Estd. 1990

UMT

Subject: DSA

Submitted To: Dr Saima Waseem

Submitted By: M. Burhan Shariq

ID: F2022065310

Section: A1

UNIVERSITY OF MANAGEMENT AND TECHNOLOGY
LAHORE

Course: CS 305 - Database Systems Management
Instructor: Dr. Saima Waseem
Topic: Binary Trees, Binary Search Trees (BSTs), and AVL Trees
Due Date: 12-9-2025
Name : Muhammad Burhan Shariq
Roll : F2022065310

Learning Objectives

Upon completing this assignment, you will be able to:

1. Traverse trees using different algorithms.
2. Implement and reason about the properties of Binary Search Trees.
3. Understand the rotation operations used to maintain an AVL tree.
4. Implement the core functionality of an AVL tree, including insertion and balancing.
5. Analyze the time complexity of operations on various tree structures.

Part 1: Theoretical Concepts

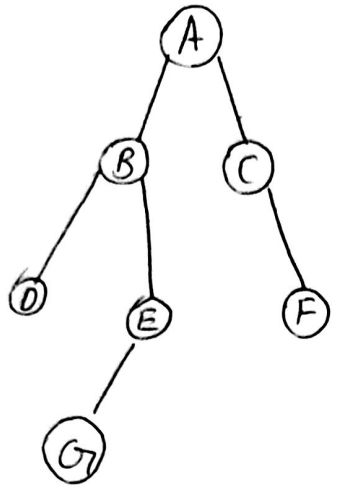
Question 1: Tree Traversal
Consider the following binary tree:

```
text
          A
         / \
        B   C
       / \   \
      D   E   F
         /
        G
```

a) Provide the node sequence for an **In-order** traversal.
b) Provide the node sequence for a **Pre-order** traversal.
c) Provide the node sequence for a **Post-order** traversal.
d) Provide the node sequence for a Breadth-First traversal.

Answer 2-

Tree



a) In order (L, Root, R): D, B, Gr, E, A, C, F

b) Pre-Order (Root, L, R): A, B, D, E, Gr, C, F

c) Post Order (L, R, Root): D, Gr, E, B, F, C, A

d) Breath-First (level-Order): A, B, C, D, E, F, Gr

Question 2: BST Properties

a) From the following sequences, which one(s) could represent a valid **in-order traversal** of a Binary Search Tree? Explain your reasoning.

i) `5, 10, 15, 20, 25, 30, 35`

ii) `10, 5, 20, 15, 30, 25, 35`

iii) `35, 30, 25, 20, 15, 10, 5`

b) Draw the **BST** that would be formed by inserting the following keys in the given order: `50, 30, 70, 20, 40, 60, 80`.

Answer

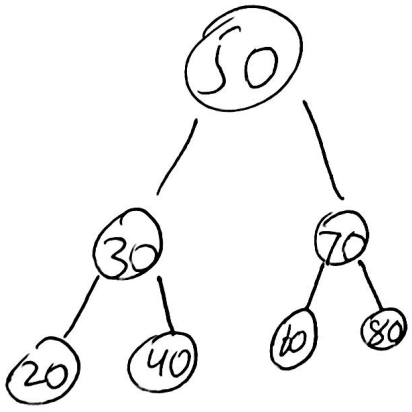
a) Which sequences can be BST in-order

i) 5, 10, 15, 20, 25, 30, 35 ☑ ascending order

ii) 10, 5, 20, 15, 30, 25, 35 ☒

iii) 35, 30, 25, 20, 15, 10, 5 ☒

b)



Question 3: AVL Tree Concepts

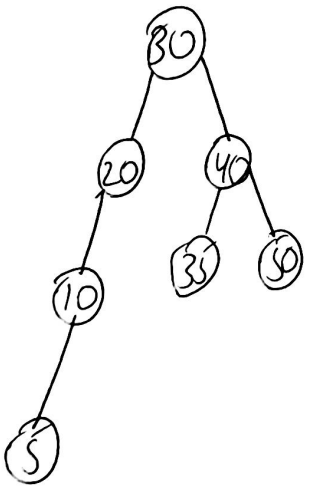
a) Calculate the **balance factor** for each node in the following tree. Is this tree an AVL tree? Justify your answer.

```
text
        30
       / \
      20  40
     /   / \
    10  35 50
   /
  5
```

c) Apply AVL on the above tree where required and, provide a simple diagram showing the tree structure before and after the necessary rotation(s).

---

a)



Balance Factor (BF = height(left) - height(right))

- Node 5 → 0
- Node 10 → +1
- Node 20 → +2 X (unbalance
- Node 35 → 0
- Node 50 → 0
- Node 40 → 0
- Node 30 → 0

Not an AVL Tree because Node 20 BF = +2

Part 2: Implementation and Analysis

Question 4: AVL Tree Insertion and Rotation
Instructions: For this question, you will **draw the trees**. Show the state of the AVL tree after each insertion. You must show the balance factor of each node next to it (e.g., `Node (BF)`). If an insertion causes the tree to become unbalanced, **identify the imbalance case** and perform the correct rotation to restore the AVL property.

Starting Tree: Begin with an empty AVL tree.
Insert the following keys in sequence: `10, 20, 30, 40, 50, 25`

Show the tree after each insertion. If a rotation is needed, show the tree *immediately before* the rotation and then *immediately after* the rotation.

Example format for your answer:

Question 5: Programming Task - AVL Tree Implementation
Implement an AVL Tree in the programming language of your choice (e.g., Python, Java, C++). Your implementation must include the following core functionalities:

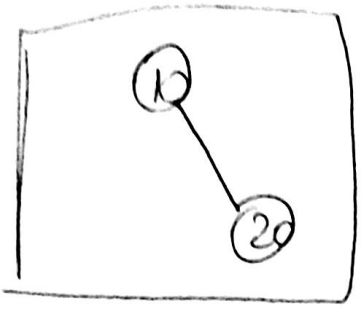
1. A `Node` class with at least the following attributes: `key`, `left`, `right`, `height`.
2. A function/method to get the `height` of a node (handle `null`/`None` pointers correctly).
3. A function/method to get the `balanceFactor` of a node.
4. Functions/methods to perform the four rotations:
    - `rotateLeft(node)`
    - `rotateRight(node)`
    - (Alternatively, you can integrate the logic for `leftLeft`, `leftRight`, etc., into your insertion method.)
5. A function/method `insert(root, key)` that inserts a new key into the tree and balances it using the necessary rotations. This method should return the new root of the subtree.
6. A function/method for **in-order traversal** to print the keys in sorted order.

Task:
a) Implement the AVL tree as described above.
b) Test your implementation by building the tree from **Question 4**. Insert the keys `10, 20, 30, 40, 50, 25` in that order.
c) **Output:**
* Print the **in-order traversal** of your final tree. (This should confirm the tree is a valid BST).

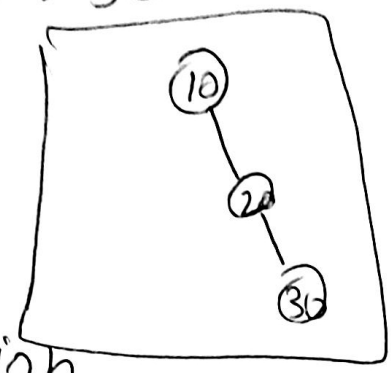
Answer no 4

- Insert 1 -> 10

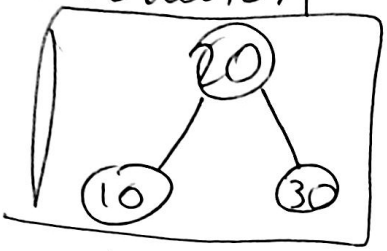
- Insert 2 -> 20



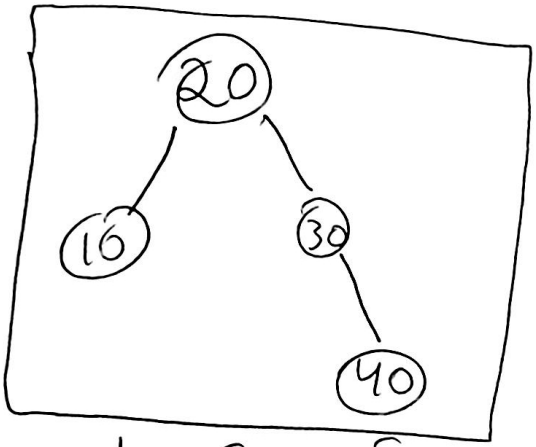
- Insert 3 -> 30



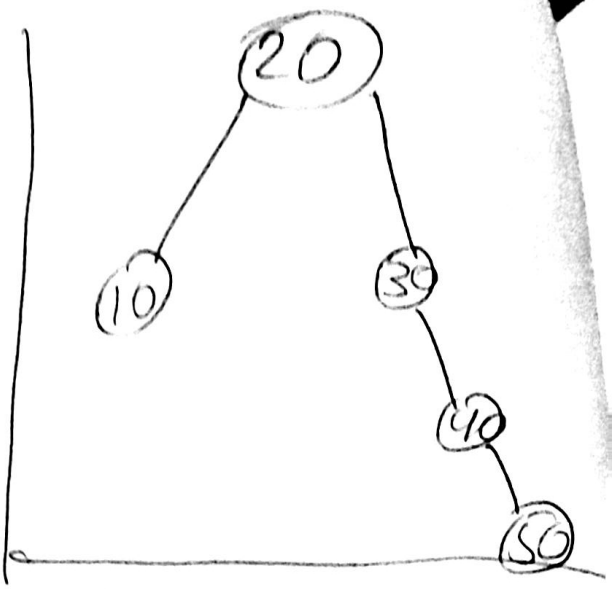
After rotation



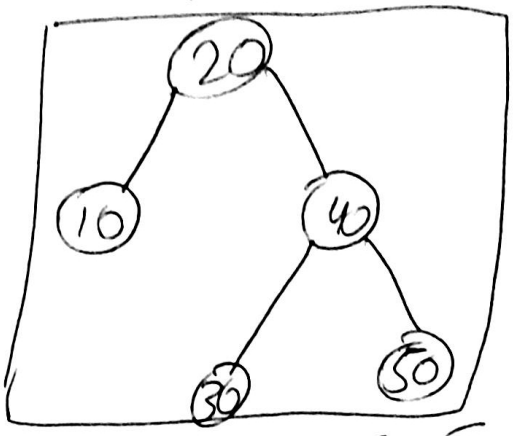
- Insert 4 -> 40



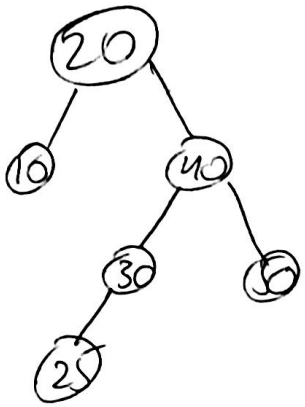
Insert 5 -> 50



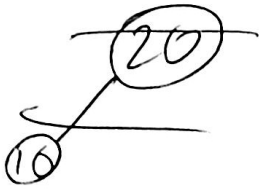
After rotation



Insert 6 -> 25

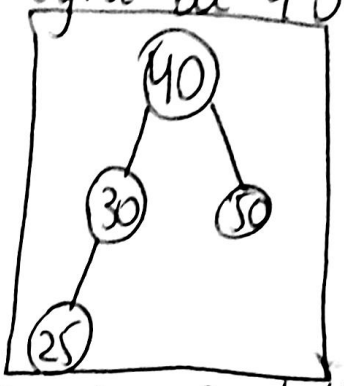


~~After rotation~~

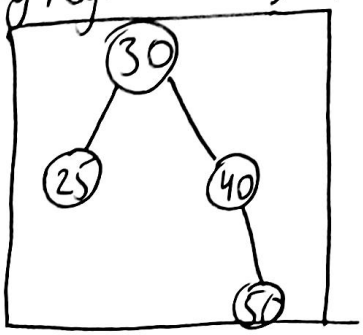


Step B

1) Rotate Right at 40 (first step

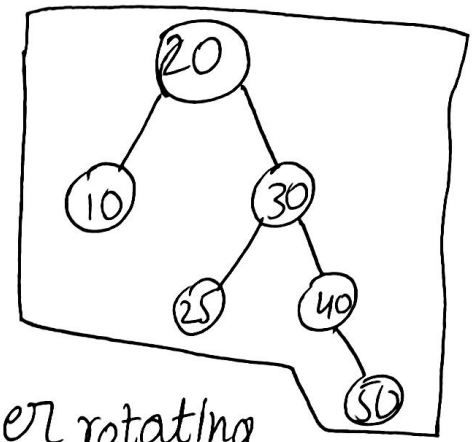


After Rotating Right (40) -> become

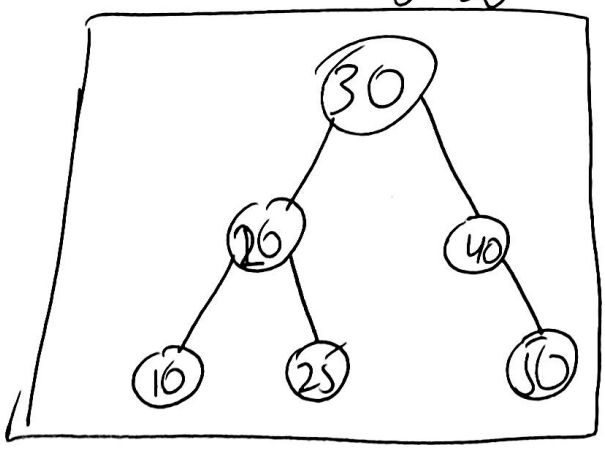


2) Rotate left at 20

30 -> 0
20 -> 0
40 -> -1



After rotating 20



Final Ans of AVL Tree

Answer 5 :-

```
#include <iostream>
#include<algorithms>
#include <queue>
using namespace std;
class Node{
public:
    int Key height;
    Node *Left *right;
    Node (int K){
        Key = K
        height = 1;
        Left = right = NULL; }
    };
Class AVLTree{
 public:
  int getHeight(Node* root){
   return root ? root->height:0;
    }
 int getBalance(Node * root){
 return root ? getheight (root->height)-getheight (root->right
                                                    :0}
 Node* rotateRight (Node* y){
 Node* x = y -> Left;
 Node* T2 = x -> right;
```

```
x->right = y
y->Left = T2
y->height = 1+ max(getHeight(y->left), getHeight(y->right));
x->height = 1+max(getHeight(x->left),getHeight(x->right));
return x; }

Node* insert(Node*root, int Key){
 if(!root) return new Node(key);
 if(Key < root->key)
   root->left = insert(root->left,Key);
 else if(Key > root-> key)
   root->right = insert(root->right,Key);
 root->height = 1+ max(getHeight(root->left),getHeight(root->
                                                 right));
 int balance = getBalance(root);
 if(balance > 1 && key < root->Left->key)
   return rotateRight(root);
 if(balance < -1 && Key > root->right->key)
   return rotateLeft(root);
 if(balance > 1 && Key > root->left->key)
   return rotateRight(root);}
 if(balance < -1 && key < root->right->key){
   return rotateLeft(root); }
   return root; }

 void inorder(Node* root) {
    if(root){
       inorder(root->left);
       cout << root->key << " ";
       inorder(root->right); }}
```

```
Void Level order (Node *root) {
 if (!root) return;
 queue <Node *> q;
 q. push (root);
 while (! q. empty()) {
        Node* node = q.front(); q. pop();
        cout << node -> Key << " ";
        if (node -> Left) q. push (node->left);
        if (node -> Right) q. push ( node -> right);
        } } };
int main() {
    AVL Tree tree;
    Node * root = NULL;
    int key[] = {10, 20, 30, 40, 50, 25};
    for (int K : Keys)
       root = tree. insert (root . K);
    cout << "In order Traversal";
     tree inorder (root);
    cout << "\n level -order Traversal";
    tree. levelorder (root);
    return 0; }
```

```
In-order Traversal: 10 20 25 30 40 50
Level-order Traversal: 30 20 40 10 25 50
Visual Tree (rotated):

            50
      40
30
            25
      20
            10
```

* Print the **level-order traversal** of your final tree. (This helps visualize the structure). You may need to implement a helper function for this.
* **Bonus:** Write a function to print the tree in a visual way (e.g., with indentation).

Question 6: Complexity Analysis

a) What is the worst-case time complexity for insertion, deletion, and search in:

c) What is the space complexity for both tree implementations?

a) Worst case time Complexity of Insertion, Deletion, Search

Operation	Unbalanced BST	AVL Tree Balance
Insertion	O(n)	O(log n)
Deletion	O(n)	O(logn)
Search	O(n)	O(logn)

Explanation

- In an unbalanced BST, if element are inserted in sorted order, the tree become skewed like a linked list) → height = n → operation take linear time (O(n)).

- In an AVL Tree rotations ensures the tree is always balance, so height = O(logn). Therefore, operations take logarithms time O(logn)

c) Space Complexity

- BST: O(n) → Each node is stored once. No extra balancing info.
- AVL Tree: O(n) → Store all nodes, plus each node has a height/balance factor (O(1) extera per node). Still O(n) overall.