

**CSE 4304-Data Structures Lab. Winter 23-24****Batch:** CSE 22**Date:** November 06, 2024**Target Group:** All**Topic:** Binary Search Trees, AVL Trees**Instructions:**

- Regardless of how you finish the lab tasks, you must submit the solutions in Google Classroom. In case I forget to upload the tasks there, CR should contact me. The deadline will always be 11:59 PM on the day the lab took place.
- 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 to recall the solution in the future easily.
- 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
2A	1 2 3 4
1B	1 2 3 4
1A	2 5 6
2B	2 5 6
Assignments	2A/1B: 1A/2B:

### Task-1: Calculating the Balance Factor of different nodes

A series of values are being inserted in a BST. Your task is to show the balance factor of every node after each insertion.

$$\text{balance\_factor} = \text{height\_of\_LeftSubtree} - \text{height\_of\_RightSubtree}$$

The following requirements must be addressed:

- Continue taking input until -1.
- After each insertion, print the nodes of the tree in an in-order fashion. Show the balance\_factor beside each node within a bracket.
- Each node has the following attributes: data, left\_pointer, right\_pointer, parent\_pointer, and height. (store balance\_factor if needed, but optional)
- Your code should have the following functions:
  - void insertion (key): **iteratively** inserts a key into the BST.
  - void Update\_height(node): update the height of a node after each insertion. Note that only the ancestors are affected after inserting a new key.
  - int height(node): returns the height of a node
  - int balance\_factor(node): returns balance factor of a node

Sample Input	Sample Output
12	12(0)
8	8(0) 12(1)
5	5(0) 8(1) 12(2)
11	5(0) 8(0) 11(0) 12(2)
20	5(0) 8(0) 11(0) 12(1) 20(0)
4	4(0) 5(1) 8(1) 11(0) 12(2) 20(0)
7	4(0) 5(0) 7(0) 8(1) 11(0) 12(2) 20(0)
17	4(0) 5(0) 7(0) 8(1) 11(0) 12(1) 17(0) 20(1)
18	4(0) 5(0) 7(0) 8(1) 11(0) 12(0) 17(-1) 18(0) 20(2)
-1	

## Task-2: Balancing a BST

Utilize the functions implemented in Task-1 to provide a complete solution for maintaining a 'Balanced BST'. The program should continue inserting values until it gets -1. It checks whether the newly inserted node has imbalanced any node for each insertion. If any imbalanced node is found, 'rotation' is used to fix the issue.

Your program must include the following functions:

- void insertion (key): **iteratively** inserts a key into the BST.
- void deletion (key): **deletes a key in the BST.**
- void Update\_height(node): update the height of a node after each insertion. Note that only the ancestors are affected after inserting a new key.
- int height(node): returns the height of a node
- int balance\_factor(node): returns the balance factor of a node
- left\_rotate(node)
- right\_rotate(node)
- check\_balance(node): check whether a node is imbalanced and call relevant rotations if needed.
- print\_avl(root): print the tree using inorder traversal. The balance factor is printed beside each node.

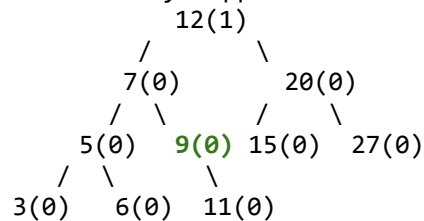
Sample Input	Sample Output
12	12(0) Balanced Root=12
9	9(0) 12(1) Balanced Root=12
5	5(0) 9(1) 12(2) Imbalance at node: 12 LL case <b>right_rotate(12)</b> Status: 5(0) 9(0) 12(0) Root=9
11	5(0) 9(-1) 11(0) 12(1) Balanced Root=9
20	5(0) 9(-1) 11(0) 12(0) 20(0) Balanced Root=9
15	5(0) 9(-2) 11(0) 12(-11) 15(0) 20(1) Imbalance at node: 9 RR case <b>Left_rotate(9)</b> Status: 5(0) 9(0) 11(0) 12(0) 15(0) 20(1) Root=12
7	5(-1) 7(0) 9(1) 11(0) 12(1) 15(0) 20(1) Balanced Root=12
3	3(0) 5(0) 7(0) 9(1) 11(0) 12(1) 15(0) 20(1)

	Balanced Root=12
6	3(0) 5(-1) 6(0) 7(1) 9(2) 11(0) 12(1) 15(0) 20(1) Imbalance at node: 9 LR Case Left_rotate(5), right_rotate(9) 3(0) 5(0) 6(0) 7(0) 9(-1) 11(0) 12(1) 15(0) 20(1) Root=12
27	3(0) 5(0) 6(0) 7(0) 9(-1) 11(0) 12(1) 15(0) 277(0) 20(0) Balanced Root=12
Delete 27	3(0) 5(0) 6(0) 7(0) 9(-1) 11(0) 12(1) 15(0) 20(1) Balanced Root=12
Delete 15	3(0) 5(0) 6(0) 7(0) 9(-1) 11(0) 12(2) 20(0) Imbalanced at node 12 3(0) 5(0) 6(0) 7(-1) 9(-1) 11(0) 12(1) 20(0)
-1	Status: 3(0) 5(0) 6(0) 7(-1) 9(-1) 11(0) 12(1) 20(0)

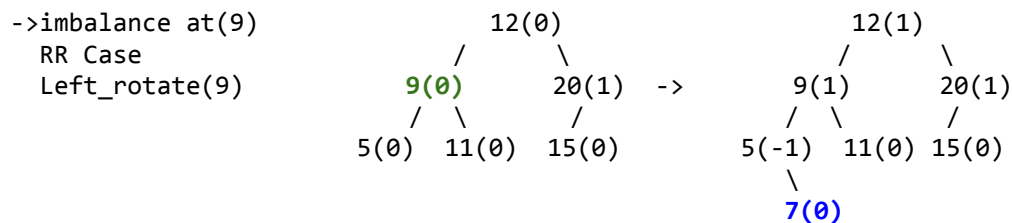
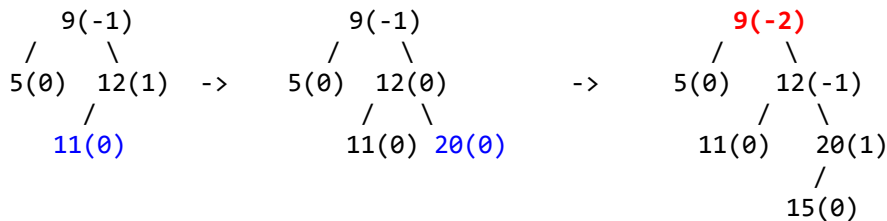
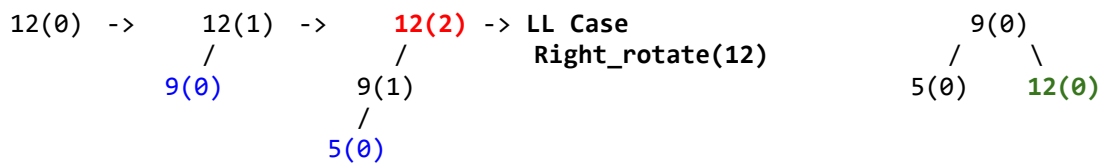
**Note:**

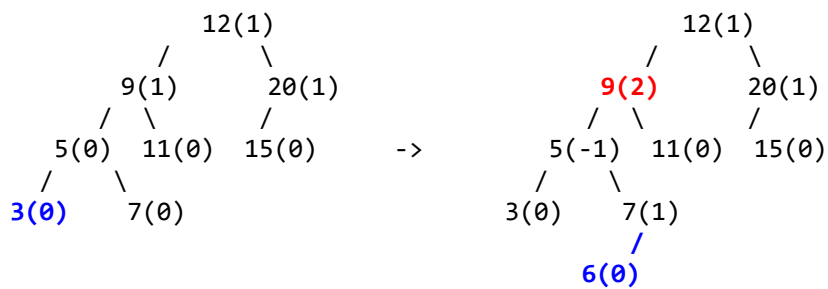
- **Do not** use any recursive implementation

The status of the tree is finally supposed to be like this:

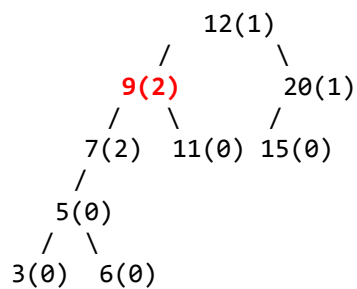


**Clarification:**

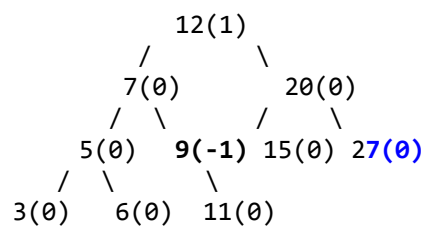
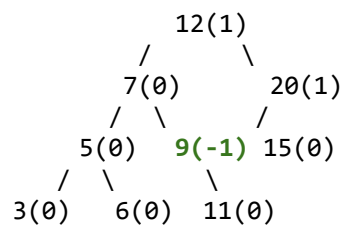




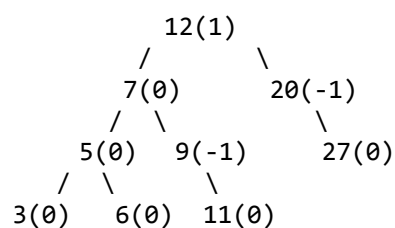
Imbalance at node(9), LR Case  
Left\_rotate(5)



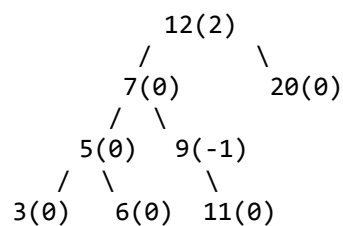
Right\_rotate(9)



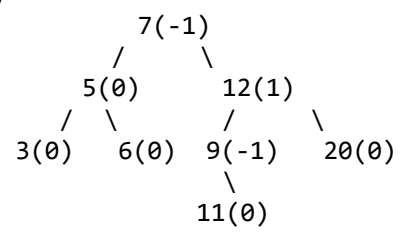
Delete (15)



Delete (27)



Imbalance at node(12)



### Task-3

[Don't start this task without completing Task 1 & 2]

Suppose a set of numbers is stored in a Balanced Binary Search Tree. An operation named 'lowerCount' is being introduced to count the total number of items less than a given query number. The obvious solution is to traverse all the items and return the count in  $O(N)$  time. Design a solution with lesser time complexity.

Sample Input	Output	Clarification
50 30 80 40 70 90 60 75 -1		<p>This input sequence should generate the following AVL tree:</p> <pre>       50      /  \     30    80      \   / \     40  70  90        / \       60  75 </pre>
50	2	30 40
40	1	30
30	0	
60	3	30 40 50
70	4	30 40 50 60
75	5	30 40 50 60 70

Hint: Use the concept of Subtree size during insertion to ensure  $O(\log N)$  complexity.

### Task 4:

[Don't start this task without completing Task 1 & 2]

Given the preorder traversal of a binary tree, find the inorder and postorder traversals.

Sample Input	Output	Clarification
A B C - - D - - E - F - -	Inorder: - C - B - D - A - E - F -  Postorder: - - C - - D B - - - F E A	The input sequence represents the following tree: <pre>       A      / \     B   E    / \ / \   C  D F  / \ / \ </pre>

Note: '-' sign denotes Null values



## Task 5:

[Don't start this task without completing Task 1 & 2]

A red-black tree is a self-balancing binary search tree that maintains efficient search, insertion, and deletion operations. It achieves this efficiency by using a specific coloring scheme for its nodes: red and black. This coloring scheme ensures that the tree remains balanced, preventing it from becoming too skewed, even after a series of insertions or deletions. When modifications are made to the tree, it automatically rearranges and recolors its nodes to maintain the balance and ensure optimal performance.

A Red-Black Tree has the following properties:

1. Every node is either red or black.
2. The root node is always black.
3. Every leaf node (NIL) is black.
4. If a node is red, then both its children are black.
5. Every path from a node to any of its descendant NIL nodes goes through the same number of black nodes.

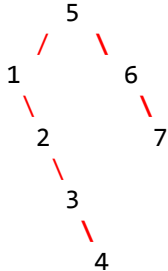
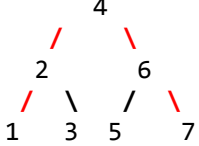
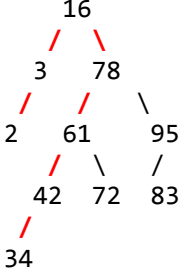
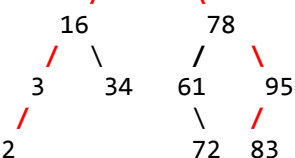
For this task you have to create the AVL trees from the given values and convert it to a Red Black Tree and print its level order traversal.

Sample Input	Sample Output	Explanation
25 15 35 30 10 20 40	(25,B) (15,R) (35,R) (10,B) (20,B) (30,B) (40,B)	<pre>       (25,B)      /    \   (15,R)  (35,R)    /  \   /  \ (10,B)(20,B)(30,B)(40,B) </pre>
13 17 8 11 1 6 22 25 27 15	(13,B) (8,R) (17,R) (1,B) (11,B) (15,B) (25,B) (6,R) (22,R) (27,R)	<pre>       (13,B)      /    \   (8,R)  (22,R)    /  \   /  \ (1,B)(11,B)(17,B)(25,B)   \   /    \ (6,R)(15,R)(27,R) </pre>

### Task 6:

[Don't start this task without completing Task 1 & 2]

The diameter of a binary tree is the length of the longest path between any two nodes in a tree. This path may or may not pass through the root. The length of a path between two nodes is represented by the number of edges between them. Find out the difference in diameter between the BST and AVL for the given set of input nodes.

Sample Input	Sample Output	Explanation
7 5 1 2 3 4 6 7	BST = 6 AVL = 4 Difference = 2	BST:  AVL: 
10 16 78 3 61 42 95 34 72 2 83	BST = 6 AVL = 6 Difference = 0	BST:  AVL: 
10 1 2 3 4 5 6 7 8 9 10	BST = 9 AVL = 5 Difference = 4	