CPE 202 Winter 2020

Lab Week 3: Queue implementations

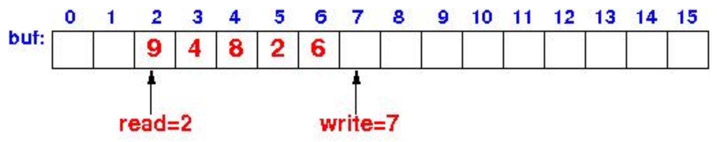
Goal:

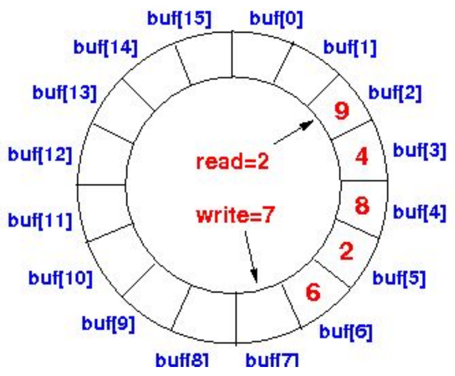
- **Implement a Queue class using a linked data structure: Class QueueLinked**
- **Implement a Queue class using a circular array: Class QueueArray**

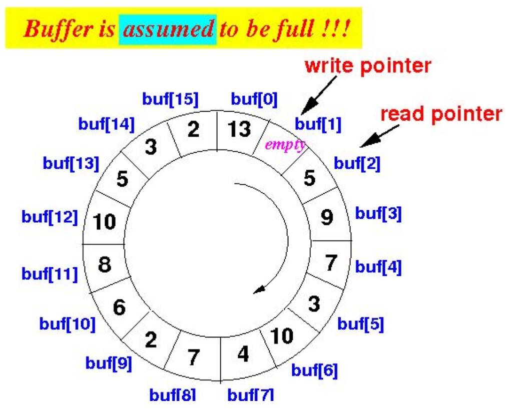
1. The first implementation you will use a linked structure similar to the linked structure used in implementing the Stack ADT. In this case, there must be a way to add items to the back of the list and remove items from the front of the list. Reuse the Node class in linked_list.py. You do not need to reuse functions in linked_list.py

2. The second implementation will use a circular array for storing the items in the queue. There are different ways of doing this but they share the idea presented in the picture. Namely elements are added to the rear of the queue using the next "free entry" in an array. (rear may be the index of the current last index storing an item in the queue or one beyond it. Elements are removed from the front (read in the picture) of the queue. Elements are added to the queue at the rear (write) of the queue. The indices in front and rear are incremented as elements are added and deleted from the queue. All arithmetic is **modulo the size of the array structure allocated to store the queue.** Care must be taken in distinguishing a full from an empty queue and this can be done in different ways. Use the following formula (**read pointer == (write pointer + 1) % (len(array)))**

Note that in this implementation the maximum number of items the queue can hold is one less than the size of the array-like structure allocated. When a user of your Queue class specifies its capacity, you may have to take this into account so that the queue can actually store capacity items. In the figure below, **buf** is an array.




buf[15] buf[0] buf[1] buf[2] buf[3] buf[4] buf[5] buf[6] buf[7] buf[8] buf[9] buf[10] buf[11] buf[12] buf[13] buf[14]
9 4 8 2 6
read=2
write=7


Buffer is assumed to be full !!!
write pointer
read pointer
empty
buf[15] buf[0] buf[1] buf[2] buf[3] buf[4] buf[5] buf[6] buf[7] buf[8] buf[9] buf[10] buf[11] buf[12] buf[13] buf[14]
13 5 9 7 3 10 4 7 2 6 8 10 5 5 3 2

You may choose to reuse array_list.py from Lab 2. In that case, you need to add an argument to __init__ of ArrayList class to pass the initial capacity, to make the arr be [None] * self.capacity. You also need to add a function update, which takes an object of ArrayList and an index, and updates the value in the arr at the index, and returns the ArrayList object, in array_list.py. You have to make sure that arr will not be resized: do not use insert nor pop functions.

Submit your queues.py to the grader and then submit the following three files to PolyLearn (zip them into one):

● **queues.py (Download this file from the polylearn)** containing an array based implementation of queue and a linked list implementation of queue. The classes must be called: **QueueArray** and **QueueLinked** . Both implementations should follow the specification and be thoroughly tested.

Raise an IndexError exception if a user of your implementation attempts to enqueue an item when the queue is full or dequeue and item with the queue is empty. A starter file is available on Polylearn.

● **test_queues.py** contains your set of tests to ensure you classes work correctly
● A file containing your linked list implementation:linked_list.py from Lab 2. Your queues.py need to import this file.
● Additionally, array_list.py if you choose to reuse it.

Make sure that each of your classes and functions (methods) has a docstring that describes its purpose and signature. Every class must have its own __eq__ and __repr__ methods implemented.