LeetCode 272

https://leetcode.com/problems/closest-binary-search-tree-value-ii/description/

Yifeng Zeng

Description

272. Closest Binary Search Tree Value II

Given a non-empty binary search tree and a target value, find k values in the BST that are closest to the target.

Note:

Given target value is a floating point.

You may assume k is always valid, that is: k ≤ total nodes.

You are guaranteed to have only one unique set of k values in the BST that are closest to the target.

Idea Report

To find k values in the BST that are closest to the target, my primitive idea is to maintain a window size of k. Since it is a BST, the inorder traversal of the BST is a increasing (or non-decreaing) sequence. So we can do an inorder traversal and update the window. We can use a arraydeque as the window, each time we add one element to the last of the arraydeque, and check the first and last element, and remove the element that has greater difference to the target, in order to keep the size of arraydeque as k. At the end, we just return the arraydeque as a list. This is basically O(n) time and O(k) space.

Code:

```
class Solution {
    // AC
    public List<Integer> closestKValues(TreeNode root, double target, int k) {
        Deque<Integer> q = new LinkedList<>();
        Deque<TreeNode> stack = new ArrayDeque<>();
        TreeNode cur = root;
        while (!stack.isEmpty() || cur != null) {
            while (cur != null) {
                stack.push(cur);
                cur = cur.left;
            }
            cur = stack.pop();
            q.offerLast(cur.val);
            if (q.size() > k) {
                int first = q.peekFirst();
                int last = q.peekLast();
                if (Math.abs((double) (first) - target)
                    > Math.abs((double) (last) - target)) {
                    q.pollFirst();
                } else {
                    q.pollLast();
                }
            }
            cur = cur.right;
        }

        List<Integer> list = new ArrayList<Integer>();
        list.addAll(q);
        return list;
    }
}
```

Since we were usin O(n) time and O(k) space, and we might want try to optimize the time to O(H), where H is the height of the BST. If it is a sorted array, we can have two pointers, one points to the closest element that is smaller than the target. Another pointer points to the closest element that is equal or larger than the target. For example we have [1,2,3,4,5,6], target is 3.1, k = 2, so left points to 3, right pionts to 4. We compare the element these two pointers point, put the element that is closer to target into result and move that pointer away by one index, until the result's size is k. So we put [3] in result, move left by one. Now compare [2] and [4], we put [4] in result and get [3,4], now result size is k = 2, we just return. Similarly in the tree, we are not able to get those elements directly by moving index, instead, we need two stacks so that we can trace back to the previous/next node in the in-order sequence. And use these stacks' top elements as the pointer. Similaryly, we compare those two nodes, add the closer one to the target into the result until the result size is k. We will have a prevStack to enumerate the nodes that is equal or smaller than taret. And we will have a nextStack to enumerate the nodes that is larger than the target. To initialize the nextStack, we search from root, if it is smaller than target, we search root.right, if it is larger than target, we put it in nextStack and search root.left, because that node may be one candidate that is larger than target. If it is equal to root, we put it in nextStack and return, because we want to find the first element that is equal or larger than target. Similary to prevStack, if root is smaller we put it in prevStack and search root.right because there may be more candidates on right. If root is larger or equal to target, we search root.left, because we want to find the first element that is smaller than target. After initializing the two pointers, we compare those two nodes, put the smaller one in the result and move that pointer away by one offset, until our result is k and return.

Code:

```
class Solution {
    // AC
    public List<Integer> closestKValues(TreeNode root, double target, int k) {
        Deque<TreeNode> nextStack = createNextStack(root, target);
        Deque<TreeNode> prevStack = createPrevStack(root, target);
        List<Integer> res = new ArrayList<>();

        while (res.size() < k) {
            if (nextStack.isEmpty()) {
                res.add(getPrevNode(prevStack));
            } else if (prevStack.isEmpty()) {
                res.add(getNextNode(nextStack));
            } else {
                double next = Math.abs(nextStack.peek().val - target);
                double prev = Math.abs(prevStack.peek().val - target);
                if (next < prev) {
                    res.add(getNextNode(nextStack));
                } else {
                    res.add(getPrevNode(prevStack));
                }
            }
        }
        return res;
    }

    private int getNextNode(Deque<TreeNode> nextStack) {
        TreeNode cur = nextStack.pop();
        int res = cur.val;
        cur = cur.right;
        while (cur != null) {
            nextStack.push(cur);
            cur = cur.left;
        }
        return res;
    }
    private int getPrevNode(Deque<TreeNode> prevStack) {
        TreeNode cur = prevStack.pop();
        int res = cur.val;
        cur = cur.left;
        while (cur != null) {
            prevStack.push(cur);
            cur = cur.right;
        }
        return res;
    }

    private Deque<TreeNode> createNextStack(TreeNode root, double target) {
        Deque<TreeNode> nextStack = new ArrayDeque<>();
        while (root != null) {
            if (root.val == target) {
                nextStack.push(root);
                return nextStack;
            }
            if (root.val < target) {
                root = root.right;
            } else {
                nextStack.push(root);
                root = root.left;
            }
        }
        return nextStack;
    }

    private Deque<TreeNode> createPrevStack(TreeNode root, double target) {
        Deque<TreeNode> prevStack = new ArrayDeque<>();
        while (root != null) {
            if (root.val < target) {
                prevStack.push(root);
                root = root.right;
            } else {
                root = root.left;
            }
        }
        return prevStack;
    }
}
```

Summary

- Two pointers move away to each other (相离双指针)