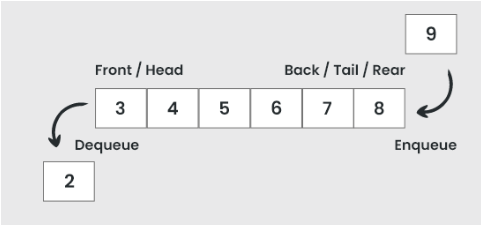
**Theory**

A **Queue** is defined as a linear data structure that is open at both ends and the operations are performed in **First In First Out (FIFO)**order.



## Basic Operations

Some of the basic operations for Queue in Data Structure are:

1. **Enqueue() –** Adds (or stores) an element to the end of the queue.
2. **Dequeue() –** Removal of elements from the queue.
3. **Peek() or front()-** Acquires the data element available at the front node of the queue without deleting it.
4. **rear() –** This operation returns the element at the rear end without removing it.
5. **isFull() –** Validates if the queue is full.
6. **isNull() –** Checks if the queue is empty.

class MyQueue<T>{

    int f,r,size;

    T arr[];

    MyQueue(int size){

        this.f=-1;

        this.r=-1;

        this.size=size;

        //cant create array with type parameter directly,

        //creating array of objects then casting to T

        this.arr= (T[])new Object[this.size];

    }

    boolean isFull() {

        return (this.r==this.size-1);

    }

    boolean isEmpty() {

        return (this.f==this.r);

    }

    void enqueue(T item) {

        if(isFull()) {

            System.out.println("Overflow");

        }

        else {

            r++;

            arr[r]=item;

        }

    }

    T dequeue() {

        T temp = null;

        if(isEmpty()) {

            System.out.println("Underflow");

        }

        else{

            f++;

            temp = arr[f];

        }

        return temp;

    }

    void display() {

        if(!isEmpty()) {

            for (int i = f+1; i <= r; i++) {

                System.out.print(arr[i] + " ");

            }

            System.out.println("\n");

        }

        else {

            System.out.println("Empty queue");

        }

    }

}

**Applications of Queue**

* **Multi programming:**Multi programming means when multiple programs are running in the main memory. It is essential to organize these multiple programs and these multiple programs are organized as queues.
* **Network:**In a network, a queue is used in devices such as a router or a switch. another application of a queue is a mail queue which is a directory that stores data and controls files for mail messages.
* **Job Scheduling:**The computer has a task to execute a particular number of jobs that are scheduled to be executed one after another. These jobs are assigned to the processor one by one which is organized using a queue.
* **Shared resources:**Queues are used as waiting lists for a single shared resource.

**Real-time application of Queue:**

* ATM Booth Line
* Ticket Counter Line
* Key press sequence on the keyboard
* CPU task scheduling
* Waiting time of each customer at call centers.

**Advantages of Queue:**

* A large amount of data can be managed efficiently with ease.
* Operations such as insertion and deletion can be performed with ease as it follows the first in first out rule.
* Queues are useful when a particular service is used by multiple consumers.
* Queues are fast in speed for data inter-process communication.
* Queues can be used in the implementation of other data structures.

**Disadvantages of Queue:**

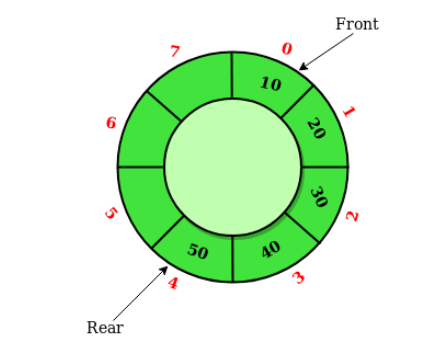
* The operations such as insertion and deletion of elements from the middle are time consuming.
* Limited Space.
* In a classical queue, a new element can only be inserted when the existing elements are deleted from the queue.
* Searching an element takes O(N) time.
* Maximum size of a queue must be defined prior.

## [Types of Queue:](https://www.geeksforgeeks.org/different-types-of-queues-and-its-applications/)

There are different types of queues:

1. **Input Restricted Queue:** This is a simple queue. In this type of queue, the input can be taken from only one end but deletion can be done from any of the ends.
2. **Output Restricted Queue:** This is also a simple queue. In this type of queue, the input can be taken from both ends but deletion can be done from only one end.
3. [Circular Queue](https://www.geeksforgeeks.org/introduction-and-array-implementation-of-circular-queue/)**:** This is a special type of queue where the last position is connected back to the first position. Here also the operations are performed in FIFO order.
4. [Double-Ended Queue (Dequeue)](https://www.geeksforgeeks.org/deque-set-1-introduction-applications/)**:** In a double-ended queue the insertion and deletion operations, both can be performed from both ends.
5. [Priority Queue](https://www.geeksforgeeks.org/priority-queue-set-1-introduction/)**:** A priority queue is a special queue where the elements are accessed based on the priority assigned to them.

**[Circular Queue](https://www.geeksforgeeks.org/circular-queue-set-1-introduction-array-implementation/)**



Circular Queue is a linear data structure in which the operations are performed based on FIFO (First In First Out) principle and the last position is connected back to the first position to make a circle. It is also called **‘Ring Buffer’**. This queue is primarily used in the following cases:

1. **Memory Management:** The unused memory locations in the case of ordinary queues can be utilized in circular queues.
2. **Traffic system:** In a computer-controlled traffic system, circular queues are used to switch on the traffic lights one by one repeatedly as per the time set.
3. **CPU Scheduling:** Operating systems often maintain a queue of processes that are ready to execute or that are waiting for a particular event to occur.

The time complexity for the circular Queue is O(1).

public boolean enqueue(int value) {

    if( (front==0 && rear==size-1) ||

    (rear==(front-1)%(size-1)) ){

        return false;

    }

    // empty queue

    if(front==-1){

        front = 0;

        rear = 0;

        queue.add(rear, value);

    }

    else if(front!=0 && rear==size-1){

        // to maintain cyclic nature

        rear = 0;

        queue.set(rear, value);

    }

    else{

        rear++;

        if(front<=rear){

            // adding new value

            queue.add(rear, value);

        }

        else{

            // The set() method of java.util.ArrayList class is used to replace the element at the specified position in this list with the specified element.

            queue.set(rear, value);

        }

    }

    return true;

}

public int dequeue() {

    int ans;

    // empty queue

    if(front==-1){

        return -1;

    }

    ans = queue.get(front);

    // single element

    if(front==rear){

        front=-1;

        rear=-1;

    }

    else if(front==size-1){

        // to maintain cyclic nature

        front=0;

    }

    else{

        front++;

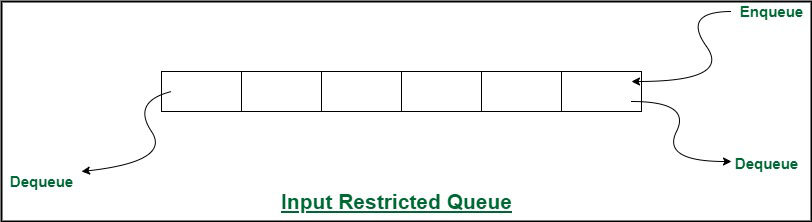
    }

    return ans;

}

**Input restricted Queue**

In this type of Queue, the input can be taken from one side only(rear) and deletion of elements can be done from both sides(front and rear). This kind of Queue does not follow FIFO(first in first out).  This queue is used in cases where the consumption of the data needs to be in FIFO order but if there is a need to remove the recently inserted data for some reason and one such case can be irrelevant data, performance issue, etc.



**Advantages of Input restricted Queue:**

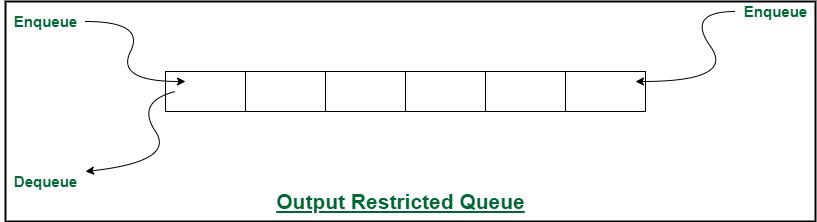
* Prevents overflow and overloading of the queue by limiting the number of items added
* Helps maintain stability and predictable performance of the system

**Disadvantages of Input restricted Queue:**

* May lead to resource wastage if the restriction is set too low and items are frequently discarded
* May lead to waiting or blocking if the restriction is set too high and the queue is full, preventing new items from being added.

**Output restricted Queue**

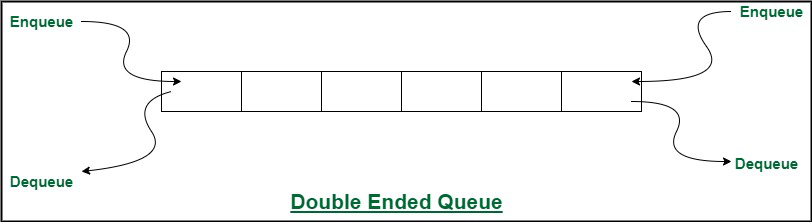
In this type of Queue, the input can be taken from both sides(rear and front) and the deletion of the element can be done from only one side(front).  This queue is used in the case where the inputs have some priority order to be executed and the input can be placed even in the first place so that it is executed first.



**[Double ended Queue](https://www.geeksforgeeks.org/deque-set-1-introduction-applications/)**

 Double Ended Queue is also a Queue data structure in which the insertion and deletion operations are performed at both the ends (front and rear). That means, we can insert at both front and rear positions and can delete from both front and rear positions.  Since Deque supports both stack and queue operations, it can be used as both. 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 both ends can be efficiently solved using Deque.

* **Multi-level undo/redo functionality:** Deques can be used to implement undo and redo functionality in applications. Each time a user performs an action, the current state of the application is pushed onto the deque. When the user undoes an action, the front of the deque is popped, and the previous state is restored. When the user redoes an action, the next state is popped from the deque.
* In computer science, deque can be used in many algorithms like *LRU Cache*, *Round Robin Scheduling*, *Expression Evaluation*.



**[Priority Queue](https://www.geeksforgeeks.org/priority-queue-set-1-introduction/)**

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. There are two types of Priority Queues. They are:

1. **Ascending Priority Queue:** Element can be inserted arbitrarily but only smallest element can be removed. For example, suppose there is an array having elements 4, 2, 8 in the same order. So, while inserting the elements, the insertion will be in the same sequence but while deleting, the order will be 2, 4, 8.
2. **Descending priority Queue:** Element can be inserted arbitrarily but only the largest element can be removed first from the given Queue. For example, suppose there is an array having elements 4, 2, 8 in the same order. So, while inserting the elements, the insertion will be in the same sequence but while deleting, the order will be 8, 4, 2.

The time complexity of the Priority Queue is O(logn).

**322\_Reverse a Queue using recursion**

**Ap 1 : using stack**

1. Pop the elements from the queue and insert into the stack now topmost element of the stack is the last element of the queue.
2. Pop the elements of the stack to insert back into the queue the last element is the first one to be inserted into the queue.

Time complexity: O( N ) Space complexity: O( N )

**Ap 2 : using recursion**

Time complexity: O( N ) Space complexity: O( N )

public Queue<Integer> rev(Queue<Integer> q) {

    if (q.isEmpty()) {

        return q; // Base case: if queue is empty, return the queue

    } else {

        int front = q.poll(); // Remove the front element of the queue

        q = rev(q); // Recursively reverse the remaining queue

        q.add(front); // Add the front element to the rear of the reversed queue

        return q;

    }

}

**323\_Reverse the first “K” elements of a queue**

Time complexity: O( N ) Space complexity: O( K )

public Queue<Integer> modifyQueue(Queue<Integer> q, int k) {

        Stack<Integer> st = new Stack<>();

        int n = q.size();

        // put first k elements in stack

        for(int i=0; i<k; i++){

            st.push(q.poll());

        }

        // fetch from stack, inert into queue

        while(!st.empty()){

            q.add(st.pop());

        }

        // remove first (n-k) elements from front of queue

        // add thoes elemen at rear

        for(int i=0; i<(n-k); i++){

            int val=q.poll();

            q.add(val);

        }

        return q;

    }

**First non-repeating character in a stream**

1. Initialize a Queue to maintain the characters in the order they appear.
2. Initialize a char array freq of size 26 to keep track of the frequency of each character.
3. Iterate through each character ch in the input string A.
   1. Increment the frequency of the current character ch by 1 in the freq array
   2. Add the current character ch to the queue
   3. While the queue is not empty, check if the frequency of the first character in the queue (peek)
      1. If it is greater than 1, remove the first character from the queue (poll).
      2. If it is not greater than 1, append the first character from the queue to the answer and break out of the loop.
   4. If the queue becomes empty after the above loop, it means there are no non-repeating characters encountered so far, so append '#' to the answer.
4. Return the answer as a string.

Time complexity: O( N ) Space complexity: O( N )

**324\_Interleave the first half of the queue with second half**

**AP 1 : using queue**

1. Define a queue
2. Push first half elements into new queue
3. Pop from both queue and add in alternative order

**AP 2 : using stack**

1. Pop first half elements from queue and push into stack
2. Pop from stack and add into queue
3. Pop first half elements from queue and add into last
4. Push first half elements from queue and push into stack
5. Pop from stack and queue, add into queue alternatively

public static void interLeaveQueue(Queue < Integer > q) {

        Stack<Integer> st = new Stack<>();

        int mid=q.size()/2;

        for(int i=0; i<mid; i++){

            st.push(q.poll());

        }

        while(!st.empty()){

            q.add(st.pop());

        }

        for(int i=0; i<mid; i++){

            int val=q.poll();

            q.add(val);

        }

        for(int i=0; i<mid; i++){

            st.push(q.poll());

        }

        while(!st.empty()){

            int val=q.poll();

            q.add(st.pop());

            q.add(val);

        }

    }

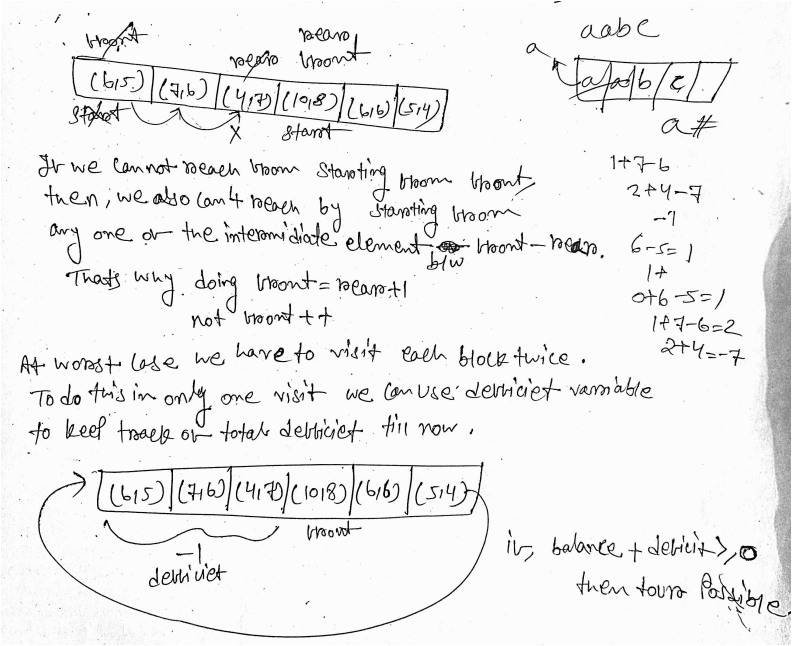
**325\_first circular tour that visits all Petrol Pumps**

**Ap 1 : nested loop**

Fix one element and check if tour possible for that element or not?

Time complexity: O( N^2 ) Space complexity: O( 1 )

**Ap 2 : concept of queue**



1. Initialize three variables: start to keep track of the starting gas station index, deficit to accumulate the deficit encountered during traversal, and balance to keep track of the current balance.
2. Iterate over each gas station starting from index 0 up to the last gas station:
   1. Calculate the balance by adding the difference between the gas at the current station and the cost of traveling to the next station.
   2. If the balance becomes negative, indicating that we can't travel further from the current station:
      1. Update the start index to the next station (i + 1).
      2. Accumulate the deficit by adding the negative balance (the amount needed to cover the deficit).
      3. Reset the balance to 0, indicating the starting point of a new circuit.
3. After the loop, check if the sum of deficit and balance is greater than or equal to zero.
   1. If it is, return the start index, indicating a valid starting gas station to complete the circuit.
   2. If not, return -1, indicating that there is no possible starting gas station to complete the circuit.

Time complexity: O( N ) Space complexity: O( 1 )

public int canCompleteCircuit(int[] gas, int[] cost) {

        int start=0;

        int deficit=0;

        int balance=0;

        for(int i=0; i<gas.length; i++){

            balance+=gas[i]-cost[i];

            if(balance<0){

                start=i+1;

                deficit+=balance;

                balance=0;

            }

        }

        if(deficit+balance>=0){

            return start;

        }

        else{

            return -1;

        }

    }

**328\_First negative integer in every window of size “k”**

**Ap 1 : using queue**

1. Initialize a deque to store indices of negative integers in the current window. Also, initialize an array ans to store the result, with size n - k + 1.
2. Iterate through the first window of size k:
   1. For each element within this window, if the element is negative, add its index to the deque.
3. If the deque is empty after processing the window,
   1. set the corresponding element in ans to 0;
   2. otherwise, set it to arr[dq.peek()]
4. Process the remaining windows:
   1. For each subsequent window, slide the window by removing the leftmost element index from the deque if it's out of the current window's range.
   2. If the current element within the window is negative, add its index to the deque.
   3. Update ans[j] with arr[dq.peek()]. If the deque is empty, set ans[j] to 0.
5. Return the ans array

Time complexity: O( N ) Space complexity: O( N )

public static int[] firstNegative(int[] arr, int n, int k) {

        Queue<Integer> dq = new ArrayDeque<>(k);

        int[] ans = new int[n-k+1];

        int j=0;

        // processing first K window

        for(int i=0; i<k; i++){

            if(arr[i] < 0){

                dq.add(i);

            }

        }

        if(dq.isEmpty()){

            ans[j++]=0;

        }

        else{

            ans[j++]=arr[dq.peek()];

        }

        // precess remaining window

        for(int i=k; i<n; i++){

            if( !dq.isEmpty() && (i-dq.peek() >= k)){

                dq.poll();

            }

            // addition

            if(arr[i]<0){

                dq.add(i);

            }

            // adding into ans

            if(dq.isEmpty()){

                ans[j++]=0;

            }

            else{

                ans[j++]=arr[dq.peek()];

            }

        }

        return ans;

    }

**Ap 2 : two-pointers**

1. Declare an array ‘ANS’ of size (N - K + 1) i.e. total windows in the array, to store our answer.
2. Initialize ‘firstNegIndex’ to 0 and ‘end’ to K-1.
3. Loop till ‘end’ < N
   1. Increment ‘firstNegIndex’ by 1 until we find a negative element or we are out of the window.
   2. If there is no negative element in the window. Update 0 in ‘ANS’.
   3. Else, Update ARR[firstNegIndex] in ‘ANS’ for the current window.
4. Return ‘ANS’.

Time complexity: O( N ) Space complexity: O( 1 )

public static int[] firstNegative(int[] arr, int n, int k) {

        int ans[] = new int[n - k + 1];

        int idx = 0;

        int firstNegIndex = 0;

        // Traversing for every window's end.

        for (int end = k - 1; end < n; end++) {

            while (firstNegIndex < end && (firstNegIndex <= end - k || arr[firstNegIndex] >= 0)) {

                firstNegIndex++;

            }

            // If we found a negative.

            if (arr[firstNegIndex] < 0) {

                ans[idx++] = arr[firstNegIndex];

            } else {

                ans[idx++] = 0;

            }

        }

        return ans;

    }

**330\_Sum of minimum and maximum elements of all subarrays of size “k”**

https://www.codingninjas.com/studio/problems/sum-of-minimum-and-maximum-elements-of-all-subarrays-of-size-k\_1171047?leftPanelTabValue=SOLUTION