Kth Smallest Element in a BST

Given a binary search tree, write a function `kthSmallest` to find the **k**th smallest element in it.

Note:
You may assume k is always valid, 1 ≤ k ≤ BST's total elements.

Follow up:
What if the BST is modified (insert/delete operations) often and you need to find the kth smallest frequently? How would you optimize the kthSmallest routine?

1. Try to utilize the property of a BST.
2. What if you could modify the BST node's structure?
3. The optimal runtime complexity is O(height of BST).

Credits:
Special thanks to @ts for adding this problem and creating all test cases.

Solution 1

Binary Search (dfs): most preferable

```
  public int kthSmallest(TreeNode root, int k) {
        int count = countNodes(root.left);
        if (k <= count) {
            return kthSmallest(root.left, k);
        } else if (k > count + 1) {
            return kthSmallest(root.right, k-1-count); // 1 is counted as current
node
        }

        return root.val;
    }

    public int countNodes(TreeNode n) {
        if (n == null) return 0;

        return 1 + countNodes(n.left) + countNodes(n.right);
    }
```

DFS in-order recursive:

```
    // better keep these two variables in a wrapper class
    private static int number = 0;
    private static int count = 0;

    public int kthSmallest(TreeNode root, int k) {
        count = k;
        helper(root);
        return number;
    }

    public void helper(TreeNode n) {
        if (n.left != null) helper(n.left);
        count--;
        if (count == 0) {
            number = n.val;
            return;
        }
        if (n.right != null) helper(n.right);
    }
```

DFS in-order iterative:

```java
public int kthSmallest(TreeNode root, int k) {
      Stack<TreeNode> st = new Stack<>();

      while (root != null) {
          st.push(root);
          root = root.left;
      }

      while (k != 0) {
          TreeNode n = st.pop();
          k--;
          if (k == 0) return n.val;
          TreeNode right = n.right;
          while (right != null) {
              st.push(right);
              right = right.left;
          }
      }

      return -1; // never hit if k is valid
}
```

written by angelvivienne original link here

Solution 2

If we could add a count field in the BST node class, it will take O(n) time when we calculate the count value for the whole tree, but after that, it will take O(logn) time when insert/delete a node or calculate the kth smallest element.

```
   public class Solution {
        public int kthSmallest(TreeNode root, int k) {
            TreeNodeWithCount rootWithCount = buildTreeWithCount(root);
            return kthSmallest(rootWithCount, k);
        }

        private TreeNodeWithCount buildTreeWithCount(TreeNode root) {
            if (root == null) return null;
            TreeNodeWithCount rootWithCount = new TreeNodeWithCount(root.val);
            rootWithCount.left = buildTreeWithCount(root.left);
            rootWithCount.right = buildTreeWithCount(root.right);
            if (rootWithCount.left != null) rootWithCount.count += rootWithCount.
left.count;
            if (rootWithCount.right != null) rootWithCount.count += rootWithCount
.right.count;
            return rootWithCount;
        }

        private int kthSmallest(TreeNodeWithCount rootWithCount, int k) {
            if (k <= 0 || k > rootWithCount.count) return -1;
            if (rootWithCount.left != null) {
                if (rootWithCount.left.count >= k) return kthSmallest(rootWithCou
nt.left, k);
                if (rootWithCount.left.count == k-1) return rootWithCount.val;
                return kthSmallest(rootWithCount.right, k-1-rootWithCount.left.co
unt);
            } else {
                if (k == 1) return rootWithCount.val;
                return kthSmallest(rootWithCount.right, k-1);
            }
        }

        class TreeNodeWithCount {
            int val;
            int count;
            TreeNodeWithCount left;
            TreeNodeWithCount right;
            TreeNodeWithCount(int x) {val = x; count = 1;};
        }
    }
```

written by WHJ425 original link here

Solution 3

Go inorder and decrease `k` at each node. Stop the whole search as soon as `k` is zero, and then the k-th element is immediately returned all the way to the recursion top and to the original caller.

Try the left subtree first. If that made `k` zero, then its answer is the overall answer and we return it right away. Otherwise, decrease `k` for the current node, and if that made `k` zero, then we return the current node's value right away. Otherwise try the right subtree and return whatever comes back from there.

```
int kthSmallest(TreeNode* root, int& k) {
    if (root) {
        int x = kthSmallest(root->left, k);
        return !k ? x : !--k ? root->val : kthSmallest(root->right, k);
    }
}
```

You might notice that I changed `k` from `int` to `int&` because I didn't feel like adding a helper just for that and the OJ doesn't mind. Oh well, here is that now:

```
int kthSmallest(TreeNode* root, int k) {
    return find(root, k);
}
int find(TreeNode* root, int& k) {
    if (root) {
        int x = find(root->left, k);
        return !k ? x : !--k ? root->val : find(root->right, k);
    }
}
```

written by StefanPochmann original link here