EXERCISE 20

Write a program to perform the following operations:

a) Insert an element into a AVL tree

b) Delete an element from a AVL tree

c) Search for a key element in a AVL tree

Aim:

To write a C program to perform the following operations on an AVL Tree:
a) Insert an element
b) Delete an element
c) Search for a key element

Algorithm:

a) Insertion

1. Insert the node using normal BST insertion.
2. Update the height of the current node.
3. Check the balance factor to see if it became unbalanced.
4. If unbalanced, perform one of the following rotations:
    - Left-Left (LL)
    - Right-Right (RR)
    - Left-Right (LR)
    - Right-Left (RL)

b) Deletion

1. Perform standard BST delete.
2. Update the height and balance factor.
3. If unbalanced, perform appropriate rotation.

c) Search

1. Traverse the tree using BST search logic (left or right).
2. Return the node if found, else return NULL.

Program:

```
#include <stdio.h>
#include <stdlib.h>
struct Node {
   int key;
   struct Node* left;
   struct Node* right;
   int height;
};
int max(int a, int b) {
   return (a > b) ? a : b;
}
int height(struct Node* N) {
   if (N == NULL)
      return 0;
   return N->height;
}
struct Node* newNode(int key) {
   struct Node* node = (struct Node*)malloc(sizeof(struct Node));
   node->key = key;
   node->left = node->right = NULL;
   node->height = 1;
   return node;
}
struct Node* rightRotate(struct Node* y) {
   struct Node* x = y->left;
```

```
    struct 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;
}
struct Node* leftRotate(struct Node* x) {
    struct Node* y = x->right;
    struct 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 getBalance(struct Node* N) {
    if (N == NULL)
        return 0;
    return height(N->left) - height(N->right);
}
struct Node* insert(struct Node* node, int key) {
    if (node == NULL)
        return newNode(key);
    if (key < node->key)
        node->left = insert(node->left, key);
```

```
    else if (key > node->key)
        node->right = insert(node->right, key);
    else
        return node;
    node->height = 1 + max(height(node->left), height(node->right));
    int balance = getBalance(node);
    if (balance > 1 && key < node->left->key)
        return rightRotate(node);
    if (balance < -1 && key > node->right->key)
        return leftRotate(node);
    if (balance > 1 && key > node->left->key) {
        node->left = leftRotate(node->left);
        return rightRotate(node);
    }
    if (balance < -1 && key < node->right->key) {
        node->right = rightRotate(node->right);
        return leftRotate(node);
    }
    return node;
}
struct Node* minValueNode(struct Node* node) {
    struct Node* current = node;
    while (current->left != NULL)
        current = current->left;
    return current;
}
```

```
struct Node* deleteNode(struct 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)) {
         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 = minValueNode(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 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;
}struct Node* search(struct Node* root, int key) {
    if (root == NULL || root->key == key)
        return root;
    if (key < root->key)
        return search(root->left, key);
    return search(root->right, key);
}
void inorder(struct Node* root) {
    if (root != NULL) {
        inorder(root->left);
        printf("%d ", root->key);
        inorder(root->right);
    }
```

```
}
int main() {
    struct Node* root = NULL;
    int choice, value;
    while (1) {
        printf("\nAVL Tree Operations:\n");
        printf("1. Insert\n2. Delete\n3. Search\n4. Display (Inorder)\n5. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
        switch (choice) {
        case 1:
            printf("Enter value to insert: ");
            scanf("%d", &value);
            root = insert(root, value);
            break;
        case 2:
            printf("Enter value to delete: ");
            scanf("%d", &value);
            root = deleteNode(root, value);
            break;
        case 3:
            printf("Enter key to search: ");
            scanf("%d", &value);
            if (search(root, value))
                printf("Key found in AVL tree.\n");
            else
```

```
            printf("Key not found.\n");
          break;
        case 4:
          printf("Inorder Traversal: ");
          inorder(root);
          printf("\n");
          break;
        case 5:
          return 0;
        default:
          printf("Invalid choice!\n");
        }
    }
    return 0;
}
```

Input and Output:

```
AVL Tree Operations:
1. Insert
2. Delete
3. Search
4. Display (Inorder)
5. Exit
Enter your choice:
AVL Tree Operations:
1. Insert
2. Delete
3. Search
4. Display (Inorder)
5. Exit
Enter your choice: 1
Enter value to insert: 30

Enter your choice: 1
Enter value to insert: 20

Enter your choice: 1
Enter value to insert: 40

Enter your choice: 1
Enter value to insert: 10

Enter your choice: 4
Inorder Traversal: 10 20 30 40
```

Result:

The program implemented successfully using AVL tree