Homework #4

Due by Tuesday 8/04, 11:55 pm

Submission instructions:

1. For this assignment, you should turn in 4 files:
    - 5 '.py' files, one for each question 1-5. Name your files: YourNetID_hw3_q2.py' and 'YourNetID_hw3_q3.py', etc.

      Note: your netID follows an abc123 pattern, not N12345678.

2. You should submit your homework via Gradescope.
    - Name all classes, functions, and methods exactly as they are in the assignment specifications.
    - Make sure there are no print statements in your code. If you have a tester code, please put it in a "main" function and do not call it.

Question 1:

Define a `LinkedQueue` class that implements the *Queue* ADT.

Implementation Requirement: All queue operations should run in $\theta(1)$ **worst- case**.
Hint: You would want to use a doubly linked list as a data member.

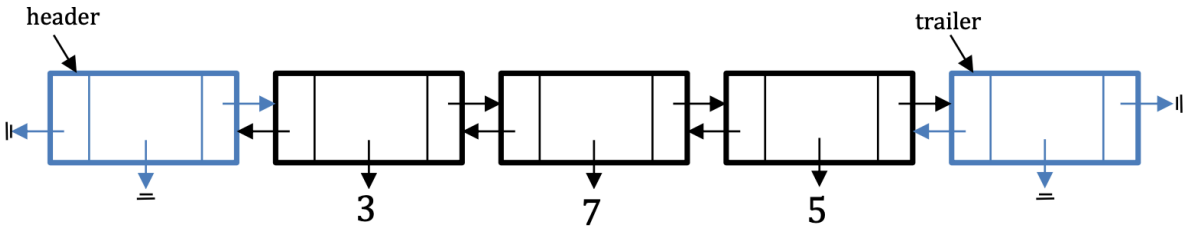
Question 2:

Many programming languages represent integers in a **fixed** number of bytes (a common size for an integer is 4 bytes). This, on one hand, bounds the range of integers that can be represented as an `int` data (in 4 bytes, only 2^{32} different values could be represented), but, on the other hand, it allows fast execution for basic arithmetic expressions (such as `+`, `-`, `*` and `/`) typically done in hardware.

Python and some other programming languages, do not follow that kind of representation for integers, and allows to represent arbitrary large integers as `int`

variables (as a result the performance of basic arithmetic is slower).

In this question, we will suggest a data structure for positive integer numbers, that can be arbitrary large.
We will represent an integer value, as a linked list of its digits.
For example, the number 375 will be represented by a 3-length list, with 3, 7 and 5 as its elements.



Note: this is not the representation Python uses. Complete the definition of the following `Integer` class:

```
class Integer:

    def __init__(self, num_str):

        ''' Initializes an Integer object representing the
        value given in the string num_str'''

    def __add__(self, other):

        ''' Creates and returns an Integer object that
        represent the sum of self and other, also of type
        Integer'''

    def __repr__(self):

        ''' Creates and returns the string representation of
        self'''
```

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

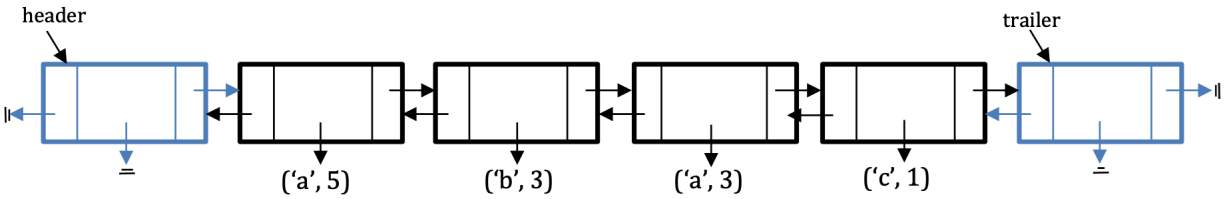
```
>>> n1 = Integer('375')

>>> n2 = Integer('4029')

>>> n3 = n1 + n2

>>> n3

4404
```

Note: When adding two `Integer` objects, implement the "Elementary School" addition technique. DO NOT convert the `Integer` objects to `int`s, add these `int`s by using Python `+` operator, and then convert the result back to an `Integer` object. This approach misses the point of this question.

Question 3:

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 = s1 + s2 #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 4:

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 5:

Define a function that when given a singly linked list find out if the linked list contains a loop and returns True if it does and False if it doesn't:

```
def findLoop(linkedList)
```

Notes:

1. Your function should run in O(n) time
2. You are only allowed to use O(1) space.