Al-Mustaqbal University
College of Healthcare and Medical Techniques
Intelligent Medical System Department

جامعة المستقبل
AL MUSTAQBAL UNIVERSITY

كلية التقنيات الطبية والصحية
قسم الانظمة الطبية الذكية

Subject: Data Structure

Class: Second

Lecturer: Asst. Prof. 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.

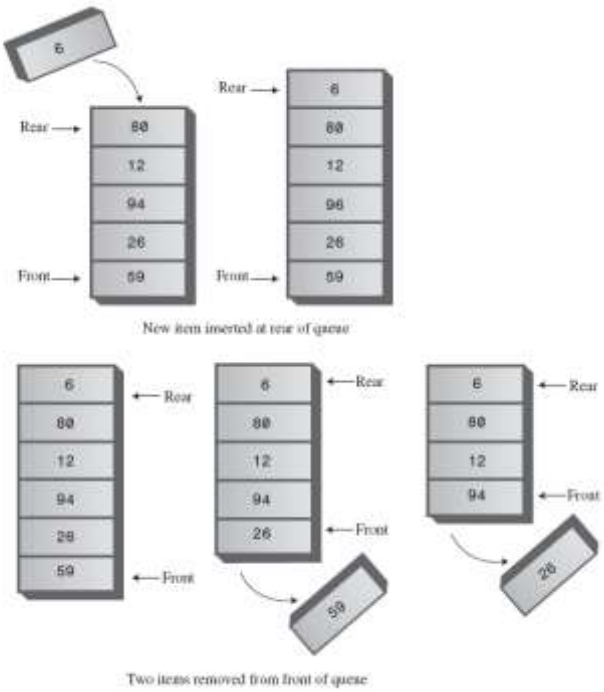


Figure 5: Operations of the Queue

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: ITEM inserted
   [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

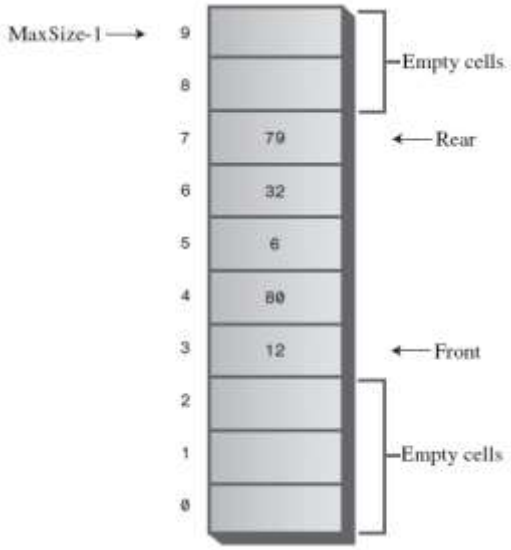


Figure 6: A Queue with some items removed

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.

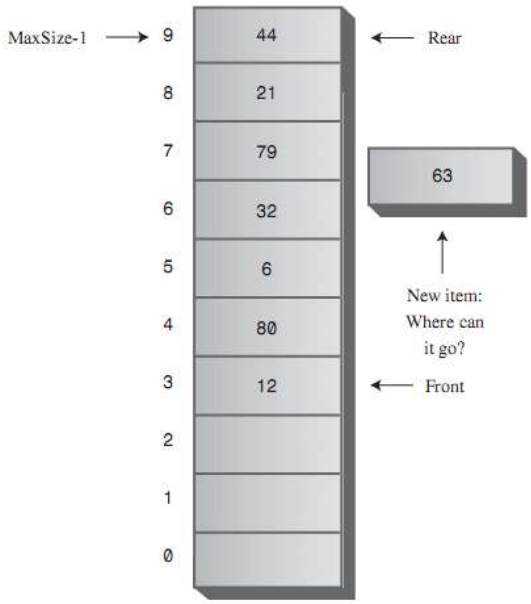


Figure 7. Rear arrow at the end of the array.

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.

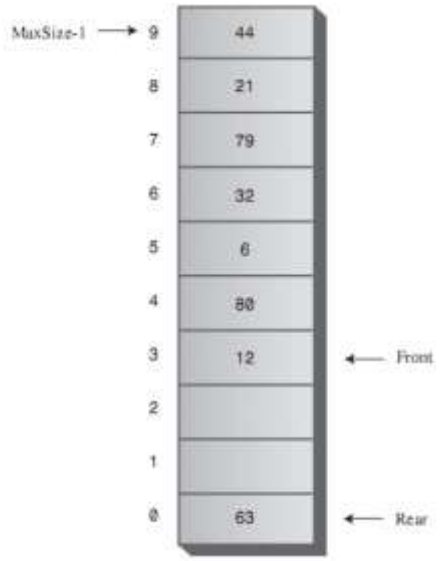


Figure 8. Rear arrow wraps around

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.,

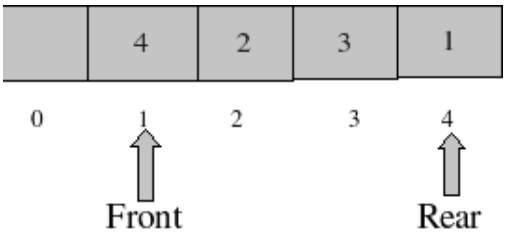
- **Insertion** can be **made only at the end** and
- **Deletion can be made only at the front**.

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

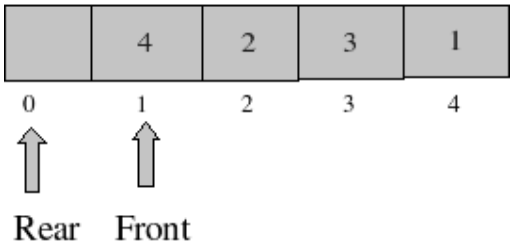
A circular queue is a Queue but a particular implementation of a queue. It is very efficient. It is also quite useful in low level code, because insertion and deletion are totally independant, which means that you don't have to worry about an interrupt handler trying to do an insertion at the same time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

Linear queue:



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

Circular queue:

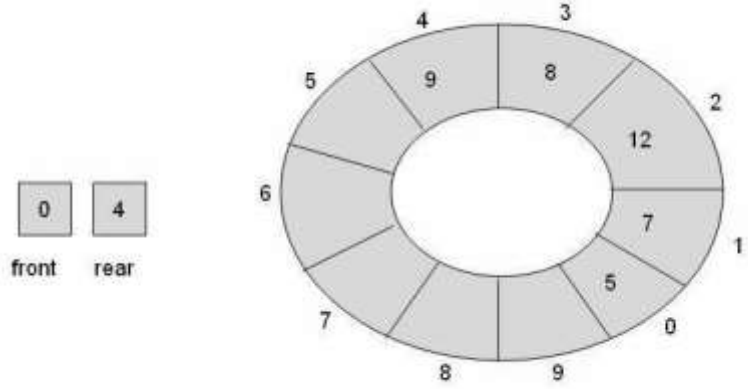


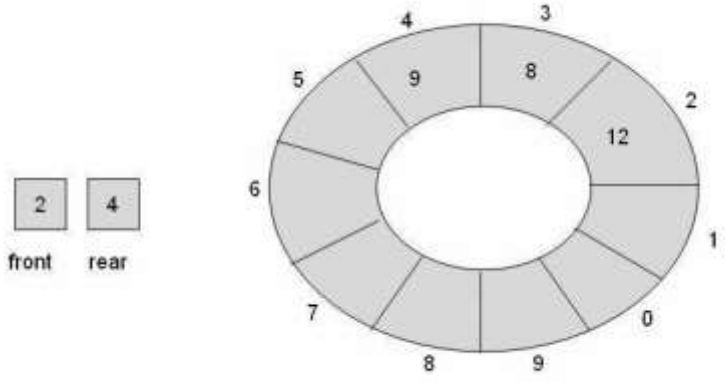
In a circular queue, after rear reaches the **end of the queue**, it can be **reset to zero**. This helps in refilling the empty spaces in between.

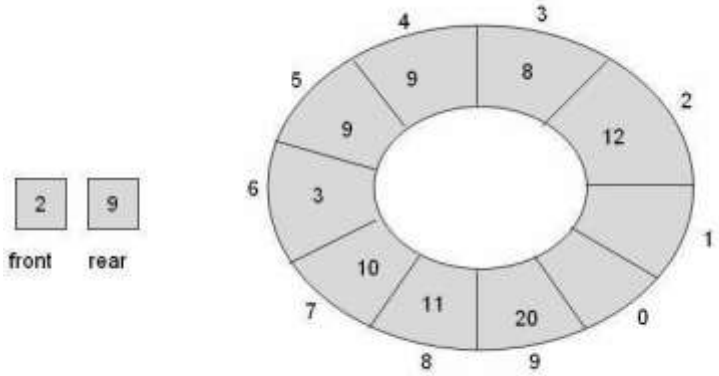
The difficulty of managing front and rear in an array-based non-circular queue can be overcome if we treat the queue position with index 0 as if it comes after the last position (in our case, index 9), i.e., we treat the queue as circular. Note that we use the same array declaration of the queue.

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**

Priority Queues

Like an ordinary queue, a **priority queue has a front and a rear**, and **items are removed from the front**. However, in a priority queue,

- items are **ordered by key value** so that the item with the lowest key (or in some implementations the highest key) is always at the **front**.
- Items are inserted in the proper position to **maintain the order**.

A **priority queue** consists of entries, each of which contains a key called the ***priority*** of the entry. A priority queue has only two operations other than the usual creation, clearing, size, full, and empty operations:

- Insert an entry.
- Remove the entry having the largest (or smallest) key.

If entries have equal keys, then any entry with the largest key may be removed first.

Applications In a time-sharing computer system, for example, a large number of tasks may be waiting for the CPU. Some of these tasks have higher priority than others. Hence the set of tasks waiting for the CPU forms a priority queue. Other applications of priority queues include simulations of time-dependent events (like the airport simulation) and solution of sparse systems of linear equations by row reduction.

PRIORITY QUEUE CONCEPT

A priority queue is best understood in comparison with a stack and a queue. To see this, imagine a supermarket checkout, where customers, each with a certain number of items in the shopping cart, arrive at the checkout counter:

ItemsInCart Customer

6	Mary	//last to arrive
12	Joe	
4	Jill	
9	Pete	
15	Stacy	
7	Bev	//first to arrive

CHECKOUT COUNTER

Suppose also that the entries <6, Mary>, <12, Joe> … <7, Bev> are placed in a data container, to reflect order of arrival, with <7, Bev> first in, <15, Stacy> next in, and so on, and <6, Mary> in last.

If the cashier serves the customers in order of arrival, that is, <7, Bev> first and <6, Mary>last, we have a conventional queue, i.e. First In, First Out, or **FIFO**.

If the cashier serves the customers starting with entries <6, Mary>, and ending with <7, Bev>, we have a stack, i.e. Last In, First Out, or **LIFO**.

If the cashier serves the customers with entries in the **order** <4, Jill>, <6, Mary>, <7, Bev>, … ending with <15, Stacy>, that is, service in order of lowest number of shopping items, we have a **priority queue**, i.e. The entry inserted with the **lowest priority key**, no matter when inserted, is the first entry out.
In this example, the first data item of each entry, the number of shopping items, serves as the **priority key**.
Notice the technical term entry. A priority queue consists of a **set of entries** into the queue, each entry consisting of a **priority key and a value**.

Applications

- Scheduling jobs on a workstation holds jobs to be performed and their priorities. When a job is finished or interrupted, highest-priority job is chosen using Extract-Max. New jobs can be added using Insert function.
- Operating System Design – resource allocation
- Data Compression -Huffman algorithm
- Discrete Event simulation

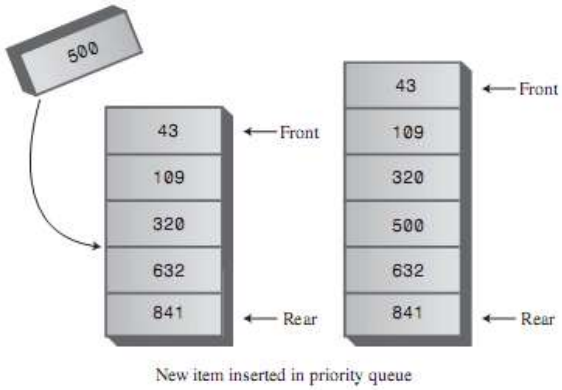
(1) Insertion of time-tagged events (time represents a priority of an event -- low time means high priority)
(2) Removal of the event with the smallest time tag

Implementation

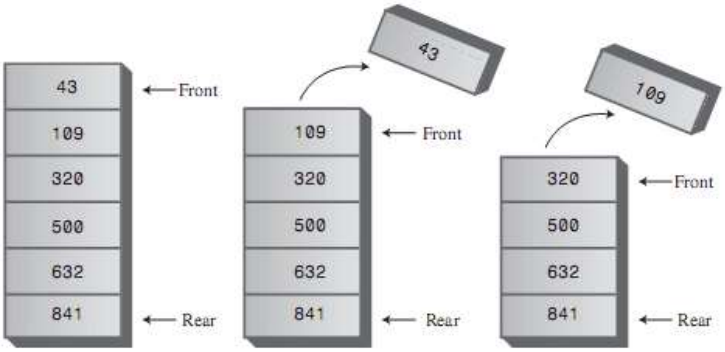
- Linked Lists
- Using a binary Heap – a special binary tree with heap property

We show **Front** and **Rear** arrows to provide a comparison with an ordinary queue, but they're not really necessary. The algorithms know that the **front** of the queue is always at the **top** of the array

at **nItems-1**, and they insert items in order, **not at the rear**. Figure below shows the operation of the PriorityQ class methods.



New item inserted in priority queue



Two items removed from front of priority queue

Key Comparison Method

Normally the entry with the **highest priority** has the **lowest priority key value**, and is extracted from the priority queue first.

That means that we need a way to **compare the key values**, so that we can say if the key of one entry is greater or less than the key of another entry, and which key has the lowest value and which the highest value.

The key comparison method may be very simple, based on integer values, as in the case of number of shopping items above.

The Priority Queue ADT

A priority queue ADT will be implemented as a container of some kind that can support the methods below.

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 one with the highest priority.

`empty`

Check whether the queue is empty.

Sorting With a Priority Queue

We can look at a priority queue as a black box. You can put entries into it, using insert(), in any key order, take out a few entries, using removeMin(), put in some more and so on, as if using a stack. But no matter how many we put in and take out, removeMin() always delivers the entry in the queue with the lowest key value. **This is obviously useful for sorting.**

Suppose we want to **sort a set** of entries (each made up of a key and a value) in ascending order:

Step 1. Use insert() to insert, in any order, all the entries into the priority queue.
Step 2. Use removeMin() to extract the entries from the queue, and print them, or place them in an array. The entries are now sorted.

Class PriorityQueue

The **insert()** method checks whether there are any items; if not, it inserts one at index **0**. **Otherwise**, it starts at the **top** of the array and shifts existing items upward until it finds the place where the **new item** should go. Then it inserts the item and **increments nItems**. Note that if there's any chance the priority queue is full, you should check for this possibility with **isFull()** before using insert().
The front and rear fields aren't necessary as they were in the Queue class because, as we noted, **front** is always at **nItems-1** and **rear** is always at **0**.
The **remove()** method is simplicity itself: It **decrements nItems** and returns the item from the **top of the array**. The **isEmpty()** and **isFull()** methods check if **nItems** is **0** or **maxSize**, respectively

```
class PriorityQueue:
    def __init__(self):
        self.queArray[] =
        self.nItems = 0

    def insert(self, item):
        if self.nItems == 0:
            self.queArray.append(item)
        else:
            j = self.nItems - 1
            while j >= 0:
```

```
            if item > self.queArray[j]:
                self.queArray[j + 1] = self.queArray[j]
            else:
                break
            j -= 1
        self.queArray[j + 1] = item
    self.nItems += 1

def remove(self):
    if self.nItems > 0:
        self.nItems -= 1
        return self.queArray.pop(self.nItems)
    else:
        raise IndexError("Priority queue is empty")

#Example usage of PriorityQueue
pq = PriorityQueue()
pq.insert(3)
pq.insert(1)
pq.insert(4)
pq.insert(2)

print("Removed items in priority order:")
while pq.nItems > 0:
    print(pq.remove())
```