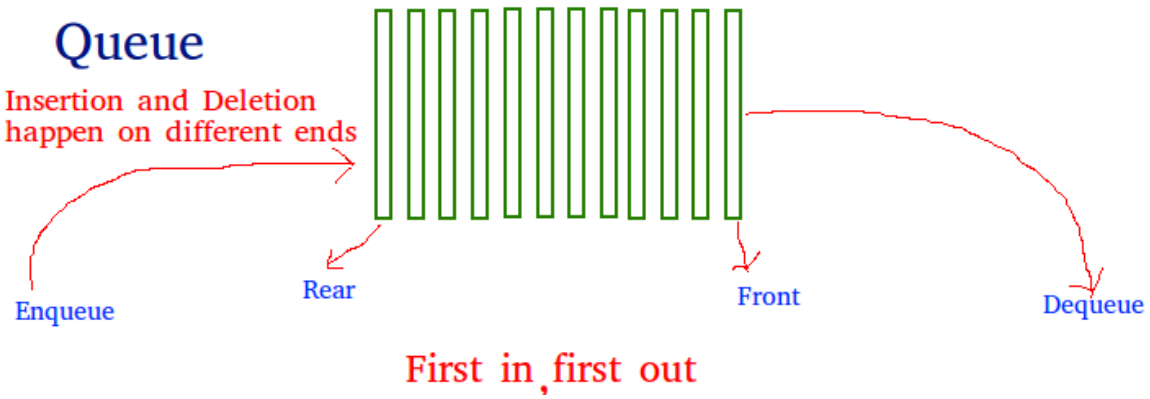
Queue :

- Queue is an abstract data structure
- It is linear and sequential
- It is open at both its ends.
- One end is always used to insert data (enqueue)…Rear end of queue
- Another end is used to remove data (dequeue)…. Front End of the queue
- Queue follows First-In-First-Out methodology, i.e., the data item stored first will be accessed first.

- A real-world example of 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 using Array :



Queue Operations

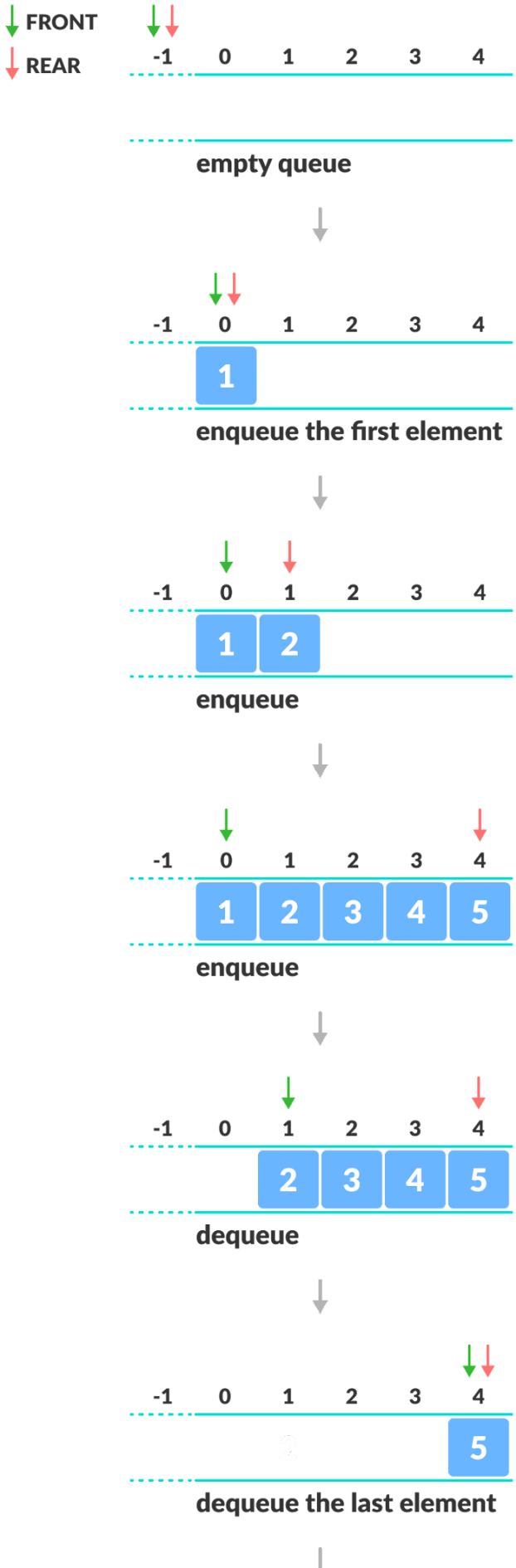
Queue operations may involve initializing or defining the queue, utilizing it, and then completely erasing it from the memory.

Basic operations

- **En(ter)queue()** − add (store) an item to the queue.
- **De(lete)queue()** − remove (access) an item from the queue.

Supportive Operations

- **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.


FRONT
REAR
-1 0 1 2 3 4
empty queue
-1 0 1 2 3 4
1
enqueue the first element
-1 0 1 2 3 4
1 2
enqueue
-1 0 1 2 3 4
1 2 3 4 5
enqueue
-1 0 1 2 3 4
2 3 4 5
dequeue
-1 0 1 2 3 4
5
dequeue the last element

peek()

- This Function helps to see the data at the **front** of the queue.

Algorithm

```
Peek()
Begin
1.  Return queue[front]
end
```

isfull()

- To check for the full queue, check if rear pointer reached at MAXSIZE

Algorithm

```
Isfull()
Begin
1. if rear = MAXSIZE
     return true
   else
     return false
End
```

Isempty()

```
begin
1.    if front < MIN  OR front > rear
//      Let Front = rear = -1
      return true
   else
     return false
   endif
end
```

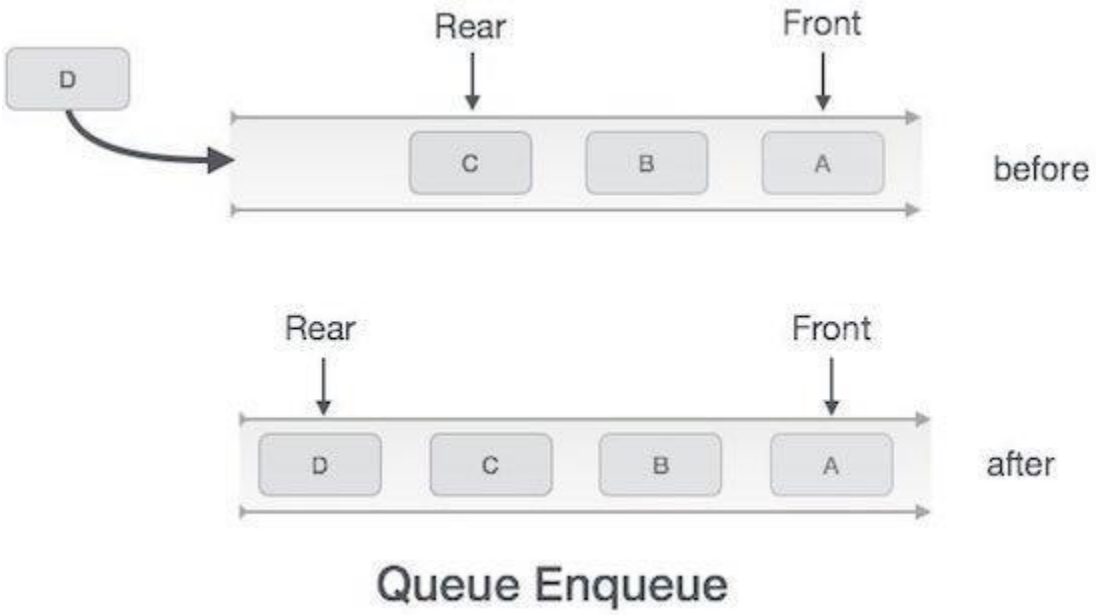
```
Begin
1. If (front = rear) = -1
      return true
   else
      return false
   endif
end
```

enqueue()

- check if the queue is full
- If Queue is full proceed for overflow and retuen 0
- for the first element, set value of FRONT to 0
- increase the REAR index by 1
- add the new element in the position pointed to by REAR

```
enqueue( data)
begin
 1.  if(isfull())
            then  return 0;
 2  rear = rear + 1;
 3  queue[rear] = data;
//3.a if front = -1 then front = 0;
 4  return 1;
 end
```

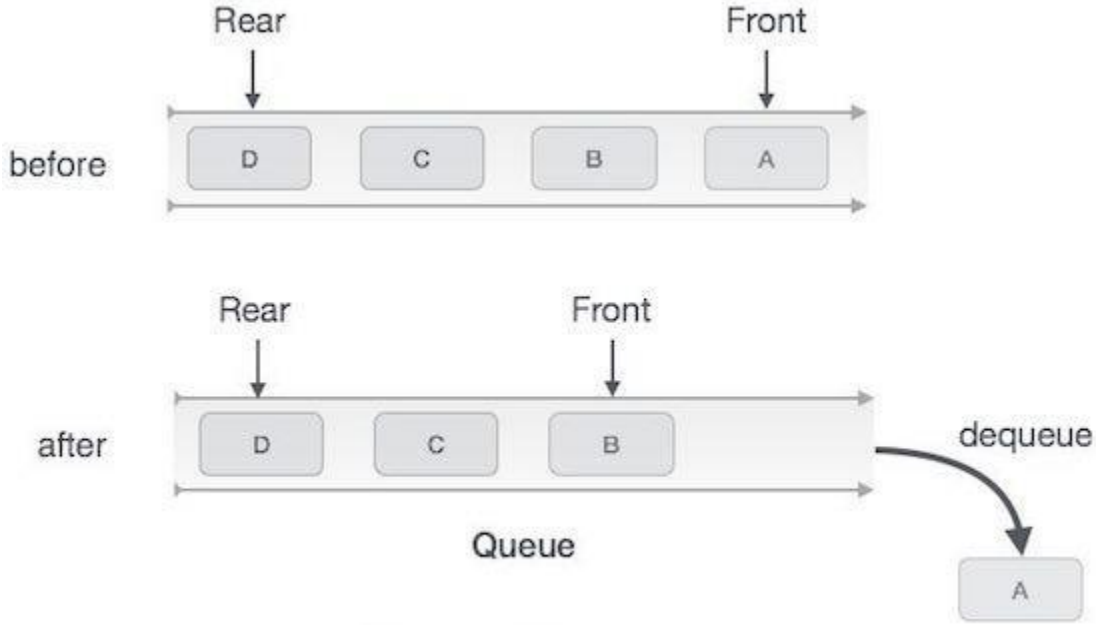
You may check Empty (front = rear = -1) then special case entry
Front = front +1
Rear = real+1
Queue[rear] = data



Queue Enqueue

Dequeue()

- check if the queue is empty
- If empty proceed for under flow and return false
- return the value pointed by FRONT
- increase the FRONT index by 1
- for the last element, reset the values of FRONT and REAR to -1

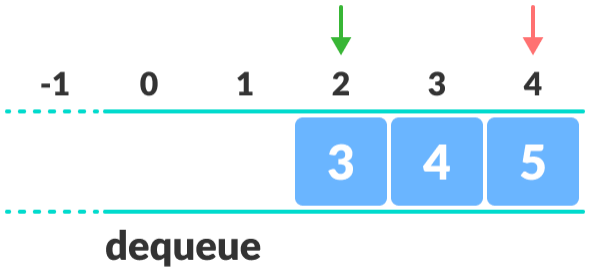


Queue Dequeue

```
procedure dequeue
Begin
1.  if (isempty()) //queue is empty
        return -1
    end if
2.  data = queue[front]
3.  front ← front + 1
4   return data
 end
```

Limitation of Queue

As you can see in the image below, after a bit of enqueuing and dequeuing, the size of the queue has been reduced.



Limitation of a queue

- The indexes 0 and 1 can only be used after the queue is reset when all the elements have been dequeued
- Defragment the queue
- After REAR reaches the last index, if we can store extra elements in the empty spaces (0 and 1), we can make use of the empty spaces. This is implemented by a modified queue called the circular queue.

Complexity Analysis

The complexity of enqueue and dequeue operations in a queue using an array is O(1).

Applications of Queue Data Structure

- CPU scheduling, Disk Scheduling
- When data is transferred asynchronously between two processes.The queue is used for synchronization. eg: IO Buffers, pipes, file IO, etc
- Handling of interrupts in real-time systems.
- Call Center phone systems use Queues to hold people calling them in an order