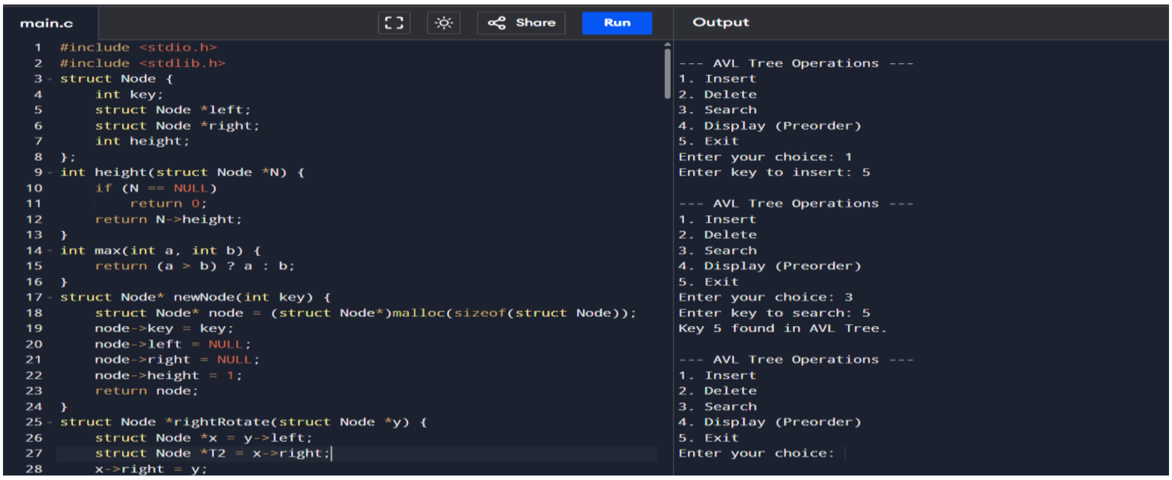
21.Write a program to perform the following operations: a) Insert an element into a AVL tree b) Delete an element from a AVL tree c) Search for a key element in a AVL tree

```
#include <stdio.h>

#include <stdlib.h>

struct Node {

   int key;

   struct Node *left;

   struct Node *right;

   int height;

};

int height(struct Node *N) {

   if (N == NULL)

      return 0;

   return N->height;

}

int max(int a, int b) {

   return (a > b) ? a : b;

}

struct Node* newNode(int key) {

   struct Node* node = (struct Node*)malloc(sizeof(struct Node));

   node->key = key;

   node->left = NULL;

   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; // Duplicate keys not allowed
    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) {
    // 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 {
             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; // Copy contents
            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;
}
int search(struct Node* root, int key) {
    if (root == NULL)
        return 0;
    if (root->key == key)
        return 1;
    else if (key < root->key)
        return search(root->left, key);
    else
        return search(root->right, key);
}
void preOrder(struct Node *root) {
    if (root != NULL) {
        printf("%d ", root->key);
        preOrder(root->left);
        preOrder(root->right);
    }
}
int main() {
    struct Node *root = NULL;
    int choice, key;
    while (1) {
        printf("\n--- AVL Tree Operations ---\n");
        printf("1. Insert\n2. Delete\n3. Search\n4. Display (Preorder)\n5. Exit\n");
        printf("Enter your 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 in AVL Tree.\n", key);
                break;
            case 4:
                printf("Preorder Traversal: ");
                preOrder(root);
                printf("\n");
                break;
            case 5:
                exit(0);
            default:
                printf("Invalid choice!\n");
        }
    }
    return 0;
}
```



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

```
--- AVL Tree Operations ---
1. Insert
2. Delete
3. Search
4. Display (Preorder)
5. Exit
Enter your choice: 1
Enter key to insert: 5

--- AVL Tree Operations ---
1. Insert
2. Delete
3. Search
4. Display (Preorder)
5. Exit
Enter your choice: 3
Enter key to search: 5
Key 5 found in AVL Tree.

--- AVL Tree Operations ---
1. Insert
2. Delete
3. Search
4. Display (Preorder)
5. Exit
Enter your choice:
```