## DSAL ASSIGNMENT 5

# BINARY SEARCH TREE

Problem Statement: Implement binary search tree and perform following operations:

a) Insert (Handle insertion of duplicate entry)

b) Delete

c) Search

d) Display tree (Traversal)

e) Display - Depth of tree

f) Display - Mirror image

g) Create a copy

h) Display all parent nodes with their child nodes

i) Display leaf nodes

j) Display tree level wise

Program :

File : bst.h

#pragma once

class node {

public:

int info;

node\* left;

node\* right;

friend class binarytree;

node();

};

class bst {

public:

node\* root;

bst();

node\* getroot();

node\* insert(node\* leaf, int z);

node\* search(node\* leaf, int a);

void postorder\_traversal(node\* leaf);

void inorder\_traversal(node\* leaf);

void preorder\_traversal(node\* leaf);

node\* Delete(node\* leaf, int a);

int disp\_depth(node\* a);

void mirror(node\* a);

void parent\_child(node\* leaf);

void disp\_leaf(node\* leaf);

void levelwise(node\* leaf, int i);

void level(node\* leaf);

};

File : bst.cpp

#include "bst.h"

#include <iostream>

using namespace std;

node::node() {

info = 0;

left = NULL;

right = NULL;

}

bst::bst() {

root = NULL;

}

node\* bst::getroot() {

return root;

}

node\* bst::insert(node\* root, int z) {

if (!root){

node\* new1 = new node();

new1->info = z;

new1->left = NULL;

new1->right = NULL;

return new1;

}

if (z > root->info){

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

}

else{

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

}

return root;

}

void bst::inorder\_traversal(node\* leaf) {

if (leaf != NULL) {

inorder\_traversal(leaf->left);

cout << leaf->info << " ";

inorder\_traversal(leaf->right);

}

}

void bst::preorder\_traversal(node\* leaf) {

if (leaf != NULL) {

cout << leaf->info << " ";

preorder\_traversal(leaf->left);

preorder\_traversal(leaf->right);

}

}

void bst::postorder\_traversal(node\* leaf) {

if (leaf != NULL) {

postorder\_traversal(leaf->left);

postorder\_traversal(leaf->right);

cout << leaf->info << " ";

}

}

node\* bst::search(node\* leaf, int a) {

node\* index;

node\* parent = NULL;

index = root;

while (index != NULL) {

if (a == index->info) {

return index;

}

else if (a < index->info) {

parent = index;

index = index->left;

}

else if (a > index->info) {

parent = index;

index = index->right;

}

}

return NULL;

}

int bst::disp\_depth(node\* a) {

if (a == NULL) {

return 0;

}

else {

int l = disp\_depth(a->left);

int r = disp\_depth(a->right);

if (l > r) {

return(l + 1);

}

else {

return(r + 1);

}

}

}

void bst :: mirror(node\* a) {

if (a != NULL) {

mirror(a->right);

cout << a->info << endl;

mirror(a->left);

}

}

void bst :: disp\_leaf(node\* root) {

if (!root) {

cout << "Empty Tree" << endl;

}

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

cout << root->info << " ";

}

if (root->left) {

disp\_leaf(root->left);

}

if (root->right) {

disp\_leaf(root->right);

}

}

void bst::level(node\* root) {

int h = disp\_depth(root);

int i;

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

levelwise(root, i);

}

}

void bst::levelwise(node\* root, int level) {

if (root == NULL) {

return;

}

if (level == 1) {

cout << root->info << " ";

}

else if (level > 1) {

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

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

}

}

void bst::parent\_child(node\* leaf) {

if (leaf == NULL) {

cout << "Empty Tree" << endl;

}

else {

if (leaf->left == NULL && leaf->right == NULL) {

cout << endl;

cout << "Parent node : " << leaf->info << endl;

cout << "Child nodes : No child nodes present" << endl;

}

else {

cout << endl;

cout << "Parent node : " << leaf->info << endl;

if (leaf->left != NULL) {

cout << "Left child : " << leaf->left->info << endl;

}

else {

cout << "No left child." << endl;

}

if (leaf->right != NULL) {

cout << "Right child : " << leaf->right->info << endl;

}

else {

cout << "No right child." << endl;

}

parent\_child(leaf->left);

parent\_child(leaf->right);

cout << endl;

}

}

}

node\* bst::Delete(node\* leaf, int a) {

if (leaf == NULL) {

cout << "TREE NOT EXIST" << endl;

return leaf;

}

if (a < leaf->info) {

leaf->left = Delete(leaf->left, a);

}

else if (a > leaf->info) {

leaf->right = Delete(leaf->right, a);

}

else {

if (leaf->left == NULL) {

node\* temp = leaf->right;

delete leaf;

return temp;

}

else if (leaf->right == NULL) {

node\* temp = leaf->left;

delete leaf;

return temp;

}

node\* temp;

temp = leaf->right;

while (temp->left != NULL) {

temp = temp->left;

}

leaf->info = temp->info;

leaf->right = Delete(leaf->right, temp->info);

}

return leaf;

}

File : bstmain.cpp

#include "bst.cpp"

#include <iostream>

using namespace std;

int main()

{

bst b;

int x = 0;

int a, c, depth, d;

node\* s;

while (x != 12) {

cout << "==============================" << endl;

cout << "Choose : " << endl;

cout << "1)Insert \n2)Inorder Traversal \n3)Preorder Traversal \n4)Postorder Travesal \n5)Delete \n6)Dispaly Depth" << endl;

cout << "7)Mirror Traversal \n8)Display parent child node pairs \n9)Display Leaf Nodes \n10)Levelwise Traversal" << endl;

cout << "11)Search \n12)Exit" << endl;

cout << "==============================" << endl;

cin >> x;

switch (x) {

case 1:

cout << "Enter value to be inserted : ";

cin >> a;

b.root = b.insert(b.getroot(), a);

cout << "==============================" << endl;

cout << " Node inserted." << endl;

cout << "==============================" << endl;

break;

case 2:

cout << "==============================" << endl;

cout << "Inorder Travesal : ";

b.inorder\_traversal(b.getroot());

cout << endl;

cout << "==============================" << endl;

break;

case 3:

cout << "==============================" << endl;

cout << "Preorder Traversal : ";

b.preorder\_traversal(b.getroot());

cout << endl;

cout << "==============================" << endl;

break;

case 4:

cout << "==============================" << endl;

cout << "Postorder Traversal : ";

b.postorder\_traversal(b.getroot());

cout << endl;

cout << "==============================" << endl;

break;

case 5:

cout << "Enter value to be deleted : " ;

cin >> c;

b.Delete(b.getroot(), c);

cout << "==============================" << endl;

cout << " Node Deleted." << endl;

cout << "==============================" << endl;

break;

case 6:

depth = b.disp\_depth(b.getroot());

cout << "==============================" << endl;

cout << "Depth : " << depth << endl;

cout << "==============================" << endl;

break;

case 7:

cout << "==============================" << endl;

cout << "Mirror Tree : " << endl;

b.mirror(b.getroot());

cout << endl;

cout << "==============================" << endl;

break;

case 8:

cout << "==============================" << endl;

b.parent\_child(b.getroot());

cout << "==============================" << endl;

break;

case 9:

cout << "==============================" << endl;

cout << "Leaf Nodes : ";

b.disp\_leaf(b.getroot());

cout << endl;

cout << "==============================" << endl;

break;

case 10:

cout << "==============================" << endl;

cout << "Levelwise Traversal : ";

b.level(b.getroot());

cout << endl;

cout << "==============================" << endl;

break;

case 11:

cout << "Enter the value to be searched : " << endl;

cin >> d;

s = b.search(b.getroot(), d);

cout << "==============================" << endl;

cout << "Parent node of node to be found : " << s->info << endl;

cout << "==============================" << endl;

break;

case 12:

break;

default:

cout << "Wrong input." << endl;

break;

}

}

}