Queues

A queue is a FIFO sequence. Addition takes place only at the tail, and removal takes place only at the head. Like a stack, queue can contain any object. The basic operations for queues are otherwise identical to those for stacks, except also for the standard names of “enqueue()” and “dequeue()” instead of “push” and “pop”.

The basic operations are:
- **enqueue(x)**: add an item at the tail
- **dequeue**: remove the item at the head
- **peek**: return the item at the head (without removing it)
- **size**: return the number of items in the queue
- **isEmpty**: return whether the queue has no items

Queue Interface

```java
public interface Queue {
    public int size();
    public boolean isEmpty(); // Is the queue empty?
    public Object front() throws EmptyQueueException; // or peek() - Examine element at front of queue
    public void enqueue(Object o); // Insert element at rear of queue
    public Object dequeue() throws EmptyQueueException; // Remove element from front of queue
}
```

Each of the above basic operations run at constant time $O(1)$ - independent of the number of items in the queue

Applications of queues?
• Line of cars at a light
• Line of people at a café
• Printer buffer

Implementation of queues
1. No built-in class exactly like a Queue, but some that are very close
2. Linked List—not as much fun, unless you keep a tail pointer.
3. Array—Use it as a circular buffer.

Array-based Queue
We can use an array of fixed capacity to store items as a queue. Just keep track of two indices:
• enqueue new items to index rear
• dequeue items from index front

What problems can that lead to?
• Rear may reach the last array index
• Move each element forward one array cell every time we perform a dequeue operation (takes \(O(n)\) where \(n\) is the current number of objects in the queue)

Solution:
To avoid moving objects once they are placed in the array, we define:
• An index to the array cell that holds the first element of the queue
• An index to the next available cell

Normal case:
• 1) insert puts the new value in the array and increments rear index.
• 2) remove takes an element out of the array and increments the front index.

How do we know if the queue is empty? [If front == rear]

What do we do when the rear pointer goes past the end of the array? There are a couple of choices:
a) allocate more space (double the size)
b) wrap around, using modular arithmetic. Assume the length of the array is \(m\).

\[
\begin{align*}
\text{front}++ & \quad \text{gets replaced with front} = (\text{front} + 1) \mod m; \\
\text{rear}++ & \quad \text{gets replaced with rear} = (\text{rear} + 1) \mod m;
\end{align*}
\]

This is called a circular array.

• Array
• Circular Array

Illustration
• Array

```
0  1  2  3  4  5

enqueue("a")

0  1  2  3  4  5
    a

enqueue("e")

0  1  2  3  4  5
    a  b  c  d  e

dequeue() returns "a"

0  1  2  3  4  5
    a  b  c  d  e

dequeue() returns "b"

0  1  2  3  4  5
    a  b  c  d  e

enqueue("f")

0  1  2  3  4  5
    a  b  c  d  e  f
```

• Circular Array
But it is still possible for the buffer to fill. How can we tell? [If the rear wraps around and catches up with the front. if (rear + 1) % n == front, we’re full.] Or we can just keep track of the count and the capacity.

If the array is full, we can also reallocate the array and copy the queue data into the new array starting at position 0 again.

**Queue Implementation using Linked List**
For the list implementation, insert new things at the end and take from the beginning.

Adding an item to the queue first stores a temporary reference to the current ‘last’ element of the list. The structure is then able to go about storing the new node instance referenced by the ‘last’ variable. If previously the collection was empty we set the ‘first’ element to this new item as well as the consistent last reference. However, in the case that there are already items present, we simply set the previous last elements ‘next’ reference to this new node.
Removing (dequeuing) elements from the collection is also a trivial task, simply returning the node referenced from the ‘first’ variable. This reference is then updated to the returned nodes ‘next’ reference, and a simple check to clear the last reference if the collection is now empty takes place.

Normal cases:
- a) insert at end, update rear
- b) remove from beginning, update front
But you also have to deal with special cases:

a) inserting an element in an empty list (front = rear = new)
b) removing the last element from a list (front = rear = null)