Al-Mustaqbal University
College of Sciences
Intelligent Medical System Department

جامعة المستقبل
AL MUSTAQBAL UNIVERSITY

كلية العلوم

قسم الانظمة الطبية الذكية

Subject: Data Structure

Class: Second

Lecturer: Prof.Dr.Mehdi Ebady Manaa

Lecture: (5)

Queues

Queues

The word queue is British for *line* (the kind you wait in). In Britain, to "queue up" means to get in line. In computer science a queue is a data structure that is somewhat like a stack, except that in a queue the first item inserted is the first to be removed (First-In-First-Out, FIFO), while in a stack, as we've seen, the last item inserted is the first to be removed (LIFO). A queue works like the line at the movies:

The first person to join the rear of the line is the first person to reach the front of the line and buy a ticket. The last person to line up is the last person to buy a ticket (or—if the show is sold out—to fail to buy a ticket). Figure 4 shows how such a queue looks.

Figure 4: A queue of people.

Examples of applications Queue:

1. Queue used to model real-world situations such as people waiting in line at a bank, airplanes waiting to take off, or data packets waiting to be transmitted over the Internet.

2. There are various queues quietly doing their job in your computer's (or the network's) operating system.

3. There's a printer queue where print jobs wait for the printer to be available.

Also there are several possible applications for queues.

4. Stores, reservation centers, and other similar services typically process customer requests according to the FIFO principle. A queue would therefore be a logical choice for a data structure to handle transaction processing for such applications. For example, it would be a natural choice for handling calls to the reservation center of an airline.

The Queue Abstract Data Type

Formally, the queue abstract data type defines a collection that keeps objects in a sequence, where element access and deletion are restricted to the first element in the sequence, which is called the ***front*** of the queue, and element insertion is restricted to the end of the sequence, which is called the ***rear*** of the queue. This restriction enforces the rule that items are inserted and deleted in a queue according to the first-in first-out (FIFO) principle. The ***queue*** abstract data type (ADT) supports the following two fundamental methods:

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

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

Additionally, similar to the case with the Stack ADT, the queue ADT includes the following supporting methods:
size(): Return the number of objects in the queue.
isEmpty(): Return a Boolean value that indicates whether the queue is empty.
front(): Return, but do not remove, the front object in the queue; an error occurs if the queue is empty.

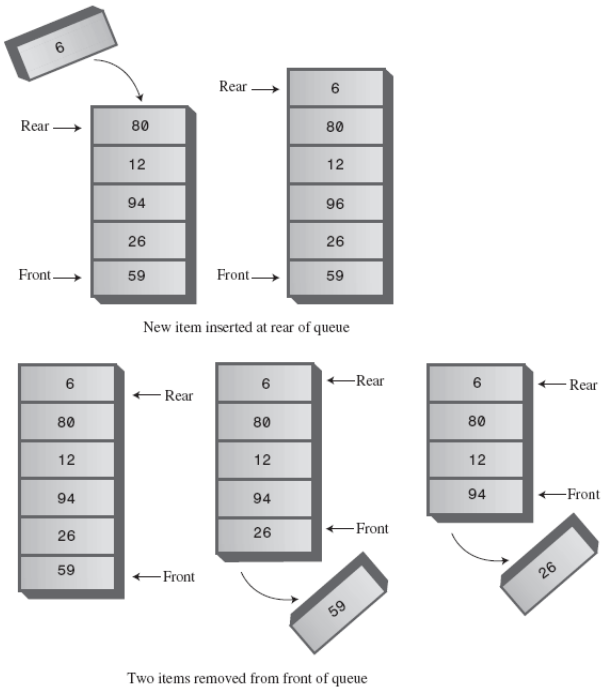


Figure 5: Operations of the Queue

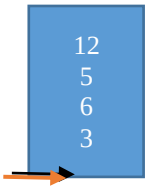
enqueue(*e*):
Description: Here QUEUE is an array with N locations. FRONT and REAR points to the front and rear of the QUEUE. ITEM is the value to be inserted.

1. If (REAR == N) Then [Check for overflow]
2. Print: Overflow
3. Else
4. If (FRONT and REAR == 0) Then [Check if QUEUE is empty]
   (a) Set FRONT = 1
   (b) Set REAR = 1
5. Else
6. Set REAR = REAR + 1 [Increment REAR by 1]
   [End of Step 4 If]
7. QUEUE[REAR] = ITEM
8. Print:

[End of Step 1 If]
9. Exit

dequeue():

Description: Here QUEUE is an array with N locations. FRONT and REAR points to the front and rear of the QUEUE.



1. If (FRONT == 0) Then [Check for underflow]
2. Print: Underflow
3. Else
4. ITEM = QUEUE[FRONT]
5. If (FRONT == REAR) Then [Check if only one element is left]
   (a) Set FRONT = 0
   (b) Set REAR = 0
6. Else
7. Set FRONT = FRONT + 1 [Increment FRONT by 1]
   [End of Step 5 If]
8. Print: ITEM deleted
   [End of Step 1 If]
9. Exit

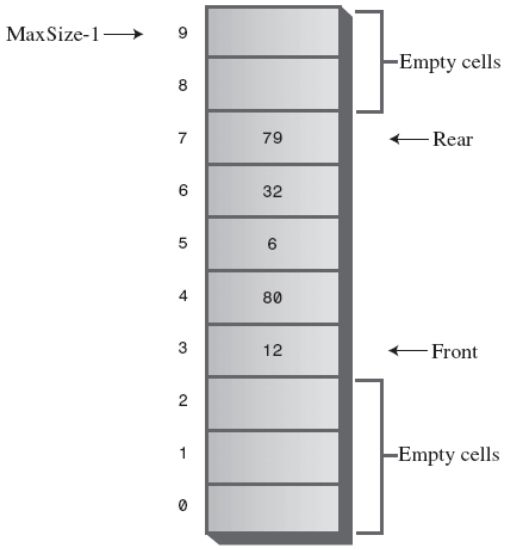


Figure 6: A Queue with some items removed

When you insert a new item in the Queue, the Front arrow moves upward, when you remove an item, Rear also moves upward.

The trouble with this arrangement is that pretty soon the rear of the queue is at the end of the array (the highest index). Even if there are empty cells at the beginning of the array, because you've removed them with F, you still can't insert a new item because Rear can't go any further. Or can it? This situation is shown in Figure 7.

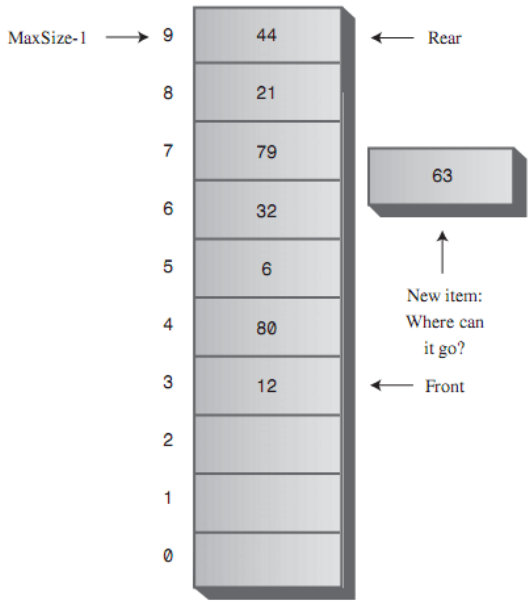


Figure 7. Rear arrow at the end of the array.

To avoid the problem of not being able to insert more items into the queue even when it's not full, the Front and Rear arrows wrap around to the beginning of the array. The results a **circular queue** (sometimes called a **ring buffer**). Insert enough items to bring the Rear arrow to the top of the array (index 9). Remove some items from the front of the array.
Now insert another item. You'll see the Rear arrow wrap around from index 9 to index 0; the new item will be inserted there. This situation is shown in Figure 8.

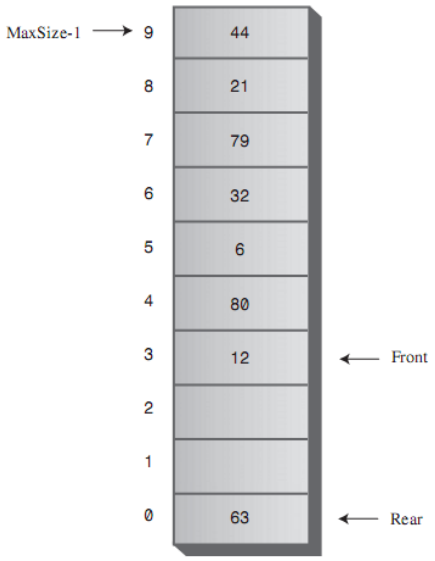


Figure 8. Rear arrow wraps around

The queue ADT

The queue ADT is defined by the following operations:
constructor
Create a new, empty queue.
insert
Add a new item to the queue.
remove
Remove and return an item from the queue. The item that is returned is the first one that was added.
empty
Check whether the queue is empty.

We know that a **linear queue** is a "first in first out " data structure,i.e.,

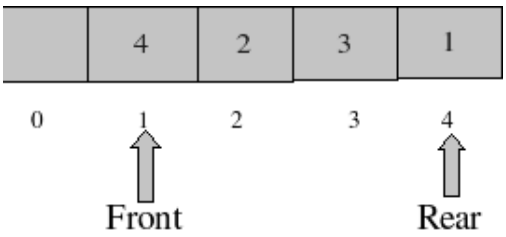
- **Insertion** can be **made only at the end** and
- **Deletion can be made only at the front**.

In a linear queue, the **traversal** through the queue is possible only **once**,i.e.,**once an element is deleted, we cannot insert another element in its position.** This disadvantage of a linear queue is overcome by a **circular queue**, thus saving memory.

Circular Queue

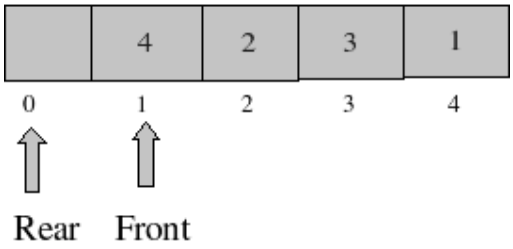
A circular queue is a Queue but a particular implementation of a queue. It is very efficient. It is also quite useful in low level code, because insertion and deletion are totally independant, which means that you don't have to worry about an interrupt handler trying to do an insertion at the same time as your main code is doing a deletion.

Linear queue:



No more elements can be inserted in a linear queue now.

Circular queue:

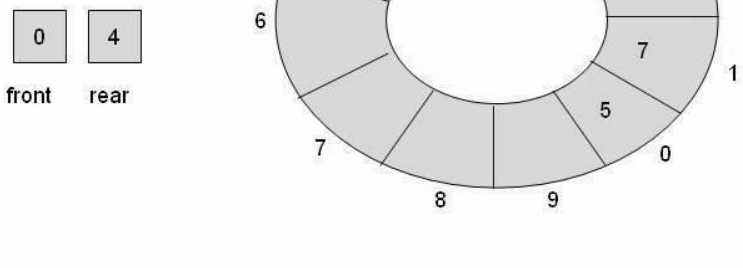


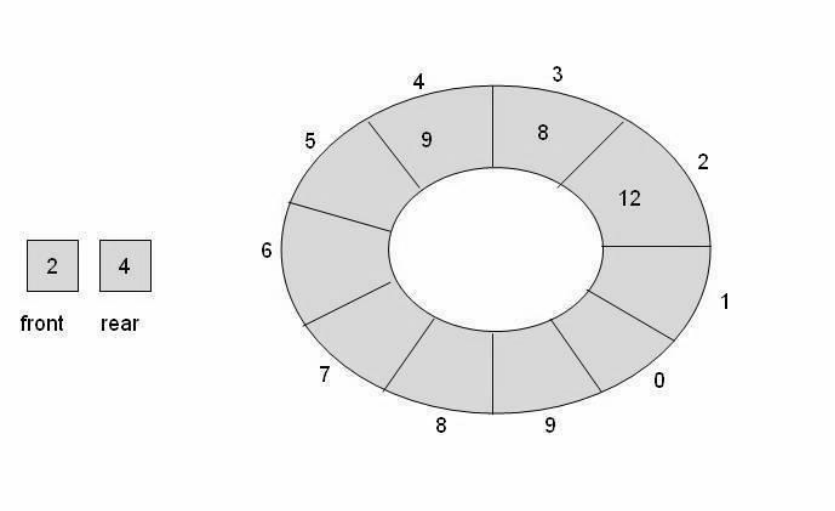
In a circular queue, after rear reaches the **end of the queue**, it can be **reset to zero**. This helps in refilling the empty spaces in between.

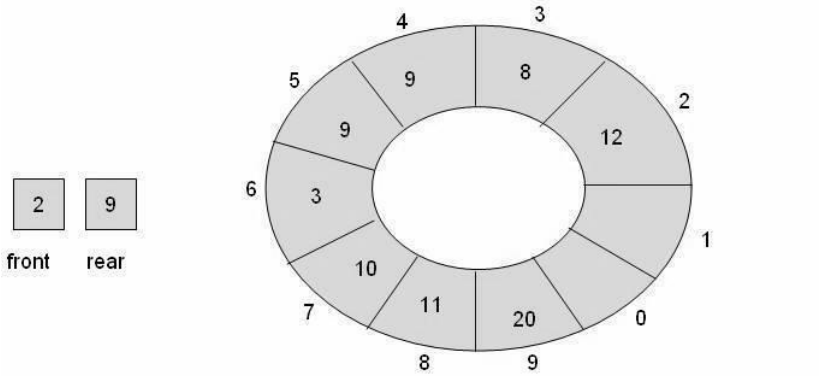
The difficulty of managing front and rear in an array-based non-circular queue can be overcome if we treat the queue position with index 0 as if it comes after the last position (in our case, index 9), i.e., we treat the queue as circular. Note that we use the same array declaration of the queue.

Empty queue:

Empty queues:







Implementation of operations on a circular queue:

Testing a circular queue for overflow

There are two conditions:

- **(front=0) and (rear=capacity-1)**
- **front=rear+1**

If any of these two conditions is satisfied, it means that **circular queue is full**.

The <u>enqueue</u> Operation on a Circular Queue

There are **three scenarios** which need to be considered, **assuming that the queue is not full**:

1. If the **queue is empty**, then the value of the **front** and the **rear** variable will be **-1** (i.e., the sentinel value), **then** both **front** and **rear** are set to **0**.
2. 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**.
3. If the queue is **not full** and the value of the rear variable is **equal to capacity -1** then rear is set to **0**.

The dequeue Operation on a Circular Queue

Again, there are **three possibilities**:

1. If there was only **one element** in the circular queue, then after the **dequeue** operation the queue will **become empty**. This state of the circular queue is reflected by setting the **front** and **rear** variables to **-1**.
2. If the value of the **front** variable is equal to **CAPACITY-1**, then set **front** variable to **0**.
3. If neither of the above conditions hold, then the **front** variable is **incremented**