**Question 3**

Divyansh Mathur, 210357

*This question revolves around binary search trees (B.S.T.)*

*Assumption made: Given BST has all distinct elements*

*Condition of BST:*

All children node value to the left of a particular node are smaller than the node value andchildren node value to the right of a particular node are greater than the node value.

1. **To identify the swapped nodes**

To identify the swapped nodes we need to traverse the BST and find the node which disturbs the condition of BST.

*Algorithm design:*

Therefore, if we devise a traversal in which we write the nodes on the left and then the node and then the nodes on the right then we will get an increasing sequence.

In other words, if we do this traversal and store it in an array the array will be sorted.

*What to expect:*

If the given BST is disturbed, then the traversal will not give increasing sequence such that there will be **at least** one node whose value will be smaller than the prev node encountered.

*Why at least:*

When a parent child pair is swapped it MAY NOT create 2 invalid nodes it may create only one invalid node, consider the case:

5

7 6

1 4

*CASE- I Traversal here: 1,7,5,6,4 (7 and 4 invalid case)*

5

4 7

1 6

*CASE-II Traversal here: 1,4,5,7,6*

Note here only 6 is invalid node!

In This case we will just swap the invalid node with its parent

*Algorithm:*

We define null node pointer prev ,first\_distortion , second\_distortion ,first\_next such that:

Prev represent just previous node visited

first\_distortion represent the previous node to the first node whose value < its prev val

first\_next represents the first node referred above

second\_distortion represents the node where the distortion occurs for the second time

If Second\_distortion remains NULL at the end that means we are having Case I then simply swap first\_distortion and first\_next,

Else Case II, swap first\_distortion and second\_distortion.

*Pseudocode:*

Nodes prev,first\_distortion, second\_distortion,first\_next all declared NULL inititally

traversal(root)// declared later

if(first\_distortion and second\_distortion both not NULL) swap(first\_distortion val, second\_distortion val)

else swap(first\_distortion val, first\_next val)

traversal(node){

    if(node == NULL) return;

    traversal(left child of node);

    if (prev is not NULL and (node val < prev val))

    {

        if ( first\_distortion is NULL )

        {

            first\_distortion = prev

            first\_next = node

        }

        else

            second\_distortion = node

    }

    prev = node;

    traversal(right child of node)

}

*To Determine Common ancestors:*

We use the following principle in BST:

If you are standing on a node then to find a value smaller than this node in its subtree you need to go to the left child and for greater value you need to go to the right child.

Once we have corrected the BST, We do traversal to find its common ancestor. On our traversal 4 case may arise:

1. Both node values is greater than the current node’s value: move to right child of this node
2. Both node values is smaller than the current node’s value: move to left child of this node
3. One node is > and other is < than the current node’ value: This was the last common ancestor, their path part ways now.
4. One of the node value == current node’s value: This was the last common ancestor, path of one node ends here.

*Pseudocode:*

Common\_ancestor(node)

{

    node is one of common ancestor

    if(node1 val > node val and node2 val > node val){

        Common\_ancestor(right child of node)

    }else if(node1 val < node val and node2 val < node val){

        Common\_ancestor(left child of node)

    }else{

        return;

    }

}

1. **To determine the value of k and which nodes were rearranged and design an algorithm of complexity O(min( G + n, nlog(n) )) for the same**

*Primitive Algorithm:*

We do inorder traversal (LEFT-NODE-RIGHT) of the BST once and compare it with the sorted version of this array and the nodes where the values differ count them. The count value will be K.

It will take T(n)= n log (n) + n time which is O(nlogn) time.

*Pseudocode*

inorder[n]

inorder\_traversal(root) //declared below

dup[] = duplicate(inorder)

sort(dup)

k=0

for i from 0 to n-1

    if (dup[i]!=inorder[i]) k++

print k

inorder\_traversal(root)

{

    if(root is NULL) return;

    inorder\_traversal(left child of root)

    push root in inorder[]

    inorder\_traversal(right child of root)

}

*Algorithm Using ‘G’ the maximum value:*

Create a Boolean array of length G, let it be called Present\_array. Next we do inorder traversal (LEFT-NODE-RIGHT) of the BST once and store it in an array named inorder, and also mark Present\_Array[node val] = true.

Now we traverse the Present\_array and inorder array simultaneously (we use 2 pointer one for earch array) .

If an element i is present i.e. Present\_array[i]==true, then we see whether was this the node which was meant to come, i.e. we check whether inorder[j] <?> i , if they are equal then simply move ahead i++ and j++ else store i and move ahead i++ j++.

It will take T(n)= G+n. Ovreral: O(n+G)

*Pseudocode*

inorder[n]

Present\_array[G] // all false by default

inorder\_traversal(root)

{

    if(root is NULL) return;

    inorder\_traversal(left child of root)

    Present\_array[node val]=true

    push root in inorder[]

    inorder\_traversal(right child of root)

}

j=0,k=0, ans\_arr[]

for i from 0 to G-1

{

    if(Present\_array==true){

        if(inorder[j]!=i ){

            store i in ans\_array

            k++

        }

        j++

    }

}

print k

*Overall Time Complexity:*

Min(O(nlogn), O(G+n))