# Unit-3

Queues

- Introduction and Definition
- Representation
- Operation on Queues
- Types of Queues: Dequeue, Circular Queue, Priority Queue
- Applications of Queue

# Queue Introduction

- Queues can be easily represented using linear arrays.
- Every queue has front and rear variables that point to the position from where deletions and insertions can be done.


**Figure 8.1** Queue


**Figure 8.2** Queue after insertion of a new element


**Figure 8.3** Queue after deletion of an element

# Insert Array Element

```
Step 1: IF REAR = MAX-1
            Write OVERFLOW
            Goto step 4
        [END OF IF]
Step 2: IF FRONT = -1 and REAR = -1
            SET FRONT = REAR = 0
        ELSE
            SET REAR = REAR + 1
        [END OF IF]
Step 3: SET QUEUE[REAR] = NUM
Step 4: EXIT
```

- For ex. front = 0 and rear = 5.
- Add =45
- Rear=Rear+1
- Front=0 and rear=6
- Every time a new element has to be added, we repeat the same procedure

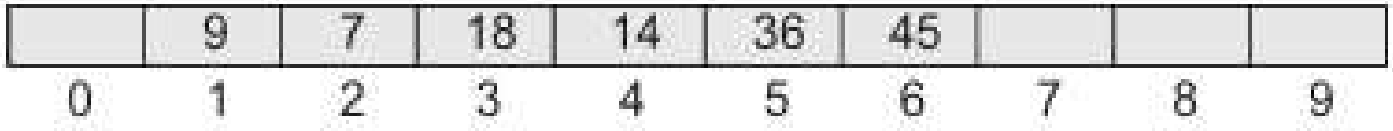
# Deletion in Queue

```
Step 1: IF FRONT = -1 OR FRONT > REAR
            Write UNDERFLOW
        ELSE
            SET VAL = QUEUE[FRONT]
            SET FRONT = FRONT + 1
        [END OF IF]
Step 2: EXIT
```

Deletions are done from front end of the queue.
Front=front+1
front = 1 and rear = 6.

Before inserting an element in a queue, we must check for overflow/underflow conditions.
An overflow →
Insert an element into a queue that is already full.
When rear = MaX – 1, where MaX is the size of the queue

An underflow→
Delete an element from a queue that is already empty. If front = –1 and rear = –1, it means there is no element in the queue



# Types of Queue

- A queue data structure can be classified into the following types:
    1. Circular Queue
    2. Deque
    3. Priority Queue

# Circular Queue

**1. Efficient Memory Utilization**

- **Linear Queue Problem**: Once the rear reaches the end of the array, no more elements can be added—even if there's space at the front due to dequeued elements.
- **Circular Queue Solution**: The rear wraps around to the front, using up that freed space.

Example:

•After enqueue: [10, 20, 30, 40, 50]

•After dequeue two elements: [_, _, 30, 40, 50]

•In a **linear queue**, rear is at index 4, so no more insertions.
•In a **circular queue**, rear wraps to index 0, allowing insertion at freed spots.

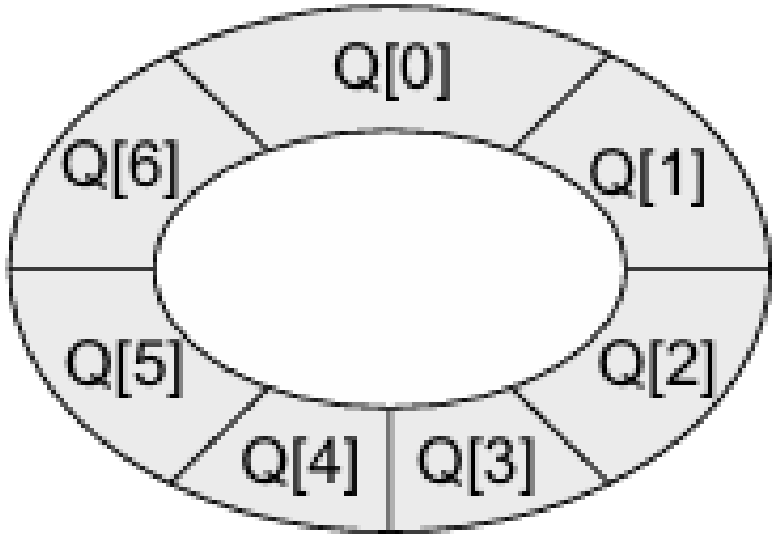
**2. Better Performance for Continuous Operations**

Circular queues are ideal for systems where data is constantly added and removed—like streaming buffers, CPU scheduling, or network routers.

**3. No Need to Shift Elements**

•In a linear queue, after multiple dequeues, you may need to shift elements to the front to make space.

•Circular queues eliminate this overhead.



# Circular insert Algorithm

```
Step 1: IF (FRONT == 0 AND REAR == MAX - 1) OR
(FRONT == REAR + 1)
        PRINT "OVERFLOW"
        GOTO Step 5
      [END OF IF]
Step 2: IF FRONT == -1
        SET FRONT = REAR = 0
      ELSE IF REAR == MAX - 1
        SET REAR = 0
      ELSE
        SET REAR = REAR + 1
      [END OF IF]
Step 3: SET QUEUE[REAR] = NUM
Step 4: PRINT "Element Inserted"
Step 5: EXIT
```

MAX = 5

QUEUE = [10, 20, 30, _, _]

FRONT = 0, REAR = 2

Insert 40:

- REAR = 3, insert at index 3 ⟶ QUEUE = [10, 20, 30, 40, _]

Insert 50:

- REAR = 4, insert at index 4 ⟶ QUEUE = [10, 20, 30, 40, 50]

Insert 60:

- Now REAR = 4, FRONT = 0 ⟶ next REAR = 0 (wrap around)
- Insert at index 0 ⟶ QUEUE = [60, 20, 30, 40, 50]
- FRONT == REAR + 1, so next insert will cause **overflow**

# Circular Delete Algorithm

```
Step 1: IF FRONT == -1
          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]
Step 4: EXIT
```

- MAX = 5
- QUEUE = [10, 20, 30, _, _]
- FRONT = 0, REAR = 2

First Deletion

- VAL = QUEUE[FRONT] = 10
- FRONT = 1
- Queue becomes: [_, 20, 30, _, _]

Second Deletion

- VAL = 20
- FRONT = 2
- Queue becomes: [_, _, 30, _, _]

Third Deletion

- Now FRONT == REAR == 2 → only one element left
- VAL = 30
- Set FRONT = REAR = -1 → queue is now empty
- Queue becomes: [_, _, _, _, _]

Fourth Deletion (Underflow)

- FRONT == -1 → queue is empty
- Output: "UNDERFLOW"

# Deque

- A deque is a linear data structure that supports:
  - Insertions at both ends
  - Deletions at both ends
- Unlike a regular queue (which follows FIFO: First In First Out), a deque gives you more flexibility. It can behave like a queue or a stack, depending on how you use it.
- Two variants of a double-ended queue.

1. Input restricted deque In this dequeue, insertions can be done only at one of the ends, while deletions can be done from both ends.

2. Output restricted deque In this dequeue, deletions can be done only at one of the ends, while insertions can be done on both ends.

# Dequeue: Insert

**Insert at Rear**

```
Step 1: IF (FRONT == 0 AND REAR == MAX - 1) OR
(FRONT == REAR + 1)
        WRITE "OVERFLOW"
        EXIT
Step 2: IF FRONT == -1
        SET FRONT = REAR = 0
      ELSE IF REAR == MAX - 1
        SET REAR = 0
      ELSE
        SET REAR = REAR + 1
Step 3: SET DEQUE[REAR] = VALUE
Step 4: EXIT
```

**Insert at Front**

```
Step 1: IF (FRONT == 0 AND REAR == MAX -
1) OR (FRONT == REAR + 1)
        WRITE "OVERFLOW"
        EXIT
Step 2: IF FRONT == -1
        SET FRONT = REAR = 0
      ELSE IF FRONT == 0
        SET FRONT = MAX - 1
      ELSE
        SET FRONT = FRONT - 1
Step 3: SET DEQUE[FRONT] = VALUE
Step 4: EXIT
```

# Deque :Deletion

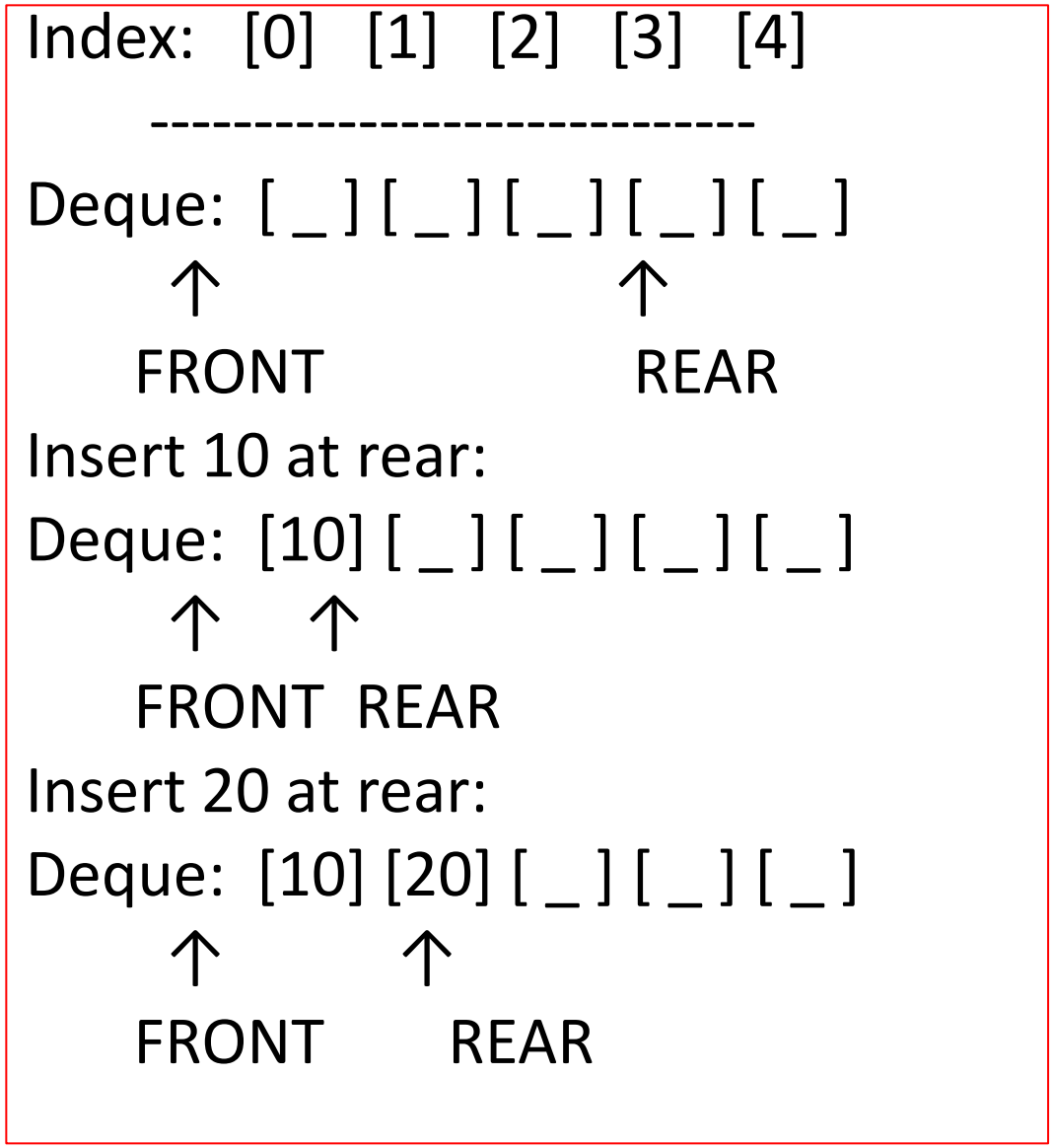
**Delete from Front**

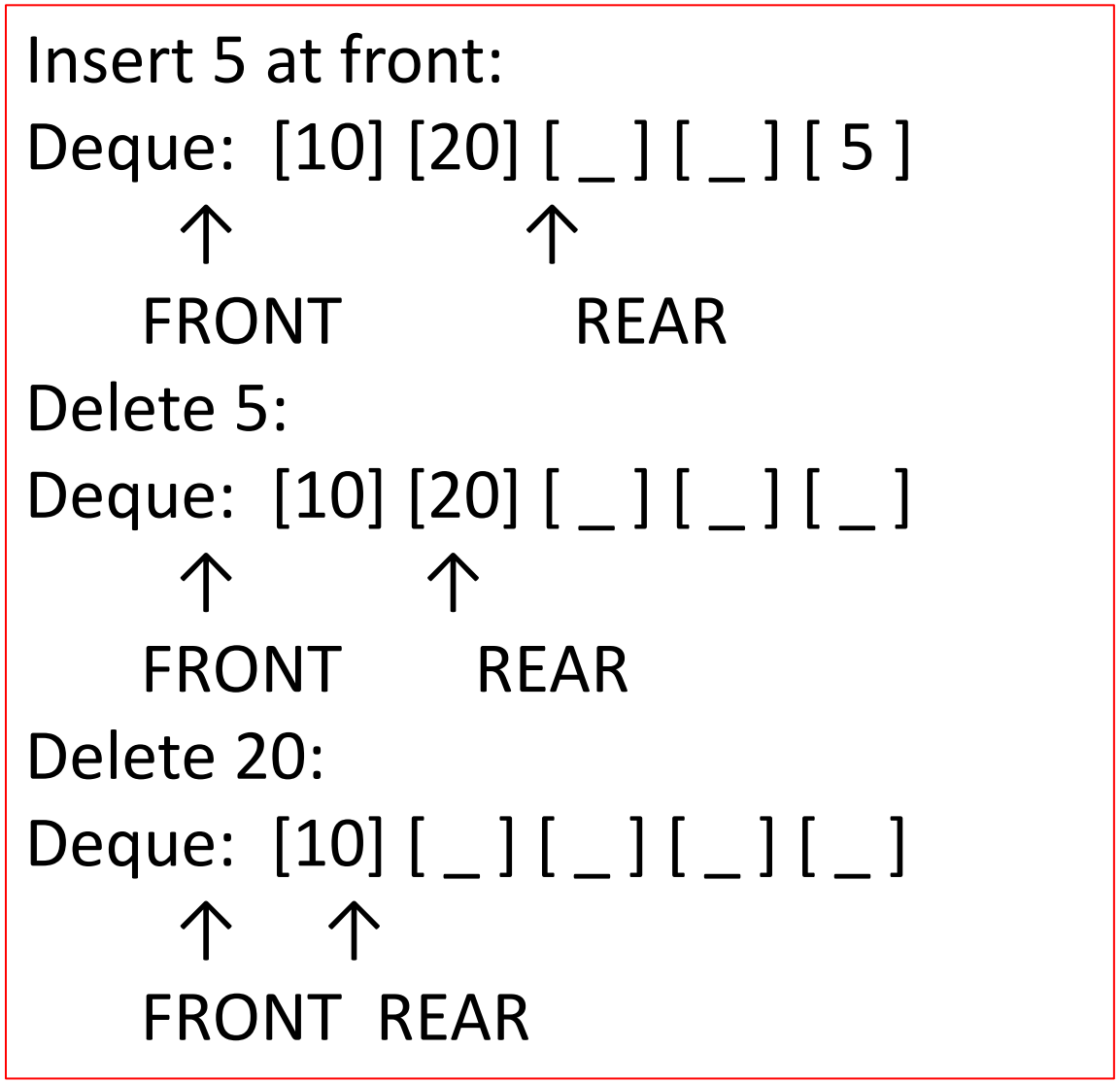
```
Step 1: IF FRONT == -1
        WRITE "UNDERFLOW"
        EXIT
Step 2: SET VALUE = DEQUE[FRONT]
Step 3: IF FRONT == REAR
        SET FRONT = REAR = -1
      ELSE IF FRONT == MAX - 1
        SET FRONT = 0
      ELSE
        SET FRONT = FRONT + 1
Step 4: EXIT
```

**Delete from Rear**

```
Step 1: IF FRONT == -1
        WRITE "UNDERFLOW"
        EXIT
Step 2: SET VALUE = DEQUE[REAR]
Step 3: IF FRONT == REAR
        SET FRONT = REAR = -1
      ELSE IF REAR == 0
        SET REAR = MAX - 1
      ELSE
        SET REAR = REAR - 1
Step 4: EXIT
```

# Examples


Index: [0] [1] [2] [3] [4]
Deque: [ _ ] [ _ ] [ _ ] [ _ ] [ _ ]
FRONT REAR
Insert 10 at rear:
Deque: [10] [ _ ] [ _ ] [ _ ] [ _ ]
FRONT REAR
Insert 20 at rear:
Deque: [10] [20] [ _ ] [ _ ] [ _ ]
FRONT REAR


Insert 5 at front:
Deque: [10] [20] [ _ ] [ _ ] [ 5 ]
FRONT REAR
Delete 5:
Deque: [10] [20] [ _ ] [ _ ] [ _ ]
FRONT REAR
Delete 20:
Deque: [10] [ _ ] [ _ ] [ _ ] [ _ ]
FRONT REAR

# Priority Queue

- Special type of queue where each element is assigned a **priority**, and elements are served based on their priority—not just their order of arrival.

Each element has:

- A value
- A priority level

Operations include:

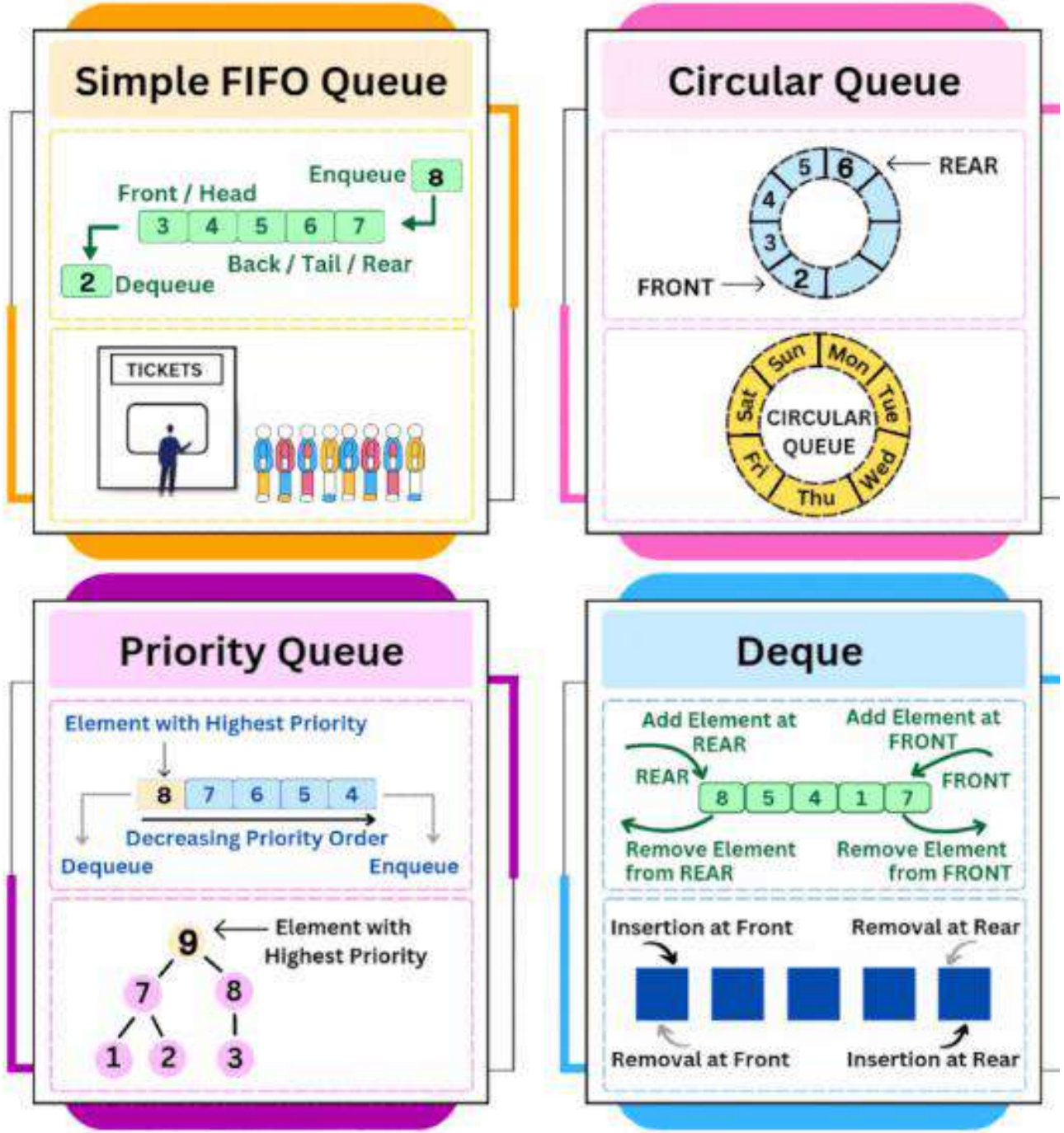
- insert(value, priority) → Adds an element with a given priority
- remove() → Removes the element with the highest priority
- peek() → Views the highest-priority element without removing it

# Example

insert("Task A", priority = 3)
insert("Task B", priority = 1)
insert("Task C", priority = 2)

max-priority queue (higher number = higher priority
→ Task A (priority 3)
→ Task C (priority 2)
→ Task B (priority 1)

min-priority queue (lower number = higher priority)
→ Task B (priority 1)
→ Task C (priority 2)
→ Task A (priority 3)

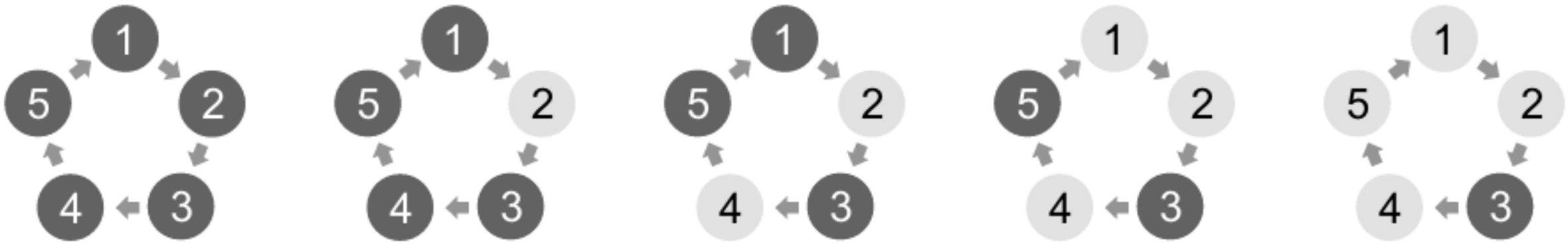

Simple FIFO Queue
Enqueue
8
Front / Head
3
4
5
6
7
Back / Tail / Rear
2
Dequeue
TICKETS
Circular Queue
REAR
5
6
4
3
2
FRONT
Sun
Mon
Tue
Wed
Thu
Fri
Sat
CIRCULAR QUEUE
Priority Queue
Element with Highest Priority
8
7
6
5
4
Decreasing Priority Order
Dequeue
Enqueue
9
Element with Highest Priority
7
8
1
2
3
Deque
Add Element at REAR
Add Element at FRONT
REAR
FRONT
8
5
4
1
7
Remove Element from REAR
Remove Element from FRONT
Insertion at Front
Removal at Rear
Removal at Front
Insertion at Rear

# Application of Queue

- Queues are widely used as waiting lists for a single shared resource like printer, disk, CPU.
- Queues are used to transfer data asynchronously (data not necessarily received at same rate as sent) between two processes (IO buffers), e.g., pipes, file IO, sockets.
- Queues are used as buffers on MP3 players and portable CD players, iPod playlist.
- Queues are used in Playlist for jukebox to add songs to the end, play from the front of the list.
- Queues are used in operating system for handling interrupts. When programming a real-time system that can be interrupted, for example, 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.

# Queue Application: Josephus Problem

- In Josephus problem, n people stand in a circle waiting to be executed.
- The 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’.
- If there are n number of people and a number k which indicates that k–1 people are skipped and k–th person in the circle is eliminated, then the problem is to choose a position in the initial circle so that the given person becomes the winner.




For example, if there are 5 (n) people and every second (k) person is eliminated, then first the person at position 2 is eliminated followed by the person at position 4 followed by person at position 1 and finally the person at position 5 is eliminated. Therefore, the person at position 3 becomes the winner.