

CSE 4304-Data Structures Lab. Winter 2024-25**Date:** 17 December 2025**Target Group:** All groups**Topic:** Binary Tree, BST**Instructions:**

- Task naming format: fullID_T01L01_2A.c/cpp
- If you find any issues in the problem description/test cases, comment in the Google Classroom.
- If you find any tricky test cases that I didn't include but that others might forget to handle, please comment! I'll be happy to add them.
- Use appropriate comments in your code. This will help you recall the solution in the future easily.
- The obtained marks will vary based on the efficiency of the solution.
- Do not use <bits/stdc++.h> library.
- Modified sections will be marked with **BLUE** color.
- You are allowed to use the STL stack unless it's specifically mentioned to use manual functions.

Group	Tasks
1A/1B/2A/2B	1 2 3 Task 2 has 20 marks
Assignment	4 5 6 7 8

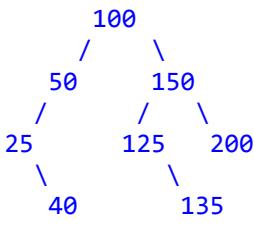
Task 1: BST insertion & Traversal Algorithms

Implement the basic insertion function of a Binary search tree and insert the given numbers one by one. After that, implement the following functions:

1. Inorder
2. Preorder
3. Postorder
4. Level_order

Insert the numbers using the ‘Binary Search Tree (BST)’ insertion policy. The first line of input will contain N, followed by N integers to be inserted in the BST.

The output must be as shown in the table. Print the parent of each node beside them. Note that you have to store the parent of each node during insertion.

Input	Output
8 100 150 50 125 135 25 40 200	Clarification: The tree looks like (Not part of output)  <pre>graph TD; 100[100] --> 50[50]; 100 --> 150[150]; 50 --> 25[25]; 50 --> 40[40]; 150 --> 125[125]; 150 --> 200[200]; 25 --> 135[135]</pre>
1	Inorder: 25(50) 40(25) 50(100) 100(null) 125(150) 135(125) 150(100) 200(150)
2	Preorder: 100(null) 50(100) 25(50) 40(25) 150(100) 125(150) 135(125) 200(150)
3	Postorder: 40(25) 25(50) 50(100) 135(125) 125(150) 200(150) 150(100) 100(null)
4	Level 1: 100(null) Level 2: 50(100) 150(100) Level 3: 25(50) 125(150) 200(150) Level 4: 40(25) 135(125)

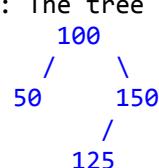
Note:

- After each insertion, print the ‘status of the tree’ using Inorder traversal. Note that the inorder traversal of a BST will always show the nodes in sorted order. (If not, there must be an error in the implementation.)
- You need to modify the Level-order Traversal algorithm to print the Level ID of each node.
- Do not write a recursive function for **insertion**. But can use recursion for the traversal algorithms.

Task 2: Implementing basic operations of BST

Implement the basic operations of a Binary Search Tree (BST). Your program should include the following operations:

1. **Insert (value):** Insert the given number by maintaining the properties of the BST. **Do not write a recursive function for insertion.** After each insertion, call the `print_tree` function to print the status of the tree in inorder fashion.
2. **print_tree:** After each insertion, print the ‘status of the tree’ using Inorder traversal. Note that the inorder traversal of a BST will always show the nodes in sorted order. (If not, there must be an error in the implementation.)
3. **Search (key):** Takes the key as input. If the key is present, print its description. Otherwise, print ‘Not Found’.
4. **Height:** Given a value, search it and return the height of that node (if present). The height of a leaf node is 0.
 - a. Note that, this height function will just return the height. **Calculation should be done during the insertion procedure.**
 - b. Write the insertion procedure in such a way that it considers height as an attribute for each node and updates height during insertion.
 - c. Do not write the typical recursive height function to visit the entire tree to return the height of a node!!
 - d. Hint: After inserting each node, follow the ancestors and call an `update_height` function, which just compares the `max(left,right)+1`.
 - e. Warning: Don’t blindly increase the parent heights! A new insertion does not necessarily mean that the height will be increased.
5. **Before_after:** Given the value of a node, you have to print the node that will appear before and after that node during inorder traversal (don’t use any traversal algorithm, or any sorting algorithm!).
6. **Max_min:** Prints the maximum and minimum value of the sub-tree below that node. Note that this function can be called with respect to any node.

Input	Output	Explanation
1 100	100	
6 100	100 100	Only 1 item is present. So it is both the max and min item
1 150	100 150	
1 50	50 100 150	
1 125	50 100 125 150	Note: The tree looks like  100 / \\\ 50 150 / 125
4 100	2	
4 125	0	
6 100	50 150	
6 150	125 150	150 is the starting point. So the minimum is 125, max is 150.

6 50	50	
1 135	50 100 125 135 150	
3 120	Not found	
3 150	Present Parent(100), Left(125), Right(Null)	
1 25	25 50 100 125 135 150	
1 40	25 40 50 100 125 135 150	
1 200	25 40 50 100 125 135 150 200	<p>Note: The tree looks like</p> <pre> graph TD 100 --- 50 100 --- 150 50 --- 25 50 --- 40 150 --- 125 150 --- 200 125 --- 135 </pre>
3 150	Present Parent(100), Left(125), Right(200)	
6 100	25 200	
6 150	125 200	As the parameter is given as 150, we check the subtree that starts from 150 as the root
3 125	Present Parent(150), Left(null), Right(135)	
3 140	Not Present	
3 40	Present Parent(25), Left(null), Right(null)	
4 100	3	
4 125	1	
4 50	2	
5 40	25 50	
5 100	50 125	
5 135	125 150	Just checking subtrees is not enough. Sometimes their value may reside in ancestors as well!
5 200	150 null	
5 25	null 40	

Task 3: Searching for LCA

In a Binary Search Tree (BST), find the Lowest Common Ancestor (LCA) for two given nodes, u and v , with the assumption that both nodes exist in the BST. The LCA of two nodes in a tree is formally defined as the nearest shared ancestor of those nodes.

The insertion process in the Binary Tree works as follows-

Insert:

- Assuming each node contains a unique value.
- Input starts with a number N (representing the number of nodes), followed by N integers in the next line that are to be inserted into the BST.

The next line of the input will be the number of queries q .

In each of the following q lines, there will be two given nodes, u_i and v_i .

Your task is to determine $LCA(u_i, v_i)$ in each query.

Input	Output	Explanation
7 4 2 6 1 3 5 7 5 5 2 1 3 7 3 5 7 6 2	4 2 4 2 4 6 4	4 / \ 2 6 / \ / \ 1 3 5 7 Note: the LCA of 5,7 is 6 (not 4). Because LCA doesn't care about the magnitude, it checks which common ancestor is the nearest one!
13 4 2 8 1 3 7 9 6 11 5 10 12 13 6 9 11 11 7 1 13 10 13 1 3 13 3	9 8 4 11 2 4	4 / \ 2 8 / \ / \ 1 3 7 9 / \ 6 11 / \ 5 10 12 \ 13

Task 4: Print paths between two nodes

A set of numbers is stored in a BST. Given two keys, your task is to print the path between the two nodes along with the length of the path (number of nodes comprising that path).

- At first, values will be inserted inside BST as long as -1 isn't given. Once the insertion is over, print the status of the tree using inorder traversal.
- Each node will show its height information in brackets.
- Your insertion algorithm should take care of the height values and store with each node. **Don't call the recursive height function for each node!** Hint: Update ancestor heights after each insertion.

Then, there will be a series of queries. Each query contains two numbers, x & y ($x < y$). Print the path to reach 'y' starting from 'x'. Additionally, count the number of nodes in that path (including x,y).

Sample Input	Sample Output	Clarification
12 7 9 20 15 27 5 3 6 11 -1	Status: 3(0) 5(1) 6(0) 7(2) 9(1) 11(0) 12(3) 15(0) 20(1) 27(0)	<pre> 12(3) / \ 7(2) 20(1) / \ / \ 5(1) 9(1) 15(0) 27(0) / \ \ 3(0) 6(0) 11(0) </pre>
5 11	5 7 9 11 4	<pre> 4 / \ 2 6 / \ / \ 1 3 5 7 </pre>
3 12	3 5 7 12 4	
3 7	3 5 7 3	
3 6	3 5 6 3	
6 7	6 5 7 3	
12 15	12 20 15 3	The LCA of 5,7 is 6 (not 4). Because LCA doesn't care about the magnitude, rather checks which common ancestor is the nearest one!
9 12	9 7 12 3	<ul style="list-style-type: none"> You don't need to print the path in sorted order. Make sure all the test cases are working. If you cannot print in the given order, you should mention it.
6 11	6 5 7 9 11 5	
7 9	7 9 2	
7 11	7 9 11 3	
3 15	3 5 7 12 20 15 6	

Task 5: K-th Smallest Element

Given a Binary Search Tree (BST), find the k-th smallest element in the tree. The insertion process in the Binary Search Tree works as follows:

- Assuming each node contains a unique value.
- Insert values maintaining the properties of a Binary Search Tree (BST).

Input:

- The first line of input will contain an integer N representing the number of nodes.
- The next line will contain N integers to be inserted into the BST.
- The next line contains an integer Q, representing the number of queries.
- Each of the next Q lines contains the integer k.

Output:

Print the k-th smallest element of the BST. If k is greater than the number of nodes, print 'Invalid'.

Input	Output	Explanation
8 100 150 50 125 135 25 40 200 5 3 10 8	50 Invalid 200	Note: The tree looks like 100 / \ 50 150 / / \ \ 25 125 200 \ \ \ \ 40 135

Task 6: Print all ancestors and Descendents

Suppose some numbers are stored in a BST. Now, given a number, you need to print all of its ancestors and descendants.

- The first line of input provides the numbers to be stored in BST (ends with -1).
- The next lines contain the value of each node.

For each node, print two lines. Ancestors and Descendants.

Input	Output	Explanation
12 7 9 20 15 27 5 3 6 11 -1	Status: 3 5 6 7 9 11 12 15 20 27	<pre> 12(3) / \ 7(2) 20(1) / \ / \ 5(1) 9(1) 15(0) 27(0) / \ \ 3(0) 6(0) 11(0) </pre>
7	12 3 5 6 9 11	
9	7 12 11	
12	NULL 3 5 6 7 9 11 15 20 27	
20	12 15 27	
3	5 7 12 Null	
5	7 12 3 6	

Note: Print the descendants in sorted order. Print Null if the ancestor/descendant does not exist.

Task 7: Diameter of a Binary Search Tree

Given a Binary Search Tree (BST), find the diameter of the tree.

The diameter of a tree is defined as the **number of nodes on the longest path** between any two nodes in the tree.

The insertion process in the Binary Search Tree works as follows:

- Assuming each node contains a unique value.
- Insert values following BST insertion rules.

Input:

The first line contains an integer N representing the number of nodes.
The next line contains N integers to be inserted into the BST.

Output:

Print the diameter of the BST.

Note:

The diameter may or may not pass through the root.

Efficient solutions will be awarded higher marks.

Input	Output	Explanation
13 4 2 8 1 3 7 9 6 11 5 10 12 13	8	<p>Note: The tree looks like</p> <pre>graph TD; 4 --> 2; 4 --> 8; 2 --> 1; 2 --> 3; 8 --> 7; 8 --> 9; 7 --> 6; 7 --> 11; 6 --> 5; 6 --> 10; 11 --> 12; 12 --> 13;</pre> <p>Longest path</p> <p>5 -> 6 -> 7 -> 8-> 9 -> 11 -> 12 -> 13</p>
10 15 10 20 8 12 16 25 6 11 18	7	<p>The tree looks like</p> <pre>graph TD; 15 --> 10; 15 --> 20; 10 --> 8; 10 --> 12; 20 --> 16; 20 --> 25; 8 --> 6; 8 --> 11; 16 --> 18;</pre> <p>Longest path</p> <p>6 -> 8 -> 10 -> 15 -> 20 -> 16 -> 18</p>

Task 8: Print Leaf Nodes of a Binary Search Tree

Given a Binary Search Tree (BST), print all the leaf nodes of the tree.
A leaf node is defined as a node that has no left child and no right child.

The insertion process in the Binary Search Tree works as follows:

- Assuming each node contains a unique value
- Insert values following BST insertion rules

Input:

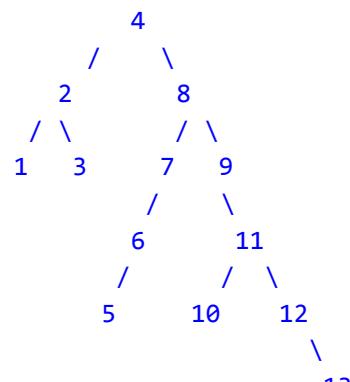
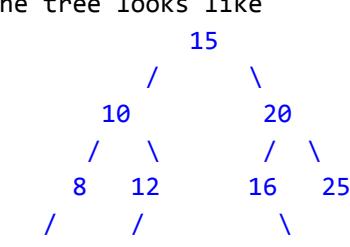
The first line contains an integer **N** representing the number of nodes.
The next line contains **N integers** to be inserted into the BST.

Output:

Print all the leaf nodes of the BST in ascending order.

Note:

- Leaf nodes appear at the bottom level of the tree
- Inorder traversal of a BST produces sorted output
- Efficient traversal-based solutions will be awarded higher marks

Input	Output	Explanation
13 4 2 8 1 3 7 9 6 11 5 10 12 13	Leaf nodes: 1, 3, 5, 10, 13	Note: The tree looks like  <pre>graph TD; 4((4)) --> 2((2)); 4 --> 8((8)); 2 --> 1((1)); 2 --> 3((3)); 8 --> 7((7)); 8 --> 9((9)); 7 --> 6((6)); 7 --> 11((11)); 6 --> 5((5)); 6 --> 10((10)); 11 --> 12((12)); 12 --> 13((13))</pre>
10 15 10 20 8 12 16 25 6 11 18	Leaf nodes: 6, 11, 18, 25	The tree looks like  <pre>graph TD; 15((15)) --> 10((10)); 15 --> 20((20)); 10 --> 8((8)); 10 --> 12((12)); 20 --> 16((16)); 20 --> 25((25)); 8 --> 6((6)); 8 --> 11((11)); 16 --> 18((18))</pre>