<https://www.lintcode.com/problem/448/>

**Description**

Given a binary search tree ([See Definition](http://www.lintcode.com/problem/validate-binary-search-tree/)) and a node in it, find the in-order successor of that node in the BST.

If the given node has no in-order successor in the tree, return null.

It's guaranteed *p* is one node in the given tree. (You can directly compare the memory address to find p)

**Example**

**Example 1:**

Input: {1,#,2}, node with value 1

Output: 2

Explanation:

1

\

2

**Example 2:**

Input: {2,1,3}, node with value 1

Output: 2

Explanation:

2

/ \

1 3

[Binary Tree Representation](https://www.lintcode.com/help/binary-tree-representation/)

**Challenge**

O(h), where h is the height of the BST.

**Attempt 1: 2022-12-20**

**Solution 1: Recursive traversal (10 min)**

**Style 1: Have return type**

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

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

\* }

\*/

public class Solution {

/\*

\* @param root: The root of the BST.

\* @param p: You need find the successor node of p.

\* @return: Successor of p.

\*/

public TreeNode inorderSuccessor(TreeNode root, TreeNode p) {

if(root == null) {

return null;

}

if(root.val > p.val) {

TreeNode left = inorderSuccessor(root.left, p);

return left != null ? left : root;

} else {

return inorderSuccessor(root.right, p);

}

}

}

Time Complexity : O(N)

Space Complexity : O(N)

**Style 2: Global variable and void return**

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

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

\* }

\*/

public class Solution {

/\*

\* @param root: The root of the BST.

\* @param p: You need find the successor node of p.

\* @return: Successor of p.

\*/

TreeNode successor = null;

public TreeNode inorderSuccessor(TreeNode root, TreeNode p) {

helper(root, p);

return successor;

}

private void helper(TreeNode root, TreeNode p) {

if(root == null) {

return;

}

if(root.val > p.val) {

successor = root;

helper(root.left, p);

} else {

helper(root.right, p);

}

}

}

Time Complexity : O(N)

Space Complexity : O(N)

**Refer to**

<https://www.lintcode.com/problem/448/solution/18213>

首先要确定中序遍历的后继:

* 如果该节点有右子节点, 那么后继是其右子节点的子树中最左端的节点
* 如果该节点没有右子节点, 那么后继是离它最近的祖先, 该节点在这个祖先的左子树内.

使用递归实现:

* 如果根节点小于或等于要查找的节点, 直接进入右子树递归
* 如果根节点大于要查找的节点, 则暂存左子树递归查找的结果, 如果是 null, 则直接返回当前根节点; 反之返回左子树递归查找的结果.

在递归实现中, 暂存左子树递归查找的结果就相当于循环实现中维护的祖先节点.

\* 4

\* / \

\* 2 6

\* / \ /

\* 1 3 5

Explanation: return (left != null) ? left : root;

let's take the successor for example, basically we always want to find p's closest node (in inorder traversal) and the node's value is larger than p's value, right?

That node can either be p's parent or the smallest node in p' right branch. When the code runs into the else block (root.val > p.val), that means the current root is either p's parent or a node in p's right branch.

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

If it's p's parent node, there are two scenarios:

1. p doesn't have right child, in this case, the recursion will eventually return null, so p's parent is the successor; (e.g find root = 4 as parent of p = 3)

2. p has right child, then the recursion will return the smallest node in the right sub tree, and that will be the answer. (e.g find root = 4 as parent of p = 2)

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

If it's p's right child, there are two scenarios:

1. the right child has left sub tree, eventually the smallest node from the left sub tree will be the answer; (e.g find root = 6 as right child of p = 4)

2. the right child has no left sub tree, the recursion will return null, then the right child (root) is our answer. (e.g find root = 3 as right child of p = 2)

**Refer to**

<https://www.lintcode.com/problem/448/solution/18539>

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

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

\* }

\*/

public class Solution {

TreeNode larger = null;

/\*

\* @param root: The root of the BST.

\* @param p: You need find the successor node of p.

\* @return: Successor of p.

\*/

public TreeNode inorderSuccessor(TreeNode root, TreeNode p) {

// write your code here

search(root, p);

return larger;

}

private void search(TreeNode node, TreeNode p) {

if (node == null) {

return;

}

if (p.val < node.val) {

larger = node;

search(node.left, p);

} else {

search(node.right, p);

}

}

}

**Solution 2: Iterative traversal (10 min)**

**Style 1: Standard inorder iterative traversal**

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

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

\* }

\*/

public class Solution {

/\*

\* @param root: The root of the BST.

\* @param p: You need find the successor node of p.

\* @return: Successor of p.

\*/

public TreeNode inorderSuccessor(TreeNode root, TreeNode p) {

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

while(root != null || !stack.isEmpty()) {

while(root != null) {

stack.push(root);

root = root.left;

}

root = stack.pop();

if(root.val > p.val) {

return root;

}

root = root.right;

}

return null;

}

}

Time Complexity : O(N)

Space Complexity : O(N)

**Style 2:**

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

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

\* }

\*/

public class Solution {

/\*

\* @param root: The root of the BST.

\* @param p: You need find the successor node of p.

\* @return: Successor of p.

\*/

public TreeNode inorderSuccessor(TreeNode root, TreeNode p) {

TreeNode successor = null;

while(root != null) {

if(root.val > p.val) {

successor = root;

root = root.left;

} else {

root = root.right;

}

}

return successor;

}

}

Time Complexity : O(N)

Space Complexity : O(N)

**Refer to**

<https://www.lintcode.com/problem/448/solution/18213>

使用循环实现:

* 查找该节点, 并在该过程中维护上述性质的祖先节点
* 查找到后, 如果该节点有右子节点, 则后继在其右子树内; 否则后继就是维护的那个祖先节点

<https://www.lintcode.com/problem/448/solution/18309>

非递归版本的解法, 用node当指标遍历，直到node变为Null为止:

首先从node = root开始, 如果node.val <= p.val, 表示successor在右子树,node往右子树移动。

若否, 则先将successor设为node, 并让node往左子树移动。之后若又出现node.val <= p.val的情况,

则successor还会被更新。

def inorderSuccessor(self, root, p):

# find closest successor candidate

if root == None or p == None:

return None

node = root

successor = None

while node != None:

if node.val <= p.val: # successor in right child

node = node.right

else: # node.val > p.val

successor = node

node = node.left

return successor