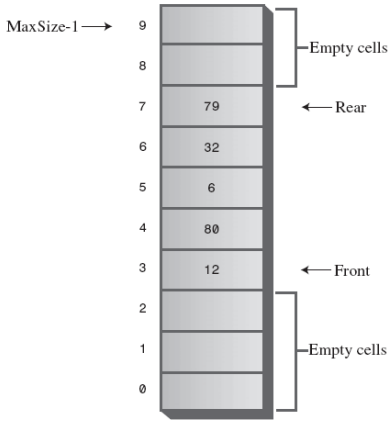
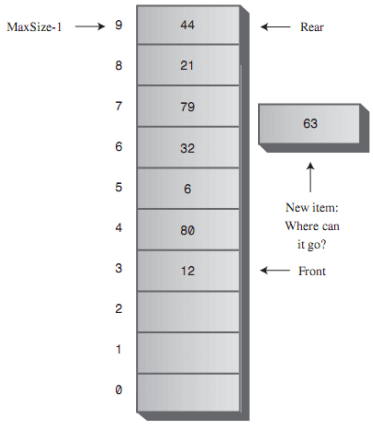
Circular and Priority Queue

Circular Queue

The trouble of linear queue 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, you still can't insert a new item because Rear can't go any further*.



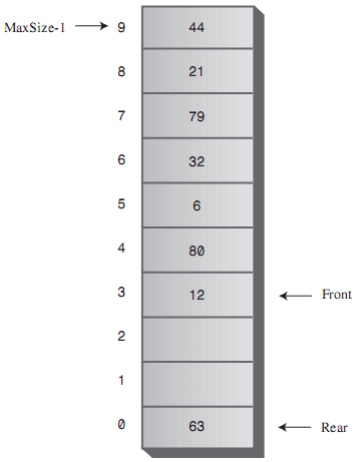
A Queue with some items removed



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.

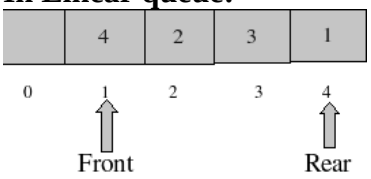
Now to insert another item. You'll see the Rear arrow wrap around from index 9 to index 0; the new item will be inserted there. This situation is shown in below Figure.



Rear arrow wraps around

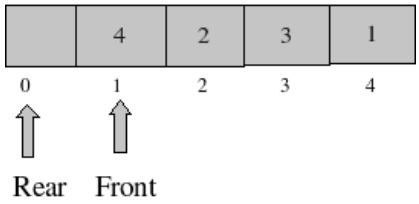
A circular queue is a Queue but a particular implementation of a queue. It is very efficient. It is also quite useful in low level code, because insertion and deletion are totally independant, which means that you don't have to worry about an interrupt handler trying to do an insertion at the same time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

In Linear queue:



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

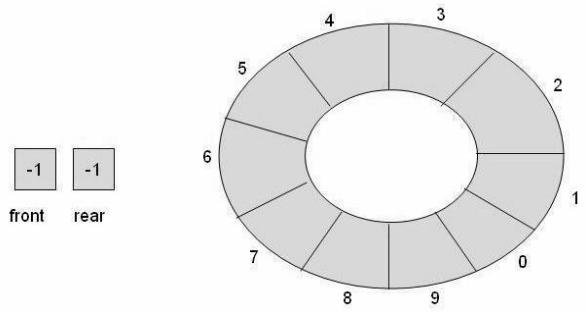
In Circular queue:



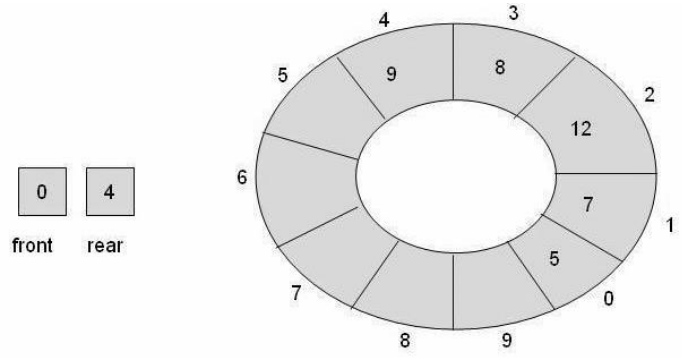
- **In a circular queue**, after rear reaches the **end of the queue**, it can be **reset to zero**. This helps in refilling the empty spaces in between.

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.

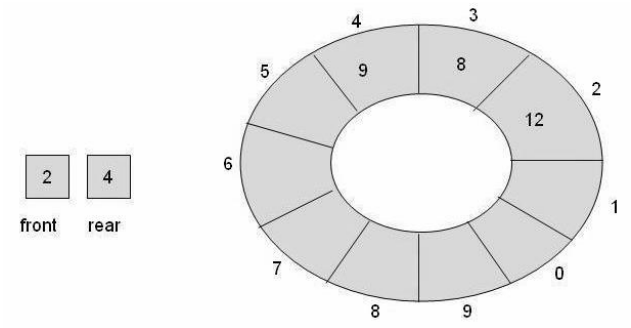
Empty queue:



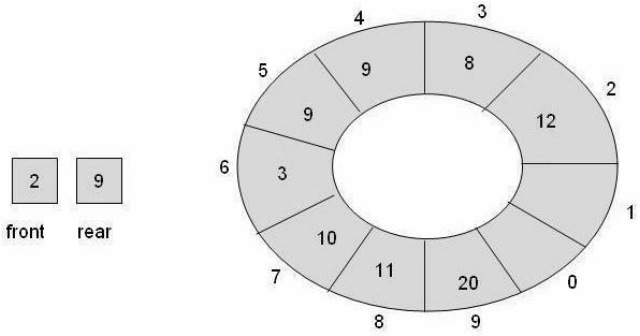
Enqueuing five items:



Dequeuing two items:



Enqueuing five items:



Implementation of operations on a circular queue:

❖ *Testing a circular queue for overflow*

There are two conditions:

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

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

❖ *The Enqueue 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*

Also, 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 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.

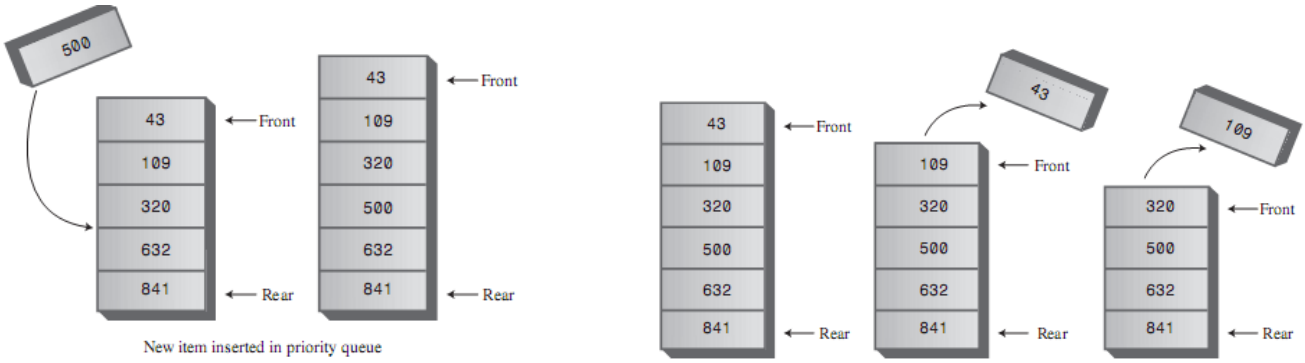
In this example, the first data item of each entry, the number of shopping items, serves as the **priority key**.

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**.

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 **front** of the queue is always at the **top** of the array at **n-1**, and they insert items in order, **not at the rear**. Figure below shows the operation of the PriorityQ.



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.

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.

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

- 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.