CMPSC 132: Programming and Computation II

Lab 5 (10 points) Due date: March 18th 2021, 11:59 PM

Goal: The goal of this lab is for you to gain a deeper understanding of the binary search tree data structure by working to implement additional functionality to the BinarySearchTree class.

General instructions:

- The work in this assignment must be your own original work and be completed alone.
- The instructor and course assistants are available on Teams and with office hours to answer any questions you may have. You may also share testing code on Teams.
- A doctest is provided to ensure basic functionality and may not be representative of the full range of test cases we will be checking. Further testing is your responsibility.
- Debugging code is also your responsibility.
- You may submit more than once before the deadline; only the latest submission will be graded.

Assignment-specific instructions:

- Download the starter code file from Canvas. Do not change the function names or given starter code in your script.
- You are not allowed to use any other data structures to manipulate or sort data.
- You are not allowed to modify the given constructor or any given code.
- You are not allowed to use the queue functions from the Python library.
- Additional examples of functionality are provided in each function's doctest
- Some methods are easier to implement if you use recursion, revisit the hands-on lecture for binary trees is necessary.
- If you are unable to complete a function, use the pass statement to avoid syntax errors

Submission format:

- Submit your code in a file named LAB5.py file to the Lab 5 Gradescope assignment before the due date.
- As a reminder, code submitted with syntax errors does not receive credit, please run your file before submitting.

Section 1: The BinarySearchTree class

We discussed the binary search tree data structure in part 1 of Module 6 and implemented some functionality in the hands-on lecture. Using parts of that code as your starter code, your assignment is to implement additional functionality on top of the BinarySearchTree class.

As a reminder, in the hands-on lecture we modified the Node class to have both a left and right pointer (instead of just next) to implement a tree data structure. Our Node class now looks like:

```
class Node:
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None
```

For the purposes of this assignment, you can assume that all values in the binary search tree will be unique numbers.

Methods

Type	Name	Description
bool	isEmpty(self)	Tests to see whether the tree is empty or not
Node	_mirrorHelper(self, node_object)	Swaps left and right children of all non-leaf nodes
int or float	getMin(self)	Gets the minimum value in the tree
int or float	getMax(self)	Gets the maximum value in the tree
bool	__contains__(self, item)	Checks if a value is present in the tree
int	getHeight(self, node_object)	Gets the height of a node in the tree

The starter code also contains the property method *getInorder* and the method *_inorderHelper*, those will not require any changes. They are provided to easily see the effects of changes to the tree to see that things work correctly. Use it for debugging and testing purposes only.

Section 1: The BinarySearchTree class

isEmpty(self) (0.5 pt)

Tests to see whether the tree is empty or not.

Output	
bool	True if the tree is empty, False otherwise.

_mirrorHelper(self, node_object) (2 pt)

The starter code contains a method called mirror, that takes the BinarySearchTree object and returns a new BinarySearchTree object that represents a mirror image of the original tree. For example, for the following tree:

```
         9
       /   \
      2     26
     / \      \
    1   7      36
```

Inorder traversal: 1, 2, 7, 9, 26, 36

A call to mirror results in a new tree as shown:

```
         9
       /   \
     26     2
     /     / \
    36    7   1
```

Inorder traversal: 36, 26, 9, 7, 2, 1

Complete the implementations of _mirrorHelper that takes a reference to a Node in the original tree and interchanges the links of the left and right children of all non-leaf nodes in the new tree. For example, if the node is a reference to 2, then in the new tree, 2.left=7 and 2.right=1. Do not modify the implementation for mirror in any way, otherwise you will not receive credit

Input		
Node	node_object	A node in the tree

Output	
Node	A reference to the root of the new tree

getMin(self), getMax(self) (1 pt each)

Property methods that return the minimum/maximum Node value in the tree. You should not use the values of getInorder in any way to complete these methods. Your methods must search in the proper sections of the tree only.

Output	
Node	The Node with the minimum/maximum value in the tree
None	None is returned if the tree is empty

__contains__(self, item) (3 pt)

Checks if a value is present in the tree by overloading the `in` operator. If you are planning on using recursion to implement this method, the nature of the special method does not allow modifications in the parameter list, adding a helper method to assist __contains__ could be useful. If you are following and iterative approach, a helper method is not required.

Input		
int or float	item	The value to check if it exists in the tree

Output	
bool	True if the value is in the tree, False otherwise

getHeight(self, node_object) (2.5 pt)

Gets the height of a node in the tree. You can assume that the node exists in the tree. As a reminder, the height of a node is the number of edges from that node to the deepest leaf, in other words, max(height_left_subtree, height_right_subtree). The height of a tree is the height of the root node. The logic defined in the traversals for the HandsOn BinaryTree class could be helpful here!

Input		
Node	node_object	The node to check the height of

Output	
int	The height of the node in the tree