# Bansilal Ramnath Agarwal Charitable Trust’s

Vishwakarma Institute of Technology, Pune-37

*(An autonomous institute of Savitribai Phule Pune University)*



|  |  |
| --- | --- |
| **Year** | Second |
| **Branch** | AIDS |
| **Division** | AI-A |
| **Batch** | 2 |
| **PRN** | 12211188 |
| **Roll no.** | 47 |
| **Name** | Aneesh Deshmukh |

**PROBLEM STATEMENT**

Tree Functions

1. Count nodes, 2. Find height, 3. Mirror Image

**THEORY**

**What is a BST?**

A BST, or Binary Search Tree, is a data structure used in computer science for organizing and managing a collection of data in a hierarchical manner. It is a type of binary tree where each node has at most two children, referred to as the left child and the right child.

In a binary search tree, the nodes are arranged such that for each node:

1. All elements in the left subtree are less than or equal to the node's value.
2. All elements in the right subtree are greater than the node's value.

This arrangement allows for efficient searching, insertion, and deletion operations. When searching for a particular element in a BST, the search operation can quickly narrow down the search space by comparing the target value with the values at each node and deciding whether to continue searching in the left or right subtree based on the comparison.

**Counting Nodes – Algorithm (Using Recursion):**

To count the nodes in a BST using recursion, we follow a simple recursive approach:

1. Base Case: If the current node is None, it means we have reached the end of a branch or subtree. In this case, we return 0, indicating that there are no nodes in this subtree.
2. Recursive Step: If the current node is not None, we count the current node itself, then recursively count the nodes in its left subtree and right subtree.
3. Combine Results: We add 1 to account for the current node, and sum up the counts from the left and right subtrees.
4. Return: The final count is the sum of nodes in the left subtree, nodes in the right subtree, and the current node.

This recursive approach traverses the entire BST, counting each node exactly once. By summing up the counts from the left and right subtrees, and adding 1 for the current node, we obtain the total count of nodes in the BST.

**Counting Nodes – Algorithm (Using Stack):**

A theoretical algorithm to count nodes in a BST using a stack:

1. Initialize: Initialize a variable count to 0 to keep track of the total count of nodes.
2. Push Root onto Stack: Push the root node of the BST onto the stack.
3. While Stack is Not Empty:

* Pop a node from the stack.
* Increment the count by 1, as we've encountered a node.
* Push the right child of the popped node onto the stack if it exists.
* Push the left child of the popped node onto the stack if it exists.

1. Return Count: After the stack is empty, return the total count.

**Height of Tree – Algorithm (Using Recursion):**

To find the height of a binary search tree (BST) using recursion, you can follow this theoretical algorithm:

1. Base Case: If the current node is None, return -1. This is because in a tree, the height of a leaf node is considered to be 0, and by returning -1 for a None node, it ensures that when we add 1 to this value during the recursive step, we correctly account for the leaf node's height.
2. Recursive Step: Otherwise, recursively find the height of the left and right subtrees of the current node.
3. Combine Results: Take the maximum of the heights of the left and right subtrees, and add 1 to it to account for the current node.
4. Return: The result from step 3 is the height of the current node. Return this value.

**Height of Tree – Algorithm (Using Stack):**

The theoretical algorithm:

1. Initialize a variable max\_depth to 0 to keep track of the maximum depth encountered.
2. Initialize an empty stack.
3. Push a tuple (root, 0) onto the stack, where the second element of the tuple represents the depth of the root, which is 0.
4. While the stack is not empty:

* Pop a node current\_node and its depth current\_depth from the stack.
* Update max\_depth if current\_depth is greater than max\_depth.
* If current\_node has a right child, push (current\_node.right, current\_depth + 1) onto the stack.
* If current\_node has a left child, push (current\_node.left, current\_depth + 1) onto the stack.

1. Return max\_depth, which represents the height of the tree.

**Mirror Tree – Algorithm (Using Recursion):**

To create a mirror image of a binary search tree (BST) using recursion, you swap the left and right subtrees of each node in the tree. Here's the theoretical algorithm:

1. Base Case: If the current node is None, return None.
2. Recursive Step: Recursively call the mirror function on the left and right subtrees of the current node.
3. Swap Subtrees: After the recursive calls return, swap the left and right subtrees of the current node.
4. Return Current Node: Finally, return the modified current node.

**Mirror Tree – Algorithm (Using Stack):**

Here's the theoretical algorithm to create a mirror image of a BST using a stack:

1. Initialize an empty stack.
2. Push the root node onto the stack.
3. While the stack is not empty:

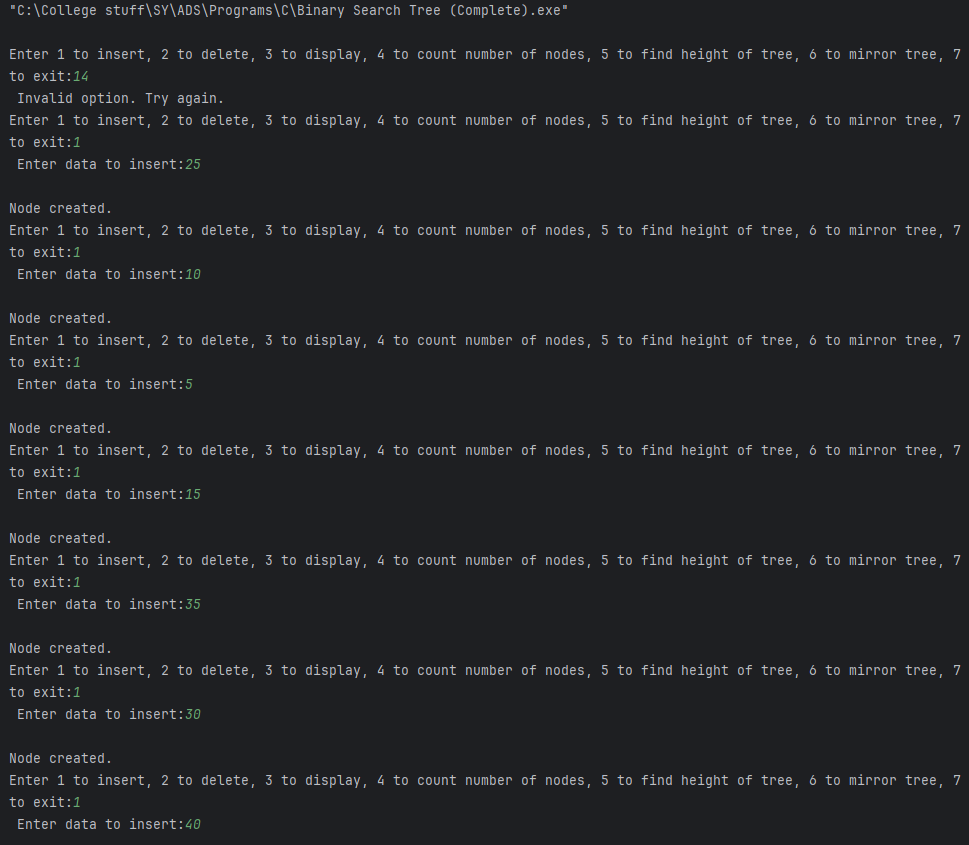
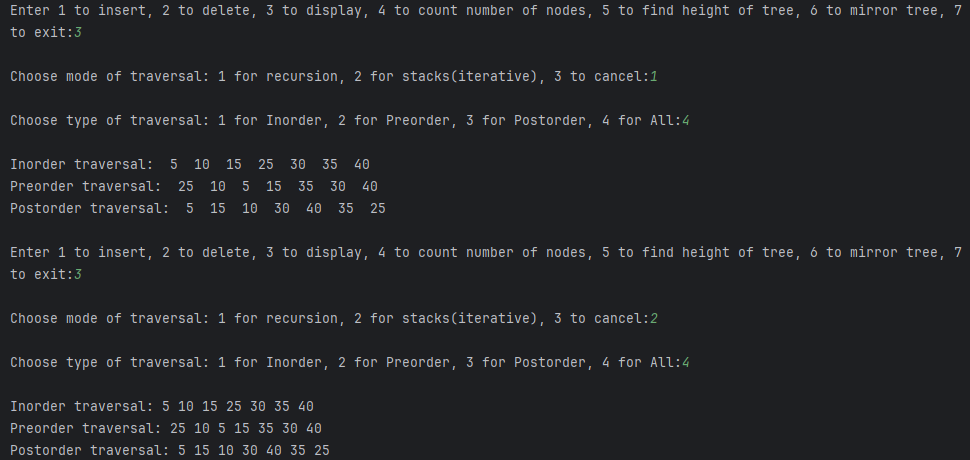
* Pop a node from the stack.
* Swap its left and right children.
* If the popped node has a left child, push it onto the stack.
* If the popped node has a right child, push it onto the stack.

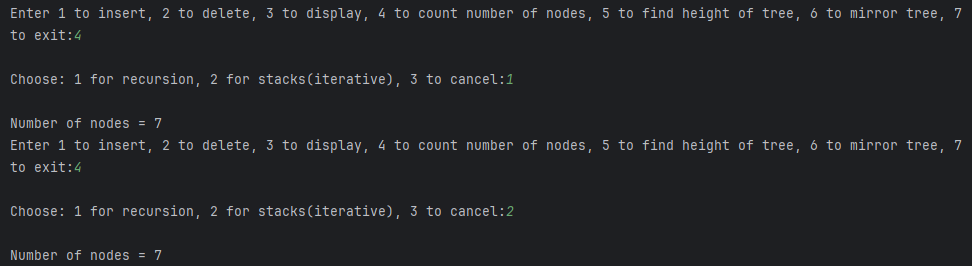
1. Return the root node of the modified tree.

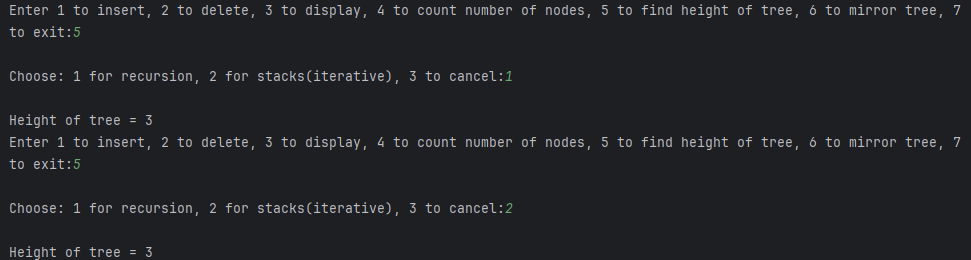
**CODE**

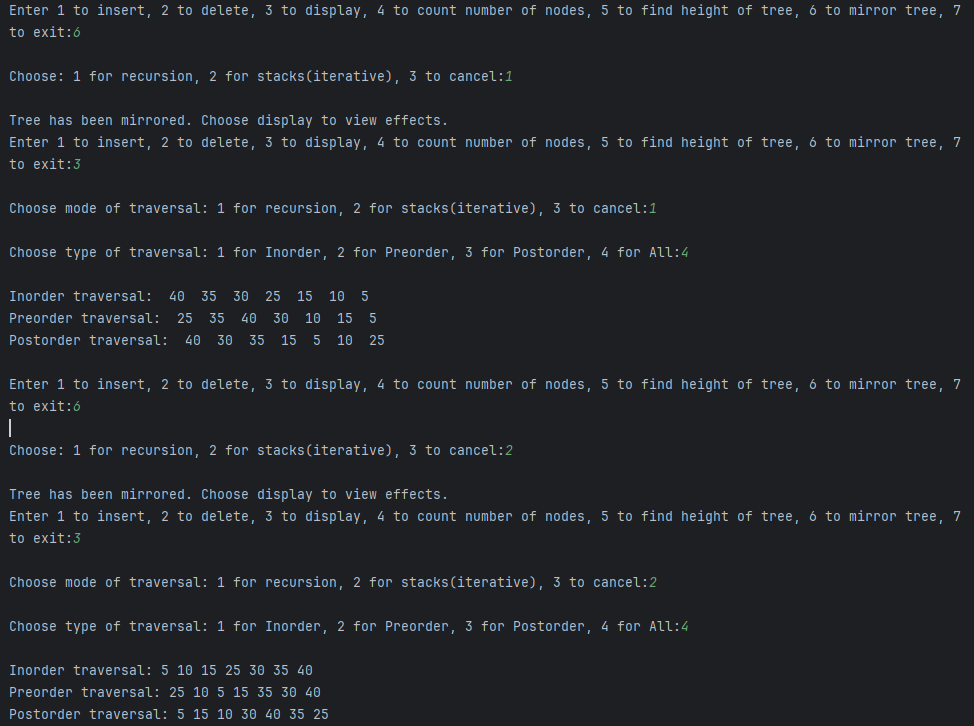
#include <stdio.h>  
#include <stdlib.h>  
  
struct BSTNode {  
 int data;  
 struct BSTNode\* left;  
 struct BSTNode\* right;  
};  
  
struct StackNode {  
 struct BSTNode\* node;  
 struct StackNode\* next;  
};  
  
struct BSTNode\* createNode(int d) {  
 struct BSTNode\* temp = (struct BSTNode\*)malloc(sizeof(struct BSTNode));  
  
 if(temp==NULL) {  
 printf("\nMemory allocation error.");  
 return createNode(d);  
 }  
  
 temp->data = d;  
 temp->left = NULL;  
 temp->right = NULL;  
  
 printf("\nNode created.");  
  
 return(temp);  
}  
  
void push(struct StackNode\*\* top, struct BSTNode\* node) {  
 struct StackNode\* snode = (struct StackNode\*)malloc(sizeof(struct StackNode));  
  
 if(snode==NULL) {  
 printf("\nMemory allocation error.");  
 return;  
 }  
  
 snode->node = node;  
 snode->next = \*top;  
 \*top = snode;  
}  
  
struct BSTNode\* pop(struct StackNode\*\* top) {  
 if(\*top==NULL)  
 return NULL;  
  
 struct StackNode\* temp = \*top;  
 \*top = (\*top)->next;  
 struct BSTNode\* popped = temp->node;  
 free(temp);  
 return popped;  
}  
  
int isEmpty(struct StackNode\* top) {  
 return top == NULL;  
}  
  
struct BSTNode\* minValueNode(struct BSTNode\* node) {  
 struct BSTNode\* current = node;  
  
 while(current && current->left != NULL)  
 current = current->left;  
  
 return current;  
}  
  
struct BSTNode\* insertNode(struct BSTNode\* root, int data) {  
 if(root==NULL)  
 return createNode(data);  
  
 if(data<root->data)  
 root->left = insertNode(root->left, data);  
 else if(data>root->data)  
 root->right = insertNode(root->right, data);  
  
 return root;  
}  
  
struct BSTNode\* deleteNode(struct BSTNode\* root, int data) {  
 if(root==NULL)  
 return root;  
  
 if(data<root->data)  
 root->left = deleteNode(root->left, data);  
 else if(data>root->data)  
 root->right = deleteNode(root->right, data);  
 else {  
 if(root->left==NULL) {  
 struct BSTNode\* temp = root->right;  
 free(root);  
 return temp;  
 }  
 else if(root->right==NULL) {  
 struct BSTNode\* temp = root->left;  
 free(root);  
 return temp;  
 }  
  
 struct BSTNode\* temp = minValueNode(root->right);  
 root->data = temp->data;  
 root->right = deleteNode(root->right, temp->data);  
 }  
  
 return root;  
}  
  
void inorderREC(struct BSTNode\* root) {  
 if(root!=NULL) {  
 inorderREC(root->left);  
 printf(" %d ", root->data);  
 inorderREC(root->right);  
 }  
}  
  
void preorderREC(struct BSTNode\* root) {  
 if(root!=NULL) {  
 printf(" %d ", root->data);  
 preorderREC(root->left);  
 preorderREC(root->right);  
 }  
}  
  
void postorderREC(struct BSTNode\* root) {  
 if(root!=NULL) {  
 postorderREC(root->left);  
 postorderREC(root->right);  
 printf(" %d ", root->data);  
 }  
}  
  
void inorderSTK(struct BSTNode\* root) {  
 if(root==NULL)  
 return;  
  
 struct StackNode\* stack = NULL;  
 struct BSTNode\* currentNode = root;  
  
 while(currentNode!=NULL || !isEmpty(stack)) {  
 while(currentNode!=NULL) {  
 push(&stack, currentNode);  
 currentNode = currentNode->left;  
 }  
  
 currentNode = pop(&stack);  
 printf("%d ", currentNode->data);  
  
 currentNode = currentNode->right;  
 }  
}  
  
void preorderSTK(struct BSTNode\* root) {  
 if(root==NULL)  
 return;  
  
 struct StackNode\* stack = NULL;  
 push(&stack, root);  
  
 while(!isEmpty(stack)) {  
 struct BSTNode\* currentNode = pop(&stack);  
 printf("%d ", currentNode->data);  
  
 if(currentNode->right!=NULL)  
 push(&stack, currentNode->right);  
  
 if(currentNode->left!=NULL)  
 push(&stack, currentNode->left);  
 }  
}  
  
void postorderSTK(struct BSTNode\* root) {  
 if(root==NULL)  
 return;  
  
 struct StackNode\* stack1 = NULL;  
 struct StackNode\* stack2 = NULL;  
 push(&stack1, root);  
  
 while(!isEmpty(stack1)) {  
 struct BSTNode\* currentNode = pop(&stack1);  
 push(&stack2, currentNode);  
  
 if(currentNode->left!=NULL)  
 push(&stack1, currentNode->left);  
  
 if(currentNode->right!=NULL)  
 push(&stack1, currentNode->right);  
 }  
  
 while(!isEmpty(stack2)) {  
 struct BSTNode\* currentNode = pop(&stack2);  
 printf("%d ", currentNode->data);  
 }  
}  
  
int rec\_count\_nodes(struct BSTNode\* root) {  
 if (root == NULL)  
 return 0;  
  
 int res = 1;  
  
 res += rec\_count\_nodes(root->left);  
 res += rec\_count\_nodes(root->right);  
  
 return res;  
}  
  
int nonrec\_count\_nodes(struct BSTNode\* root) {  
 int res = 0, top = -1;  
 struct BSTNode\* stack[100];  
  
 struct BSTNode\* current = root;  
  
 while (current!=NULL || top!=-1) {  
 while (current!=NULL) {  
 stack[++top] = current;  
 current = current->left;  
 }  
 current = stack[top--];  
 res++;  
 current = current->right;  
 }  
 return res;  
}  
  
int rec\_height (struct BSTNode\* root) {  
 if(root==NULL)  
 return 0;  
 else {  
 int lDepth = rec\_height(root->left);  
 int rDepth = rec\_height(root->right);  
  
 if(lDepth>rDepth)  
 return (lDepth+1);  
 else  
 return (rDepth+1);  
 }  
}  
  
int nonrec\_height (struct BSTNode\* root) {  
 if(root==NULL) {  
 return 0;  
 }  
  
 int height = 0, top = -1;  
 struct BSTNode\* stack[100];  
  
 struct BSTNode\* current = root;  
 struct BSTNode\* lastVisited = NULL;  
  
 while(current!=NULL || top!=-1) {  
 while(current!=NULL) {  
 stack[++top] = current;  
 current = current->left;  
 }  
  
 current = stack[top];  
  
 if(current->right==NULL || current->right==lastVisited) {  
 if(top+1>height) {  
 height = top+1;  
 }  
 lastVisited = current;  
 top--;  
 current = NULL;  
 }  
 else {  
 current = current->right;  
 }  
 }  
  
 return height;  
}  
  
void rec\_mirror (struct BSTNode\* root) {  
 if(root==NULL)  
 return;  
 else {  
 struct node\* temp;  
  
 rec\_mirror(root->left);  
 rec\_mirror(root->right);  
  
 temp = root->left;  
 root->left = root->right;  
 root->right = temp;  
 }  
}  
  
void nonrec\_mirror (struct BSTNode\* root) {  
 if (root == NULL) {  
 return;  
 }  
  
 struct BSTNode\* stack[100];  
 int top = -1;  
 stack[++top] = root;  
  
 while (top != -1) {  
 struct BSTNode\* current = stack[top--];  
  
 struct BSTNode\* temp = current->left;  
 current->left = current->right;  
 current->right = temp;  
  
 if (current->left != NULL) {  
 stack[++top] = current->left;  
 }  
 if (current->right != NULL) {  
 stack[++top] = current->right;  
 }  
 }  
}  
  
int main() {  
 struct BSTNode\* root = NULL;  
 int choice, subchoice, subsubchoice, data, res;  
  
 do {  
 printf("\nEnter 1 to insert, 2 to delete, 3 to display, 4 to count number of nodes, 5 to find height of tree, 6 to mirror tree, 7 to exit: ");  
 scanf("%d", &choice);  
  
 switch (choice) {  
 case 1:  
 printf("Enter data to insert: ");  
 scanf("%d", &data);  
 root = insertNode(root, data);  
 break;  
 case 2:  
 printf("Enter data to delete: ");  
 scanf("%d", &data);  
 root = deleteNode(root, data);  
 break;  
 case 3:  
 printf("\nChoose mode of traversal: 1 for recursion, 2 for stacks(iterative), 3 to cancel: ");  
 scanf("%d", &subchoice);  
  
 switch (subchoice) {  
 case 1:  
 printf("\nChoose type of traversal: 1 for Inorder, 2 for Preorder, 3 for Postorder, 4 for All: ");  
 scanf("%d", &subsubchoice);  
  
 switch(subsubchoice) {  
 case 1:  
 printf("\nInorder traversal: ");  
 inorderREC(root);  
 break;  
 case 2:  
 printf("\nPreorder traversal: ");  
 preorderREC(root);  
 break;  
 case 3:  
 printf("\nPostorder traversal: ");  
 postorderREC(root);  
 break;  
 case 4:  
 printf("\nInorder traversal: ");  
 inorderREC(root);  
 printf("\nPreorder traversal: ");  
 preorderREC(root);  
 printf("\nPostorder traversal: ");  
 postorderREC(root);  
 printf("\n");  
 break;  
 default:  
 printf("\nEnter valid choice.");  
 }  
 break;  
 case 2:  
 printf("\nChoose type of traversal: 1 for Inorder, 2 for Preorder, 3 for Postorder, 4 for All: ");  
 scanf("%d", &subsubchoice);  
  
 switch(subsubchoice) {  
 case 1:  
 printf("\nInorder traversal: ");  
 inorderSTK(root);  
 break;  
 case 2:  
 printf("\nPreorder traversal: ");  
 preorderSTK(root);  
 break;  
 case 3:  
 printf("\nPostorder traversal: ");  
 postorderSTK(root);  
 break;  
 case 4:  
 printf("\nInorder traversal: ");  
 inorderSTK(root);  
 printf("\nPreorder traversal: ");  
 preorderSTK(root);  
 printf("\nPostorder traversal: ");  
 postorderSTK(root);  
 printf("\n");  
 break;  
 default:  
 printf("\nEnter valid choice.");  
 }  
 break;  
 case 3:  
 break;  
 default:  
 printf("\nEnter valid choice.");  
 }  
 break;  
 case 4:  
 printf("\nChoose: 1 for recursion, 2 for stacks(iterative), 3 to cancel: ");  
 scanf("%d", &subchoice);  
  
 switch(subchoice) {  
 case 1:  
 res = rec\_count\_nodes(root);  
 printf("\nNumber of nodes = %d", res);  
 break;  
 case 2:  
 res = nonrec\_count\_nodes(root);  
 printf("\nNumber of nodes = %d", res);  
 break;  
 case 3:  
 break;  
 }  
 break;  
 case 5:  
 printf("\nChoose: 1 for recursion, 2 for stacks(iterative), 3 to cancel: ");  
 scanf("%d", &subchoice);  
  
 switch(subchoice) {  
 case 1:  
 res = rec\_height(root);  
 printf("\nHeight of tree = %d", res);  
 break;  
 case 2:  
 res = nonrec\_height(root);  
 printf("\nHeight of tree = %d", res);  
 break;  
 case 3:  
 break;  
 }  
 break;  
 case 6:  
 printf("\nChoose: 1 for recursion, 2 for stacks(iterative), 3 to cancel: ");  
 scanf("%d", &subchoice);  
  
 switch(subchoice) {  
 case 1:  
 rec\_mirror(root);  
 printf("\nTree has been mirrored. Choose display to view effects.");  
 break;  
 case 2:  
 nonrec\_mirror(root);  
 printf("\nTree has been mirrored. Choose display to view effects.");  
 break;  
 case 3:  
 break;  
 }  
 break;  
 case 7:  
 exit(0);  
 default:  
 printf("Invalid option. Try again.");  
 }  
 } while (1);  
}

**OUTPUT**

****

****

****

****