

CHAPTER 4

Queues

Chapter Objectives

- To learn how to represent a waiting line (queue) and how to use the methods in the `Queue` interface for insertion (`offer` and `add`), removal (`remove` and `poll`), and for accessing the element at the front (`peek` and `element`)
- To understand how to implement the `Queue` interface using a single-linked list, a circular array, and a double-linked list
- To become familiar with the `Deque` interface and how to use its methods to insert and remove items from either end of a deque
- To understand how use `Queues` and random number generators to simulate the operation of a physical system that has one or more waiting lines

Queue

- The queue, like the stack, is a widely used data structure
- A queue differs from a stack in one important way
 - ▣ A stack is LIFO list – *Last-In, First-Out*
 - ▣ while a queue is FIFO list, *First-In, First-Out*



Queue Abstract Data Type

Section 4.1

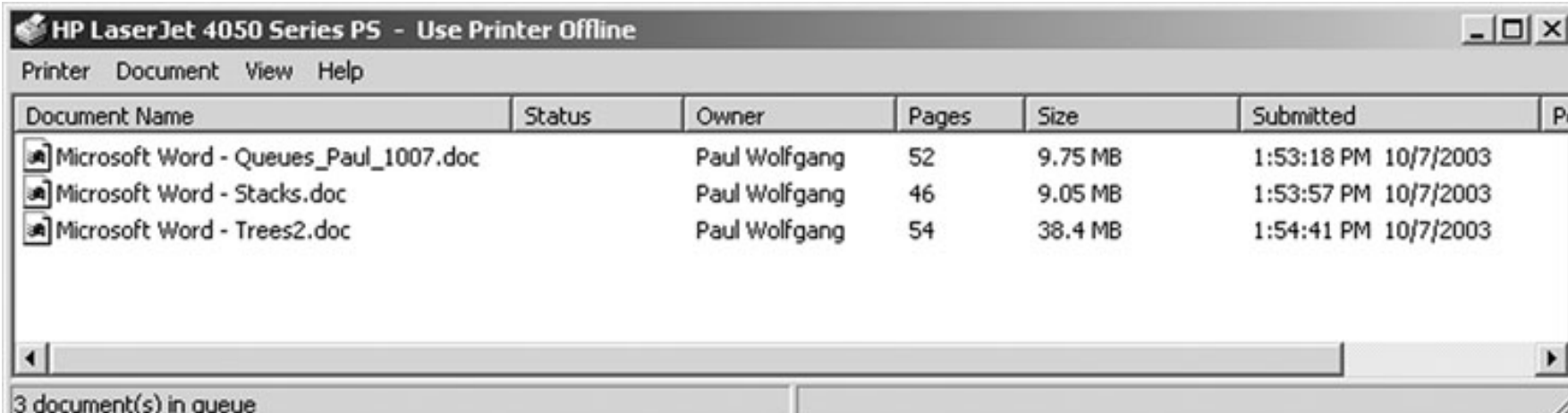
Queue Abstract Data Type

- A queue can be visualized as a line of customers waiting for service
- The next person to be served is the one who has waited the longest
- New elements are placed at the end of the line



Print Queue

- Operating systems use queues to
 - ▣ keep track of tasks waiting for a scarce resource
 - ▣ ensure that the tasks are carried out in the order they were generated
- Print queue: printing is much slower than the process of selecting pages to print, so a queue is used



HP LaserJet 4050 Series PS - Use Printer Offline

Printer Document View Help

Document Name	Status	Owner	Pages	Size	Submitted	Pi
Microsoft Word - Queues_Paul_1007.doc		Paul Wolfgang	52	9.75 MB	1:53:18 PM 10/7/2003	
Microsoft Word - Stacks.doc		Paul Wolfgang	46	9.05 MB	1:53:57 PM 10/7/2003	
Microsoft Word - Trees2.doc		Paul Wolfgang	54	38.4 MB	1:54:41 PM 10/7/2003	

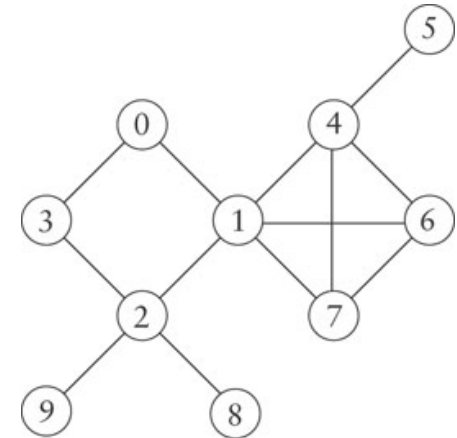
3 document(s) in queue

Unsuitability of a Print Stack

- ❑ Stacks are Last-In, First-Out (LIFO)
- ❑ The most recently selected document would be the next to print
- ❑ Unless the printer stack is empty, your print job may never be executed if others are issuing print jobs

Using a Queue for Traversing a Multi-Branch Data Structure

- A graph models a network of nodes, with links connecting nodes to other nodes in the network
- A node in a graph may have several neighbors
- Programmers doing a *breadth-first traversal* often use a queue to ensure that nodes closer to the starting point are visited before nodes that are farther away
- You can learn more about graph traversal in Chapter 10



Queue applications...

- ❑ Process scheduling
- ❑ Disk scheduling
- ❑ IPC
- ❑ IO buffers
- ❑ Interrupt scheduling
- ❑ Anything where the job itself is slower than the arrival of new jobs.

Specification for a Queue Interface

Method	Behavior
<code>boolean offer(E item)</code>	Inserts <code>item</code> at the rear of the queue. Returns true if successful; returns false if the item could not be inserted.
<code>E remove()</code>	Removes the entry at the front of the queue and returns it if the queue is not empty. If the queue is empty, throws a <code>NoSuchElementException</code> .
<code>E poll()</code>	Removes the entry at the front of the queue and returns it; returns null if the queue is empty.
<code>E peek()</code>	Returns the entry at the front of the queue without removing it; returns null if the queue is empty.
<code>E element()</code>	Returns the entry at the front of the queue without removing it. If the queue is empty, throws a <code>NoSuchElementException</code> .

- The `Queue` interface implements the `Collection` interface (and therefore the `Iterable` interface), so a full implementation of `Queue` must implement all required methods of `Collection` (and the `Iterable` interface)

Class `LinkedList` Implements the `Queue` Interface

- The `LinkedList` class provides methods for inserting and removing elements at either end of a double-linked list, which means all `Queue` methods can be implemented easily
- The Java 5.0 `LinkedList` class implements the `Queue` interface

```
Queue<String> names = new LinkedList<String>();
```

- ▣ creates a new `Queue` reference, `names`, that stores references to `String` objects
- ▣ The actual object referenced by `names` is of type `LinkedList<String>`, but because `names` is a type `Queue<String>` reference, you can apply only the `Queue` methods to it



Maintaining a Queue of Customers

Section 4.2

Maintaining a Queue of Customers

- Write a menu-driven program that maintains a list of customers
- The user should be able to:
 - ▣ insert a new customer in line
 - ▣ display the customer who is next in line
 - ▣ remove the customer who is next in line
 - ▣ display the length of the line
 - ▣ determine how many people are ahead of a specified person

Designing a Queue of Customers

- ❑ Use `JOptionPane.showOptionDialog()` for the menu
- ❑ Use a queue as the underlying data structure
- ❑ Write a `MaintainQueue` class which has a `Queue<String>` component `customers`

Data Field	Attribute
<code>private Queue<String> customers</code>	A queue of customers.
Method	Behavior
<code>public static void processCustomers()</code>	Accepts and processes each user's selection.

Designing a Queue of Customers (cont.)

Algorithm for `processCustomers`

1. `while` the user is not finished
2. Display the menu and get the selected operation
3. Perform the selected operation

Algorithm for determining the position of a Customer

1. Get the customer name
2. Set the count of customers ahead of this one to 0
3. `for each` customer in the queue
4. `if` the customer is not the one sought
5. increment the counter
6. `else`
7. display the count of customers and exit the loop
8. `if` all the customers were examined without success
9. display a message that the customer is not in the queue

Implementing a Queue of Customers

- Listing 4.1(MaintainQueue, page 202)
- Listing 4.2 (method processCustomers in Class MaintainQueue, pages 203-204)



Implementing the Queue Interface

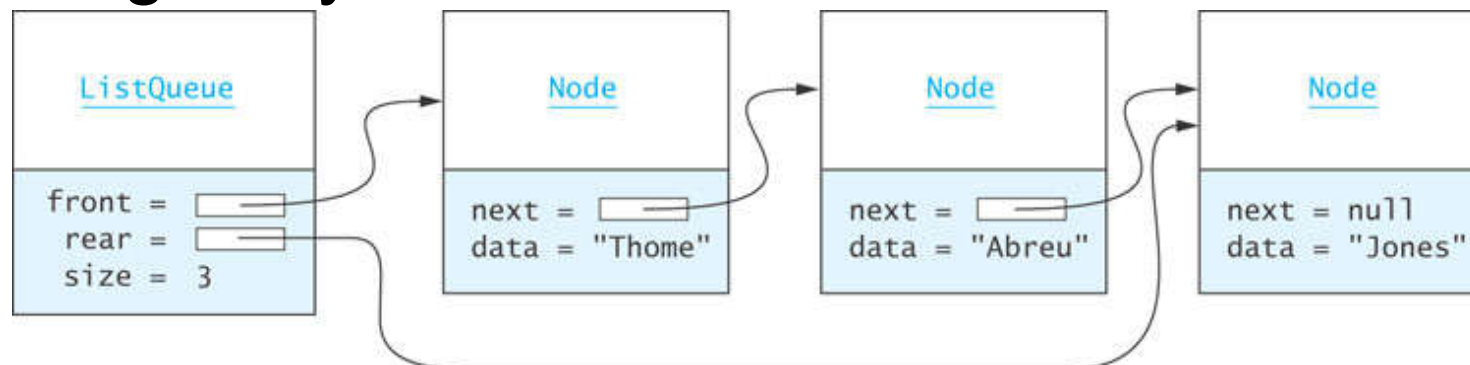
Section 4.3

Using a Double-Linked List to Implement the `Queue` Interface

- Insertion and removal from either end of a double-linked list is $O(1)$ so either end can be the front (or rear) of the queue
- Java designers decided to make the head of the linked list the front of the queue and the tail the rear of the queue
- Problem: If a `LinkedList` object is used as a queue, it will be possible to apply other `LinkedList` methods in addition to the ones required and permitted by the `Queue` interface
- Solution: Create a new class with a `LinkedList` component and then code (by delegation to the `LinkedList` class) only the public methods required by the `Queue` interface

Using a Single-Linked List to Implement a Queue

- ❑ Insertions are at the rear of a queue and removals are from the front
- ❑ We need a reference to the last list node so that insertions can be performed at $O(1)$
- ❑ The number of elements in the queue is changed by methods insert and remove

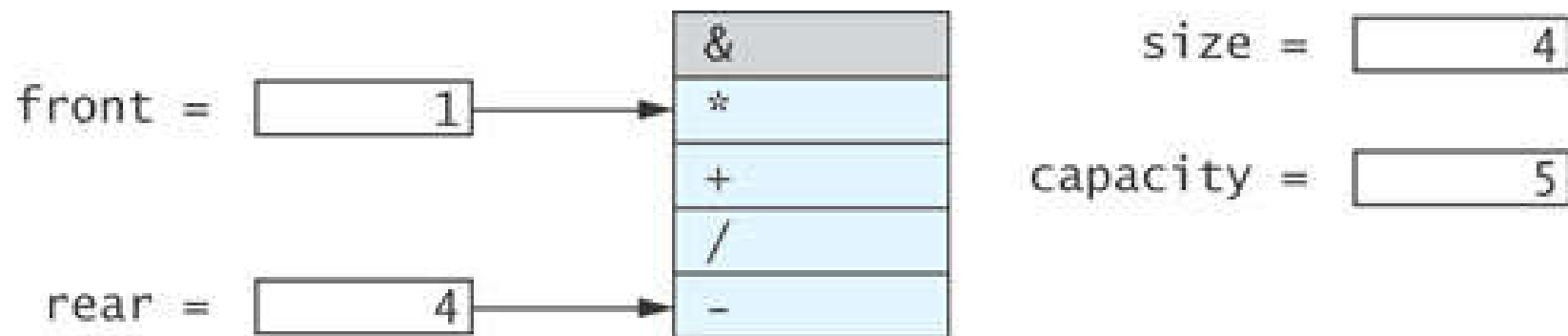
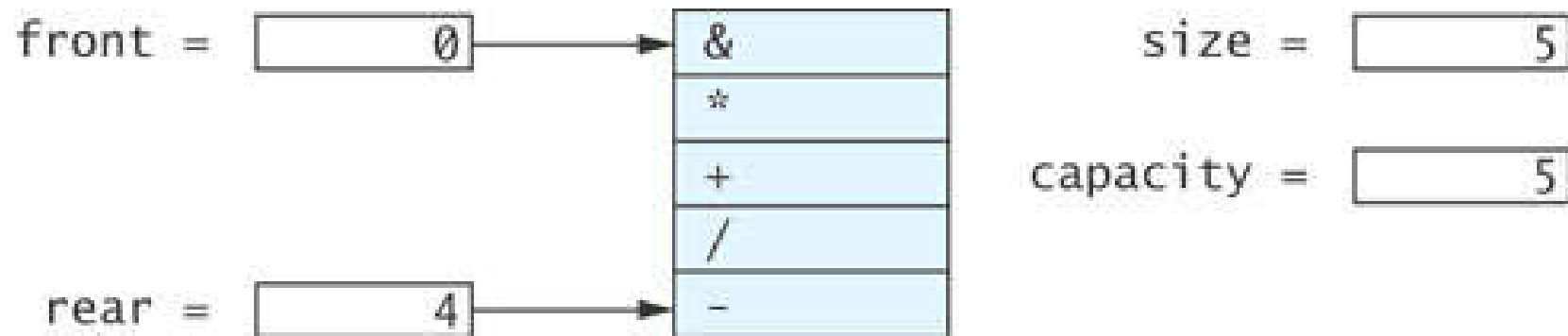


- ❑ Listing 4.3 (`ListQueue`, pages 208-209)

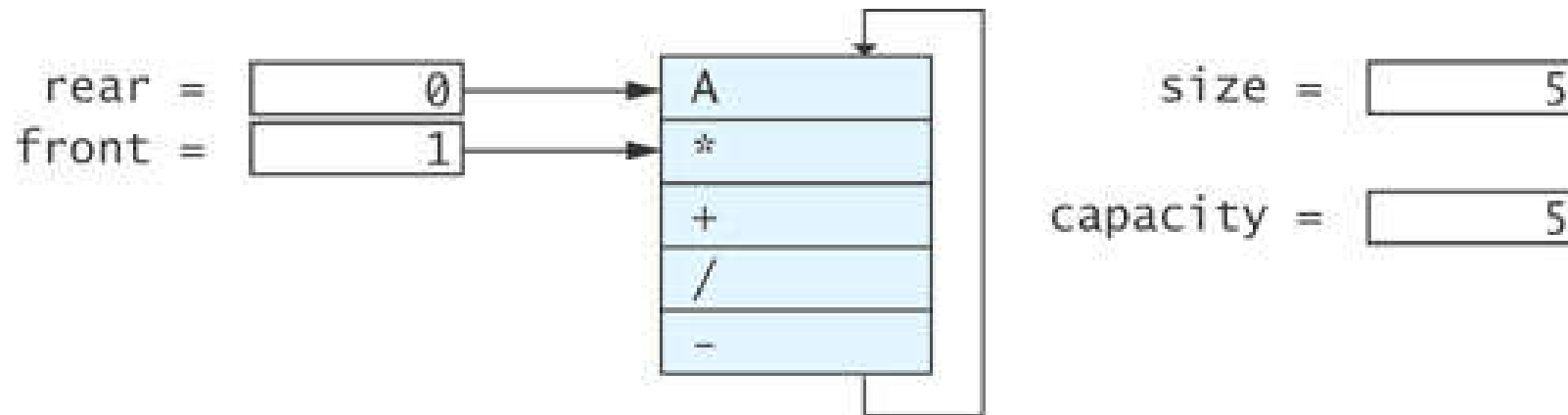
Implementing a Queue Using a Circular Array

- The time efficiency of using a single- or double-linked list to implement a queue is acceptable
- However, there are some space inefficiencies
- Storage space is increased when using a linked list due to references stored in the nodes
- Array Implementation
 - ▣ Insertion at rear of array is constant time $O(1)$
 - ▣ Removal from the front is linear time $O(n)$
 - ▣ Removal from rear of array is constant time $O(1)$
 - ▣ Insertion at the front is linear time $O(n)$
- We now discuss how to avoid these inefficiencies in an array

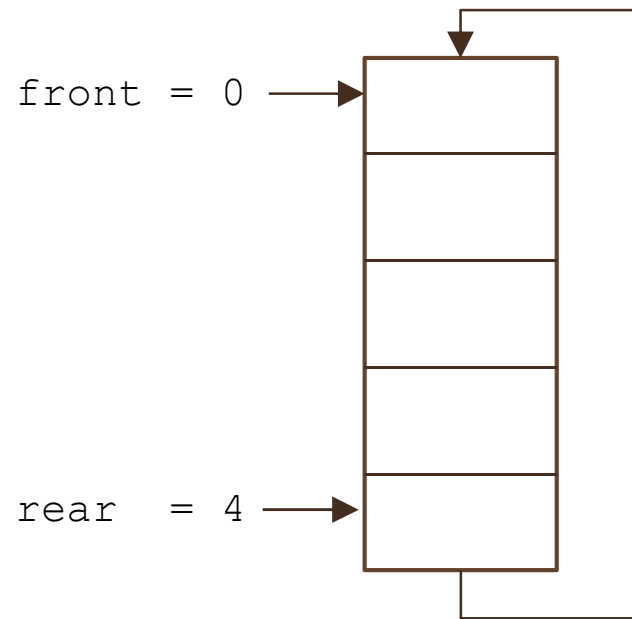
Implementing a Queue Using a Circular Array (cont.)



Implementing a Queue Using a Circular Array (cont.)



Implementing a Queue Using a Circular Array (cont.)



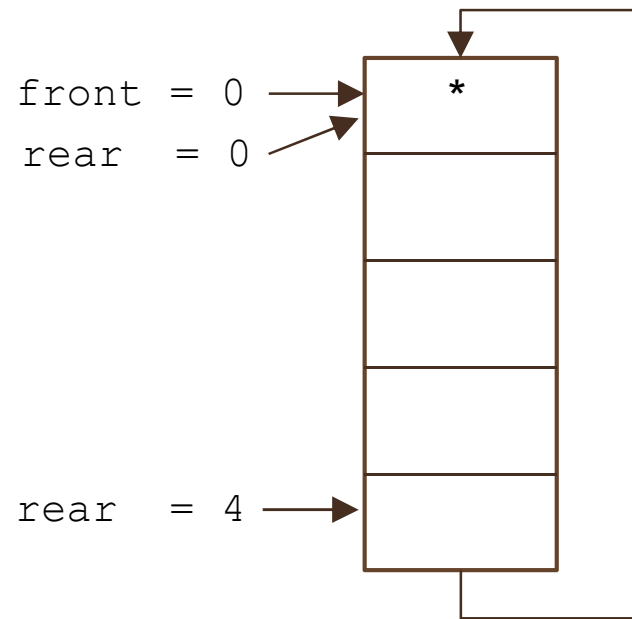
```
ArrayQueue q = new ArrayQueue(5);
```

```
size      = 0
```

```
capacity = 5
```

```
public ArrayQueue(int initCapacity) {  
    capacity = initCapacity;  
    theData = (E[])new Object[capacity];  
    front = 0;  
    rear = capacity - 1;  
    size = 0;  
}
```

Implementing a Queue Using a Circular Array (cont.)



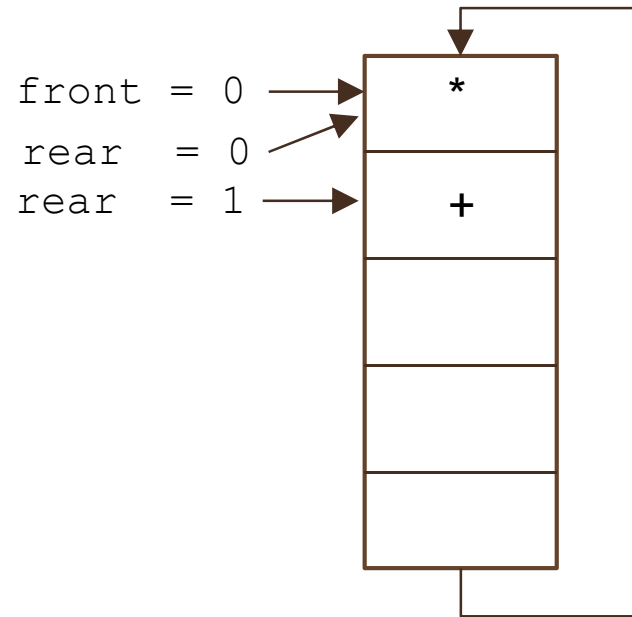
```
q.offer('*');
```

```
size      = 1
```

```
capacity = 5
```

```
⇒ public boolean offer(E item) {  
    ⇒ if (size == capacity) {  
        reallocate();  
    }  
    size++;  
    rear = (rear + 1) % capacity;  
    theData[rear] = item;  
    ⇒ return true;  
}
```


Implementing a Queue Using a Circular Array (cont.)



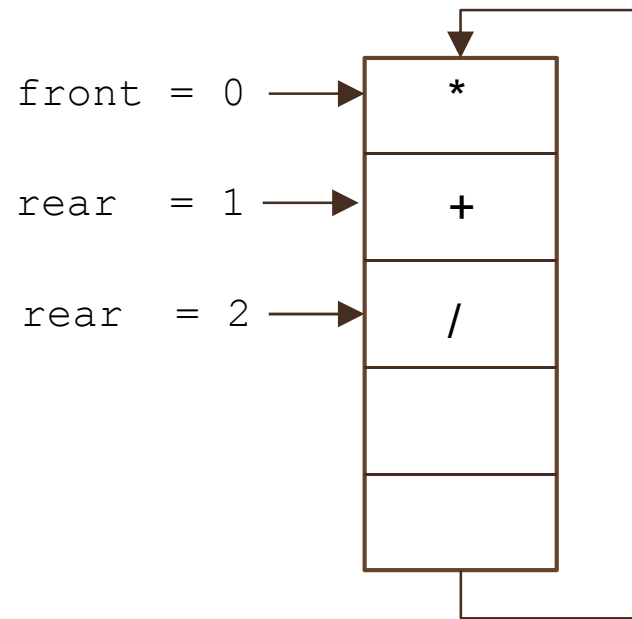
```
q.offer('+');
```

```
size      = 2
```

```
capacity = 5
```

```
⇒ public boolean offer(E item) {  
    ⇒ if (size == capacity) {  
        reallocate();  
    }  
    size++;  
    rear = (rear + 1) % capacity;  
    theData[rear] = item;  
    ⇒ return true;  
}
```

Implementing a Queue Using a Circular Array (cont.)



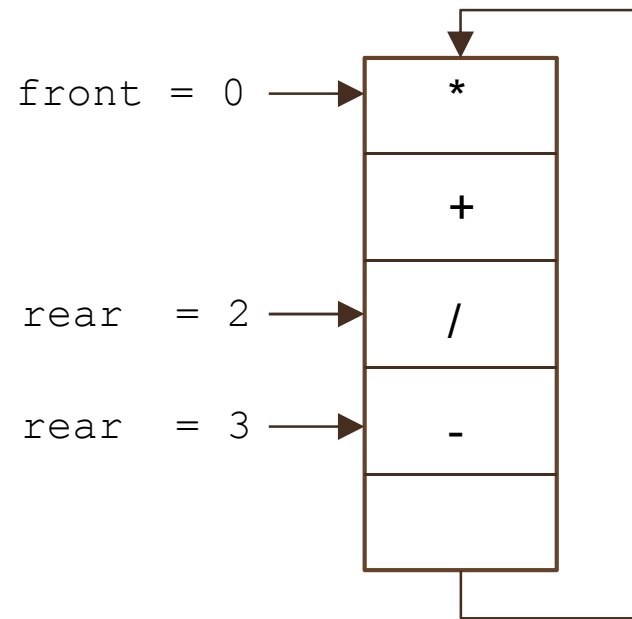
`q.offer('/') ;`

size = 3

capacity = 5

```
⇒ public boolean offer(E item) {  
    ⇒ if (size == capacity) {  
        reallocate();  
    }  
    size++;  
    rear = (rear + 1) % capacity;  
    theData[rear] = item;  
    ⇒ return true;  
}
```

Implementing a Queue Using a Circular Array (cont.)



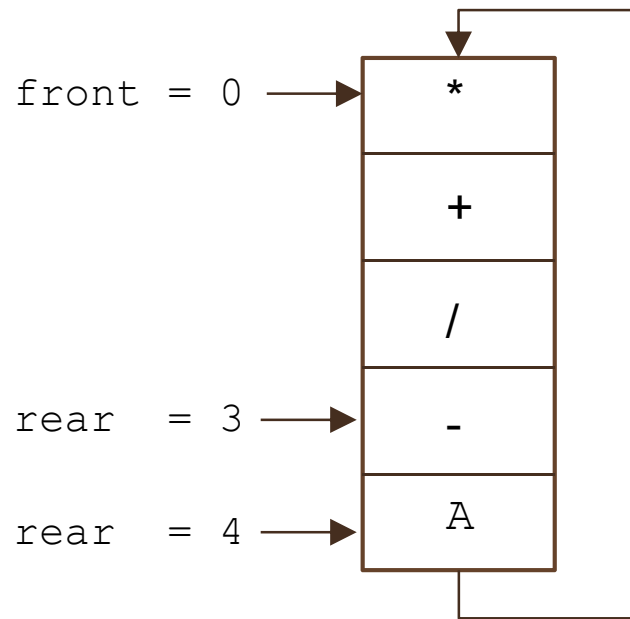
```
q.offer('-');
```

```
size      = 4
```

```
capacity = 5
```

```
⇒ public boolean offer(E item) {  
    ⇒ if (size == capacity) {  
        reallocate();  
    }  
    size++;  
    rear = (rear + 1) % capacity;  
    theData[rear] = item;  
    ⇒ return true;  
}
```

Implementing a Queue Using a Circular Array (cont.)



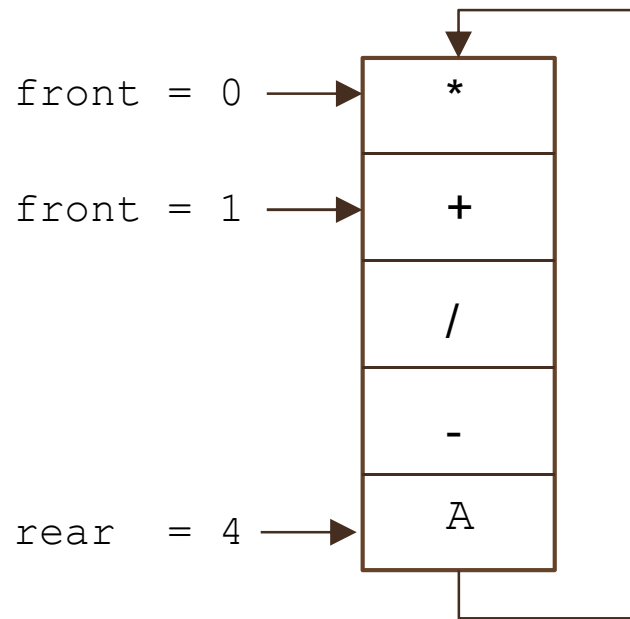
```
q.offer('A');
```

```
size      = 5
```

```
capacity = 5
```

```
⇒ public boolean offer(E item) {  
    ⇒ if (size == capacity) {  
        reallocate();  
    }  
    size++;  
    rear = (rear + 1) % capacity;  
    theData[rear] = item;  
    ⇒ return true;  
}
```

Implementing a Queue Using a Circular Array (cont.)



result = '*'

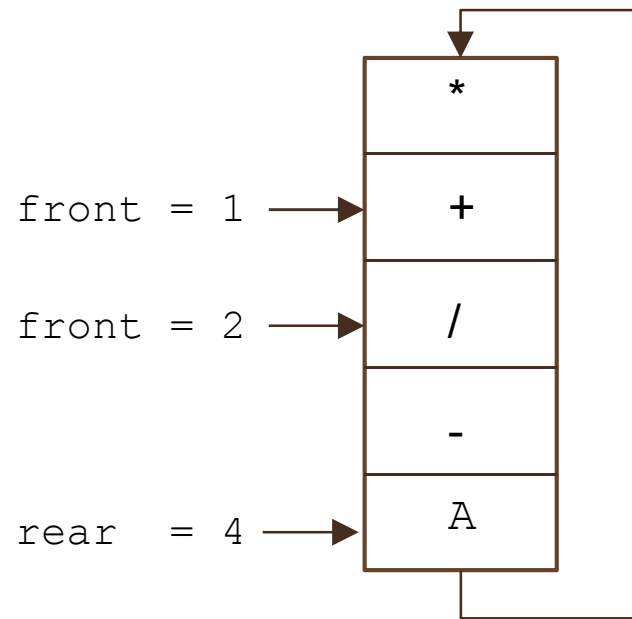
```
next = q.poll();
```

size = 4

capacity = 5

```
⇒ public E poll() {  
    ⇒ if (size == 0) {  
        return null  
    }  
    E result = theData[front];  
    front = (front + 1) % capacity;  
    size--;  
    ⇒ return result;  
}
```

Implementing a Queue Using a Circular Array (cont.)



result = '+'

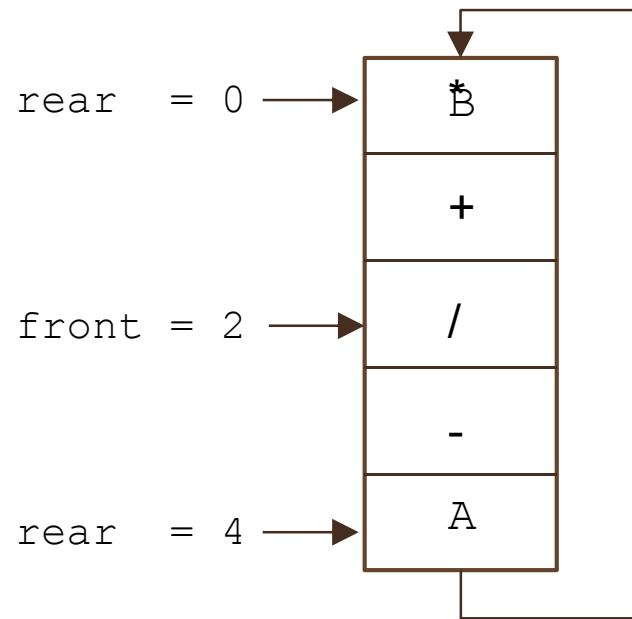
```
next = q.poll();
```

```
size      = 3
```

```
capacity = 5
```

```
⇒ public E poll() {  
    ⇒ if (size == 0) {  
        return null  
    }  
    E result = theData[front];  
    front = (front + 1) % capacity;  
    size--;  
    ⇒ return result;  
}
```

Implementing a Queue Using a Circular Array (cont.)



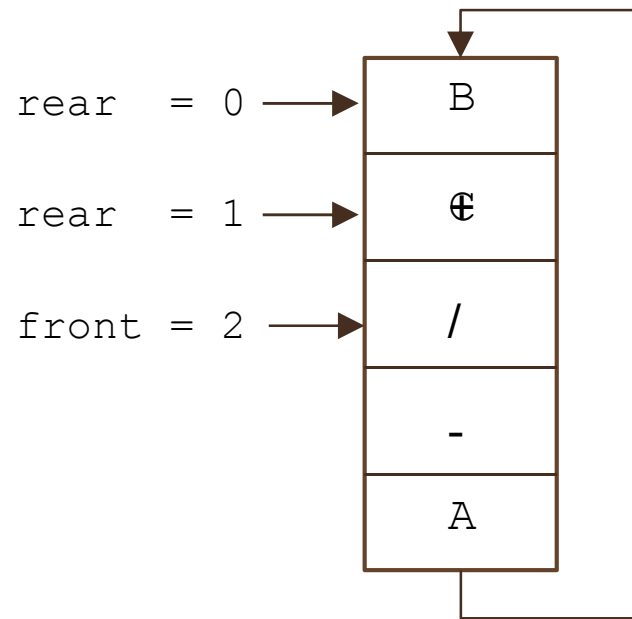
```
q.offer('B');
```

```
size      = 4
```

```
capacity = 5
```

```
⇒ public boolean offer(E item) {  
    ⇒ if (size == capacity) {  
        reallocate();  
    }  
    size++;  
    rear = (rear + 1) % capacity;  
    theData[rear] = item;  
    ⇒ return true;  
}
```

Implementing a Queue Using a Circular Array (cont.)



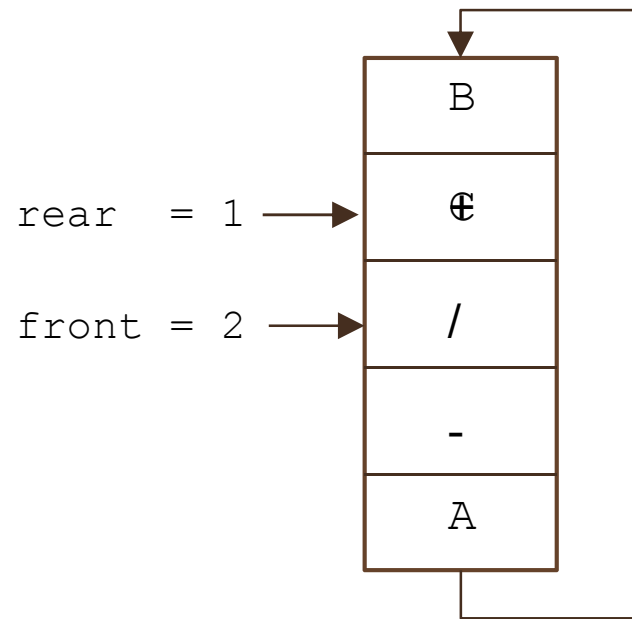
```
q.offer('C');
```

```
size      = 5
```

```
capacity = 5
```

```
⇒ public boolean offer(E item) {  
    ⇒ if (size == capacity) {  
        reallocate();  
    }  
    size++;  
    rear = (rear + 1) % capacity;  
    theData[rear] = item;  
    ⇒ return true;  
}
```


Implementing a Queue Using a Circular Array (cont.)



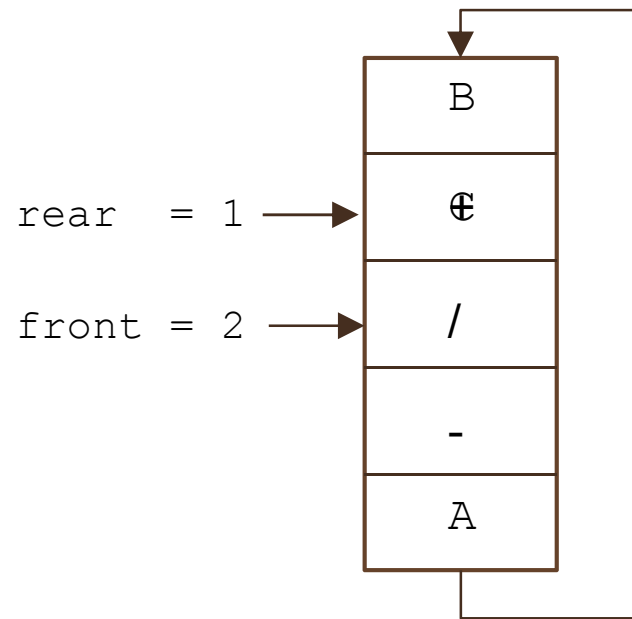
```
q.offer('D');
```

```
size      = 5
```

```
capacity = 5
```

```
⇒ public boolean offer(E item) {  
    ⇒ if (size == capacity) {  
    ⇒     reallocate();  
    }  
    size++;  
    rear = (rear + 1) % capacity;  
    theData[rear] = item;  
    return true;  
}
```

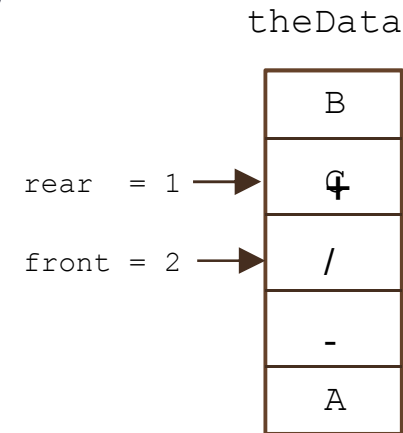
Implementing a Queue Using a Circular Array (cont.)



`q.offer('D');`

`size = 5`

`capacity = 5`



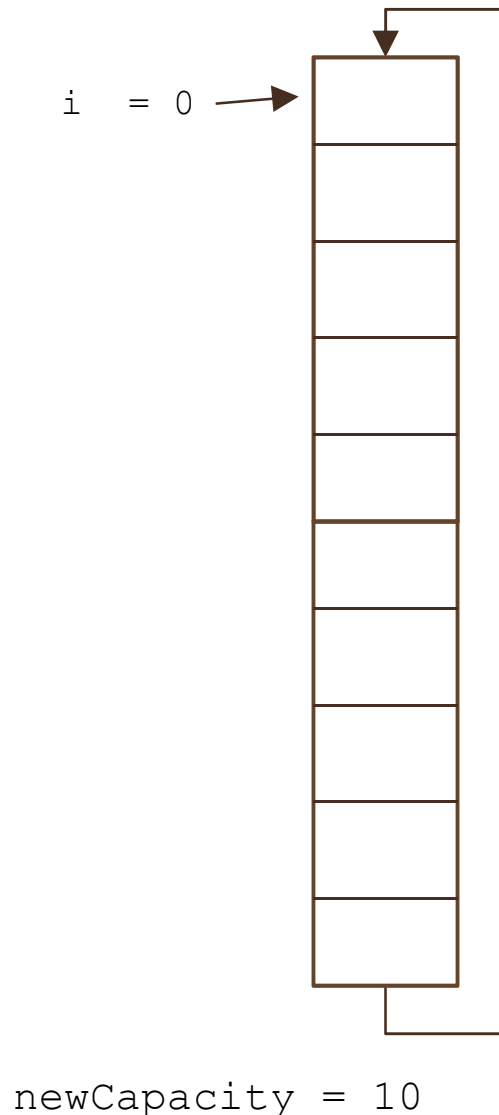
```

=> private void reallocate() {
=>     int newCapacity = 2 * capacity;
    E[] newData = (E[])new Object[newCapacity];
    int j = front;
    for (int i = 0; i < size; i++) {
        newData[i] = theData[j];
        j = (j + 1) % capacity;
    }
    front = 0;
    rear = size - 1;
    capacity = newCapacity;
    theData = newData;
}

```

`newCapacity = 10`

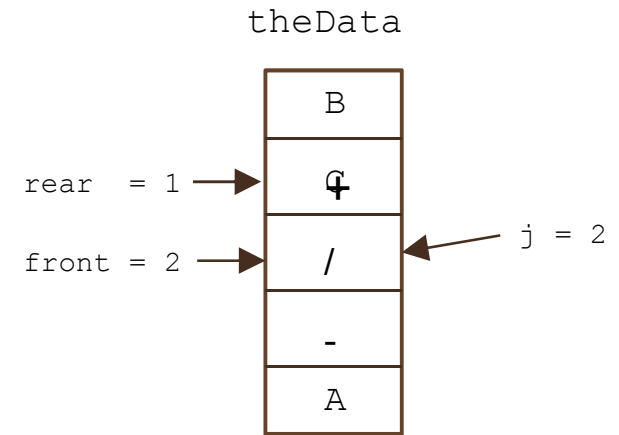
Implementing a Queue Using a Circular Array (cont.)



`q.offer('D');`

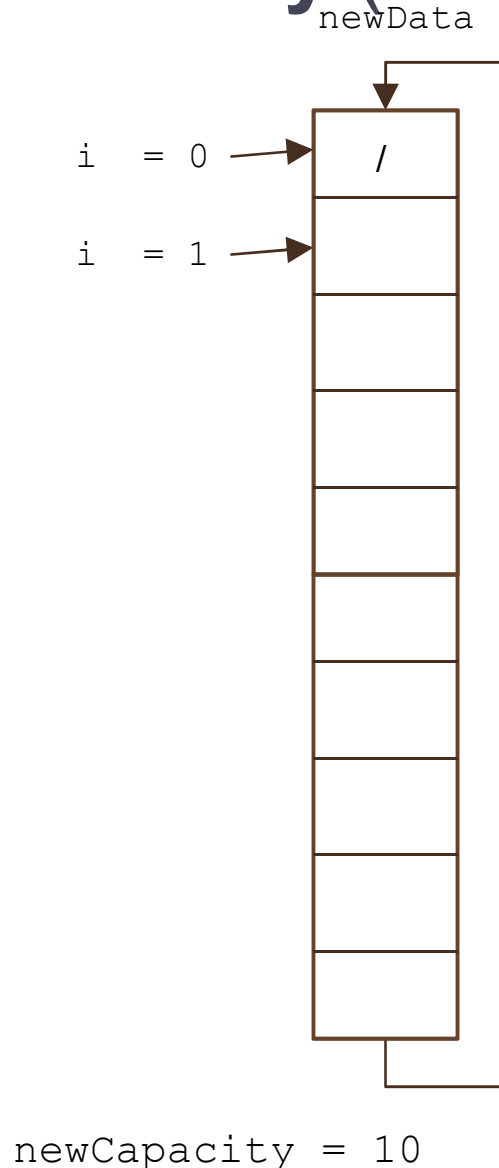
`size = 5`

`capacity = 5`



```
private void reallocate() {
    int newCapacity = 2 * capacity;
    E[] newData = (E[])new Object[newCapacity];
    int j = front;
    for (int i = 0; i < size; i++) {
        newData[i] = theData[j];
        j = (j + 1) % capacity;
    }
    front = 0;
    rear = size - 1;
    capacity = newCapacity;
    theData = newData;
}
```

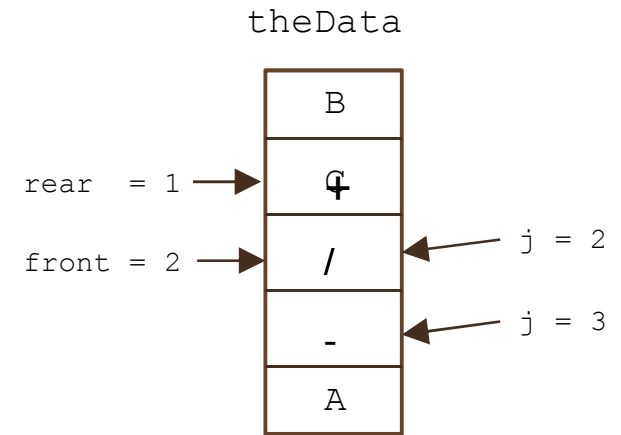
Implementing a Queue Using a Circular Array (cont.)



q.offer('D');

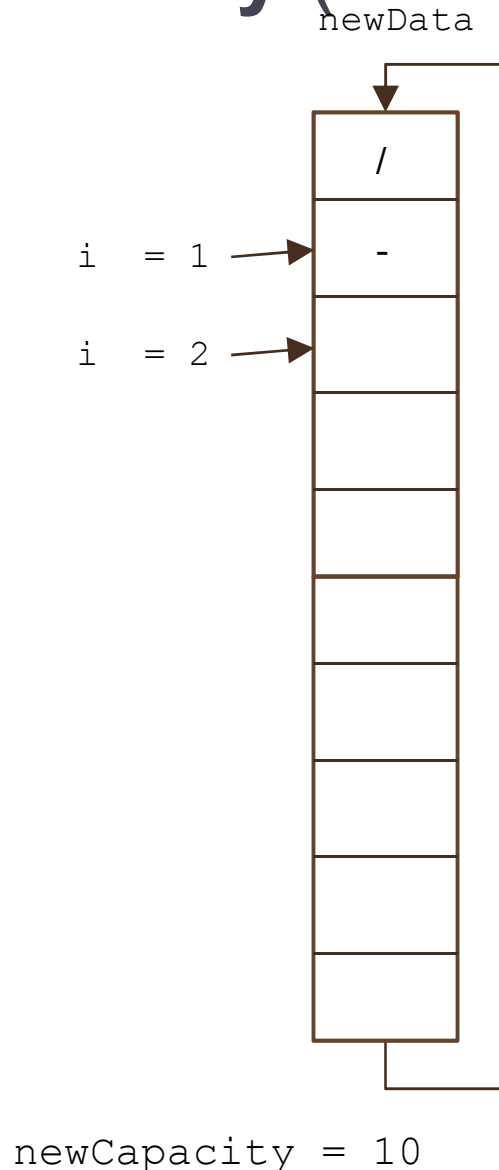
size = 5

capacity = 5



```
private void reallocate() {
    int newCapacity = 2 * capacity;
    E[] newData = (E[])new Object[newCapacity];
    int j = front;
    → for (int i = 0; i < size; i++) {
        newData[i] = theData[j];
        j = (j + 1) % capacity;
    }
    front = 0;
    rear = size - 1;
    capacity = newCapacity;
    theData = newData;
}
```

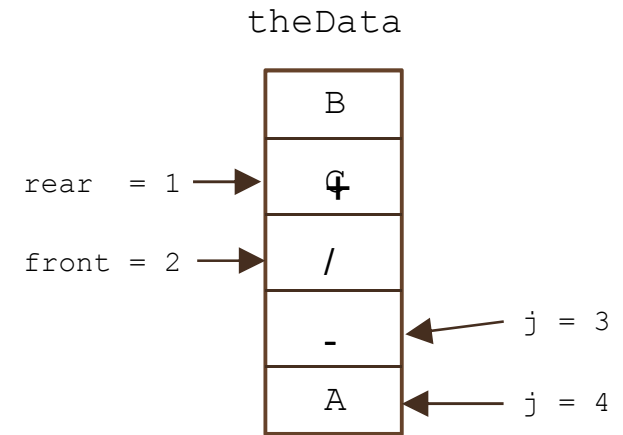
Implementing a Queue Using a Circular Array (cont.)



`q.offer('D');`

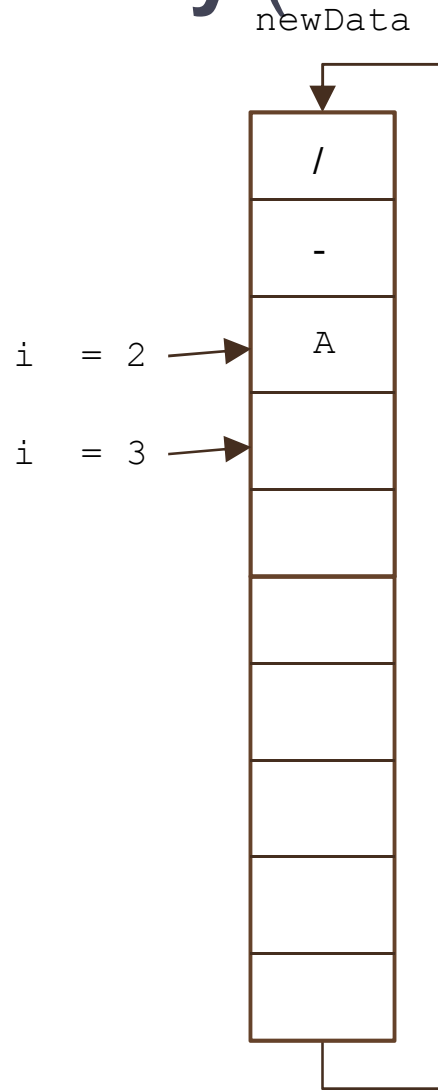
size = 5

capacity = 5



```
private void reallocate() {
    int newCapacity = 2 * capacity;
    E[] newData = (E[])new Object[newCapacity];
    int j = front;
    → for (int i = 0; i < size; i++) {
        newData[i] = theData[j];
        j = (j + 1) % capacity;
    }
    front = 0;
    rear = size - 1;
    capacity = newCapacity;
    theData = newData;
}
```

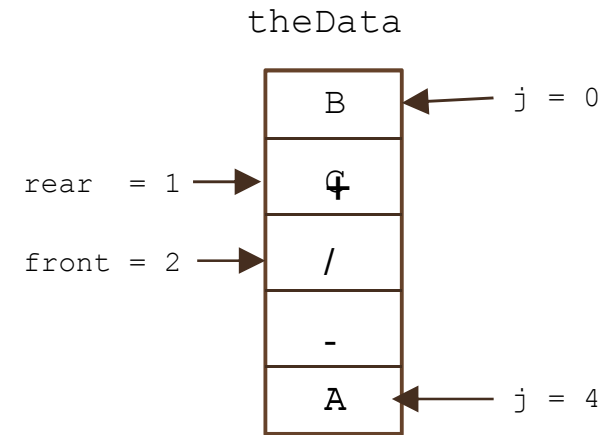
Implementing a Queue Using a Circular Array (cont.)



q.offer('D');

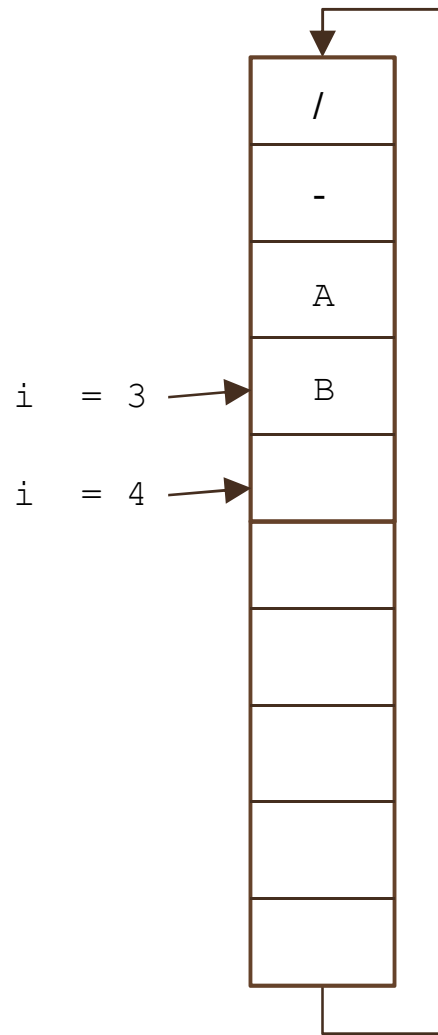
size = 5

capacity = 5



```
private void reallocate() {
    int newCapacity = 2 * capacity;
    E[] newData = (E[])new Object[newCapacity];
    int j = front;
    → for (int i = 0; i < size; i++) {
        newData[i] = theData[j];
        j = (j + 1) % capacity;
    }
    front = 0;
    rear = size - 1;
    capacity = newCapacity;
    theData = newData;
}
```

Implementing a Queue Using a Circular Array (cont.)

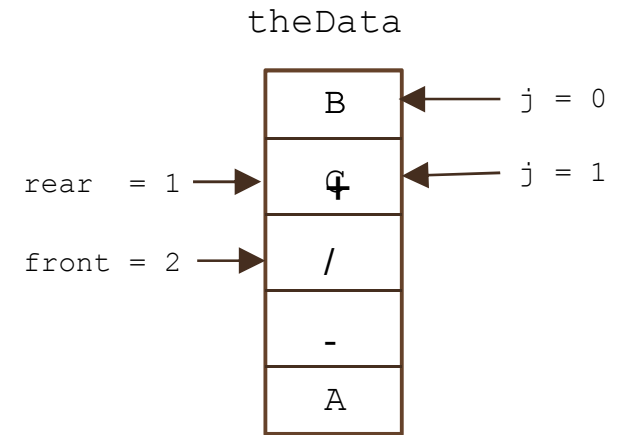


newCapacity = 10

```
q.offer('D');
```

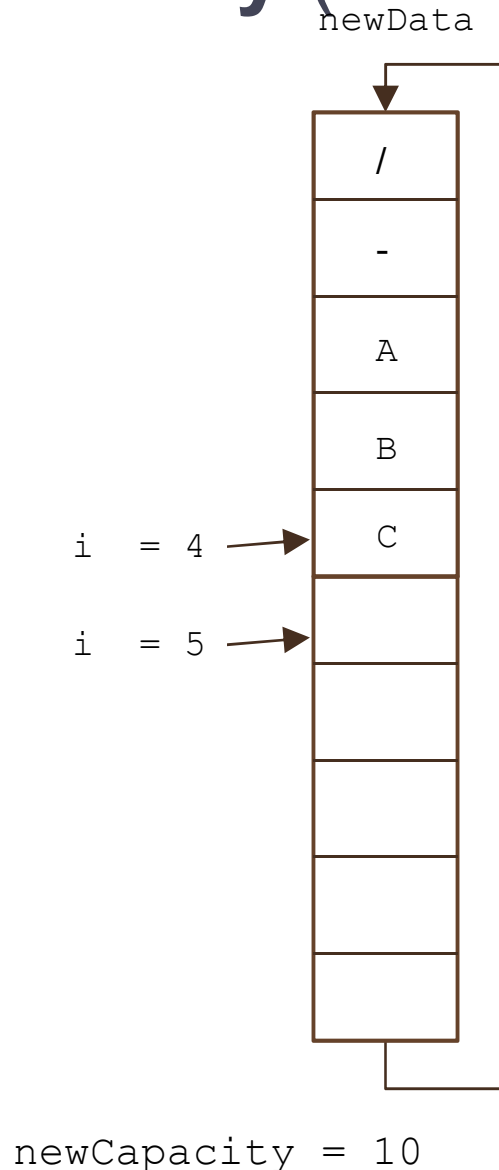
```
size      = 5
```

```
capacity = 5
```



```
private void reallocate() {
    int newCapacity = 2 * capacity;
    E[] newData = (E[])new Object[newCapacity];
    int j = front;
    → for (int i = 0; i < size; i++) {
        newData[i] = theData[j];
        j = (j + 1) % capacity;
    }
    front = 0;
    rear = size - 1;
    capacity = newCapacity;
    theData = newData;
}
```

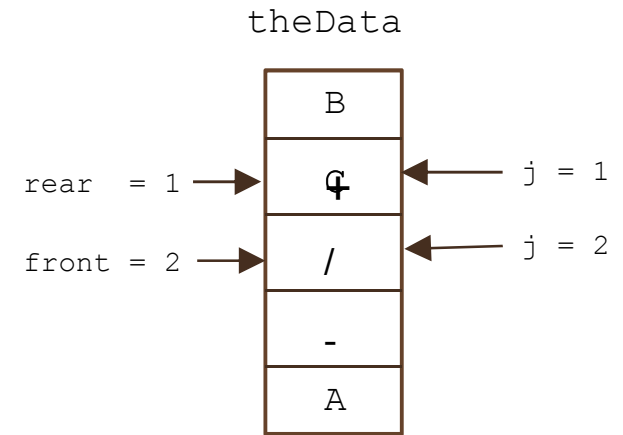
Implementing a Queue Using a Circular Array (cont.)



q.offer('D');

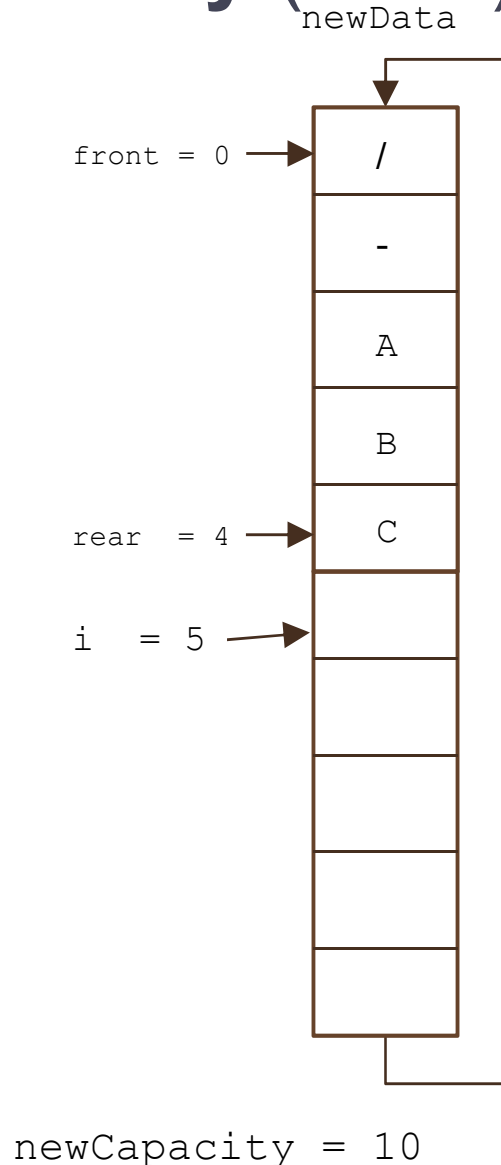
size = 5

capacity = 5



```
private void reallocate() {
    int newCapacity = 2 * capacity;
    E[] newData = (E[])new Object[newCapacity];
    int j = front;
    → for (int i = 0; i < size; i++) {
        newData[i] = theData[j];
        j = (j + 1) % capacity;
    }
    front = 0;
    rear = size - 1;
    capacity = newCapacity;
    theData = newData;
}
```

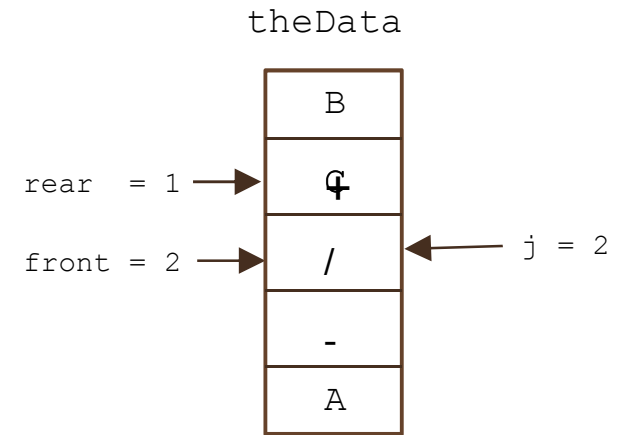

Implementing a Queue Using a Circular Array (cont.)



```
q.offer('D');
```

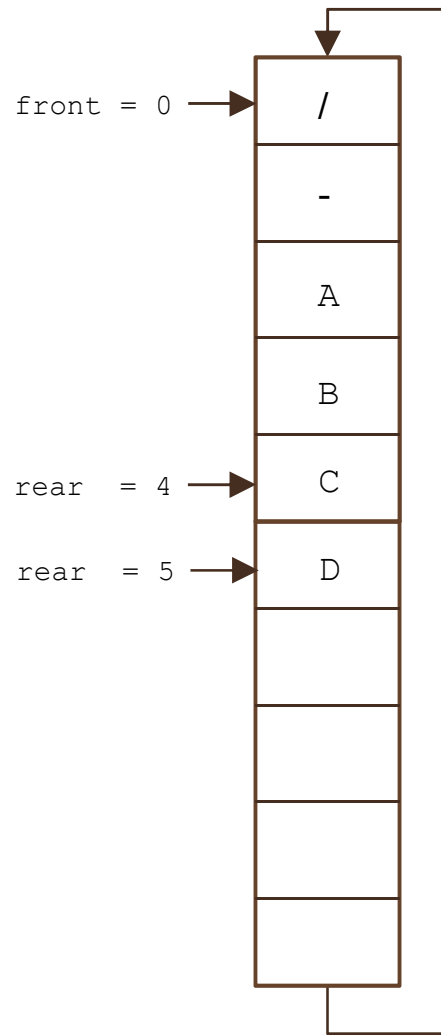
```
size      = 5
```

```
capacity = 10
```



```
private void reallocate() {
    int newCapacity = 2 * capacity;
    E[] newData = (E[])new Object[newCapacity];
    int j = front;
    for (int i = 0; i < size; i++) {
        newData[i] = theData[j];
        j = (j + 1) % capacity;
    }
    front = 0;
    rear = size - 1;
    capacity = newCapacity;
    theData = newData;
}
```

Implementing a Queue Using a Circular Array (cont.)



```
q.offer('D');
```

```
size      = 6
```

```
capacity = 10
```

```
public boolean offer(E item) {  
    if (size == capacity) {  
        reallocate();  
    }  
    size++;  
    rear = (rear + 1) % capacity;  
    theData[rear] = item;  
    return true;  
}
```

Implementing a Queue Using a Circular Array (cont.)

- Listing 4.4 (`ArrayQueue`, pages 212-214)

Implementing Class

ArrayQueue<E>.Iter (cont.)

- Just as for class `ListQueue<E>`, we must implement the missing `Queue` methods and an inner class `Iter` to fully implement the `Queue` interface

```
private class Iter implements
Iterator<E> {
    private int index;
    private int count = 0;

    public Iter() {
        index = front;
    }

    @Override
    public boolean hasNext() {
        return count < size;
    }

    ....
}
```

Implementing Class

ArrayQueue<E>.Iter (cont.)

- Just as for class `ListQueue<E>`, we must implement the missing `Queue` methods and an inner class `Iter` to fully implement the `Queue` interface

`index` stores the subscript of the next element to be accessed

```
private class Iter implements
Iterator<E> {
    private int index;
    private int count = 0;

    public Iter() {
        index = front;
    }

    @Override
    public boolean hasNext() {
        return count < size;
    }
}
```

....

Implementing Class

ArrayQueue<E>.Iter (cont.)

- Just as for class `ListQueue<E>`, we must implement the missing `Queue` methods and an inner class `Iter` to fully implement the `Queue` interface

The constructor initializes `index` to `front` when a new `Iter` object is created

```
private class Iter implements  
    Iterator<E> {
```

```
    private int index;  
    private int count = 0;
```

```
    public Iter() {  
        index = front;  
    }
```

```
    @Override  
    public boolean hasNext() {  
        return count < size;  
    }
```

```
    ....
```

Implementing Class

ArrayQueue<E>.Iter (cont.)

- Just as for class `ListQueue<E>`, we must implement the missing `Queue` methods and an inner class `Iter` to fully implement the `Queue` interface

`count` keeps track of the number of items accessed so far

```
private class Iter implements
Iterator<E> {
    private int index;
    private int count = 0;

    public Iter() {
        index = front;
    }

    @Override
    public boolean hasNext() {
        return count < size;
    }

    ....
}
```

Implementing Class

ArrayQueue<E>.Iter (cont.)

- Just as for class `ListQueue<E>`, we must implement the missing `Queue` methods and an inner class `Iter` to fully implement the `Queue` interface

`hasNext()` returns true if count is less than size

```
private class Iter implements  
    Iterator<E> {
```

```
    private int index;
```

```
    private int count = 0;
```

```
    public Iter() {
```

```
        index = front;
```

```
    }
```

```
    @Override
```

```
    public boolean hasNext() {
```

```
        return count < size;
```

```
    }
```

```
    ....
```


Implementing Class

ArrayQueue<E>.Iter (cont.)

- Just as for class `ListQueue<E>`, we must implement the missing `Queue` methods and an inner class `Iter` to fully implement the `Queue` interface

`next()` returns the element at position `index` and increments `Iter`'s fields `index` and `count`

```
@Override
public E next() {
    if (!hasNext()) {
        throw new
            NoSuchElementException();
    }
    E returnValue = theData[index];
    index = (index + 1) % capacity;
    count++;
    return returnValue;
}

@Override
public void remove {
    throw new
        UnsupportedOperationException();
}
}
```

Implementing Class

ArrayQueue<E>.Iter (cont.)

- Just as for class `ListQueue<E>`, we must implement the missing `Queue` methods and an inner class `Iter` to fully implement the `Queue` interface

`remove()` throws an exception because removing an item other than the first item violates the queue's contract

```
@Override
public E next() {
    if (!hasNext()) {
        throw new
            NoSuchElementException();
    }
    E returnValue = theData[index];
    index = (index + 1) % capacity;
    count++;
    return returnValue;
}

@Override
public void remove {
    throw new
        UnsupportedOperationException();
}
}
```

Comparing the Three Implementations

- Computation time
 - ▣ All three implementations are comparable in terms of computation time
 - ▣ All operations are $O(1)$ regardless of implementation
 - ▣ Although reallocating an array is $O(n)$, its is amortized over n items, so the cost per item is $O(1)$

Comparing the Three Implementations (cont.)

□ Storage

- Linked-list implementations require more storage due to the extra space required for the links
 - Each node for a single-linked list stores two references (one for the data, one for the link)
 - Each node for a double-linked list stores three references (one for the data, two for the links)
- A double-linked list requires 1.5 times the storage of a single-linked list
- A circular array that is filled to capacity requires half the storage of a single-linked list to store the same number of elements,
- but a recently reallocated circular array is half empty, and requires the same storage as a single-linked list



The Deque Interface

Section 4.4

Deque **Interface**

- A deque (pronounced "deck") is short for double-ended queue
- A double-ended queue allows insertions and removals from both ends
- The Java Collections Framework provides two implementations of the `Deque` interface
 - ▣ `ArrayDeque`
 - ▣ `LinkedList`
- `ArrayDeque` uses a resizable circular array, but (unlike `LinkedList`) does not support indexed operations
- `ArrayDeque` is the recommend implementation

Deque Interface (cont.)

Method	Behavior
<code>boolean offerFirst(E item)</code>	Inserts <code>item</code> at the front of the deque. Returns true if successful; returns false if the item could not be inserted.
<code>boolean offerLast(E item)</code>	Inserts <code>item</code> at the rear of the deque. Returns true if successful; returns false if the item could not be inserted.
<code>void addFirst(E item)</code>	Inserts <code>item</code> at the front of the deque. Throws an exception if the item could not be inserted.
<code>void addLast(E item)</code>	Inserts <code>item</code> at the rear of the deque. Throws an exception if the item could not be inserted.
<code>E pollFirst()</code>	Removes the entry at the front of the deque and returns it; returns null if the deque is empty.
<code>E pollLast()</code>	Removes the entry at the rear of the deque and returns it; returns null if the deque is empty.
<code>E removeFirst()</code>	Removes the entry at the front of the deque and returns it if the deque is not empty. If the deque is empty, throws a <code>NoSuchElementException</code> .
<code>E removeLast()</code>	Removes the item at the rear of the deque and returns it. If the deque is empty, throws a <code>NoSuchElementException</code> .
<code>E peekFirst()</code>	Returns the entry at the front of the deque without removing it; returns null if the deque is empty.
<code>E peekLast()</code>	Returns the item at the rear of the deque without removing it; returns null if the deque is empty.
<code>E getFirst()</code>	Returns the entry at the front of the deque without removing it. If the deque is empty, throws a <code>NoSuchElementException</code> .
<code>E getLast()</code>	Returns the item at the rear of the deque without removing it. If the deque is empty, throws a <code>NoSuchElementException</code> .
<code>boolean removeFirstOccurrence(Object item)</code>	Removes the first occurrence of <code>item</code> in the deque. Returns true if the item was removed.
<code>boolean removeLastOccurrence(Object item)</code>	Removes the last occurrence of <code>item</code> in the deque. Returns true if the item was removed.
<code>Iterator<E> iterator()</code>	Returns an iterator to the elements of this deque in the proper sequence.
<code>Iterator<E> descendingIterator()</code>	Returns an iterator to the elements of this deque in reverse sequential order.

Deque **Example**

Deque Method	Deque d	Effect
d.offerFirst('b')	b	'b' inserted at front
d.offerLast('y')	by	'y' inserted at rear
d.addLast('z')	byz	'z' inserted at rear
d.addFirst('a')	abyz	'a' inserted at front
d.peekFirst()	abyz	Returns 'a'
d.peekLast()	abyz	Returns 'z'
d.pollLast()	aby	Removes 'z'
d.pollFirst()	by	Removes 'a'

Deque **Interface** (cont.)

- ❑ The `Deque` interface extends the `Queue` interface, so it can be used as a queue
- ❑ A deque can be used as a stack if elements are pushed and popped from the front of the deque
- ❑ Using the `Deque` interface is preferable to using the legacy `Stack` class (based on `Vector`)

Stack Method	Equivalent Deque Method
<code>push(e)</code>	<code>addFirst(e)</code>
<code>pop()</code>	<code>removeFirst()</code>
<code>peek()</code>	<code>peekFirst()</code>
<code>empty()</code>	<code>isEmpty()</code>



Simulating Waiting Lines Using Queues

Section 4.5

Simulating Waiting Lines Using Queues

- *Simulation* is used to study the performance of a physical system by using a physical, mathematical, or computer model of the system
- Simulation allows designers of a new system to estimate the expected performance before building it
- Simulation can lead to changes in the design that will improve the expected performance of the new system
- Simulation is useful when the real system would be too expensive to build or too dangerous to experiment with after its construction

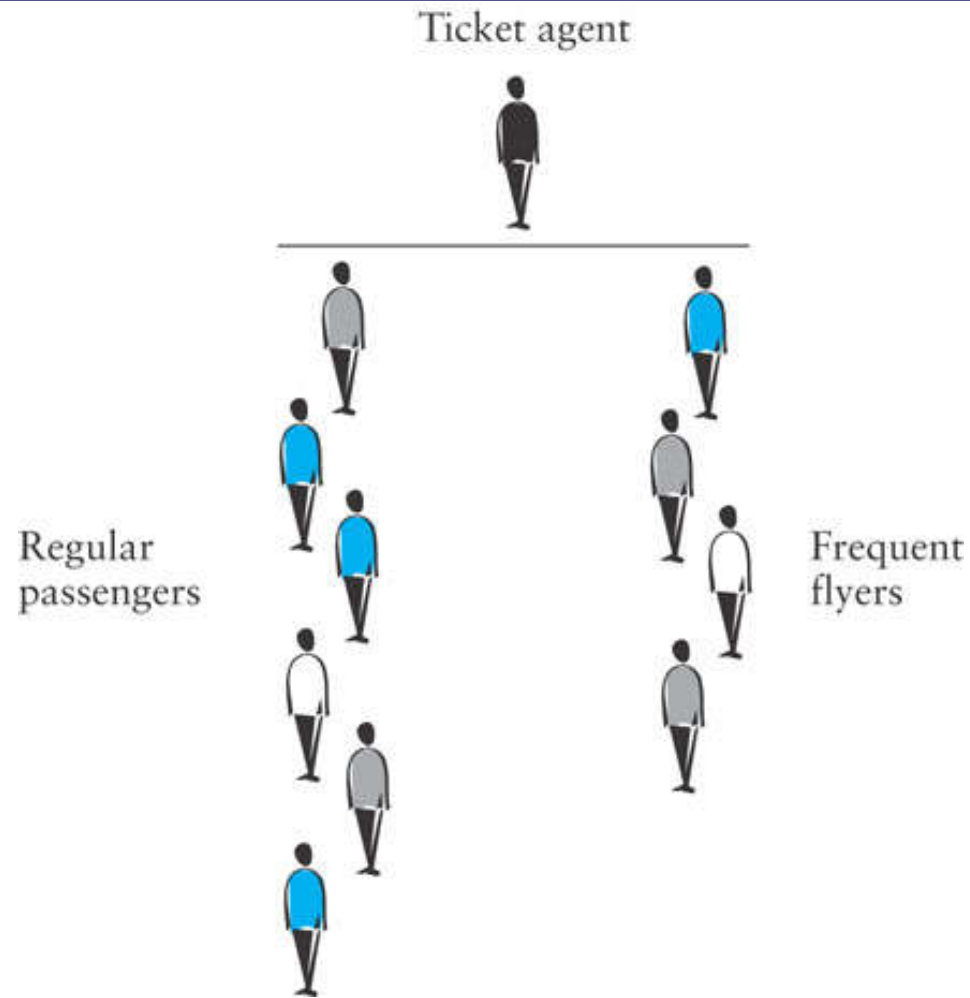
Simulating Waiting Lines Using Queues (cont.)

- System designers often use computer models to simulate physical systems
 - ▣ Example: an airline check-in counter
- A branch of mathematics called *queuing theory* studies such problems

Case Study

- Blue Skies Airlines (BSA) would like to have two waiting lines:
 - ▣ regular customers
 - ▣ frequent flyers
- Assuming only one ticket agent, BSA would like to determine the average wait time for taking passengers from the waiting lines using various strategies:
 - ▣ take turns serving passengers from both lines (one frequent flyer, one regular, one frequent flyer, etc.)
 - ▣ serve the passenger waiting the longest
 - ▣ serve any frequent flyers before serving regular passengers

Case Study (cont.)



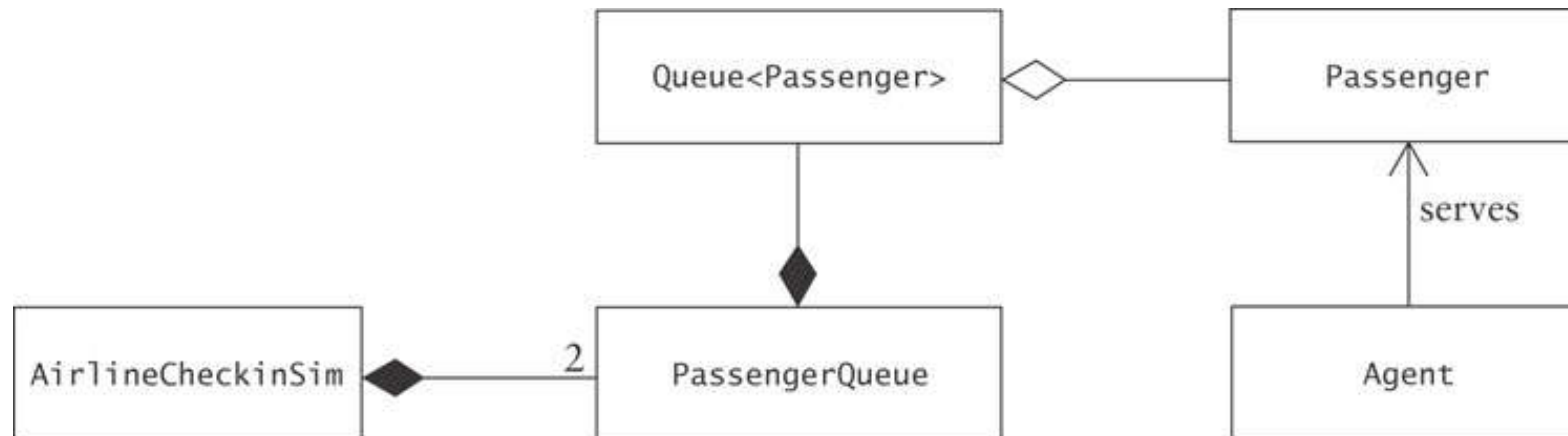
Case Study: Analysis

- To run the simulation, we must keep track of the current time by maintaining a clock set to an initial time of zero
- The clock will increase by one time unit until the simulation is finished
- During each time interval, one or more of the following events occur(s):
 1. a new frequent flyer arrives in line
 2. a new regular flyer arrives in line
 3. the ticket agent finishes serving a passenger and begins to serve a passenger from the frequent flyer line
 4. the ticket agent finishes serving a passenger and begins to serve a passenger from the regular passenger line
 5. the ticket agent is idle because there are no passengers to serve

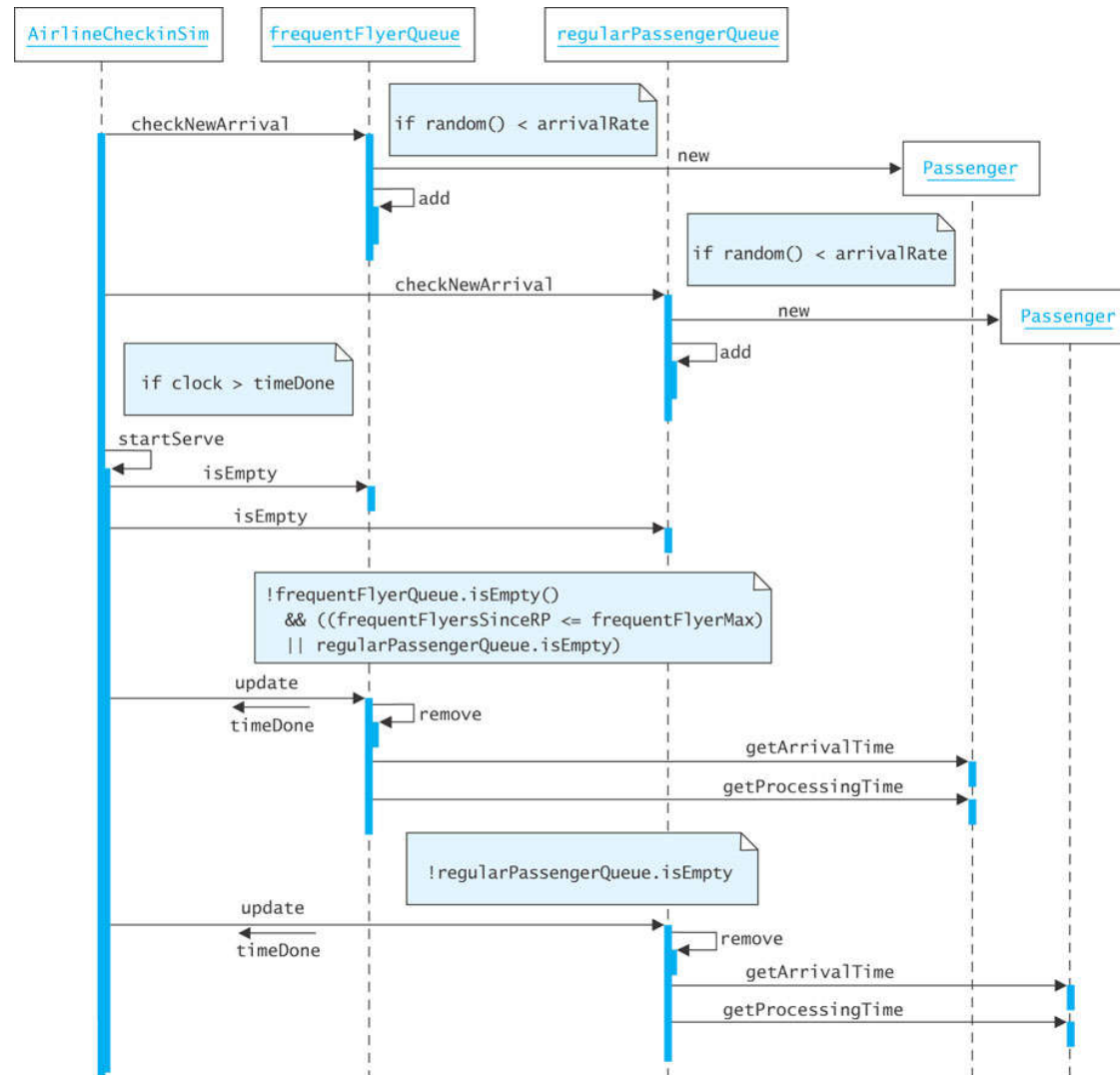
Case Study: Analysis (cont.)

- We can simulate different serving strategies by introducing a simulation variable, `frequentFlyerMax` (> 0)
- `frequentFlyerMax` represents the number of consecutive frequent flyer passengers served between regular passengers
- When `frequentFlyerMax` is:
 - 1, every other passenger served will be a regular passenger
 - 2, every third passenger served will be a regular passenger
 - a very large number, any frequent flyers will be served before regular passengers

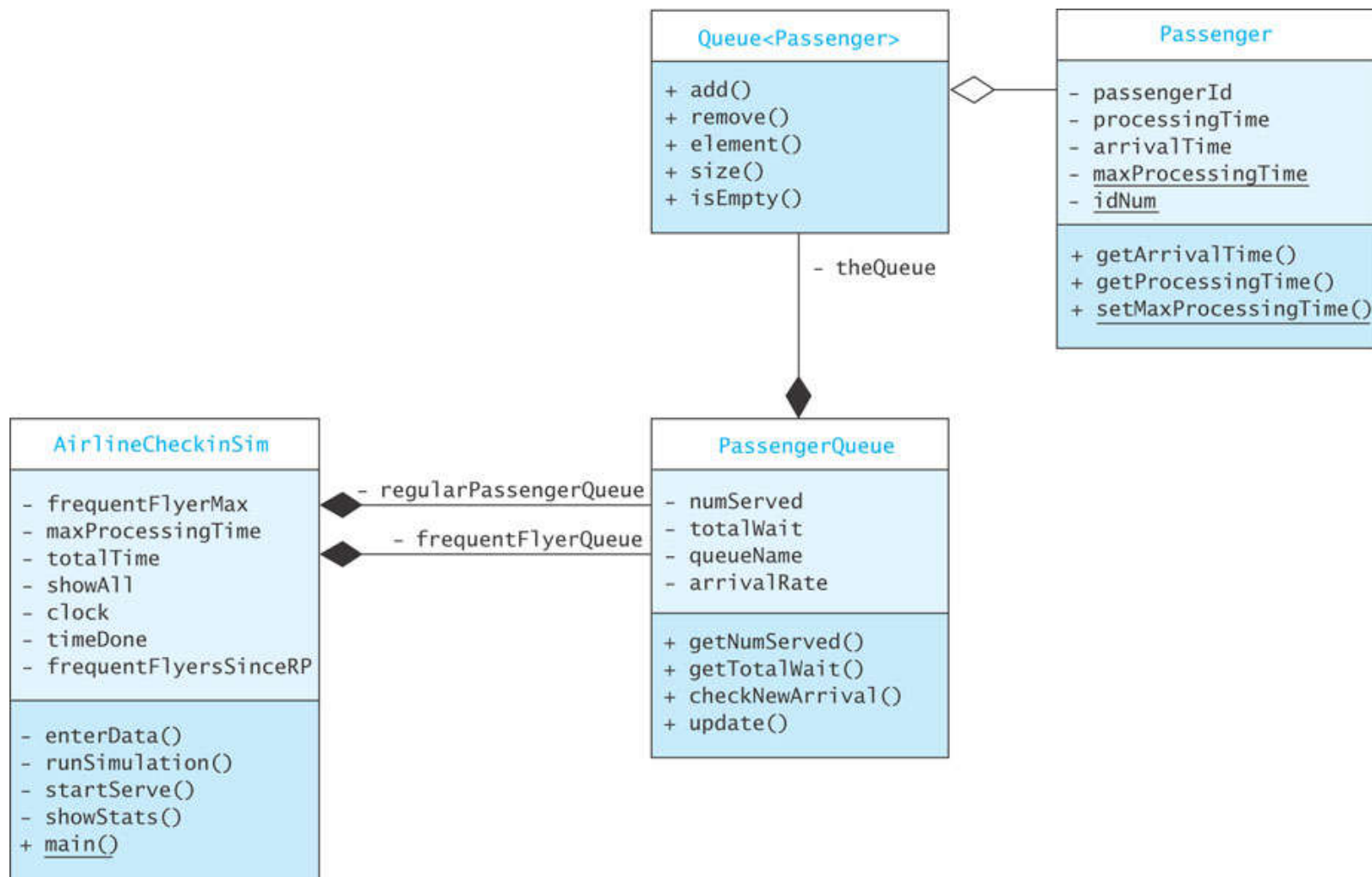
Case Study: Design (cont.)



Case Study: Design (cont.)



Case Study: Design (cont.)



Case Study: Design (cont.)

Data Field	Attribute
<code>private PassengerQueue frequentFlyerQueue</code>	The queue of frequent flyers.
<code>private PassengerQueue regularPassengerQueue</code>	The queue of regular passengers.
<code>private int frequentFlyerMax</code>	The maximum number of frequent flyers to serve between regular passengers.
<code>private int maxProcessingTime</code>	The maximum time to serve a passenger.
<code>private int totalTime</code>	The total time to run the simulation.
<code>private boolean showAll</code>	A flag indicating whether to trace the simulation.
<code>private int clock</code>	The current clock time (initially zero).
<code>private int timeDone</code>	The time that the current passenger will be finished.
<code>private int frequentFlyersSinceRP</code>	The number of frequent flyers served since the last regular passenger.
Method	Behavior
<code>public static void main(String[] args)</code>	Starts the execution of the simulation by calling <code>enterData</code> and <code>runSimulation</code> .
<code>private void runSimulation()</code>	Controls the simulation. Executes the steps shown in Figure 4.15.
<code>private void enterData()</code>	Reads in the data for the simulation.
<code>private void startServe()</code>	Initiates service for a passenger.
<code>private void showStats()</code>	Displays the summary statistics.

Case Study: Design (cont.)

Data Field	Attribute
<code>private Queue<Passenger> theQueue</code>	The queue of passengers.
<code>private int numServed</code>	The number from this queue who were served.
<code>private int totalWait</code>	The total time spent waiting by passengers who were in this queue.
<code>private String queueName</code>	The name of this queue.
<code>private double arrivalRate</code>	The arrival rate for this queue.
Method	Behavior
<code>public PassengerQueue(String queueName)</code>	Constructs a new queue with the specified name.
<code>private void checkNewArrival(int clock, boolean showAll)</code>	Checks whether there was a new arrival for this queue and, if so, inserts the passenger into the queue.
<code>private int update(int clock, boolean showAll)</code>	Updates the total waiting time and number of passengers served when a passenger from this queue is served.
<code>public int getTotalWait()</code>	Returns the total waiting time for passengers in this queue.
<code>public int getNumServed()</code>	Returns the number of passengers served from this queue.

Case Study: Design (cont.)

Method	Behavior
<code>public Passenger(int arrivalTime)</code>	Constructs a new passenger, assigns it a unique ID and the specified arrival time. Computes a random processing time in the range 1 to <code>maxProcessingTime</code> .
<code>public int getArrivalTime()</code>	Returns the value of <code>arrivalTime</code> .
<code>public int getProcessingTime()</code>	Returns the value of <code>processingTime</code> .
<code>public static void setMaxProcessingTime(int maxProcessingTime)</code>	Sets the <code>maxProcessingTime</code> used to generate the random processing time.

Case Study: Design (cont.)

Internal Variable	Attribute	Conversion
frequentFlyerQueue.arrivalRate	Expected number of frequent flyer arrivals per hour.	Divide input by 60 to obtain arrivals per minute.
regularPassengerQueue.arrivalRate	Expected number of regular passenger arrivals per hour.	Divide input by 60 to obtain arrivals per minute.
maxProcessingTime	Maximum service time in minutes.	None.
totalTime	Total simulation time in minutes.	None.
showAll	Flag. If true , display minute-by-minute trace of simulation.	Input beginning with 'Y' or 'y' will set this to true ; other inputs will set it to false .

Case Study: Implementation

- Listing 4.6 (`Passenger.java`, pages 234-235; `PassengerQueue.java`, page 231, `AirlineCheckinSim.java`, page 228)

Case Study: Testing

```
Command Prompt
Expected number of frequent flyer arrivals per hour: 15
Expected number of regular passenger arrivals per hour: 30
The maximum number of frequent flyers
served between regular passengers: 5
Maximum service time in minutes: 4
The total simulation time in minutes: 10
Display minute-by-minute trace of simulation (Y or N): y
Time is 0: Server is idle
Time is 1: Regular Passenger arrival, new queue size is 1
Time is 1: Serving Regular Passenger with time stamp 1, service time is 3
Time is 3: Regular Passenger arrival, new queue size is 1
Time is 4: Serving Regular Passenger with time stamp 3, service time is 1
Time is 5: Frequent Flyer arrival, new queue size is 1
Time is 5: Serving Frequent Flyer with time stamp 5, service time is 1
Time is 6: Regular Passenger arrival, new queue size is 1
Time is 6: Serving Regular Passenger with time stamp 6, service time is 3
Time is 7: Regular Passenger arrival, new queue size is 1
Time is 8: Regular Passenger arrival, new queue size is 2
Time is 9: Frequent Flyer arrival, new queue size is 1
Time is 9: Regular Passenger arrival, new queue size is 3
Time is 9: Serving Frequent Flyer with time stamp 9, service time is 1

The number of regular passengers served was 3
with an average waiting time of 0.3333333333333333
The number of frequent flyers served was 2
with an average waiting time of 0.0
Passengers in frequent flyer queue: 0
Passengers in regular queue: 3
```