# Data Structures and Algorithms

# QUEUES

# Queues

- A **Queue** is a special kind of list, where items are inserted at one end (**the rear**) And deleted at the other end (**the front**).

- Accessing the elements of queues follows a First In,First Out (FIFO) order.
- Example
  - Like customers standing in a check-out line in a store, the first customer in is the first customer served.

# Common Operations on Queues

- **MAKENULL:**
- **FRONT*(Q)*:** Returns the first element on Queue Q.
- **ENQUEUE(*x*,*Q*):** Inserts element x at the end of Queue Q.
- **DEQUEUE(*Q*):** Deletes the first element of *Q.*
- **ISEMPTY(*Q*):** Returns true if and only if Q is an empty queue.
- **ISFULL(Q):** Returns true if and only if Q is full.

# Enqueue and Dequeue

- Primary queue operations: Enqueue and Dequeue

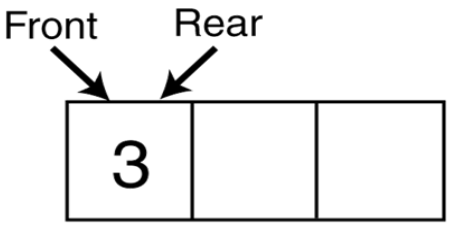
- **Enqueue** – insert an element at the rear of the queue.
- **Dequeue** – remove an element from the front of the queue.

# Queues Implementations

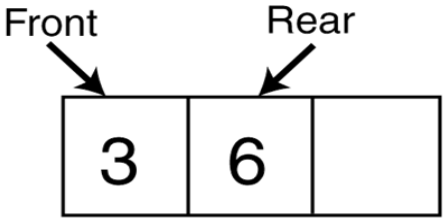
- Static
  - Queue is implemented by an array, and size of queue remains fix
- Dynamic
  - A **queue** can be **implemented** as a **linked list**, and *expand* or *shrink* with each *enqueue* or *dequeue* operation.

# Static Implementation of Queues

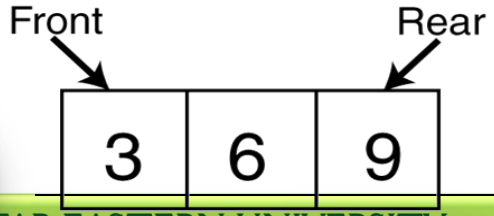
Enqueue(3);



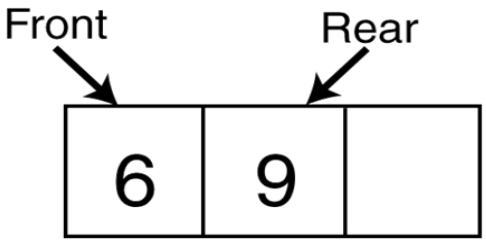

Enqueue(6);



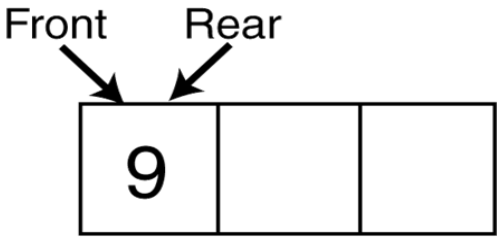

Enqueue(9);



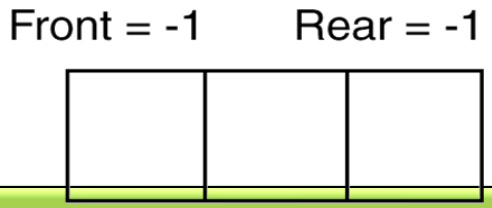

Dequeue();




Dequeue();

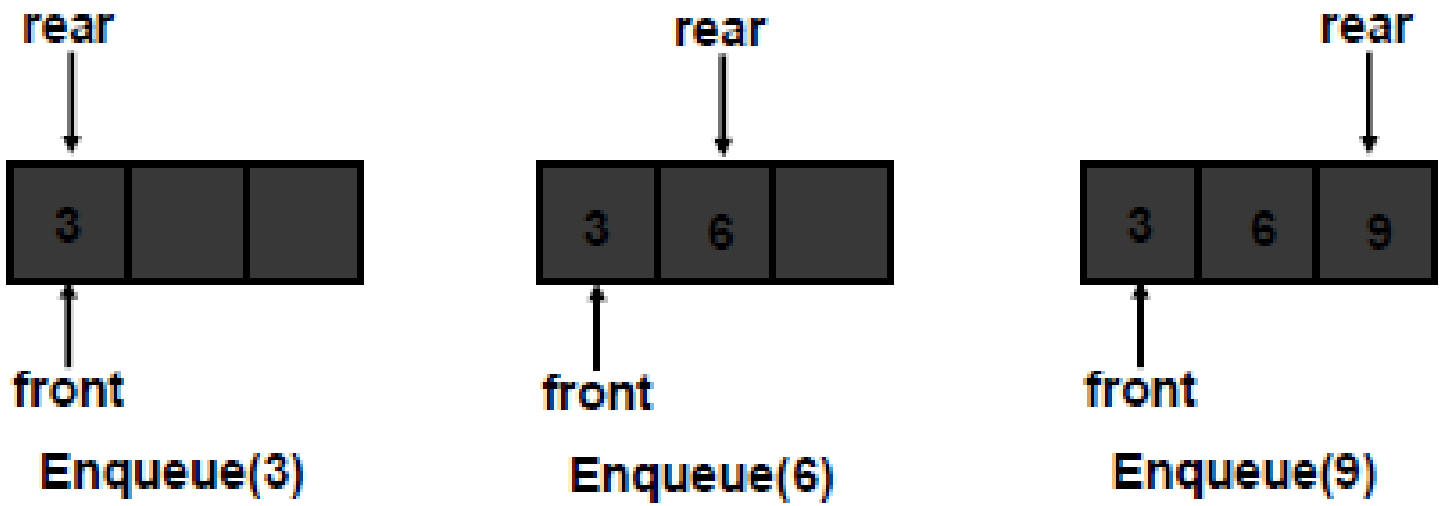



Dequeue();



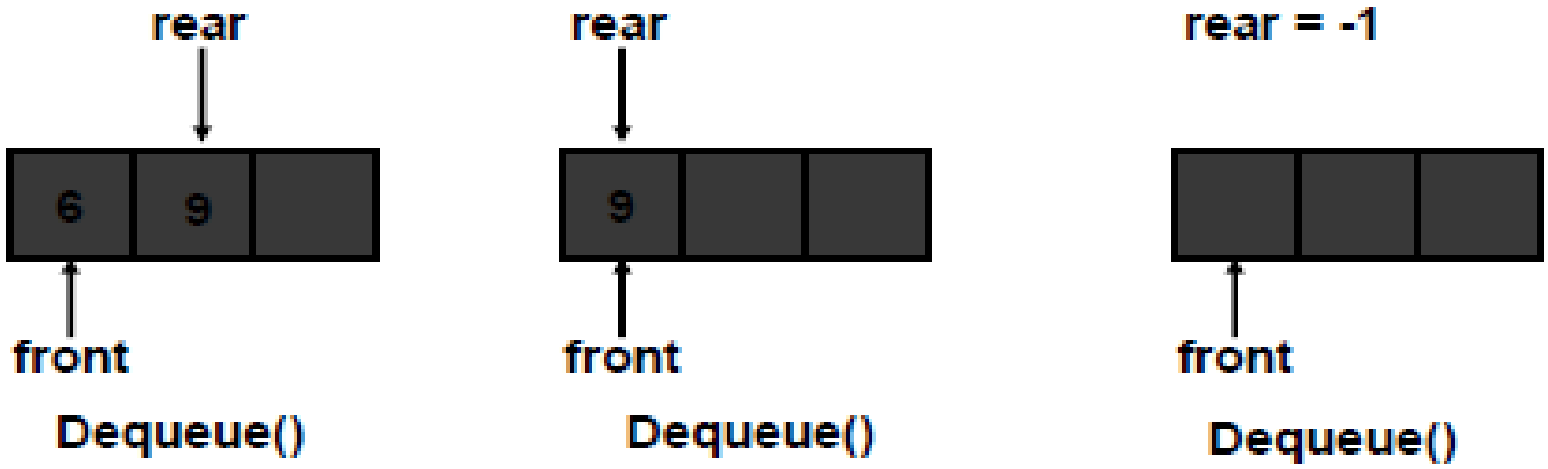
# Static Implementation of Queue

- Static implementation is done using arrays
- In this implementation, we should know the exact number of elements to be stored in the queue.
- When enqueuing, the front index is always fixed and the rear index moves forward in the array.



# Static Implementation of Queue

- When dequeuing, the front index is fixed, and the element at the front of the queue is removed. Move all the elements after it by one position. (Inefficient!!!)



# Static Implementation of Queue

- **A better way**
  - When an item is enqueued, the rear index moves forward.
  - When an item is dequeued, the front index also moves forward by one element
- **Example:**

**X = occupied, and O = empty**

- (front) XXXXOOOOO (rear)
- OXXXXOOOO (after 1 dequeue, and 1 enqueue)
- OOXXXXXOO (after another dequeue, and 2 enqueues)
- OOOOXXXXX (after 2 more dequeues, and 2 enqueues)
- **The problem here is that the rear index cannot move beyond the last element in the array.**

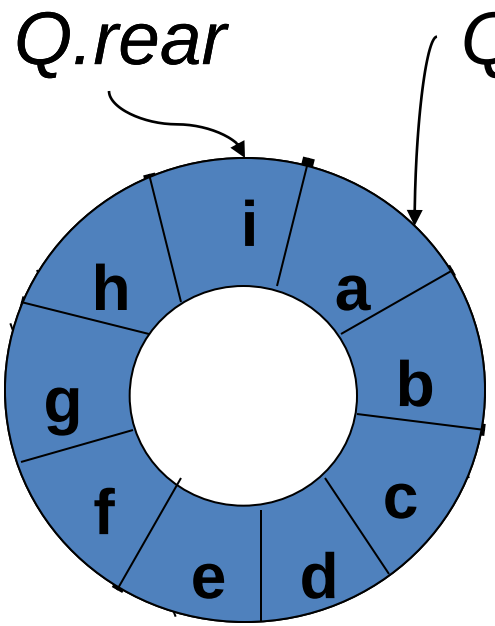
# Static Implementation of Queue

- To overcome the above limitation, we can use **circular array implementation of queues.**
- In this implementation, first position follows the last.

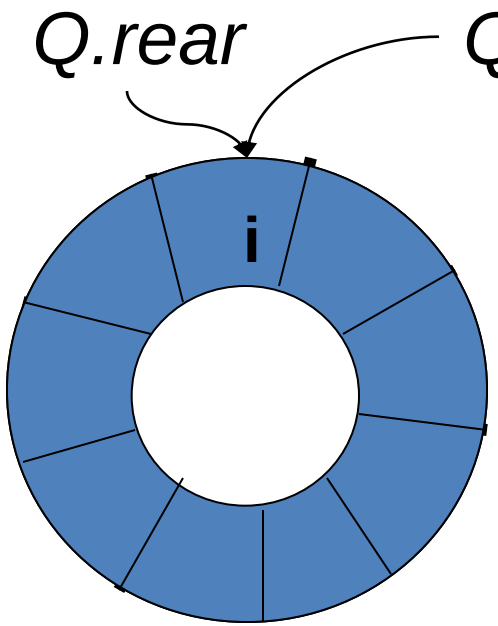
- **When an element moves past the end of a circular array, it wraps around to the beginning, e.g**
  - OOOOO7963 ->4OOOO7963 (after Enqueue(4))
  - After Enqueue(4), the rear index moves from 3 to 4.

- **How to detect an empty or full queue, using a circular array algorithm?**
  - Use a counter of the number of elements in the queue.

# Circular Queue

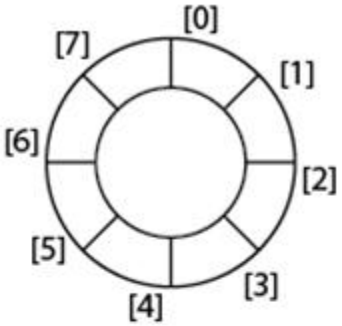



A Completely Filled Queue

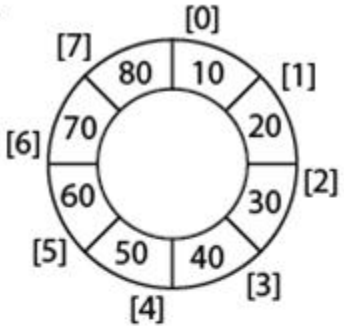



A Queue with Only 1 Element

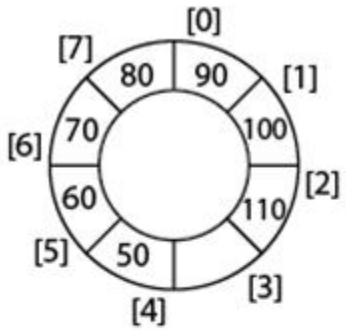
size: 8 (from index 0 to 7)



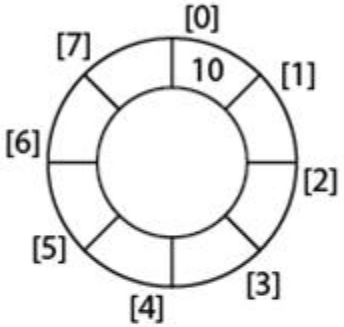

queue is empty
front: -1
rear: -1



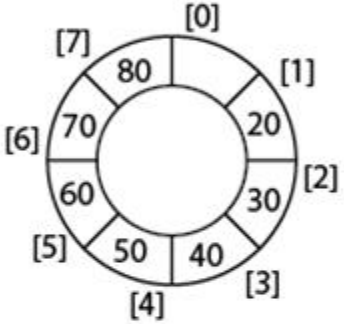

enqueue(90)
queue is full
front: 0
rear: 7



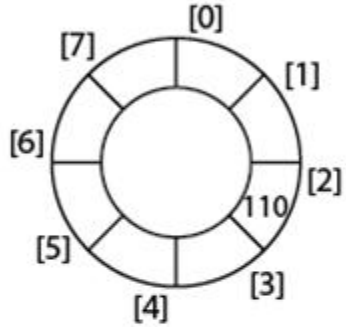

enqueue(100)
enqueue(110)
front: 4
rear: 2



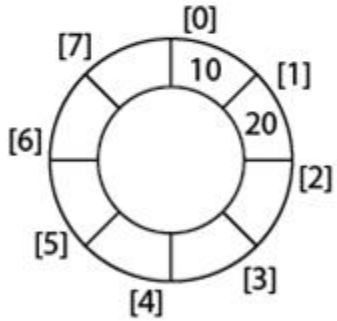

enqueue(10)
front: 0
rear: 0



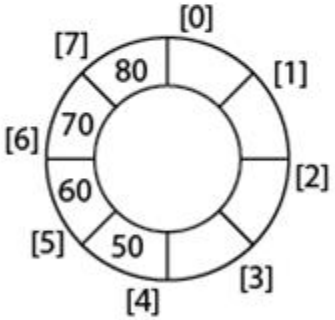

dequeue( )
front: 1
rear: 7



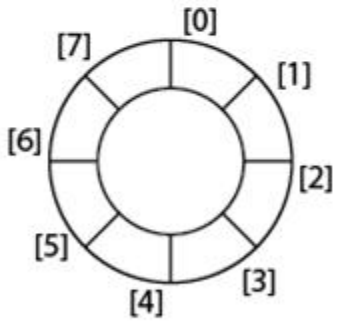

dequeue( )
dequeue( )
dequeue( )
dequeue( )
dequeue( )
dequeue( )
front: 2
rear: 2



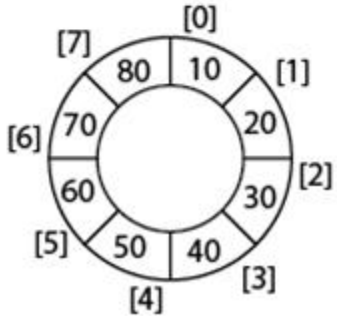

enqueue(20)
front: 0
rear: 1



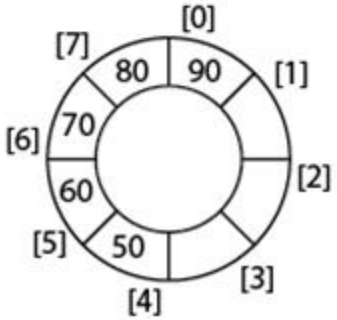

dequeue( )
dequeue( )
dequeue( )
front: 4
rear: 7



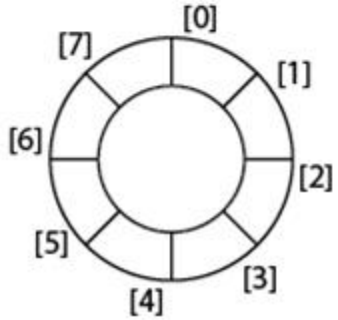

dequeue( )
front: -1
rear: -1




enqueue(30)
enqueue(40)
enqueue(50)
enqueue(60)
enqueue(70)
enqueue(80)
front: 0
rear: 7

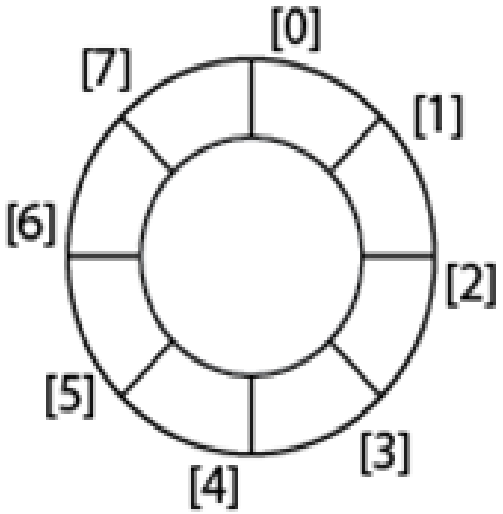



enqueue(90)
front: 4
rear: 0

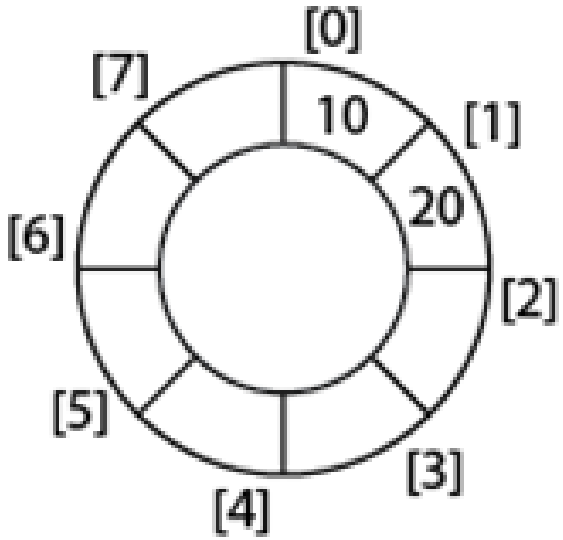



dequeue( )
queue is empty
front: -1
rear: -1

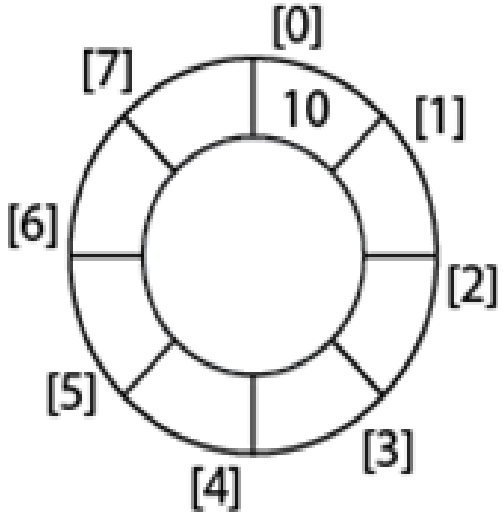
size: 8 (from index 0 to 7)



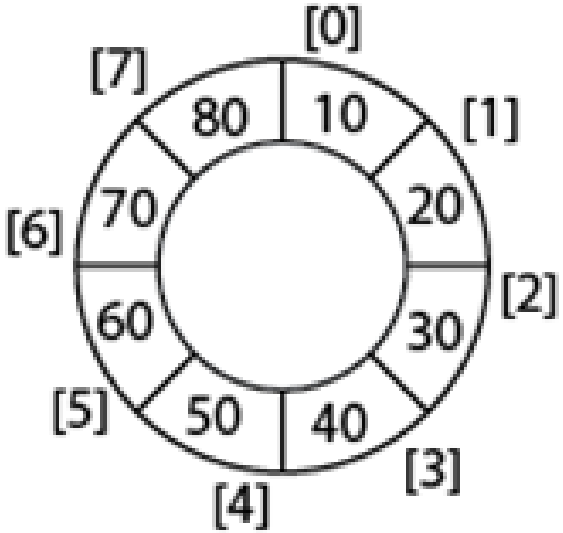

queue is empty
front: -1
rear: -1



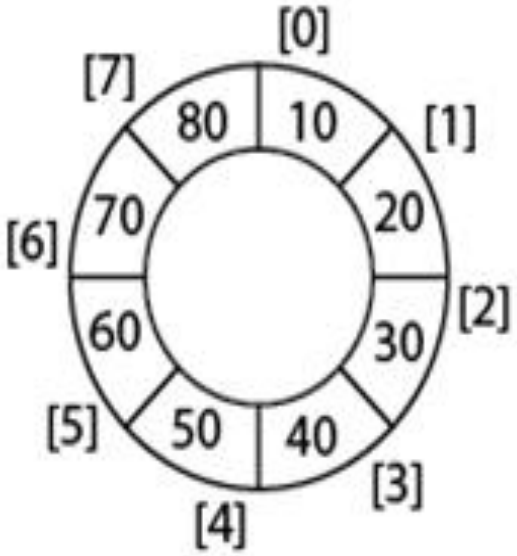

enqueue(20)
front: 0
rear: 1



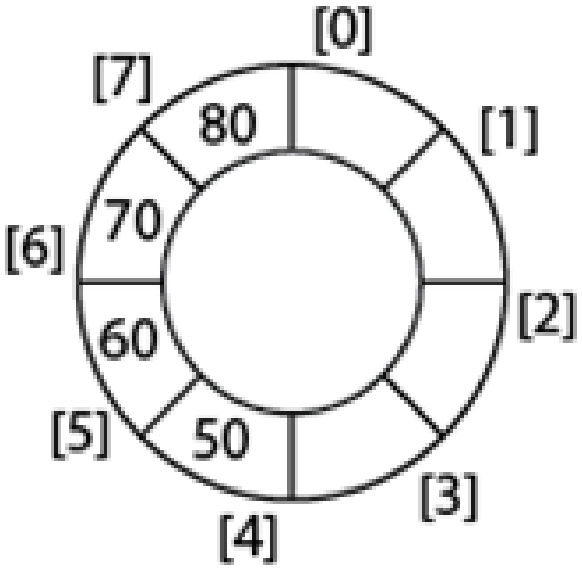

enqueue(10)
front: 0
rear: 0



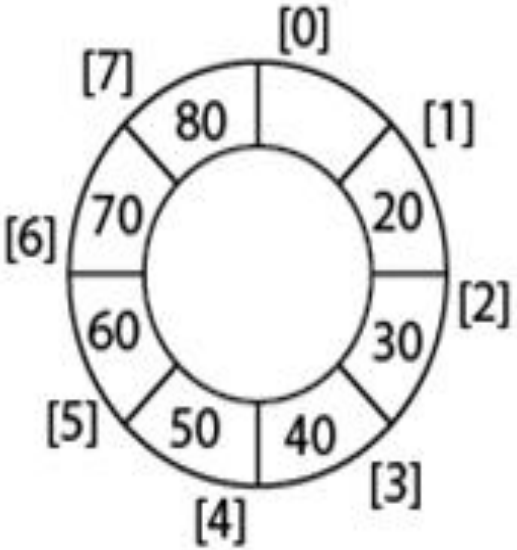

enqueue(30)
enqueue(40)
enqueue(50)
enqueue(60)
enqueue(70)
enqueue(80)
front: 0
rear: 7



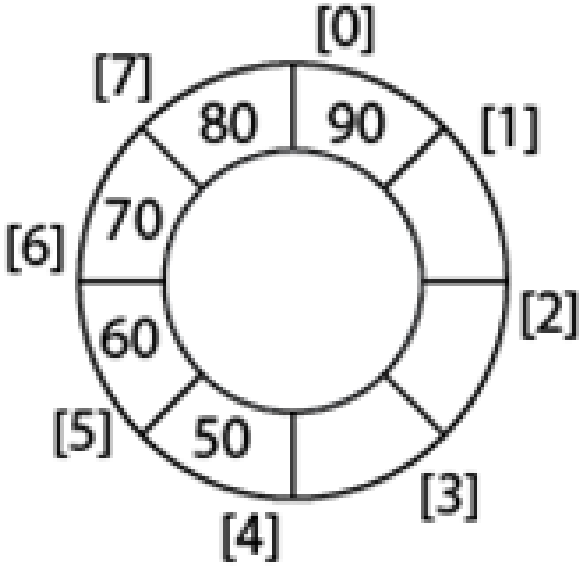

enqueue(90)
queue is full
front: 0
rear: 7



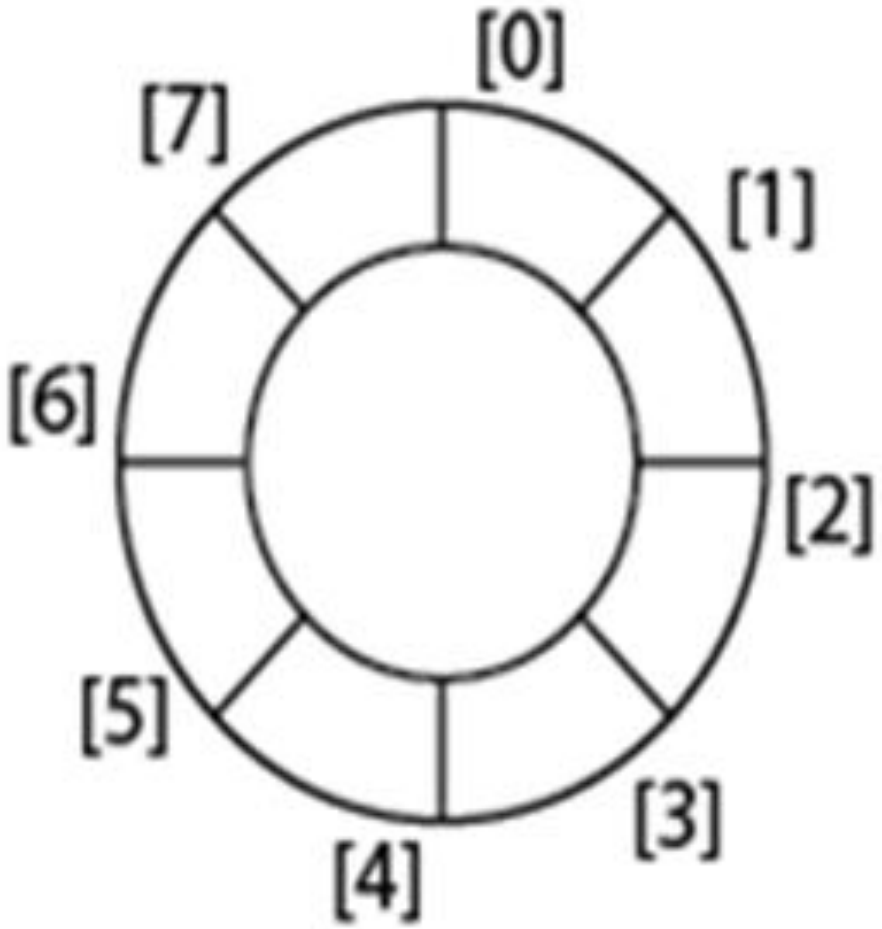

dequeue( )
dequeue( )
dequeue( )
front: 4
rear: 7



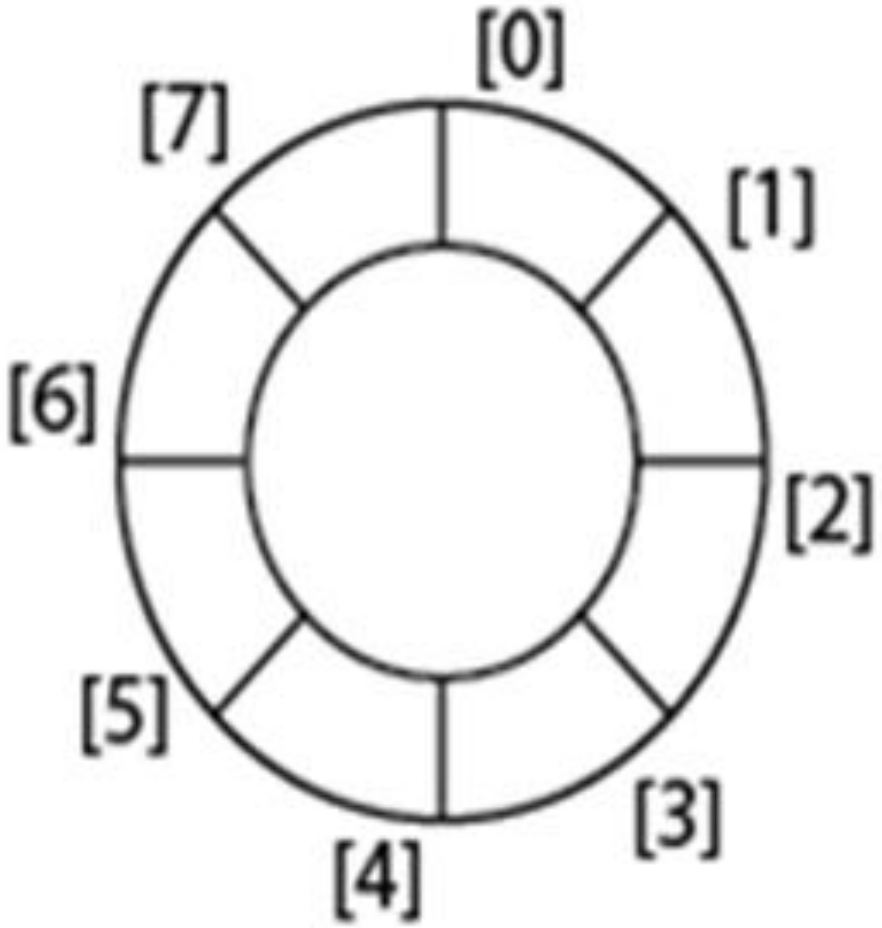

dequeue( )
front: 1
rear: 7



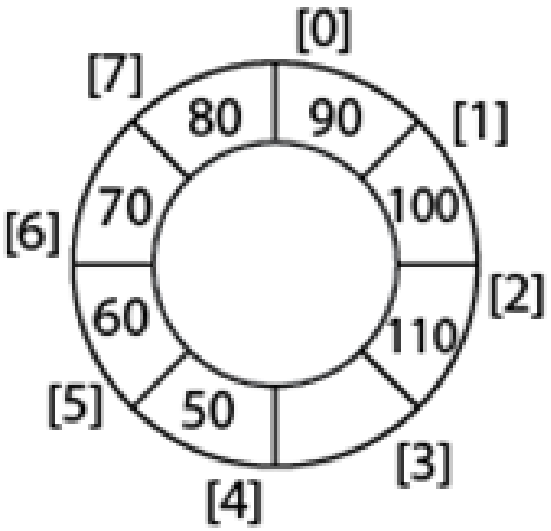

enqueue(90)
front: 4
rear: 0



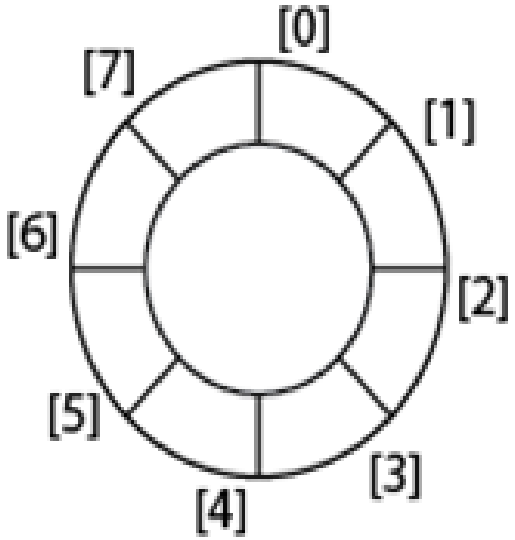

enqueue(100)
enqueue(110)



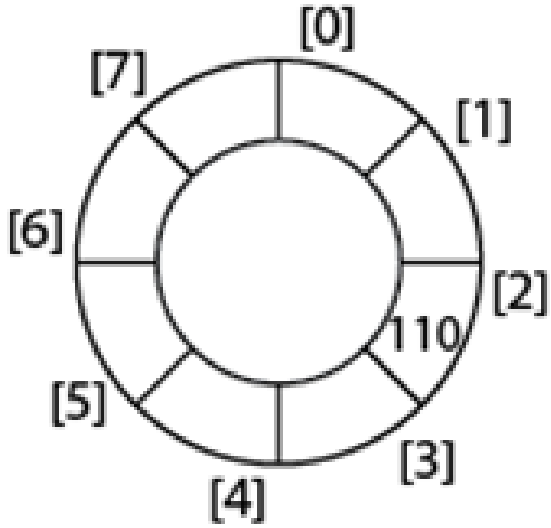

dequeue( )
dequeue( )
dequeue( )
dequeue( )
dequeue( )
dequeue( )



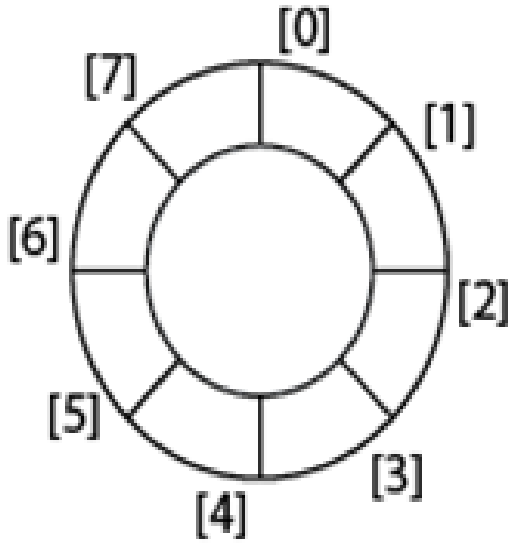

enqueue(100)
enqueue(110)
front: 4
rear: 2




dequeue( )
front: -1
rear: -1




dequeue( )
dequeue( )
dequeue( )
dequeue( )
dequeue( )
dequeue( )
front: 2
rear: 2




dequeue( )
queue is empty
front: -1
rear: -1

# ALGORITHM FOR QUEUE OPERATIONS

Let Q be the array of some specified size say SIZE

**INSERTING AN ELEMENT INTO THE QUEUE**

1. Initialize front=0 rear = –1
2. Input the value to be inserted and assign to variable “data”
3. If (rear >= SIZE)

   (*a) Display “Queue overflow”*

   (*b) Exit*
4. Else

(*a) Rear = rear +1*

5. Q[rear] = data
6. Exit

# ALGORITHM FOR QUEUE OPERATIONS

## DELETING AN ELEMENT FROM QUEUE

1. If (rear< front)
   (*a) Front = 0, rear = –1*
   (*b) Display "The queue is empty"*
   (*c) Exit*
2. Else
   (*a) Data = Q[front]*
3. Front = front +1
4. Exit

# OTHER QUEUES

There are three major variations in a simple queue. They are

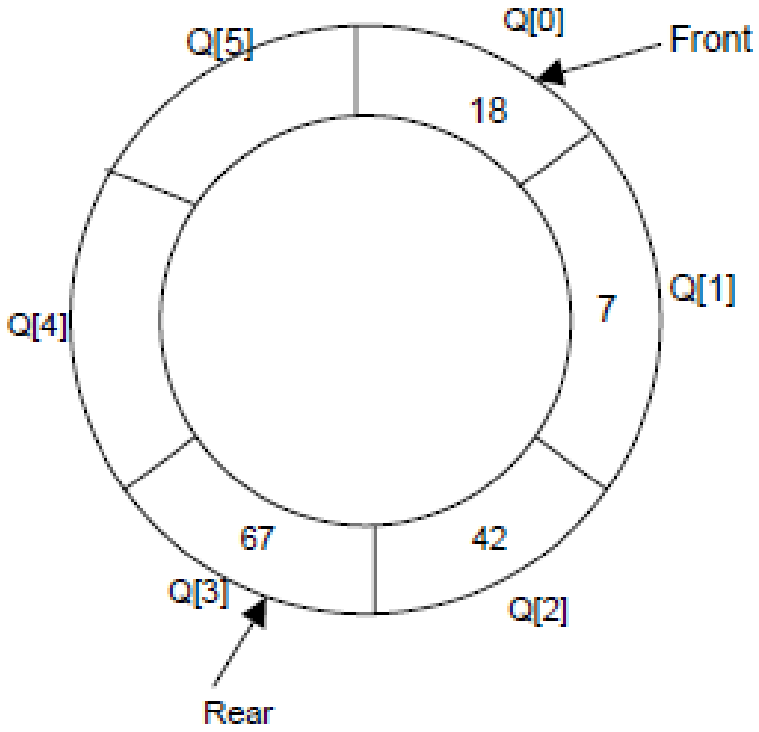
1. Circular queue
2. Double ended queue (de-queue)
3. Priority queue

# CIRCULAR QUEUES

-In circular queues the elements Q[0],Q[1],Q[2] .... Q[*n – 1] is represented in a circular* fashion with Q[1] following Q[n].

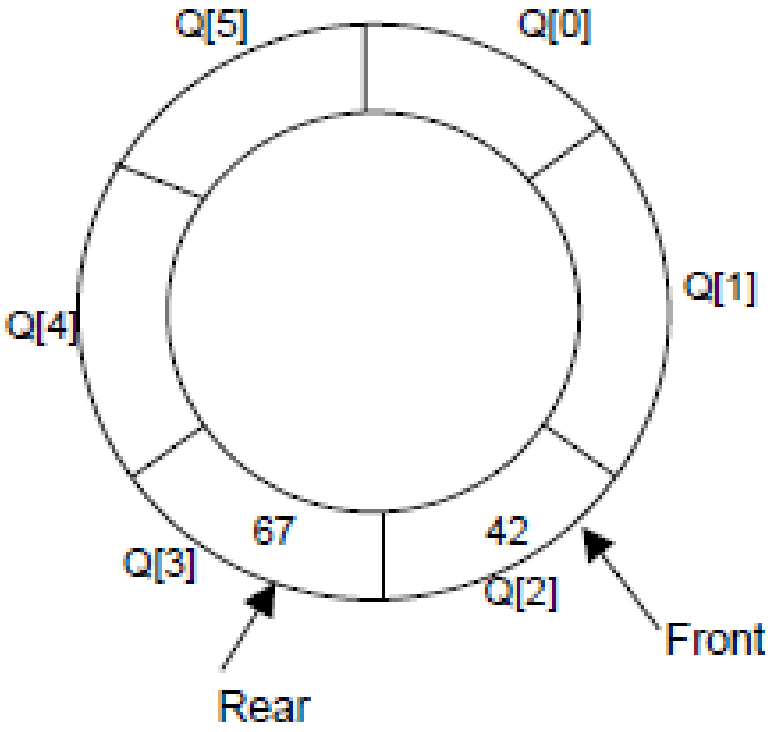
- one in which the insertion of a new element is done at the very first location of the queue if the last location at the queue is full

# CIRCULAR QUEUES



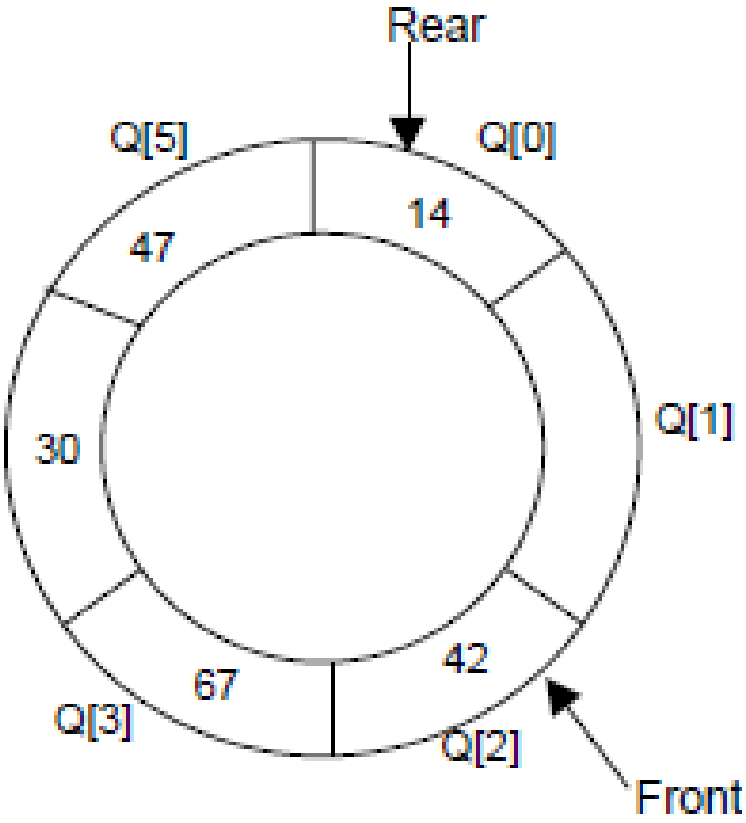

**After inserting 18, 7, 42, 67**

# CIRCULAR QUEUES




**After popping 18, 7**

# CIRCULAR QUEUES




**After pushing 30,47,14**

# ALGORITHM OF CIRCULAR QUEUES

Let Q be the array of some specified size say SIZE. FRONT and REAR are two pointers where the elements are deleted and inserted at two ends of the circular queue. DATA is the element to be inserted.

# ALGORITHM OF CIRCULAR QUEUES

**Inserting an element to circular Queue**

1. Initialize FRONT = – 1; REAR = 1
2. REAR = (REAR + 1) % SIZE
3. If (FRONT is equal to REAR)

(a) Display “Queue is full” (b) Exit

4. Else

(a) Input the value to be inserted and assign to variable “DATA”

5. If (FRONT is equal to – 1)

(a) FRONT = 0 (b) REAR = 0

6. Q[REAR] = DATA
7. Repeat steps 2 to 5 if we want to insert more elements
8. Exit

# ALGORITHM OF CIRCULAR QUEUES

**Deleting an element from a circular queue**

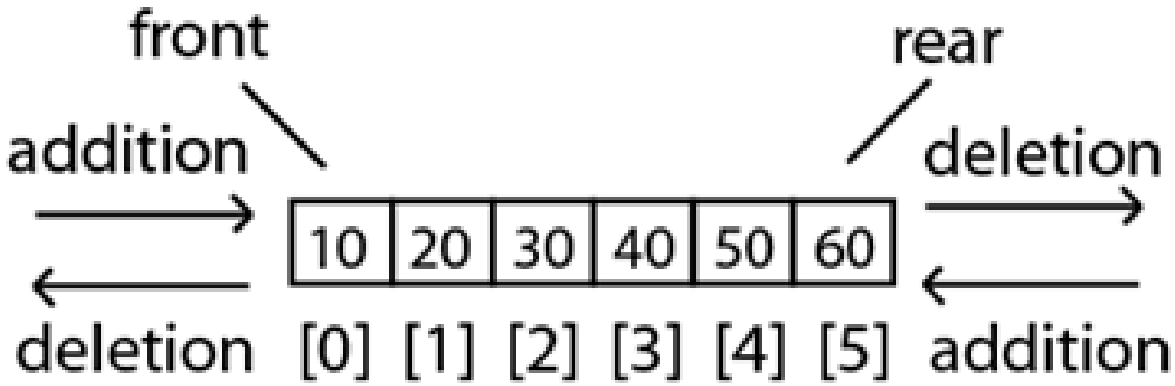
1. If (FRONT is equal to – 1)
(*a) Display "Queue is empty"*
(*b) Exit*
2. Else
(*a) DATA = Q[FRONT]*
3. If (REAR is equal to FRONT)
(*a) FRONT = –1*
(*b) REAR = –1*
4. Else
(*a) FRONT = (FRONT +1) % SIZE*
5. Repeat the steps 1, 2 and 3 if we want to delete more elements
6. Exit

# Double Ended Queue (Deque)

- A deque is a linear list in which all insertions and deletions are made at the end of the list.
- A deque is pronounced as ‘deck’ or ‘de queue’.
- It is a sort of FLIFLO (First In Last In or First Out Last Out).
- Refers to the right end or left end of a queue.

# Deque (cont.)

- A deque is commonly implemented as a circular array with two variables left and right taking care of the active endsof the deque.



# Two Variants of Deque

- Input restricted deque
  - Insertions are allowed at one end only while deletions are allowed at both ends.
- Output restricted deque
  - Allows insertions at both ends of the deque but permits deletions only at one end.

# ALGORITHMS FOR INSERTING AN ELEMENT

**INSERT AN ELEMENT AT THE RIGHT SIDE OF THE DE-QUEUE**

1. Input the DATA to be inserted
2. If ((left == 0 && right == MAX–1) || (left == right + 1))

(*a) Display "Queue Overflow"* (*b) Exit*

3. If (left == –1)

(*a) left = 0* (*b) right = 0*

4. Else

(*a) if (right == MAX –1)*

(*i) left = 0*

(*b) else*

(*i) right = right+1*

5. Q[right] = DATA
6. Exit

# ALGORITHMS FOR INSERTING AN ELEMENT

**INSERT AN ELEMENT AT THE LEFT SIDE OF THE DE-QUEUE**

1. Input the DATA to be inserted
2. If ((left == 0 && right == MAX–1) || (left == right+1))

(*a) Display "Queue Overflow"* (*b) Exit*

3. If (left == – 1)

(*a) Left = 0* (*b) Right = 0*

4. Else

(*a) if (left == 0)*

(*i) left = MAX – 1*

(*b) else*

(*i) left = left – 1*

5. Q[left] = DATA
6. Exit

# ALGORITHMS FOR DELETING AN ELEMENT

**DELETE AN ELEMENT FROM THE RIGHT SIDE OF THE DE-QUEUE**

1. If (left == – 1)

(*a) Display "Queue Underflow"* (*b) Exit*

2. DATA = Q [right]

3. If (left == right)

(*a) left = – 1* (*b) right = – 1*

4. Else

(*a) if(right == 0)*

(*i) right = MAX-1*

(*b) else*

(*i) right = right-1*

5. Exit

# ALGORITHMS FOR DELETING AN ELEMENT

**DELETE AN ELEMENT FROM THE LEFT SIDE OF THE DE-QUEUE**

1. If (left == – 1)

(*a) Display "Queue Underflow"* (*b) Exit*

2. DATA = Q [left]

3. If(left == right)

(*a) left = – 1* (*b) right = – 1*

4. Else

(*a) if (left == MAX-1)*

(*i) left = 0*

(*b) Else*

(*i) left = left +1*

5. Exit

# Applications of a Linear Queue

- Time-sharing system
- A CPU endowed with memory resources

# Applications of aQueue

1. Round robin techniques for processor scheduling is implemented using queue.
2. Printer server routines (in drivers) are designed using queues.
3. All types of customer service software (like Railway/Air ticket reservation) are

designed using queue to give proper service to the customers.

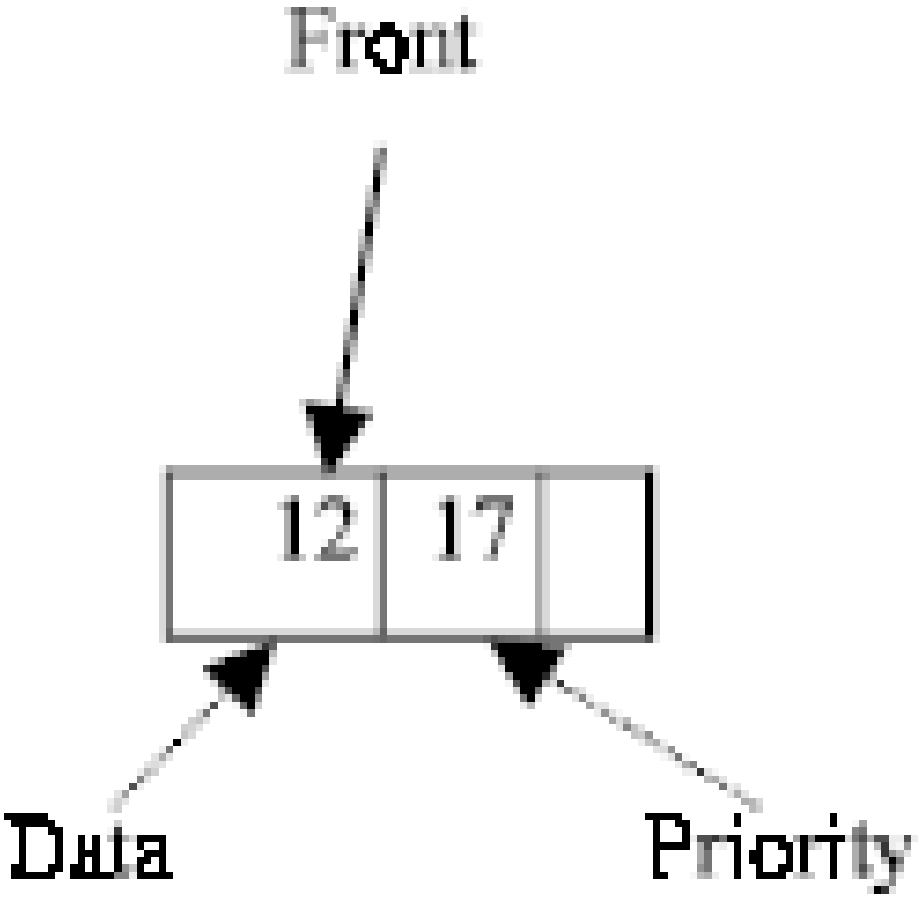
# Priority Queues

# PRIORITY QUEUES

**Priority Queue** is a queue where each element is assigned a priority. In priority queue, the elements are deleted and processed by following rules.

1. An element of higher priority is processed before any element of lower priority.
2. Two elements with the same priority are processed according to the order in which they were inserted to the queue.

# LINKED LIST REPRESENTATION OF PRIORITY QUEUE



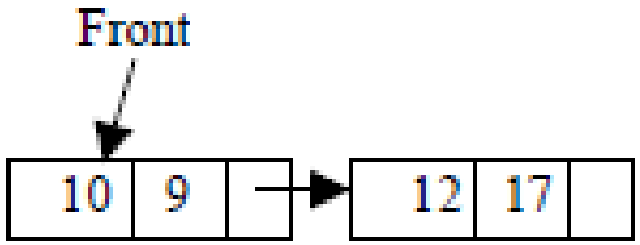



**Fig. 5.37.** push(DATA =10, PRIORITY = 9)

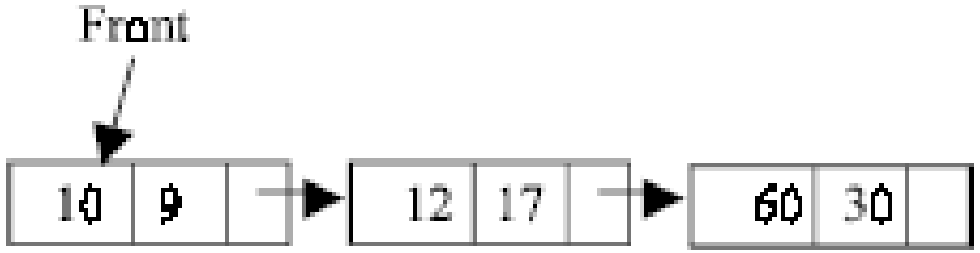



**Fig. 5.38.** push(DATA = 60, PRIORITY = 30)


**Fig. 5.39.** push(DATA = 13, PRIORITY = 46)

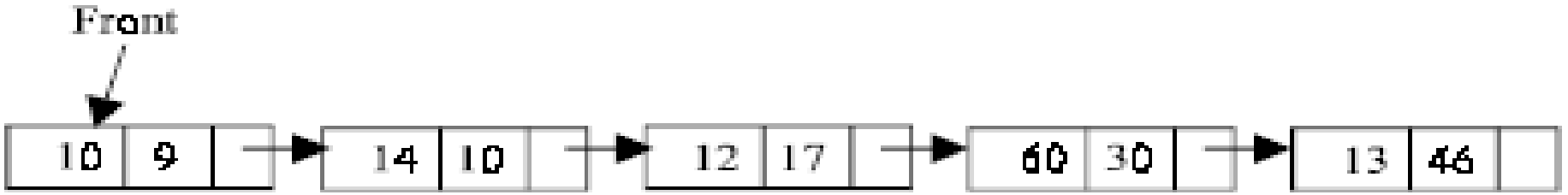



**Fig. 5.40.** push(DATA = 14, PRIORITY = 10)

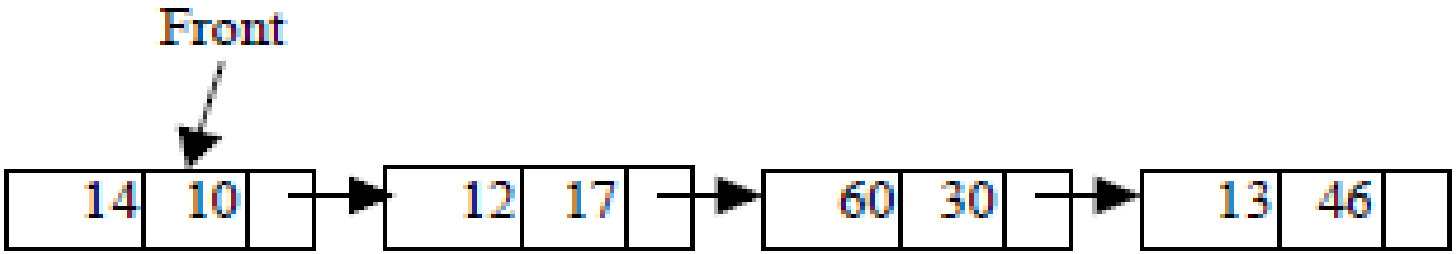



**Fig. 5.41.** $x$ = pop() (*i.e.*, 10)

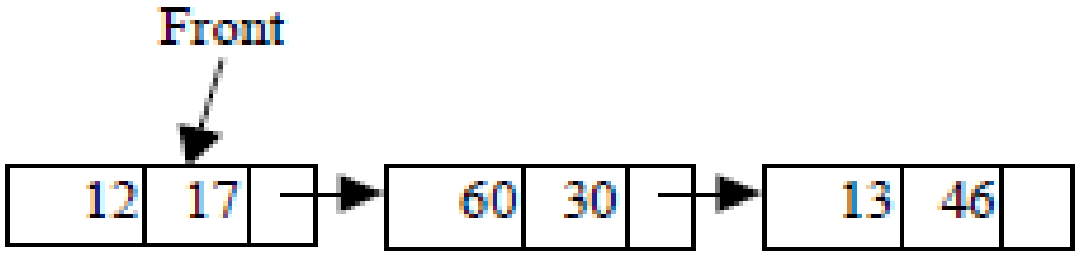



**Fig. 5.42.** $x$ = pop() (*i.e.*, 14)

# Introduction

- Stack and Queue are data structures whose elements are ordered based on a sequence in which they have been inserted
- E.g. pop() function removes the item pushed last in the stack
- Intrinsic order among the elements themselves (e.g. numeric or alphabetic order etc.) is ignored in a stack or a queue

# Definition

- A priority queue is a data structure in which prioritized insertion and deletion operations on elements can be performed according to their priority values.

- There are two types of priority queues:
  - Ascending Priority queue, and a
  - Descending Priority queue

Applications commonly require comparing and ranking objects according to parameters or properties, called "**keys**," that are assigned to each object in a collection.

***key*** *to be an object that is assigned to an element as a* specific attribute for that element and that can be used to identify, rank, or weigh that element.

**Example: (comparing)**
-companies earning
-restaurant
-priority of standby passengers-

**The fundamental functions of a priority queue *P* are as follows:**

**insert(*e*):** *Insert the element e (with an implicit associated key value)* into *P.*

**min():** Return an element of *P with the smallest associated key* value, that is, an element whose key is less than or equal to that of every other element in *P.*

**removeMin():** Remove from *P the element min().*

| *Operation* | *Output* | *Priority Queue* |
|---|---|---|
| insert(5) | – | {5} |
| insert(9) | – | {5,9} |
| insert(2) | – | {2,5,9} |
| insert(7) | – | {2,5,7,9} |
| min() | [2] | {2,5,7,9} |
| removeMin() | – | {5,7,9} |
| size() | 3 | {5,7,9} |
| min() | [5] | {5,7,9} |
| removeMin() | – | {7,9} |
| removeMin() | – | {9} |
| removeMin() | – | {} |
| empty() | *true* | {} |
| removeMin() | *"error"* | {} |

# Types of Priority Queue

- **Ascending Priority queue**: a collection of items into which items can be inserted ***randomly*** but only the ***smallest*** item can be removed
- If "**A-Priority-Q**" is an ascending priority queue then
  - Enqueue() will insert item 'x' into **A-Priority-Q**,
  - minDequeue() will remove the minimum item from **A-Priority-Q** and return its value

# Types of Priority Queue

- **Descending Priority queue**: a collection of items into which items can be inserted ***randomly*** but only the ***largest*** item can be removed
- If "**D-Priority-Q**" is a descending priority queue then
  - Enqueue() will insert item x into **D-Priority-Q**,
  - maxDequeue( ) will remove the maximum item from **D-Priority-Q** and return its value

# Generally

- In both the above types, if elements with equal priority are present, the FIFO technique is applied.
- Both types of priority queues are similar in a way that both of them remove and return the element with the highest **"Priority"** when the function remove() is called.
  - For an ascending priority queue item with smallest value has maximum "priority"
  - For a descending priority queue item with highest value has maximum "priority"
- This implies that we must have criteria for a priority queue to determine the Priority of its constituent elements.
- the elements of a priority queue can be numbers, characters or any complex structures such as phone book entries, events in a simulation

# Priority Queue Issues

- In what manner should the items be inserted in a priority queue
  - Ordered (so that retrieval is simple, but insertion will become complex)
  - Arbitrary (insertion is simple but retrieval will require elaborate search mechanism)
- Retrieval
  - In case of un-ordered priority queue, what if minimum number is to be removed from an ascending queue of n elements (n number of comparisons)
- In what manner should the queue be maintained when an item is removed from it
  - Emptied location is kept blank (how to recognize a blank location ??)
  - Remaining items are shifted