CSCI-SHU 210 Data Structures

Recitation 8 Binary trees

In this recitation, we focus on implementation of the binary tree.
You should keep in mind that, in this recitation, we are learning a Binary Tree, not a Binary Search Tree.

1. Getting familiar with 500 lines of code.

   You will be given 4 files:

   - `linked_queue.py`: The queue ADT implemented with linked list. This file is included because we need a queue to perform level order traversal.
   - `tree.py`: Abstract base class for BinaryTree
     - depth(self) # Task 4
     - height(self) # Task 1
     - __iter__(self)

   - `binary_tree.py`: Abstract base class for LinkedBinaryTree
     - sibling(self, p)
     - children(self, p)
     - inorder(self)
     - levelorderPrint(self, p) # Task 5
     - inorderPrint(self, p) # Task 5
     - preorderPrint(self,p) # Task 5
     - postorderPrint(self,p) # Task 5
   - `linked_binary_tree.py`:
     - class Position
     - class _Node
     - _validate(self, p)
     - _make_position(self, node)
     - root(self)
     - parent(self, p)
     - left(self, p)
     - right(self, p)
     - num_children(self, p)
     - add_root(self, e)
     - add_left(self, p, e)
     - add_right(self, p, e)
     - replace(self, p, e)
     - delete(self, p)
     - attach(self, p, t1, t2)
     - flip(self, p) # Task 6
     - flip_subtree(self, p) # Task 7
     - return_max(self, p) # Task 8
     - pretty_print(self)

Your task 1: Implement height(self) in `tree.py`. The height function is required to display the tree. Your function should be recursive.

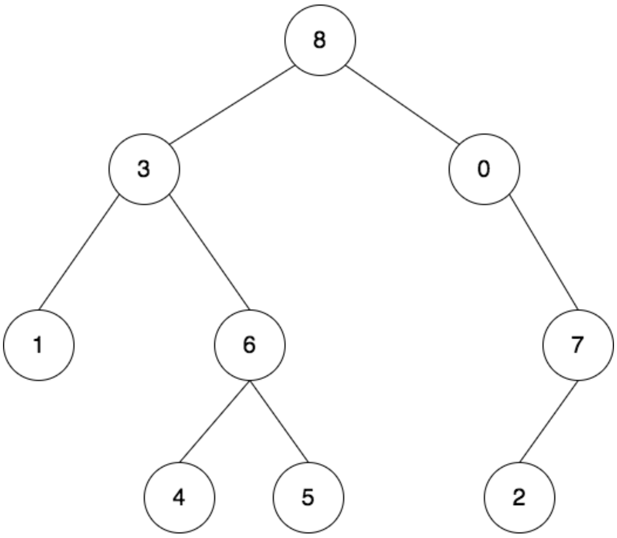
Your task 2: Take some time to look at those .py files. Run `linked_binary_tree.py`. You should see a binary tree being displayed.

Your task 3: Write some code in `linked_binary_tree.py`

```
#################### Task 3 draw a tree ###################
# Code here!
```

Your code should construct this tree:

then display it by calling LinkedBinaryTree.pretty_print(self)



2. Implementing depth function

Your task 4: Implement depth(self) in `tree.py`. Recall depth:

"""Return the number of levels separating Position p from the root."""

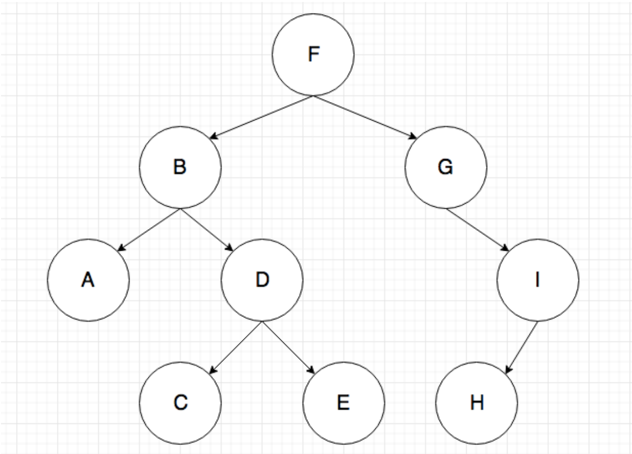
Your function should be recursive.

3. Tree traversals

Your task 5: Implement the following tree traversal in `binary_tree.py`:

```
preorderPrint(self,p)
postorderPrint(self,p)
inorderPrint(self,p)
levelorderPrint(self,p)
```

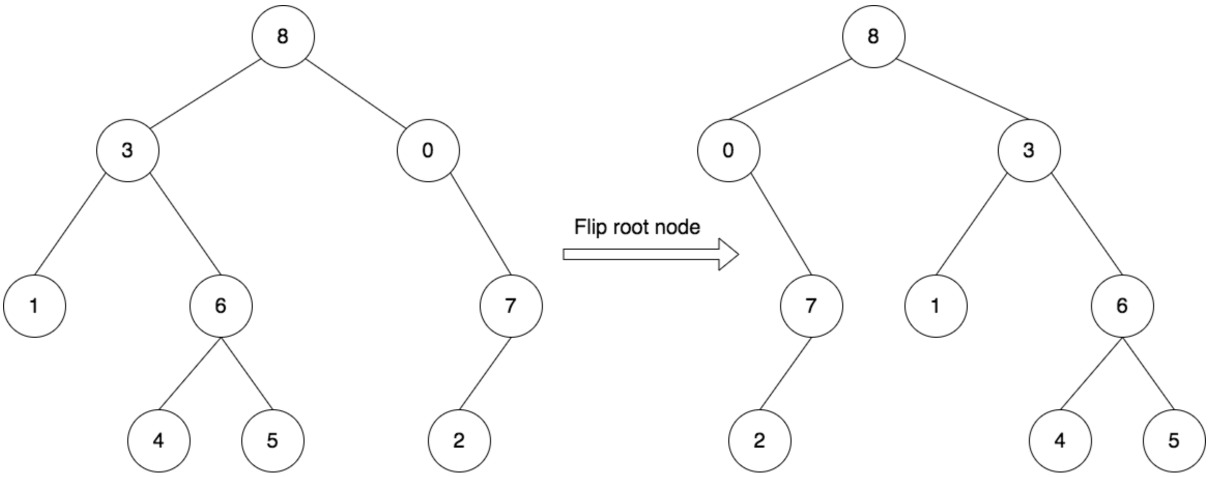
To get yourself started, take a look at algorithms and desired outputs:




```
VisitNode, do something (printing in our case)
RecursionLeft
RecursionRight                 Preorder result: F, B, A, D, C, E, G, I, H

RecursionLeft
RecursionRight
VisitNode, do something        Postorder result: A, C, E, D, B, H, I, G, F

RecursionLeft
VisitNode, do something
RecursionRight                 Inorder result: A, B, C, D, E, F, G, H, I

Enqueue root
While queue is not empty:
        deque front, do something
        Enqueue front's left child
        Enqueue front's right child
                              Levelorder Result: F, B, G, A, D, I, C, E, H
```

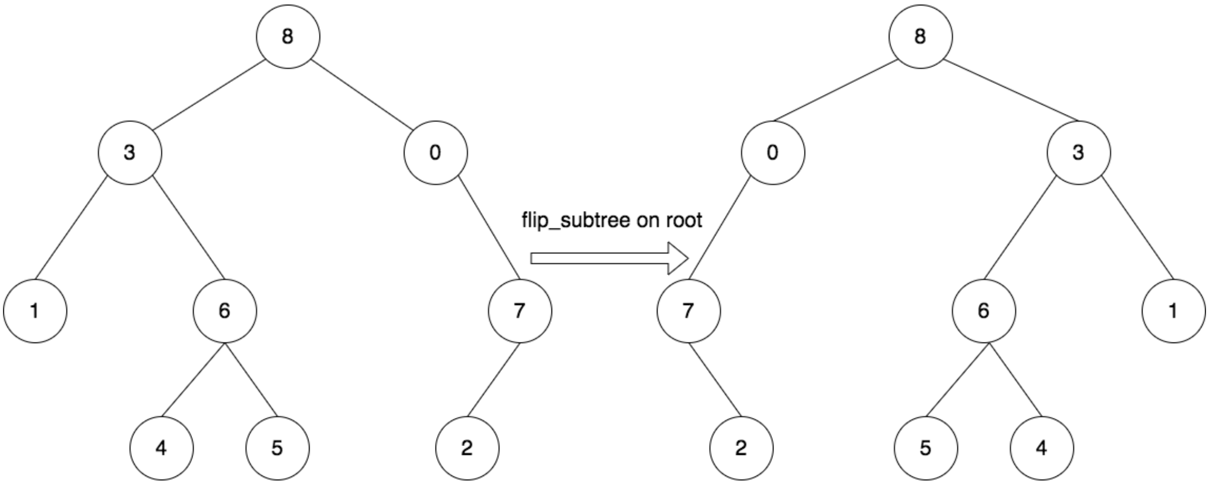
4. Flip a node, Flip tree

Your task 6: Implement method flip(self,p) in `linked_binary_tree.py.` which flips the left and right children of position p.



Your task 7: Implement method flip_subtree(self,p=None) in `linked_binary_tree.py` which flips the left and right children all nodes in the subtree of p, and if p is omitted it flips the entire tree.

Your method must be recursive.



5. Finding the maximum value in a Tree.

Your task 8: Implement method return_max(self) in `linked_binary_tree.py.` Traverse the tree and return the maximum value stored within the tree.

6. Finding the maximum value in a PositionalList. (If we have time)

```
Your task 9: Implement method return_max(self) in PositionalList.py.
Traverse the positional list and return the maximum value stored within
the list.
```

7. Sorting the PositialList using insertion sort. (If we have time)

```
Your task 10: Implement method insertion_sort(L) in PositionalList.py.
This sorting function sorts PositionalList of comparable elements into non
decreasing order.
```