<https://www.lintcode.com/problem/915/>

**Description**

Given a binary search tree and a node in it, find the in-order predecessor of that node in the BST.

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

**Example**

**Example1**

Input: root = {2,1,3}, p = 1

Output: null

**Example2**

Input: root = {2,1}, p = 2

Output: 1

**Attempt 1: 2023-01-03**

**Solution 1:  Recursive traversal (30 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 p: the given node

\* @return: the in-order predecessor of the given node in the BST

\*/

TreeNode result = null;

public TreeNode inorderPredecessor(TreeNode root, TreeNode p) {

if(root == null) {

return null;

}

helper(root, p);

return result;

}

private void helper(TreeNode root, TreeNode p) {

if(root == null) {

return;

}

helper(root.left, p);

// Recursively update result in recursion call till find

// most close one smaller than p, since its inorder

// traversal, the last update is guaranteed to be the most

// close one smaller than p

if(root.val < p.val) {

result = root;

}

helper(root.right, p);

}

}

Time Complexity: O(n)

Space Complexity: O(n)

**Refer to**

<https://www.lintcode.com/problem/915/solution/17855>

Just inorder traversal. Probably the most clean code.

Keep recording the node with value smaller than the value of the given node.

class Solution {

private:

TreeNode \* predecessor = NULL;

public:

/\*\*

\* @param root: the given BST

\* @param p: the given node

\* @return: the in-order predecessor of the given node in the BST

\*/

TreeNode \* inorderPredecessor(TreeNode \* root, TreeNode \* p) {

helper(root, p);

return predecessor;

}

void helper(TreeNode\* root, TreeNode\* p) {

if (root == NULL) return;

helper(root->left, p);

if (root->val < p->val) predecessor = root;

helper(root->right, p);

}

};

**Another two recursive traversal styles:**

**Refer to**

<https://www.lintcode.com/problem/915/solution/16963>

简单易懂精简的dfs方法

模板同样适用于inorder successor

只需要修改if else的顺序即可

public class Solution {

/\*\*

\* @param root: the given BST

\* @param p: the given node

\* @return: the in-order predecessor of the given node in the BST

\*/

TreeNode pre = null;

public TreeNode inorderPredecessor(TreeNode root, TreeNode p) {

// write your code here

dfs(root, p);

return pre;

}

private void dfs(TreeNode root, TreeNode p) {

if (root == null) {

return;

}

if (p.val > root.val) {

pre = root;

dfs(root.right, p);

} else {

dfs(root.left, p);

}

}

}

**Refer to**

<https://www.lintcode.com/problem/915/solution/19090>

public TreeNode inorderPredecessor(TreeNode root, TreeNode p) {

// recursive way

if (root == null) {

return root;

}

if (p.val <= root.val) {

return inorderPredecessor(root.left, p);

}

TreeNode node = inorderPredecessor(root.right, p);

return node == null ? root : node;

}

**Solution 2:  Iterative traversal (10 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 p: the given node

\* @return: the in-order predecessor of the given node in the BST

\*/

public TreeNode inorderPredecessor(TreeNode root, TreeNode p) {

if(root == null) {

return null;

}

TreeNode prev = null;

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

stack.push(root);

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

while(root != null) {

stack.push(root);

root = root.left;

}

root = stack.pop();

// Check 'root.val == p.val' before assign 'prev = root'

// because when given p's value equal to smallest node

// value in the BST, then no predecessor for p in this BST,

// and current 'prev' still as null we can return directly

if(root.val == p.val) {

return prev;

}

prev = root;

root = root.right;

}

return null;

}

}

Time Complexity: O(n)

Space Complexity: O(n)

**Q: Why we have to create a  'prev' variable to hold the predecesssor rather than no variable needed ?**

**A: Because if we during inorder traversal, the successor is naturally the very first node after the find target node, but in predecessor check the first node found smaller than target node is not necessary the very first node before the target node, for example, we have a inorder traversal for a BST has {1,2,3,4} four nodes and target node = 3, the first node after 3 is naturally 4 as successor, but the first node before 3 is 1 not the correct answer as 2 as predecessor, that's why we have to create a variable 'prev' to iteratively hold root's most recent previous node.**

**The wrong iterative traversal way copied directly from L285/Lint448.Inorder Successor in BST**

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;

}

}

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

Copied to below is WRONG by just modifying root.val < p.val

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 inorderPredecessor(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;

}

}

**Refer to**

<https://www.lintcode.com/problem/915/solution/19090>

两种方法：1.Recursive版本。2：non-recursive版本。

具体说一下非递归版本，实际上就是中序遍历树，找到p节点。从一开始设置一个pre节点，一直保持在root的前面。这样当root节点==p节点的时候，

前面的pre节点就刚好是我们要的predecessor.

非递归版本时间复杂度我个人认为和递归版本的时间复杂度一样都是O(n)。因为都是遍历直到p。 但是非递归版本的空间复杂度是O(h). 递归版本的虽然没有额外

辅助空间的开销，但是因为递归本身，会消耗栈空间，从空间复杂度上看，貌似非递归版本的要更优一些。

public TreeNode inorderPredecessor(TreeNode root, TreeNode p) {

// recursive way

if (root == null) {

return root;

}

if (p.val <= root.val) {

return inorderPredecessor(root.left, p);

}

TreeNode node = inorderPredecessor(root.right, p);

return node == null ? root : node;

}

// non-recursive

public TreeNode inorderPredecessor(TreeNode root, TreeNode p) {

if (root == null) {

return root;

}

TreeNode predecessor = null;

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

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

while (root != null) {

stack.push(root);

root = root.left;

}

root = stack.pop();

if (root == p) { return predecessor; }

predecessor = root;

root = root.right;

}

return null; // will never reach here

}