NYU, Tandon School of Engineering

CS-UY 1134 Data Structures and Algorithms — Fall 2017

# Homework #6
# Due by Friday 11/3, 11:55pm

**Submission instructions:**

1. You should submit your homework in the NYU Classes system.
2. For this assignment you should turn in 5 files. One '.py' file for each question. Name your files 'YourNetID_hw6_q1.py', 'YourNetID_hw6_q2.py', 'YourNetID_hw6_q3.py', etc.

## 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:

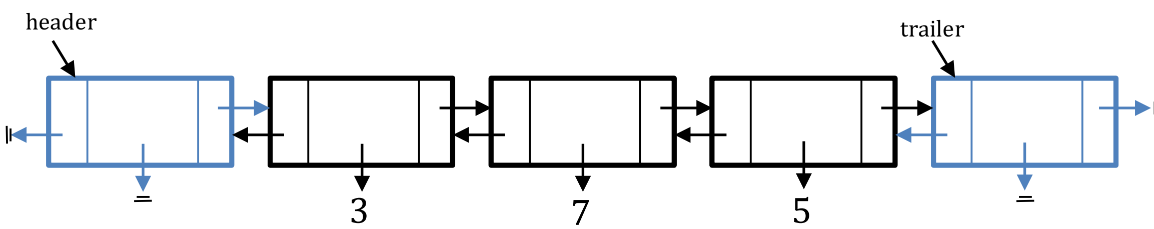
Many programming languages represent integers in a **fixed** number of bytes (a common size for an integer is 4 bytes). This, on one hand, bounds the range of integers that can be represented as an `int` data (in 4 bytes, only $2^{32}$ different values could be represented), but, on the other hand, it 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 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 __str__(self):
        ''' Creates and returns the string representation
         of self'''

    def __repr__(self):
        return str(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.

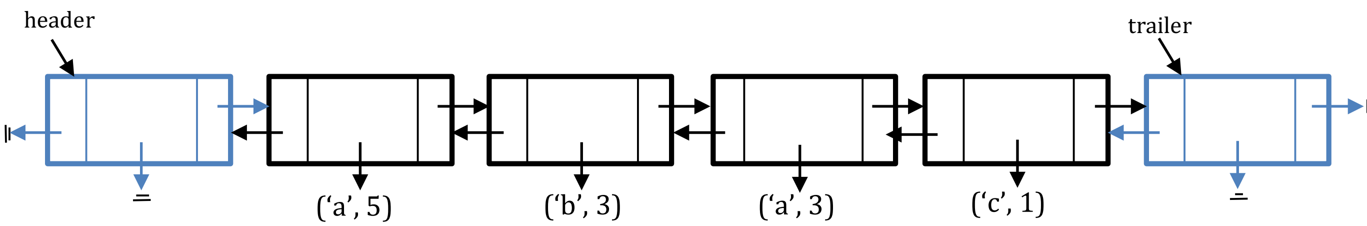
**Extra Credit:**

Support also the multiplication of two `Integer` objects (by implementing the "Elementary School" multiplication technique):

```
    def __mul__(self, other):
        ''' Creates and returns an Integer object that
        represent the multiplication of self and other,
        also of type Integer'''
```

**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 __str__(self):
        ''' Creates and returns the string representation
        (of type str) of self'''

    def __repr__(self):
        return str(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 4:

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.first_node()
>>> e1_1 = e1.data.first_node()
>>> e1_1.data = 10

>>> e2 = lnk_lst2.first_node()
>>> e2_1 = e2.data.first_node()
>>> 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.first_node()
>>> e1_1 = e1.data.first_node()
>>> e1_1.data = 10

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

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

**Question 5:**

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 should call a helper function, `merge_sublists`, which **is recursive**.
Complete the implementation given below for the `merge_linked_lists` function, and define the `merge_sublists` function:

```
def merge_linked_lists(srt_lnk_lst1, srt_lnk_lst2):
    return merge_sublists( ______________________________________ )
```

**Notes:**

1. You need to decide on the signature of `merge_sublists`.
2. 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 $\theta(n_1 + n_2)$.