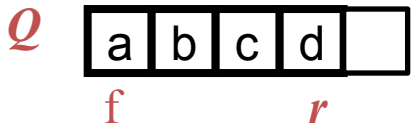
Queue

Tutorial #9

Queue

- A (**simple**) queue is a collection of objects that can be accessed in "**first-in, first-out**" order: The only element that is visible and that can be removed is the oldest remaining element.

- Three variables keep track of the head, tail, and size
  - **Head (front)** :index of the front element
  - **Tail (rear)**: index of the last element, where we add new elements (enqueue)
- **size** is number of entries in the queue



- Queues occur in real life a lot :
  - Queues at checkout
  - Queues in banks
- Queues In software systems:
  - Queue of requests at a web server

Insertion and Remove

- Insertion (enquque):
  - Placing an item in a queue, which is done at the end of the queue; rear

- Deletion (dequque):
  - Removing an item from a queue which is done at the front end.

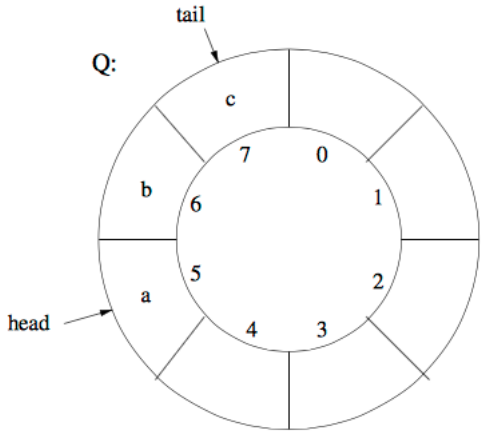
The size of the queue :

front –rear+1

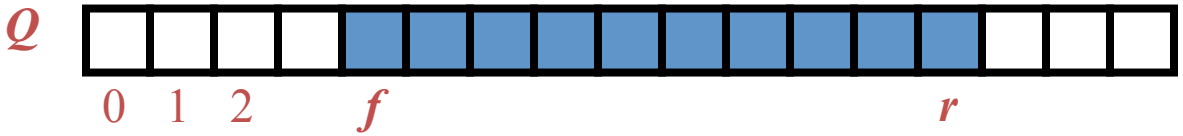
Circular Queue

Its structure can be like the figure

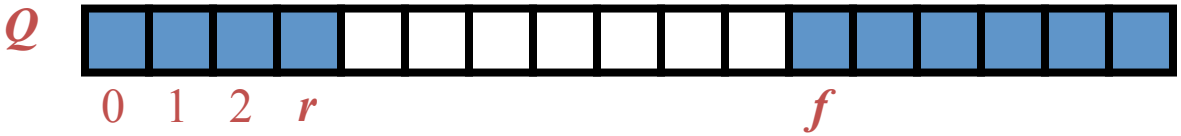
- In circular queue once the queue is full, the ***first*** element of the queue becomes the ***rear*** most element ; if and only if the ***front*** has moved forward .
- Otherwise it will be a “queue overflow ” state



normal configuration



wrapped-around configuration



The mod operator

- `mod` can be used to calculate the front and back positions in a circular array, therefore avoiding comparisons to the array size
  - The back of the queue is:
    - `(front + size - 1) % N`
    - where `size` is the number of items currently in the queue
  - After removing an item the front of the queue is:
    - `(front + 1) % N;`

- Give **pseudocode** for each queue operation when it is being implemented as a circular queue. Your pseudocode should be detailed enough to be clear that the queue *is* being implemented as a circular queue, rather than in some other way.
  - Enqueue()
  - Dequeue()
  - Size()
  - IsEmpty()

```
Algorithm size()
    return size

Algorithm isEmpty()
    return size == 0
```

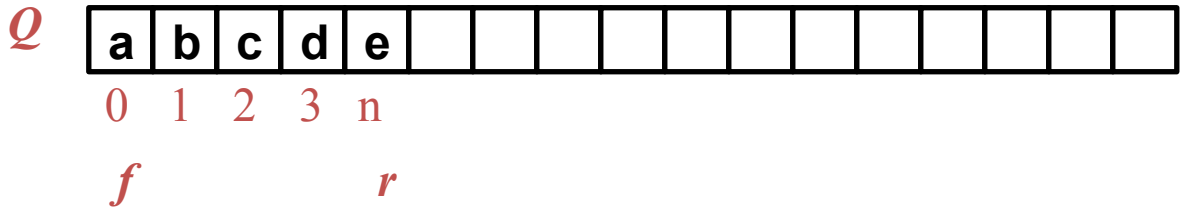
```
Algorithm enqueue(o)
    if size() = N then
        throw FullQueueException
     else
        r ← (r + 1) mod N
        Q[r] ← o
        size=size+1
```

```
Algorithm dequeue()
    if isEmpty() then
        throw EmptyQueueException
     else
        o ← Q[f]
        f ← (f + 1) mod N
        size =size -1
        return o
```

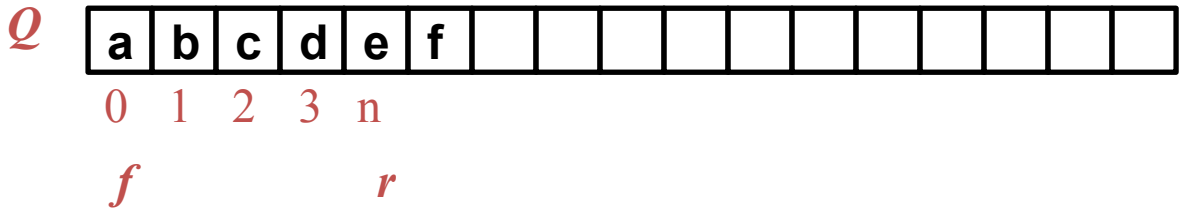
- Consider the following implementation:
  - the elements of the queue are always stored at the beginning of the array: If the queue currently includes n elements (where $n \leq N$) then these should be stored in positions 0, 1, …, n−1 of the array.
  - What is wrong with this implementation?

What is wrong with this implementation?

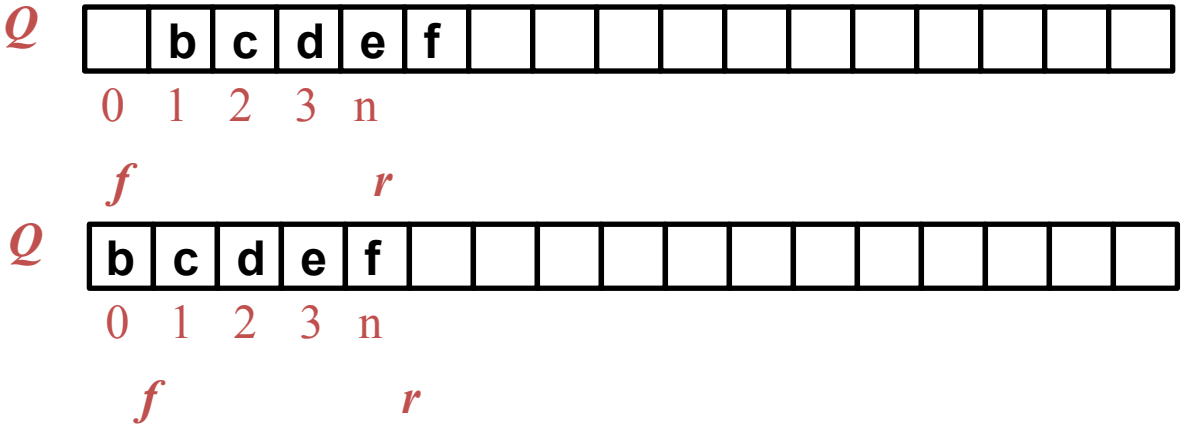
– What happens each time you try to add/remove something?



- **Enqueue("f")**

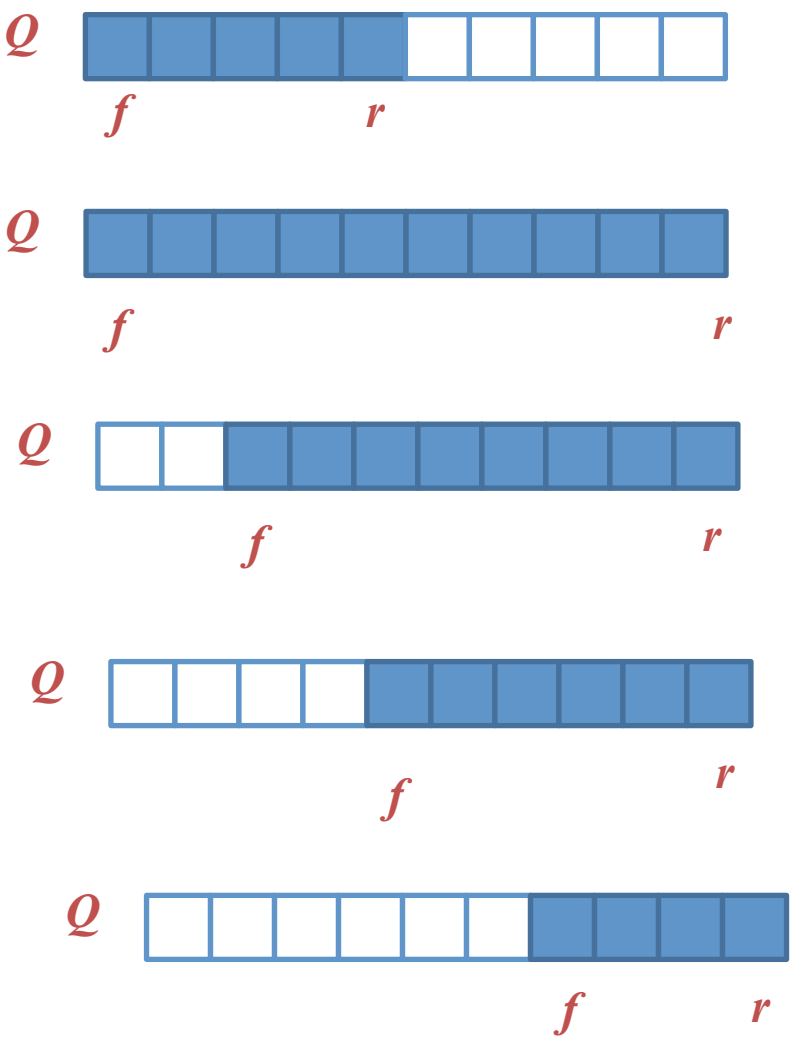


- **Dequeue()**



- Consider the following implementation:
  - Allow the queue to be used just like it is as described in lectures, except that we cannot wrap around: An attempt to enqueue an element should cause an exception to be thrown, and the queue to be unchanged, if “rear” has value N−1.
    - What is wrong with this implementation?

No Wrap around!



An implementation of an unbounded queue using static array: The amortized Cost

- Dequeue → Θ(1)
- Enqueue →
  - Think about if you double every time array fills up, what is the average cost per insertion? Suppose array size is 100.
  - First 100 insertions are cheap (constant time). Next one costs 100 operations.Then, the next 100 will be cheap. Next one costs 200.Then, the next 200 will be cheap, next one costs 400, etc.
  - One expensive operation makes subsequent ones cheap. Average performance per insert is at most one element copy + one element update. It is still **constant time**

Reference: http://web.eecs.umich.edu/~aprakash/eecs282/lectures/14-queues.pdf