Queues

A queue is an ordered collection of items from which items may be deleted at one end (collect the **front** of the queue) and into which items may be inserted at the other end (called the **Rear** of the queue). The figure(1) illustrates a queue containing three elements **A, B** and **C**.

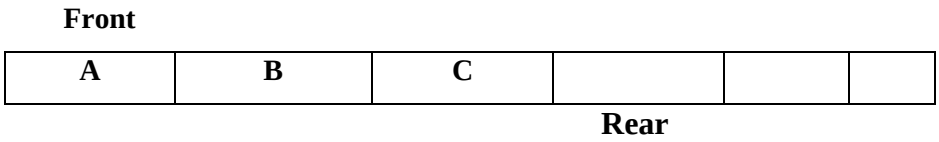


Figure -1-

In figure (2) an element has been deleted from the queue. Since may be deleted only from the front of the queue, A is removed and B is now at the front.

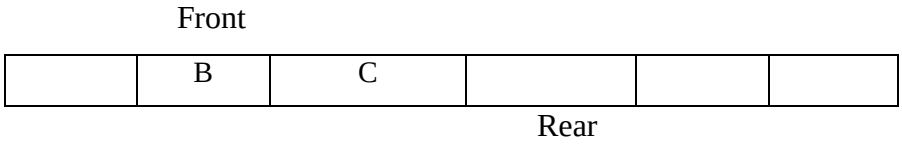


Figure -2-

In figure (3) , when items D and E are inserted , they may be inserted at the rear of the queue.

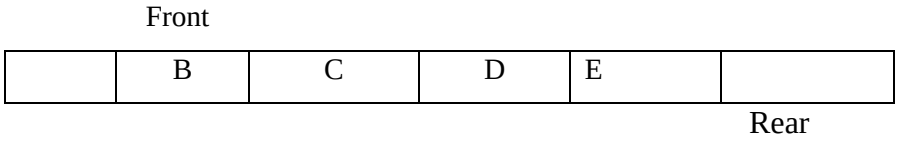


Figure -3-

The Queue Abstract Data Type

- Queue is an ADT data structure similar to stack, except that the first item to be inserted is the first one to be removed.
- This mechanism is called First-In-First-Out (FIFO).
- Placing an item in a queue is called “insertion or enqueue”, which is done at the end of the queue called “rear”.
- Removing an item from a queue is called “deletion or dequeue”, which is done at the other end of the queue called “front”.
- Used extensively in operating systems
    - Queues of processes, I/O requests, and much more

Problem:

After a few insert and delete operations the rear might reach the end of the queue and no more items can be inserted although the items from the front of the queue have been deleted and there is space in the queue.

Solution: Circular Queue

The figure(4) illustrates a Circular Queue containing five elements **A, B** , C , D and **E**.

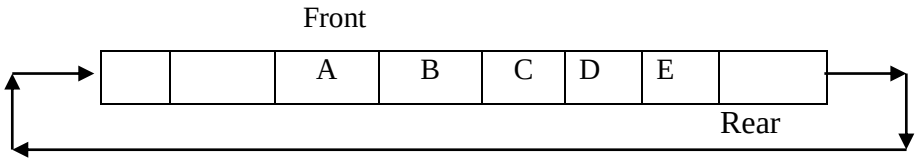


Figure -4-

In figure (5) , when items F are inserted , they may be inserted at the rear of the queue.

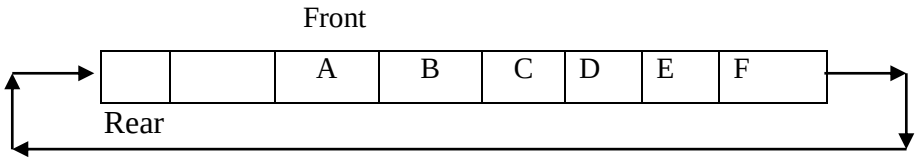


Figure -5-

Class specification :

Queue
-MaxSize:int **- QueueArray[]:object** **-front:int** **-rear:int**
+Queue(int) **+ enqueue (object):void** **+ dequeue():object** **+Front():object** **+IsFull():bool** **+IsEmpty():bool** **+Size():int**

The queue abstract data type (ADT) supports the following Methods:

ADT : Queue

{

Data: a non zero positive integer number representing MaxSize . and array of object elements represent the QueueArray.

Operations:

A constructor(queue) :initialize the data to some Data object certain value.

enqueue (element): Insert object element at the rear of the queue.

Input : Object; Output: None.

dequeue(): Remove and return from the queue the object at the front ; an error occurs if the queue is empty.

Input : None; Output: Object.

Notice that, as with the stack pop() operation, the dequeue () operation returns the object that was removed (an alternative to avoid unnecessary copying by defining dequeue() so that no value is returned) . The queue ADT also includes the following supporting member methods:

Front(): Return, but do not remove, a reference to the front element in the queue ; an error occurs if the queue is empty.

Input : None; Output: Object.

Size(): Return the number of objects in the queue .

Input : None; Output: Integer.

IsEmpty() : Return a Boolean value indicating if the queue is empty.

Input : None; Output: Boolean.

IsFull() : Return a Boolean value indicating if the queue is full.

Input : None; Output: Boolean.

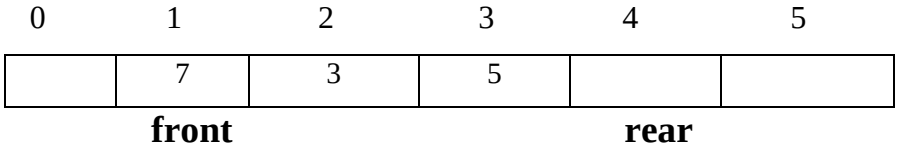
} end ADT Queue

We illustrate the operation in the queue ADT in the following example:

Example: The following table shows a series of queue operation and their effect on an initially empty queue Q of integer .

Queue q=new Queue [6];

Operation	Output	front	Rear	Queue
		0	0	
enqueue(5)	–	0	1	(5)
enqueue(3)	–	0	2	(5, 3)
dequeue()	5	1	2	(3)
enqueue(7)	–	1	3	(3, 7)
dequeue()	3	2	3	(7)
Front()	7	2	3	(7)
dequeue()	7	3	3	()
dequeue()	"error"	3	3	()
IsEmpty()	true	0	0	()
enqueue(9)	–	0	1	(9)
enqueue(7)	–	0	2	(9, 7)
Size()	2	0	2	(9, 7)
enqueue(3)	–	0	3	(9, 7, 3)
enqueue(5)	–	0	4	(9, 7, 3, 5)
dequeue()	9	1	4	(7, 3, 5)



class Queue

{

// data member or data value

private int front , rear, MaxSize ;
private object[] QueueArray;

// Constructer or default Constructer

```
public Queue(int n)
  {
      MaxSize = n;
      QueueArray = new object[MaxSize];
       rear = 0;
      front= 0;
   }
}
```

Queue Operations

We describe how to use this method to implement a queue in code :

Implementation Array-based Queue

To Summarize the discussion of Queues , let us list all the methods we have discussed for implementing queue:

1 – **The physical model:** a linear array with the front always in the first position and all entries moved up the array whenever the front is removed . This is generally a poor method for use in computer .(O(n)).

Pseudocode deQueue():

Physical mode	c.f.
if isEmpty() then	1
Throw a queueEmpty Exception	1
else	
{	
item ← QueueArray[0]	1
for(i←0 to (size()-2))	n
{	
QueueArray[i] ← QueueArray[i+1]	n
}	
rear←rear-1	1
return item	1
}	= 2n+5
	=O(n)

2- A linear array with two indices always increasing . this is a good method if the queue can be emptied all at once . (all methods O(1))

Pseudocode Size():

```
return ( rear-front)
```

Pseudocode IsEmpty():

```
if (front equal rear ) { front ← 0; rear ← 0; return true } else return false
```

Pseudocode Front():

```
if IsEmpty() then
        Throw a queueEmpty Exception
Return QueueArray[front]
```

Pseudocode dequeue():

```
if  IsEmpty() then
        Throw a queueEmpty Exception
else
{
   item ← QueueArray[front]
   front←front+1
 return item
}
```

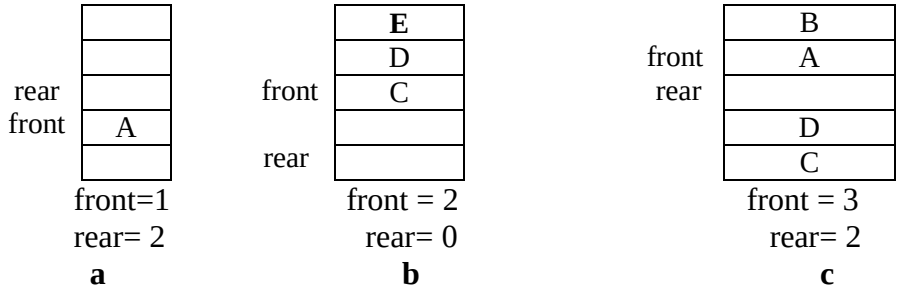
Pseudocode IsFull():

```
If ( rear equal  Maxsize)   return true  else   return false
```

Pseudocode enqueue(element):

```
if  IsFull()  then
        Throw a queueFullException
else
{
     QueueArray[rear] ←element
      rear←rear+1
}
```

3- <u>A circular array with use a gap cell within the array to check for fullness or emptiness</u>



In figure (a) removing the last element, leaving the queue empty (front=rear) . In figure (c) in adding an element to the last free slat in the queue leaving the queue full (front = rear). The value of front and rear in the two situation are identical.

One solution is to use a **gap cell** within the array to check for fullness or emptiness

Pseudocode Size():

```
return (MaxSize -  front+rear )mod  MaxSize
```

Pseudocode IsEmpty():

```
if (front  equal  rear )  {   front ← 0 ;   rear ← 0;   return true  }   else      return  false
```

Pseudocode Front():

```
if  IsEmpty() then
        Throw a queueEmpty Exception
Return QueueArray[front]
```

Pseudocode dequeue():

```
if  IsEmpty() then
        Throw a queueEmpty Exception
else
{
    item ← QueueArray[front]
    front← (front+1) mod MaxSize
   return item
}
```

Pseudocode IsFull():

```
If ((rear+1)% MaxSize  equal front)    return true  else  return false
```

Pseudocode enqueue(element):

Circular Queue (gap cell)

```
if  IsFull ()   then
        Throw a queueFullException
else
{
   QueueArray[rear] ← element
   rear ← (rear+1) mod MaxSize
}
```

Example1 : Consider the following operations on a circular queue data structure that stores integer values?

```
Queue   Q1=new Queue  [10];
  for (int i=0; i< 7  ; i++)
      Q1.enqueue (i) ;
  for (int i=1; i< 5  ; i++)
      Q1.enqueue(Q1.dequeue ()) ;
```

What queue will look like after the code above executes(what position of rear and front)?

Solution

0	1	2	3	4	5	6	7	8	9
3				4	5	6	0	1	2
	Rear			Front					

Example2 : Consider the following operations on a circular queue data structure that stores integer values?

Queue q=new Queue(9);

q.enqueue(6); q.enqueue(1); q.enqueue(8); q.enqueue(q.dequeue()); q.enqueue(q.dequeue()); q.enqueue(5); q.enqueue(9); q.enqueue(q.dequeue()); q.enqueue(q.dequeue());q.enqueue(3);

what q will look like after the code above executes(what position of rear and front)?

Solution

0	1	2	3	4	5	6	7	8
3				1	5	9	8	6
	Rear			Front				

The table show the running times of methods in realization of a queue by an array

Method	Time
Size	O(1)
isEmpty	O(1)
Front	O(1)
Enqueue	O(1)
Dequeue	O(1)

Exercises

Q1/Write method to print Queue?
Q2/Write method to print Circular Queue ?
Q3/ Consider the following operations on a circular queue data structure that stores integer values?

Queue q=new Queue [10];
q.enqueue(6);
q.enqueue(1);
q.enqueue(q.dequeue());
q.enqueue(8);
q.enqueue(2);
q.enqueue(5);
q.enqueue(9);
q.enqueue(q.dequeue());
q.enqueue(q.dequeue());
what q will look like after the code above executes?

Q4/ Consider the following operations on a circular queue data structure that stores integer values?

Queue q=new Queue [6];
q.enqueue(9);
q.enqueue(3);
q.enqueue(5);
q.enqueue(q.dequeue());
q.enqueue(4);
q.enqueue(2);
q.enqueue(q.dequeue());
q.enqueue(q.dequeue());
q.enqueue(8);

what q will look like after the code above executes?

Q5/ Describe the output for the following sequence of queue operation?

```
Queue  q=new Queue [7];
q.enqueue(5);
q.enqueue(3);
q.dequeue()
q.enqueue(2);
 q.enqueue(8);
q.dequeue();
q.dequeue();
q.enqueue(7);
```

Q6/ Describe the output for the following sequence of circular queue operation?

```
Queue   []  q=new Queue [6];;
q.enqueue(15);
q.enqueue(13);
q.dequeue()
q.enqueue(21);
 q.enqueue(8);
q.dequeue();
q.dequeue();
q.enqueue(7);
q.enqueue(9);
q.enqueue(1);
q.enqueue(8);
```