NYU, Tandon School of Engineering

CS-UY 1134 Data Structures and Algorithms — Fall 2017

Homework #7
Due by Friday 11/10, 11:55pm

Submission instructions:

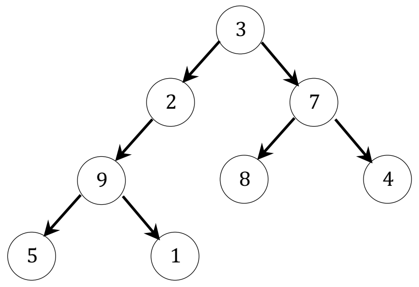
1. You should submit your homework in the NYU Classes system.
2. For this assignment, you should turn in one file containing the definition of the `LinkedBinaryTree` class, including all the additional methods implemented in this assignment. Also include the functions you wrote for questions 1, 3 and 5. Name this file 'YourNetID_hw7.py'.

Question 1:

Define the following function:

```
def min_and_max(bin_tree)
```

When called on a `LinkedBinaryTree`, containing numerical data in all its nodes, it will **return a tuple**, containing the maximum and minimum values in the tree.
For example, given the following tree:



Calling `min_and_max` on the tree above, should return (1, 9).

Implementation requirements:

1. Define one additional, **recursive**, helper function:
   ```
   def subtree_min_and_max(bin_tree, subtree_root)
   ```
   That is given `bin_tree`, a `LinkedBinaryTree` object, and `subtree_root`, a reference to a node in `bin_tree`. When called, it should return the minimum and maximum tuple for the subtree rooted by `subtree_root`.
2. In your implementations, you are not allowed to use any method from the `LinkedBinaryTree` class. Specifically, you are not allowed to iterate over the tree, using any of the traversals.
3. Your function should run in **linear time**.
4. Since the maximum and minimum are not defined on an empty set of elements, if the function is called on an empty tree you should raise an exception.

Question 2:

Add the following method to the `LinkedBinaryTree` class:

```
def leaves_list(self)
```

When called on a tree, it will create and return a list, containing the values stored at the leaves of the tree, ordered from left to right.
For example, if called on the tree from question 1, it should return: `[5, 1, 8, 4]`

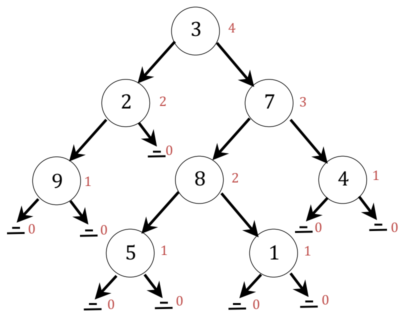
Implementation requirement:

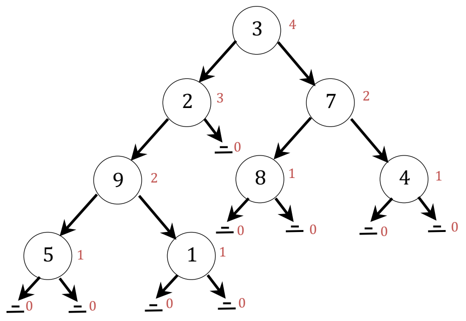
1. You may want to define an additional, **recursive**, helper function.
2. In your implementations, you are not allowed to use any method from the `LinkedBinaryTree` class. Specifically, you are not allowed to iterate over the tree, using any of the traversals.
3. Your method should run in **linear time**.

Question 3:
Although we originally defined the height of a subtree rooted at *r* to be the number of *edges* on the longest path from *r* to a leaf, for the simplicity of the definitions in this question, we will modify this definition so the height of a subtree rooted at *r* will be the number of ***nodes*** on such a longest path.
By this definition, a leaf node has height 1, while we trivially define the height of a "*None*" child to be 0.

We give the following definition, for what is considered to be a balanced tree:
We say that a binary tree *T* satisfies the ***Height-Balance Property*** if for every node *p* of *T*, the heights of the children of *p* differ by at most 1.

For example, consider the following two trees. Note that in these figures we showed the height of each subtree to the right of each such root, in a (small) red font:





The tree on the left satisfies the height-balance property, while the tree on the right does not (since the subtree rooted by the node containing 2 has one child with height 2 and the second child with height 0).

Implement the following function:
```
    def is_height_balanced(bin_tree)
```

Given `bin_tree`, a `LinkedBinaryTree` object, it will return `True` if the tree satisfies the height-balance property, or `False` otherwise.

Implementation requirement: Your function should run in **linear time**.

Hint: To meet the runtime requirement, you may want to define an additional, recursive, helper function, that returns more than one value (multiple return values could be collected as a tuple).

Question 4:

Add the following method to the `LinkedBinaryTree` class:

```
def iterative_inorder(self)
```

When called on a tree, it will create a generator, allowing to iterate over the values of the tree in an in-order order.
For example, if `t` is the tree from question 1, when running the following code:

```
for item in t.iterative_inorder():
     print(item, end=' ')
print()
```

You should expect the following output:

```
5  9  1  2  3  8  7  4
```

Implementation requirements:

1. You should make a purely iterative implementation. **Recursion is not allowed**.
2. Your method is not allowed to call any helper methods or functions. **All the work should be done inside this method**.
3. In addition to the tree, you may use only $\boldsymbol{\theta(1)}$ **additional memory**. That is, you are not allowed to use any additional data structure (such as stack, list, etc.).
4. The total runtime to iterate over an entire tree should be **linear**.

Note: When you will be done with this question, you will truly appreciate the simplicity and elegancy of recursion ☺

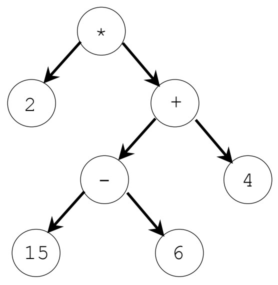
Question 5:

a. Implement the following function:

```
def create_expression_tree(prefix_exp_str)
```

The function is given a string `prefix_exp_str`, which contains an arithmetic expression in a prefix notation.
When called, it creates and returns a `LinkedBinaryTree` object, that is the expression tree representing `prefix_exp_str`.

For example, the call: `create_expression_tree('* 2 + - 15 6 4')` will create and return the following tree:



Note:
Assume that prefix expression will come in the following format:

1. All operators in the expression will be taken from the four basic arithmetic operations (+, -, * and /).
2. All operands will be positive integers.
3. The tokens in the expression will be separated by a space.

Implementation requirements:

1. The nodes containing operators, should have the operator as a `string`, and nodes containing numerals, should have the number as an `int`.
2. The runtime for creating the expression tree should be **linear**.
3. You are **not allowed to use a stack** in your implementation.

Hint: There is more than one way to solve this problem. One way is to start by creating a list with the expression's tokens (by using the split method), and then calling a helper function:
`def create_expression_tree_helper(prefix_exp, start_pos)`
This helper function would be recursive, and in order to efficiently pass the subexpression that the call should consider, besides the list with the entire expression, it also gets `start_pos`, which is the index where the subexpression starts (we don't know how long the subexpression will be, so we can't pass the ending position too).
The call to the helper function would return two values (as a tuple): the first would be the subtree it created (that represent the subexpression starting at `start_pos`), and the second would be the size of that subtree (so you can know where the next subexpression would start…).

b. Use the function you implemented in section (a), to implement the following function:

```
def prefix_to_postfix(prefix_exp_str)
```

The function is given a string `prefix_exp_str`, which contains an arithmetic expression in a prefix notation (in the format described in section (a)).
When called, it returns the string representation of the postfix equivalent of `prefix_exp_str`.

For example, the call: `prefix_to_postfix('* 2 + - 15 6 4')`, will return: `'2 15 6 - 4 + *'`

Implementation requirement: The runtime of this function should be **linear**.