**Question 1**

**a. Reconstructed Binary Tree**

**Step-by-Step Reconstruction**

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

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

**Recursive application**

**Left subtree**

* E, B, C (they are preorder after the root) they are preorder subset for the subtree that is left.
* As the root is E( as mentioned in the preorder subset first) so the B, E, C is the in order subset.
* C is the right child, B(as in order before E no element is left) so it is Left child of E

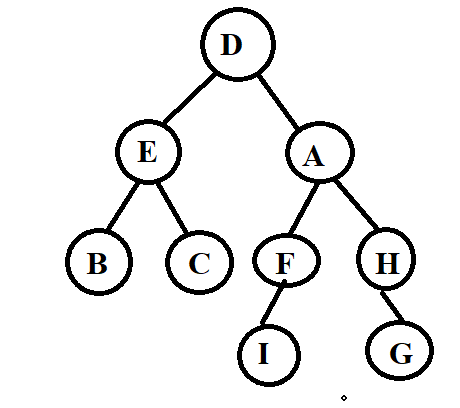
**Right subtree**

* A, F, I , G, H (After left subtree is preorder) is a Preorder subset for the right subtree
* So A (In preorder subset is first) is the root based on the in order subset that is I, F, G, A, H.
* I, F, G (In order Left for A) is A Left Sub tree and H is the right child.

**Right subtree of A**

* F, I, G are the Preorder subset of the Left subtree for A.
* As the root is F( In Preorder subset is first) so the In order subset is I, F, G.
* I is the Left Child and G(As in order Right for F is G)

**Tree Diagram:**



**b. General Reconstruction Process**

**Algorithm Overview**

**Root Identification**   
Root of the tree is the preorder traversal first element.

**Subtree Splitting**   
Partitioning the inorder traversal for left and right subtrees use the root.

**Recursive Reconstruction**

* Recursively applying same process for left subarray that is inorder traversal(subtree left) and to the preorder subarray corresponding.
* Recursively applying same process for Right subarray that is inorder traversal(subtree right) and to the preorder subarray corresponding.

**Key Insight**

Root first is given by the Preorder while left and right subtrees are separated by the inorder so recursive decomposition will be enabled

**c. Time Complexity Analysis**

**Best Case**

* When tree gets balanced or balanced like a complete binary tree.
* At every recursive step Array is divided into two nearly equal halves.
* O (n) is the time complexity using the hash map for looking the root position in time O (1).

**Worst Case**

* When tree is skewed like a linked list
* As the recursion depth is O (n) and every step will split the array into one element and rest.
* O(n2) is the time complexity as because of the repeated linear scans of the whole array( If hash map is not used)

|  |  |  |
| --- | --- | --- |
| **Scenario** | **With Hash Map** | **Without Hash Map** |
| Balanced Tree | O(n) | O(n log n) |
| Skewed Tree | O(n) | O(n²) |

**Question 2**

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

For checking the pass by value problem in python 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 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 current node's value is greater than the previous value

if node.value <= prev[0]:

return False

Update the previous value to the current node's value

prev[0] = node.value

Traverse right subtree

return inorder\_traversal(node.right)

return inorder\_traversal(root)

**Explanation**

* **Inorder Traversal:** Algorithm is performing inorder traversal that will visit all the nodes in ascending for the validation of the BST.
* **Mutable State with List:** Last visited nodes value is stored by the ‘prev’ list. As the lists are mutable so all recursive calls are referring same list that will make sure that the values that were previous will be updated accurately in the whole traversal process.
* **Validation Check:** At every node current value is compared with the prev[0]. Tree will not be BST if the current value is not be greater than prev[0].

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

**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 Validation of the BST because correct algorithm always visits all the nodes in worst case.
* **Space Complexity:** Recursion stack is used for determining the space complexity that will grow up as tree height (h) will grow.
* **Worst Case:** Height is n that will lead to space complexity of O (n) for skewed tree.
* **Best Case:** Height is O (log n) that will result into space complexity of O(log n) for the balanced tree.
* In the ‘prev’ list having a constant space O(1). So 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****

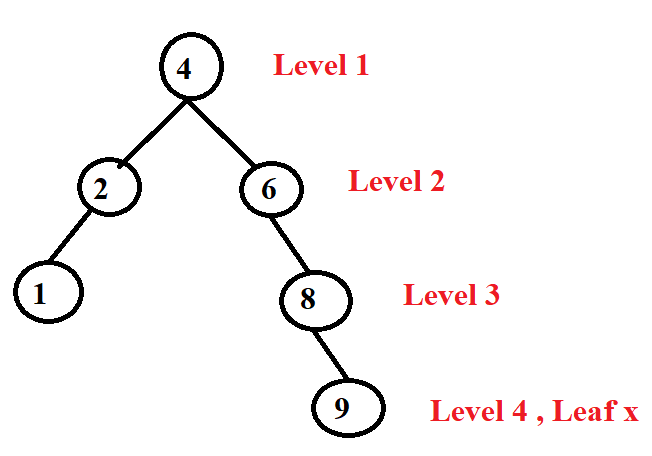
#### ****Objective:****

Design a Balanced Binary Search Tree (BST) that consists of two leaf nodes x and y that is:

∣level(x)−level(y)∣=2

Add the value 1 through the n in this way that 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 tree is not learning excessively in one direction) by following the rules of BST

**BST Validation**

* In order traversal that is : 1,2,4,6,8,9 they are increasing strictly
* Properties hold by the BST are
* Left subtree values are less that the node
* Right subtree values are greater than node
* All subtrees nodes are BSTs by 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 like

4, 2, 1, 6, 8, 9

* 4 will become root node
* 2 is inserted to left of root node 4
* 1 is inserted to left of subtree 2
* 6 is inserted to right of root node 4
* 8 becomes right child of subtree 6
* 9 becomes right child of subtree 8

**Question 4**

**a. Analysis of AVL Tree Leaf Level Difference**

**Conclusion**

The claim is **true**.

**Proof and Reasoning**

**1. AVL Tree Property**

* AVL tree is a type of self-balancing binary search tree (BST) type.
* For each node in the AVL tree, the difference of height (balanced factor) between the left and the right subtree is maximum of 1.

Balance Factor=Height (left subtree)−Height(right subtree)∈{−1,0,1}

**2. Depth and Level Definitions**

* The level(also called depth) of a node is called the number of edges that are from root to that node
* Leaf nodes are the nodes that does not have the child and is always available the different depths

**3. Leaf Level Difference in AVL Trees**

* Let consider the height of AVL tree to be h.
* The shallowest leaf can be available at the earlier level is h-2h
* The deepest leaf will appear at the level h.

So the maximum difference that is possible in levels at any two leaves is :

h− (h−2) =2

**4. Why Level Difference Cannot Exceed 2**

* Suppose if we want to construct the tree at the place where difference between the two leaves level is more than 2
* For this implementation, than some node of the one subtree should be given at the deeper levels of at least 3 as compare to others.
* It will violates the condition of AVL as it is a subtree with imbalance height ( as difference is more than or equal to 2) that will consider the balancing factors beyond the given range

**5. General Observations:**

* AVL trees are always strictly height balanced.
* AVL tree structure always prevents the leaves from being “too far apart” in the depth.
* That’s why AVL tree that is not valid would have the two leaves that have difference of level greater than 2.

**b. Insert and Delete in a Binary Search Tree (BST)**

**Answer**

No, the tree that will be obtained as the resulting will not always identical to the original tree.

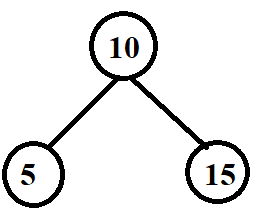
**Explanation with Cases**

**Case 1: Inserted Node is a Leaf (No Children)**

* When a node is inserted it is added at correct position of the leaf in BST.
* When immediately deleted, than there is no need of restructuring.
* The tree will exactly revert to the original state.

**Example**:

Original Tree:



* Insert 17, then immediately delete 17
* Structure of tree will remain unchanged.
* So, Tree is identical to the original.

**Case 2: Inserted Node is an Internal Node (Has Children)**

* When node is inserted and it will get the children (because of further insertion before the deletion) in such a way deletion will involve the replacing of it with its inorder successor or the predecessor.
* So the tree structure will change due to restructuring .

**Example**

Insert 12 into tree and then insert 13 as its right child. Now delete 12

* 13 (its child) takes 12's place.
* Tree structure is now different from the original.

**Case 3: Even Leaf Insertion Can Cause Pointer/Structure Changes**

* In languages or the implementations where references of memory or the identities of node matter, even when inserting and deleting of the leaf will affect the layout of the internal memory or structure of the parent-child pointer.
* This difference will not be seen in the diagram but will affect the algorithms relying on identity of the node

**Question 5 – Data Structure Design for Composite Keys**

**Proposed Data Structure**

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

**Structure**

* **Primary Index**: The BST that is balanced have an index of *k1*
* **Secondary Index**: For every *k1*node will contains a sub-BST or the 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 accessing fast.

**Operation Details**

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

**Steps**:

* Check if *k1*​ will exists in hash map:
* If yes than insert *k2*into sub-BST or the list if it is not present already
* If No than create a new sub-BST or the list with *k2* than insert *k1* into primary BST and then update the hash map.

**Time Complexity**:

*O (log m + log ni)*

Where:

* m= It is number of unique *k1*​ values,
* ni​ = It is number of entries for that *k1*​.

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

**Steps**:

* Retrieve sub-BST or the list for *k1*​from hash map.
* Delete *K2*​from substructure.
* If substructure will become empty than remove *k1*​from both BST and hash map.

**Time Complexity**:

*O (log m + log ni)*

**3. DELETE-ALL(S, *k1*)**

**Steps**:

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

**Time Complexity**:

*O (log m + log ni)*