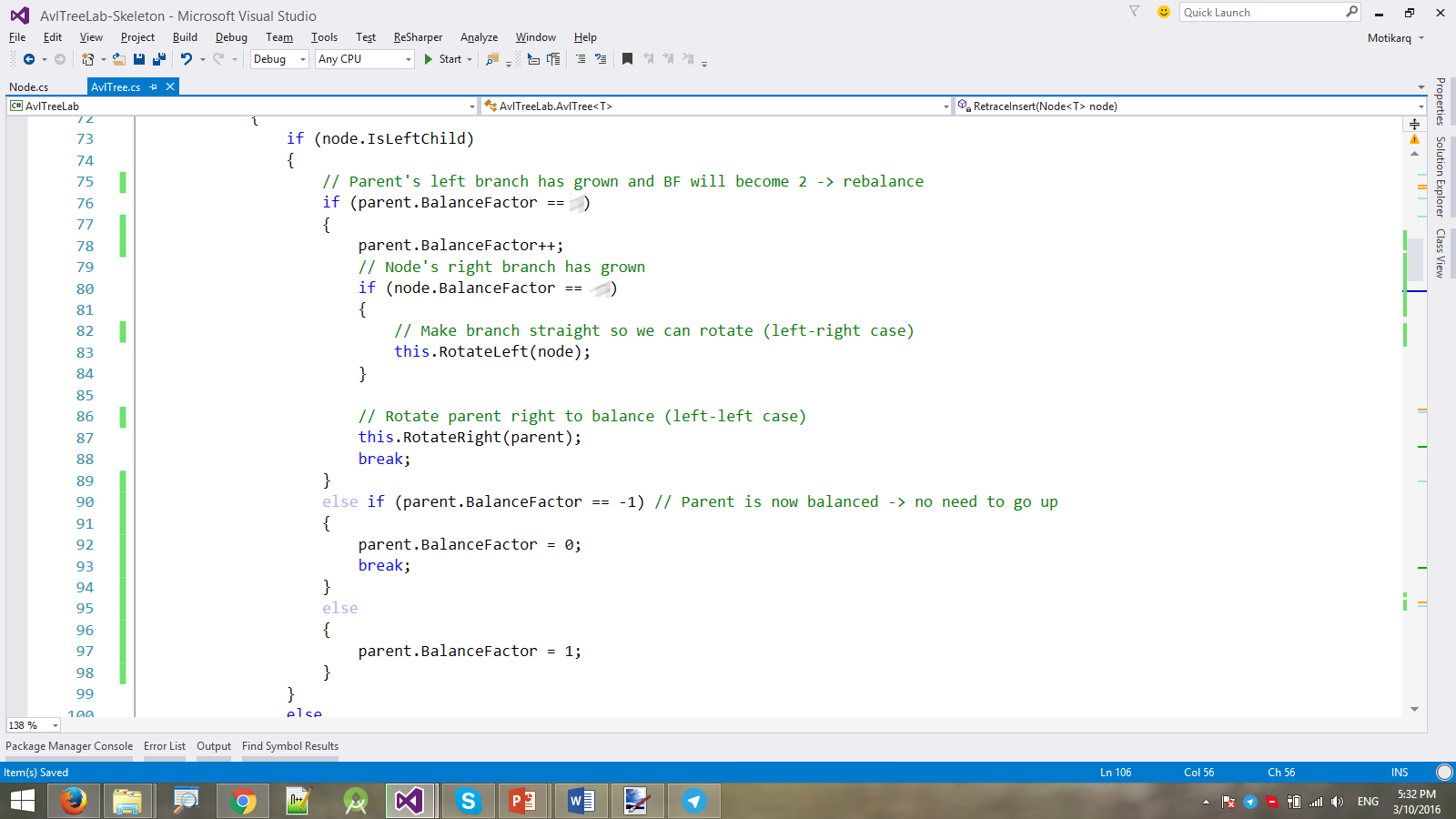


If so, we call method **RetraceInsert(Node<T> startNode)** and begin retracing from the new node's parent to the root.

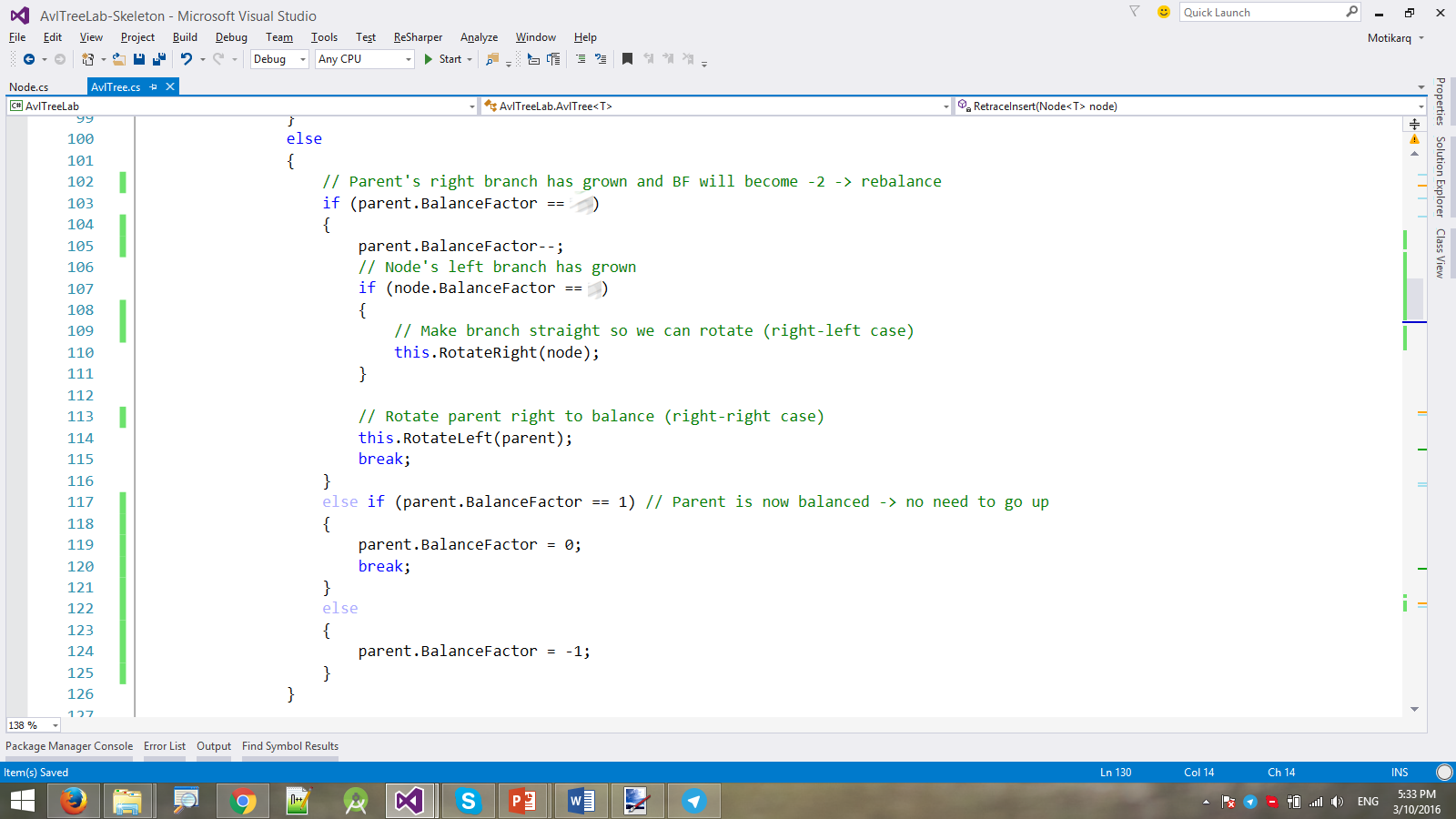


|  |  |
| --- | --- |
| The **retracing loop** "climbs up" back to the root and **modifies the balance factors**, depending on the **direction** we come from.   * If we come from **right** 🡪 reduce **parent BF** * If we come from **left** 🡪 increase **parent BF** | ***22*** *is inserted in the tree*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

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

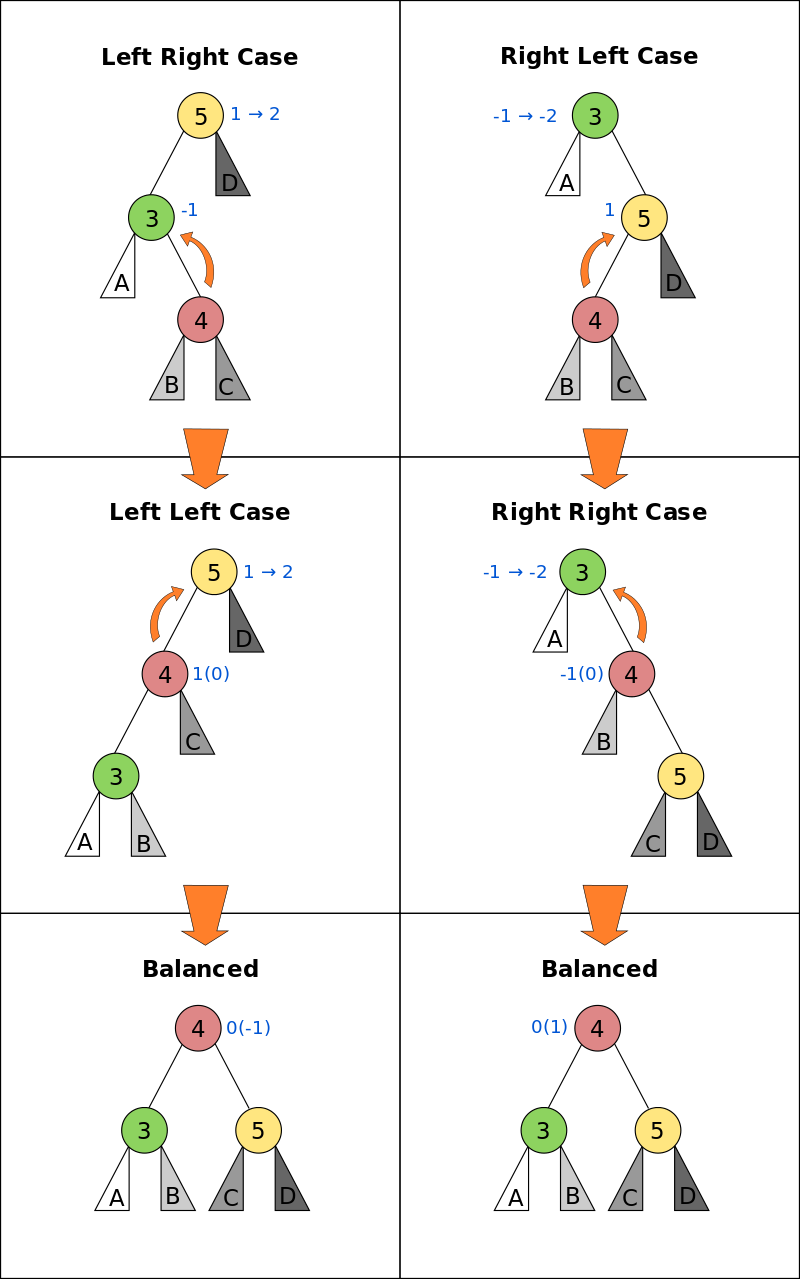


We do the same thing for when we come from right:

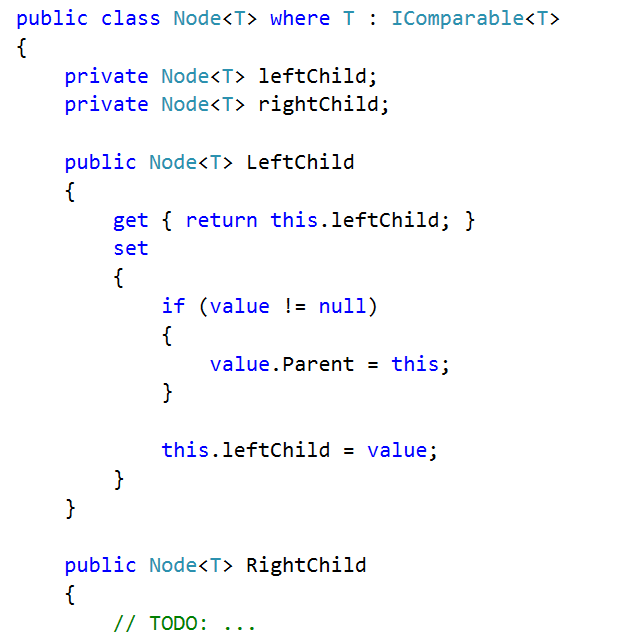


### Rotations

These are 4 different cases when rotating:



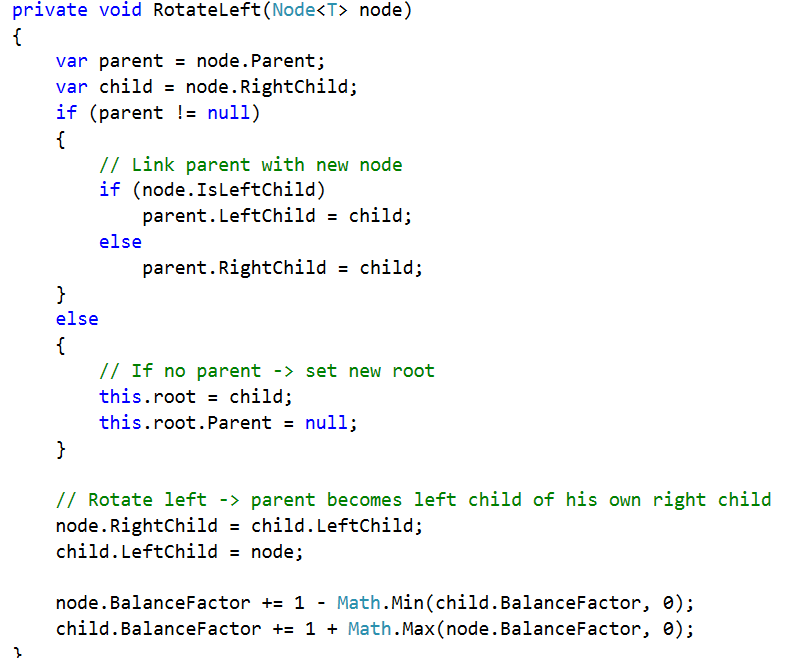
Add another helper method (property in this case). Adding a **node N** as left or right child to **node P**, automatically sets his parent link as well.



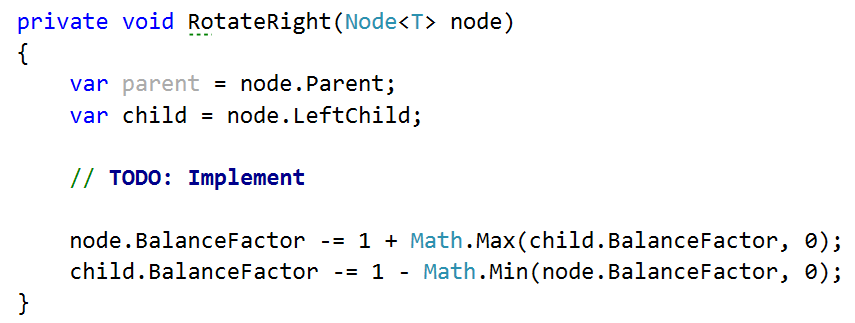
An AVL tree has 2 types of rotations - left and right as seen below:

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

The method performing the left rotation looks as follows:

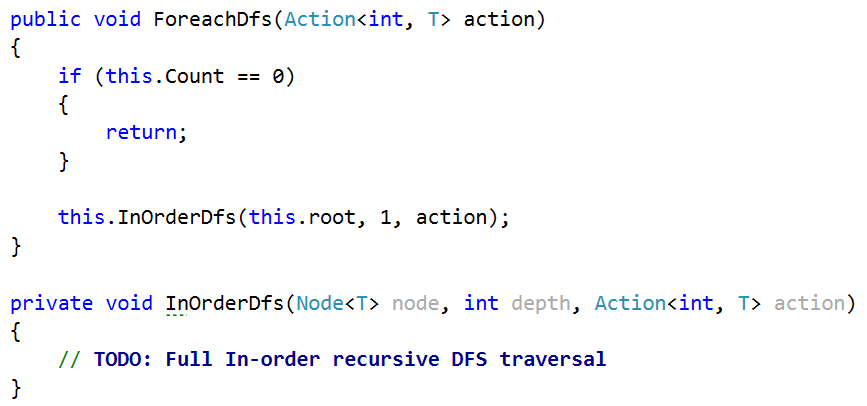


Likewise, implement the rotate right method. Use the illustrations above for reference.



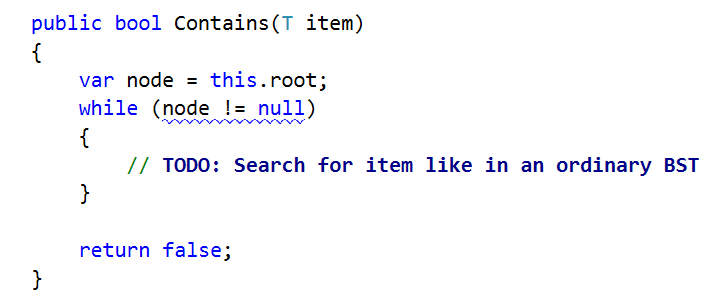
And voila! Your insertion into the AVL tree should be ready.

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.

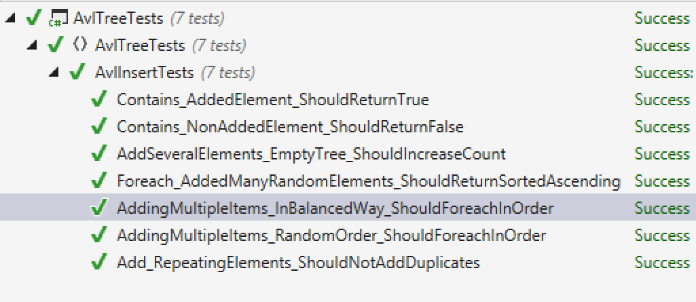


## Contains Method

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!