Data Structure & Algorithms

Lec 05: Queues

Fall 2017 - University of Windsor
Dr. Sherif Saad

Agenda

1. Motivation & Background
2. Queue Definition, Properties and Applications
3. Queue as an ADT
4. Self Assessment

Learning Outcome

By the end of this class, you should be able to

- Explain the queue operations and its applications
- Implement the queue as ADT
- Use queue to solve problems that require queue insertion and deletion behaviors

Motivation and Background

A **queue** is a linear data structure similar to arrays, linked list, and queues. The queue defines one restriction over elements insertion and deletion. The elements are added at one end (rear) and removed from the (front) the other end.

The first element inserted is the first element to be removed from the queue.

The way the queue handle elements insertion and deletion is called First In First Out (FIFO) or Last In Last Out (LILO).

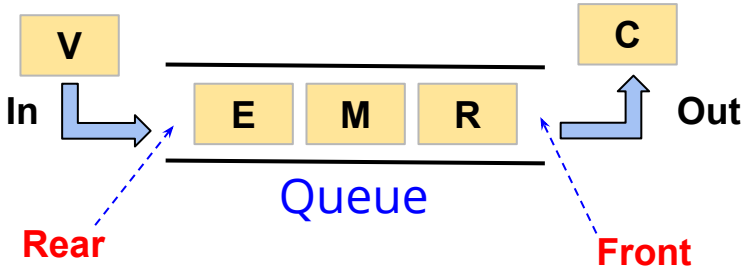
The insertion of an element into the queue called enqueue operation and the deletion of an element from the queue called a dequeue operation.

How does the Queue work?

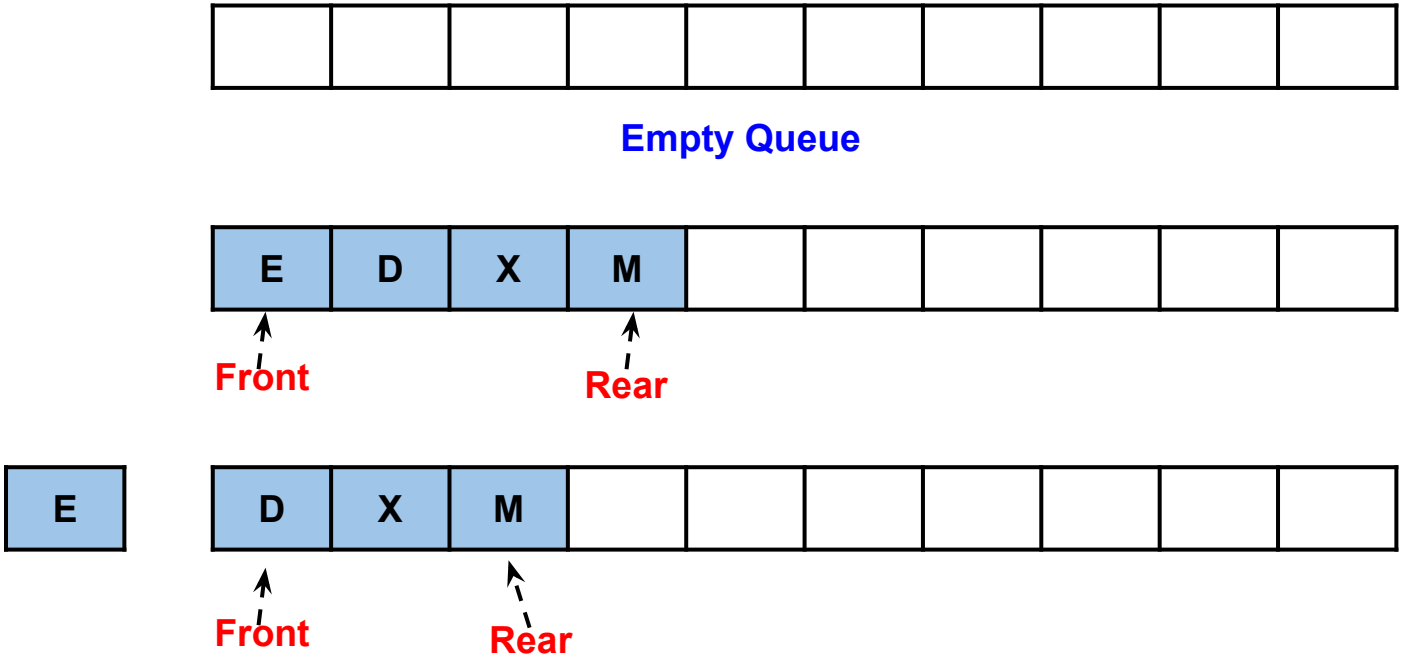
We need two special pointers Front and Rear. We add elements to the rear of the queue and remove elements from the front of the queue.

Trying to add an element to a full queue result in overflow condition

Trying to remove an element from an empty queue result in underflow condition



How does the Queue work?



Queue Applications

Many real-life applications require queue behaviors (FIFO) structures.

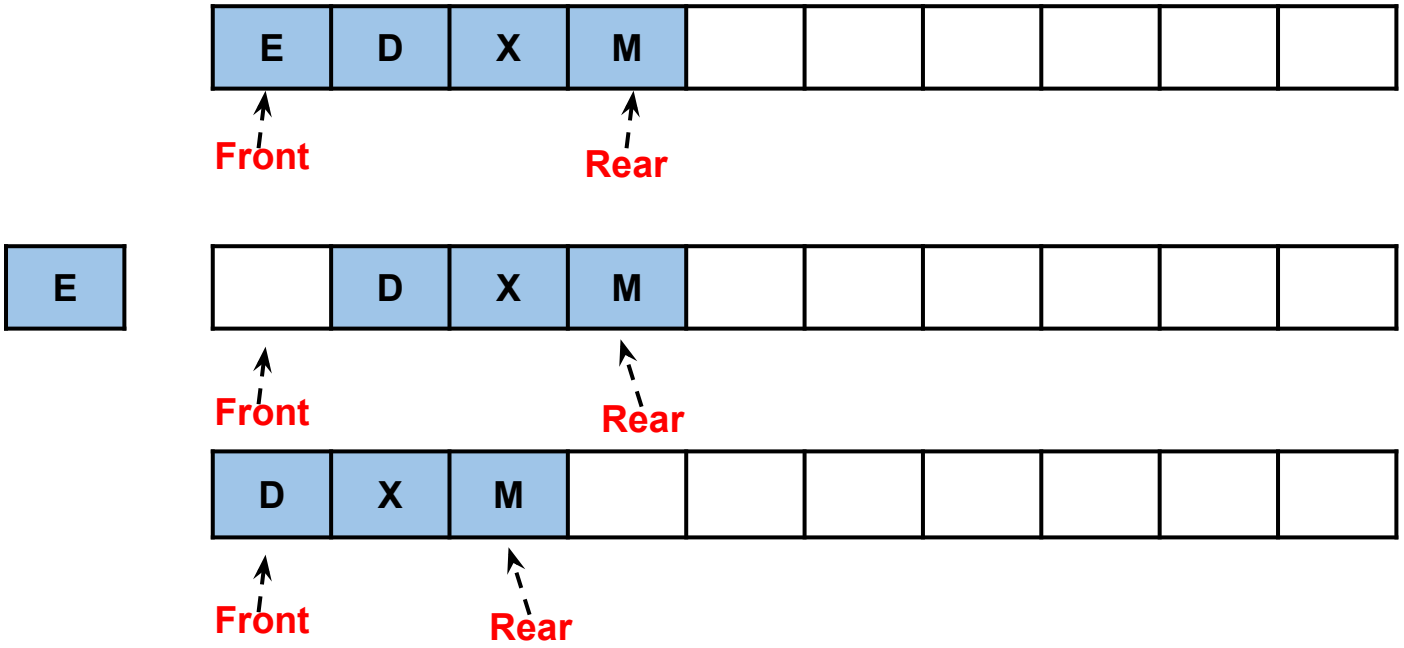
1. Operating System schedule jobs (with equal priority like print queue)
2. Multiprogramming (Message Queue)
3. Cloud Computing
4. Asynchronous Data Transfer.
5. Graph traversing algorithms (e.g. Breadth First Search)

Queue as ADT

1. Add an Element into the queue
2. Delete an Element from the queue
3. Initialize the queue
4. Check if the queue is full
5. Check if the queue is empty
6. Measure the size of the queue
7. Processing queue elements
8. Delete all elements in the queue
9. Check the top of the queue

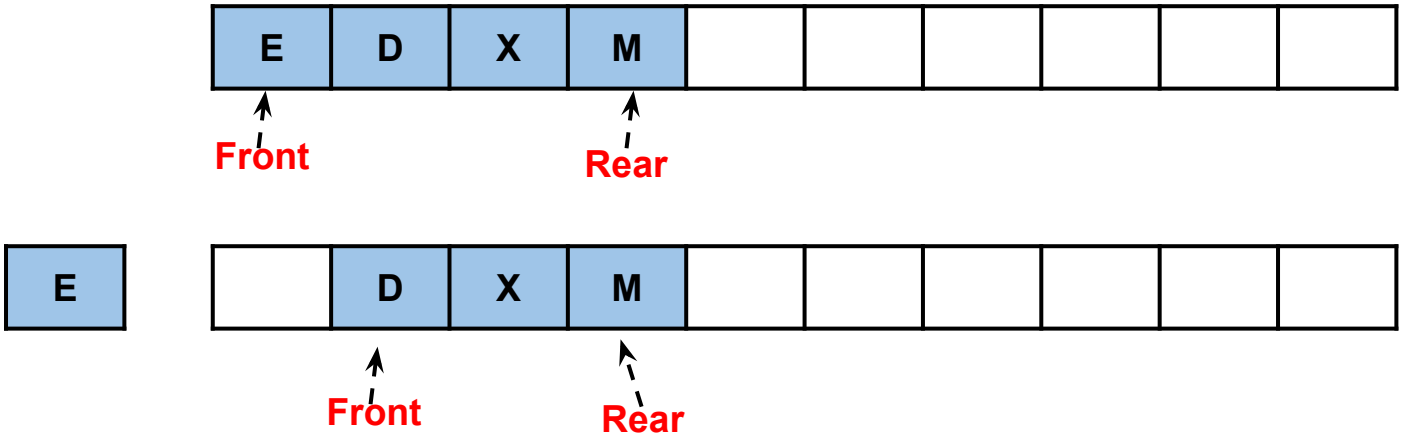
Queue ADT: Array Based Implementation

Using simple array to implement a queue data structure is inefficient



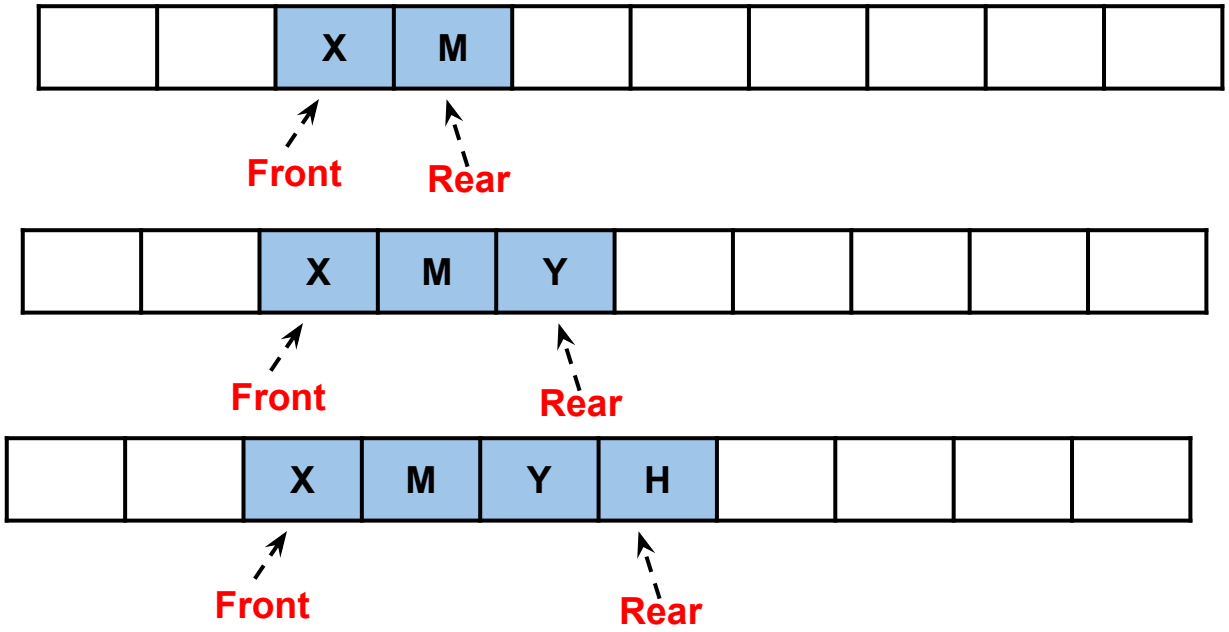
Queue ADT: Array Based Implementation

It is better to use a circular array (we move the front and the rear indices and not shifting the elements)



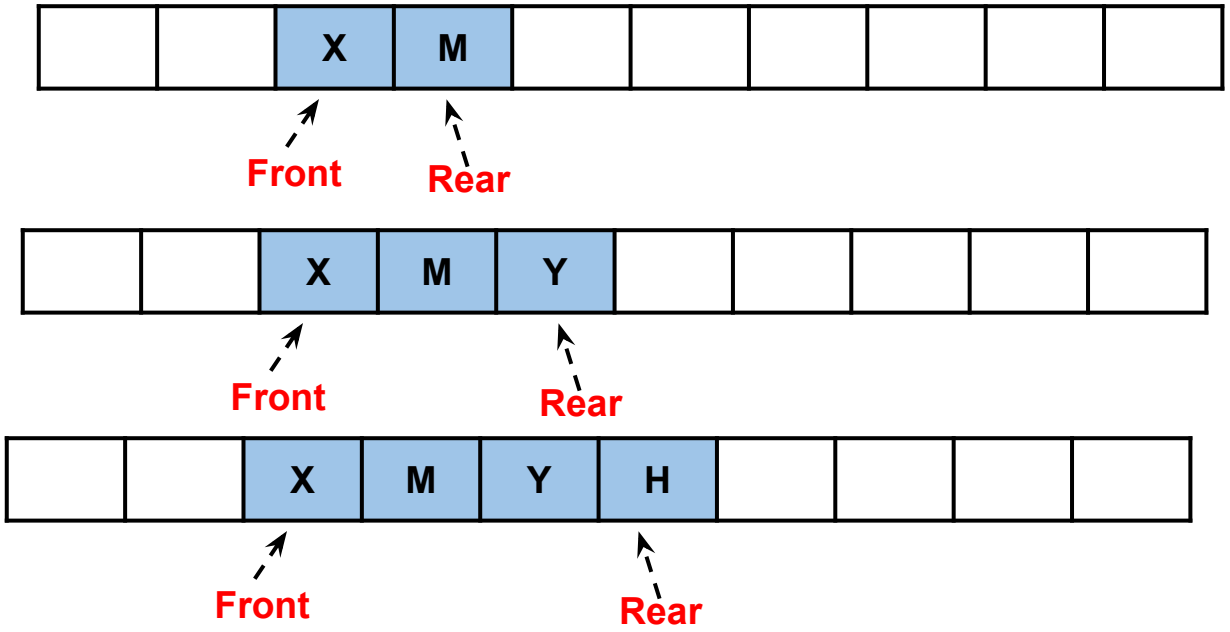
Queue ADT: Array Based Implementation

Shifting the elements has a time complexity of O(n)

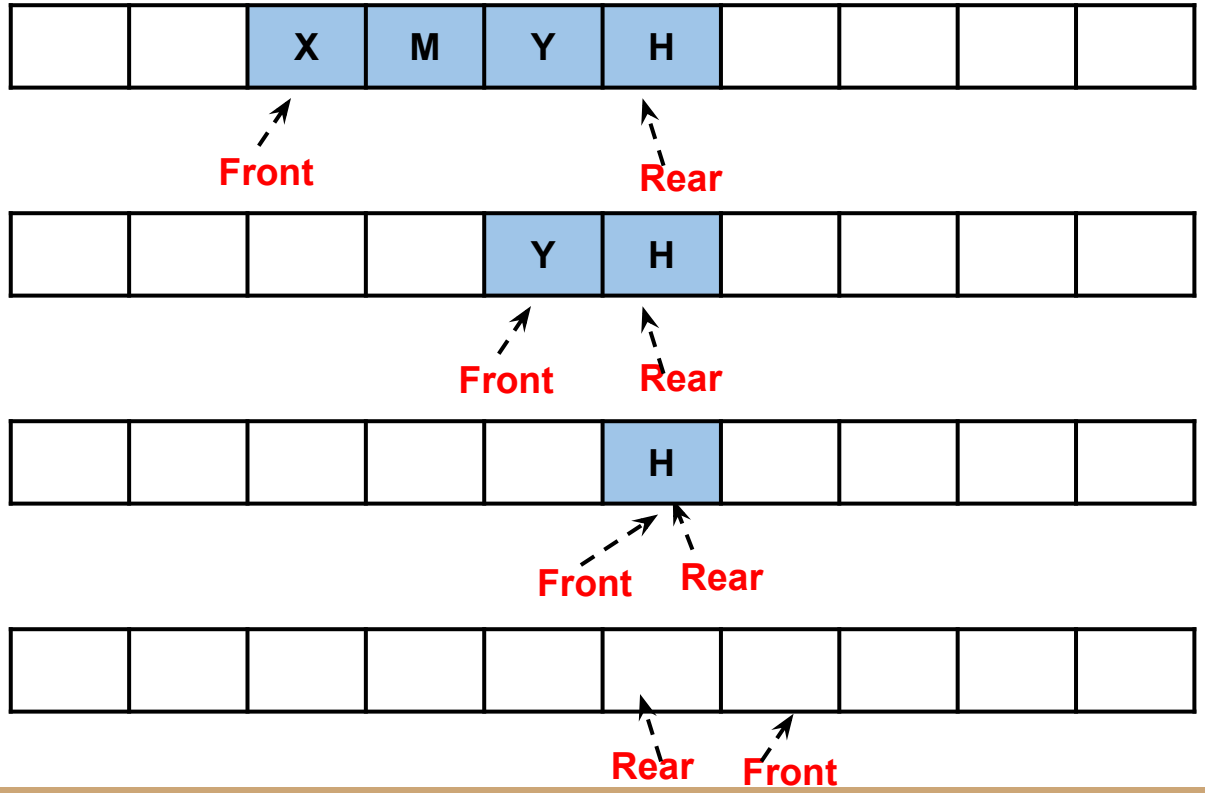


Queue ADT: Array Based Implementation

Moving the front and end indices give a time complexity of O(1)

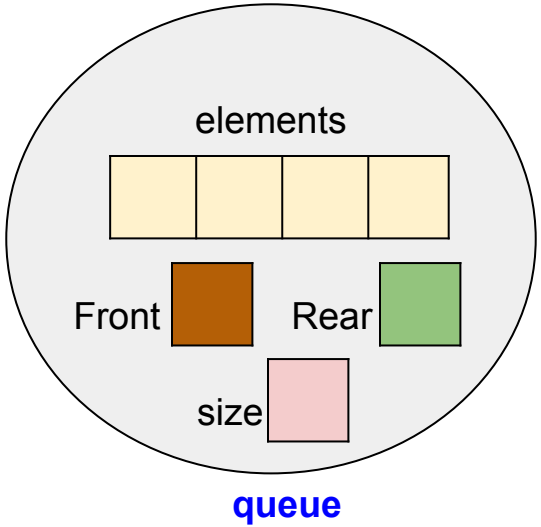


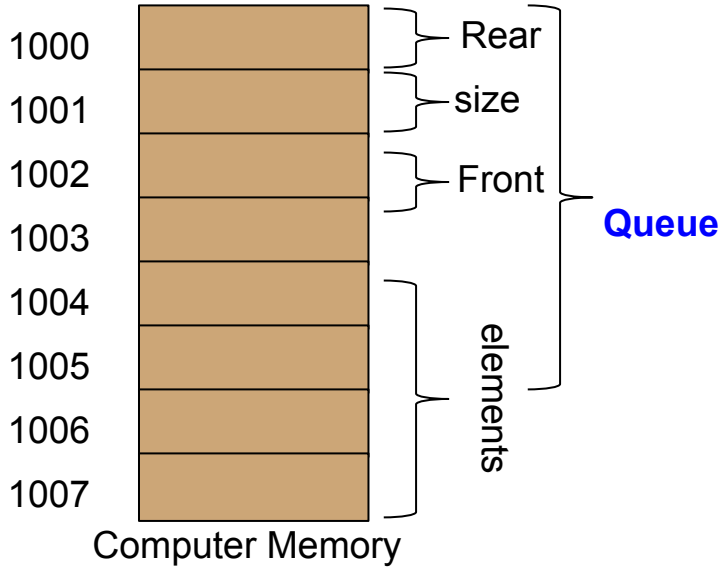
Queue ADT: Array Based Implementation



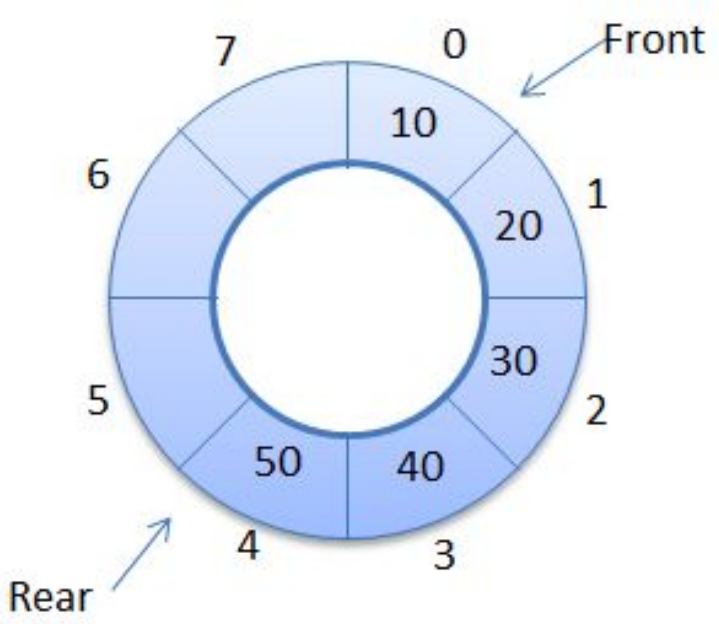
Queue ADT: Array based Implementation

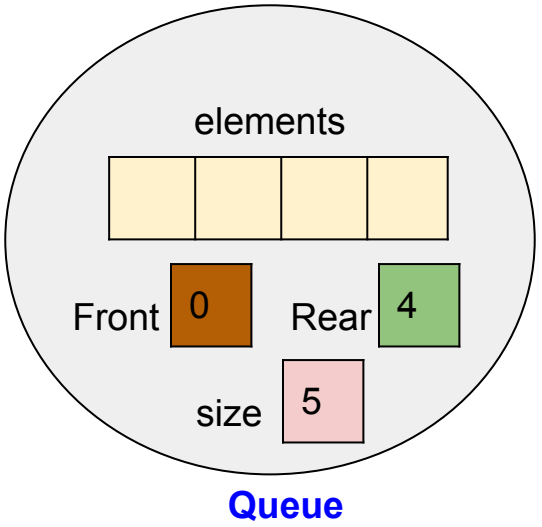
With circular array we need to store the front and rear indices of our queue and also store the size (the current non empty cells in the array). We use the size to check if the queue is full or empty.





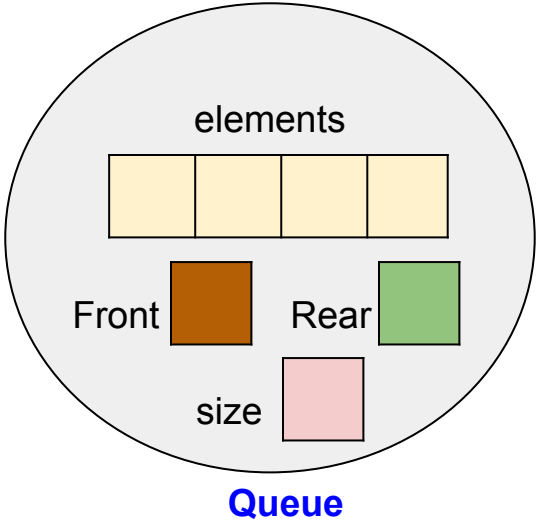
Queue ADT: Circular Array based Implementation





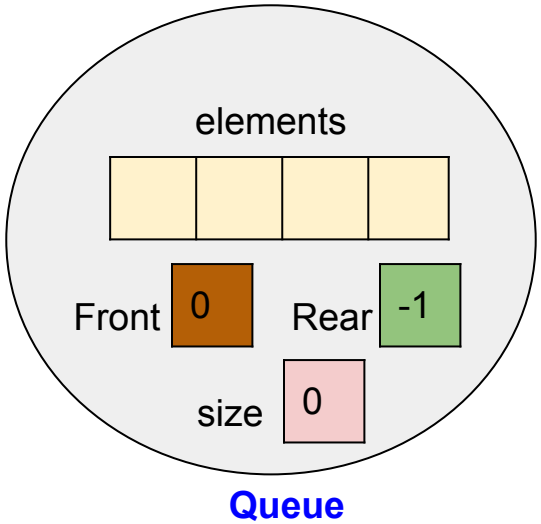
Queue ADT: Queue Structure

```
typedef struct Queue{

    int size;
    int front;
    int rear;

    QueueElement elements[MAXQUE];

}Queue;
```



Queue ADT: Queue Initialization

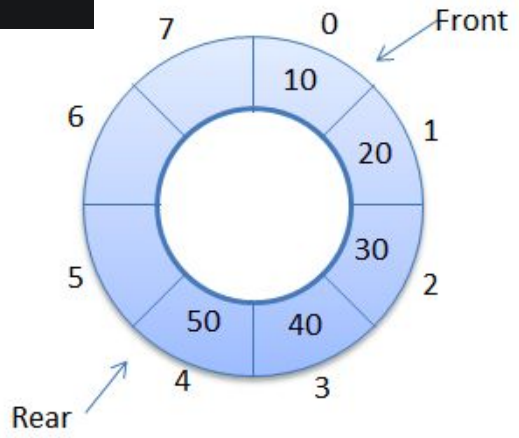
```
13 void InitiateQueue(Queue *ptrQ){
14
15     ptrQ->front=0;
16
17     ptrQ->rear=-1;
18
19     ptrQ->size=0;
20 }
```



Queue ADT: Add Element (EnQueue)

```
24 void Enqueue(QueueElement e, Queue *ptrQ){
25
26     ptrQ->rear = ptrQ->rear+1;
27
28     ptrQ->elements[ptrQ->rear] = e;
29
30     ptrQ->size++;
31 }
```

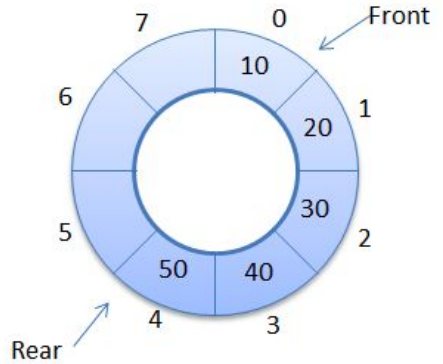
Remember we are working with a circular queue



Queue ADT: Add Element (EnQueue)

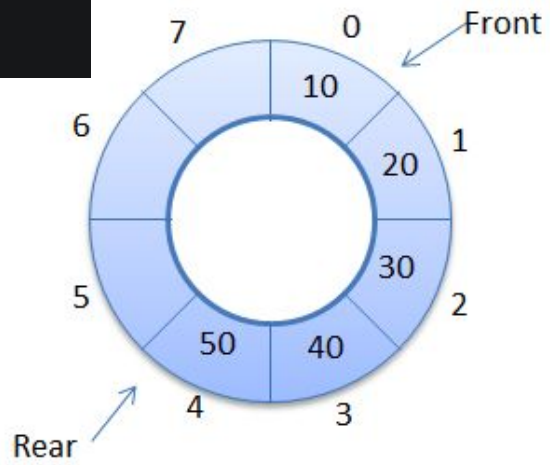
```
24 void Enqueue(QueueElement e, Queue *ptrQ){
25
26     if(ptrQ->rear == MAXQUE-1){
27         ptrQ->rear = 0;
28     }else{
29         ptrQ->rear++;
30     }
31
32     ptrQ->elements[ptrQ->rear] = e;
33
34     ptrQ->size++;
35 }
```

Remember we are working with a circular queue



Queue ADT: Add Element (EnQueue)

```
24 void Enqueue(QueueElement e, Queue *ptrQ){
25
26     ptrQ->rear = (ptrQ->rear+1) % MAXQUE;
27
28     ptrQ->elements[ptrQ->rear] = e;
29
30     ptrQ->size++;
31 }
```



Queue ADT: Queue Full, Queue Empty, Queue Size

```
3  int IsFullQueue(Queue *ptrQ){
4
5      return ptrQ->size == MAXQUE;
6  }
7
8
9  int IsEmptyQueue(Queue *ptrQ){
10
11     return ptrQ->size == 0;
12 }
13
14 int QueueSize(Queue *ptrQ){
15
16     return  ptrQ->size;
17 }
```

Queue ADT: Remove an Element (DeQueue)

```
16 void Dequeue(QueueElement *ptrE, Queue *ptrQ){
17
18     *ptrE = ptrQ->elements[ptrQ->front];
19
20     ptrQ->front = (ptrQ->front+1) % MAXQUE;
21
22     ptrQ->size--;
23
24 }
```

Queue ADT: Delete | Empty Queue and Queue Front

```
1  void DeleteQueue(Queue *ptrQ){
2
3      ptrQ->front=0;
4
5      ptrQ->rear=-1;
6
7      ptrQ->size=0;
8  }
9
10 void QueueFront(QueueElement *ptrE, Queue*ptrQ){
11
12     *ptrE = ptrQ->elements[ptrQ->front];
13 }
```

Queue ADT: Traverse Elements

```
 6 void TraverseQueue(Queue *ptrQ, void (*ptrFun)(QueueElement)){
 7
 8     int count =1;
 9
10     int index = ptrQ->front;
11
12     while (count<=ptrQ->size) {
13         (*ptrFun)(ptrQ->elements[index]);
14
15         index = (index + 1) % MAXQUE;
16
17         count++;
18     }
19 }
```

Run-Time Complexity

The run time complexity of the Queue operations with array-based implementation are:

Operation	Run-Time Complexity
Add an Element	O(1)
Remove an Element	O(1)
Queue Size	O(1)
Is Empty Queue	O(1)
Is Full Queue	O(1)
Delete | Queue queue	O(1)

Self-Assessment

Write an algorithm to reverse a queue of n elements using only the queue ADT methods?

Explain how could we implement a FIFO (queue) using two stacks?