AAST

# بسم الله الرحمن الرحيم

# Lecture 10: Queues

# Queues cont'd

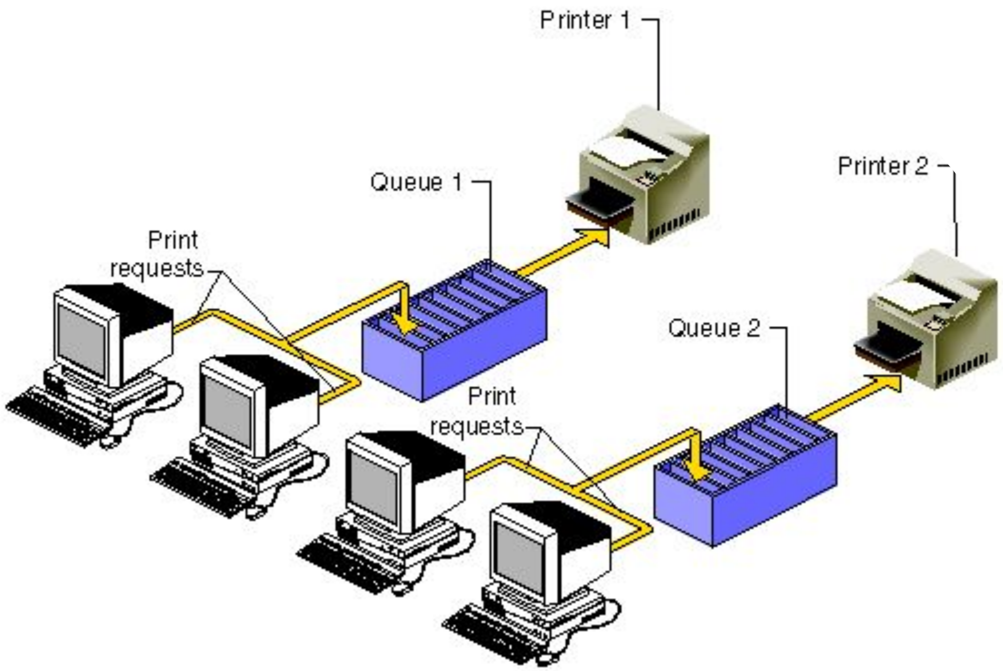
AAST

- What is a Queue ?
- Examples of Queues
- Design of a Queue
- Different Implementations of the Queue

# What is a Queue ?

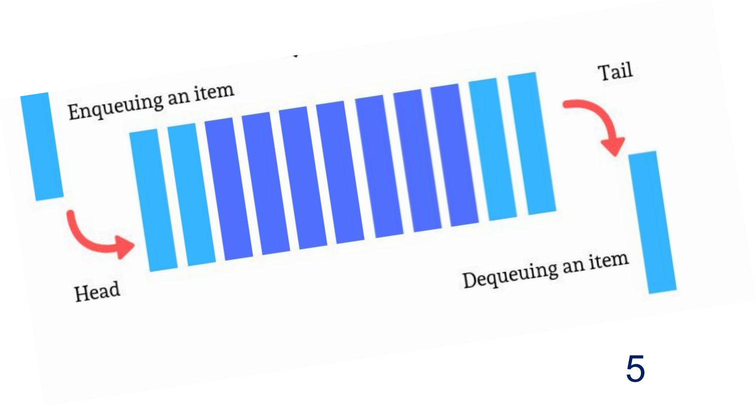
- Linear list.
- One end is called front.
- Other end is called rear.
- Additions are done at the rear only.
- Removals are made from the front only.

# Queues

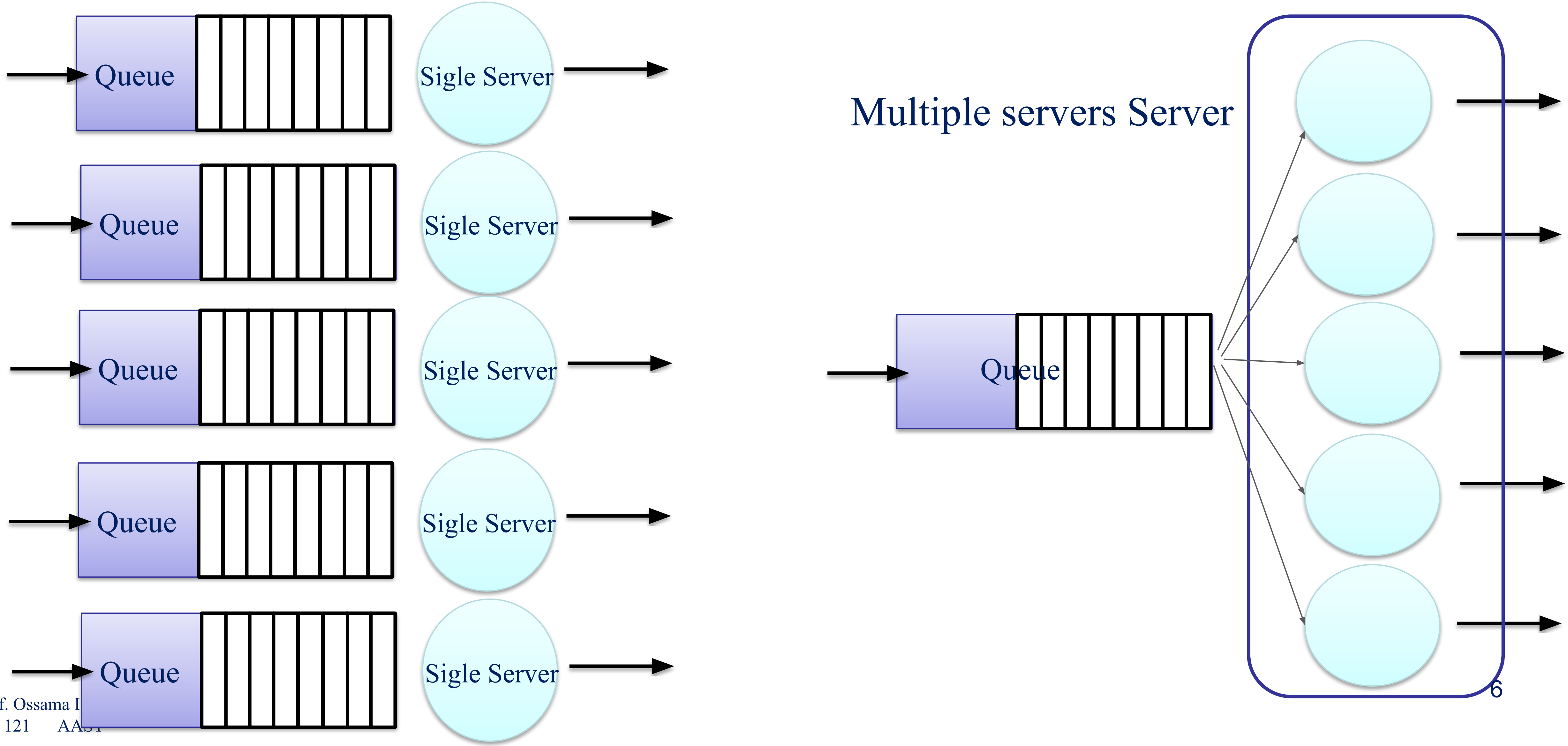




rear

front



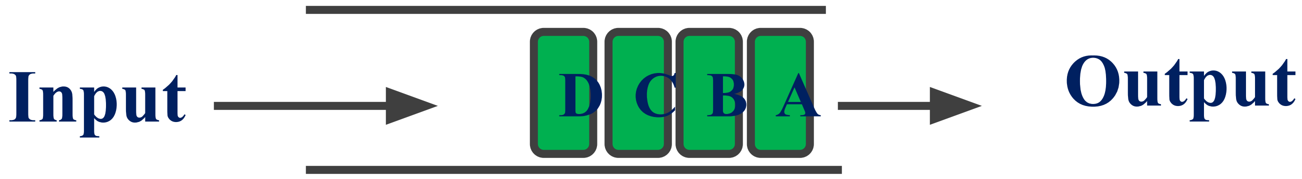
# ?Which is faster



# What is a Queue ?

AAST

A queue is an ordered collection of items where the addition of new items happens at one end, called the "rear," and the removal of existing items occurs at the other end, commonly called the "front." As an element enters the queue it starts at the rear and makes its way toward the front, waiting until that time when it is the next element to be removed.




Queue: *First In First Out (FIFO)*

# What is a Queue? Cont'd

A ***queue*** is a linear abstract data type such that insertions are made at one end, called the rear, and removals are made at the other end, called the front.

Queues are sometimes called **FIFOs**: first-in first-out.

**enqueue()  Queue  dequeue()**

The two basic operations are:
enqueue: adds an element to the rear of the queue.
dequeue: removes and returns the element at the front of the **queue.**

Queues are used extensively in operating systems
Queues of processes, I/O requests, and much more

# What is a Queue? Cont'd

- A queue is open at two ends.
- You can only
  - add entry (**enqueue**) at the rear
  - delete entry (**dequeue**) at the front

# Examples : Queue

Examples:

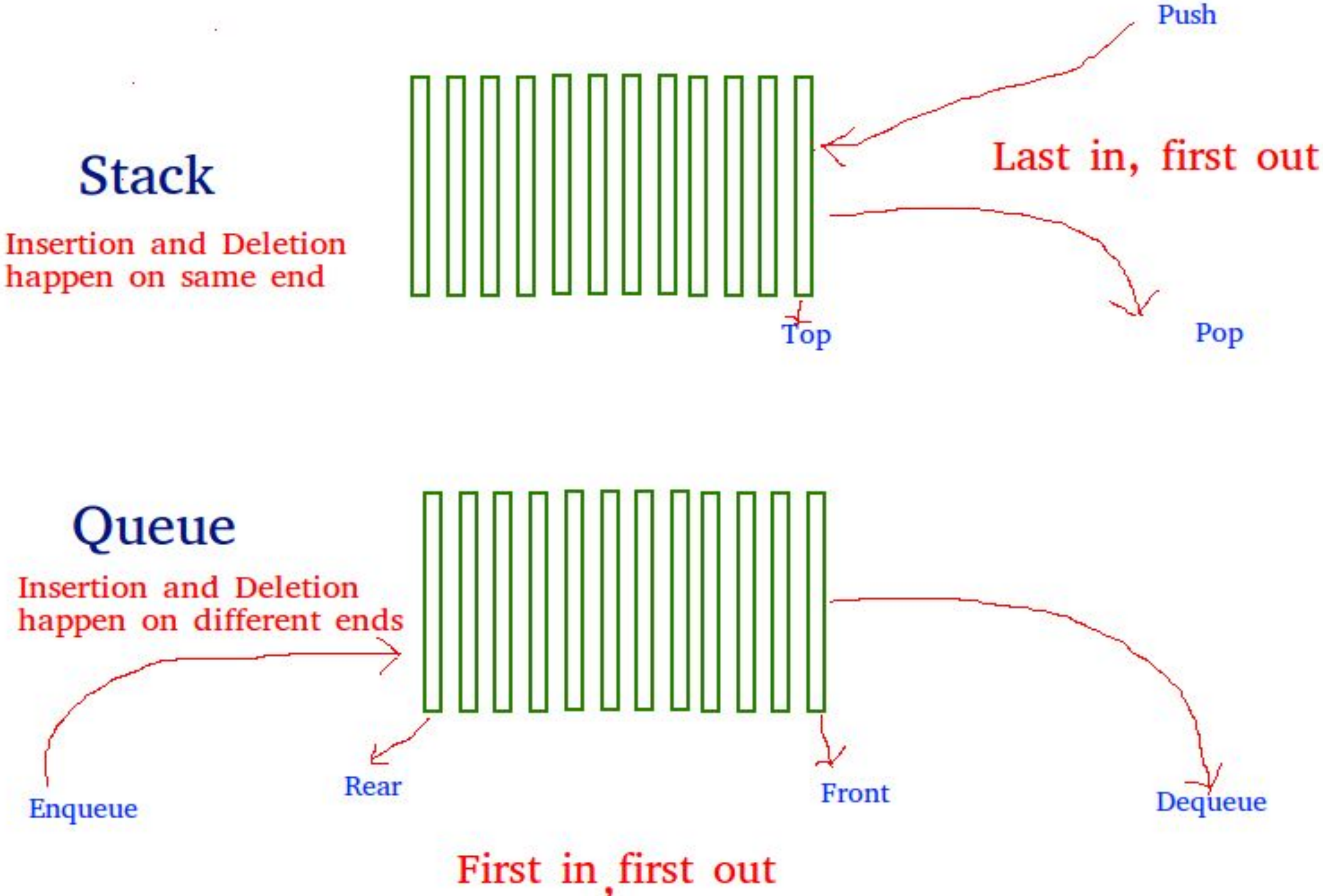
- Toll Station

  Car comes, pays, leaves
- Check-out in Big Y market

  Customer comes, checks out and leaves
- Bank, ATM

More examples: Printer, Office Hours, …

# Differences between Stacks and Queues



# Revisit Of Stack Applications

- Applications in which the stack cannot be replaced with a queue.
  - Parentheses matching.
  - Towers of Hanoi.
  - Switchbox routing.
  - Method invocation and return.
  - Try-catch-throw implementation.
- Application in which the stack may be replaced with a queue.
  - Rat in a maze.
    - Results in finding shortest path to exit.

# Uses of Queues

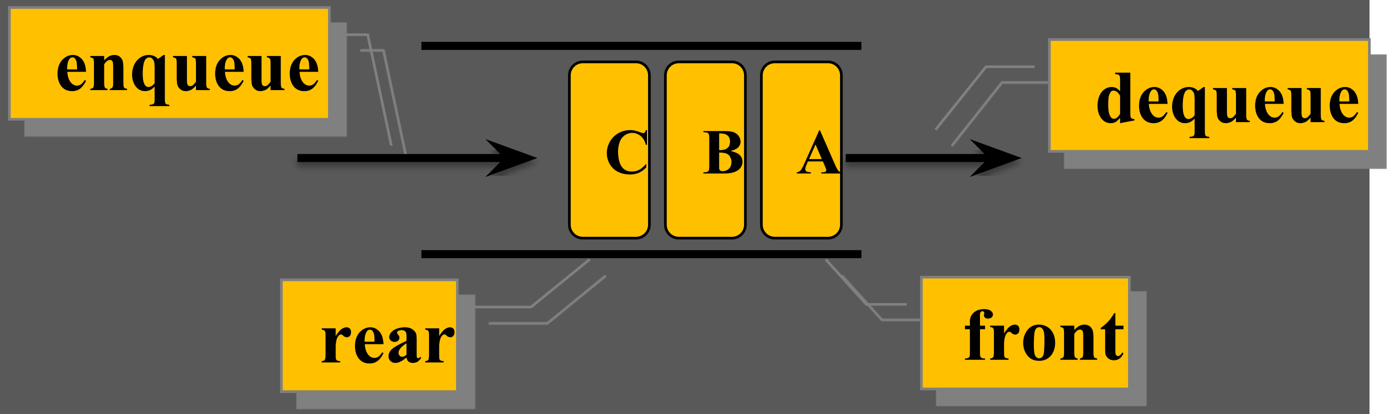
AAST

## Queues are used for:

- Direct applications - Waiting lists,
- Shared resources management (system programming):
  - Access to the processor;
  - Access to the peripherals such as disks and printers.
- Application programs:
  - Simulations;
  - Generating sequences of increasing length over a finite size alphabet;
  - Navigating through a maze.

# Queue Operations

- Queue
  - Operating on both ends
  - Operations: EnQueue(in), DeQueue(out)



# Queue Operations

add(Object item)

  a.k.a. enqueue(Object item)

Object get()

  a.k.a. Object front()

Object remove()

  a.k.a. Object dequeue()

boolean isEmpty()

Specify in an interface, allow varied implementations

# Implementing a Queue Using Arrays

# Array Representation of Queues

- Queues can be easily represented using arrays.
- Every queue has front and rear variables that point to the position from where deletions and insertions can be done, respectively.
- Consider the queue shown in figure

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 12 | 9 | 7 | 18 | 14 | 36 | | | | |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

- Here, front = 0 and rear = 5.
- If we want to add one more value in the list say with value 45, then rear would be incremented by 1 and the value would be stored at the position pointed by rear.

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 12 | 9 | 7 | 18 | 14 | 36 | 45 | | | |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

# Array Representation of Queues

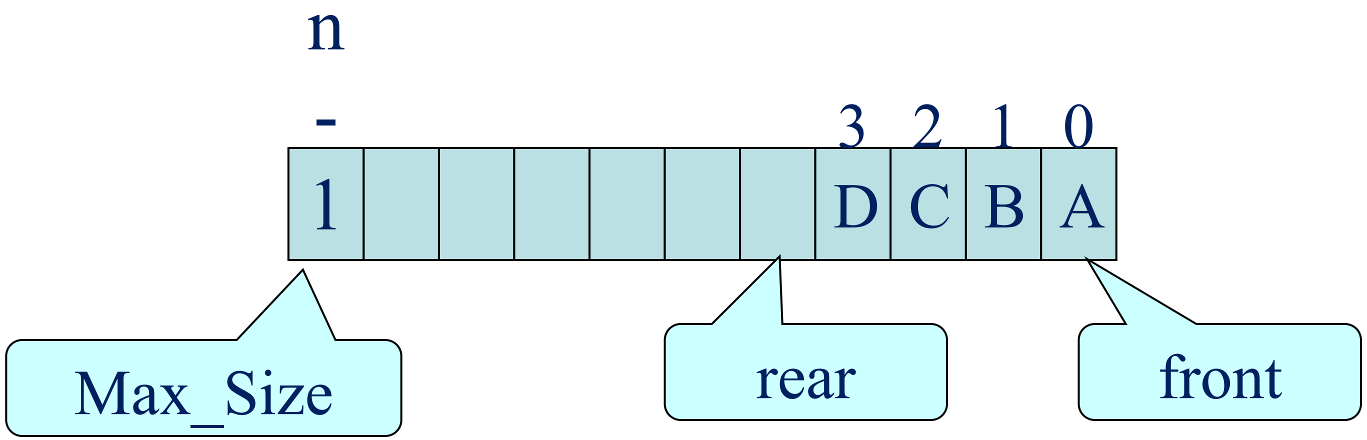
- **Now, front = 0 and rear = 6. Every time a new element has to be added, we will repeat the same procedure.**
- **Now, if we want to delete an element from the queue, then the value of front will be incremented. Deletions are done from only this end of the queue.**

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | 9 | 7 | 18 | 14 | 36 | 45 | | | |

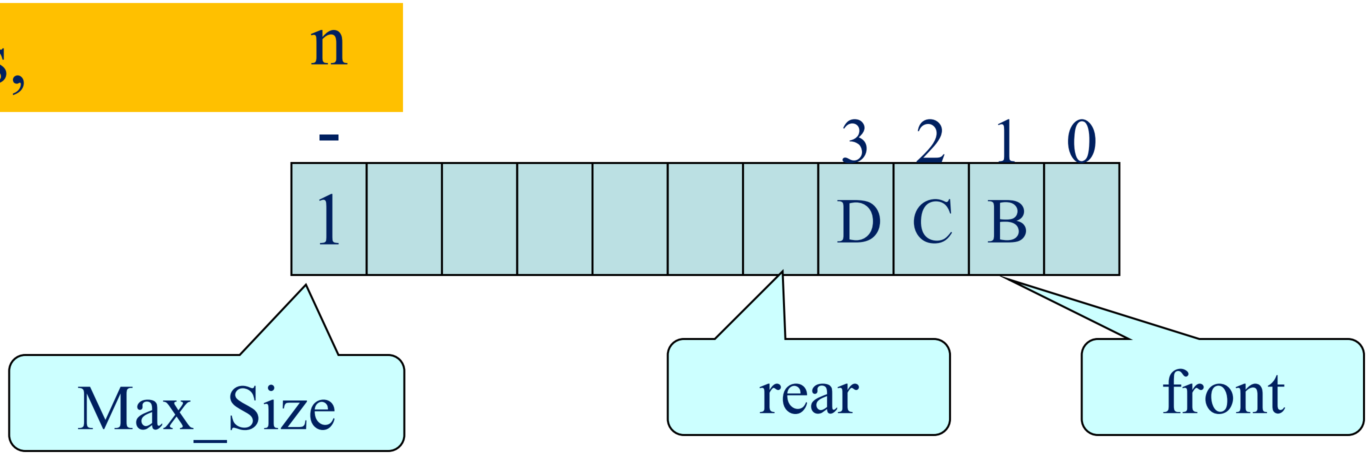
0 1 2 3 4 5 6 7 8 9

- **Now, front = 1 and rear = 6.**

# Array Representation of Queues



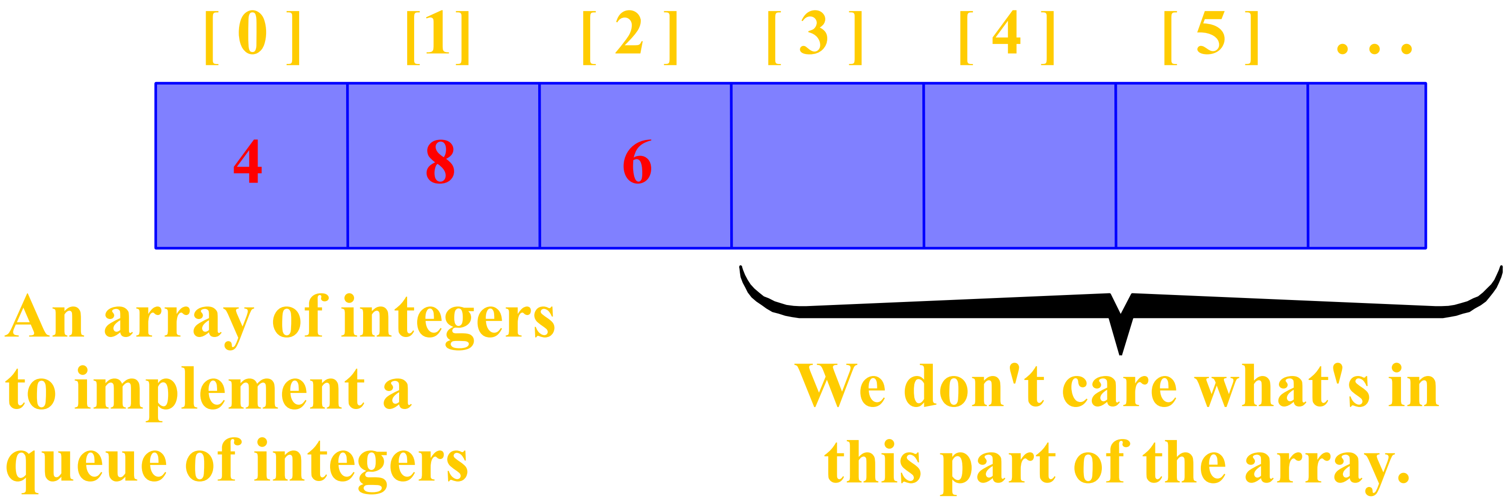

After A leaves,



# Array Implementation

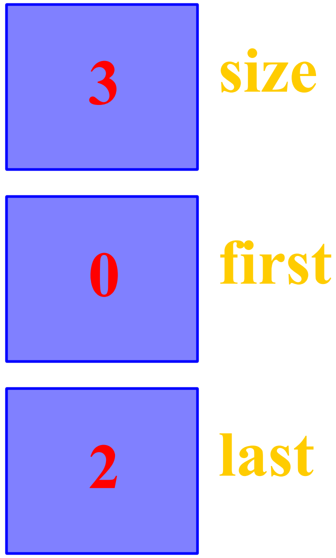
AAST

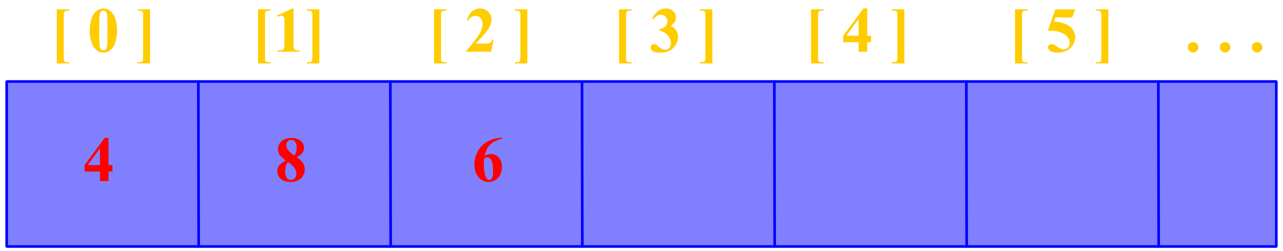
- A queue can be implemented with an array, as shown here. For example, this queue contains the integers 4 (at the front), 8 and 6 (at the rear).



# Array Implementation

- The easiest implementation also keeps track of the number of items in the queue and the index of the first element (at the front of the queue), the last element (at the rear).

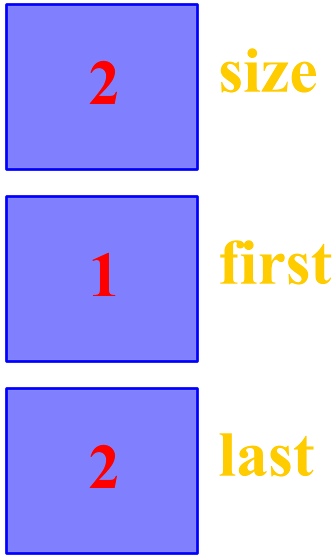


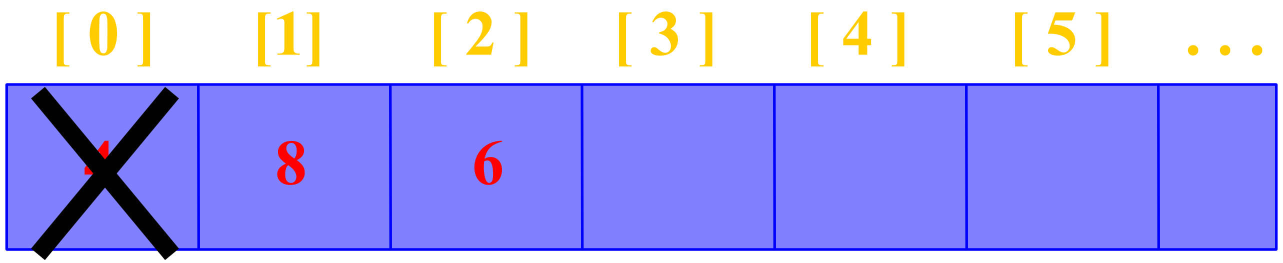



# A Dequeue Operation

AAST

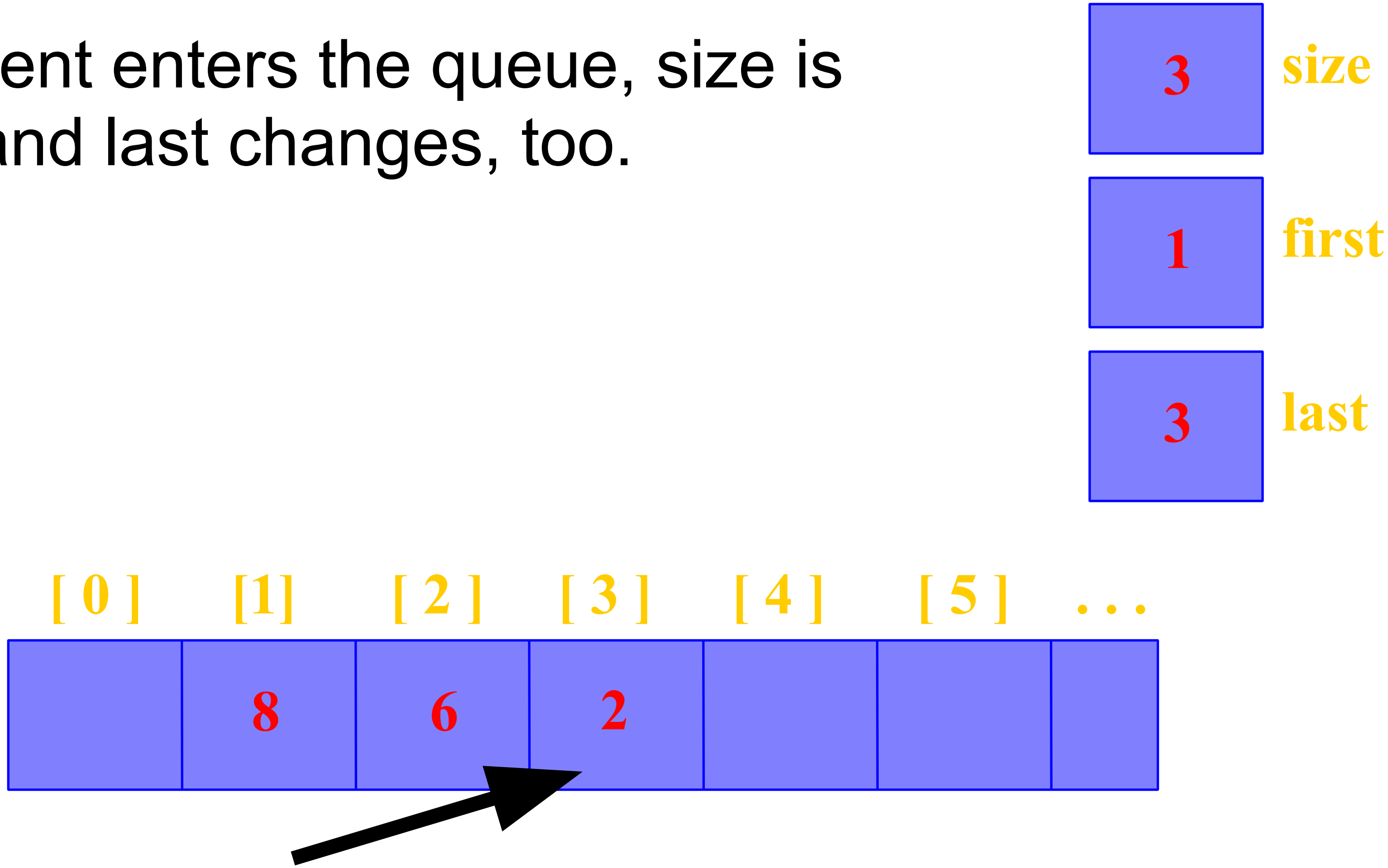
- When an element leaves the queue, size is decremented, and first changes, too.



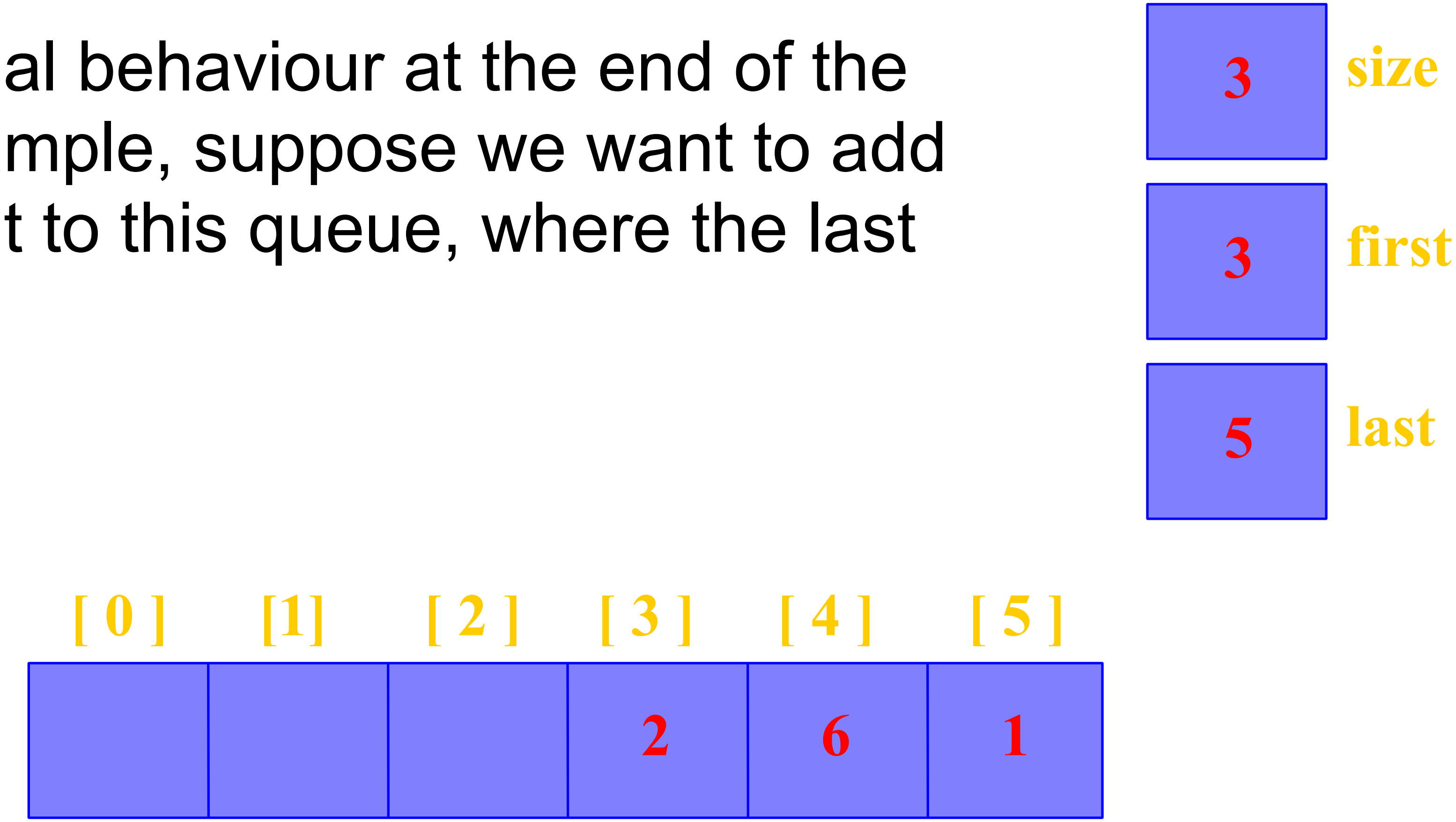
# An Enqueue Operation

- When an element enters the queue, size is incremented, and last changes, too.



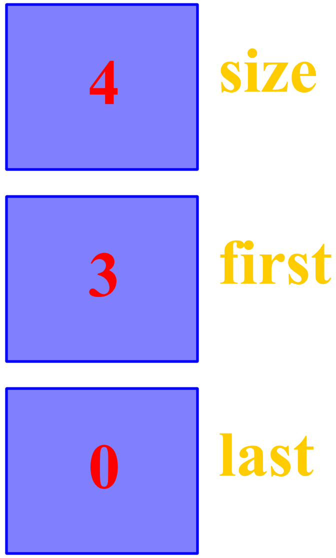
# At the End of the Array

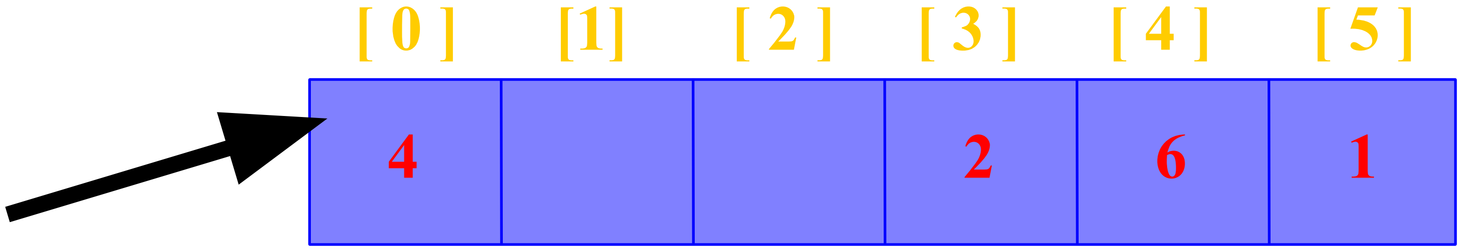
- There is special behaviour at the end of the array. For example, suppose we want to add a new element to this queue, where the last index is [5]:



# At the End of the Array

- The new element goes at the front of the array (if that spot isn't already used):

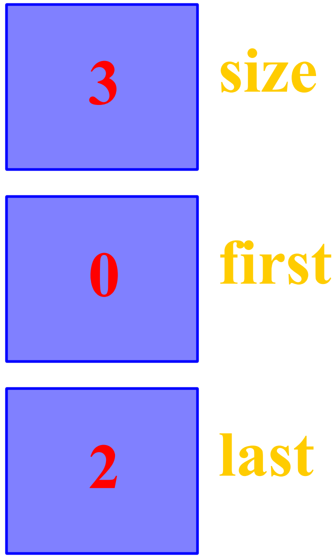


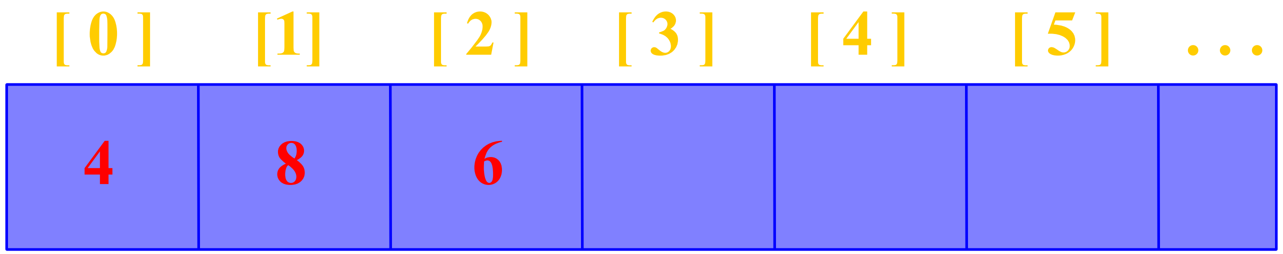



# Array Implementation



- Easy to implement
- But it has a limited capacity with a fixed array
- Or you must use a dynamic array for an unbounded capacity
- Special behaviour is needed when the rear reaches the end of the array.

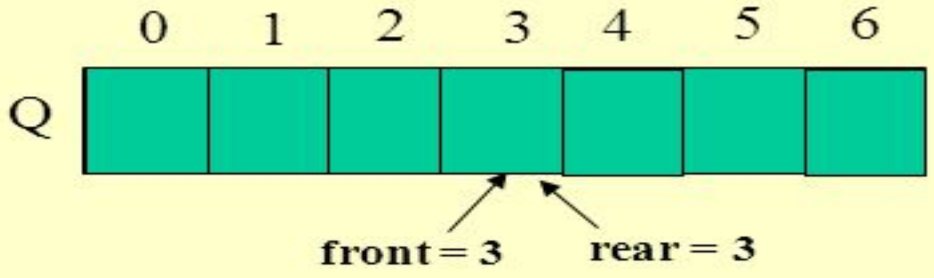


# Array Implementation of Queues

- We can implement a queue of at most "N" elements with an array int Q[N] and 3 variables: int front, int rear, int noItems as follows

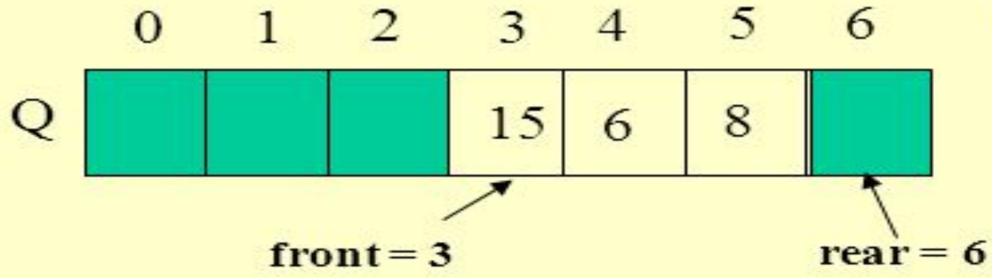
**An Empty Queue**




**noItems = 0**

- Front and rear are equal to each other and noItems = 0 in an empty Queue

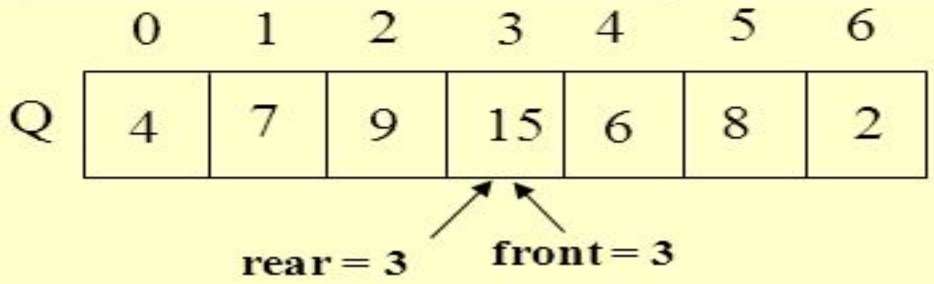
**A partially-full Queue**




**noItems = 3**

- Front points to the first element in the Queue
- Rear points to the next slot after the last element in the Queue

**A Full Queue**




**noItems = 7**

- Front and rear are equal to each other and noItems = N=7 in a full Queue

# Array Representation of Queues

- Before inserting an element in the queue we must check for overflow conditions.
- An overflow occurs when we try to insert an element into a queue that is already full, i.e. when rear = MAX – 1, where MAX specifies the maximum number of elements that the queue can hold.
- Similarly, before deleting an element from the queue, we must check for underflow condition.
- An underflow occurs when we try to delete an element from a queue that is already empty. If front = -1 and rear = -1, this means there is no element in the queue.

# Algorithm for Insertion Operation

AAST

```
Algorithm to insert an element in a queue
Step 1: IF REAR=MAX-1, then;
         Write OVERFLOW
           Goto Step 4
        [END OF IF]
Step 2: IF FRONT == -1 and REAR = -1, then
         SET FRONT = REAR = 0
        ELSE
         SET REAR = REAR + 1
        [END OF IF]
Step 3: SET QUEUE[REAR] = NUM
Step 4: Exit
```

**Time complexity: O(1)**

# Algorithm for Deletion Operation

AAST

```
Algorithm to delete an element from a queue
Step 1: IF FRONT = -1 OR FRONT > REAR, then
          Write UNDERFLOW
             Goto Step 2
         ELSE
          SET VAL = QUEUE[FRONT]
             SET FRONT = FRONT + 1
        [END OF IF]
Step 2: Exit
```

**Time complexity: O(1)**

# !! Problem when using array

An array has **limited size**, once rear is at the end of this array, and there is new item coming in, what can we do?

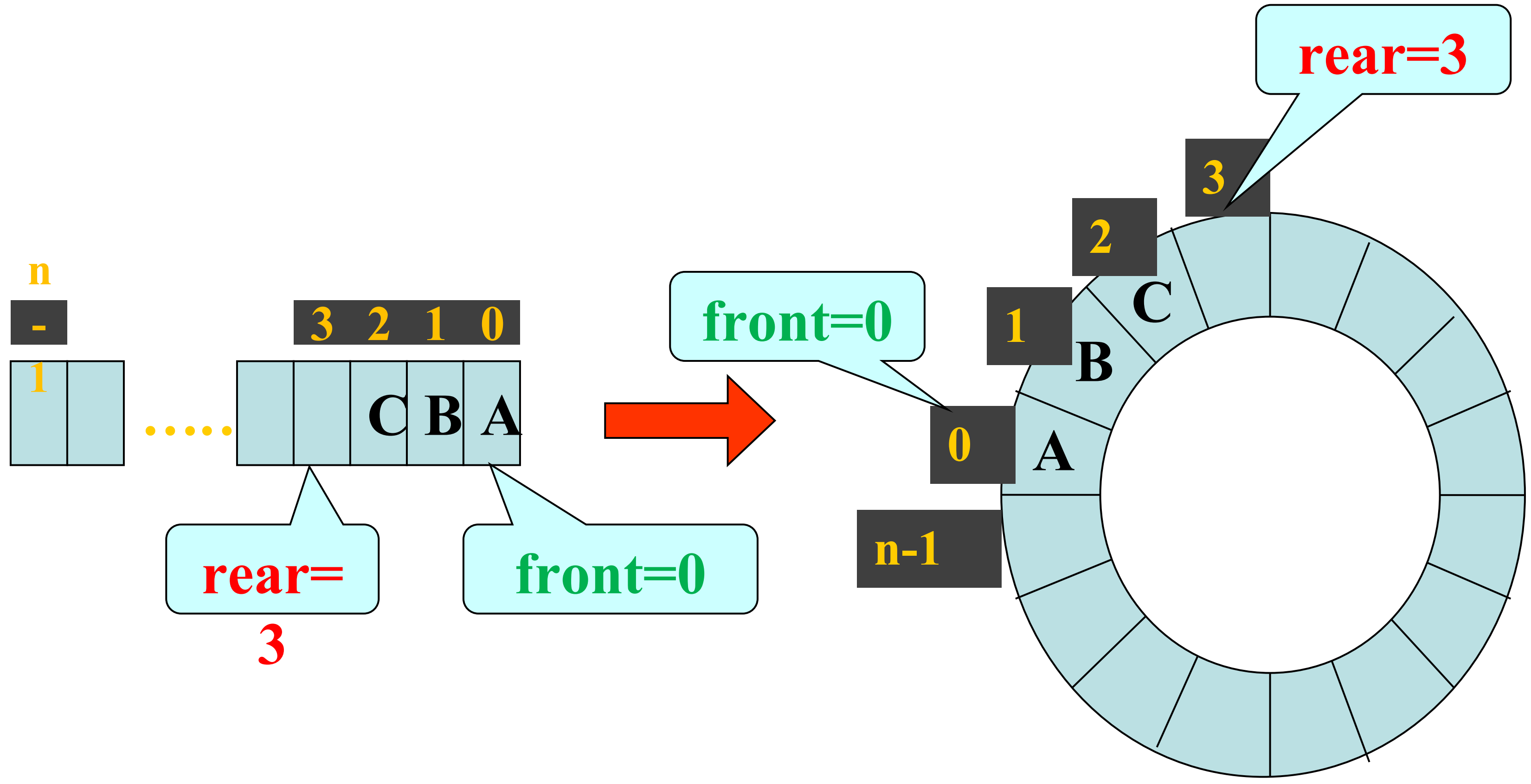
:There are two solutions

1. Shifting all items to front in the array when dequeue operation. ( **Too Costly…** )
2. Wrapped around array ---- **Circular Array**

# Implementing a Queue Using Circular Arrays

AAST

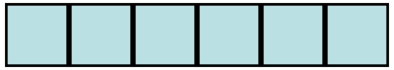
## Wrapped around array



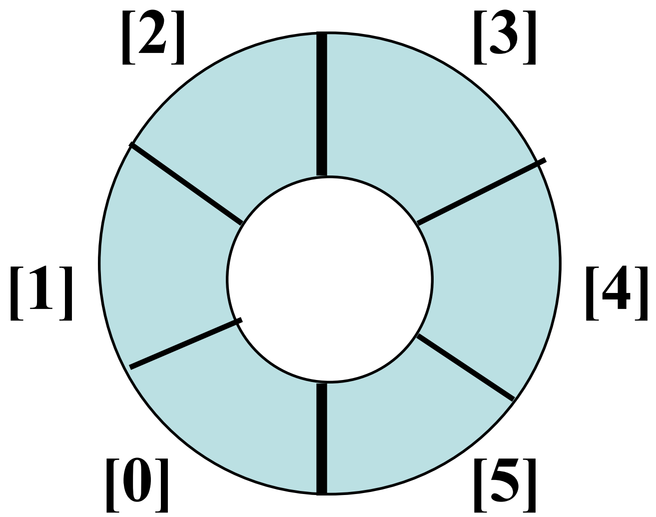
# Queue Implementation Using Circular Array

- Use a 1D array queue.

queue[]

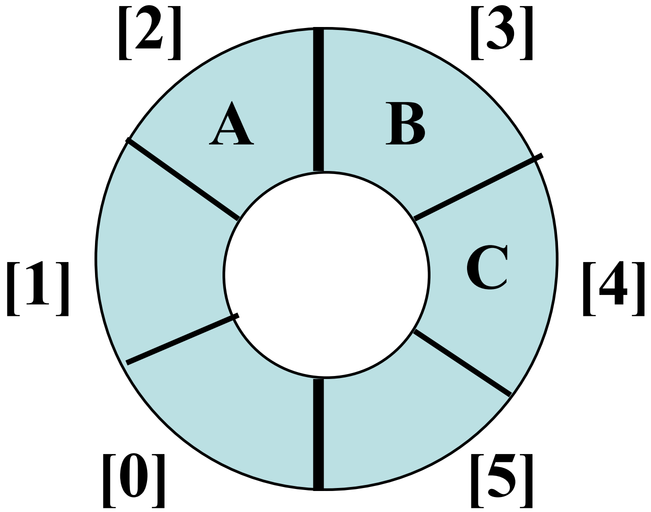


- **Circular view of array.**



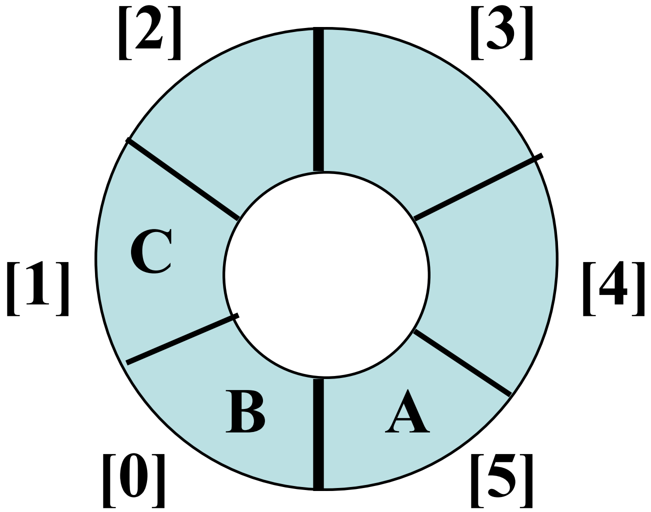
# Queue Implementation Using Circular Array

- **Possible configuration with 3 elements**



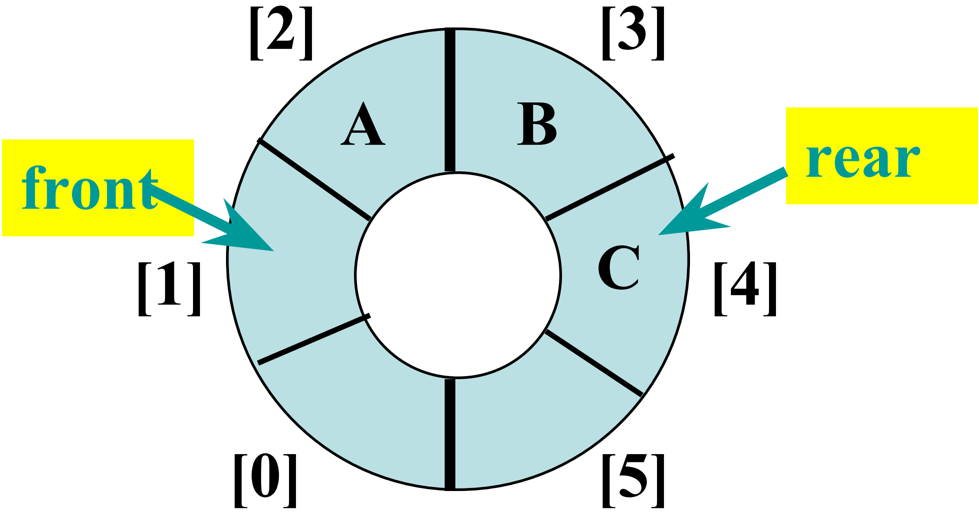
# Queue Implementation Using Circular Array

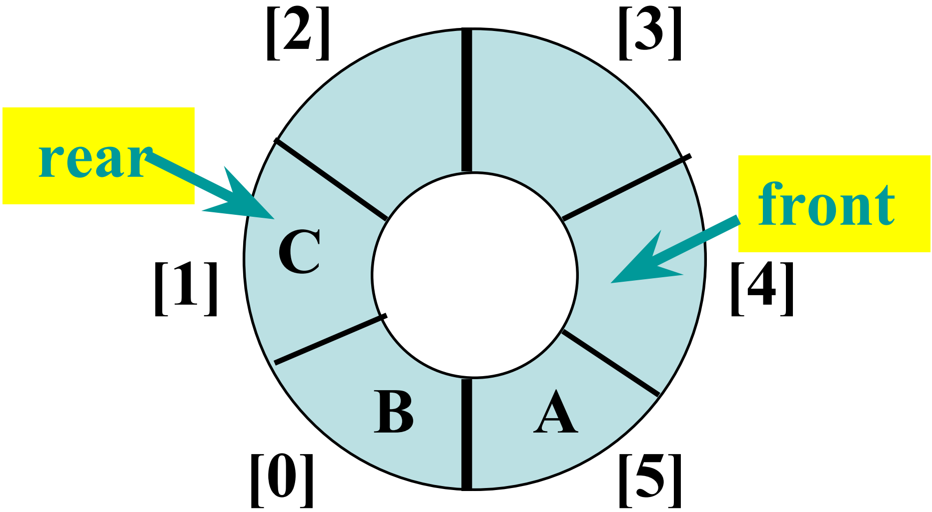
- **Another possible configuration with 3 elements**



# Queue Implementation Using Circular Array

- **Use integer variables front and rear.**
  - **front is one position counterclockwise from first element**
  - **rear gives position of last element**

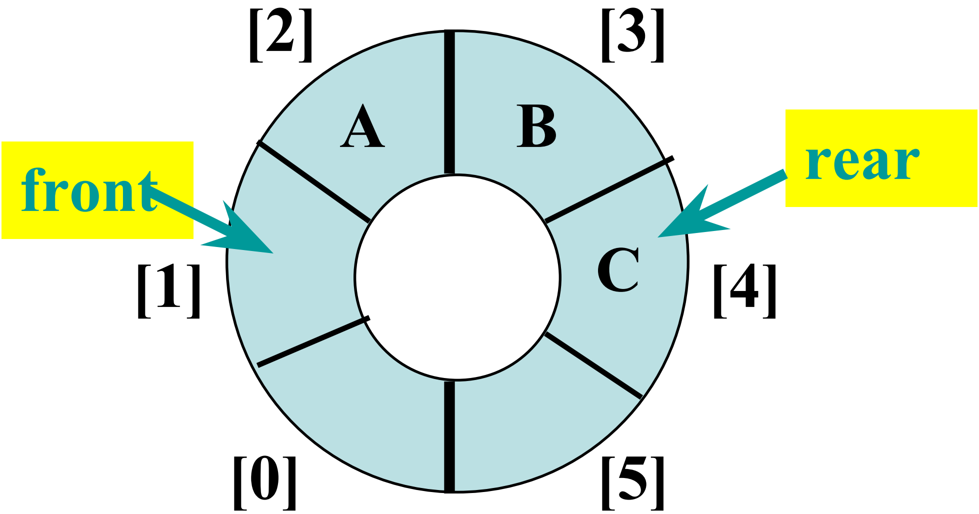


AAST

## Add an Element to the Queue

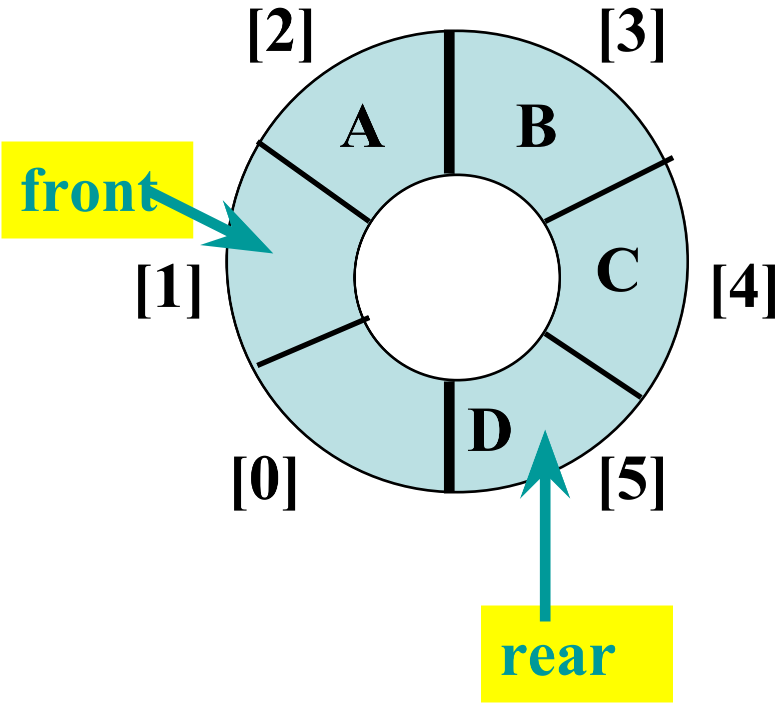
- **Move rear one clockwise**



AAST

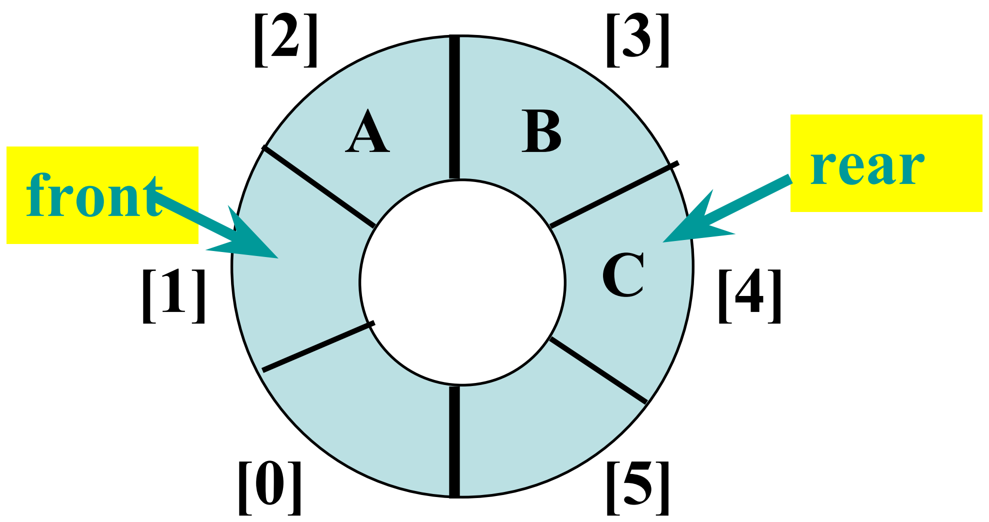
## Add an Element to the Queue

- **Move rear one clockwise.**
- **Then put into queue[rear].**



## Remove an Element from the Queue

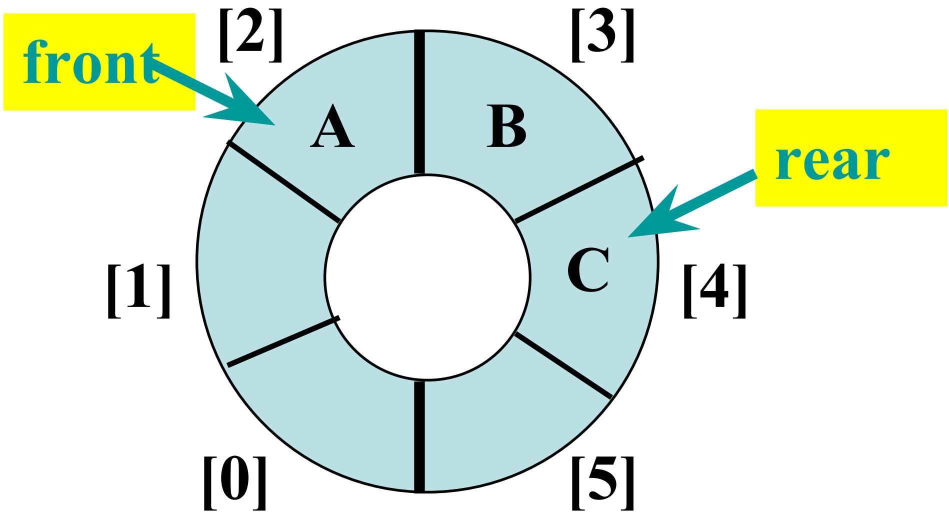
- **Move front one clockwise.**



AAST

## Remove an Element from the Queue

- Move front one clockwise.
- Then extract from queue[front].

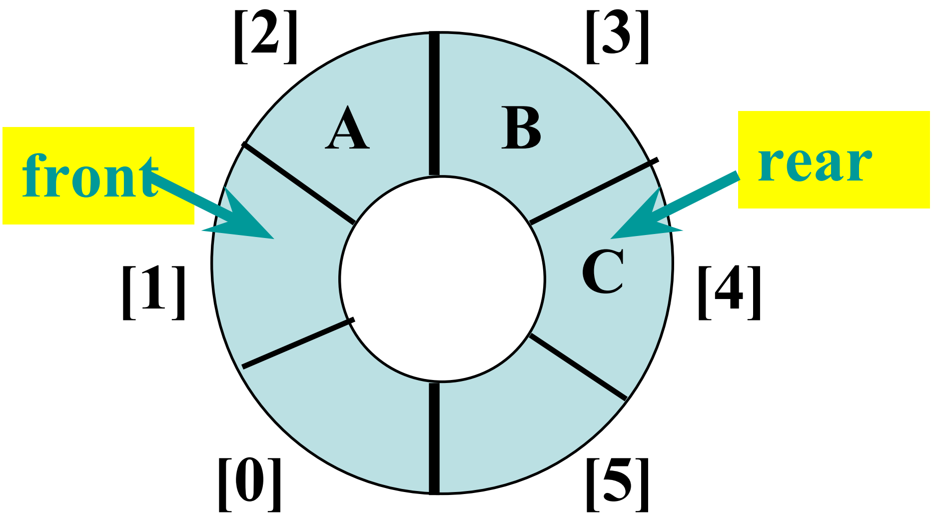


# Queue Implementation Using Circular Array

## Moving rear Clockwise

- rear++;

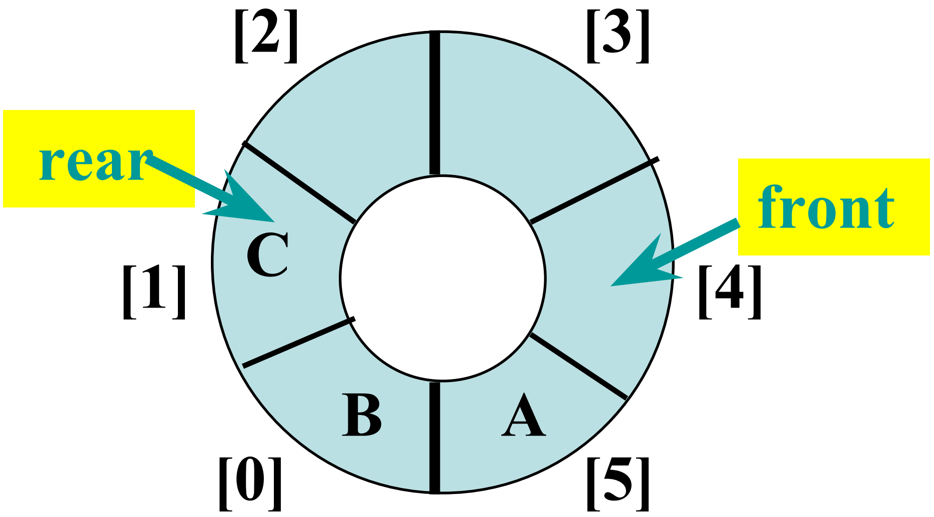
  if (rear = = queue.length) rear = 0;



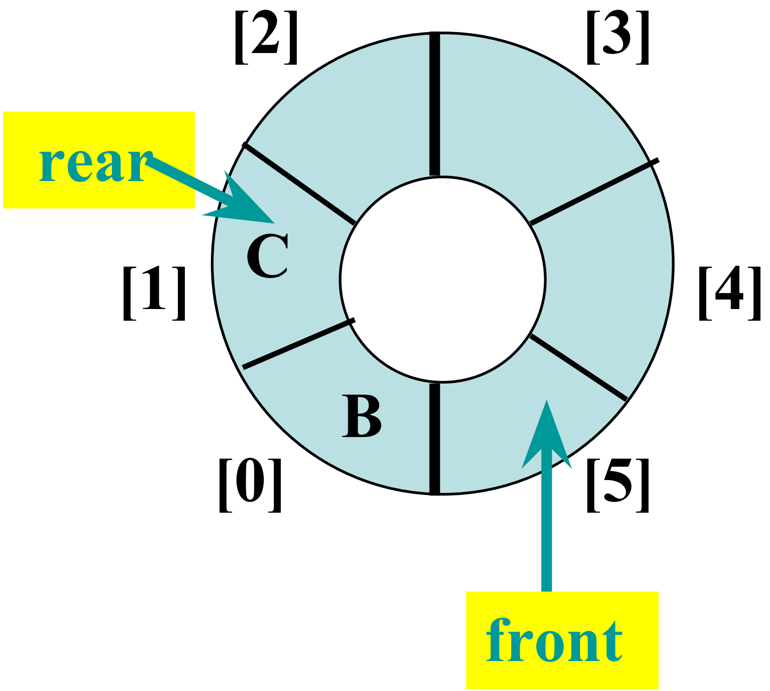

- rear = (rear + 1) % queue.length;

# Queue Implementation Using Circular Array

## Empty That Queue



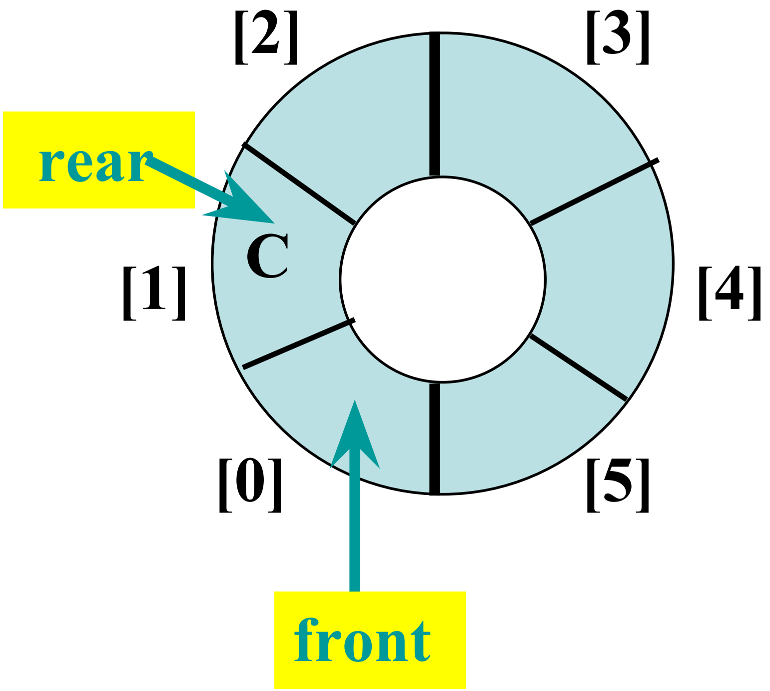
## Empty That Queue



# Queue Implementation Using Circular Array

AAST

## Empty That Queue

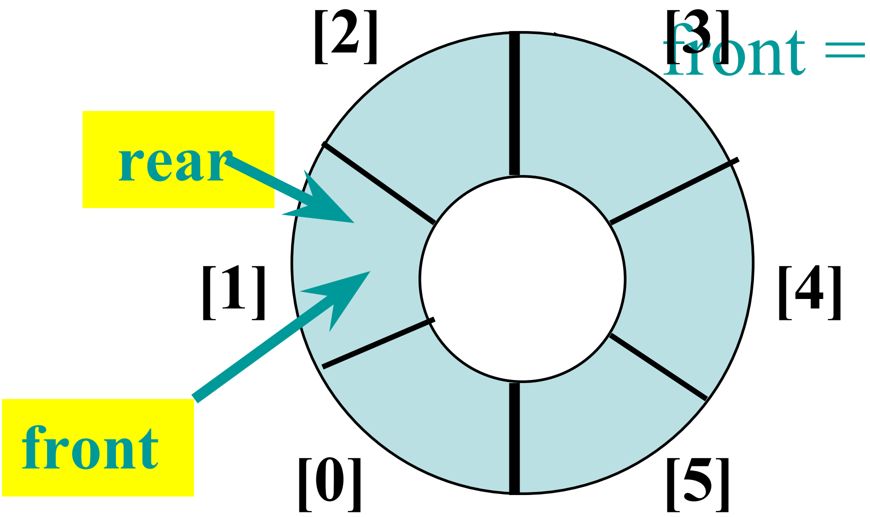


# Queue Implementation Using Circular Array

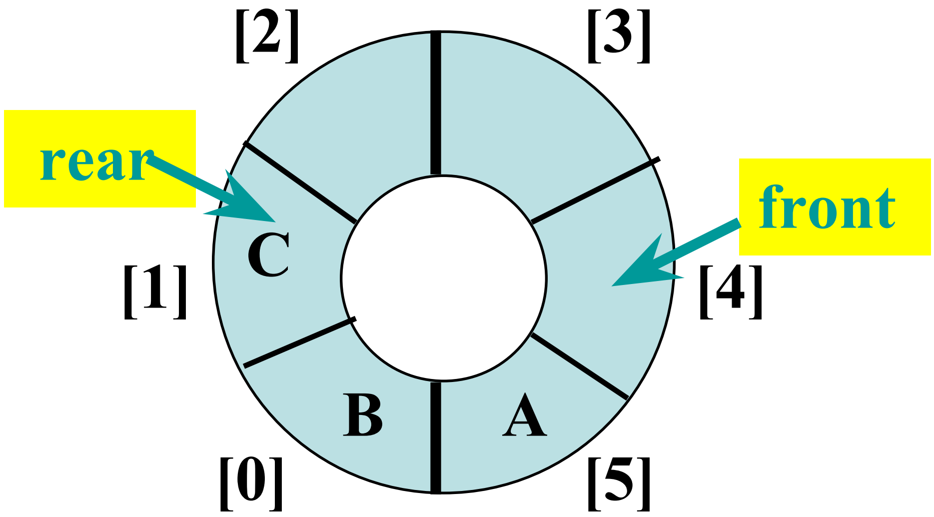
- When a series of removes causes the queue to become empty ==> front = rear.

Or

- When a queue is constructed, it is empty initialize front = rear = 0.

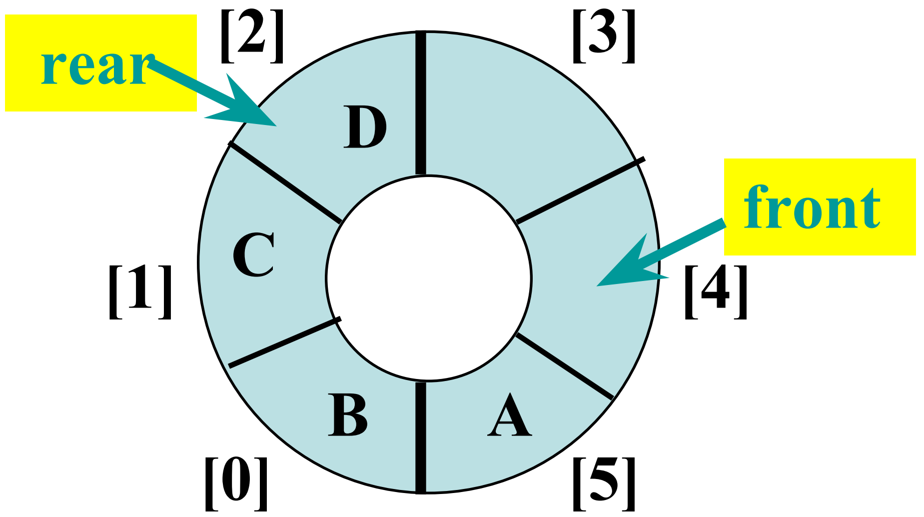


## A Full Queue



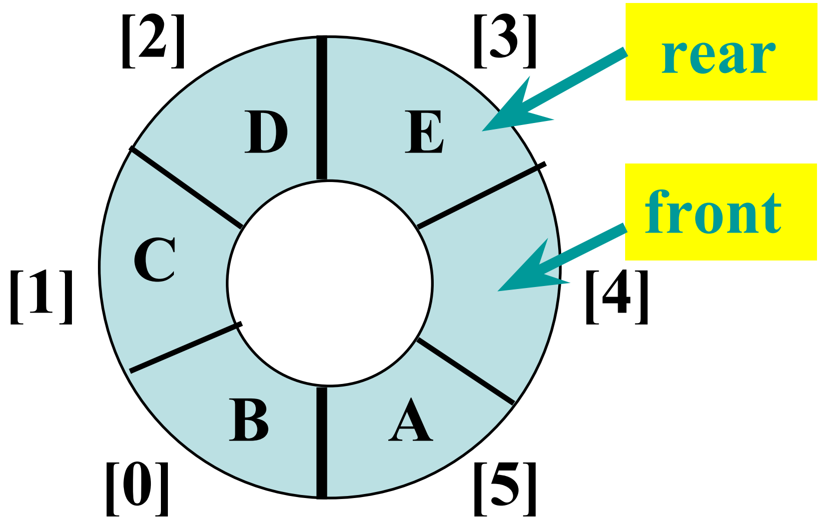
AAST

## A Full Queue



# Queue Implementation Using Circular Array

## A Full Queue

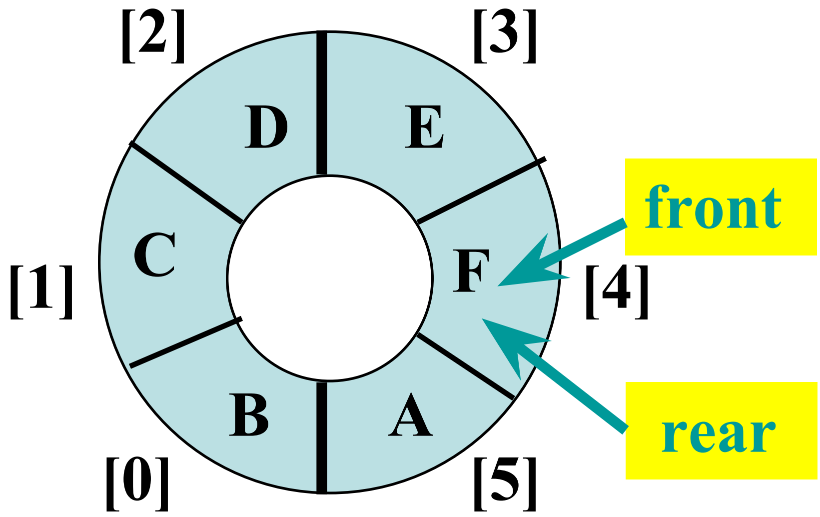


# Queue Implementation Using Circular Array

- When a series of adds causes the queue to become full

  ==> front = rear.

- So we cannot distinguish between a full queue and an empty queue!



# !!!Problems



- Remedies.
  - Don't let the queue get full.
    - When the addition of an element will cause the queue to be full, increase array size.
    - This is what the text does.
  - Define a boolean variable lastOperationIsPut.
    - Following each put set this variable to true.
    - Following each remove set to false.
    - Queue is empty iff (front == rear) && !lastOperationIsPut
    - Queue is full iff (front == rear) && lastOperationIsPut

# !!!Problems

- Remedies (continued).
  - Define an integer variable size.
    - Following each put do size++.
    - Following each remove do size--.
    - Queue is empty iff (size == 0)
    - Queue is full iff (size == queue.length)
  - Performance is slightly better when first strategy is used.

# EnQueue & DeQueue In Circular Array

EnQueue

rear = (rear + 1) mod n

DeQueue

front = (front + 1) mod n

## Empty/Full In Circular Array

When rear equals front, Queue is empty

When (rear + 1) mod n equals front, Queue is full

Circular array with capacity *n* at most can hold *n-1* items.

# EnQueue & DeQueue In Circular Array

The problem is that we implement this circular queue into **Linear Arrays**

# Circular Queues

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | 7 | 18 | 14 | 36 | 45 | 21 | 99 | 72 |

0 1 2 3 4 5 6 7 8 9

- We will explain the concept of circular queues using an example.
- In this queue, front = 2 and rear = 9.
- Now, if you want to insert a new element, it cannot be done because the space is available only at the left of the queue.
- If rear = MAX – 1, then OVERFLOW condition exists.
- This is the major drawback of an array queue. Even if space is available, no insertions can be done once rear is equal to MAX – 1.
- This leads to wastage of space. In order to overcome this problem, we use circular queues.
- In a circular queue, the first index comes right after the last index.
- A circular queue is full, only when front=0 and rear = Max – 1.

# Inserting an Element in a Circular Queue

- **For insertion we check for three conditions which are as follows:**
- **If front=0 and rear= MAX – 1, then the circular queue is full.**

| 90 | 49 | 7 | 18 | 14 | 36 | 45 | 21 | 99 | 72 |
|---|---|---|---|---|---|---|---|---|---|
| front=0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | rear = 9 |

- **If rear != MAX – 1, then the rear will be incremented and value will be inserted**

| 90 | 49 | 7 | 18 | 14 | 36 | 45 | 21 | 99 | |
|---|---|---|---|---|---|---|---|---|---|
| front=0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | rear= 8 | 9 |

- **If front!=0 and rear=MAX -1, then it means that the queue is not full. So, set rear = 0 and insert the new element.**

| | 49 | 7 | 18 | 14 | 36 | 45 | 21 | 99 | 72 |
|---|---|---|---|---|---|---|---|---|---|
| | front=1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | rear= 9 |

# Inserting an Element in a Circular Queue

- **rear = front -1 overflow**

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 9 | 10 | 7 | 18 | 14 | 36 | 45 | 21 | 99 | 72 |
| | rear=1 | front=2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

# Algorithm to Insert an Element in a Circular Queue

```
Step 1: IF FRONT=0 and REAR=MAX-1, or REAR = FRONT - 1 then
          Write "OVERFLOW"
              Goto Step 4
          [END OF IF]

Step 2: IF FRONT = -1 and REAR = -1, then;
          SET FRONT = REAR = 0
      ELSE IF REAR = MAX - 1 and FRONT != 0
           SET REAR = 0
      ELSE
           SET REAR = REAR + 1
          [END OF IF]
Step 3: SET QUEUE[REAR] = VAL
Step 4: Exit
```

# Deleting an Element from a Circular Queue

- To delete an element again we will check for three conditions:
- If front = -1, then it means there are no elements in the queue. So an underflow condition will be reported.

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | |

- If the queue is not empty and after returning the value on front, if front = rear, then it means now the queue has become empty and so front and rear are set to -1.

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | front=rear= 9 |
|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | 81 |

Delete this element and set rear = front = -1

- If the queue is not empty and after returning the value on front, if front = MAX -1, then front is set to 0.

| 0 | 1 | 2 | 3 | 4 | rear= 5 | 6 | 7 | 8 | front= 9 |
|---|---|---|---|---|---|---|---|---|---|
| 72 | 63 | 9 | 18 | 27 | 39 | | | | 81 |

# Algorithm to Delete an Element from a Circular Queue

```
Step 1: IF FRONT = -1, then
        Write "Underflow"
        Goto Step 4
        [END OF IF]
Step 2: SET VAL = QUEUE[FRONT]
Step 3: IF FRONT = REAR
      SET FRONT = REAR = -1
        ELSE
      IF FRONT = MAX -1
         SET FRONT = 0
      ELSE
         SET FRONT = FRONT + 1
      [END OF IF]
        [END OF IF]
Step 4: EXIT
```

# C-Implementation for Circular Queue

```
int CQueue[MAX_SIZE], frontp=-1, rearp=-1;
        // Global declarations - Queue
                frontp = pointer to front
                rearp = pointer to rear
```

# C-Implementation for Circular Queue

AAST

```
//  Function to Check Circular Queue Full
int  CQueue_full()
{
   if( (frontp == rearp+1) || (frontp == 0 && rearp== MAX_SIZE-1)) return 1;
   return 0;
}
```

```
//  Function to Check Circular Queue Empty
int CQueue_empty()
{
   if(frontp== -1) return 1;
   return 0;
}
```

# C-Implementation for Circular Queue

```
// Function for enqueue == Insert operation
void CQueue_insert(int element)
{
    if( CQueue_full()) printf("\n\n Overflow  ….. !\n");
    else
    {
        if(frontp ==-1)frontp =0;
        rearp =(rearp+1) % MAX_SIZE;
        CQueue[rearp] = element;
    }
}
```

# C-Implementation for Circular Queue

AAST

```
// Function for dequeue == Delete operation
int CQueue_delete()
{
    int element;
    if(CQueue_empty()){
        printf("\n\n Underflow  ______!\n\n");
        return(-1);
    }
    else
    {
        element=CQueue[frontp];
        if(frontp==rearp){ frontp =-1; rearp=-1;} //  CQueue has only one element ?
        else
            frontp=(frontp+1) % MAX_SIZE;
        return(element);
    }
}
```

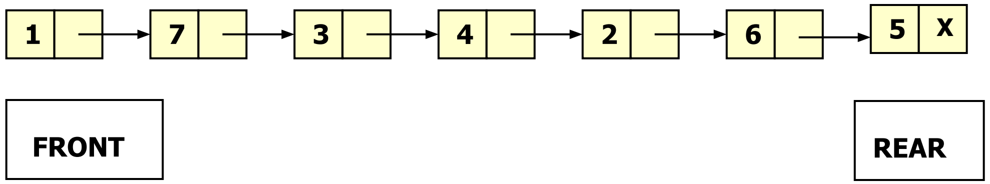
# Implementing a Queue Using Linked Lists

# Linked List Implementation of Queues

- **Using a singly linked list to hold queue elements,**
  - **Using FRONT pointer pointing to the start element**
  - **Using REAR pointer pointing to the last element**

- **Insertions is done at the rear end using REAR pointer**
- **Deletions is done at the front end using FRONT pointer**

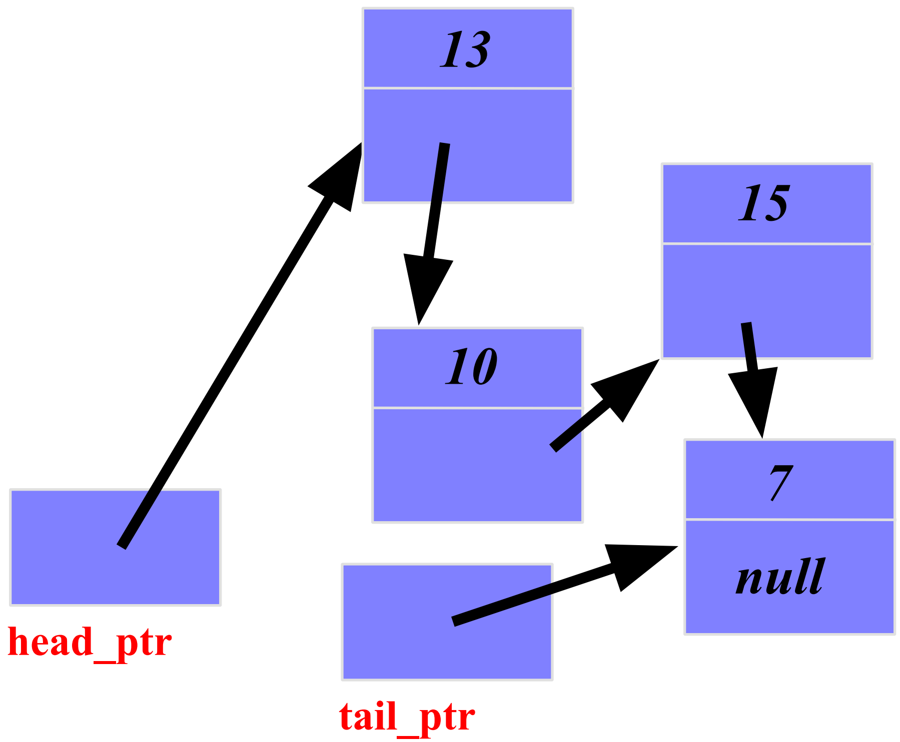
- **If FRONT = REAR = NULL, then the queue is empty.**



# Linked List Implementation of Queues

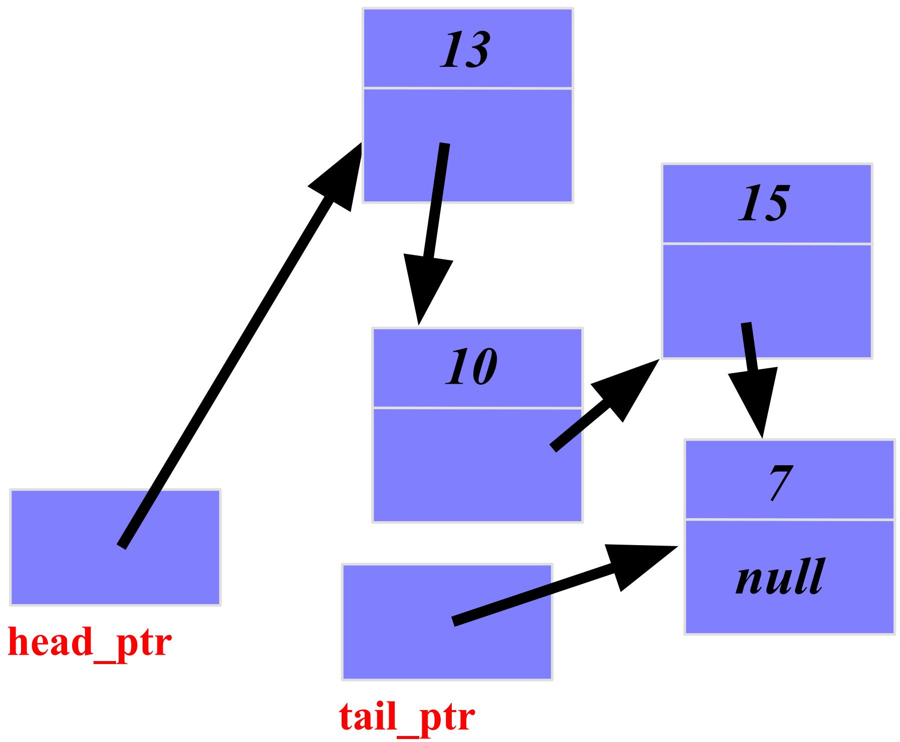


- A queue can also be implemented with a linked list with both a head and a tail pointer.



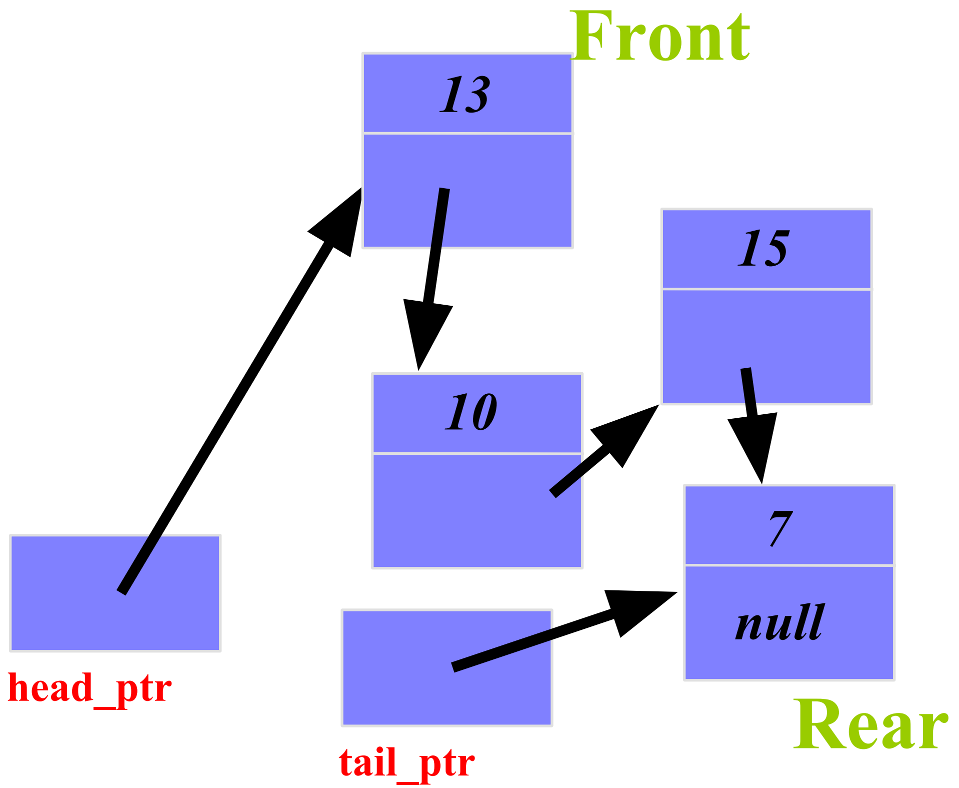
# Linked List Implementation of Queues

- Which end do you think is the front of the queue? Why?



# Linked List Implementation of Queues

- The head_ptr points to the front of the list.
- Because it is harder to remove items from the tail of the list.



# Inserting an Element in a Linked List Queue

```
Algorithm to insert an element in a linked queue

Step 1: Allocate memory for the new node and
        name the pointer as PTR
Step 2: SET PTR->DATA = VAL
Step 3: IF FRONT = NULL, then
      SET FRONT = REAR = PTR
      SET FRONT->NEXT = REAR->NEXT = NULL
        ELSE
      SET REAR->NEXT = PTR
      SET REAR = PTR
      SET REAR->NEXT = NULL
        [END OF IF]
Step 4: END

Time complexity: O(1)
```

# Deleting an Element from a Linked List Queue

```
Algorithm to delete an element from a
linked queue

Step 1: IF FRONT = NULL, then
        Write "Underflow"
        Go to Step 5
            [END OF IF]
Step 2: SET PTR = FRONT
Step 3: FRONT = FRONT->NEXT
Step 4: FREE PTR
Step 5: END
```

**Time complexity: O(1)**

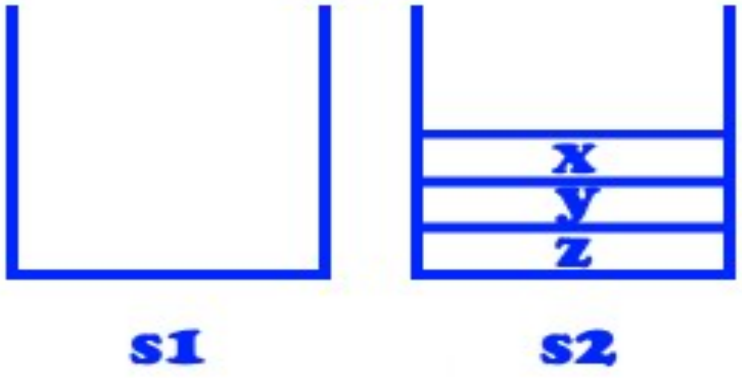
# Implementing a Queue Using Stacks

# Implementing a Queue Using Two Stacks

isEmpty
size
enqueue
dequeue
front

```
enqueue(Element e)
    s1.push(e)

dequeue
    If ( s2.isEmpty )
        While ( !s1.isEmpty ) do
            s2.push(s1.pop)
        EndWhile
    EndIf
    Return s2.pop
```



# Deque

# Deques

- A deque (double-ended queue) is a list in which elements can be inserted or deleted at either end.
- Also known as a head-tail linked list because elements can be added to or removed from the front (head) or back (tail).
- A deque can be implemented either using a circular array or a circular doubly linked list.
- In a deque, two pointers are maintained, LEFT and RIGHT which point to either end of the deque.

# Deque variants

AAST

- There are two variants of deques:
  - *Input restricted deque:*
    - insertions can be done only at one end
    - deletions can be done from both ends
  - *Output restricted deque:*
    - deletions can be done only at one ends
    - insertions can be done on both ends

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | | 29 | 37 | 45 | 54 | 63 | | |
| 0 | 1 | 2 | LEFT = 3 | 4 | 5 | 6 | RIGHT = 7 | 8 | |

9

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 42 | | | | | | | 63 | 27 | 18 |

RIGHT = 0 1 2 3 4 5 6 LEFT = 7 8 9

# Priority Queues

# Priority Queues

- A priority queue is a queue in which each element is assigned a priority
- The priority of elements is used to determine the order in which these elements will be processed
- The general rule of processing elements of a priority queue can be given as:
  - An element with higher priority is processed before an element with lower priority
  - Two elements with same priority are processed on a first come first served (FCFS) basis
- Priority queues are widely used in operating systems to execute the highest priority process first
- In computer's memory priority queues can be represented using arrays or linked lists

# Priority Queues

Two kinds of priority queues:

- Min priority queue.
- Max priority queue.

# Min Priority Queue

- Collection of elements
- Each element has a priority or key
- Supports following operations:
  - isEmpty
  - size
  - add/put an element into the priority queue
  - get element with min priority
  - remove element with min priority

# Max Priority Queue

- Collection of elements
- Each element has a priority or key
- Supports following operations:
  - isEmpty
  - size
  - add/put an element into the priority queue
  - get element with max priority
  - remove element with max priority

# Complexity Of Operations

- Two good implementations are heaps and leftist trees
- isEmpty, size, and get => $O(1)$ time
- put and remove => $O(\log n)$ time where $n$ is the size of the priority queue

# Applications

## Sorting

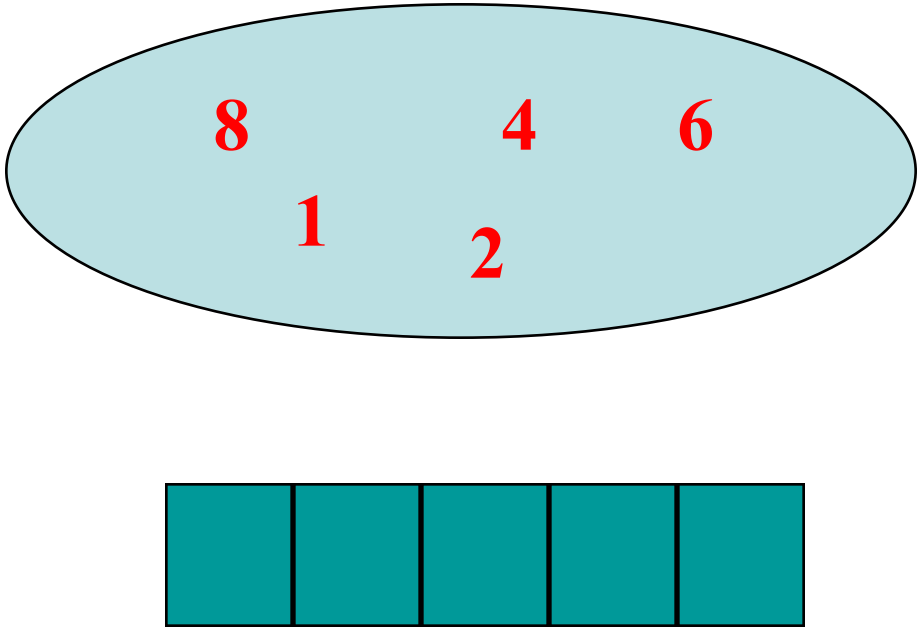
- use element key as priority
- put elements to be sorted into a priority queue
- extract elements in priority order
  - if a min priority queue is used, elements are extracted in ascending order of priority (or key)
  - if a max priority queue is used, elements are extracted in descending order of priority (or key)

# Sorting Example

Sort five elements whose keys are 6, 8, 2, 4, 1 using a max priority queue

- Put the five elements into a max priority queue
- Do five remove max operations placing removed elements into the sorted array from right to left

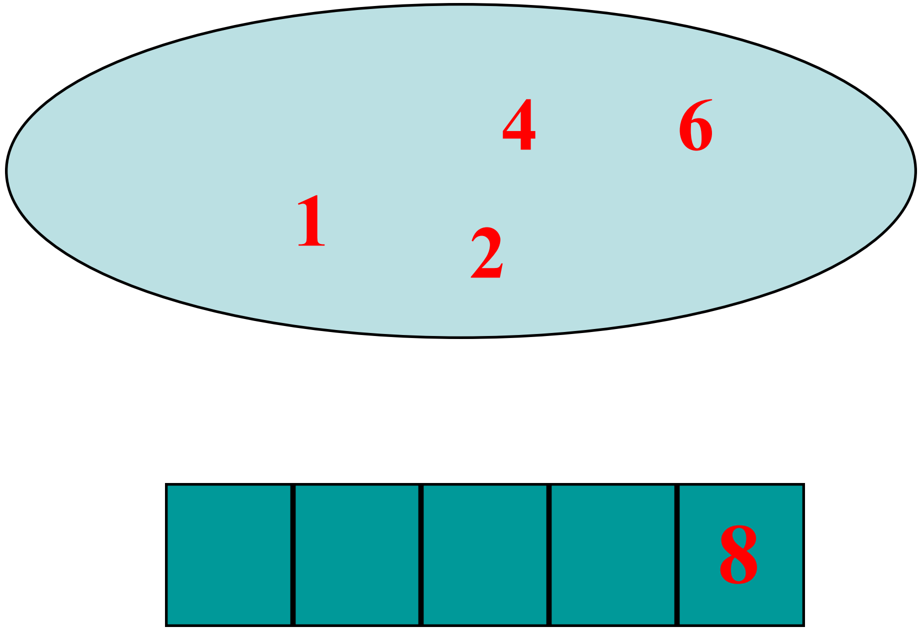
# After Putting Into Max Priority Queue




**Max Priority Queue**

Sorted Array

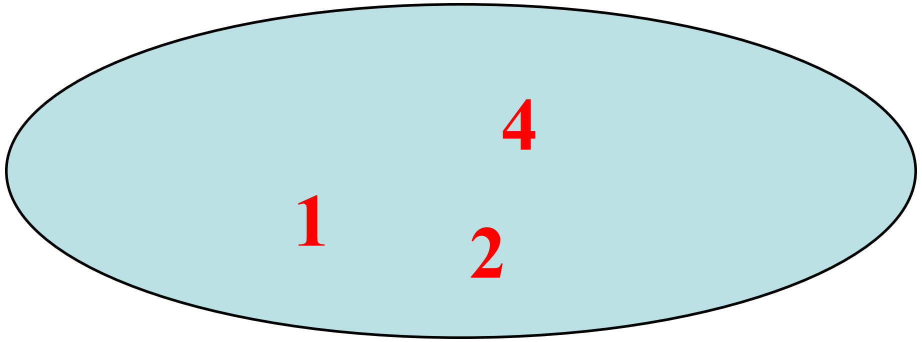
# After First Remove Max Operation



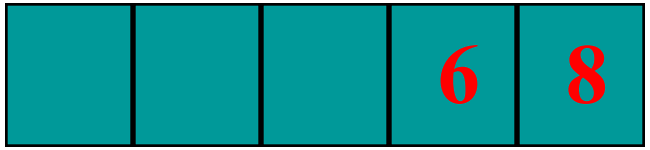

**Max Priority Queue**

Sorted Array

# After Second Remove Max Operation



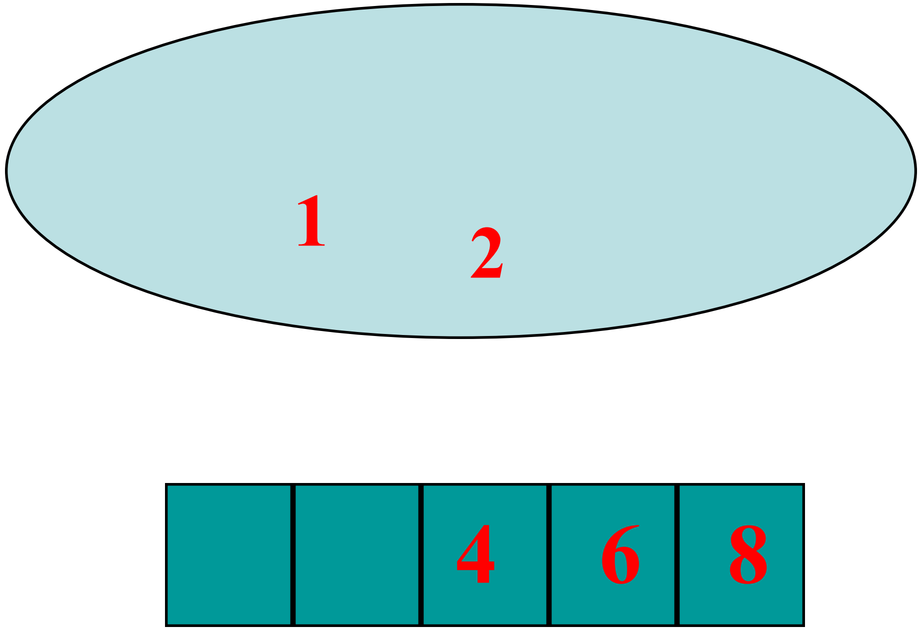

**Max Priority Queue**




Sorted Array

# After Third Remove Max Operation

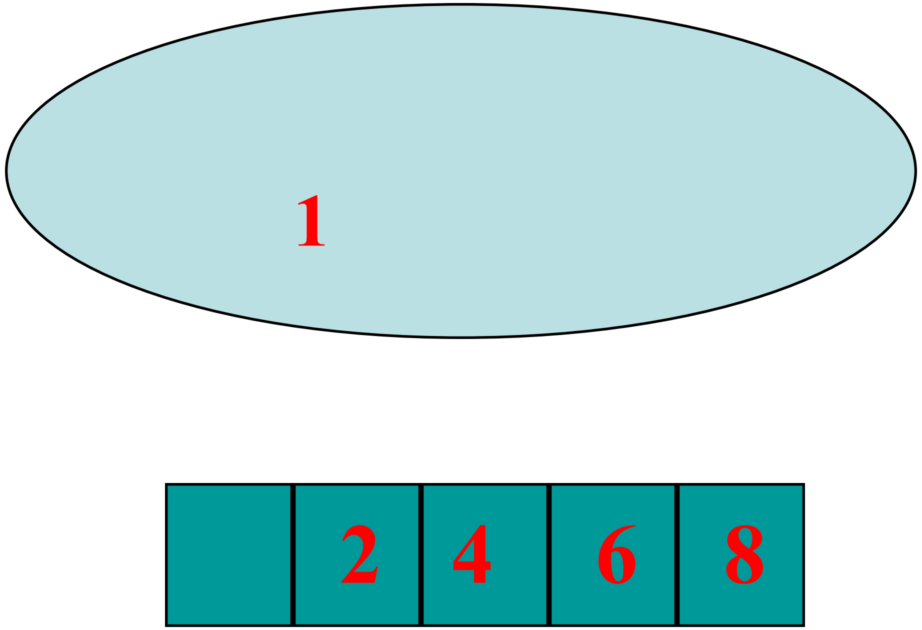
AAST




**Max Priority Queue**

Sorted Array

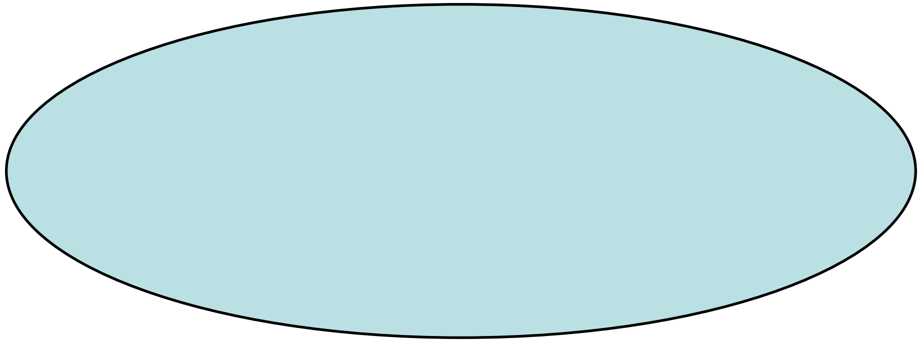
# After Fourth Remove Max Operation




**Max Priority Queue**

Sorted Array

# After Fifth Remove Max Operation



**Max Priority Queue**

| | | | | |
|---|---|---|---|---|
| 1 | 2 | 4 | 6 | 8 |

Sorted Array

# Complexity Of Sorting

AAST

Sort n elements

- n put operations => $O(n \log n)$ time
- n remove max operations => $O(n \log n)$ time
- Total time is $O(n \log n)$
- Compare with $O(n^2)$ for sort methods

# Array Representation of Priority Queues

- When arrays are used to implement a priority queue, then a separate queue for each priority number is maintained
- Each of these queues will be implemented using circular arrays or circular queues
- Every individual queue will have its own FRONT and REAR pointers
- We can use a two-dimensional array for this purpose where each queue will be allocated same amount of space
- Given the front and rear values of each queue, a two dimensional matrix can be formed

# Linked List Representation of Priority Queues

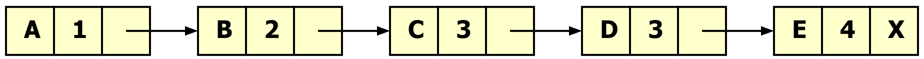
- **When a priority queue is implemented using a linked list, then every node of the list contains three parts:**

  **(i) the information or data part (A, B, C, …)**

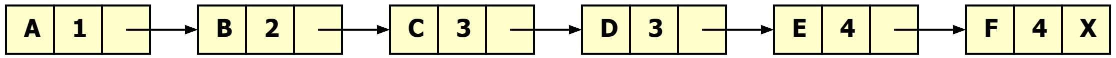
  **(ii) the priority number of the element (1, 2, 3, …)**

  **(iii) and address of the next element**

- **If we are using a sorted linked list, then element having higher priority will precede the element with lower priority.**




**Priority queue after insertion of a new node (F, 4)**



# Applications of Queues

# Applications of Queues

1. **Queues are widely used as <u>waiting lists</u> for a single shared resource like printer, disk, CPU.**

2. **Queues are used to <u>transfer data asynchronously</u> e.g., pipes, file IO, sockets.**

3. **Queues are used as <u>buffers</u> on MP3 players and portable CD players, iPod playlist.**

# Applications of Queues

5. **Queues are used in Playlist for jukebox to add songs to the end, from the front of the list. play**

6. **Queues are used in OS for handling interrupts. When programming a real-time system that can be interrupted (e.g., by a mouse click), it is necessary to process the interrupts immediately before proceeding with the current job. If the interrupts have to be handled in the order of arrival, then a FIFO queue is the appropriate data structure**

# Applications of Queues

In **Josephus problem**,

- **n** people stand in a circle waiting to be executed
- Counting starts at some point in the circle and proceeds in a specific direction around the circle
- In each step, a certain number of people are skipped and the next person is executed (or eliminated)
- The elimination of people makes the circle smaller and smaller
- At the last step, only one person remains who is declared the ‘winner’

# Questions