**Code:-**

void deletion(Node\* root, int item)

{

Node\* parent = NULL;

Node\* cur = root;

search(cur, item, parent);

if (cur == NULL)

return;

if (cur->left == NULL && cur->right == NULL)

{

if (cur != root)

{

if (parent->left == cur)

parent->left = NULL;

else

parent->right = NULL;

}

else

root = NULL;

free(cur);

}

else if (cur->left && cur->right)

{

Node\* succ = findMinimum(cur- >right);

int val = succ->data;

deletion(root, succ->data);

cur->data = val;

}

else

{

Node\* child = (cur->left)? Cur- >left: cur->right;

if (cur != root)

{

if (cur == parent->left)

parent->left = child;

else

parent->right = child;

}

else

root = child;

free(cur);

}

}

Node\* findMinimum(Node\* cur)

{

while(cur->left != NULL) {

cur = cur->left;

}

return cur;

}

**Check if a BST is AVL :**

**Code:-**

int height(Node\* node) {

if(node == NULL)

return 0;

return 1 + max(height(node->l), height(node->r));

}

bool AVL(Node \*root) {

int lh;

int rh;

if(root == NULL)

return true;

lh = height(root->l);

rh = height(root->r);

if(abs(lh-rh) <= 1 && AVL(root->l) && AVL(root->r))

return true;

return false;

}

**Insertion and Deletion in AVL tree**

**Code:-**

#include <iostream>

using namespace std;

class Node {

public:

int key;

Node \*left;

Node \*right;

int height;

};

int max(int a, int b) {

return (a > b) ? a : b;

}

int height(Node \*N) {

if (N == NULL)

return 0;

return N->height;

}

Node \*newNode(int key) {

Node \*node = new Node();

node->key = key;

node->left = NULL;

node->right = NULL;

node->height = 1;

return (node);

}

**Code:-**

#include <iostream>

using namespace std;

class Node {

public:

int key;

Node \*left;

Node \*right;

int height;

};

int max(int a, int b) {

return (a > b) ? a : b;

}

int height(Node \*N) {

if (N == NULL)

return 0;

return N->height;

}

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 = 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;

}

Node \*nodeWithMimumValue(Node \*node) {

Node \*current = node;

while (current->left != NULL)

current = current->left;

return current;

}

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

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) {  
 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);  
}