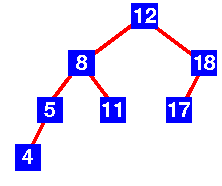
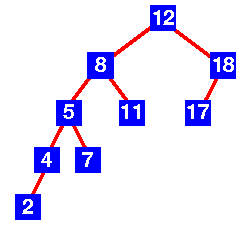
# AVL Tree | Set 1 (Insertion)

AVL tree is a self-balancing Binary Search Tree (BST) where the difference between heights of left and right subtrees cannot be more than one for all nodes.

**An Example Tree that is an AVL Tree**  
[](http://d1hyf4ir1gqw6c.cloudfront.net/wp-content/uploads/AVLTREE1.gif)  
The above tree is AVL because differences between heights of left and right subtrees for every node is less than or equal to 1.

**An Example Tree that is NOT an AVL Tree**  
[](http://d1hyf4ir1gqw6c.cloudfront.net/wp-content/uploads/AVLTREE2.gif)  
The above tree is not AVL because differences between heights of left and right subtrees for 8 and 18 is greater than 1.

Images are taken from [here](https://www.cs.auckland.ac.nz/~jmor159/PLDS210/AVL.html).

**Why AVL Trees?**  
Most of the BST operations (e.g., search, max, min, insert, delete.. etc) take O(h) time where h is the height of the BST. The cost of these operations may become O(n) for a skewed Binary tree. If we make sure that height of the tree remains O(Logn) after every insertion and deletion, then we can guarantee an upper bound of O(Logn) for all these operations. The height of an AVL tree is always O(Logn) where n is the number of nodes in the tree (See [this](http://www.youtube.com/watch?v=TbvhGcf6UJU)video lecture for proof).

**Insertion**  
To make sure that the given tree remains AVL after every insertion, we must augment the standard BST insert operation to perform some re-balancing. Following are two basic operations that can be performed to re-balance a BST without violating the BST property (keys(left) < key(root) < keys(right)). 1) Left Rotation 2) Right Rotation

T1, T2 and T3 are subtrees of the tree rooted with y (on left side)

or x (on right side)

y x

/ \ Right Rotation / \

x T3 – – – – – – – > T1 y

/ \ < - - - - - - - / \

T1 T2 Left Rotation T2 T3

Keys in both of the above trees follow the following order

keys(T1) < key(x) < keys(T2) < key(y) < keys(T3)

So BST property is not violated anywhere.

**Steps to follow for insertion**  
Let the newly inserted node be w  
**1)** Perform standard BST insert for w.  
**2)** Starting from w, travel up and find the first unbalanced node. Let z be the first unbalanced node, y be the child of z that comes on the path from w to z and x be the grandchild of z that comes on the path from w to z.  
**3)** Re-balance the tree by performing appropriate rotations on the subtree rooted with z. There can be 4 possible cases that needs to be handled as x, y and z can be arranged in 4 ways. Following are the possible 4 arrangements:  
a) y is left child of z and x is left child of y (Left Left Case)  
b) y is left child of z and x is right child of y (Left Right Case)  
c) y is right child of z and x is right child of y (Right Right Case)  
d) y is right child of z and x is left child of y (Right Left Case)

Following are the operations to be performed in above mentioned 4 cases. In all of the cases, we only need to re-balance the subtree rooted with z and the complete tree becomes balanced as the height of subtree (After appropriate rotations) rooted with z becomes same as it was before insertion. (See [this](http://www.youtube.com/watch?v=TbvhGcf6UJU)video lecture for proof)

**a) Left Left Case**

T1, T2, T3 and T4 are subtrees.

z y

/ \ / \

y T4 Right Rotate (z) x z

/ \ - - - - - - - - -> / \ / \

x T3 T1 T2 T3 T4

/ \

T1 T2

**b) Left Right Case**

z z x

/ \ / \ / \

y T4 Left Rotate (y) x T4 Right Rotate(z) y z

/ \ - - - - - - - - -> / \ - - - - - - - -> / \ / \

T1 x y T3 T1 T2 T3 T4

/ \ / \

T2 T3 T1 T2

**c) Right Right Case**

z y

/ \ / \

T1 y Left Rotate(z) z x

/ \ - - - - - - - -> / \ / \

T2 x T1 T2 T3 T4

/ \

T3 T4

**d) Right Left Case**

z z x

/ \ / \ / \

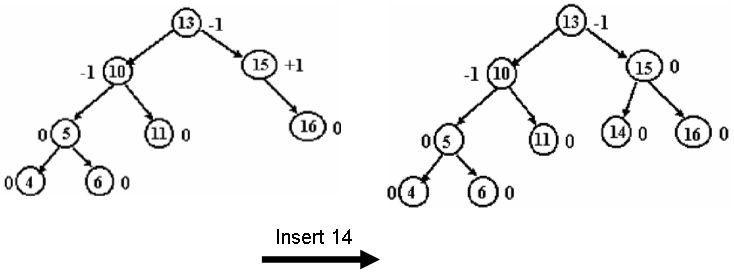
T1 y Right Rotate (y) T1 x Left Rotate(z) z y

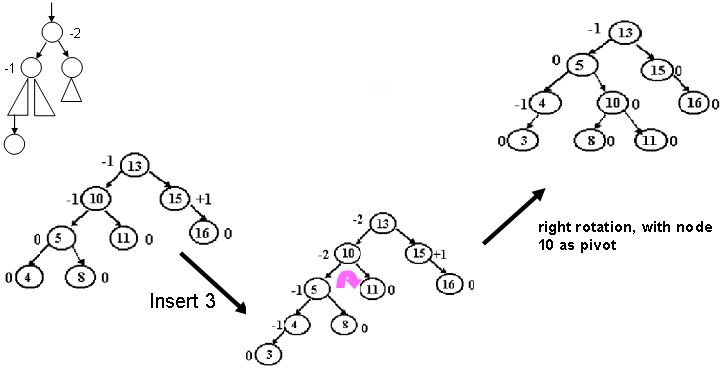
/ \ - - - - - - - - -> / \ - - - - - - - -> / \ / \

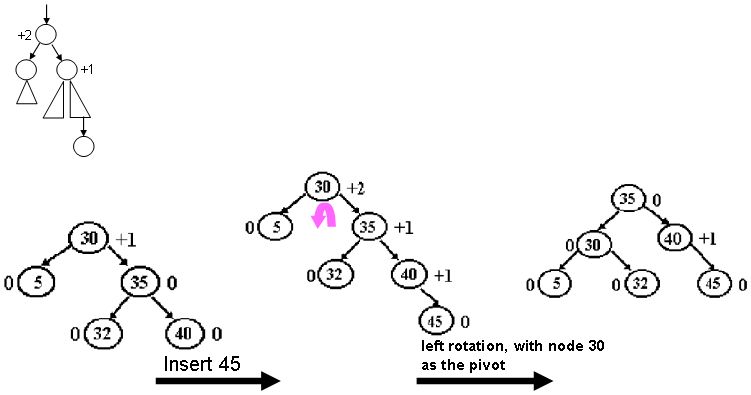
x T4 T2 y T1 T2 T3 T4

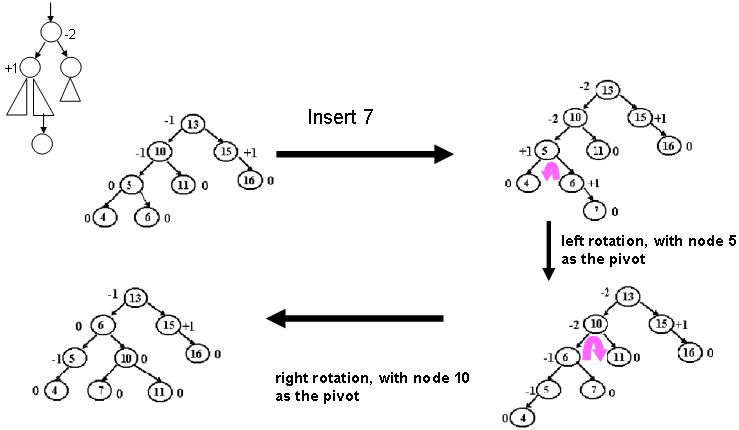
/ \ / \

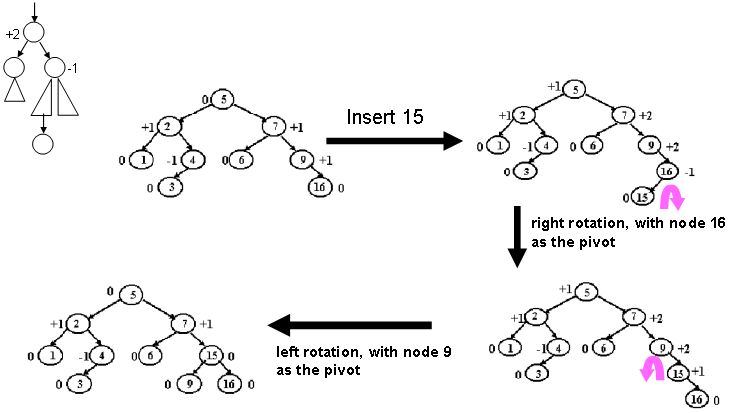
T2 T3 T3 T4

**Insertion Examples:**  
[](http://d1hyf4ir1gqw6c.cloudfront.net/wp-content/uploads/AVLINSERT1.jpg)

[](http://d1hyf4ir1gqw6c.cloudfront.net/wp-content/uploads/AVLINSERT2.jpg.jpg)

[](http://d1hyf4ir1gqw6c.cloudfront.net/wp-content/uploads/AVLINSERT3.jpg)

[](http://d1hyf4ir1gqw6c.cloudfront.net/wp-content/uploads/AVLINSERT4.jpg)

[](http://d1hyf4ir1gqw6c.cloudfront.net/wp-content/uploads/AVLINSERT5.jpg)

The above images are taken from [here](http://faculty.kfupm.edu.sa/ICS/said/files/ics202Labs/Lab08/lab08.htm).

## [We strongly recommend that you click here and practice it, before moving on to the solution.](http://www.practice.geeksforgeeks.org/problem-page.php?pid=700382)

**implementation**  
Following is the implementation for AVL Tree Insertion. The following implementation uses the recursive BST insert to insert a new node. In the recursive BST insert, after insertion, we get pointers to all ancestors one by one in bottom up manner. So we don’t need parent pointer to travel up. The recursive code itself travels up and visits all the ancestors of the newly inserted node.  
1) Perform the normal BST insertion.  
2) The current node must be one of the ancestors of the newly inserted node. Update the height of the current node.  
3) Get the balance factor (left subtree height – right subtree height) of the current node.  
4) If balance factor is greater than 1, then the current node is unbalanced and we are either in Left Left case or left Right case. To check whether it is left left case or not, compare the newly inserted key with the key in left subtree root.  
5) If balance factor is less than -1, then the current node is unbalanced and we are either in Right Right case or Right Left case. To check whether it is Right Right case or not, compare the newly inserted key with the key in right subtree root.