Assignment 4

Problem 1 - Height Balance Verification in a Tree

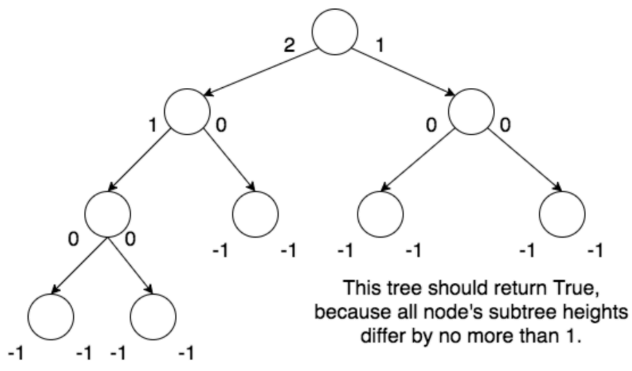
For this problem, we will define tree height and height balance as follows:

- An empty tree has height 0
- The height of any node N_i is computed with max(height of left branch of N_i, height of right branch of N_i) + 1.
- As an example, by this definition, a leaf node has height 1
- We consider a tree T as height balanced, if for every node N_i within T:

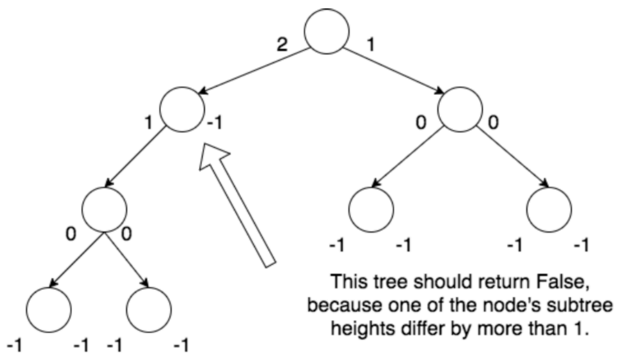
$$| \text{height of left branch of } N_i - \text{height of right branch of } N_i | \leq 1$$

In class `Tree` of file [`trees.py`](trees.py), implement method `is_height_balanced(self)`

It will return `True` if called on an instance of `Tree` that is height balanced, `False` otherwise.



This tree should return True, because all node's subtree heights differ by no more than 1.



This tree should return False, because one of the node's subtree heights differ by more than 1.

Examples:

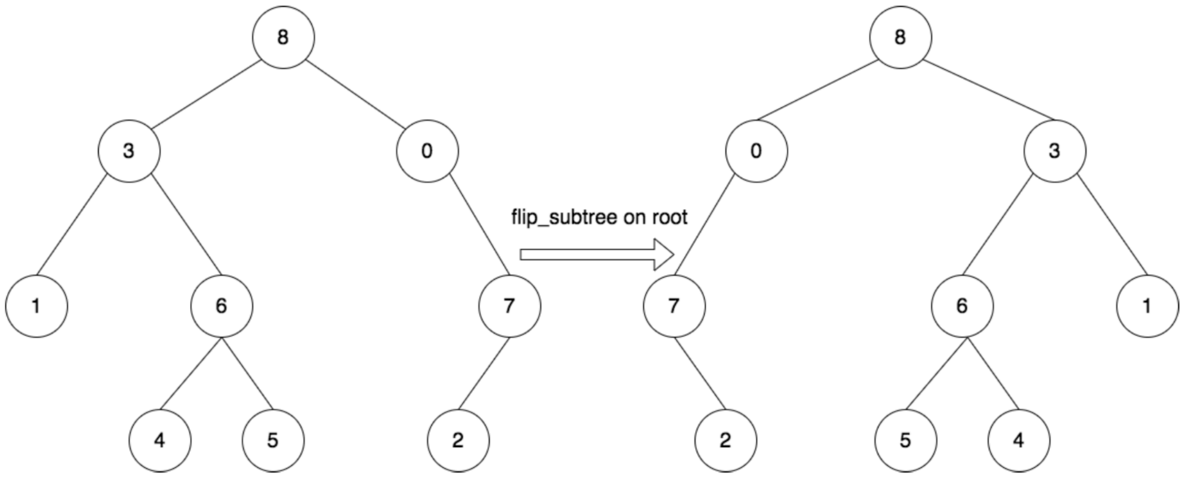
```
➢ In [0]: T1.is_height_balanced() # Suppose T1 is the left tree above
➢ Out[0]: True
➢
➢ In [1]: T2.is_height_balanced() # Suppose T2 is the right tree above
➢ Out[1]: False
```

Problem 2 - Flip a Tree

In class `Tree` of file trees.py, implement method `flip_tree(self, node=None)`

This method flips the left child with the right child of every node in the subtree of the node given as an argument. If called without any argument, the method flips the entire tree.

```
Example function call:
>>> T.flip_tree()
```



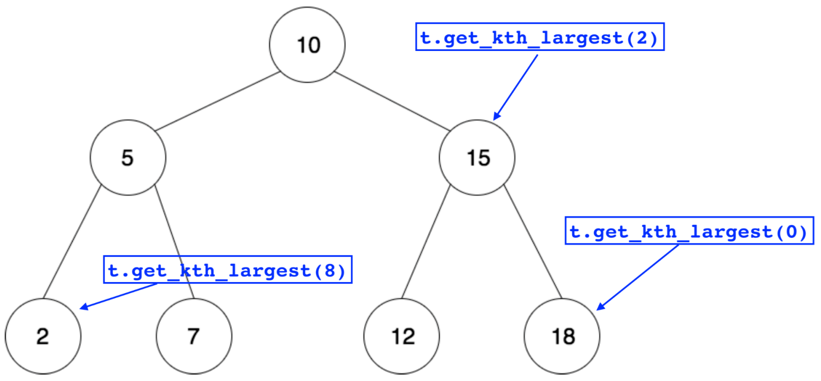
Important: you can declare additional methods in class `Tree` to solve this problem.

Problem 3 - Get the k^th Largest Node in a Binary Search Tree

In class `BinarySearchTree` of file trees.py, implement method `get_kth_largest(self, k)`

It returns the `k`-th largest node of the calling `BinarySearchTree` object. It returns the node with the smallest value if `k` is larger than the size of the tree; the node with the largest value if `k` is 0 or less.

Examples:



Important: (1) your function must return an instance of the `TreeNode` class; (2) you can use any method from class `BinarySearchTree` or from class `Tree`.

Problem 4 - Cuckoo Hashing

The cuckoo is a bird that does not make its own nest; rather it finds a nest of another bird, kicks that bird's eggs out, and places its own eggs there. *Cuckoo hashing* is just as ruthless, but it comes with a nice result: the runtime complexity of searches is constant! You may read more about this technique here and here.

Your task is to implement a cuckoo hashing table in file `cuckoo_hashing.py`

Implement the following functions:

```
def __getitem__(self, key) -> object:
def __setitem__(self, k, v) -> None:
def __delitem__(self, k) -> None:
def __len__(self) -> int:
def __contains__(self, key) -> boolean:
```

Notes:

- *Hash functions _hash1 and _hash2 are provided.*
- *You may define as many new functions as you like.*
- *Make sure you pass the test code.*
- *This is a Map ADT; it should behave like a dictionary.*
- *Implementing another strategy than cuckoo hashing may pass some (maybe all) tests. Be advised that we'll verify your implementation by hand.*

Submission format

The files below constitute your coding canvas for this assignment.

1. `trees.py`
2. `cuckoo_hashing.py`

You need to complete them, and then submit them directly (do not put them in a directory or zip) to gradescope:

https://www.gradescope.com/courses/399287/assignments/2080897