Queues
Outline and Reading

- The Queue ADT
- Implementation with a circular array
- Growable array-based queue
- Queue interface in C++
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 o): inserts an element o 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
- empty(): returns a Boolean indicating whether no elements are stored

Exceptions
- Attempting the execution of dequeue or front on an empty queue throws an EmptyQueueException
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.

Normal configuration:

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

Wrapped-around configuration:

```
Q:  0  1  2  r  f
```
Queue Operations

Hint: use the modulo operator

Algorithm `size()`

```plaintext
return (N - f + r) mod N
```

[or = ((N+r)-f) mod N]

Algorithm `empty()`

```plaintext
return (f = r)
```
Queue Operations (cont.)

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

Algorithm `enqueue(o)`

```plaintext
if size() = N - 1 then
    throw FullQueueException
else
    Q[r] ← o
    r ← (r + 1) mod N
```

Diagram:
```
Q:
<p>| | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>f</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>r</td>
</tr>
</tbody>
</table>
```

```
Q:
<p>| | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>r</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>f</td>
</tr>
</tbody>
</table>
```
Queue Operations (cont.)

- Operation dequeue throws an exception if the queue is empty.

- This exception is specified in the queue ADT.

Algorithm `dequeue()`

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

```
Q: [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ]
   0 1 2  f

Q: [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ]
   0 1 2  r
   f
```
Linked-List based Queue

Will see later...
Growable Array-based Queue

- In an enqueue operation, when the array is full, instead of throwing an exception, we can replace the array with a larger one.
- Similar to what we did for an array-based stack.
- The enqueue operation has amortized running time:
  - $O(n)$ with the incremental strategy
  - $O(1)$ with the doubling strategy
Informal C++ Queue Interface

- Informal C++ interface for our Queue ADT
- Requires the definition of class EmptyQueueException
- A corresponding built-in STL class exists

```cpp
template <typename Object>
class Queue {
public:
    int size();
    bool isEmpty();
    Object& front() 
        throw(EmptyQueueException);
    void enqueue(Object o);
    Object dequeue() 
        throw(EmptyQueueException);
};
```