CSC148 Summer 2018: Lab 8

Introduction

The goals of this lab are:

- To get you familiar with how Binary Search Trees work
- To give you practice working with Binary Search Trees
- To give you practice thinking about runtime analysis

Don't hesitate to make use of other resources for this lab, including the course notes, your TAs, instructor, or other students.

General Lab Notes

1. Make sure you have lab_pyta.txt downloaded and placed in the directory (or directories) where you'll be working.
2. To use PythonTA, include the following code (if you already have a main block, just add the body to the end of it):

```
if __name__ == '__main__':
    import python_ta
    python_ta.check_all(config="lab_pyta.txt")
```

Your lab_pyta.txt should be in the same folder as the .py files you're running. PythonTA will raise errors regarding style, specifying the lines you need to fix. You should get familiar with what the errors mean, and how to fix them: this will be important for your exercises and assignments.

Getting Started

This lab will have you using BinarySearchTrees which only have very basic functionality. Download lab_binary_search_trees.py or write your own BinarySearchTree classes. This class contains only the ___init___ and ___str___ methods for BinarySearchTrees, as well as an insert function.

BinarySearchTrees follow the same class design as BinaryTrees; so you can work with them in the same way. The only rule that we need to follow is that:

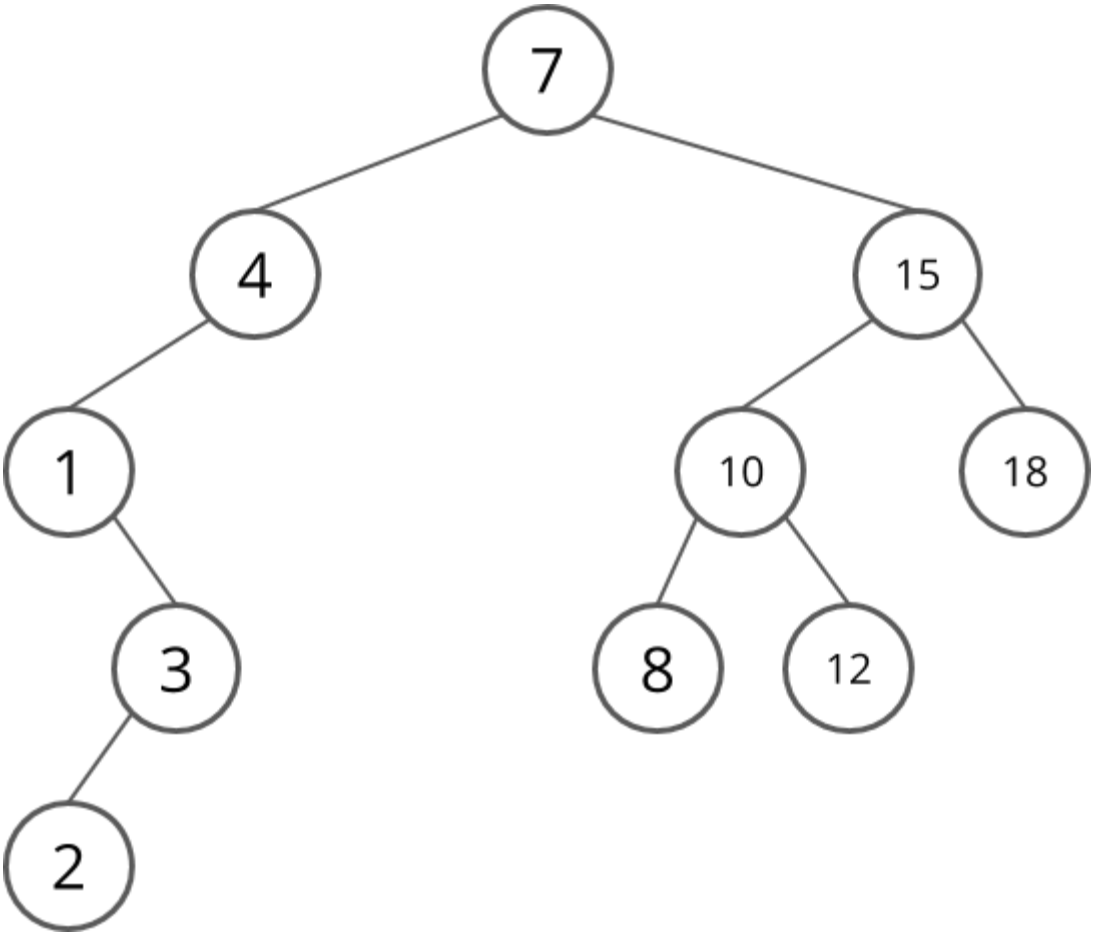
- All items in the left subtree of a BinarySearchTree must have values less than the root's value.
- All items in the right subtree of a BinarySearchTree must have values greater than the root's value.

Assume that we'll never have duplicates for BinarySearchTrees.

Creating a BinarySearchTree using insert

Within the `if __name__ == '__main__'` block, the BinarySearchTree t was created. Make sure you understand how it was created, what it looks like, and what the subtrees in it are.

Create a new tree called `large_tree` using the `insert()` function such that it looks like the following:



What order did you have to insert values in?

Insertion Order

Look at the BinarySearchTree (BST) t which was defined in the `if __name__ == '__main__'` block. If we use insert(), we are guaranteed to always have a BST that doesn't violate BST properties. If we call print(t) we see that it looks like:

```
          10
    8         12
6     9
```

What would happen if we changed the order of insertions? What if we inserted 6 before 8? If we inserted 9 before 8?

Draw out the different BinarySearchTrees that result from the different orders, and make sure you understand how the insertion order affects the structure of the BinarySearchTrees.

Finding the minimum value in a BST (find_min)

Write a function called find_min() that takes in a BinarySearchTree and returns the smallest value in it. Avoid making any unnecessary recursive calls.

In the worst case scenario, how many nodes would you have to look at (relative to the size of a BST) to find the minimum? What is the worst case scenario?

What about the best case scenario?

Deleting and swapping with the right subtree (delete_right)

In lecture, we deleted a node with two children by swapping values with the largest value in the left subtree, and then deleting the node that had that value from the left subtree.

Write a function called delete_right() that takes in a BinarySearchTree or None and a value, and deletes the node with that value. In the case of a node with 2 children, swap with the smallest value in the right subtree instead.

Checking if a BinarySearchTree Violates any BST properties (is_valid_bst)

If we create a BinarySearchTree without using insert(), there's a possibility that BST properties could be violated (i.e. a number could be in the incorrect spot).

Write a function called is_valid_bst() that takes in a BinarySearchTree or None and returns True if the BinarySearchTree is a valid BinarySearchTree (i.e. it doesn't violate any BST properties).

How many nodes do you have to check when calling is_valid_bst()? If you have to use a function like find_min() or find_max(), make sure you account for the number of nodes you check when calling those too.

What's the runtime of this for a BinarySearchTree that's completely balanced? What about a BinarySearchTree that's completely unbalanced?

Counting the number of nodes in a BST (count_nodes)

Write a function called count_nodes() that takes in a BinarySearchTree or None and returns the total number of nodes in it.

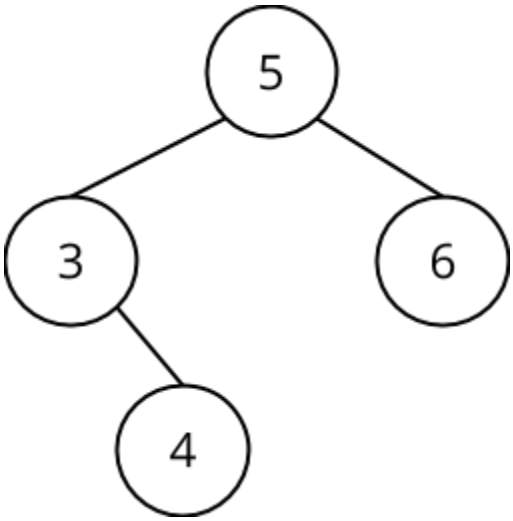
How many nodes do you have to check when calling count_nodes()? How does this number grow as you change the size of the BinarySearchTree? Does this change for differently balanced BinarySearchTrees?

Finding the nth smallest number in a BST (find_nth_smallest)

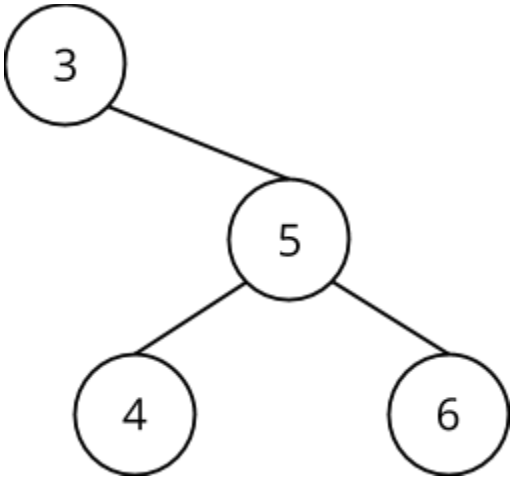
Write a function called find_nth_smallest() that takes in a BinarySearchTree or None and returns the n^{th} smallest value in the BinarySearchTree. You will likely want to use count_nodes() within find_nth_smallest().

Rotating a BST clockwise (rotate_right)

We can adjust BinarySearchTrees while still keeping all BinarySearchTree properties. One way to do this is to rotate the entire BinarySearchTree itself. For example, if we have a BinarySearchTree that looks like this:



We can rotate it clockwise in order to get this BinarySearchTree:



For an animated illustration, see this GIF from Wikipedia -- this illustrates both a right rotation, and a left rotation.

In this, the following changes occur:

- The original root of the BinarySearchTree becomes the root of the left subtree.
- The original root of the BinarySearchTree becomes the right subtree of the new root.
- What used to be the right subtree of the original left subtree is now the left subtree of the original root.

All BinarySearchTree properties are maintained in this rotation. Why? What properties does the left subtree have? What properties does the left subtree's right subtree have? What do we know about its values? In the rotated BinarySearchTree, what do we know about the values of the new right subtree and its left subtree?

Write a function called rotate_right() that takes in a BinarySearchTree and returns the root of the rotated BinarySearchTree. If there is no left child, then no rotation should occur.