**Queue Data Structure**

A **Queue Data Structure**is a fundamental concept in computer science used for storing and managing data in a specific order.

* It follows the principle of "**First in, First out**" **(FIFO)**, where the first element added to the queue is the first one to be removed.
* It is used as a buffer in computer systems where we have speed mismatch between two devices that communicate with each other. For example, CPU and keyboard and two devices in a network
* Queue is also used in Operating System algorithms like CPU Scheduling and Memory Management, and many standard algorithms like Breadth First Search of Graph, Level Order Traversal of a Tree.

**FIFO Principle in Queue:**

FIFO Principle states that the first element added to the Queue will be the first one to be removed or processed. So, Queue is like a line of people waiting to purchase tickets, where the first person in line is the first person served. (i.e. First Come First Serve).

A screenshot of a computer

AI-generated content may be incorrect.

**Basic Terminologies of Queue**

* **Front:**Position of the entry in a queue ready to be served, that is, the first entry that will be removed from the queue, is called the front of the queue. It is also referred as the head of the queue.
* **Rear:** Position of the last entry in the queue, that is, the one most recently added, is called the rear of the queue. It is also referred as the tail of the queue.
* **Size:** Size refers to the current number of elements in the queue.
* **Capacity:** Capacity refers to the maximum number of elements the queue can hold.

A diagram of a type of queue

AI-generated content may be incorrect.

**1. Simple Queue**

A simple queue follows the FIFO (First In, First Out) principle.

* Insertion is allowed only at the rear (back).
* Deletion is allowed only from the front.
* Can be implemented using a linked list or a circular array.

When an array is used, we often prefer a **circular queue**, which is mainly an efficient array implementation of a simple queue. It efficiently utilizes memory by reusing the empty spaces left after deletion, avoiding wastage that occurs in a normal linear array implementation..

**2. Double-Ended Queue (Deque)**

In a deque, insertion and deletion can be performed from both ends.

**3. Priority Queue**

A queue where each element is assigned a **priority**, and deletion always happens based on priority (not just position).

**Basic Operations for Queue Data Structure**

Queue is a linear data structure that follows the FIFO (First In First Out) principle, where insertion is done at the rear end and deletion is done from the front end.

The following are some fundamental operations that allow us to add, remove, and access elements efficiently.

1. **Enqueue**: Adds an element to the end (rear) of the queue. If the queue is full, an overflow error occurs.
2. **Dequeue**: Removes the element from the front of the queue. If the queue is empty, an underflow error occurs.
3. **Peek/Front**: Returns the element at the front without removing it.
4. **Size**: Returns the number of elements in the queue.
5. **isEmpty**: Returns true if the queue is empty, otherwise false.
6. **isFull**: Returns true if the queue is full, otherwise false.

* **enqueue() -** Insertion of elements to the queue.
* **dequeue() -** Removal of elements from the queue.
* **getFront()-** Acquires the data element available at the front node of the queue without deleting it.
* **getRear() -** This operation returns the element at the rear end without removing it.
* **isEmpty() -** Checks if the queue is empty.
* **size() -**This operation returns the size of the queue i.e. the total number of elements it contains

A table with text and numbers

AI-generated content may be incorrect. A screenshot of a computer code

AI-generated content may be incorrect.

**enqueue()**

Enqueue operation inserts an element at the end of the queue i.e. at the rear end.

A screenshot of a computer screen

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect. A screenshot of a computer

AI-generated content may be incorrect.

**dequeue()**

Dequeue operation removes element that is at the front end of the queue.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect. A white background with yellow text

AI-generated content may be incorrect.

**Time Complexity:** O(1), since deleting from the front takes constant time.  
In JavaScript, there’s no built-in queue, so we use arrays. Removing elements with q.shift() takes O(n) time because all elements are re-indexed.

**Auxiliary Space:** O(1)

**Note:**If a queue is empty, deleting an element will cause an underflow condition.

**getFront()**

getFront will returns the element at the front end of the queue without removing it.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer code

AI-generated content may be incorrect.

**Time Complexity:** O(1), since accessing the front element takes constant time.  
**Auxiliary Space:** O(1)

**getRear()**

getRear will returns the element at the rear end without removing it.

A screenshot of a game

AI-generated content may be incorrect.

A computer screen shot of a code

AI-generated content may be incorrect.

**Time Complexity:** O(1)  
**Auxiliary Space:** O(1)

**isEmpty**

isEmpty returns a boolean value that indicates whether the queue is empty or not

A white rectangular object with black text

AI-generated content may be incorrect.

**Size()**

Size returns the size of the queue i.e. the total number of elements it contains.

A screenshot of a number of squares

AI-generated content may be incorrect.