3. 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.

AVL tree got its name after its inventor Georgy Adelson-Velsky and Landis.

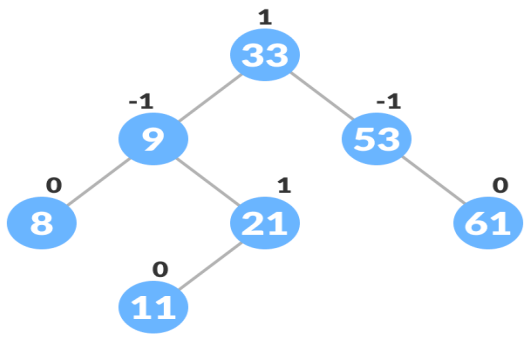
Balance Factor

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.

An example of a balanced avl tree is:



Avl tree

Operations on an AVL tree

Various operations that can be performed on an AVL tree are:

Rotating the subtrees in an AVL Tree

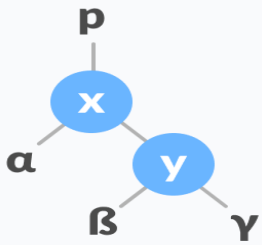
In rotation operation, the positions of the nodes of a subtree are interchanged.

There are two types of rotations:

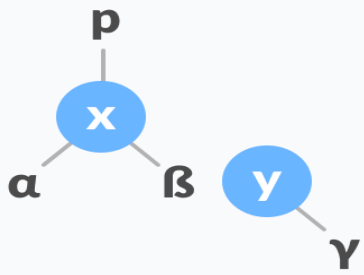
Left Rotate

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

Algorithm

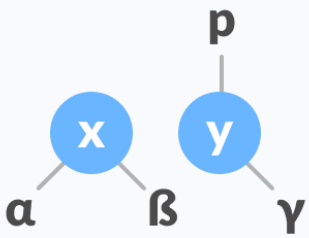


1. Let the initial tree be: Left rotate
2. If `y` has a left subtree, assign `x` as the parent of the left subtree of `y`.

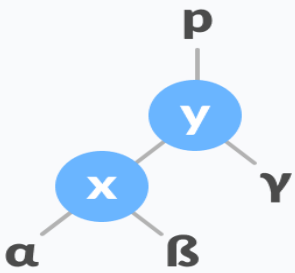


Assign x as the parent of the left subtree of y

3. If the parent of `x` is `NULL`, make `y` as the root of the tree.
4. Else if `x` is the left child of `p`, make `y` as the left child of `p`.



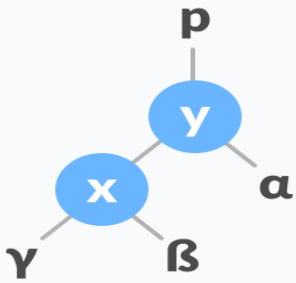
5. Else assign `y` as the right child of `p`. Change the parent of x to that of y



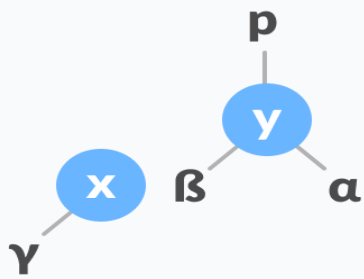
6. Make `y` as the parent of `x`. Assign y as the parent of x.

Right Rotate

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

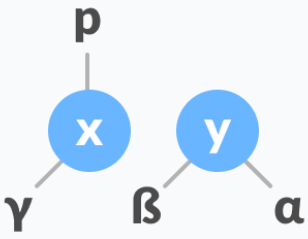


1. Let the initial tree be: Initial tree
2. If `x` has a right subtree, assign y as the parent of the right subtree of `x`.

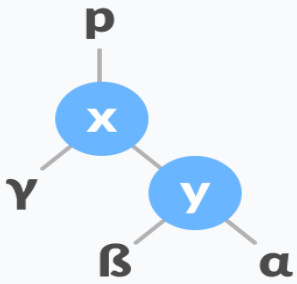


Assign y as the parent of the right subtree of x

3. If the parent of `y` is `NULL`, make `x` as the root of the tree.
4. Else if `y` is the right child of its parent `p`, make `x` as the right child of `p`.



5. Else assign `x` as the left child of `p`. Assign the parent of y as the parent of x.



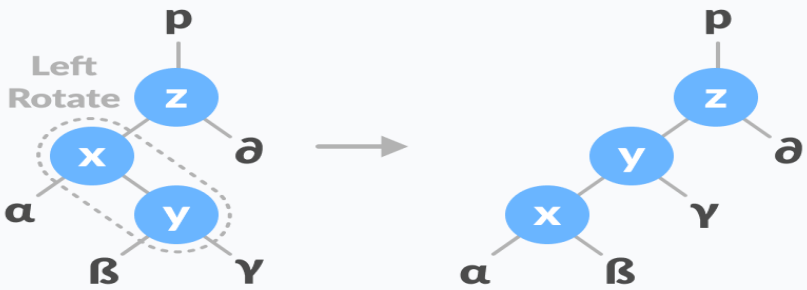
6. Make `x` as the parent of `y`. Assign x as the parent of y

---

Left-Right and Right-Left Rotate

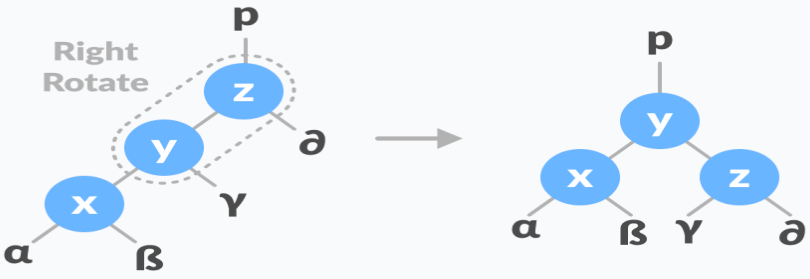
In left-right rotation, the arrangements are first shifted to the left and then to the right.

1. Do left rotation on x-y.



Left rotate x-y

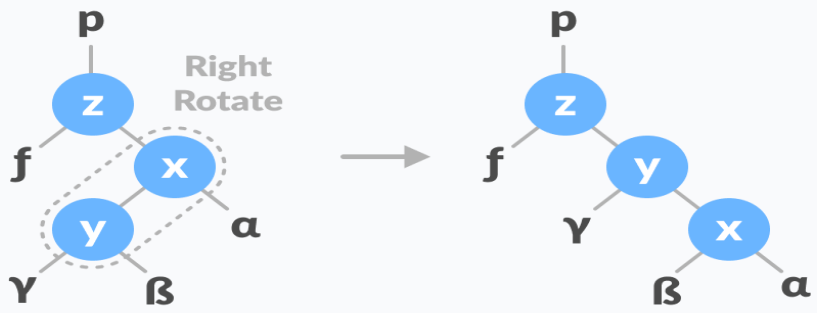
2. Do right rotation on y-z.



Right rotate z-y

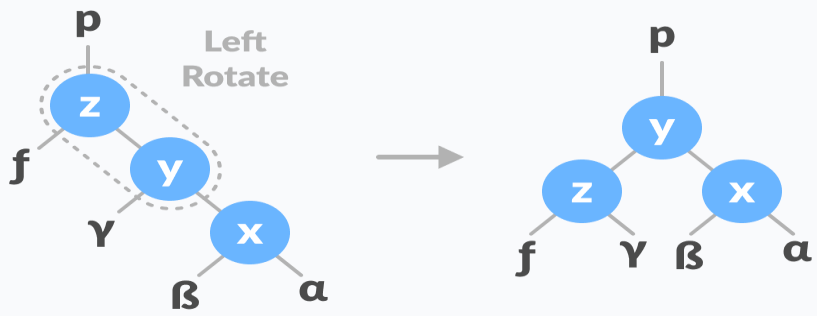
In right-left rotation, the arrangements are first shifted to the right and then to the left.

1. Do right rotation on x-y.



Right rotate x-y

2. Do left rotation on z-y.

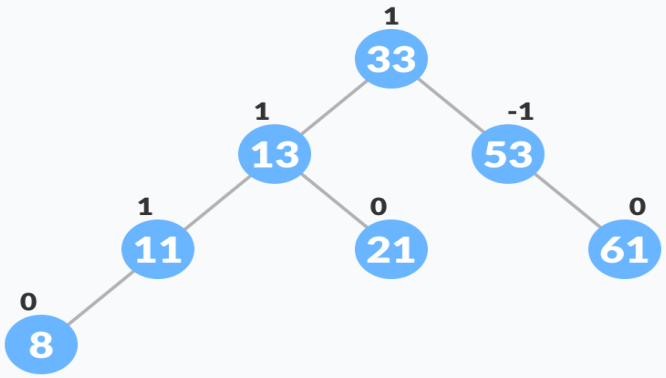


Left rotate z-y

Algorithm to insert a newNode

A `newNode` is always inserted as a leaf node with balance factor equal to 0.

1. Let the initial tree be: Initial tree for

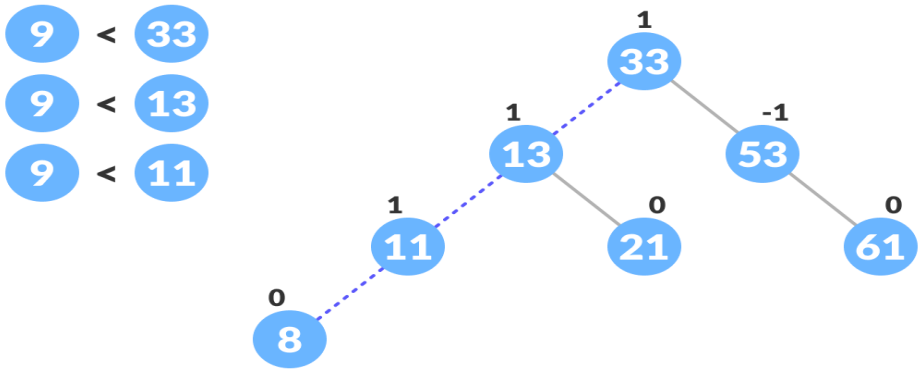


initial tree for insertion

Let the node to be inserted be:

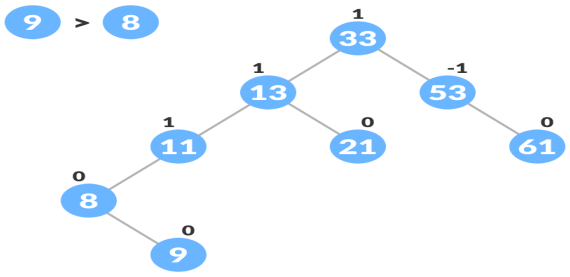
New node

2. Go to the appropriate leaf node to insert a `newNode` using the following recursive steps. Compare `newKey` with `rootKey` of the current tree.
    1. If `newKey` < `rootKey`, call insertion algorithm on the left subtree of the current node until the leaf node is reached.
    2. Else if `newKey` > `rootKey`, call insertion algorithm on the right subtree of current node until the leaf node is reached.
    3. Else, return `leafNode`.



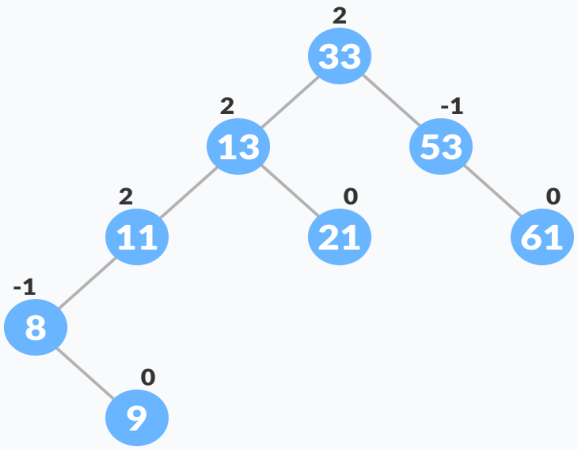
Finding the location to insert newNode

3. Compare `leafKey` obtained from the above steps with `newKey`:
    1. If `newKey` < `leafKey`, make newNode as the `leftChild` of `leafNode`.
    2. Else, make `newNode` as rightChild of `leafNode`.



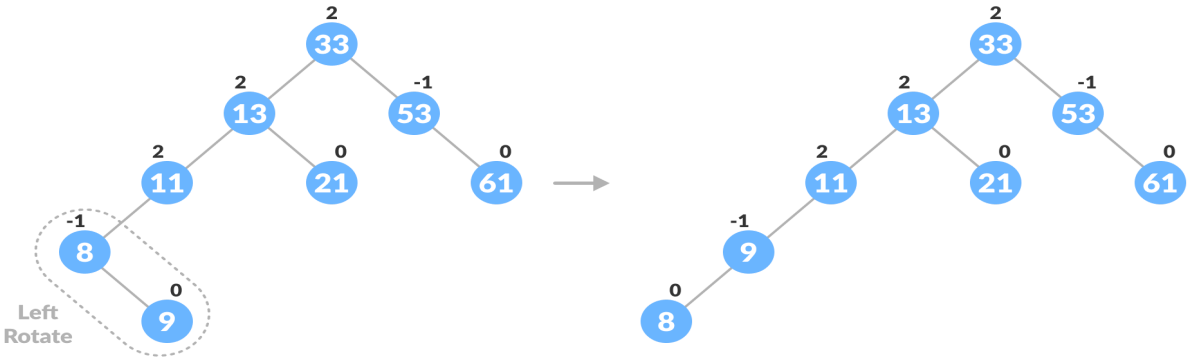
Inserting the new node

4. Update `balanceFactor` of the nodes.

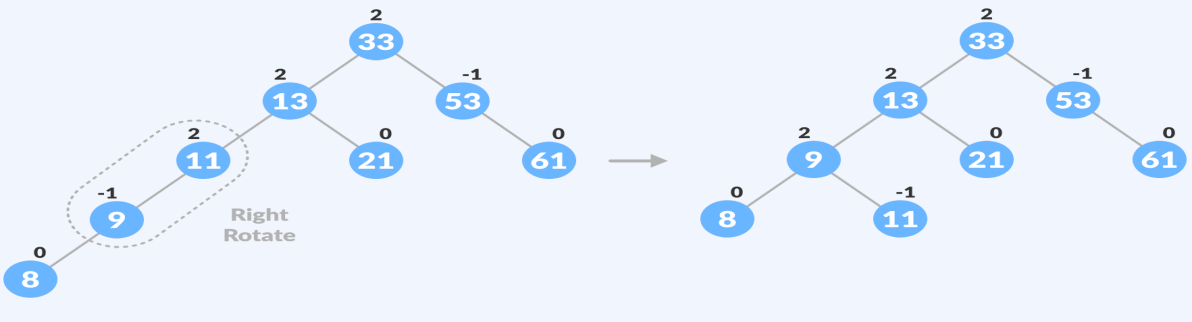


Updating the balance factor after insertion

5. If the nodes are unbalanced, then rebalance the node.
    1. If `balanceFactor` > 1, it means the height of the left subtree is greater than that of the right subtree. So, do a right rotation or left-right rotation
        1. If `newNodeKey` < `leftChildKey` do right rotation.
        2. Else, do left-right rotation.

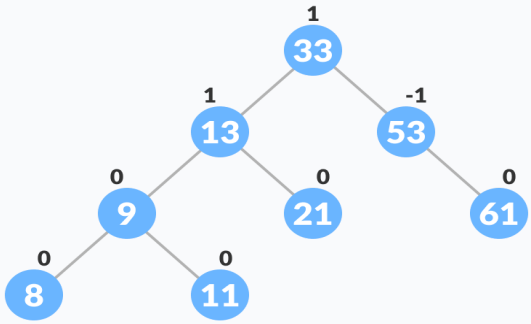


Balancing the tree with rotation



Balancing the tree with rotation

2. If `balanceFactor` < -1, it means the height of the right subtree is greater than that of the left subtree. So, do right rotation or right-left rotation
    1. If `newNodeKey` > `rightChildKey` do left rotation.
    2. Else, do right-left rotation

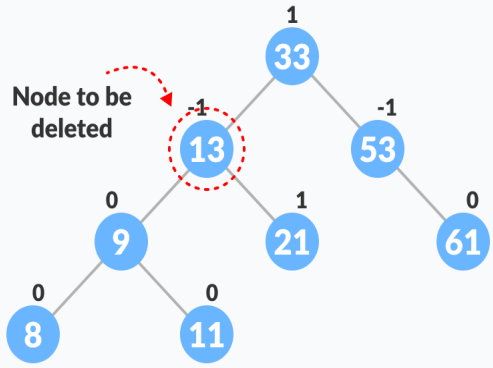


6. The final tree is: Final balanced tree

Algorithm to Delete a node

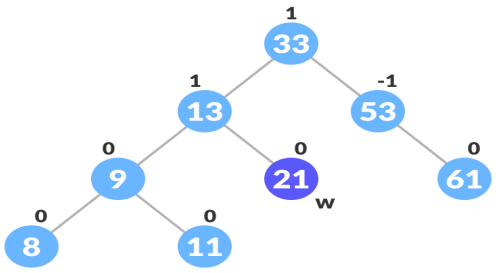
A node is always deleted as a leaf node. After deleting a node, the balance factors of the nodes get changed. In order to rebalance the balance factor, suitable rotations are performed.

1. Locate `nodeToBeDeleted` (recursion is used to find `nodeToBeDeleted` in the code used below).



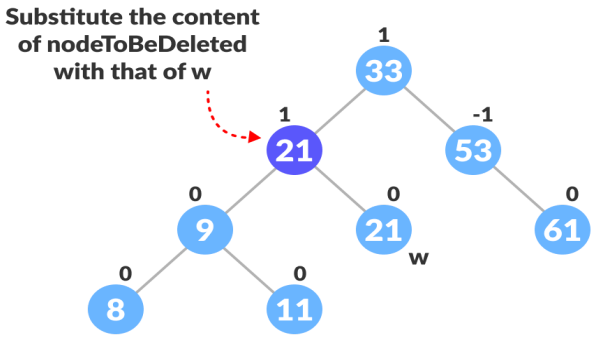
Locating the node to be deleted

2. There are three cases for deleting a node:
    1. If `nodeToBeDeleted` is the leaf node (ie. does not have any child), then remove `nodeToBeDeleted`.
    2. If `nodeToBeDeleted` has one child, then substitute the contents of `nodeToBeDeleted` with that of the child. Remove the child.
    3. If `nodeToBeDeleted` has two children, find the inorder successor `w` of `nodeToBeDeleted` (ie. node with a minimum value of key in the right subtree).



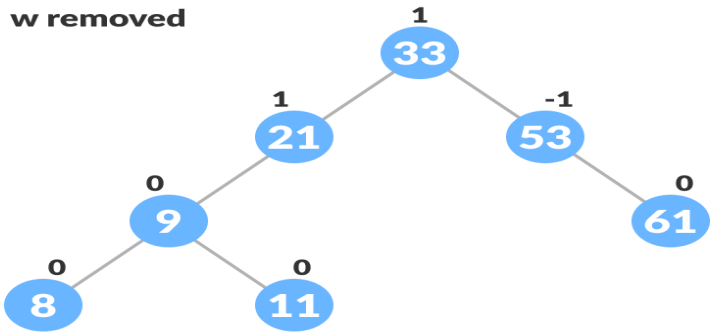
Finding the successor

1. Substitute the contents of `nodeToBeDeleted` with that of `w`.



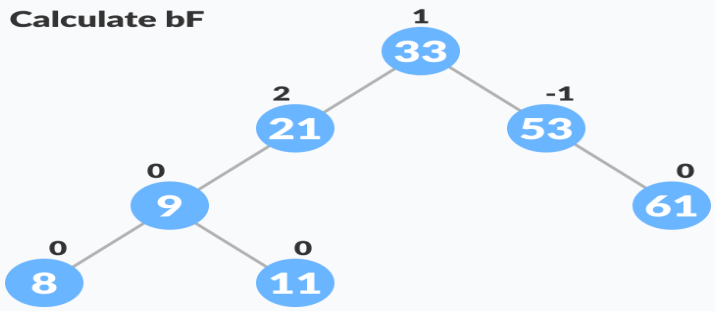
Substitute the node to be deleted

2. Remove the leaf node `w`.



Remove w

3. Update `balanceFactor` of the nodes.

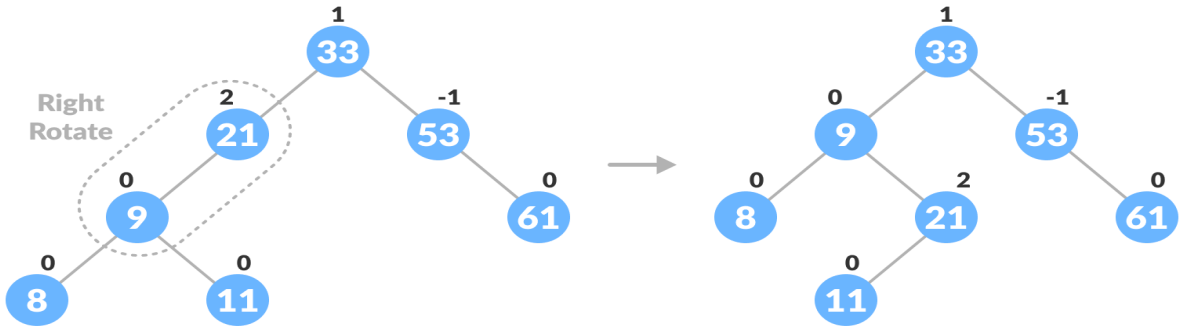


Update bf

4. **Rebalance the tree if the balance factor of any of the nodes is not equal to -1, 0 or 1.**

    1.If `balanceFactor` of `currentNode` > 1,

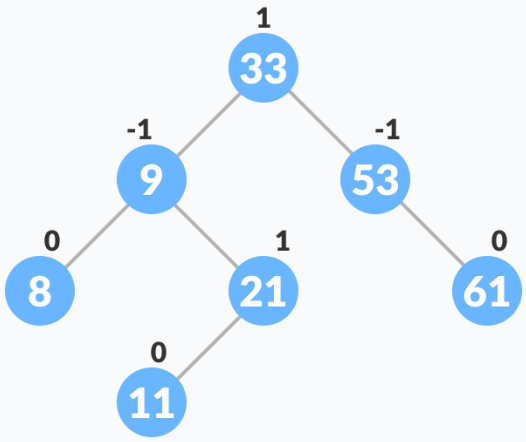
        a. If `balanceFactor` of `leftChild` >= 0, do right rotation.
        b. Else do left-right rotation.



Right-rotate for balancing the tree

2. If `balanceFactor` of `currentNode` < -1,

    a. If `balanceFactor` of `rightChild` <= 0, do left rotation.
    b. Else do right-left rotation.



5. **The final tree is:** Avl tree final

```
#include <bits/stdc++.h>
using namespace std;

class Node {
   public:
  int key;
  Node *left;
  Node *right;
  int height;
};
```

```
Node *nodeWithMimumValue(Node *node)
{
  Node *current = node;
  while (current->left != NULL)
    current = current->left;
  return current;
}

Node *deleteNode(Node *root, int key)
{
```

```
int max(int a, int b);

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

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

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 *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;
}

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;
}

int getBalanceFactor(Node *N) {
  if (N == NULL)
    return 0;
  return height(N->left) -
       height(N->right);
}

Node *insertNode(Node *node, int key)
{
  if (node == NULL)
    return (newNode(key));
  if (key < node->key)
    node->left =
  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;
    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;
}
void printTree(Node *root, string
indent, bool last) {
  if (root != nullptr) {
```

```
insertNode(node->left, key);
  else if (key > node->key)
    node->right =
insertNode(node->right, key);
  else
    return node;
    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;
}
```

```
    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);
}
```

Complexities of Different Operations on an AVL Tree

Insertion	Deletion	Search
O(log n)	O(log n)	O(log n)

AVL Tree Applications

- For indexing large records in databases
- For searching in large databases