**DS LAB 12: Binary Search Tree**

Name: Vivian VIjay Ludrick

Roll no: 9914

Branch: SE Comps A Batch C

Code:

*// after mirroring search doesn't work*

#include <stdio.h>

#include <stdlib.h>

typedef struct node

{

int data;

struct node \*left;

struct node \*right;

} Node;

typedef struct tree

{

Node \*root;

} Tree;

*// this function returns the address of the new node that is created*

Node \***create\_node**(int *data*)

{

Node \*new\_node = (Node \*)**malloc**(sizeof(Node));

new\_node->data = *data*;

new\_node->left = **NULL**;

new\_node->right = **NULL**;

return new\_node;

}

*// tranverse till the leftmost node is found as it is the node with minimum value*

Node \***find\_min**(Node \**node*)

{

while (*node*->left != **NULL**)

{

*node* = *node*->left;

}

return *node*;

}

*// tranverse till the leftmost node is found as it is the node with minimum value*

Node \***find\_max**(Node \**node*)

{

while (*node*->right != **NULL**)

{

*node* = *node*->right;

}

return *node*;

}

*// insert a new node into an existing binary search tree*

void **insert\_node**(Tree \**t*, int *data*)

{

Node \*new\_node = **create\_node**(*data*);

if (*t*->root == **NULL**)

{

*t*->root = new\_node;

return;

}

Node \*current = *t*->root;

while (1)

{

if (*data* < current->data)

{

if (current->left == **NULL**)

{

current->left = new\_node;

return;

}

current = current->left;

}

else

{

if (current->right == **NULL**)

{

current->right = new\_node;

return;

}

current = current->right;

}

}

}

*// LVR transversal in BST. Gives ascending order of the node->data*

void **Inorder\_traversal**(Node \**node*)

{

if (*node* == **NULL**)

{

return;

}

**Inorder\_traversal**(*node*->left);

**printf**("%d ", *node*->data);

**Inorder\_traversal**(*node*->right);

}

*// VLR transversal in BST.*

void **Preorder\_traversal**(Node \**node*)

{

if (*node* == **NULL**)

{

return;

}

**printf**("%d ", *node*->data);

**Preorder\_traversal**(*node*->left);

**Preorder\_traversal**(*node*->right);

}

*// LRV traversal in BST*

void **Postorder\_traversal**(Node \**node*)

{

if (*node* == **NULL**)

{

return;

}

**Postorder\_traversal**(*node*->left);

**Postorder\_traversal**(*node*->right);

**printf**("%d ", *node*->data);

}

*// search for a node->data in BST*

void **search**(Node \**root*, int *data*)

{

Node \*current = *root*;

while (current != **NULL**)

{

*// if the data is found print it and return to close the function*

if (*data* == current->data)

{

**printf**("Data %d is found\n", *data*);

return;

}

else if (*data* < current->data)

{

current = current->left;

}

else

{

current = current->right;

}

}

*// if the data is not found i.e the while loop reaches the leaf node then the while loop will end*

**printf**("Data %d is not found\n", *data*);

}

*// check if the node->data exists recursively*

int **searchRecursive**(Node \**current*, int *data*)

{

if (*current* == **NULL**)

{

**printf**("Data %d not found\n", *data*);

return 0;

}

if (*data* == *current*->data)

{

**printf**("Data %d is found\n", *data*);

return 1;

}

if (*data* < *current*->data)

{

return **searchRecursive**(*current*->left, *data*);

}

else

{

return **searchRecursive**(*current*->right, *data*);

}

}

*// count the number of node recursively. return 1 for each node found and add it to the total number of left and right nodes*

int **countNodesRecursive**(Node \**current*)

{

if (*current* == **NULL**)

{

return 0;

}

*// Count the current node and recursively count nodes in left and right subtrees*

int leftCount = **countNodesRecursive**(*current*->left);

int rightCount = **countNodesRecursive**(*current*->right);

return 1 + leftCount + rightCount;

}

*// count the number of leaves recursively. first count all the left nodes then the right node at the end add all the node recursively at each step to get the final count*

int **countLeafNodesRecursive**(Node \**current*)

{

if (*current* == **NULL**)

{

return 0;

}

if (*current*->left == **NULL** && *current*->right == **NULL**)

{

*// The current node is a leaf node*

return 1;

}

*// Recursively count leaf nodes in left and right subtrees*

int leftCount = **countLeafNodesRecursive**(*current*->left);

int rightCount = **countLeafNodesRecursive**(*current*->right);

return leftCount + rightCount;

}

*// recursivly calculates the height of the BST*

int **calculateHeightRecursive**(Node \**current*)

{

if (*current* == **NULL**)

{

return 0;

}

*// Recursively calculate the height of the left and right subtrees*

int leftHeight = **calculateHeightRecursive**(*current*->left);

int rightHeight = **calculateHeightRecursive**(*current*->right);

*// The height of the tree is the maximum of the left and right subtree heights, plus 1 for the current node.*

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

}

*// switch the left and right nodes at each level with top-> bottom approach.*

void **mirrorTreeRecursive**(Node \**current*)

{

if (*current* == **NULL**)

{

return;

}

*// Swap the left and right subtrees*

Node \*temp = *current*->left;

*current*->left = *current*->right;

*current*->right = temp;

*// Recursively mirror the left and right subtrees*

**mirrorTreeRecursive**(*current*->left);

**mirrorTreeRecursive**(*current*->right);

}

*// delete the node based on the three case it can be*

Node \***deleteNodeNonRecursive**(Node \**root*, int *key*)

{

Node \*current = *root*;

Node \*parent = **NULL**;

*// Search for the node to delete*

while (current != **NULL** && current->data != *key*)

{

parent = current;

if (*key* < current->data)

{

current = current->left;

}

else

{

current = current->right;

}

}

*// If the node is not found, return the original root*

if (current == **NULL**)

{

return *root*;

}

*// Handle three cases for deletion*

*// Case 1: Node with no child*

if (current->left == **NULL** && current->right == **NULL**)

{

if (parent == **NULL**)

{

**free**(current);

return **NULL**; *// Root node is deleted*

}

else if (parent->left == current)

{

parent->left = **NULL**;

}

else

{

parent->right = **NULL**;

}

**free**(current);

}

*// Case 2: Node with one child*

else if (current->left == **NULL**)

{

Node \*temp = current->right;

if (parent == **NULL**)

{

**free**(current);

return temp;

}

if (parent->left == current)

{

parent->left = temp;

}

else

{

parent->right = temp;

}

**free**(current);

}

else if (current->right == **NULL**)

{

Node \*temp = current->left;

if (parent == **NULL**)

{

**free**(current);

return temp;

}

if (parent->left == current)

{

parent->left = temp;

}

else

{

parent->right = temp;

}

**free**(current);

}

*// Case 3: Node with two children*

else

{

Node \*successor = **find\_min**(current->right);

int successorData = successor->data;

**deleteNodeNonRecursive**(*root*, successorData); *// Recursively delete the successor*

current->data = successorData; *// Copy the successor data to the current node*

}

return *root*;

}

int **main**()

{

Tree t;

t.root = **NULL**;

Node \*temp, \*min\_node, \*max\_node;

int choice;

int data;

int del;

while (1)

{

**printf**("\nBinary Tree Menu:\n");

**printf**("1. Insert\t\t");

**printf**("2. Delete\t");

**printf**("3. Inorder\t");

**printf**("4. Preorder\n");

**printf**("5. Postorder\t\t");

**printf**("6. min and max\t");

**printf**("7. Search\t");

**printf**("8. Count nodes\n");

**printf**("9. Count leaf nodes\t");

**printf**("10. height\t");

**printf**("11. Mirror\t");

**printf**("0. Exit\n");

**printf**("Enter your choice: ");

**scanf**("%d", &choice);

switch (choice)

{

case 1:

**printf**("Enter the value to insert: ");

**scanf**("%d", &*data*);

**insert\_node**(&*t*, data);

break;

case 2:

**printf**("Enter the data to be deleted: ");

**scanf**("%d", &*del*);

t.root = **deleteNodeNonRecursive**(t.root, del); *// Update the root with the result of the deletion*

**printf**("Tree after deletion: ");

**Inorder\_traversal**(t.root);

break;

case 3:

**printf**("Inorder traversal: ");

**Inorder\_traversal**(t.root);

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

break;

case 4:

**printf**("Preorder traversal: ");

**Preorder\_traversal**(t.root);

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

break;

case 5:

**printf**("Postorder traversal: ");

**Postorder\_traversal**(t.root);

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

break;

case 6:

*// Find minimum and maximum values here*

min\_node = **find\_min**(t.root);

max\_node = **find\_max**(t.root);

*// when we reverse the minimum and the maximum reverse. So to get consistent result this logic is necessary*

if(min\_node > max\_node){

temp = min\_node;

min\_node = max\_node;

max\_node = temp;

}

**printf**("Minimum value in the tree: %d\n", min\_node->*data*);

**printf**("Maximum value in the tree: %d\n", max\_node->*data*);

break;

case 7:

*// Search for a value here*

**printf**("Enter the value to search for: ");

**scanf**("%d", &*data*);

**search**(t.root, data);

**searchRecursive**(t.root, data);

**printf**("It prints twice cause we search recursively as well as non recursively.\n");

break;

case 8:

**printf**("The number of nodes is %d\n", **countNodesRecursive**(t.root));

break;

case 9:

**printf**("The number of leaf nodes is %d\n", **countLeafNodesRecursive**(t.root));

break;

case 10:

**printf**("The height of the tree is %d\n", **calculateHeightRecursive**(t.root));

break;

case 11:

**mirrorTreeRecursive**(t.root);

**printf**("After mirroring: ");

**Inorder\_traversal**(t.root);

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

break;

case 0:

**printf**("Exiting the program.\n");

return 0;

default:

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

break;

}

}

return 0;

}

OUTPUT:



