**Given the root of a Binary Search Tree (BST), convert it to a Greater Tree such that every key of the original BST is changed to the original key plus the sum of all keys greater than the original key in BST. As a reminder, a binary search tree is a tree that satisfies these constraints: • The left subtree of a node contains only nodes with keys less than the node's key. • The right subtree of a node contains only nodes with keys greater than the node's key. • Both the left and right subtrees must also be binary search trees. The above output shows a level order traversal of BST. Prototype: TreeNode\* bstToGst(TreeNode\* root) { } Note: Your program should be menu based, and only exit on pressing 4 • Press 1 to insert values to the tree • Press 2 to display the inorder/preorder/post order/level order traversal of tree (You are free to choose one form of traversal). • Press 3 to generate greater sum tree. • Press 4 to exit**

#include <iostream>

using namespace std;

struct TreeNode {

int val;

TreeNode\* left;

TreeNode\* right;

TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}

};

class BST {

private:

TreeNode\* root;

TreeNode\* insert(TreeNode\* node, int val) {

if (!node) return new TreeNode(val);

if (val < node->val) {

node->left = insert(node->left, val);

}

else {

node->right = insert(node->right, val);

}

return node;

}

void inorderTraversal(TreeNode\* node) {

if (!node) return;

inorderTraversal(node->left);

cout << node->val << " ";

inorderTraversal(node->right);

}

int convertToGreaterTree(TreeNode\* node, int& sum) {

if (!node) return sum;

convertToGreaterTree(node->right, sum);

int currentValue = node->val;

node->val += sum; // Update current node value with sum of greater values

sum += currentValue; // Update sum to include current node's original value

convertToGreaterTree(node->left, sum);

return sum;

}

public:

BST() : root(nullptr) {}

void insert(int val) {

root = insert(root, val);

}

void display() {

cout << "Inorder Traversal: ";

inorderTraversal(root);

cout << endl;

}

void bstToGst() {

int sum = 0;

convertToGreaterTree(root, sum);

}

};

int main() {

BST tree;

int choice, value;

while (true) {

cout << "Menu:\n";

cout << "1. Insert value into the tree\n";

cout << "2. Display inorder traversal of the tree\n";

cout << "3. Generate Greater Tree\n";

cout << "4. Exit\n";

cout << "Enter your choice: ";

cin >> choice;

switch (choice) {

case 1:

cout << "Enter value to insert: ";

cin >> value;

tree.insert(value);

break;

case 2:

tree.display();

break;

case 3:

tree.bstToGst();

cout << "Greater Tree generated.\n";

break;

case 4:

cout << "Exiting program.\n";

return 0;

default:

cout << "Invalid choice. Please try again.\n";

}

}

return 0;

}

**Your task is to implement BST through recursion. Your BST class should have following functionalities: 1. Insert Element 2. Create Mirror Restriction: All these functions MUST be recursive. • InsertElement: It should insert elements according to rules of BST. • CreateMirror: It should convert BST into its mirror BST. 20 Marks 20 Marks After implementing BST class, write the following menu-based driver program and test all the options on the tree given in the image below (Note: all codes must be generic and should work for values other than the ones shown in the image below).**

#include <iostream>

using namespace std;

struct TreeNode {

int data;

TreeNode\* left;

TreeNode\* right;

TreeNode(int value) : data(value), left(nullptr), right(nullptr) {}

};

class BST {

private:

TreeNode\* root;

TreeNode\* insertElement(TreeNode\* node, int value) {

if (!node) {

return new TreeNode(value);

}

if (value < node->data) {

node->left = insertElement(node->left, value);

}

else {

node->right = insertElement(node->right, value);

}

return node;

}

void createMirror(TreeNode\* node) {

if (!node) return;

swap(node->left, node->right);

createMirror(node->left);

createMirror(node->right);

}

void inorderTraversal(TreeNode\* node) {

if (!node) return;

inorderTraversal(node->left);

cout << node->data << " ";

inorderTraversal(node->right);

}

public:

BST() : root(nullptr) {}

void insert(int value) {

root = insertElement(root, value);

}

void mirror() {

createMirror(root);

}

void displayInOrder() {

inorderTraversal(root);

cout << endl;

}

};

int main() {

BST bst;

// Insert values into the BST

bst.insert(10);

bst.insert(5);

bst.insert(15);

bst.insert(3);

bst.insert(7);

bst.insert(12);

bst.insert(18);

// Display in-order traversal of the original BST

cout << "Original In-Order Traversal: ";

bst.displayInOrder();

// Create the mirror of the BST

bst.mirror();

cout << "In-Order Traversal after Creating Mirror: ";

bst.displayInOrder();

return 0;

}

**Some Important Functions**void insert(int value, node\*& root) {

if (root == nullptr) {

root = new node(value);

return;

}

if (value > root->data) {

insert(value, root->right);

}

else if (value < root->data) {

insert(value, root->left);

}

}

Height

int height(node\* root) {

if (root == nullptr) {

return 0;

}

int l = height(root->left);

int r = height(root->right);

return (max(l, r) + 1);

}

Identical

bool identical(node\* root1, node\* root2) {

if (root1 == nullptr && root2 == nullptr) {

return true;

}

else if (root1 == nullptr || root2 == nullptr || root1->data != root2->data) {

return false;

}

return identical(root1->left, root2->left) &&

identical(root1->right, root2->right);

}

Mirror

bool mirror(node\* root1, node\* root2) {

if (root1 == nullptr && root2 == nullptr) {

return true;

}

else if (root1 == nullptr || root2 == nullptr || root1->data != root2->data) {

return false;

}

return mirror(root1->left, root2->right) &&

mirror(root1->right, root2->left);

}

Symmetry or Itself Mirror

bool symetric(node \* root) {

return mirror(root, root);

}

Diameter

int Height(node\* root, int& diam) {

if (root == nullptr) {

return 0;

}

int l = Height(root->left, diam);

int r = Height(root->right, diam);

diam = max(diam, l + r + 1);

return (max(l, r) + 1);

}

int diameter(node\* root) {

int d = 0;

Height(root, d);

return d;

}

Balanced

bool balanced(node\* root) {

if (root == nullptr) {

return true;

}

int h1 = height(root->left);

int h2 = height(root->right);

if (abs(h1 - h2) <= 1 && balanced(root->left) && balanced(root->right)) {

return true;

}

return false;

}

[11:56 PM, 9 / 23 / 2024] M.Ali UCP : DELETE

void deleteroot(node\*& root, int x) {

if (root == nullptr) {

cout << "empty ";

return;

}

if (x < root->data) {

deleteroot(root->left, x);

}

else if (x > root->data) {

deleteroot(root->right, x);

}

else if (x == root->data) {

if (root->left == nullptr && root->right == nullptr) {

delete root;

root = nullptr;

}

else if (root->left == nullptr) {

node\* temp = root->right;

delete root;

root = temp;

}

else if (root->right == nullptr) {

node\* temp = root->left;

delete root;

root = temp;

}

else {

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

root->data = temp->data;

deleteroot(root->right, temp->data);

}

}