Data Structures and Algorithms

Mr. Tahir Iqbal

tahir.iqbal@bahria.edu.pk

Lecture 08: Queues

Queues

- A **Queue** is a special kind of list, where items are inserted at one end (**the rear**) And deleted at the other end (**the front**).

- Accessing the elements of queues follows a First In,First Out (FIFO) order.
- Example
  - Like customers standing in a check-out line in a store, the first customer in is the first customer served.

Common Operations on Queues

- **MAKENULL:**
- **FRONT*(Q*):** Returns the first element on Queue Q.
- **ENQUEUE(*x*,*Q*):** Inserts element x at the end of Queue Q.
- **DEQUEUE(*Q*):** Deletes the first element of *Q.*
- **ISEMPTY(*Q*):** Returns true if and only if Q is an empty queue.
- **ISFULL(Q):** Returns true if and only if Q is full.

Enqueue and Dequeue

- Primary queue operations: Enqueue and Dequeue

- **Enqueue** – insert an element at the rear of the queue.
- **Dequeue** – remove an element from the front of the queue.

Applications of Queues

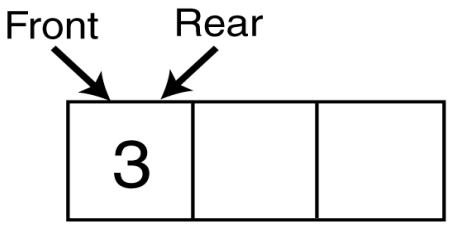
- Operating system
  - multi-user/multitasking environments, where several users or task may be requesting the same resource simultaneously.
- Printer SPOOL
- Communication Software
  - queues to hold *information* received over networks and dial up connections. (Information can be transmitted faster than it can be processed, so is placed in a queue waiting to be processed)

Queues Implementations

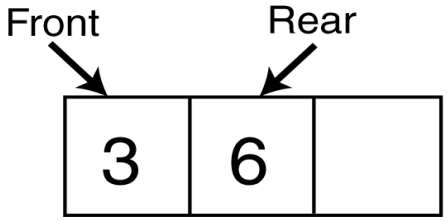
- Static
  - Queue is implemented by an array, and size of queue remains fix
- Dynamic
  - A **queue** can be **implemented** as a **linked list**, and *expand* or *shrink* with each *enqueue* or *dequeue* operation.

Static Implementation of Queues

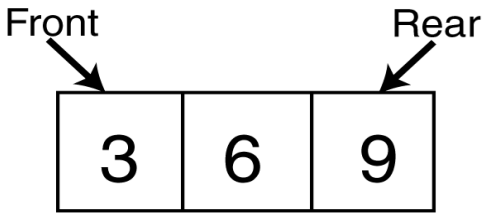
Enqueue(3);



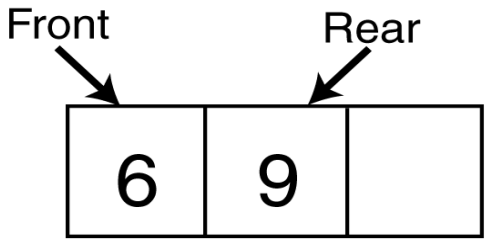
Enqueue(6);



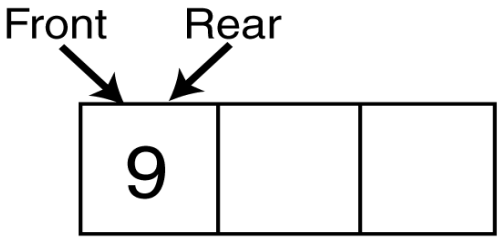
Enqueue(9);



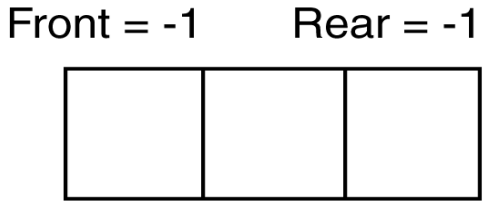
Dequeue();



Dequeue();

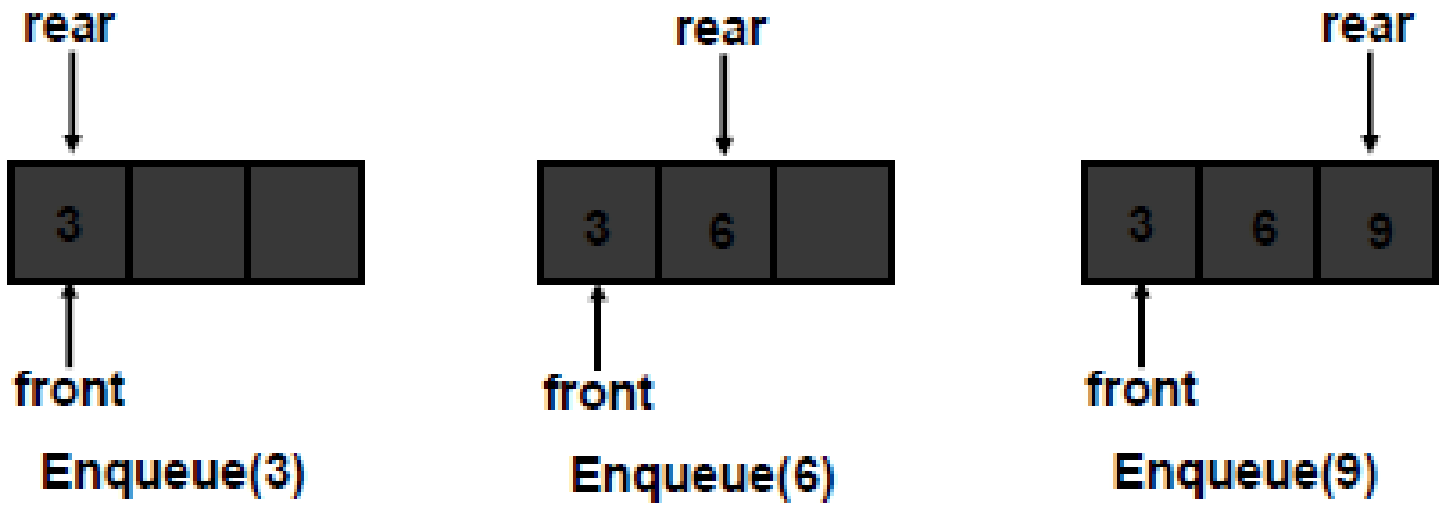


Dequeue();



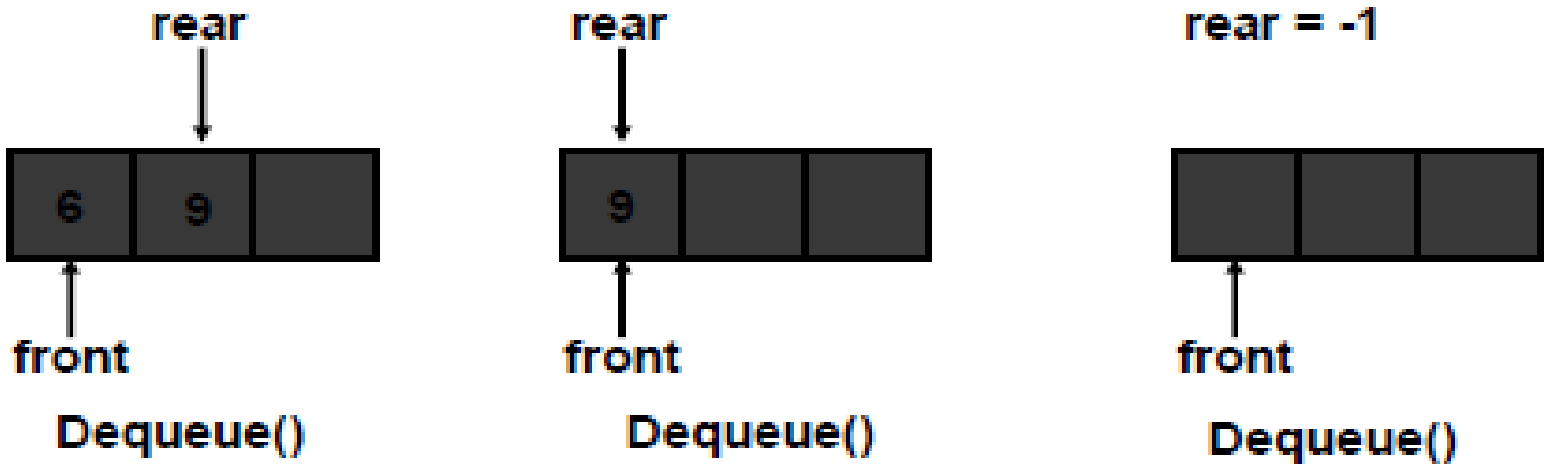
Static Implementation of Queue

- Static implementation is done using arrays
- In this implementation, we should know the exact number of elements to be stored in the queue.
- When enqueuing, the front index is always fixed and the rear index moves forward in the array.



Static Implementation of Queue

- When dequeuing, the front index is fixed, and the element at the front of the queue is removed. Move all the elements after it by one position. (Inefficient!!!)



Static Implementation of Queue

- **A better way**
  - When an item is enqueued, the rear index moves forward.
  - When an item is dequeued, the front index also moves forward by one element
- **Example:**

X = occupied, and O = empty

- (front) XXXXOOOOO (rear)
- OXXXXOOOO (after 1 dequeue, and 1 enqueue)
- OOXXXXXOO (after another dequeue, and 2 enqueues)
- OOOOXXXXX (after 2 more dequeues, and 2 enqueues)

- **The problem here is that the rear index cannot move beyond the last element in the array.**

Queue using Arrays

```
const int length=10
struct queue
{
  int items[length];
  int front, rear;
}

struct queue q;
q.front=q.rear=-1;
```

IsEmpty() and **dequeue** ()

```
bool isempty(queue *q)
{
   if (q->rear== -1)
      return true;
   else
      return false;
}

int dequeue (queue* q)
{
   int x=q->items[q->front++];
   return x;
}
```

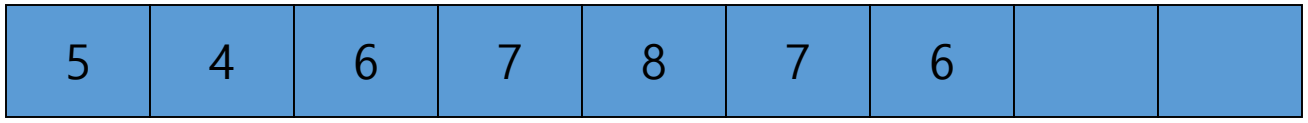
```
int dequeue (queue* q)
{
   if (q->rear== -1)
      return -1;

   int x=q->items[q->front++];
   return x;
}
```

Isfull() and Enqueue

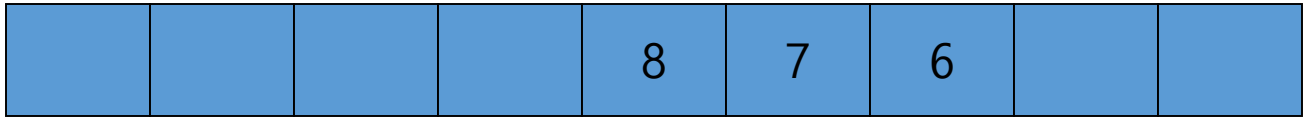
```
bool isfull(struct queue *q)
{
   if (q->rear +1 == length)
            return true;
   else
            return false;
}

void enqueue(struct queue *q, int x)
   {
            q.items[++q->rear]=x;
   }
```

Still What Wrong with this Queue ? … Hint! Front and rear …

Array Implementation



Front=0
Rear=6



Front=4
Rear=6

Front=5
Rear=8

How can we insert more elements? Rear index can not move beyond the last element….

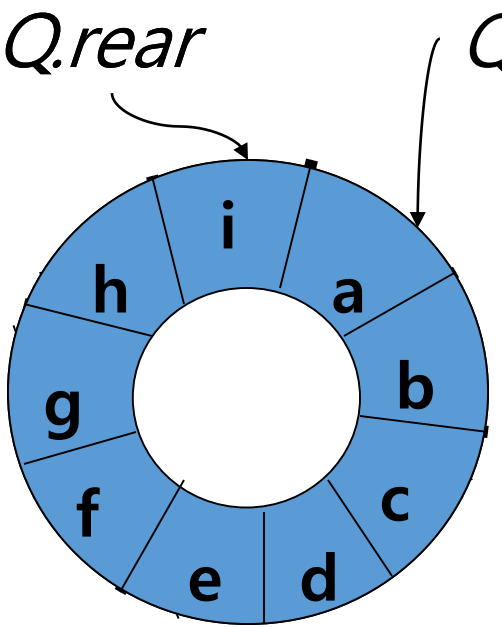
Circular Queue

Static Implementation of Queue

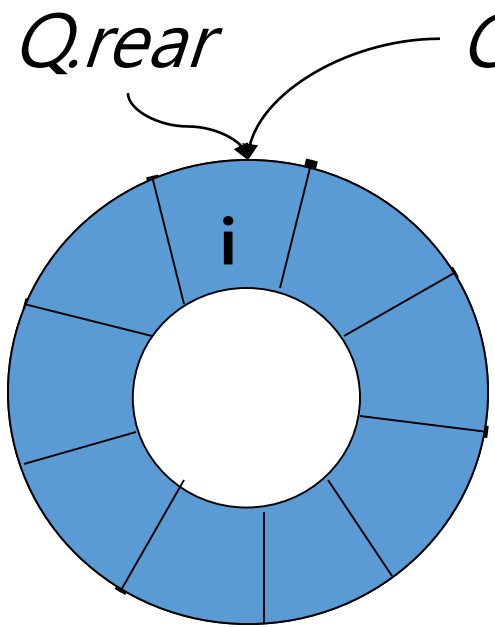
- To overcome the above limitation, we can use **circular array implementation of queues.**
- In this implementation, first position follows the last.

- **When an element moves past the end of a circular array, it wraps around to the beginning, e.g**
  - OOOOO7963 ->4OOOO7963 (after Enqueue(4))
  - After Enqueue(4), the rear index moves from 3 to 4.

Circular Queue



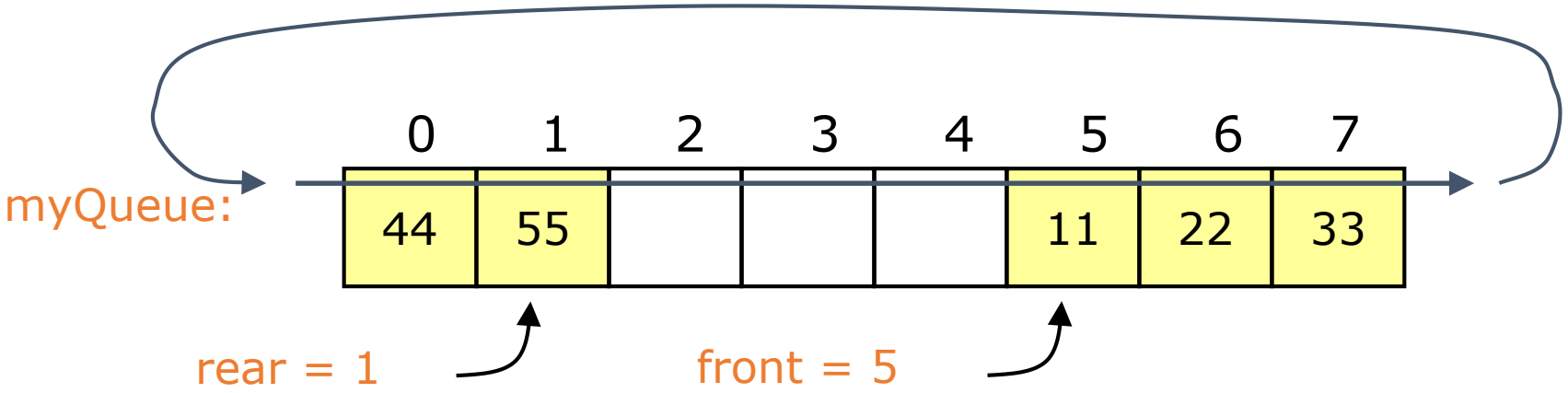
A Completely Filled Queue



A Queue with Only 1 Element

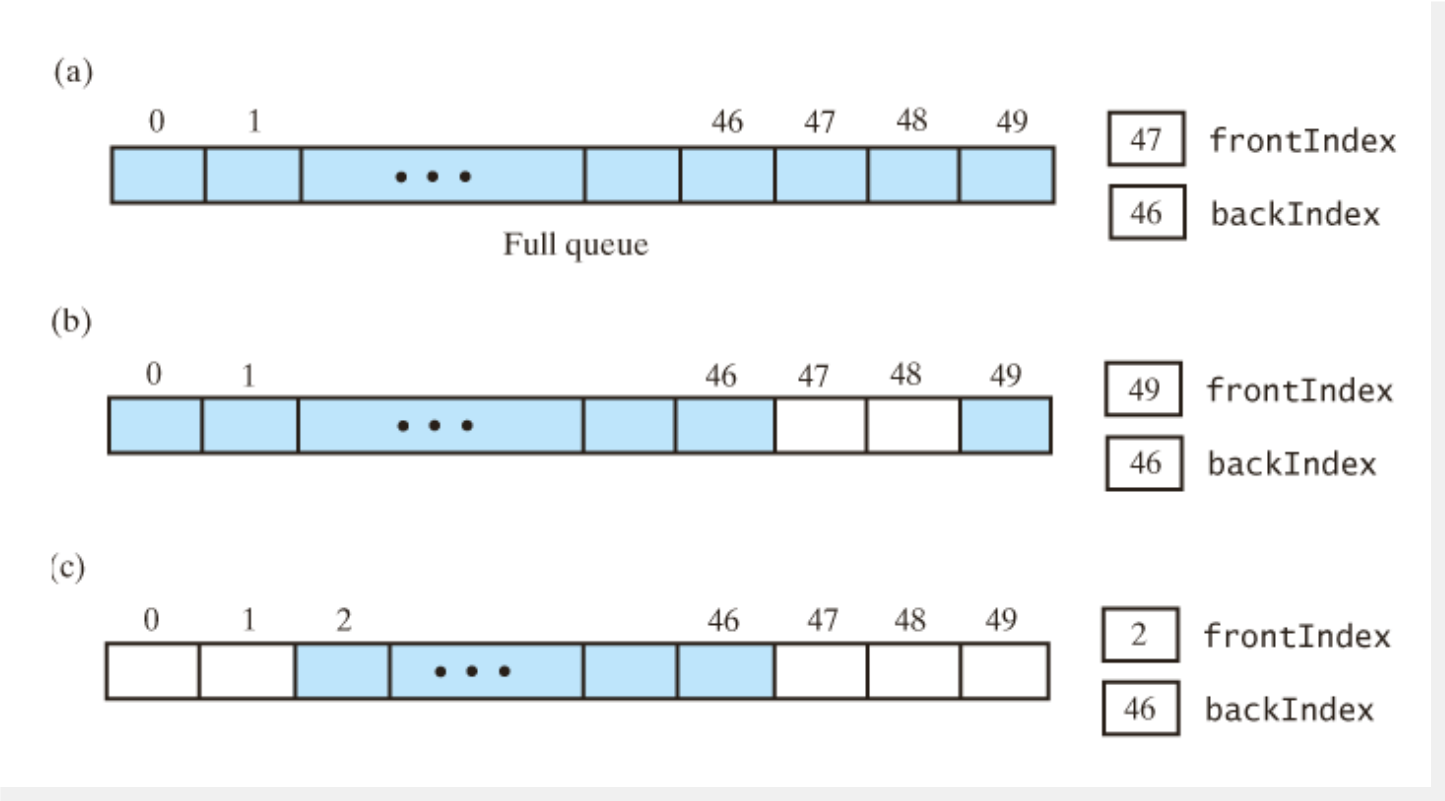
Circular arrays

- We can treat the array holding the queue elements as circular (joined at the ends)



- Elements were added to this queue in the order 11, 22, 33, 44, 55, and will be removed in the same order
- Use: `front = (front + 1) % myQueue.length;`
  and: `rear = (rear + 1) % myQueue.length;`

A Circular Array



A circular array that represents a queue:
(a) when full; (b) after removing 2 entries; (c) after removing 3 more entries;

Enqueue Algorithm

- Enqueue Procedure:
  - Enqueue(Queue, MaxSize, Front, Rear, Item)
    (This procedure inserts an ITEM into a queue)

1. If Front = 1 and Rear = MaxSize, or if Front = Rear+1, then:
   Print OVERFLOW, and Return.
2. [Find new value of Rear]
   If Front = NULL. Then, [Queue initially empty]
   Set Front := 1 and Rear := 1.
   Else if Rear = MaxSize, then:
   Set Rear := 1.
   Else:
   Set Rear := Rear + 1.
3. Set Queue[Rear] := Item.
4. Return.

Dequeue Algorithm

- Dequeue Procedure:
  - Dequeue(Queue, MaxSize, Front, Rear, Item)

    (This procedure deletes an element from the queue and assigns it to ITEM)

1. If Front = NULL, then:

   Print UNDERFLOW, and Return.
2. Set Item := Queue[Front].
3. [Find new value of Front]

   If Front = Rear. Then, [Queue has only one element to start]

   Set Front := NULL and Rear := NULL.

   Else if Front = MaxSize, then:

   Set Front := 1.

   Else:

   Set Front := Front + 1.
4. Return.

Using circular queue

- Allow rear to wrap around the array.

```
if(rear == queueSize-1)
                rear = 0;
else
                rear++;
```

- Or use module arithmetic
  rear = (rear + 1) % queueSize;

Front=5
Rear=8

Enqueue 39 **Rear=(Rear+1) mod Queue Size = (8+1) mod 9 = 0**

Front=5
Rear=0

Implementation

```
Struct queue
{
   int items[length];
   int front, rear;
};
Struct queue q;
q.front=q.rear= –1;
```

```
bool IsEmpty(struct queue *pq)
{
   return(pq->front==pq->rear);
}
```

```
int dequeue (struct queue *pq)
{
   if (IsEmpty(pq)){
            cout<<“queue underflow”
            return 0;
   }
temp = pq->items[pq->front];
   If (pq->front==length-1)
            pq->front=0;
   else
            (pq->front)++;
Return(temp);
}
```

Implementation cont.

```
int enqueue(struct queue *pq, int x)
{
    //make room for new element
    If (pq->rear==length-1)
            pq->rear=0;
    else
            (pq->rear)++;

    //Check for overflow
    If (pq->rear==pq->front)
    {
            cout<<“queue overflow”;
            exit(1);
    }
    pq->items[pq->rear]=x;
            return;
}
```

How to determine empty and full Queues?

- It can be somewhat tricky
- Number of approaches
  - A counter indicating number of values in the queue can be used (We will use this approach)

Implementation

```
Struct queue
{
   int items[length];
   int front, rear, numItems;
};
Struct queue q;
q.front=q.rear=  –1;
numItems=0;
```

- ```
  bool isEmpty()
  {
       if (numItems)
              return false;
       else
              return true;
  }
  ```

- ```
  bool isFull()
  {
       if (numItems==length)
              return false;
       else
              return true;
  }
  ```

Priority Queues

Definition

- A priority queue is a data structure in which prioritized insertion and deletion operations on elements can be performed according to their priority values.

- There are two types of priority queues:
  - Ascending Priority queue, and a
  - Descending Priority queue

Types of Priority Queue

- **Ascending Priority queue**: a collection of items into which items can be inserted ***randomly*** but only the ***smallest*** item can be removed
- If "**A-Priority-Q**" is an ascending priority queue then
  - Enqueue() will insert item 'x' into **A-Priority-Q**,
  - minDequeue() will remove the minimum item from **A-Priority-Q** and return its value

Types of Priority Queue

- **Descending Priority queue**: a collection of items into which items can be inserted ***randomly*** but only the ***largest*** item can be removed
- If "**D-Priority-Q**" is a descending priority queue then
  - Enqueue() will insert item x into **D-Priority-Q**,
  - maxDequeue( ) will remove the maximum item from **D-Priority-Q** and return its value

Generally

- In both the above types, if elements with equal priority are present, the FIFO technique is applied.

- Both types of priority queues are similar in a way that both of them remove and return the element with the highest **“Priority”** when the function remove() is called.
  - For an ascending priority queue item with smallest value has maximum “priority”
  - For a descending priority queue item with highest value has maximum “priority”

- This implies that we must have criteria for a priority queue to determine the Priority of its constituent elements.
- the elements of a priority queue can be numbers, characters or any complex structures such as phone book entries, events in a simulation

Priority Queue Issues

- In what manner should the items be inserted in a priority queue
  - Ordered (so that retrieval is simple, but insertion will become complex)
  - Arbitrary (insertion is simple but retrieval will require elaborate search mechanism)
- Retrieval
  - In case of un-ordered priority queue, what if minimum number is to be removed from an ascending queue of n elements (n number of comparisons)
- In what manner should the queue be maintained when an item is removed from it
  - Emptied location is kept blank (how to recognize a blank location ??)
  - Remaining items are shifted