```
// AVLApp.cpp : This file contains the 'main' function. Program execution begins and ends there.
//

#include "pch.h"
#include <iostream>
// C program to insert a node in AVL tree
#include<stdio.h>
#include<stdlib.h>

using namespace std;

struct Node
{
	int data;
	 Node *left;
	 Node *right;
};

/*
 * Class Declaration
 */
class AVL
{
private:
	Node* createNode(int);
	int diff(Node *);
	Node *LeftRotation(Node *);
	Node *RightRotation(Node *);
	Node *LR_Rotation(Node *);
	Node *RL_Rotation(Node *);
	Node* balanceFactor(Node *);
public:
	Node *root;
	AVL();
	int height(Node *);
	Node* insert(Node *, int);
	void display(Node *, int);
	void inorder(Node *);
	void preorder(Node *);
	void postorder(Node *);
	Node*Delete(struct Node*, int );
	Node* findMax(Node*);
	Node* findMin(Node*);
};
//constuctor

AVL::AVL()
{
	root = NULL;
}

Node* AVL::createNode(int val)
{
	Node* node = new Node;
	node->data = val;
	node->left = NULL;
	node->right = NULL;
	return node;
}
```

```
int AVL::height(Node *temp)

{

    int h = 0;

    if (temp != NULL)

    {

        int l_height = height(temp->left);

        int r_height = height(temp->right);

        int max_height = l_height> r_height? l_height: r_height;

        h = max_height + 1;

    }

    return h;

}


int AVL::diff(Node *temp)

{

    int l_height = height(temp->left);

    int r_height = height(temp->right);

    int b_factor = l_height - r_height;

    return b_factor;

}



Node *AVL::LeftRotation(Node *parent)

{

    Node *temp;

    temp = parent->right;

    parent->right = temp->left;

    temp->left = parent;

    return temp;

}

Node *AVL::RightRotation(Node *parent)
```

```
{
        Node *temp;
        temp = parent->left;
        parent->left = temp->right;
        temp->right = parent;
        return temp;
}

Node *AVL::LR_Rotation(Node *parent)
{
        Node *temp;
        temp = parent->left;
        parent->left = LeftRotation(temp);
        return RightRotation(parent);
}

Node *AVL::RL_Rotation(Node *parent)
{
        Node *temp;
        temp = parent->right;
        parent->right = RightRotation(temp);
        return LeftRotation(parent);
}

Node *AVL::balanceFactor(Node *temp)
{
        int bal_factor = diff(temp);
        if (bal_factor > 1)
        {
                if (diff(temp->left) > 0)
                        temp = RightRotation(temp);
                else
                        temp = LR_Rotation(temp);
        }
        else if (bal_factor < -1)
        {
                if (diff(temp->right) > 0)
                        temp = RL_Rotation(temp);
                else
                        temp = LeftRotation(temp);
        }
        return temp;
}

Node *AVL::insert(Node *node, int value)
{
        if (node == NULL)
        {
                node = createNode(value);
                return node;
        }
        else if (value < node->data)
        {
                node->left = insert(node->left, value);
                node = balanceFactor(node);
        }
        else if (value >= node->data)
        {
                node->right = insert(node->right, value);
                node = balanceFactor(node);
        }
        return node;
```

```
}
struct Node* AVL::Delete(struct Node* r, int value)
{
    if (r == NULL)
        return r;
    else if (value < r->data)
        r->left = Delete(r->left, value);
    else if (value > r->data)
        r->right = Delete(r->right, value);
    // Element to delete if found
    else {
        // Case 1:  No child
        if (r->left == NULL && r->right == NULL) {
            delete r;
            r = NULL;
        }
        //Case 2: One child
        else if (r->left == NULL) {
            Node *temp = r;
            r = r->right;
            delete temp;
            temp = NULL;
        }
        else if (r->right == NULL) {
            Node *temp = r;
            r = r->left;
            delete temp;
            temp = NULL;
        }
        // case 3: 2 children
        else {
            // in the left subtree find maximum
            Node *temp = findMax(r->left);
            r->data = temp->data;
            r->left = Delete(r->left, temp->data);
            /*
            // in the right subtree find minimum
            Node *temp = findMin(r->right);
            r->data = temp->data;
            r->right = Delete(r->right, temp->data);
            */
        }
    }

    // return if there is only one node
    if (r == NULL)
        return r;
    r = balanceFactor(r);
    return r;
}
Node* AVL::findMax(Node *r)
{
    while (r->right != NULL)
        r = r->right;
    return r;
}
Node* AVL::findMin(Node *r)
{
    while (r->left != NULL)
        r = r->left;
    return r;
}
```

```
/*
 * Inorder Traversal of AVL Tree
 */
void AVL::inorder(Node *tree)
{
        if (tree == NULL)
                return;
        inorder(tree->left);
        cout << tree->data << "  ";
        inorder(tree->right);
}
/*
 * Preorder Traversal of AVL Tree
 */
void AVL::preorder(Node *tree)
{
        if (tree == NULL)
                return;
        cout << tree->data << "  ";
        preorder(tree->left);
        preorder(tree->right);

}

/*
 * Postorder Traversal of AVL Tree
 */
void AVL::postorder(Node *tree)
{
        if (tree == NULL)
```

```
                return;
        postorder(tree->left);
        postorder(tree->right);
        cout << tree->data << "  ";
}



int main()
{
        AVL avl;
        avl.root = avl.insert(avl.root, 10);
        avl.root = avl.insert(avl.root, 20);
        avl.root = avl.insert(avl.root, 30);
        avl.root = avl.insert(avl.root, 40);
        avl.root = avl.insert(avl.root, 50);
        avl.root = avl.insert(avl.root, 70);
        avl.root = avl.insert(avl.root, 5);
        avl.display(avl.root, avl.height(avl.root));

        avl.root = avl.Delete(avl.root, 30);
        cout << endl << endl << endl;
        system("pause");
        return 0;
}
```