# 701. Insert into a BST (/problems/insert-into-a-binary-search-tree/)

April 24, 2019 | 6.2K views

Average Rating: 4.73 (11 votes)

Given the root node of a binary search tree (BST) and a value to be inserted into the tree, insert the value into the BST. Return the root node of the BST after the insertion. It is guaranteed that the new value does not exist in the original BST.

Note that there may exist multiple valid ways for the insertion, as long as the tree remains a BST after insertion. You can return any of them.

For example,

```
Given the tree:
        4
       / \
      2   7
     / \
    1   3
And the value to insert: 5
```

You can return this binary search tree:

```
         4
       /   \
      2     7
     / \   /
    1   3 5
```

This tree is also valid:

```
      5
    /   \
   2     7
  / \
 1   3
      \
       4
```

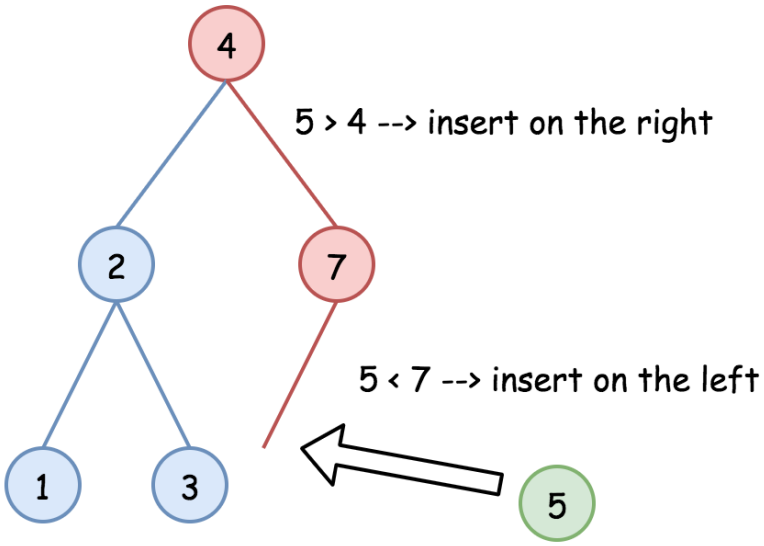
# Solution

## Intuition

One of the huge BST advantages is a search (https://leetcode.com/problems/search-in-a-binary-search-tree/) for *arbitrary* element in $\mathcal{O}(\log N)$ time. Here we'll see that the insert time is $\mathcal{O}(\log N)$ , too, in the average case.

The problem solution is very simple - one could always insert new node as a child of the leaf. To define which leaf to use, one could follow the standard BST logic :

- If `val > node.val` - go to insert into the right subtree.
- If `val < node.val` - go to insert into the left subtree.

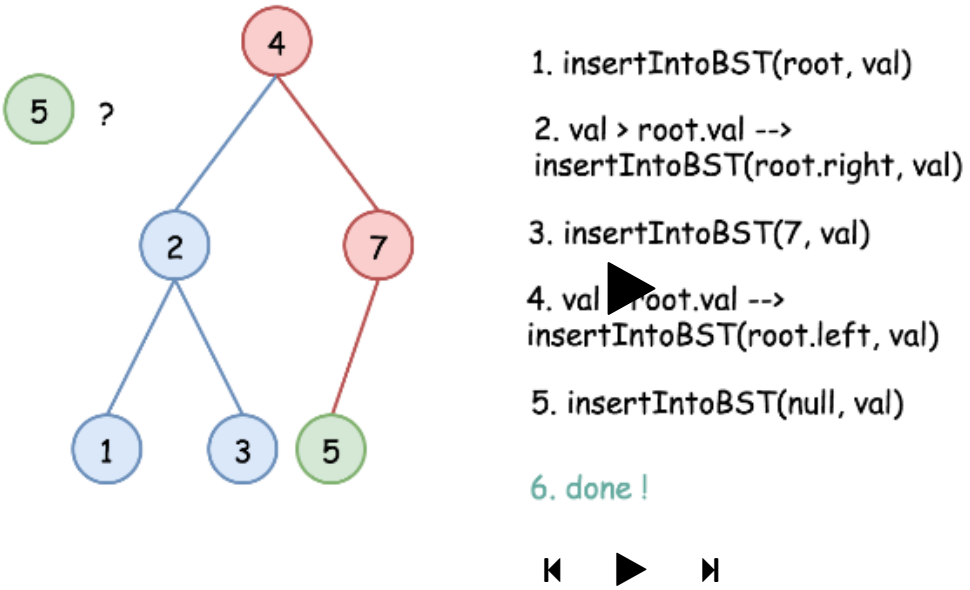


## Approach 1: Recursion



The recursion implementation is very straightforward :

- If `root` is null - return `TreeNode(val)` .
- If `val > root.val` - go to insert into the right subtree.
- If `val < root.val` - go to insert into the left subtree.
- Return `root` .




7 / 7

Java Python Copy

```
class Solution:
    def insertIntoBST(self, root: TreeNode, val: int) -> TreeNode:
        if not root:
            return TreeNode(val)

        if val > root.val:
            # insert into the right subtree
            root.right = self.insertIntoBST(root.right, val)
        else:
            # insert into the left subtree
            root.left = self.insertIntoBST(root.left, val)
        return root
```

**Complexity Analysis**

- Time complexity : $\mathcal{O}(H)$, where $H$ is a tree height. That results in $\mathcal{O}(\log N)$ in the average case, and $\mathcal{O}(N)$ in the worst case.

  Let's compute time complexity with the help of master theorem (https://en.wikipedia.org/wiki/Master_theorem_(analysis_of_algorithms)) $T(N) = aT\left(\frac{N}{b}\right) + \Theta(N^d)$. The equation

represents dividing the problem up into $a$ subproblems of size $\frac{N}{b}$ in $\Theta(N^d)$ time. Here at step there is only one subproblem `a = 1`, its size is a half of the initial problem `b = 2`, and all this happens in a constant time `d = 0`, as for the binary search. That means that $\log_b a = d$ and hence we're dealing with case 2 (https://en.wikipedia.org/wiki/Master_theorem_(analysis_of_algorithms)#Case_2_example) that results in $\mathcal{O}(n^{\log_b a} \log^{d+1} N) = \mathcal{O}(\log N)$ time complexity.

- Space complexity : $\mathcal{O}(H)$ to keep the recursion stack, i.e. $\mathcal{O}(\log N)$ in the average case, and $\mathcal{O}(N)$ in the worst case.

## Approach 2: Iteration

The recursion above could be converted into the iteration

Java Python Copy

```
class Solution:
    def insertIntoBST(self, root: TreeNode, val: int) -> TreeNode:
        node = root
        while node:
            # insert into the right subtree
            if val > node.val:
                # insert right now
                if not node.right:
                    node.right = TreeNode(val)
                    return root
                else:
                    node = node.right
            # insert into the left subtree
            else:
                # insert right now
                if not node.left:
                    node.left = TreeNode(val)
                    return root
                else:
                    node = node.left
        return TreeNode(val)
```

**Complexity Analysis**

- Time complexity : $\mathcal{O}(H)$, where $H$ is a tree height. That results in $\mathcal{O}(\log N)$ in the average case, and $\mathcal{O}(N)$ in the worst case.

  Let's compute time complexity with the help of master theorem (https://en.wikipedia.org

/wiki/Master_theorem_(analysis_of_algorithms)) $T(N) = aT\left(\frac{N}{b}\right) + \Theta(N^d)$. The equation represents dividing the problem up into $a$ subproblems of size $\frac{N}{b}$ in $\Theta(N^d)$ time. Here at step there is only one subproblem `a = 1`, its size is a half of the initial problem `b = 2`, and all this happens in a constant time `d = 0`, as for the binary search. That means that $\log_b a = d$ and hence we're dealing with case 2 (https://en.wikipedia.org /wiki/Master_theorem_(analysis_of_algorithms)#Case_2_example) that results in $\mathcal{O}(n^{\log_b a} \log^{d+1} N) = \mathcal{O}(\log N)$ time complexity.

- Space complexity : $\mathcal{O}(1)$ since it's a constant space solution.

Analysis written by @liaison (https://leetcode.com/liaison/) and @andvary (https://leetcode.com /andvary/)

### Rate this article:

Previous (/articles/binary-search/) Next (/articles/delete-node-in-a-bst/)

## Comments: 2

Sort By

Type comment here... (Markdown is supported)

Preview Post

alexspark (alexspark) ★ 2 April 27, 2019 8:00 PM

(/alexspark)

Thank you for this solution. I was able to come up with similar recursive and iterative solutions. Can we do variants of this question such as if we wanted to optimize for a height balanced BST because with the above solution, if we always added new values that are greater than all values in the BST, we'd get a very skewed BST.

Read More

2 Share Reply

SHOW 1 REPLY

foobarfoo (foobarfoo) ★ 11 April 25, 2019 5:32 AM

(/foobarfoo)

Can you explain why both the `return new TreeNode(val);` and `return root` are valid solution ?

1 Share Reply

SHOW 5 REPLIES