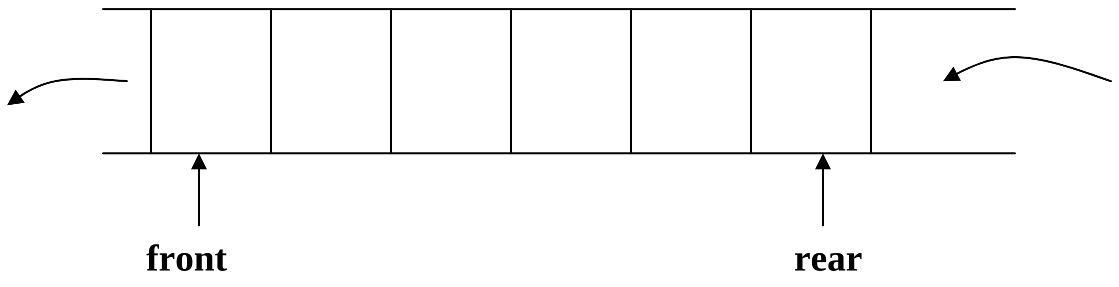
QUEUES

A queue is simply a waiting line that grows by adding elements to its end and shrinks by removing elements from the front. Compared to stack, it reflects the more commonly used maxim in real-world, namely, “first come, first served”. Waiting lines in supermarkets, banks, food counters are common examples of queues.

A formal definition of queue as a data structure: It is a list from which items may be deleted at one end (front) and into which items may be inserted at the other end (rear). It is also referred to as a first-in-first-out (FIFO) data structure.



Queues have many applications in computer systems:

- Handling jobs in a single processor computer
- print spooling
- transmitting information packets in computer networks.

- **Primitive Queue operations**

  `enqueue (q, x):`

  inserts item x at the rear of the queue q

  `x = dequeue (q):`
  removes the front element from q and returns its value.

  `isEmpty(q)` : Check to see if the queue is empty.

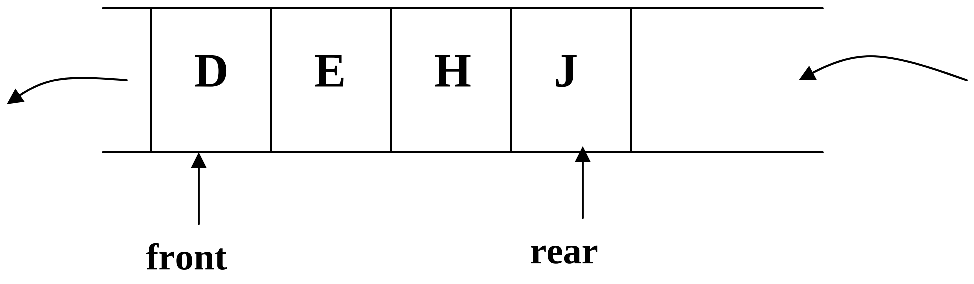
  `isFull(q)` : checks to see if there is space to insert more items in the queue.

Example:

Consider the following sequence of operations being performed on a queue “q” which stores single characters:

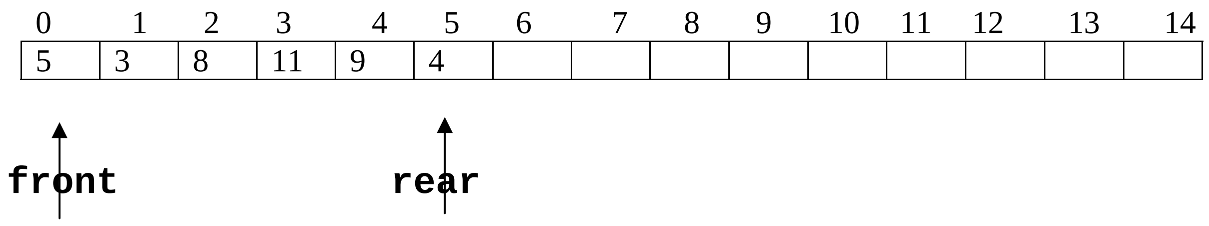
```
enqueue(q, ‘A’);
enqueue(q, ‘B’);
enqueue(q, ‘C’);
x = dequeue(q);
enqueue(q, ‘D’);
enqueue(q, ‘E’);
x = dequeue(q);
enqueue(q, ‘H’);
x = dequeue(q);
enqueue(q, ‘J’);
```

The contents of the queue “q” after these operations would be



Array Implementation

The array to implement the queue would need two variables (indices) called *front* and *rear* to point to the first and the last elements of the queue.



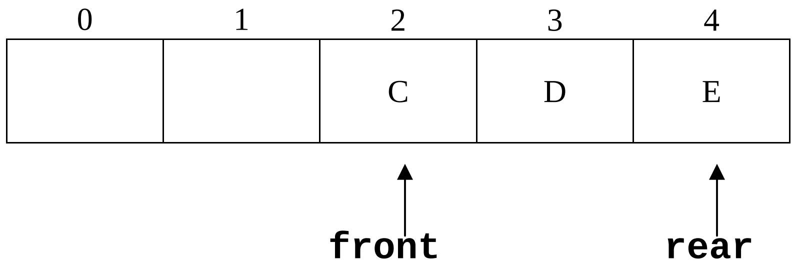
Initially:

```
q->rear = -1;
q->front = -1;
```

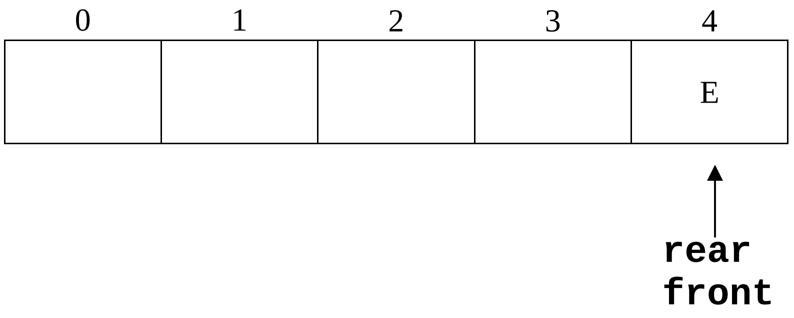
For each ***enqueue*** operation *rear* is incremented by one, and for each ***dequeue*** operation , *front* is incremented by one.
While the ***enqueue*** and ***dequeue*** operations are easy to implement, there is a big disadvantage in this set up. The size of the array needs to be huge, as the number of slots would go on increasing as long as there are items to be added to the list (irrespective of how many items are deleted, as these two are independent operations.)

Problems with this representation:

Although there is space in the following queue, we may not be able to add a new item. An attempt will cause an overflow.



It is possible to have an empty queue yet no new item can be inserted.(when *front* moves to the point of *rear*, and the last item is deleted.)

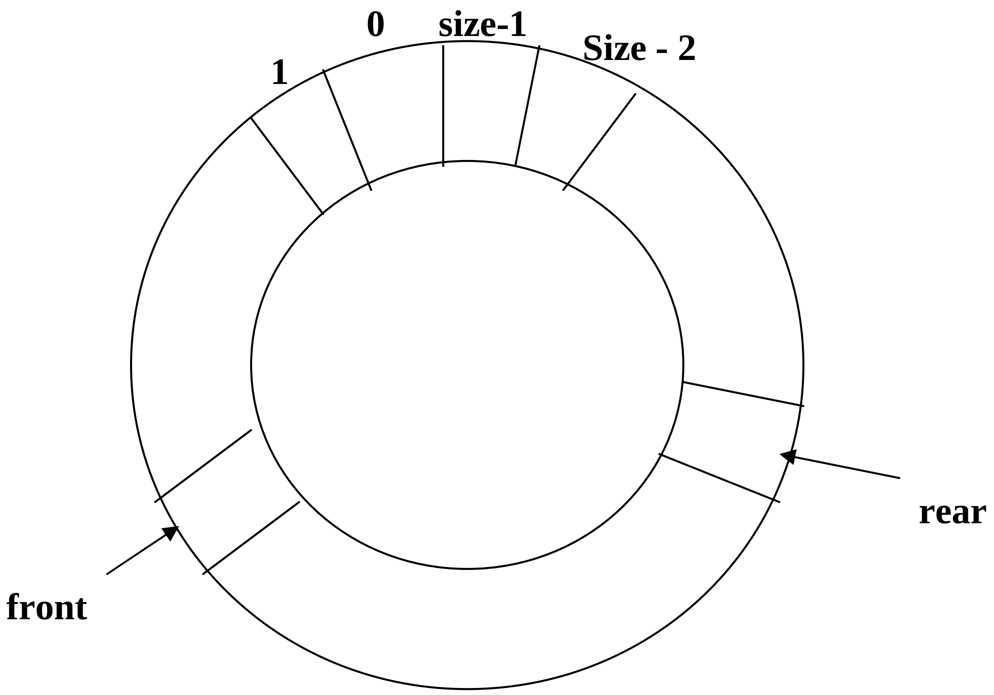


A solution: Circular Array

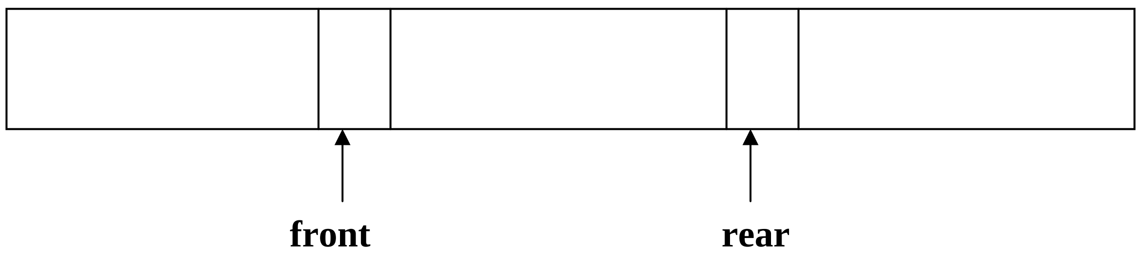
Let us now imagine that the above array is wrapped around a cylinder, such that the first and last elements of the array are next to each other. When the queue gets apparently full, it can continue to store elements in the empty spaces in the beginning of the array. This makes efficient use of the array slots. It is also referred to as a circular array. This enables us to utilize the unavailable slots, provided the indices *front* and *rear* are handled carefully.

Here again *front* refers to the index of the element to be next removed from the queue, and *rear* refers to the index of the last element added to the queue.

–



equivalently:



The queue is implemented on a circular array. With each enqueue, the *rear* index is incremented, and with each dequeue the *front* index is incremented. The wrap around is achieved by taking the mod function with the capacity of the array.

The queue is initialized with *front* = *rear* = 0. When the first element is enqueued, the *rear* index becomes 1 while the *front* index remains 0. When the second element is enqueued, the *rear* index becomes 2 while the *front* index remains 0.

If we now dequeue, the *front* index becomes 1, and at the next dequeue operation, the *front* index becomes 2. The queue is now empty as two elements were stored and these two were retrieved. The index values are *front* = *rear* = 2.

In fact, at any stage the condition *front* = *rear* indicates that the queue is empty.

Consider a queue with *n* slots. Suppose the *front* index is zero, and we go on filling up the queue. When the last but one slot is filled up the *rear* index will have value *n-1* . Now if we want to insert another element in the queue, the *rear* index will be *n.* But since we do not have the slot *n,* it would point to the slot *0.* Thus, while the queue is actually full we have *front* = *rear* = 0, which is not possible because that is a condition for the queue to be empty. Here we are faced with a contradiction.

To overcome this problem, the last element is never filled up in a circular queue, and when only a single slot is empty, we declare the queue to be full.

Queue Size :

The *front* and *rear* indices can be used to find out the current *size* of the queue, that is, the number of elements currently in the queue. The value *rear* – *front* gives us the *size*, and when this value is negative, we simply add the capacity to this to give us the size. Thus in general we have

size = (rear – front + capacity) % capacity.

Enqueue and Dequeue algorithms:

Here are the enqueue and the dequeue algorithms. It is assumed here that the variables, rear, front and capacity are globally visible.

```
initially,

rear = 0;
front = 0;


void enqueue(int Q[], int d) {
     rearplus1 =  (rear+1)% capacity;
     if (rearplus1 == front)
          print “Q full”;
     else
     {
          Q[rear]= d;
          rear = rearplus1;
     }
 }




void dequeue( int Q[ ], *dd ) {
     if(front == rear)
     {
          *dd = -9999 ;
           print “Q empty”
     }
     else
     {
          *dd = Q[front];
          fron =(front +1)%capacity;
      }
}
```