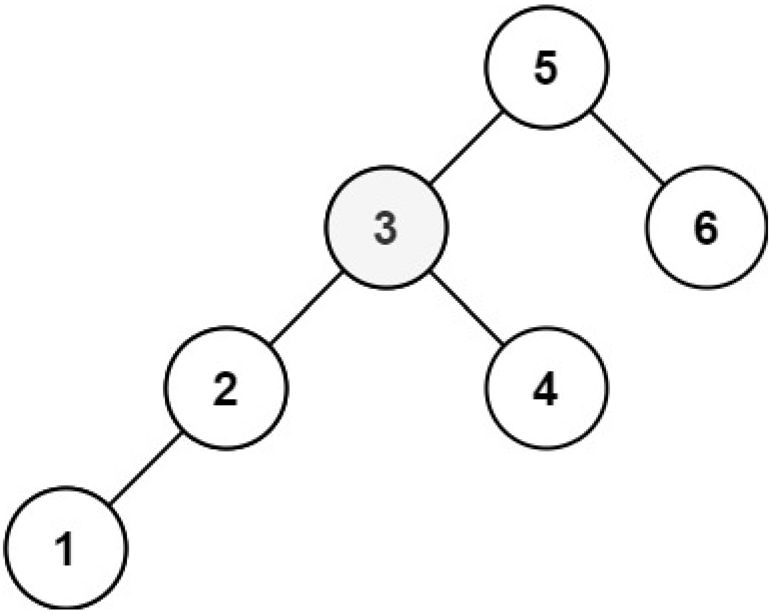
230. Kth Smallest Element in a BST

Given the `root` of a binary search tree, and an integer `k`, return *the* `kth` (**1-indexed**) *smallest element in the tree*



Input: root = [5,3,6,2,4,null,null,1], k = 3
Output: 3
Follow up: If the BST is modified often (i.e., we can
do insert and delete operations) and you need to find the kth smallest
frequently, how would you optimize?
1. My attempt:

```
def kthSmallest(self, root: TreeNode, k: int) -> int:
        if root is None:
            return
        ans = [0]
        count = [0]
        self.helper(root,k,ans,count)
        return ans[0]

    def helper(self,root,k,ans,count):
        if root is None:
            return
        self.helper(root.left,k,ans,count)
        count[0] = count[0]+1
        if count[0]==k:
```

```
            ans[0] = root.val
        self.helper(root.right,k,ans,count)
```

Sol:1.

```
def kthSmallest(self, root, k):
        """
        :type root: TreeNode
        :type k: int
        :rtype: int
        """
        stack = []

        while True:
            while root:
                stack.append(root)
                root = root.left
            root = stack.pop()
            k -= 1
            if not k:
                return root.val
```

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?

Insert and delete in a BST were discussed last week, the time complexity of these operations is O(H)\mathcal{O}(H)O(H), where HHH is a height of binary tree, and H=logNH = \log NH=logN for the balanced tree.

Hence without any optimisation insert/delete + search of kth element has O(2H+k)\mathcal{O}(2H + k)O(2H+k) complexity. How to optimise that?

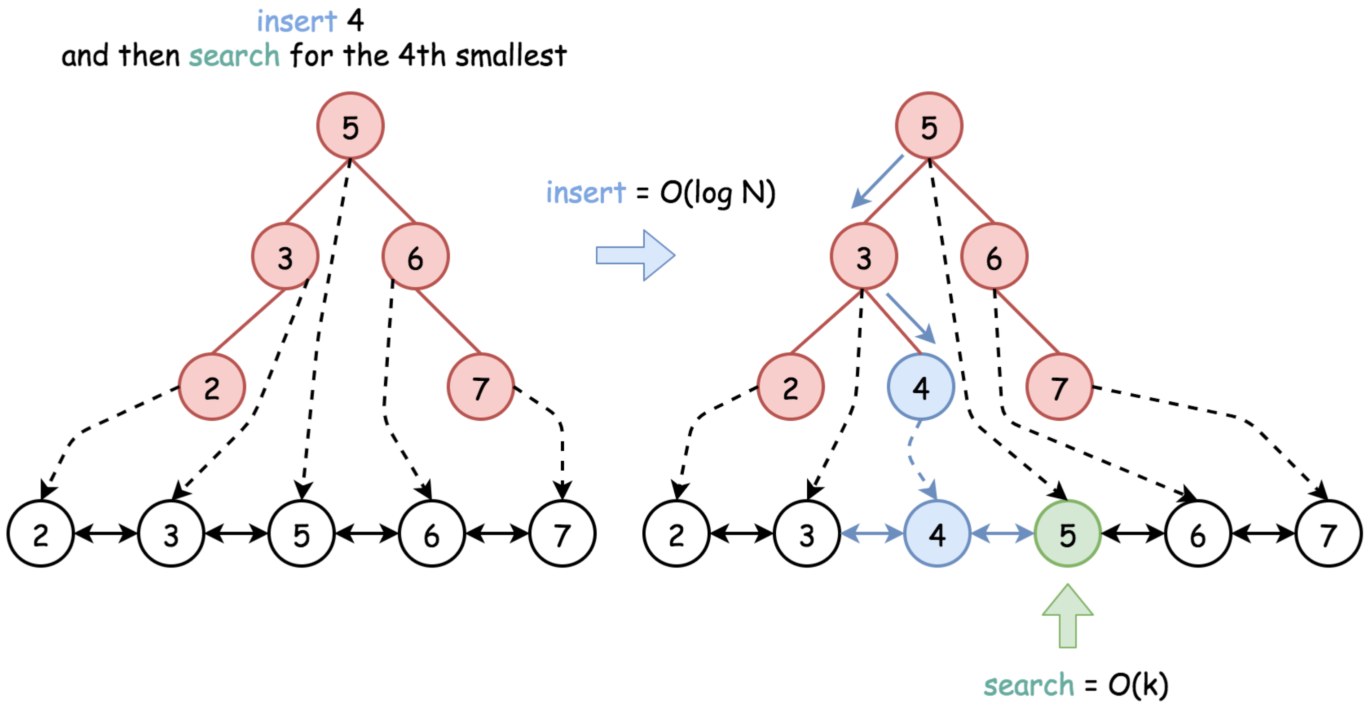
That's a design question, basically we're asked to implement a structure which contains a BST inside and optimises the following operations :

- Insert
- Delete
- Find kth smallest

Seems like a database description, isn't it? Let's use here the same logic as for LRU cache design, and combine an indexing structure (we could keep BST here) with a double linked list.

Such a structure would provide:

- $\mathcal{O}(H)$ time for the insert and delete.
- $\mathcal{O}(k)$ for the search of kth smallest.



The overall time complexity for insert/delete + search of kth smallest is $\mathcal{O}(H + k)$ instead of $\mathcal{O}(2H + k)$.

Complexity Analysis

- Time complexity for insert/delete + search of kth smallest: $\mathcal{O}(H + k)$, where H is a tree height. $\mathcal{O}(\log N + k)$ in the average case, $\mathcal{O}(N + k)$ in the worst case.
- Space complexity : $\mathcal{O}(N)$ to keep the linked list.