Homework #4
Due by Wednesday 08/11/2021, 11:55pm

Submission instructions:

1. You should write 5 '.py' files: one for each question.
   Name your files: 'YourNetID_hw4_q1.py', 'YourNetID_hw4_q2.py', etc.
   Note: your netID follows an abc123 pattern, not N12345678.

2. In this assignment, we provided 'DoublyLinkedList.py' file (with the implementation of a doubly linked list).
   In questions where you need to use `DoublyLinkedList`, make the definition in a separate file, and use an **`import`** statement to import the `DoublyLinkedList` class.
   You should use the linked lists as a black box. **You are not allowed to put any changes in 'DoublyLinkedList.py'.** Such a change is considered a break of an abstraction barrier.

3. In this assignment, we provided 'LinkedBinaryTree.py' file (with the implementation of a binary tree).

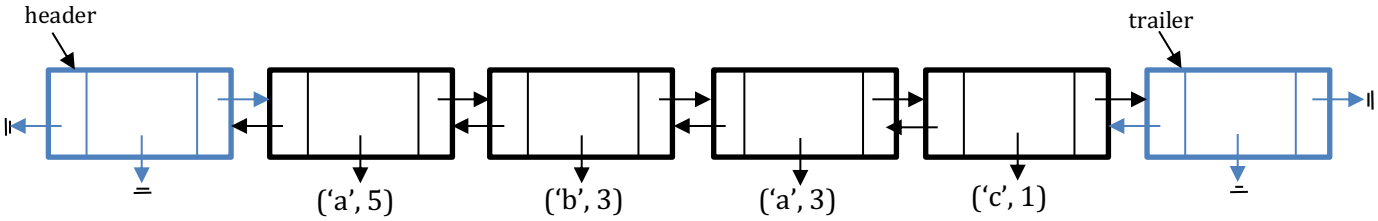
4. You should submit your homework via Gradescope. For Gradescope's autograding feature to work:
   a. Name all functions and methods exactly as they are in the assignment specifications.
   b. Make sure there are no print statements in your code. If you have tester code, please put it in a "main" function and do not call it.
   c. You don't need to submit the 'DoublyLinkedList.py' file

Question 1:

In this question, we will suggest a data structure for storing strings with a lot of repetitions of successive characters.

We will represent such strings as a linked list, where each maximal sequence of the same character in consecutive positions, will be stored as a single tuple containing the character and its count.

For example, the string "aaaaabbbaaac" will be represented as the following list:



Complete the definition of the following `CompactString` class:

```
class CompactString:
    def __init__(self, orig_str):
        ''' Initializes a CompactString object
        representing the string given in orig_str'''

    def __add__(self, other):
        ''' Creates and returns a CompactString object that
        represent the concatenation of self and other,
        also of type CompactString'''

    def __lt__(self, other):
        ''' returns True if"f self is lexicographically
        less than other, also of type CompactString'''

    def __le__(self, other):
        ''' returns True if"f self is lexicographically
        less than or equal to other, also of type
        CompactString'''

    def __gt__(self, other):
        ''' returns True if"f self is lexicographically
        greater than other, also of type CompactString'''

    def __ge__(self, other):
        ''' returns True if"f self is lexicographically
        greater than or equal to other, also of type
        CompactString'''

    def __repr__(self):
        ''' Creates and returns the string representation
        (of type str) of self'''
```

For example, after implementing the `CompactString` class, you should expect the following behavior:

```
>>> s1 = CompactString('aaaaabbbaaac')
>>> s2 = CompactString('aaaaaaacccaaaa')
>>> s3 = s2 + s1 #in s3's linked list there will be 6 'real' nodes
>>> s1 < s2
False
```

Note: Here too, when adding and comparing two `CompactString` objects, DO NOT convert the `CompactString` objects to `str`s, do the operation on `str`s (by using Python +, <, >, <=, >= operators), and then convert the result back to a `CompactString` object. This approach misses the point of this question.

Question 2:

In this question, we will demonstrate the difference between shallow and deep copy. For that, we will work with *nested doubly linked lists of integers*. That is, each element is an integer or a `DoublyLinkedList`, which in turn can contain integers or `DoublyLinkedList`s, and so on.

a. Implement the following function:
   `def` `copy_linked_list(lnk_lst)`
   The function is given a nested doubly linked lists of integers `lnk_lst`, and returns a **shallow copy** of `lnk_lst`. That is, a new linked list where its elements reference the same items in `lnk_lst`.

   For example, after implementing `copy_linked_list`, you should expect the following behavior:

```
>>> lnk_lst1 = DoublyLinkedList()
>>> elem1 = DoublyLinkedList()
>>> elem1.add_last(1)
>>> elem1.add_last(2)
>>> lnk_lst1.add_last(elem1)
>>> elem2 = 3
>>> lnk_lst1.add_last(elem2)

>>> lnk_lst2 = copy_linked_list(lnk_lst1)

>>> e1 = lnk_lst1.header.next
>>> e1_1 = e1.data.header.next
>>> e1_1.data = 10

>>> e2 = lnk_lst2.header.next
>>> e2_1 = e2.data.header.next
>>> print(e2_1.data)
10
```

b. Now, implement:
def `deep_copy_linked_list(lnk_lst)`
The function is given a nested doubly linked lists of integers `lnk_lst`, and returns a **deep copy** of `lnk_lst`.

For example, after implementing `deep_copy_linked_list`, you should expect the following behavior:

```
>>> lnk_lst1 = DoublyLinkedList()
>>> elem1 = DoublyLinkedList()
>>> elem1.add_last(1)
>>> elem1.add_last(2)
>>> lnk_lst1.add_last(elem1)
>>> elem2 = 3
>>> lnk_lst1.add_last(elem2)

>>> lnk_lst2 = deep_copy_linked_list(lnk_lst1)

>>> e1 = lnk_lst1.header.next
>>> e1_1 = e1.data.header.next
>>> e1_1.data = 10

>>> e2 = lnk_lst2.header.next
>>> e2_1 = e2.data.header.next
>>> print(e2_1.data)
1
```

Note: `lnk_lst` could have **multiple levels** of nesting.

Question 3:

In this question, we will implement a function that merges two sorted linked lists:
`def merge_linked_lists(srt_lnk_lst1, srt_lnk_lst2)`

This function is given two doubly linked lists of integers `srt_lnk_lst1` and `srt_lnk_lst2`. The elements in `srt_lnk_lst1` and `srt_lnk_lst2` are sorted. That is, they are ordered in the lists, in an ascending order.
When the function is called, it will **create and return a new** doubly linked list, that contains all the elements that appear in the input lists in a sorted order.

For example:
if `srt_lnk_lst1`= [1 <--> 3 <--> 5 <--> 6 <--> 8],
and `srt_lnk_lst2`=[2 <--> 3 <--> 5 <--> 10 <--> 15 <--> 18],
calling: `merge_linked_lists(srt_lnk_lst1, srt_lnk_lst2)`, should create and return a doubly linked list that contains:
[1 <--> 2 <--> 3 <--> 3 <--> 5 <--> 5 <--> 6 <--> 8 <--> 10 <--> 15 <--> 18].

The `merge_linked_lists` function is not recursive, but it defines and calls `merge_sublists` - a nested helper **recursive** function.

Complete the implementation given below for the `merge_linked_lists` function:

```
def merge_linked_lists(srt_lnk_lst1, srt_lnk_lst2):
    def merge_sublists( _____ ):
        _____
        _____
        _____
        _____
        _____

    return merge_sublists( _____ )
```

Notes:

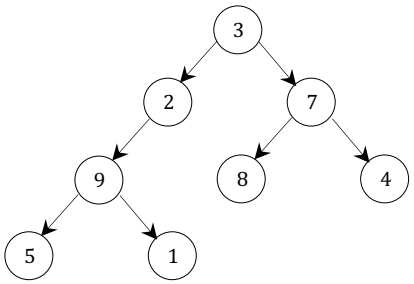
1. You need to decide on the signature of `merge_sublists`.
2. `merge_sublists` has to be **recursive**.
3. An efficient implementation of `merge_sublists` would allow `merge_linked_lists` to run in **linear time.** That is, if n_1 and n_2 are the sizes of the input lists, the runtime would be $\theta(n_1 + n_2)$.

Question 4:

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(root)
   ```

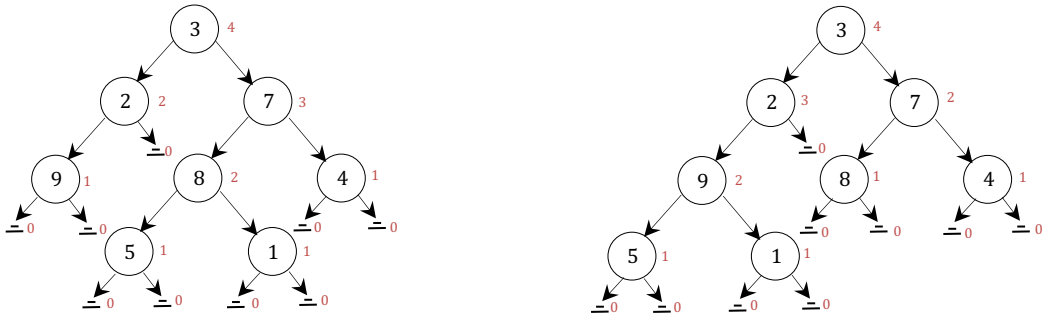
   That is given `root`, a reference to a node in a `LinkedBinaryTree` tree. When called, it should return the minimum and maximum tuple for the subtree rooted by `root`.
2. In your implementations, you are not allowed to use any method from the `LinkedBinaryTree` class. Specifically, you are not allowed 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 5:

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).