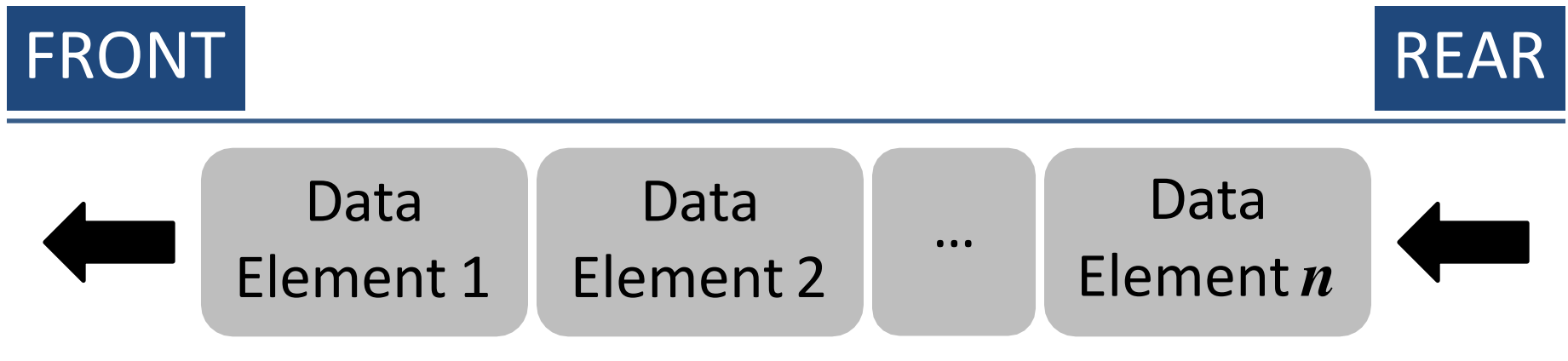


# Queues

# Introduction

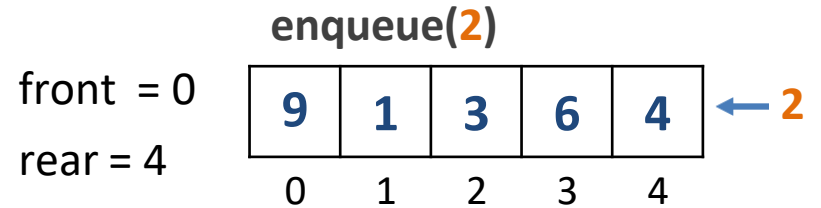
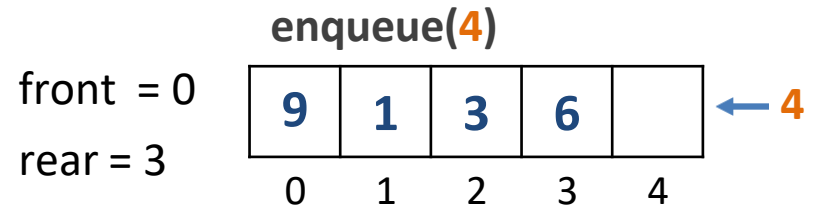
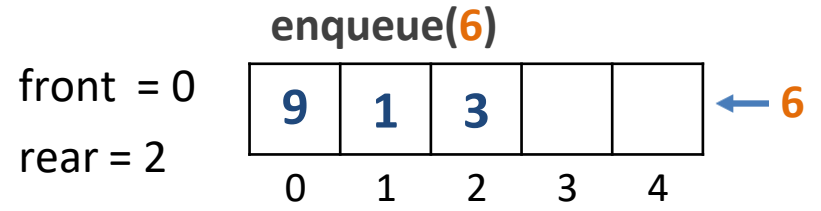
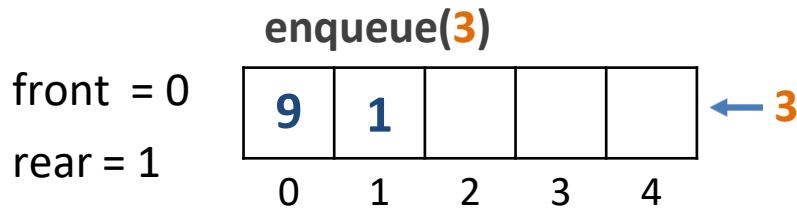
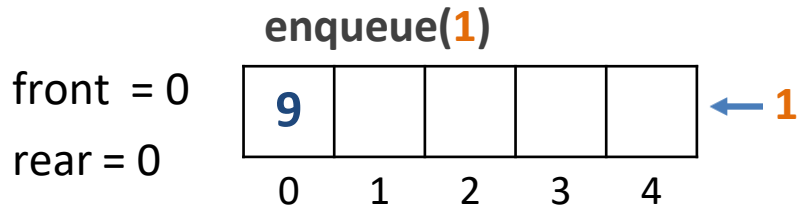
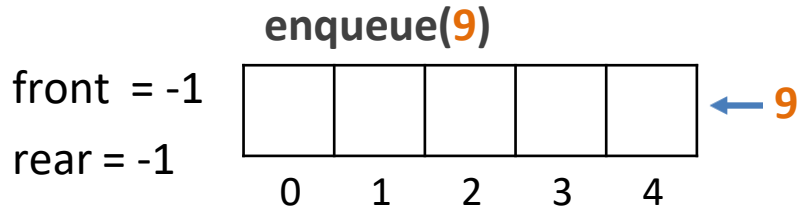
- First in, first out (FIFO) structure (equivalent to Last in, last out (LIFO) structure).
- An ordered list of homogeneous elements in which
  - Insertions take place at one end (REAR).
  - Deletions take place at the other end (FRONT).



# Operations

- Two primary operations:
  - **enqueue()** – Adds an element to the *rear* of a queue.
  - **dequeue()** – Removes the *front* element of the queue.
- Other operations for effective functionality:
  - **isFull()** – Check if queue is full. *OVERFLOW*
  - **isEmpty()** – Check if queue is empty. *UNDERFLOW*
  - **size()** – Returns the number of elements in the queue.
  - **peek()** – Returns the element at the front of the queue.

# Queue – Enqueue

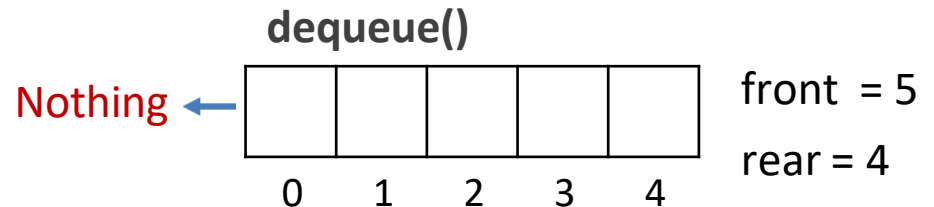
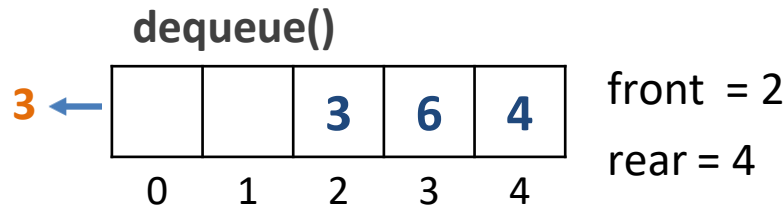
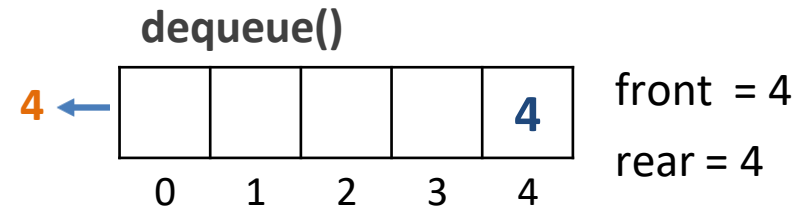
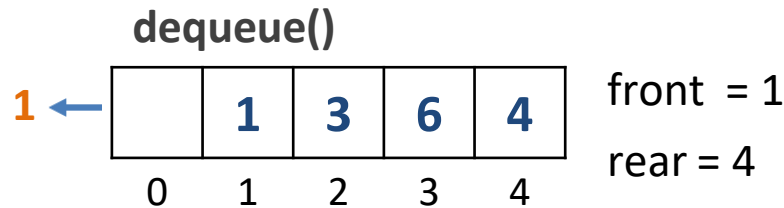
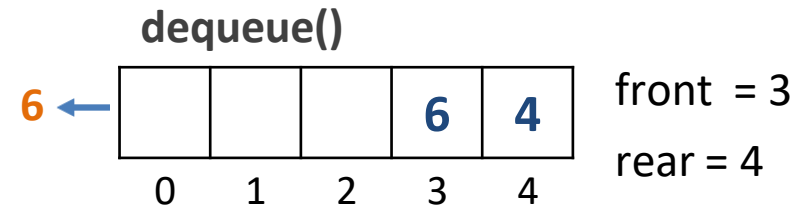
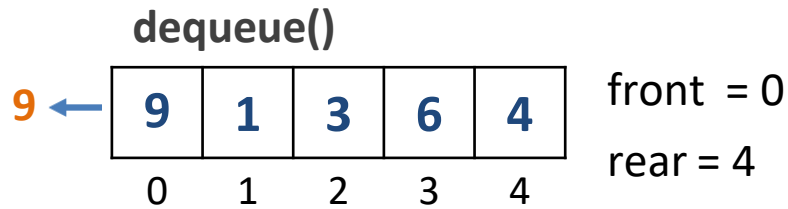


The queue is full, no more elements  
can be added. **OVERFLOW**

OVERFLOW

←

# Queue – Dequeue



The queue is empty, no element  
can be removed. **UNDERFLOW**

UNDERFLOW

# Queue as an ADT

- A queue is an ordered list of elements of same data type.
- Elements are always inserted at one end (rear) and deleted from another end (front).
- Following are its basic operations:
  - $Q = \textit{init}()$  – Initialize an empty queue.
  - $\textit{size}()$  – Returns the number of elements in the queue.
  - $\textit{isEmpty}(Q)$  – Returns "true" if and only if the queue  $Q$  is empty, i.e., contains no elements.

# Contd...

- *isFull(Q)* – Returns "true" if and only if the queue Q has a bounded size and holds the maximum number of elements it can.
- *front(Q)* – Returns the element at the front of the queue Q.
- $Q = \text{enqueue}(Q, x)$  – Inserts an element x at the rear of the queue Q.
- $Q = \text{dequeue}(Q)$  – Removes an element from the front of the queue Q.
- *print(Q)* – Prints the elements of the queue Q from front to rear.

# Implementation

- Using static arrays
  - Realizes queues of a maximum possible size.
  - Front is maintained at the smallest index and rear at the maximum index values in the array.
- Using dynamic linked lists
  - Choose beginning of the list as the front and tail as rear of the queue.



# Static Array Implementation

# Enqueue Operation

- Let,
  - QUEUE be an array with  $N$  locations.
  - FRONT and REAR points to the front and rear of the QUEUE.
  - ITEM is the value to be inserted.
  - 1. If  $(\text{REAR} == N - 1)$
  - 2.       Print[Overflow]
  - 3. Else
  - 4.       If  $(\text{FRONT} == -1 \ \&\& \ \text{REAR} == -1)$
  - 5.             Set  $\text{FRONT} = 0$  and  $\text{REAR} = 0$ .
  - 6.       Else
  - 7.             Set  $\text{REAR} = \text{REAR} + 1$ .
  - 8.        $\text{QUEUE}[\text{REAR}] = \text{ITEM}$ .

# Deque Operation

- Let,
  - QUEUE be an array with N locations.
  - FRONT and REAR points to the front and rear of the QUEUE.
  - ITEM holds the value to be deleted.
  - 1. If (FRONT == -1 || FRONT > REAR)
  - 2.       Print[Underflow]
  - 3. Else
  - 4.       ITEM = QUEUE[FRONT]
  - 5.       Set FRONT = FRONT + 1

# Static Array Implementation

```
1. #define MAXLEN 100
2. typedef struct
3. { int element[MAXLEN];
4.   int front, rear; } queue;
5. queue init ()
6. { queue Q;
7.   Q.front = Q.rear = -1;
8.   return Q; }
9. int size( queue Q )
10. { return ( Q.rear - Q.front + 1 ); }
11. int isEmpty ( queue Q )
12. { return ((Q.front == -1) ||
           (Q.front > Q.rear)); }
13. int isFull ( queue Q )
14. { return (Q.rear == MAXLEN - 1); }
15. int front ( queue Q )
16. { if (isEmpty(Q))
17.   printf("Empty queue\n");
18.   else
19.     return Q.element[Q.front]; }
```

# Contd...

```
20. queue enqueue ( queue Q , int x )
21. {   if (isFull(Q))
22.     printf("OVERFLOW\n");
23.     else if (isEmpty(Q))
24.     {   Q.front = Q.rear = 0;
25.         Q.element[Q.rear] = x;
26.     }
27.     else
28.     {   ++Q.rear;
29.         Q.element[Q.rear] = x;
30.     }
31.     return Q; }

32. queue dequeue ( queue Q )
33. {   if (isEmpty(Q))
34.     printf("UNDERFLOW\n");
35.     else
36.     Q.front++;
37.     return Q; }
```

# Contd...

**38.** void print ( queue Q )

**39.** {     int i;

**40.**       for (i = Q.front; i <= Q.rear; i++)

**41.**           printf("%d ",Q.element[i]); }

**42.** int main ()

**43.** {   queue Q;

**44.**     Q = init();

**45.**     Q = enqueue(Q,5);

**46.**     Q = enqueue(Q,3);

**47.**     Q = dequeue(Q);

**48.**     Q = enqueue(Q,7);

**49.**     Q = dequeue(Q);

**50.**     printf("Current queue : "); print(Q);

**51.**     printf(" with front = %d.\n", front(Q));

**52.**     Q = enqueue(Q,9);

**53.**     Q = enqueue(Q,3);

**54.**     Q = enqueue(Q,1);

**55.**     printf("Current queue : "); print(Q);

**56.**     printf(" with front = %d.\n", front(Q));

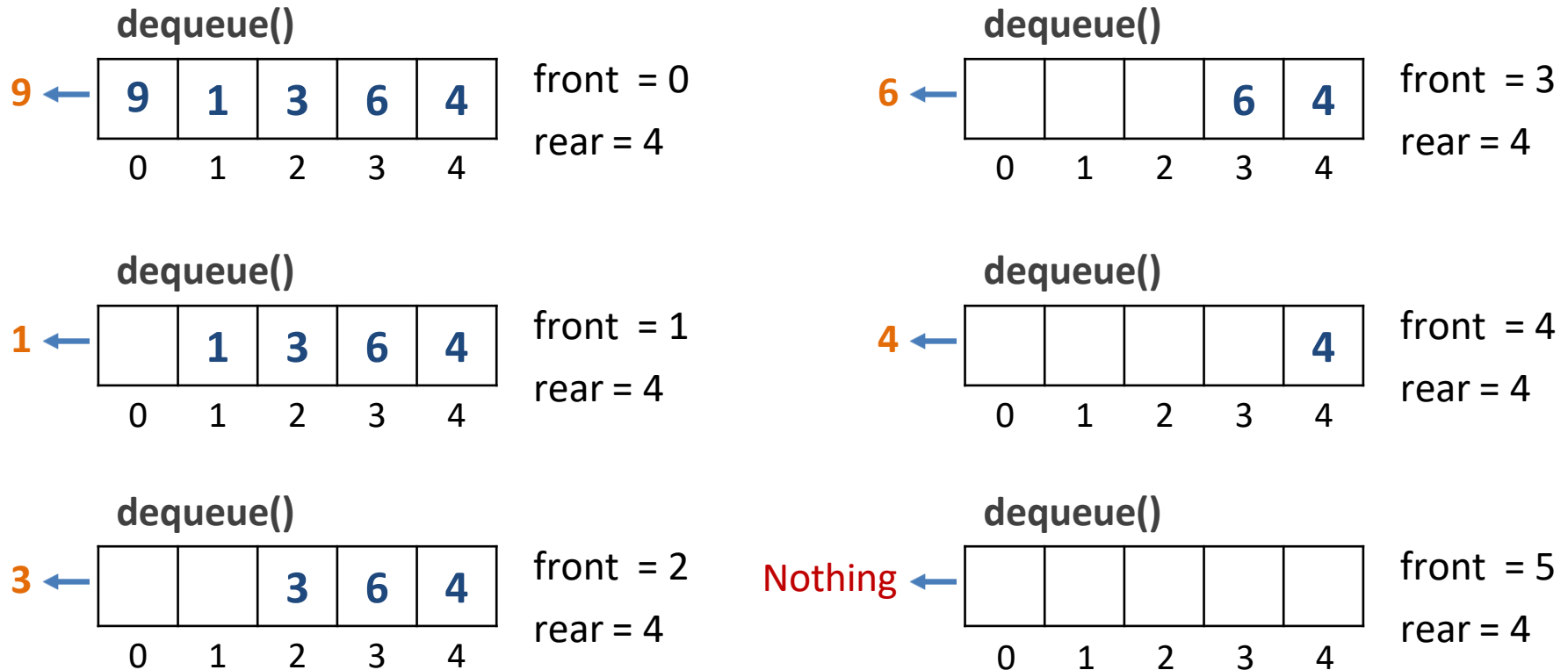
**57.**     printf("Size is %d.",size(Q));

**58.**     return 0;   }

# Queues

## Variants

# Problem with Simple Queues



UNDERFLOW

The queue is empty, still no  
element can be added as  
**REAR = N-1 (Queue Full)**



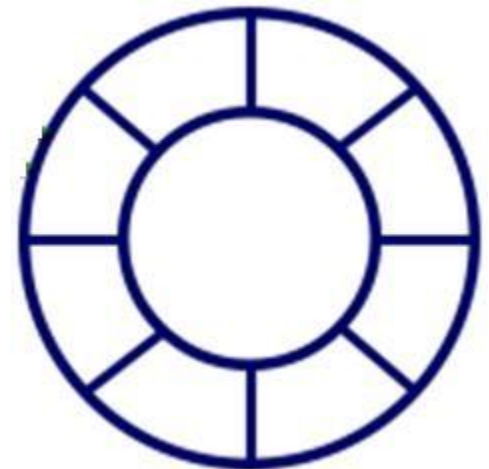
# Circular Queues

- The front and rear ends of a queue are joined to make the queue circular.
- Also known as circular buffer, circular queue, cyclic buffer or ring buffer.
- Overflow

$\text{front} == (\text{rear} + 1) \% \text{MAXLEN}$

- Underflow

$(\text{front} == \text{rear}) \ \&\& \ (\text{rear} == -1)$



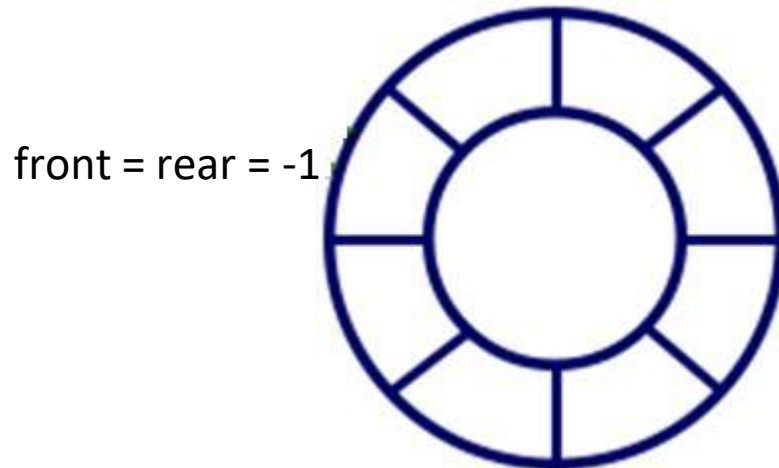
# Contd...

- Overflow

$\text{front} == (\text{rear} + 1) \% 8$

- Underflow

$(\text{front} == \text{rear}) \ \&\& \ (\text{rear} == -1)$



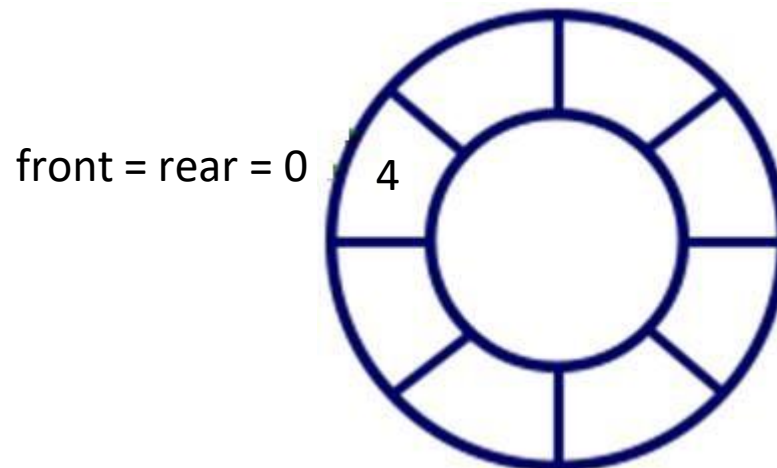
# Contd...

- Overflow

$\text{front} == (\text{rear} + 1) \% 8$

- Underflow

$(\text{front} == \text{rear}) \ \&\& \ (\text{rear} == -1)$



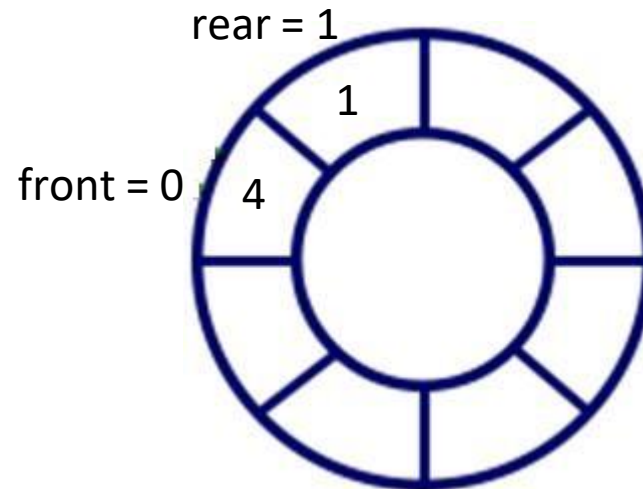
# Contd...

- Overflow

$\text{front} == (\text{rear} + 1) \% 8$

- Underflow

$(\text{front} == \text{rear}) \ \&\& \ (\text{rear} == -1)$



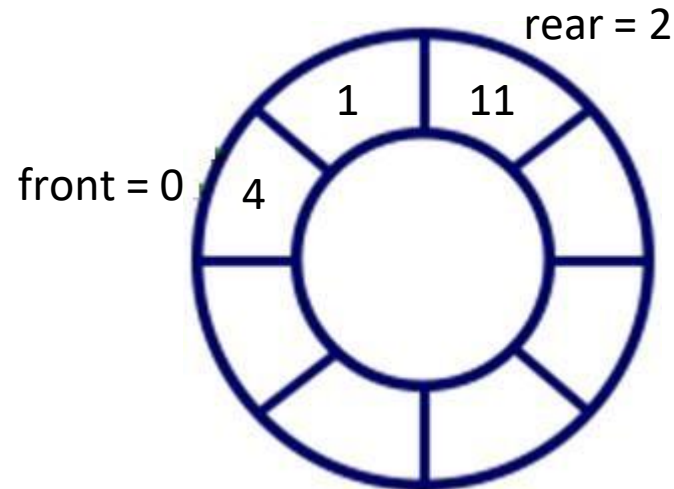
# Contd...

- Overflow

$\text{front} == (\text{rear} + 1) \% 8$

- Underflow

$(\text{front} == \text{rear}) \ \&\& \ (\text{rear} == -1)$



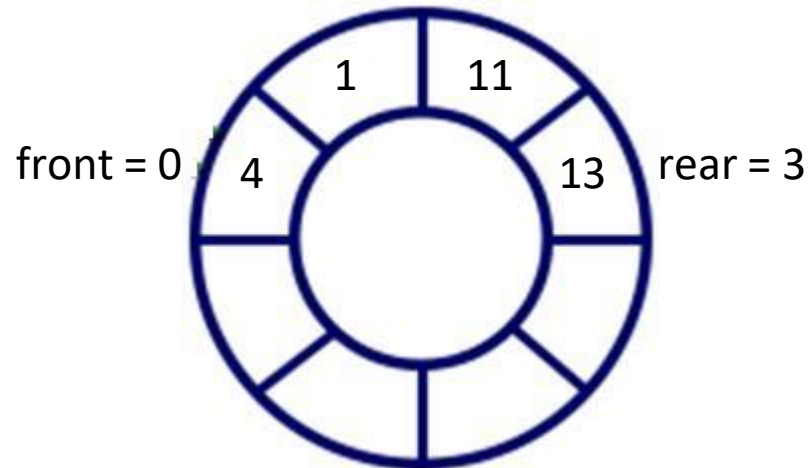
# Contd...

- Overflow

$\text{front} == (\text{rear} + 1) \% 8$

- Underflow

$(\text{front} == \text{rear}) \ \&\& \ (\text{rear} == -1)$



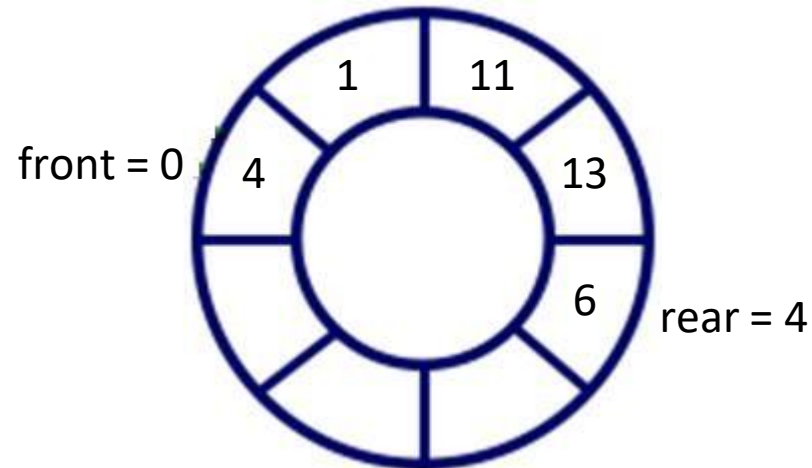
# Contd...

- Overflow

$\text{front} == (\text{rear} + 1) \% 8$

- Underflow

$(\text{front} == \text{rear}) \ \&\& \ (\text{rear} == -1)$



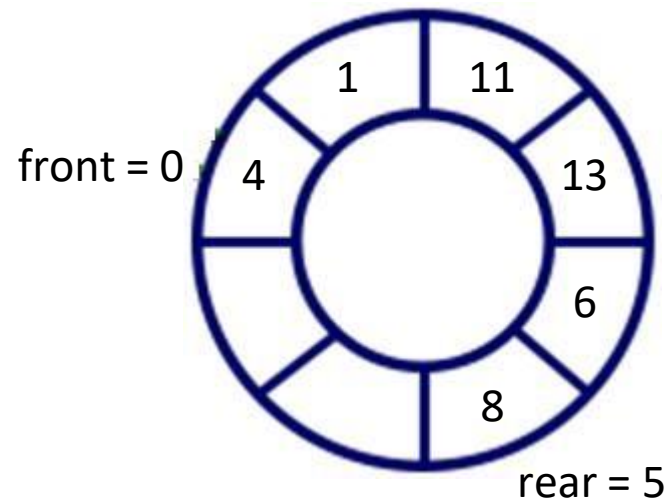
# Contd...

- Overflow

$\text{front} == (\text{rear} + 1) \% 8$

- Underflow

$(\text{front} == \text{rear}) \ \&\& \ (\text{rear} == -1)$





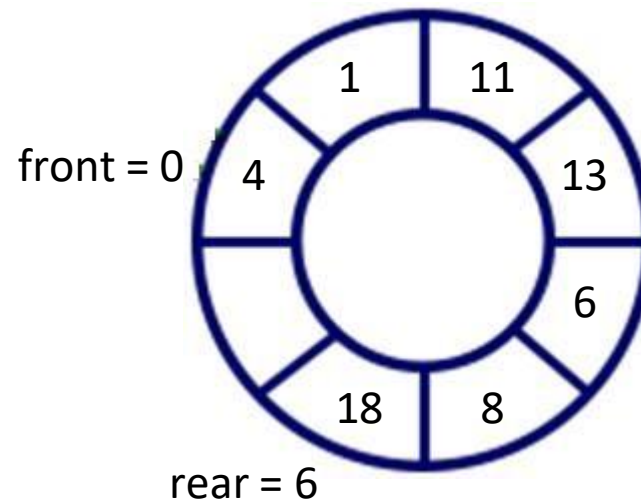
# Contd...

- Overflow

$\text{front} == (\text{rear} + 1) \% 8$

- Underflow

$(\text{front} == \text{rear}) \ \&\& \ (\text{rear} == -1)$



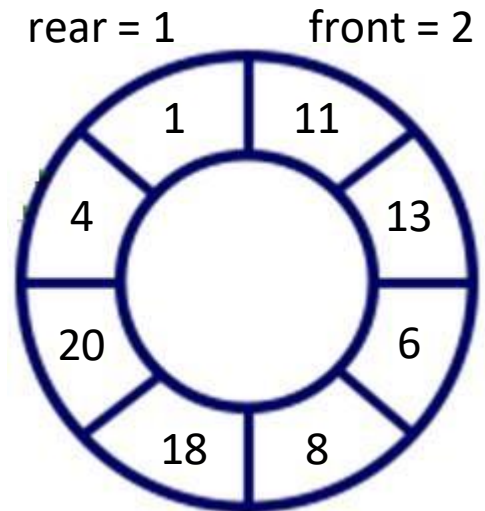
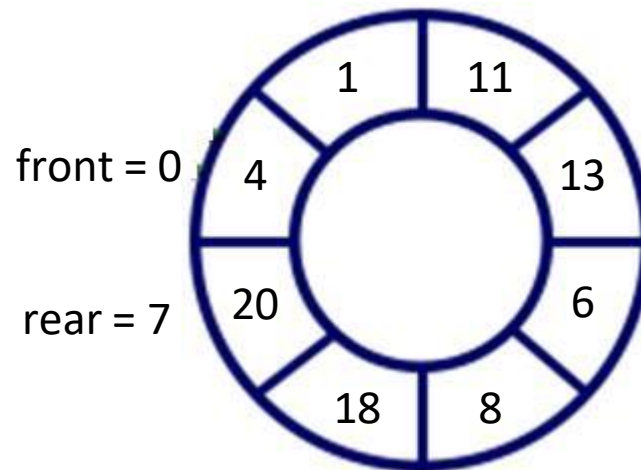
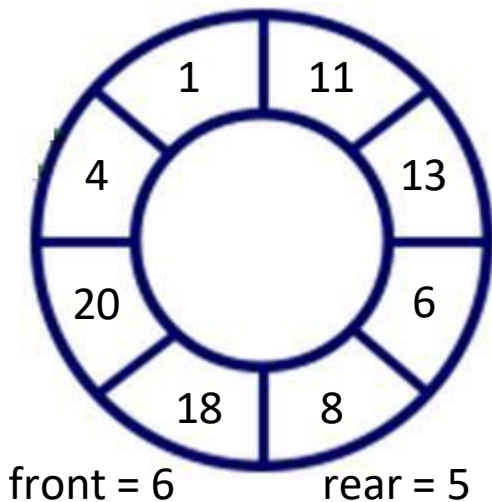
# Contd...

- **Overflow**

**$\text{front} == (\text{rear} + 1) \% 8$**

- Underflow

$(\text{front} == \text{rear}) \ \&\& \ (\text{rear} == -1)$



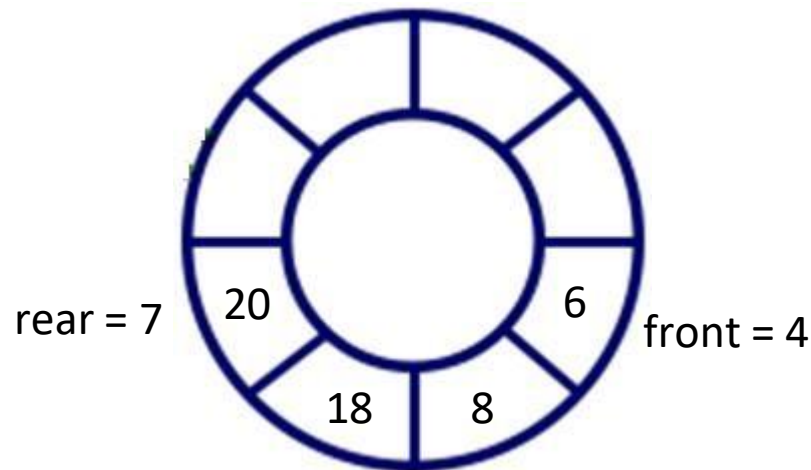
# Contd...

- Overflow

$\text{front} == (\text{rear} + 1) \% 8$

- Underflow

$(\text{front} == \text{rear}) \ \&\& \ (\text{rear} == -1)$



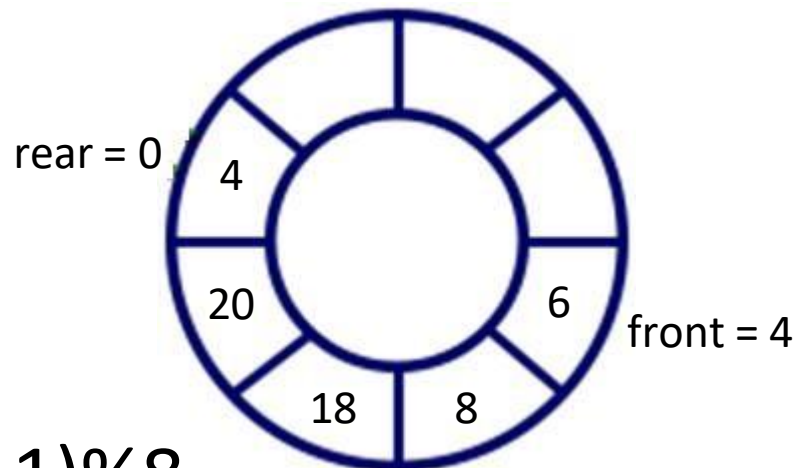
# Contd...

- Overflow

$\text{front} == (\text{rear} + 1) \% 8$

- Underflow

$(\text{front} == \text{rear}) \ \&\& \ (\text{rear} == -1)$



$\text{REAR} = (\text{REAR} + 1) \% 8$

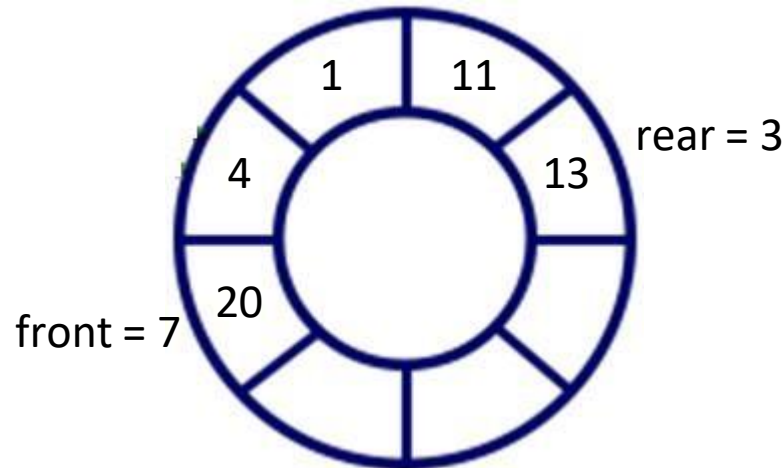
# Contd...

- Overflow

$\text{front} == (\text{rear} + 1) \% 8$

- Underflow

$(\text{front} == \text{rear}) \ \&\& \ (\text{rear} == -1)$



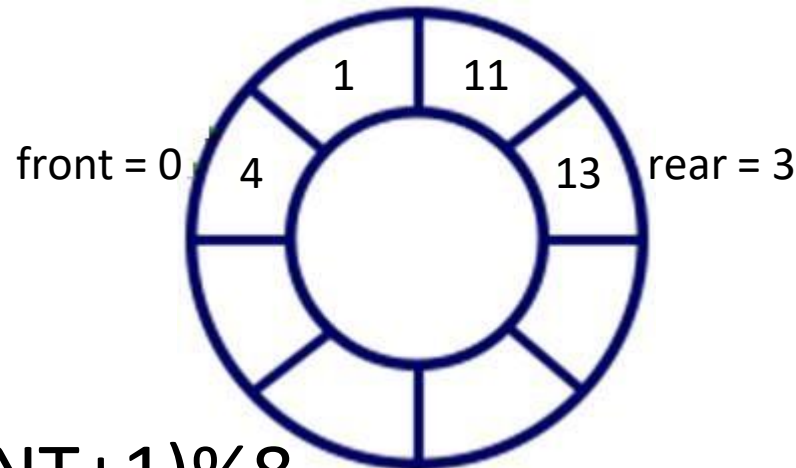
# Contd...

- Overflow

$\text{front} == (\text{rear} + 1) \% 8$

- Underflow

$(\text{front} == \text{rear}) \ \&\& \ (\text{rear} == -1)$



$\text{FRONT} = (\text{FRONT} + 1) \% 8$

# Enqueue Operation

- Let,
  - QUEUE be an array with MAX locations.
  - FRONT and REAR points to the front and rear of the QUEUE.
  - ITEM is the value to be inserted.
- 1. if  $(\text{FRONT} == (\text{REAR} + 1) \% \text{MAX})$
- 2.           Print [Overflow]
- 3. else
- 4.           Set  $\text{REAR} = (\text{REAR} + 1) \% \text{MAX}$
- 5.           Set  $\text{QUEUE}[\text{REAR}] = \text{element}$
- 6.           If  $(\text{FRONT} == -1)$
- 7.                 Set  $\text{FRONT} = 0$

# Dequeue Operation

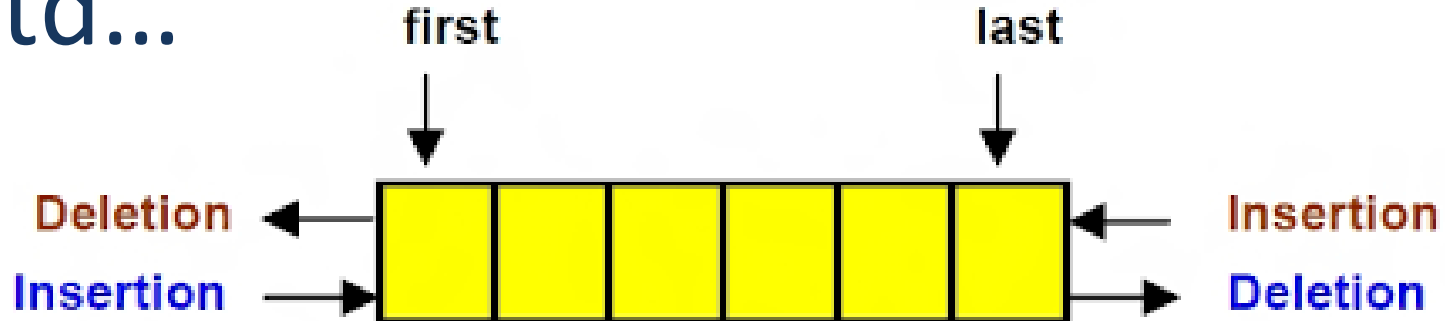
- Let,
    - QUEUE be an array with MAX locations.
    - FRONT and REAR points to the front and rear of the QUEUE.
    - ITEM holds the value to be deleted.
1. if ((FRONT == REAR) && (REAR == -1))
  2.       Print [Underflow]
  3. else
  4.       ITEM = Q[FRONT]
  5.       If (FRONT == REAR)
  6.             FRONT = REAR = -1
  7.       Else
  8.             FRONT = (FRONT + 1) % MAX



# Deque

- **Double-ended queue.**
- Generalization of queue data structure.
- Elements can be added to or removed from either of the two ends.
- A hybrid linear structure that provides all the capabilities of stacks and queues in a single data structure.
- Does not require the LIFO and FIFO orderings.

# Contd...



## Types

- Input-restricted deque.
  - Deletion can be made from both ends, but insertion can be made at one end only.
- Output-restricted deque.
  - Insertion can be made at both ends, but deletion can be made from one end only.

# Priority Queues

- Another variant of queue data structure.
- Each element has an associated priority.
- Insertion may be performed based on the priority.
- Deletion is performed based on the priority.
- Elements having the same priority are served or deleted according to first come first serve order.
- Two types:
  - Min-priority queues (Ascending priority queues)
  - Max-priority queues (Descending priority queues)