CS-UY 1134: Data Structures and Algorithms
Lab 9: Binary Search Trees

Friday, August 2, 2019

Instructions

This lab will focus on binary search trees.

Try to solve all of the problems by the end of the lab time. As added practice, it is a good idea to try writing your code with pen and paper first, as you will have to do this on the exam.

When you type up your code, write some test cases. Try multiple test cases for each function you have to implement, and consider edge cases carefully.

As a reminder, you *may* (and are encouraged to!) discuss these problems with your classmates during lab and help each other solve them. However, make sure the code you submit is your own.

What to submit

If you believe you have solved all the problems, let a TA or the professor know and they will check your work. This is a good chance to get feedback on problems similar to what you might see on the exam.

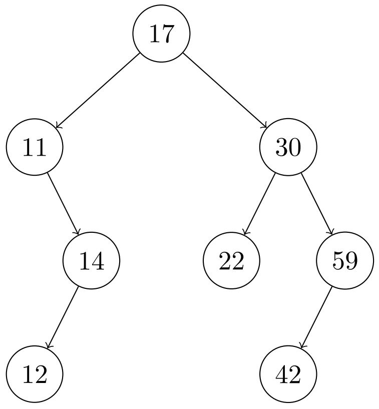
If you don't finish by the end of the lab, you may submit something on NYU Classes by Sunday August 4, 11:55 PM.

Either when a TA has checked your solutions, or by the Sunday submission deadline, submit one ".py" file containing your solutions to the programming problems, and one additional file containing your answers to the written problems. Submit your files to the "Lab 9: Linked Lists and Binary Trees" assignment on NYU Classes.

1 Written Problems (Do These!)

Complete the following written problems, and submit your answers on Classes along with your code. Submit one file with all of your answers (so you should submit a

1. Consider the following binary search tree.

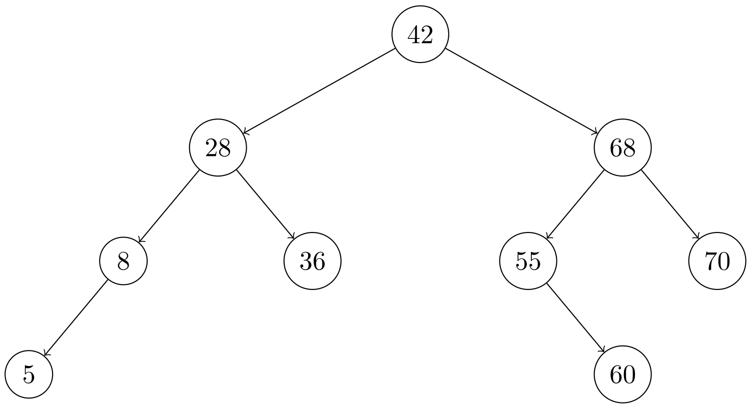


Perform the following operations on the tree cumulatively. (This means perform each operation on the tree that resulted from the previous operation.)

(a) Insert 15
(b) Insert 28
(c) Delete 30
(d) Insert 30
(e) Delete 11
(f) Delete 17

Note: Do not attempt to write code to create the tree and perform the operations. Follow the process by hand to show how the insertions and deletions would modify the tree.

2. Consider the following binary search tree.



Perform the following operations:

(a) Pre-order traversal
(b) Post-order traversal
(c) Inorder traversal

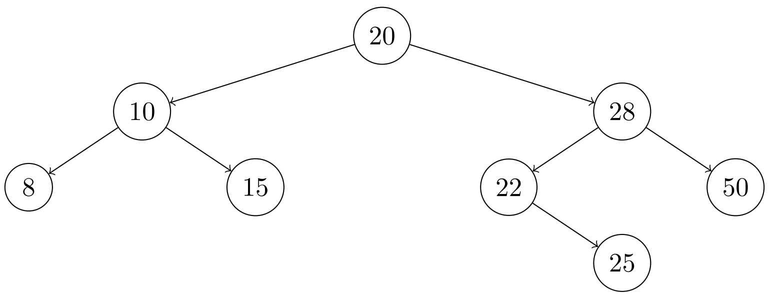
2 Check if two Binary Search Trees Have The Same Keys

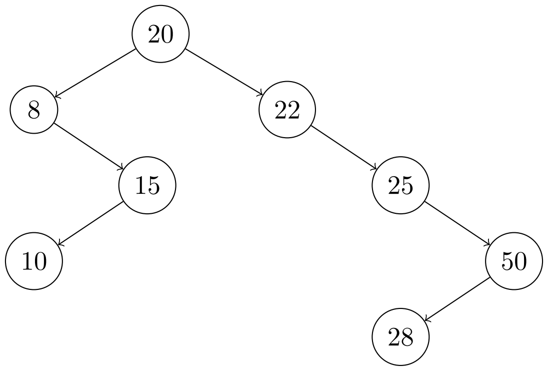
For this problem, we care only about the keys of the binary search tree.

Write a function that takes in two binary search tree objects and checks if the two contain the same keys. Implement this as

```
def are_bst_keys_same(bst1, bst2)
```

For example, given the two trees:





the function call to `are_bst_keys_same` should return True.

Implementation Requirement

1. Your implementation must run in **linear time**.

There is no space requirement, but try to think about what the space complexity of your solution is.

3 Check if Tree is BST

Implement a function to check if a binary tree is a binary search tree (that is, it has the binary search tree property).

You will do this **recursively**, using a recursive helper function, `is_bst_helper`. The function `is_bst` will make an initial call to the helper function. Your code should look like this:

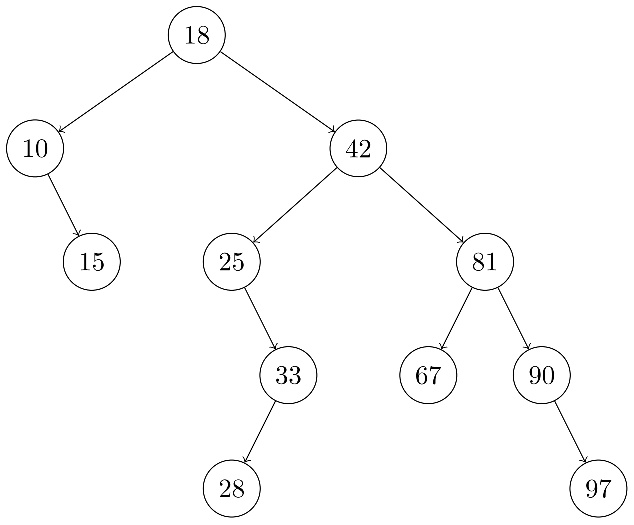
```
def is_bst(binary_tree):
    return is_bst_helper(binary_tree.root)[0]

def is_bst_helper(subtree_root):
    # Your code here
```

Note that the function `is_bst` takes a parameter `binary_tree`, *of type* ***`LinkedBinaryTree`***. Similarly, `is_bst_helper` takes as parameter a node, of type `LinkedBinaryTree.Node`.

Hint: The function `is_bst_helper` should return multiple values as a tuple: a boolean value, and the min and max of the subtree rooted by `subtree_root`.

4 Lowest Common Ancestor

Given two nodes in a binary tree, their *lowest common ancestor* is the deepest node which is an ancestor of both nodes. For example, in the tree below, the lowest common ancestor of 28 and 90 is 42; and the lowest common ancestor of 81 and 97 is 81.



Implement a function that, given two nodes in a *binary search tree*, returns the node that is the lowest common ancestor of the given nodes. Implement this as:

```
def lca_bst(bst, node1, node2):
    '''
    params: bst -> BinarySearchTreeMap
            node1 -> BinarySearchTreeMap.Node
            node2 -> BinarySearchTreeMap.Node

    returns: the lowest common ancestor of node1 and node2 -> BinarySearchTree.Node
    '''
```

Implementation Requirement: Your implementation *must* run in time $O(h)$, where h is the height of the Binary Search Tree.

Think about: How might you implement this for a general binary trees, where the input is not necessarily a binary search tree? Can you still do it in time $O(h)$?