Previous (/articles/insert-into-a-bst/) Next (/articles/kth-smallest-element-in-a-bst/)

450. Delete node in a BST (/problems/delete-node-in-a-bst/)

April 26, 2019 | 25.2K views

Given a root node reference of a BST and a key, delete the node with the given key in the BST. Return the root node reference (possibly updated) of the BST.

Basically, the deletion can be divided into two stages:

1. Search for a node to remove.
2. If the node is found, delete the node.

Note: Time complexity should be O(height of tree).

Example:

```
root = [5,3,6,2,4,null,7]
key = 3

    5
   / \
  3   6
 / \   \
2   4   7

Given key to delete is 3. So we find the node with value 3 and delete it.

One valid answer is [5,4,6,2,null,null,7], shown in the following BST.

    5
   / \
  4   6
 /     \
2       7

Another valid answer is [5,2,6,null,4,null,7].

    5
   / \
  2   6
   \   \
    4   7
```

Solution

Three facts to know about BST

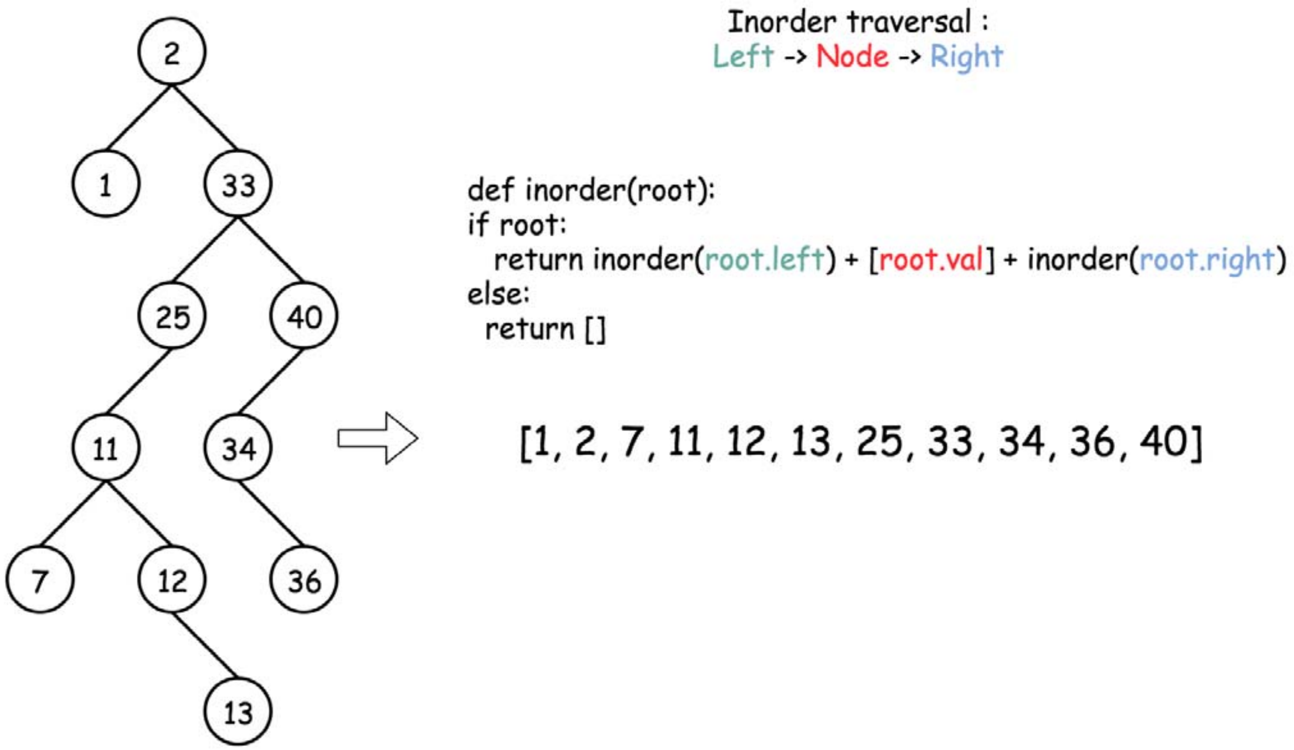
Here is list of facts which are better to know before the interview.

> Inorder traversal of BST is an array sorted in the ascending order.

To compute inorder traversal follow the direction `Left -> Node -> Right`.

Java Python Copy

```
def inorder(root):
    return inorder(root.left) + [root.val] + inorder(root.right) if root else []
```



> Successor = "after node", i.e. the next node, or the smallest node *after* the current one.

It's also the *next* node in the inorder traversal. To find a successor, go to the right once and then as many times to the left as you could.

Java Python Copy

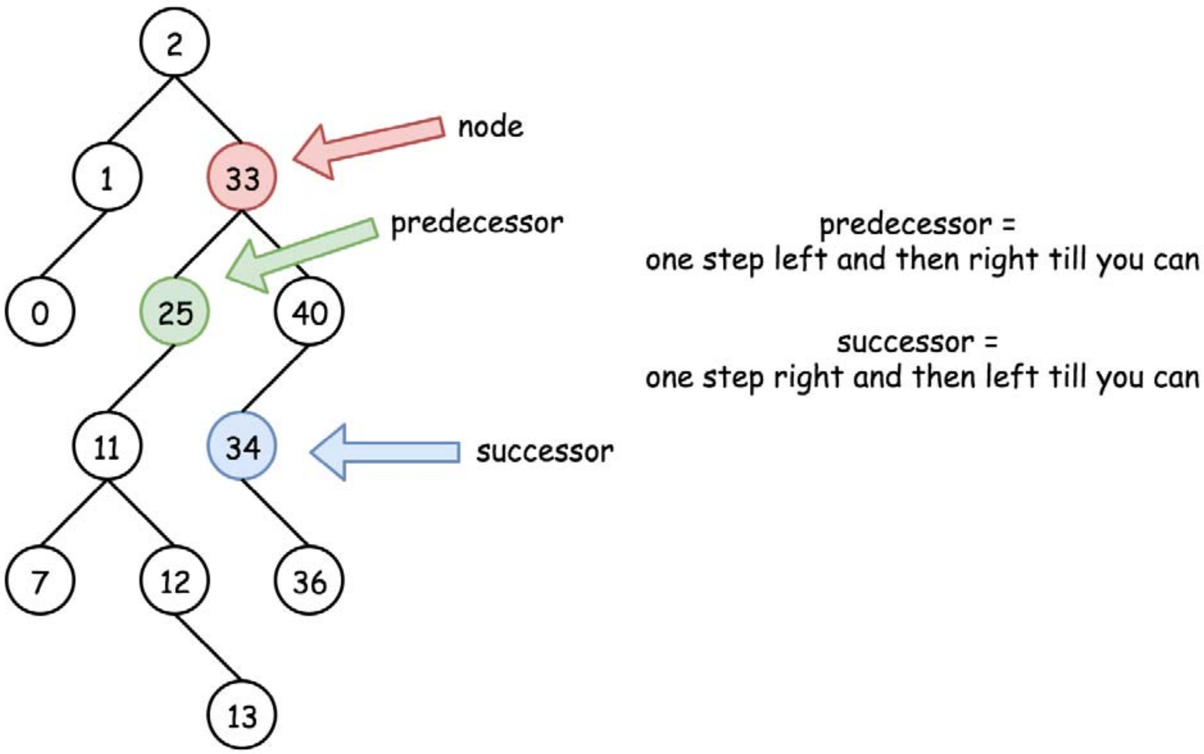
```
def successor(root):
    root = root.right
    while root.left:
        root = root.left
    return root
```

> Predecessor = "before node", i.e. the previous node, or the largest node *before* the current one.

It's also the *previous* node in the inorder traversal. To find a predecessor, go to the left once and then as many times to the right as you could.

Java Python Copy

```
def predecessor(root):
    root = root.left
    while root.right:
        root = root.right
    return root
```

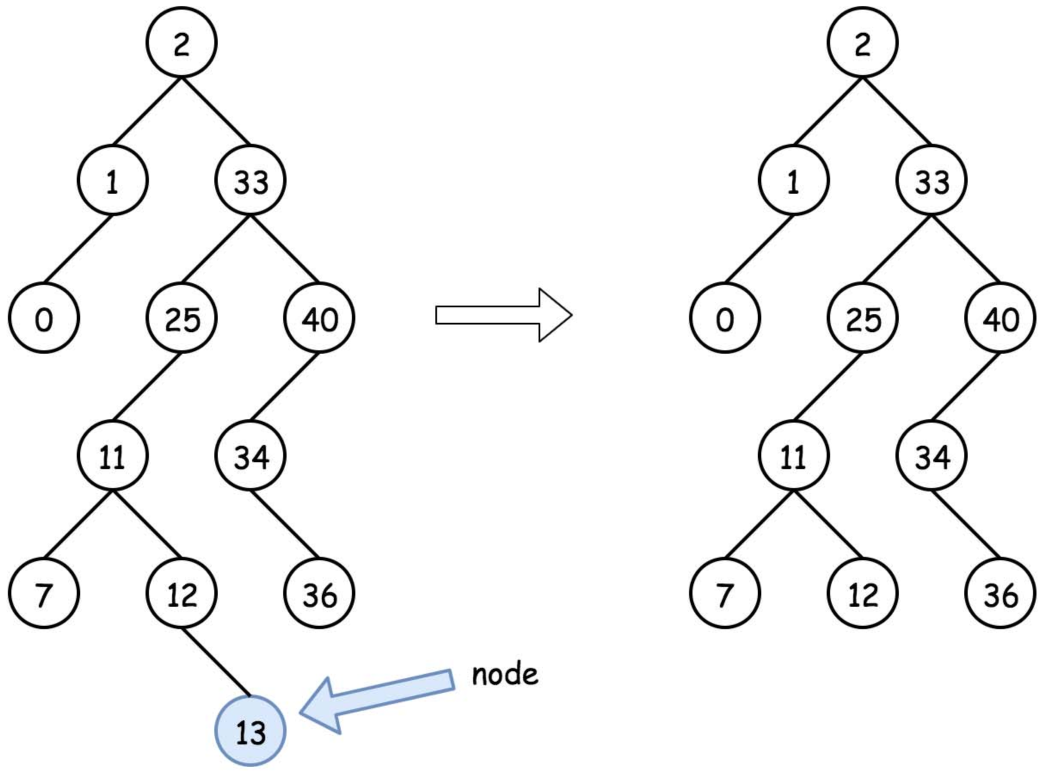


Approach 1: Recursion

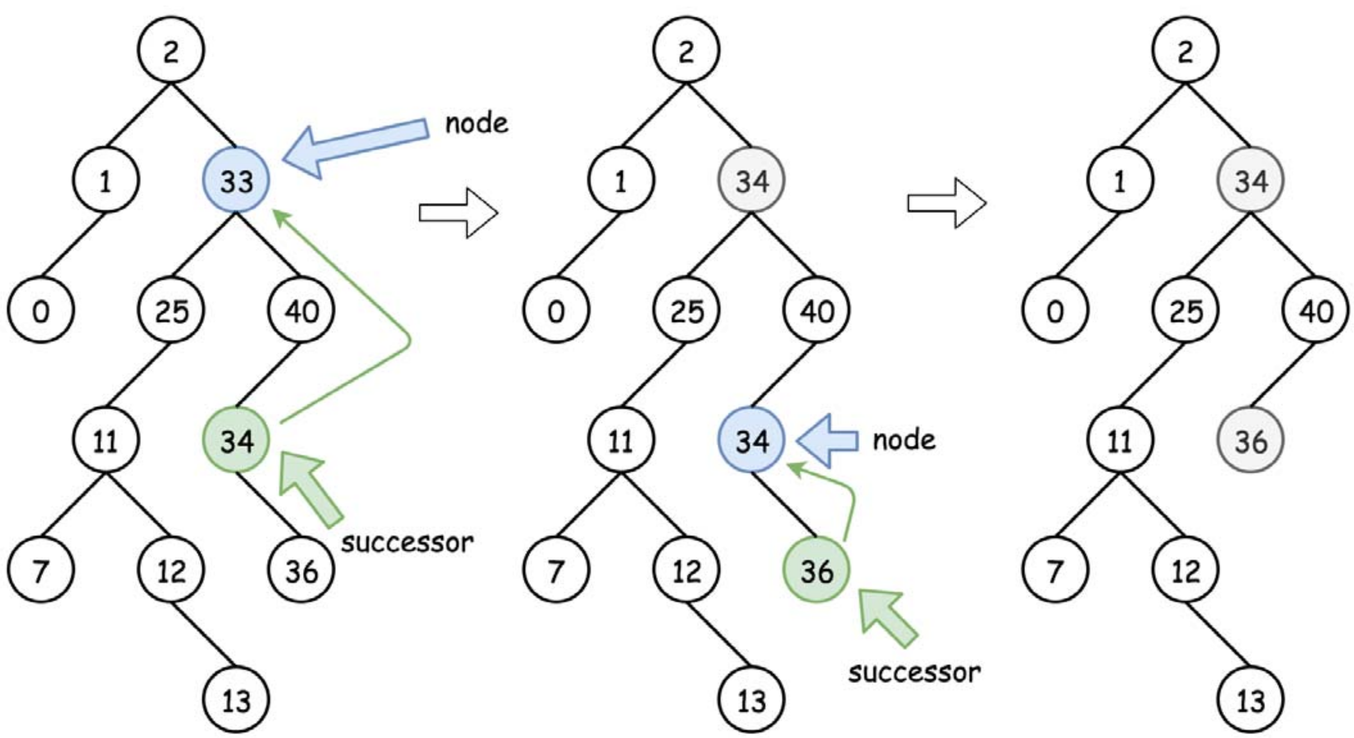
Intuition

There are three possible situations here :

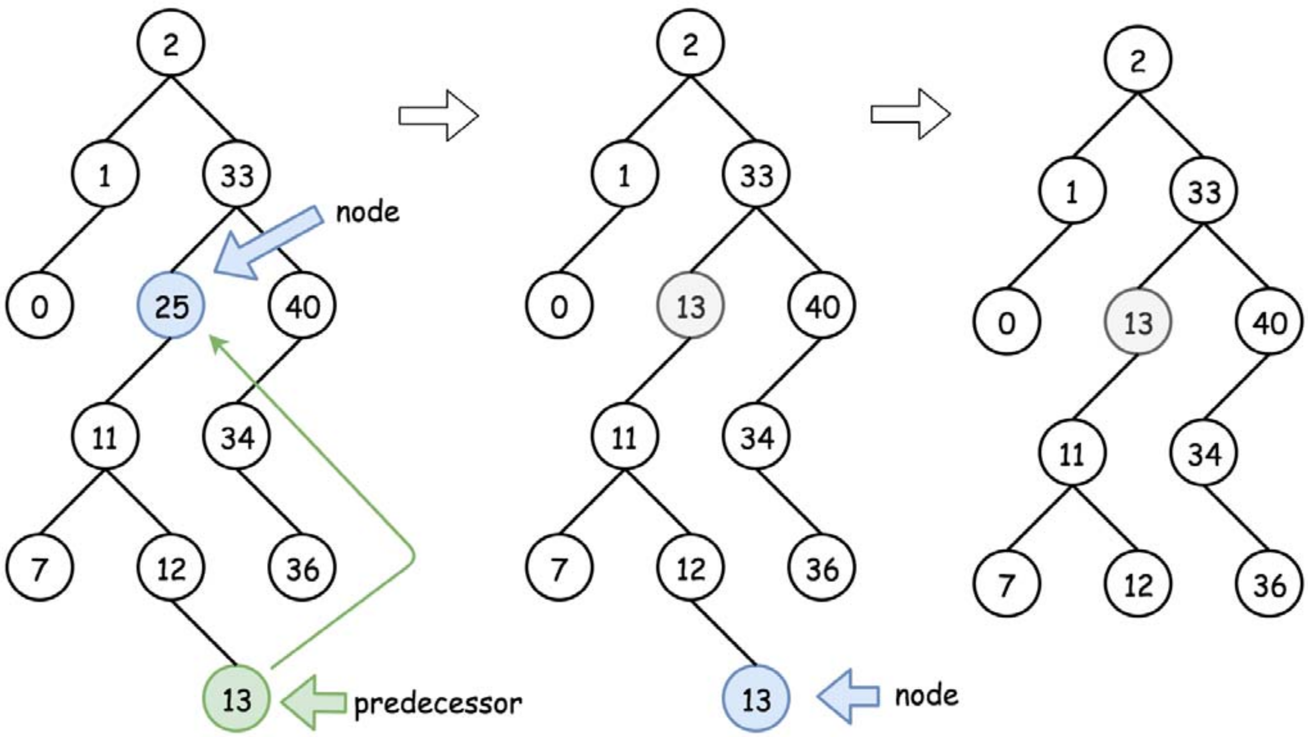
- Node is a leaf, and one could delete it straightforward : `node = null`.



- Node is not a leaf and has a right child. Then the node could be replaced by its *successor* which is somewhere lower in the right subtree. Then one could proceed down recursively to delete the successor.



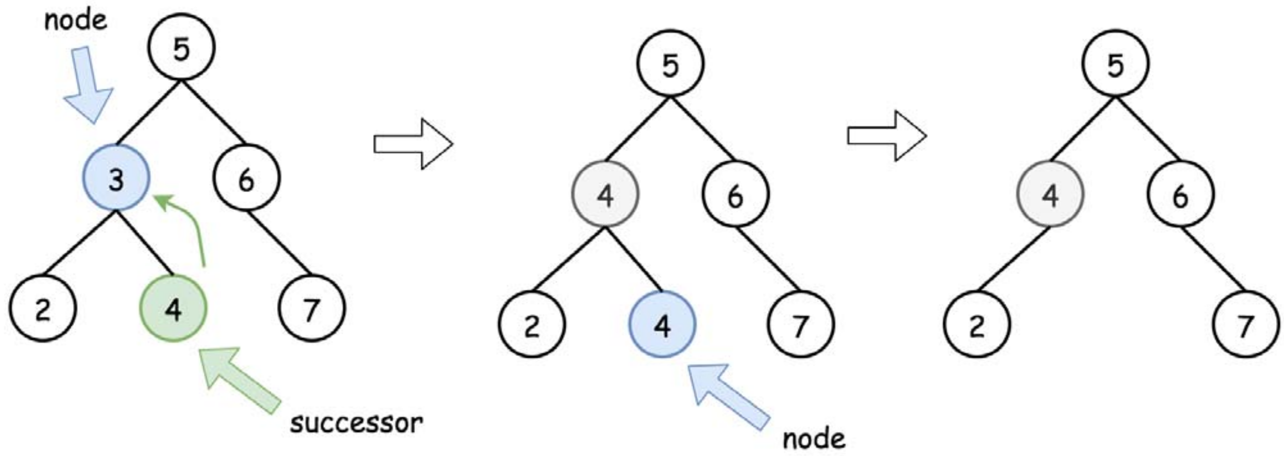
- Node is not a leaf, has no right child and has a left child. That means that its *successor* is somewhere upper in the tree but we don't want to go back. Let's use the *predecessor* here which is somewhere lower in the left subtree. The node could be replaced by its *predecessor* and then one could proceed down recursively to delete the predecessor.



Algorithm

- If `key > root.val` then delete the node to delete is in the right subtree `root.right = deleteNode(root.right, key)`.
- If `key < root.val` then delete the node to delete is in the left subtree `root.left = deleteNode(root.left, key)`.
- If `key == root.val` then the node to delete is right here. Let's do it :
  - If the node is a leaf, the delete process is straightforward : `root = null`.
  - If the node is not a leaf and has the right child, then replace the node value by a successor value `root.val = successor.val`, and then recursively delete the successor in the right subtree `root.right = deleteNode(root.right, root.val)`.
  - If the node is not a leaf and has only the left child, then replace the node value by a predecessor value `root.val = predecessor.val`, and then recursively delete the predecessor in the left subtree `root.left = deleteNode(root.left, root.val)`.
- Return `root`.

Implementation



Java Python Copy

```
class Solution:
    def successor(self, root):
        """
        One step right and then always left
        """
        root = root.right
        while root.left:
            root = root.left
        return root.val

    def predecessor(self, root):
        """
        One step left and then always right
        """
        root = root.left
        while root.right:
            root = root.right
        return root.val

    def deleteNode(self, root: TreeNode, key: int) -> TreeNode:
        if not root:
            return None

        # delete from the right subtree
        if key > root.val:
            root.right = self.deleteNode(root.right, key)
        # delete from the left subtree
```

Complexity Analysis

- Time complexity : $\mathcal{O}(\log N)$. During the algorithm execution we go down the tree all the time - on the left or on the right, first to search the node to delete ($\mathcal{O}(H_1)$ time complexity as already discussed (https://leetcode.com/articles/insert-into-a-bst/)) and then to actually delete it. H_1 is a tree height from the root to the node to delete. Delete process takes $\mathcal{O}(H_2)$ time, where H_2 is a tree height from the root to delete to the leafs. That in total results in $\mathcal{O}(H_1 + H_2) = \mathcal{O}(H)$ time complexity, where H is a tree height, equal to $\log N$ in the case of the balanced tree.

- Space complexity : $\mathcal{O}(H)$ to keep the recursion stack, where H is a tree height. $H = \log N$ for the balanced tree.

Rate this article:

Previous (/articles/insert-into-a-bst/)

Next (/articles/kth-smallest-element-in-a-bst/)

Help Center (/support/) | Terms (/terms/) | Privacy (/privacy/) United States (/region/)