afteracademy.com

Recover Binary Search Tree-Interview Problem

7-9 minutes

Difficulty: Hard

Asked in: Amazon, Microsoft

Understanding the Problem:

Given a Binary Search Tree such that two of the nodes of this tree have been swapped by mistake. You need to write a program that will recover this BST while also maintaining its original structure.

For Example:

```
Input: Given a BST in which two of the nodes are
swapped by mistake.
                  10
                 /  \
                7    9
               / \
              5   15
Output:

                  10
                 /  \
                7    15
               / \
              5   9
Explanation: As you can easily see, in the above
binary search tree, the two nodes 9 and 15 have been
swapped with each other. This has been fixed in the
output tree.
```

Possible questions to ask the interviewer:

- Can I use auxiliary space to solve this problem? (**Ans:** *Try to optimise your solution for constant space complexity.*)
- Two of the adjacent nodes can also be swapped with each other? (**Ans:** *Yes!*)

Before proceeding to solutions below, don't you want to give this problem a try? Click *here*.

Solutions

We are going to discuss two possible solutions for this problem. We will start with a simple naive solution that will definitely come to our minds and then will see its complexity and will come up with a less complex, efficient solution.

- **Solution I- Using Extra Space**
- **Solution II- Using Constant Space**

Solution I- Using Extra Space

Solution idea

The idea is fairly simple, we know that in-order traversal of the binary search tree will give us elements in the sorted order. Now, in this case, since the two elements are swapped with each other, the in-order traversal will not result in a list of sorted elements. Better say, there will be two misplacements of elements in the sorted list. By taking the in-order traversal of the given BST and sorting it, we can easily have a check on the swapped elements. This check will now help us in finding the corresponding nodes in the tree and exchanging it with each other. This will help us in recovering the original tree and we will also be storing the structure of the given tree.

Solution steps

- We will have the in-order traversal of the given BST stored in a list.
- We will sort this list by making a copy of this list.
- Then, we will compare both the lists to find the elements which have been swapped.
- After that, search the nodes in the tree and update the value of

those nodes.

Pseudo-code

```
void inorderTraversal(TreeNode root, int
arrInorder[])
{
    inorderTraversal(root.left)
    arrInorder.add(root.data)
    inorderTraversal(root.right)
}
void searchAndUpdate(TreeNode root, int element1, int
element2)
{
    if(root == NULL)
       return
    searchAndUpdate(root.left, element1, element2)
    if(root.data == element1)
       root.data = element2
    else if(root.data == element2)
       root.data == element1
    searchAndUpdate(root.right, element1, element2)
}
void fixBst(TreeNode root)
{
    int[] arrInorder
    inorderTraversal(root, arrInorder)
    int[] copyArr = arrInorder.copy()   //copies
array into new one
    copyArr.sort()    //will sort the array
    for(int i = 0 to arrInorder.length; i+=1)
```

```
    {
        if(arrInorder[i] != copyArr)
            searchAndUpdate(root, arrInorder[i],
copyArr[i])
            break
    }
}
```

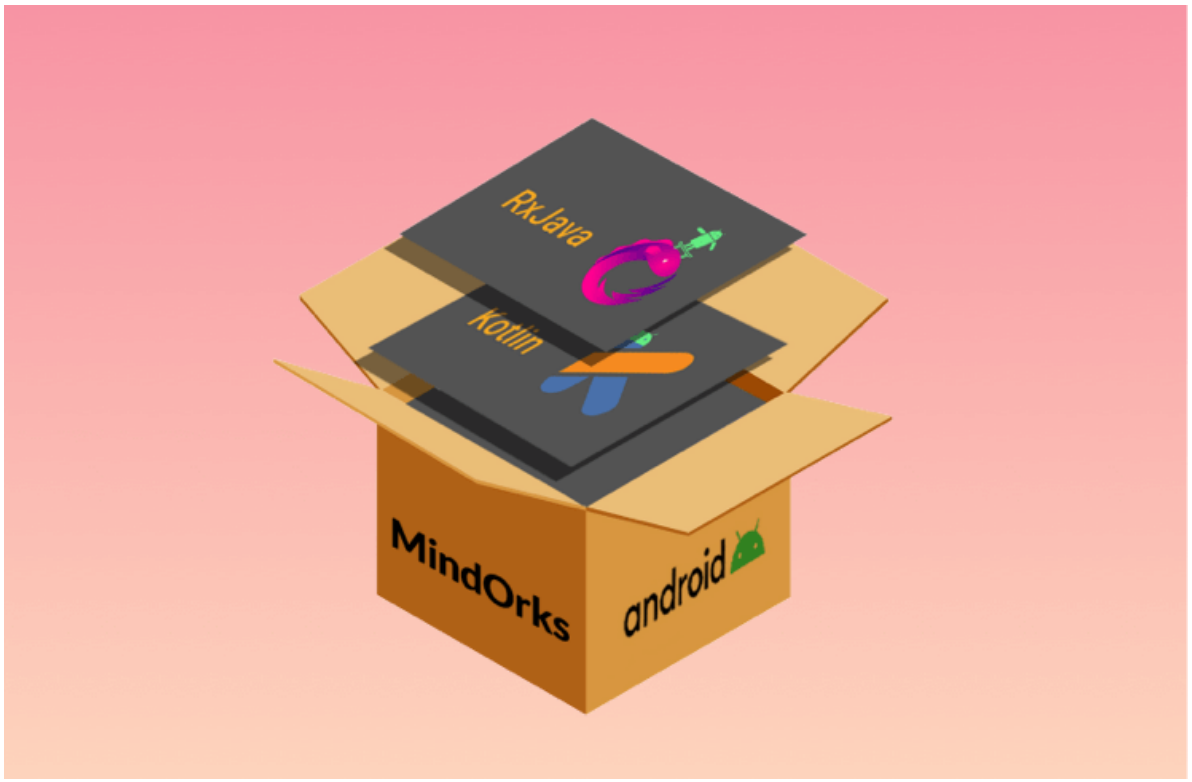
Complexity Analysis

Time Complexity: O(logN) (*We are sorting the list here but do you think the complexity will go upto logN?* ***Hint:*** *The array will be almost sorted.)*

Space Complexity: O(N)

Critical ideas to think!

- Can you directly sort the first list without creating a copy and update the tree? What will be the approach in that case and also the complexity



Solution II- Using Constant Space

Solution idea

In the above solution, we were comparing the entire sorted array which we got from the in-order traversal of the BST. Since we know that the in-order traversal of a BST will always give us a sorted list of elements, this problem can be reduced to a problem where two elements of a sorted array have been swapped.

For example:

```
In-order Traversal of the given BST {5, 7, 9, 10, 15}
In-order Traversal of the BST when 9 and 15 are
swapped {5, 7, 15, 10, 9}
```

Looking at the above example, you can conclude that maintaining two-pointers and updating the pointers whenever you encounter distortion in the sorted array will give you two swapped elements. This seems okay till now. But look at the below given in-order traversal of a BST and the in-order traversal after the two nodes are swapped.

```
In-order Traversal of the BST {5, 7, 9, 12, 16}
In-order Traversal of the BST after two nodes 7 and 9
are swapped
```

```
{5, 9, 7, 12, 16}
```

Do you still think that our previous approach will work in this case? No, right? So what should be our ideal approach then? We will maintain three-pointers first, middle, and last. Every time when we will update our first pointer (when the previous node's data is greater than than the current node), we will update the middle one also with the current node. If we encounter a similar situation anywhere else and our first pointer is already being updated we will update the last pointer with the current node. In case of adjacent elements, our last pointer will be NULL. And our first and middle pointers will be having the values of swapped elements.

Solution steps

- Create three nodes of the BST first, second and middle and initialize them all with NULL.
- Create one more node previous that will store the previous node to compare with the current node's data. Initially, it will also be NULL.
- We will make a util function for fixing our BST and pass the root and all the four newly created nodes.
- We will have a base condition in the util function that will check if the previous node's data is greater than the current(root) node's data. If this is true and the value of first is still NULL, we will update first with the previous node and middle with the current node(root).
- If the previous node's value is greater than the current node's value and first is not NULL, we will update last with the current(root) node.
- We will also update the previous with root in each case.
- We will recursively do this for the left and right subtrees.

Pseudo-code

```
void fixBstUtil(TreeNode root, TreeNode first,
TreeNode middle, TreeNode last, TreeNode previous)
{
    if(root is NULL) return
   fixBstUtil(root.left, first, middle, last,
previous)
    if(previous and previous.data > root.data)
    {
       if(first is NULL)
       {
          first = previous
          middle = root
       }
       else
           last = root
      previous = root
    }
    fixBstUtil(root.right, first, middle, last,
previous)
}
void fixBst(TreeNode root)
{
    TreeNode first, middle, last, previous
    first = middle = last = previous = NULL
    fixBstUtil(root, first, middle, last, previous)
    if(first and last)
       swap(first.data, last.data)  //function for
swapping values
```

```
    else
        swap(first.data, middle.data)
}
```

Complexity Analysis

Time Complexity: O(N)

Space Complexity: O(1)

Critical ideas to think!

- If the question does not have constraint that the BST should have its original structure, what else approach you can think of?

Comparison of Different Solutions Discussed

Approach	Time Complexity	Space Complexity
Solution I- Using Extra Space	**O(logN)**	**O(N)**
Solution II- Using Constant Space	**O(N)**	**O(1)**

Suggested Problems to Solve

- Check if a given binary tree can be converted to binary search tree by just swapping one element.
- Count number of BST nodes in a given range.
- Find the shortest distance between two nodes in a BST.
- Delete a node from a binary tree to make it a binary search tree.

Happy Coding!

Team AfterAcademy!!