QUEUES

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.

In these slides we show Java code for queues without the "generics" provided in the newer Java 1.5. That's because we'd like to work with arrays, which are not so easy to use with generics. In general, that might not be the ideal choice.

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.
  Precondition: The queue is not empty.
- **dequeue():** Remove and return the front element of the queue.
  Precondition: The queue is not empty.
- **size():** Return the number of elements in the queue.

A Java Interface

Remember that an interface is like a class, but without any instance variables or method bodies. It defines what a class must do, but is not a class itself. We need to define a class later that “implements” the interface.

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

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

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

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

Implementing Queue

```
/** A Queue with fixed capacity.
 */
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()).
      * Precondition: size() != 0. */
    public Object dequeue() {
        Object head = contents[0];
        // Move all the other elements up one spot.
        for (int i = 0; i != size - 1; ++i) {
            contents[i] = contents[i+1];
        }
        --size;
        return head;
    }

    /** Return my front element.
      * Precondition: 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 `ArrayList` has operations that let it be used like a `Queue` (Exercise: look up which ones).

Why not just use `ArrayList` 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

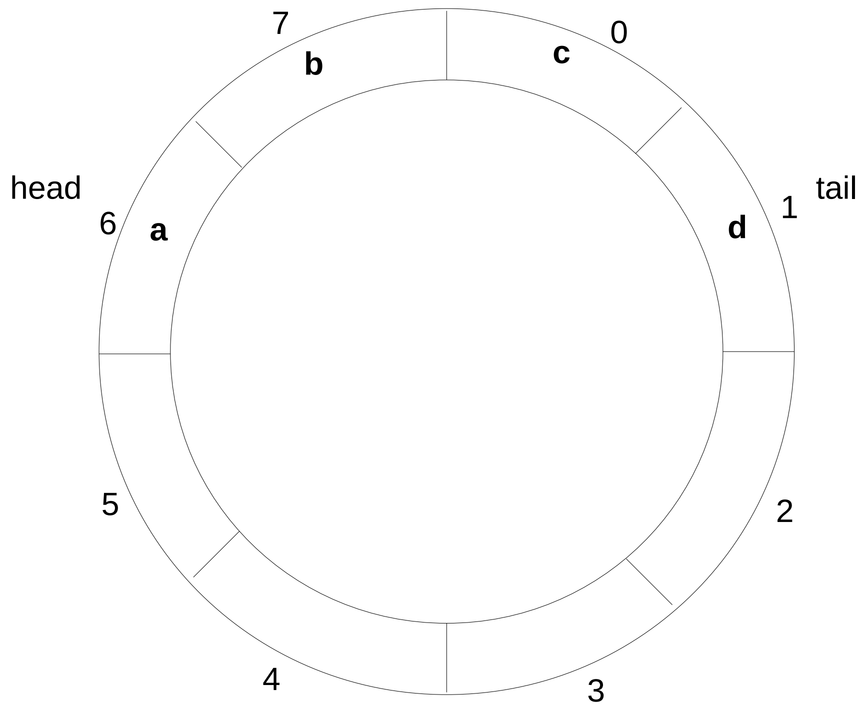
- Keep a variable that says where the head is (since it's going to move down).
- What if the queue eventually slides down to the end of the array?

Use a circular array

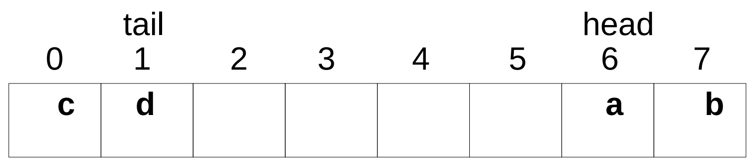
- As above, but when the tail (or head) gets to the end, wrap around to the beginning.
- That is, treat the array as if it were circular.

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];
        tail = contents.length - 1;
    }
    /** Append o to me. */
    public void enqueue(Object o) {
        tail = (tail+1) % contents.length;
        contents[tail] = o;
        ++size;
    }
    /** Remove and return my front element.
        Precondition: size() != 0. */
    public Object dequeue() {
        Object result = contents[head];
        head = (head+1) % contents.length;
        --size;
        return result;
    }
    /** Return my front element.
        Precondition: 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?