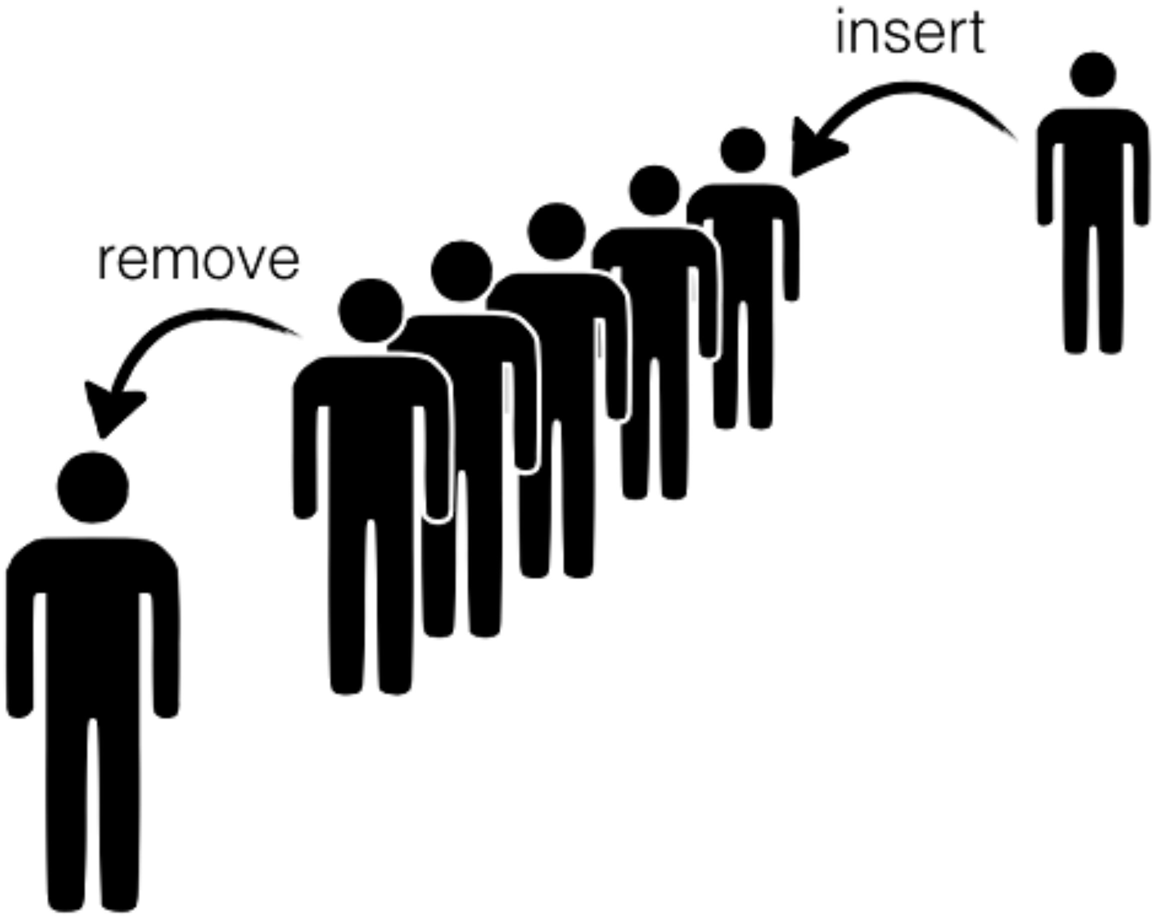
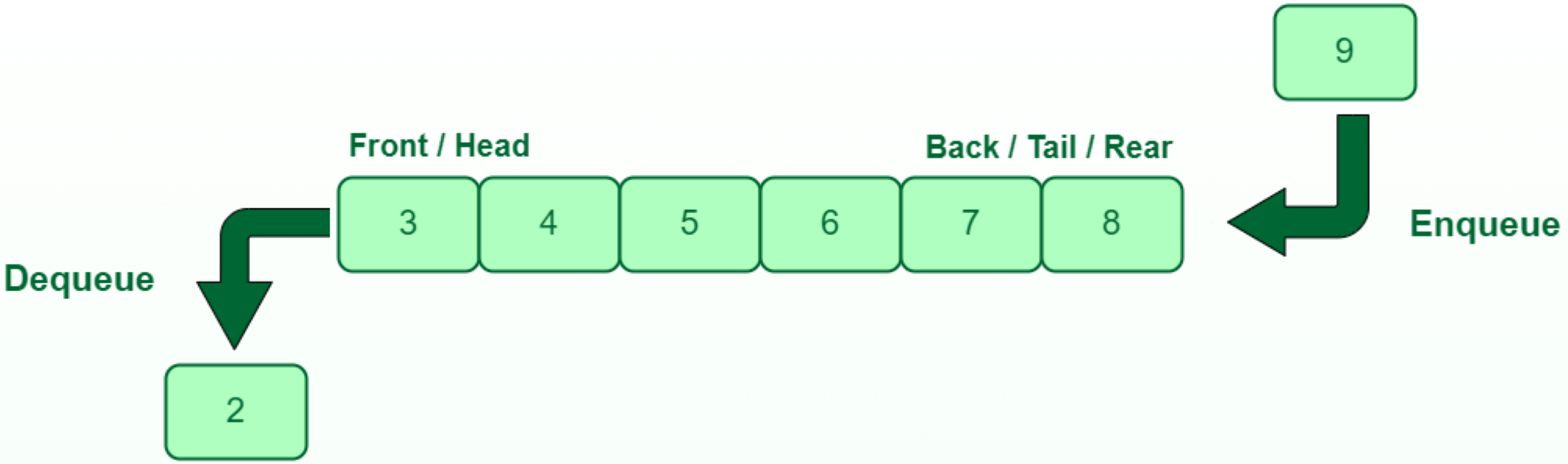
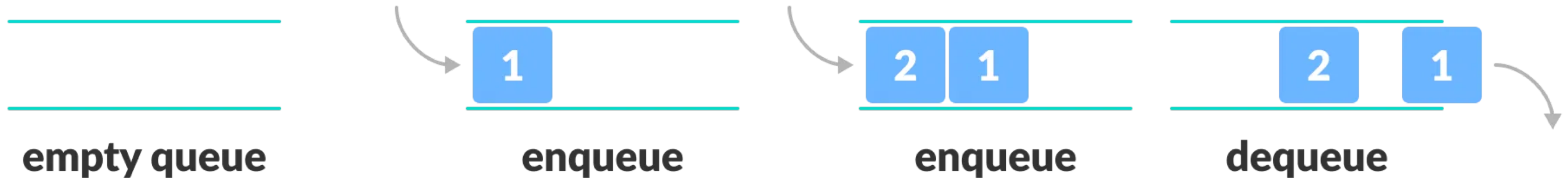
Data Structures and Algorithms
(ES221)

Queue (FIFO)

Dr. Zubair Ahmad


insert
remove


9
Front / Head
Back / Tail / Rear
3
4
5
6
7
8
Enqueue
Dequeue
2


1
2
1
2
1
empty queue
enqueue
enqueue
dequeue

Queues

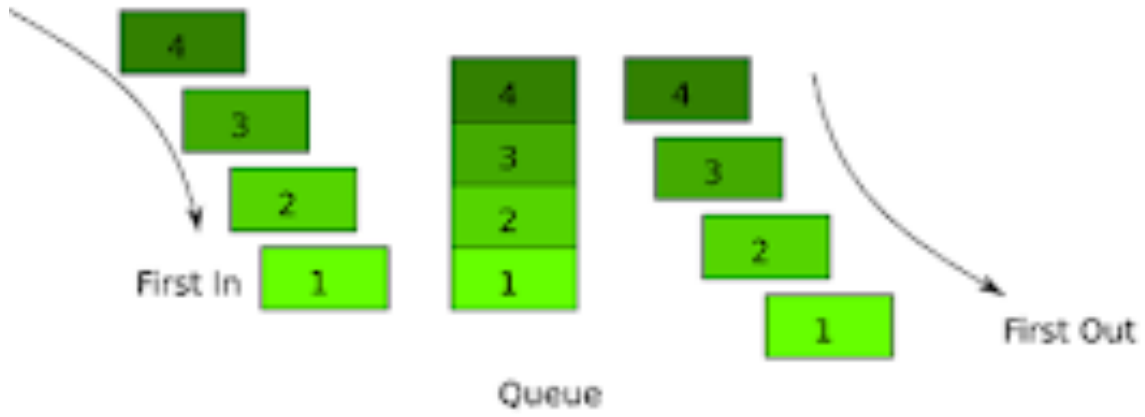
"A ***Queue*** is a special kind of list, where items are inserted at one end (***the rear***) And deleted at the other end (***the front***)"

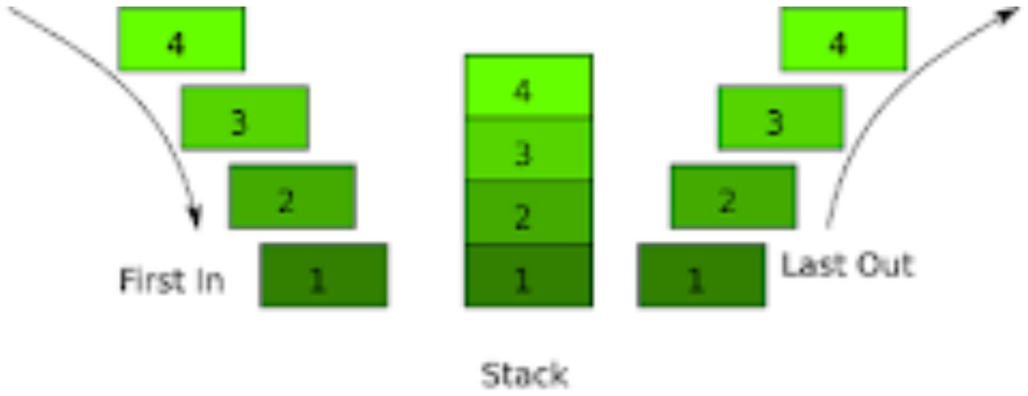
Other Name:

- First In First Out (FIFO)

Difference from Stack:

Insertion go at the end of the list, rather than the beginning of the list.





Common Operations on Queues

1. **MAKENULL(Q):** Makes Queue Q be an empty list.
2. **FRONT*(Q)*:** Returns the first element on Queue Q.
3. **ENQUEUE(*x*,*Q*):** Inserts element x at the end of Queue Q.
4. **DEQUEUE(*Q*):** Deletes the first element of *Q.*
5. **EMPTY(*Q*):** Returns true if and only if Q is an empty queue.

Example??

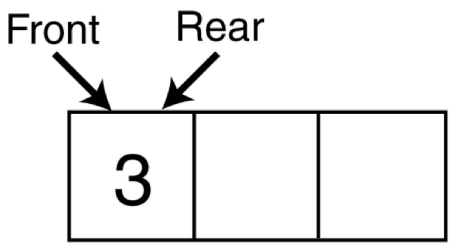
Applications of Queues

- Operating system
  - multi-user/multitasking environments, where several users or task may be requesting the same resource simultaneously.
- Communication Software
  - queues to hold *information* received over networks and dial up connections. (Information can be transmitted faster than it can be processed, so is placed in a queue waiting to be processed)
- Some other?

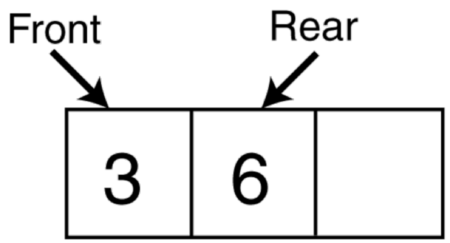
Implementation

- Static
  - Queue is implemented by an array, and size of queue remains fix
- Dynamic
  - A **queue** can be **implemented** as a **linked list**, and *expand* or *shrink* with each *enqueue* or *dequeue* operation.

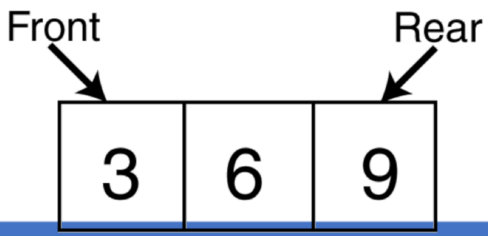
Enqueue(3);



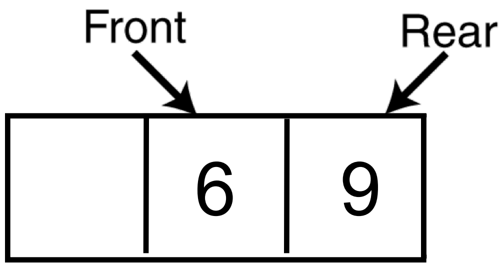
Enqueue(6);



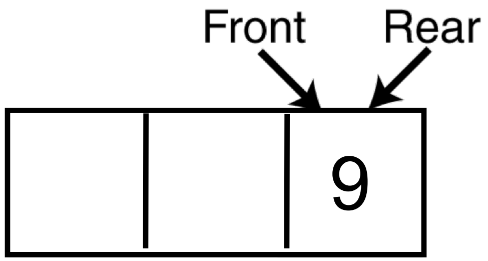
Enqueue(9);



Dequeue();

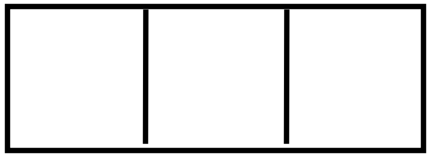


Dequeue();



Dequeue();

Front = -1 Rear = -1



ENQUEUE(*NEwdata,Queue*)

```
1. if ( (front = 0 and rear = N-1) or (Front = Rear + 1) )
     (Queue already full ? )
     • Print Overflow and return
2. Find the new value of Rear
   if front = -1  ( i.e., queue is initially empty)
     • Front = 0 rear = 0
   else If rear = N -1( rear is at the end)
     • Rear = 0        (Move around the rear to the start of the queue)
   else
     • rear = rear +1
3. QUEUE[rear] = NEWDATA (Add the new data at the end)
4. return
```

Dequeue(*Queue*)

Case 1: Initially Empty Queue

Front = -1, Rear = -1
Inserting 10:
Front = 0, Rear = 0
Queue: [10, _, _, _, _]

Case 2: Normal Insertions

Inserting 20:
Rear = 1
Queue: [10, 20, _, _, _]

Inserting 30:
Rear = 2
Queue: [10, 20, 30, _, _]

Inserting 40:
Rear = 3
Queue: [10, 20, 30, 40, _]

Inserting 50:
Rear = 4
Queue: [10, 20, 30, 40, 50]

Case 3: Circular Wrap Around
If we **delete** two elements (10, 20), front moves forward:

Front = 2, Rear = 4
Queue: [_, _, 30, 40, 50]

Now, inserting 60:

Rear wraps around to 0 (Rear = 0)
Queue: [60, _, 30, 40, 50]

Dequeue(*Queue*)

```
1. if front = -1  (queue already empty?)
     • Print underflow and return
2. DATATOREMOVE= QUEUE[front]
3. (Find the new value of front)
 if front = rear (QUEUE has only one element)
     • front = -1
     • Rear = -1
else if front = N-1
     • front = 0
 else
     • front = front + 1
```

Dequeue(*Queue*)

Lets assume we have a **circular queue** of size N = 5, represented as:

Index:	0	1	2	3	4
Queue:	[10]	[20]	[30]	[40]	[50]

Initially:

front = 0 (pointing to 10)
rear = 4 (pointing to 50)

Dequeue(*Queue*)

Dequeue the first element

- Check if the queue is empty → **No (front != -1)**

- .DATATOREMOVE = QUEUE[front] = 10

- **Update front:**

front != rear (so not a single element case)
front != N-1, so move front forward.
front = front + 1 = 1

LINKED LIST IMPLEMENTATION OF QUEUE

```
struct queue {
        int data;
        queue *next;
        } *head, *tail;
```

ENQUEUE(*NEwdata*, *Queue*)

```
1.  Find the new value of tail
    q = new queue;
    if head == NULL  ( i.e., queue is initially empty)
        • head = q; tail = q;
    else
        • tail->next = q; tail = q;
2. tail->data = NEWDATA (Add the new data at the end)
3. return
```

Dequeue(*Queue*)

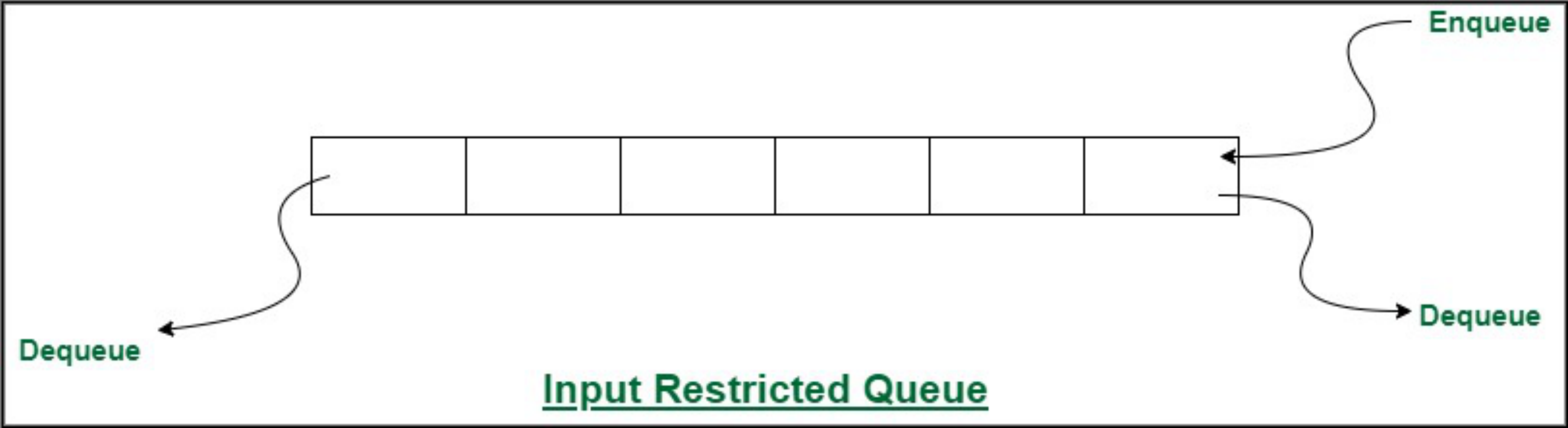
```
1. if head==NULL  (queue already empty?)
    • Print underflow and return
2. DATATOREMOVE= head->data
3. (Find the new value of tail)
 if head = tail (QUEUE has only one element)
    • head = NULL
    • tail = NULL
   else
    • head=head->next
```

Input restricted 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 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

Input restricted Queue

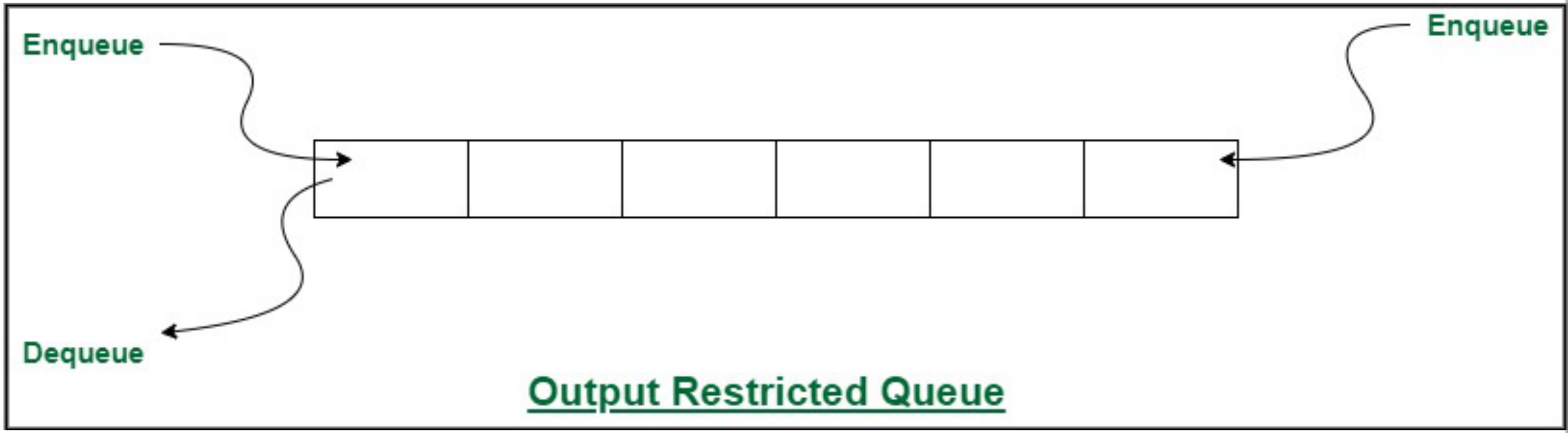


Output restricted Queue

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.

Output restricted Queue

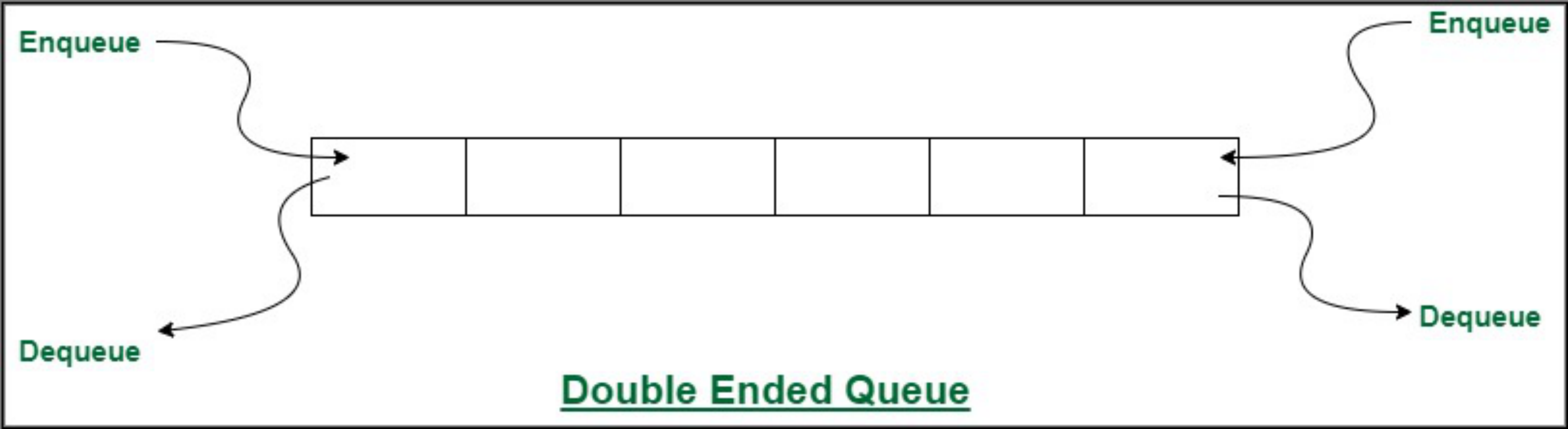


Double Ended Queue

The insertion and deletion operations are performed at both the ends (front and rear)

Since Deque supports both stack and queue operations, it can be used as both

Double Ended Queue



Questions?

zahmaad.github.io