Algorithms and Data Structures for Beginners

19 / 35

18 - BST Insert and Remove

Mark Lesson Complete View Code Prev Next

18 Remove

Suggested Problems

Status	Star	Problem	Difficulty	Video Solution	Code
☐	☆	Insert Into A Binary Search Tree	Medium		
☐	☆	Delete Node In A Bst	Medium		

Insertion and Removal from a Binary Search Tree

We mentioned previously that the main benefit of using BSTs over sorted arrays is that we can perform removal and insertion in $O(log\ n)$ time, assuming that the tree is balanced. Let's dig a little deeper into the insertion and deletion.

Insertion

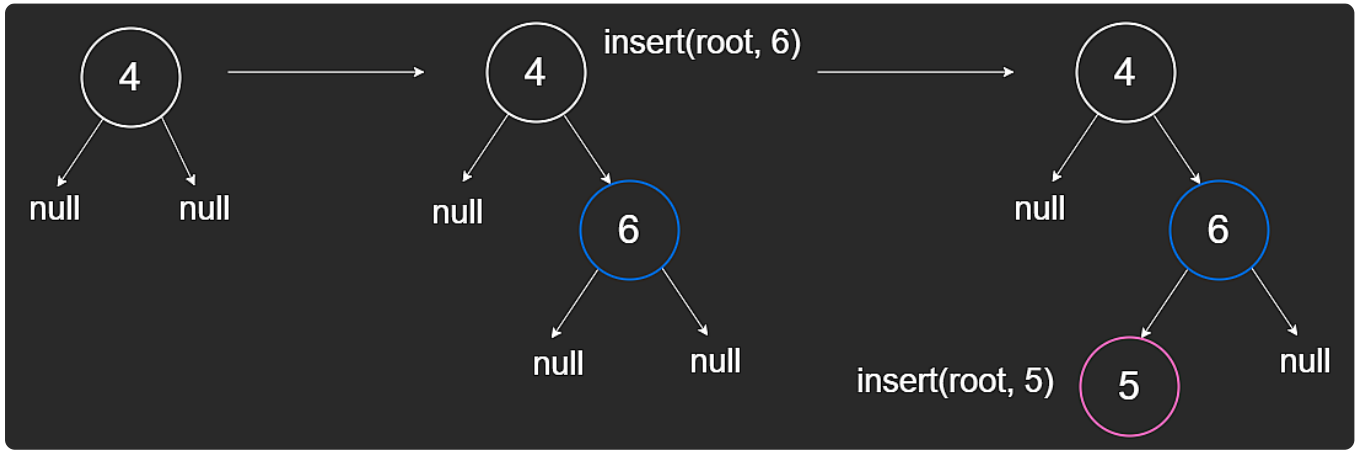
If we wish to insert a new node into the BST, we first have to traverse the BST to find the right position, and then insert this node.

If we have a BST `[4]`, and wish to insert 6, we could either end up with `[4,null,6]` or `[6,4,null]`. Both of these would be valid BSTs. In the first example, we added the 6 as a leaf node, which is an easier process than the second example.

Let's add `5` to previously resulting tree `[4,null,6]`, which results in `[4,null,6,5,null]`.

This process is demonstrated by the pseudocode below.

```
fn insert(root, val):
    // Base case, if we have found the position of insertion
    if root is null:
        return new TreeNode(val)
    // If value is greater than the current node, it belongs in the rig
    if val > root.val:
        root.right = insert(root.right, val)
    // If value is smaller than the current node, it belongs in the lef
    else if val < root.val:
        root.left = insert(root.left, val)
    return root
```



The visual above demonstrates how insertion is done. 6 is greater than the root node, so it ends up in the right sub-tree. 5 is greater than the root node but smaller than 6 so it ends up in the left subtree of the tree rooted at 6.

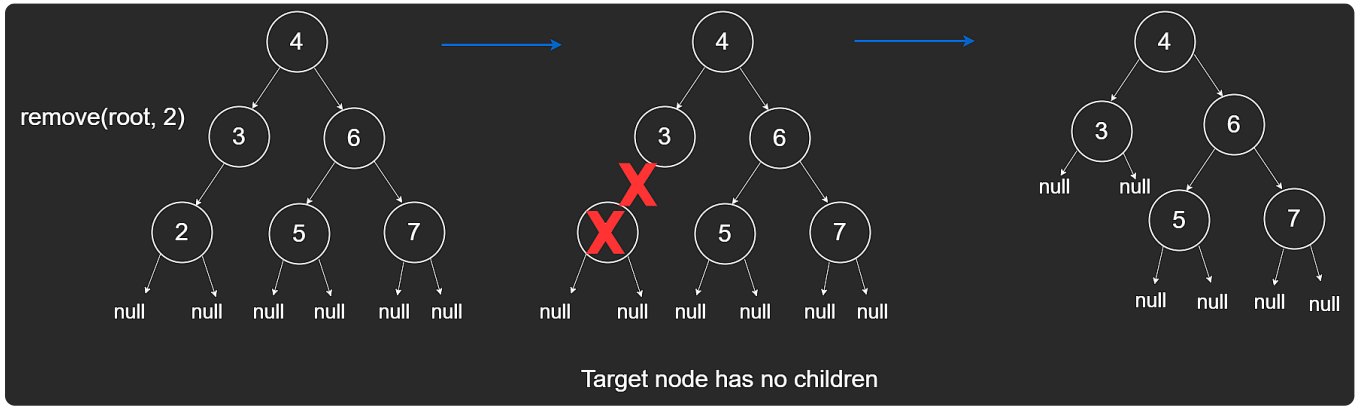
Removal

Before removing a node from a BST, we need to consider two cases:

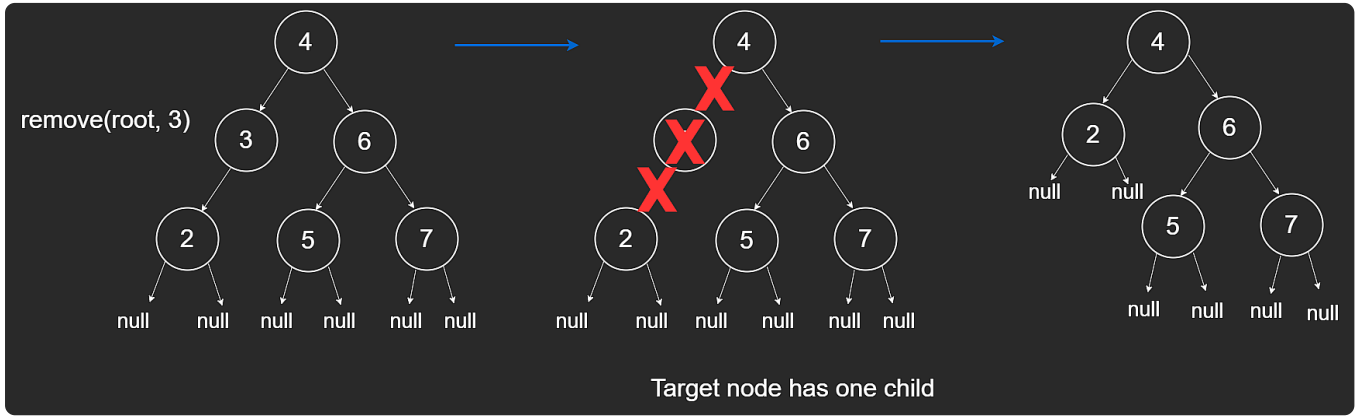
1. The target node has 0 or 1 child
2. The target node has 2 children

Case 1 - The target node has one child or no children

If we wish to delete node `2`, which has no children, the `left_child` pointer of `3` now points to `null`.



If we wish to delete node `3`, which has one child, the `left_child` pointer of the root node will point to `2` instead of `3`.



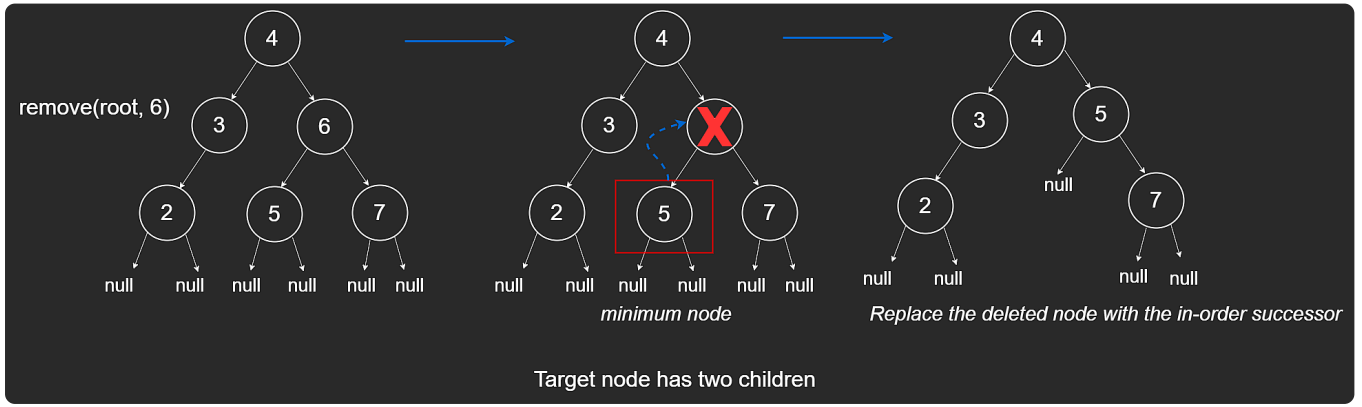
Case 2 - The target node has two children

If we wanted to delete a node with two children, say, `6`, we replace the node with its **in-order successor.**

The in-order successor is the left-most node in the right subtree of the target node. Another way of looking at it is that it is the smallest node among all the nodes that

are greater than the target node. This will ensure that the resulting tree is still a valid binary search tree.

The visual below shows process of deletion of nodes with two children.




```
// Find the minimum node, which is the in-order successor
fn minValueNode(root):
    curr = root
    while curr is not null and curr.left is not null:
        curr = curr.left
    return curr

// Remove a node and return the root of the BST.
fn remove(root, val):
    if root is null:
        return null
```

```
// If the value is bigger than the current node, we go right
if val > root.val:
    root.right = remove(root.right, val)
// Otherwise, the value is smaller than the current node, so we go
else if val < root.val:
    root.left = remove(root.left, val)
// We have found the target node
else:
    // Target node has one child
    if root.left is null:
        return root.right
    else if root.right is null:
        return root.left
    // Target node has two children
    else:
        minNode = minValueNode(root.right)
        root.val = minNode.val
        root.right = remove(root.right, minNode.val)
return root
```

Time Complexity

If the given tree is a balanced binary tree, the height will be in $log\ n$, for reasons that are very similar to what we discussed in merge sort. However, it is a possibility that in the worst case, the tree provided is either left-skewed or right-skewed. In that

case, the height of the tree will be in $O(n)$ and the total work done for all the operations described above is $O(n)$.

Github Privacy Terms