Lab 5 (10 points) **Due date:** October 22, 2024, 11:59 PM

Before you start…

Keep in mind these tips from your instructor:

- Watch the lectures in Module 6.1 before you start this assignment, especially the hands-on videos
- Do not compare yourself to others: An assignment might take you hours, compared to other fellow students, but every student learns in a different way. Please reach out to the course staff id you are stuck and need help
- Outline a strategy before you start typing code and share it with the course staff for additional support
- There are several ways to traverse a binary tree, all forms were described and implemented in the lectures
- Binary search trees have an order property, that means, certain methods do not require the traversal of the entire tree
- Your tree traversals should not modify the original tree!
- Ask questions using our Lab 5 channel in Microsoft Teams

REMINDER: As you work on your coding assignments, it is important to remember that passing the examples provided does not guarantee full credit. While these examples can serve as a helpful starting point, it is ultimately your responsibility to thoroughly test your code and ensure that it is functioning correctly and meets all the requirements and specifications outlined in the assignment instructions. Failure to thoroughly test your code can result in incomplete or incorrect outputs, which will lead to deduction in points for each failed case.

Create additional test cases and share them with you classmates on Teams, we are in this together!

90% > Passing all test cases 10% > Clarity and design of your code

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	Short Description
bool	isEmpty(self)	Tests to see whether the tree is empty or not
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
bool	isBalanced_helper(self, node_object)	Checks if the tree is balanced

NOTE: None of the methods in this assignment should mutate (modify) the given binary search tree. The use of Python lists is not allowed anywhere in this assignment.

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.

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

Property methods that return the minimum/maximum Node value in the tree. Your methods must search in the proper sections of the tree only.

Output	
int or float	The value of the node with the minimum/maximum value in the tree
None	None is returned if the tree is empty

__contains__(self, item) (2.5 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 (not required), 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) (2 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 (Recursively call the function for the left and right subtrees of the current node)

Input		
Node	node	The node to check the height of

Output	
int	The height of the node in the tree

isBalanced_helper(self, node) (3 pt)

In lectures, we presented the concept of balanced binary trees. In general, a tree is said to be balanced when the balance condition is satisfied:

Balance condition: balance of every node is between -1 and 1

where balance(node) = height(node's left subtree) – height(node's right subtree)

This function returns True if the Binary Search Tree is balanced, False otherwise. This method will be called by the isBalanced property method provided in the starter code (do not modify the code provided)

Hint:

To verify a tree is balanced, you must check that for every node *u* in the tree, the heights of the children of *u* differ by at most 1. This means that you must perform a full traversal of the tree (using the same strategy presented in the preorder, inorder and postorder implementations from the HandsOn Lectures). For each node calculate the heights of the left and right subtrees by calling the getHeight method you implemented for the class, and finally, check the balance condition.

Input		
Node	node	The node to check the balance condition of

Output	
bool	True if tree is balanced, False otherwise