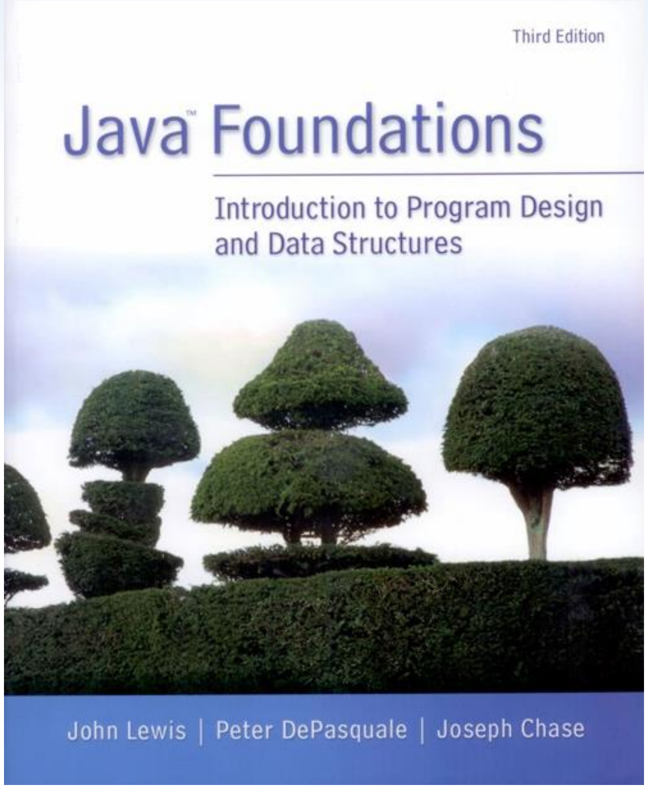


Chapter 14

Queues

Chapter Scope

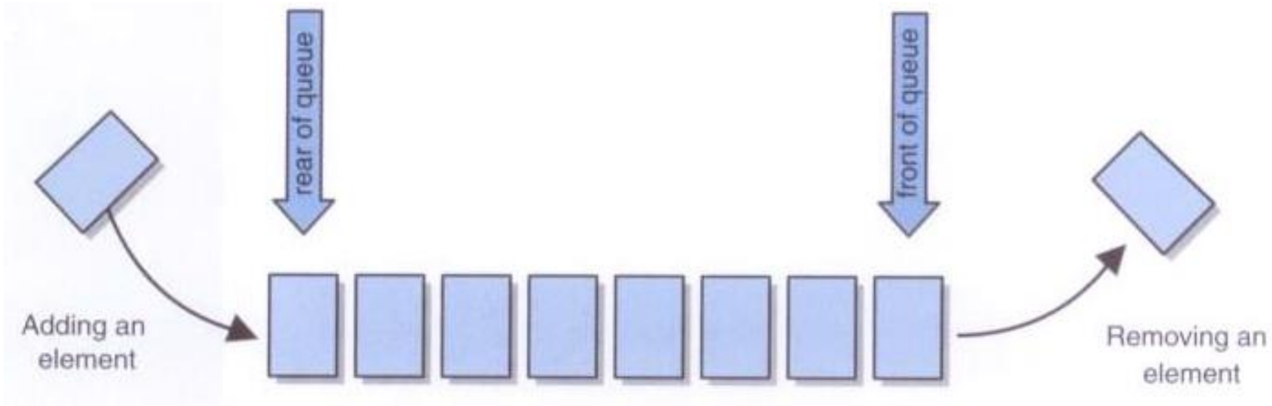
- Queue processing
- Using queues to solve problems
- Various queue implementations
- Comparing queue implementations

Queues

- A *queue* is a collection whose elements are added on one end and removed from the other
- Therefore a queue is managed in a *FIFO* fashion: first in, first out
- Elements are removed in the same order they arrive
- Any waiting line is a queue:
  - the check out line at a grocery store
  - the cars at a stop light
  - an assembly line

Queues

- A queue, conceptually:



Queues

- Standard queue operations:

Operation	Description
enqueue	Adds an element to the rear of the queue.
dequeue	Removes an element from the front of the queue.
first	Examines the element at the front of the queue.
isEmpty	Determines if the queue is empty.
size	Determines the number of elements on the queue.
toString	Returns a string representation of the queue.

Queues in the Java API

- The Java Collections API is not consistent about its implementation of collections
- For queues, the API provides a `Queue` interface, then various classes such as `LinkedList` implement the interface
- Furthermore, the `Queue` interface defines two versions of the methods that add and remove elements from the queue
- The difference in the two versions is how exceptional situations are handled

Queues in the Java API

- The `add` method throws an exception if the element cannot be added, and the `offer` method returns a boolean indicating success or failure
- When the queue is empty, the `remove` method throws an exception and the `poll` method returns null
- The goal is to give the user the option of handling exceptional cases in whatever way is preferred

Coded Messages

- Let's use a queue to help us encode and decode messages
- A *Ceasar cipher* encodes a message by shifting each letter in a message by a constant amount k
- If k is 5, A becomes F, B becomes G, etc.
- However, this is fairly easy to break
- An improvement can be made by changing how much a letter is shifted depending on where the letter is in the message

Coded Messages

- A *repeating key* is a series of integers that determine how much each character is shifted
- For example, consider the repeating key

3 1 7 4 2 5

- The first character in the message is shifted 3, the next 1, the next 7, and so on
- When the key is exhausted, we just start over at the beginning of the key

Coded Messages

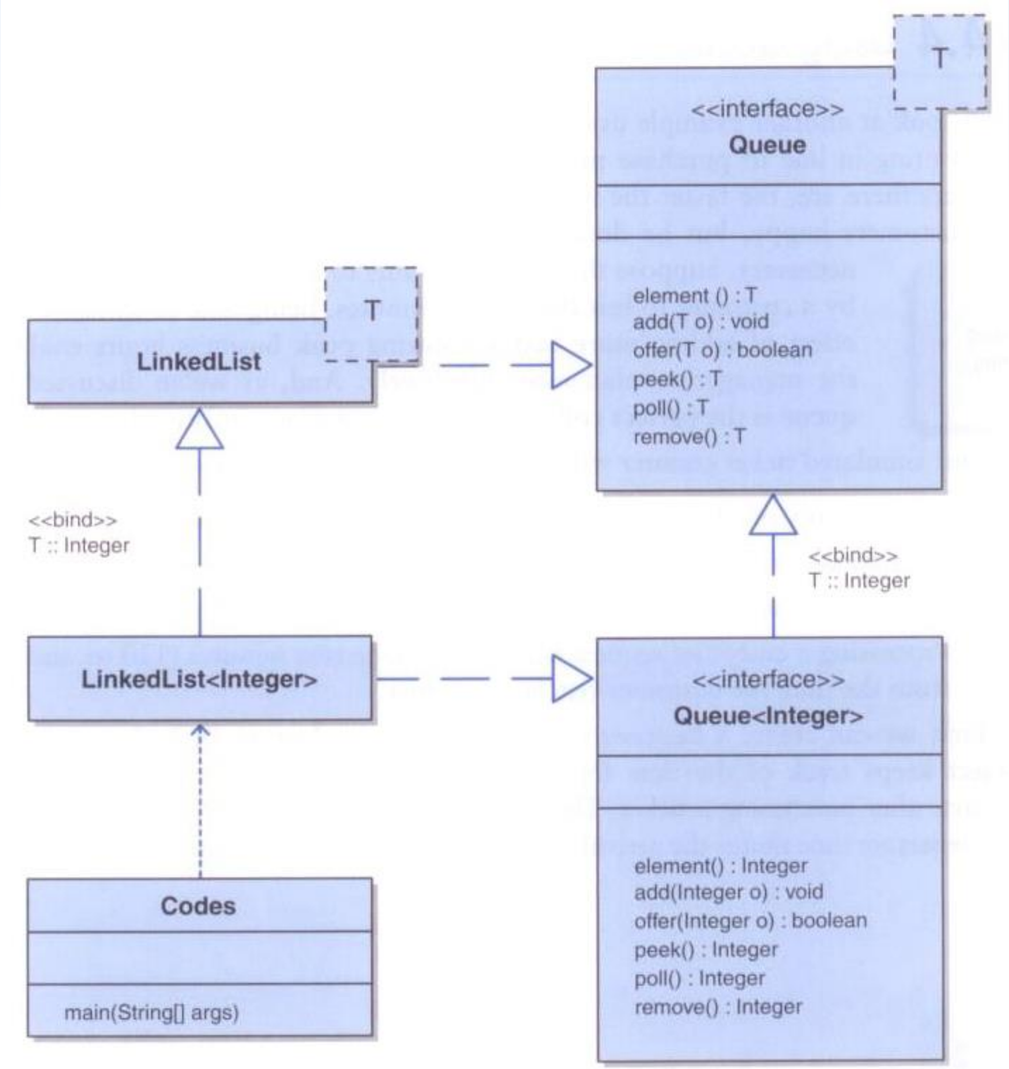
- Decoding a message using a repeating key:

Encoded Message:	n	o	v	a	n	j	g	h	l		m	u		u	r	x	l	v
Key:	3	1	7	4	2	5	3	1	7		4	2		5	3	1	7	4
Decoded Message:	k	n	o	w	l	e	d	g	e		i	s		p	o	w	e	r

```java
/**
 * Codes demonstrates the use of queues to encrypt and decrypt messages.
 *
 * @author Java Foundations
 * @version 4.0
 */
public class Codes
{
    /**
     * Encode and decode a message using a key of values stored in
     * a queue.
     */
    public static void main(String[] args)
    {
        int[] key = {5, 12, -3, 8, -9, 4, 10};
        Integer keyValue;
        String encoded = "", decoded = "";
        String message = "All programmers are playwrights and all " +
                         "computers are lousy actors.";
        Queue<Integer> encodingQueue = new LinkedList<Integer>();
        Queue<Integer> decodingQueue = new LinkedList<Integer>();

        // load key queues
        for (int scan = 0; scan < key.length; scan++)
        {
            encodingQueue.add(key[scan]);
            decodingQueue.add(key[scan]);
        }
```

```
        // encode message
        for (int scan = 0; scan < message.length(); scan++)
        {
            keyValue = encodingQueue.remove();
            encoded += (char) (message.charAt(scan) + keyValue);
            encodingQueue.add(keyValue);
        }

        System.out.println ("Encoded Message:\n" + encoded + "\n");

        // decode message
        for (int scan = 0; scan < encoded.length(); scan++)
        {
            keyValue = decodingQueue.remove();
            decoded += (char) (encoded.charAt(scan) - keyValue);
            decodingQueue.add(keyValue);
        }

        System.out.println ("Decoded Message:\n" + decoded);
    }
}
```

Coded Messages



Ticket Counter Simulation

- Now let's use a queue to simulate the waiting line at a movie theatre
- The goal is to determine how many cashiers are needed to keep the wait time below 7 minutes
- We'll assume:
  - customers arrive on average every 15 seconds
  - processing a request takes two minutes once a customer reaches a cashier

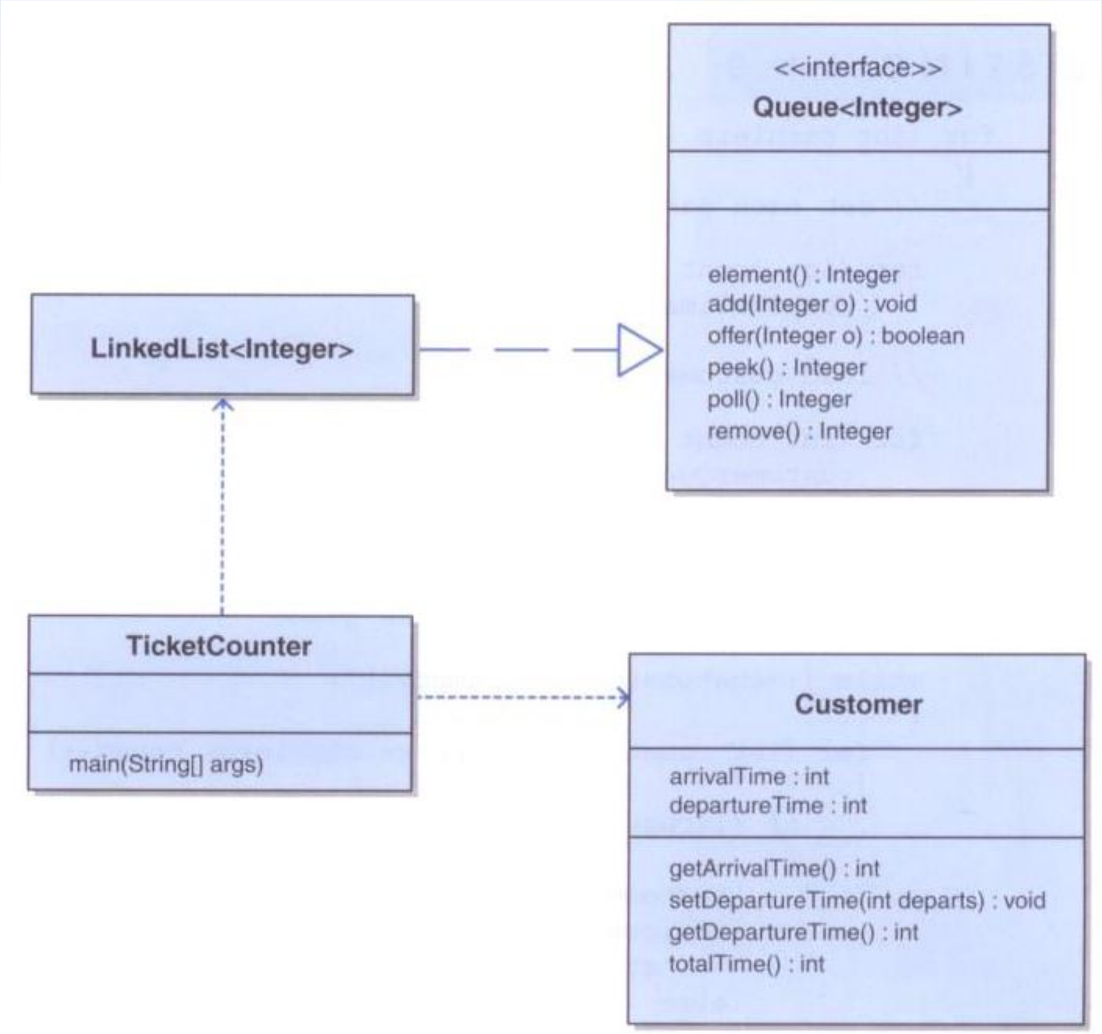
```java
/**
 * Customer represents a waiting customer.
 *
 * @author Java Foundations
 * @version 4.0
 */
public class Customer
{
    private int arrivalTime, departureTime;

    /**
     * Creates a new customer with the specified arrival time.
     * @param arrives the arrival time
     */
    public Customer(int arrives)
    {
        arrivalTime = arrives;
        departureTime = 0;
    }

    /**
     * Returns the arrival time of this customer.
     * @return the arrival time
     */
    public int getArrivalTime()
    {
        return arrivalTime;
    }
```

```java
    /**
     * Sets the departure time for this customer.
     * @param departs the departure time
     **/
    public void setDepartureTime(int departs)
    {
        departureTime = departs;
    }

    /**
     * Returns the departure time of this customer.
     * @return the departure time
     */
    public int getDepartureTime()
    {
        return departureTime;
    }

    /**
     * Computes and returns the total time spent by this customer.
     * @return the total customer time
     */
    public int totalTime()
    {
        return departureTime - arrivalTime;
    }
}
```

```java
import java.util.*;

/**
 * TicketCounter demonstrates the use of a queue for simulating a line of customers.
 *
 * @author Java Foundations
 * @version 4.0
 */
public class TicketCounter
{
    private final static int PROCESS = 120;
    private final static int MAX_CASHIERS = 10;
    private final static int NUM_CUSTOMERS = 100;

    public static void main(String[] args)
    {
        Customer customer;
        Queue<Customer> customerQueue = new LinkedList<Customer>();
        int[] cashierTime = new int[MAX_CASHIERS];
        int totalTime, averageTime, departs, start;

        // run the simulation for various number of cashiers
        for (int cashiers = 0; cashiers < MAX_CASHIERS; cashiers++)
        {
            // set each cashiers time to zero initially
            for (int count = 0; count < cashiers; count++)
                cashierTime[count] = 0;

            // load customer queue
            for (int count = 1; count <= NUM_CUSTOMERS; count++)
                customerQueue.add(new Customer(count * 15));
```

```java
            totalTime = 0;

            // process all customers in the queue
            while (!(customerQueue.isEmpty()))
            {
                for (int count = 0; count <= cashiers; count++)
                {
                    if (!(customerQueue.isEmpty()))
                    {
                        customer = customerQueue.remove();
                        if (customer.getArrivalTime() > cashierTime[count])
                                  start = customer.getArrivalTime();
                        else
                            start = cashierTime[count];
                              departs = start + PROCESS;
                            customer.setDepartureTime(departs);
                        cashierTime[count] = departs;
                        totalTime += customer.totalTime();
                    }
                }
            }

            // output results for this simulation
            averageTime = totalTime / NUM_CUSTOMERS;
            System.out.println("Number of cashiers: " + (cashiers + 1));
            System.out.println("Average time: " + averageTime + "\n");
        }
    }
}
```

Ticket Counter Simulation



Ticket Counter Simulation

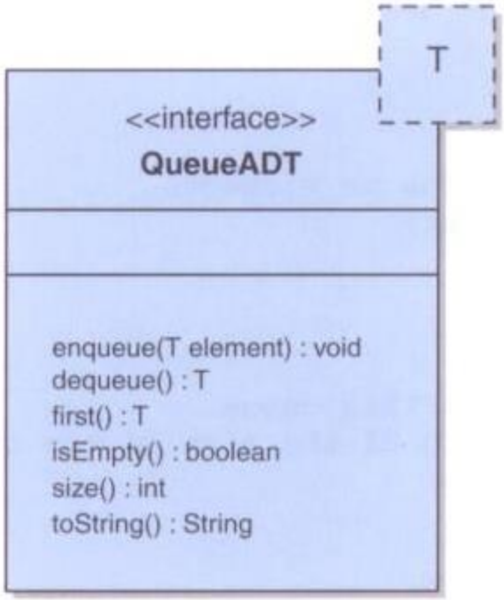
- The results of the simulation:

Number of Cashiers:	1	2	3	4	5	6	7	8	9	10
Average Time (sec):	5317	2325	1332	840	547	355	219	120	120	120

- With the goal of an average time of less than seven minutes (420 secs), six cashiers will suffice
- Adding more than eight cashiers will not improve the situation

A Queue ADT

- Defining our own interface for a queue, using only the classic operations:

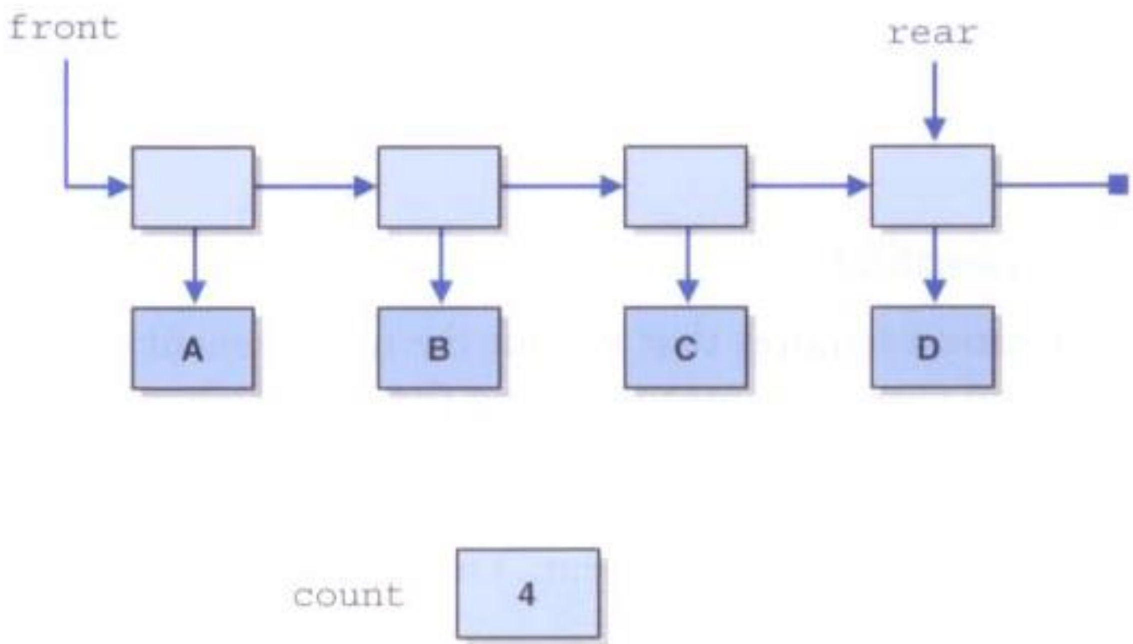


```java
package jsjf;

/**
 * QueueADT defines the interface to a queue collection.
 *
 * @author Java Foundation
 * @version 4.0
 */
public interface QueueADT<T>
{
    /**
     * Adds one element to the rear of this queue.
     * @param element  the element to be added to the rear of the queue
     */
    public void enqueue(T element);

    /**
     * Removes and returns the element at the front of this queue.
     * @return the element at the front of the queue
     */
    public T dequeue();

    /**
     * Returns without removing the element at the front of this queue.
     * @return the first element in the queue
     */
    public T first();
```

```java
    /**
     * Returns true if this queue contains no elements.
     * @return true if this queue is empty
     */
    public boolean isEmpty();

    /**
     * Returns the number of elements in this queue.
     * @return the integer representation of the size of the queue
     */
    public int size();

    /**
     * Returns a string representation of this queue.
     * @return the string representation of the queue
     */
    public String toString();
}
```

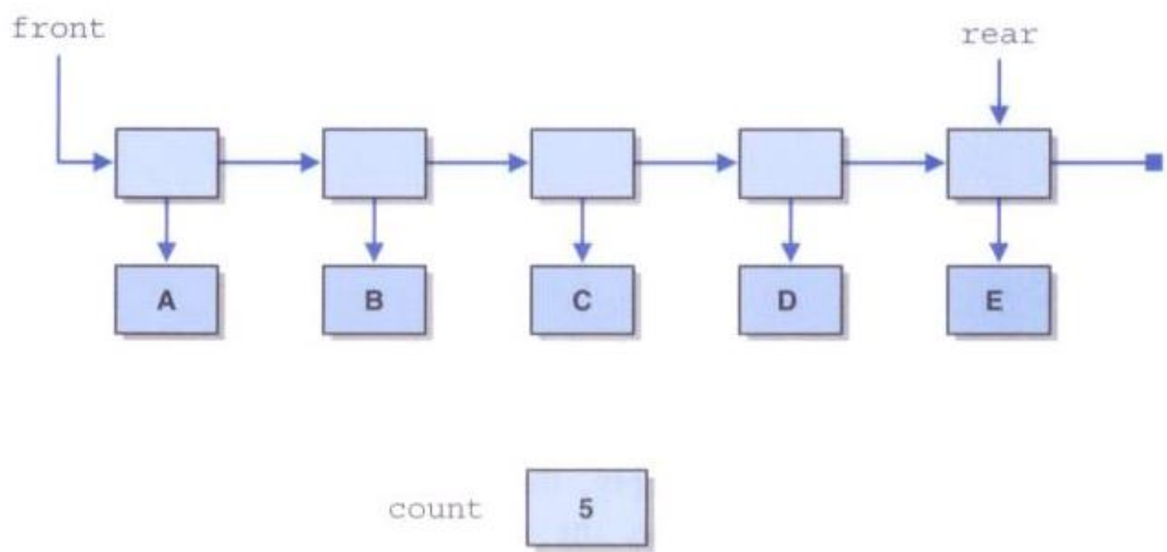
Implementing a Queue with Links

- Since operations work on both ends of the queue, we'll use both front and rear references



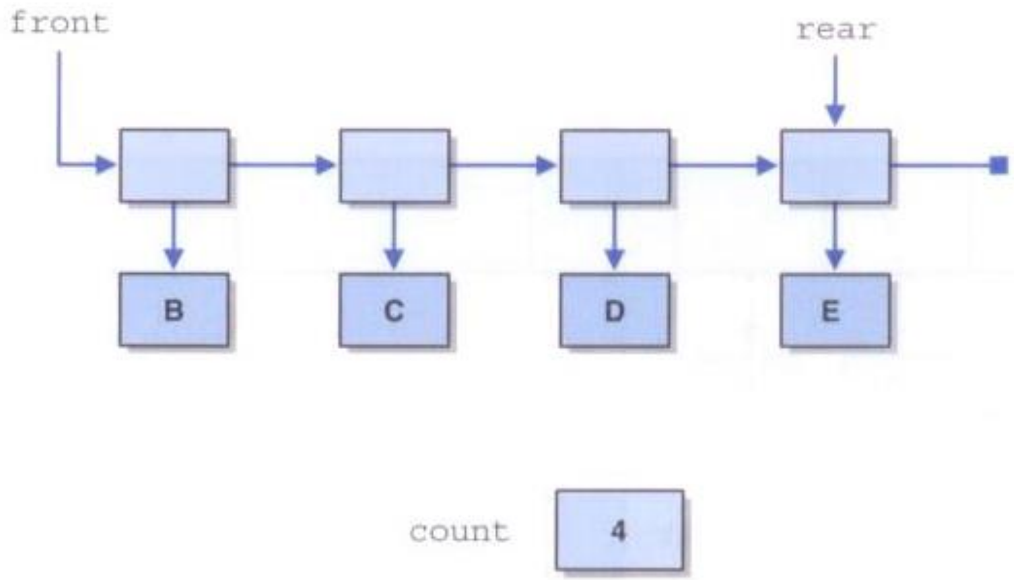
Implementing a Queue with Links

- After adding element E to the rear of the queue:



Implementing a Queue with Links

- After removing an element from the front:



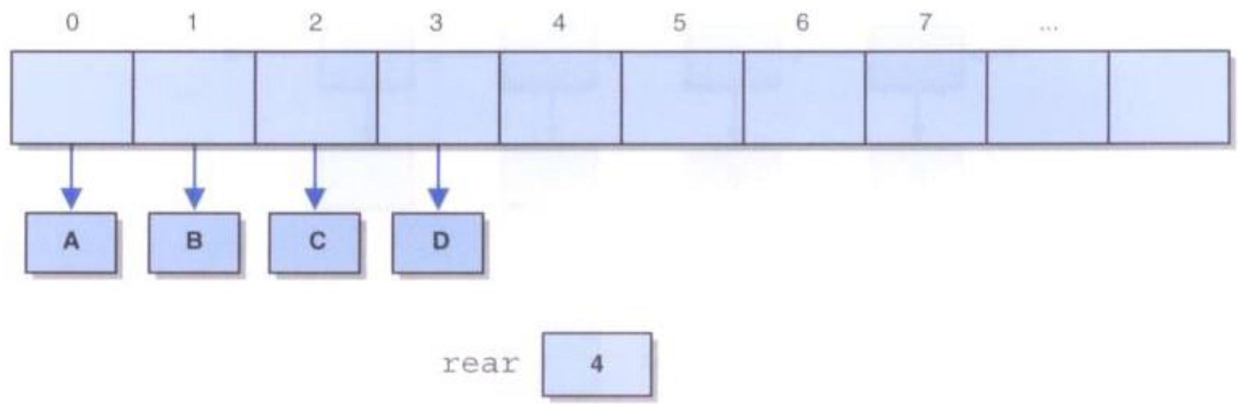
```java
package jsjf;

import jsjf.exceptions.*;

/**
 * LinkedQueue represents a linked implementation of a queue.
 *
 * @author Java Foundations
 * @version 4.0
 */
public class LinkedQueue<T> implements QueueADT<T>
{
    private int count;
    private LinkedNode<T> head, tail;

    /**
     * Creates an empty queue.
     */
    public LinkedQueue()
    {
        count = 0;
        head = tail = null;
    }
```

```java
/**
 * Adds the specified element to the tail of this queue.
 * @param element the element to be added to the tail of the queue
 */
public void enqueue(T element)
{
    LinearNode<T> node = new LinearNode<T>(element);

    if (isEmpty())
        head = node;
    else
        tail.setNext(node);

    tail = node;
    count++;
}
```

```java
/**
 * Removes the element at the head of this queue and returns a
 * reference to it.
 * @return the element at the head of this queue
 * @throws EmptyCollectionException if the queue is empty
 */
public T dequeue() throws EmptyCollectionException
{
    if (isEmpty())
        throw new EmptyCollectionException("queue");

    T result = head.getElement();
    head = head.getNext();
    count--;

    if (isEmpty())
        tail = null;

    return result;
}
```

Implementing a Queue with an Array

- If we implement a queue as we did a stack, one end would be fixed at index 0:



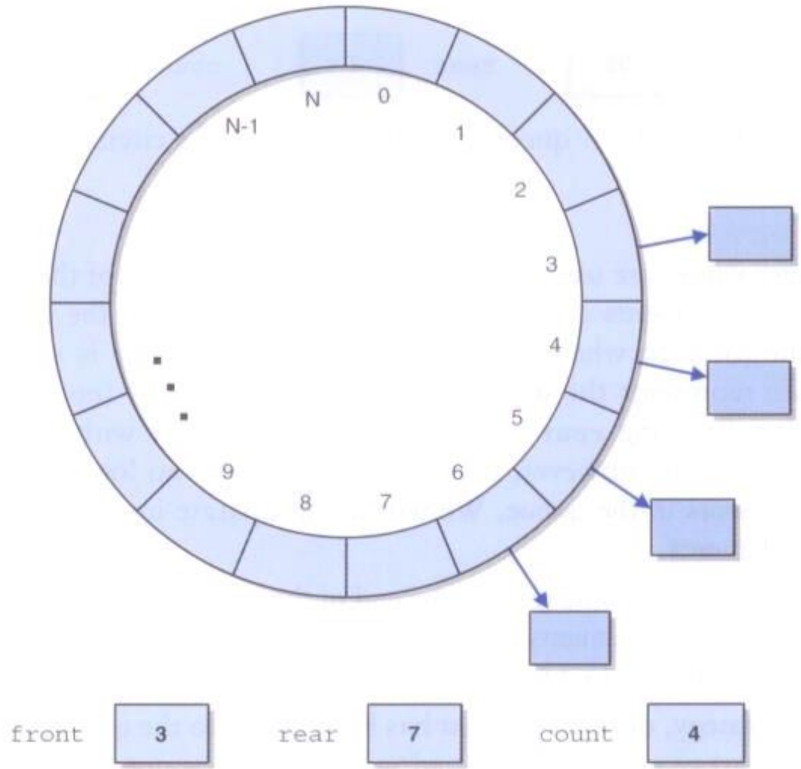
- The problem is that (unlike a stack) a queue operates at both ends
- To be efficient, we must avoid shifting elements

Implementing a Queue with an Array

- A better solution is to treat the array as circular
- A *circular array* is a a regular array that is treated as if it loops back around on itself
- That is, the last index is thought to precede index 0
- We use two integers to keep track of where the front and rear of the queue are at any given time

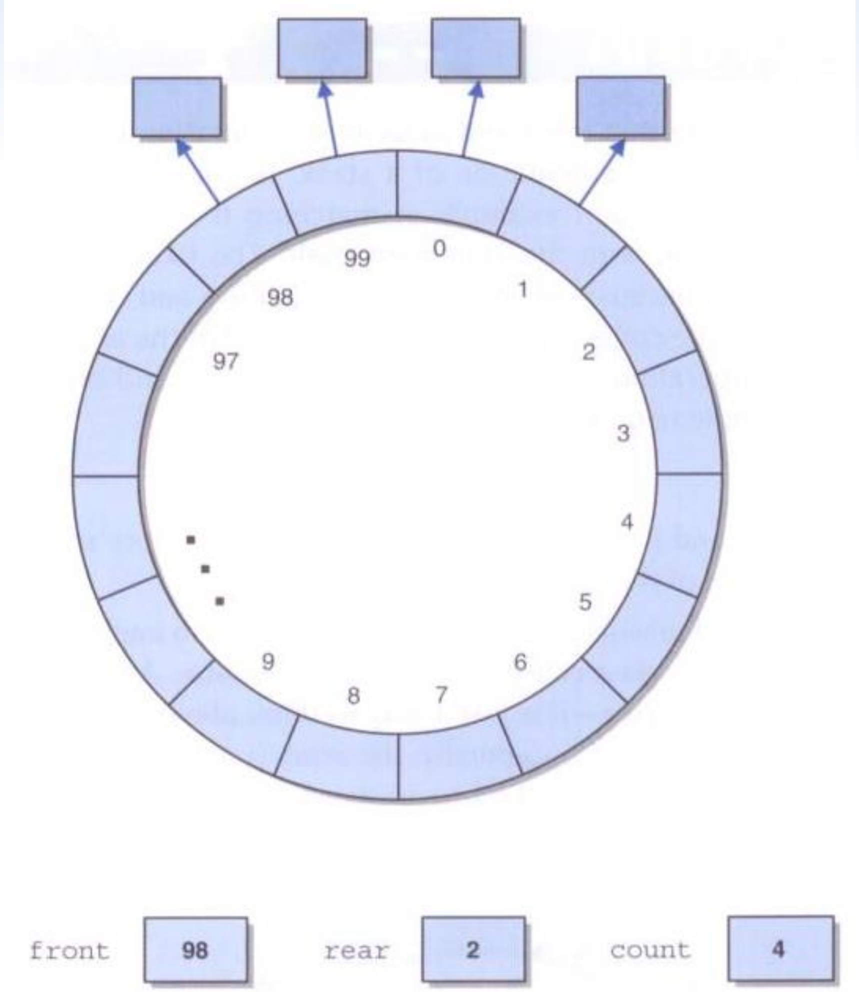
Implementing a Queue with an Array

- A queue implemented using a circular queue:

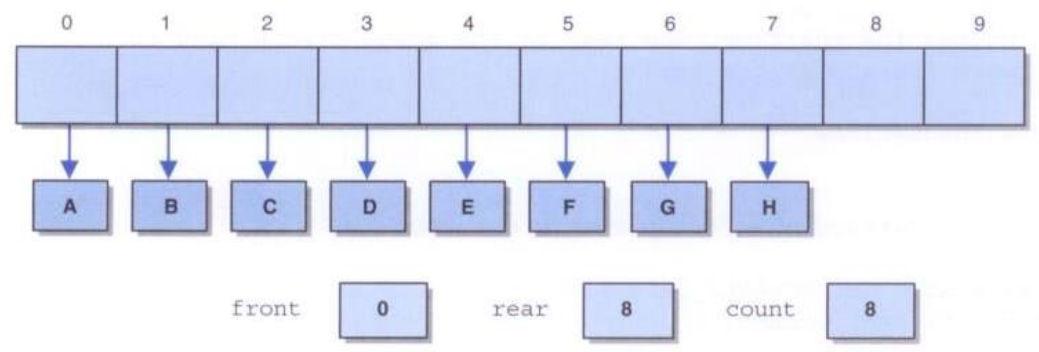


Implementing a Queue with an Array

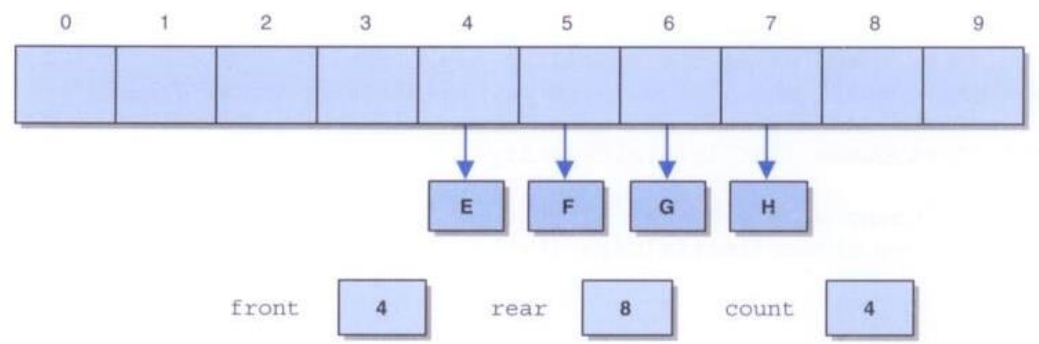
- At some point, the elements of the queue may straddle the end of the array:



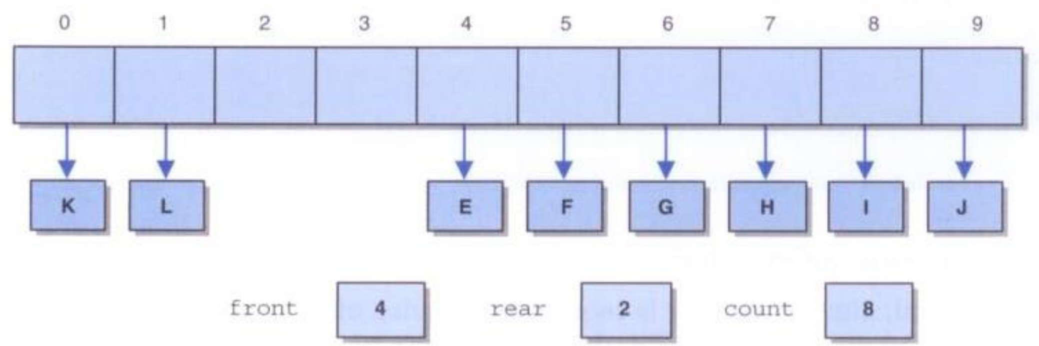
- After A-H have been enqueued:



- After A-D have been dequeueed:

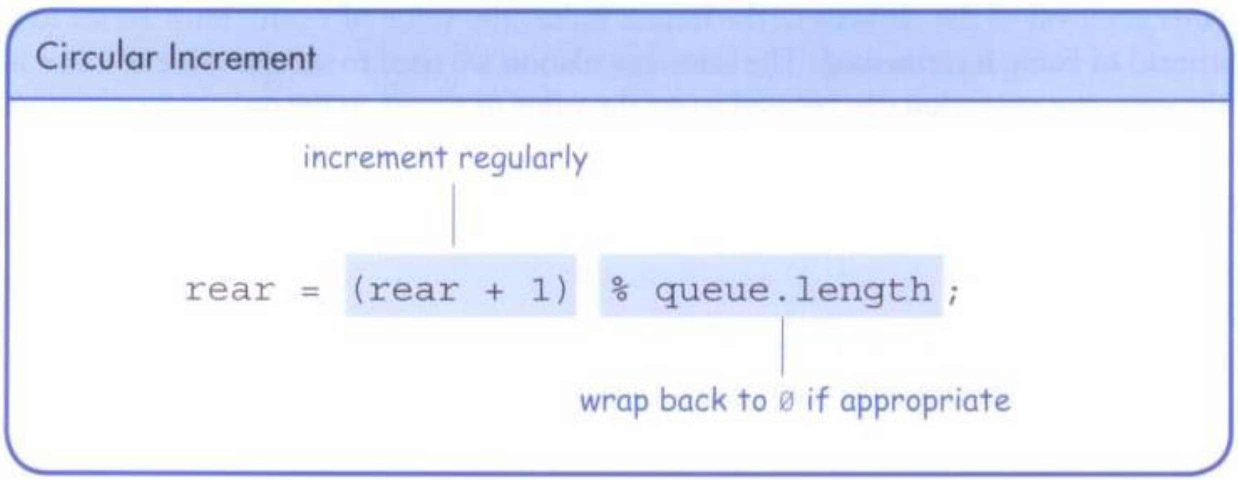


- After I, J, K, and L have been enqueued:



Implementing a Queue with an Array

- Both the `front` and `rear` index values are incremented, wrapping back to 0 when necessary



```java
package jsjf;

import jsjf.exceptions.*;

/**
 * CircularArrayQueue represents an array implementation of a queue in
 * which the indexes for the front and rear of the queue circle back to 0
 * when they reach the end of the array.
 *
 * @author Java Foundations
 * @version 4.0
 */
public class CircularArrayQueue<T> implements QueueADT<T>
{
    private final static int DEFAULT_CAPACITY = 100;
    private int front, rear, count;
    private T[] queue;
```

```java
/**
 * Creates an empty queue using the specified capacity.
 * @param initialCapacity the initial size of the circular array queue
 */
public CircularArrayQueue (int initialCapacity)
{
    front = rear = count = 0;
    queue = (T[]) (new Object[initialCapacity]);
}

/**
 * Creates an empty queue using the default capacity.
 */
public CircularArrayQueue()
{
    this(DEFAULT_CAPACITY);
}
```

```java
/**
 * Adds the specified element to the rear of this queue, expanding
 * the capacity of the queue array if necessary.
 * @param element the element to add to the rear of the queue
 */
public void enqueue(T element)
{
    if (size() == queue.length)
        expandCapacity();

    queue[rear] = element;
    rear = (rear+1) % queue.length;

    count++;
}
```

```java
/**
 * Creates a new array to store the contents of this queue with
 * twice the capacity of the old one.
 */
private void expandCapacity()
{
    T[] larger = (T[]) (new Object[queue.length *2]);

    for (int scan = 0; scan < count; scan++)
    {
        larger[scan] = queue[front];
        front = (front + 1) % queue.length;
    }

    front = 0;
    rear = count;
    queue = larger;
}
```

```java
/**
 * Removes the element at the front of this queue and returns a
 * reference to it.
 * @return the element removed from the front of the queue
 * @throws EmptyCollectionException  if the queue is empty
 */
public T dequeue() throws EmptyCollectionException
{
    if (isEmpty())
        throw new EmptyCollectionException("queue");

    T result = queue[front];
    queue[front] = null;
    front = (front+1) % queue.length;

    count--;

    return result;
}
```