#include <stdio.h>

#include <stdlib.h>

// Node structure

struct Node {

int key;

struct Node\* left;

struct Node\* right;

int height;

};

// Get height of a node

int height(struct Node\* node) {

return node ? node->height : 0;

}

// Get maximum of two integers

int max(int a, int b) {

return (a > b) ? a : b;

}

// Create a new node

struct Node\* createNode(int key) {

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

node->key = key;

node->left = node->right = NULL;

node->height = 1;

return node;

}

// Right rotate

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;

}

// Left rotate

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;

}

// Get balance factor

int getBalance(struct Node\* node) {

return node ? height(node->left) - height(node->right) : 0;

}

// Insert a key

struct Node\* insert(struct Node\* node, int key) {

if (!node)

return createNode(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);

// 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 minimum value node

struct Node\* minValueNode(struct Node\* node) {

struct Node\* current = node;

while (current->left)

current = current->left;

return current;

}

// Delete a key

struct Node\* delete(struct Node\* root, int key) {

if (!root)

return root;

if (key < root->key)

root->left = delete(root->left, key);

else if (key > root->key)

root->right = delete(root->right, key);

else {

if (!root->left || !root->right) {

struct Node\* temp = root->left ? root->left : root->right;

if (!temp) {

temp = root;

root = NULL;

} else

\*root = \*temp;

free(temp);

} else {

struct Node\* temp = minValueNode(root->right);

root->key = temp->key;

root->right = delete(root->right, temp->key);

}

}

if (!root)

return root;

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

int balance = getBalance(root);

// Rebalance tree

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)

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) {

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--- AVL 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 key to insert: ");

scanf("%d", &key);

root = insert(root, key);

break;

case 2:

printf("Enter key to delete: ");

scanf("%d", &key);

root = delete(root, key);

break;

case 3:

printf("Enter key to search: ");

scanf("%d", &key);

if (search(root, key))

printf("Key %d found in the 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:

printf("Exiting program.\n");

return 0;

default:

printf("Invalid choice. Try again.\n");

}

}

}