Linear Data Structure: Queue

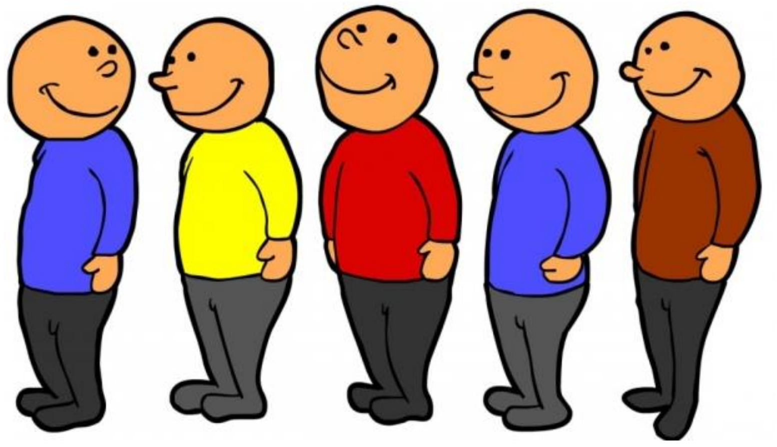
Dr. Jyoti Srivastava

Definition of Queue

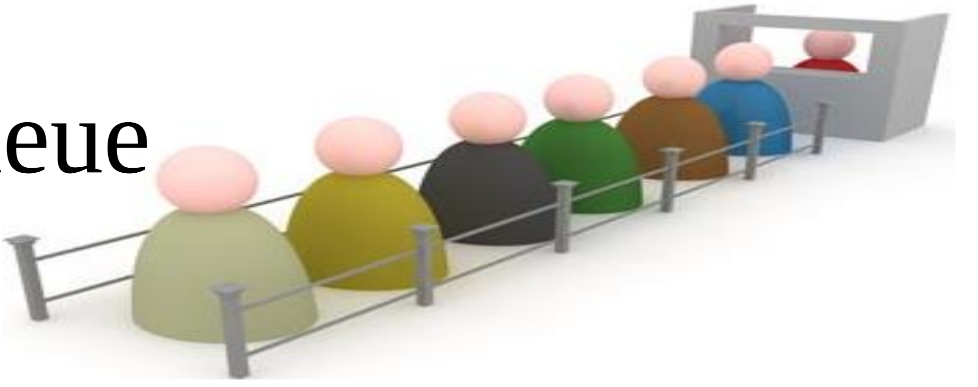
- A **Queue** is an ordered collection of items

- from which items may be deleted at one end (called the ***front*** (also called **head**) of the queue)

- and into which items may be inserted at the other end (the ***rear*** (also called **tail**) of the queue).



Introduction to Queue



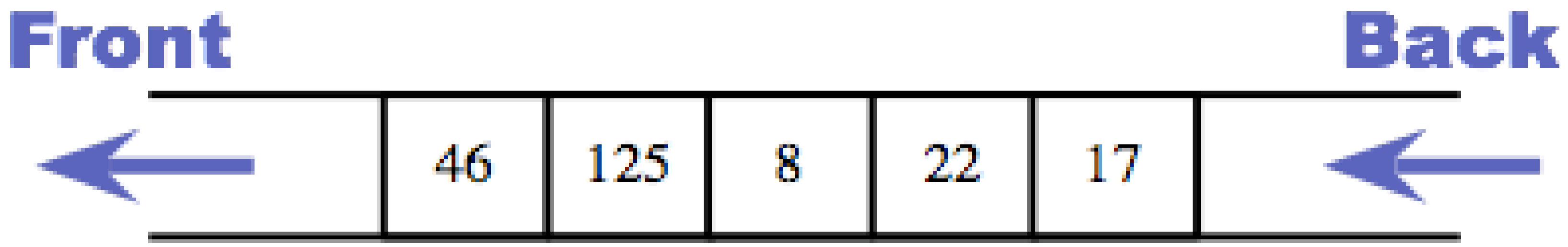
- The first element inserted into the queue is the first element to be removed.

- For this reason a queue is sometimes called a ***fifo*** (first-in first-out) or ***fcfs***(first-come-first-servre) list as opposed to the stack, which is a ***lifo*** (last-in first-out).

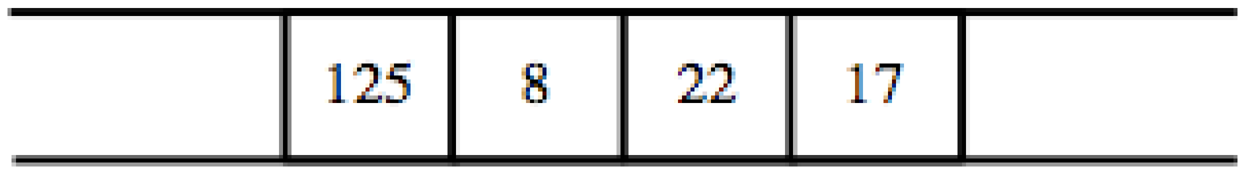
- Process to add an element into queue is called **Enqueue.**

- Process of removal of an element from queue is called **Dequeue**.

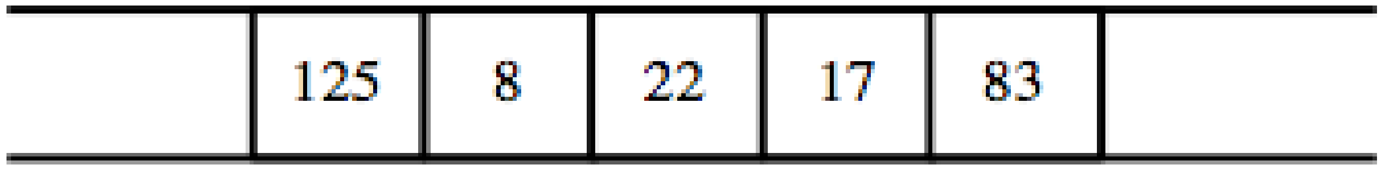
Array representation of Queue



Items enter queue at back and leave from front



After dequeue()



After enqueue(83)

Queue Operations

- Insert to the rear of the queue
- Remove (Delete) from the front of the queue
- Is the Queue Empty
- Is the Queue Full
- What is the size of the Queue

Insert in Queue

```
Step 1: IF REAR =  MAX – 1 then;
            Write OVERFLOW;
            RETURN to Caller function
      [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
```

Insert in Queue

Before insertion

2	3	9		
Front 0	1	Rear 2	3	4

Front = 0 and Rear = 2

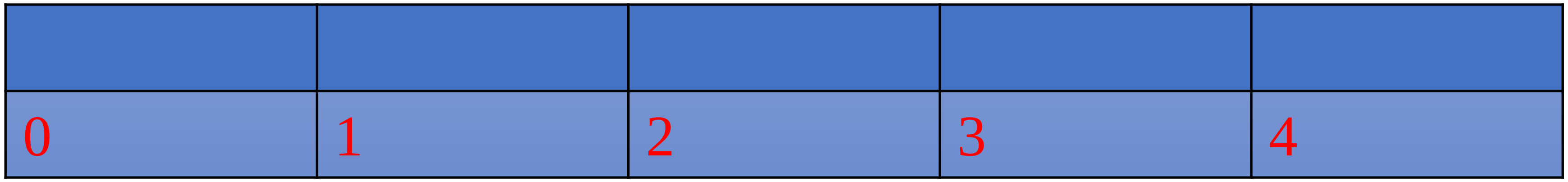
Insert 1 in Queue

2	3	9	1	
Front 0	1	2	Rear 3	4

Front = 0 and Rear = 3

Delete in Queue

Size = 5



Front = -1 and Rear = -1

Front ==Rear ==-1// queue is empty

Front ==Rear // Only one value in the queue

Delete in Queue

```
Step 1: IF FRONT = -1, then;
            Write UNDERFLOW
        ELSE
            SET VAL = QUEUE[FRONT]
            IF FRONT = REAR
                SET FRONT = REAR = -1
            ELSE
                SET FRONT = FRONT + 1
            RETURN VAL
        [END OF IF]
    [END OF IF]
Step 2: Exit
```

Delete in Queue

Before deletion

Front = 0 and Rear = 2

2	3	9		
Front		Rear		

After deletion

Front = 1 and Rear = 2

	3	9		
	Front	Rear		

Queue Implementation using Array versus Linked List

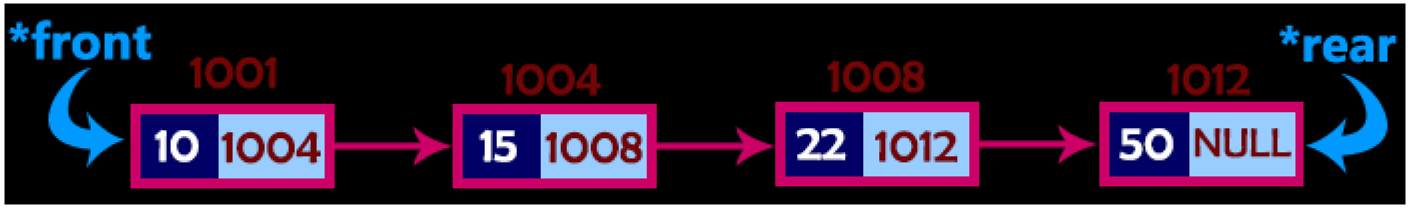
- **Queue Implementation using Array:**
  - Major problem: it works only for fixed number of data values.
  - So the amount of data must be specified at the beginning of the implementation itself.
  - It is not suitable, when we don't know the size of data which we are going to use.

- **Queue Implementation using Linked List:**
  - The queue implemented using linked list can work for unlimited number of values.
  - So, there is no need to fix the size at the beginning of the implementation.

Linked Implementation of Queue

- last inserted node is always pointed by '**rear**’
- and the first node is always pointed by '**front**’.

- Example:



- last inserted node is 50 and it is pointed by '**rear**’
- first inserted node is 10 and it is pointed by '**front**'

Linked Implementation of Queue

```
struct Node {
        int data;
        struct Node *next;
}*front = NULL, *rear = NULL;
```

enQueue(value) - Inserting an element into the Queue

Step 1: Create a **newNode** with given value and set '**newNode → next**' to **NULL**.

Step 2: Check whether queue is **Empty** (**rear == NULL**)

Step 3: If it is **Empty** then, set **front = newNode** and **rear = newNode**.

Step 4: If it is **Not Empty** then, set **rear → next = newNode** and **rear = newNode**.

deQueue() - Deleting an Element from Queue

```
Step 1: Check whether queue
            is Empty (front == NULL).

Step 2: If it is Empty, then
            display "Queue is Empty!!!
            Exit

Step 3: If it is Not Empty then,
            define a Node pointer 'temp' and set it to 'front'.
Step 4: If  set 'front == rear' then,
            set front = NULL, and rear = NULL
Step 5: else set 'front = front → next'
             delete 'temp' (free(temp)).
```

display() - Displaying the elements of Queue

Step 1: Check whether queue
is **Empty** (**front == NULL**).

Step 2: If it is **Empty** then,
display **'Queue is Empty!!!'**
Exit.

Step 3: If it is **Not Empty** then,
define a Node pointer **'temp'** and initialize with **front**.
(temp = front)

Step 4: While(**temp != NULL**)
Display '**temp → data**'
temp = temp → next.

Drawback of Queue

- After deletion of any two number from Queue Front = 2 and Rear = 4

		9	7	8
0	1	Front = 2	3	Rear = 4

- If we want to add any element to Queue, it will prompt ‘OVERFLOW’

- Major drawback: Wastage of Space

Circular Queue

- Since all the items in the queue are required to shift when an item is deleted, this method is not preferred.

		9	7	8
0	1	Front = 2	3	Rear = 4

- The other method is ***circular queue.***

- When rear = MAXQUEUE-1, the next element is entered at items[0] in case that spot is free.

Circular Queue

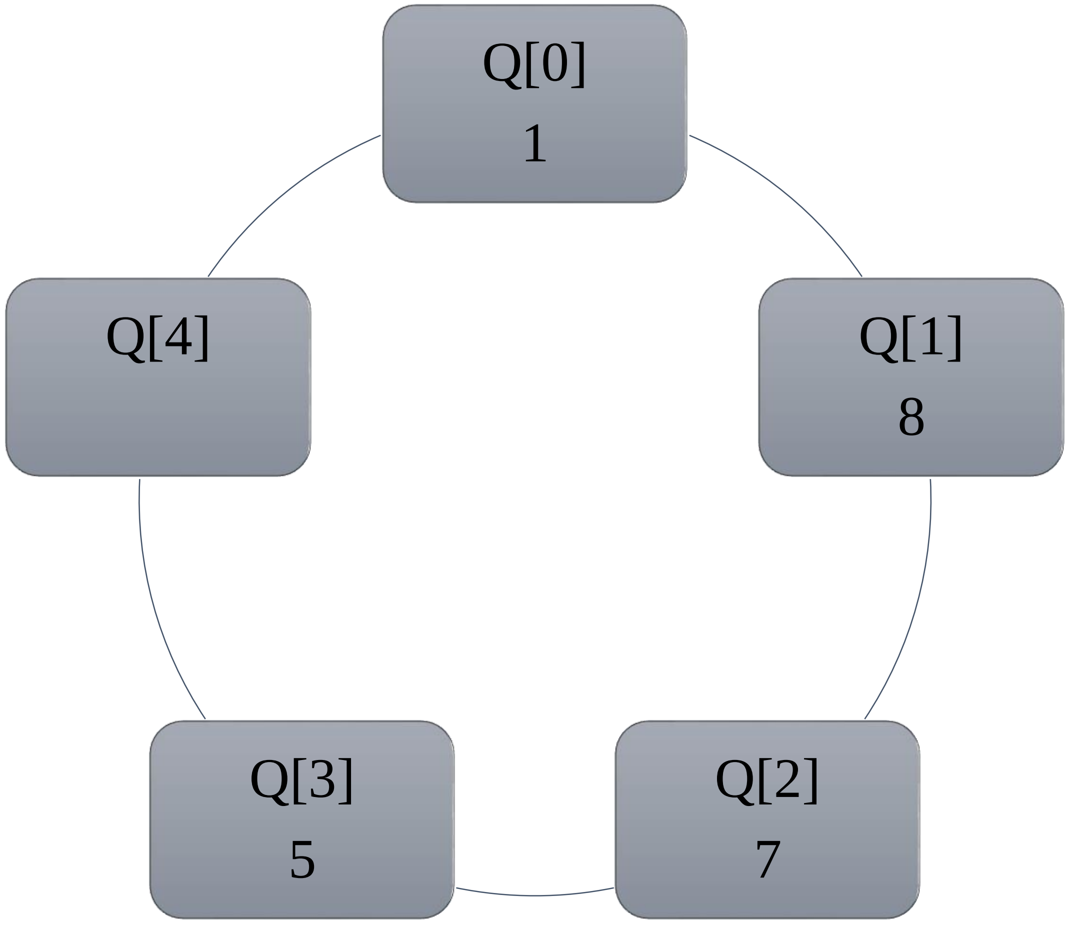
- If FRONT = 0 and REAR = MAX-1, then circular queue is full.

1	3	9	7	8
Front = 0	1	2	3	Rear = 4

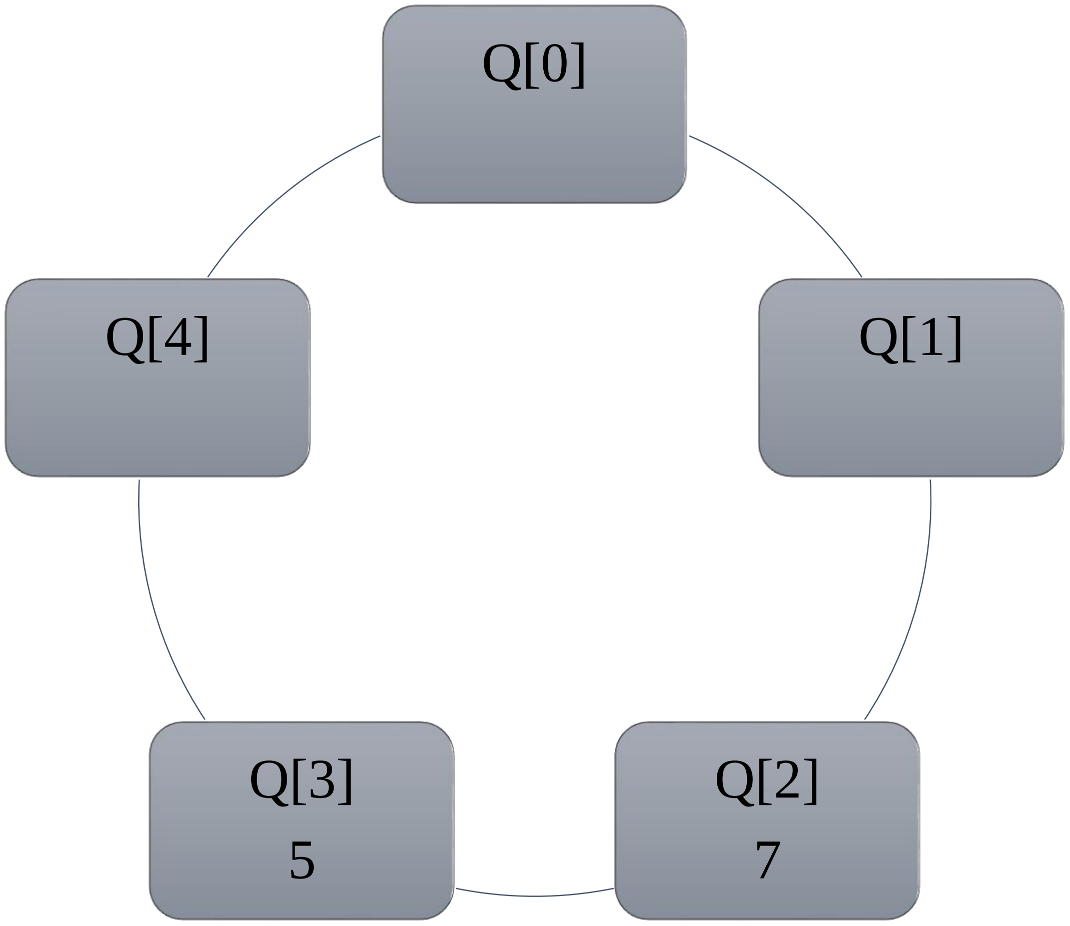
- If FRONT = REAR = -1, then circular queue is empty.

0	1	2	3	4

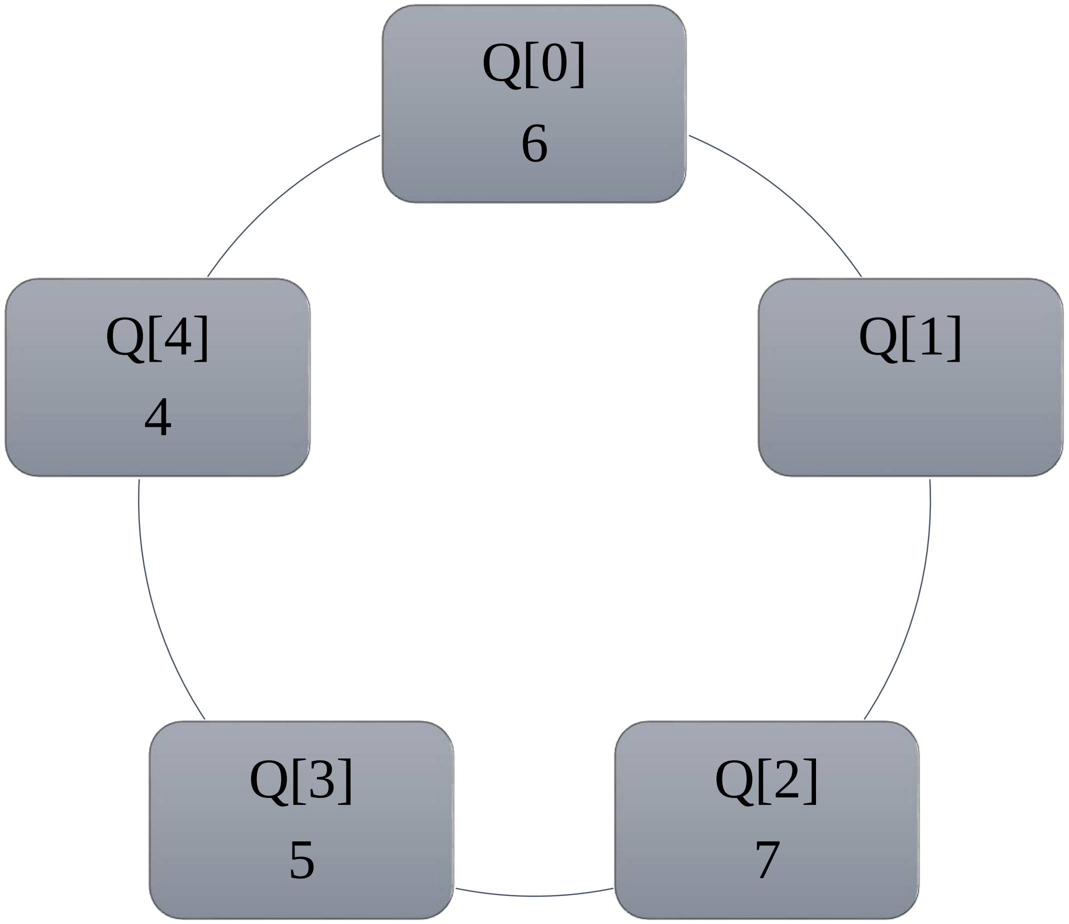
Circular Queue with values 1, 8, 7, 5



Circular Queue deleting values 1, 8



Circular Queue
inserting values 4,6



Insert in Circular Queue

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

Delete in Circular Queue

```
Step 1: IF FRONT = -1, then;
            Write UNDERFLOW
            RETURN to Caller function
        [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
```

Example

Circular Queue, capacity of 6 elements

REAR = 3, FRONT = 1

Queue: **_, L, M, N, _, _** (where _ denotes empty cell)

Describe the queue as the following operations take place:

1. Add O
2. Add P
3. Delete two letters
4. Add Q, R, S
5. Delete one letter

Example

- Initially

	L	M	N		
0	FRONT = 1	2	REAR=3	4	5

- Add O

	L	M	N	O	
0	FRONT = 1	2	3	REAR=4	5

- Add P

	L	M	N	O	P
0	FRONT = 1	2	3	4	REAR=5

Example

- Delete two letters

			N	O	P
0	1	2	FRONT = 3	4	REAR=5

- Add Q, R, S

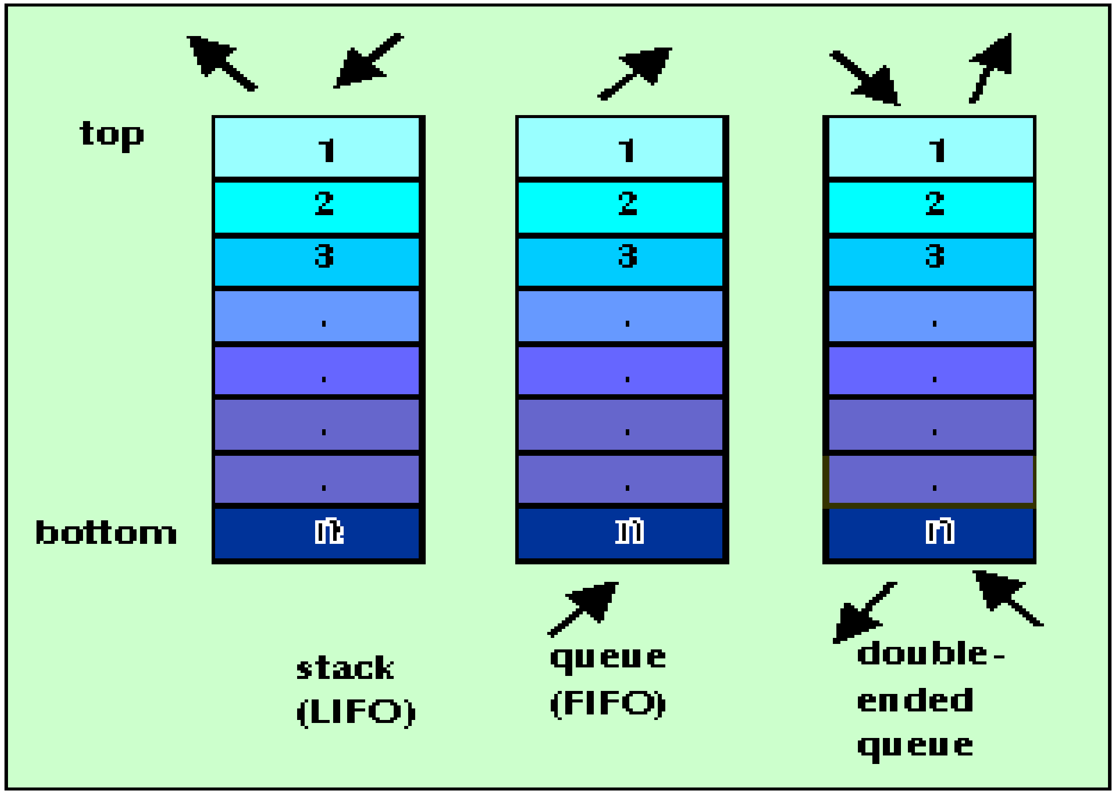
Q	R	S	N	O	P
0	1	REAR = 2	FRONT = 3	4	5

- Delete one letter

Q	R	S		O	P
0	1	REAR = 2	3	FRONT = 4	5

DEQUE – double-ended queue

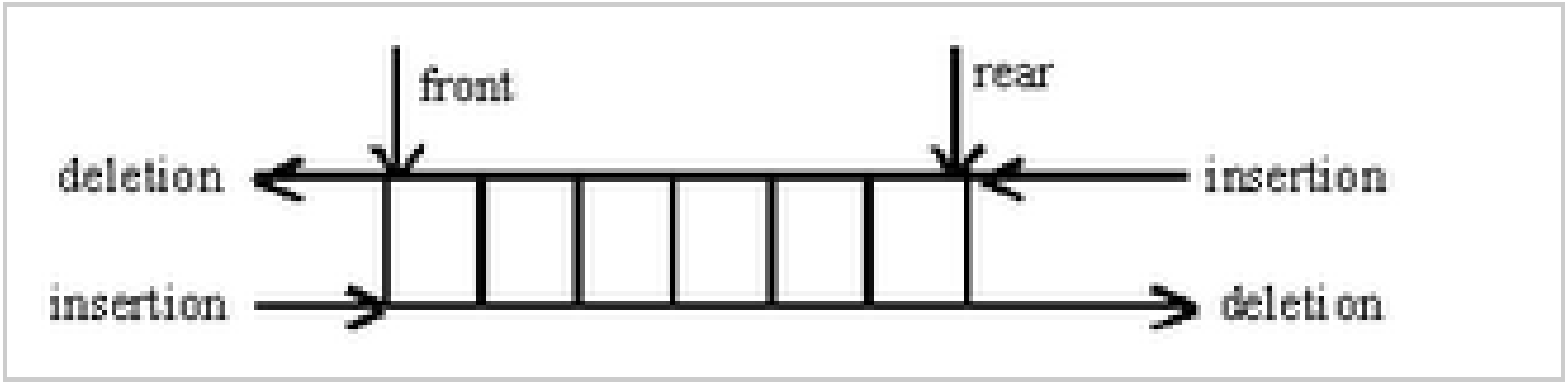
Deque is sometimes written *dequeue*, but this use is generally deprecated in technical literature or technical writing because *dequeue* is also a verb meaning *"to remove from a queue"*.



double-ended queue

DEQUE

- *DeQueue is a data structure in which elements may be added to or deleted from the front or the rear.*
- Dequeue can be behave like a queue and like a stack by using limited functions.



DEQUE and its sub-types

- This differs from the QUEUE (FIFO), where elements can only be added to one end and removed from the other.

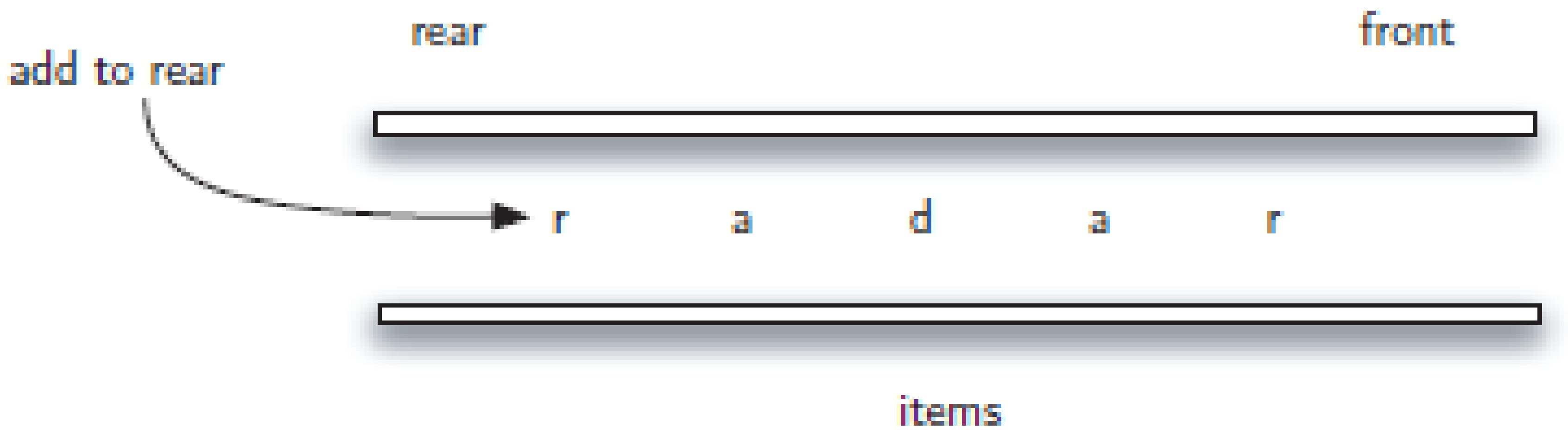
- This general data class has some possible sub-types:

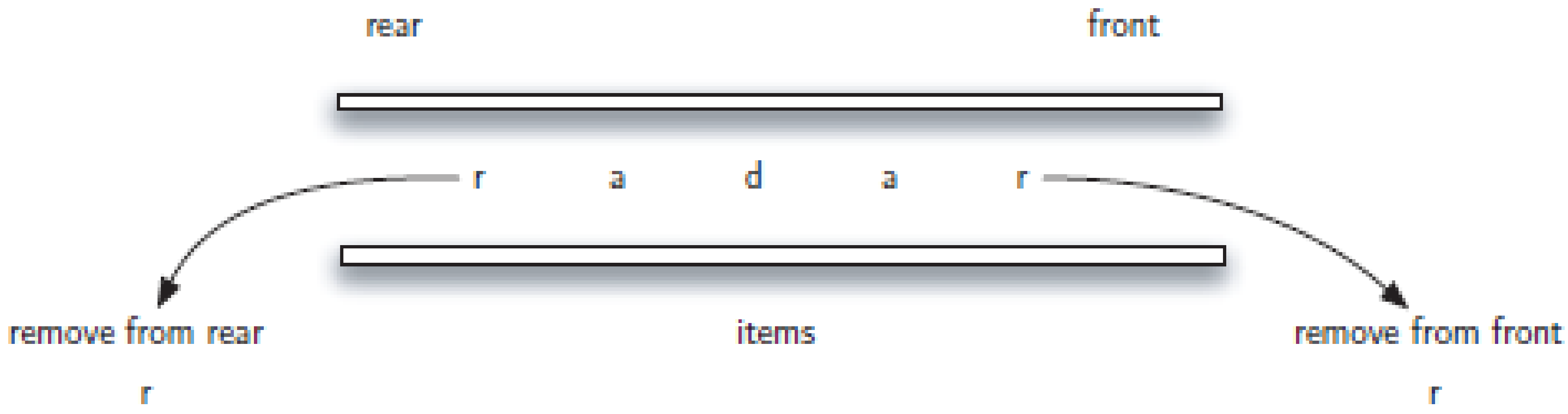
1. *Input-restricted deque* is one where deletion can be made from both ends, but insertion can only be made at one end.

2. *Output-restricted deque* is one where insertion can be made at both ends, but deletion can be made from one end only.

DEQUE – Palindrome Checker

Add "radar" to the rear





Remove from front and rear

Example

DEQUE, circular array with 6 memory cells

LEFT = 2   RIGHT = 4

DEQUE: _, A, C, D, _, _

1. F is added to right of the queue
2. Two letters are deleted from right
3. K, L, M are added to left
4. One letter on left is deleted
5. R is added to right
6. S is added to left

Example

- Initially

	A	C	D		
1	LEFT = 1	3	RIGHT = 4	5	6

- F is added to right of the queue

	A	C	D	F	
1	LEFT = 2	3	4	RIGHT = 5	6

- Two letters are deleted from right

	A	C			
1	LEFT = 2	RIGHT = 3	4	5	6

Example

- K, L, M are added to left

K	A	C		M	L
1	2	RIGHT = 3	4	LEFT = 5	6

- One letter on left is deleted

K	A	C			L
1	2	RIGHT = 3	4	5	LEFT = 6

- R is added to right

K	A	C	R		L
1	2	3	RIGHT = 4	5	LEFT = 6

Example

- S is added to left

K	A	C	R	S	L
1	2	3	RIGHT = 4	LEFT = 5	6

Priority Queue

- A **priority queue** is an abstract data type which is like a regular queue or stack data structure, but where additionally each element has a *"priority"* associated with it.

A	**B**	**C**	**D**	**E**
P1	P2	P3	P4	P5
FRONT				**REAR**

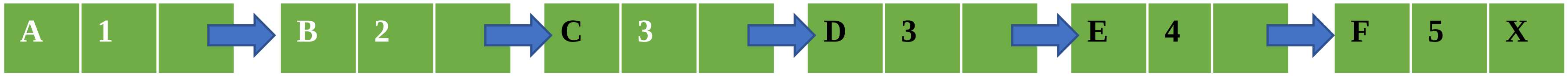
- It is possible that element 'D' may be deleted before an element pointed out by FRONT.

Priority Queue Rules

1. In a priority queue, an element with high priority is served before an element with low priority.

2. If two elements have the same priority, they are served according to their order in the queue.

Linked representation



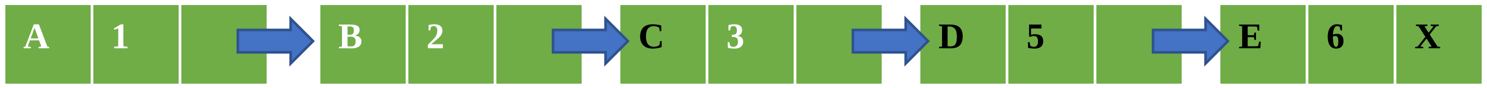
- From this queue, we can not decide whether A was inserted before E or E was inserter before A, as this is not sorted based on *FCFS*.

- Element with higher priority comes before the element of lower priority.

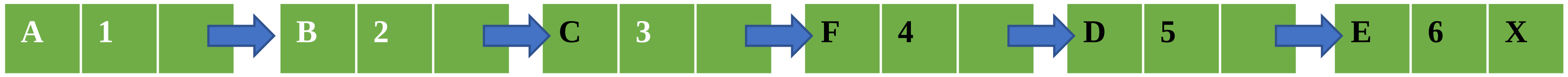
- When two elements with same priority
  - arranged and processed on *FCFS* principle.

Note: **lower number means higher priority**

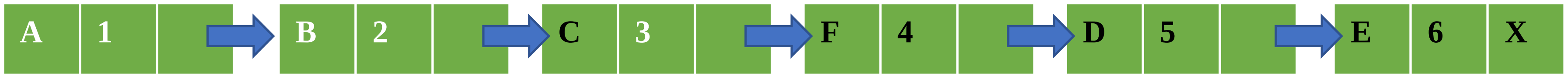
Insertion



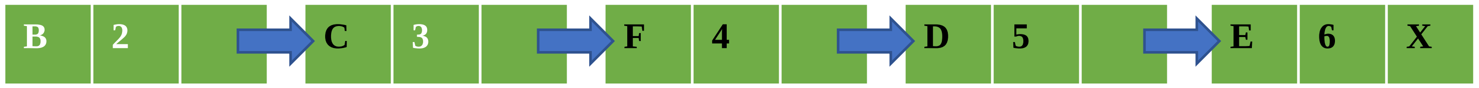
- To insert a value, we have to traverse the entire list until we find a node with lower than that of the new element.
- New element will be inserted before the node with a lower priority.
- If there is node with same priority then new node will be inserted after that element.



Deletion



- Deletion is very simple in this case, first node will be deleted and the data of that node will be processed first.



Array representation of Priority Queue

- When array is used for priority queue, a separate queue for each priority number is maintained.
- We use two-dimensional array for this purpose where each queue will be allocated the same amount of space.
- Row will provide priority while column specifies element, that we want to store.

Values	**Priority**
A	1
B	2
D	3
E	4
F	3

FRONT	**REAR**
1	1
1	1
1	2
1	1

	1	**2**	**3**	**4**	**5**
1	A				
2	B				
3	D	F			
4	E				

Array representation - Insert

Values	**Priority**
A	1
B	2
D	3
E	4
F	3
G	4

FRONT	**REAR**		**1**	**2**	**3**	**4**	**5**
1	1	1	A				
1	1	2	B				
1	2	3	D	F			
1	2	4	E	G			

Array representation - Delete

Values	**Priority**
A	1
B	2
D	3
E	4
F	3
G	4

FRONT	**REAR**		**1**	**2**	**3**	**4**	**5**
1	1	1	A				
1	1	2	B				
1	2	3	D	F			
1	2	4	E	G			

- Removal will always be based on priority,

  A -> B -> D -> F -> E -> G

Application of Queues

- Widely used **as waiting lists for single shared resource like printer, CPU, etc.**

- Queues are used **as buffers on MP3 players** and **portable CD players**.

- In real life, **Call Center phone systems use Queues**, to hold people calling them in an order, until a service representative is free.

- Computer systems must often provide a “holding area” for messages **between two processes, two programs, or even two systems**. This holding area is usually called a “**buffer**” and is often implemented as a queue.