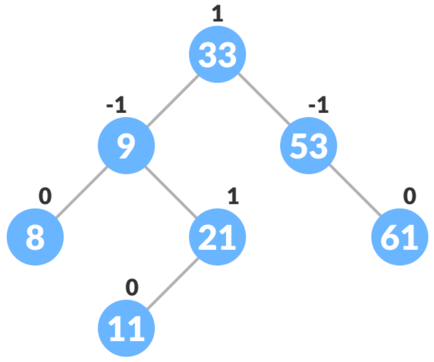
AVL Tree

AVL tree is a self-balancing binary search tree in which each node maintains extra information called a balance factor whose value is either -1, 0 or +1.

Balance Factor

Balance factor of a node in an AVL tree is the difference between the height of the left subtree and that of the right subtree of that node.

Balance Factor = (Height of Left Subtree - Height of Right Subtree) or (Height of Right Subtree - Height of Left Subtree)



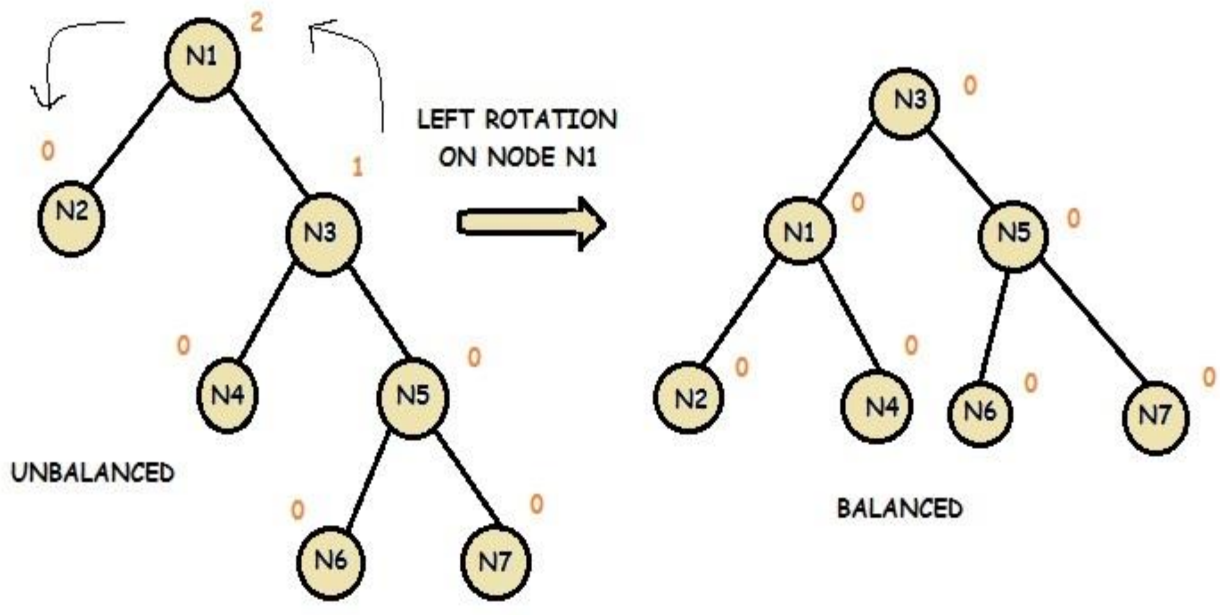
The self-balancing property of an AVL tree is maintained by the balance factor. The value of balance factor should always be -1, 0 or +1.

Operations on an AVL tree

Rotating the subtrees in an AVL Tree

Left Rotate

In left-rotation, the arrangement of the nodes on the right is transformed into the arrangements on the left node.



LEFT ROTATION ON AVL TREE (by OpenGenus)

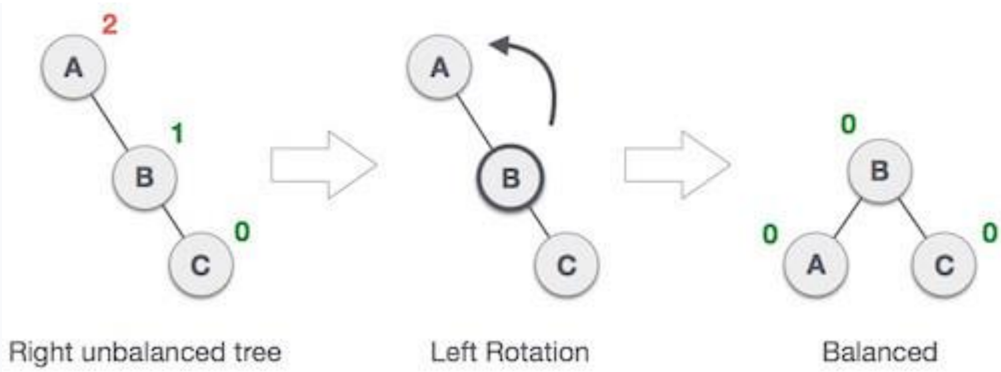
Complexity

Algorithm	Average case	Worst case
Space	o(n)	o(n)
Search	o(log n)	o(log n)
Insert	o(log n)	o(log n)
Delete	o(log n)	o(log n)

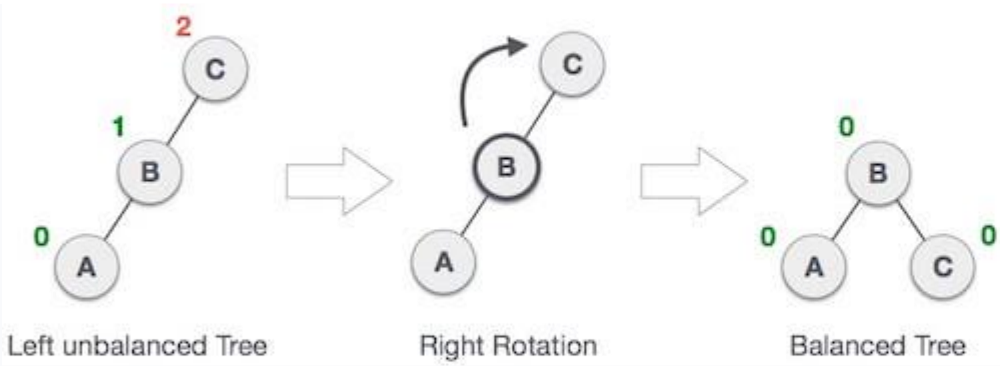
Operations on AVL tree

Operation	Description
Insertion	Insertion in AVL tree is performed in the same way as it is performed in a binary search tree. However, it may lead to violation in the AVL tree property and therefore the tree may need balancing. The tree can be balanced by applying rotations.
Deletion	Deletion can also be performed in the same way as it is performed in a binary search tree. Deletion may also disturb the balance of the tree therefore, various types of rotations are used to rebalance the tree.

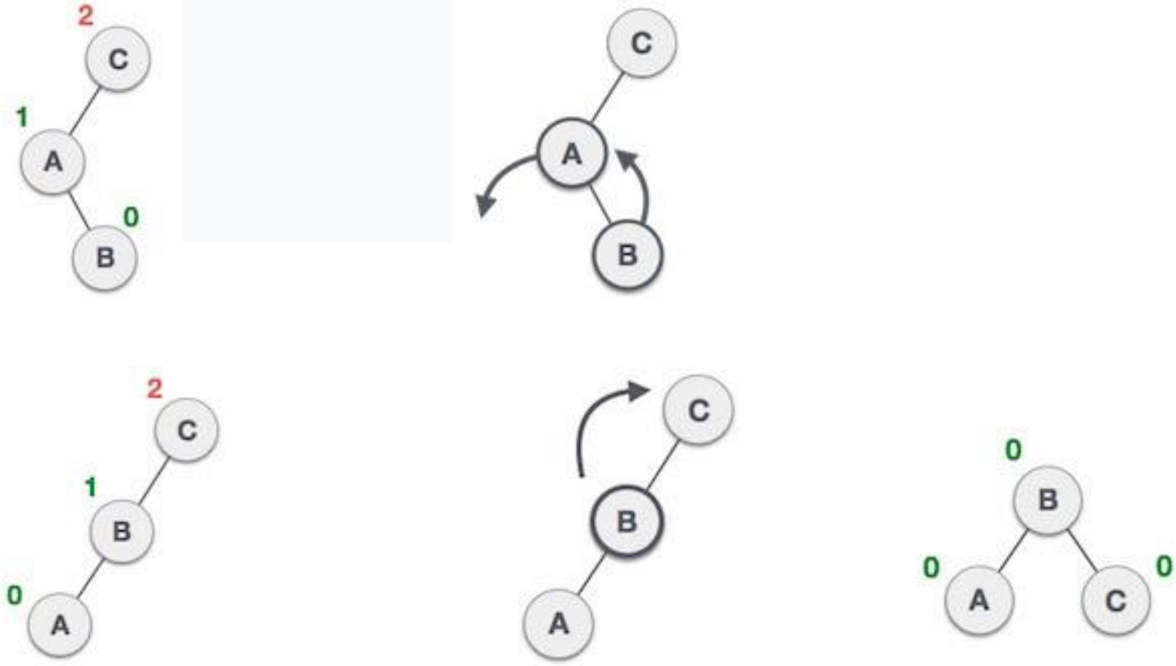
1. RR Rotation



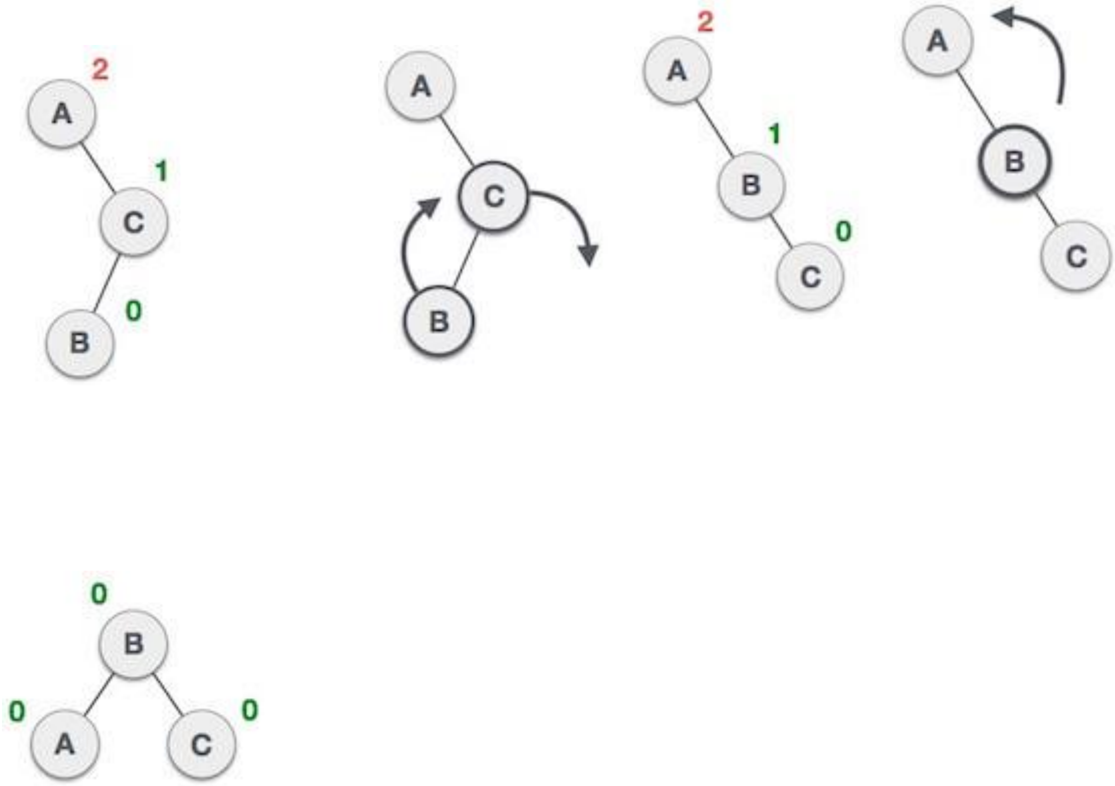
2. LL Rotation



3. LR Rotation



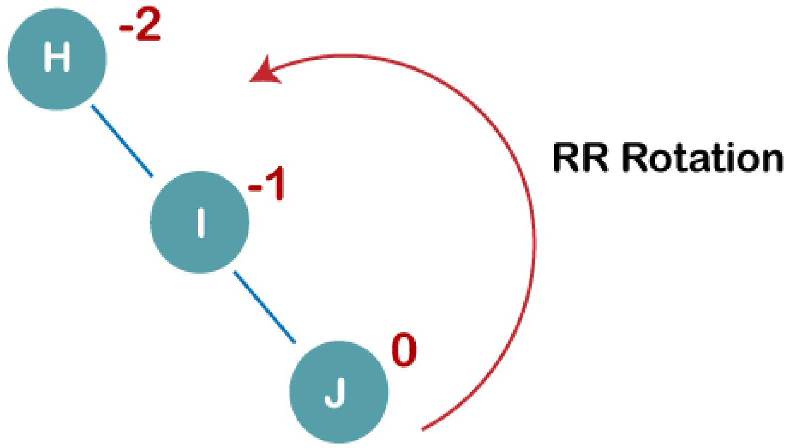
4. RL Rotation

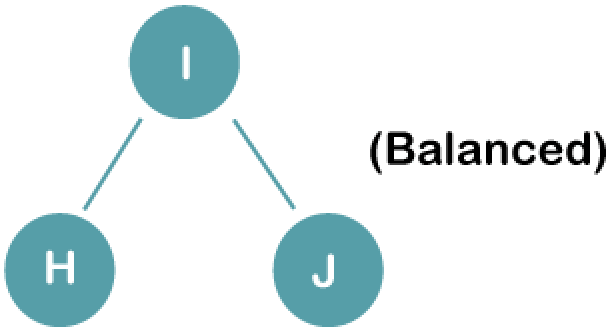


Q: Construct an AVL tree having the following elements

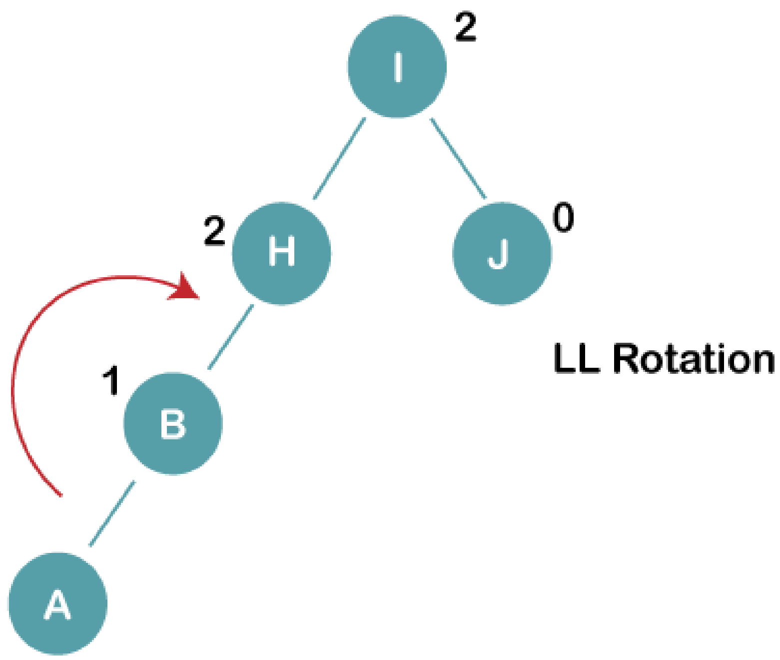
H, I, J, B, A, E, C, F, D, G, K, L

1. **Insert H, I, J**

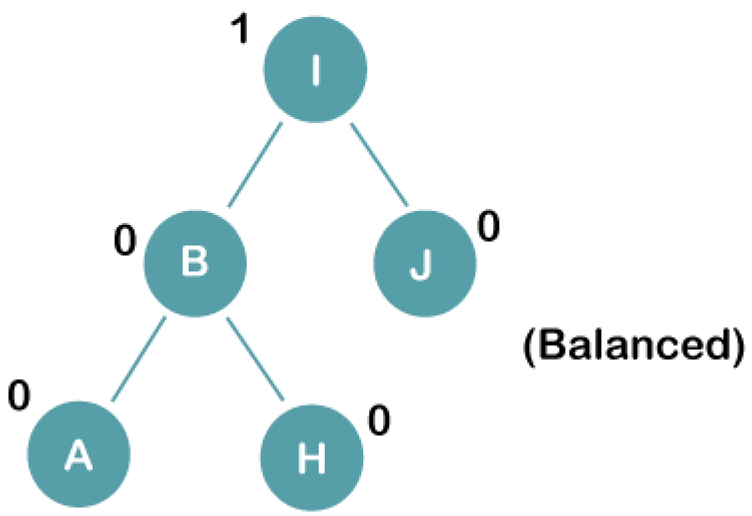




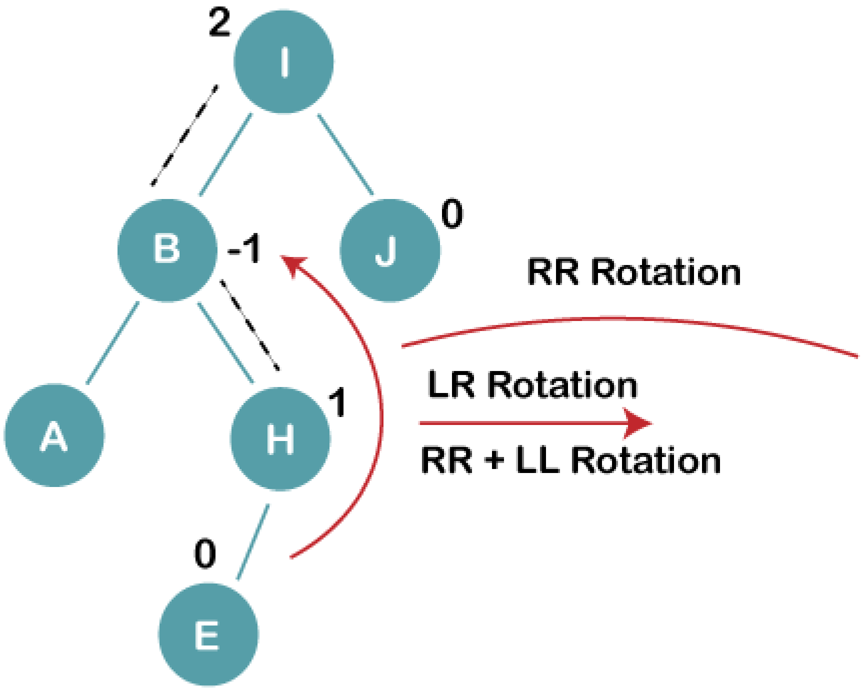
2. **Insert B, A**



The resultant balance tree is:

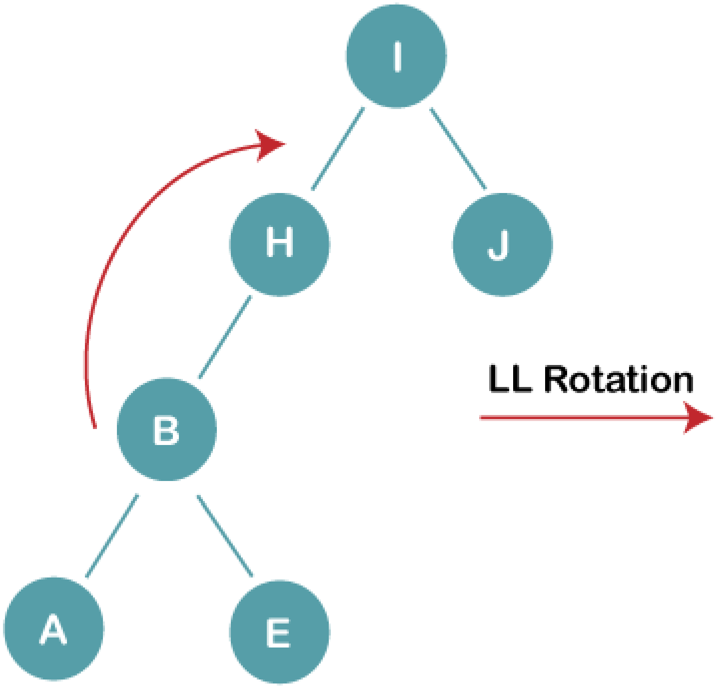


3. **Insert E**



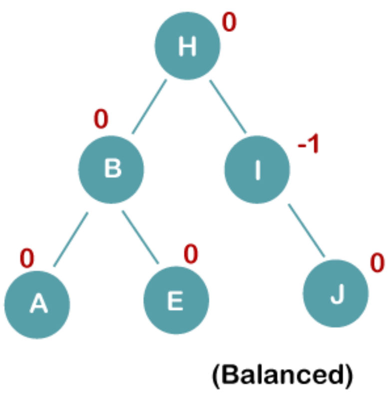
3 a) We first perform RR rotation on node B

The resultant tree after RR rotation is:

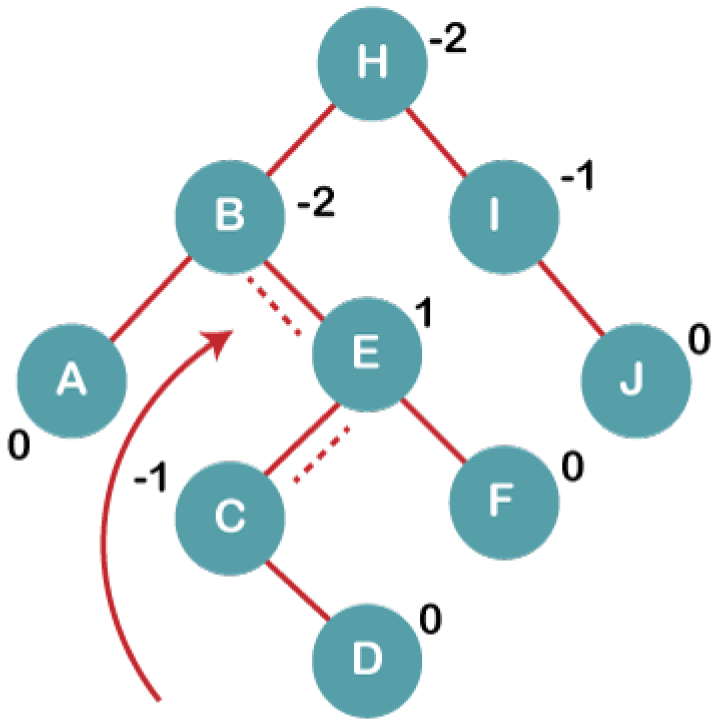


3b) We first perform LL rotation on the node I

The resultant balanced tree after LL rotation is:

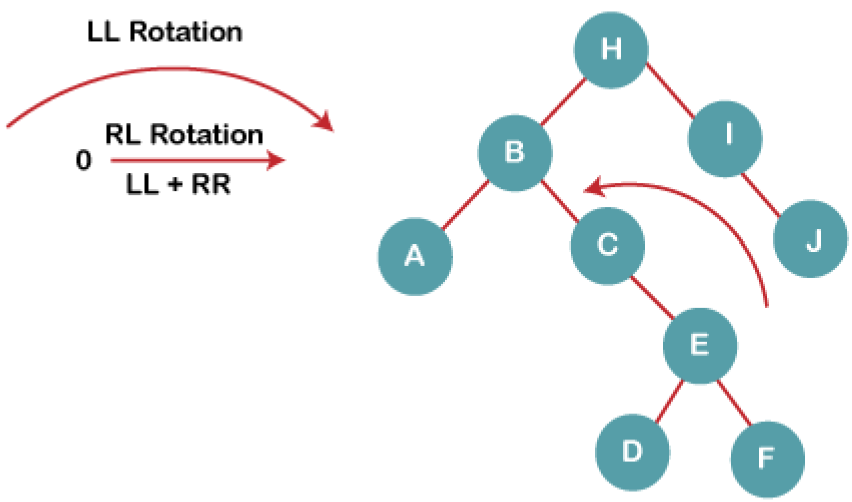


4. Insert C, F, D



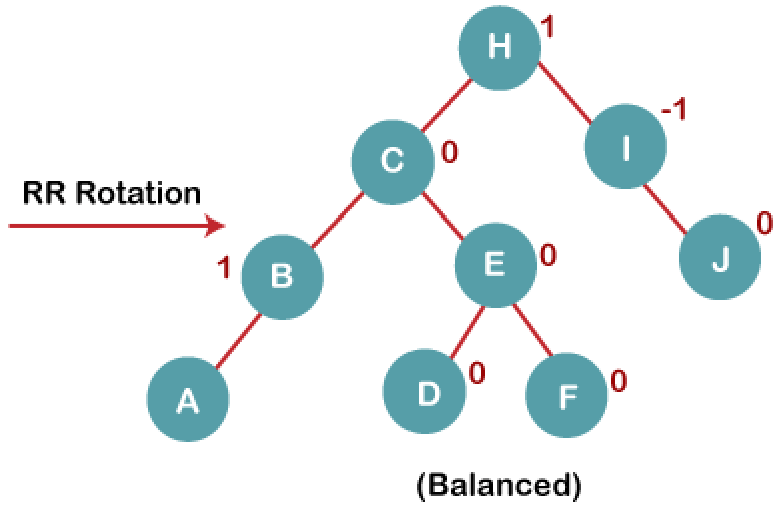
4a) We first perform LL rotation on node E

The resultant tree after LL rotation is:

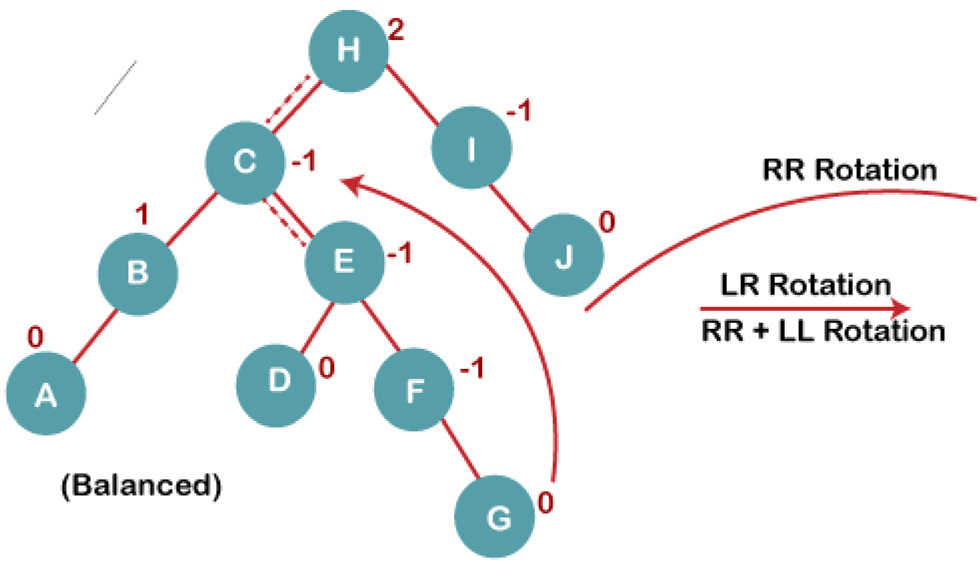


4b) We then perform RR rotation on node B

The resultant balanced tree after RR rotation is:

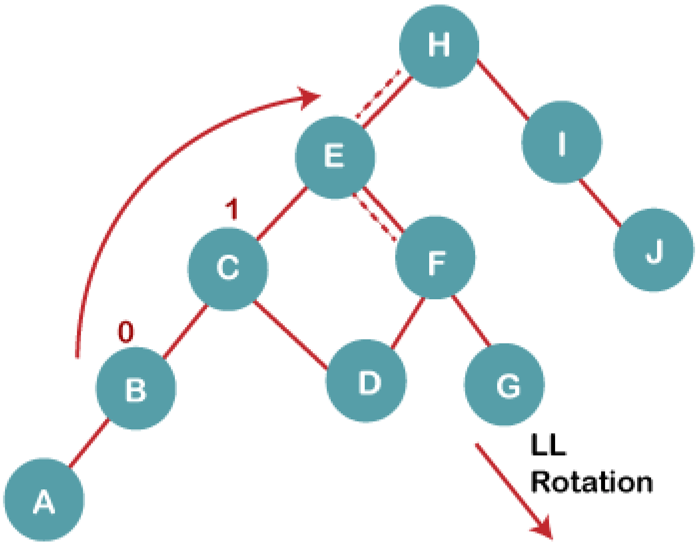


4. **Insert G**



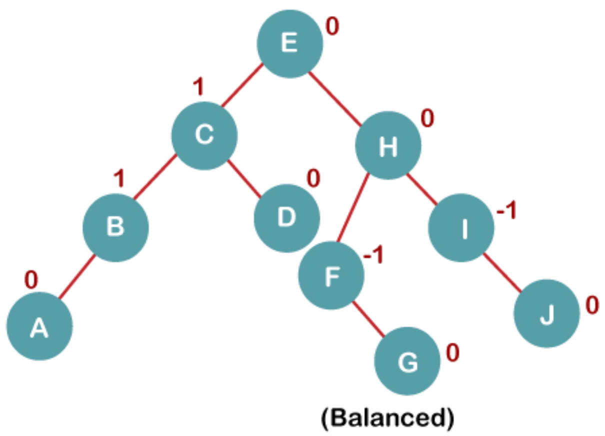
5 a) We first perform RR rotation on node C

The resultant tree after RR rotation is:

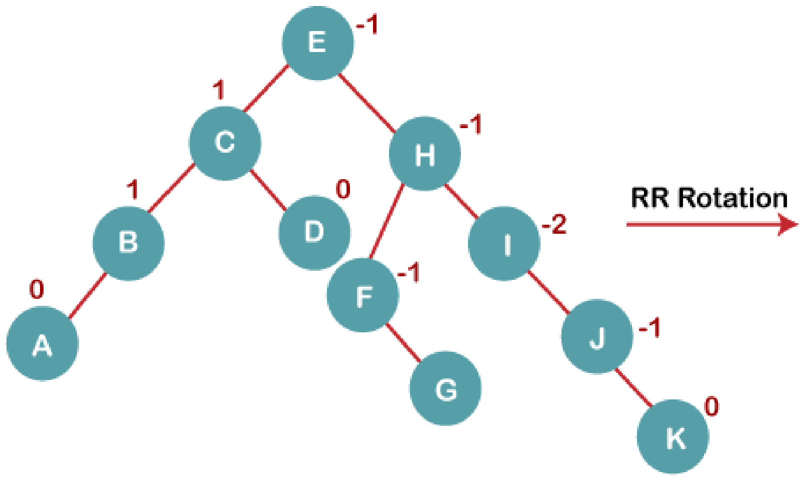


5 b) We then perform LL rotation on node H

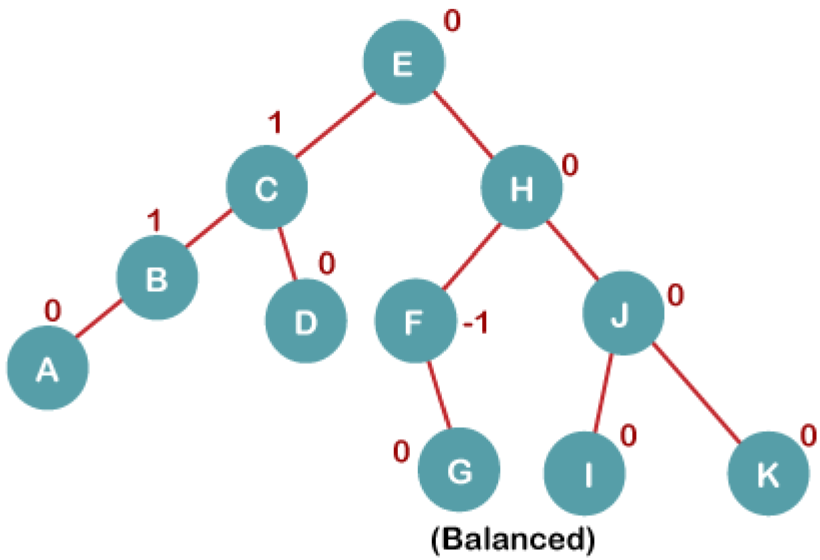
The resultant balanced tree after LL rotation is:



6. Insert K



The resultant balanced tree after RR rotation is:



Program Implementation

```
// AVL tree implementation in C++
#include <iostream>
using namespace std;
class Node {
   public:
  int key;
  Node *left;
  Node *right;
  int height;
};

int max(int a, int b);

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

int max(int a, int b) {
  return (a > b) ? a : b;
}

// New node creation
Node *newNode(int key) {
  Node *node = new Node();
  node->key = key;
  node->left = NULL;
```

```
  node->right = NULL;
  node->height = 1;
  return (node);
}

// Rotate right
Node *rightRotate(Node *y) {
  Node *x = y->left;
  Node *T2 = x->right;
  x->right = y;
  y->left = T2;
  y->height = max(height(y->left),
          height(y->right)) +
        1;
  x->height = max(height(x->left),
          height(x->right)) +
        1;
  return x;
}

// Rotate left
Node *leftRotate(Node *x) {
  Node *y = x->right;
  Node *T2 = y->left;
  y->left = x;
  x->right = T2;
  x->height = max(height(x->left),
          height(x->right)) +
        1;
  y->height = max(height(y->left),
          height(y->right)) +
```

```
      1;
  return y;
}

// Get the balance factor of each node
int getBalanceFactor(Node *N) {
  if (N == NULL)
    return 0;
  return height(N->left) -
       height(N->right);
}

// Insert a node
Node *insertNode(Node *node, int key) {
  // Find the correct postion and insert the node
  if (node == NULL)
    return (newNode(key));
  if (key < node->key)
    node->left = insertNode(node->left, key);
  else if (key > node->key)
    node->right = insertNode(node->right, key);
  else
    return node;

  // Update the balance factor of each node and
  // balance the tree
  node->height = 1 + max(height(node->left),
             height(node->right));
  int balanceFactor = getBalanceFactor(node);
  if (balanceFactor > 1) {
    if (key < node->left->key) {
```

```
    return rightRotate(node);
  } else if (key > node->left->key) {
    node->left = leftRotate(node->left);
    return rightRotate(node);
  }
 }
 if (balanceFactor < -1) {
  if (key > node->right->key) {
    return leftRotate(node);
  } else if (key < node->right->key) {
    node->right = rightRotate(node->right);
    return leftRotate(node);
  }
 }
 return node;
}

// Node with minimum value
Node *nodeWithMimumValue(Node *node) {
 Node *current = node;
 while (current->left != NULL)
  current = current->left;
 return current;
}

// Delete a node
Node *deleteNode(Node *root, int key) {
 // Find the node and delete it
 if (root == NULL)
  return root;
 if (key < root->key)
```

```
  root->left = deleteNode(root->left, key);
else if (key > root->key)
  root->right = deleteNode(root->right, key);
else {
  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;
    free(temp);
  } else {
    Node *temp = nodeWithMimumValue(root->right);
    root->key = temp->key;
    root->right = deleteNode(root->right,
            temp->key);
  }
}

if (root == NULL)
  return root;

// Update the balance factor of each node and
// balance the tree
root->height = 1 + max(height(root->left),
          height(root->right));
int balanceFactor = getBalanceFactor(root);
if (balanceFactor > 1) {
  if (getBalanceFactor(root->left) >= 0) {
```

```
      return rightRotate(root);
    } else {
      root->left = leftRotate(root->left);
      return rightRotate(root);
    }
  }
  if (balanceFactor < -1) {
    if (getBalanceFactor(root->right) <= 0) {
      return leftRotate(root);
    } else {
      root->right = rightRotate(root->right);
      return leftRotate(root);
    }
  }
  return root;
}

// Print the tree
void printTree(Node *root, string indent, bool last) {
  if (root != nullptr) {
    cout << indent;
    if (last) {
      cout << "R----";
      indent += "   ";
    } else {
      cout << "L----";
      indent += "|  ";
    }
    cout << root->key << endl;
    printTree(root->left, indent, false);
    printTree(root->right, indent, true);
```

```
  }
}

int main() {
  Node *root = NULL;
  root = insertNode(root, 33);
  root = insertNode(root, 13);
  root = insertNode(root, 53);
  root = insertNode(root, 9);
  root = insertNode(root, 21);
  root = insertNode(root, 61);
  root = insertNode(root, 8);
  root = insertNode(root, 11);
  printTree(root, "", true);
  root = deleteNode(root, 13);
  cout << "After deleting " << endl;
  printTree(root, "", true);
}
```