Previous Next

173. Binary Search Tree Iterator

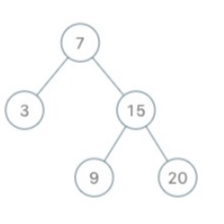
June 6, 2019 | 84.3K views

Average Rating: 4.94 (174 votes)

Implement an iterator over a binary search tree (BST). Your iterator will be initialized with the root node of a BST.

Calling `next()` will return the next smallest number in the BST.

Example:




```
BSTIterator iterator = new BSTIterator(root);
iterator.next();    // return 3
iterator.next();    // return 7
iterator.hasNext(); // return true
iterator.next();    // return 9
iterator.hasNext(); // return true
iterator.next();    // return 15
iterator.hasNext(); // return true
iterator.next();    // return 20
iterator.hasNext(); // return false
```

Note:

- `next()` and `hasNext()` should run in average O(1) time and uses O(*h*) memory, where *h* is the height of the tree.
- You may assume that `next()` call will always be valid, that is, there will be at least a next smallest number in the BST when `next()` is called.

Solution

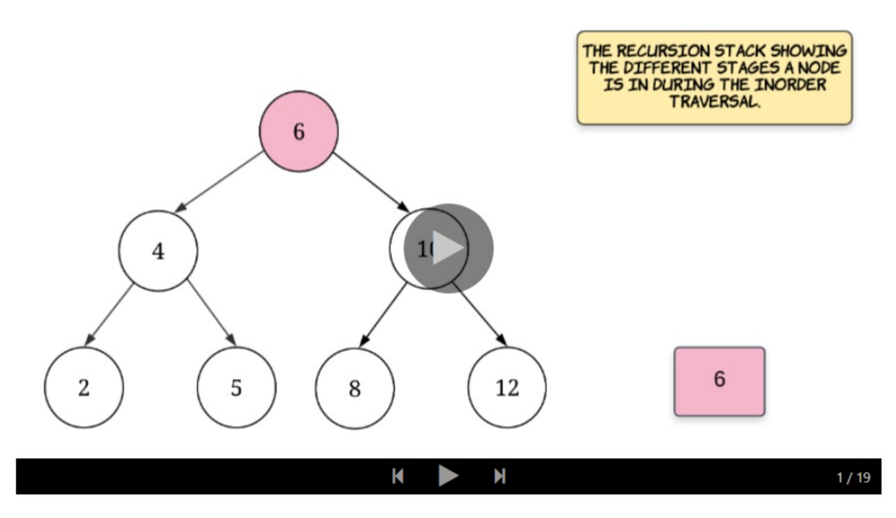
Before looking at the solutions for this problem, let's try and boil down what the problem statement essentially asks us to do. So, we need to implement an iterator class with two functions namely `next()` and `hasNext()`. The `hasNext()` function returns a boolean value indicating whether there are any more elements left in the binary search tree or not. The `next()` function returns the next smallest element in the BST. Therefore, the first time we call the `next()` function, it should return the smallest element in the BST and likewise, when we call `next()` for the very last time, it should return the largest element in the BST.

You might be wondering as to what could be the use case for an iterator. Essentially, an iterator can be used to *iterate over* any container object. For our purpose, the container object is a binary search tree. If such an iterator is defined, then the traversal logic can be abstracted out and we can simply make use of the iterator to process the elements in a certain order.

```
1. new_iterator = BSTIterator(root);
2. while (new_iterator.hasNext())
3.     process(new_iterator.next());
```

Now that we know the motivation behind designing a good iterator class for a data structure, let's take a look at another interesting aspect about the iterator that we have to build for this problem. Usually, an iterator simply goes over each of the elements of the container one by one. For the BST, we want the iterator to return elements in an ascending order.

> An important property of the binary search tree is that the inorder traversal of a BST gives us the elements in a sorted order. Thus, the inorder traversal will be the core of the solutions that we will look ahead.



Approach 1: Flattening the BST

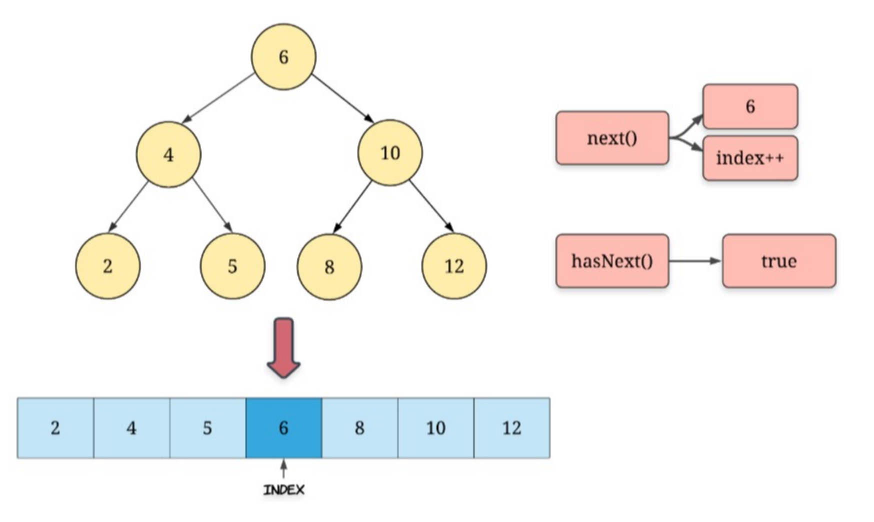
Intuition

In computer programming, an iterator is an object that enables a programmer to traverse a container, particularly lists. This is the Wikipedia definition of an iterator. Naturally, the easiest way to implement an iterator would be on an array like container interface. So, if we had an array, all we would need is a pointer or an index and we could easily implement the two required functions `next()` and `hasNext()`.

Hence, the first approach that we will look at is based on this idea. We will be using additional memory and we will flatten the binary search tree into an array. Since we need the elements to be in a sorted order, we will do an inorder traversal over the tree and store the elements in a new array and then build the iterator functions using this new array.

Algorithm

1. Initialize an empty array that will contain the nodes of the binary search tree in the sorted order.
2. We traverse the binary search tree in the inorder fashion and for each node that we process, we add it to our array `nodes`. Note that before processing a node, its left subtree has to be processed (or recursed upon) and after processing a node, its right subtree has to be recursed upon.
3. Once we have all the nodes in an array, we simply need a pointer or an index in that array to implement the two functions `next` and `hasNext`. Whenever there's a call to `hasNext`, we simply check if the index has reached the end of the array or not. For the call to `next` function, we simply return the element pointed by the index. Also, after a the `next` function call is made, we have to move the index one step forward to simulate the progress of our iterator.



Java | Python | Copy

```
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None

class BSTIterator:

    def __init__(self, root: TreeNode):

        # Array containing all the nodes in the sorted order
        self.nodes_sorted = []

        # Pointer to the next smallest element in the BST
        self.index = -1

        # Call to flatten the input binary search tree
        self._inorder(root)

    def _inorder(self, root):
        if not root:
            return
        self._inorder(root.left)
        self.nodes_sorted.append(root.val)
        self._inorder(root.right)

```

Complexity analysis

- Time complexity : $O(N)$ is the time taken by the constructor for the iterator. The problem statement only asks us to analyze the complexity of the two functions, however, when implementing a class, it's important to also note the time it takes to initialize a new object of the class and in this case it would be linear in terms of the number of nodes in the BST. In addition to the space occupied by the new array we initialized, the recursion stack for the inorder traversal also occupies space but that is limited to $O(h)$ where h is the height of the tree.
  - `next()` would take $O(1)$
  - `hasNext()` would take $O(1)$
- Space complexity : $O(N)$ since we create a new array to contain all the nodes of the BST. This doesn't comply with the requirement specified in the problem statement that the maximum space complexity of either of the functions should be $O(h)$ where h is the height of the tree and for a well balanced BST, the height is usually $logN$. So, we get great time complexities but we had to compromise on the space. Note that the new array is used for both the function calls and hence the space complexity for both the calls is $O(N)$.

Approach 2: Controlled Recursion

Intuition

The approach we saw earlier uses space which is linear in the number of nodes in the binary search tree. However, the reason we had to resort to such an approach was because we can control the iteration over the array. We can't really *pause* a recursion in between and then start it off sometime later.

> However, if we could simulate a controlled recursion for an inorder traversal, we wouldn't really need to use any additional space other than the space used by the stack for our recursion simulation.

So, this approach essentially uses a custom stack to simulate the inorder traversal i.e. we will be taking an iterative approach to inorder traversal rather than going with the recursive approach and in doing so, we will be able to easily implement the two function calls without any other additional space.

Things however, do get a bit complicated as far as the time complexity of the two operations is concerned and that is where we will spend a little bit of time to understand if this approach complies with all the asymptotic complexity requirements of the question. Let's move on to the algorithm for now to look at this idea more concretely.

Algorithm

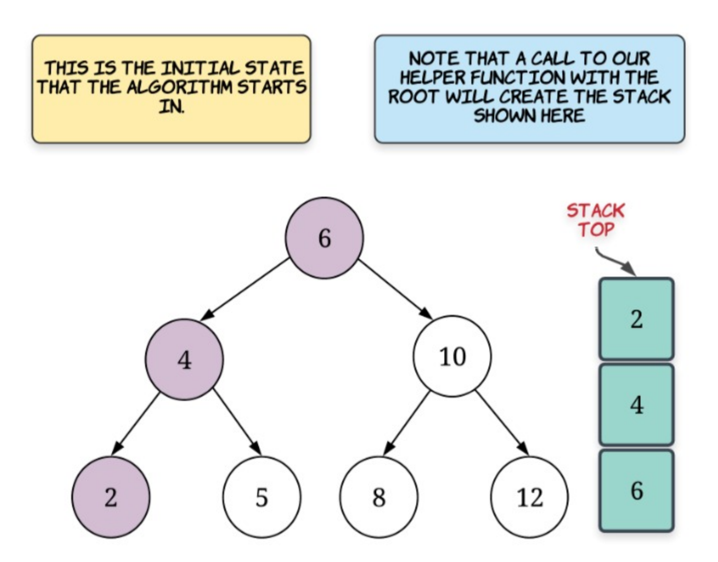
1. Initialize an empty stack `S` which will be used to simulate the inorder traversal for our binary search tree. Note that we will be following the same approach for inorder traversal as before except that now we will be using our own stack rather than the system stack. Since we are using a custom data structure, we can *pause* and *resume* the recursion at will.
2. Let's also consider a helper function that we will be calling again and again in the implementation. This function, called `_inorder_left` will essentially add all the nodes in the leftmost branch of the tree rooted at the given node `root` to the stack and it will keep on doing so until there is no `left` child of the `root` node. Something like the following code:

   ```
   def inorder_left(root):
     while (root):
       S.append(root)
       root = root.left
   ```

   For a given node `root`, the next smallest element will *always* be the leftmost element in its tree. So, for a given root node, we keep on following the leftmost branch until we reach a node which doesn't have a left child and that will be the next smallest element. For the root of our BST, this leftmost node would be the smallest node in the tree. Rest of the nodes are added to the stack because they are pending processing. Try and relate this with a dry run of a simple recursive inorder traversal and things will make a bit more sense.

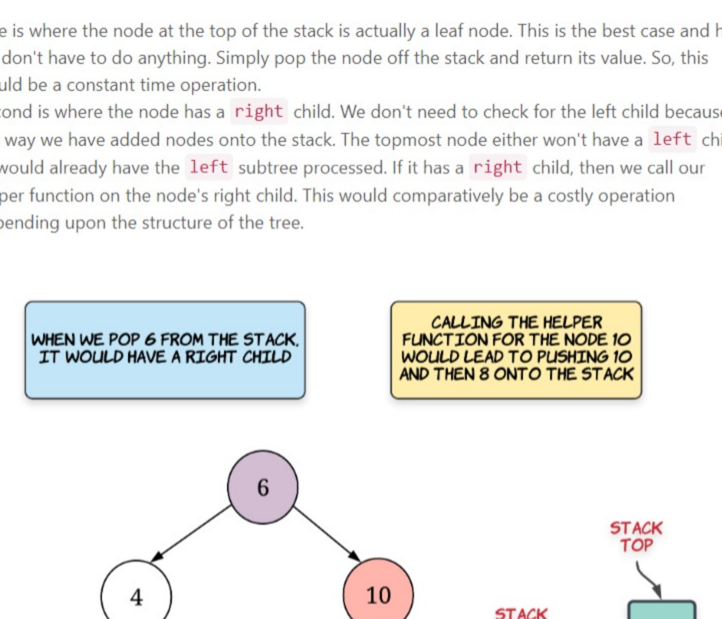
3. The first time `next()` function call is made, the smallest element of the BST has to be returned and then our simulated recursion has to move one step forward i.e. move onto the next smallest element in the BST. The invariant that will be maintained in this algorithm is that the stack top always contains the element to be returned for the `next()` function call. However, there is additional work that needs to be done to maintain that invariant. It's very easy to implement the `hasNext()` function since all we need to check is if the stack is empty or not. So, we will only focus on the `next()` call from now.

4. Initially, given the root node of the BST, we call the function `_inorder_left` and that ensures our invariant holds. Let's see this first step with an example.



5. Suppose we get a call to the `next()` function. The node which we have to return i.e. the next smallest element in the binary search tree iterator is the one sitting at the top of our stack. So, for the example above, that node would be `2` which is the correct value. Now, there are two possibilities that we have to deal with:

   - One is where the node at the top of the stack is actually a leaf node. This is the best case and here we don't have to do anything. Simply pop the node off the stack and return its value. So, this would be a constant time operation.
   - Second is where the node has a `right` child. We don't need to check for the left child because of the way we have added nodes onto the stack. The topmost node either won't have a `left` child or would already have the `left` subtree processed. If it has a `right` child, then we call our helper function on the node's right child. This would comparatively be a costly operation depending upon the structure of the tree.



6. We keep on maintaining the invariant this way in the function call for `next` and this way we will always be able to return the next smallest element in the BST from the top of the stack. Again, it's important to understand that obtaining the next smallest element doesn't take much time. However, some time is spent in maintaining the invariant that the stack top will *always* have the node we are looking for.

Java | Python | Copy

```
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None

class BSTIterator:

    def __init__(self, root: TreeNode):

        # Stack for the recursion simulation
        self.stack = []

        # Remember that the algorithm starts with a call to the helper function
        # with the root node as the input
        self._leftmost_inorder(root)

    def _leftmost_inorder(self, root):

        # For a given node, add all the elements in the leftmost branch of the tree
        # under it to the stack.
        while root:
            self.stack.append(root)
            root = root.left

    def next(self) -> int:
```

Complexity analysis

- Time complexity : The time complexity for this approach is very interesting to analyze. Let's look at the complexities for both the functions in the class:
  - `hasNext` is the easier of the lot since all we do in this is to return true if there are any elements left in the stack. Otherwise, we return false. So clearly, this is an $O(1)$ operation every time. Let's look at the more complicated function now to see if we satisfy all the requirements in the problem statement
  - `next` involves two major operations. One is where we pop an element from the stack which becomes the next smallest element to return. This is a $O(1)$ operation. However, we then make a call to our helper function `_inorder_left` which iterates over a bunch of nodes. This is clearly a linear time operation i.e. $O(N)$ in the worst case. This is true.

    > However, the important thing to note here is that we only make such a call for nodes which have a right child. Otherwise, we simply return. Also, even if we end up calling the helper function, it won't always process N nodes. They will be much lesser. Only if we have a skewed tree would there be N nodes for the root. But that is the only node for which we would call the helper function.

    Thus, the amortized (average) time complexity for this function would still be $O(1)$ which is what the question asks for. We don't need to have a solution which gives constant time operations for *every* call. We need that complexity on average and that is what we get.
- Space complexity: The space complexity is $O(h)$ which is occupied by our custom stack for simulating the inorder traversal. Again, we satisfy the space requirements as well as specified in the problem statement.

Rate this article: ★ ★ ★ ★ ★

Previous Next

Comments: 51

Sort By

Type comment here... (Markdown is supported)

Preview Post

lifran ★ 122 August 12, 2019 3:08 AM

When analyzing amortized time complexities, I find it easiest to reason that each node gets pushed and popped exactly once in next() when iterating over all N nodes.
That comes out to 2N * O(1) over N calls to next(), making it O(1) on average, or O(1) amortized.

109 Share Reply

SHOW 7 REPLIES

prateekiiest ★ 51 June 14, 2019 10:33 AM

This is a great analysis.

Thanks

42 Share Reply

Alquimista3301 ★ 11 June 18, 2019 4:07 AM

This is a great article 👏👏👏 thanks for sharing

11 Share Reply

SHOW 1 REPLY

zhuangjianing ★ 14 April 7, 2020 12:33 AM

If I correctly understand it, the second solution is just an iterative inorder traversal. Am I right?

7 Share Reply

SHOW 2 REPLIES

pingrunhuang ★ 6 June 17, 2019 5:29 AM

Can someone explain me why this is just using O(h) of memory? From my point of view, the nodes_sorted from the first solution occupied O(N) where N is the number of nodes.

5 Share Reply

SHOW 7 REPLIES

apg3141 ★ 3 May 8, 2020 7:55 PM

Did anyone actually think/implement of the solution 2 by themselves, it feels really far fetched. I am not sure how we can solve it in an hour long interview. I guess the idea of using a stack and populating left branch is so novel that you either know it or you don't . Just validating if anyone else feels this way

3 Share Reply

SHOW 2 REPLIES

gakhilesh ★ 7 May 23, 2020 11:16 PM

Very basic java solution :

class BSTIterator {

private LinkedList<Integer> list;

Read More

1 Share Reply

zeus1985 ★ 64 January 20, 2020 5:31 AM

The question says that each opreation should be done in O(1), but the second approach is at O(longest branch) in the worst case. I've implemented the same one, but the array implementation is faster as per the submission. It shows 6%/9% for the second implementation.

1 Share Reply

SHOW 1 REPLY

zekunf ★ 1 July 3, 2019 7:50 AM

Space can be O(1). Check posts in Discuss if interested :)

1 Share Reply

SHOW 1 REPLY

RG0887 ★ 13 July 21, 2019 12:09 AM

Thanks for the detailed write-up!

0 Share Reply

‹ 1 2 3 4 5 6 ›