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 operations**

  - enqueue \((q, x)\):
    
    inserts item \(x\) at the rear of the queue \(q\)

  - 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:

```c
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

```
D E H J
```

front rear
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.

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>3</td>
<td>8</td>
<td>11</td>
<td>9</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Initially:

```c
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.

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>C</td>
<td>D</td>
<td>E</td>
</tr>
</tbody>
</table>

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

\begin{center}
\begin{tabular}{|c|c|c|c|c|}
\hline
0 & 1 & 2 & 3 & 4 \\
\hline
\end{tabular}
\end{center}

\textbf{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 \textit{front} and \textit{rear} are handled carefully.

Here again \textit{front} refers to the index of the element to be next removed from the queue, and \textit{rear} refers to the index of the last element added to the queue.
equivalently:
To illustrate the use of circular array, let us take an example of a Queue with 7 slots, i.e.
size is 7. The indices front and rear can take on values from 0 to (size – 1)
corresponding to elements present in the queue.
Initially there are no elements, so let us keep both front and rear equal to −1.

Now we let us study the effect of following operations and see how front and rear indices
are changed.

enqueue 20

Since rear = −1, element 20 will be stored in slot 0.

\[
\begin{array}{ccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 \\
20 & & & & & & \\
\end{array}
\]

Now there is a single element in the queue in slot 0. It is the first element, so front = 0.
It is also the last element, so rear = 0. This means whenever front = −1 and an
element is added, we have to make front = 0. The indices have the following values:

<table>
<thead>
<tr>
<th>Front</th>
<th>rear</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

enqueue 15

\[
\begin{array}{ccccccc}
20 & 15 & & & & & \\
\end{array}
\]

Note this operation does not affect the index front, which remains at 0. There are two
elements in the queue, so the last element added moves value of index rear to 1.

<table>
<thead>
<tr>
<th>Front</th>
<th>rear</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

enqueue 6
enqueue 11

Check if there is still space in the queue. If rear is not yet equal to size − 1, slots are still
available in the queue. So increment rear and put the new element there. This will take
rear to 3.

\[
\begin{array}{ccccccc}
20 & 15 & 6 & 11 & & & \\
\end{array}
\]

<table>
<thead>
<tr>
<th>Front</th>
<th>Rear</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3</td>
</tr>
</tbody>
</table>

At this point we ask the Queue to remove an element.

\[x = \text{dequeue}\]
It is going to remove the element being pointed by front, which is 20. This means the element at front is now 15 located at position 1.

```
15  6  11
Front 1  Rear 3
```

Now we perform the operations

```
enqueue 9
enqueue 7
x = dequeue
x = dequeue
```

This will cause 15 and 6 be removed from the queue, and the index front to move to 3, as 11 is the element currently in front.

```
11  9  7
Front 3  Rear 5
```

```
enqueue 8
```

This will apparently show that there is no more space in the queue after the last slot is filled up.

```
11  9  7  8
Front 3  Rear 6
```

```
enqueue 4
```

What will happen now? There are no slots after 8, but part of the queue is still empty. So the new element can be put in the first position, which gives us the rule “if rear = size – 1, then put the element in slot 0 and assign rear = 0.”
enqueue 6  
enqueue 1

The picture at this stage looks like this

<table>
<thead>
<tr>
<th>4</th>
<th>6</th>
<th>1</th>
<th>11</th>
<th>9</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Front</td>
<td>rear</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

enqueue 3

We cannot carry out this operation, as the Queue is physically full. We can see that ourselves. But how would a program find out that the area allocated to the queue is full? Observe how front and rear are related. You will always find that the queue is full whenever

front = rear + 1

**Special cases of Dequeue**

Now just suppose at some point in time, this is how the queue looks like:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Front</td>
<td>rear</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

And we perform

x = dequeue

The list is going to be empty after that. To indicate that the queue is now completely empty, we must set

front = rear = – 1

Remember this is how we had initialized the queue.
There can be another situation, where again we are going to remove the element in last position of the array, but there are other elements in the queue. Consider this:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Front: 6  rear: 1

\( x = \text{Dequeue} \)

Here we simply wrap around the array and set front = 0.

Thus, for Dequeue operation
if front not equal to rear, and front = size – 1
Then we have to assign front = 0.
For all other cases we simply do front ++.

So when do we say that the Queue is empty?
when front = – 1.

When do we say that the Queue is full?
There are two situations:

1) front = 0 and rear = size – 1 OR
2) front = rear + 1.