UTM
UNIVERSITI TEKNOLOGI MALAYSIA

DATA STRUCTURE & ALGORITHM

Chapter 09

Queue

Part 1 -Queue Implementation Linked List

Nor Bahiah Hj Ahmad & Dayang Norhayati A.Jawawi
School of Computing

Objectives

At the end of the lesson students are expected to be able to:

- Understand queue concepts and applications.
- Understand queue structure and operations that can be done on queue.
- Understand and know how to implement queue using array and linked list : linear array, circular array, linear link list and circular list.

Introduction to Queue

New items enter at the back, or rear, of the queue

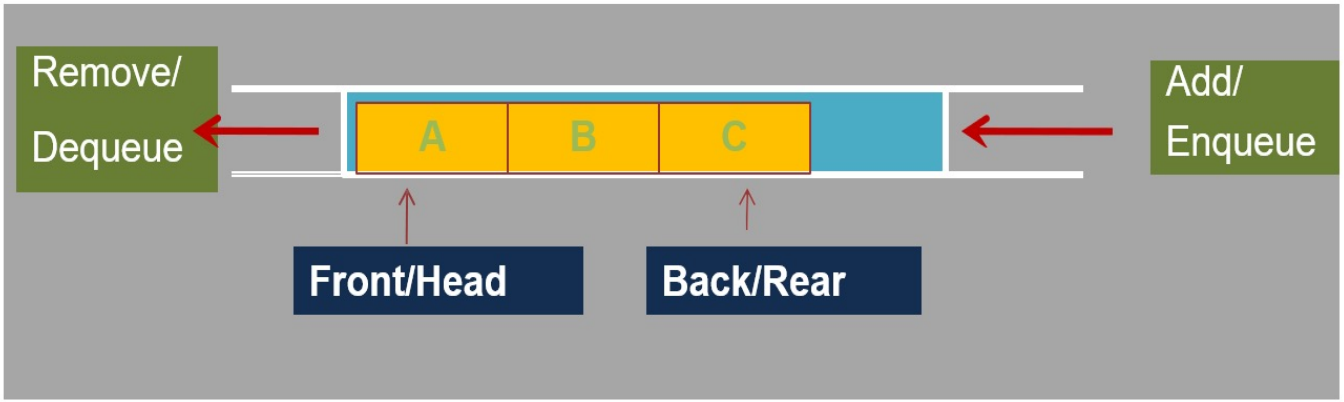
- Items leave from the front of the queue
- First-in, first-out (FIFO) property
  - The first item inserted into a queue is the first item to leave
  - Middle elements are logically inaccessible
- Important in simulation & analyzing the behavior of complex systems

Queue Applications

Real-World Applications
- Cashier lines in any store
- Check out at a bookstore
  - Bank / ATM
  - Call an airline

- Computer Science Applications
  - Print lines of a document
  - Printer sharing between computers
  - Recognizing palindromes
  - Shared resource usage (CPU, memory access, ...)
- Simulation
  - A study to see how to reduce the wait involved in an application

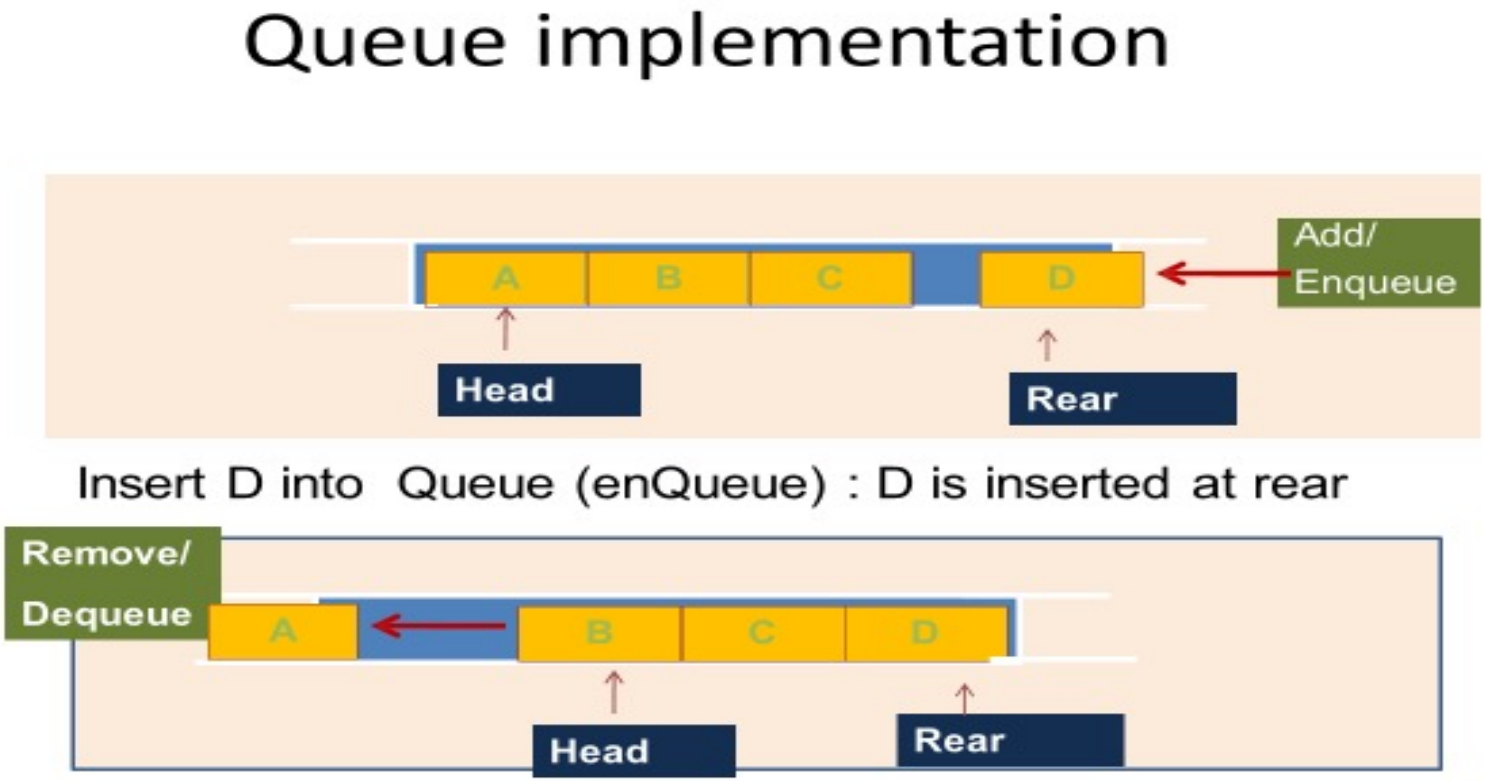
Queue Implementation



Basic Structure of a Queue:

- Data structure that hold the queue
- head
- rear

Queue Implementation



Abstract Data Type Queue

Implementation:

- Array-based (Linear or Circular)
- Pointer-based : Link list (Linear or Circular)

Abstract Data Type Queue

- ADT queue operations
  - Create an empty queue
- Destroy a queue
- Determine whether a queue is empty
- Add a new item to the queue
- Remove the item that was added earliest
- Retrieve at Front
- Retrieve at Back the item that was added earliest

Queue: Linear Array Implementation

- Number of elements in Queue are fixed during declaration.
- Need ***isFull()*** operation to determine whether a queue is full or not.
- Queue structure need at least 3 elements:

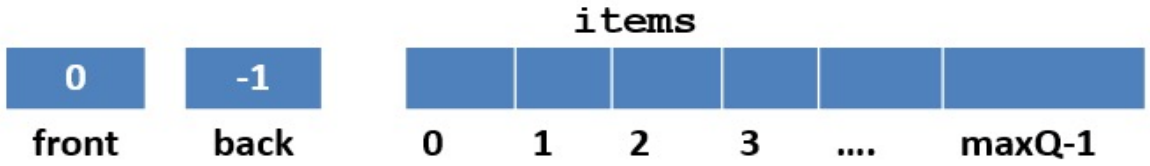
1) Element to store items in Queue
2) Element to store index at head
3) Element to store index at rear

Queue
front *rear* *items*
createQueue() *destroyQueue()* *isEmpty();* *isFull();* *enQueue();* *deQueue();* *getFront();* *getRear();*

CreateQueue() operation

- Linear Array implementation
  - *front* & *back* are indexes in the array
  - Initial condition: *front =0* & *back = -1*

```
Queue::Queue()
{ front = 0;
  back = -1;
}
```



Initial state for a queue linear array

Queue operations

- **Destroy Queue distructor : All elements in the queue will be disposed.**

```
queue::~queue()
{ delete [ ] items; }
```

- **Check whether a queue is empty**
  - Queue Empty Condition : back < front

```
bool queue::isEmpty()
{ return bool(back < front); }
```

Queue operations

Check whether a queue is Full

- Queue Full Condition : back = size -1

```
bool queue::isFull()
{ return bool(back == size - 1); }
```

- No more item can be insert into a queue, when a queue is full.

Queue operations

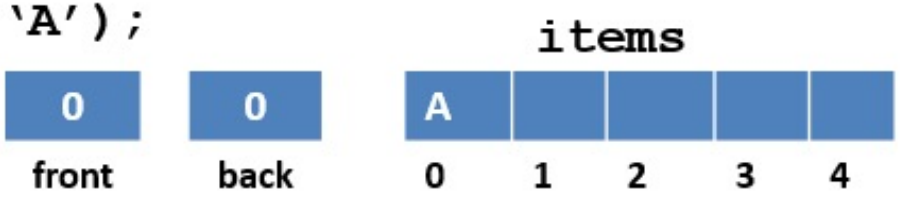
- Insert into a queue (enQueue)
  - Increment back
  - Insert item in `items[ back ]`

```
void queue::enQueue(char insertItem)
{ if (isFull())
      cout<< "\nCannot Insert. Queue is full!";
  else
  { //insert at back
    back++;
    items[back] = insertItem;
  } // end else if
}
```

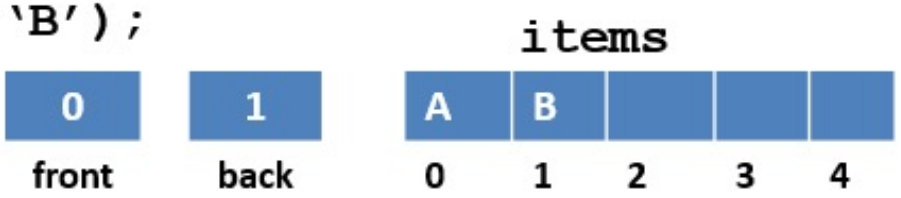
enQueue operations for a queue with size = 5

```
Queue myQueue;
```


```
myQueue.enQueue('A');
```

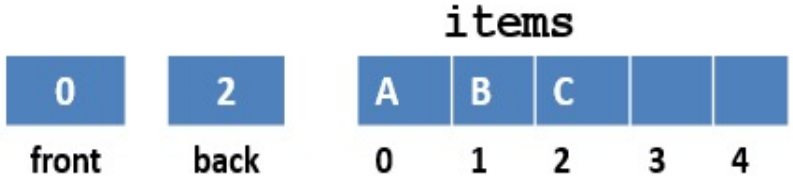



```
myQueue.enQueue('B');
```




```
myQueue.enQueue('C');
```

Queue operations



- Item at front and back can be retrieved from queue

```
char queue:: getFront() // get item at Front
{ return items[front] ; }
```

```
char queue::getRear() // get item at Back
{ return items[back] ; }
```

```
cout << myQueue.getFront(); //output is A
cout << myQueue.getRear(); // output is C
```

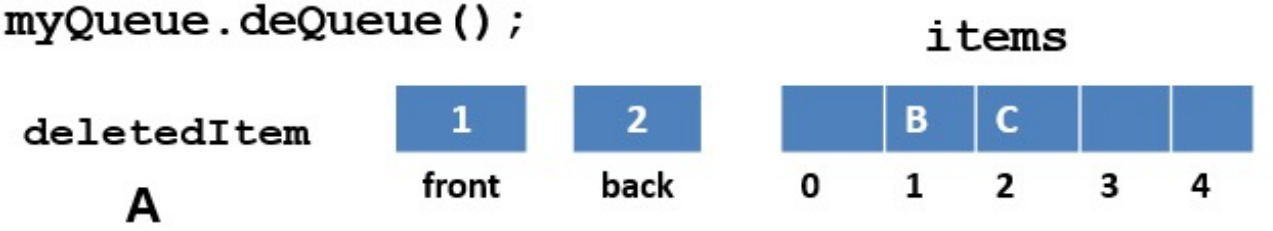
Queue operations

- Delete from a queue (deQueue)
  - Increment front

```
void queue::deQueue()
{ if (isEmpty())
    cout<< "\nCannot remove item. Empty Queue!";
  else
  { //retrieve item at front
    deletedItem = items[front];
    front++;
  } // end else if
}
```

deQueue operations

```
myQueue.deQueue();
```




```
myQueue.deQueue();
```

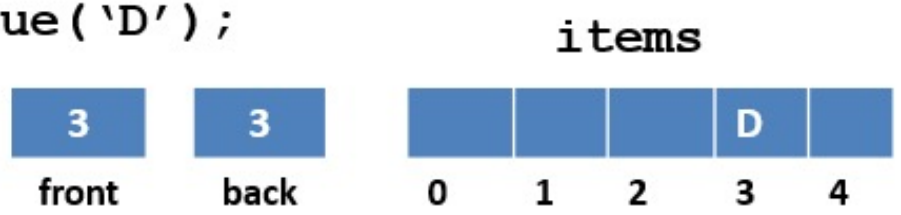

```
myQueue.deQueue();
```


```
myQueue.deQueue();
```

```
Cannot remove item.
Queue is Empty with back < front
```

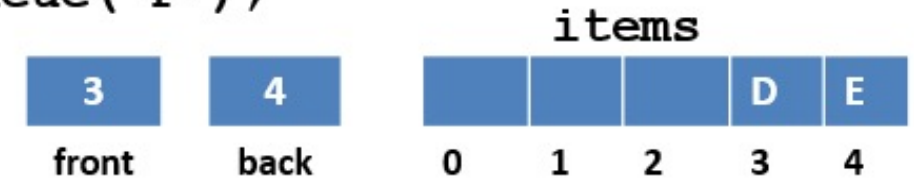
Queue operations - enQueue

```
myQueue.enQueue('D');
```




```
myQueue.enQueue('E');
```


```
myQueue.enQueue('F');
```

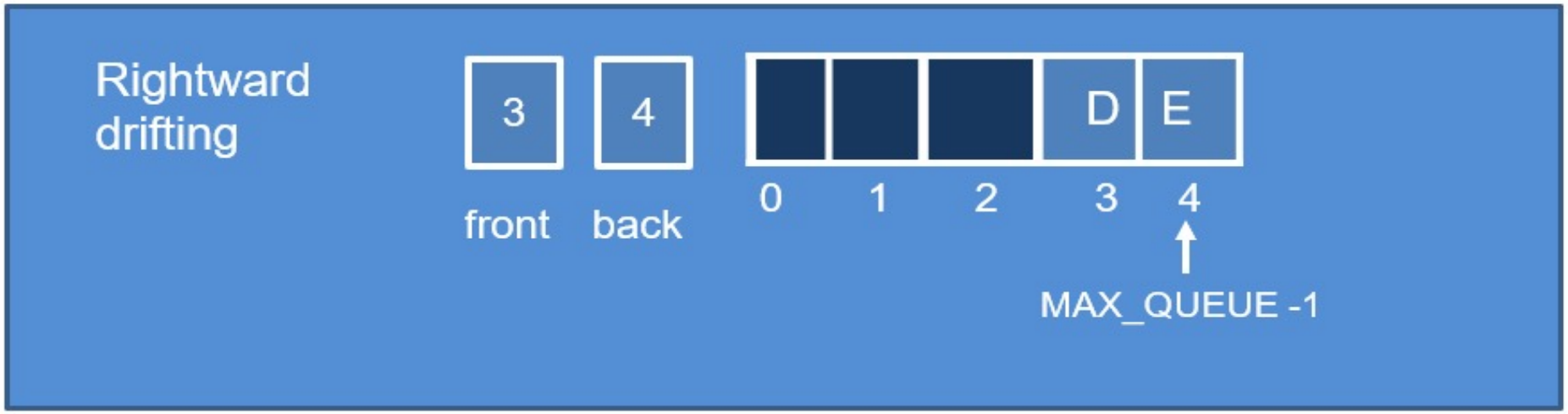



```
Cannot insert F, even though there are empty spaces
in front of the queue array.
Currently, Queue is FULL with back == size - 1.
```

Linear array implementation - drawback

Problem: Rightward-Drifting:

- After a sequence of additions & removals, items will drift towards the end of the array
- Even though, there are empty spaces in front of the queue array, enQueue operation cannot be performed on the queue, since back = max_queue – 1.



Rightward drifting solutions

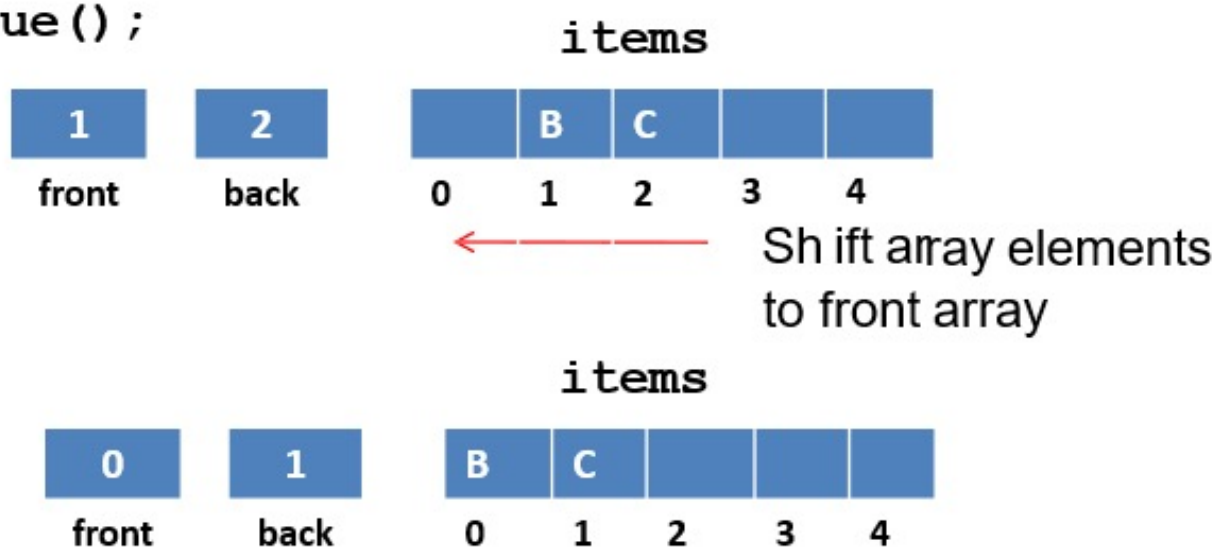
To optimize space and to solve rightward drifting:

1. Shift array elements after each deletion.

```
myQueue.deQueue();
```

deletedItem

A



However, shifting is not effective and dominates the cost of the implementation.

Queue circular array

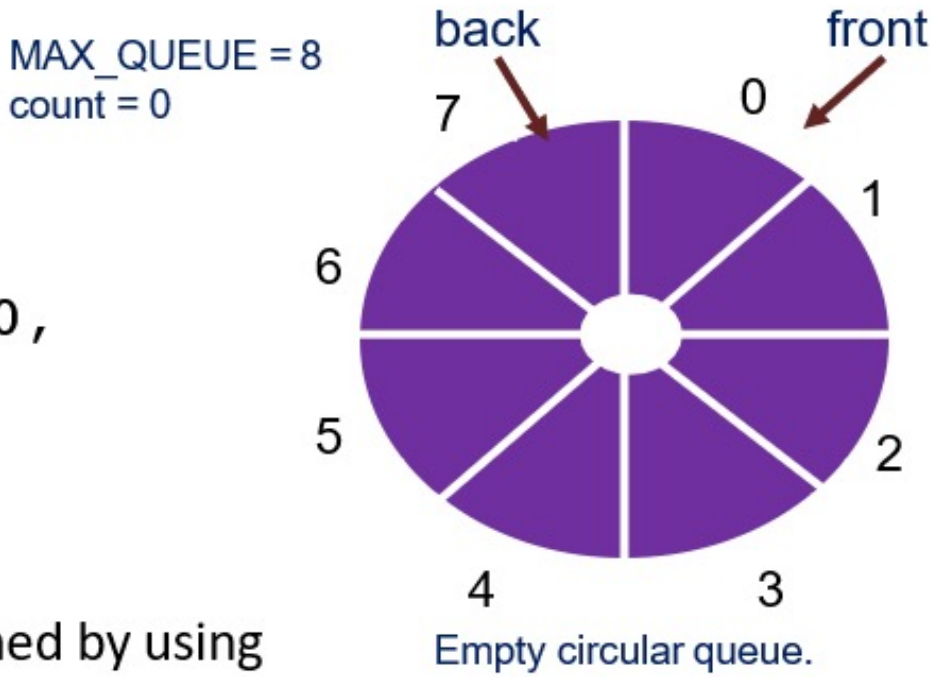
- Problem:
  - front & back no longer can be used to distinguish between queue-full & queue-empty conditions
- Solution:
  - Use a counter
  - count == 0 means empty queue
  - count == MAX_QUEUE means full queue
- Disadvantage
  - Overhead of maintaining a counter or flag

Circular Array Implementation

- Queue declarations

```
const int MAX_QUEUE = maximum-size-of-queue;
QueueItemType items [MAX_QUEUE];
int   front;
int   back;
int   count
```

- Initial condition:
  - `count = 0, front = 0,`
  - `back = MAX_QUEUE - 1`



Empty circular queue.

- The Wrap-around effect is obtained by using arithmetic (%-operator)

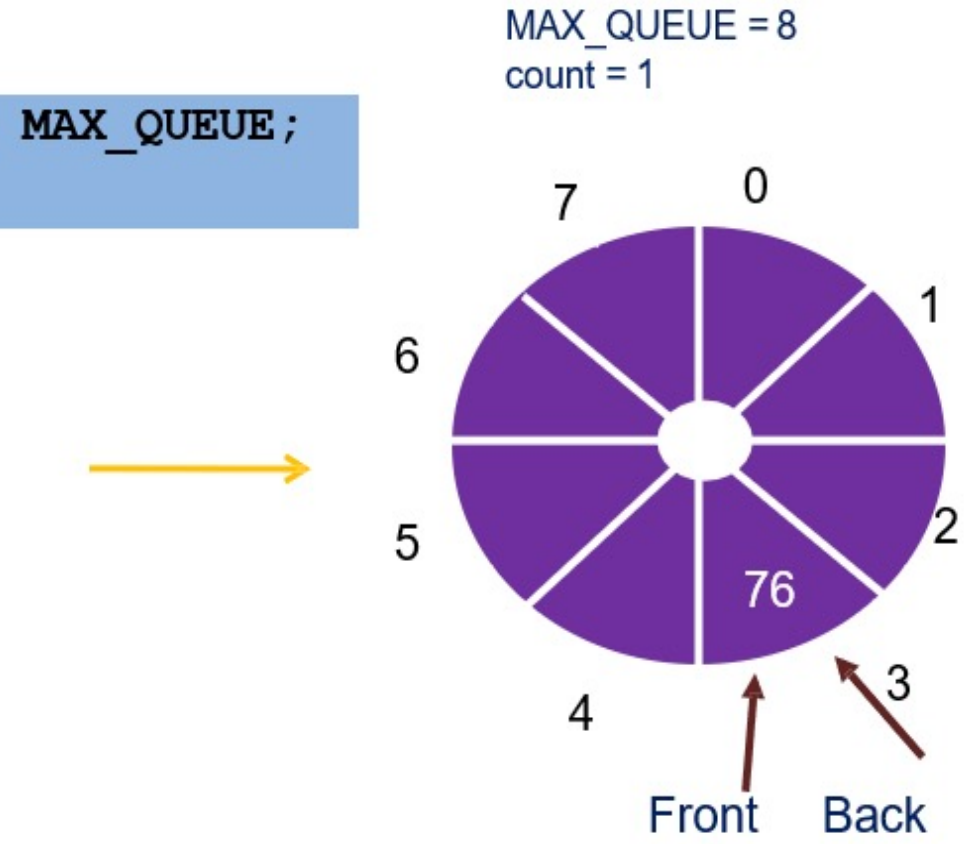
Circular Array Implementation

Deletion

Increment *front* using modulo arithmetic
Decrement *count*

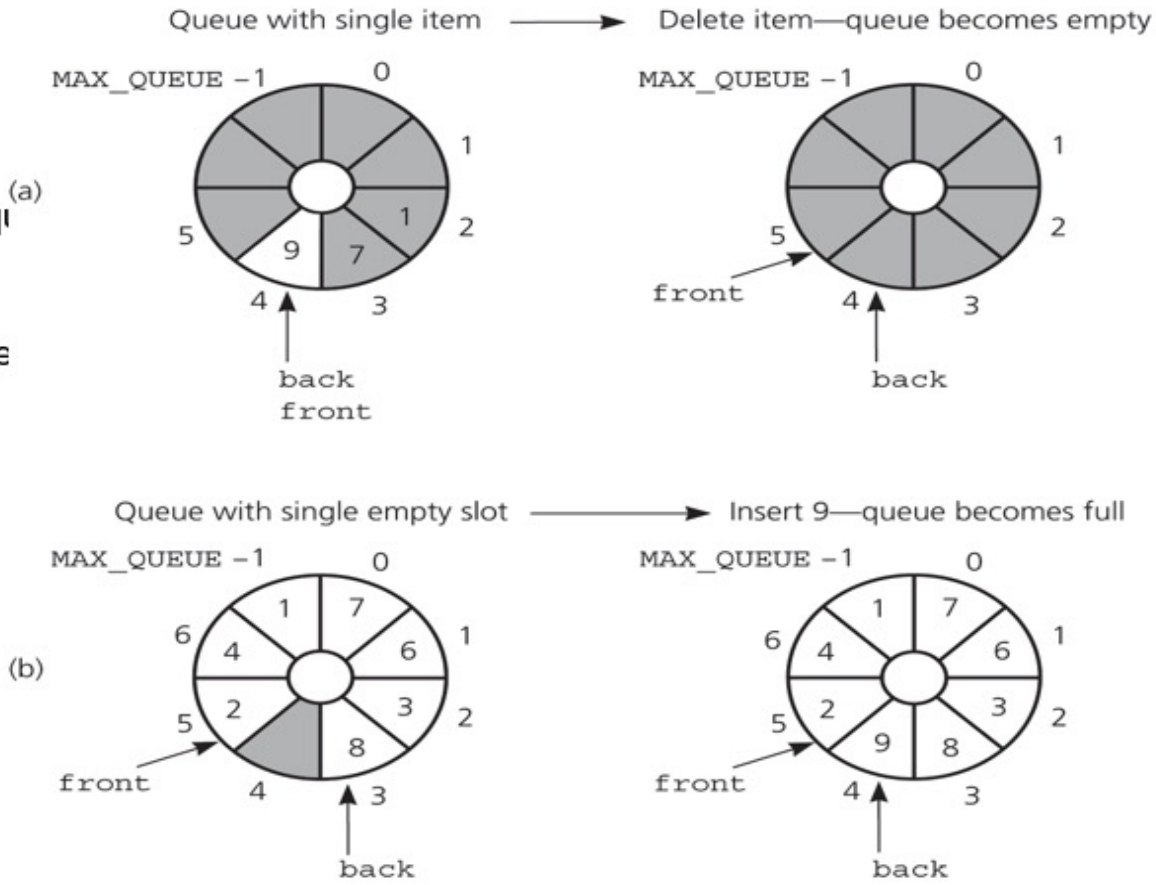
```
front = ( front + 1 ) % MAX_QUEUE;
--count;
```

After delete 20, 45 and 51 sequentially from circular queue



Circular Array Implementation

(a) *front* passes *back* when the qu
becomes empty;
(b) *back* catches up to *front* whe
the queue becomes full



Next.....
Queue implementation Linked List