CS-UY 1134: Data Structures and Algorithms
Lab 7: Linked Lists, Plus More Fun Stacks and Queues

Friday, July 12, 2019

Instructions

This lab will focus on linked list data structures, as well as some more practice with stacks and queues.

Try to solve all of the problems by the end of the lab time. As added practice, it is a good idea to try writing your code with pen and paper first, as you will have to do this on the exam.

When you type up your code, write some test cases. Try multiple test cases for each function you have to implement, and consider edge cases carefully.

As a reminder, you *may* (and are encouraged to!) discuss these problems with your classmates during lab and help each other solve them. However, make sure the code you submit is your own.

What to submit

If you believe you have solved all the problems, let a TA or the professor know and they will check your work. This is a good chance to get feedback on problems similar to what you might see on the exam.

If you don't finish by the end of the lab, you may submit something on NYU Classes by Sunday July 14, 11:55 PM.

Either when a TA has checked your solutions, or by the Sunday submission deadline, submit your solutions to the "Lab 7: Linked Lists" assignment on NYU Classes.

1 LinkedStack

Implement a `class LinkedStack` which should provide the Stack ADT implemented uring a linked list. Use the `DoublyLinkedList` class from lecture (you can get the code on NYU Classes). Your `LinkedStack` should include an instance of this `DoublyLinkedList` class as a data member, and should **not** have any dynamic array as a member. Your implementation should provide all of the Stack operations in *worst case* $O(1)$ running time. A skeleton of the class, showing all the functions you should implement, is provided below. You *must* implement all of these methods.

```
class LinkedStack:
    def __init__(self):
        '''Initialize an empty stack'''
        self.data = DoublyLinkedList()
        # You will likely need to add more to this constructor,
        # but at minimum you should have the linked list data member as above

    def __len__(self):
        '''Return the number of elements in the stack'''
```

```
    def is_empty(self):
        '''Returns True if the stack is empty'''

    def push(self, elem):
        '''Adds elem to the top of the stack'''

    def pop(self):
        '''Remove and return the element at the top of the stack
        or raise an Exception if the stack is empty'''

    def top(self):
        '''Returns (without removing) the element at the top of the stack
        or raise an Exception if the stack is empty'''
```

2 LeakyStack

A *Leaky Stack* is similar to a stack except that the Leaky Stack has a bound on the number of items that can be stored in it at once. During initialization, the maximum size N is given. If the Leaky Stack has N elements, it is considered full. When an element is added to a Leaky Stack that is full, the element at the *bottom* of the stack is removed to make room for the new element, which is then placed at top of the stack as usual.

For example, consider the following code:

```
leaky_s = LeakyStack(5)
leaky_s.push(17)
leaky_s.push(42)
leaky_s.push(6)
leaky_s.push(31)
leaky_s.push(28)
```

The Leaky Stack has a maximum size of 5 (specified when it is created with the constructor) so it is now full (because we have `push`ed 5 items onto it). If we now make a call to `leaky_s.push(2)`, the item at the bottom of the stack, 17, is removed to make space for the new item, 2. This is shown below, where we show the contents of the Leaky Stack both *before* and *after* the call to `leaky_s.push(2)`:

Before:

28
31
6
42
17

Before:

2
28
31
6
42

A class that implements the *Leaky Stack* ADT would need to provide the following methods:

```
def __init__(self, max_num_of_elems):
    '''Initialize an empty leaky stack'''

def __len__(self):
    '''Return the number of elements in the stack'''

def is_empty(self):
    '''Returns True if the stack is empty'''

def push(self, elem):
```

```
    '''Adds elem to the top of the stack'''

def pop(self):
    '''Remove and return the element at the top of the stack
    or raise an Exception if the stack is empty'''

def top(self):
    '''Returns (without removing) the element at the top of the stack
    or raise an Exception if the stack is empty'''
```

In this problem, you will provide *two* different implementations for the *Leaky Stack* ADT, using different data structures.

1. Implement a Leaky Stack using a dynamic array, in a class called `class ArrayLeakyStack`. All operations **must** run in time $O(1)$ in the worst case.

   Hint: To handle the "leaky" part of removing the bottom element when pushing to a full stack, think of using a similar "circular" approach as we did with queues.

2. Implement a Leaky Stack using a linked list, in a class called `class LinkedLeakyStack`. Your class should include a data member that is an instance of the `DoublyLinkedList` class we wrote in lecture (you can get the code from NYU Classes). All operations **must** run in time $O(1)$ in the worst case.

3 Stacks with Queues

In this problem, try to implement a stack using a queue. Write a class called `class QStack` that has implements the stack ADT. It should have an instance of the class `ArrayQueue` as a data member (use the `ArrayQueue` class we wrote in lecture — you can get the code from NYU Classes). You can only access and modify this ArrayQueue instance by using the methods that are defined as part of the ArrayQueue class (that is, the methods that make up the interface of the Queue ADT). Your `QStack` class may also use $O(1)$ additional space for other data members as necessary (this is a *constant* amount of extra space — so you may **not** include extra data structures or collections of non-constant size).

Recall that as the QStack will implement the Stack ADT, you must define all the operations of the Stack ADT. A skeleton of the class along with the methods you must implement is below:

```
class QStack:
    def __init__(self):
        '''Initialize an empty stack'''
        self.data_queue = ArrayQueue()
        # You may need to add to this constructor, such as adding more data members
        # But you should at minimum have the ArrayQueue data member as above

    def __len__(self):
        '''Return the number of elements in the stack'''

    def is_empty(self):
        '''Returns True if the stack is empty'''

    def push(self, elem):
        '''Adds elem to the top of the stack'''

    def pop(self):
        '''Remove and return the element at the top of the stack
```

```
        or raise an Exception if the stack is empty'''

    def top(self):
        '''Returns (without removing) the element at the top of the stack
        or raise an Exception if the stack is empty'''
```

Hint: You will *not* be able to achieve $O(1)$ worst-case running times for all operations. In fact, some operations may be very inefficient. That is *okay*: There is no running time requirement for this problem (however, see below).

Analyze the running time of the operations in your implementation, particularly `push` and `pop`. **In the code you submit, include the running time of each of your operations in a comment at the top of each method's implementation**. While there is no requirement for efficiency, you *must* analyze the running times and **include them in your submission** for full credit.

Also think about, but don't submit: Which of the operations, `push` or `pop`, is more efficient in your implementation? If your `pop` is more efficient, can you think of an alternate implementation where `push` would be more efficient (possibly at the expense of making `pop` less efficient)? If your `push` is more efficient, can you think of an alternate implementation where `pop` would be more efficient (at the expense of making `push` more expensive)?