CSE 12
The Queue Abstract Data Type

- The Queue ADT
- Designing a test plan for Queue
- The Adapter pattern again
- Alternative implementations of Queue
- Circular arrays
- Time costs of Queue implementations
The Queue ADT

Queue: a First-In-First-Out (FIFO) data structure
- the first item put into the structure will be the first item to be taken out of it

• Values: a sequence of data items of type T
• Operations:
  - enqueue(T)
  - T dequeue()
  - T peek()
  - boolean isEmpty()
  - int size()

these are the essential ones!
Examples of Queue applications

- Operating systems schedulers: processes can be placed in a queue awaiting their turn to use the processor; the process that has been waiting the longest will be selected to run next

- Print spoolers: print jobs on a first-come-first-served basis

- Network packet buffering: packets arriving on the input port of a router will be queued up on one of the router’s output ports for the next hop

- Simulations of any queue-like behavior (e.g., planes on a runway, customers in a checkout line)
Queue Properties and Attributes

**Properties**
- A Queue is a FIFO structure.
- Considered as a linear structure, an addition is done at one end (the ‘back’ or ‘rear’ or ‘tail’) and a deletion at the other end (the ‘front’ or ‘head’)
- This queue has a fixed upper bound (capacity) on the number of elements it can store

**Attributes**
- capacity: The maximum number of elements that can be in the queue
- size: The number of elements in the queue: $0 \leq size \leq \text{capacity}$
- front: Indicates the end of the queue from which elements are removed
- rear: Indicates the end of the queue to which elements are added

A Queue could be “boundless”, but *this* example has a limited capacity
A picture of a queue

Head or Front of the Queue

Tail or Back or Rear of the Queue
Queue Operations

Queue()
pre-condition: none
responsibilities: constructor - initialize the queue attributes
post-condition: size is set to 0
capacity is set to DEFAULT_CAPACITY
front refers to a null value (not part of the queue)
rear refers the position where the first insertion will be made
returns: nothing

Queue( int maxElements )
pre-condition: maxElements > 0
responsibilities: constructor — initialize the queue attributes
post-condition: size is set to 0
capacity is set to maxElements
front refers to a null value
rear refers to the position where the first insertion will be made
returns: nothing
throws: invalid argument exception if pre-condition is not met

Note features needed to implement an upper bound on the collection’s capacity.
enqueue( T element )

pre-condition: isFull() is false

responsibilities: insert element at the rear of the queue
post-condition: element is placed at the rear of the queue

size is incremented by 1

rear is advanced to the position where the next element will be inserted, or some null value if the queue is full

returns: nothing

throws: full queue exception if pre-condition is not met
Queue Operations: dequeue()

dqueue()
  pre-condition: isEmpty() is false
  responsibilities: remove and return the element at front
  post-condition: the front element is no longer in the queue
                  size is decremented by 1
                  front refers to the element that is now at the front of the queue,
                  or null if the queue is empty
  returns: the element removed
  throws: empty queue exception if pre-condition is not met
# Behavior of Queue’s operations

<table>
<thead>
<tr>
<th>Method</th>
<th>Purpose</th>
<th>New Object State</th>
<th>Returned Value</th>
</tr>
</thead>
</table>
| `Queue<String> q = new LinkedQueue<String>()` | Create an empty Queue q                         | `capacity = 5
size = 0
front = null  rear = 1st queue slot`                       | a LinkedQueue object for Strings using the default capacity |
| q.enqueue (“A”)                | Add “A” to q; add an element to an empty queue   | `size = 1
“A”  “A”  rear = 2nd queue slot`                       |                                       |
| q.enqueue (“B”)                | Add “B” to q; add an element to a non-empty queue| `size = 2
“A”  “B”  “B”  rear = 3rd queue slot`                        |                                       |
| q.isFull()                     | See if q is full                                 |                                                       | false                                |
| q.enqueue (“C”)                | Add “C” to q                                    | `size = 3
“A”  “B”  “C”  “C”  rear = 4th queue slot`                      |                                       |
| String str = q.peek()          | Peek at the front element of q                   | “A”                                                    |                                       |
| str = q.dequeue()              | Delete the front element from q                  | `size = 2
“B”  “C”  “C”  rear = 3rd queue slot`                      | “A”                                   |
| q.isEmpty()                    | See if q is empty                                |                                                       | false                                |
| str = q.dequeue ()             | Delete front element from q                     | `size = 1
“C”  “C”  “C”  rear = 2nd queue slot`                      | “B”                                   |
| str = q.dequeue ()             | Delete front element from q                     | `size = 0
front = null  rear = 1st queue slot`                       | “C”                                   |
| q.isEmpty()                    | See if q is empty                                |                                                       | true                                  |
| str = q.peek()                 | Peek at the front element of q                   |                                                       | empty queue exception                |
Designing a Test Plan

The usual suspects:
- correct instantiation
- accessor methods return the correct values
- insertions really do insertions and in the right place
- deletions really do deletions and return the correct element

What does having an upper bound on size add to the plan?
- adds ‘corner cases’ that should be explicitly tested:
  - test adding when queue is one element short of capacity
  - test removing when queue is at capacity
  - test adding when queue is at capacity (should throw exception?)
## Test Case: Enqueueing to capacity

<table>
<thead>
<tr>
<th>Method</th>
<th>Purpose</th>
<th>Object State</th>
<th>Expected Result</th>
</tr>
</thead>
</table>
| `Queue<String> q = new LinkedQueue<String>(3)` | Create an empty Queue with a fixed capacity | capacity = 3  
size = 0  
front = null  
rear = 1st queue slot | a LinkedQueue object for Strings with a capacity of 3 |
| `q.enqueue("A")` | Enqueue into empty queue | size = 1  
"A”  
front = “A”  
rear = 2nd queue slot | |
| `q.enqueue("B")` | Enqueue into a nonempty queue | size = 2  
“A” “B”  
front = “A”  
rear = 3rd queue slot | |
| `q.enqueue("C")` | Enqueue into a nonempty queue and reach the queue’s capacity | size = 3  
“A” “B” “C”  
front = “A”  
rear = undefined | |
| `q.size()` | Verify queue state | 3 | |
| `q.isEmpty()` | Verify queue state | false | |
| `q.isFull()` | Verify queue state | true | |
| `q.peek()` | Verify queue state | “A” | |
| `q.enqueue("oops")` | Verify test for pre-condition for `enqueue()` | full queue exception | |
Implementing Queue

• We are implementing a particular variant of the Queue ADT, which is a limited capacity queue

• As with any interface specification, this ADT can be implemented in many ways

• We will consider two approaches:
  – Use the Adapter pattern, and Adapt an existing List class
  – A direct implementation using a circular array
ListQueue: Applying the Adapter Pattern

Tasks to complete to implement a Queue by adapting List:

1. Choose which implementation of List is most appropriate for representing Queue
2. Decide which attributes from List will play the roles of the Queue attributes
3. Select methods from List that will provide the behavior needed for the Queue API
4. Deciding how to support any Queue behavior or attributes that are not supported by List
5. Define the exception classes
   FullQueueException and EmptyQueueException
Choose the List Implementation to use

• What are the essential characteristics of a Queue?
  – It is a linear structure that is accessed and modified only at the ends

• What are the essential characteristics of the List implementations in `java.util`?
  – `ArrayList` – modifying at index 0 is expensive $O(n)$ while modifications at end are cheap $O(1)$
  – `LinkedList` – accesses at either end are $O(1)$ because it is doubly-linked and maintains a reference to the head and tail of the list

  **LinkedList wins!**
Map Queue Attributes to List Attributes and/or Methods

<table>
<thead>
<tr>
<th>Queue Attribute</th>
<th>List Attribute or Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td>size()</td>
</tr>
<tr>
<td>front</td>
<td>head – position 0 of the list</td>
</tr>
<tr>
<td>rear</td>
<td>tail – position size() of the list</td>
</tr>
<tr>
<td>capacity</td>
<td>none</td>
</tr>
</tbody>
</table>

What are we going to do about this?
## Map Queue Methods to List Methods

<table>
<thead>
<tr>
<th>Queue Operation</th>
<th>List Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>void enqueue( element )</td>
<td>void add( list.size(), element )</td>
</tr>
<tr>
<td>E dequeue()</td>
<td>E remove( 0 )</td>
</tr>
<tr>
<td>E front()</td>
<td>E get( 0 )</td>
</tr>
<tr>
<td>int size()</td>
<td>int size()</td>
</tr>
<tr>
<td>boolean isEmpty()</td>
<td>boolean isEmpty()</td>
</tr>
<tr>
<td>void clear()</td>
<td>void clear()</td>
</tr>
<tr>
<td>boolean isFull()</td>
<td>none</td>
</tr>
<tr>
<td>capacity()</td>
<td>none</td>
</tr>
</tbody>
</table>

What are we going to do about this?
Providing for Queue Characteristics Not Supported by List Characteristics

• List has no built-in support for enforcing a fixed upper bound on size (i.e. a limited capacity)

• So the ListQueue implementation can’t delegate this to a List; it will have to manage this itself...
What does the implementation look like?

```java
import java.util.LinkedList;
import java.util.List;
import java.lang.IllegalArgumentException;

/**
 * Queue ADT implemented using the Adapter design pattern and java.util.LinkedList as the storage data structure. This version of Queue assumes an upper bound on the number of elements that can be stored in the queue.
 */
public class ListQueue<E> implements Queue<E> {
    private List<E> queue;
    private int capacity;

    /**
     * Default constructor. Create an empty queue with the default capacity.
     */
    public ListQueue() {
        this( DEFAULT_CAPACITY );
    }
}
```

Aspects relating to enforcing a limited capacity
What does the implementation look like?

```java
/**
 * Add <tt>element</tt> to the end of the queue.
 * @throws FullQueueException if the queue is full
 */
public void enqueue(E element) {
    if (this.isFull()) {
        throw new FullQueueException("The queue is full");
    }
    this.queue.add(this.queue.size(), element);
}

/**
 * Determine if this queue has room for more elements.
 * @return <tt>true</tt> if this queue has room for more elements
 * <tt>false</tt> otherwise.
 */
public boolean isFull() {
    return this.queue.size() == this.capacity;
}
```

Enforcing a limited capacity
Define Custom Exception Classes as needed

```java
/**
 * Thrown when there is an attempt to access the front
 * of an empty queue.
 */
public class EmptyQueueException extends RuntimeException {
    public EmptyQueueException() {
        super();
    }
    public EmptyQueueException( String errMsg ) {
        super(" " + errMsg);
    }
}
```

Make it an *unchecked* exception... why?

Make it an unchecked exception... why?

FullQueueException would look very similar...
Another implementation of Queue

• As we have seen, if we are going to adapt an existing List class to implement Queue, it makes sense to adapt LinkedList instead of ArrayList

• But let’s look more closely at the idea of using an array (directly, not using the Adapter pattern) to implement a Queue

• This will lead to the idea of a circular array…
ArrayQueue: Using an Array underlying data structure

A Queue is a linear structure that is accessed at both ends. How do we map *front* and *rear* to the two ends of an array? Here are two options:

(a) Queue.*front* is always at 0 – shift elements *left* on dequeue().

(b) Queue.*rear* is always at 0 – shift elements *right* on enqueue().
ArrayQueue: another option

- Neither of those solutions is very good as they both involve moving all the existing data elements, which has high time cost.

- Idea: Instead of moving data elements to a fixed position for front when removing, let front advance through the array.

Hmmm…what do we do when we now add an element to that queue at the rear? What happens when we remove several elements, and front catches up with rear?...
ArrayQueue: Using a *circular* array underlying data structure

**Solution**: Be more creative!

View the array as *circular* and allow both *front* and *rear* to advance through (around) the array.

This will require *no* data movement for enqueues or dequeues!
A Circular Array Implementation of Queue

Task List:

1. Map Queue ADT attributes to a circular array implementation.

2. Implement the Queue ADT operations as methods.
Queue attributes for array implementation

```java
public class ArrayQueue<E> implements Queue<E> {
    private E[] queue;
    private int capacity;
    private int size;
    private int front;
    private int rear;
}
```

This is the easy part.

**Harder:** how should `front` and `rear` be initialized and how should they be updated?

This is important because the implementation of the queue operations will depend on the answer!
Initializing Queue Attributes

- **rear** – can initialize `rear` to 0

- **front** – the specification says it has a “null” value when the queue is empty...
  - But can't assign null to an int variable!
  - Could use -1 instead (it can't be a legal index), or...
  - ...we can just check for `size == 0` to determine queue is empty
  - So, can initialize `front` to 0, but be sure to check `size != 0` in dequeue method
The important thing is that the indexes front and rear need to be updated by the enqueue() and dequeue() operations so everything works correctly.

Different ways of doing this are possible...

As the implementer, it is up to you to make the choice.

Let's consider two possibilities.

It is really important to think about these details. It pays – lots of debugging can be avoided!
Updating Queue Attributes: plan 1

front holds the index in the array of the first element of the queue, if size!=0

- doing a dequeue() means extracting the value at front in the array, setting that array element to null, then incrementing front (with wraparound)

rear holds the index in the array where the next insertion will be made, if size < capacity

- doing an enqueue() means doing the insertion at rear, then incrementing rear (with wraparound), so that it is ready for the next enqueue() operation
Updating Queue Attributes: plan 2

front holds the index in the array of the first element of the queue, if size!=0

  – doing a dequeue() means extracting the value at position front in the array, setting that array element to null, then incrementing front (with wraparound)

rear holds the index in the array of the last element of the queue, if size!=0

  – doing an enqueue() means incrementing rear (with the wraparound), then inserting the element at rear
An implementation of enqueue, plan 1

```java
/**
* Insert element at the rear of the queue.
* @param element the element to be inserted
* post element is placed at the rear of the queue
* size is incremented by 1
* rear is position where the next element will be inserted
* @throws FullQueueException if isFull() is true
*/
public void enqueue( E element ) throws FullQueueException{
    1. if the queue is full throw a FullQueueException
    if ( this.isFull() )
        throw new FullQueueException();
    2. put element at the rear of the queue
    this.queue[rear] = element;
    3. advance rear with wraparound
    this.rear = (this.rear + 1 ) % this.capacity;
    4. increment size by 1
    this.size++;
}
```
An implementation of clear

```java
/**
 * Empty the queue of elements.
 */
public void clear() {
    // empty the queue, making all the array cells null
    while ( !this.isEmpty() )
        this.dequeue();
    // loop post-conditions:
    // this.size == 0
    // this.front == this.rear
}
```

We own a method to remove elements – use it!
Testing the Implementations

• Basic ‘black box’ tests are developed based on the API

• Knowing aspects of the implementation permits some ‘white box’ tests

• Create tests to ensure that front/rear wraparound works
  – Insert elements up to capacity
  – Remove some elements
  – Insert and remove additional elements forcing front and rear to wraparound
  – Check that all the queue API operations still work!
### Implementation Time Costs

<table>
<thead>
<tr>
<th>ListQueue Operation</th>
<th>List Operation</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>enqueue( element )</td>
<td>add( size(), element )</td>
<td>O(1)</td>
</tr>
<tr>
<td>dequeue()</td>
<td>remove( 0 )</td>
<td>O(1)</td>
</tr>
<tr>
<td>peek()</td>
<td>get( 0 )</td>
<td>O(1)</td>
</tr>
<tr>
<td>clear()</td>
<td>clear()</td>
<td>O(1)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Circular ArrayQueue Operation</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>enqueue( element )</td>
<td>O(1)</td>
</tr>
<tr>
<td>dequeue()</td>
<td>O(1)</td>
</tr>
<tr>
<td>peek()</td>
<td>O(1)</td>
</tr>
<tr>
<td>clear()</td>
<td>O(n)</td>
</tr>
</tbody>
</table>

Implement Queue by adapting java.util.LinkedList

Implement Queue using a circular array

Why does it take time O(n) to clear an ArrayQueue with size n?
Next time

• Java Generics
• The Java Type System and Type Wildcards
• Copying Java Objects
• Deep vs. Shallow Copy

Reading: Gray, Ch 4