QUEUE

The Idea of a Queue

A queue is like a lineup in a bank: objects enter at the back, and leave from the front.

Queues are used in many kinds of software, such as:

- Operating systems.
  Queues keep track of processes waiting for a turn in the CPU.
- Simulations.
  In a Laundromat simulation, a queue might keep track of loads of laundry waiting for a dryer.
- Graphical user interfaces (GUIs).
  Queues keep track of events waiting to be handled, like multiple button clicks.

Definition of a Queue

Data: A sequence of objects. The objects are removed in the order they were inserted; this is referred to as "first in, first out", or FIFO. The first element, at the "front" of the queue, is called the "head". The last element is called the "tail".

Operations:

- **enqueue(o):** Append o to the queue.
- **head():** Return front element of the queue.
  Requires: The queue is not empty.
- **dequeue():** Remove and return the front element of the queue.
  Requires: The queue is not empty.
- **size():** Return the number of elements in the queue.

Note that this is not written in Java! The description is quite general. For example, it does not restrict:

- what can go in a queue
- what a queue is used for
- how a queue is stored
- how the operations work
- what programming language it might be built in

We have described only what the *user* of a queue needs: its "interface" or "front-end".

This is similar to the interface for a physical device, like a pop machine:

- It has buttons you can push to get pop.
- It doesn't reveal irrelevant details like how pop is stored, how change is made, etc.

A Java Interface

We can define the interface for a queue using a Java `interface`. It's like a class, but without any instance variables or method bodies.

(We'll define the instance variables and method bodies *separately*, in a class (or classes) that "implement" the interface.)

```
public interface Queue {
    /** Append o to me. */
    void enqueue(Object o);

    /** Return my front element.
        Requires: size() != 0. */
    Object head();

    /** Remove and return my front element.
        Requires: size() != 0. */
    Object dequeue();

    /** Return the number of elements in me. */
    int size();
}
```


All interface members are automatically `public`. According to the Java Language Specification: "It is [. . .] strongly discouraged as a matter of style, to redundantly specify the public modifier for interface methods."

Two Views of an Interface

Here are two ways to think of an `interface`:

- It's a **guarantee** to the client code that any class that implements the interface definitely has methods with these headers (and maybe other methods too).
- It's an **obligation** of the implementer, who must write these methods.

Using Queue

We can't create instances of `Queue` (it doesn't have any code that *does* anything).

```
public class Broken {
    public static void main(String[] args) {
        Queue q = new Queue(15);
        for (int i=0; i != 15; i++) {
            q.enqueue(new Integer(i));
        }
    }
}
```

However, we can write code to *use* (an instance of a class implementing) `Queue`.

```
public class Okay {
    public static void fill(Queue q, int n) {
        for (int i=0; i != n; i++) {
            q.enqueue(new Integer(i));
        }
    }
    public static void main(String[] args) {
        // To call fill(), we must construct some
        // kind of Queue, so we need a class that
        // implements Queue.
    }
}
```


Implementing an Interface

In order to implement an `interface`, we must write a class that declares that it implements the interface.

For example, a class that implements the `Queue` interface must begin like this:

```
class ... implements Queue {
```

The class must contain two things:

- Bodies of the methods guaranteed by the interface to exist.
- Instance variables to hold the queue contents (or whatever). They should be `private`.

We are allowed to add other things to our class, but these are our obligations.

Implementing Queue

```
/**
 * A Queue with fixed capacity.
 * Operations are constant-time except for
 * construction which is O(capacity) and
 * dequeue which is O(size()).
 */
public class ArrayQueue implements Queue {

    /** The number of elements in me. */
    private int size;

    /** contents[0 .. size-1] contains my elements. */
    private Object[] contents;

    // Representation invariant:
    //   size >= 0.
    //   If size is 0, I am empty.
    //   If size > 0:
    //     contents[0] is the head
    //     contents[size-1] is the tail
    //     contents[0 .. size-1] contains the
    //       Objects in the order they were inserted.

    /** An ArrayQueue with capacity for n elements. */
    public ArrayQueue(int n) {
        contents = new Object[n];
    }
```

(Continued on next slide)


```
    /** Append o to me. */
    public void enqueue(Object o) {
        contents[size++] = o;
    }

    /** Remove and return my front element.
      * This is O(size()).
      * Requires: size() != 0. */
    public Object dequeue() {
        Object head = contents[0];

        // Move all the other elements up one spot.
        for (int i = 0; i != size; ++i) {
            contents[i] = contents[i+1];
        }

        --size;
        return head;
    }

    /** Return my front element.
      * Requires: size() != 0. */
    public Object head() {
        return contents[0];
    }

    /** Return the number of elements in me. */
    public int size() {
        return size;
    }
}
```


Questions

Why is the constructor $O($`capacity`$)$?

Would the following have worked in `dequeue()`?

```
    for (int i = size - 1; i != 0; --i) {
        contents[i-1] = contents[i];
    }
```

Using ArrayQueue

We couldn't construct a `Queue`, but we can construct an `ArrayQueue`.

Example (compiles and runs):

```
public class AlsoOkay {
    public static void fill(Queue q, int n) {
        for (int i=0; i != n; i++) {
            q.enqueue(new Integer(i));
        }
    }

    public static void main(String[] args) {
        // Could also be declared as ArrayQueue.
        Queue q = new ArrayQueue(15);
        fill(q, 15);
    }
}
```


Queue improvements

A Java `Vector` has operations that let it be used like a `Queue` (Exercise: look up which ones).

Why not just use `Vector` all the time?

Because we can optimize our `Queue` implementation(s) for the specific `Queue` operations.

Question: What operation takes the most time in `ArrayQueue`?

Some alternative techniques

Don't bother shifting

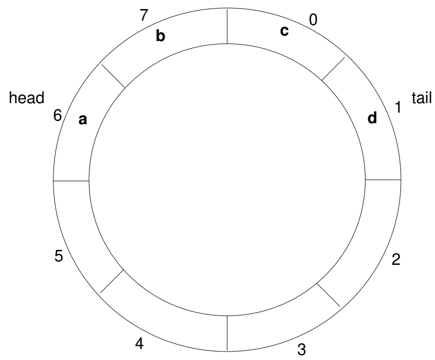
- Keep a variable that says where the head is (since it's going to move down).
- What if the queue eventually slides down to the end of the array?

Use a circular array

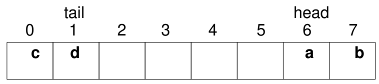
- As above, but when the tail (or head) gets to the end, wrap around to the beginning.
- That is, treat the array as if it were circular.

Here are two ways of drawing a queue containing a b c d:

Intuitively:



Reality:



Question: Devise a sequence of operations that would lead to this situation.

Class CircularQueue

```
/** A Queue with fixed capacity and (except for creation)
  * constant-time operations. */
public class CircularQueue implements Queue {

    /** The number of elements in me. */
    private int size;
    /** The index of the head and tail of the queue. */
    private int head, tail;
    /** The items in me,
        stored in contents[head .. tail],
        with wraparound. */
    private Object[] contents;

    // Representation invariant:
    //   Let "capacity" be contents.length.
    //   size >= 0.
    //   If size is 0, I am empty.
    //   If size > 0:
    //     contents[head] is the head
    //     contents[tail] is the tail
    //     if head <= tail,
    //       contents[head .. tail] contains
    //       the Objects in the order they were
    //       inserted; size=tail-head+1.
    //     if head > tail,
    //       contents[head .. capacity-1, 0 .. tail]
    //       contains the Objects in the order they
    //       were inserted; size=tail-head+1+capacity.
```



```
    /** A CircularQueue with capacity for n elements. */
    public CircularQueue(int n) {
        contents = new Object[n];
    }
    /** Append o to me. */
    public void enqueue(Object o) {
        tail = (tail+1) % contents.length;
        contents[tail] = o;
        ++size;
    }
    /** Remove and return my front element.
        Requires: size() != 0. */
    public Object dequeue() {
        Object result = contents[head];
        head = (head+1) % contents.length;
        --size;
        return result;
    }
    /** Return my front element.
        Requires: size() != 0. */
    public Object head() {
        return contents[head];
    }
    /** Return the number of elements in me. */
    public int size() {
        return size;
    }
}
```

Question: Why not calculate `size` instead of storing it?