Computer Science I CSCI-141
Binary Search Tree Homework

1 Problem

Eyecu is a company that keeps track of everybody in the city of Orwellia. At any time, they need to track some subset of the citizens, and this collection is constantly changing, with individuals added and removed for reasons known only to Eyecu. They have decided that the right way to maintain this information is with a variant of a binary search tree. As we know, binary search trees are efficient data structures for maintaining sorted datasets that are constantly being updated with insertions and deletions.

To keeps things from getting too complicated for this exercise, we will only deal with insertions to the binary search tree, and not deal with deletions. Also, we will just use unique numbers to represent individuals. That is, the data field for each node in the tree will just be a non-negative integer.

Eyecu has some specific tasks that they need to be able to perform quickly at any time. For any arbitrary node in the tree, at any time, they need to be able to:

- Determine the height of the subtree rooted at that node.
- Determine if the size of the subtree rooted at that node is equal to the data value of that node.
- Determine if the subtree rooted at that node ever becomes excessively imbalanced.

They have asked you to design an enhanced binary search tree data structure that can allow them to quickly compute this information.

2 Definitions

The terms used in the requirements are defined as follows:

- The height of a node in a binary search tree is defined as the number of steps (hopping from a node to one of its children) along the longest downward path from that node to a leaf. We define the height of an empty tree to be equal to -1. The height of a leaf node is 0 (it has no children). Recursively, the height of a node is equal to 1 + the maximum of the height of its two subtrees.
- The size of a subtree rooted at any arbitrary node is equal to the number of nodes in the tree rooted at that node. The size of an empty tree is 0. The size of a leaf node is 1. Recursively, the size of a tree is equal to 1 + the size of the left subtree + the size of the right subtree.
- The imbalance of a subtree rooted at any arbitrary node is equal to the absolute value of the difference in height of its two subtrees. We define the imbalance of an empty tree to be 0.

3 Starter Code and Requirements

Eyecu is giving you the following class definition to work with:

```
class EyecuBST(rit_object):
    __slots__ = ('left', 'right', 'parent', 'value', 'height', 'size', 'imbalance')
    _types   =  ('EyecuBST', 'EyecuBST, 'EyecuBST', int, int, int, int)
```

The `EyecuBST` class will be used to represent an enhanced binary search tree. It has the characteristics of a binary search tree plus a `parent` reference that allows a node to be traced upward in the tree toward the root. It also contains additional data associated with each node in the tree. When you create an `EyecuBST` object (we'll refer to it below as `eBST`), its slots will have the following usage:

- `left`: a reference to the left sub-tree of `eBST`.
- `right`: a reference to the right sub-tree of `eBST`.
- `parent`: a reference to the node (tree) that is the parent of `eBST`. All nodes except for the root of a tree have a parent node; the parent of the root is specified as `None`.
- `value`: an integer representing the data associated with this node. For our problem, each person being tracked has a unique non-negative integer `value`.
- `height`: the height (as defined above) of the `eBST` tree.
- `size`: the size (as defined above) of the `eBST` tree.
- `imbalance`: the imbalance (as defined above) of the `eBST` tree.

Along with this class definition, Eyecu has provided you a function to create an `EyecuBST` object:

- `createEyecuBST`: creates and returns an `EyecuBST` object. This function should be used whenever a new value is to be inserted into the tree as a leaf node.

You will implement the following binary search tree functionality that Eyecu is interested in:

- `insert`: takes as inputs an `EyecuBST` tree (or `None` representing an empty tree) and an element to be added to the tree. It creates a new node in the tree for the element, and places it in the appropriate location of the tree according to the rules of binary search trees. It returns the updated tree.
- `treeHeight`: takes an `EyecuBST` tree as input and computes and returns the height of the tree. If the input tree is empty (represented by `None`), it returns the value -1.
- `treeSize`: takes an `EyecuBST` tree as input and computes and returns the size of the tree. If the input tree is empty (represented by `None`), it returns the value 0.
- `treeImbalance`: takes an `EyecuBST` tree as input and computes and returns the imbalance of the tree. If the input tree is empty (represented by `None`), it returns the value 0.
- `findNode`: takes as input an `EyecuBST` tree and a target value (a non-negative integer) and searches for the node in the tree with that value. It returns a reference to that node in the tree. It returns `None` if the value is not present in the tree.

You may find it useful to define additional helper functions. However, the only mandatory functions are the listed ones.

The file `eyecuFunc.py` is being provided to you. It contains a class definition, a function to create an `EyecuBST` object, and stubs for the functions that you must write. You can download it as part of the `eyecu.zip` file from the course website in the same folder where this writeup is contained.

4 The EyecuBST Tester

In order to evaluate whether your data structures and algorithms meet Eyecu's requirements, they have provided a test module that you can run against your solution to see if it gives correct answers, and gives them fast enough.

The test module is named `eyecuTester.py` and is contained in the `eyecu.zip` file listed above. The test module should be kept in the same folder as your `eyecuFunc.py` file.

The test module has two lines at the bottom that can be uncommented to test your program in two different ways. Other than that, you do not need to modify the test module in any way.

4.1 Testing Your Insert Function

The first test function can be used to check that you have correctly implemented insert such that a valid binary search tree is constructed. It is recommended that you implement your `insert` function first, and confirm that you are correctly building a binary search tree, before focusing on the other functionality.

This test function creates an input list of numbers to put into a tree, and calls your `insert` function repeatedly, confirming after each insert that you return a valid binary search tree.

4.2 Testing All Functionality for Correctness and Speed

The second test function can be used to check your entire program for correctness and speed. The test module does the following:

- It prompts for four different inputs that Eyecu is interested in for a given test. The inputs are:
  - The number of elements (non-negative integers representing people) to put in the binary search tree.
  - The specific target person (as a non-negative integer) being evaluated in this test. (Below, the tree node having this value is referred to as the target node).
  - The maximum height of a tree rooted at the target node that should be allowed. The test will automatically terminate if at any stage the tree rooted at the target node exceeds this maximum height.
  - The maximum imbalance of a tree rooted at the target node that should be allowed. The test will automatically terminate if at any stage the tree rooted at the target node exceeds this maximum imbalance.

- Creates a randomized list containing non-negative numbers for the test. A seed is provided to the random number generator which causes the same random sequence to occur for a given number of elements. This allows the tester to have randomness and repeatability.
- A timer records how long it takes to insert all of the elements into the binary search tree, and also checks to make sure that the conditions for continuing are met after each insertion.
- The tester keeps track of whether or not the target node is in the tree yet. It doesn't do any checking until the target node appears in the tree.
- After the target node is in the tree, for each iteration the tester checks that:
  - the height of the tree rooted at the target node doesn't exceed the provided maximum.
  - the size of the tree rooted at the target node doesn't equal the value of the target.
  - the imbalance of the tree rooted at the target node doesn't exceed the provided maximum.

  If any of these checks fails, the test is automatically terminated at that point. Otherwise, the test continues until the complete set of values has been added to the binary search tree.
- When the test is complete, the tester reports back the time it took to complete the test, along with the characteristics of the tree rooted at the target node at the time the test was terminated.

5 Where You Come Into Play

The tester calls your functions: `insert, findNode, treeHeight, treeSize, treeImbalance`. These need to provide correct answers. Furthermore, they need to provide them quickly. There are different ways to implement this functionality. Not all implementations provide the speed that Eyecu requires.

6 Example Test Output

The following test outputs are provided so that you can check the correctness of your solution, as well as the speed of your solution. Use these exact inputs to check your results. You should do additional testing on your own to confirm the correctness of your algorithms.

- Example 1: 8 element BST. The random seed results in the values entered in the following order: [3,4,6,7,2,5,0,1]. This test terminates when the size of the tree rooted at the target node matches the value of the target node (2).

```
Input number of elements to put in BST: 8
Input specific target number to evaluate: 2
Input maximum height of target tree to allow: 2
Input maximum imbalance of target tree to allow: 2
```

```
Test terminated after 7 nodes entered into tree.
Elapsed time:  0.0006770000000000109 seconds
Target Value: 2
Height of tree rooted at target: 1
Size of tree rooted at target: 2
Imbalance of tree rooted at target: 1
```

- Example 2: 8 element BST. The random seed results in the values entered in the following order: [3,4,6,7,2,5,0,1]. This test terminates when the imbalance of the tree rooted at the target node (value 4) exceeds input maximum of 1.

```
Input number of elements to put in BST: 8
Input specific target number to evaluate: 4
Input maximum height of target tree to allow: 2
Input maximum imbalance of target tree to allow: 1
Test terminated after 4 nodes entered into tree.
Elapsed time:  0.0005429999999999879 seconds
Target Value: 4
Height of tree rooted at target: 2
Size of tree rooted at target: 3
Imbalance of tree rooted at target: 2
```

- Example 3: 10 element BST. The random seed results in the values entered in the following order: [7,3,2,8,5,6,9,4,0,1]. This test uses the input target value of -1 to request that the root of the tree (in this case the value 7) be used as the target. This test terminates when the height of the tree rooted at the target node exceeds the input maximum of 2.

```
Input number of elements to put in BST: 10
Input specific target number to evaluate: -1
Input maximum height of target tree to allow: 2
Input maximum imbalance of target tree to allow: 2
Test terminated after 6 nodes entered into tree.
Elapsed time:  0.0006260000000000154 seconds
Target Value: 7
Height of tree rooted at target: 3
Size of tree rooted at target: 6
Imbalance of tree rooted at target: 2
```

- Example 4: 100,000 element BST. At this level, you will notice if your algorithms are running efficiently. If it takes on the order of 10 seconds to complete this task, you are in good shape. If it takes on the order of 500 seconds to complete this task, Eyecu will not be satisfied.

```
Input number of elements to put in BST: 100000
```

```
Input specific target number to evaluate: 25302
Input maximum height of target tree to allow: 30
Input maximum imbalance of target tree to allow: 20
Test terminated after 100000 nodes entered into tree.
Elapsed time:  8.680177 seconds
Target Value: 25302
Height of tree rooted at target: 27
Size of tree rooted at target: 8263
Imbalance of tree rooted at target: 0
```

7 Submission

Submit your `eyecuFunc.py` file to the MyCourses dropbox before the deadline. You do not need to submit any other files.

8 Grading

- 35% `insert` correctly builds a binary search tree.
- 15% `treeHeight` correctly computes the height of the tree.
- 15% `treeSize` correctly computes the size of the tree.
- 15% `treeImbalance` correctly computes the imbalance of the tree.
- 15% the functionality meets Eyecu requirements for speed.
- 5% for correct style, with docstrings for each function.