Queues

Queue

- A Queue is a linear structure which follows a particular order in which the operations are performed. The order is First In First Out (FIFO).
- A good example of a queue is any queue of consumers for a resource where the consumer that came first is served first.
- The difference between stacks and queues is in removing. In a stack we remove the item the most recently added; in a queue, we remove the item the least recently added.

Queue

- **Queue** is also an abstract data type or a linear data structure, just like stack data structure, in which the first element is inserted from one end called the **REAR**(also called **tail**), and the removal of existing element takes place from the other end called as **FRONT**(also called **head**).

- This makes queue as **FIFO**(First in First Out) data structure, which means that element inserted first will be removed first.

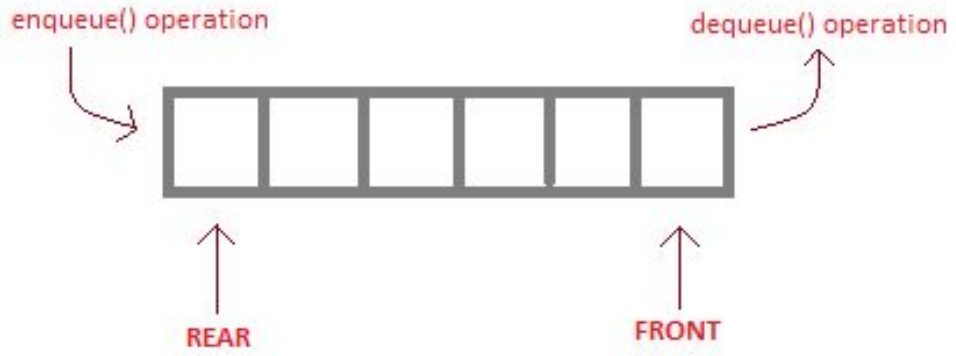
- Which is exactly how queue system works in real world. If you go to a ticket counter to buy movie tickets, and are first in the queue, then you will be the first one to get the tickets. Right? Same is the case with Queue data structure. Data inserted first, will leave the queue first.

Basic Operations of Queue

- A queue is an object or more specifically an abstract data structure(ADT) that allows the following operations:
- **Enqueue**: Add an element to the end of the queue
- **Dequeue**: Remove an element from the front of the queue
- **IsEmpty**: Check if the queue is empty
- **IsFull**: Check if the queue is full
- **Peek**: Get the value of the front of the queue without removing it

Queue

- The process to add an element into queue is called **Enqueue** and the process of removal of an element from queue is called **Dequeue**.



enqueue( ) is the operation for adding an element into Queue.

dequeue( ) is the operation for removing an element from Queue .

QUEUE DATA STRUCTURE

Basic features of Queue

1. Like stack, queue is also an ordered list of elements of similar data types.
2. Queue is a FIFO(First in First Out) structure.
3. Once a new element is inserted into the Queue, all the elements inserted before the new element in the queue must be removed, to remove the new element.
4. Peek() function is oftenly used to return the value of first element without dequeuing it.

Applications of Queue

- Queue, as the name suggests is used whenever we need to manage any group of objects in an order in which the first one coming in, also gets out first while the others wait for their turn, like in the following scenarios:

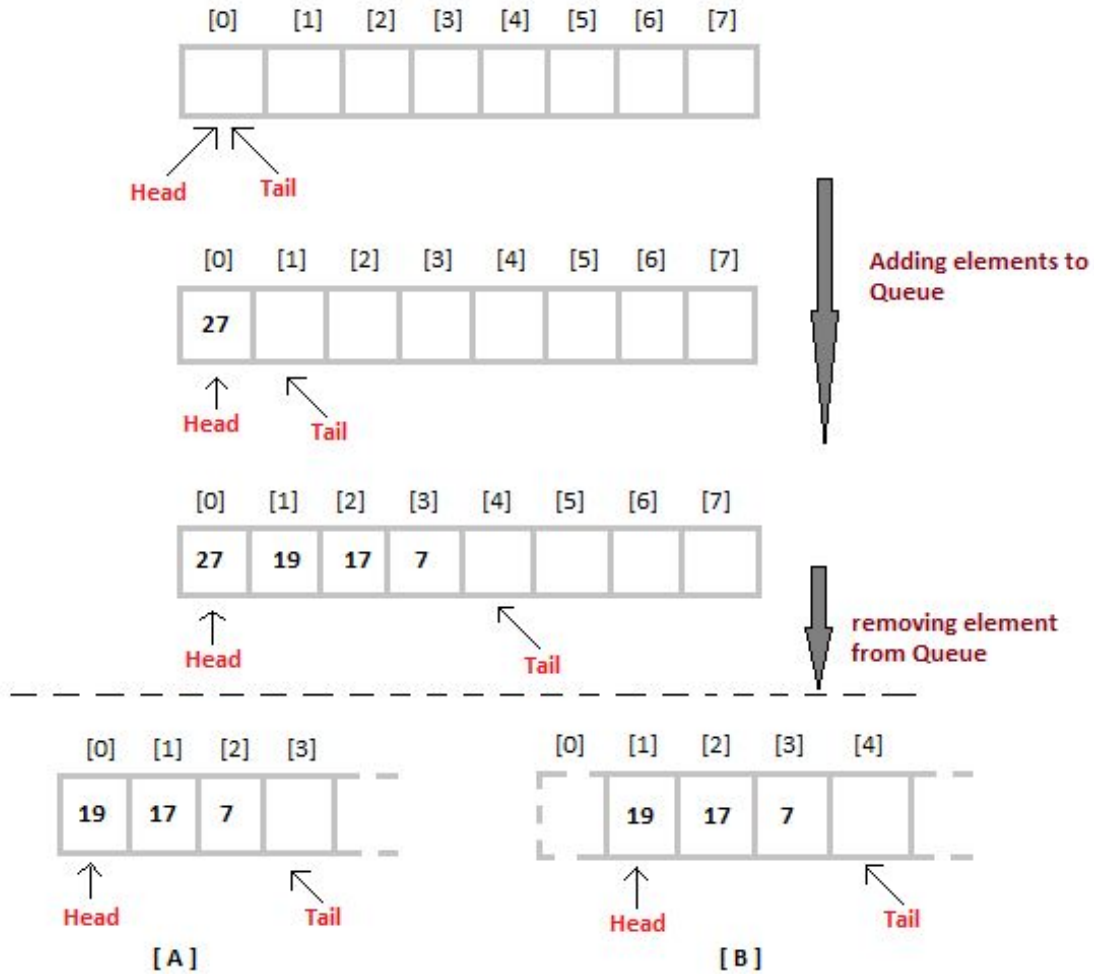
1. Serving requests on a single shared resource, like a printer, CPU task scheduling etc.
2. In real life scenario, Call Center phone systems uses Queues to hold people calling them in an order, until a service representative is free.
3. Handling of interrupts in real-time systems. The interrupts are handled in the same order as they arrive i.e First come first served.

Implementation of Queue Data Structure

- Queue can be implemented using an Array, Stack or Linked List. The easiest way of implementing a queue is by using an Array.

- Initially the **head**(FRONT) and the **tail**(REAR) of the queue points at the first index of the array (starting the index of array from 0).

- As we add elements to the queue, the **tail** keeps on moving ahead, always pointing to the position where the next element will be inserted, while the **head** remains at the first index.


[0] [1] [2] [3] [4] [5] [6] [7]
Head
Tail
Adding elements to Queue
27
19
17
7
removing element from Queue
[A]
[B]

- When we remove an element from Queue, we can follow two possible approaches (mentioned [A] and [B] in above diagram). In [A] approach, we remove the element at **head** position, and then one by one shift all the other elements in forward position.
- In approach [B] we remove the element from **head** position and then move **head** to the next position.
- In approach [A] there is an **overhead of shifting the elements one position forward** every time we remove the first element.
- In approach [B] there is no such overhead, but whenever we move head one position ahead, after removal of first element, the **size on Queue is reduced by one space** each time.

Algorithm for ENQUEUE operation

1. Check if the queue is full or not.
2. If the queue is full, then print overflow error and exit the program.
3. If the queue is not full, then increment the tail and add the element.

Algorithm for DEQUEUE operation

1. Check if the queue is empty or not.
2. If the queue is empty, then print underflow error and exit the program.
3. If the queue is not empty, then print the element at the head and increment the head.

Complexity Analysis of Queue Operations

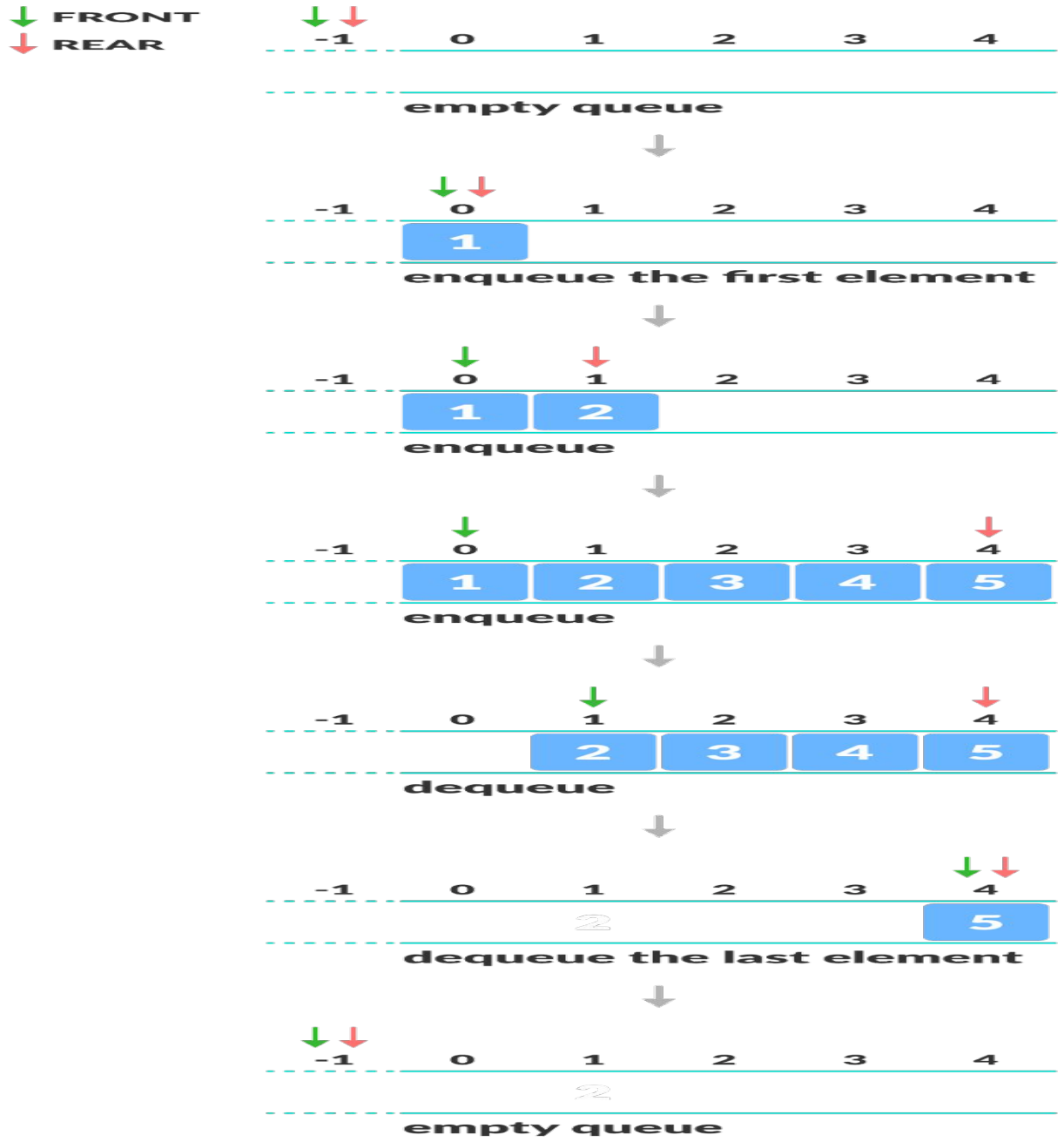
- Just like Stack, in case of a Queue too, we know exactly, on which position new element will be added and from where an element will be removed, hence both these operations requires a single step.
- Enqueue: **O(1)**
- Dequeue: **O(1)**
- Size: **O(1)**

Working of Queue

- Queue operations work as follows:
- two pointers FRONT and REAR
- FRONT track the first element of the queue
- REAR track the last elements of the queue
- initially, set value of FRONT and REAR to -1

- **Enqueue Operation**
- check if the queue is full
- 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

- **Dequeue Operation**
- check if the queue is empty
- 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

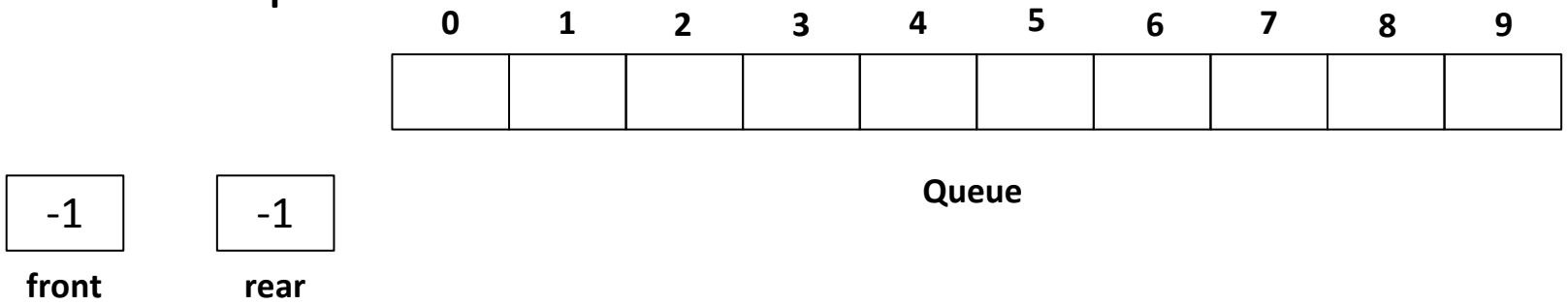

FRONT
REAR
-1 0 1 2 3 4
empty queue
-1 0 1 2 3 4
1
enqueue the first element
-1 0 1 2 3 4
1 2
enqueue
-1 0 1 2 3 4
1 2 3 4 5
enqueue
-1 0 1 2 3 4
2 3 4 5
dequeue
-1 0 1 2 3 4
2
5
dequeue the last element
-1 0 1 2 3 4
2
empty queue

Representation of Queues in Memory

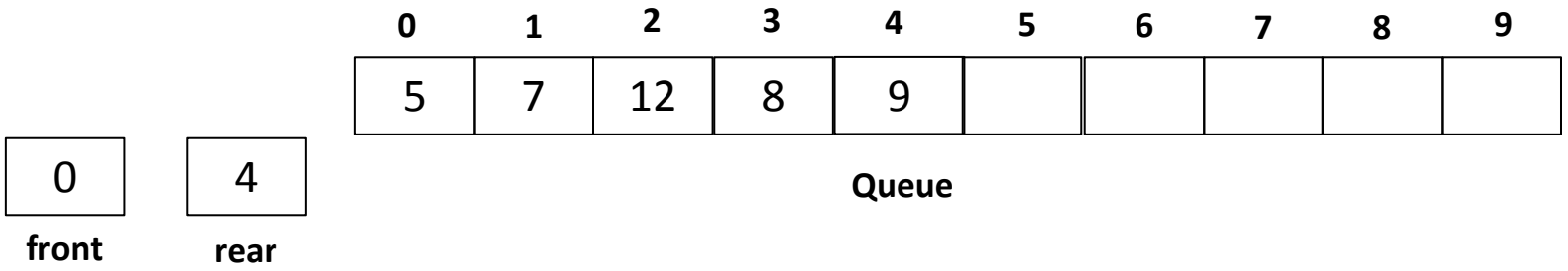
- A Queue can be represented in memory using a linear array or a linked list.

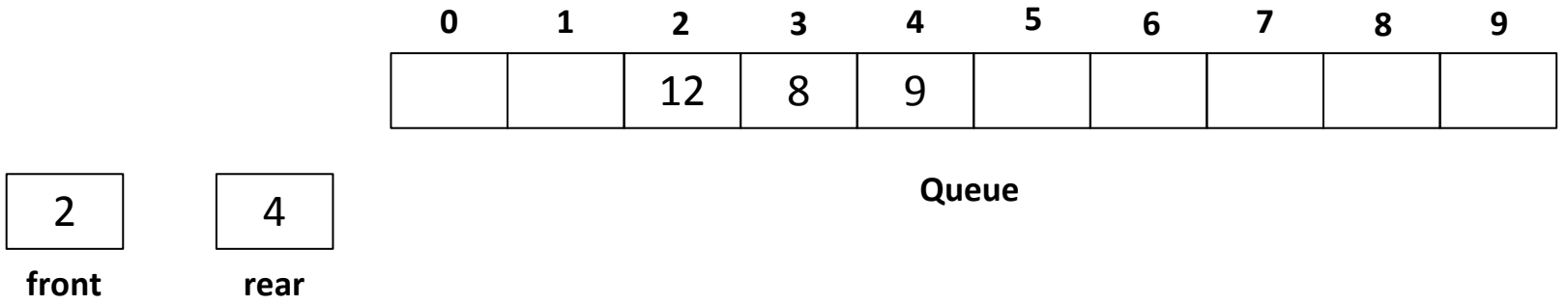
Representation using Array

- A Queue is represented by a linear array, say *queue*, and two integer variables, *front* and *rear*. Here variable *front* holds the index of the first element of the queue and variable *rear* holds the index of the last element of the queue.

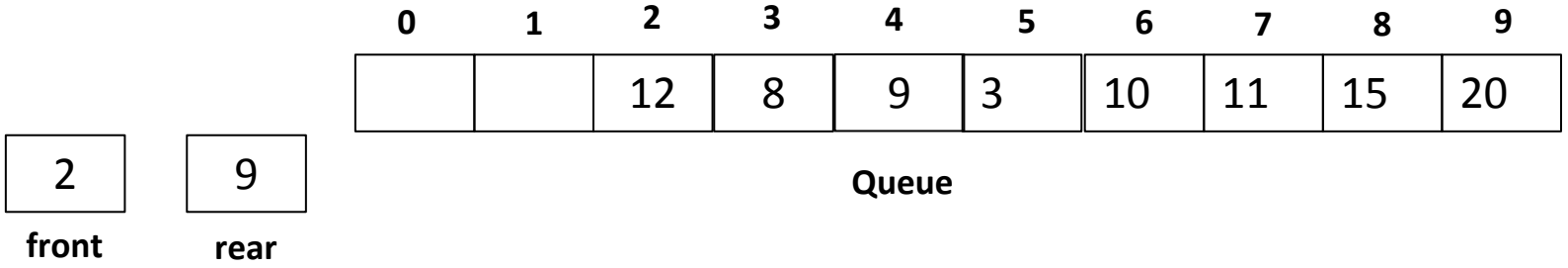


Representation of Queues in Memory

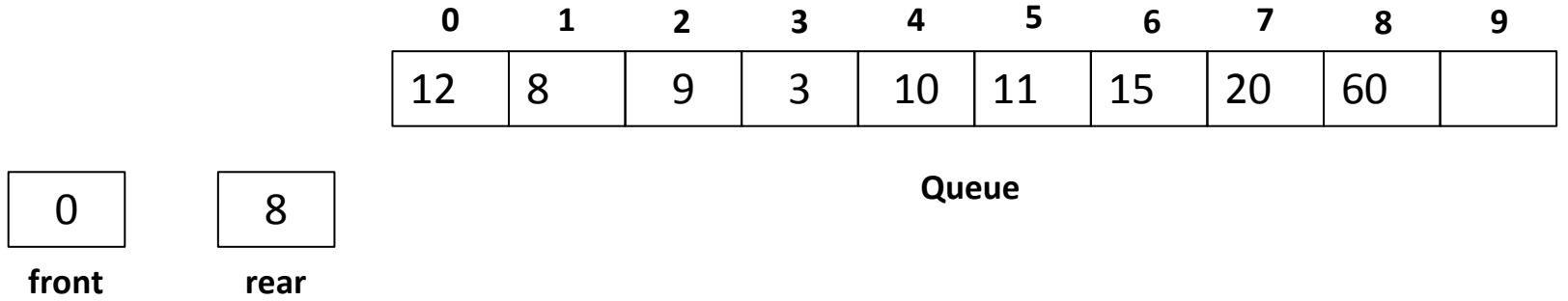




Representation of Queues in Memory



It is not possible to enqueue more elements as such, though two positions in the linear queue are vacant. To overcome this problem, the elements of the queue are moved forward, so that the vacant positions are shifted towards the rear end of the linear queue. After shifting, front and rear are adjusted properly, and then the element is enqueued in the linear queue as usual.



However, this difficulty can be overcome if we treat the queue position with index 0 as a position that comes after position with index 9, i.e., we treat the queue as circular.

Creating an Empty Queue

- Before we can use a queue, it is to be initialized/created. As the index of array elements can take value in the range 0 to (size -1), the purpose of initializing the queue is served by assigning value -1 to the ***front*** and ***rear*** variables.

- **Algorithm for creating stack**

  CreateQueue(Q)

  This sub algorithm assigns value -1 to variable front and rear to put the linear queue Q in the initial state, i.e., empty.

  1. Set front = rear = -1
  2. Return

Testing Queue for Underflow

- Before we remove an item from a queue, it is necessary to test whether queue still have some elements, i.e., to test that whether the queue is empty or not.
- If it is not empty then the dequeue operation can be performed to remove the front element.
- This test is performed by comparing the value of front with sentinel value -1

Algorithm for testing Underflow

IsEmpty (Q, status)

This sub algorithm compares the value of ***front*** with the sentinel value (-1). If the value of the ***front*** is -1 it sets the Boolean variable ***status*** to true(underflow) else value false.

1. If (front = -1) then
   Set status = true
   Else
   Set status = false
   EndIf
2. Return

Testing Queue for Overflow

- Before we insert an item into a Queue, it is necessary to test whether queue still have some space to accommodate the new elements, i.e., to test that whether the queue is full or not. If it is not full then the enqueue operation can be performed to insert the element at the rear end of the queue. This test is performed by test condition:

(front =0) and (rear = size -1)

- i.e., front element is a position with index 0 and the last element is at position with index size-1, which indicates that no position in linear queue is vacant.

Algorithm to check Overflow

IsFull (Q, status)

This sub algorithm compares the value of front with zero and value of rear with (size-1). If the value of the front is zero and rear is (size-1) it sets the variable ***status*** to true(overflow) else value false.

1. If (front = 0 and rear = (size-1)) then
   Set status = true
   Else
   Set status = false
   EndIf
2. Return

Enqueue Operation on Linear Queue

- There are two conditions, which can occur, even if queue is not full. These are:
- If a linear queue is empty, then the value of front and rear variables will be -1, then both front and rear are set to 0.
  - If a linear queue is not empty, then there are further two possibilities:
  - If the value of the rear variable is less than size-1, then the rear variable is incremented.
- If the value of the rear variable is equal to size-1, then the elements of the linear queue are moved forward, if front is not equal to 0, and the front and rear variables are adjusted accordingly.

Enqueue (Q, item)
This sub algorithm calls sub algorithm IsFull to see whether the Q if full, and if the answer is yes, it flags "overflow" condition and returns. However, if the Q is not full, it enqueues (add/insert) new element item at the rear of linear queue Q

- 1. Call IsFull (Q, status)
- 2. If (status = true) then
  - Write : "Overflow"
  - Return
- EndIf
- 3. If (front =-1) then
    - Set front = rear = 0
- Else If (rear = size-1)
  - For i = front to rear in steps of +1
    - Set Q[i-front] = Q[i]
    - EndFor
    - Set rear = rear –front + 1
    - Set front = 0
    - Else
    - Set rear = rear + 1
    - EndIf
- 4. Set Q[rear] = item
- 5. Return

Dequeue Operation on Linear Queue

- The front element of a linear queue is assigned to a local variable, which later on will be returned via the return statement. After assigning the front element of a linear queue to a local variable, the value of front variable is modified so that it points to the new front. There are two possibilities:
- If there was only one element in a linear queue, then after dequeue operation queue will become empty. This state of a linear queue is reflected by setting front and rear variables to -1.
- Otherwise the value of the front variable is incremented.

Dequeue (Q, item)
This sub algorithm calls sub algorithm IsEmpty to see whether the Q if empty, and if the answer is yes, it flags "underflow" condition and returns. However, if the Q is not empty, it dequeues (remove/delete) element from front of linear queue Q and assigns to local variable item

- 1. Call IsEmpty (Q, status)
- 2. If (status = true) then
  - Write : "Underflow"
  - Return
- EndIf
- 3. Set item = Q[front]
- 4. If (front = rear) then
- Set front = rear = -1
- Else
- Set front = front + 1
- EndIf
- 4. Return

Assessing the Front Element

- **Peep (Q, item)**
- This sub algorithm calls sub algorithm IsEmpty to see whether the Q is empty, and if yes, it flags “underflow” condition and returns. However, if the Q is not empty, it assigns the front element of the linear queue Q to local variable item
- 1. Call IsEmpty (Q, status)
- 2. If (status = true) then
- Write : “underflow”
- Return
- Endif
- 3. Set item = Q[front]
- 4. Return

Limitations of Linear Queue

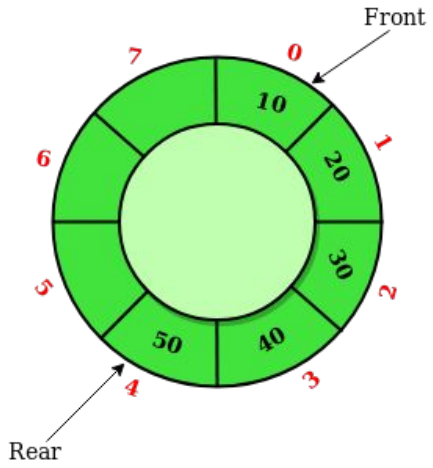
- The only limitation of linear queue is that:

- *If the last position of the queue is occupied, it is not possible to enqueue any more elements even though some positions are vacant towards the front positions of the queue*

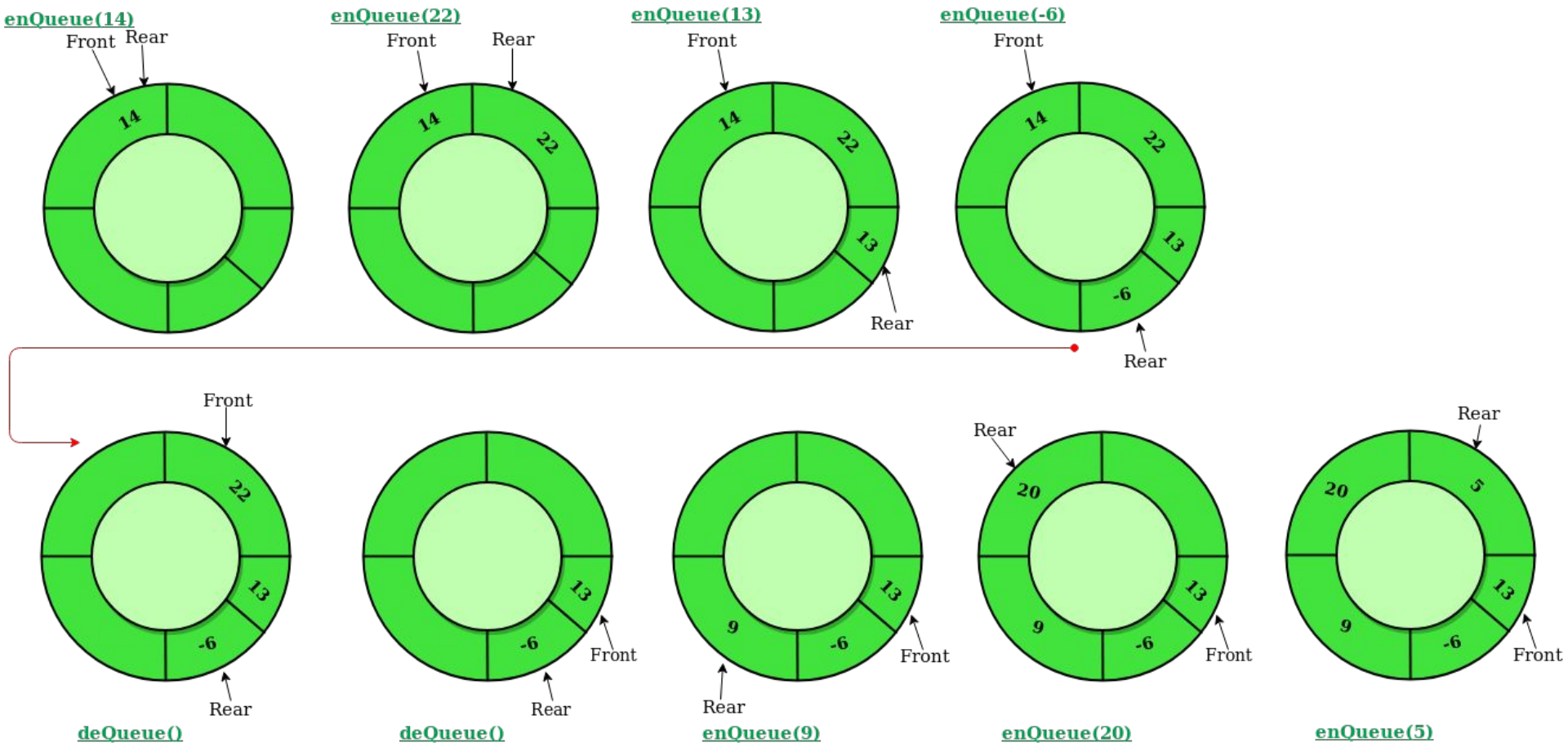
- However, this limitation can be overcome by moving the elements forward and finally adjusting the front and rear positions appropriately.
- But this operation can be time consuming if the queue is very long. This limitation can be overcome if we treat that the queue position with index 0 comes immediately after the last queue position with index (size-1). The resulting queue is known as circular queue.

Circular Queue

- Circular Queue is a linear data structure in which the operations are performed based on FIFO (First In First Out) principle and the last position is connected back to the first position to make a circle. It is also called **'Ring Buffer'**.

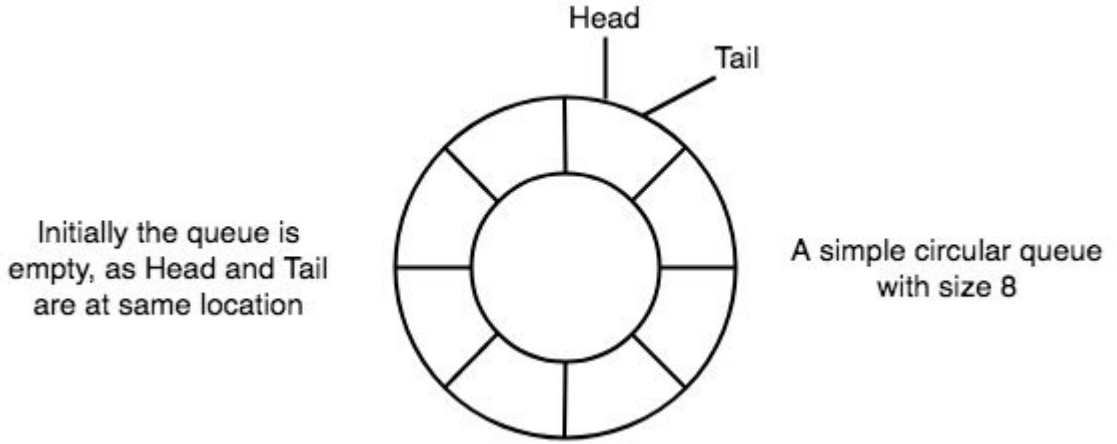


In a normal Queue, we can insert elements until queue becomes full. But once queue becomes full, we can not insert the next element even if there is a space in front of queue.

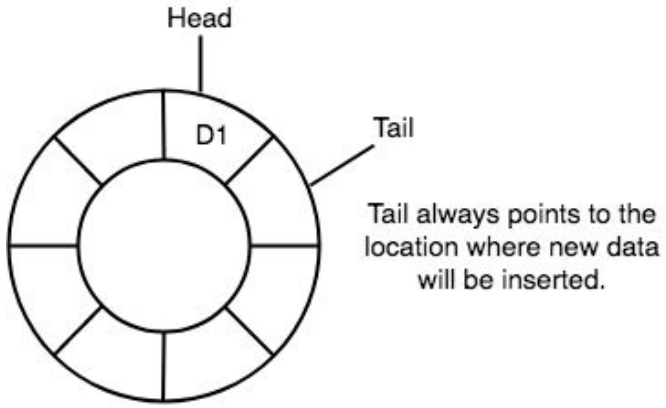


Basic features of Circular Queue

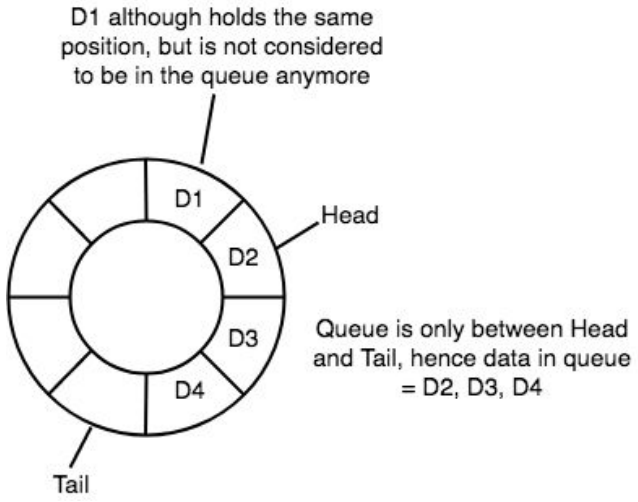
- In case of a circular queue, head pointer will always point to the front of the queue, and tail pointer will always point to the end of the queue.
- Initially, the head and the tail pointers will be pointing to the same location, this would mean that the queue is empty



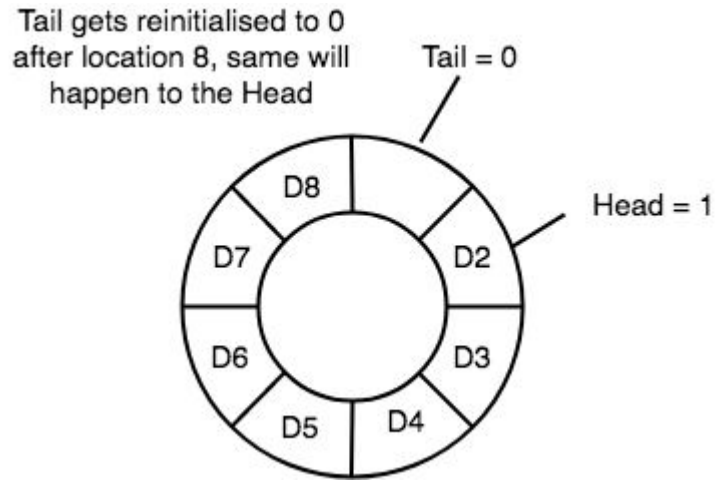
- New data is always added to the location pointed by the tail pointer, and once the data is added, tail pointer is incremented to point to the next available location.



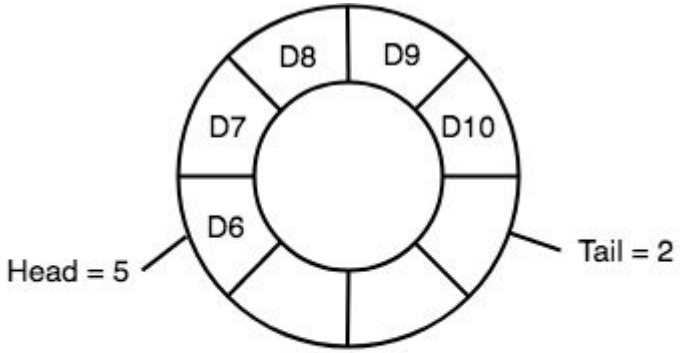
- In a circular queue, data is not actually removed from the queue. Only the head pointer is incremented by one position when **dequeue** is executed. As the queue data is only the data between head and tail, hence the data left outside is not a part of the queue anymore, hence removed.



- The head and the tail pointer will get reinitialized to **0** every time they reach the end of the queue



- Also, the head and the tail pointers can cross each other. In other words, head pointer can be greater than the tail. Sounds odd? This will happen when we dequeue the queue a couple of times and the tail pointer gets reinitialized upon reaching the end of the queue.



In such a situation the value of the Head pointer will be greater than the Tail pointer

Operations on Circular Queue

- The operations of initialization, testing for underflow for circular queue is similar to that of a linear queue.
- However, other operations are slightly different.

Testing a circular Queue for overflow

- Before inserting new element in a circular queue, it is necessary to test whether a circular queue still have some space to accommodate the incoming element i.e., to test that whether a circular queue is full or not. If it is not full then the enqueue operation can be performed to insert the element at rear of a circular queue. This test is performed by test condition:

(front = 0) and (rear = size-1) OR front = rear+1

- If any of these condition is satisfied, it means circular queue is full

- **IsFull (Q, status)**
- This sub algorithm sets Boolean variable status to Boolean value true if circular queue Q is empty else to Boolean value false

1. If (((front = 0) and (rear = size-1)) OR (front = rear+1)) then
   Set status = true
   Else
   Set status = false
2. Return

Enqueue Operation on Circular Queue

Before the enqueue operation, there are three conditions, which can occur, even if a circular queue is not full. These are:

- If the queue is empty, then the value of the front and rear will be -1, then both front and rear are set to 0.
- If the queue is not empty then the value of the rear will be the index of the last element of the queue, then the rear variable is incremented.
- If the queue is not full and the value of rear variable is equal to size-1, then rear variable is set to 0.

Enqueue (Q, item)
This sub algorithm calls sub algorithm IsFull to see whether the Q if full, and if the answer is yes, it flags "overflow" condition and returns. However, if the Q is not full, it enqueues (add/insert) new element item at the rear of circular queue Q

- 1. Call IsFull (Q, status)
- 2. If (status = true) then
  Write : "Overflow"
  Return
  EndIf
- 3. If (front =-1) then
  Set front = rear = 0
  Else If (rear = size-1) then
  Set rear = 0
  Else
  Set rear = rear+1
  EndIf
- 4. Set Q[rear] = item
- 5. Return

Dequeue Operation on a circular queue

- There are two possibilities:
    - If there was only one element in a circular queue, then after dequeue operation the circular queue will become empty. This state of a circular queue is reflected by setting front and rear variables to -1.
    - If value of the front variable is equal to size-1, then set front variable to 0.
- If none of the above conditions hold, then the front variable is incremented

Dequeue (Q, item)
This sub algorithm calls sub algorithm IsEmpty to see whether the Q if empty, and if the answer is yes, it flags "underflow" condition and returns. However, if the Q is not empty, it dequeue (remove/delete) element from front of circular queue Q and assigns to local variable item

- 1. Call IsEmpty (Q, status)
- 2. If (status = true) then
  - Write : "Underflow"
  - Return
- EndIf
- 3. Set item = Q[front]
- 4. If (front = rear) then
- Set front = rear = -1
- Else if (front = size-1) then
- Set front = 0
- Else
- Set front = front + 1
- 5. Return

Special Types of Queues

Deque

- A deque, pronounced as deck, is a kind of linear queue in which elements can be added/removed at/from either end but not in the middle.

- There are two variations of deque:
- Input restricted deque: which allows insertions at one ened but allows deletions at both the ends
- Output restricted deque: which allows insertions at both ends but allows deletions only at one end.

Special Types of Queue

Priority Queue

- A priority queue is a kind of queue in which each element is assigned a priority and the order in which elements are deleted and processed comes from the following rules:
  - An element with highest priority is processed first before any element of lower priority.
  - Two or more elements with the same priority are processed according to the order in which they were added to the queue.

There can be different criteria's for determining the priority. Some are:

- A shortest job is given priority over the longer one
- An important job is given higher priority over a routine type job.