**21.** **Write a C program to Graph traversal using Breadth First Search**

#include <stdio.h>

#include <stdlib.h>

// AVL Tree Node structure

typedef struct Node {

int key;

struct Node \*left;

struct Node \*right;

int height;

} Node;

// Function to get height of a node

int height(Node \*node) {

if (node == NULL)

return 0;

return node->height;

}

// Function to get maximum of two integers

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;

}

// Right rotate subtree rooted with y

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 x;

}

// Left rotate subtree rooted with x

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 y;

}

// Get balance factor of node

int getBalance(Node \*node) {

if (node == NULL)

return 0;

return height(node->left) - height(node->right);

}

// Insert a key into AVL tree

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

// 1. Perform normal BST insertion

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 not allowed

return node;

// 2. Update height of this ancestor node

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

// 3. Get balance factor to check if unbalanced

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 key value

Node\* minValueNode(Node\* node) {

Node\* current = node;

while (current->left != NULL)

current = current->left;

return current;

}

// Delete a key from AVL tree

Node\* deleteNode(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 {

// Node with only one child or no child

if ((root->left == NULL) || (root->right == NULL)) {

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

// No child case

if (temp == NULL) {

temp = root;

root = NULL;

} else // One child case

\*root = \*temp; // Copy contents

free(temp);

} else {

// Node with two children

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

root->key = temp->key;

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

}

}

// If tree had only one node then return

if (root == NULL)

return root;

// 2. Update height

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

// 3. Get balance factor

int balance = getBalance(root);

// 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 root;

}

// Search for a key in AVL tree

Node\* search(Node\* root, int key) {

if (root == NULL || root->key == key)

return root;

if (root->key < key)

return search(root->right, key);

return search(root->left, key);

}

// Inorder traversal

void inorder(Node \*root) {

if (root != NULL) {

inorder(root->left);

printf("%d ", root->key);

inorder(root->right);

}

}

// Main function

int main() {

Node \*root = NULL;

int choice, key;

Node \*result;

printf("AVL Tree Operations:\n");

printf("1. Insert\n");

printf("2. Delete\n");

printf("3. Search\n");

printf("4. Display (Inorder)\n");

printf("5. Exit\n");

while (1) {

printf("\nEnter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

printf("Enter key to insert: ");

scanf("%d", &key);

root = insert(root, key);

printf("Key %d inserted successfully.\n", key);

break;

case 2:

printf("Enter key to delete: ");

scanf("%d", &key);

root = deleteNode(root, key);

printf("Key %d deleted successfully.\n", key);

break;

case 3:

printf("Enter key to search: ");

scanf("%d", &key);

result = search(root, key);

if (result != NULL)

printf("Key %d found in the tree.\n", key);

else

printf("Key %d not found in the tree.\n", key);

break;

case 4:

printf("Inorder traversal: ");

inorder(root);

printf("\n");

break;

case 5:

exit(0);

default:

printf("Invalid choice! Please try again.\n");

}

}

return 0;

}