LJ University
University with a Difference

DATA STRUCTURE USING JAVA

LECTURE NOTES FOR UNIT - 5

UNIT -5
QUEUE – II

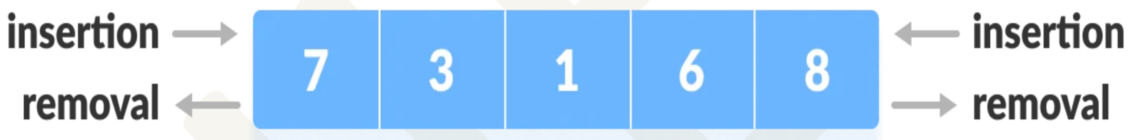
PREPARED BY:
MR. VISMAY SHAH

Contents

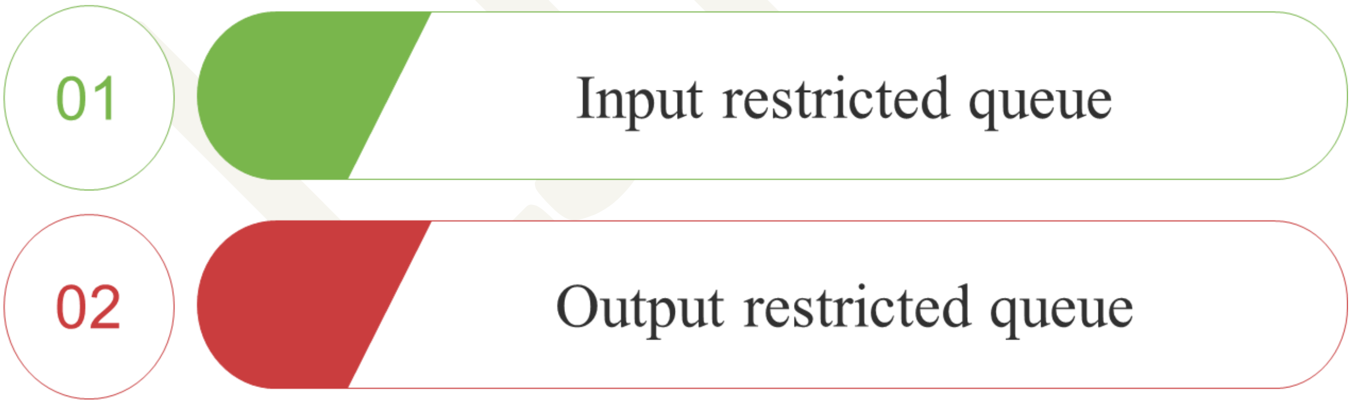
1. DOUBLE ENDED QUEUE

✓ Deque or Double Ended Queue is a type of queue in which insertion and removal of elements can either be performed from the front or the rear. **Thus, it does not follow FIFO rule (First in First Out).**

Operations	Description
insertFront ()	Adds an item at the front of Deque.
insertLast ()	Adds an item at the rear of Deque.
deleteFront ()	Deletes an item from the front of Deque.
deleteLast ()	Deletes an item from the rear of Deque.
getFront ()	Gets the front item from the queue.
getRear ()	Gets the last item from queue
isEmpty ()	Checks whether Deque is empty or not.
isFull ()	Checks whether Deque is full or not.



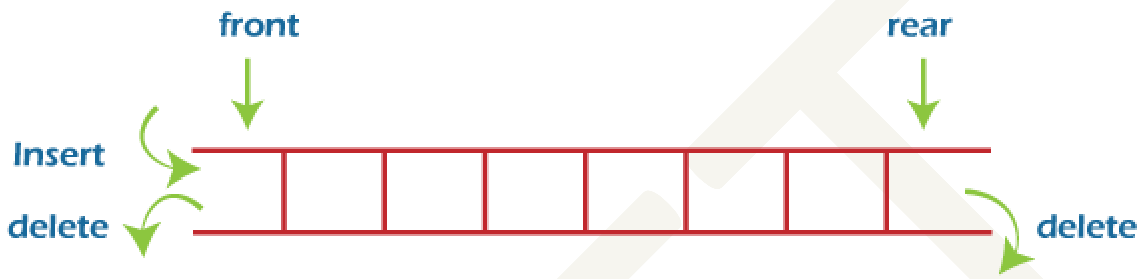
2. Types of Deque



3. Input Restricted Queue

INPUT RESTRICTED QUEUE

✓ In input restricted queue, insertion operation can be performed at only one end, while deletion can be performed from both ends.

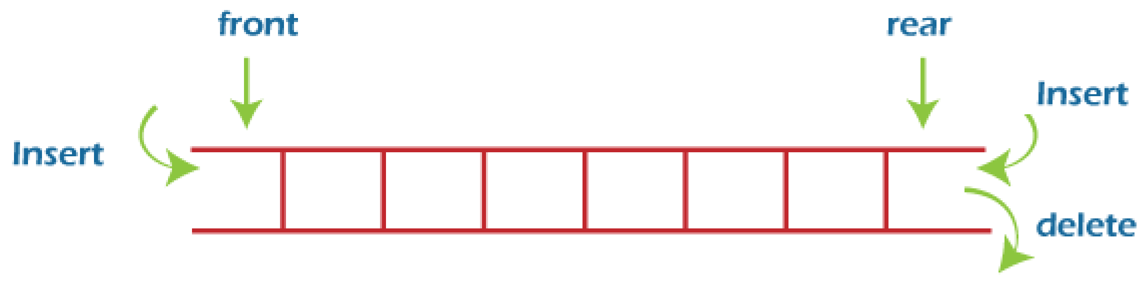


input restricted double ended queue

4. Output Restricted Queue

OUTPUT RESTRICTED QUEUE

✓ In output restricted queue, deletion operation can be performed at only one end, while insertion can be performed from both ends.



Output restricted double ended queue

5. Operations Performed on Deque

Operations Performed on deque

01 Insertion at front

02 Insertion at rear

03 Deletion at front

04 Deletion at rear

✓ We can also perform peek operations in the deque along with the operations listed above. Through peek operation, we can get the deque's front and rear elements of the deque. So, in addition to the above operations, following operations are also supported in dequeue:

- Get the front item from the deque
- Get the rear item from the deque
- Check whether the deque is full or not
- Checks whether the deque is empty or not

Now, let's understand the operation performed on deque using an example.

6. Insertion At The Front End

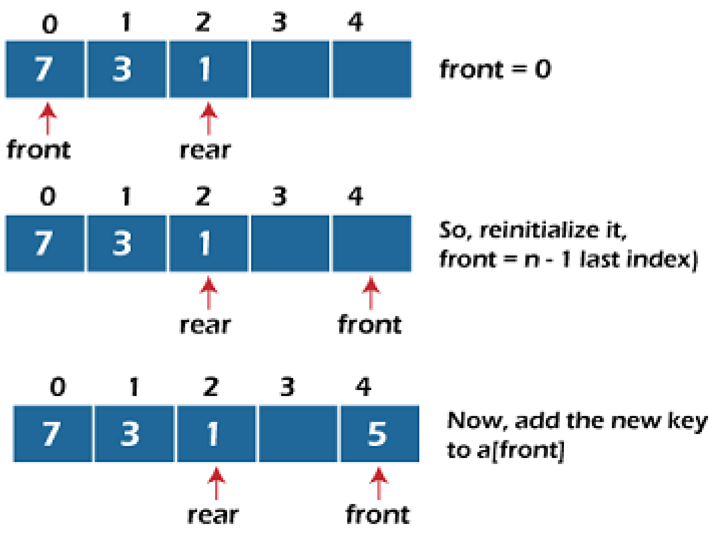
INSERTION AT THE FRONT END

✓ In this operation, the element is inserted from the front end of the queue.

✓ Before implementing the operation, we first have to check whether the queue is full or not.

✓ If the queue is not full, then the element can be inserted from the front end by using the below conditions –

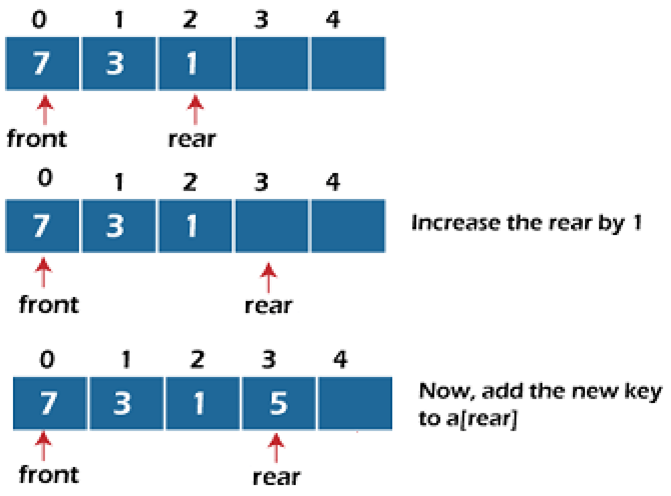
- If the queue is empty, both rear and front are initialized with 0. Now, both will point to the first element.
- Otherwise, check the position of the front if the front is less than 1 (front < 1), then reinitialize it by front = n - 1, i.e., the last index of the array.



7. Insertion At The Rear End

INSERTION AT THE REAR END

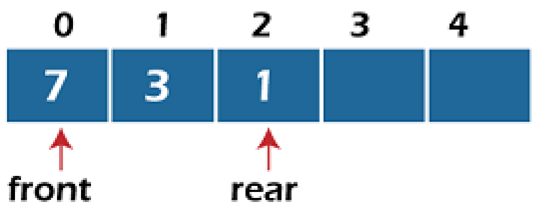
- ✓ In this operation, the element is inserted from the rear end of the queue.
- ✓ Before implementing the operation, we first have to check again whether the queue is full or not.
- ✓ If the queue is not full, then the element can be inserted from the rear end by using the below conditions –
  - If the queue is empty, both rear and front are initialized with 0. Now, both will point to the first element.
  - Otherwise, increment the rear by 1. If the rear is at last index (or size - 1), then instead of increasing it by 1, we have to make it equal to 0.

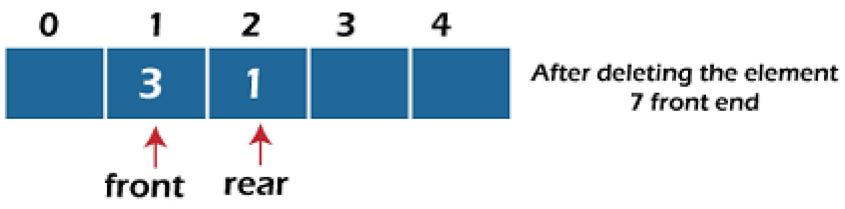


8. Deletion At The Front End

DELETION AT THE FRONT END

- ✓ In this operation, the element is deleted from the front end of the queue.
- ✓ Before implementing the operation, we first have to check whether the queue is empty or not.
- ✓ If the queue is empty, i.e., front = -1, it is the underflow condition, and we cannot perform the deletion.
- ✓ If the queue is not full, then the element can be inserted from the front end by using the below conditions -
  - If the deque has only one element, set rear = -1 and front = -1.
  - Else if front is at end (that means front = size - 1), set front = 0.
  - Else increment the front by 1, (i.e., front = front + 1).

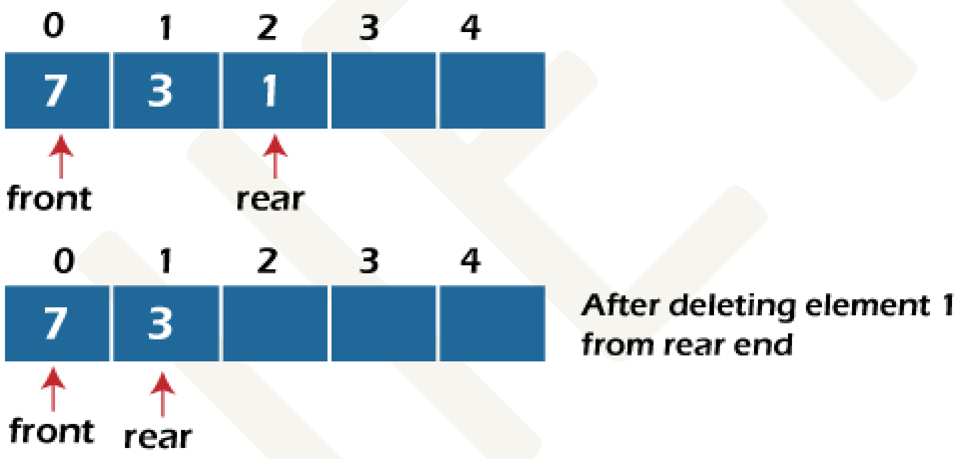




9. Deletion At The Rear End

DELETION AT THE REAR END

- ✓ In this operation, the element is deleted from the rear end of the queue.
- ✓ Before implementing the operation, we first have to check whether the queue is empty or not.
- ✓ If the queue is empty, i.e., front = -1, it is the underflow condition, and we cannot perform the deletion.
  - If the deque has only one element, set rear = -1 and front = -1.
  - If rear = 0 (rear is at front), then set rear = n - 1.
  - Else, decrement the rear by 1 (or, rear = rear -1).



✓ Check empty

- This operation is performed to check whether the deque is empty or not. If front = -1, it means that the deque is empty.

✓ Check full

- This operation is performed to check whether the deque is full or not. If front = rear + 1, or front = 0 and rear = n - 1 it means that the deque is full.
- The time complexity of all of the above operations of the deque is O(1), i.e., constant.

✓ getFront()

- Gets the front item from queue.

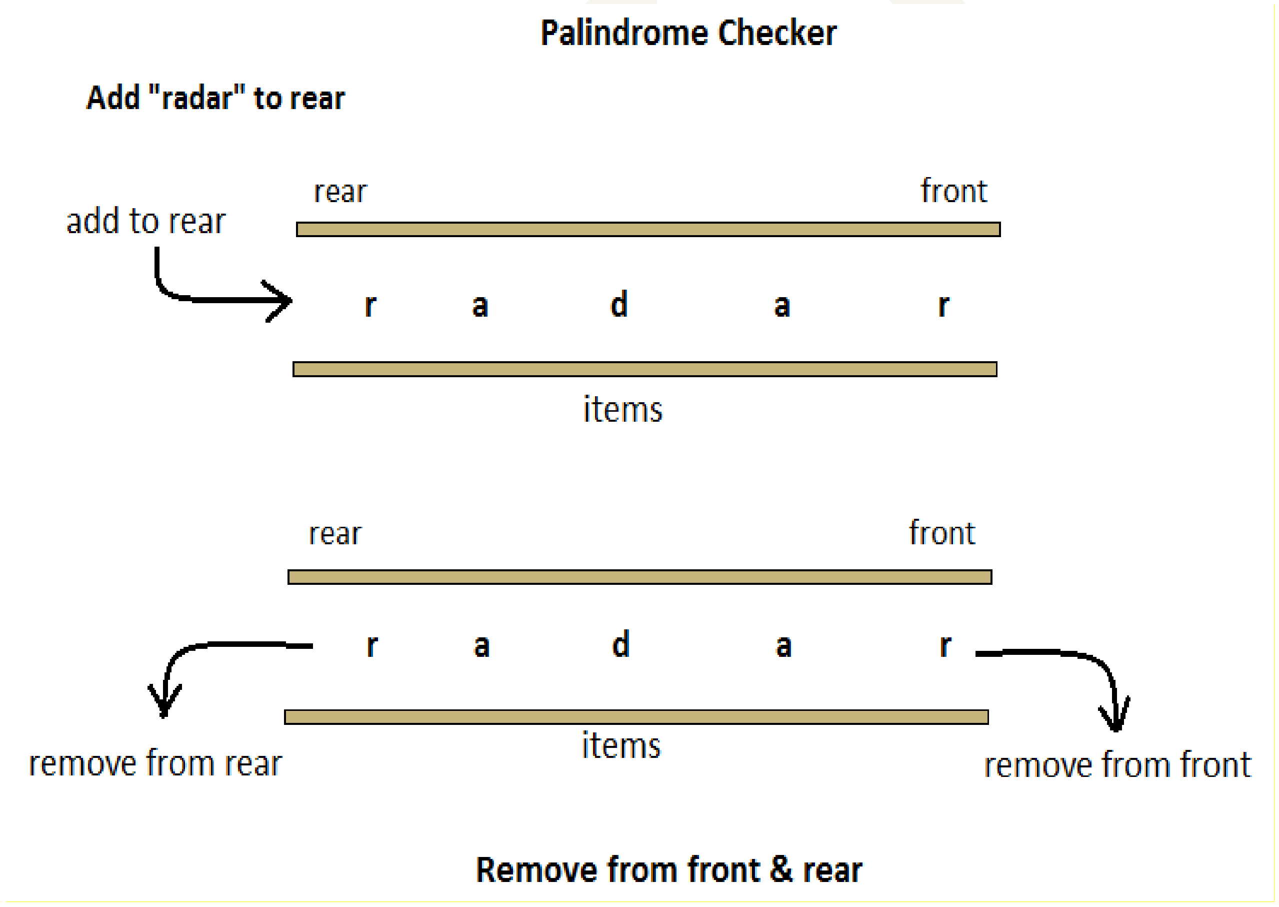
✓ getRear()

- Gets the last item from queue.

10. Application of Queue

Applications of deque

- ✓ Deque can be used as both stack and queue, as it supports both operations.
- ✓ Deque can be used as a palindrome checker means that if we read the string from both ends, the string would be the same.
- ✓ The Deque data structure supports clockwise and anticlockwise rotations in O(1) time which can be useful in certain applications. Also, the problems where elements need to be removed and or added to both ends can be efficiently solved using Deque.



A-Steal job scheduling algorithm

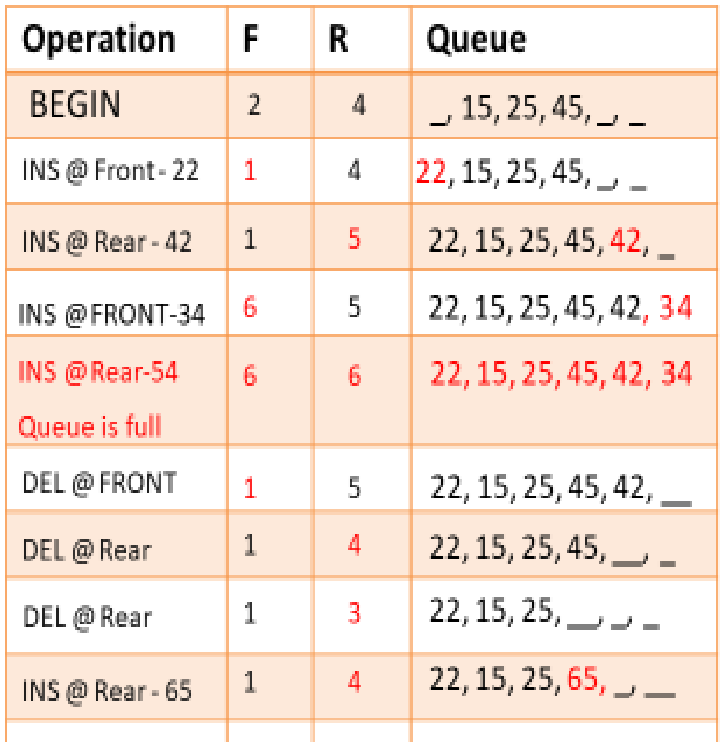
The A-Steal algorithm implements task scheduling for several processors (multiprocessor scheduling). The processor gets the first element from the deque. When one of the processors completes execution of its own threads it can steal a thread from another processor. It gets the last element from the deque of another processor and executes it.

Some Practical Applications of Deque:

- ✓ Applied as both stack and queue, as it supports both operations.
- ✓ Storing a web browser's history.
- ✓ Storing a software application's list of undo operations.
- ✓ Job scheduling algorithm

Consider an Example below –

Q. Trace the F & R for the given below Queue (given that – initially at front is at 2 and rear is at 4. Size of Queue is 8.



Operation	F	R	Queue
BEGIN	2	4	_, 15, 25, 45, _, _
INS @ Front - 22	1	4	22, 15, 25, 45, _, _
INS @ Rear - 42	1	5	22, 15, 25, 45, 42, _
INS @FRONT-34	6	5	22, 15, 25, 45, 42, 34
INS @Rear-54 Queue is full	6	6	22, 15, 25, 45, 42, 34
DEL @ FRONT	1	5	22, 15, 25, 45, 42, __
DEL @ Rear	1	4	22, 15, 25, 45, __, _
DEL @ Rear	1	3	22, 15, 25, __, _, _
INS @ Rear - 65	1	4	22, 15, 25, 65, _, __

11. Write A Java Program to Implement the Operation of Circular Deque using Array.

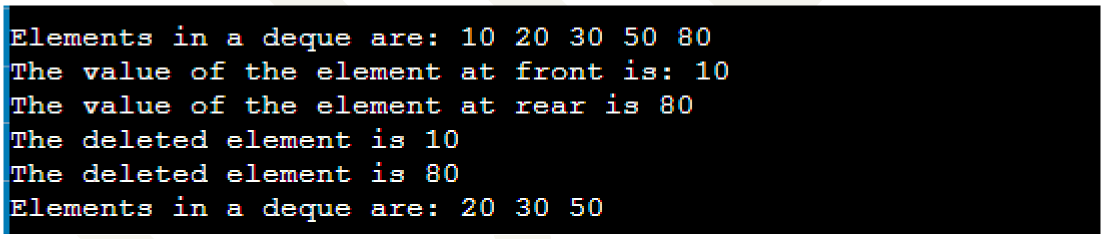
```java
import java.util.*;
class CdeQueue //created a class named CdeQueue where we will create all the method of inster, delete display//
{
    int front , rear, size;
    int a[];
    CdeQueue(int size) //created a parameterized constructor to create a Queue//
    {
        this.size = size; //the size inserted by user will be given to the variable size using this keyword//
        front = -1; // setting front and rear at -1//
        rear = -1;
        a = new int[size]; //declaring an array of the size which was entered by user//
    }
    void insertFront(int x) //created a method to insert an element at front//
    {
        if((front==0 && rear==size-1) || (front==rear+1))    //checking the overflow condition
        {
            System.out.println("Overflow"); //if true print overflow/
        }
        else if((front==-1) && (rear==-1))    //if both front & rear 0 then insert element//
        {
            front=rear=0;
            a[front]=x;
        }
        else if(front==0)
        {
            front=size-1;
            a[front]=x;
        }
        else
        {
            front=front-1;
            a[front]=x;
        }
    }
    void insertRear(int x) //created a method to insert an element at rear//
    {
        if((front==0 && rear==size-1) || (front==rear+1)) //checking the overflow condition//
        {
            System.out.println("Overflow");
        }
        else if((front==-1) && (rear==-1))    //if front and rear both -2 then insert at rear//
        {
            rear=0;
            a[rear]=x;
        }
        else if(rear==size-1)    //if rear is size-1 then set rear to 0 and then insert element//
        {
            rear=0;
            a[rear]=x;
        }
        else
        {
            rear++;
            a[rear]=x;
        }
    }
    void deleteFront() //created a method to delete an element from front//
    {
        if((front==-1) && (rear==-1)) //checking if the Dequeue is empty or not//
        {
            System.out.println("Deque is empty");
        }
        else if(front==rear) //if front & rear are same then a[front] gets deleted and front and rear set to -1//
        {
            System.out.println("\nThe deleted element is " + a[front]);
            front=-1;
            rear=-1;

        }
         else if(front==(size-1))
         {
            System.out.println("\nThe deleted element is " + a[front]);
             front=0;
         }
         else
         {
            System.out.println("\nThe deleted element is "+ a[front]);
              front=front+1;
         }
    }
    void deleteRear() //created a method to delete an element from rear//
    {
        if((front==-1) && (rear==-1))
        {
            System.out.println("Deque is empty");    //checking if the Dequeue is empty or not//
        }
        else if(front==rear)    //if front & rear are same then a[front] gets deleted and front and rear set to -1//
        {
            System.out.println("\nThe deleted element is "+ a[rear]);
            front=-1;
            rear=-1;

        }
        else if(rear==0)
        {
            System.out.println("\nThe deleted element is "+ a[rear]);
            rear=size-1;
        }
        else
        {
            System.out.println("\nThe deleted element is "+ a[rear]);
            rear=rear-1;
        }
    }
    void getFront() //created a method to get an element from front//
    {
        if(front != -1)
        {
            System.out.println("The front is at position " + front + "and having value " + a[front]);
        }
        else
        {
            System.out.println("The front is at position -1 ");
        }
    }
    void getRear() //created a method to get an element from rear//
    {
        if(rear != -1)
        {
            System.out.println("The rear is at position " + rear + "and having value " + a[rear]);
        }
        else
        {
            System.out.println("The rear is at position -1 ");
        }
    }
    void display() //created a method to display the queue//
    {
        int i=front;
        System.out.println("\nElements in a deque are: ");

        while(i!=rear)
        {
            System.out.print(a[i] + "-");
            i=(i+1)%size;
        }
        System.out.println(a[rear]);
    }
}
class Return //create a run class//
{
    public static void main(String args[])
    {
        CdeQueue cq = new CdeQueue(5);
        cq.insertFront(20);
        cq.insertFront(10);
        cq.insertRear(30);
        cq.insertRear(50);
        cq.insertRear(80);
        cq.display();  // Calling the display function to retrieve the values of deque
        cq.getFront();  // Retrieve the value at front-end
        cq.getRear();  // Retrieve the value at rear-end
        cq.deleteFront();
        cq.deleteRear();
        cq.display(); // calling display function to retrieve values after deletion
    }
}
```

OUTPUT:



```
Elements in a deque are: 10 20 30 50 80
The value of the element at front is: 10
The value of the element at rear is 80
The deleted element is 10
The deleted element is 80
Elements in a deque are: 20 30 50
```

12. Priority Queue

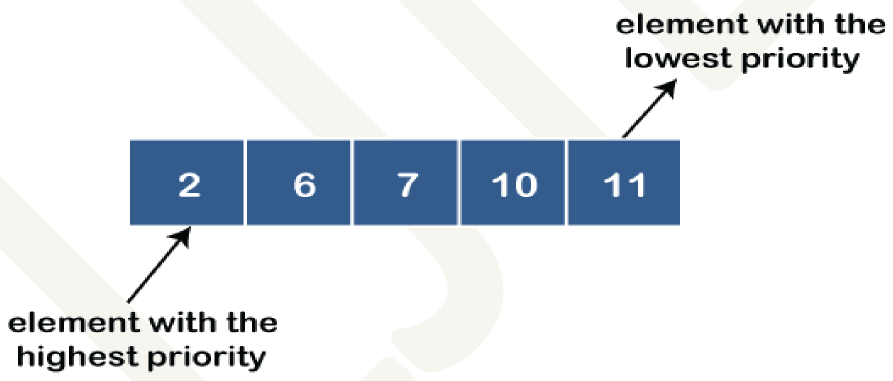
Priority Queue

- ✓ A priority queue is a special type of queue in which each element is associated with a priority and is served according to its priority
  - ❑ The insertion and deletion in queue occur according to the priority.
  - ❑ The element with higher priority inserts/deletes first

13. Types of Priority Queue: -

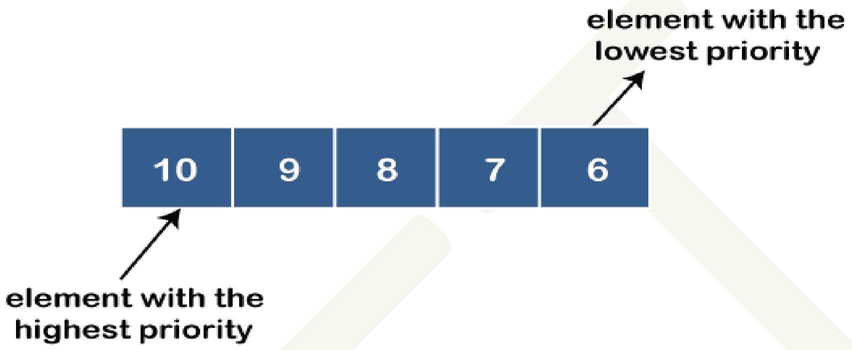
a. Ascending Priority Queue (Low to High Priority)

- ✓ As the name suggests, in ascending order priority queue, the element with a lower priority value is given a higher priority in the priority list. For example, if we have the following elements in a priority queue arranged in ascending order like 4,6,8,9,10. Here, 4 is the smallest number, therefore, it will get the highest priority in a priority queue.


element with the lowest priority
2
6
7
10
11
element with the highest priority

b. Descending Priority Queue (High to Low Priority)

✓ As the name suggests, in descending order priority queue, the element with a higher priority value is given a higher priority in the priority list. For example, if we have the following elements in a priority queue arranged in descending order like 10,9,8,6,4. Here, 10 is the largest number, therefore, it will get the highest priority in a priority queue.


element with the lowest priority
10
9
8
7
6
element with the highest priority

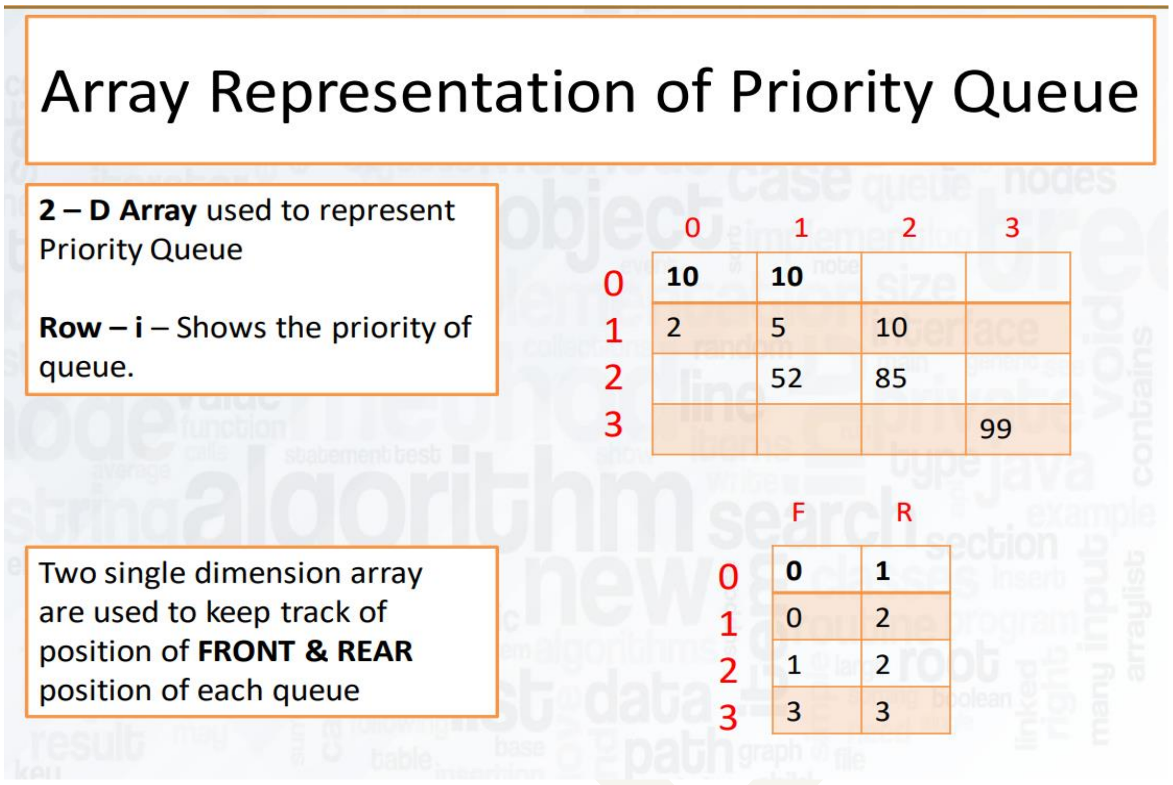
Suppose we are keeping track of a line of patients entering a hospital emergency room. In a queue, the first person to enter the emergency room is the first person to see the doctor (FIFO).

But what if a patient comes in who needs immediate medical attention (priority 1)? In a priority queue, each element is assigned a unique priority.

A priority queue is different from a "normal" queue, because instead of being a "first-in-first-out" data structure, values come out in order by priority. A priority queue might be used, for example, to handle the jobs sent to the Computer Science Department's printer: Jobs sent by the department chair should be printed first, then jobs sent by professors, then those sent by graduate students, and finally those sent by undergraduates. The values put into the priority queue would be the priority of the sender (e.g., using 4 for the chair, 3 for professors, 2 for grad students, and 1 for undergrads), and the associated information would be the document to print.

Each time the printer is free, the job with the highest priority would be removed from the print queue, and printed. (Note that it is OK to have multiple jobs with the same priority; if there is more than one job with the same highest priority when the printer is free, then any one of them can be selected.)

Priority queues are among the most useful of all collections. A priority queue is a collection in which items can be added at any time, but the only item that can be removed is the one with the highest priority.



Array Representation of Priority Queue

2 – D Array used to represent Priority Queue

Row – i – Shows the priority of queue.

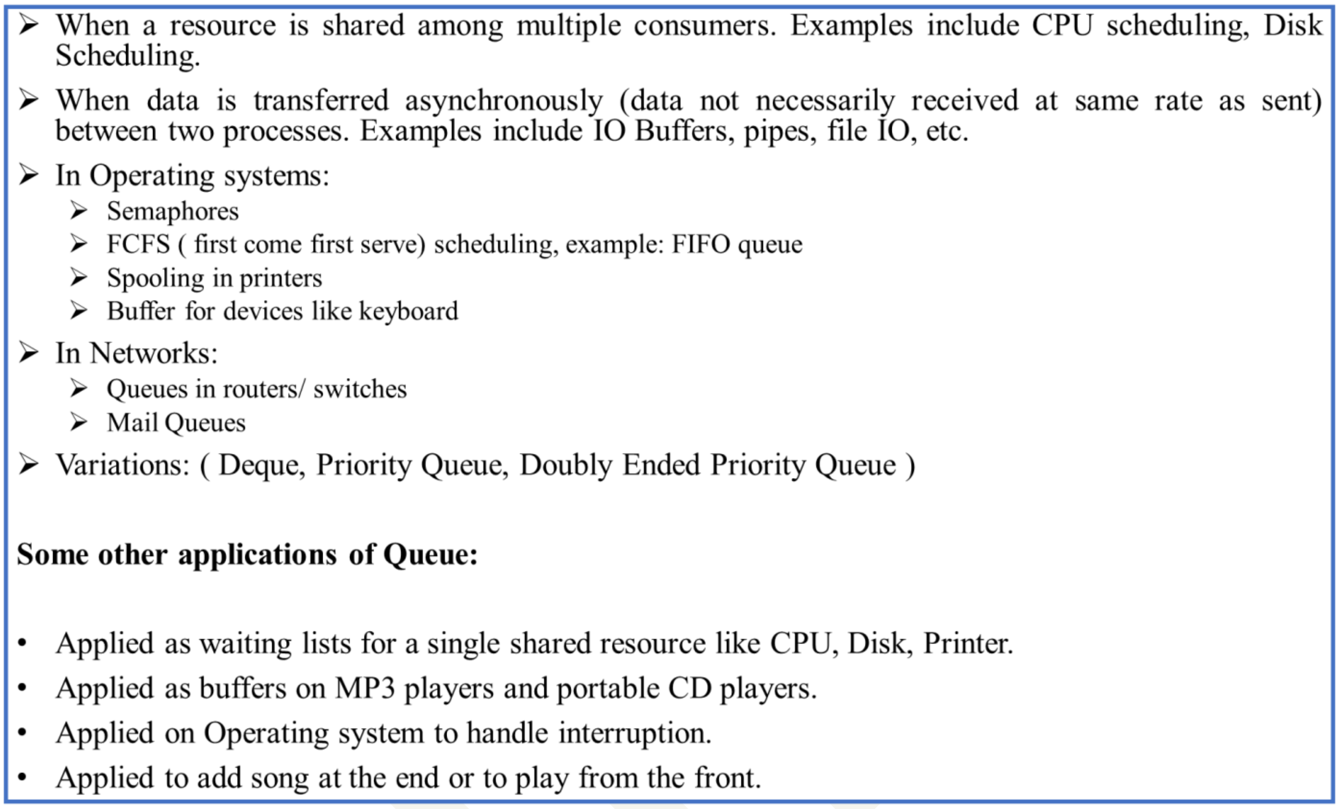
	0	1	2	3
0	**10**	**10**		
1	2	5	10	
2		52	85	
3				99

Two single dimension array are used to keep track of position of **FRONT & REAR** position of each queue

	F	R
0	**0**	**1**
1	0	2
2	1	2
3	3	3

14. Application of Queue

Applications of Queue



- When a resource is shared among multiple consumers. Examples include CPU scheduling, Disk Scheduling.
- When data is transferred asynchronously (data not necessarily received at same rate as sent) between two processes. Examples include IO Buffers, pipes, file IO, etc.
- In Operating systems:
  - Semaphores
  - FCFS (first come first serve) scheduling, example: FIFO queue
  - Spooling in printers
  - Buffer for devices like keyboard
- In Networks:
  - Queues in routers/ switches
  - Mail Queues
- Variations: (Deque, Priority Queue, Doubly Ended Priority Queue)

Some other applications of Queue:

- Applied as waiting lists for a single shared resource like CPU, Disk, Printer.
- Applied as buffers on MP3 players and portable CD players.
- Applied on Operating system to handle interruption.
- Applied to add song at the end or to play from the front.