1. Write a C program to perform the following operations:

a) Insert an element into an AVL tree.

Code:

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

// Structure for AVL tree node
typedef struct Node {
    int key;
    struct Node *left;
    struct Node *right;
    int height;
} Node;

// Utility function to get the height of the tree
int height(Node *N) {
    if (N == NULL) return 0;
    return N->height;
}

// Utility function to get the maximum of two integers
int max(int a, int b) {
    return (a > b) ? a : b;
}

// Utility function to create a new node
Node* newNode(int key) {
    Node* node = (Node*) malloc(sizeof(Node));
    node->key = key;
    node->left = NULL;
    node->right = NULL;
    node->height = 1; // New node is initially at height 1
```

```
    return node;
}

// Right rotate utility
Node* rightRotate(Node *y) {
    Node *x = y->left;
    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 utility
Node* leftRotate(Node *x) {
    Node *y = x->right;
    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
int getBalance(Node *N) {
    if (N == NULL) return 0;
    return height(N->left) - height(N->right);
}

// Insert a node into the AVL tree
Node* insert(Node* node, int key) {
    // 1. Perform the normal BST insert
    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 // Duplicate keys are not allowed in the AVL tree
        return node;

    // 2. Update height of this ancestor node
    node->height = 1 + max(height(node->left), height(node->right));

    // 3. Get the balance factor of this ancestor node to check whether
    // this node became unbalanced
    int balance = getBalance(node);
```

```
    // If this node becomes unbalanced, then there are 4 cases

    // Left Left Case
    if (balance > 1 && key < node->left->key)
        return rightRotate(node);

    // Right Right Case
    if (balance < -1 && key > node->right->key)
        return leftRotate(node);

    // Left Right Case
    if (balance > 1 && key > node->left->key) {
        node->left = leftRotate(node->left);
        return rightRotate(node);
    }

    // Right Left Case
    if (balance < -1 && key < node->right->key) {
        Node ->right = rightRotate(node->right);
        return leftRotate(node);
    }

    // Return the (unchanged) node pointer
    return node;
}

// Utility function to print the AVL tree (Inorder Traversal)
void inorder(Node *root) {
    if (root != NULL) {
        inorder(root->left);
        printf("%d ", root->key);
```

```
        inorder(root->right);
    }
}

// Driver program to test the above functions
int main() {
    Node *root = NULL;

    // Inserting elements
    root = insert(root, 10);
    root = insert(root, 20);
    root = insert(root, 30);
    root = insert(root, 40);
    root = insert(root, 50);
    root = insert(root, 25);

    // Print the AVL tree
    printf("Inorder traversal of the constructed AVL tree is:\n");
    inorder(root);
    printf("\n");

    return 0;
}
```

Output: inorder traversal of the constructed AVL tree is:

10 20 25 30 40 50

b) Delete an element from an AVL tree.

Code:

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

// Structure for AVL tree node
typedef struct Node {
```

```
    int key;
    struct Node *left;
    struct Node *right;
    int height;
} Node;

// Utility function to get the height of the tree
int height(Node *N) {
    if (N == NULL) return 0;
    return N->height;
}

// Utility function to get the maximum of two integers
int max(int a, int b) {
    return (a > b) ? a : b;
}

// Utility function to create a new node
Node* newNode(int key) {
    Node* node = (Node*) malloc(sizeof(Node));
    node->key = key;
    node->left = NULL;
    node->right = NULL;
    node->height = 1; // New node is initially at height 1
    return node;
}

// Right rotate utility
Node* rightRotate(Node *y) {
```

```
    Node *x = y->left;
    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 utility
Node* leftRotate(Node *x) {
    Node *y = x->right;
    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
int getBalance(Node *N) {
    if (N == NULL) return 0;
    return height(N->left) - height(N->right);
}

// Find the node with the minimum key value
Node* minValueNode(Node* node) {
    Node* current = node;
    while (current->left != NULL)
        current = current->left;
    return current;
}

// Delete a node from the AVL tree
Node* deleteNode(Node* root, int key) {
    // STEP 1: PERFORM STANDARD BST DELETE
    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 {
        // Node with only one child or no child
        if (root->left == NULL) {
```

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

        // Node with two children: Get the inorder successor (smallest in the
right subtree)
        Node* temp = minValueNode(root->right);

        // Copy the inorder successor's content to this node
        root->key = temp->key;

        // Delete the inorder successor
        root->right = deleteNode(root->right, temp->key);
    }

    // STEP 2: UPDATE HEIGHT OF THE CURRENT NODE
    root->height = 1 + max(height(root->left), height(root->right));

    // STEP 3: GET THE BALANCE FACTOR OF THIS NODE
    int balance = getBalance(root);

    // If this node becomes unbalanced, then there are 4 cases
```

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

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

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

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

    // Return the (unchanged) node pointer
    return root;
}

// Utility function to print the AVL tree (Inorder Traversal)
void inorder(Node *root) {
    if (root != NULL) {
        inorder(root->left);
        printf("%d ", root->key);
```

```
        inorder(root->right);
    }
}

// Driver program to test the above functions
int main() {
    Node *root = NULL;

    // Inserting elements
    root = insert(root, 10);
    root = insert(root, 20);
    root = insert(root, 30);
    root = insert(root, 40);
    root = insert(root, 50);
    root = insert(root, 25);

    printf("Inorder traversal of the constructed AVL tree is:\n");
    inorder(root);
    printf("\n");

    // Deleting elements
    root = deleteNode(root, 10);
    root = deleteNode(root, 20);
    root = deleteNode(root, 30);

    printf("Inorder traversal after deletions:\n");
    inorder(root);
    printf("\n");
```

```
    return 0;
}
```

Output:Inorder traversal of the constructed AVL tree is:

10 20 25 30 40 50

Inorder traversal after deletions:

25 40 50

c) Search for a key element in an AVL tree.

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

// Structure for AVL tree node
typedef struct Node {
    int key;
    struct Node *left;
    struct Node *right;
    int height;
} Node;

// Utility function to get the height of the tree
int height(Node *N) {
    if (N == NULL) return 0;
    return N->height;
}

// Utility function to get the maximum of two integers
int max(int a, int b) {
    return (a > b) ? a : b;
}
```

```
// Utility function to create a new node
Node* newNode(int key) {
    Node* node = (Node*) malloc(sizeof(Node));
    node->key = key;
    node->left = NULL;
    node->right = NULL;
    node->height = 1; // New node is initially at height 1
    return node;
}

// Right rotate utility
Node* rightRotate(Node *y) {
    Node *x = y->left;
    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 utility
```

```
Node* leftRotate(Node *x) {
    Node *y = x->right;
    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
int getBalance(Node *N) {
    if (N == NULL) return 0;
    return height(N->left) - height(N->right);
}

// Insert a node into the AVL tree
Node* insert(Node* node, int key) {
    // 1. Perform the normal BST insert
    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 // Duplicate keys are not allowed in the AVL tree
    return node;

// 2. Update height of this ancestor node
node->height = 1 + max(height(node->left), height(node->right));

// 3. Get the balance factor of this ancestor node to check whether
// this node became unbalanced
int balance = getBalance(node);

// If this node becomes unbalanced, then there are 4 cases

// Left Left Case
if (balance > 1 && key < node->left->key)
    return rightRotate(node);

// Right Right Case
if (balance < -1 && key > node->right->key)
    return leftRotate(node);

// Left Right Case
if (balance > 1 && key > node->left->key) {
    node->left = leftRotate(node->left);
    return rightRotate(node);
}

// Right Left Case
```

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

    // Return the (unchanged) node pointer
    return node;
}

// Search for a key in the AVL tree
Node* search(Node* root, int key) {
    // Base Cases: root is null or key is present at root
    if (root == NULL || root->key == key)
        return root;

    // Key is greater than root's key
    if (root->key < key)
        return search(root->right, key);

    // Key is smaller than root's key
    return search(root->left, key);
}

// Utility function to print the AVL tree (Inorder Traversal)
void inorder(Node *root) {
    if (root != NULL) {
        inorder(root->left);
        printf("%d ", root->key);
        inorder(root->right);
```

```
    }
}

// Driver program to test the above functions
int main() {
    Node *root = NULL;

    // Inserting elements
    root = insert(root, 10);
    root = insert(root, 20);
    root = insert(root, 30);
    root = insert(root, 40);
    root = insert(root, 50);
    root = insert(root, 25);

    // Print the AVL tree
    printf("Inorder traversal of the constructed AVL tree is:\n");
    inorder(root);
    printf("\n");

    // Search for keys
    int keys_to_search[] = {10, 25, 30, 100}; // Example keys
    for (int i = 0; i < sizeof(keys_to_search) / sizeof(keys_to_search[0]); i++) {
        Node *result = search(root, keys_to_search[i]);
        if (result != NULL) {
            printf("Key %d found in the AVL tree.\n", keys_to_search[i]);
        } else {
            printf("Key %d not found in the AVL tree.\n", keys_to_search[i]);
        }
```

```
    }

    return 0;
}
```

Output: Inorder traversal of the constructed AVL tree is:

10 20 25 30 40 50

Key 10 found in the AVL tree.

Key 25 found in the AVL tree.

Key 30 found in the AVL tree.

Key 100 not found in the AVL tree.