CSCI 204: Data Structures & Algorithms

Circular Queue

Revised based on textbook author's notes.

Queue

- A restricted access container that stores a linear collection.
  - Very common for solving problems in computer science that require data to be processed in the order in which it was received.
  - Provides a **first-in first-out** (FIFO) protocol.
- New items are added at the **back** while existing items are removed from the **front** of the queue.

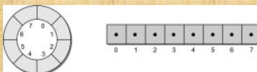
The Queue ADT

- A *queue* stores a linear collection of items with access limited to a first-in first-out order.
  - New items are added to the back.
  - Existing items are removed from the front.

- Queue()
- is_empty()
- len()
- enqueue(item)
- dequeue()

Queue: Circular Array

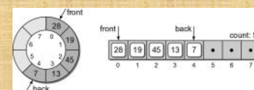
- **circular array** – an array viewed as a circle instead of a line.



- Items can be added/removed without having to shift the remaining items in the process.
- Introduces the concept of a maximum-capacity queue that can become full.

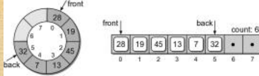
Queue: Circular Array

- How should the data be organized within the array?
  - **count field** – number of items in the queue.
  - **front and back markers** – indicate the array elements containing the queue items.



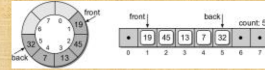
Queue: Circular Array

- To enqueue an item:
  - new item is inserted at the position following back
  - back is advanced by one position
  - count is incremented by one.
- Suppose we enqueue 32:



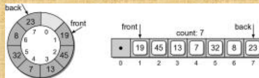
Queue: Circular Array

- To dequeue an item:
  - the value in the front position is saved
  - front is advanced by one position.
  - count is decremented by one.
- Suppose we dequeue an item:



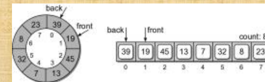
Queue: Circular Array

- Suppose we enqueue items 8 and 23:



Queue: Circular Array

- What happens if we enqueue 39?
  - Since we are using a circular array, the same steps are followed.
  - But since back is at the end of the array, it wraps around to the front.



Queue: Circular Array

arrayqueue.py

```
class Queue :
  def __init__( self, max_size ) :
    self._count = 0
    self._front = 0
    self._back = max_size - 1
    self._qarray = Array( max_size )

  def is_empty( self ) :
    return self._count == 0

   # A new operation specifically for the circular array.
  def is_full( self ) :
    return self._count == len(self._qarray)

  def __len__( self ) :
    return self._count

# ...
```


Queue: Circular Array

arrayqueue.py

```
class Queue :
# ...
  def enqueue( self, item ):
    assert not self.is_full(), "Cannot enqueue to a full queue."
    max_size = len(self._qarray)
    self._back = (self._back + 1) % max_size
    self._qarray[self._back] = item
    self._count += 1

  def dequeue( self ):
    assert not self.is_empty(), "Cannot dequeue from an empty queue."
    item = self._qarray[ self._front ]
    max_size = len(self._qarray)
    self._front = (self._front + 1) % max_size
    self._count -= 1
    return item
```

Queue Analysis: Circular Array

Queue Operation	Worst Case
q = Queue()	O(1)
len(q)	O(1)
q.is_empty()	O(1)
q.is_full()	O(1)
q.enqueue(x)	O(1)
x = q.dequeue()	O(1)

13

Your Exercise

- The circular queue we just implemented uses a `count` to control how the queue operates.
- Your exercise is to implement the same circular queue without the count variable.
- The basic idea is to use the relation between `front` and `back` to manage the queue.
- Note that without a count, one can't tell the difference between a full queue or empty queue if `front == back`, so the two have to be different when queue is empty or full.