Binary Search Tree Iterator

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.

Note: `next()` and `hasNext()` should run in average O(1) time and uses O(*h*) memory, where *h* is the height of the tree.

Credits:
Special thanks to @ts for adding this problem and creating all test cases.

Solution 1

I use Stack to store directed left children from root. When next() be called, I just pop one element and process its right child as new root. The code is pretty straightforward.

So this can satisfy O(h) memory, hasNext() in O(1) time, But next() is O(h) time.

I can't find a solution that can satisfy both next() in O(1) time, space in O(h).

Java:

```
public class BSTIterator {
    private Stack<TreeNode> stack = new Stack<TreeNode>();

    public BSTIterator(TreeNode root) {
        pushAll(root);
    }

    /** @return whether we have a next smallest number */
    public boolean hasNext() {
        return !stack.isEmpty();
    }

    /** @return the next smallest number */
    public int next() {
        TreeNode tmpNode = stack.pop();
        pushAll(tmpNode.right);
        return tmpNode.val;
    }

    private void pushAll(TreeNode node) {
        for (; node != null; stack.push(node), node = node.left);
    }
}
```

C++:

```
class BSTIterator {
    stack<TreeNode *> myStack;
public:
    BSTIterator(TreeNode *root) {
        pushAll(root);
    }

    /** @return whether we have a next smallest number */
    bool hasNext() {
        return !myStack.empty();
    }

    /** @return the next smallest number */
    int next() {
        TreeNode *tmpNode = myStack.top();
        myStack.pop();
        pushAll(tmpNode->right);
        return tmpNode->val;
    }

private:
    void pushAll(TreeNode *node) {
        for (; node != NULL; myStack.push(node), node = node->left);
    }
};
```

Python:

```
class BSTIterator:
    # @param root, a binary search tree's root node
    def __init__(self, root):
        self.stack = list()
        self.pushAll(root)

    # @return a boolean, whether we have a next smallest number
    def hasNext(self):
        return self.stack

    # @return an integer, the next smallest number
    def next(self):
        tmpNode = self.stack.pop()
        self.pushAll(tmpNode.right)
        return tmpNode.val

    def pushAll(self, node):
        while node is not None:
            self.stack.append(node)
            node = node.left
```

written by xcv58 original link here

Solution 2

My idea comes from this: My first thought was to use inorder traversal to put every node into an array, and then make an index pointer for the next() and hasNext(). That meets the O(1) run time but not the O(h) memory. O(h) is really much more less than O(n) when the tree is huge.

This means I cannot use a lot of memory, which suggests that I need to make use of the tree structure itself. And also, one thing to notice is the "average O(1) run time". It's weird to say average O(1), because there's nothing below O(1) in run time, which suggests in most cases, I solve it in O(1), while in some cases, I need to solve it in O(n) or O(h). These two limitations are big hints.

Before I come up with this solution, I really draw a lot binary trees and try inorder traversal on them. We all know that, once you get to a TreeNode, in order to get the smallest, you need to go all the way down its left branch. So our first step is to point to pointer to the left most TreeNode. The problem is how to do back trace. Since the TreeNode doesn't have father pointer, we cannot get a TreeNode's father node in O(1) without store it beforehand. Back to the first step, when we are traversal to the left most TreeNode, we store each TreeNode we met (They are all father nodes for back trace).

After that, I try an example, for next(), I directly return where the pointer pointing at, which should be the left most TreeNode I previously found. What to do next? After returning the smallest TreeNode, I need to point the pointer to the next smallest TreeNode. When the current TreeNode has a right branch (It cannot have left branch, remember we traversal to the left most), we need to jump to its right child first and then traversal to its right child's left most TreeNode. When the current TreeNode doesn't have a right branch, it means there cannot be a node with value smaller than itself father node, point the pointer at its father node.

The overall thinking leads to the structure Stack, which fits my requirement so well.

```java
/**
 * Definition for binary tree
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */

public class BSTIterator {

    private Stack<TreeNode> stack;
    public BSTIterator(TreeNode root) {
        stack = new Stack<>();
        TreeNode cur = root;
        while(cur != null){
            stack.push(cur);
            if(cur.left != null)
                cur = cur.left;
            else
                break;
        }
    }

    /** @return whether we have a next smallest number */
    public boolean hasNext() {
        return !stack.isEmpty();
    }

    /** @return the next smallest number */
    public int next() {
        TreeNode node = stack.pop();
        TreeNode cur = node;
        // traversal right branch
        if(cur.right != null){
            cur = cur.right;
            while(cur != null){
                stack.push(cur);
                if(cur.left != null)
                    cur = cur.left;
                else
                    break;
            }
        }
        return node.val;
    }
}

/**
 * Your BSTIterator will be called like this:
 * BSTIterator i = new BSTIterator(root);
 * while (i.hasNext()) v[f()] = i.next();
 */
```

Solution 3

the idea is same as using stack to do Binary Tree Inorder Traversal

```
public class BSTIterator {

        Stack<TreeNode> stack =  null ;
        TreeNode current = null ;

        public BSTIterator(TreeNode root) {
              current = root;
              stack = new Stack<> ();
        }

        /** @return whether we have a next smallest number */
        public boolean hasNext() {
              return !stack.isEmpty() || current != null;
        }

            /** @return the next smallest number */
        public int next() {
            while (current != null) {
                stack.push(current);
                current = current.left ;
            }
            TreeNode t = stack.pop() ;
            current = t.right ;
            return t.val ;
        }
    }
```

written by scott original link here