QUEUE

Prof. Navrati Saxena

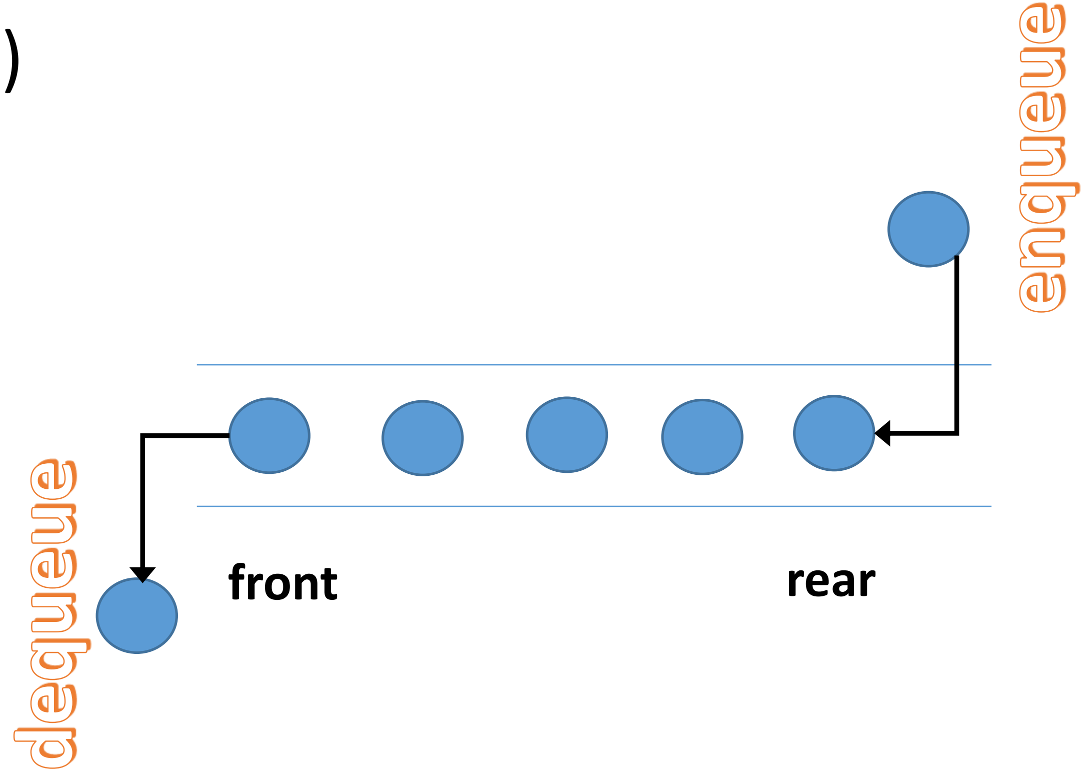
Queue

- Linear data structure
- Based on the principle of First-In-First-Out (FIFO)
  - Data entered first will be accessed first
- Ordered collection of items where an item is inserted at one end called the rear and an existing item is removed from the other end, called the front

- En-Queue: Add items on the Rear end
- De-Queue: Remove items from the Front end

Operations in an Queue

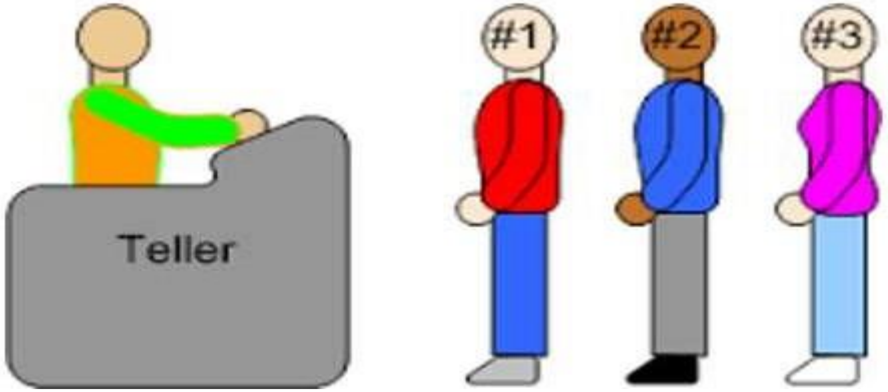
- Operations performed from both ends
  - Head-tail or front-back.
  - **En-Queue**: insert an item into back (rear) of the queue
  - **De-Queue**: Remove an item from the front of the queue



- People standing at MacDonalds are perfect example of an Queue

Applications of Queue

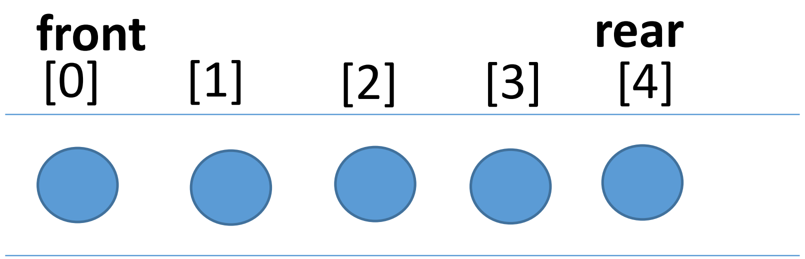
- Checkout line
- Printer/keyboard/mouse etc. queue
- Airport take-off
- Operating system
  - Job Scheduling
  - Multi-programming
- Different types of scheduling algorithms: Round Robin



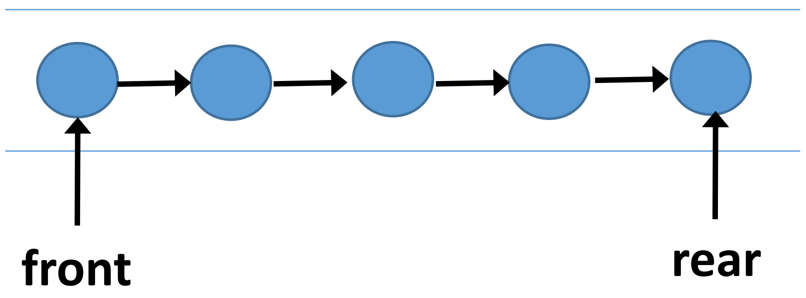
Queue

- The queue can be implemented into two ways:
  - Using arrays (Static implementation)
  - Using pointer (Dynamic implementation)

ARRAY / STATIC REPRESENTATION OF QUEUE

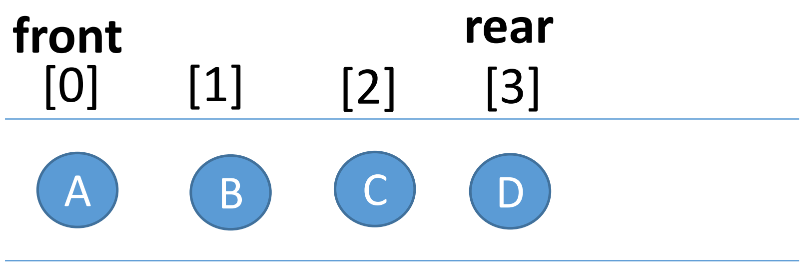


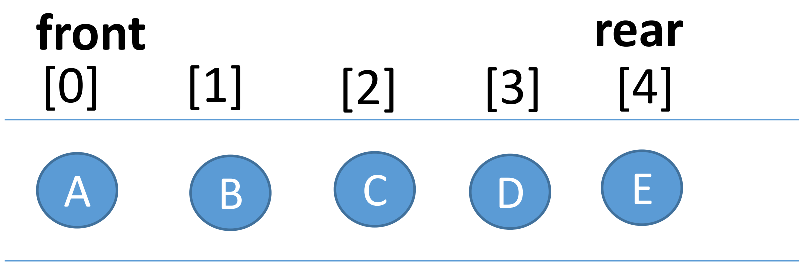
POINTER / DYNAMIC REPRESENTATION OF QUEUE



Queue Representation using Array (1/2)

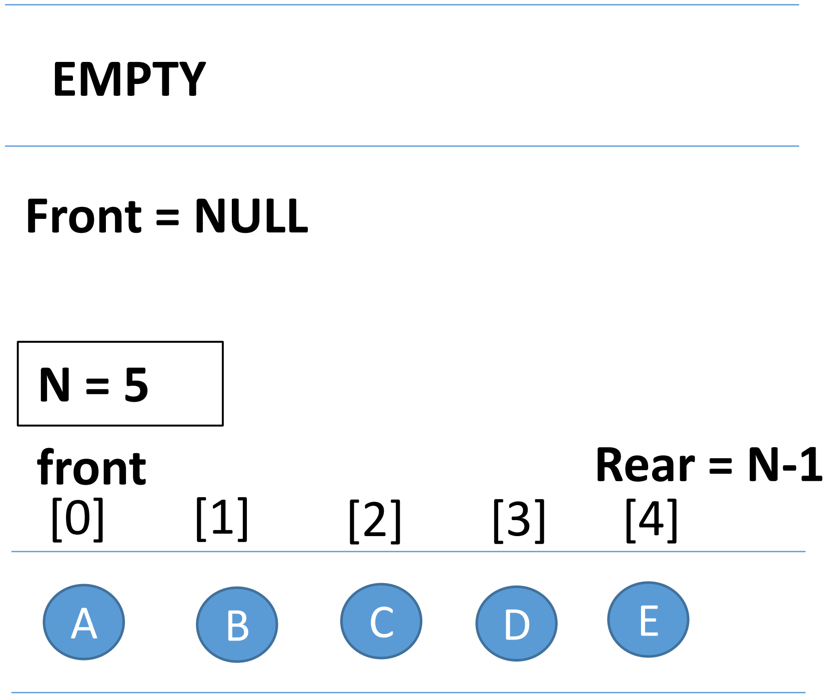
- Let QUEUE be an array
- Two variables: FRONT & REAR
- FRONT contains the location of the element to be removed or deleted
- REAR contains location of the last element inserted



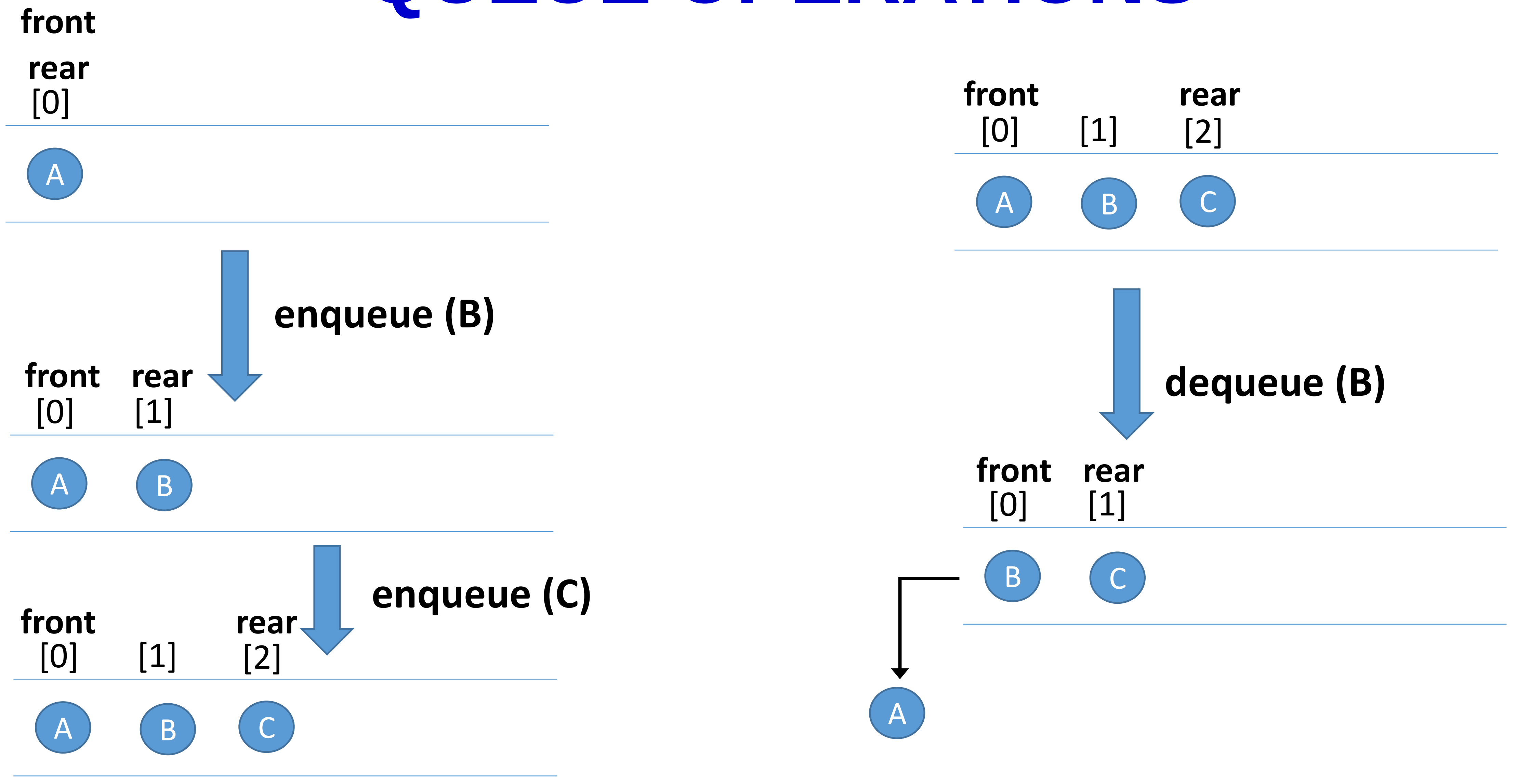


Queue Representation using Array (2/2)

- The conditions:
  - FRONT = NULL indicates empty queue
  - REAR = N-1 indicates that queue is full



QUEUE OPERATIONS



Algorithm: Enqueue (1/2)

ENQUEUE (QUEUE, REAR, FRONT, ITEM)

QUEUE is an array with N elements;

FRONT is the pointer that contains the location of the element to be deleted;

REAR contains the location of the inserted element; ITEM is the element to be inserted.

Step 1: if REAR = N-1 then [Check Overflow]

PRINT "QUEUE is Full or Overflow"

Exit [End if]

Algorithm: Enqueue (2/2)

Step 1: if REAR = N-1 then [Check Overflow]

PRINT "QUEUE is Full or Overflow"

Exit

Step 2: if FRONT = NULL [Check if empty Queue]

FRONT = -1

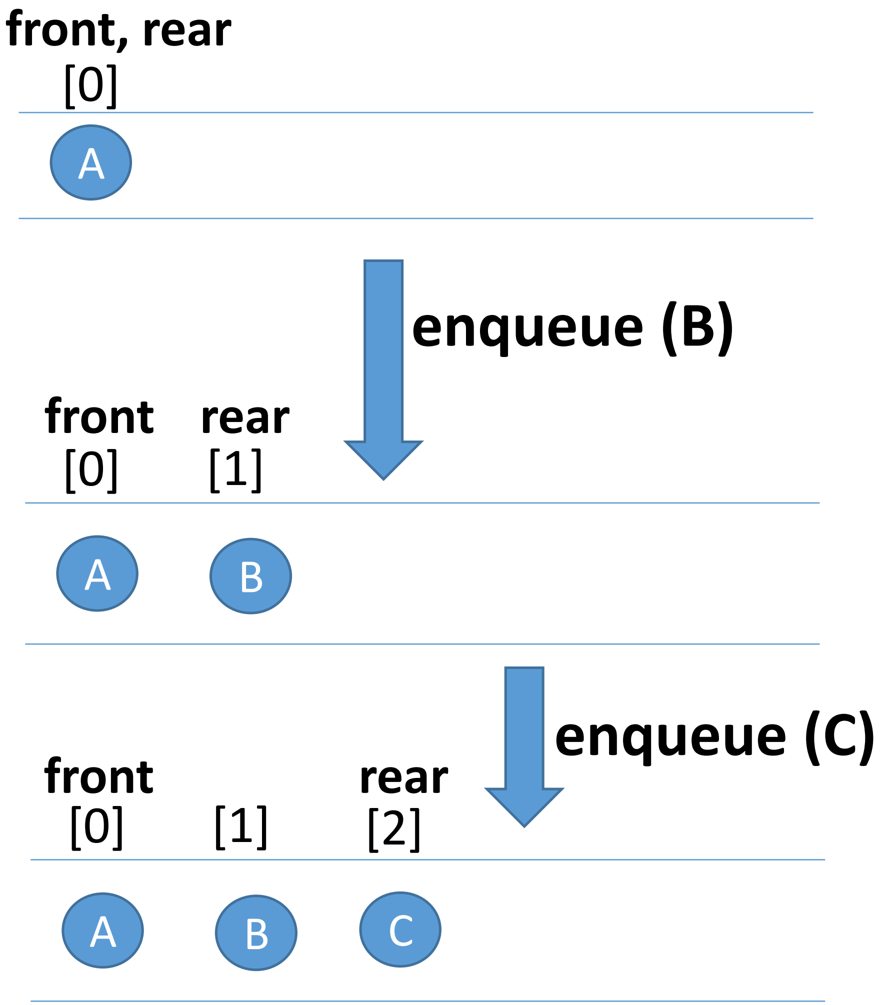
REAR = -1

Step 3: REAR = REAR + 1 //[Increment REAR]

QUEUE[REAR] = ITEM

//[Copy ITEM to REAR]

Step 4: Return



Algorithm: Dequeue (1/2)

ALGORITHM: DEQUEUE (QUEUE, REAR, FRONT, ITEM)

QUEUE is the array with N elements; FRONT is the pointer that contains the location of the element to be deleted; REAR contains the location of the inserted element. ITEM will store the element to be deleted.

```
Step 1: if FRONT = NULL [Check Empty Queue]
                    PRINT "QUEUE is Empty or Underflow"
                    Exit
          [End if]
```

Algorithm: Dequeue (2/2)

Step 2: ITEM = QUEUE[FRONT]

Step 3: if FRONT = REAR

[QUEUE has only one element]

FRONT = NULL

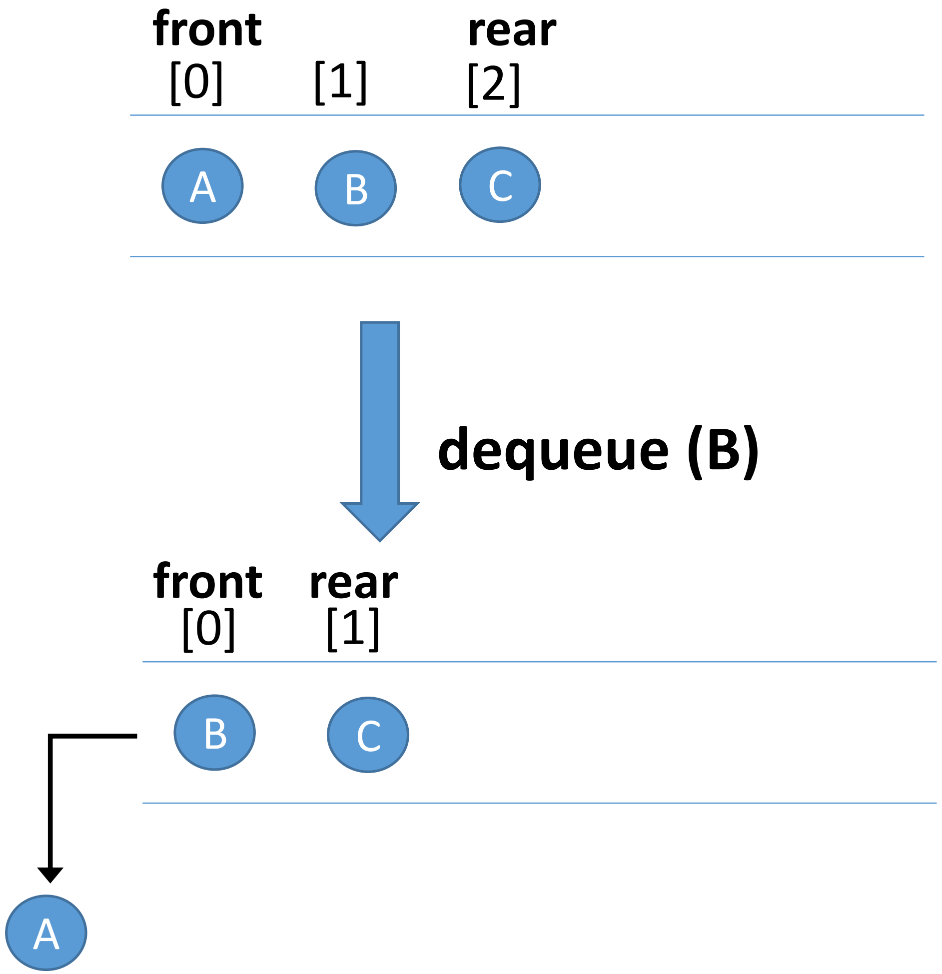
REAR = NULL

else

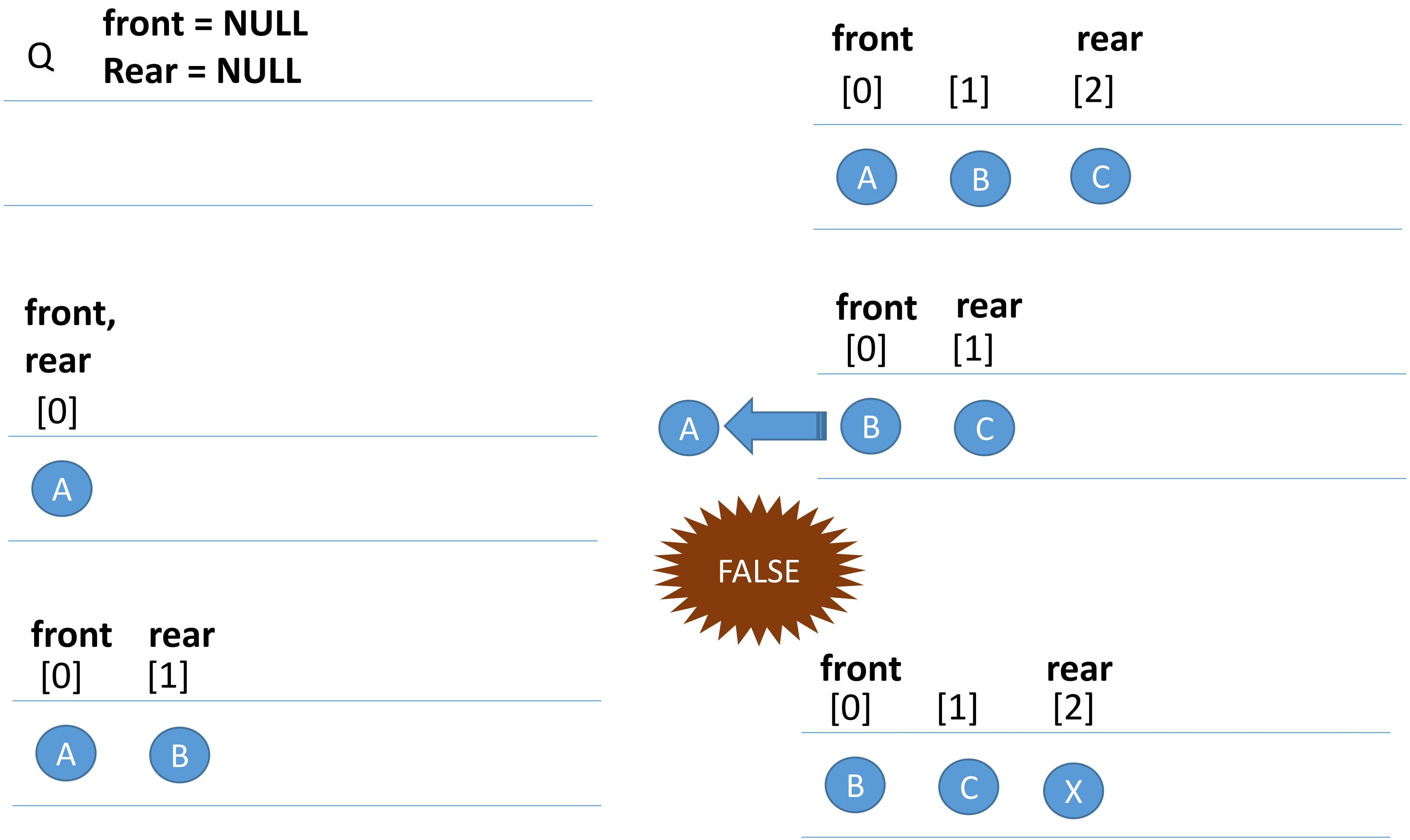
FRONT = FRONT + 1

[Increment FRONT pointer]

Step 4: Return

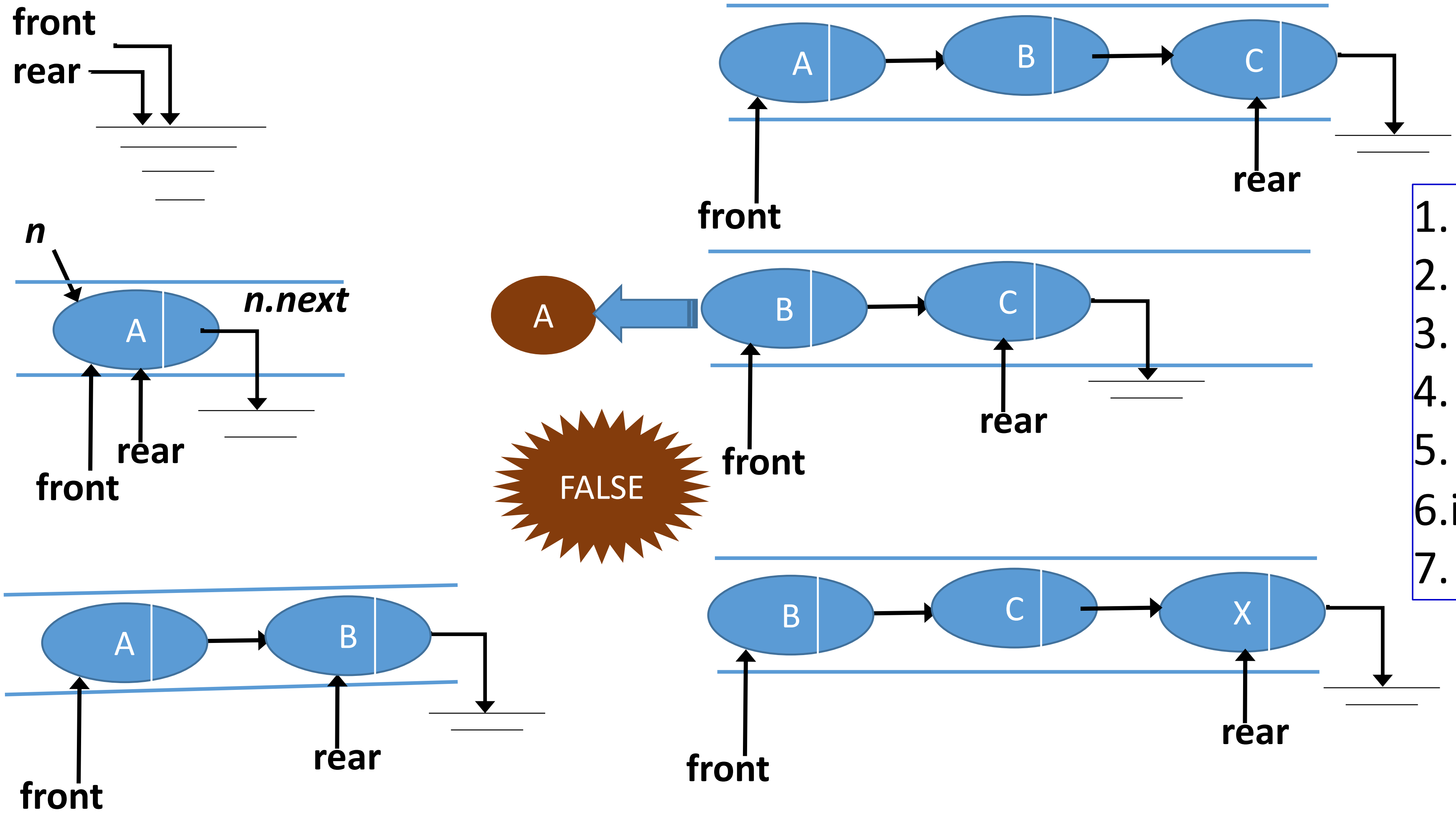


Array Implementation of Queue



1. makeQueue(Q)
2. enQueue(A, Q)
3. enQueue(B, Q)
4. enQueue(C, Q)
5. deQueue(Q)
6.isQueueEmpty(Q)
7. enQueue(X, Q)

Pointer Implementation of Queue



1. makeQueue(Q)
2. enQueue(A, Q)
3. enQueue(B, Q)
4. enQueue(C, Q)
5. deQueue(Q)
6.isQueueEmpty(Q)
7. enQueue(X, Q)

RECOMMENDED FUNCTIONS

- **Queue()** creates a new, empty queue; no parameters and returns an empty queue
- **enqueue(item)** adds a new item to rear of queue; needs the item and returns nothing
- **dequeue()** removes front item; needs no parameters, returns item, queue is modified
- **isEmpty()** test if queue is empty; needs no parameters, returns a boolean value
- **size()** returns number of items in the queue; needs no parameters; returns an integer

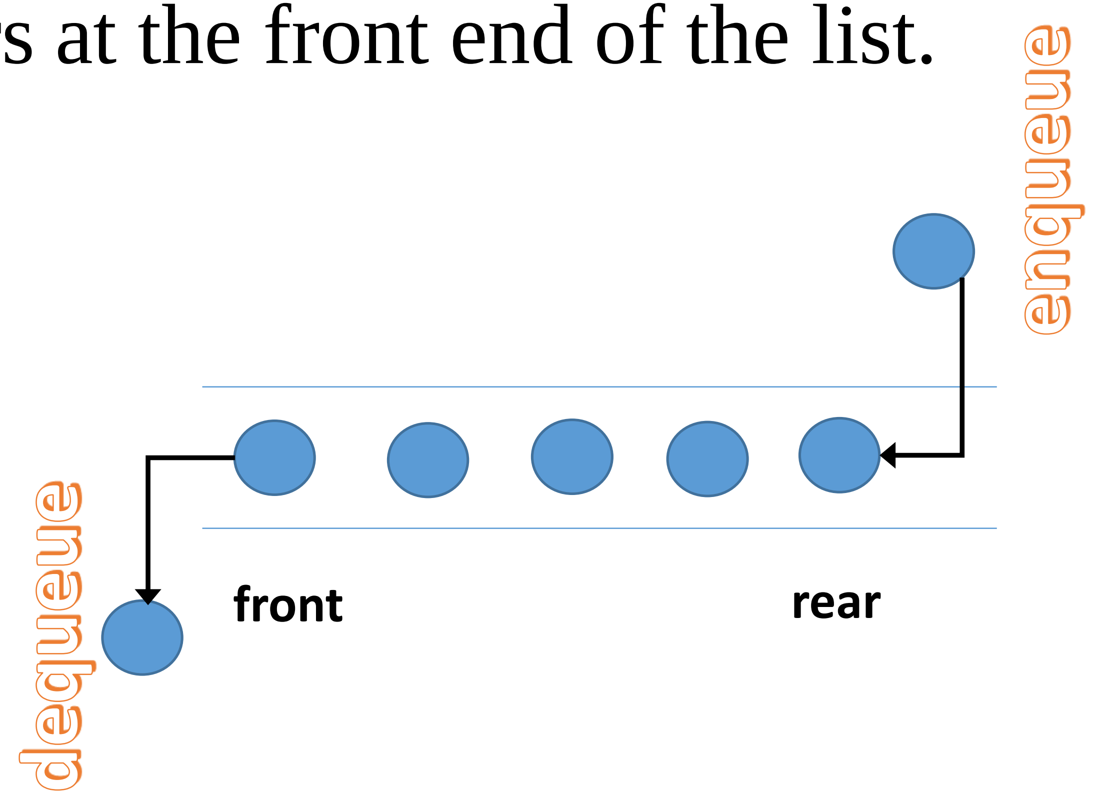
FOUR Types of Queues

- Simple Queue
- Circular Queue
- Priority Queue
- De-queue (Double Ended Queue) – not being covered

Simple Queue

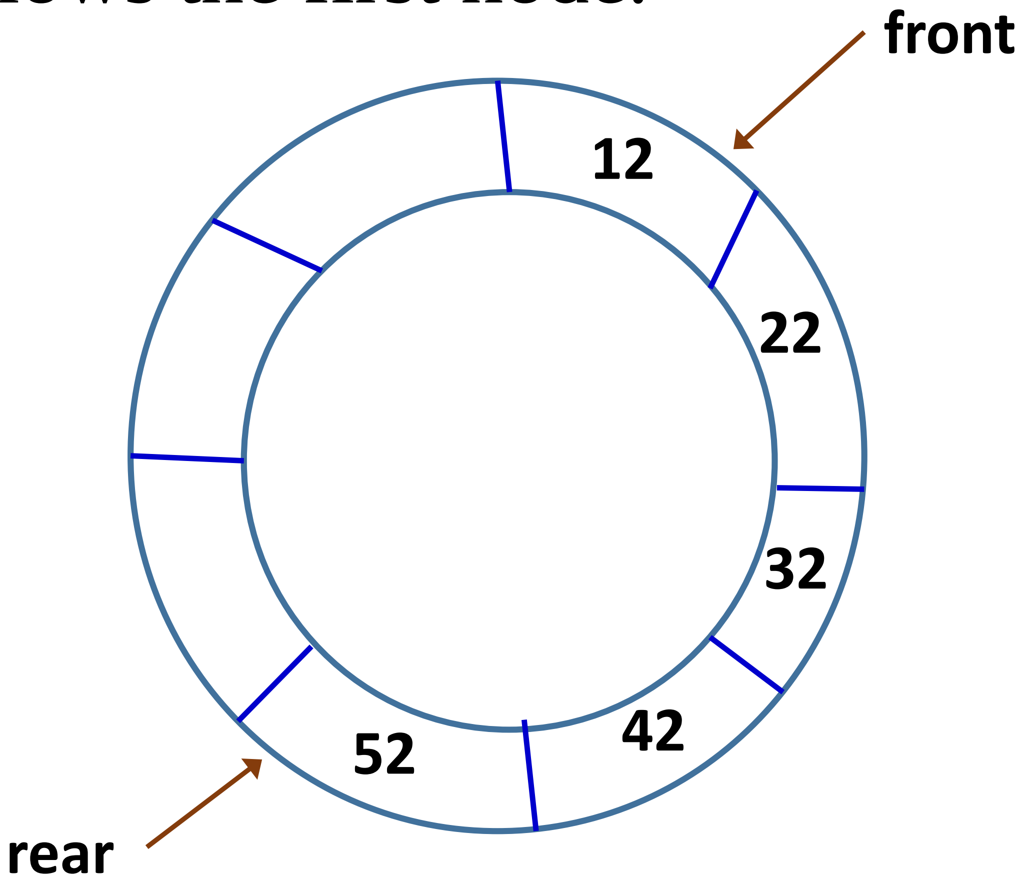
Simple Queue:

- Insertion occurs at the rear end of the list
- Deletion occurs at the front end of the list.

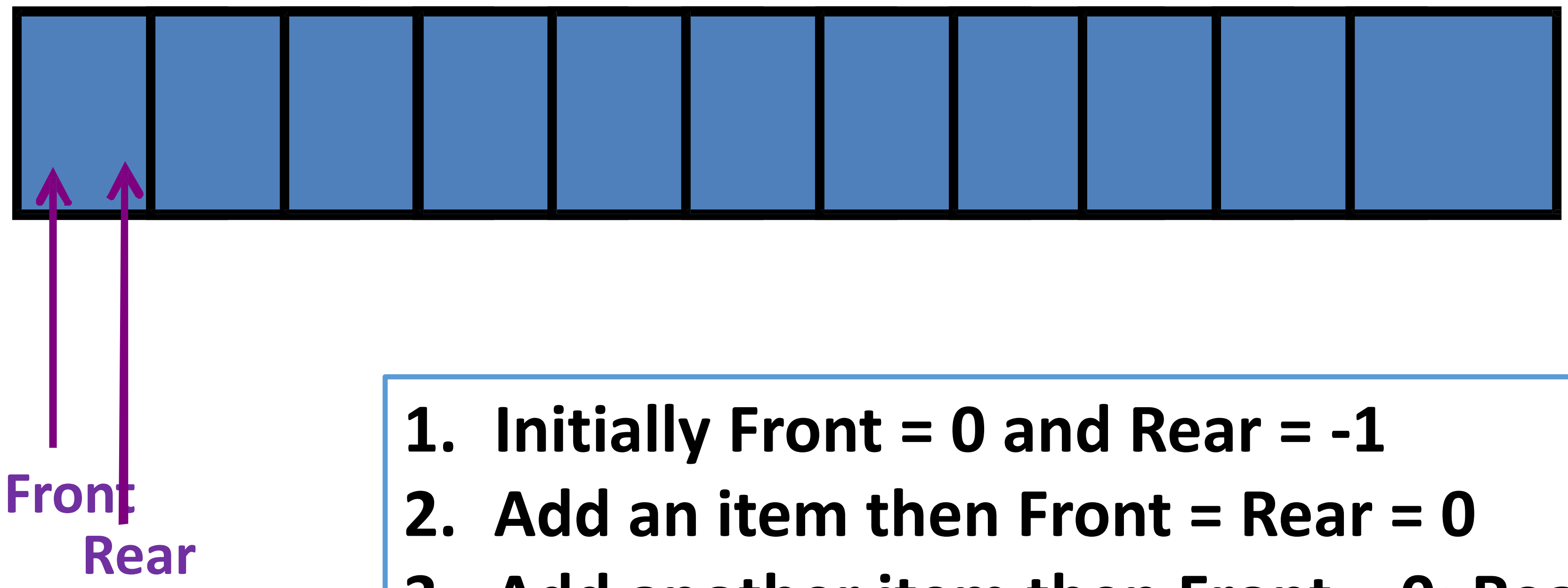


Circular Queue

Circular Queue: All nodes are treated as a circle such that the last node follows the first node.



Array Implementation: Circular Queue



1. Initially Front = 0 and Rear = -1
2. Add an item then Front = Rear = 0
3. Add another item then Front = 0; Rear = 1
4. Add three items then Front = 0; Rear = 4

Logical Circularity of Queue (1/2)

Addition causes the increment in REAR.

When REAR reaches MAX-1 position then Increment in REAR causes REAR to reach at first position that is 0

```
If (rear == MAX-1)
        rear = 0
Else
        rear = rear +1
```

rear = (rear + 1) % MAX;

Logical Circularity of Queue (2/2)

Deletion causes the increment in FRONT.

When FRONT reaches the MAX-1 position, then increment in FRONT, causes FRONT to reach at first position that is 0

```
If (front== MAX-1)
        front = 0
Else
        front= front +1
```

Front = (front + 1) % MAX;

Drawbacks of Circular Queue

- **Boundary case problem:** Difficult to distinguish full and empty queues
- In circular queue it is necessary that:
  - Before insertion, fullness of Queue must be checked (for overflow)
  - Before deletion, emptiness of Queue must be checked (for underflow)
- **Condition to check FULL Circular Queue**

```
If (( front == MAX-1) || ( front ==0 && rear == MAX - 1))
```

- **Condition to check EMPTY Circular Queue**

```
if (( front == MAX-1) || ( front ==0 && rear == MAX - 1))
```

To solve the drawback of circular Queue:
Use **count variable** to hold the current position (in case of insertion or deletion)

Circular Queue using count

Operations	Algorithms
Initialize Operation	**INIT**(QUEUE,FRONT,REAR,COUNT)
Is_Full check	**INSERT-ITEM**(QUEUE, FRONT, REAR, MAX, COUNT, ITEM)
Is_Empty check	**REMOVE-ITEM**(QUEUE, FRONT, REAR, COUNT, ITEM)
Insertion operation	**FULL-CHECK**(QUEUE,FRONT,REAR,MAX,COUNT,FULL)
Deletion operation	**EMPTY-CHECK**(QUEUE,FRONT,REAR,MAX,COUNT,EMPTY)

Algorithms

INIT(QUEUE,FRONT,REAR,COUNT)

1. FRONT = 1
2. REAR = 0
3. COUNT = 0
4. Return

FULL-CHECK(QUEUE,FRONT,REAR,MAX,COUNT,FULL)

If (COUNT = MAX) then

FULL = true

Otherwise

FULL = false

Return

EMPTY-CHECK(QUEUE,FRONT,REAR,MAX,COUNT,EMPTY)

This is used to check queue is empty or not.

If(COUNT = 0)

then EMPTY = true

Otherwise

EMPTY = false

Return

Insertion Operation

```
INSERT-ITEM( QUEUE, FRONT, REAR, MAX, COUNT, ITEM)
This algorithm is used to insert or add item into circular queue
If ( COUNT = MAX ) then
    Display “Queue overflow”
    Return
Otherwise
    If ( REAR == MAX ) then
         REAR = 1
    Otherwise
         REAR = REAR + 1
    QUEUE(REAR) = ITEM
    COUNT = COUNT + 1
Return
```

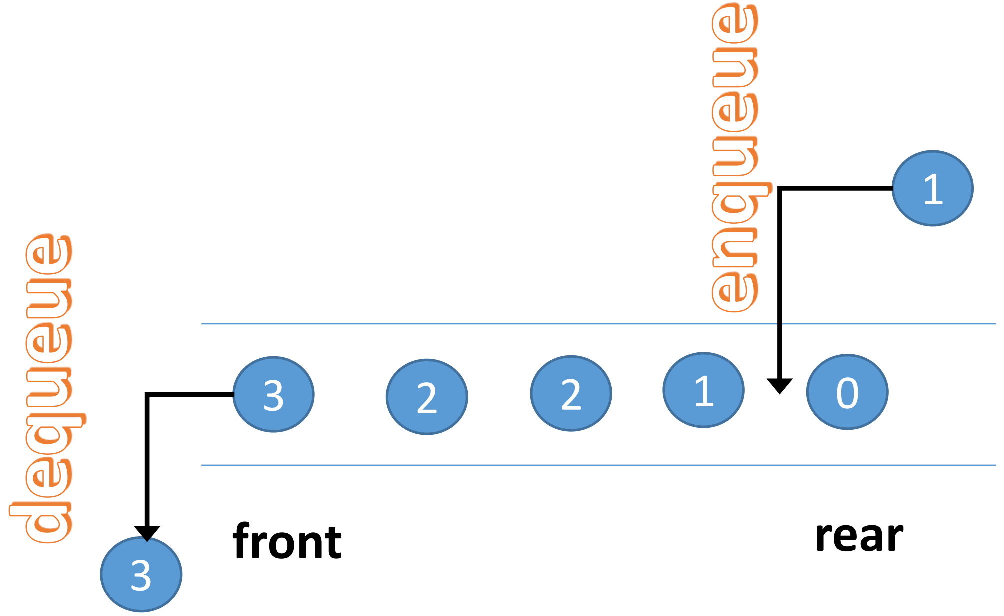
Deletion Operation

```
REMOVE-ITEM( QUEUE, FRONT, REAR, COUNT, ITEM)

If ( COUNT = 0 ) then
    Display “Queue underflow”
    Return
Otherwise
    ITEM = QUEUE(FRONT)l
    If ( FRONT =MAX ) then
        FRONT = 1
    Otherwise
        FRONT = FRONT + 1
        COUNT = COUNT + 1
    Return
```

Priority Queue (1/3)

- Each items have some pre-defined priority.
- An element can be inserted or removed from any position depending upon their priority.



Higher Value
=> Higher priority

Priority Queue (2/3)

- Stores item into queue with associated priority
- Does not support FIFO (First In First Out)
- It supports the following three operations:
  - **InsertWithPriority**: Insert an item to the queue with associated priority
  - **GetNext**: remove the element from the queue which has highest priority.

    Also know as **PopElement()** or **GetMaximum**
  - **PeekAtNext**(optional): Get the item with highest priority without removing it

Priority Queue (3/3)

- Every item has a priority associated with it
- An element with high priority is dequeued before an element with low priority
- If two elements have the same priority, they are served according to their order in the queue
- A typical priority queue supports following operations

  insert(item, priority): Inserts an item with given priority

  getHighestPriority(): Returns the highest priority item

  deleteHighestPriority(): Removes the highest priority item

Implementing Priority Queue

- **insert()** Adding an item at end of array in O(1) time
- **getHighestPriority()** Linearly search the highest priority item in array. This operation takes O(n) time
- **deleteHighestPriority()** First linearly search an item, then remove the item by moving all subsequent items one position back
- Using Linked List, time complexity of all operations remains same as array.
- The advantage with linked list is deleteHighestPriority() can be more efficient as we don't have to move items

Some Applications of Queue

1. Modeling and analysis of Computer Networks
2. CPU Scheduling
3. Graph algorithms like Dijkstra’s shortest path algorithm, Prim’s Minimum Spanning Tree.
4. All queue applications where priority is involved
5. Heap Sort

Thank you!