Queue

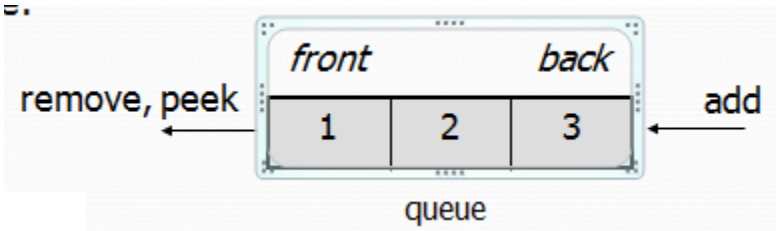
First-In-First-Out (FIFO)

Queue: Retrieves elements in the order they were added. First-In, First-Out ("FIFO")

Queue is a collection of data that is accessed in a first-in-first-out (FIFO) manner .

Basic queue operations:

add (enqueue): Add an element to the back.
remove (dequeue): Remove the front element.
peek: Examine the front element.



Basic Operations in queue:

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.

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

- Let's first learn about supportive functions of a queue −
- peek() : This function helps to see the data at the **front** of the queue. The algorithm of peek() function is as follows −

- peek() : This function helps to see the data at the **front** of the queue. The algorithm of peek() function is as follows −
- **Algorithm**

```
begin procedure peek
   return queue[front]
end procedure
```

- isfull() : As we are using single dimension array to implement queue, we just check for the rear pointer to reach at MAXSIZE to determine that the queue is full.
- **Algorithm**

```
begin procedure isfull
     if rear equals to MAXSIZE
      return true
    else
       return false
   endif
end procedure
```

- isempty(): Algorithm of isempty() function –
- **Algorithm**

begin procedure isempty

if front is less than MIN OR front is greater than rear

return true

else return false

endif

End procedure

- If the value of **front** is less than MIN or 0, it tells that the queue is not yet initialized, hence empty.

- **Enqueue**: Queues maintain two data pointers, **front** and **rear**.
- The following steps should be taken to enqueue (insert) data into a queue −

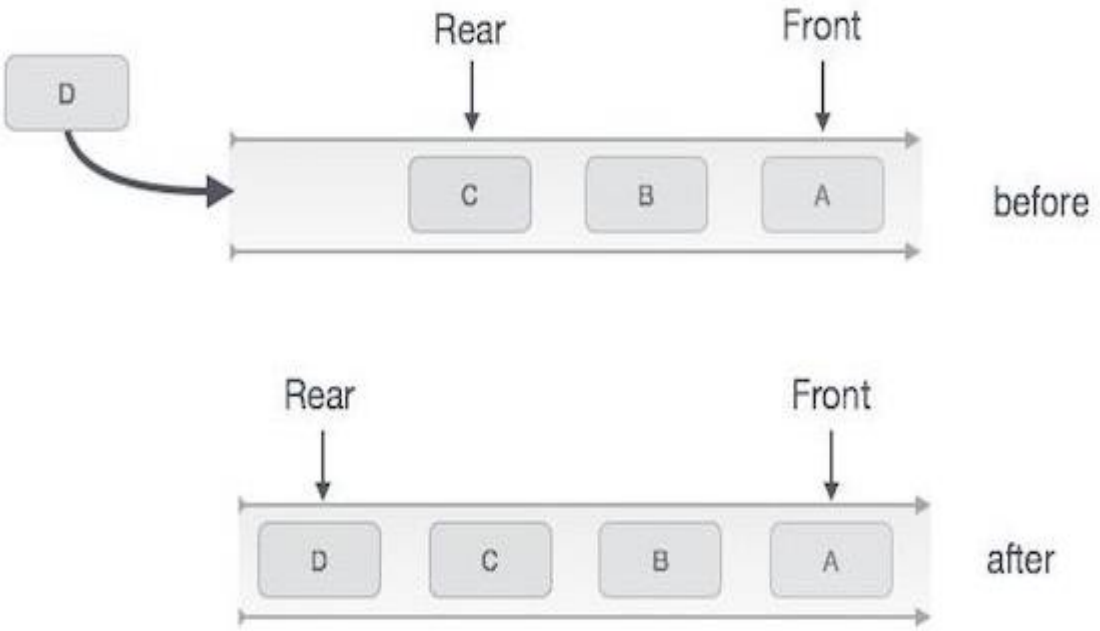
Step 1 − Check if the queue is full.

Step 2 − If the queue is full, produce overflow error and exit.

Step 3 − If the queue is not full, increment **rear** pointer to point the next empty space.

Step 4 − Add data element to the queue location, where the rear is pointing.

Step 5 − return success.




```
procedure enqueue(data)
   if queue is full
      return overflow
   endif

   rear ← rear + 1

   queue[rear] ← data

   return true

end procedure
```

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

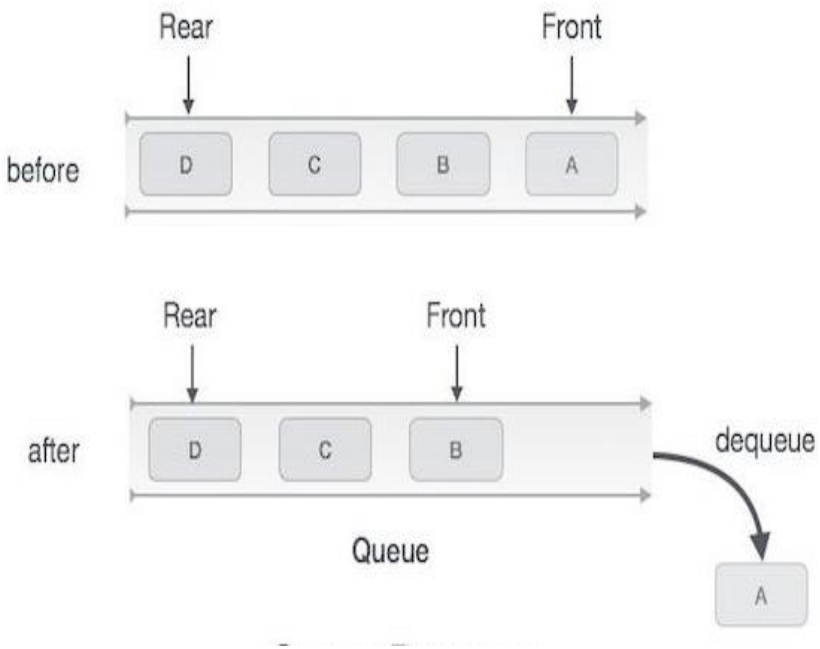
Step 1 − Check if the queue is empty.

Step 2 − If the queue is empty, produce underflow error and exit.

Step 3 − If the queue is not empty, access the data where **front** is pointing.

Step 4 − Increment **front** pointer to point to the next available data element.

Step 5 − Return success.



Queue Dequeue

Algorithm for dequeue operation

```
procedure dequeue
   if queue is empty
      return underflow
   end if

   data = queue[front]
   front ← front + 1

   return true
end procedure
```

- **Exercise**
- Write a method `stutter` that accepts a queue of integers as a parameter and replaces every element of the queue with two copies of that element.
  - `front [1, 2, 3] back`
    becomes
    `front [1, 1, 2, 2, 3, 3] back`

- Write a method `mirror` that accepts a queue of strings as a parameter and appends the queue's contents to itself in reverse order.
  - `front [a, b, c] back`
    becomes
    `front [a, b, c, c, b, a] back`