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

// Define the structure for a node in the AVL tree
typedef struct Node {
    int key;
    struct Node* left;
    struct Node* right;
    int height;
} Node;

// Function to calculate the height of a node
int height(Node* node) {
    if (node == NULL)
        return 0;
    return node->height;
}

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

// 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;
    return node;
}

// Function to perform left rotation
Node* leftRotate(Node* z) {
    Node* y = z->right;
    Node* T2 = y->left;
```

```
    y->left = z;
    z->right = T2;

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

    return y;
}

// Function to perform right rotation
Node* rightRotate(Node* y) {
    Node* x = y->left;
    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;
}

// Function to get the balance factor of a node
int getBalance(Node* node) {
    if (node == NULL)
        return 0;
    return height(node->left) - height(node->right);
}

// Function to insert a new node into the AVL tree
Node* insertNode(Node* node, int key) {
    if (node == NULL)
        return newNode(key);

    if (key < node->key)
        node->left = insertNode(node->left, key);
    else if (key > node->key)
```

```
        node->right = insertNode(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;
}

// Function to delete a node from the AVL tree
Node* deleteNode(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) {
```

```
        Node* temp = root->left ? root->left : root->right;
        if (temp == NULL) {
            temp = root;
            root = NULL;
        } else {
            *root = *temp;
        }
        free(temp);
    } else {
        Node* temp = root->right;
        while (temp->left != NULL)
            temp = temp->left;
        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;
}

// Function to search for a key in the AVL tree
Node* searchNode(Node* root, int key) {
    if (root == NULL || root->key == key)
        return root;

    if (root->key < key)
        return searchNode(root->right, key);

    return searchNode(root->left, key);
}

// Function to print the AVL tree in inorder
void printInorder(Node* root) {
    if (root != NULL) {
        printInorder(root->left);
        printf("%d ", root->key);
        printInorder(root->right);
    }
}

int main() {
    Node* root = NULL;

    // Insert elements into the AVL tree
    root = insertNode(root, 10);
    root = insertNode(root, 20);
    root = insertNode(root, 30);
    root = insertNode(root, 40);
    root = insertNode(root, 50);
    root = insertNode(root, 25);

    printf("Inorder traversal of the AVL tree: ");
    printInorder(root);
```

```
    printf("\n");

    // Delete an element from the AVL tree
    root = deleteNode(root, 20);

    printf("Inorder traversal of the AVL tree after deletion: ");
    printInorder(root);
    printf("\n");

    // Search for a key in the AVL tree
    Node* result = searchNode(root, 30);
    if (result != NULL)
        printf("Key found: %d\n", result->key);
    else
        printf("Key not found\n");

    return 0;
}

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

#define MAX 100

int graph[MAX][MAX], visited[MAX];
int queue[MAX], front = -1, rear = -1, vertices;

void BFS(int start) {
    visited[start] = 1;
    queue[++rear] = start;

    while (front != rear) {
        int current = queue[++front];
        printf("%d ", current);

        for (int i = 0; i < vertices; i++) {
            if (graph[current][i] == 1 && !visited[i]) {
                visited[i] = 1;
```

```
            queue[++rear] = i;
        }
    }
  }
}
```