Insertion in AVL tree

1. Only 3 nodes, with root node being unbalanced can be balanced at a time.
2. In case of two different nodes being unbalanced, balance the successor node first, the ancestor or parent node will get automatically balanced
3. Insert node normally as in BST, try the Balance factor of each node at insertion of a node, and if unbalance, according to the cause of unbalance (LL, RR, RL, LR), balance the tree

```
class Node
{
    int key;
    Node* leftchild;
    Node* rightchild;
    int height;
};

int height( Node* N)
{
    if (N == NULL)
        return 0;
    return N -> height;
```

```
}

Node* newNode(int key)
{
    Node* node = new Node();
    node->key = key;
    node->leftchild = NULL;
    node->rightchild = NULL;
    node->height = 1;
    return (node);
}

// Function to right rotate
// Function to left rotate

Node* insert(Node* node, int key)
{
    if (node == NULL)                    // Empty tree
        return(newNode(key));
    if (key < node->key)
        node->leftchild = insert(node->leftchild, key);
    else if (key > node->key)
        node->rightchild = insert(node->rightchild, key);
    else
        return node; // Equal keys are not allowed
    // update height of ancestor
    node->height = 1 + max(height(node->leftchild),
                           height(node->rightchild));
    // Get balance factor of this node
    int balance = getBalance(node);
```

```
// Check for imbalance in the node
   for following 4 possibilities

//RR  if (balance > 1 && key < node->leftchild->key)
        // Right rotate(node);

//LL  if (balance < -1 && key > node->right->key)
        // Leftrotate(node);

//LR  if (balance > 1 && key > node->left->key)
        // Left Rotate
           then Right Rotate

    if (balance < -1 && key < node->right->key)
        // Right Rotate
           then left Rotate

    return node;
}

// Delete Function
Node* del(Node* root, int key)
{
    if (root == NULL)
        return root;

    if (key < root->key)
        root->leftchild = del(root->leftchild, key);
```

```
else if (key > root->key)
    root->right = del(root->right, key);

// If key is same as root's key,
    this node is to be deleted

else
{
    // node with only one child or no child
    if (root->left == NULL) || (root->right == NULL)
    {
        Node* temp = root->left ?
                     root->left : root->right;
        // No child case
        if (temp == NULL)
        {
            temp = root;
            root = NULL;
        }
        else
        *root = *temp; // Copy contents of
                          non-empty child
        free(temp);
    }
    else
    {
        Node* temp = minValue(root->right);
        root->key = temp->key;
        root->right = del(root->right,
                          temp->key);
    }
```

```
}

// Get the balance factor
int balance = getBalance(root);

// If unbalanced, 4 possibilities arise

//LL
//RR
//LR
//RL

return root;
}
```

Deletion

1. Leaf node
2. One child
3. Two children

After deletion check for balance factor of each node, and if any unbalance, balance the tree.