Implementation of AVL Tree

Write a function in C program to insert a new node with a given value into an AVL tree. Ensure that the tree remains balanced after insertion by performing rotations if necessary. Repeat the above operation to delete a node from AVL tree.

PROGRAM:

```
#include <stdio.h>
#include <stdlib.h>

// Definition of the AVL tree node
struct Node {
    int data;
    struct Node* left;
    struct Node* right;
    int height;
};

// Function to get the height of the tree
int height(struct Node* node) {
    if (node == NULL)
        return 0;
    return node->height;
}

// Function to create a new node
struct Node* createNode(int data) {
    struct Node* node = (struct Node*)malloc(sizeof(struct Node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;
    node->height = 1;  // New node is initially added at leaf
    return node;
}

// Function to get the maximum of two integers
```

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

// Right rotate subtree rooted with y
struct Node* rightRotate(struct Node* y) {
    struct Node* x = y->left;
    struct Node* T2 = x->right;

    // Perform rotation
    x->right = y;
    y->left = T2;

    // Update heights
    y->height = max(height(y->left), height(y->right)) + 1;
    x->height = max(height(x->left), height(x->right)) + 1;

    // Return new root
    return x;
}

// Left rotate subtree rooted with x
struct Node* leftRotate(struct Node* x) {
    struct Node* y = x->right;
    struct Node* T2 = y->left;

    // Perform rotation
    y->left = x;
    x->right = T2;

    // Update heights
    x->height = max(height(x->left), height(x->right)) + 1;
    y->height = max(height(y->left), height(y->right)) + 1;
```

```
    // Return new root
    return y;
}

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

// Function to insert a node into the AVL tree
struct Node* insert(struct Node* node, int data) {
    if (node == NULL)
        return createNode(data);

    if (data < node->data)
        node->left = insert(node->left, data);
    else if (data > node->data)
        node->right = insert(node->right, data);
    else
        return node;

    node->height = 1 + max(height(node->left), height(node->right));

    int balance = getBalance(node);

    if (balance > 1 && data < node->left->data)
        return rightRotate(node);

    if (balance < -1 && data > node->right->data)
        return leftRotate(node);

    if (balance > 1 && data > node->left->data) {
```

```
        node->left = leftRotate(node->left);
        return rightRotate(node);
    }

    if (balance < -1 && data < node->right->data) {
        node->right = rightRotate(node->right);
        return leftRotate(node);
    }

    return node;
}

// Function to find the node with the minimum value in the tree
struct Node* findMin(struct Node* node) {
    struct Node* current = node;

    while (current->left != NULL)
        current = current->left;

    return current;
}

// Function to delete a node from the AVL tree
struct Node* deleteNode(struct Node* root, int data) {
    if (root == NULL)
        return root;

    if (data < root->data)
        root->left = deleteNode(root->left, data);
    else if (data > root->data)
        root->right = deleteNode(root->right, data);
    else {
        if ((root->left == NULL) || (root->right == NULL)) {
            struct Node* temp = root->left ? root->left : root->right;
```

```
        if (temp == NULL) {
            temp = root;
            root = NULL;
        } else
            *root = *temp;
        free(temp);
    } else {
        struct Node* temp = findMin(root->right);

        root->data = temp->data;

        root->right = deleteNode(root->right, temp->data);
    }
}

if (root == NULL)
    return root;

root->height = 1 + max(height(root->left), height(root->right));

int balance = getBalance(root);

if (balance > 1 && getBalance(root->left) >= 0)
    return rightRotate(root);

if (balance > 1 && getBalance(root->left) < 0) {
    root->left = leftRotate(root->left);
    return rightRotate(root);
}

if (balance < -1 && getBalance(root->right) <= 0)
    return leftRotate(root);
```

```
    if (balance < -1 && getBalance(root->right) > 0) {
        root->right = rightRotate(root->right);
        return leftRotate(root);
    }

    return root;
}

// Function to display the tree in in-order traversal
void inorder(struct Node* root) {
    if (root != NULL) {
        inorder(root->left);
        printf("%d ", root->data);
        inorder(root->right);
    }
}

// Main function to test the AVL tree operations
int main() {
    struct Node* root = NULL;
    int choice, data;

    while (1) {
        printf("\nAVL Tree Operations:\n");
        printf("1. Insert\n");
        printf("2. Delete\n");
        printf("3. Display\n");
        printf("4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter the value to insert: ");
```

```
            scanf("%d", &data);
            root = insert(root, data);
            break;
        case 2:
            printf("Enter the value to delete: ");
            scanf("%d", &data);
            root = deleteNode(root, data);
            break;
        case 3:
            printf("AVL tree in-order traversal: ");
            inorder(root);
            printf("\n");
            break;
        case 4:
            exit(0);
        default:
            printf("Invalid choice! Please try again.\n");
      }
   }

   return 0;
}
```

OUTPUT:

```
AVL Tree Operations:
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 1
Enter the value to insert: 30

AVL Tree Operations:
1. Insert
```

2. Delete

3. Display

4. Exit

Enter your choice: 1

Enter the value to insert: 20

AVL Tree Operations:

1. Insert

2. Delete

3. Display

4. Exit

Enter your choice: 1

Enter the value to insert: 40

AVL Tree Operations:

1. Insert

2. Delete

3. Display

4. Exit

Enter your choice: 3

AVL tree in-order traversal: 20 30 40

AVL Tree Operations:

1. Insert

2. Delete

3. Display

4. Exit

Enter your choice: 1

Enter the value to insert: 10

AVL Tree Operations:

1. Insert

2. Delete

3. Display

4. Exit

Enter your choice: 3

AVL tree in-order traversal: 10 20 30 40

AVL Tree Operations:

1. Insert

2. Delete

3. Display

4. Exit

Enter your choice: 2

Enter the value to delete: 20

AVL Tree Operations:

1. Insert

2. Delete

3. Display

4. Exit

Enter your choice: 3

AVL tree in-order traversal: 10 30 400