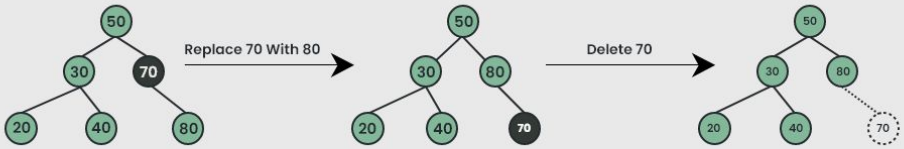
DATA STRUCTURE
WEEK 10 / 11

Contact Details:
Email: sobia.iftikhar@nu.edu.pk
GCR:

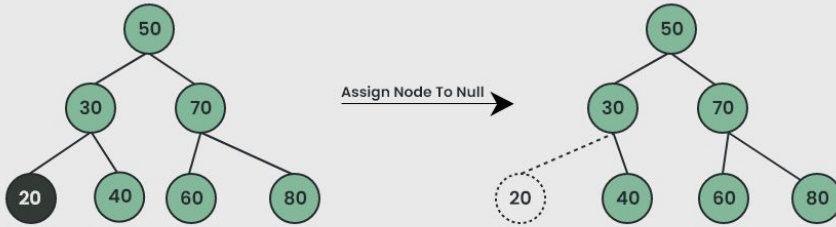
Case 2: Delete A Node With Single Child In BST



Delete Node 70

After Deletion

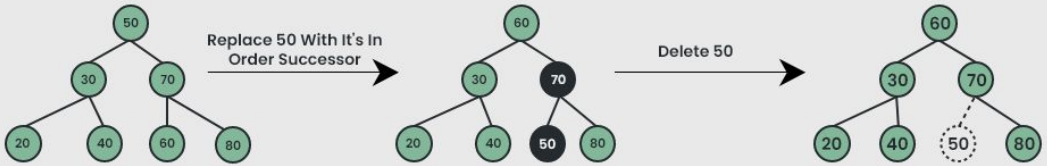
Case 1 : Delete A Leaf Node In BST



Delete Node 20

Deleted Node 20

Case 3 : Delete A Node With Both Children In BST



Delete Node 50

After Deletion

Delete Implementation

```
Node* deleteNode(Node* root, int k)
{
    // Base case
    if (root == NULL)
        return root;
    if (root->key > k) {
        root->left = deleteNode(root->left, k);
        return root;
    }
    else if (root->key < k) {
        root->right = deleteNode(root->right, k);
        return root;
    }
    if (root->left == NULL) {
        Node* temp = root->right;
        delete root;
        return temp;
    }
    else if (root->right == NULL) {
        Node* temp = root->left;
        delete root;
        return temp;
    }
```

```
    else {

        Node* succParent = root;

        // Find successor
        Node* succ = root->right;
        while (succ->left != NULL) {
            succParent = succ;
            succ = succ->left;
        }
        if (succParent != root)
            succParent->left = succ->right;
        else
            succParent->right = succ->right;

        // Copy Successor Data to root
        root->key = succ->key;

        // Delete Successor and return root
        delete succ;
        return root;
    }
}
```

Finding Height of a BST

```
Int height (node * head)
{
        if(head == NULL)
                return NULL;
        else
        {
                int h_left= height(head->left);
                int h_right = height (head ->right);
                if (h_left > h_right)
                        return (h_left + 1);
                else return ( h_right + 1);
        }
}
```

Thinking

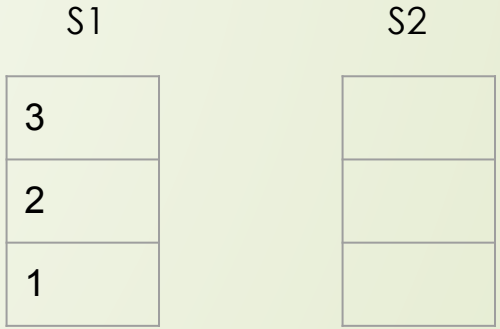
Implementing Queue using stack?

Implementing stacks using queue?

Stacks with Queue

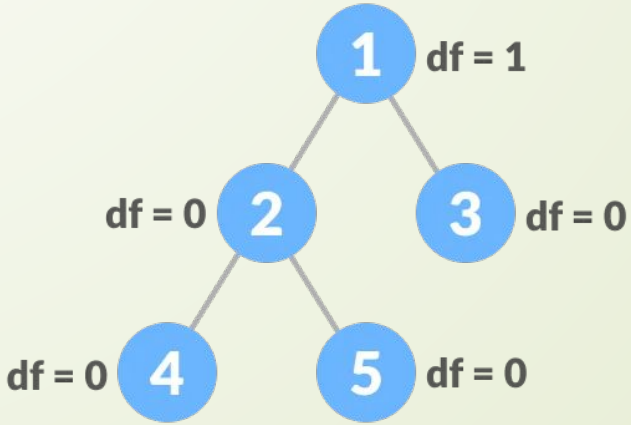
Using two stacks implement QUEUE functionality

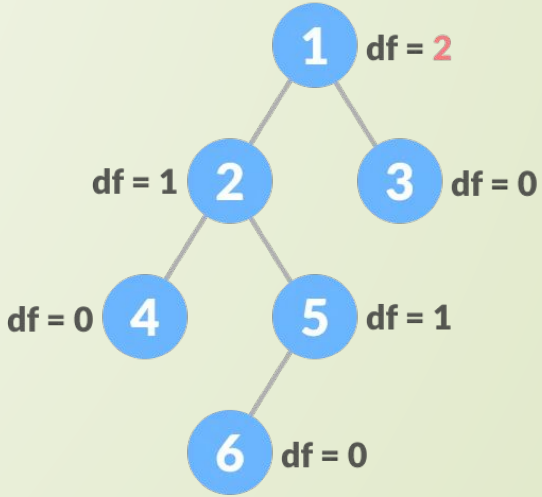
```
Class Queue{
Stack s1;
Stack s2;
Void push(int x){
        s1.push(s);
}
int Qpop(){
while(!s1. empty()){
        S2.push(s1.pop());
}
Int ans= s2.pop();// deleted
while(!s2. empty()){
        S1.push(s2.pop());
}
Return ans;
}
}
```



Balance Binary Tree

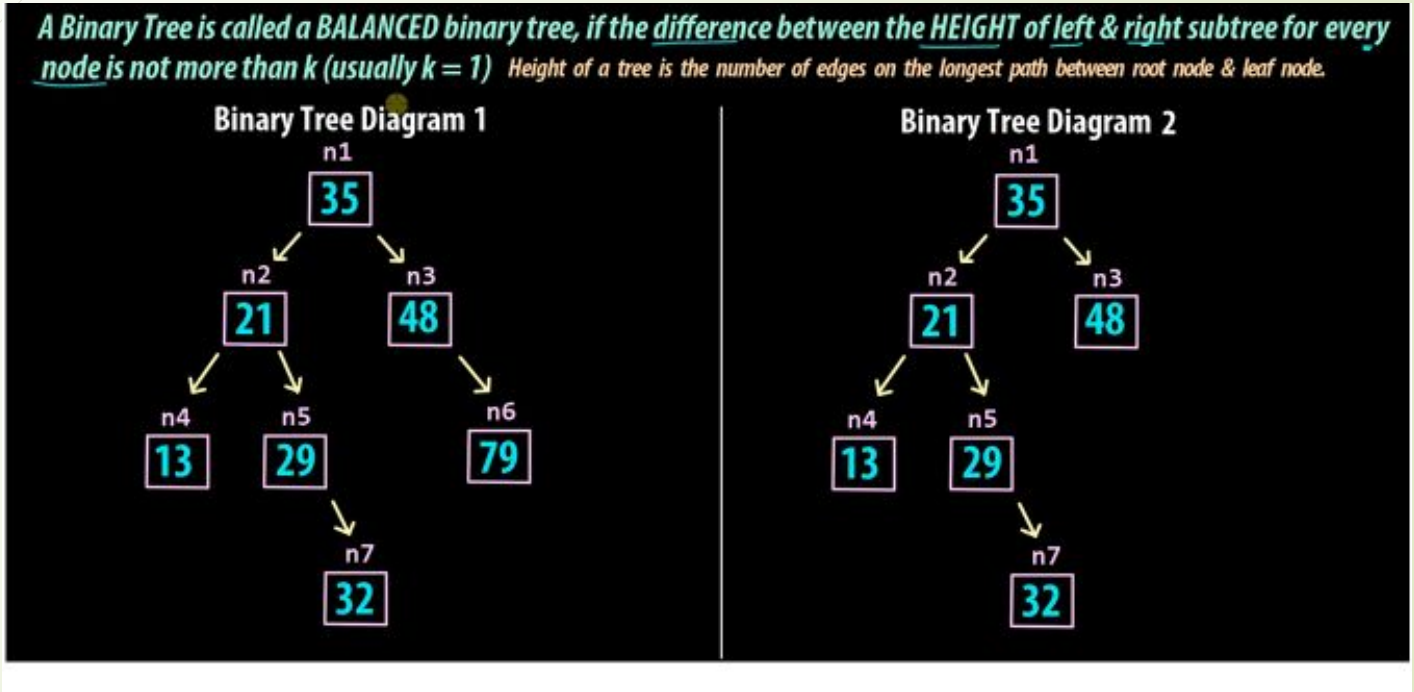
A balanced binary tree, also referred to as a height-balanced binary tree, is defined as a binary tree in which the height of the left and right subtree of any node differ by not more than 1.



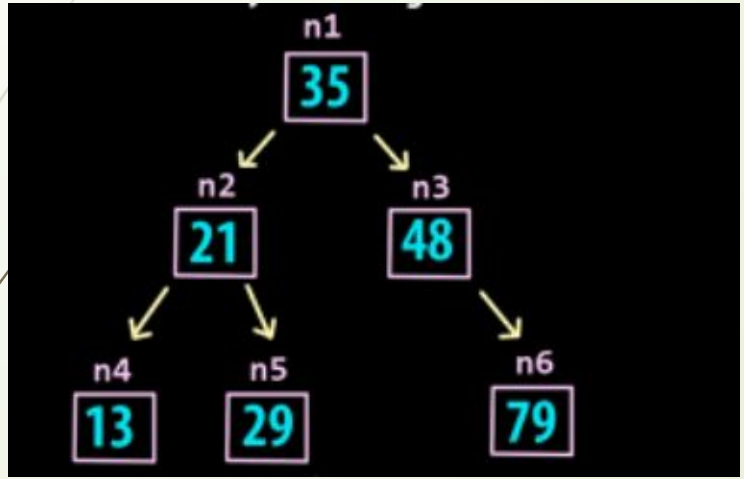


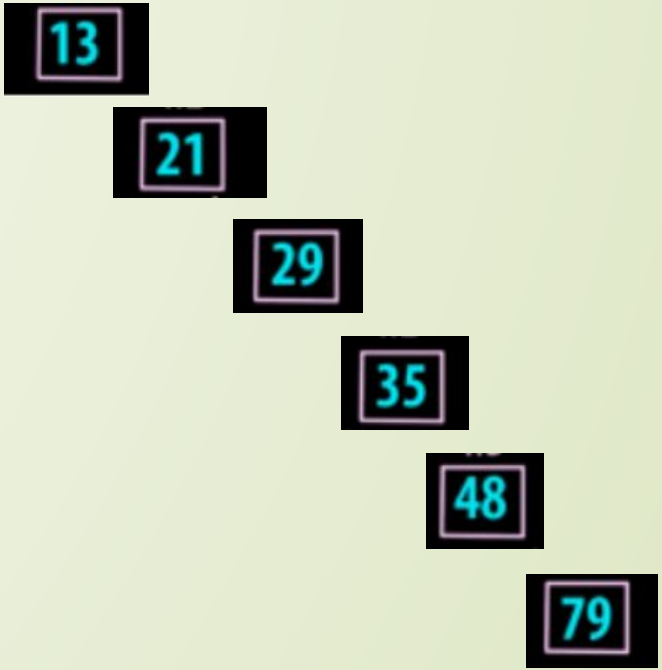
df = |height of left child - height of right child|

Balance Binary Tree



Why we need a balanced binary tree





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.

Avl operation :

1. Insert
1. Delete
2. Search

Right Right Rotation

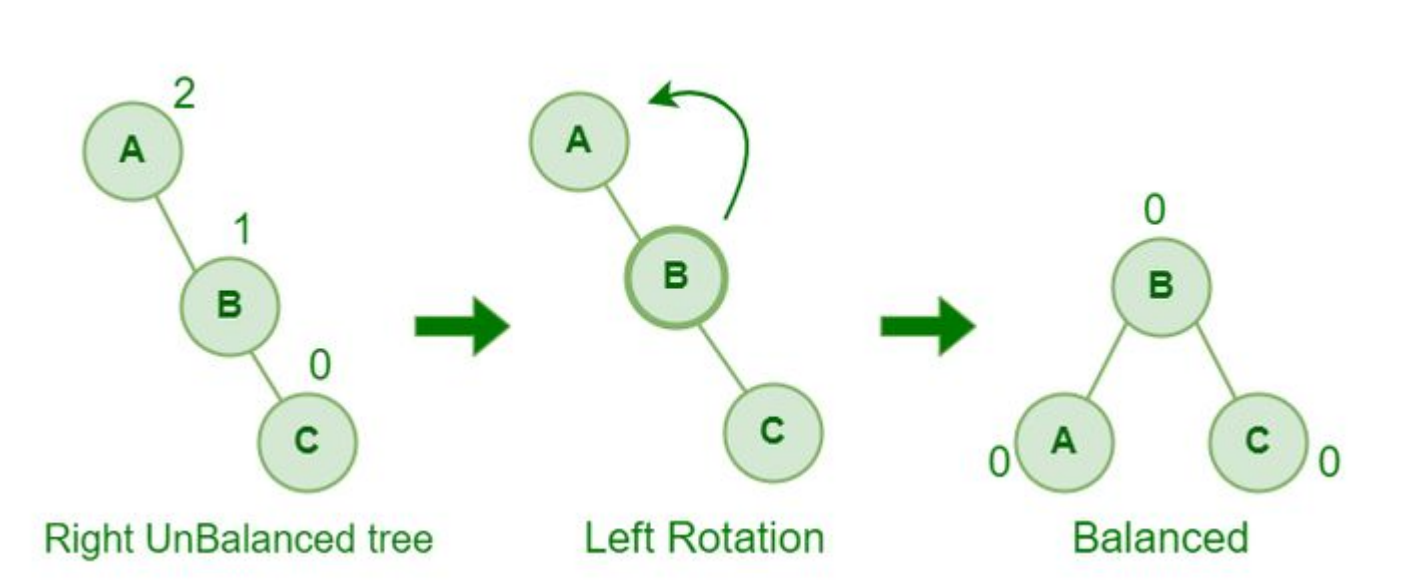
Left Left Rotation

Right Left Rotation

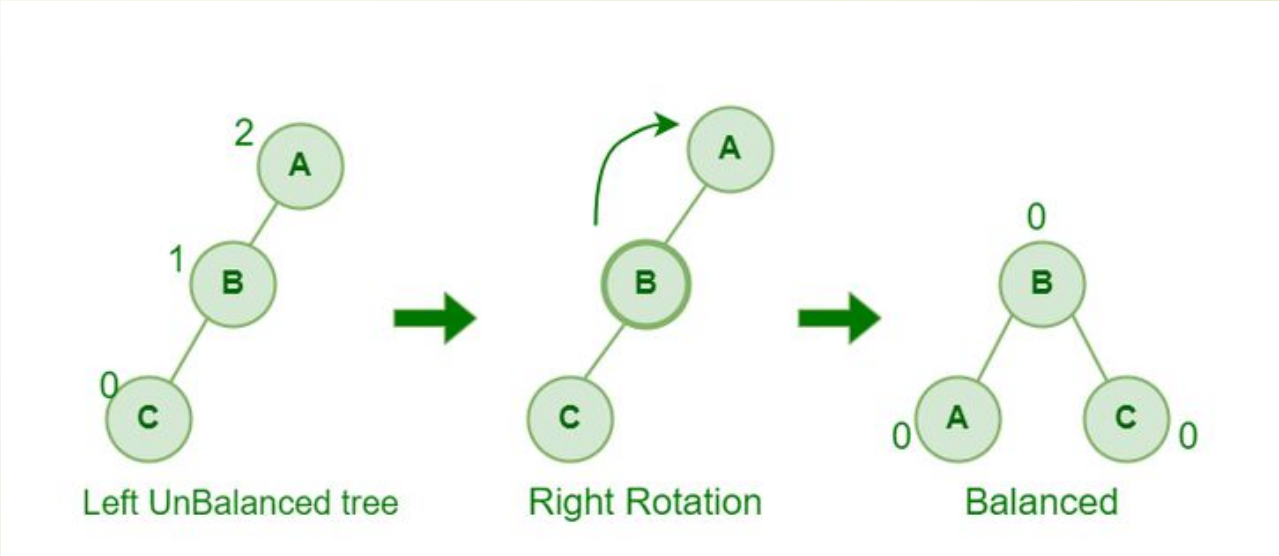
Left Right Rotation

Avl

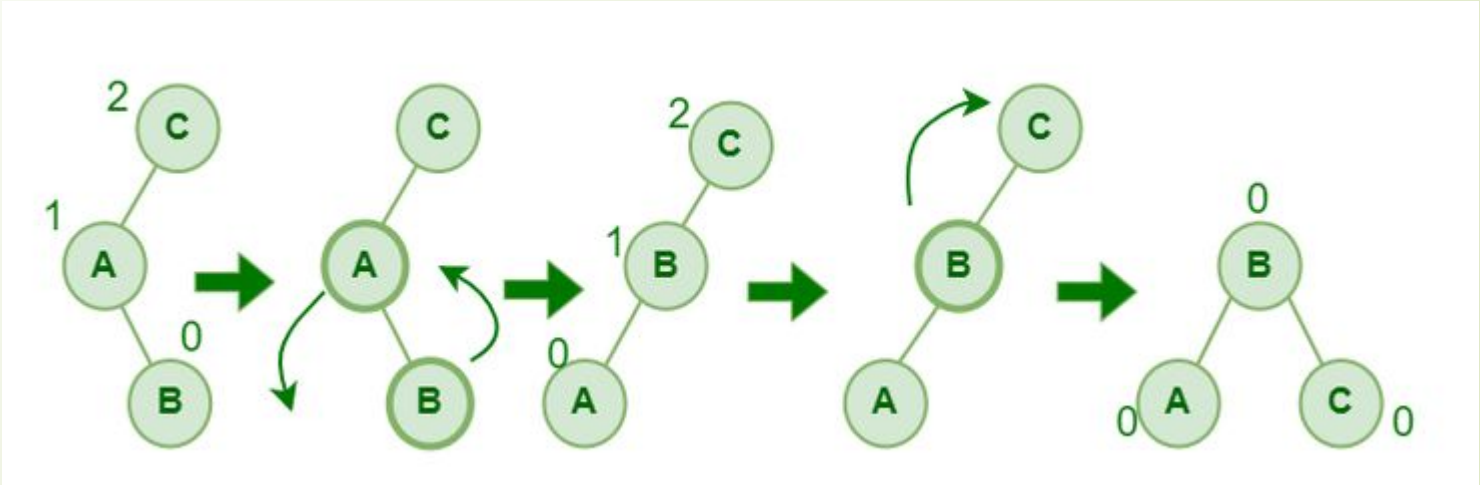
Left Rotation:



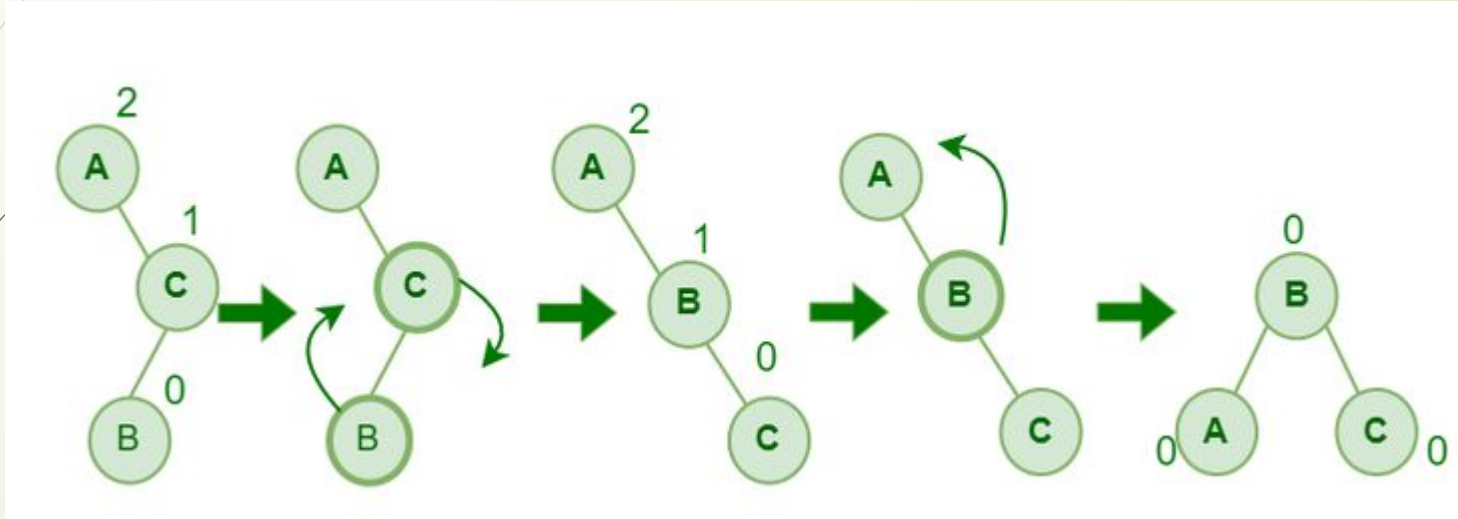
Right Rotation:



Left-Right Rotation

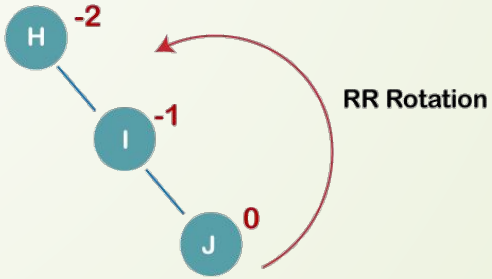


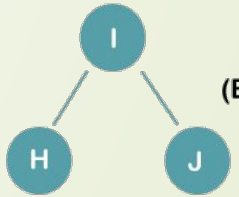
Right-Left Rotation

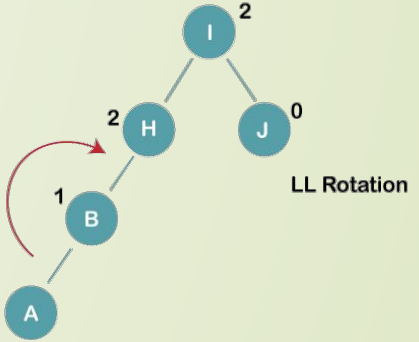


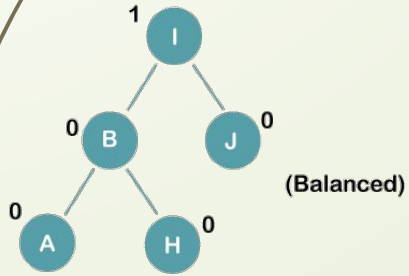
Construct an AVL tree having the following elements

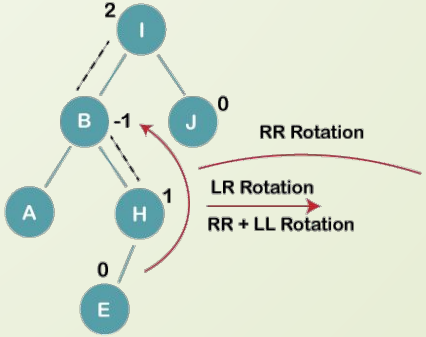
H, I, J, B, A, E, C, F, D, G, K, L

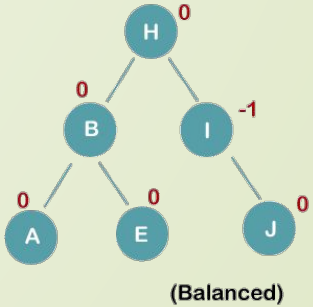






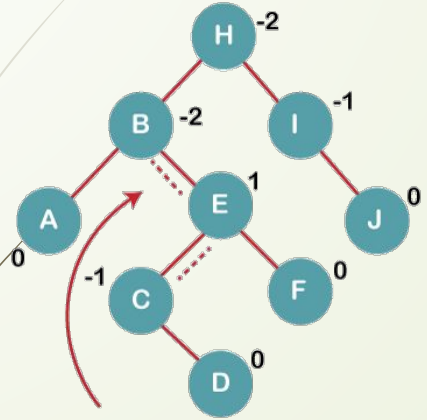


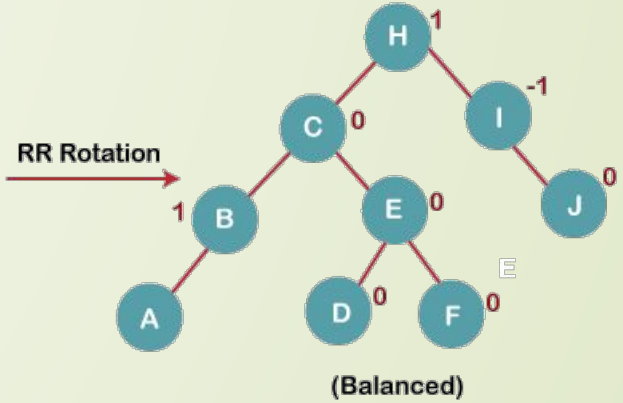




Construct an AVL tree having the following elements

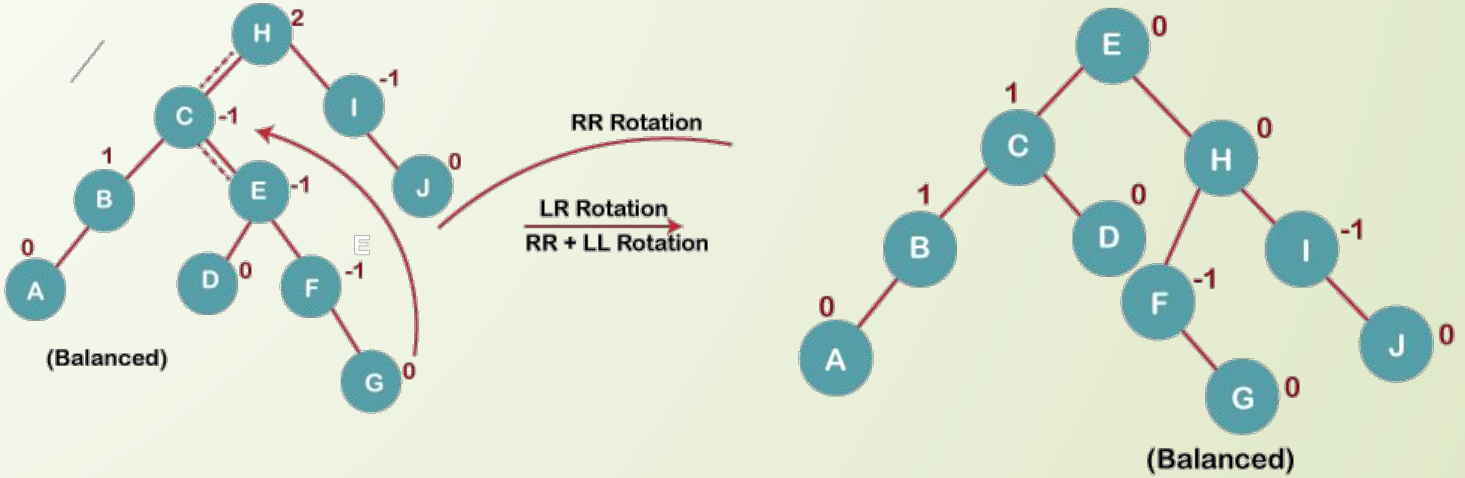
H, I, J, B, A, E, C, F, D, G, K, L





Construct an AVL tree having the following elements

H, I, J, B, A, E, C, F, D, G, K, L



Implementation

```
Node* insert(Node* node, int key)
{
    /* 1. Perform the normal BST insertion */
    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 are not allowed in BST
        return node;

    node->height = 1 + max(height(node->left),
                           height(node->right));
    int balance = getBalance(node);
```

```
    // 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;
}
```

```
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))
    y->height = max(height(y->left),height(y->right))

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

Exercise

Construct Avl

14,17,11,7,53,4,13,12,8,60,19,16,20

Deletion in AVL

For Deletion there are three cases:

1. If it is leaf node then just delete it
2. If node have one child then lin the child with parent of node
3. If it node have two child then there are two case:
    a. Find max node from left subtree
    b. Or find min node from right subtree