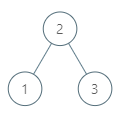
# Question

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

The successor of a node p is the node with the smallest key greater than p.val.

**Example 1:**

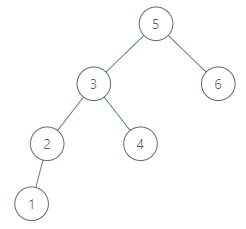


**Input:** root = [2,1,3], p = 1

**Output:** 2

**Explanation:** 1's in-order successor node is 2. Note that both p and the return value is of TreeNode type.

**Example 2:**



**Input:** root = [5,3,6,2,4,null,null,1], p = 6

**Output:** null

**Explanation:** There is no in-order successor of the current node, so the answer is null.

**Note:**

1. If the given node has no in-order successor in the tree, return null.
2. It's guaranteed that the values of the tree are unique.

# Solution

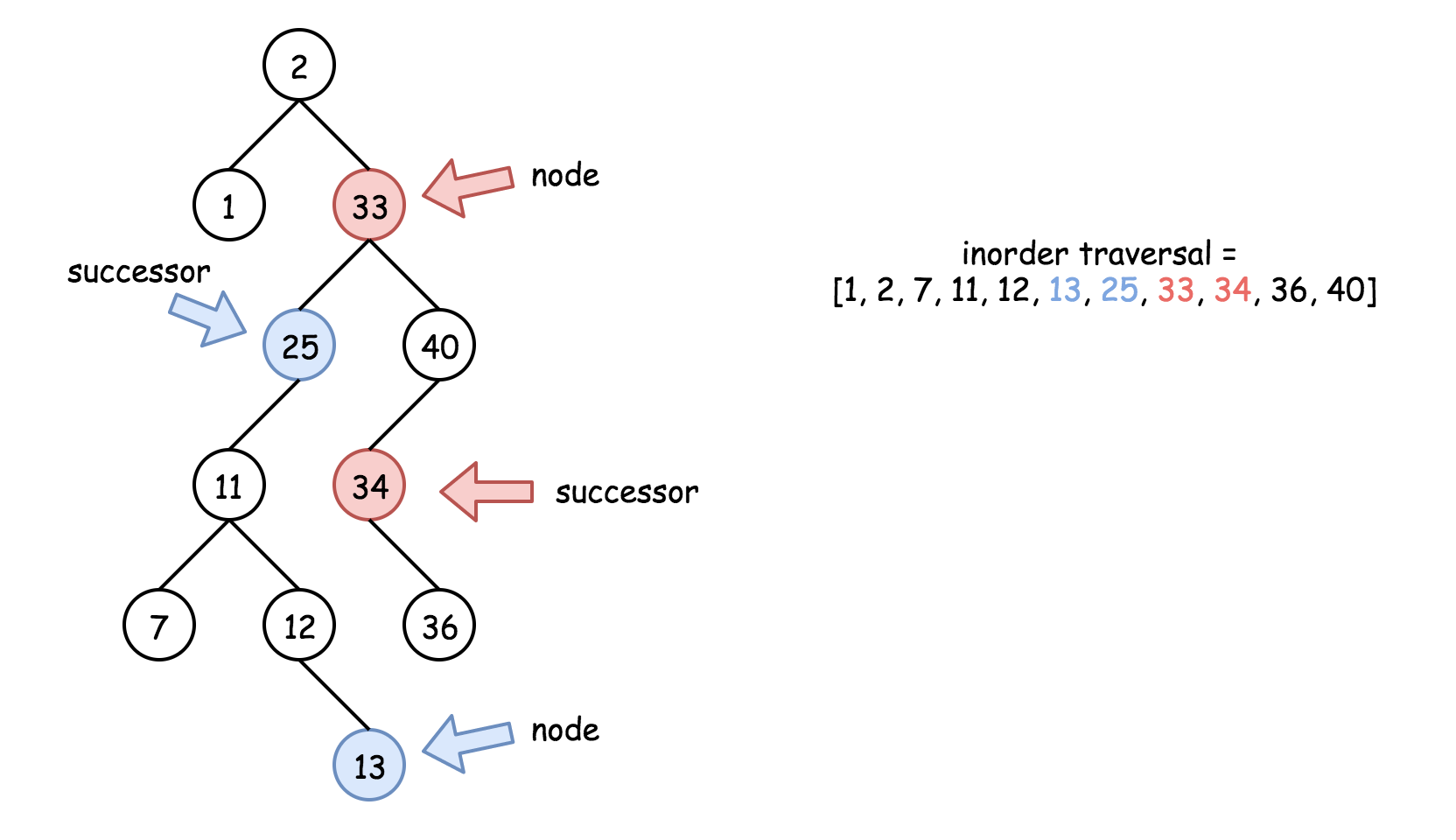
#### **Approach 1: Iterative Inorder Traversal**

[Inorder traversal of BST is an array sorted in the ascending order](https://leetcode.com/articles/delete-node-in-a-bst/).

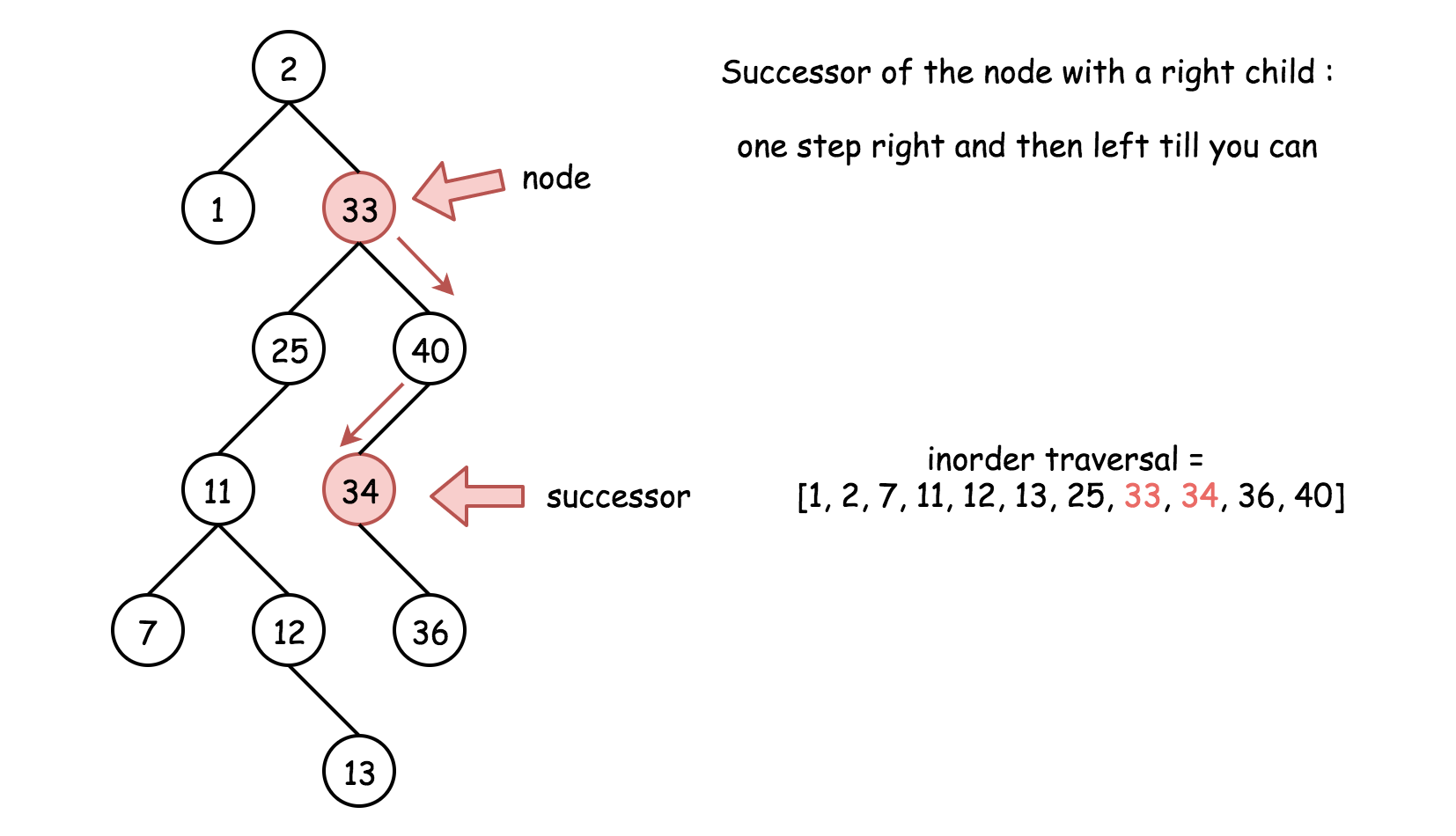
Successor is the smallest node in the inorder traversal after the current one.

There could be two situations :

1. If the node has a right child, the successor is somewhere lower in the tree, see red nodes on the Fig. below.
2. Otherwise, the successor is somewhere upper in the tree, see blue nodes on the Fig.



Let's first check the simple case 1. To find a successor, go one step right and then left till you can.



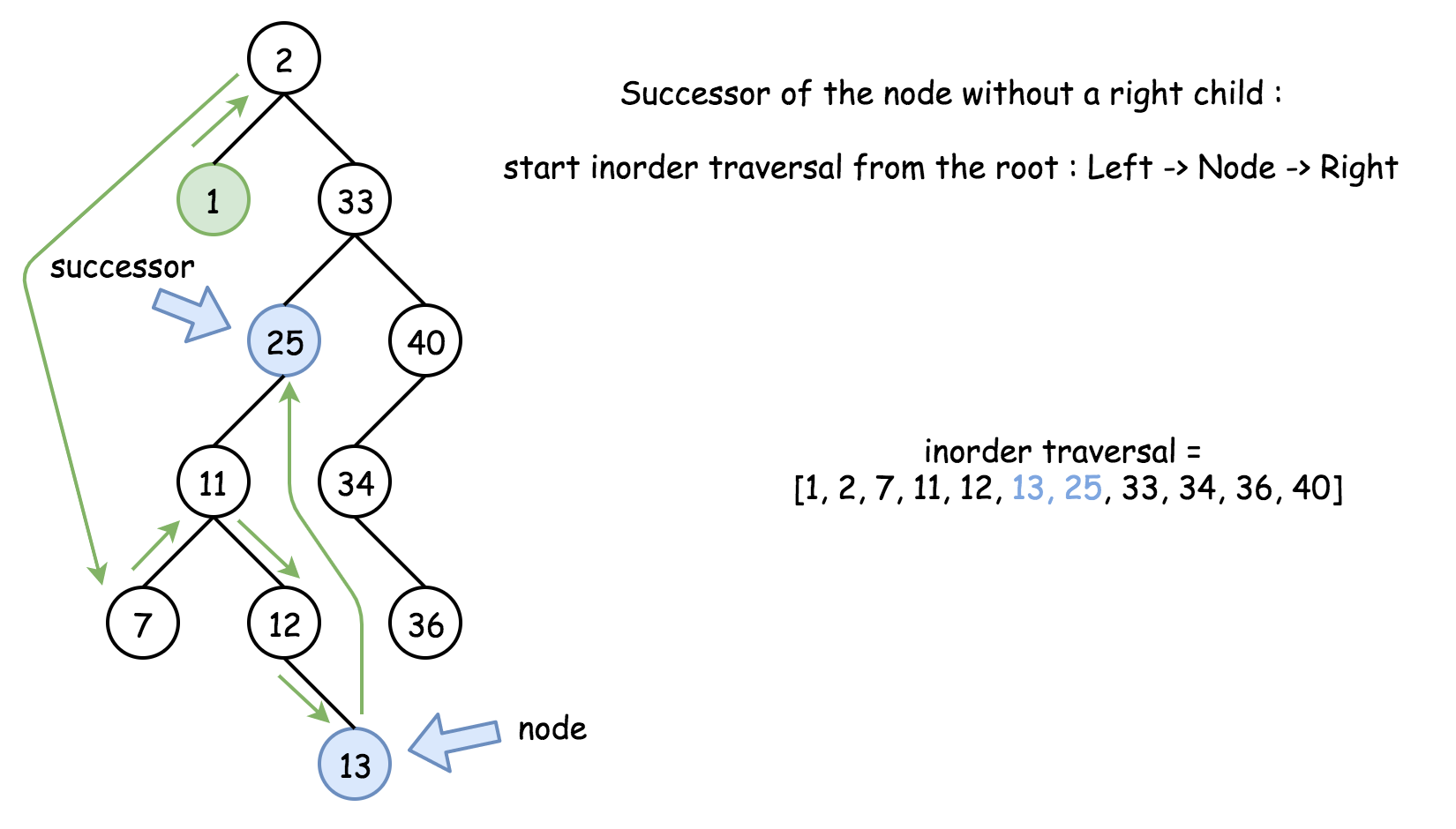
This approach has \mathcal{O}(H\_p)O(*Hp*​) time complexity, where H\_p*Hp*​ is a height of the node p*p*.

Now let's consider the case 2. There is no access to the parent nodes here, and hence one has to traverse the tree starting from the root and not from the node.

[Inorder traversal could be implemented in three different ways: recursive, iterative and Morris](https://leetcode.com/articles/recover-binary-search-tree/). Here it's better to choose the iterative traversal in order to optimise the performance and minimize the space at the same time.

Iterative inorder traversal is simple: go left till you can, and then one step right. Repeat till the end of nodes in the tree.

The idea is to keep just one previous node during the inorder traversal. If that previous node is equal to p, then the current node is a successor of p.



This approach has \mathcal{O}(H)O(*H*) time complexity, where H*H* is a tree height. Basically, this approach is universal and could be used for the case 1 as well. We simply don't do that because the approach 1 is faster.

**Algorithm**

* If the node has a right child, go one step right and then left till you can. Return the successor.
* Otherwise, implement iterative inorder traversal. While there are still nodes in the tree or in the stack:
  + Go left till you can, adding nodes in stack.
  + Pop out the last node. If its predecessor is equal to p, return that last node. Otherwise, save that node to be the predecessor in the next turn of the loop.
  + Go one step right.
* If we're here that means the successor doesn't exit. Return null.

**Implementation**

|  |
| --- |
| class Solution {  public TreeNode inorderSuccessor(TreeNode root, TreeNode p) {  // the successor is somewhere lower in the right subtree  // successor: one step right and then left till you can  if (p.right != null) {  p = p.right;  while (p.left != null) p = p.left;  return p;  }  // the successor is somewhere upper in the tree  ArrayDeque<TreeNode> stack = new ArrayDeque<>();  int inorder = Integer.MIN\_VALUE;  // inorder traversal : left -> node -> right  while (!stack.isEmpty() || root != null) {  // 1. go left till you can  while (root != null) {  stack.push(root);  root = root.left;  }  // 2. all logic around the node  root = stack.pop();  // if the previous node was equal to p  // then the current node is its successor  if (inorder == p.val) return root;  inorder = root.val;  // 3. go one step right  root = root.right;  }  // there is no successor  return null;  }  } |

**Complexity Analysis**

* Time complexity : \mathcal{O}(H\_p)O(*Hp*​) in the best case, when node p has a right child. Here H\_p*Hp*​ is a height of node p. \mathcal{O}(H)O(*H*) in the worst case of no right child. Here H*H* is a tree height.
* Space complexity : \mathcal{O}(1)O(1) in the best case, when node p has a right child. Otherwise, up to \mathcal{O}(H)O(*H*) to keep the stack.

Other:

**Successor**

public TreeNode successor(TreeNode root, TreeNode p) {

if (root == null)

return null;

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

return successor(root.right, p);

} else {

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

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

}

}

**Predecessor**

public TreeNode predecessor(TreeNode root, TreeNode p) {

if (root == null)

return null;

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

return predecessor(root.left, p);

} else {

TreeNode right = predecessor(root.right, p);

return (right != null) ? right : root;

}

}