AVL Tree | Set 2 (Deletion)

We have discussed AVL insertion in the previous post. In this post, we will follow a similar approach for deletion.

Steps to follow for 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:

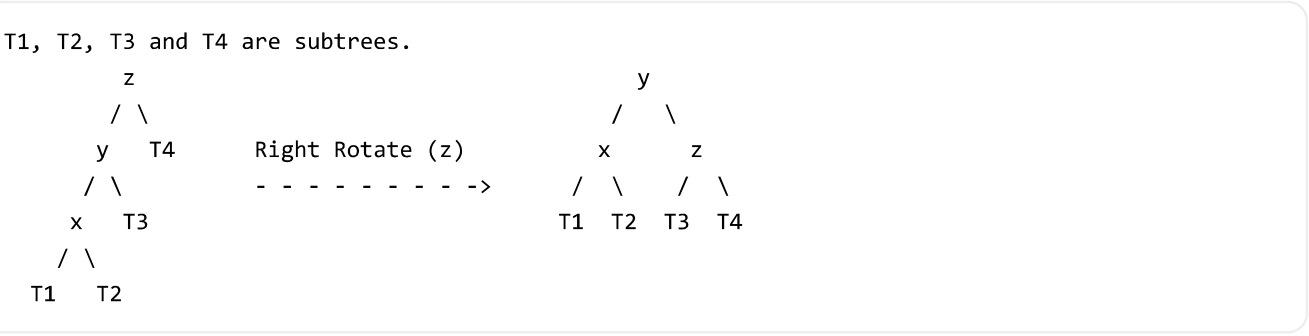
a) y is left child of z and x is left child of y (Left Left Case)

b) y is left child of z and x is right child of y (Left Right Case)

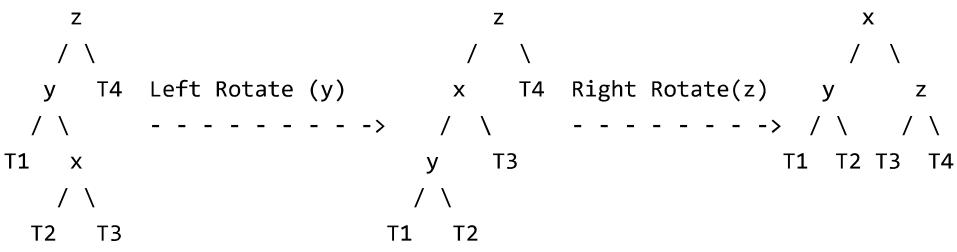
c) y is right child of z and x is right child of y (Right Right Case)
d) y is right child of z and x is left child of y (Right Left Case)

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 (See this video lecture for proof)

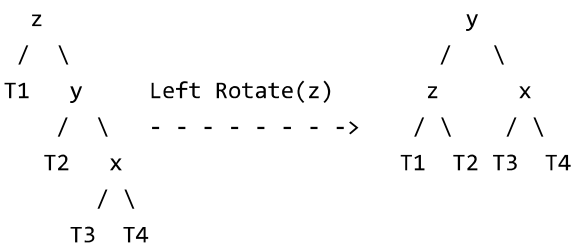
a) Left Left Case



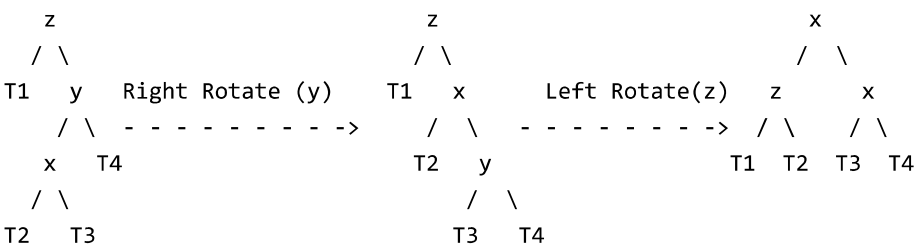
b) Left Right Case



c) Right Right Case

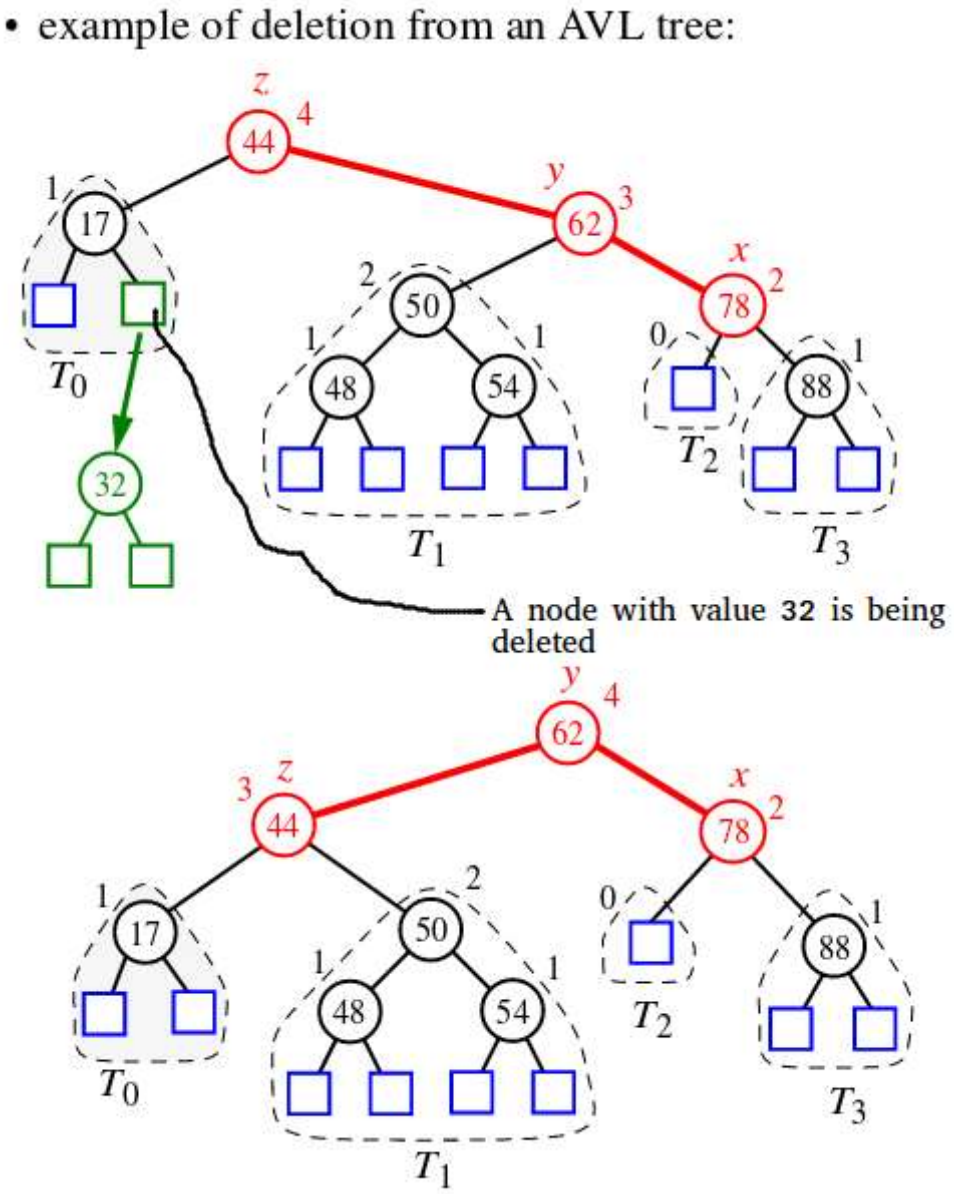


d) Right Left Case



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 unbalanaced node which is 44. We mark it as z, its higher height child as y which is 52, 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.

Image source: https://www.cs.purdue.edu/homes/ayg/CS251/slides/chap7b.pdf

Recommended: Please solve it on "*PRACTICE*" first, before moving on to solution.

C 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.

C

```c
// C program to delete a node from AVL Tree
#include<stdio.h>
#include<stdlib.h>

// An AVL tree node
struct Node
{
    int key;
    struct Node *left;
    struct Node *right;
    int height;
};

// A utility function to get maximum of two integers
int max(int a, int b);

// A utility function to get height of the tree
int height(struct 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;
}

/* Helper function that allocates a new node with the given key and
    NULL left and right pointers. */
struct Node* newNode(int key)
{
```

```
    struct Node* node = (struct Node*)
                        malloc(sizeof(struct Node));
    node->key   = key;
    node->left   = NULL;
    node->right  = NULL;
    node->height = 1;  // new node is initially added at leaf
    return(node);
}

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

// A utility function to left rotate subtree rooted with x
// See the diagram given above.
struct Node *leftRotate(struct Node *x)
{
    struct Node *y = x->right;
    struct 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(struct Node *N)
{
    if (N == NULL)
        return 0;
    return height(N->left) - height(N->right);
}

struct Node* insert(struct Node* node, int key)
{
    /* 1.  Perform the normal BST rotation */
    if (node == NULL)
        return(newNode(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
          unbalanced */
    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;
}

/* 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. */
struct Node * minValueNode(struct Node* node)
{
    struct Node* current = node;

    /* loop down to find the leftmost leaf */
    while (current->left != NULL)
        current = current->left;

    return current;
}

// Recursive function to delete a node with given key
// from subtree with given root. It returns root of
// the modified subtree.
struct Node* deleteNode(struct 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) )
        {
            struct Node *temp = root->left ? root->left :
                                             root->right;

            // No child case
            if (temp == NULL)
            {
                temp = root;
                root = NULL;
            }
            else // One child case
             *root = *temp; // Copy the contents of
                            // the non-empty child
            free(temp);
        }
        else
        {
            // node with two children: Get the inorder
            // successor (smallest in the right subtree)
            struct 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);
        }
    }

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

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

    // 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(struct Node *root)
{
    if(root != NULL)
    {
        printf("%d ", root->key);
        preOrder(root->left);
        preOrder(root->right);
    }
}

/* Drier program to test above function*/
int main()
{
  struct Node *root = NULL;

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

    /* The constructed AVL Tree would be
            9
           /  \
          1    10
        /  \     \
       0    5     11
      /    /  \
     -1   2    6
    */

    printf("Preorder traversal of the constructed AVL "
           "tree is \n");
    preOrder(root);

    root = deleteNode(root, 10);

    /* The AVL Tree after deletion of 10
            1
           /  \
          0    9
        /     /  \
       -1    5     11
           /  \
          2    6
    */

    printf("\nPreorder traversal after deletion of 10 \n");
    preOrder(root);

    return 0;
}
```

Java

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

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

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

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;
    }

    // A utility function to left rotate subtree rooted with x
    // See the diagram given above.
    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;
    }

    /* 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);
            }
        }

        // 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
```

Run on IDE

Python3

```
# Python code to delete a node in AVL tree
# Generic tree node class
class TreeNode(object):
    def __init__(self, val):
        self.val = val
        self.left = None
        self.right = None
        self.height = 1

# AVL tree class which supports insertion,
# deletion operations
class AVL_Tree(object):

    def insert(self, root, key):

        # Step 1 - Perform normal BST
        if not root:
            return TreeNode(key)
        elif key < root.val:
            root.left = self.insert(root.left, key)
        else:
            root.right = self.insert(root.right, key)

        # Step 2 - Update the height of the
        # ancestor node
        root.height = 1 + max(self.getHeight(root.left),
                              self.getHeight(root.right))

        # Step 3 - Get the balance factor
        balance = self.getBalance(root)

        # Step 4 - If the node is unbalanced,
        # then try out the 4 cases
        # Case 1 - Left Left
        if balance > 1 and key < root.left.val:
            return self.rightRotate(root)

        # Case 2 - Right Right
        if balance < -1 and key > root.right.val:
            return self.leftRotate(root)

        # Case 3 - Left Right
        if balance > 1 and key > root.left.val:
```

```
            root.left = self.leftRotate(root.left)
            return self.rightRotate(root)

        # Case 4 - Right Left
        if balance < -1 and key < root.right.val:
            root.right = self.rightRotate(root.right)
            return self.leftRotate(root)

        return root

    # Recursive function to delete a node with
    # given key from subtree with given root.
    # It returns root of the modified subtree.
    def delete(self, root, key):

        # Step 1 - Perform standard BST delete
        if not root:
            return root

        elif key < root.val:
            root.left = self.delete(root.left, key)

        elif key > root.val:
            root.right = self.delete(root.right, key)

        else:
            if root.left is None:
                temp = root.right
                root = None
                return temp

            elif root.right is None:
                temp = root.left
                root = None
                return temp

            temp = self.getMinValueNode(root.right)
            root.val = temp.val
            root.right = self.delete(root.right,
                                     temp.val)

        # If the tree has only one node,
        # simply return it
        if root is None:
            return root

        # Step 2 - Update the height of the
        # ancestor node
        root.height = 1 + max(self.getHeight(root.left),
                              self.getHeight(root.right))

        # Step 3 - Get the balance factor
        balance = self.getBalance(root)

        # Step 4 - If the node is unbalanced,
        # then try out the 4 cases
        # Case 1 - Left Left
        if balance > 1 and self.getBalance(root.left) >= 0:
            return self.rightRotate(root)

        # Case 2 - Right Right
        if balance < -1 and self.getBalance(root.right) <= 0:
            return self.leftRotate(root)

        # Case 3 - Left Right
        if balance > 1 and self.getBalance(root.left) < 0:
            root.left = self.leftRotate(root.left)
            return self.rightRotate(root)

        # Case 4 - Right Left
```

```
        if balance < -1 and self.getBalance(root.right) > 0:
            root.right = self.rightRotate(root.right)
            return self.leftRotate(root)

        return root

    def leftRotate(self, z):

        y = z.right
        T2 = y.left

        # Perform rotation
        y.left = z
        z.right = T2

        # Update heights
        z.height = 1 + max(self.getHeight(z.left),
                           self.getHeight(z.right))
        y.height = 1 + max(self.getHeight(y.left),
                           self.getHeight(y.right))

        # Return the new root
        return y

    def rightRotate(self, z):

        y = z.left
        T3 = y.right

        # Perform rotation
        y.right = z
        z.left = T3

        # Update heights
        z.height = 1 + max(self.getHeight(z.left),
                           self.getHeight(z.right))
        y.height = 1 + max(self.getHeight(y.left),
                           self.getHeight(y.right))

        # Return the new root
        return y

    def getHeight(self, root):
        if not root:
            return 0

        return root.height

    def getBalance(self, root):
        if not root:
            return 0

        return self.getHeight(root.left) - self.getHeight(root.right)

    def getMinValueNode(self, root):
        if root is None or root.left is None:
            return root

        return self.getMinValueNode(root.left)

    def preOrder(self, root):

        if not root:
            return

        print("{0} ".format(root.val), end="")
        self.preOrder(root.left)
        self.preOrder(root.right)
```

```
myTree = AVL_Tree()
root = None
nums = [9, 5, 10, 0, 6, 11, -1, 1, 2]

for num in nums:
    root = myTree.insert(root, num)

# Preorder Traversal
print("Preorder Traversal after insertion -")
myTree.preOrder(root)
print()

# Delete
key = 10
root = myTree.delete(root, key)

# Preorder Traversal
print("Preorder Traversal after deletion -")
myTree.preOrder(root)
print()

# This code is contributed by Ajitesh Pathak
```

Run on IDE

Output:

```
Preorder traversal of the constructed AVL tree is
9 1 0 -1 5 2 6 10 11
Preorder traversal after deletion of 10
1 0 -1 9 5 2 6 11
```

Time Complexity: The rotation operations (left and right rotate) take constant time as only few pointers are being changed there. Updating the height and getting the balance factor also take constant time. So the time complexity of AVL delete remains same as BST delete which is O(h) where h is height of the tree. Since AVL tree is balanced, the height is O(Logn). So time complexity of AVL delete is O(Log n).