**DSA LAB**

**Lab Assignment number 14**

**Name:** Aamir Ansari  **Batch:** A **Roll no:** 01

// code

#include <stdio.h>

#include <stdlib.h>

struct node { // stucture of node

struct node \*left;

int data;

struct node \*right;

int height;

};

// declaring root

struct node \*root = NULL;

struct node \*findMax(struct node \*root) {

while (root->right != NULL) {

root = root->right;

}

return root;

}

struct node \*findMin(struct node \*root) {

while (root->left != NULL) {

root = root->left;

}

return root;

}

int max (int n1, int n2) {

return ((n1 > n2) ? n1 : n2);

}

int height (struct node \*root) {

if (root == NULL) {

return 0;

}

return root->height;

}

struct node \*getNewNode(int data) { // initialises and allocates memory for newNode

struct node \*newNode;

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

newNode->data = data;

newNode->left = NULL;

newNode->right = NULL;

newNode->height = 1;

return newNode;

}

int getBalance(struct node \*root) {

if (root == NULL) {

return 0;

}

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

}

struct node \*rightRotate(struct node \*root) {

struct node \*rootLeft = root->left;

struct node \*rootLeftRight = rootLeft->right;

// rotation

rootLeft->right = root;

root->left = rootLeftRight;

// updation of height

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

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

// back tracking of root

return rootLeft;

}

struct node \*leftRotate(struct node \*root) {

struct node \*rootRight = root->right;

struct node \*rootRightLeft = rootRight->left;

// rotation

rootRight->left = root;

root->right = rootRightLeft;

// updation of height

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

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

// back tracking of root

return rootRight;

}

struct node \*insert (struct node \*root, int data) { // inserts in the avl tree

if (root == NULL) { // base case

root = getNewNode(data);

return root;

}

if (data < root->data) { // insertion in right sub-tree

root->left = insert(root->left, data);

}

else if (data > root->data) { // insertion in left sub-tree

root->right = insert(root->right, data);

}

else { // return root if value is equal

return root;

}

// updating height of ancestor node

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

// balance factor

int balance = getBalance(root);

// ROTATIONS

if ((balance > 1) && (data < root->left->data)) { // LEFT-LEFT

return rightRotate(root);

}

else if ((balance < -1) && (data > root->right->data)) { // RIGHT-RIGHT

return leftRotate(root);

}

else if ((balance > 1) && (data > root->left->data)) { // LEFT-RIGHT

root->left = leftRotate(root->left);

return rightRotate(root);

}

else if ((balance < -1) && (data < root->right->data)) { // RIGHT-LEFT

root->right = rightRotate(root->right);

return leftRotate(root);

}

return root;

}

struct node \*delete(struct node \*root, int val) {

// deletion of node

if (root == NULL) { // empty tree

return root;

}

else if (val < root->data) { // finding node in left sub-tree

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

}

else if (val > root->data) { // finding node in right sub-tree

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

}

else { // found the node

if (root->right == NULL && root->left == NULL) { // deleting leaf node

free(root);

root = NULL;

} else if (root->right == NULL) { // deleting a node with only left sub-tree

struct node \*temp = root;

root = root->left;

free(temp);

} else if (root->left == NULL) { // deleting a node with only right sub-tree

struct node \*temp = root;

root = root->right;

free(temp);

} else { // deleting nodes with two sub-trees

// storing address of node with min value in right sub-tree

struct node \*temp = findMin(root->right);

root->data = temp->data;

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

}

}

// updation of height

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

// check balance factor

int balance = getBalance(root);

// ROTATIONS

if ((balance > 1) && (getBalance(root->left)>=0)) { // LEFT-LEFT

return rightRotate(root);

}

else if ((balance < -1) && (getBalance(root->right)<=0)) { // RIGHT-RIGHT

return leftRotate(root);

}

else if ((balance > 1) && (getBalance(root->left)<0)) { // LEFT-RIGHT

root->left = leftRotate(root->left);

return rightRotate(root);

}

else if ((balance < -1) && (getBalance(root->right)>0)) { // RIGHT-LEFT

root->right = rightRotate(root->right);

return leftRotate(root);

}

return root;

}

void search(struct node \*root, int val) {

if (root->data == val) {

printf("\n%d is present in the tree", val);

return;

}

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

printf("\nNot present");

return;

}

if (val <= root->data) { // search in left sub-tree

search(root->left, val);

}

else { // search in right sub-tree

search(root->right, val);

}

}

int countAllNodes(struct node \*root) {

if (root == NULL) {

return 0;

}

else {

return countAllNodes(root->left) + countAllNodes(root->right) + 1;

}

}

void inOrderTraversal (struct node \*root) {

if (root == NULL) {

return;

}

inOrderTraversal(root->left);

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

inOrderTraversal(root->right);

}

void display(struct node \*root, int space) {

if (root == NULL)

return;

// Increase distance between levels

space += 7;

// Process right child first

display(root->right, space);

// Print current node after space

printf("\n");

for (int i = 5; i < space; i++) {

printf(" ");

}

printf("%d\n", root->data);

// Process left child

display(root->left, space);

}

int main() {

struct node \*temp;

int data, i, choice, val;

while (1) {

printf("\n(1) Insert");

printf("\n(2) Delete");

printf("\n(3) Search");

printf("\n(4) Height");

printf("\n(5) INORDER");

printf("\n(6) TOTAL number of nodes");

printf("\n(7) Display");

printf("\n(8) EXIT");

printf("\nEnter your choice : ");

scanf("%d", &choice);

switch (choice) {

case 1:

printf("\nEnter data to insert : ");

scanf("%d", &data);

root = insert(root, data);

printf("\n%d is inserted!", data);

break;

case 2:

printf("\nEnter a value to delete : ");

scanf("%d", &val);

root = delete (root, val);

printf("\n%d is deleted!", val);

break;

case 3:

printf("\nEnter a number to Search");

scanf("%d", &data);

search(root, data);

break;

case 4:

printf("\nHeight of tree is : %d", height(root));

break;

case 5:

printf("\nIN-ORDER : ");

inOrderTraversal(root);

break;

case 6:

printf("\nTotal number of nodes : %d", countAllNodes(root));

break;

case 7:

display(root, 0);

break;

case 8:

printf("\n\*\*\* E X I T I N G \*\*\*\n");

exit(1);

break;

default:

printf("\n\*\*\* I N V A L I D \*\*\*");

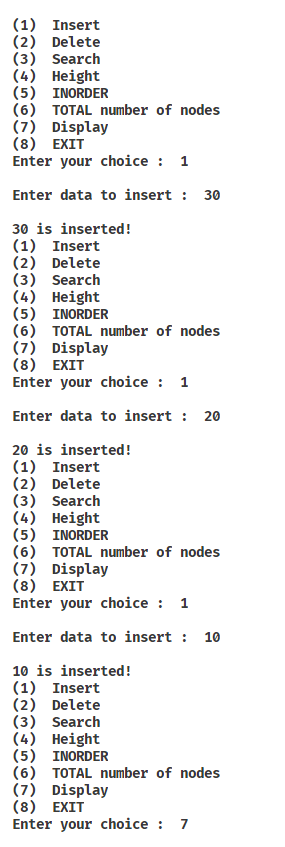
}

}

return 0;

}

// output

.

