

**LDD Training Project**  
**Project done on**  
**BINARY SEARCH TREE(BST)**

**SUBMITTED BY,**  
**BHAVANA**  
**[bhavanaudupa4@gmail.com](mailto:bhavanaudupa4@gmail.com)**

# **1. INTRODUCTION**

In computer science, Binary Search Trees (BSTs) are essential data structures for effective insertion, deletion, and searching. Their ability to provide logarithmic time complexity for these operations makes them indispensable in a variety of applications, including compilers and databases.

This documentation acts as an all-inclusive manual for comprehending and applying BST functions. Insertion, deletion, search, traversal, and BST property checks are all covered. It also incorporates CPU time measurement, which is a crucial component in assessing algorithm performance.

## **2. PROBLEM STATEMENT**

### **Binary Search Tree (BST) Implementation:**

Create a Node structure with data, left, and right pointers.

### **Implement functions for:**

Insertion: Recursively add nodes while maintaining the BST property (left subtree < node < right subtree).

Deletion: Handle different cases (leaf node, single child, two children).

Search: Recursively search for a specific value, returning the node or NULL if not found.

In-order Traversal: Print nodes in ascending order (left subtree, node, right subtree).

### **Pre-order and Post-order Traversal of a Binary Tree:**

Modify the in-order traversal function from question 1 to perform pre-order (root, left, right) and post-order (left, right, root) traversals.

### **Check if a Binary Tree is a Binary Search Tree (BST):**

Implement a recursive function that checks if the left subtree is less than the current node and the right subtree is greater than the current node.

Traverse the tree and return false if any violation is found, otherwise return true.

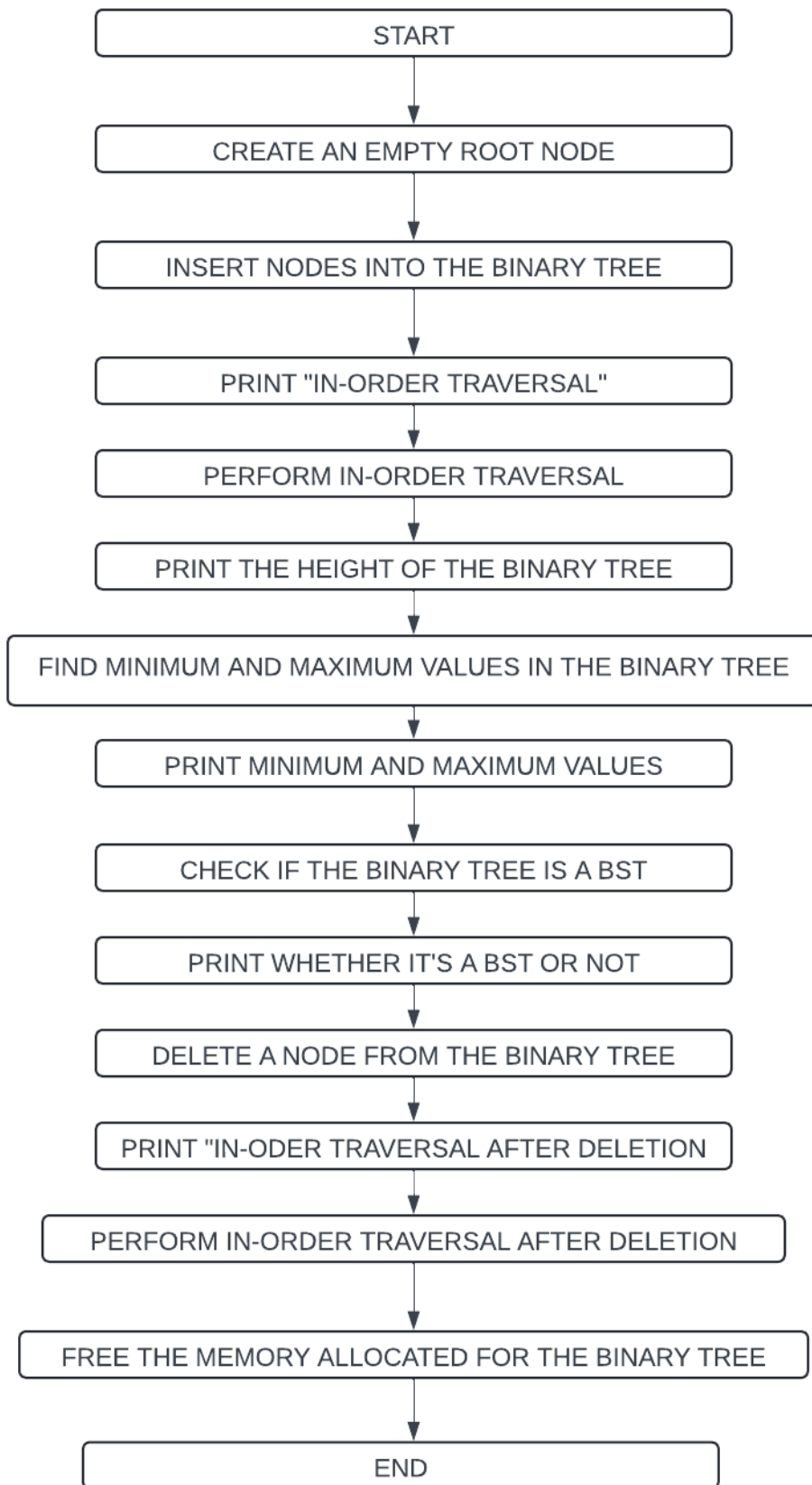
### **Find the Height of a Binary Tree:**

Implement a recursive function that returns the maximum depth (height) from the root to a leaf node.

### **Find the Minimum and Maximum Values in a Binary Tree:**

Modify the in-order traversal function to track the minimum and maximum values encountered so far.

## **FLOW CHART:**



# Program Execution

## 1. BST Initialization:

A pointer to the root node of the BST is declared and initialized to NULL.

## 2. BST Operations:

- Nodes with values 50, 30, 20, 40, 70, 60, and 80 are inserted into the BST.
- In-order, pre-order, and post-order traversals are performed on the BST, and the results are printed.
- The program searches for the value 70 in the BST and prints whether it is found or not.
- Minimum and maximum values in the BST are found and printed.
- The program checks if the binary tree is a BST and prints the result.
- The height of the binary tree is calculated and printed.

## 3. CPU Time Measurement:

CPU time is measured between two clock() calls to evaluate the time taken by a CPU-intensive task.

# CODE IMPLEMENTATION

## BST.H -> header file

```
#ifndef BST_H // Header guard to prevent multiple inclusion of the header
#define BST_H
#include <stdbool.h> // Include the standard library for boolean data type

// Node structure for BST
typedef struct Node {
    int data; // Data stored in the node
    struct Node* left; // Pointer to the left child node
    struct Node* right; // Pointer to the right child node
} Node; // Definition of the Node structure

// Function declarations
Node* createNode(int data); // Function to create a new node
Node* insert(Node* root, int data); // Function to insert a node in the BST
Node* deleteNode(Node* root, int data); // Function to delete a node from the BST
Node* search(Node* root, int data); // Function to search for a node in the BST
void inorderTraversal(Node* root); // Function to perform in-order traversal of the BST
```

```

void preorderTraversal(Node* root); // Function to perform pre-order traversal of the BST
void postorderTraversal(Node* root); // Function to perform post-order traversal of the BST
bool isBST(Node* root); // Function to check if the binary tree is a BST
int maxHeight(Node* root); // Function to find the maximum height of the binary tree
void findMinMax(Node* root, int* min, int* max); // Function to find the minimum and
maximum values in the binary tree

```

```

#endif /* BST_H */ // End of header guard directive

```

## EXPLANATION:

Your BST header file seems well-structured and includes the necessary function declarations. Here's a brief overview of each component:

1. **Header Guard**: Prevents multiple inclusions of the header file within the same translation unit, ensuring that the declarations are processed only once.
2. **Inclusion of Standard Libraries**: You've included `<stdbool.h>` for the boolean data type, which is a good practice.
3. **Node Structure**: Defines the structure of a node in the binary search tree (BST), which includes the data and pointers to the left and right child nodes.
4. **Function Declarations**: Declarations for various functions related to BST operations, such as node creation, insertion, deletion, searching, and different traversal methods. Also included are functions for checking if the tree is a binary search tree, finding the maximum height, and finding the minimum and maximum values in the tree.
5. **End of Header Guard Directive**: Marks the end of the header guard directive.

Overall, header file provides a clear and concise interface for working with binary search trees in C. If you have implemented corresponding functions for each declaration elsewhere, this header file will enable users to include it in their programs and utilize the BST operations seamlessly.

## BST.C

```

#include "bst.h" // Include the header file defining the structures and function prototypes

```

```

#include <stdio.h> // Include standard I/O functions

```

```

#include <stdlib.h> // Include standard library functions

```

```

#include <limits.h> // Include library for INT_MIN and INT_MAX

```

```

// Function to create a new node with the given data

```

```

Node* createNode(int data) {
    Node* newNode = (Node*)malloc(sizeof(Node)); // Allocate memory for the new node
    newNode->data = data; // Assign the data to the new node
    newNode->left = newNode->right = NULL; // Set left and right child pointers to NULL
    return newNode; // Return the newly created node
}

```

```

// Function to insert a node with the given data into the BST

```

```

Node* insert(Node* root, int data) {
    if (root == NULL) // If the root is NULL, create a new node and return it
        return createNode(data);
    if (data < root->data) // If data is less than the root's data, insert into the left subtree
        root->left = insert(root->left, data);
    else if (data > root->data) // If data is greater than the root's data, insert into the right subtree
        root->right = insert(root->right, data);
}

```

```

    return root; // Return the root of the modified BST
}

// Function to delete a node with the given data from the BST
Node* deleteNode(Node* root, int data) {
    if (root == NULL) // If the root is NULL, return NULL
        return root;
    if (data < root->data) // If data is less than the root's data, delete from the left subtree
        root->left = deleteNode(root->left, data);
    else if (data > root->data) // If data is greater than the root's data, delete from the right subtree
        root->right = deleteNode(root->right, data);
    else { // If the node to be deleted is found
        if (root->left == NULL) { // If the node has no left child
            Node* temp = root->right; // Store the right child temporarily
            free(root); // Free the memory occupied by the node
            return temp; // Return the right child to connect with the parent
        } else if (root->right == NULL) { // If the node has no right child
            Node* temp = root->left; // Store the left child temporarily
            free(root); // Free the memory occupied by the node
            return temp; // Return the left child to connect with the parent
        }
        // If the node has both left and right children
        Node* temp = root->right; // Find the minimum node in the right subtree
        while (temp->left != NULL)
            temp = temp->left;
        root->data = temp->data; // Copy the data of the minimum node to the current node
        root->right = deleteNode(root->right, temp->data); // Delete the minimum node from the right subtree
    }
    return root; // Return the root of the modified BST
}

// Function to search for a node with the given data in the BST
Node* search(Node* root, int data) {
    if (root == NULL || root->data == data) // If root is NULL or data is found at the root, return root
        return root;
    if (data < root->data) // If data is less than the root's data, search in the left subtree
        return search(root->left, data);
    return search(root->right, data); // If data is greater than the root's data, search in the right subtree
}

// Function to perform in-order traversal of the BST
void inorderTraversal(Node* root) {
    if (root != NULL) { // If the root is not NULL
        inorderTraversal(root->left); // Recursively traverse the left subtree
        printf("%d ", root->data); // Print the data of the current node
        inorderTraversal(root->right); // Recursively traverse the right subtree
    }
}

```

```
    }  
}
```

```
// Function to perform pre-order traversal of the BST
```

```
void preorderTraversal(Node* root) {  
    if (root != NULL) { // If the root is not NULL  
        printf("%d ", root->data); // Print the data of the current node  
        preorderTraversal(root->left); // Recursively traverse the left subtree  
        preorderTraversal(root->right); // Recursively traverse the right subtree  
    }  
}
```

```
// Function to perform post-order traversal of the BST
```

```
void postorderTraversal(Node* root) {  
    if (root != NULL) { // If the root is not NULL  
        postorderTraversal(root->left); // Recursively traverse the left subtree  
        postorderTraversal(root->right); // Recursively traverse the right subtree  
        printf("%d ", root->data); // Print the data of the current node  
    }  
}
```

```
// Utility function to check if the binary tree rooted at the given node is a BST
```

```
bool isBSTUtil(Node* root, int min, int max) {  
    if (root == NULL) // If the root is NULL, it's a BST  
        return true;  
    if (root->data < min || root->data > max) // If the data of the root is outside the valid range,  
        not a BST  
        return false;  
    // Check recursively for left and right subtrees  
    return isBSTUtil(root->left, min, root->data - 1) && isBSTUtil(root->right, root->data + 1,  
max);  
}
```

```
// Function to check if the binary tree is a BST
```

```
bool isBST(Node* root) {  
    return isBSTUtil(root, INT_MIN, INT_MAX); // Call utility function with initial range as  
INT_MIN and INT_MAX  
}
```

```
// Function to find the maximum height of the binary tree
```

```
int maxHeight(Node* root) {  
    if (root == NULL) // If the root is NULL, height is 0  
        return 0;  
    // Calculate heights of left and right subtrees recursively  
    int leftHeight = maxHeight(root->left);  
    int rightHeight = maxHeight(root->right);  
    // Return the maximum of the heights plus 1 (for the current node)  
    return (leftHeight > rightHeight) ? leftHeight + 1 : rightHeight + 1;  
}
```



```
// Function to find the minimum and maximum values in the binary tree
void findMinMax(Node* root, int* min, int* max) {
    if (root == NULL) // If the root is NULL, return
        return;
    findMinMax(root->left, min, max); // Recursively find min/max in the left subtree
    if (root->data < *min) // Update min if the data of the current node is smaller
        *min = root->data;
    if (root->data > *max) // Update max if the data of the current node is larger
        *max = root->data;
    findMinMax(root->right, min, max); // Recursively find min/max in the right subtree
}
```

```
#include <stdio.h> // Include standard I/O functions
#include <stdlib.h> // Include standard library functions
#include <time.h> // Include time functions
#include <limits.h> // Include library for INT_MIN and INT_MAX
#include "bst.h" // Include the header file defining the structures and function prototypes
```

## EXPLANATION:

Your implementation file (`bst.c`) defines the functions declared in the header file (`bst.h`) for working with binary search trees. Here's a breakdown of what each function does:

1. **createNode**: Allocates memory for a new node, assigns the given data to it, and initializes its left and right child pointers to NULL. Returns a pointer to the newly created node.
2. **insert**: Inserts a node with the given data into the binary search tree. If the tree is empty, it creates a new node as the root. Otherwise, it recursively traverses the tree to find the appropriate position for insertion based on the data value.
3. **deleteNode**: Deletes a node with the given data from the binary search tree. It handles different cases such as deleting a leaf node, a node with only one child, and a node with two children. It recursively finds the node to be deleted and adjusts the tree accordingly.
4. **search**: Searches for a node with the given data in the binary search tree. It recursively traverses the tree, comparing the data value with the current node's data until it finds a match or reaches a leaf node.
5. **inorderTraversal**: Performs an in-order traversal of the binary search tree, visiting nodes in non-decreasing order of their data values.
6. **preorderTraversal**: Performs a pre-order traversal of the binary search tree, visiting the root node first, then recursively traversing the left and right subtrees.
7. **postorderTraversal**: Performs a post-order traversal of the binary search tree, recursively traversing the left and right subtrees before visiting the root node.
8. **isBSTUtil** and **isBST**: Utility functions to check if the binary tree is a binary search tree by validating the ordering property of the tree's nodes.
9. **maxHeight**: Computes the maximum height of the binary tree, which is the length of the longest path from the root node to a leaf node.
10. **findMinMax**: Finds the minimum and maximum values in the binary search tree by traversing the tree recursively and updating the minimum and maximum values accordingly.

Implementation file also includes necessary standard library headers and the header file defining the structures and function prototypes for the binary search tree. Overall, it provides a complete set of functions for creating, modifying, and traversing binary search trees in C.

## MAIN.C

```

int main() {
    Node* root = NULL; // Declare a pointer to the root node and initialize it to NULL
    root = insert(root, 50); // Insert nodes into the BST
    insert(root, 30);
    insert(root, 20);
    insert(root, 40);
    insert(root, 70);
    insert(root, 60);
    insert(root, 80);

    printf("Inorder traversal: "); // Print the result of inorder traversal
    inorderTraversal(root);
    printf("\n");
    printf("Preorder traversal: "); // Print the result of preorder traversal
    preorderTraversal(root);
    printf("\n");
    printf("Postorder traversal: "); // Print the result of postorder traversal
    postorderTraversal(root);
    printf("\n");
    int value = 70; // Value to be searched in the BST
    Node* found = search(root, value); // Search for the value in the BST
    if (found)
        printf("%d is found in the tree.\n", value); // Print if the value is found
    else
        printf("%d is not found in the tree.\n", value); // Print if the value is not found
    int min = INT_MAX, max = INT_MIN; // Initialize variables to store min and max values
    findMinMax(root, &min, &max); // Find the minimum and maximum values in the BST
    printf("Minimum value: %d\n", min); // Print the minimum value
    printf("Maximum value: %d\n", max); // Print the maximum value
    printf("Is the binary tree a BST? %s\n", isBST(root) ? "Yes" : "No"); // Check if the binary
tree is a BST
    printf("Height of the binary tree: %d\n", maxHeight(root)); // Print the height of the binary
tree
    // CPU time calculation
    clock_t start = clock(); // Start the clock
    // Perform some CPU intensive task here
    clock_t end = clock(); // End the clock
    double cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC; // Calculate CPU
time used
    printf("CPU Time: %f seconds\n", cpu_time_used); // Print CPU time used
    return 0; // Return 0 to indicate successful completion of the program
}

```

## EXPLANATION:

`main` function demonstrates the usage of the binary search tree (BST) functions you've implemented. Here's a breakdown of what it does:

1. **\*\*Root Initialization and Insertion\*\***: Initializes a pointer ``root`` to the root node of the BST and inserts several nodes with different values into the tree.
2. **\*\*Traversal\*\***: Performs in-order, pre-order, and post-order traversals of the BST and prints the results. These traversals display the nodes of the BST in different orders.
3. **\*\*Search\*\***: Searches for a specific value (``value = 70``) in the BST using the ``search`` function and prints whether it's found in the tree or not.
4. **\*\*Minimum and Maximum Values\*\***: Finds the minimum and maximum values in the BST using the ``findMinMax`` function and prints them.
5. **\*\*Check if BST\*\***: Checks if the binary tree is a binary search tree (BST) using the ``isBST`` function and prints the result.
6. **\*\*Height Calculation\*\***: Calculates the height of the binary tree using the ``maxHeight`` function and prints it.

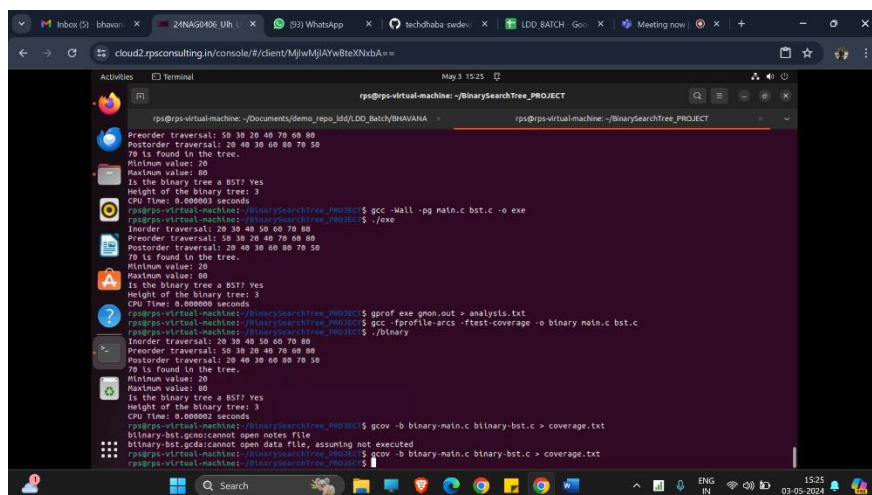
7. **CPU Time Calculation**: Measures CPU time used for a CPU-intensive task. In your code, you'll need to replace the placeholder with the actual CPU-intensive task.

8. **Return Value**: Returns `0` to indicate successful completion of the program.

Your `main` function provides a comprehensive test of the functionality of the BST operations, including insertion, traversal, search, finding minimum and maximum values, checking if it's a BST, and calculating the height of the binary tree. Additionally, it includes a placeholder for measuring CPU time, which you can replace with your actual CPU-intensive task.

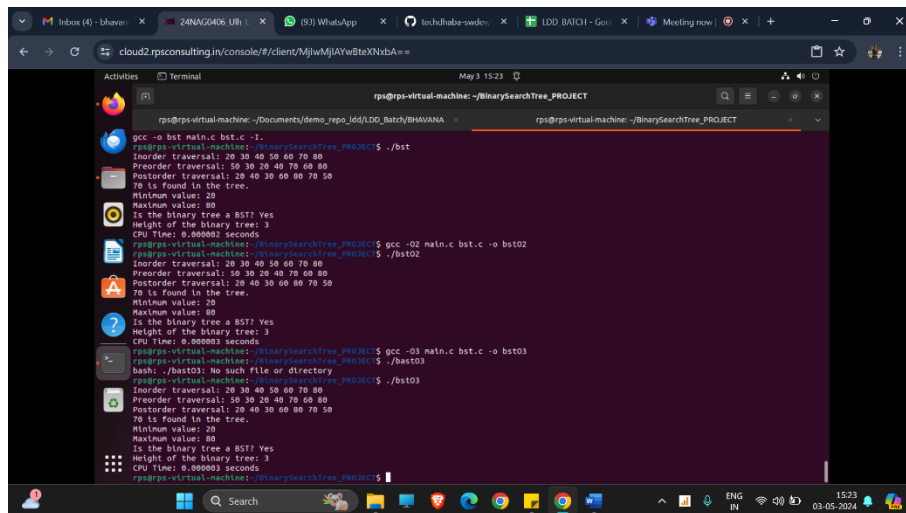
## **OUTPUT:**

## **ANALYSIS AND COVERAGE REPORT:**



```
rspr@ps-virtual-machine: ~/Documents/demo_repo_id5/LDD_batch/BHAVANA
rspr@ps-virtual-machine: ~/BinarySearchTree_PROJECT
Preorder traversal: 50 30 20 40 70 60 80
Postorder traversal: 20 40 30 60 80 70 50
70 is found in the tree.
Minimum value: 20
Maximum value: 80
Is the binary tree a BST? Yes
Height of the binary tree: 3
CPU Time: 0.000003 seconds
rspr@ps-virtual-machine: ~/BinarySearchTree_PROJECT$ gcc -Wall -pg main.c bst.c -o exe
rspr@ps-virtual-machine: ~/BinarySearchTree_PROJECT$ ./exe
Preorder traversal: 50 30 20 40 70 60 80
Postorder traversal: 20 40 30 60 80 70 50
70 is found in the tree.
Minimum value: 20
Maximum value: 80
Is the binary tree a BST? Yes
Height of the binary tree: 3
CPU Time: 0.000000 seconds
rspr@ps-virtual-machine: ~/BinarySearchTree_PROJECT$ gprof exe gmon.out > analysts.txt
rspr@ps-virtual-machine: ~/BinarySearchTree_PROJECT$ gcc -fprofile-arcs -ftest-coverage -o binary main.c bst.c
rspr@ps-virtual-machine: ~/BinarySearchTree_PROJECT$ ./binary
Preorder traversal: 50 30 20 40 70 60 80
Postorder traversal: 20 40 30 60 80 70 50
70 is found in the tree.
Minimum value: 20
Maximum value: 80
Is the binary tree a BST? Yes
Height of the binary tree: 3
CPU Time: 0.000000 seconds
rspr@ps-virtual-machine: ~/BinarySearchTree_PROJECT$ gcov -b binary-main.c binary-bst.c > coverage.txt
rspr@ps-virtual-machine: ~/BinarySearchTree_PROJECT$ gcov -b binary-main.c binary-bst.c > coverage.txt
rspr@ps-virtual-machine: ~/BinarySearchTree_PROJECT$
```

## **OPTIMIZING THE CODE USING COMMANDS:**



```
gcc -o bst main.c bst.c -l.
rps@rps-virtual-machine: ~/BinarySearchTree_PROJECT
rps@rps-virtual-machine: ~/BinarySearchTree_PROJECT
rps@rps-virtual-machine: ~/BinarySearchTree_PROJECT$ ./bst
Inorder traversal: 20 30 40 50 60 70 80
Preorder traversal: 50 30 20 40 70 60 80
Postorder traversal: 20 40 30 60 80 70 50
70 is found in the tree.
Minimum value: 20
Maximum value: 80
Is the binary tree a BST? Yes
Height of the binary tree: 3
CPU Time: 0.000002 seconds
rps@rps-virtual-machine: ~/BinarySearchTree_PROJECT$ gcc -O2 main.c bst.c -o bst02
rps@rps-virtual-machine: ~/BinarySearchTree_PROJECT$ ./bst02
Inorder traversal: 20 30 40 50 60 70 80
Preorder traversal: 50 30 20 40 70 60 80
Postorder traversal: 20 40 30 60 80 70 50
70 is found in the tree.
Minimum value: 20
Maximum value: 80
Is the binary tree a BST? Yes
Height of the binary tree: 3
CPU Time: 0.000003 seconds
rps@rps-virtual-machine: ~/BinarySearchTree_PROJECT$ gcc -O3 main.c bst.c -o bst03
rps@rps-virtual-machine: ~/BinarySearchTree_PROJECT$ ./bst03
bash: ./bst03: No such file or directory
rps@rps-virtual-machine: ~/BinarySearchTree_PROJECT$ ./bst03
Inorder traversal: 20 30 40 50 60 70 80
Preorder traversal: 50 30 20 40 70 60 80
Postorder traversal: 20 40 30 60 80 70 50
70 is found in the tree.
Minimum value: 20
Maximum value: 80
Is the binary tree a BST? Yes
Height of the binary tree: 3
CPU Time: 0.000003 seconds
rps@rps-virtual-machine: ~/BinarySearchTree_PROJECT$
```

## CONCLUSION

To sum up, this documentation has given a thorough rundown of the CPU time measurement and Binary Search Tree (BST) procedures that are included in the code. BSTs are useful in a variety of applications because they are adaptable data structures with effective search, insertion, and deletion capabilities.

Since BST operations are the foundation of many algorithms and data structures, software engineers must understand them and how to apply them. Furthermore, calculating CPU time facilitates code optimization for improved speed and helps evaluate the effectiveness of methods.

Through exploring the complexities of BST operations and CPU time measurement, developers can improve their comprehension of algorithm design and optimization, which will ultimately result in the creation of software solutions that are more reliable and efficient.

