AVL TREE PROGRAM CONTAINS ALL OPERATIONS

```
#include <stdio.h>

#include <stdlib.h>



struct TreeNode {

	int data;

	struct TreeNode* left;

	struct TreeNode* right;

	int height;

};



int height(struct TreeNode* node) {

	if (node == NULL)

		return 0;

	return node->height;

}



int max(int a, int b) {

	return (a > b) ? a : b;

}



struct TreeNode* createNode(int key) {
```

```
    struct TreeNode* newNode = (struct TreeNode*)malloc(sizeof(struct TreeNode));
    if (newNode != NULL) {
        newNode->data = key;
        newNode->left = NULL;
        newNode->right = NULL;
        newNode->height = 1;
    }
    return newNode;
}

struct TreeNode* rightRotate(struct TreeNode* y) {
    struct TreeNode* x = y->left;
    struct TreeNode* 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;
}
```

```
struct TreeNode* leftRotate(struct TreeNode* x) {
	struct TreeNode* y = x->right;
	struct TreeNode* 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 getBalance(struct TreeNode* node) {
	if (node == NULL)
		return 0;
	return height(node->left) - height(node->right);
}
```

```
struct TreeNode* insert(struct TreeNode* root, int key) {

    if (root == NULL)
        return createNode(key);

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


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


    int balance = getBalance(root);


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


    if (balance < -1 && key > root->right->data)
        return leftRotate(root);
```

```
    if (balance > 1 && key > root->left->data) {
        root->left = leftRotate(root->left);
        return rightRotate(root);
    }

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

    return root;
}

struct TreeNode* minValueNode(struct TreeNode* node) {
    struct TreeNode* current = node;
    while (current->left != NULL)
        current = current->left;
    return current;
}

struct TreeNode* deleteNode(struct TreeNode* root, int key) {
```

```
if (root == NULL)
        return root;



if (key < root->data)
        root->left = deleteNode(root->left, key);
else if (key > root->data)
        root->right = deleteNode(root->right, key);
else {


        if ((root->left == NULL) || (root->right == NULL)) {
                struct TreeNode* temp = root->left ? root->left : root->right;



                if (temp == NULL) {
                        temp = root;
                        root = NULL;
                } else // One child case
                        *root = *temp;


                free(temp);
        } else {


                struct TreeNode* temp = minValueNode(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;
}

void inOrderTraversal(struct TreeNode* root) {
    if (root != NULL) {
        inOrderTraversal(root->left);
        printf("%d ", root->data);
        inOrderTraversal(root->right);
    }
}

void freeAVLTree(struct TreeNode* root) {
    if (root != NULL) {
```

```
        freeAVLTree(root->left);
        freeAVLTree(root->right);
        free(root);
    }
}

int main() {
    struct TreeNode* root = NULL;
    int choice, key;

    do {
        printf("\nAVL Tree Operations:\n");
        printf("1. Insert a node\n");
        printf("2. Delete a node\n");
        printf("3. In-order Traversal\n");
        printf("4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter the key to insert: ");
                scanf("%d", &key);
                root = insert(root, key);
                break;
            case 2:
```

```
                printf("Enter the key to delete: ");
                scanf("%d", &key);
                root = deleteNode(root, key);
                break;
            case 3:
                printf("In-order Traversal: ");
                inOrderTraversal(root);
                printf("\n");
                break;
            case 4:

                freeAVLTree(root);
                printf("Exiting...\n");
                break;
            default:
                printf("Invalid choice! Please enter a valid option.\n");
        }

    } while (choice != 4);

    return 0;
}
```

OUTPUT

AVL Tree Operations:

1. Insert a node

2. Delete a node

3. In-order Traversal

4. Exit

Enter your choice: 1

Enter the key to insert: 2 3

AVL Tree Operations:

1. Insert a node

2. Delete a node

3. In-order Traversal

4. Exit

Enter your choice: In-order Traversal: 2

AVL Tree Operations:

1. Insert a node

2. Delete a node

3. In-order Traversal

4. Exit

Enter your choice: 2 3

Enter the key to delete:

AVL Tree Operations:

1. Insert a node

2. Delete a node

3. In-order Traversal

4. Exit

Enter your choice: 2

Enter the key to delete: 3

AVL Tree Operations:

1. Insert a node

2. Delete a node

3. In-order Traversal

4. Exit

Enter your choice: 3

In-order Traversal: 2

AVL Tree Operations:

1. Insert a node

2. Delete a node

3. In-order Traversal

4. Exit

Enter your choice:

=== Session Ended. Please Run the code again ===