Queues:
Usage and 3 Implementations

One of the most useful collections in data structures!
A queue is consistent with the general concept of
- a waiting line to buy movie tickets
- a request to print a document
- crawling the web to retrieve documents

A queue is a linear collection whose elements are added on one end and removed from the other

Queue elements are processed in a first in, first out (FIFO) manner

Elements are removed from the queue in the same order in which they are placed on the queue
15.6 – Queues

Adding an element

Removing an element

rear of queue

front of queue
public interface Queue<T> {
    // Adds element to rear of the queue
    public void enqueue(T element);
    // Removes and returns element at front of queue
    public T dequeue();
    // Return reference to first element w/o removing
    public T first();
    // Returns true if queue contains no elements
    public boolean isEmpty();
    // Returns number of elements
    public int size();
}
Example using a Queue

Queue waitList = new ArrayQueue<String>();
System.out.println("Is new waitlist empty? " +
        waitList.isEmpty());
waitList.enqueue("Aly");
waitList.enqueue("Bea");
// one spot opened
System.out.println("Enroll: " + waitList.dequeue());
waitList.enqueue("Cat");
waitList.enqueue("Duo");
System.out.println("Number of students in waitList: " +
        waitList.size());
// Allow everyone to enroll
while (!waitList.isEmpty()){
        System.out.println("Enroll: " + waitList.dequeue());
}
Exercise: Print a Queue without destroying it
ArrayQueue: implementation of the Queue interface in javafoundations using an array
package javafoundations;

public interface Queue<T> {

    // Adds element to rear of the queue
    public void enqueue (T element);

    // Removes and returns element at front of queue
    public T dequeue();

    // Returns reference to first element without removing
    public T first();

    // Returns true if queue contains no elements
    public boolean isEmpty();

    // Returns number of elements
    public int size();

    // Returns string representation
    public String toString();
}
Implementing Queues with Arrays

- Adds a new element to the rear of the queue, which is stored at the high end of the array

```
0 1 2 3 4 5 6 7 ... ...
```

Left-shift to correct queue

```
A B C D E Enqueue ‘E’
```

```
count 4
```
package javafoundations;

import javafoundations.exceptions.*;

public class ArrayQueue<T> implements Queue<T> {
    private final int DEFAULT_CAPACITY = 10;
    private int count;
    private T[] queue;

    // 0-args constructor: Creates empty queue
    public ArrayQueue() {
        count = 0;
        queue = (T[]) (new Object[DEFAULT_CAPACITY]);
    }
}
```java
public boolean isEmpty() {
    return (count == 0);
}

public int size() {
}

public T first() throws EmptyCollectionException {
    if (count == 0) throw new EmptyCollectionException
    ("First failed. Queue empty");
}

public void enqueue(T element) {
}
```
public T dequeue() throws EmptyCollectionException {
    if (count == 0) throw new EmptyCollectionException("Dequeue failed. Queue empty");
    T result = queue[0];
    count--;
    // shift elements to keep the front at element 0
    for (int index = 0; index < count; index++)
        queue[index] = queue[index+1];
    queue[count] = null;
    return result;
}

// Left as programming project:
// public String toString() {}
CircularArrayQueue: implementation of the Queue interface in Java Foundations using a circular array

Wait! What??
Circular Array?
Is there such a thing?
Implementing Queues with Circular Arrays

- As elements are dequeued, the front of the queue will move further into the array.
- As elements are enqueued, the rear of the queue will also move further into the array.
- The challenge comes when the rear of the queue reaches the end of the array.
- When this occurs, it “wraps around” to the front of the array.
- Use two variables, `front` and `rear`, to represent the location where the first element is stored, and where the next available slot in the array is located (respectively).
15.9 – Implementing Queues with Circular Arrays

The diagram illustrates a circular array used to implement a queue. The array is labeled with indices from 0 to n-1. The operations performed on the queue are:

- FRONT: 4
- REAR: 7
- COUNT: 4

The elements in the queue are labeled A, B, C, and D. The front of the queue is at index 4, and the rear is at index 7. The count of the queue is 4.
15.9 – The Changing State of a Circular Array Q

- Front: 7
- Rear: 2
- Count: 5
package javafoundations;

import javafoundations.exceptions.*;

public class CircularArrayQueue<T> implements Queue<T> {
    private final int DEFAULT_CAPACITY = 10;
    private int front, rear, count;
    private T[] queue;

    (more...)
javafoundations.CircularArrayQueue

public CircularArrayQueue() {
    front = rear = count = 0;
    queue = (T[]) (new Object[DEFAULT_CAPACITY]);
}

public void enqueue (T element) {
    if (count == queue.length) expandCapacity();
    queue[rear] = element;
    rear = (rear+1) % queue.length;
    count++;
}

(more...)
javafoundations.CircularArrayQueue

public void expandCapacity() {
    T[] larger = (T[])(new Object[queue.length*2]);

    for (int index=0; index < count; index++)
        larger[index] = queue[(front+index) % queue.length];

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

// The following methods are left as Programming Projects.

// public T dequeue () throws EmptyCollectionException {
// public T first () throws EmptyCollectionException
// public int size() {
// public boolean isEmpty() {
// public String toString() {
}
Case Study: DDOS

Distributed Denial of Service is an infamous attack on computer networks.
Distributed Denial of Service (DDOS)

To connect two computers on the internet we use routers: simple devices that control a queue

- They’re CircularArrayQueues **without** expandCapacity()

A router connects to several channels, chooses the best to send a message it received

When (portions of) a message arrives, the router stores it temporarily, decides which channel to use next, dequeues and sends the message.

A DDOS is caused when attackers flood routers with messages for delivery!
LinkedQueue: implementation of the Queue interface in java foundations using LinearNodes

Linking LinearNode Objects
Like we used to implement a Stack!
15.10 – Implementing Queues with Links

A → B → C → D

front

count 4

rear

1-23
package javafoundations;

import javafoundations.exceptions.*;

public class LinkedQueue<T> implements Queue<T> {
    private int count;
    private LinearNode<T> front, rear;

    // Creates an empty queue.
    public LinkedQueue() {
        count = 0;
        front = rear = null;
    }

    (more...)

    // Represents a linked implementation of a queue.
    // *******************************************************************************
javafoundations.LinkedQueue

// Adds the specified element to the rear of this queue.
public void enqueue (T element)
{
    LinearNode<T> node = new LinearNode<T>(element);
    if (count == 0)
        front = node;
    else
        rear.setNext(node);
    rear = node;
    count++;
}

// The following methods are left as Programming Projects.
// public T dequeue () throws EmptyCollectionException { }
// public T first () throws EmptyCollectionException { }
// public boolean isEmpty() { }
// public int size() { }
// public String toString() { }
Analysis of Stack and Queue Implementations

- All operations for a stack (push, pop, peek, etc.) are $O(1)$
- Almost all operations for a queue are $O(1)$
- The only exception is the dequeue operation for the ArrayQueue implementation – the shifting of elements makes it $O(n)$
- The dequeue operation for the CircularArrayQueue is $O(1)$ because of the ability to eliminate the shifting of elements
- Both stacks and queues can be implemented very efficiently