700. Search in a BST (/problems/search-in-a-binary-search-tree/)

Sept. 13, 2019 | 25.2K views

Average Rating: 4.92 (13 votes)

Given the root node of a binary search tree (BST) and a value. You need to find the node in the BST that the node's value equals the given value. Return the subtree rooted with that node. If such node doesn't exist, you should return NULL.

For example,

```
Given the tree:
        4
       / \
      2   7
     / \
    1   3

And the value to search: 2
```

You should return this subtree:

```
      2
     / \
    1   3
```

In the example above, if we want to search the value `5`, since there is no node with value `5`, we should return `NULL`.

Note that an empty tree is represented by `NULL`, therefore you would see the expected output (serialized tree format) as `[]`, not `null`.

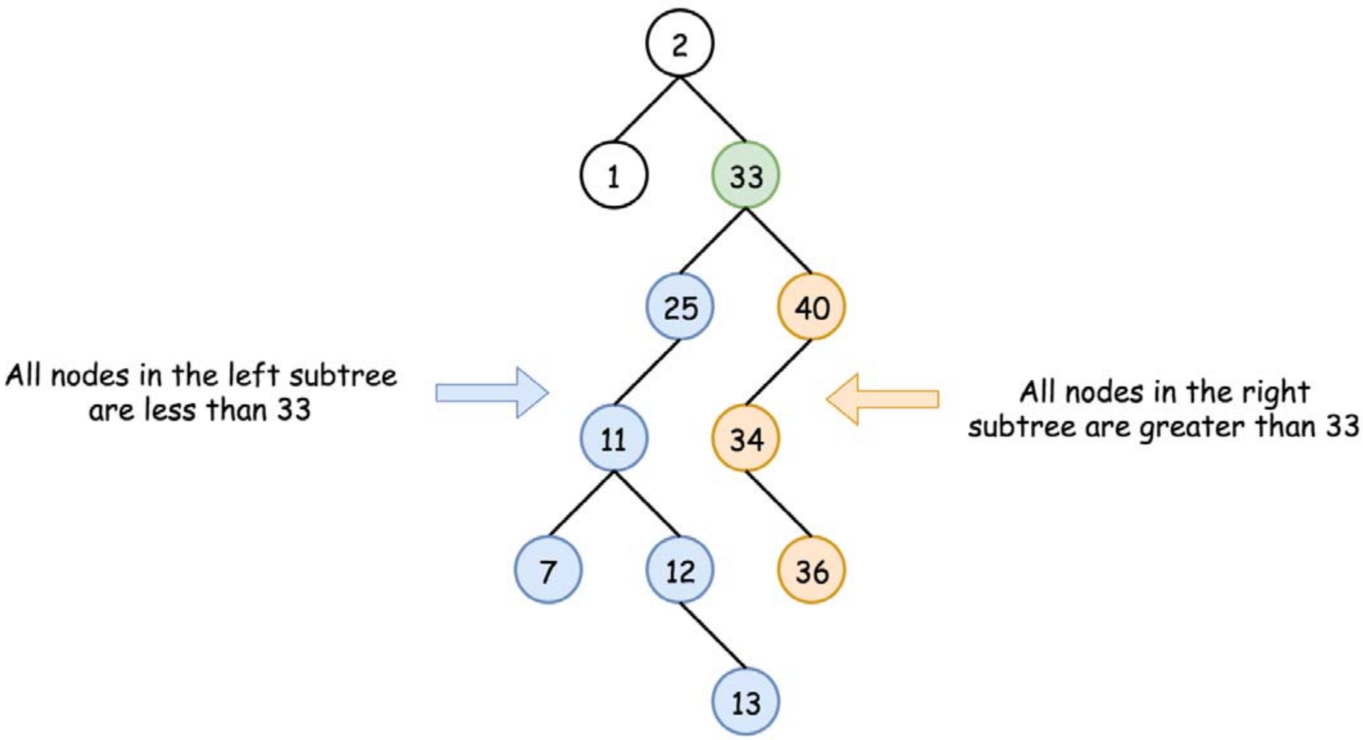
Solution

Binary Search Tree.

Binary Search Tree is a binary tree where the key in each node

- is greater than any key stored in the left sub-tree,
- and less than any key stored in the right sub-tree.

Here is an example:



Such data structure provides the following operations in a logarithmic time:

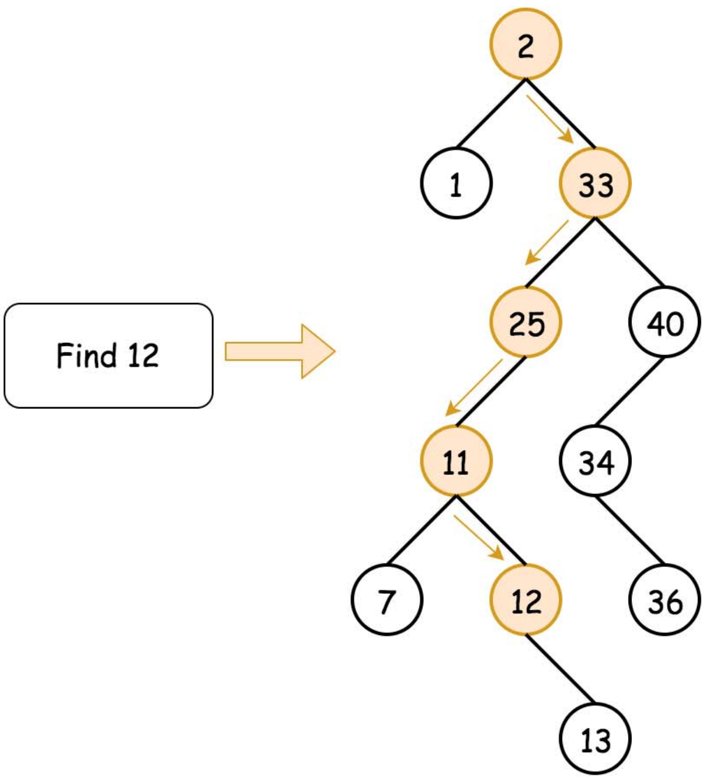
- Search.
- Insert (https://leetcode.com/articles/insert-into-a-bst/).
- Delete (https://leetcode.com/articles/delete-node-in-a-bst/).

Approach 1: Recursion

Algorithm

The recursion implementation is very straightforward:

- If the tree is empty `root == null` or the value to find is here `val == root.val` - return root.
- If `val < root.val` - go to search into the left subtree `searchBST(root.left, val)`.
- If `val > root.val` - go to search into the right subtree `searchBST(root.right, val)`.
- Return `root`.



Implementation

Java Python Copy

```
class Solution {
  public TreeNode searchBST(TreeNode root, int val) {
    if (root == null || val == root.val) return root;

    return val < root.val ? searchBST(root.left, val) : searchBST(root.right, 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.

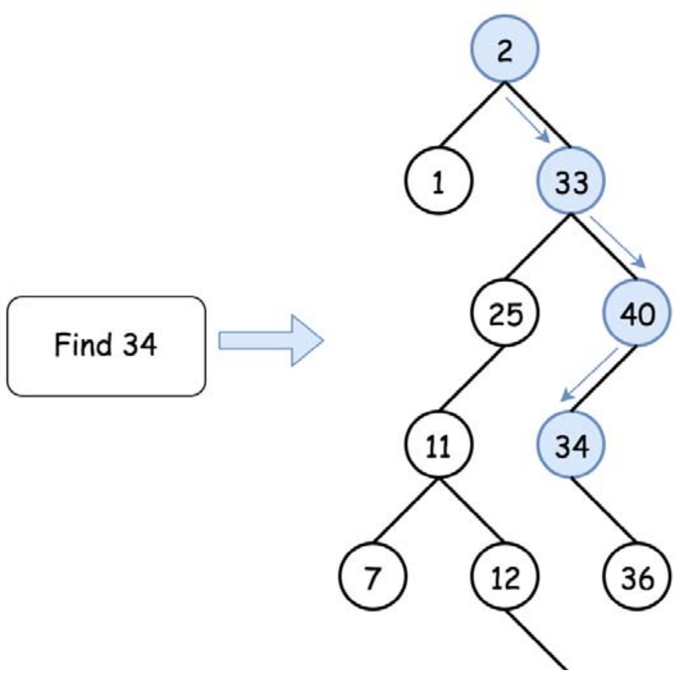
Let's compute time complexity with the help of master theorem (https://en.wikipedia.org/wiki/Master_theorem_(analysis_of_algorithms)) $T(N) = aT\left(\frac{N}{b}\right) + \Theta(N^d)$. The equation represents dividing the problem up into a subproblems of size $\frac{N}{b}$ in $\Theta(N^d)$ time. Here 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 : $\mathcal{O}(H)$ 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

To reduce the space complexity, one could convert recursive approach into the iterative one:

- While the tree is not empty `root != null` and the value to find is *not* here `val != root.val`:
  - If `val < root.val` - go to search into the left subtree `root = root.left`.
  - If `val > root.val` - go to search into the right subtree `root = root.right`.
- Return `root`.



13

Implementation

Java Python Copy

```
class Solution {
  public TreeNode searchBST(TreeNode root, int val) {
    while (root != null && val != root.val)
      root = val < root.val ? root.left : root.right;
    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 : $\mathcal{O}(1)$ since it's a constant space solution.

Rate this article:

Previous (/articles/repeated-dna-sequences/)

Next (/articles/search-in-a-sorted-array-of-unknown-size/)

Comments: 5

Sort By

Type comment here... (Markdown is supported)

Preview

Post

ajaay (/ajaay) 19 September 13, 2019 4:58 PM

If the compiler supports tail-recursion optimization, I think the recursive solution would also use O(1) memory. GCC supports this if the -O2 or -O3 flags are used.

(/ajaay)

18 Share Reply

SHOW 3 REPLIES

qiuqiushasha (/qiuqiushasha) 41 September 14, 2019 11:24 AM

@tailrec in Scala is my favoriate

(/qiuqiushasha)

2 Share Reply

PokemonChampKRW (/pokemonchampkrw) 4 June 17, 2020 11:10 PM

The June daily challenge is very inconsistent with problem difficulty...

(/pokemonchampkrw)

0 Share Reply

anonymous1127 (/anonymous1127) 89 June 15, 2020 9:18 AM

How to do this in Haskell

(/anonymous1127)

0 Share Reply

sriharik (/sriharik) 73 May 12, 2020 10:11 PM

Straightforward recursion and iterative version.

(/sriharik)

```
    public TreeNode searchBST(TreeNode root, int val) {
        if (root == null) return null;
        if (root.val == val) return root;
```

Read More

0 Share Reply

SHOW 1 REPLY

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