CS206

Queues
Queues
Queueing Theory

Agner Krarup Erlang
Queues

- Insertions and deletions are First In First Out – FIFO
  - Insert at the back
  - Delete from the front
Queue Interface

- null is returned from peek() and poll() when queue is empty
- add(), remove(), element() are identical to offer(), poll(), peek() but for throw.

```java
public interface QueueIntf<Q> {
    boolean isEmpty();
    int size();
    boolean add(Q q)
        throws IllegalStateException;
    Q remove()
        throws NoSuchElementException;
    Q element()
        throws NoSuchElementException;
    boolean offer(Q q);
    Q poll();
    Q peek();
}
```
## Example

<table>
<thead>
<tr>
<th>Operation</th>
<th>output</th>
<th>Queue Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>offer(5)</td>
<td>TRUE</td>
<td>{5}</td>
</tr>
<tr>
<td>offer(3)</td>
<td>TRUE</td>
<td>{5,3}</td>
</tr>
<tr>
<td>poll()</td>
<td>5</td>
<td>{3}</td>
</tr>
<tr>
<td>offer(7)</td>
<td>TRUE</td>
<td>{3, 7}</td>
</tr>
<tr>
<td>poll()</td>
<td>3</td>
<td>{3, 7}</td>
</tr>
<tr>
<td>peek()</td>
<td>7</td>
<td>{7}</td>
</tr>
<tr>
<td>poll()</td>
<td>7</td>
<td>{}</td>
</tr>
<tr>
<td>poll()</td>
<td>null</td>
<td>{}</td>
</tr>
</tbody>
</table>
Array-based Queue

- An array of size $n$ in a circular fashion
- Two ints to track front and size
  - $f$: index of the front element
  - $co$: number of stored elements

<table>
<thead>
<tr>
<th>Q</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>$f$</th>
<th>$r$</th>
</tr>
</thead>
<tbody>
<tr>
<td>normal configuration</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Q</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>$r$</th>
<th>$f$</th>
</tr>
</thead>
<tbody>
<tr>
<td>wrapped-around configuration</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Circular Array and Queue

- When the queue has fewer than $n$ elements, location $r = (f + c_0) \mod n$ is the first empty slot past the rear of the queue.

**normal configuration**

```
Q
0 1 2  f   r
```

**wrapped-around configuration**

```
Q
0 1 2  r   f
```
public class ArrayQueue<Q> implements QueueIntf<Q> {

    /** the default capacity for the backing array */
    private static final int CAPACITY = 40;

    /** The array in which the queue data is stored */
    private Q[] backingArray;

    /** The array location of the head of the queue */
    private int count;

    /** The array location of the end of the queue */
    private int frontLoc;

    /**
     * Create an array backed queue with the default capacity.
     */
    public ArrayQueue() {
        this(CAPACITY);
    }

    /**
     * Create an array backed queue with the given capacity
     * @param qSize the capacity for the queue
     */
    public ArrayQueue(int qSize) {
        count = 0;
        frontLoc = 0;
        backingArray = (Q[]) new Object[qSize];
    }
}
offer(), add()

- must handle case if the array becomes full
  - Limitation of the array-based implementation
    - offer returns false
    - add throws exception
Performance and Limitations for array-based Queue

- **Performance**
  - let $n$ be the number of objects in the queue
  - The space used is $O(n)$
  - Each operation runs in time $O(1)$

- **Limitations**
  - Max size is limited and cannot be changed
  - Pushing onto a full stack queue in an exception
Simulating a Bank