<https://www.lintcode.com/problem/901/>

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

* 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.

**Example**

**Example 1:**

Input:

{1}

0.000000

1

Output:

[1]

Explanation：

Binary tree {1}, denote the following structure:

1

**Example 2:**

Input:

{3,1,4,#,2}

0.275000

2

Output:

[1,2]

Explanation：

Binary tree {3,1,4,#,2}, denote the following structure:

3

/ \

1 4

\

2

**Challenge**

Assume that the BST is balanced, could you solve it in less than O(n) runtime (where n = total nodes)?

**Attempt 1: 2022-12-10**

**Solution 1:  Brute force based on L270/Lint900.Closest Binary Search Tree Value and PriorityQueue (30 min)**

**Note: minPQ or maxPQ doesn't matter, both can implement pick only top k minimum delta nodes**

/\*\*

\* Definition of TreeNode:

\* public class TreeNode {

\* public int val;

\* public TreeNode left, right;

\* public TreeNode(int val) {

\* this.val = val;

\* this.left = this.right = null;

\* }

\* }

\*/

public class Solution {

/\*\*

\* @param root: the given BST

\* @param target: the given target

\* @param k: the given k

\* @return: k values in the BST that are closest to the target

\* we will sort your return value in output

\*/

public List<Integer> closestKValues(TreeNode root, double target, int k) {

PriorityQueue<Integer> minPQ = new PriorityQueue<Integer>(k, (a, b) -> a - b);

inorder(root, target, minPQ, k);

List<Integer> result = new ArrayList<Integer>();

while(!minPQ.isEmpty()) {

result.add(minPQ.poll());

}

return result;

}

private void inorder(TreeNode root, double target, PriorityQueue<Integer> minPQ, int k) {

if(root == null) {

return;

}

inorder(root.left, target, minPQ, k);

// When minPQ size smaller than k, keep adding any node value

// When minPQ size no less than k, compare delta between current visiting node value

// and target with existing minPQ peek value and target, if current delta is remove

// existing peek value and update with current visiting node value

if(minPQ.size() < k) {

minPQ.offer(root.val);

} else {

if(Math.abs(root.val - target) < Math.abs(target - minPQ.peek())) {

minPQ.poll();

minPQ.offer(root.val);

}

}

inorder(root.right, target, minPQ, k);

}

}

Time complexity: O(k + (n - k)logk)

Space complexity: O(n)

**Refer to**

<http://buttercola.blogspot.com/2015/09/leetcode-closest-binary-search-tree_8.html>

**Brute-force solution:**

The straight-forward solution would be to use a heap. We just treat the BST just as a usual array and do a in-order traverse. Then we compare the current element with the minimum element in the heap, the same as top k problem.

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

public class Solution {

private PriorityQueue&lt;Integer&gt; minPQ;

private int count = 0;

public List&lt;Integer&gt; closestKValues(TreeNode root, double target, int k) {

minPQ = new PriorityQueue&lt;Integer&gt;(k);

List&lt;Integer&gt; result = new ArrayList&lt;Integer&gt;();

inorderTraverse(root, target, k);

// Dump the pq into result list

for (Integer elem : minPQ) {

result.add(elem);

}

return result;

}

private void inorderTraverse(TreeNode root, double target, int k) {

if (root == null) {

return;

}

inorderTraverse(root.left, target, k);

if (count &lt; k) {

minPQ.offer(root.val);

} else {

if (Math.abs((double) root.val - target) &lt; Math.abs((double) minPQ.peek() - target)) {

minPQ.poll();

minPQ.offer(root.val);

}

}

count++;

inorderTraverse(root.right, target, k);

}

}

**Solution 2:  Two Stacks (360 min)**

/\*\*

\* Definition of TreeNode:

\* public class TreeNode {

\* public int val;

\* public TreeNode left, right;

\* public TreeNode(int val) {

\* this.val = val;

\* this.left = this.right = null;

\* }

\* }

\*/

public class Solution {

/\*\*

\* @param root: the given BST

\* @param target: the given target

\* @param k: the given k

\* @return: k values in the BST that are closest to the target

\* we will sort your return value in output

\*/

public List<Integer> closestKValues(TreeNode root, double target, int k) {

List<Integer> result = new ArrayList<Integer>();

Stack<TreeNode> prev = new Stack<TreeNode>();

Stack<TreeNode> next = new Stack<TreeNode>();

TreeNode node = root;

while(node != null) {

if(node.val < target) {

prev.push(node);

node = node.right;

} else {

next.push(node);

node = node.left;

}

}

while(result.size() < k) {

double distp = prev.isEmpty() ? Integer.MAX\_VALUE : Math.abs(prev.peek().val - target);

double distn = next.isEmpty() ? Integer.MAX\_VALUE : Math.abs(next.peek().val - target);

if(distp < distn) {

result.add(0, prev.peek().val);

getPrev(prev);

} else {

result.add(next.peek().val);

getNext(next);

}

}

return result;

}

// FIND THE PREDECESSOR NODE OF A BINARY SEARCH TREE

private void getPrev(Stack<TreeNode> stack) {

TreeNode l = stack.pop().left;

while(l != null) {

stack.push(l);

l = l.right;

}

}

// FIND THE SUCCESSOR NODE OF A BINARY SEARCH TREE

private void getNext(Stack<TreeNode> stack) {

TreeNode r = stack.pop().right;

while(r != null) {

stack.push(r);

r = r.left;

}

}

}

**Refer to**

<https://www.lintcode.com/problem/901/solution/16651>

对令狐老师和其他同学的解法小小总结一下, O(h + k)的时间复杂度。h为树的高度，平均为logn。

【确认条件】

（1）沟通BST的定义。

（2）确认是否需要判断tree和k是否valid。

（3）确认不会存在两个与target距离相等的值，否则输出list的时候还得判断哪一个放在前面。

（4）确认k是否小于等于tree中的节点数（虽然解法中遇到这种情况会通过break跳出）。

【解题思路】

（1）**通过get\_stacks()虚拟寻找target的插入位置，并将一路上经过的点根据值的大小分别放入prev\_stack和next\_stack。用两个栈的好处是：之后在实现get\_next()和get\_prev()的时候会相对简单一些，不需要像完整版BST iterator那么复杂。**

（2）实现get\_next()，利用next\_stack寻找next\_value。在一般的BST iterator中，寻找下一节点的算法是：如果当前点存在右子树，那么就是右子树中一直向左走到底的那个点；如果当前点不存在右子树，则对到达当前点的路径进行反向遍历（一直pop stack），寻找第一个（离当前点最近的）左拐的点。

e.g

4

/ \

2 6

/ \ /

1 3 5

比如4存在右子树(node.right != null)，那么就是右子树一直向左走到底的那个点(node = node.right -> loop node = node.left)，这里是5

比如3不存在右子树(node.right == null)，则对到达当前点的路径进行反向遍历(一直pop stack, loop node = stack.pop())，寻找第一个(离当前点最近的)左拐的点(node.left != null)，这里是4

然而在本题中，因为已经分离prev\_stack和next\_stack，所以在当前节点不存在右子树的情况下，当前节点在next\_stack中的前一个位置自然就是要找的下一个点。因此代码中只需处理当前节点存在右子树时的情况，即先取当前节点的右子树，再一路向左走到底。

（3）实现get\_prev()，利用prev\_stack寻找prev\_value。对get\_next()的处理方式取反，即先取当前节点的左子树，再一路向右走到底。若不存在左子树，在pop出当前节点后，stack[-1]自然处于下一个prev节点的位置。

（4）for循环k次，每次比较prev\_stack和next\_stack栈顶节点的值，把与target距离近的那个放进results中。

【实现要点】

（1）实现get\_stacks()的时候，在把节点分入两个栈的时候注意思考一下，别把大小写，左右子树弄反了。另外对于本题，不需要对root.val == target的情况专门处理。

（2）实现get\_next()和get\_prev()注意细节（完整版BST iterator其实需要背诵，本题中再对其简化）。

（3）比较大小的时候引入sys.maxsize作为异常情况处理。

【复杂度】

时间复杂度：O(h + k)，O(h)来自于对树的搜索，O(k)是获取k个结果。

空间复杂度：O(h)

class Solution:

def closestKValues(self, root, target, k):

results = []

if root is None or k == 0:

return results

next\_stack, prev\_stack = self.get\_stacks(root, target)

for \_ in range(k):

if len(next\_stack) == 0 and len(prev\_stack) == 0:

break

next\_diff = sys.maxsize if len(next\_stack) == 0 else abs(next\_stack[-1].val - target)

prev\_diff = sys.maxsize if len(prev\_stack) == 0 else abs(prev\_stack[-1].val - target)

if next\_diff < prev\_diff:

results.append(self.get\_next(next\_stack))

else:

results.append(self.get\_prev(prev\_stack))

return results

def get\_stacks(self, root, target):

next\_stack, prev\_stack = [], []

while root:

if root.val < target:

prev\_stack.append(root)

root = root.right

else:

next\_stack.append(root)

root = root.left

return next\_stack, prev\_stack

def get\_next(self, next\_stack):

value = next\_stack[-1].val

node = next\_stack.pop().right

while node:

next\_stack.append(node)

node = node.left

return value

def get\_prev(self, prev\_stack):

value = prev\_stack[-1].val

node = prev\_stack.pop().left

while node:

prev\_stack.append(node)

node = node.right

return value

**Refer to**

<https://www.lintcode.com/problem/901/solution/16534>

使用令狐老师的基本思路重写，让代码更易读。

思路等同于从指定节点开始分别向前和向后遍历，直到找到k个最接近target的节点。

使用prev和next两个栈分别记录前驱和后继，goPrev相当于反向中序遍历，goNext相当于正向中序遍历。

public class Solution {

public List<Integer> closestKValues(TreeNode root, double target, int k) {

Stack<TreeNode> next = new Stack<TreeNode>();

Stack<TreeNode> prev = new Stack<TreeNode>();

TreeNode node = root;

// find the nodes closest to target

while (node != null) {

if (node.val < target) {

prev.push(node);

node = node.right;

} else {

next.push(node);

node = node.left;

}

}

List<Integer> ret = new LinkedList<Integer>();

while (ret.size() < k) {

double distp = prev.isEmpty() ? Integer.MAX\_VALUE : Math.abs(prev.peek().val - target);

double distn = next.isEmpty() ? Integer.MAX\_VALUE : Math.abs(next.peek().val - target);

// compare and find the closest node, and move the corresponding stack.

if (distp < distn) {

ret.add(0, prev.peek().val);

goPrev(prev);

} else {

ret.add(next.peek().val);

goNext(next);

}

}

return ret;

}

private void goNext(Stack<TreeNode> st) {

TreeNode r = st.pop().right;

while (r != null) {

st.push(r);

r = r.left;

}

}

private void goPrev(Stack<TreeNode> st) {

TreeNode l = st.pop().left;

while (l != null) {

st.push(l);

l = l.right;

}

}

}

**Step by step process**

/\*\*

\* k=3

\* 4

\* / \

\* 2 5

\* / \

\* 1 3

\*/

========================================

Round 1:

prev next

----- -----

3 4 distp = Math.abs(prev.peek().val - target) = |3 - 3.714286| = 0.714286

----- ----- ===> distn = Math.abs(next.peek().val - target) = |4 - 3.714286| = 0.285714

2

-----

distp > distn

result.add(next.peek().val) = {4}

goNext(next) = goNext({4})

next next next

----- ----- -----

4 ===> pop() ===> empty ===> 4.right = 5 ===> push(5) ===> 5

----- ----- -----

========================================

Round 2:

prev next

----- -----

3 5 distp = Math.abs(prev.peek().val - target) = |3 - 3.714286| = 0.714286

----- ----- ===> distn = Math.abs(next.peek().val - target) = |5 - 3.714286| = 1.285714

2

-----

distp < distn

result.add(0, prev.peek().val) = {3, 4}

goPrev(prev) = goPrev({2, 3})

prev prev

----- -----

3 2

----- ===> pop() ===> ----- ===> 3.left = null

2

-----

========================================

Round 3:

prev next

----- -----

2 5 distp = Math.abs(prev.peek().val - target) = |2 - 3.714286| = 1.714286

----- ----- ===> distn = Math.abs(next.peek().val - target) = |5 - 3.714286| = 1.285714

distp > distn

result.add(next.peek().val) = {3, 4, 5}

goNext(next) = goNext({5})

next next

----- -----

5 ===> pop() ===> empty ===> 4.right = null

----- -----

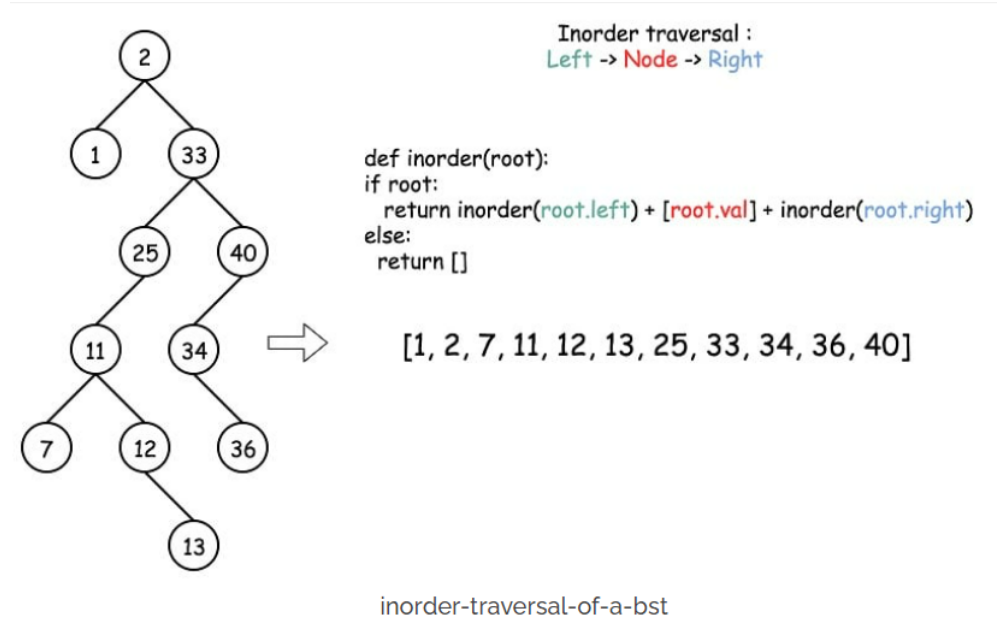
========================================

**Important tips: Finding the Predecessor and Successor Node of a Binary Search Tree**

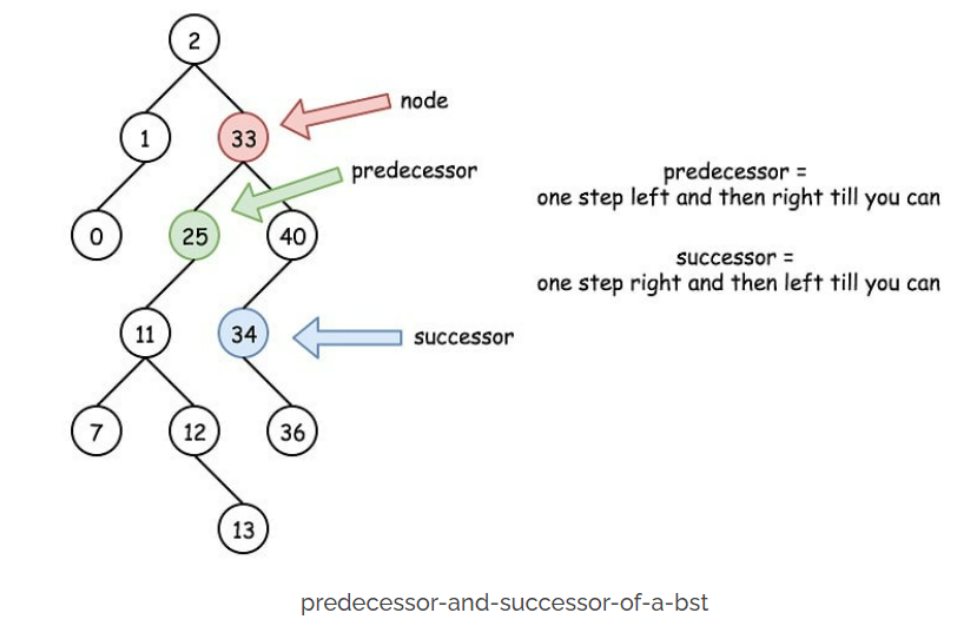
<https://helloacm.com/finding-the-predecessor-and-successor-node-of-a-binary-search-tree/>

A Binary Search Tree (BST) is a commonly used data structure that can be used to search an item in O(LogN) time. A BST should have the following characteristics: its left nodes are smaller than the root and its right nodes are larger than the root.

If we perform an [inorder traversal](https://helloacm.com/how-to-do-binary-tree-inorder-traversal-in-cc/): left nodes first, current node, and then right nodes – we will have a fully sorted sequence.



To find the Predecessor or Successor Node of a BST – we can perform the following algorithms:



### FIND THE PREDECESSOR NODE OF A BINARY SEARCH TREE

The predecessor node is the largest node that is smaller than the root (current node) – thus it is on the left branch of the [Binary Search Tree](https://helloacm.com/how-to-construct-binary-search-tree-from-preorder-traversal-c-and-java/), and the rightmost leaf (largest on the left branch).

And below is the Java implementation to get the predecessor node of a Binary Search Tree:

public int predecessor(TreeNode root) {

if (root == null) return null;

root = root.left;

while (root.right != null) root = root.right;

return root;

}

### FIND THE SUCCESSOR NODE OF A BINARY SEARCH TREE

On the other hand, the successor node is the smallest node that is bigger than the root/current – thus it is on the right branch of the BST, and also on the leftmost leaf (smallest on the right branch).

Java method to get the successor:

public int successor(TreeNode root) {

if (root == null) return null;

root = root.right;

while (root.left != null) root = root.left;

return root;

}

All implementation of finding [successor](https://helloacm.com/teaching-kids-programming-algorithms-to-find-the-inorder-successor-of-a-binary-search-tree/) or predecessor takes O(1) constant space and run O(N) time (when BST is just a degraded linked list) – however, on average, the complexity is O(LogN) where the binary tree is balanced.

Finding successor or predecessor is very useful – for example, we can use this to [delete a node in a binary search tree](https://helloacm.com/how-to-delete-a-node-from-a-binary-search-tree/).