Self-Balancing Trees (AVL)

Background:

- Issues with BST, the runtime of search, insert, and deletion is proportional to the height of the tree
- The expected height of the tree is logn, so O(logn)
- However the worst case is when the data is inserted in sorted order, the height of the tree is n, O(n)
- We use a self balancing tree to achieve a worst case height of O(logn)
    - AVL Trees
    - Red Black Trees

AVL Tree:

- A binary search tree that balances it height after insertion and deletion.
- Height balanced tree:
    - This means that for any node, the heights of its two subtrees differ by at most 1
    - The height of an AVL tree with n nodes is O(logn)
    - **Proof**
        - Let n(h) = the minimum number of nodes in a tree of height h.
        - This minimal tree has a root and subtrees of heights h-1 and h-2,
        - where each subtree is also a minimal tree. Therefore,
        - $n(h) = 1 + n(h-1) + n(h-2)$
        - $n(h) > n(h-1) + n(h-2)$
        - $n(h) > n(h-2) + n(h-2)$ (because $n(h-1) > n(h-2)$)
        - $n(h) > 2 \cdot n(h-2)$
        - $n(h) > 2^2 \cdot n(h-4)$ (because $n(h-2) > 2n(h-4)$)
        - ... (repeat)
        - $n(h) > 2^{(h-1)/2} \cdot n(1)$ (if h is odd)
        - $n(h) > 2^{(h-2)/2} \cdot n(2)$ (if h is even)
        - $n(h) > 2^{(h-2)/2}$ (true in both cases)
        - $\log(n(h)) > (h-2)/2$ (take the log of both sides)
        - $h < 2\log(n(h)) + 2$ (multiply by 2, then add 2)
        - $h < 2\log n + 2$ (because $n(h) \leq n$)
        - $h = O(\log n)$
- **Balanced/Unbalanced:**
    - Given a tree T, we say that a position is **balanced** if, the absolute value of the difference in height between its children is at most 1. Otherwise it is **unbalanced**
- **Insertion and Deletions:**
    - We begin with a normal insertion or deletion as with BST. From the new/deleted node, we move upward through the tree until we find the first unbalanced node (a node whose subtrees differ in height by more than 1).
    - Let Z be the unbalanced node

- Let Y be the child of Z with greater height, and X be the child of Y with greater height
- In insertion, all 3 will be ancestors of the new node. In deletion, if Y has 2 children with the same height, choose X such that both directions are the same.
- y is left child of z and x is left child of y (Left Left Case)
- y is left child of z and x is right child of y (Left Right Case)
- y is right child of z and x is right child of y (Right Right Case)
- y is right child of z and x is left child of y (Right Left Case)

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
  / \      - - - - - - - - ->    /  \      - - - - - - - ->  / \
T1   x                          y    T3                    T1  T2
    / \                        / \
  T2   T3                    T1   T2
```

c) Right Right Case

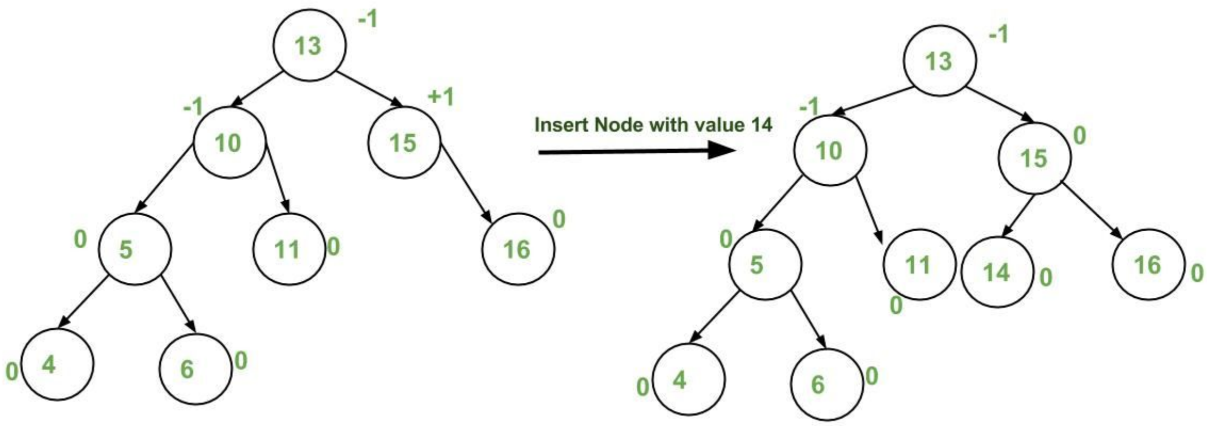
```
  z                                y
 /  \                            /   \
T1   y     Left Rotate(z)       z      x
    /  \   - - - - - - - ->    / \    / \
   T2   x                     T1  T2 T3  T4
       / \
     T3  T4
```

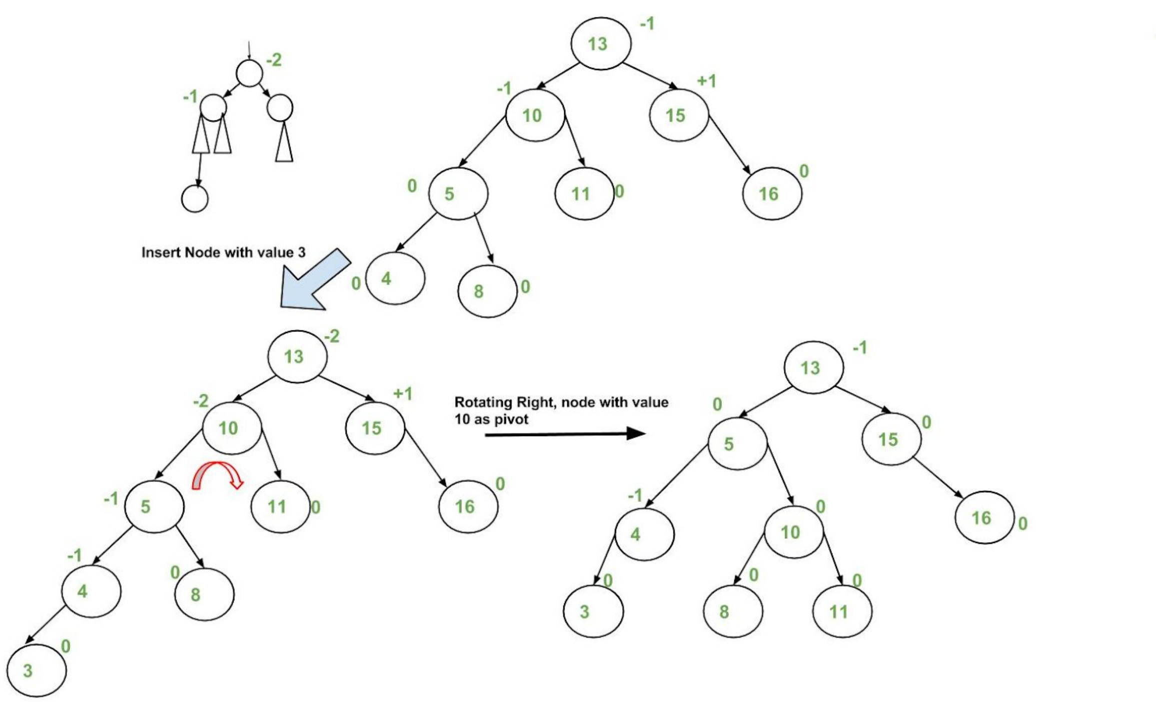
d) Right Left Case

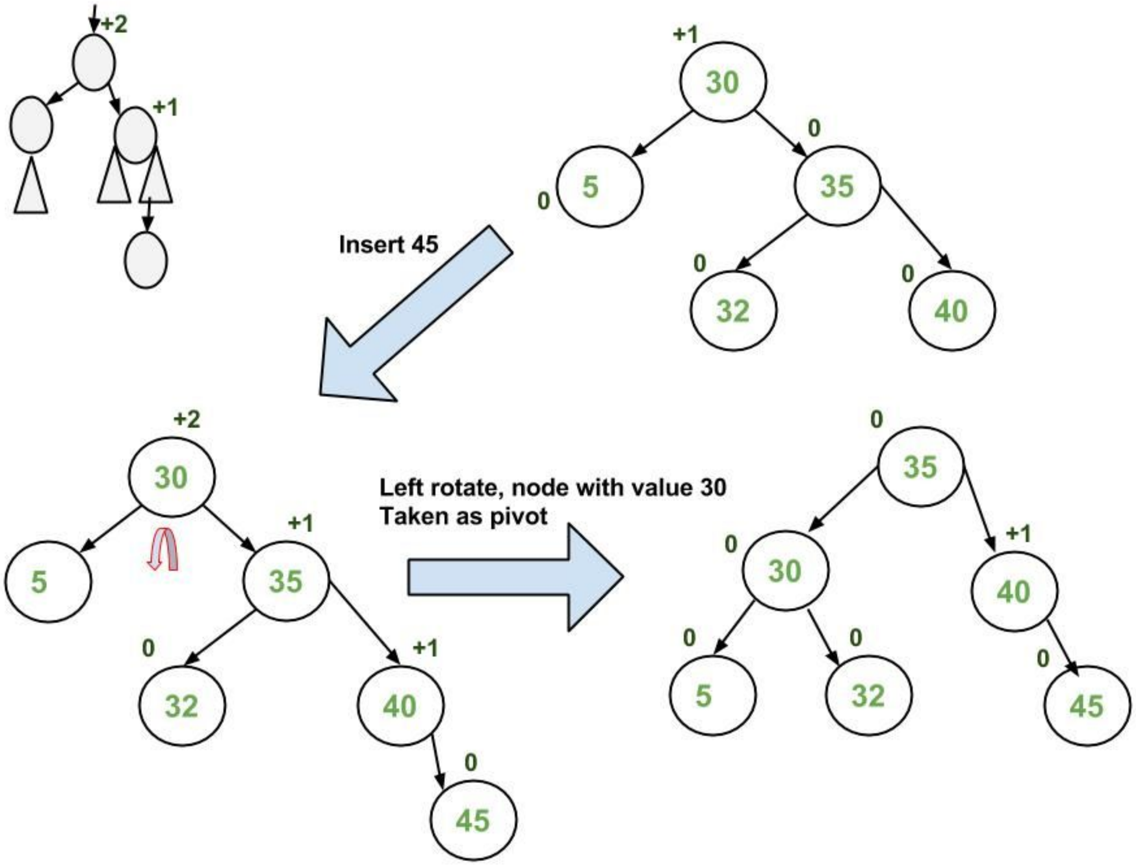
```
   z                            z                            x
  / \                          / \                          /  \
T1   y   Right Rotate (y)    T1   x      Left Rotate(z)   z      y
    / \  - - - - - - - - ->     /  \   - - - - - - - ->  / \    /
   x  T4                       T2   y                   T1  T2 T3
  / \                              /  \
T2   T3                          T3   T4
```

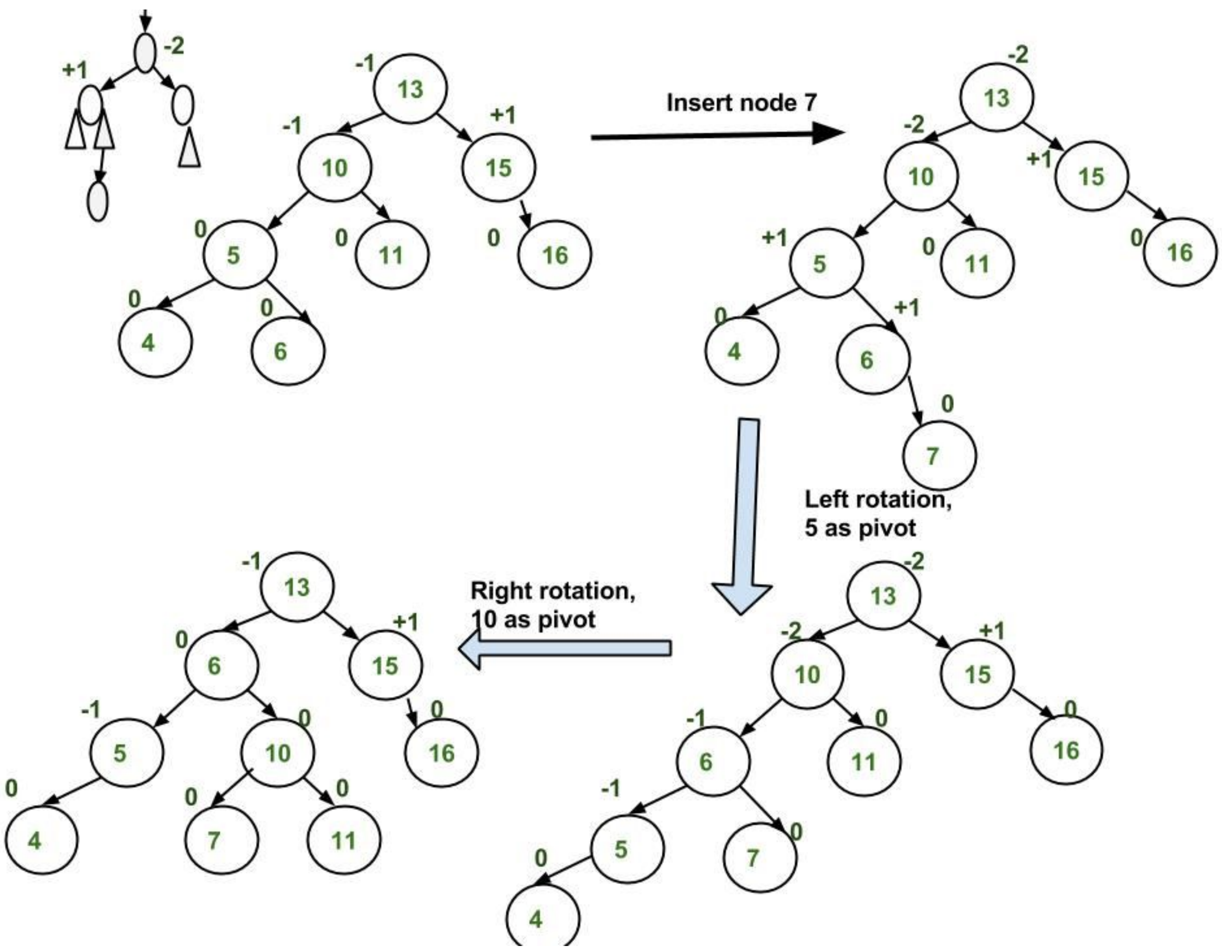
- **Rotations:**

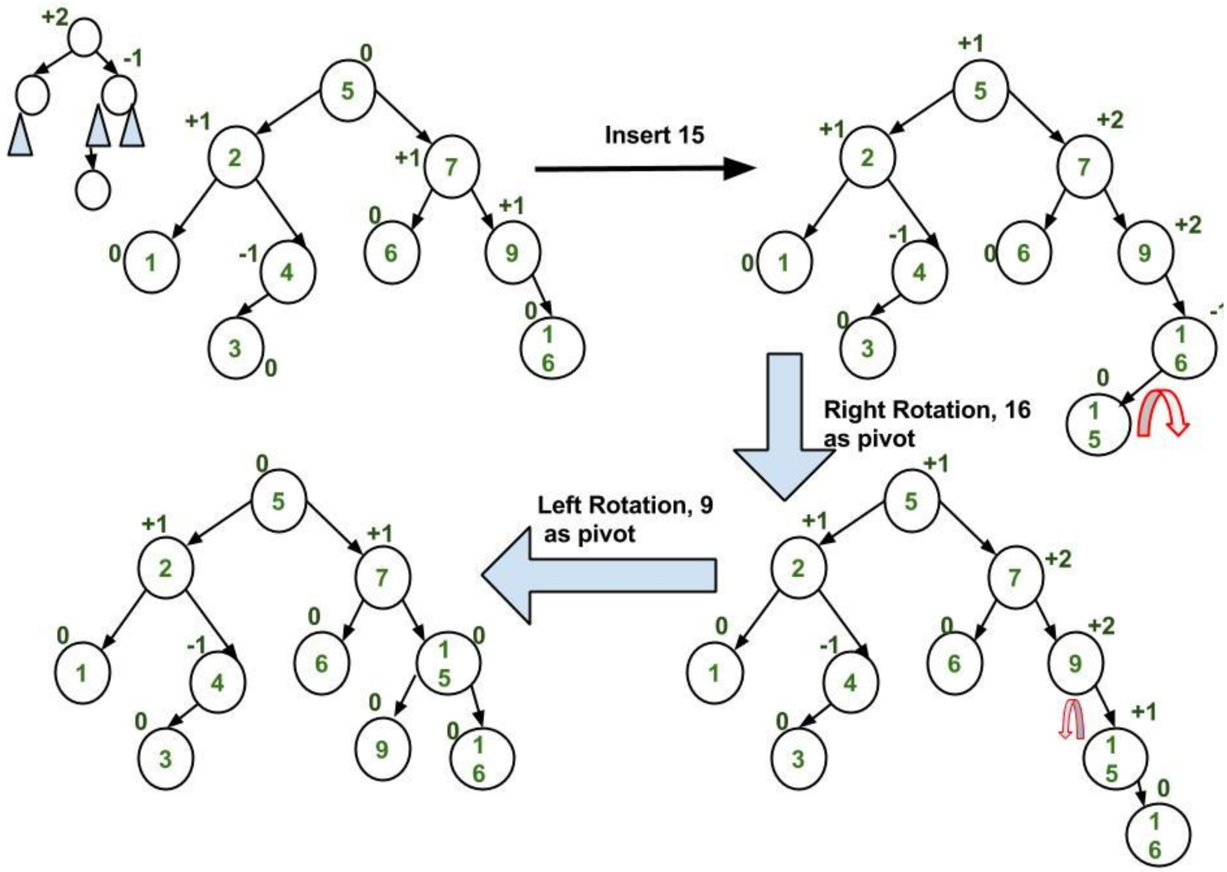
- After any insertion or deletion, if the tree is unbalanced, we perform a single or double rotation to rebalance the tree
- A rotation changes the structure of a subtree so that one of the root's children becomes the new root.
    - There are two types of rotation, a left rotation and a right rotation






Insert 45
Left rotate, node with value 30
Taken as pivot


Insert node 7
Left rotation,
5 as pivot
Right rotation,
10 as pivot


Insert 15
Right Rotation, 16 as pivot
Left Rotation, 9 as pivot