# LDD TRAINING PROJECT

## PROJECT DONE ON

## BINARY SEARCH TREE (BST)

SUBMITTED

BY

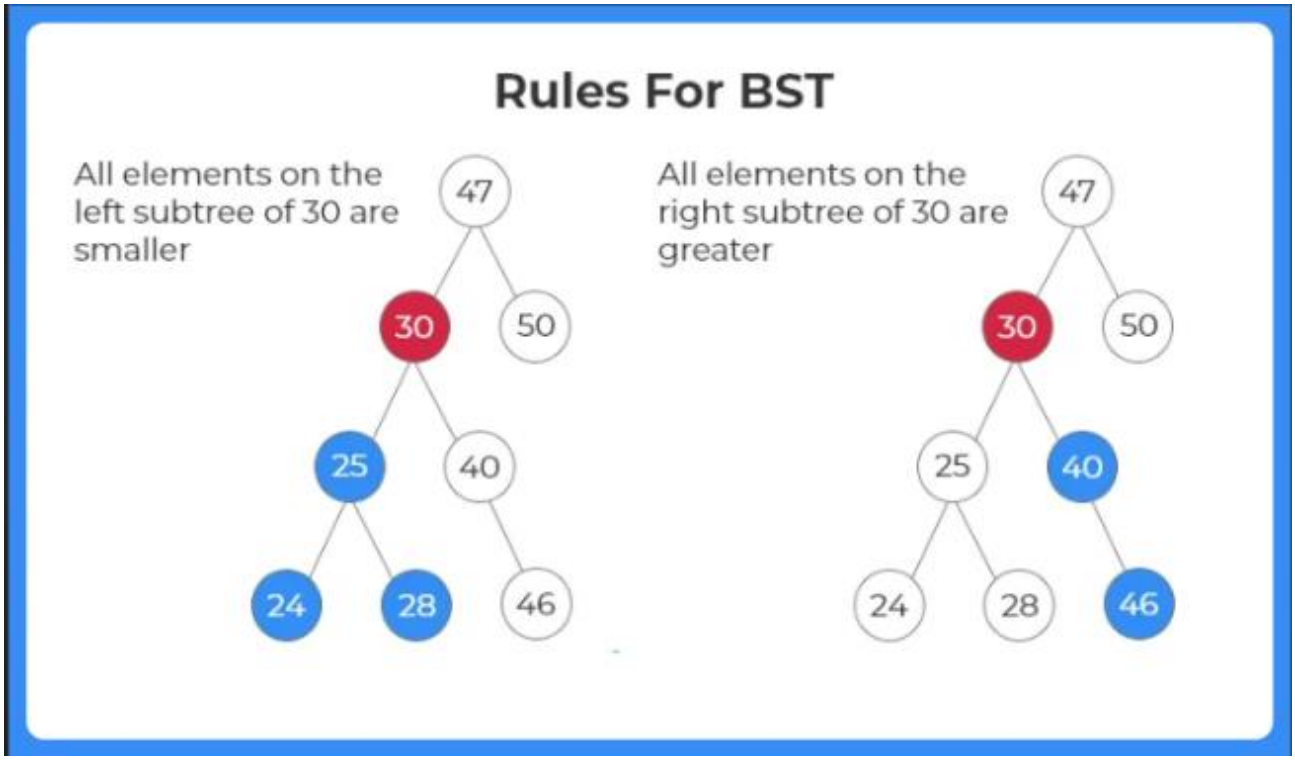
BADRINATH YADAV BIRRU

birrubadrinath@gmail.com

# INTRODUCTION

1. A **Binary Search Tree** is a data structure used in computer science for organizing and storing data in a sorted manner.
2. Each node in a **Binary Search Tree** has at most two children, a **left** child and a **right** child, with the **left** child containing values less than the parent node and the **right** child containing values greater than the parent node.
3. This hierarchical structure allows for efficient **searching**, **insertion**, and **deletion** operations on the data stored in the tree.

# BASIC STRUCTURE OF BST




1. Here in the above picture we can see the structure of binary tree.
2. For this project we also implemented operations like in-order traversal , pre-order traversal , post – order traversal , binary tree is BST , max and min values in BST , height of binary tree.

# Problem statement

## 1.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).

## 2. 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.

## 3.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.

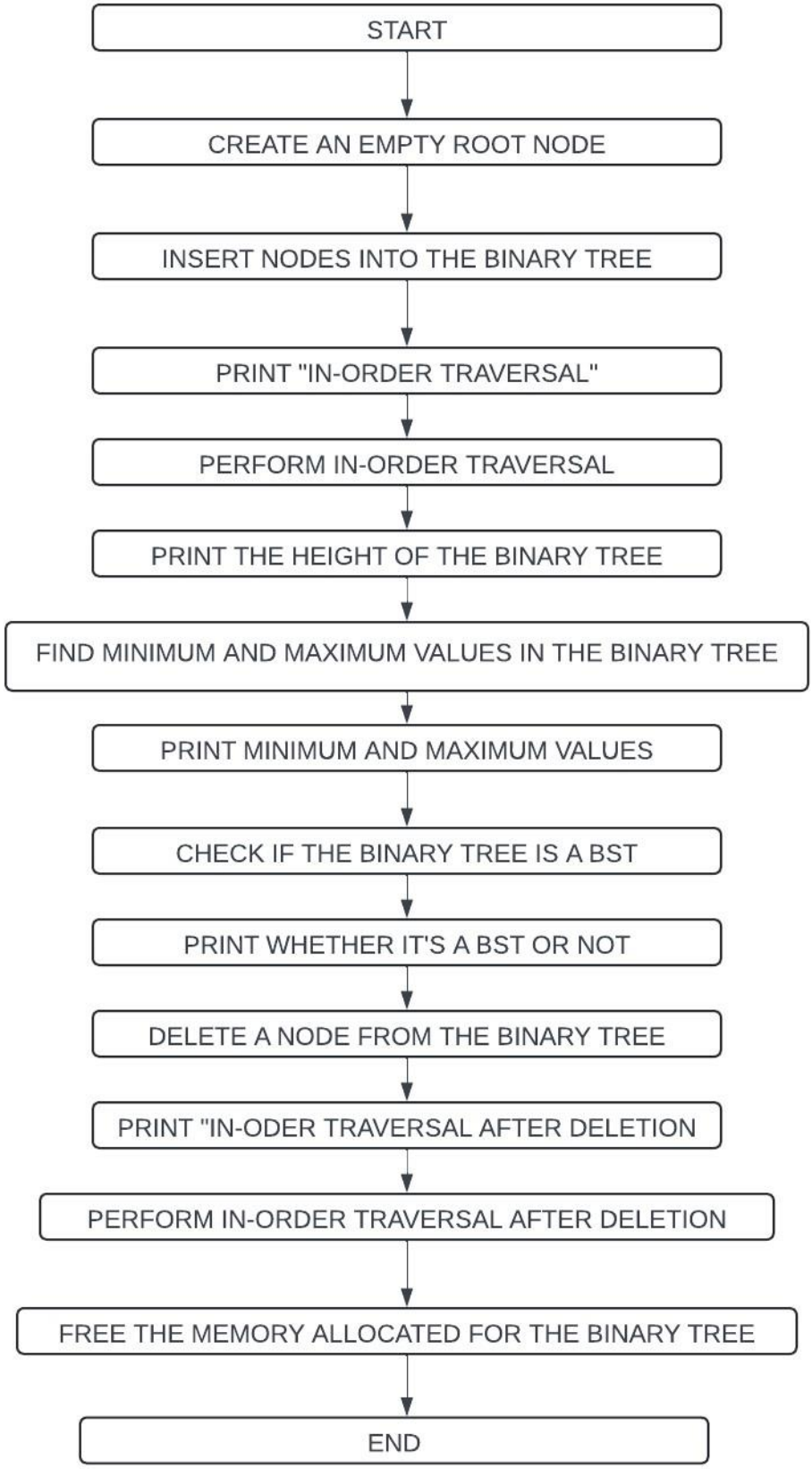
## 4.Find the Height of a Binary Tree:

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

## 5. 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.

BASIC FLOW DIAGRAM:


START
CREATE AN EMPTY ROOT NODE
INSERT NODES INTO THE BINARY TREE
PRINT "IN-ORDER TRAVERSAL"
PERFORM IN-ORDER TRAVERSAL
PRINT THE HEIGHT OF THE BINARY TREE
FIND MINIMUM AND MAXIMUM VALUES IN THE BINARY TREE
PRINT MINIMUM AND MAXIMUM VALUES
CHECK IF THE BINARY TREE IS A BST
PRINT WHETHER IT'S A BST OR NOT
DELETE A NODE FROM THE BINARY TREE
PRINT "IN-ODER TRAVERSAL AFTER DELETION
PERFORM IN-ORDER TRAVERSAL AFTER DELETION
FREE THE MEMORY ALLOCATED FOR THE BINARY TREE
END

# Program Execution

1. **BST Initialization:**

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

2. **BST Operations**:

   1.Nodes with values 50, 30, 20, 40, 70, 60, and 80 are inserted into the BST.

   2.In-order, pre-order, and post-order traversals are performed on the BST, and the results are printed.

   3.The program searches for the value 70 in the BST and prints whether it is found or not.

   4.Minimum and maximum values in the BST are found and printed.

   5.The program checks if the binary tree is a BST and prints the result.

   6.The height of the binary tree is calculated and printed.

3. **CPU Time Measurement:**

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

## NODE.H-> header file

```
 #ifndef NODE_H//ifndef header guard checks wheather header file is included previously or not

#define NODE_H// it is first step of header file and it prevent node.h from multiple includions

#include "header.h" // includes header file that contains all header functions

// Node structure for binary search tree

struct Node {

   int data; // Data of the node to be stored

   struct Node *left;  // Pointer to the left child node

   struct Node *right; // Pointer to the right child node

};

//function declarations

struct Node* createNode(int data);//function to create a newnode

struct Node* insert(struct Node* root, int data);//function to insert node

void inOrderTraversal(struct Node* root);//function to print data in order traversal

struct Node* search(struct Node* root, int data);//function for searching node data
```

```
struct Node* findMin(struct Node* root);//function to find min value in binary tree

struct Node* findMax(struct Node* root);// function to find max value in binary tree

struct Node* deleteNode(struct Node* root, int data);//function to delete node

bool isBSTUtil(struct Node* root, int min, int max);//function to search for binary tree is binary search tree

int height(struct Node* root);//function to find height of the binary tree

void preOrderTraversal(struct Node* root);//function to find pre order traversal

void postOrderTraversal(struct Node* root);//function to find pre order traversal

#endif // end of the indef header guard
```

## EXPLANATION:

Your node 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 `header.h` which contains system headers 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.

## HEADER.H

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

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

#include <stdbool.h>  // Include the standard library for boolean data type

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

## NEWNODE.C

```
#include "node.h"//includes node.h containing node fuction declaration
// Function to create a new node
```

```
struct Node* createNode(int data) {
struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));// Allocate memory for a new node
  if (newNode == NULL) {// Check if memory allocation failed
  fprintf(stderr, "Memory allocation failed\n");// Print error message to os
  exit(1);// Exit program if allocation failed unsuccesfull termination of program
   }
   newNode->data = data;// Assign data to the new node
   newNode->left = NULL;// Initialize left child pointer to NULL
   newNode->right = NULL;// Initialize right child pointer to NULL
   return newNode;// Return the new node
}
```

INSERTNODE.C

```
#include "node.h"//includes node.h header file which contains node structure

// Function to insert a new node

struct Node* insert(struct Node* root, int data) {

   if (root == NULL) {//checks  If the root is NULL or not

      return createNode(data); //returns to  CreateNode function to create a new node as the root if the root is null

   }

   if (data < root->data) {// If the data is less than root data

      root->left = insert(root->left, data);// Recursively insert into the left subtree

   } else if (data > root->data) {// If the data is greater than root data

      root->right = insert(root->right, data);// Recursively insert into the right subtree

   }

   return root;// Return the root to the insert function

}
```

INORDER.C

```
#include "node.h"
// Function to perform in-order traversal (left, root, right)
void inOrderTraversal(struct Node* root) {
   if (root != NULL) {//checks If root is not NULL
      inOrderTraversal(root->left);// Traverse left subtree
      printf("%d ", root->data);// Print root data
      inOrderTraversal(root->right);// Traverse right subtree
   }
}
```

PREORDER.C

```
#include "node.h"
// Function to perform pre-order traversal (root, left, right)
void preOrderTraversal(struct Node* root) {
   if (root != NULL) {//checks weather noot is null or not
      printf("%d ", root->data);// Print root data
      preOrderTraversal(root->left);// Traverse left subtree
      preOrderTraversal(root->right);// Traverse right subtree
   }
}
```

POSTORDER.C#include "node.h"

```
// Function to perform post-order traversal (left, right, root)
void postOrderTraversal(struct Node* root) {
   if (root != NULL) {
      postOrderTraversal(root->left);// Traverse left subtree
      postOrderTraversal(root->right);// Traverse right subtree
```

```
        printf("%d ", root->data);// Print root data
    }
}
```

MIN.C

```
#include "node.h"
// Function to perform post-order traversal (left, right, root)
void postOrderTraversal(struct Node* root) {
    if (root != NULL) {
        postOrderTraversal(root->left);// Traverse left subtree
        postOrderTraversal(root->right);// Traverse right subtree
        printf("%d ", root->data);// Print root data
    }
}
```

MAX.C

```
#include "node.h"
// Function to find the maximum value in the BST
struct Node* findMax(struct Node* root) {
    if (root == NULL) {// If root is NULL
        return NULL;  // Return NULL
    }
    if (root->right == NULL) {  // If right child is NULL
        return root; // Return the root
    }
    return findMax(root->right); // Recursively find maximum in the right subtree
}
```

DELETENODE.C

```
#include "node.h"
// Function to delete a node from the BST
struct Node* deleteNode(struct Node* root, int data) {
   if (root == NULL) { // If root is NULL
      return root; // Return root
   }
   if (data < root->data) {  // If data is less than root data
      root->left = deleteNode(root->left, data);  // Recursively delete from left subtree
   } else if (data > root->data) {   // If data is greater than root data
      root->right = deleteNode(root->right, data);  // Recursively delete from right subtree
   } else {    // If data matches root data
      if (root->left == NULL) {     // If left child is NULL
         struct Node* temp = root->right;  // Assign right child to temporary node
         free(root);    // Free memory of root
         return temp;     // Return right child as root
      } else if (root->right == NULL) {     // If right child is NULL
         struct Node* temp = root->left; // Assign left child to temporary node
         free(root);  // Free memory of root
         return temp;  // Return left child as root
      }
      struct Node* temp = findMin(root->right);  // Find minimum in right subtree
      root->data = temp->data;  // Copy minimum value to root
      root->right = deleteNode(root->right, temp->data); // Delete the minimum value node
```

```
    }
    return root; // Return root
}
```

SEARCH.C

```
#include "node.h"
// Function to search for a value in the BST
struct Node* search(struct Node* root, int data) {
    if (root == NULL || root->data == data) {// If root is NULL or matches the data
        return root; // Return the root
    }
    if (data < root->data) {// If data is less than root data
        return search(root->left, data);// Search in the left subtree
    }
    return search(root->right, data);// Search in the right subtree

}
```

HEIGHT.C

```
#include "node.h"
// Function to find the height of the BST
int height(struct Node* root) {
    if (root == NULL) {   // If root is NULL
        return -1;  // Height is -1
    }
    int leftHeight = height(root->left); // Height of left subtree
    int rightHeight = height(root->right);  // Height of right subtree
    return (leftHeight > rightHeight) ? (leftHeight + 1) : (rightHeight + 1);  //
Return maximum height + 1
```

```
}
```

BT_BST.C

```
#include "node.h"
// Function to check if a binary tree is a BST
bool isBSTUtil(struct Node* root, int min, int max) {
   if (root == NULL) {   // If root is NULL
      return true;  // It is a BST
   }
   if (root->data < min || root->data > max) { // If root data violates BST property
      return false;    // It is not a BST
   }
   return (isBSTUtil(root->left, min, root->data - 1) &&
         isBSTUtil(root->right, root->data + 1, max));   // Check left subtree  and Check right subtree
}

bool isBST(struct Node* root) {
   return isBSTUtil(root, INT_MIN, INT_MAX);   // Check if BST property is satisfied
}
```

**EXPLANATION:**

Your implementation file (`node.h`) contains the functions declared in the node.h file along with header file (`header.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

```
#include "node.h"//includes node header file which contains node strucutre with function declarations and header files
#include <time.h>//time.h includes time calculating functions like CLOCK_T
int main() {
clock_t start = clock();//function for calculating the program start time
   struct Node* root = NULL;  // Initialize root pointer


   // Insert elements into BST
   root = insert(root, 50);  //inserting 50 as root value
   insert(root, 30);//inserting data 30
   insert(root, 20);//inserting data 20
   insert(root, 40);//inserting data 40
```

```
    insert(root, 70);//inserting data 70
    insert(root, 60);//inserting data 60
    insert(root, 80);//inserting data 80

    // In-order traversal
    printf("In-order traversal: ");   // Print in-order traversal
    inOrderTraversal(root);//calling in order traversal that returns inorder traversal of root nodes data
    printf("\n");//print next line

    // Search
    int searchValue = 70; // Value to search
    struct Node* searchedNode = search(root, searchValue);  // Search for value
    if (searchedNode != NULL) {  //checking weather searchnode is not null
        printf("%d found in the BST.\n", searchValue);  // Print if found
    } else {
        printf("%d not found in the BST.\n", searchValue); // Print if not found
    }

    // Minimum and Maximum
    printf("Minimum value in the BST: %d\n", findMin(root)->data); // Print minimum value
    printf("Maximum value in the BST: %d\n", findMax(root)->data); // Print maximum value
    // Pre-order traversal
printf("Pre-order traversal: ");//print the givenstring
preOrderTraversal(root);//print preorder travel of root nodes
```

```
printf("\n");//prins next line


// Post-order traversal
printf("Post-order traversal: ");//printd the entered string
postOrderTraversal(root);//prints post order traversal of root node values
printf("\n");//print next line


    // Deletion
    int deleteValue = 30; // Value to delete
    root = deleteNode(root, deleteValue); // Delete node
    printf("In-order traversal after deleting %d: ", deleteValue);// Print in-order traversal
    inOrderTraversal(root);
    printf("\n");


    // Check if the tree is a BST
    printf("Is the tree a BST? %s\n", isBST(root) ? "Yes" : "No");// Check if BST property is satisfied


    // Height
    printf("Height of the BST: %d\n", height(root));// Print height of the tree
clock_t end = clock();//function to store the ending time of execution
double time_used = ((double)(start-end))/CLOCKS_PER_SEC;//function to calculate total time taken by program for execution
        printf("time taken : %f\n",time_used);//prints the total time taken
    return 0;//successful termination of 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 CPUintensive 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.

MAKEFILE :

```
CC =gcc//macro stores the gcc command
CFLAGS = -I.//includes the files from present working directory
DEPS = header.h node.h//includes all header files on which the .c source files depends
%.o : %.c $(DEPS)
    $(CC) -c -o $@ $< $(CFLAGS)
tree : newnode.c insertnode.c inorder.c preorder.c postorder.c min.c max.c deletenode.c search.c height.c bt_bst.c main.c
    gcc -o tree newnode.c insertnode.c inorder.c preorder.c postorder.c min.c max.c deletenode.c search.c height.c bt_bst.c main.c
```

- Make file is used to link all the header file with souce file and generate object file and stroes them in tree library.
- By typing make or make -f make file command we can create executable file tree

- By typing ./tree we can see the out put

OUTPUT:

In-order traversal: 20 30 40 50 60 70 80

70 found in the BST.

Minimum value in the BST: 20

Maximum value in the BST: 80

Pre-order traversal: 50 30 20 40 70 60 80

Post-order traversal: 20 40 30 60 80 70 50

In-order traversal after deleting 30: 20 40 50 60 70 80

Is the tree a BST? Yes

Height of the BST: 2

time taken : -0.000120

ANALYSIS AND COVERAGE REPORT:

- gcc -Wall -pg newnode.c insertnode.c inorder.c preorder.c postorder.c min.c max.c deletenode.c search.c height.c bt_bst.c main.c tet_gmon
- we created executable file for creating gmon.out file.
- ./test_gmon and this will create gmon.out file which gives the exact time , function caling etc. gives a clear idea of how the program is executing.
- gprof test_gmon gmon.out > analysis.txt

COMMANDS FOR COVERAGE:

- command to create coverage file for all program gcc -fprofile-arcs -ftest-coverage pg newnode.c insertnode.c inorder.c preorder.c postorder.c min.c max.c deletenode.c search.c height.c bt_bst.c main.c test_cov
- ./test_cov executable file creates coverage file test_cov-main.gcda like files
- Coverage file gives basic idea for average time taken for execution and lines of code executed . Helps to find the errors easily.
- gcov test_cov-newnode.c test_cov-insertnode.c test_cov-inorder.c test_cov-preorder.c test_cov-postorder.c test_cov-min.c test_cov-max.c

test_cov-deletenode.c test_cov-search.c test_cov-height.c test_cov-bt_bst.c test_cov-main.c

- command to store the program analysis or coverage file text in a .txt file.
- gcov -b test_cov-newnode.c test_cov-insertnode.c test_cov-inorder.c test_cov-preorder.c test_cov-postorder.c test_cov-min.c test_cov-max.c > coverage.txt


# 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