Queue

The queue is an abstract data structure, somewhat similar to Stacks. Unlike stacks, a queue is open at both ends. One end is always used to insert data (enqueue), and the other is used to remove data (dequeue). Queue follows First-In-First-Out methodology, i.e., the data item stored first will be accessed first.

1. A queue can be defined as an ordered list that enables insert operations to be performed at one end called **REAR** and delete operations to be performed at another end called **FRONT**.

2. Queue is referred to be as the First In First Out list.

3. For example, people waiting in line for a rail ticket form a queue.

A real-world example of the queue can be a single-lane one-way road, where the vehicle enters first, exits first. More real-world examples can be seen as queues at the ticket windows and bus stops.

Queue Representation

As we now understand that in the queue, we access both ends for different reasons. The following diagram given below tries to explain queue representation as a data structure −

As in stacks, a queue can also be implemented using Arrays, Linked-lists, Pointers, and Structures. For the sake of simplicity, we shall implement queues using the one-dimensional array.

Basic Operations

Queue operations may involve initializing or defining the queue, utilizing it, and then completely erasing it from the memory. Here we shall try to understand the basic operations associated with queues −

- enqueue() − add (store) an item to the queue.
- dequeue() − remove (access) an item from the queue.

A few more functions are required to make the above-mentioned queue operation efficient. These are −

- peek() − Gets the element at the front of the queue without removing it.
- isfull() − Checks if the queue is full.
- isempty() − Checks if the queue is empty.

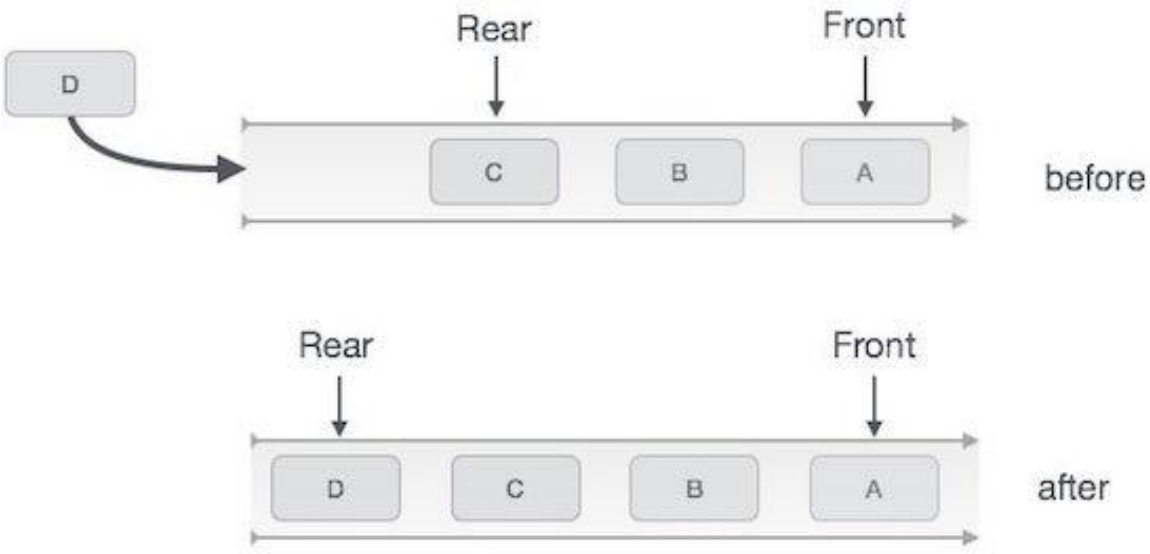
In the queue, we always dequeue (or access) data, pointed by the front pointer, and while enqueuing (or storing) data in the queue we take the help of the rear pointer.

Enqueue Operation

Queues maintain two data pointers, front, and rear. Therefore, its operations are comparatively difficult to implement than stacks.

The following steps should be taken to enqueue (insert) data into a queue −

- Step 1 − Check if the queue is full.
- Step 2 − If the queue is full, produce an overflow error and exit.
- Step 3 − If the queue is not full, increment the rear pointer to point to the next empty space.
- Step 4 − Add the data element to the queue location, where the rear is pointing.
- Step 5 − return success.

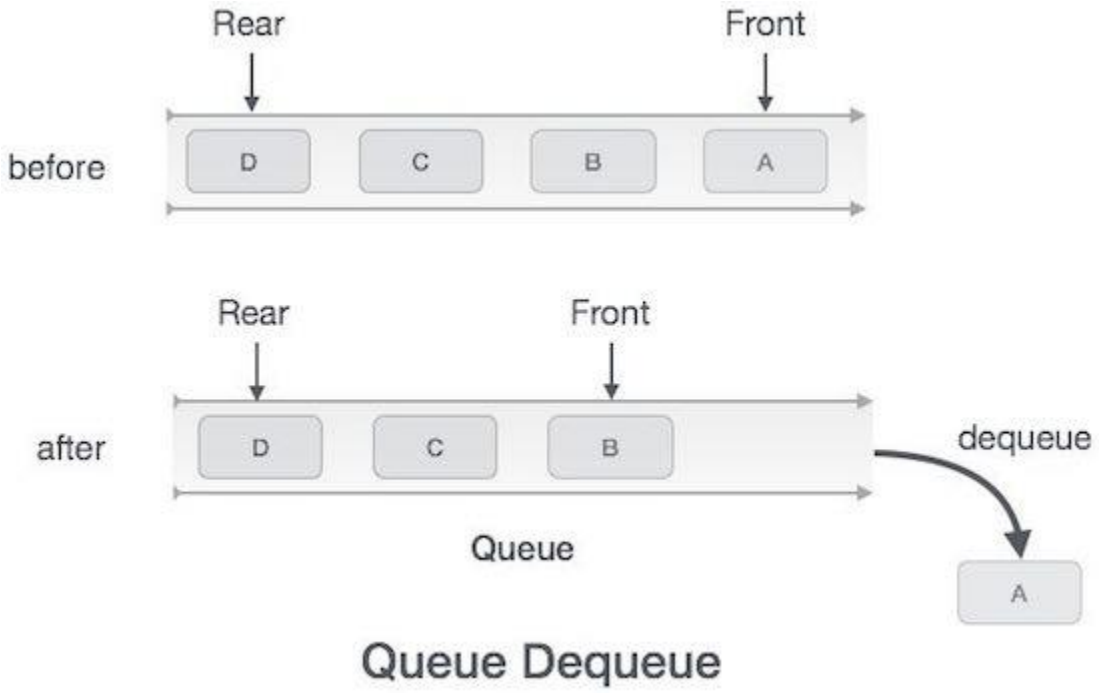


Queue Enqueue

Dequeue Operation

Accessing data from the queue is a process of two tasks − access the data where the front is pointing and remove the data after access. The following steps are taken to perform dequeue operation −

- Step 1 − Check if the queue is empty.
- Step 2 − If the queue is empty, produce an underflow error and exit.
- Step 3 − If the queue is not empty, access the data where the front is pointing.
- Step 4 − Increment front pointer to point to the next available data element.
- Step 5 − Return success.



Queue Dequeue

Applications of Queue

Due to the fact that queue performs actions on a first in first out basis which is quite fair for the ordering of actions. There are various applications of queues discussed below.

1. Queues are widely used as waiting lists for a single shared resource like printer, disk, CPU.
2. Queues are used in the asynchronous transfer of data (where data is not being transferred at the same rate between two processes) for eg. pipes, file IO, sockets.
3. Queues are used as buffers in most applications like MP3 media players, CD players, etc.
4. Queues are used to maintain the playlist in media players in order to add and remove the songs from the playlist.
5. Queues are used in operating systems for handling interrupts.

Complexity

Data Structure	**Time Complexity**								**Space Complexity**
	Average				**Worst**				**Worst**
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Queue	θ(n)	θ(n)	θ(1)	θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)