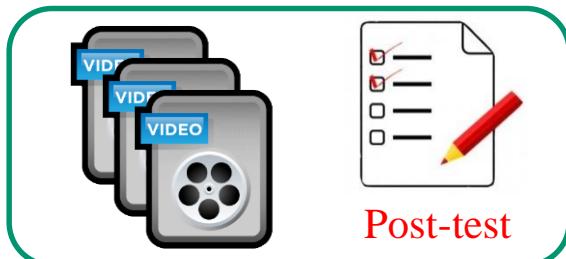




CMPUT 175

Introduction to Foundations of Computing

Queue, bounded Queue, Circular Queue



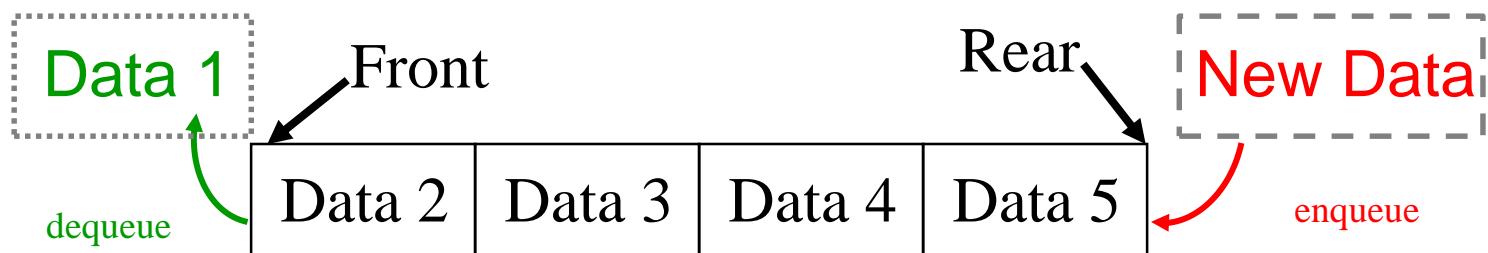
You should view the vignettes:
Queues

Objectives

- In this lecture we will learn about another linear structure called Queue.
- We will learn about how to implement the Queue data structure in python.
- We will discuss Other implementations such as **bounded Queues** and **circular Queues**.

Queues

- Ordered collection where items are added at one end (called **rear** or **tail**) and removed at the other end (called **front** or **head**)



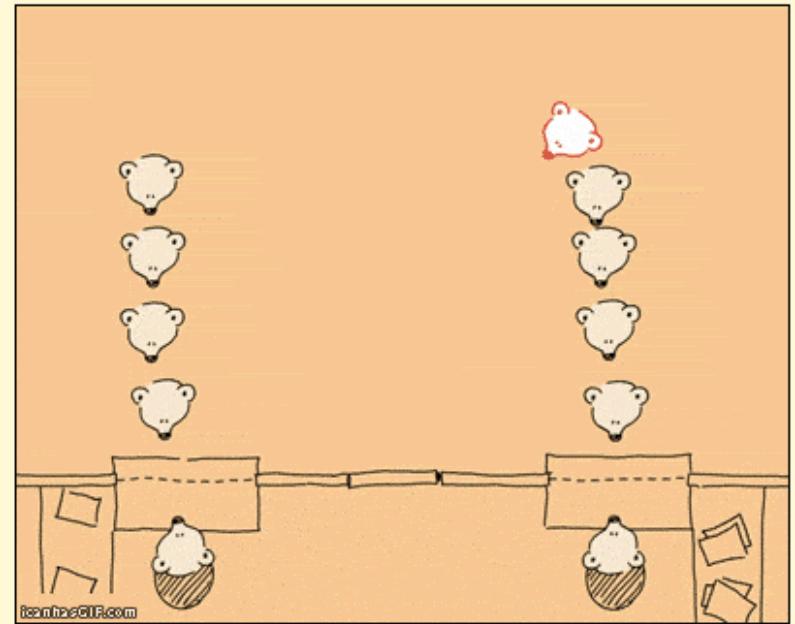
- First in - first out (FIFO)
 - Add element: enqueue (from rear of queue)
 - Remove element: dequeue (from front of queue)
 - is empty
 - check size (also reset)



Funny Queues



Story of my life...

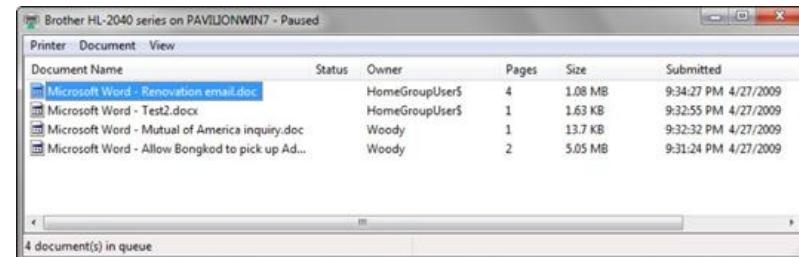
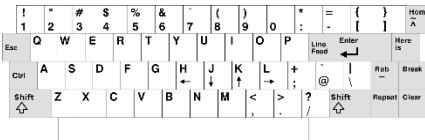
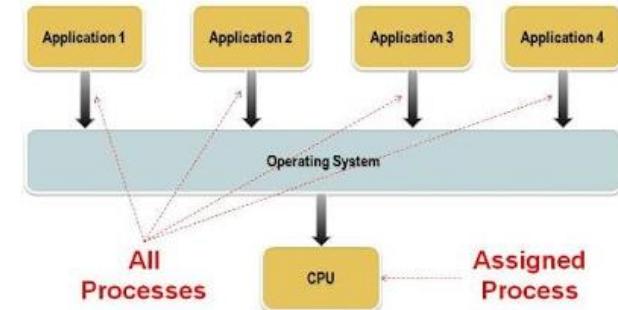


more awesome pictures at THEMETAPICTURE.COM

How useful are Queues

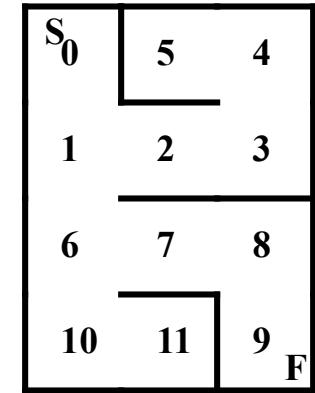
We will see some examples:

- Printing queue for waiting printing tasks
- Computing processes waiting list
- Keyboard buffer
- Traversing a maze
- ...



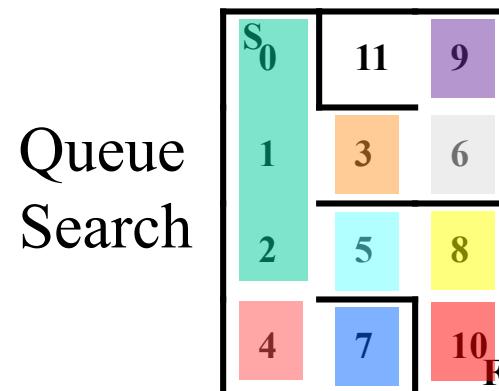
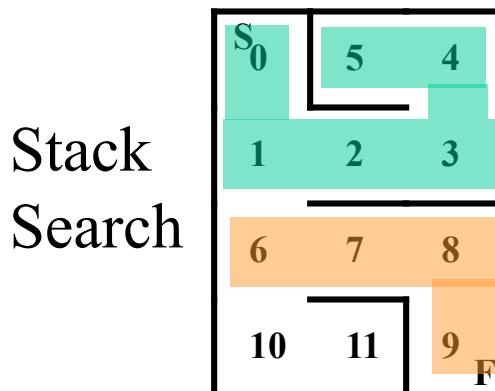
Queue Example - Maze Algorithm

- Like Stacks, Queues can also be used to store unsearched paths.
 - Repeat as long as the current square is not null and is not the finish square:
 - “Visit” the square and mark it as visited.
 - Enqueue one square on the queue for each unvisited legal move from the current square.
 - Dequeue the queue into the current square or bind the current square to null if the queue is empty.
 - If the current square is the goal we are successful, otherwise there is no solution

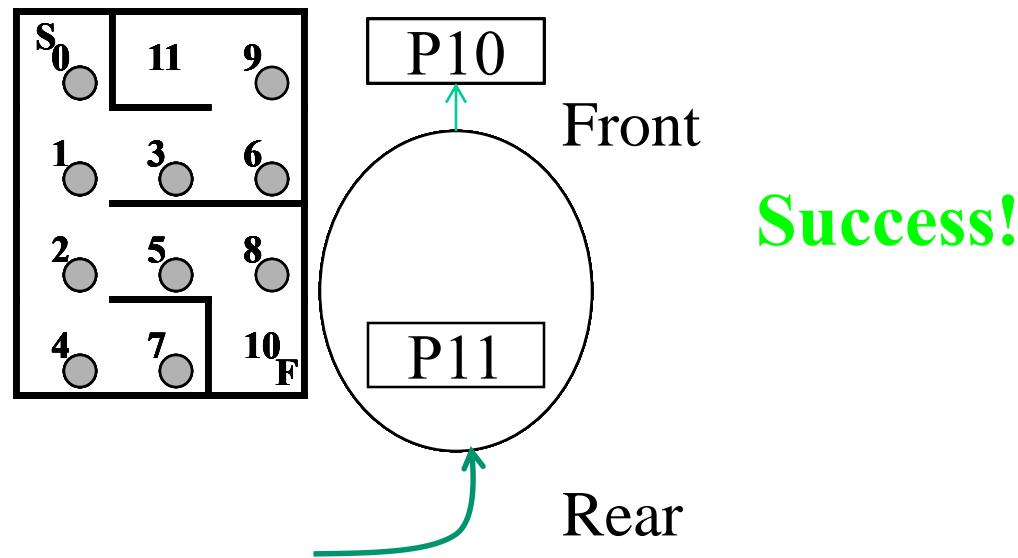


Queue Example - Searching

- The algorithm seems the same so what is the difference between using a Stack and a Queue?
- When a Stack is used, the search goes as deep along a single path as possible, before trying another path.
- When a Queue is used, the search expands a frontier of nodes equidistant from the start.



Queue Example - Maze Trace



Queue Abstract data Type

- **Queue()**

- Create a new queue that is empty.
- It needs no parameters and returns an empty queue

- **enqueue(item)**

- Adds a new item to the rear of the queue.
- It needs an item and returns nothing

- **dequeue()**

- Remove the front item from the queue
- It needs no parameters and returns the item.
- The queue is modified.

- **isEmpty()**

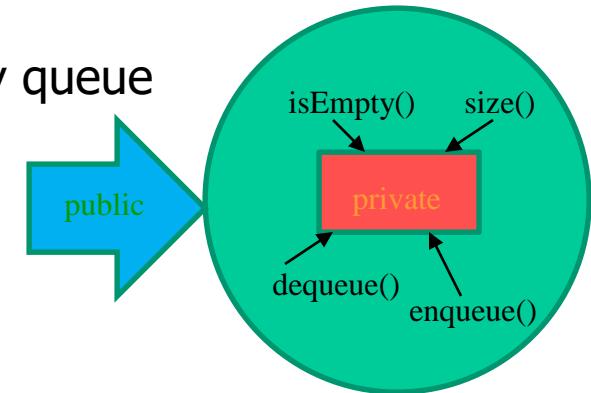
- Test to see whether the queue is empty
- It needs no parameters and returns a Boolean value

- **size()**

- Returns the number of items on the queue
- It needs no parameters and returns an integer

- **clear()**

- Empty the queue.

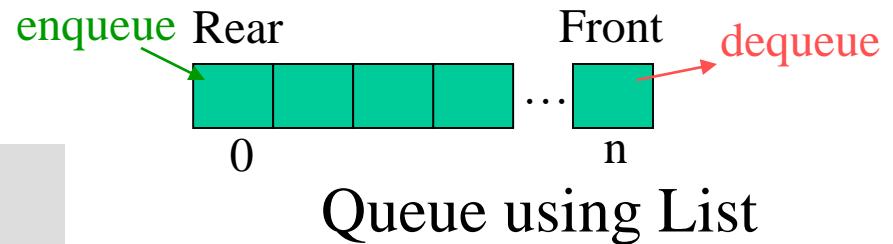


Queue Implementation in Python

- How to store the elements in the queue and allow the queue to grow and shrink one element at a time?
 - Using a python List
 - We chose the front (head) and rear (tail) of the queue to correspond to some fixed end of the list
- Implement each and every method as specified in the Queue ADT (enqueue, dequeue, isEmpty, size)
- Implement the class and instance constructor

Implementation

- Assuming we chose to have the rear on the left (position 0) of the list and the front on the right of a list.



```
class Queue:
```

```
    def __init__(self):  
        self.items = []
```

```
    def enqueue(self, item):  
        self.items.insert(0,item)
```

```
    def dequeue(self):  
        return self.items.pop()
```

```
def isEmpty(self):  
    return self.items == []
```

```
def size(self):  
    return len(self.items)
```

```
def clear(self):  
    self.items=[]
```

Printing the queue

- How to display the queue instance?
- The queue is implemented as a list and python knows how to display it.
- It is better to define a method to display the queue. Let's call it show()

```
def show(self):  
    print (self.items)
```

```
def __str__(self):  
    return str(self.items)
```

Converts the object into a string

Let's test it

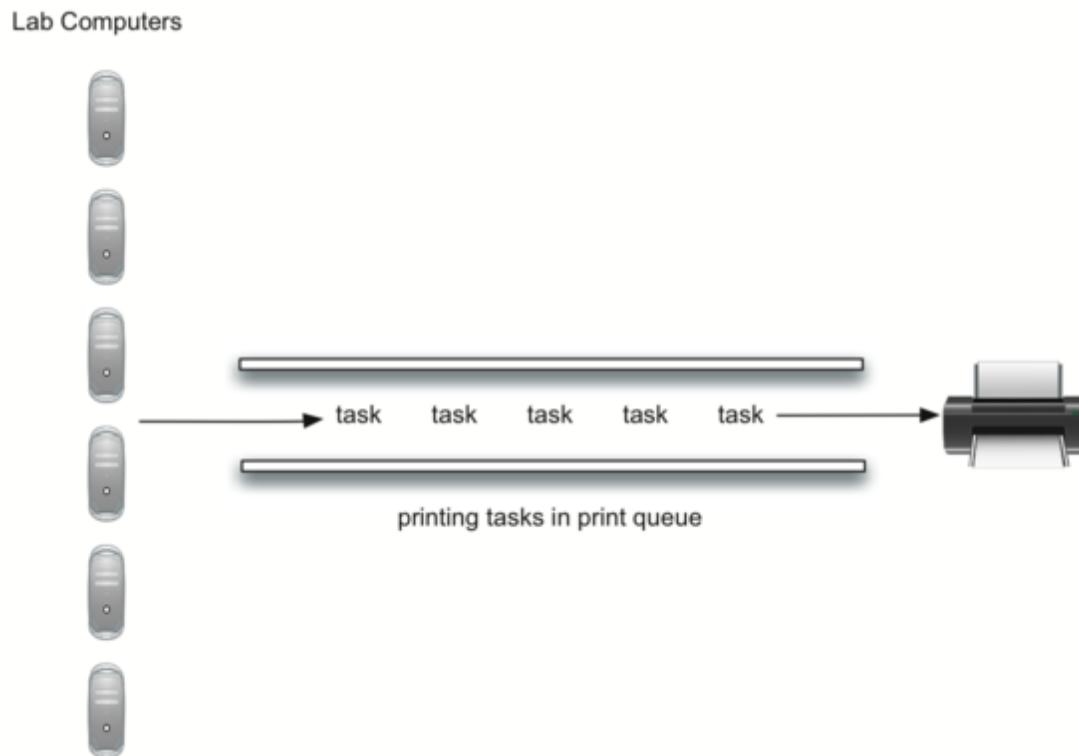
```
q=Queue()  
q.show()  
print (q.isEmpty())  
q.enqueue("bob")  
q.show()  
print (q.isEmpty())  
q.enqueue("eva")  
q.enqueue("paul")  
q.show()  
print (q.size())  
item=q.dequeue()  
q.show()  
print (item,"was first in the queue")  
print (q.size())
```

```
[]  
True  
['bob']  
False  
['paul', 'eva', 'bob']  
3  
['paul', 'eva']  
bob was first in the queue  
2
```

It seems to work as designed but these are very rudimentary tests. More stringent tests done in isolation are always required.

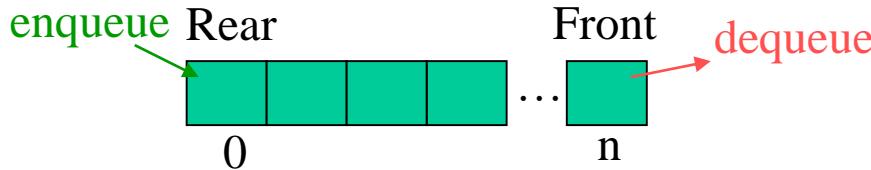
Printing Queue

- The textbook gives an illustrative example of use of Queues to manage printing tasks in a computing lab. and provides a simulation.

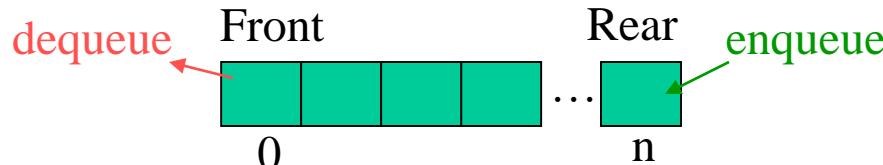


Implementation Options

- Using a list, we implemented the Queue by selecting the rear (tail) to be at position 0.



- We can opt to have the front (head) at position 0.



- We need only to rewrite enqueue() and dequeue().

```
class Queue:  
    def __init__(self):  
        self.items = []  
    def isEmpty(self):  
        return self.items == []
```

```
def size(self):  
    return len(self.items)
```

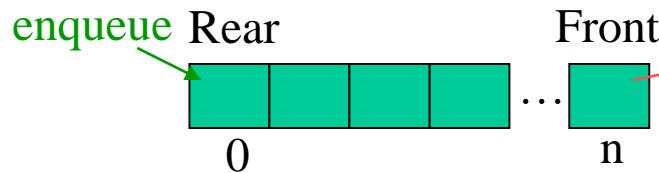
```
def enqueue(self, item):  
    self.items.append(item)
```

```
def dequeue(self):  
    return self.items.pop(0)
```

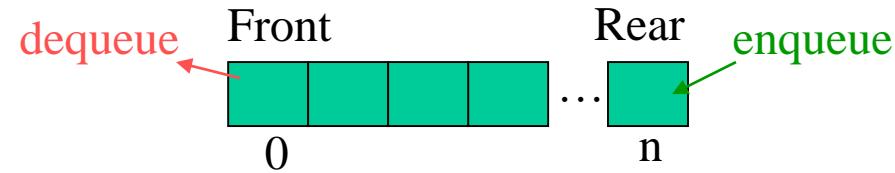
Both implementations provide the same performance
One shifts elements on enqueue, the other shifts elements on dequeue

Comparision

- Rear at position 0.



- Front at position 0.



- dequeue operation is $O(1)$
- enqueue operation is $O(n)$

- dequeue operation is $O(n)$
- enqueue operation is $O(1)$

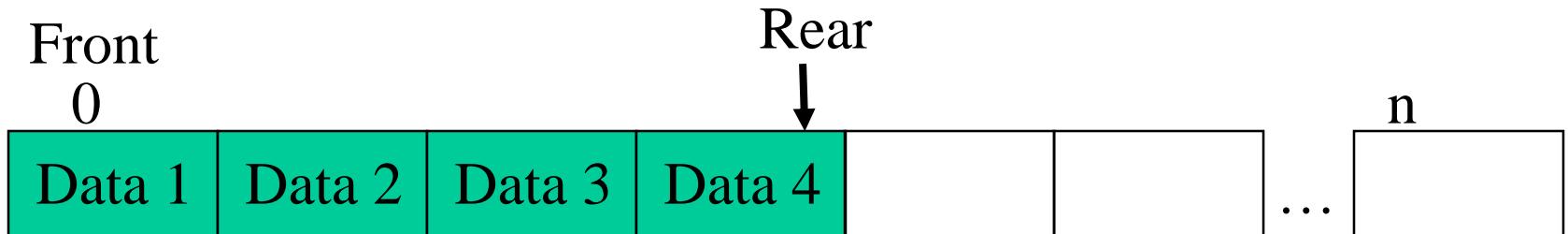
- There is an efficient implementation of the ADT queue that allows adding and removing element with $O(1)$: **Doubly-Linked-List**
- We will see this data structure later after covering Linked Lists

Bounded Queues

- A bounded queue is a queue limited to a fixed number of items.
- The capacity is fixed at creation of the queue and can not be changed
- This is possible when we know a-priori the maximum size of the queue we would need.
- We can impose the front to be at position 0 of the list and have an index for the rear that we slide along the list.

Bounded Queues Visual

- This queue of capacity n has 4 elements.
- Front is always at position 0
- Current Rear is at position 3



Bounded Queue ADT

- **BQueue(capacity)**

- Create a new queue of size *capacity* that is empty.

- **enqueue(item)**

- Adds a new item *item* to the rear of the queue (if there is room).

- **dequeue()**

- Remove the front item from the queue and return it.

- **peek()**

- Return the front item from the queue without removing it

- **isEmpty()**

- Test to see whether the queue is empty and return a Boolean value

- **isFull()** ←

- Test to see whether the queue reached full capacity and return a Boolean value

- **size()**

- Return the number of items on the queue

- **capacity()** ←

- Return the capacity of the queue

- **clear()**

- Empty the queue.

We need to raise exceptions when enqueue and reaching capacity or dequeue when queue is empty

Bounded Queue Constructor

```
class BoundedQueue:
```

```
    # Constructor, which creates a new empty queue:
```

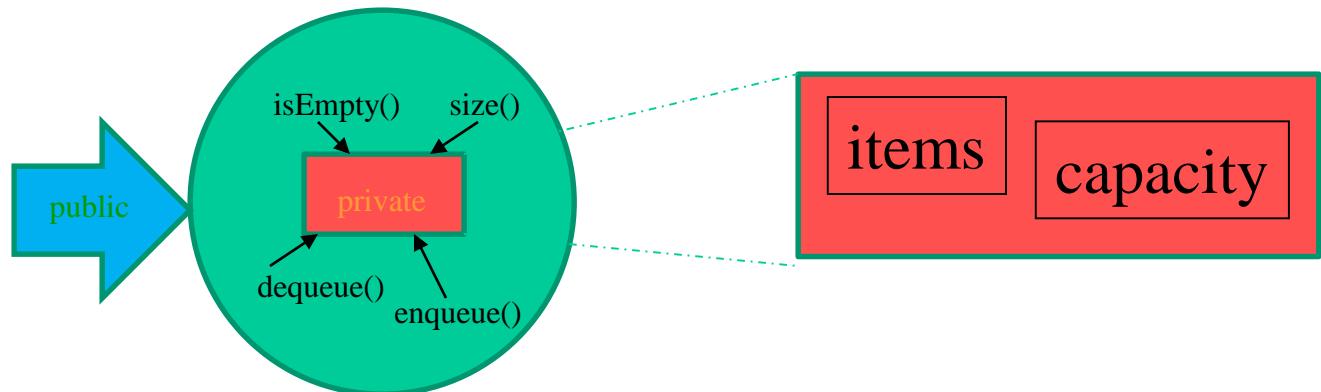
```
    def __init__(self, capacity):
```

```
        assert isinstance(capacity, int), ('Error: Type error: %s' % (type(capacity)))
```

```
        assert capacity > 0, ('Error: Illegal capacity: %d' % (capacity))
```

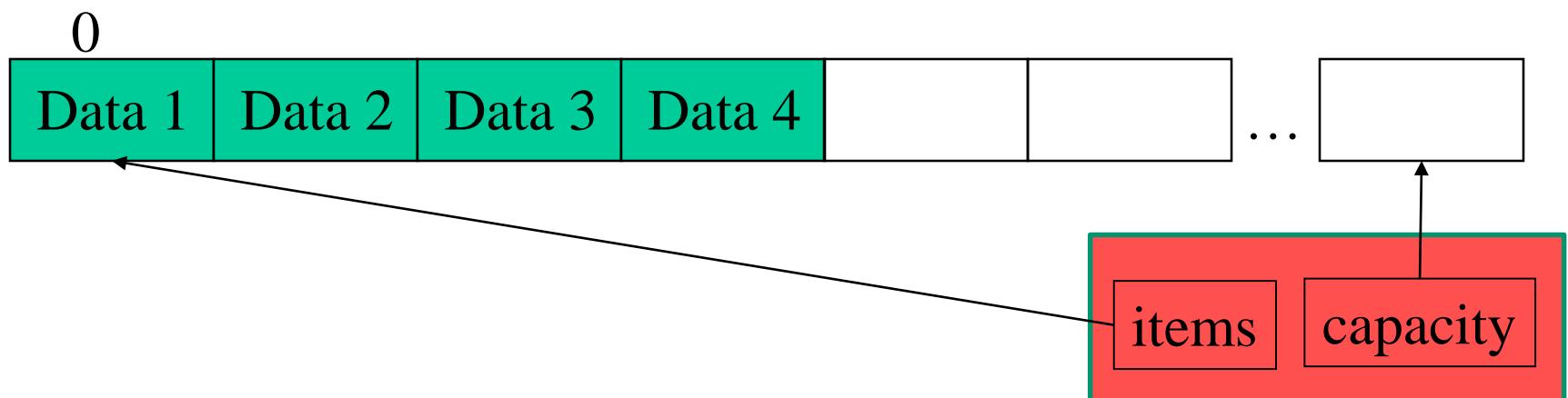
```
        self.__items = []
```

```
        self.__capacity = capacity
```



Bounded Queue enqueue()

```
# Adds a new item to the back of the queue, and returns nothing:  
def enqueue(self, item):  
    if len(self.__items) >= self.__capacity:  
        raise Exception('Error: Queue is full')  
    self.__items.append(item)
```



Bounded Queue dequeue() and peek()

```
# Removes and returns the front-most item in the queue.
```

```
# Returns nothing if the queue is empty.
```

```
def dequeue(self):
```

```
    if len(self.__items) <= 0:
```

```
        raise Exception('Error: Queue is empty')
```

```
    return self.__items.pop(0)
```

```
# Returns the front-most item in the queue, and DOES NOT change the queue.
```

```
def peek(self):
```

```
    if len(self.__items) <= 0:
```

```
        raise Exception('Error: Queue is empty')
```

```
    return self.__items[0]
```

Bounded Queue isEmpty(), IsFull(), size() and capacity()

Returns True if the queue is empty, and False otherwise:

```
def isEmpty(self):  
    return len(self.__items) == 0
```

Returns True if the queue is full, and False otherwise:

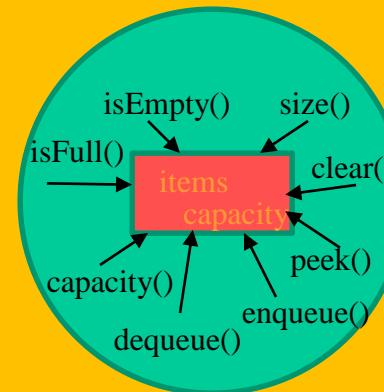
```
def isFull(self):  
    return len(self.__items) == self.__capacity
```

Returns the number of items in the queue:

```
def size(self):  
    return len(self.__items)
```

Returns the capacity of the queue:

```
def capacity(self):  
    return self.__capacity
```



Bounded Queue clear() and str()

```
# Removes all items from the queue, and sets the size to 0  
# clear() should not change the capacity
```

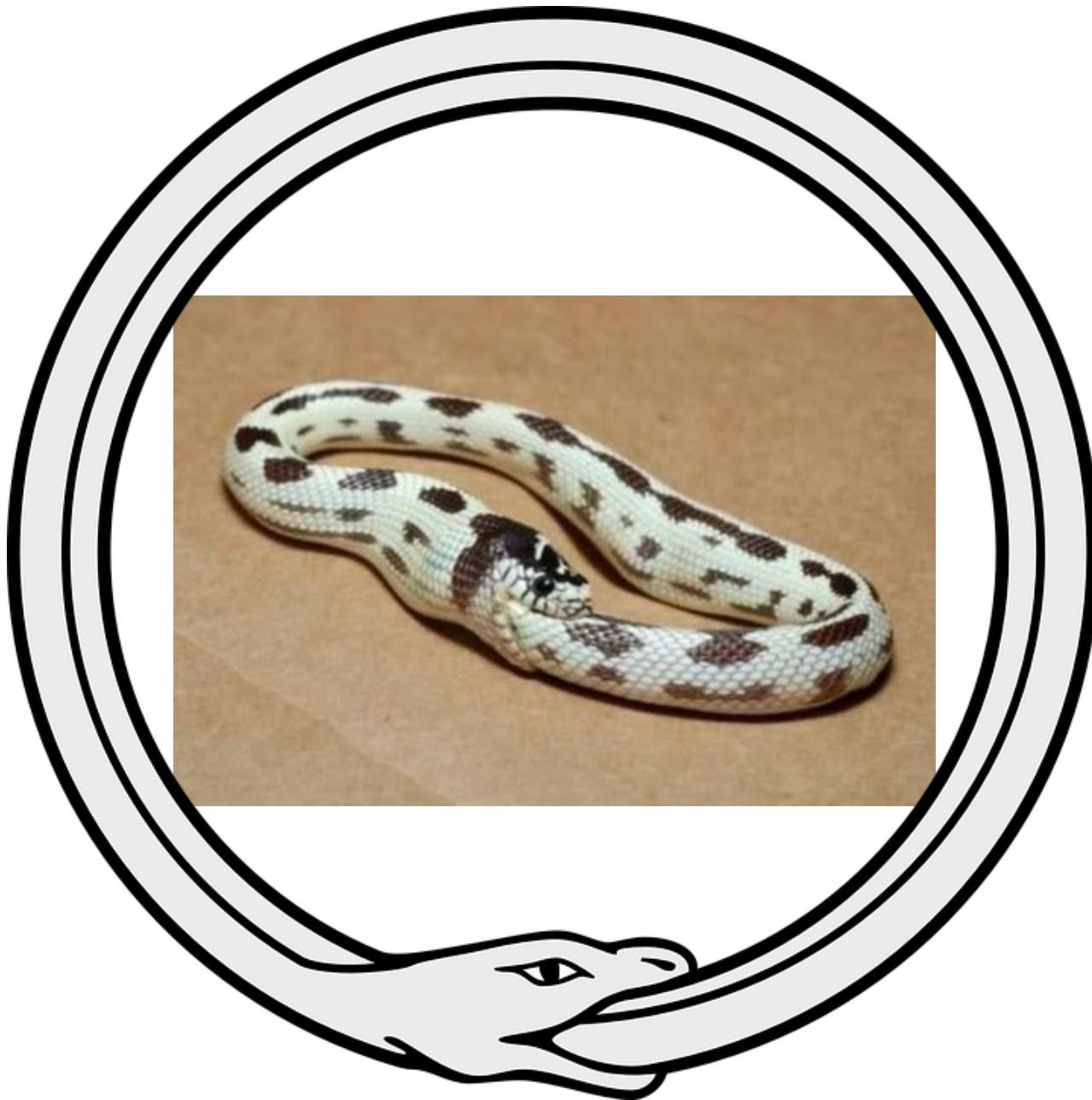
```
def clear(self):  
    self.__items = []
```

```
# Returns a string representation of the queue:
```

```
def __str__(self):  
    str_exp = "<"  
    for item in self.__items:  
        str_exp += (str(item) + " ")  
    return str_exp + "<"
```

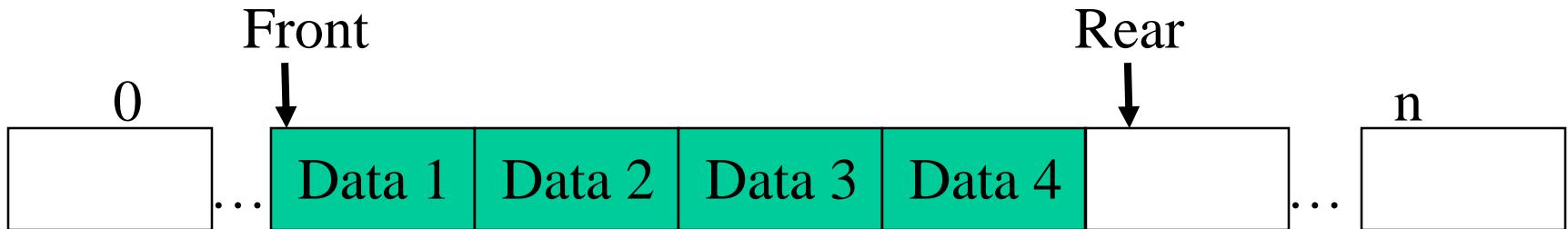
```
# Returns a string representation of the object  
# bounded queue:  
def __repr__(self):  
    return str(self) + " Max=" + str(self.__capacity)
```

<1 2 3 4 5 < Max=100

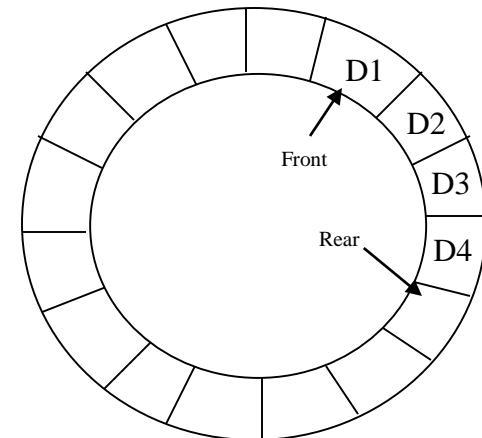


Circular Queues

- A circular queue is a bounded queue but instead of pinning the front (head) at position 0, both rear and front have indexes that slide

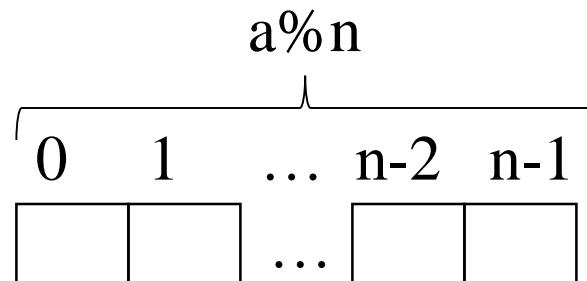


- Front is “chasing” rear modulo the capacity. This is why we call it circular



Let's talk about Modulus

- The modulo operation finds the remainder after division of one number a by another n
- a modulo n or $a \bmod n$ or $a \% n$
- *Remainder of a/n*
- *$a \% n$ is always between 0 and $n-1$*



Visualize Modulus



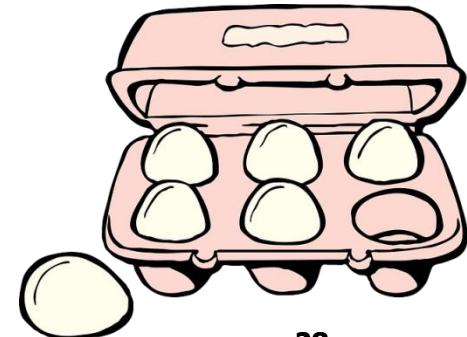
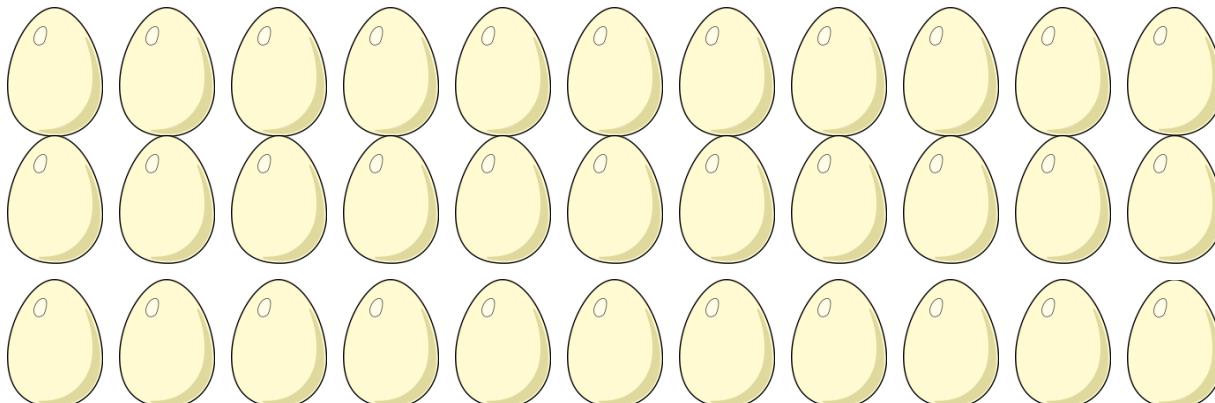
Clock is hour modulo 12



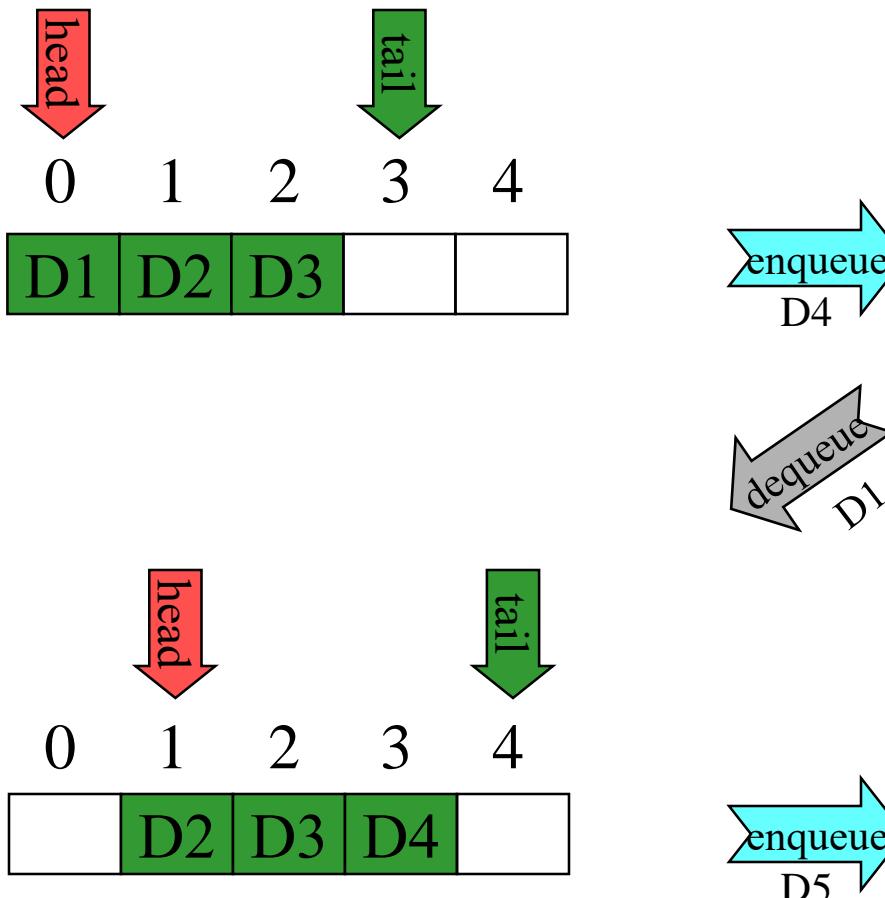
0,1,2,3,4,5,6,7,8,9,10,11,0,1,2,3,4,5,6,7,8,9,10,11,0,1,2...

$$33 \% 6 = ?$$

$$33 \% 6 = 3$$

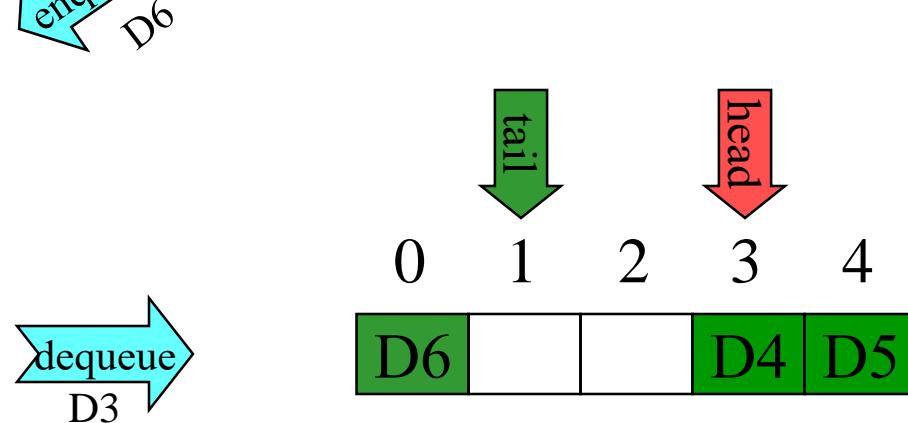
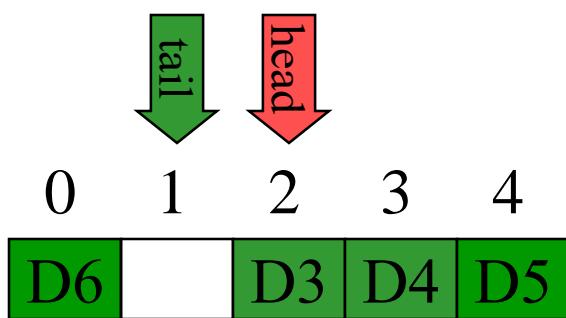
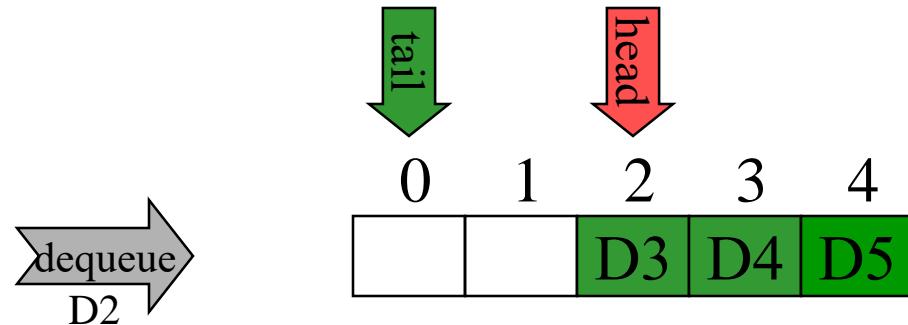
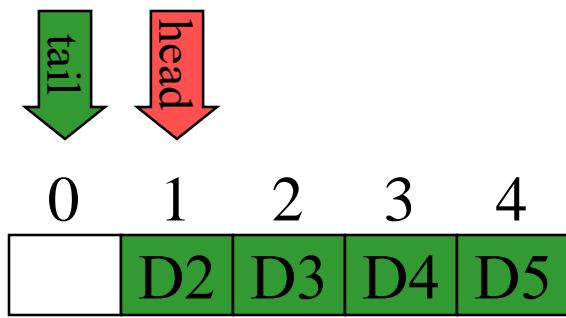


Sliding Indexes (slide 1)



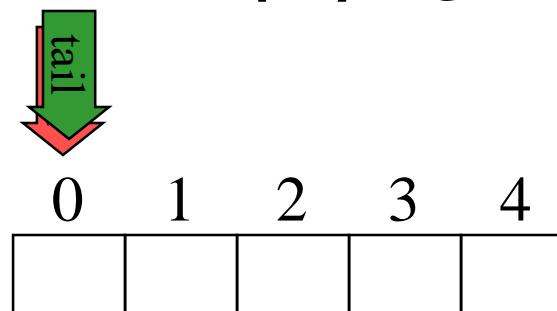
Remove from *head* then increment *head*

Sliding Indexes (slide 1)

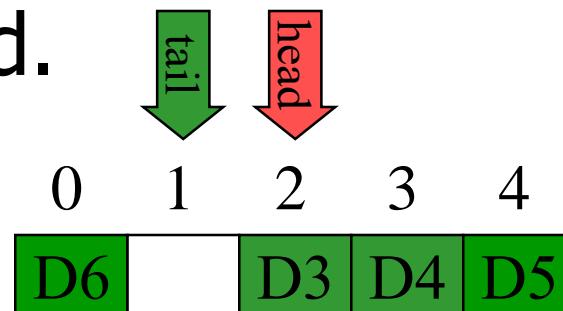


Circular Queues - Empty and Full

- We leave one empty entry in the Queue.
- The condition for an empty Queue is:
 $\text{head} == \text{tail}$.



- The condition for a full Queue is: tail is one “behind” head.



Circular Queues - Empty and Full

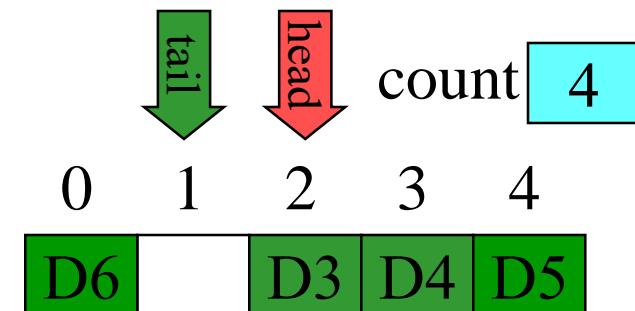
- We can avoid leaving one empty entry in the Queue by caching the current size of the queue
- The condition for an empty Queue is: the size is 0.
- The condition for a full Queue is: the size is equal to the queue maximum capacity.
- When enqueueing the cached size is incremented
- When dequeuing the cached size is decremented

Dequeue:

remove from head
increment head

Enqueue:

add to tail
increment tail



Circular Queue ADT

- **CQueue(capacity)**

- Create a new queue of size *capacity* that is empty.

- **enqueue(item)**

- Adds a new item *item* to the rear of the queue (if there is room).

- **dequeue()**

- Remove the front item from the queue and return it.

- **peek()**

- Return the front item from the queue without removing it

- **isEmpty()**

- Test to see whether the queue is empty and return a Boolean value

- **isFull()**

- Test to see whether the queue reached full capacity and return a Boolean value

- **size()**

- Return the number of items on the queue

- **capacity()**

- Return the capacity of the queue

- **clear()**

- Empty the queue.

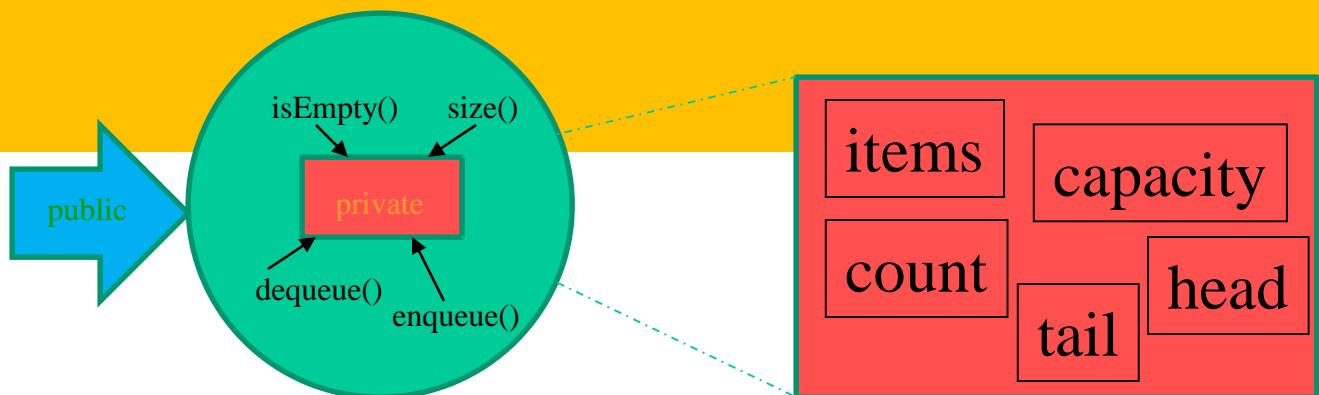
We need to raise exceptions when enqueue and reaching capacity or dequeue when queue is empty

Circular Queue Constructor

```
class CircularQueue:  
    # Constructor, which creates a new empty queue:  
    def __init__(self, capacity):  
        if type(capacity) != int or capacity<=0:  
            raise Exception ('Capacity Error')
```

```
        self.__items = []  
        self.__capacity = capacity  
        self.__count=0  
        self.__head=0  
        self.__tail=0
```

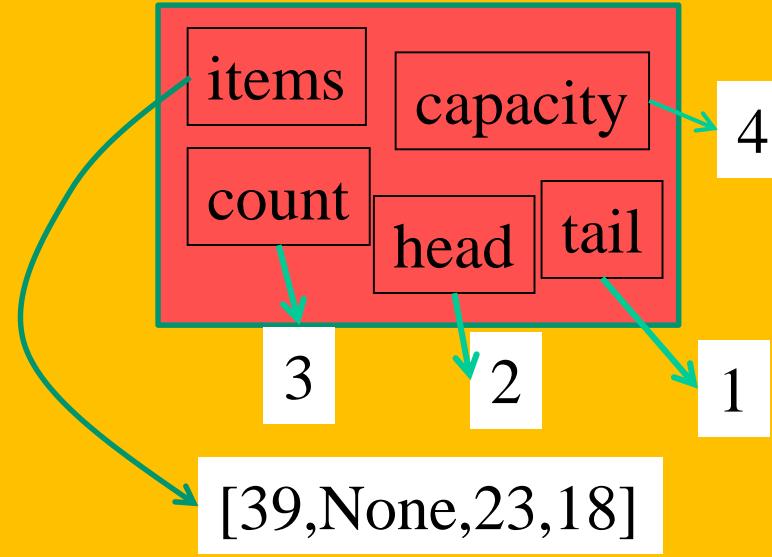
Different way to
test input
parameter



Circular Queue enqueue()

Adds a new item to the back of the queue, and returns nothing:

```
def enqueue(self, item):  
    if self.__count==self.__capacity:  
        raise Exception('Error: Queue is full')  
    if len(self.__items)<self.__capacity:  
        self.__items.append(item)  
    else:  
        self.__items[self.__tail]=item  
    self.__count +=1  
    self.__tail=(self.__tail +1) % self.__capacity
```



q=CircularQueue(4)
q.enqueue(45)
q.enqueue(67)
q.enqueue(23)
q.dequeue()
q.enqueue(18)
q.dequeue()
q.enqueue(39)

q→[]
q→[45]
q→[45,67]
q→[45, 67, 23]
q→[None,67,23]
q→[None,67,23,18]
q→[None,None,23,18]
q→[39,None,23,18]

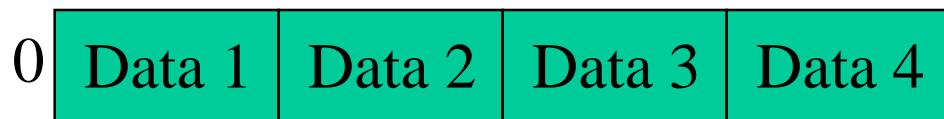
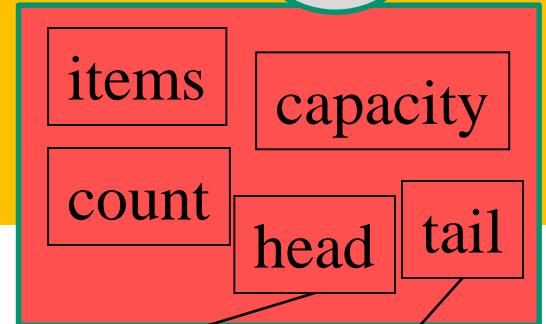
instance created
append
append
append
overwrite with None
append
overwrite with None
overwrite a None

Circular Queue enqueue()

Adds a new item to the back of the queue, and returns nothing:

```
def enqueue(self, item):  
    if self.__count == self.__capacity:  
        raise Exception('Error: Queue is full')  
    if len(self.__items) < self.__capacity:  
        self.__items.append(item)  
    else:  
        self.__items[self.__tail] = item  
        self.__count += 1  
        self.__tail = (self.__tail + 1) % self.__capacity
```

We could have avoided it if the constructor had:
for i in range(0, capacity):
 self.items.append(None)



q=CircularQueue(4)

q→[None,None,None,None]

Circular Queue dequeue() & peek()

```
# Removes and returns the front-most item in the queue.
```

```
# Returns nothing if the queue is empty.
```

```
def dequeue(self):
```

```
    if self.__count == 0:
```

```
        raise Exception('Error: Queue is empty')
```

```
    item= self.__items[self.__head]
```

```
    self.__items[self.__head]=None
```

```
    self.__count -=1
```

```
    self.__head=(self.__head+1) % self.__capacity
```

```
    return item
```

```
# Returns the front-most item in the queue, and DOES NOT change the queue.
```

```
def peek(self):
```

```
    if self.__count == 0:
```

```
        raise Exception('Error: Queue is empty')
```

```
    return self.__items[self.__head]
```

Circular Queue isEmpty(), IsFull(), size() and capacity()

```
# Returns True if the queue is empty, and False otherwise:
```

```
def isEmpty(self):  
    return self.__count == 0
```

```
# Returns True if the queue is full, and False otherwise:
```

```
def isFull(self):  
    return self.__count == self.__capacity
```

```
# Returns the number of items in the queue:
```

```
def size(self):  
    return self.__count
```

```
# Returns the capacity of the queue:
```

```
def capacity(self):  
    return self.__capacity
```

Circular Queue clear() & str()

```
# Removes all items from the queue, and sets the size to 0
```

```
# clear() should not change the capacity
```

```
def clear(self):
```

```
    self.__items = []
```

```
    self.__count=0
```

```
    self.__head=0
```

```
    self.__tail=0
```

```
# Returns a string representation of the queue:
```

```
def __str__(self):
```

```
    str_exp = "]"
```

```
    i=self.__head
```

```
    for j in range(self.__count):
```

```
        str_exp += str(self.__items[i]) + " "
```

```
        i=(i+1) % self.__capacity
```

```
    return str_exp + "]"
```

]2 3 4 5 6]

Circular Queue repr()

```
# # Returns a string representation of the object CircularQueue
def __repr__(self):
    return str(self.__items) + " H=" + str(self.__head) + " T=" + str(self.__tail) + " (" + str(self.__count) + "/" + str(self.__capacity) + ")"
```

]None, None, 2, 3, 4, 5, 6, None] H=2 T=7 (5/8)

]8, None, None, 3, 4, 5, 6, 7] H=3 T=1 (6/8)

Purpose of `__str__` and `__repr__`

- Both are used to represent an object
- `__str__` returns the informal string representation of an instance
- `__str__` is called by the built-in function `str()` and by a print statement
- `__repr__` returns an official string representation of an instance
- `__repr__` is called by the built-in function `repr()`

Double ended Queue = Deque

- Also written Dequeue (head-tail linked list)
- Addition of elements can be done on both ends and deletion can be done on both ends
- Can be input-restricted or output-restricted
- `append()` and `appendLeft()` to add
- `pop()` and `popLeft()` to remove
- `peek()` and `peekLeft()` to see first available
- Use eg. Scheduling jobs in multithread processor (job stealing)