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:**
  - `first()`: returns the element at the front without removing it.
  - `size()`: returns the number of elements stored.
  - `isEmpty()`: indicates whether no elements are stored.
- **Boundary cases:**
  - Attempting the execution of `dequeue` or `first` on an empty queue returns `null`.
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>first()</td>
<td>7</td>
<td>(7)</td>
</tr>
<tr>
<td>dequeue()</td>
<td>7</td>
<td>()</td>
</tr>
<tr>
<td>dequeue()</td>
<td>null</td>
<td>()</td>
</tr>
<tr>
<td>isEmpty()</td>
<td>true</td>
<td>()</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 size
  - $f$: index of the front element
  - $sz$: number of stored elements
- When the queue has fewer than $N$ elements, array location $r = (f + sz) \mod N$ is the first empty slot past the rear of the queue

Queue Operations

- We use the modulo operator (remainder of division)

```
Algorithm size()
return sz
```

```
Algorithm isEmpty()
return (sz == 0)
```
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 IllegalStateException
else
    r ← (f + sz) mod N
    Q[r] ← o
    sz ← (sz + 1)
```

---

Queue Operations (cont.)

- Note that operation dequeue returns null if the queue is empty.

**Algorithm dequeue()**

```
if isEmpty() then
    return null
else
    o ← Q[f]
    f ← (f + 1) mod N
    sz ← (sz - 1)
    return o
```
Queue Interface in Java

- Java interface corresponding to our Queue ADT
- Assumes that first() and dequeue() return null if queue is empty

public interface Queue<E> {
    int size();
    boolean isEmpty();
    E first();
    void enqueue(E e);
    E dequeue();
}

Array-based Implementation

```java
/** Implementation of the queue ADT using a fixed-length array. */
public class ArrayQueue<E> implements Queue<E> {
    // instance variables
    private E[] data; // generic array used for storage
    private int f = 0; // index of the front element
    private int sz = 0; // current number of elements

    // constructors
    public ArrayQueue() {this(CAPACITY);} // constructs queue with default capacity
    public ArrayQueue(int capacity) { // constructs queue with given capacity
        data = (E[]) new Object[capacity]; // safe cast; compiler may give warning
    }

    // methods
    /** Returns the number of elements in the queue. */
    public int size() { return sz; }
    /** Tests whether the queue is empty. */
    public boolean isEmpty() { return (sz == 0); }
```
Array-based Implementation (2)

```java
21 /** Inserts an element at the rear of the queue. */
22 public void enqueue(E e) throws IllegalStateException {
23     if (sz == data.length) throw new IllegalStateException("Queue is full");
24     int avail = (f + sz) % data.length;   // use modular arithmetic
25     data[avail] = e;
26     sz++;
27 }
28
29 /** Returns, but does not remove, the first element of the queue (null if empty). */
30 public E first() {
31     if (isEmpty()) return null;
32     return data[f];
33 }
34
35 /** Removes and returns the first element of the queue (null if empty). */
36 public E dequeue() {
37     if (isEmpty()) return null;
38     E answer = data[f].
39     data[f] = null;   // dereference to help garbage collection
40     f = (f + 1) % data.length;
41     sz--;
42     return answer;
43 }
```

Comparison to java.util.Queue

- Our Queue methods and corresponding methods of java.util.Queue:

<table>
<thead>
<tr>
<th>Our Queue ADT</th>
<th>Interface java.util.Queue</th>
</tr>
</thead>
<tbody>
<tr>
<td>enqueue(e)</td>
<td>add(e)</td>
</tr>
<tr>
<td></td>
<td>offers(e)</td>
</tr>
<tr>
<td>dequeue()</td>
<td>remove()</td>
</tr>
<tr>
<td></td>
<td>poll()</td>
</tr>
<tr>
<td>first()</td>
<td>element()</td>
</tr>
<tr>
<td></td>
<td>peek()</td>
</tr>
<tr>
<td>size()</td>
<td>size()</td>
</tr>
<tr>
<td>isEmpty()</td>
<td>isEmpty()</td>
</tr>
</tbody>
</table>

© 2014 Goodrich, Tamassia, Goldwasser
Application: Round Robin Schedulers

We can implement a round robin scheduler using a queue $Q$ by repeatedly performing the following steps:

1. $e = Q.dequeue()$
2. Service element $e$
3. $Q.enqueue(e)$