CHAPTER IV

BALANCED TREE: AVL

4.1 Learning Objectives

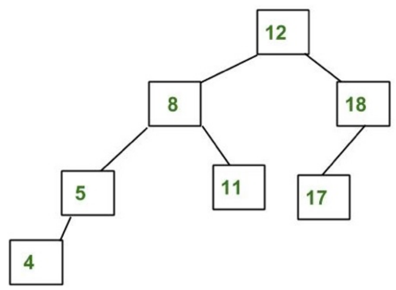
1. Able to understand the concept of AVL Tree
2. Able to implement AVL Tree

4.2 Materials

4.2.1 AVL Tree

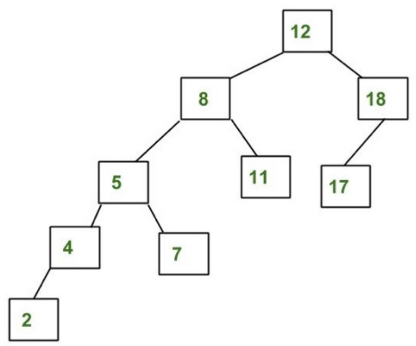
AVL tree is a self-balancing Binary Search Tree (BST) where the difference between heights of left and right subtrees cannot be more than one for all nodes.

An Example Tree that is an AVL Tree



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



The above tree is not AVL because differences between heights of left and right subtrees for 8 and 12 is greater than 1.

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.

4.2.2 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 the left side) or x (on
the 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.

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                    T1T2T3T4
    / \                       / \
  T2   T3                   T1   T2
```

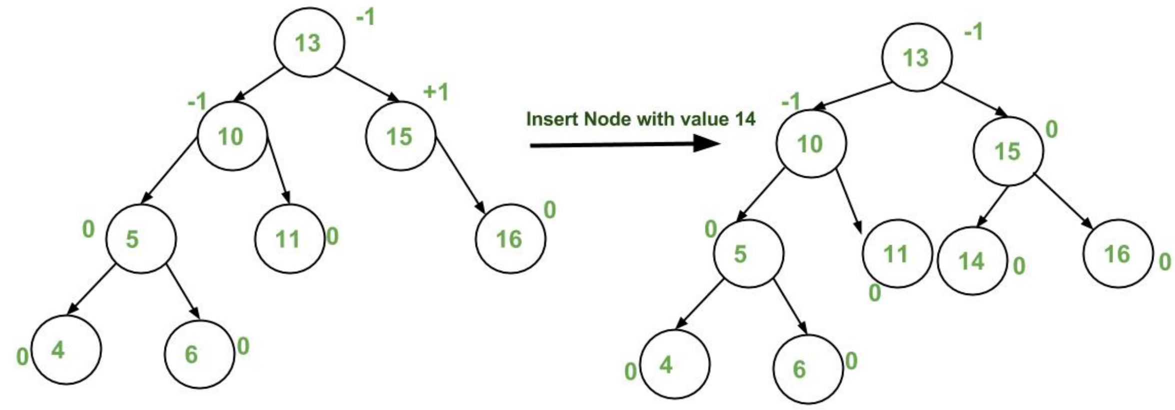
c) Right Right Case

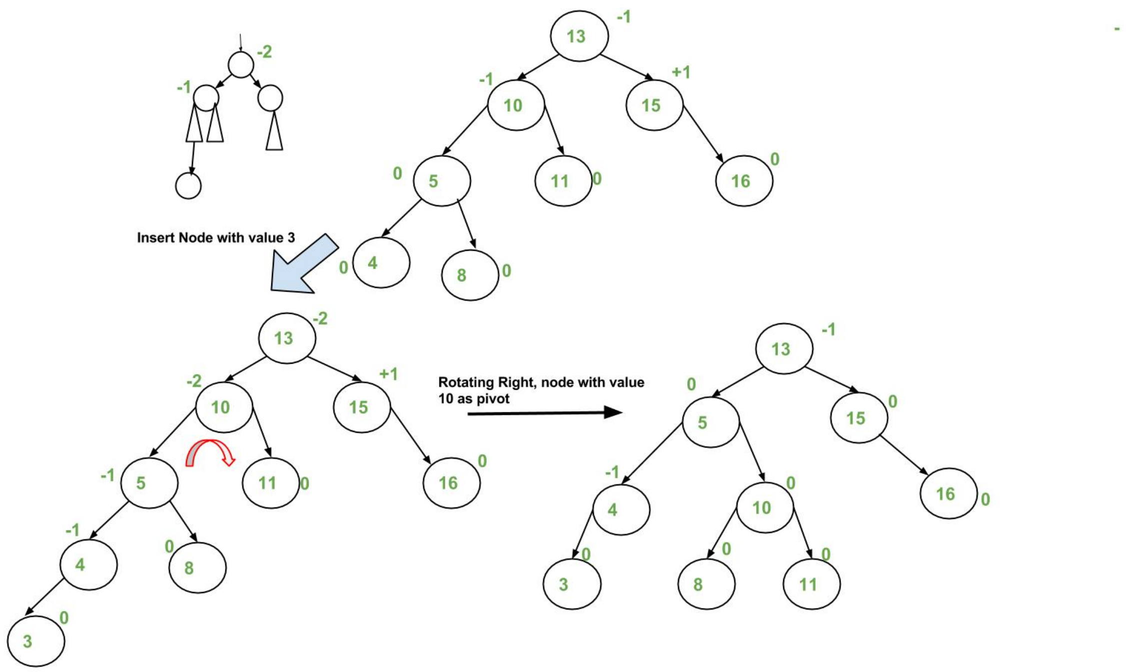
```
  z                                y
 /  \                            /   \
T1   y     Left Rotate(z)       z      x
    /  \   - - - - - - - ->    / \    / \
   T2   x                     T1  T2 T3  T4
       / \
     T3  T4
```

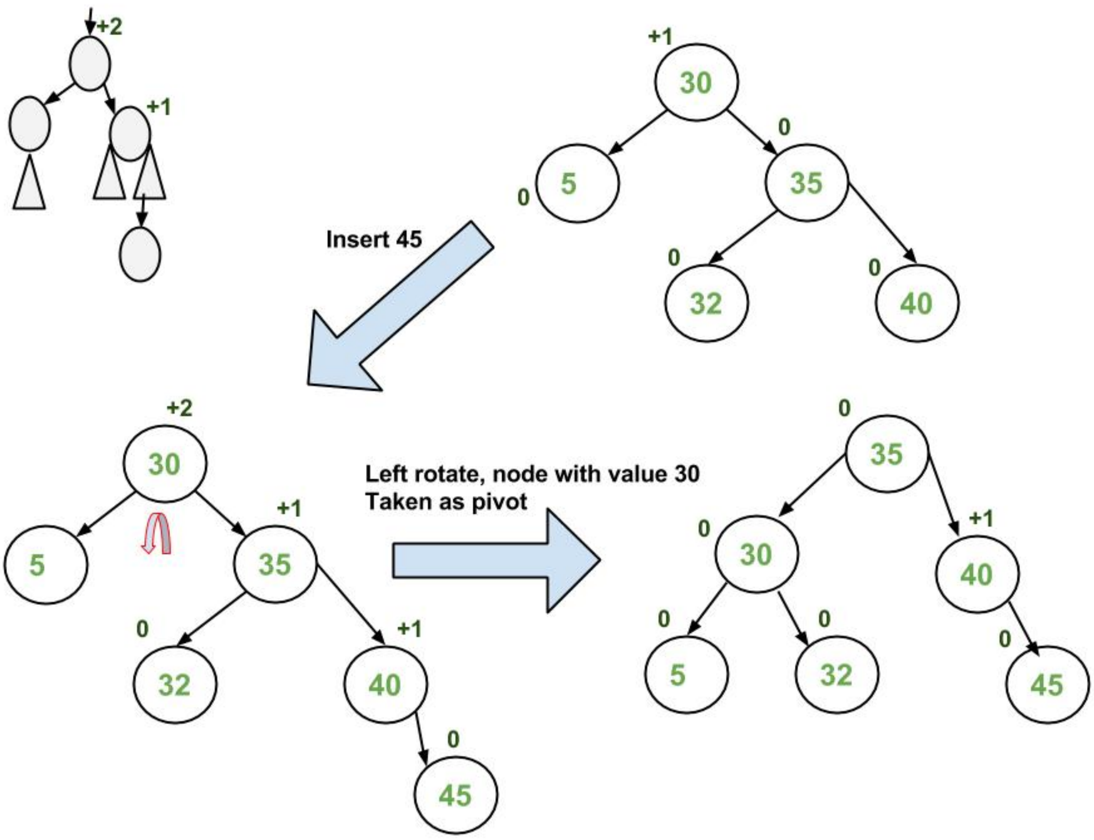
d) Right Left Case

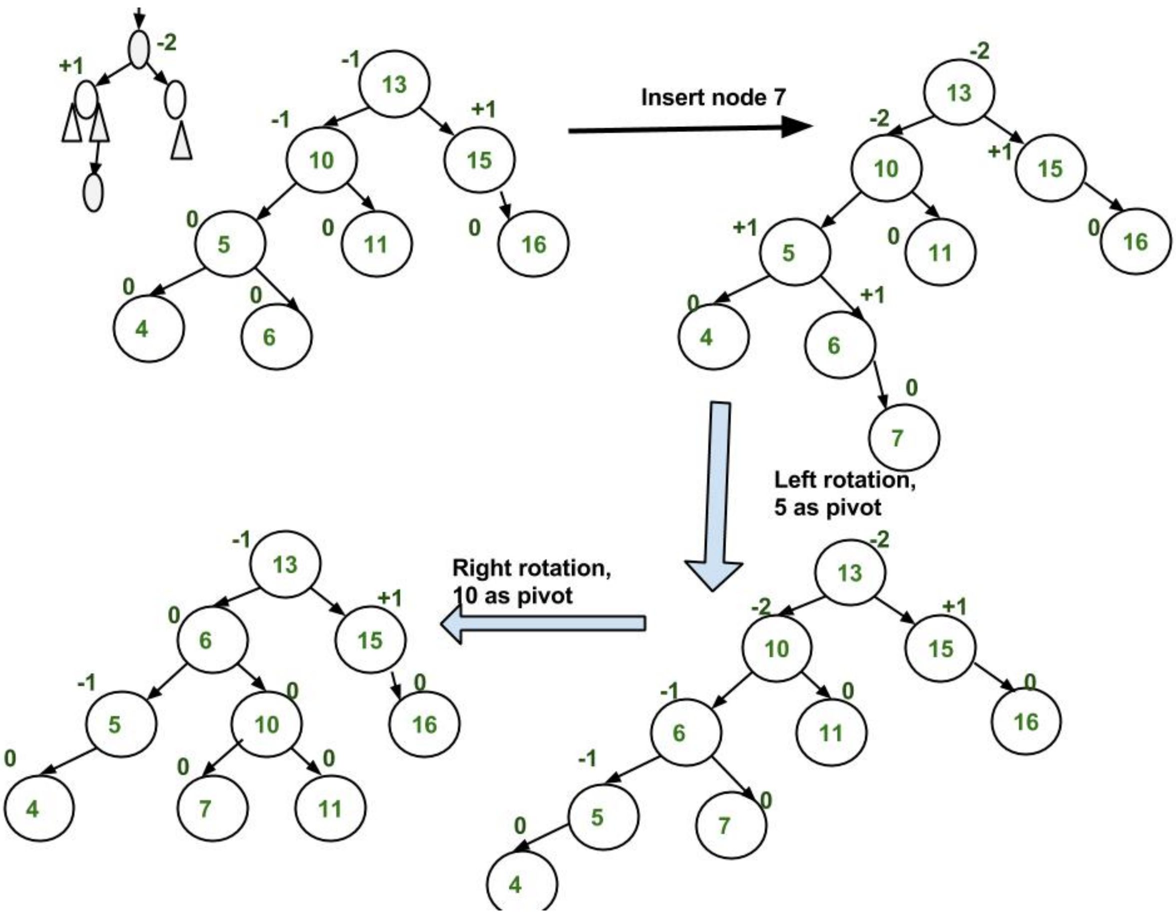
```
   z                            z                            x
  / \                          / \                         /   \
T1   y   Right Rotate (y)    T1   x      Left Rotate(z)   z      y
    / \  - - - - - - - - ->     /   \   - - - - - - - ->  / \    / \
   x   T4                     T2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

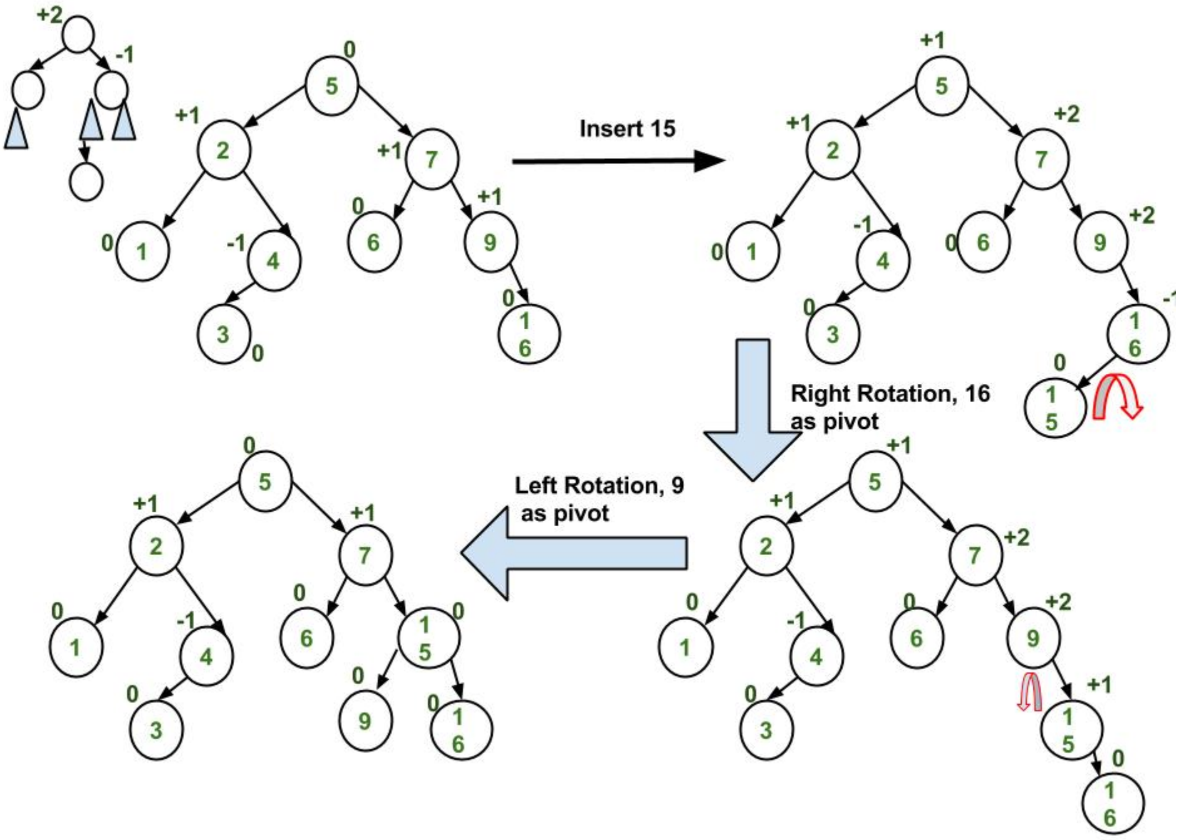
Insertion Examples:




Insert Node with value 3
Rotating Right, node with value
10 as pivot
13
10
15
5
11
16
4
8
3


Insert 45
Left rotate, node with value 30
Taken as pivot
30
5
35
32
40
45


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



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 a 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.

4.2.2 Deletion

To make sure that the given tree remains AVL after every deletion, we must augment the standard BST delete 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.
```

Let w be the node to be deleted

1) Perform standard BST delete for w.

2) Starting from w, travel up and find the first unbalanced node. Let z be the first unbalanced node, y be the larger height child of z, and x be the larger height child of y. Note that the definitions of x and y are different from insertion here.

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)

Like insertion, following are the operations to be performed in above mentioned 4 cases. Note that, unlike insertion, fixing the node z won't fix the complete AVL tree. After fixing z, we may have to fix ancestors of z as well

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

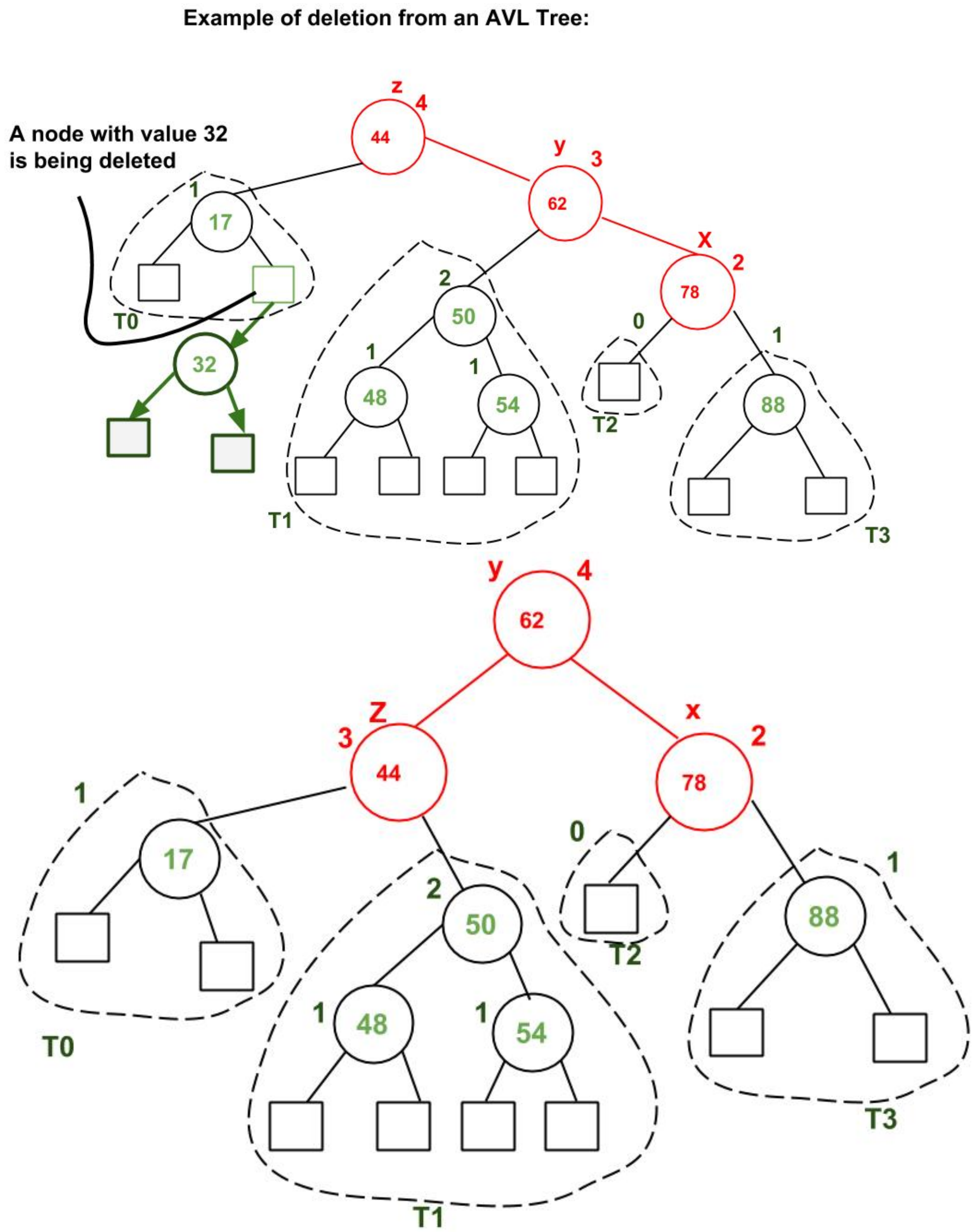
```
  z                                y
 /  \                            /   \
T1   y     Left Rotate(z)       z      x
    /  \   - - - - - - - ->    / \    / \
   T2   x                     T1  T2 T3  T4
       / \
     T3  T4
```

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
```

Unlike insertion, in deletion, after we perform a rotation at z, we may have to perform a rotation at ancestors of z. Thus, we must continue to trace the path until we reach the root.

Example:



A node with value 32 is being deleted. After deleting 32, we travel up and find the first unbalanced node which is 44. We mark it as z, its higher height child as y which is 62, and y's higher height child as x which could be either 78 or 50 as both are of same height. We have considered 78. Now the case is Right Right, so we perform left rotation.

Implementation

Following is the C implementation for AVL Tree Deletion. The following C implementation uses the recursive BST delete as basis. In the recursive BST delete, after deletion, 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 deleted node.

1) Perform the normal BST deletion.

2) The current node must be one of the ancestors of the deleted 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 Left Right case, get the balance factor of left subtree. If balance factor of the left subtree is greater than or equal to 0, then it is Left Left case, else Left Right case.

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 Right Left case, get the balance factor of right subtree. If the balance factor of the right subtree is smaller than or equal to 0, then it is Right Right case, else Right Left case.

```java
// Java program for deletion in AVL Tree

class Node
{
    int key, height;
    Node left, right;

    Node(int d)
    {
        key = d;
        height = 1;
    }
}
```

```java
class AVLTree
{
    Node root;

    // A utility function to get height of the tree
    int height(Node N)
    {
        if (N == null)
            return 0;
        return N.height;
    }

    // A utility function to get maximum of two integers
    int max(int a, int b)
    {
        return (a > b) ? a : b;
    }

    // A utility function to right rotate subtree rooted with y
    // See the diagram given above.
    Node rightRotate(Node y)
    {
        Node x = y.left;
        Node T2 = x.right;

        // Perform rotation
        x.right = y;
        y.left = T2;

        // Update heights
        y.height = max(height(y.left), height(y.right)) + 1;
        x.height = max(height(x.left), height(x.right)) + 1;

        // Return new root
        return x;
    }
```

```
Node leftRotate(Node x)
{
    Node y = x.right;
    Node T2 = y.left;

    // Perform rotation
    y.left = x;
    x.right = T2;

    // Update heights
    x.height = max(height(x.left), height(x.right)) + 1;
    y.height = max(height(y.left), height(y.right)) + 1;

    // Return new root
    return y;
}

// Get Balance factor of node N
int getBalance(Node N)
{
    if (N == null)
        return 0;
    return height(N.left) - height(N.right);
}

Node insert(Node node, int key)
{
    /* 1. Perform the normal BST rotation */
    if (node == null)
        return (new Node(key));

    if (key < node.key)
        node.left = insert(node.left, key);
    else if (key > node.key)
        node.right = insert(node.right, key);
    else // Equal keys not allowed
        return node;

    /* 2. Update height of this ancestor node */
    node.height = 1 + max(height(node.left),
                        height(node.right));

    /* 3. Get the balance factor of this ancestor
    node to check whether this node became
    Wunbalanced */
    int balance = getBalance(node);

    // If this node becomes unbalanced, then
    // there are 4 cases Left Left Case
    if (balance > 1 && key < node.left.key)
        return rightRotate(node);

    // Right Right Case
    if (balance < -1 && key > node.right.key)
        return leftRotate(node);

    // Left Right Case
    if (balance > 1 && key > node.left.key)
    {
        node.left = leftRotate(node.left);
        return rightRotate(node);
    }

    // Right Left Case
    if (balance < -1 && key < node.right.key)
    {
        node.right = rightRotate(node.right);
        return leftRotate(node);
    }

    /* return the (unchanged) node pointer */
    return node;
}
```

```java
    /* Given a non-empty binary search tree, return the
    node with minimum key value found in that tree.
    Note that the entire tree does not need to be
    searched. */
    Node minValueNode(Node node)
    {
        Node current = node;

        /* loop down to find the leftmost leaf */
        while (current.left != null)
        current = current.left;

        return current;
    }

    Node deleteNode(Node root, int key)
    {
        // STEP 1: PERFORM STANDARD BST DELETE
        if (root == null)
            return root;

        // If the key to be deleted is smaller than
        // the root's key, then it lies in left subtree
        if (key < root.key)
            root.left = deleteNode(root.left, key);

        // If the key to be deleted is greater than the
        // root's key, then it lies in right subtree
        else if (key > root.key)
            root.right = deleteNode(root.right, key);

        // if key is same as root's key, then this is the node
        // to be deleted
        else
        {

            // node with only one child or no child
            if ((root.left == null) || (root.right == null))
            {
                Node temp = null;
                if (temp == root.left)
                    temp = root.right;
                else
                    temp = root.left;

                // No child case
                if (temp == null)
                {
                    temp = root;
                    root = null;
                }
                else // One child case
                    root = temp; // Copy the contents of
                                // the non-empty child
            }
            else
            {

                // node with two children: Get the inorder
                // successor (smallest in the right subtree)
                Node temp = minValueNode(root.right);

                // Copy the inorder successor's data to this node
                root.key = temp.key;

                // Delete the inorder successor
                root.right = deleteNode(root.right, temp.key);
            }
        }
```

```java
        // If the tree had only one node then return
        if (root == null)
            return root;

        // STEP 2: UPDATE HEIGHT OF THE CURRENT NODE
        root.height = max(height(root.left), height(root.right)) + 1;

        // STEP 3: GET THE BALANCE FACTOR OF THIS NODE (to check whether
        // this node became unbalanced)
        int balance = getBalance(root);

        // If this node becomes unbalanced, then there are 4 cases
        // Left Left Case
        if (balance > 1 && getBalance(root.left) >= 0)
            return rightRotate(root);

        // Left Right Case
        if (balance > 1 && getBalance(root.left) < 0)
        {
            root.left = leftRotate(root.left);
            return rightRotate(root);
        }

        // Right Right Case
        if (balance < -1 && getBalance(root.right) <= 0)
            return leftRotate(root);

        // Right Left Case
        if (balance < -1 && getBalance(root.right) > 0)
        {
            root.right = rightRotate(root.right);
            return leftRotate(root);
        }

        return root;
    }
```

```
    // A utility function to print preorder traversal of
    // the tree. The function also prints height of every
    // node
    void preOrder(Node node)
    {
        if (node != null)
        {
            System.out.print(node.key + " ");
            preOrder(node.left);
            preOrder(node.right);
        }
    }

    public static void main(String[] args)
    {
        AVLTree tree = new AVLTree();

        /* Constructing tree given in the above figure */
        tree.root = tree.insert(tree.root, 9);
        tree.root = tree.insert(tree.root, 5);
        tree.root = tree.insert(tree.root, 10);
        tree.root = tree.insert(tree.root, 0);
        tree.root = tree.insert(tree.root, 6);
        tree.root = tree.insert(tree.root, 11);
        tree.root = tree.insert(tree.root, -1);
        tree.root = tree.insert(tree.root, 1);
        tree.root = tree.insert(tree.root, 2);

        /* The constructed AVL Tree would be
        9
        / \
        1 10
        / \ \
        0 5 11
        / / \
        -1 2 6
        */
        System.out.println("Preorder traversal of "+
                           "constructed tree is : ");
        tree.preOrder(tree.root);

        tree.root = tree.deleteNode(tree.root, 10);

        /* The AVL Tree after deletion of 10
        1
        / \
        0 9
        /   / \
        -1 5 11
        / \
        2 6
        */
        System.out.println("");
        System.out.println("Preorder traversal after "+
                       "deletion of 10 :");
        tree.preOrder(tree.root);
    }
}

// This code has been contributed by Mayank Jaiswal
```

4.3 Assignment

1. Implement AVL from this chapter with data{0, 1, 2, 3, 4, 5, 6, 7, 8, 9} in sequence. Draw the created AVL Tree!
2. From Binary Search Tree implementation from chapter 3, create one method to check wheter the Binary Search Tree is AVL Tree or not.