Revanth.NM
1BM18CS081

AVL Tree

```
Node * newNode (int Key)
{
    Node * node = new Node();
    node -> Key = Key;
    node -> left = NULL;
    node -> right = NULL;
    node -> height = 1;
    return (node);
}
Node *rightRotate (Node * node)
{
    Node * left = node-> left;
    Node * right = left -> right;

    left-> right = node;
    node -> left = right;
    node -> height= max(height(node->left), height(node -> right))+1;
    left -> height = max( height (left -> left), height(left -> right))+1;
    return left;
}
Node * leftRotate (Node * node)
{
    Node * right = node-> right;
    Node * left = ~~node~~ right -> left;

    right -> left = node;
    node -> right = left;
    node -> height = max (height (node -> left), height (node -> right)) +1;
    right-> height = max( height(right -> left), height ( right -> right)) +1;

    return right;
}
```

```
Node * insert (Node * node, int key)
{
        if (node == NULL)
            return (newNode(key));

        if (key < node -> key)                                      Creating a
              node -> left = insert(node -> left, key);             new node at
        else if (key > node -> key)                                 position.
              node -> right = insert (node -> right, key);

        else
              return node;

        node -> height = 1+ max (height(node -> left), height(node -> right));

        int balance = getBalance (node);     } getting Balance at each
                                                         node
        If (balance > 1 && key < node -> left -> key)
      return rightRotate (node);
        If ( balance < -1 && key > node -> right -> key)              Rotating the
            return leftRotate (node);                                tree if there is any
         If (Balance > 1  &&  key > node -> left -> key)             inbalance in tree.
              node -> left = leftRotate (node -> left);
            return  rightRotate (node);

        If (Balance < -1  &&  key < node -> right -> key)
            node -> right = rightRotate (node -> right);
            return  leftRotate (node);

        return node;

}
```

```
Node * deleteNode (Node * root, int Key)
{
        If (root == NULL)
            return root;
        If (root -> Key > Key)
            root -> left = deleteNode (Node -> left, Key);
        else If ( root -> Key < Key)
            root -> right = deleteNode( root -> right, Key);
```

Finding the correct node that is to be deleted.

```
        else
        {
```

← Once found below steps will be executed.

```
            If((root -> left == NULL) || (root -> right== NULL))
            {
                Node * temp = root -> left ? root -> left : root -> right;

                If (temp == NULL)
                {
                    temp = root;
                    root = NULL;
                }
                else
                    *root = * temp;

                ~~temp;~~ free(temp);
            }
```

deleting a node with 1 or 0 childs

```
            else
            {
                Node * temp = minValueNode (root -> right);
```

with more than 1 child

```
                root -> key = temp -> Key;
                root -> right = deleteNode (root -> right, temp -> Key);
            }
        }
        If (root==NULL)
            return root;
```

resetting the Tree values and Balancing it by Rotating appropriatly.

```
        root -> height = 1+max(height(root -> left), height(root -> right));
```

Balance the tree same as in insert function if its unbalanced (By Rotating)

```
        return root;
}
```