Queues
The Queue ADT

- The Queue ADT stores arbitrary objects
- Insertions and deletions follow the first-in first-out scheme
- Insertions are at the rear of the queue and removals are at the front of the queue
- Main queue operations:
  - `enqueue(object)`: inserts an element at the end of the queue
  - `dequeue()`: removes and returns the element at the front of the queue
- Auxiliary queue operations:
  - `front()`: returns the element at the front without removing it
  - `size()`: returns the number of elements stored
  - `isEmpty()`: indicates whether no elements are stored
- Exceptions
  - Attempting the execution of dequeue or front on an empty queue throws an EmptyQueueException
### Example

<table>
<thead>
<tr>
<th>Operation</th>
<th>Output</th>
<th>Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>enqueue(5)</td>
<td>–</td>
<td>(5)</td>
</tr>
<tr>
<td>enqueue(3)</td>
<td>–</td>
<td>(5, 3)</td>
</tr>
<tr>
<td>dequeue()</td>
<td>5</td>
<td>(3)</td>
</tr>
<tr>
<td>enqueue(7)</td>
<td>–</td>
<td>(3, 7)</td>
</tr>
<tr>
<td>dequeue()</td>
<td>3</td>
<td>(7)</td>
</tr>
<tr>
<td>front()</td>
<td>7</td>
<td>(7)</td>
</tr>
<tr>
<td>dequeue()</td>
<td>7</td>
<td>()</td>
</tr>
<tr>
<td>dequeue()</td>
<td>“error”</td>
<td>()</td>
</tr>
<tr>
<td>isEmpty()</td>
<td></td>
<td>true</td>
</tr>
<tr>
<td>enqueue(9)</td>
<td>–</td>
<td>(9)</td>
</tr>
<tr>
<td>enqueue(7)</td>
<td>–</td>
<td>(9, 7)</td>
</tr>
<tr>
<td>size()</td>
<td>2</td>
<td>(9, 7)</td>
</tr>
<tr>
<td>enqueue(3)</td>
<td>–</td>
<td>(9, 7, 3)</td>
</tr>
<tr>
<td>enqueue(5)</td>
<td>–</td>
<td>(9, 7, 3, 5)</td>
</tr>
<tr>
<td>dequeue()</td>
<td>9</td>
<td>(7, 3, 5)</td>
</tr>
</tbody>
</table>
Applications of Queues

- Direct applications
  - Waiting lists, bureaucracy
  - Access to shared resources (e.g., printer)
  - Multiprogramming

- Indirect applications
  - Auxiliary data structure for algorithms
  - Component of other data structures
Array-based Queue

- Use an array of size $N$ in a circular fashion.
- Two variables keep track of the front and rear:
  - $f$ index of the front element
  - $r$ index immediately past the rear element
- Array location $r$ is kept empty.
Queue Operations

- We use the modulo operator (remainder of division)

Algorithm \textit{size}()
return \((N - f + r) \mod N\)

Algorithm \textit{isEmpty}()
return \((f = r)\)
Queue Operations (cont.)

- Operation enqueue throws an exception if the array is full
- This exception is implementation-dependent

**Algorithm enqueue(o)**

if size() = $N - 1$ then
  throw FullQueueException
else
  $Q[r] \leftarrow o$
  $r \leftarrow (r + 1) \mod N$
Queue Operations (cont.)

- Operation dequeue throws an exception if the queue is empty.
- This exception is specified in the queue ADT.

Algorithm `dequeue()`:

```
if isEmpty() then
    throw EmptyQueueException
else
    o ← Q[f]
    f ← (f + 1) mod N
    return o
```

```markdown
<table>
<thead>
<tr>
<th>Q</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>f</th>
<th>r</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Q</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>r</th>
<th>f</th>
</tr>
</thead>
</table>
```

© 2010 Goodrich, Tamassia
Queue Interface in Java

- Java interface corresponding to our Queue ADT
- Requires the definition of class EmptyQueueException
- No corresponding built-in Java class

```java
public interface Queue<E> {
    public int size();
    public boolean isEmpty();
    public E front() throws EmptyQueueException;
    public void enqueue(E element);
    public E dequeue() throws EmptyQueueException;
}
```
Application: Round Robin Schedulers

- We can implement a round robin scheduler using a queue Q by repeatedly performing the following steps:
  1. \( e = Q.\text{dequeue}() \)
  2. Service element \( e \)
  3. \( Q.\text{enqueue}(e) \)