NAME: Manasvi Mahadev Patil.

Branch: I.T.

Division: S. Y. - C

Roll no.:03

**AIM:-**

Implement the program in C++ to create a Binary Search Tree, insert elements of user choice, and provide functionalities to find height, count nodes, leaf nodes, minimum and maximum nodes, mirror image, print leaf nodes, and print nodes level-wise:

```cpp

#include <iostream>

#include <queue>

using namespace std;

// Binary Tree Node struct and class definitions go here...

int main() {

// Code for creating the Binary Search Tree and implementing functionalities...

return 0;

}

```.

1. A Binary Search Tree (BST) is a binary tree where each node has at most two children - left child and right child. The left child's value is less than the node's value, and the right child's value is greater, enabling efficient search, insertion, and deletion.

2. A node in a binary search tree contains data and pointers to its left and right children, forming the basic building block.

3. The left child is the node appearing to the left of its parent in the binary tree.

4. The right child is the node appearing to the right of its parent in the binary tree.

5. The root is the topmost node, serving as the tree's starting point, and there is only one root node.

6. Insertion involves adding a new node with a given value while preserving the BST property.

7. Height is the binary tree's maximum depth, measured by the longest path from the root to any leaf node.

8. Count Nodes represents the total number of nodes in the tree, including internal and leaf nodes.

9. A leaf node has no children, indicated by both its left and right child pointers being NULL.

10. The minimum node in a BST is the one with the smallest value, located at the leftmost end of the tree. The maximum node holds the largest value and is at the rightmost end.

11. Printing Leaf Nodes displays the values of all nodes without children (leaf nodes).

12. Print Nodes Level-wise involves displaying the node values level by level, starting from the root, utilizing a queue for traversal.

13. Delete Tree refers to deallocating memory and freeing resources for the entire tree to prevent memory leaks.

**CODE:-**

**CODE:**

**#include <stdio.h>**

**#include <stdlib.h>**

**#include <stdbool.h>**

**struct TreeNode {**

**int data;**

**struct TreeNode\* left;**

**struct TreeNode\* right;**

**};**

**struct TreeNode\* createNode(int val) {**

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

**newNode->data = val;**

**newNode->left = NULL;**

**newNode->right = NULL;**

**return newNode;**

**}**

**struct TreeNode\* insert(struct TreeNode\* root, int val) {**

**if (root == NULL) {**

**return createNode(val);**

**}**

**if (val < root->data) {**

**root->left = insert(root->left, val);**

**} else {**

**root->right = insert(root->right, val);**

**}**

**return root;**

**}**

**int height(struct TreeNode\* root) {**

**if (root == NULL) {**

**return -1;**

**}**

**int leftHeight = height(root->left);**

**int rightHeight = height(root->right);**

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

**}**

**int countNodes(struct TreeNode\* root) {**

**if (root == NULL) {**

**return 0;**

**}**

**return countNodes(root->left) + countNodes(root->right) + 1;**

**}**

**int countLeafNodes(struct TreeNode\* root) {**

**if (root == NULL) {**

**return 0;**

**}**

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

**return 1;**

**}**

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

**}**

**int findMin(struct TreeNode\* root) {**

**if (root == NULL) {**

**printf("Error: Tree is empty\n");**

**return -1;**

**}**

**while (root->left != NULL) {**

**root = root->left;**

**}**

**return root->data;**

**}**

**int findMax(struct TreeNode\* root) {**

**if (root == NULL) {**

**printf("Error: Tree is empty\n");**

**return -1;**

**}**

**while (root->right != NULL) {**

**root = root->right;**

**}**

**return root->data;**

**}**

**void printLeafNodes(struct TreeNode\* root) {**

**if (root == NULL) {**

**return;**

**}**

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

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

**}**

**printLeafNodes(root->left);**

**printLeafNodes(root->right);**

**}**

**void printLevelWise(struct TreeNode\* root) {**

**if (root == NULL) {**

**return;**

**}**

**struct TreeNode\* queue[1000];**

**int front = 0, rear = 0;**

**queue[rear++] = root;**

**while (front < rear) {**

**struct TreeNode\* currNode = queue[front++];**

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

**if (currNode->left != NULL) {**

**queue[rear++] = currNode->left;**

**}**

**if (currNode->right != NULL) {**

**queue[rear++] = currNode->right;**

**}**

**}**

**}**

**void deleteTree(struct TreeNode\* root) {**

**if (root == NULL) {**

**return;**

**}**

**deleteTree(root->left);**

**deleteTree(root->right);**

**free(root);**

**}**

**int main() {**

**struct TreeNode\* root = NULL;**

**int n;**

**printf("Enter the number of elements to insert: ");**

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

**printf("Enter the elements:\n");**

**for (int i = 0; i < n; i++) {**

**int element;**

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

**root = insert(root, element);**

**}**

**int h = height(root);**

**printf("Height of the BST: %d\n", h);**

**int numNodes = countNodes(root);**

**printf("Number of nodes in the BST: %d\n", numNodes);**

**int numLeafNodes = countLeafNodes(root);**

**printf("Number of leaf nodes in the BST: %d\n", numLeafNodes);**

**int minNode = findMin(root);**

**int maxNode = findMax(root);**

**printf("Minimum node value: %d\n", minNode);**

**printf("Maximum node value: %d\n", maxNode);**

**printf("Leaf nodes of the BST: ");**

**printLeafNodes(root);**

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

**printf("Nodes of the BST level-wise: ");**

**printLevelWise(root);**

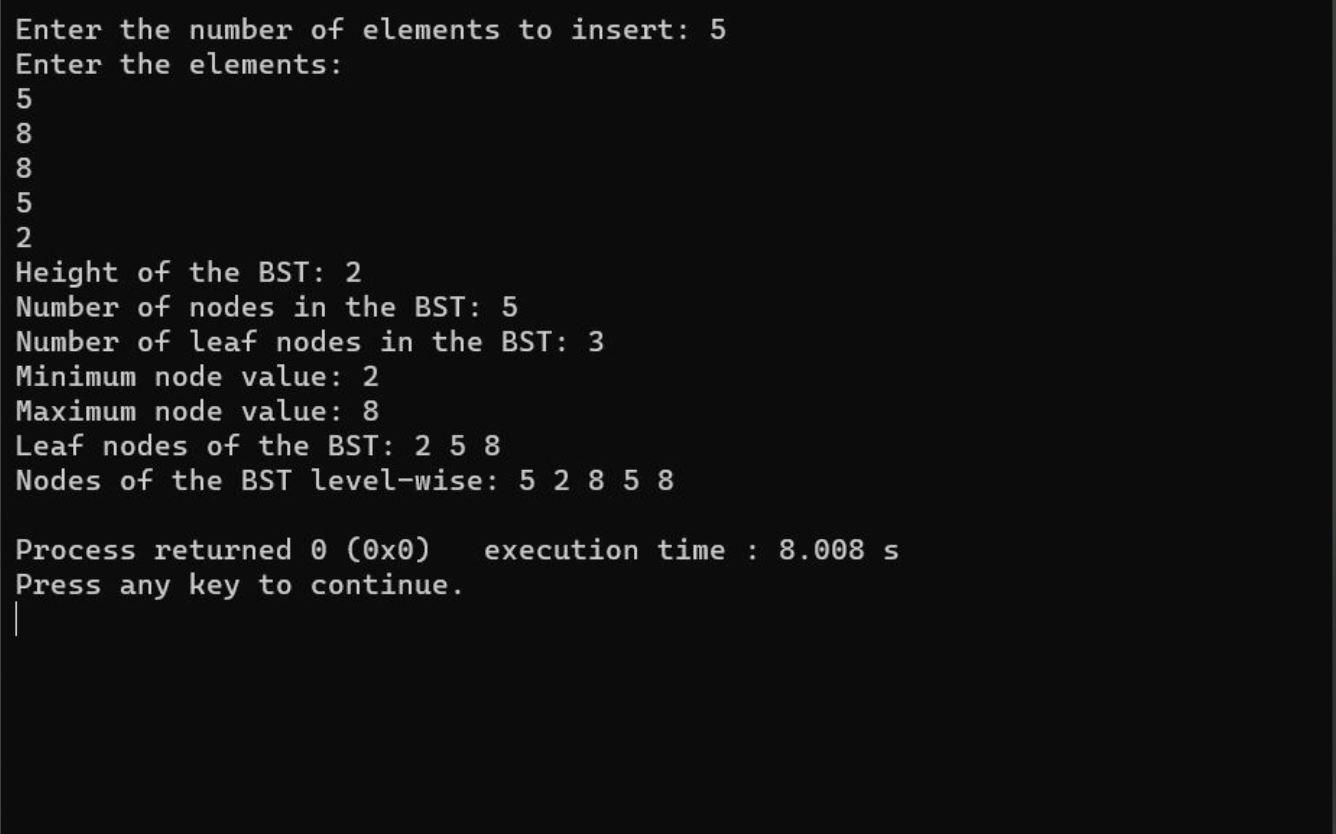
**printf("\n");**

**deleteTree(root);**

**return 0;**

**}**

**OUTPUT:-**

****

**CONCLUSION:-**

The implementation of Binary Search Tree (BST) in C demonstrates a binary tree data structure where the left child of each node holds smaller values, and the right child contains larger values. The code showcases key operations such as insertion, finding height, counting nodes, leaf nodes, and finding minimum and maximum nodes, highlighting the efficiency of BST for ordered data storage.