Question 1

a. Reconstructed Binary Tree

Step-by-Step Reconstruction

1. Preorder root identification

The first element that is available in the preorder traversal is D, which makes it the root node.

The root node was necessary to find because when reconstructing a binary tree, the starting point must be known. In a preorder traversal, the first element is guaranteed to be the root.

2. Inorder split

Split the inorder into the subtree (B, E, C) that is the left subtree and the subtree (I, F, G, A, H) that is the right subtree using the D as the root Node.

In inorder, everything left of D belongs to the left subtree, everything right belongs to the right subtree.

Inorder: [B, E, C] D [I, F, G, A, H]

Root node: D

Left subtree: B, E, C

Right subtree: I, F, G, A, H

3. Recursively apply the same logic to the subtrees.

Left subtree

E, B, C are the nodes that appear after D in the preorder traversal. E, B, and C come right after D in the preorder list, and they belong to the left subtree of D.

E is the root node as it comes before B and C in the preorder traversal.

Looking at the inorder traversal B, E, C, the left child is B since it comes before E while the right child is C since it appears after E.

Right subtree

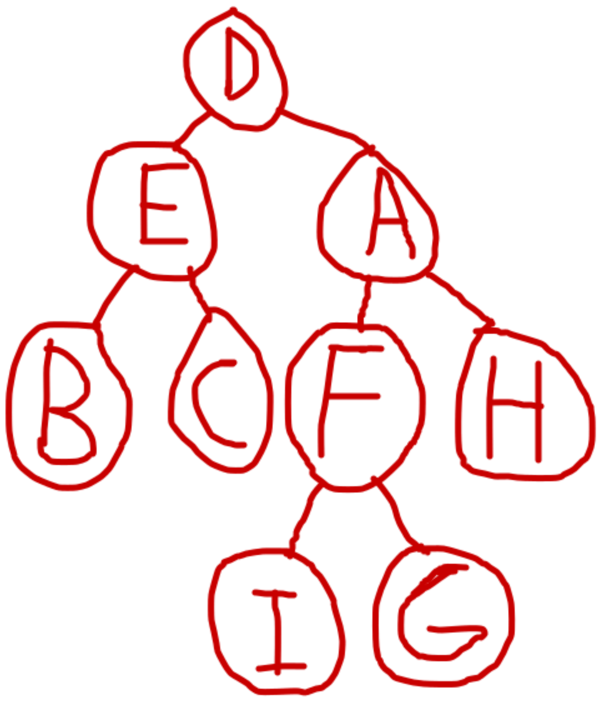
After building the left subtree of D, the remaining preorder nodes are **A, F, I, G, H**. These form the **preorder subset for the right subtree of D**.

- The first node in this subset is **A**, so A is the **root of the right subtree**.
- In the inorder traversal, the corresponding subset is **I, F, G, A, H**.
    - The nodes to the **left of A** in this inorder subset (**I, F, G**) make up **A's left subtree**.
    - The node to the **right of A**, which is **H**, is **A's right child**.

Now, looking at the **left subtree of A**:

- The preorder subset for this subtree is **F, I, G**.
- The first node, **F**, is the **root of A's left subtree**.
- In the corresponding inorder subset (**I, F, G**):
    - **I** is on the left of F, so it becomes the **left child of F**.
    - **G** is on the right of F, so it becomes the **right child of F**.

Tree Diagram:



b. General Reconstruction Process

1. The first node in preorder is the root of the current subtree.
2. Locate this root in the inorder list:
   a. Elements left of the root make up the left subtree.
   b. Elements right of the root make up the right subtree.
3. Use the size of the left subtree (from inorder) to split the preorder list:
   a. Next n nodes will form the left subtree.
   b. Remaining nodes will form the right subtree.
4. Recur on left and right subtree slices.

c. Time Complexity Analysis

<u>Best case:</u>

Occurs when the tree is balanced or nearly balanced, such as a complete binary tree.

At each recursive step, the arrays are split into two nearly equal halves.

If a hash map is used to find the root's position in the inorder array in O(1) time, the overall time complexity is O(n).

<u>Worst case:</u>
The worst case transpires when the tree is skewed, like a linked list (all nodes to one side).

The recursion depth becomes O(n), and each step splits the array into just one element and the rest.

If a hash map isn't used, the algorithm must scan the inorder array linearly at every step, leading to a time complexity of $O(n^2)$. With a hashmap, the index in inorder can be found in O(1) time. Whenever you need to find the index of a value in the inorder traversal, you can do so by accessing the value that corresponds to the node value.

E.g.
Preorder traversal: D, E, B, C, A, F, I, G, H
Inorder traversal: B, E, C, D, I, F, G, A, H

```
inorder_index_map = {
  B: 0,
  E: 1,
  C: 2,
  D: 3,
  I: 4,
  F: 5,
  G: 6,
  A: 7,
  H: 8
}

node_value = B
node_index = inorder_index_map[node_value]
```

Question 2

a. Algorithm to Check if a Binary Tree is a BST

For checking the pass-by-value problem in Python, a mutable container (that is a list) is used by the algorithm for tracking the values of the previous nodes in recursive calls. It will make sure that inorder traversal will accurately maintain the increasing order of monotonic that is needed for the BST.

Pseudocode

```
def is_bst(root):
    Use a list to maintain a mutable state (prev value) across recursive calls
    prev = [float('-inf')]

    def inorder_traversal(node):
        if node is None:
            return True

         Traverse left subtree

        if not inorder_traversal(node.left):
            return False
         Check if the current node's value is greater than the previous value

        if node.value <= prev[0]:
            return False

         Update the value of the previous node to the current node's value
        prev[0] = node.value

         Traverse the right subtree
        return inorder_traversal(node.right)

    return inorder_traversal(root)
```

Explanation

- Inorder Traversal: This algorithm performs inorder traversal that will visit all the nodes in ascending for the validation of the BST.
- Mutable State with List: The value of the node that was last visited is stored by the ‘prev’ list. As the lists are mutable all recursive calls refer to the same list which ensures that the previous values will be updated accurately throughout the traversal process.
- Validation Check: At every node, the current value is compared with the prev[0]. The binary tree will not be BST if the current value is not greater than prev[0].

This methodology will be considered effective for the verification process of BST properties by leveraging a sorted nature for inorder traversal for Python’s variable mutability and scoping with proper handling.

<u>Input(s):</u>

A pointer to the root node of a binary tree.

<u>Output:</u>

True or false depending on whether the binary tree is a valid binary search tree.

<u>A binary tree is considered to be a binary search tree if for every node:</u>

- All node values in the left subtree are less than the node's value.
- All node values in the right subtree are greater than the node's value.
- Both left and right subtrees are also binary search trees.

1. Start at the root node of the binary tree.
2. Define a valid range for node values.
   At the very beginning, before checking any nodes, the root node can be of any value. There are no restrictions yet, so the allowed range goes from the smallest possible value to the largest possible value.
3. At each node, check if the node's value falls within the valid range.
   - If it does:
     Recurse on the left subtree, updating the max bound to the current node's value.
     Recurse on the right subtree, updating the min bound to the current node's value.
   - Otherwise, the binary tree is not a binary search tree, so return false.
4. If all nodes satisfy the binary search tree condition, return true as the binary tree is a binary search tree.

b. Time and Space Complexity Analysis

- Time Complexity: During inorder traversal algorithm will visit every node exactly once so the time complexity is O(n). Here n is the number of nodes in the tree. It is considered optimal for the validation of the BST because the correct algorithm always visits all the nodes in the worst case.

- Space Complexity: Recursion stack is used for determining the space complexity that will grow as tree height (h) will grow.
- Worst Case:
  Height is n which will lead to a space complexity of O(n) for the skewed tree.
- Best Case:
  Height is O(log n) which results in a space complexity of O(log n) for the balanced tree.
- The ‘prev’ list has a constant space O(1). Thus, the recursion stack is the dominant factor.

Question 3 (15 points; 7 for a, 8 for b)

a. Balanced BST with Leaf Level Difference of 2

Objective:
Design a Balanced Binary Search Tree (BST) that consists of two leaf nodes x and y that is:
$|\text{level}(x) - \text{level}(y)| = 2$

Add values 1 through n in this way that the tree will justify the BST properties and will be reasonably balanced (Not compulsory to be AVL, but also not skewed either)

Valid Tree Structure (n = 7)

Explanation:
The leaf y=1 at level 3

The leaf x=9 at level 5

Level difference:
|5-3|= 2 so Condition is satisfied |5-3|=2
This given structure is balanced reasonably (as the tree is not skewed excessively in one direction) by following the rules of BST

BST Validation
Inorder traversal, that is: 1, 2, 4, 6, 8, 9 are increasing strictly
Properties held by the BST are

- Left subtree values less than the node
- Right subtree values are greater than the node
- All subtree nodes are BSTs themselves

b. Number of Nodes and Insertion Order
The Total Number of Nodes (n = 7)
The tree consists of 7 nodes: 1, 2, 4, 6, 8, and 9. Additionally the root node 4.

Insertion Order
To construct the tree while preserving the BST property, insert the nodes in this order:

For the construction of the tree while maintaining the properties of the BST that insert the nodes always in order as such:
4, 2, 1, 6, 8, 9

- 4 will become the root node
- 2 is inserted to the left of root node 4
- 1 is inserted to the left of subtree 2
- 6 is inserted to the right of root node 4

- 8 becomes a right child of subtree 6
- 9 becomes a right child of subtree 8

<u>**Question 4**</u>

a. Analysis of AVL Tree Leaf Level Difference

Conclusion

The claim is true.

Proof and Reasoning

1. AVL Tree Property

An AVL tree is a type of self-balancing binary search tree (BST) type.

For each node in the AVL tree, the difference in height/balanced factor between the left and the right subtree is a maximum of 1.

Balance Factor = Height (left subtree) − Height(right subtree) $\in \{-1,0,1\}$

2. Depth and Level Definitions

The level or depth of a node is called the number of edges that are from the root to that node.

Leaf nodes are the nodes that do not have children and are always available at different depths.

3. Leaf Level Difference in AVL Trees

Suppose the height of the AVL tree is h.

The earliest level that the shallowest leaf is available at is h - 2h.

The deepest leaf will appear at level h.

Ultimately, the maximum possible difference in levels between any two leaves is:
$h - (h - 2) = 2$

4. Why Level Difference Cannot Exceed 2
Suppose we want to construct the tree at a place where the difference between the two leaf levels exceeds 2.
For this implementation, some nodes in one of the subtrees would be at deeper levels of at least 3 relative to others.
It would violate the AVL condition since the subtree has an imbalanced height.
Having a height difference of 2 or more falls outside the allowed balance factor range.

5. General Observations:
AVL trees are always strictly height-balanced.
AVL tree structure always prevents the leaves from being "too far apart" in terms of depth.
That's why an AVL tree that is not valid would have two leaf nodes with a difference greater than 2 levels.

b. Insert and Delete in a Binary Search Tree (BST)
Answer
No, the tree that will be obtained as the result will not always be identical to the original tree.

Explanation with Cases

Case 1: Inserted Node is a Leaf (No Children)
When a node is inserted it is added at the correct position of the leaf in a BST.
When immediately deleted, there is no need for restructuring.

The tree will exactly revert to its original state.

Example:
Original Tree:
Insert 17, then immediately delete 17.
The structure of the tree will remain unchanged.
So, the tree is identical to the original.

Case 2: Inserted Node is an Internal Node (Has Children)
When a node is inserted and later acquires children (due to further insertions before it's deleted), deleting it will require replacing it with either its inorder successor or predecessor.

Therefore, the tree structure will change due to restructuring.

Example
Insert 12 into a tree, insert 13 as its right child, then delete 12.
13 (its child) takes 12's place.
The tree structure is now different from the original.

Case 3: Even Leaf Insertion Can Cause Pointer/Structure Changes
In languages or implementations where memory references or node identities matter, inserting or deleting a leaf can affect the internal memory layout or alter the structure of parent-child pointers.

This difference will not be seen in the diagram but will affect the algorithms relying on the identity of the node.

Question 5 – Data Structure Design for Composite Keys

Proposed Data Structure

We will implement the hybrid approach by joining a balanced binary search tree (BST) and hash map for efficient access.

Structure

- Primary Index: The BST that is balanced will have an index of k1
- Secondary Index: Every k1 node will contain a sub-BST or sorted list for all corresponding values of k2
- Hash Map: It will maintain a mapping from every k1 to its sub-BST or the list head for quick access

Operation Details

1. INSERT(S, (k1, k2))

Steps:

Check if k1 exists in the hash map:

If yes then insert k2 into the sub-BST or list if it is not present already.

If not, then create a new sub-BST or list with k2. Then insert k1 into the primary BST while updating the hash map.

Time Complexity:

O (log m + log ni)

Where:

- m is the number of unique k1 values
- ni is the number of entries for that k1

2. DELETE(S, (k1, k2))

Steps:

- Retrieve sub-BST or the list for k1 from the hash map.

- Delete k2 from the substructure.
- If the substructure will become empty, remove k1 from both the BST and hash map.

Time Complexity:
O(log m + log ni)

3. DELETE-ALL(S, k1)
Steps:

- Use a hash map for finding the sub-BST or list for k1.
- Delete all elements from the list.
- Remove k1 from the primary BST and hash map.

Time Complexity:
O(log m + log ni)