Data Structures

QUEUES

- A queue is an ordered collection of items from which items may be deleted at one end (called the *front* of the queue) and into which items may be inserted at the other end (called the *rear* of the queue)

Definitions

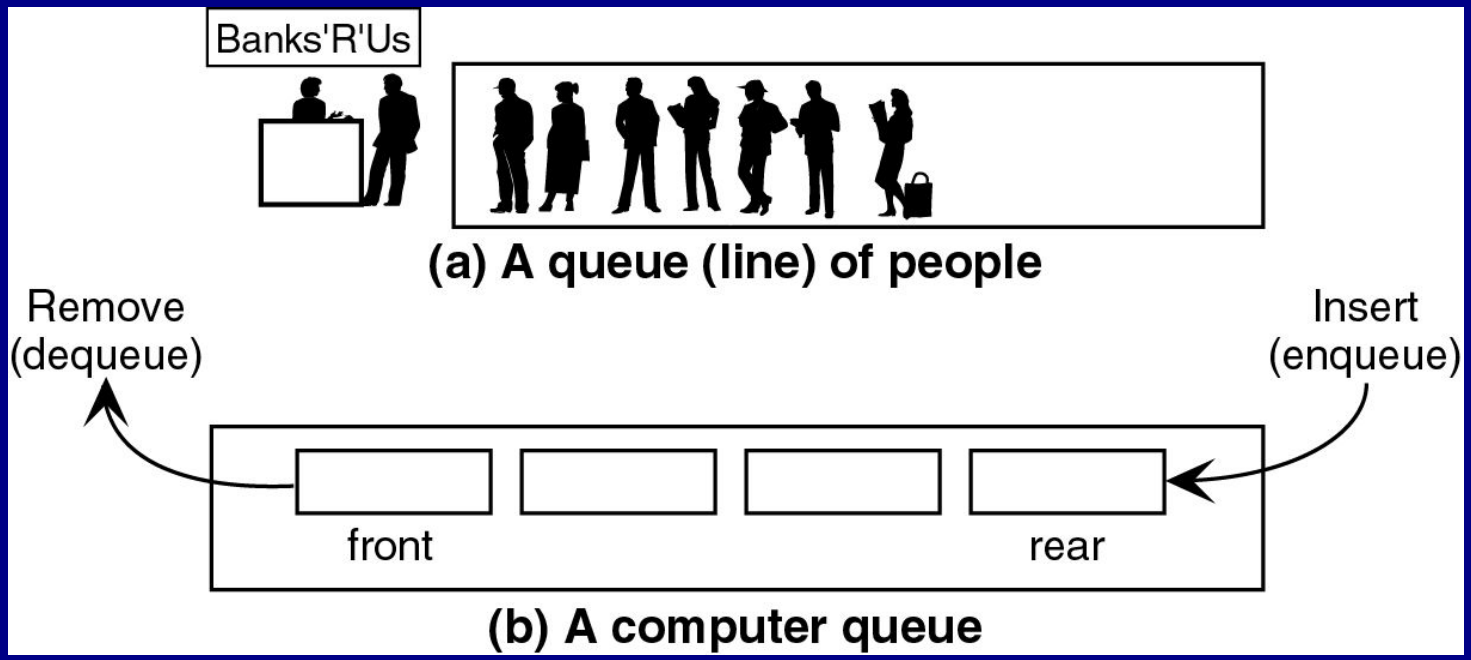
- Data structure which implements a first-in, first-out list; e.g. print queue, which contains a list of jobs to be printed in order.
- In *programming*, a queue is a *data structure* in which elements are removed in the same order they were entered. This is often referred to as FIFO (first in, first out).
- In contrast, a *stack* is a data structure in which elements are removed in the reverse order from which they were entered. This is referred to as LIFO (last in, first out).

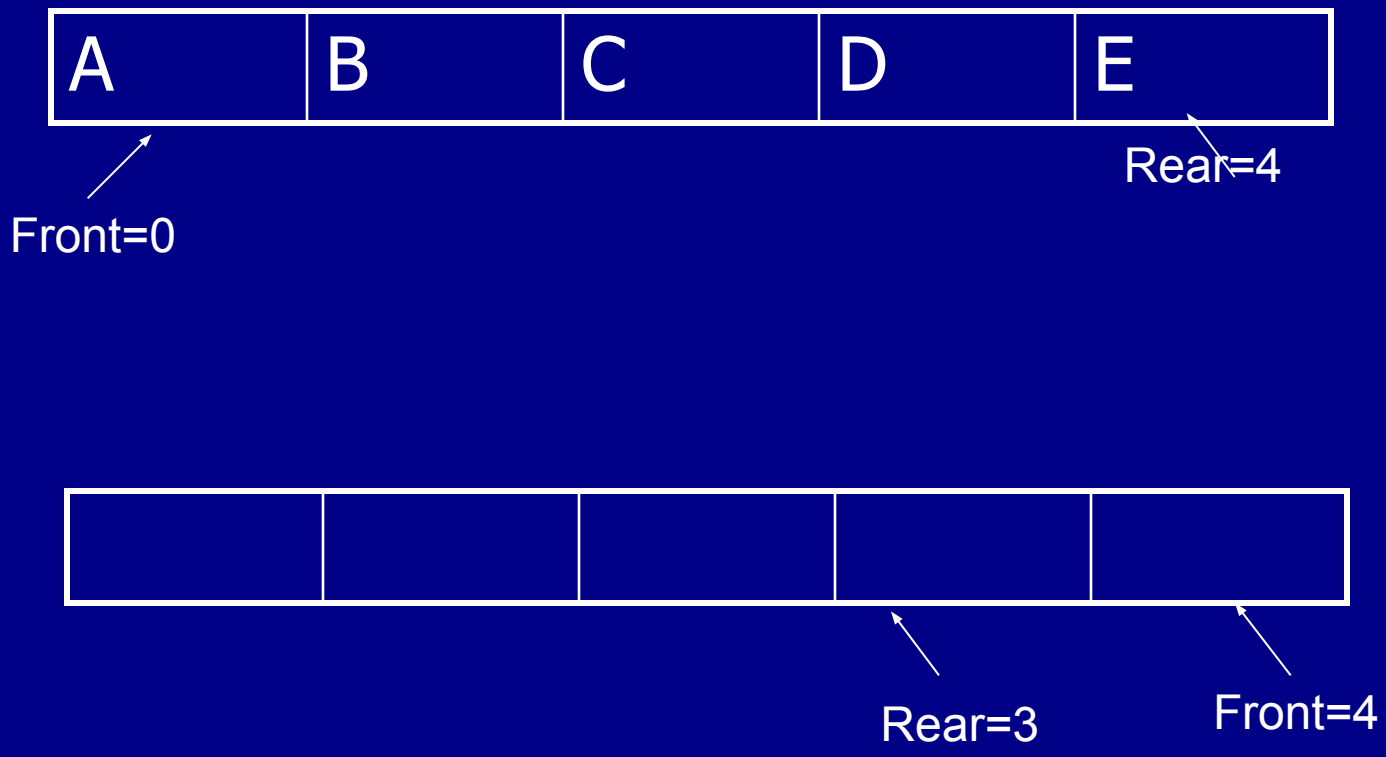
Introduction to Queues

- The queue data structure is very similar to the stack.
- In a stack, all insertions and deletions occur at one end, the top, of the list.
- In the queue, as in the stack, all deletions occur at the head of the list.
- However, all insertions to the queue occur at the tail of the list.

Introduction to Queues

- Basically, data enters the queue at one end and exits at the other end.




A
B
C
D
E
Front=0
Rear=4
Rear=3
Front=4

Introduction to Queues

- You get in line at the end and get serviced when you get to the front of the line.
- This characteristic gives a queue its first-in, first-out (FIFO) behavior.

Applications

- Ticketing counter
- Bus stop line
- Bank Customers
- Printer SPOOL
- CPU Scheduling

Queue Operations

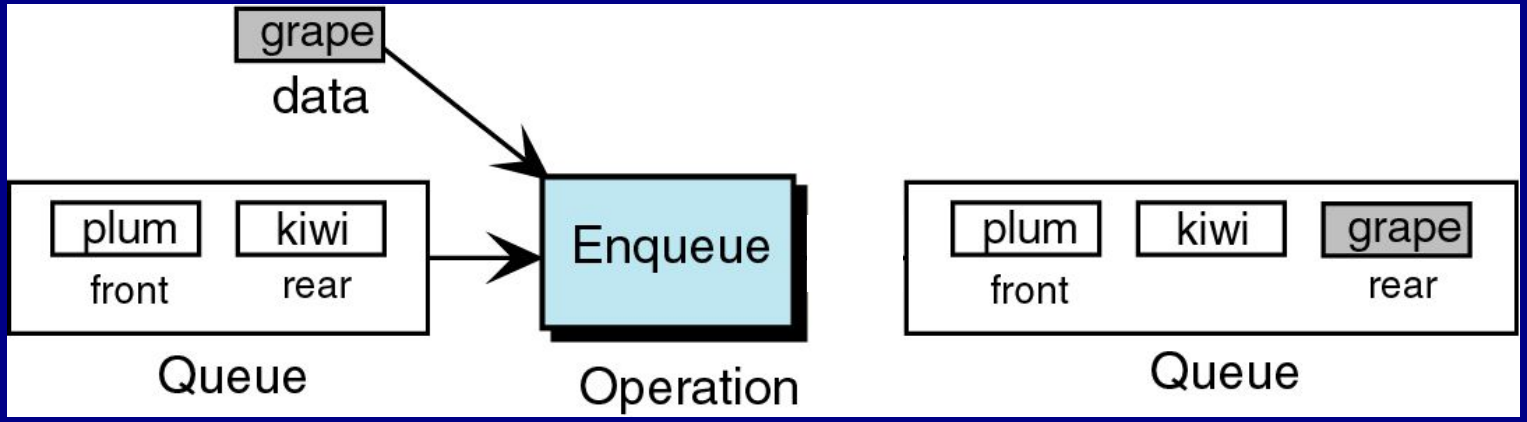
- Initialize the queue, Q, to be the empty queue.
- Determine whether or not if the queue Q is empty.
- Determine whether or not if the queue Q is full.
- Insert (enqueue) a new item onto the rear of the queue Q.
- Remove (dequeue) an item from the front of Q, provided Q is nonempty.

Queue Operations

- **isEmpty()** – check to see if the queue is empty.
- **isFull()** – check to see if the queue is full.
- **enqueue(element)** - put the element at the end of the queue.
- **dequeue()** – take the first element from the queue.
- **first()** – return the first element in the queue without removing it.

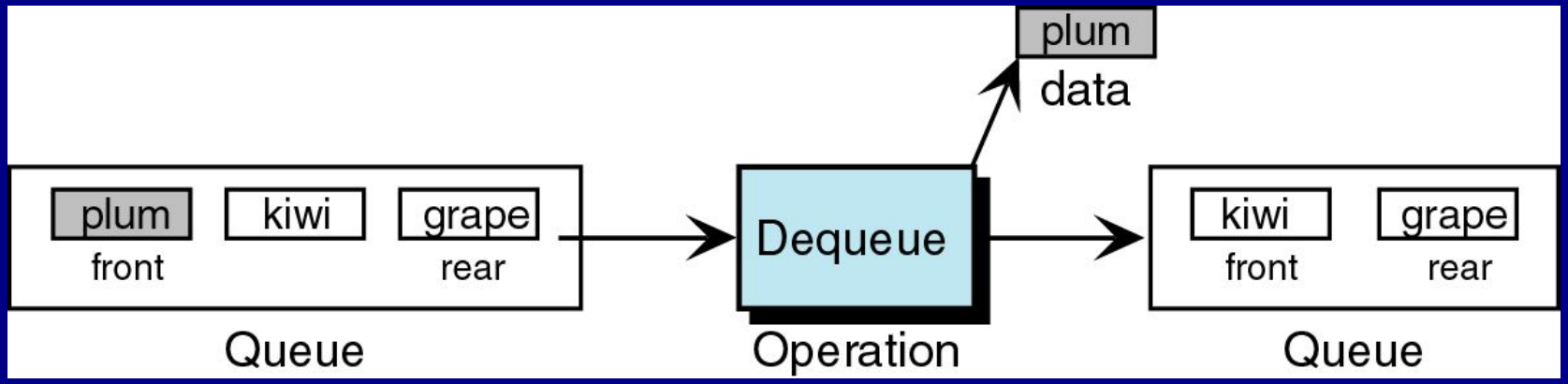
Enqueue

- The queue insert is known as ***enqueue***.
- After the data has been inserted, this new element becomes the rear of the queue.



Dequeue

- The queue delete operation is known as ***dequeue***.
- The data at the front of the queue is returned to the user and deleted from the queue.



Types of Queue

- Ordinary Queue
- Circular Queue
- Double Ended Queue(DQueue)
- Priority Queue

Array Implementation

Any implementation of a queue requires:
storage for the data as well as
markers ("pointers") for the front and for the back of the queue.

An ***array-based*** implementation would need structures like
`items`, an array to store the elements of the queue
`Front`, an index to track the front queue element
`Rear`, an index to track the position *following* last queue element

`Additions` to the queue would result in incrementing ***Rear***.
`Deletions` from the queue would result in incrementing ***Front***.
Clearly, we'd run out of space soon!

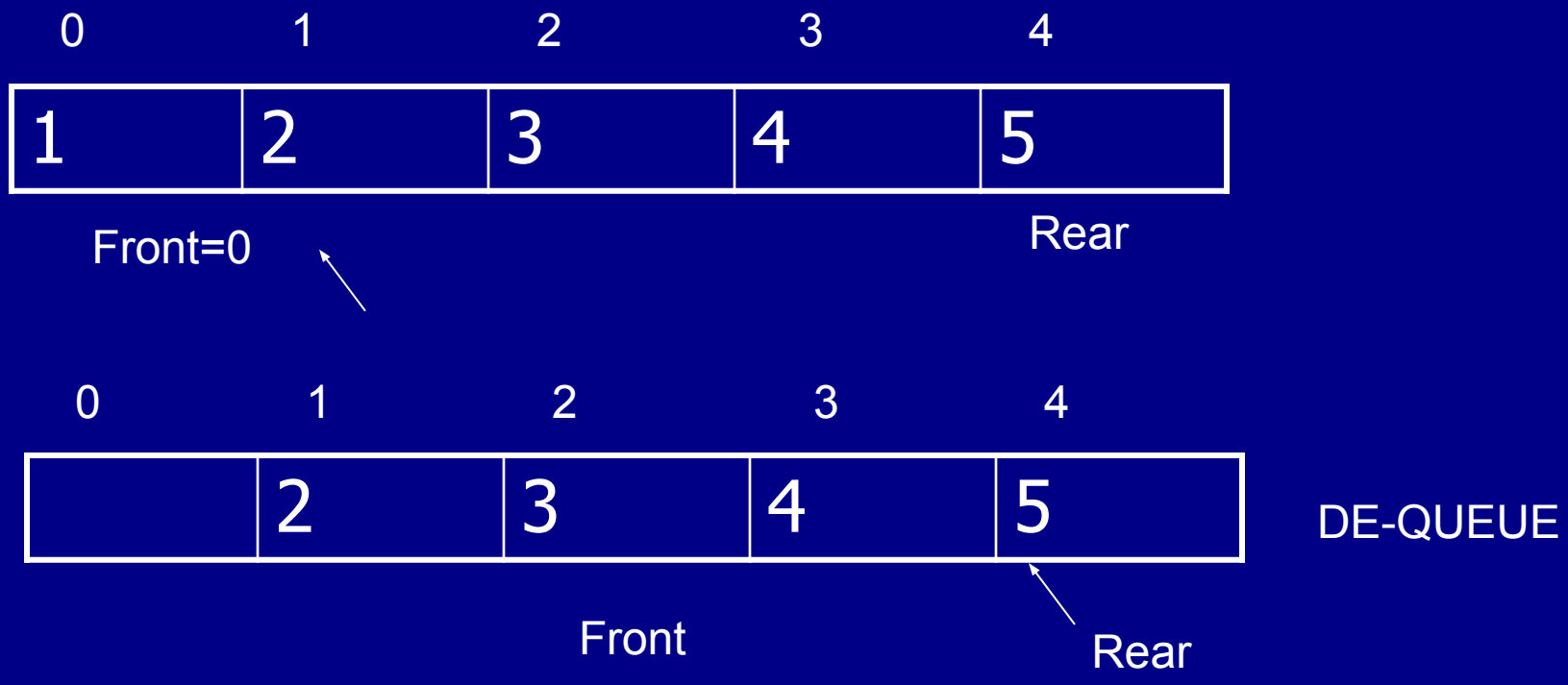
Queue using Arrays

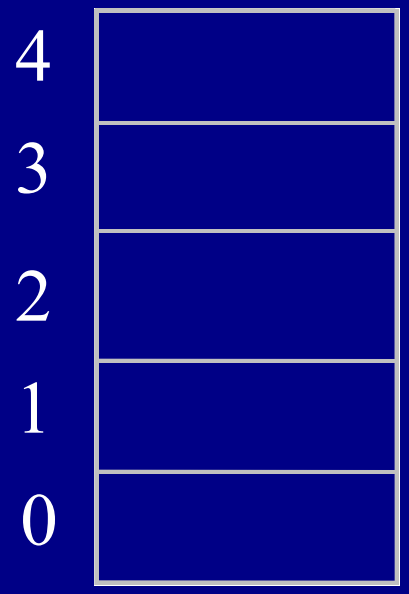
```
#define SIZE 10
Struct queue
{
    int items[SIZE];
    int front, rear;
}
```

- Insert(q,x)
  `q.items[++q.rear]=x;`

- X=remove(q)
  `x=q.items[q.front++];`

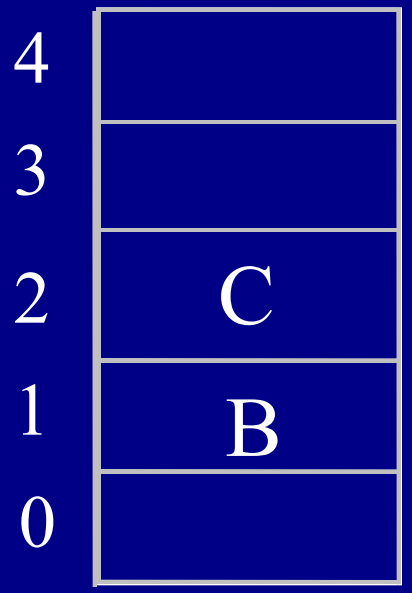
Dequeue





q.front=0

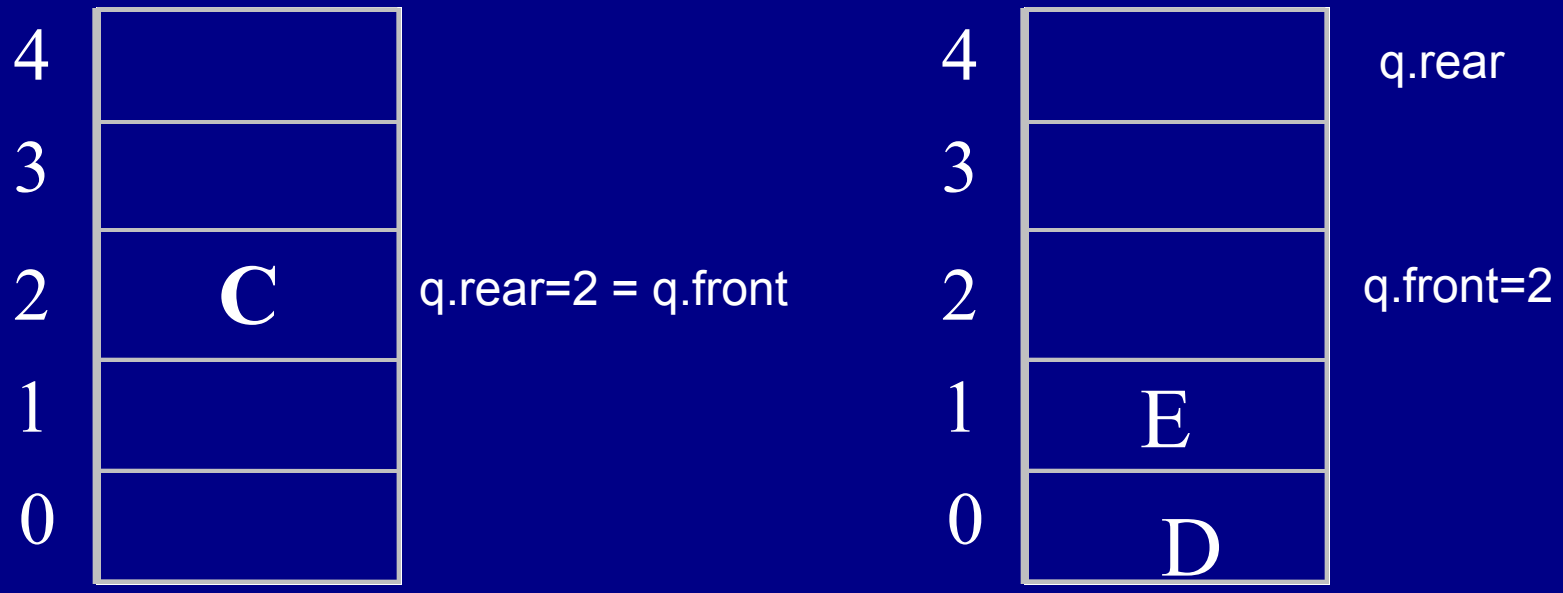
q.rear=-1



q.rear=2

q.front=1

- The queue is empty whenever q.rear<q.front
- The number of elements in the queue at any time is equal to the value of ***q.rear –q.front +1***



- Now there are 3 elements the queue but there is room for 5
- If to insert *F* in the queue the q.rear must be increased by 1 to 5 and q. items[5] must be set to the value F. but q.items is an array of only 5 elements so this insertion cannot be made

Solution to Problem

Clearly, we'have run out of space

Solutions include:

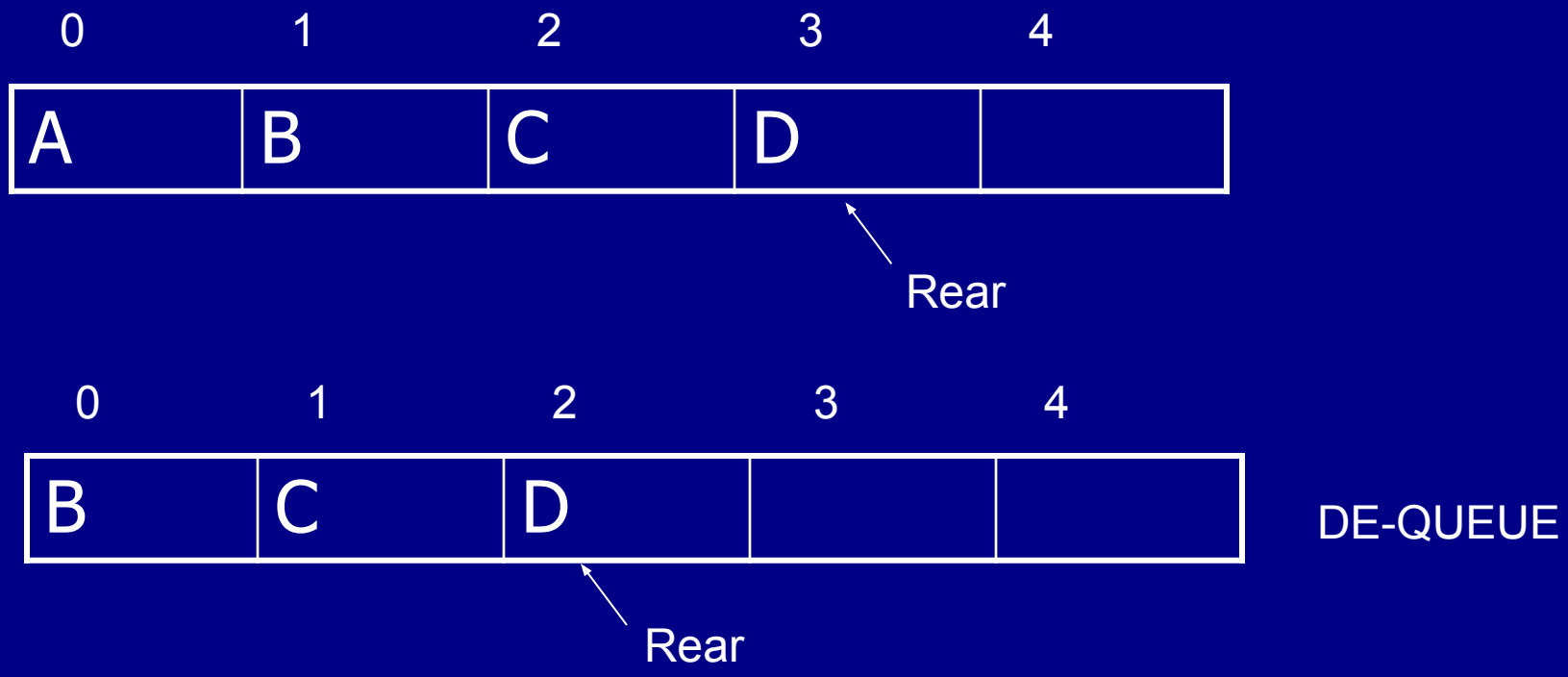
- Shifting the elements downward with each deletion

- Viewing array as a circular buffer, i.e. wrapping the end to the front

Solution 1

- For arrays there are two methods
- First is do it as we do in real world
  - Check if array is not empty
  - Simply dequeue from the first location of array say array[0] i.e. the zeroth index
  - After dequeue shift all the elements of the array from array[index] to array[index-1]

Dequeue



Dequeue Operation

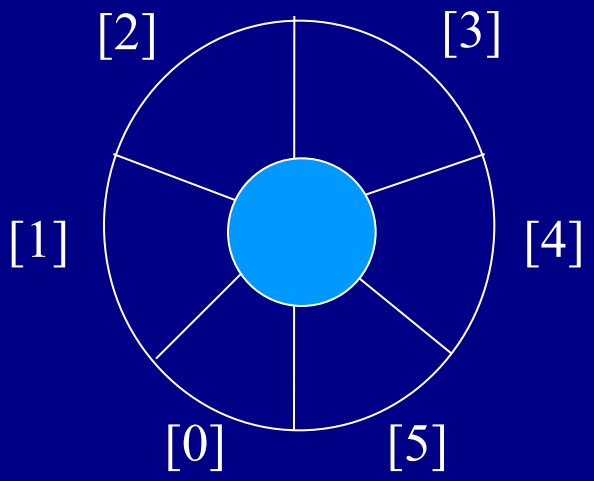
```
x=q.items[0];
for(i=0; i<q.rear; i++)
    q.items[i]=q.items[i+1]
q.rear--;
```

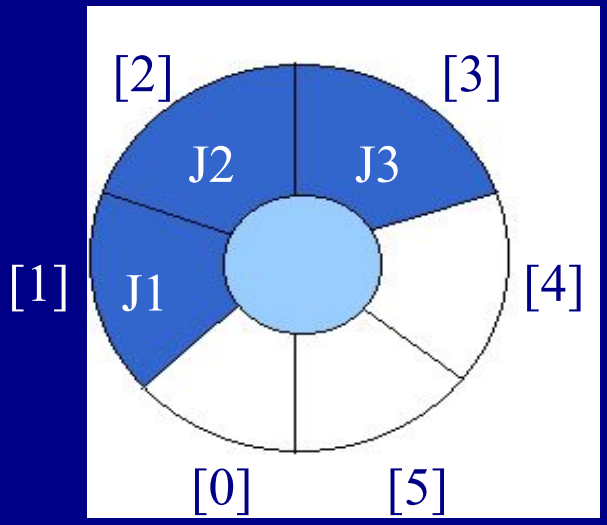
- Method is costly as we have to move all the elements of the array
- Do we need Front Index in this case?
  - No, Because we are always dequeue(ing) from the first index

Solution2: Circular Queue

- To avoid the costly operation shifting all the elements again we employ another method called circular queue
- Simply dequeue the element and move the front pointer to next index

Circular Queue





Can be seen as a circular queue

Problems with above solution

- How to know if the queue is empty
- How to know if the queue is full
- What is the relation between front and back when queue is full?

- Solution

Keep it simple… add counter to the queue

Deque

- It is a double-ended queue.
- Items can be inserted and deleted from either ends.
- More versatile data structure than stack or queue.
- E.g. policy-based application (e.g. low priority go to the end, high go to the front)

Priority Queues

- More specialized data structure.
- Similar to Queue, having front and rear.
- Items are removed from the front.
- Items are ordered by key value so that the item with the lowest key (or highest) is always at the front.
- Items are inserted in proper position to maintain the order.

Basic methods of implementing Priority Queue

- Method 1: Insert in any order and delete min/max element. This requires shifting after deletion in array representation to fill up the hole created.

- Method 2: Insert By order and delete front most item as the highest prioritized item will be at the front of the queue. This method requires shifting during insertion

Both of the above discussed methods are not efficient as shifting is costlier in terms of time.

Implementation of the above methods are discussed in class.

Note: Efficient methods of implementing priority queue using Binary Search Tree and Heap data structures will be discussed in unit IV of the course.