**Implementation of Binary Search Tree**

**//** code

#include <stdio.h>

#include <stdlib.h>

// Declaration of node of tree

struct node {

struct node \*left;

int data;

struct node \*right;

};

// declaring root node

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;

}

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;

return newNode;

}

struct node \*insert(struct node \*root, int data) {

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

root = getNewNode(data);

return root;

}

if (data <= root->data) { // inserting in left subtree

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

}

else { // inserting in right subtree

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

}

// returning original root of the tree

return root;

}

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

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);

}

}

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 height(struct node \*root) {

int leftHeight, rightHeight;

if (root == NULL) {

return 0;

}

else {

leftHeight = height(root->left);

rightHeight = height(root->right);

return (leftHeight > rightHeight) ? leftHeight + 1 : rightHeight + 1;

}

}

int countAllNodes(struct node \*root) {

if (root == NULL) {

return 0;

}

else {

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

}

}

int countLeafNodes(struct node \*root) {

if (root == NULL) {

return 0;

}

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

return 1;

}

else {

return countLeafNodes(root->left) + countLeafNodes(root->right);

}

}

int countNonLeafNodes(struct node \*root) {

return (countAllNodes(root) - countLeafNodes(root));

}

void printOneLevel(struct node \*root, int level) { // print elements on given level

if (root == NULL) {

return;

}

if (level == 1) {

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

}

else if (level > 1) {

printOneLevel(root->left, level-1);

printOneLevel(root->right, level-1);

}

}

void printCompleteTree(struct node \*root) { // calls printOneLevel for all the levels in the trr

int h = height(root);

int i;

for (i=1 ; i<=h ; i++) {

printOneLevel(root, i);

printf("\n");

}

}

void mirrorTree(struct node \*root) {

if (root == NULL) {

return;

}

struct node \*temp = root;

// get to all nodes of tree

mirrorTree(root->left);

mirrorTree(root->right);

// swap the pointer

temp = root->left;

root->left = root->right;

root->right = temp;

}

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

if (root != NULL) {

deleteCompleteTree(root->left);

deleteCompleteTree(root->right);

free(root);

}

}

void preOrderTraversal(struct node \*root) {

if (root == NULL) {

return;

}

// print the data of the node

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

// recursion on left sub-tree

preOrderTraversal(root->left);

//recursion on right sub-tree

preOrderTraversal(root->right);

}

void inOrderTraversal(struct node \*root) {

if (root == NULL) {

return;

}

// recursion on left sub-tree

inOrderTraversal(root->left);

// print the data of the node

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

//recursion on right sub-tree

inOrderTraversal(root->right);

}

void postOrderTraversal(struct node \*root) {

if (root == NULL) {

return;

}

// recursion on left sub-tree

postOrderTraversal(root->left);

//recursion on right sub-tree

postOrderTraversal(root->right);

// print the data of the node

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

}

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) PREORDER");

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

printf("\n(8) TOTAL number of nodes");

printf("\n(9) Number of LEAF nodes");

printf("\n(10) Number of NON-LEAF nodes");

printf("\n(11) Find MIN");

printf("\n(12) Find MAX");

printf("\n(13) Display");

printf("\n(14) Mirror");

printf("\n(15) Excise Tree");

printf("\n(16) 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("\nPRE-ORDER : ");

preOrderTraversal(root);

break;

case 7:

printf("\nPOST-ORDER : ");

postOrderTraversal(root);

break;

case 8:

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

break;

case 9:

printf("\nNumber of LEAF nodes : %d", countLeafNodes(root));

break;

case 10:

printf("\nNumber of NON-LEAF nodes : %d", countNonLeafNodes(root));

break;

case 11:

temp = findMin(root);

printf("\nMINIMUM in tree : %d", temp->data);

break;

case 12:

temp = findMax(root);

printf("\nMAXIMUM in tree : %d", temp->data);

break;

case 13:

printf("\n\*\*\*TREE\*\*\*\n");

printCompleteTree(root);

break;

case 14:

printf("\n\*\*\*MIRROR\*\*\*\n");

mirrorTree(root);

printCompleteTree(root);

break;

case 15:

deleteCompleteTree(root);

printf("\nEntire tree is deleted! you happy now, huh?");

break;

case 16:

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

exit(1);

break;

default:

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

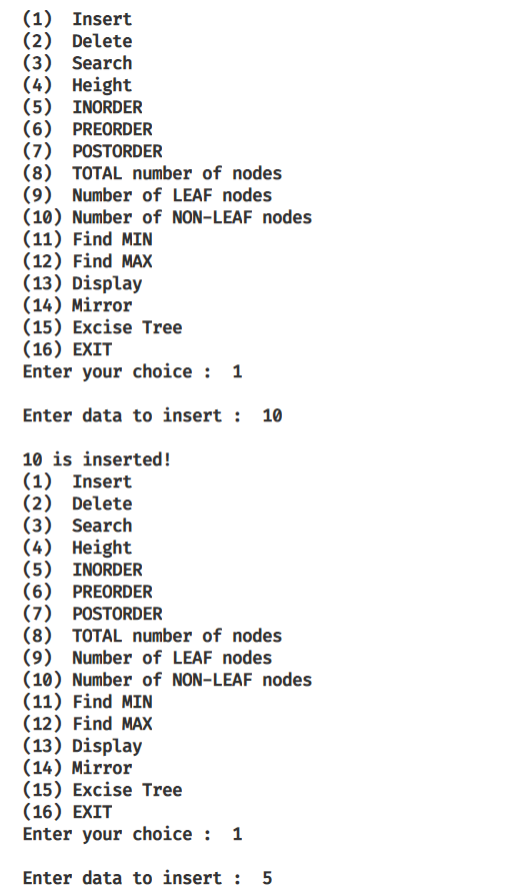
}

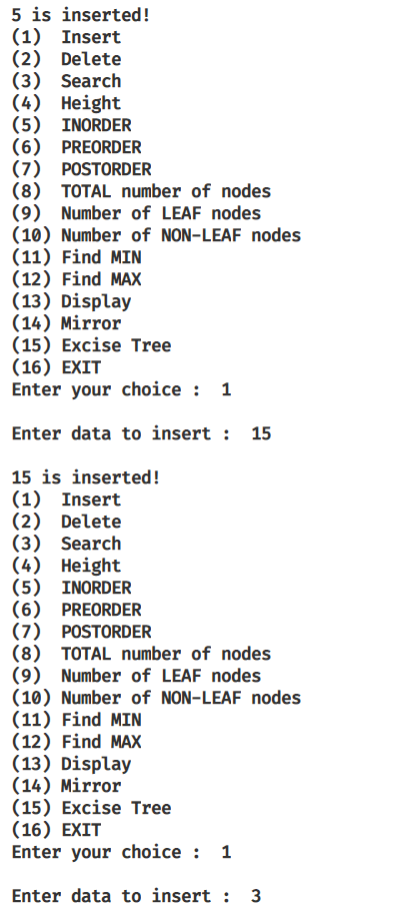
}

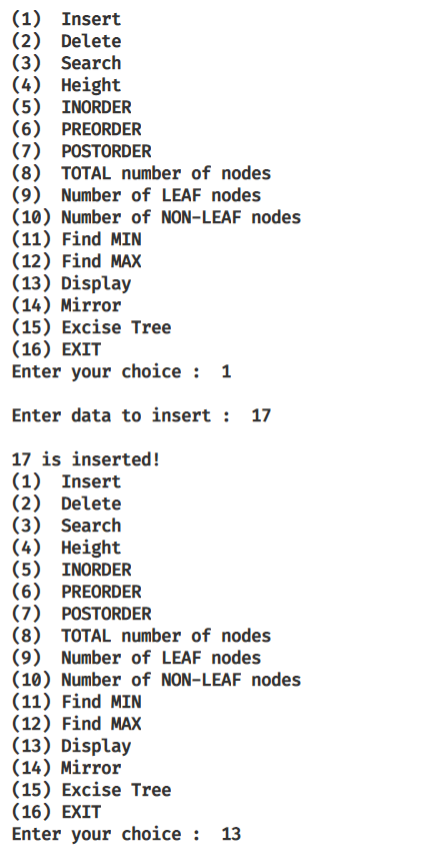
return 0;

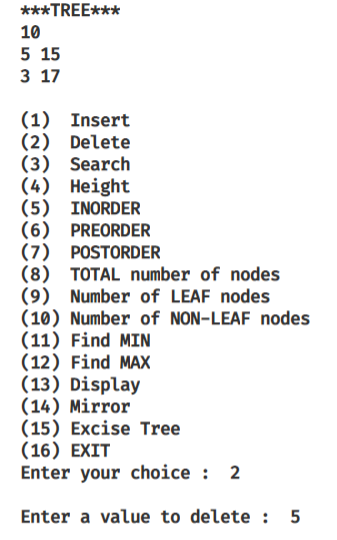
}

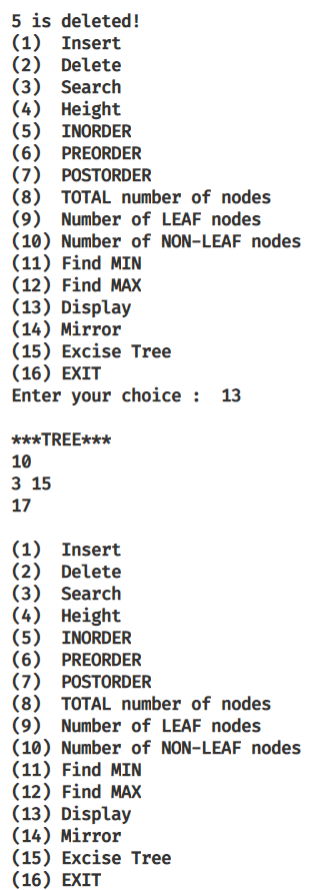
// output

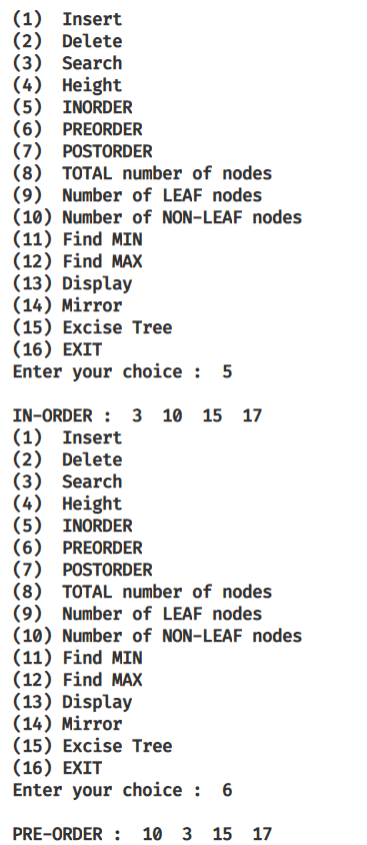
.

.

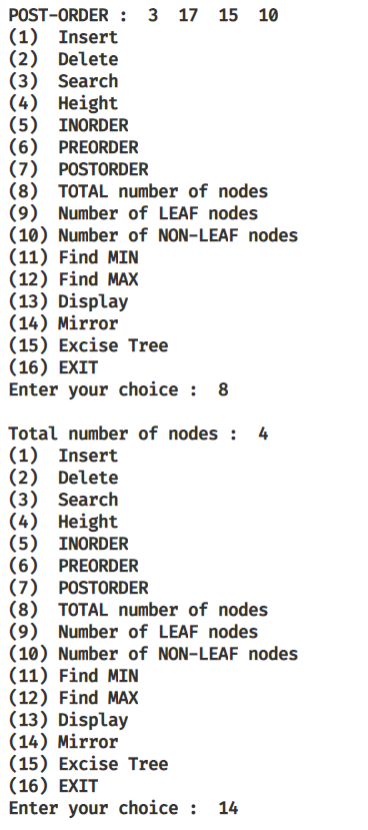
.

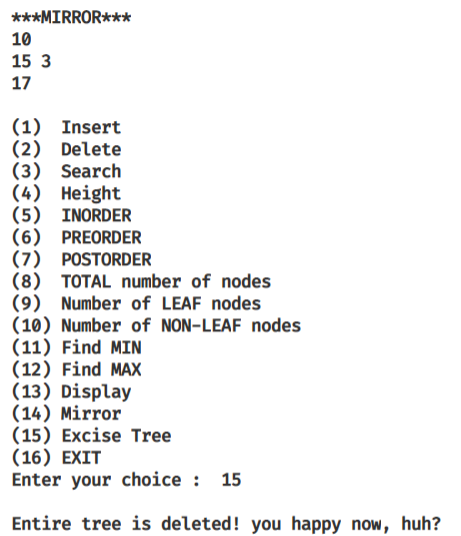
.

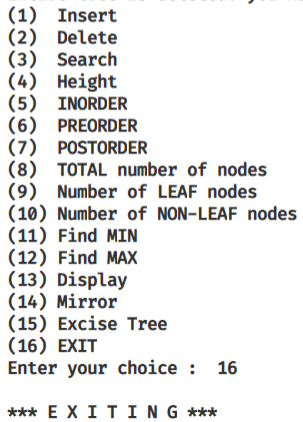
****

****

.

****

****

****