```
/*WAP to implement AVL Tree.*/
#include<iostream>
#include <algorithm>
#define pow2(n) (1 << (n))
using namespace std;
/*
* Node Declaration
*/
struct avl_node
{
   int data;
   struct avl_node *left;
   struct avl_node *right;
}*root;
/*
* Class Declaration
*/
class avlTree
{
public:
   int height(avl_node *);
   int diff(avl_node *);
   avl_node *rr_rotation(avl_node *);
   avl_node *ll_rotation(avl_node *);
   avl_node *lr_rotation(avl_node *);
   avl_node *rl_rotation(avl_node *);
   avl_node* balance(avl_node *);
   avl_node* insert(avl_node *, int);
   void display(avl_node *, int);
   void inorder(avl_node *);
   void preorder(avl_node *);
   void postorder(avl_node *);
   avl_node* remove(avl_node* t, int x);
   avl_node* findMin(avl_node*);
   avl_node* findMax(avl_node*);
   avlTree()
   {
        root = NULL;
    }
};
/*
* Main Contains Menu
*/
int main()
{
    int choice, item;
    avlTree avl;
    while (1)
    {
        cout << "\n---------------------" << endl;
        cout << "AVL Tree Implementation" << endl;
        cout << "\n---------------------" << endl;
        cout << "1.Insert Element into the tree" << endl;
        cout << "2.Delete Element into the tree" << endl;
        cout << "3.Display Balanced AVL Tree" << endl;
        cout << "4.InOrder traversal" << endl;
        cout << "5.PreOrder traversal" << endl;
        cout << "6.PostOrder traversal" << endl;
        cout << "7.Exit" << endl;
        cout << "Enter your Choice: ";
        cin >> choice;
        switch (choice)
        {
        case 1:
            cout << "Enter value to be inserted: ";
            cin >> item;
            root = avl.insert(root, item);
            break;
        case 2:
            cout << "Enter value to be deleted: ";
            cin >> item;
            root = avl.remove(root, item);
            break;
        case 3:
            if (root == NULL)
            {
                cout << "Tree is Empty" << endl;
                continue;
            }
            cout << "Balanced AVL Tree:" << endl;
            avl.display(root, 1);
            break;
        case 4:
            cout << "Inorder Traversal:" << endl;
            avl.inorder(root);
            cout << endl;
            break;
        case 5:
            cout << "Preorder Traversal:" << endl;
            avl.preorder(root);
            cout << endl;
            break;
        case 6:
            cout << "Postorder Traversal:" << endl;
            avl.postorder(root);
            cout << endl;
            break;
        case 7:
            exit(1);
            break;
        default:
            cout << "Wrong Choice" << endl;
        }
    }
    return 0;
}
/*
* Height of AVL Tree
*/
int avlTree::height(avl_node *temp)
{
    int h = 0;
    if (temp != NULL)
    {
        int l_height = height(temp->left);
        int r_height = height(temp->right);
        int max_height = max(l_height, r_height);
        h = max_height + 1;
    }
    return h;
}
/*
* Height Difference
*/
int avlTree::diff(avl_node *temp)
{
    int l_height = height(temp->left);
    int r_height = height(temp->right);
    int b_factor = l_height - r_height;
    return b_factor;
}
/*
* Right- Right Rotation
*/
avl_node *avlTree::rr_rotation(avl_node *parent)
{
    avl_node *temp;
    temp = parent->right;
    parent->right = temp->left;
    temp->left = parent;
    return temp;
}
/*
* Left- Left Rotation
*/
avl_node *avlTree::ll_rotation(avl_node *parent)
{
    avl_node *temp;
    temp = parent->left;
    parent->left = temp->right;
    temp->right = parent;
    return temp;
}
/*
* Left - Right Rotation
*/
avl_node *avlTree::lr_rotation(avl_node *parent)
{
    avl_node *temp;
    temp = parent->left;
    parent->left = rr_rotation(temp);
    return ll_rotation(parent);
}
/*
* Right- Left Rotation
*/
avl_node *avlTree::rl_rotation(avl_node *parent)
{
    avl_node *temp;
    temp = parent->right;
    parent->right = ll_rotation(temp);
    return rr_rotation(parent);
}
/*
* Balancing AVL Tree
*/
avl_node *avlTree::balance(avl_node *temp)
{
    int bal_factor = diff(temp);
    if (bal_factor > 1)
    {
        if (diff(temp->left) > 0)
            temp = ll_rotation(temp);
        else
            temp = lr_rotation(temp);
    }
    else if (bal_factor < -1)
    {
        if (diff(temp->right) > 0)
            temp = rl_rotation(temp);
        else
            temp = rr_rotation(temp);
    }
    return temp;
}
/*
* Insert Element into the tree
*/
avl_node *avlTree::insert(avl_node *root, int value)
{
    if (root == NULL)
    {
        root = new avl_node;
        root->data = value;
        root->left = NULL;
        root->right = NULL;
        return root;
    }
    else if (value < root->data)
    {
        root->left = insert(root->left, value);
        root = balance(root);
    }
    else if (value >= root->data)
    {
        root->right = insert(root->right, value);
        root = balance(root);
    }
    return root;
}
/*
* Display AVL Tree
```

```
*/
void avlTree::display(avl_node *ptr, int level)
{
    int i;
    if (ptr != NULL)
    {
        display(ptr->right, level + 1);
        printf("\n");
        if (ptr == root)
            cout << "Root -> ";
        for (i = 0; i < level && ptr != root; i++)
            cout << "        ";
        cout << ptr->data;
        display(ptr->left, level + 1);
    }
}
/*
* Inorder Traversal of AVL Tree
*/
void avlTree::inorder(avl_node *tree)
{
    if (tree == NULL)
        return;
    inorder(tree->left);
    cout << tree->data << "  ";
    inorder(tree->right);
}
/*
* Preorder Traversal of AVL Tree
*/
void avlTree::preorder(avl_node *tree)
{
    if (tree == NULL)
        return;
    cout << tree->data << "  ";
    preorder(tree->left);
    preorder(tree->right);
```

```
}
avl_node* avlTree::findMin(avl_node* t)
{
   if (t == NULL) return NULL;
   else if (t->left == NULL) return t; // if element traverse on max left then return
   else return findMin(t->left); // or recursively traverse max left
}
avl_node* avlTree:: findMax(avl_node* t)
{
   if (t == NULL) return NULL;
   else if (t->right == NULL) return t;
   else return findMax(t->right);
}
/*
* Postorder Traversal of AVL Tree
*/
void avlTree::postorder(avl_node *tree)
{
   if (tree == NULL)
      return;
   postorder(tree->left);
   postorder(tree->right);
   cout << tree->data << "  ";
}
avl_node* avlTree:: remove(avl_node* t, int x)
{
   avl_node* temp;
   // element not found
   if (t == NULL) return NULL;
   // searching element
   else if (x < t->data) t->left = remove(t->left, x);
   else if (x >t->data) t->right = remove(t->right, x);

   // element found
   // element has 2 children
   else if (t->left && t->right)
```

```
    {
        temp = findMin(t->right);
        t->data = temp->data;
        t->right = remove(t->right, t->data);
    }
    // if element has 1 or 0 child
    else
    {
        temp = t;
        if (t->left == NULL) t = t->right;
        else if (t->right == NULL) t = t->left;
        delete temp;
    }
    if (t == NULL) return t;
    // check balanced)
    t = balance(t);
}
```