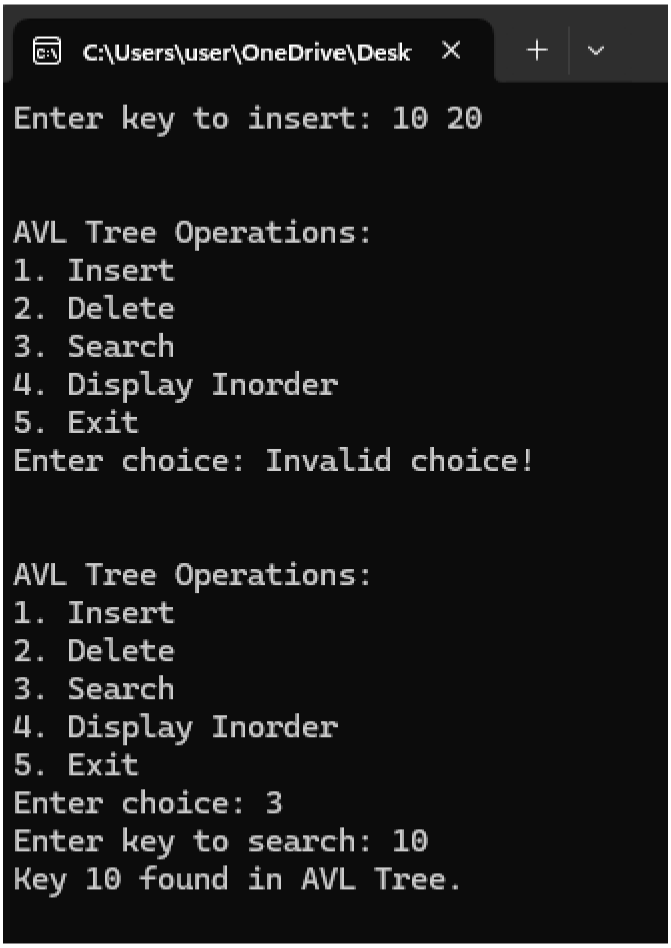
```
#include <stdio.h>
#include <stdlib.h>

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

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

// Get maximum of two integers
int max(int a, int b) {
    return (a > b) ? a : b;
}

// Allocate a new node
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; // New node is initially added at leaf
    return node;
}

// Right rotate
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 x;
}
```

```
// Left rotate
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 y;
}

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

// Insert node
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; // Duplicate keys not allowed

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

    // Balance
    int balance = getBalance(node);

    // 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 node;
}

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

// Delete node
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 {
        // Node with one or no child
        if ((root->left == NULL) || (root->right == NULL)) {
            struct Node *temp = root->left ? root->left : root->right;

            if (temp == NULL) { // No child
                temp = root;
                root = NULL;
            } else { // One child
                *root = *temp;
            }
            free(temp);
        } else {
            // Node with two children
            struct Node *temp = minValueNode(root->right);
            root->key = temp->key;
            root->right = deleteNode(root->right, temp->key);
        }
    }

    if (root == NULL)
        return root;

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

    // Balance
    int balance = getBalance(root);

    // Balance cases
    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;
}

// Search for a key
int search(struct Node *root, int key) {
    if (root == NULL)
        return 0;
    if (key == root->key)
        return 1;
    else if (key < root->key)
        return search(root->left, key);
    else
        return search(root->right, key);
}

// Inorder traversal
void inorder(struct Node *root) {
    if (root != NULL) {
        inorder(root->left);
        printf("%d ", root->key);
        inorder(root->right);
    }
}

// Main function
int main() {
    struct Node *root = NULL;
    int choice, key;

    while (1) {
        printf("\n\nAVL Tree Operations:\n");
        printf("1. Insert\n2. Delete\n3. Search\n4. Display Inorder\n5. Exit\n");
        printf("Enter choice: ");
        scanf("%d", &choice);

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

        case 2:
            printf("Enter key to delete: ");
            scanf("%d", &key);
            root = deleteNode(root, key);
            break;

        case 3:
            printf("Enter key to search: ");
            scanf("%d", &key);
            if (search(root, key))
                printf("Key %d found in AVL Tree.\n", key);
            else
                printf("Key %d not found.\n", key);
            break;

        case 4:
            printf("Inorder traversal: ");
            inorder(root);
            printf("\n");
            break;

        case 5:
            exit(0);
        default:
            printf("Invalid choice!\n");
        }
    }

    return 0;
}
```



```
Enter key to insert: 10 20


AVL Tree Operations:
1. Insert
2. Delete
3. Search
4. Display Inorder
5. Exit
Enter choice: Invalid choice!


AVL Tree Operations:
1. Insert
2. Delete
3. Search
4. Display Inorder
5. Exit
Enter choice: 3
Enter key to search: 10
Key 10 found in AVL Tree.
```