Queues

• Queue Concept
• Queue Design Considerations
• Queues in Java Collections APIs
• Queue Applications
• Reading L&C 5.1-5.8, 9.3
Queue Abstract Data Type

- A *queue* is a linear collection where the elements are added to one end and removed from the other end
- The processing is *first in, first out (FIFO)*
- The first element put on the queue is the first element removed from the queue
- Think of a line of people waiting for a bus (The British call that “queuing up”)
A Conceptual View of a Queue

Rear of Queue (or Tail)

Front of Queue (or Head)

Adding an Element

Removing an Element
Queue Terminology

• We *enqueue* an element on a queue to add one
• We *dequeue* an element off a queue to remove one
• We can also examine the *first* element without removing it
• We can determine if a queue is *empty* or not and how many elements it contains (its *size*)
• The L&C QueueADT interface supports the above operations and some typical class operations such as *toString()*
Queue Design Considerations

• Although a queue can be empty, there is no concept for it being full. An implementation must be designed to manage storage space.

• For `first` and `dequeue` operation on an empty queue, this implementation will throw an exception.

• Other implementations could return a value `null` that is equivalent to “nothing to return.”
The java.util.Queue Interface

• The java.util.Queue interface is in the Java Collections API (extends Collection)
• However, it is only an interface and you must use an implementing class
• LinkedList is the most commonly used implementing class
• For a queue of type objects:
  
  ```java
  Queue<type> myQueue = new LinkedList<type>();
  ```
The java.util.Queue Interface

• The names of the methods are different
• Enqueue is done using:
  boolean offer(T element) // returns false if full
• Dequeue is done using either:
  T poll()    // returns null value if empty
  T remove()  // throws an exception if empty
• Peek is done using either:
  T peek()    // returns null value if empty
  T element() // throws an exception if empty
Applications for a Queue

• A queue can be used as an underlying mechanism for many common applications
  – Cycling through a set of elements in order
  – Simulation of client-server operations
  – Radix Sort
  – Scheduling processes in an operating system such as printer queues
Cycling through Code Keys

• The Caesar cipher is simple letter shifting
• Each letter is treated as its number 0-25 in the alphabet and each letter is encoded as:
  
  \[
  \text{cipher value} = (\text{letter value} + \text{constant}) \mod 26
  \]

• The message is decoded letter by letter:
  
  \[
  \text{letter value} = (\text{cipher value} - \text{constant}) \mod 26
  \]
  
  if (letter value < 0) letter value += 26

• Using the constant 7, the word “queue” would be coded as “xblblbl”
• Note: the word’s “pattern” is recognizable
Cycling through Code Keys

• The Caesar cipher is easy to solve because there are only 26 possible “keys” to try
• It can be made harder by cycling through a key set of values such as 3, 1, 7, 4, 2, 5
• We put that sequence of numbers in a queue
• As we encode each letter, we dequeue a number for the constant and re-enqueue it - cycling through the entire key set as many times as needed for the message length
Cycling through Code Keys

- Using that queue of numbers as the constant values, the word “queue” becomes “tvlyg”
- Note: the word’s “pattern” is not recognizable
- If we are encoding a message containing the entire Unicode character set, we can omit the modulo 26 operator as in the text book code
- See L&C, Listing 5.1
Ticket Counter Simulation

• See L&C Listing 5.2 and 5.3
• The simulation in this example sets up a queue with each customer arriving at regular 15 second intervals
• This is not a very meaningful analysis because it doesn’t take into account the typical variations in arrival rates
• E.G. One customer every 15 seconds could mean 8 customers arriving at one time and then 2 minutes with no arriving customers
Ticket Counter Simulation

• Textbook code always gives same values:

<table>
<thead>
<tr>
<th>cashiers</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5317</td>
</tr>
<tr>
<td>2</td>
<td>2325</td>
</tr>
<tr>
<td>3</td>
<td>1332</td>
</tr>
<tr>
<td>4</td>
<td>840</td>
</tr>
<tr>
<td>5</td>
<td>547</td>
</tr>
<tr>
<td>6</td>
<td>355</td>
</tr>
<tr>
<td>7</td>
<td>219</td>
</tr>
<tr>
<td>8</td>
<td>120</td>
</tr>
<tr>
<td>9</td>
<td>120</td>
</tr>
<tr>
<td>10</td>
<td>120</td>
</tr>
</tbody>
</table>
Ticket Counter Simulation

• A more sophisticated simulation would use probability distributions for the arrival rate and for the processing time
  – With an average serving time that sets a maximum capacity for handling customers based on the number of servers
  – And an average arrival time with parameters for the distribution of arrivals over time

• A statistical analysis is more important for an ice cream shop next to a movie theater (during a movie versus as a movie lets out)
Ticket Counter Simulation

• Poisson Distribution is commonly used for estimating arrival times in simulations
  – Lambda is the average number of arrivals per time interval
  – $P(X=k)$ is the probability that $k$ is the number of arrivals during this time interval
Ticket Counter Simulation

• Replacement for textbook code in listing 5.3:

```java
/** load customer queue
   improved to use random Poisson arrival times*/
Poisson myDist = new Poisson(lambda);
for (int count=0; count < NUM_INTERVALS / lambda; count++)
{
    int numberOfCustomers = myDist.getValue();
    for (int i = 0; i < numberOfCustomers; i++)
        customerQueue.offer(new Customer(count*15*lambda));
}

• Introduces random arrival times based on the Poisson distribution for each time interval
Ticket Counter Simulation

• For a Markov/Markov/1 (M/M/1) process (one queue and one server), the expected waiting time can be calculated in closed form

\[
\text{Waiting Time} = \frac{\text{Service Time}}{1 - \text{load/capacity}}
\]

• This produces a graph of waiting time versus load/capacity with infinite waiting time at load equal to 100% or more of capacity
Ticket Counter Simulation

Waiting Time versus Load / Capacity
Radix Sort - IBM Card Sorter
Radix Sort - Algorithm

• See L&C Listing 9.3
• Like the old IBM punched card sorters

Original List

1. Sort into Queues by Radix

   Q0  Q1  Q10

   *  *  *

2. Empty Each Queue in Order and Add to List