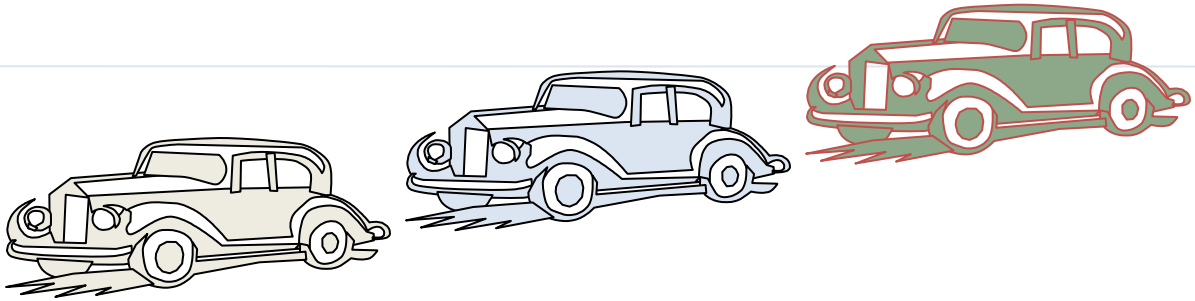
QUEUES



Presentation for use with the textbook

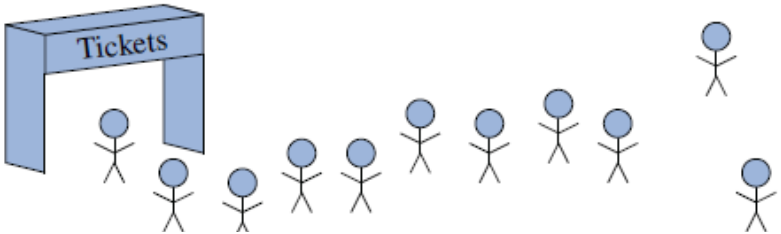
1. Data Structures and Algorithms in Java, 6th edition, by M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, Wiley, 2014
2. Data Structures Abstraction and Design Using Java, 2nd Edition by Elliot B. Koffman & Paul A. T. Wolfgang, Wiley, 2010

QUEUE

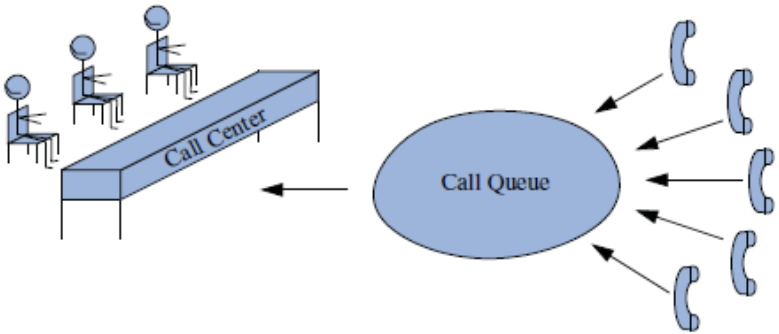
- The Queue ADT stores arbitrary objects
- The queue, like the stack, is a widely used data structure
- A queue differs from a stack in one important way Insertions and deletions follow
  - A stack is LIFO list – *Last-In, First-Out*
  - while a queue is FIFO list, *First-In, First-Out*
- <u>Insertions are at the rear of the queue and removals are at the front of the queue</u>

APPLICATIONS OF QUEUES

- Direct applications
  - Waiting lists, bureaucracy
  - Access to shared resources (e.g., printer)
  - Multiprogramming
- Indirect applications
  - Auxiliary data structure for algorithms
  - Component of other data structures



People waiting in line to purchase tickets;



phone calls being routed to a customer service center.

THE QUEUE ADT

- Main queue operations:
  - **enqueue(object):** inserts an element at the end of the queue
  - **object dequeue():** removes and returns the element at the front of the queue

```
public interface Queue<E> {
  /** Returns the number of elements in the queue. */
  int size();
  /** Tests whether the queue is empty. */
  boolean isEmpty();
  /** Inserts an element at the rear of the queue. */
  void enqueue(E e);
  /** Returns, but does not remove, the first element of the queue (null if empty). */
  E first();
  /** Removes and returns the first element of the queue (null if empty). */
  E dequeue();
}
```

- Auxiliary queue operations:
  - object **first**(): returns the element at the front without removing it
  - integer **size**(): returns the number of elements stored
  - boolean **isEmpty**(): indicates whether no elements are stored
- Boundary cases:
  - Attempting the execution of dequeue or first on an empty queue returns null

EXAMPLE

Operation		***Output***	*Q*
enqueue(5)		–	(5)
enqueue(3)		–	(5, 3)
dequeue()		*5*	(3)
enqueue(7)		–	(3, 7)
dequeue()		*3*	(7)
first()		*7*	(7)
dequeue()		*7*	()
dequeue()		*null*	()
isEmpty()	*true*	()	
enqueue(9)		–	(9)
enqueue(7)		–	(9, 7)
size()		*2*	(9, 7)
enqueue(3)		–	(9, 7, 3)
enqueue(5)		–	(9, 7, 3, 5)
dequeue()		*9*	(7, 3, 5)

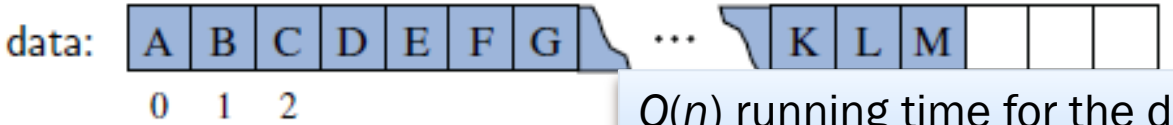
COMPARISON TO JAVA.UTIL.QUEUE

- Our Queue methods and corresponding methods of java.util.Queue:

Our Queue ADT	Interface java.util.Queue	
	throws exceptions	returns special value
enqueue(e)	add(e)	offer(e)
dequeue( )	remove( )	poll( )
first( )	element( )	peek( )
size( )	size( )	
isEmpty( )	isEmpty( )	

ARRAY-BASED QUEUE 1

Using an array to store elements of a queue, such that the first element inserted, "A", is at cell 0, the second element inserted, "B", at cell 1, and so on.



$O(n)$ running time for the dequeue method.

ARRAY-BASED QUEUE 2

- Replace a dequeued element in the array with a null reference
- A variable to keep track of the front
  - f index of the front element

O(1) deque operation but, if we repeatedly let the front of the queue drift rightward over time, the back of the queue would reach the end of the underlying array even when there are fewer than N elements currently in the queue.

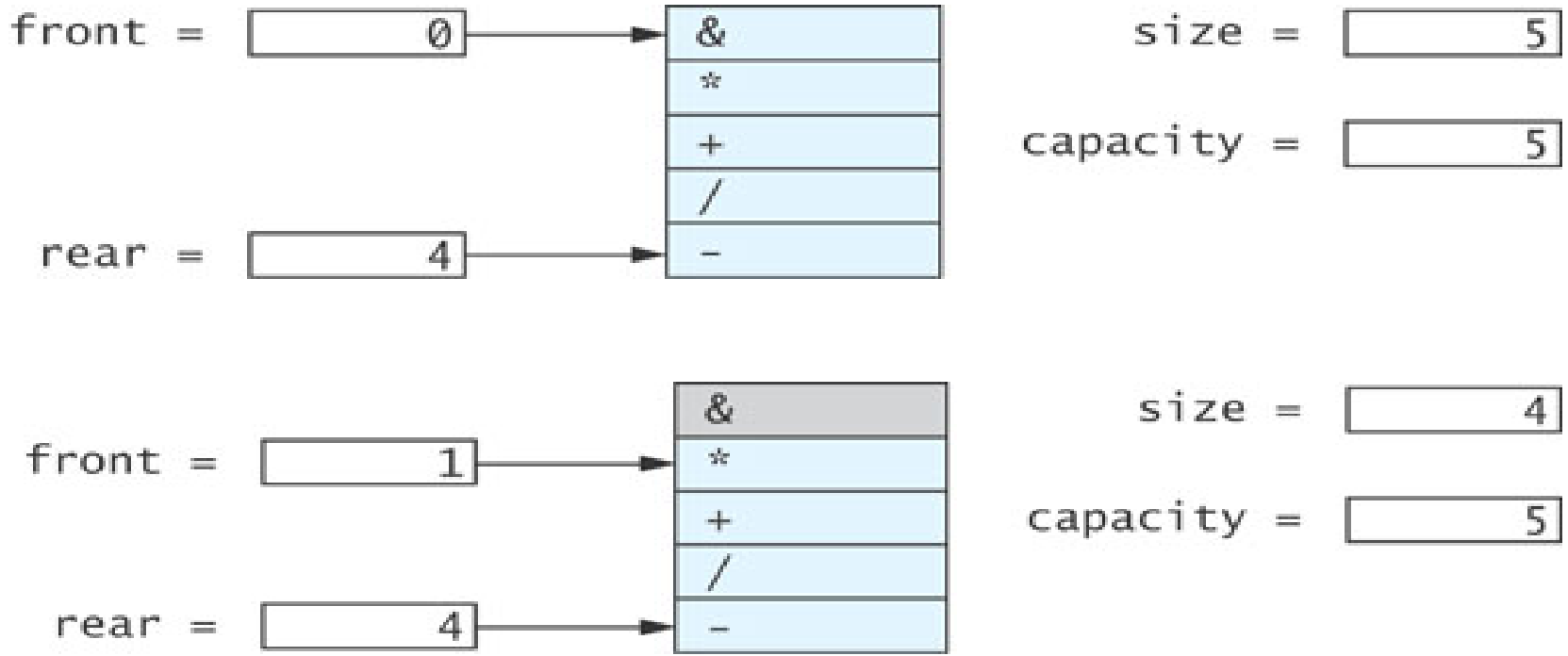
ARRAY-BASED QUEUE 3: CIRCULAR ARRAY

- Use an array of size N in a circular fashion
- Two variables keep track of <u>the front and size</u>
  - f index of the front element
  - sz number of stored elements
- How to store additional elements in such a configuration:
  - When the queue has fewer than N elements, array location $r = (f + sz) \bmod N$ is the first empty slot past the rear of the queue

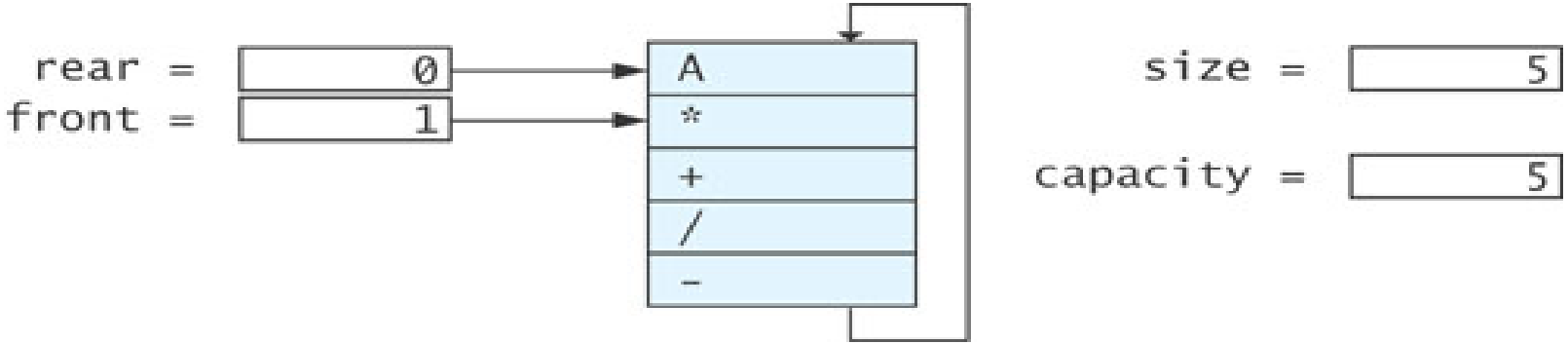
normal configuration

wrapped-around configuration

IMPLEMENTING A QUEUE USING A CIRCULAR ARRAY (CONT.)



IMPLEMENTING A QUEUE USING A CIRCULAR ARRAY (CONT.)



QUEUE INTERFACE IN JAVA

- Java interface corresponding to our Queue ADT
- Assumes that first() and dequeue() return null if queue is empty

```
public interface Queue<E> {
  int size();
  boolean isEmpty();
  E first();
  void enqueue(E e);
  E dequeue();
}
```

ARRAY QUEUE

```
/** Implementation of the queue ADT using a fixed-length array. */
public class ArrayQueue<E> implements Queue<E> {
  // instance variables
  private E[ ] data;                          // generic array used for storage
  private int f = 0;                          // index of the front element
  private int sz = 0;                         // current number of elements

  // constructors
  public ArrayQueue() {this(CAPACITY);}       // constructs queue with default capacity
  public ArrayQueue(int capacity) {           // constructs queue with given capacity
    data = (E[ ]) new Object[capacity];       // safe cast; compiler may give warning
  }
```

QUEUE OPERATIONS

- We use the **modulo** operator (%) : remainder of division

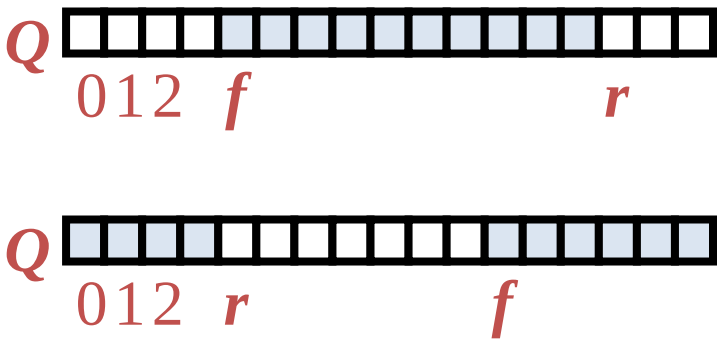
```
// methods
/** Returns the number of elements in the queue. */
public int size() { return sz; }

/** Tests whether the queue is empty. */
public boolean isEmpty() { return (sz == 0); }
```

```
/** Returns, but does not remove, the first element of the queue (null if empty). */
public E first() {
  if (isEmpty()) return null;
  return data[f];
}
```

QUEUE OPERATIONS (CONT.): **ENQUEUE**

- Operation **enqueue** throws an exception if the array is full
- This exception is implementation-dependent



Algorithm *enqueue*(o)
  if $size() = N - 1$ **then**
    throw *IllegalStateException*
  else
    $r \leftarrow (f + sz) \bmod N$
    $Q[r] \leftarrow o$
    $sz \leftarrow (sz + 1)$

QUEUE OPERATIONS (CONT.): **ENQUEUE**

- Formula for calculating the index of the new

  avail = (f + sz) % data.length;

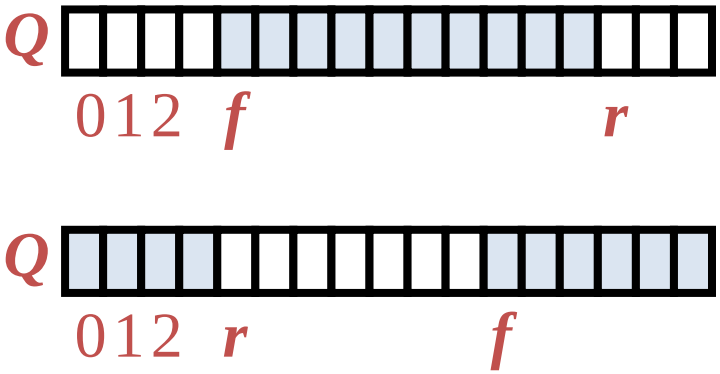
- NOTE: the size of the queue as it exists *prior* to the addition of the new elem ent.

```
/** Inserts an element at the rear of the queue. */
public void enqueue(E e) throws IllegalStateException {
  if (sz == data.length) throw new IllegalStateException("Queue is full");
  int avail = (f + sz) % data.length;      // use modular arithmetic
  data[avail] = e;
  sz++;
}
```

- Example:
  - capacity 10, current size 3, and first element at index 5, its three elements are stored at in dices 5, 6, and 7, and the next element should be added at index 8, computed as (5+3) % 10.
  - capacity 10, current size 3, and first element at index 8, its three elements are stored at in dices 8, 9, and 0, and the next element should be added at index 1, computed as (8+3) % 10.

QUEUE OPERATIONS (CONT.): **DEQUEUE**

- Note that operation **dequeue** returns NULL if the queue is empty



Algorithm ***dequeue***()
  if ***isEmpty***() **then**
    return ***null***
  else
    $o \leftarrow Q[f]$
    $f \leftarrow (f + 1) \bmod N$
    $sz \leftarrow (sz - 1)$
    return o

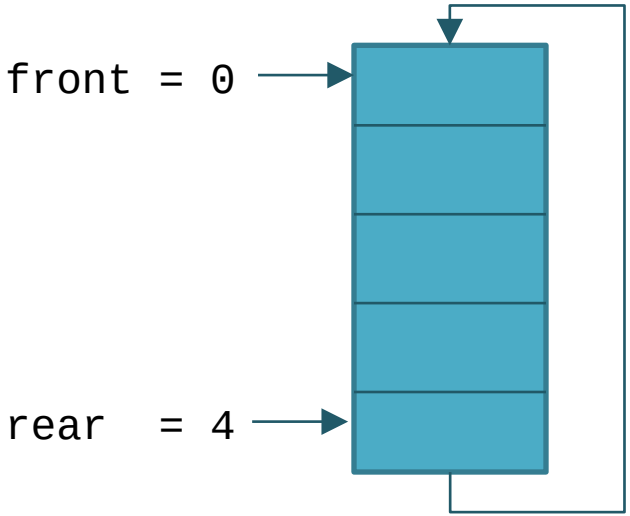
QUEUE OPERATIONS (CONT.): **DEQUEUE**

- Update of *f :* because of the possibility of a wraparound co nfiguration, we rely on modular arithmetic, computing

  f = (f+1) % data.length.

```
/** Removes and returns the first element of the queue (null if empty). */
public E dequeue() {
  if (isEmpty()) return null;
  E answer = data[f];
  data[f] = null;                    // dereference to help garbage collection
  f = (f + 1) % data.length;
  sz--;
  return answer;
}
```

IMPLEMENTING A QUEUE USING A CIRCULAR ARRAY (CONT.)



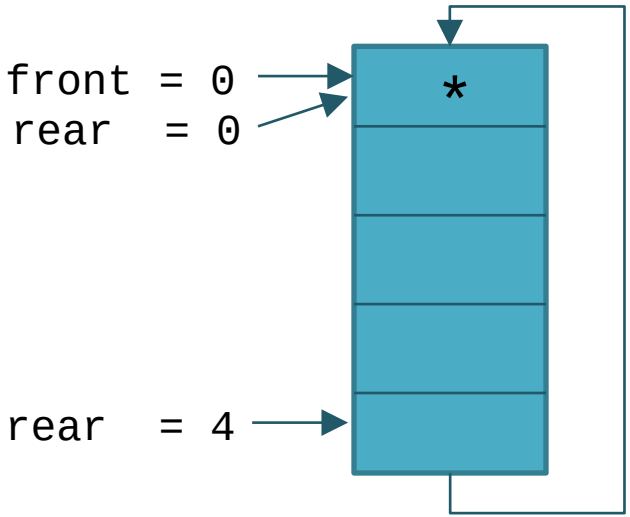

```
ArrayQueue q = new ArrayQueue(5);

size     = 0
capacity = 5
```

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

Our Queue ADT	Interface java.util.Queue	
	throws exceptions	returns special value
enqueue(e)	add(e)	offer(e)
dequeue( )	remove( )	poll( )
first( )	element( )	peek( )
size( )	size( )	
isEmpty( )	isEmpty( )	

IMPLEMENTING A QUEUE USING A CIRCULAR ARRAY (CONT.)




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

size     = 1
capacity = 5
```

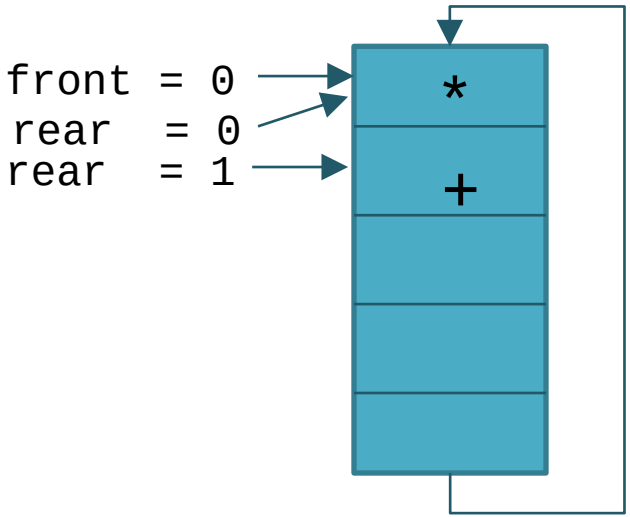
GROWABLE ARRAY

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

```
/** Inserts an element at the rear of the queue. */
public void enqueue(E e) throws IllegalStateException {
  if (sz == data.length) throw new IllegalStateException("Queue is full");
  int avail = (f + sz) % data.length;      // use modular arithmetic
  data[avail] = e;
  sz++;
}
```

CONSTANT CAPACITY

IMPLEMENTING A QUEUE USING A CIRCULAR ARRAY (CONT.)

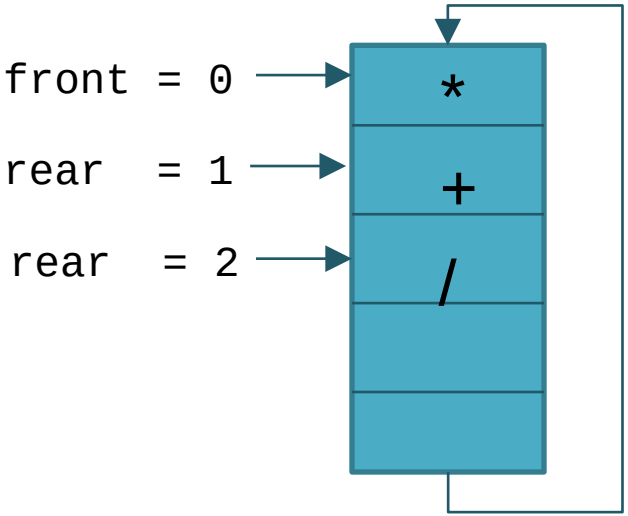



```
q.offer('+');
```

```
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.)

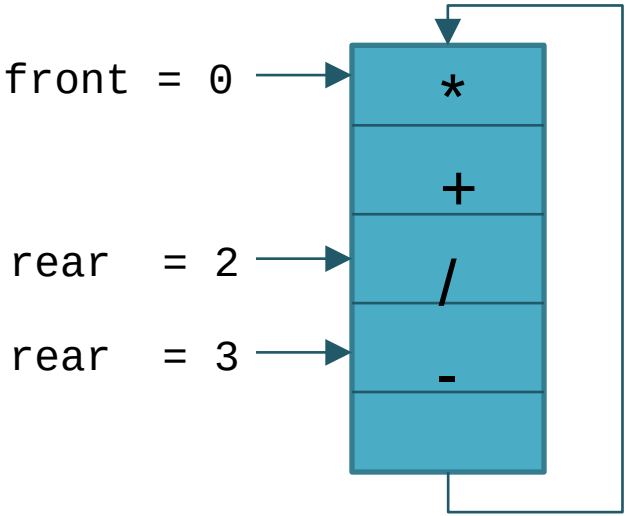



```
q.offer('/');

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.)

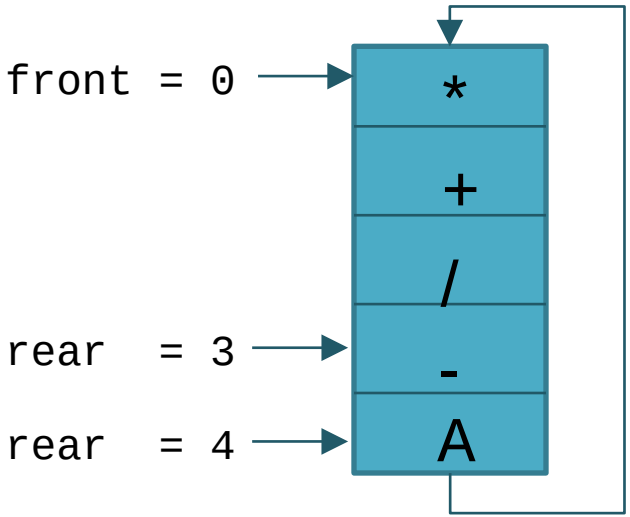



```
q.offer('-');

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.)

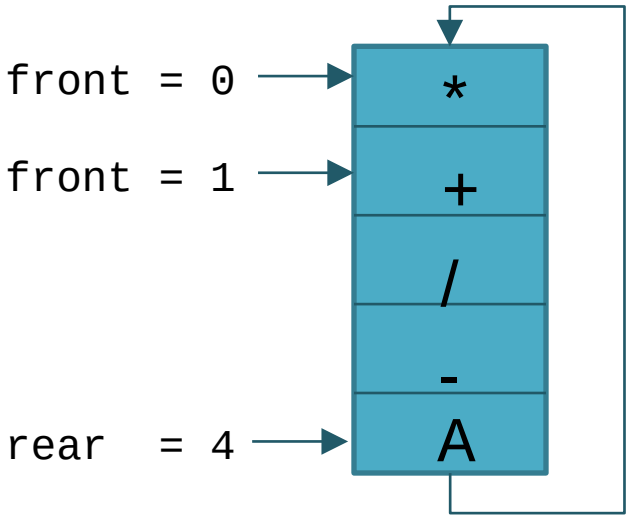



```
q.offer('A');

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.)



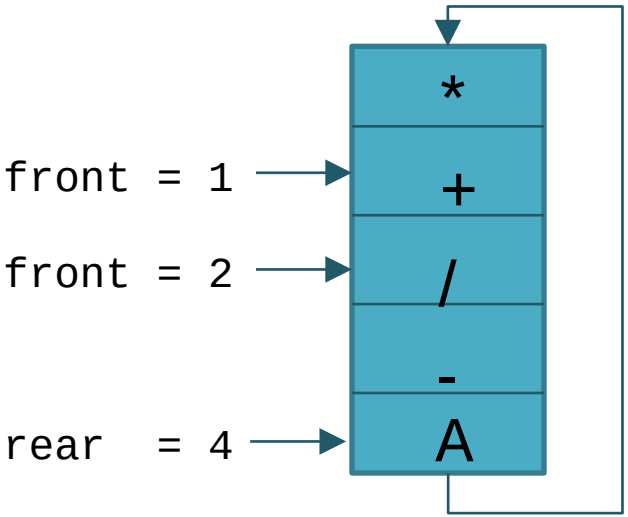

```
next = q.poll();

size     = 4
capacity = 5
```

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

result = '*'

IMPLEMENTING A QUEUE USING A CIRCULAR ARRAY (CONT.)



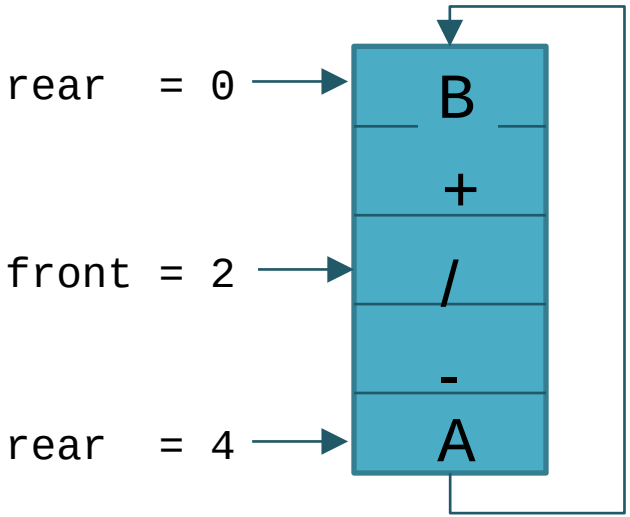
result = '+'

```
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.)

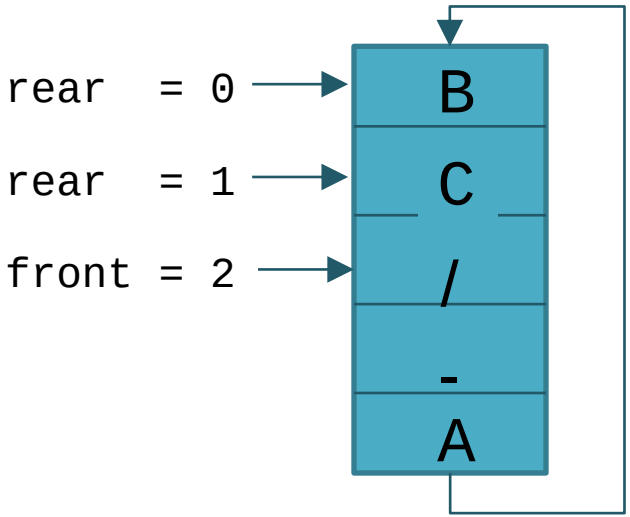



```
q.offer('B');

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.)

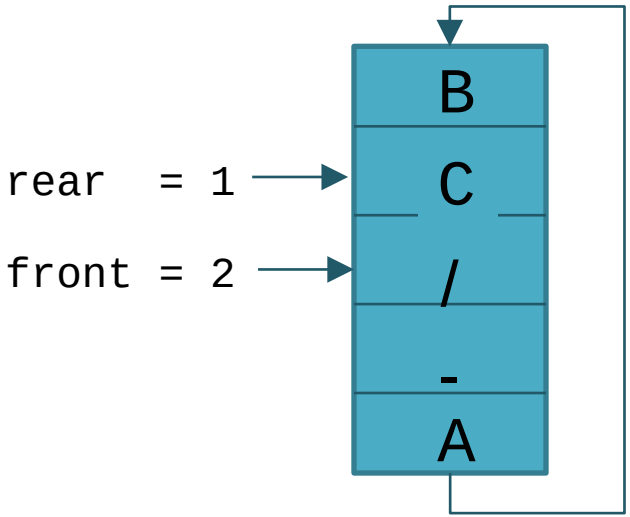



```
q.offer('C');

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.)

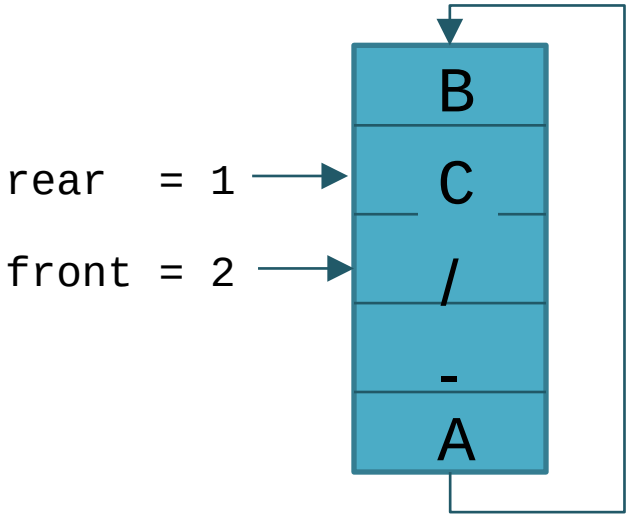



```
q.offer('D');

size     = 5
capacity = 5
```

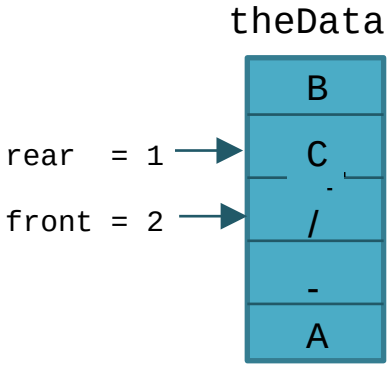
```
public boolean offer(E item) {
  if (size == capacity) {
    reallocate();
  }
  size++;
  rear = (rear + 1) % capacity;
  theData[rear] = item;
  return true;
}
```

IMPLEMENTING A QUEUE USING A CIRCULAR ARRAY (CONT.)




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

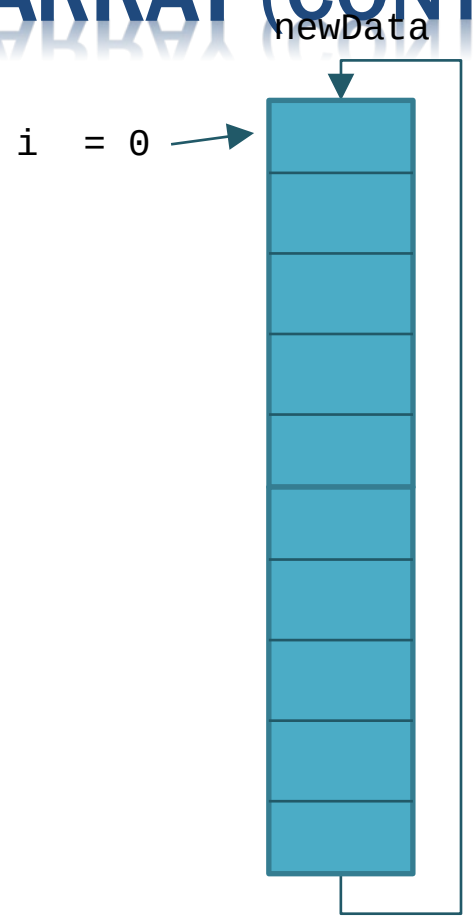
size = 5
capacity = 5




```
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

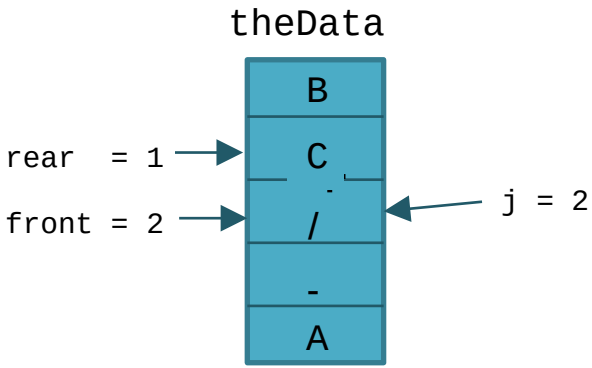
IMPLEMENTING A QUEUE USING A CIRCULAR ARRAY (CONT.)



newCapacity = 10

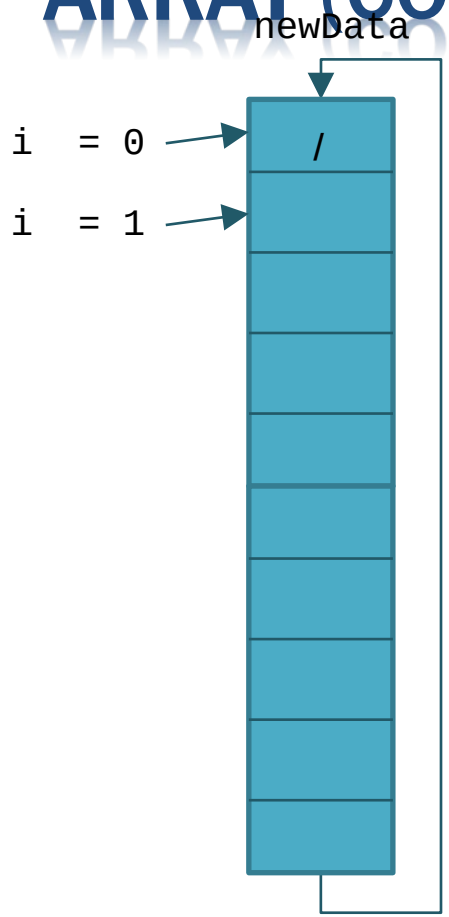
q.offer('D');

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;
}
```

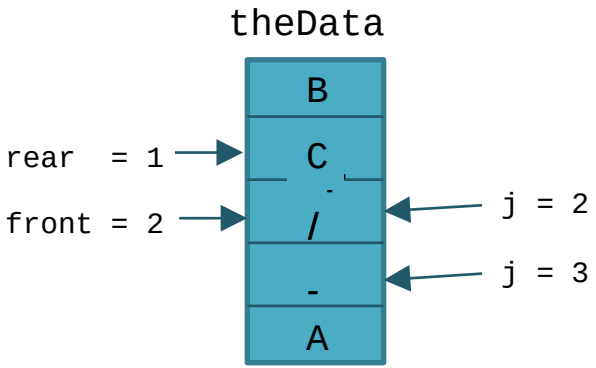
IMPLEMENTING A QUEUE USING A CIRCULAR ARRAY (CONT.)



newCapacity = 10

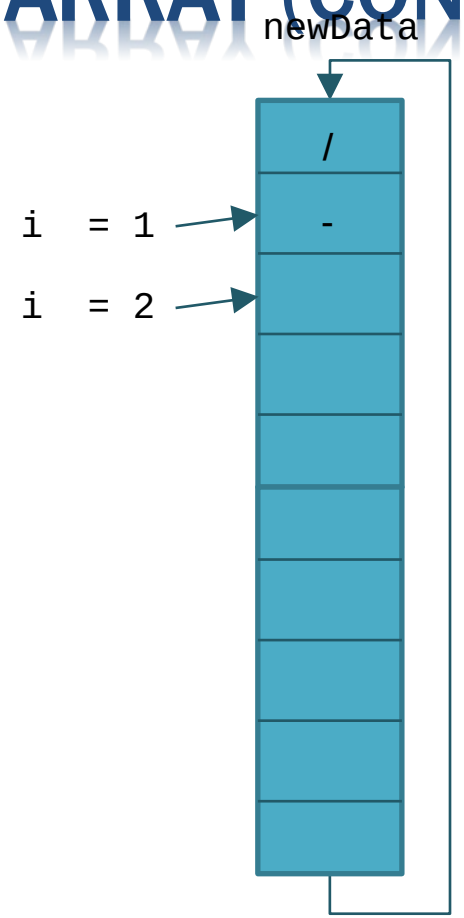
q.offer('D');

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;
}
```

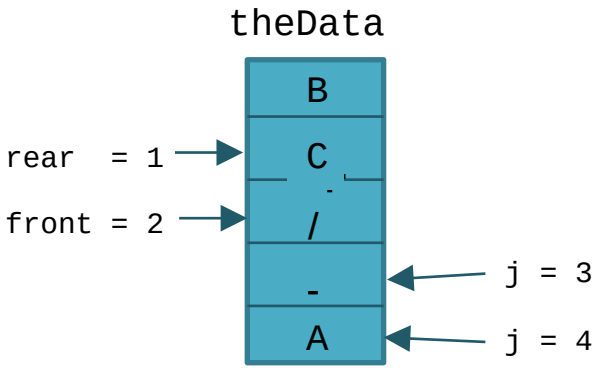
IMPLEMENTING A QUEUE USING A CIRCULAR ARRAY (CONT.)



newCapacity = 10

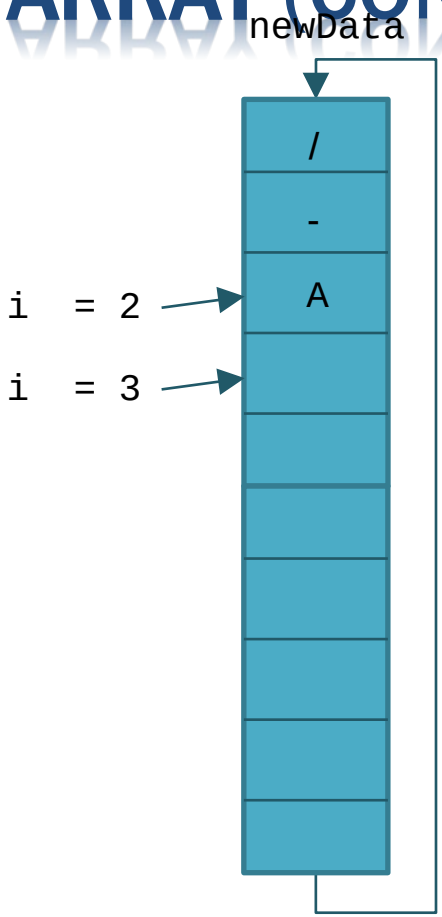
q.offer('D');

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;
}
```

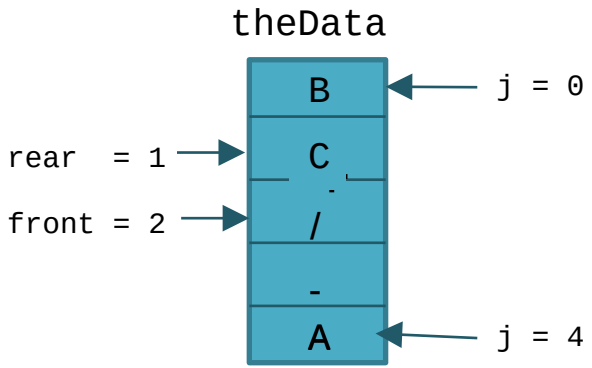
IMPLEMENTING A QUEUE USING A CIRCULAR ARRAY (CONT.)



newCapacity = 10

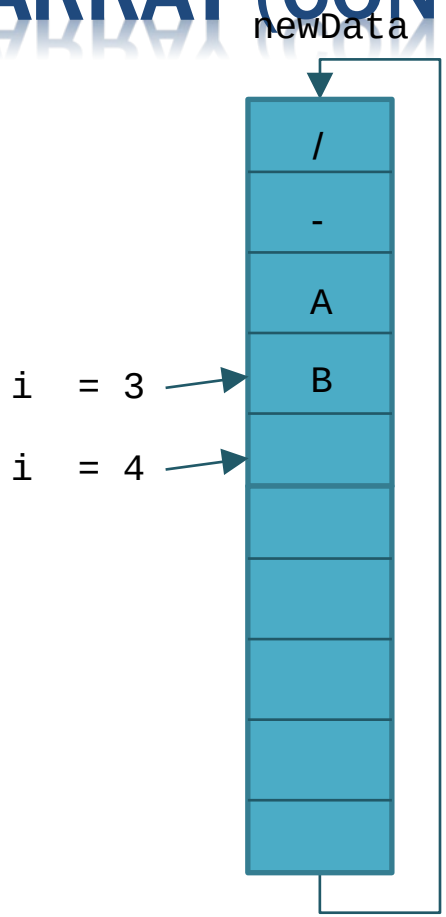
q.offer('D');

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;
}
```

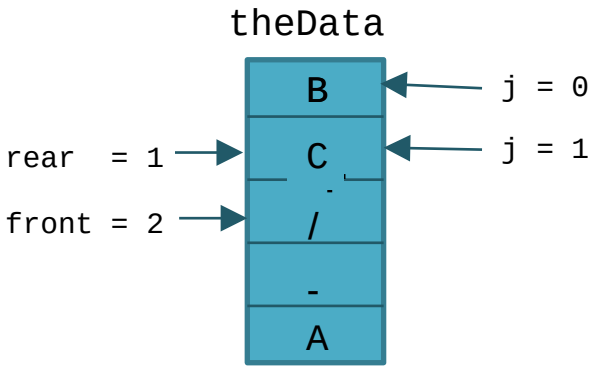
IMPLEMENTING A QUEUE USING A CIRCULAR ARRAY (CONT.)



newCapacity = 10

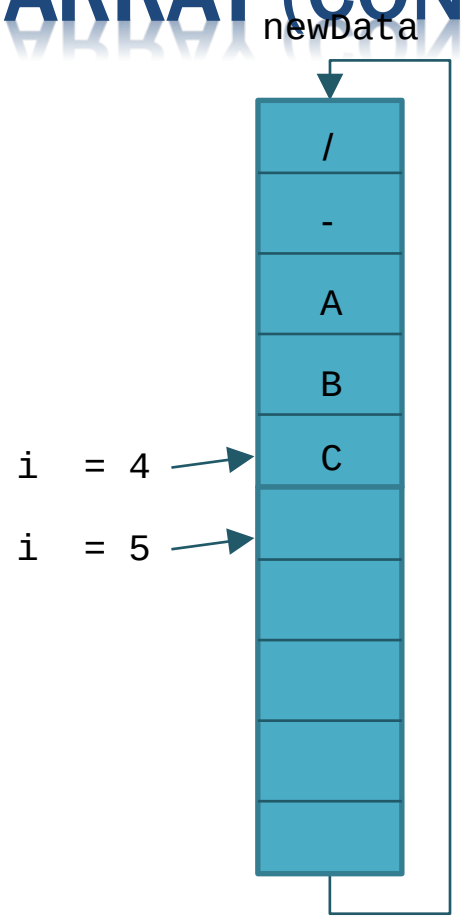
q.offer('D');

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;
}
```

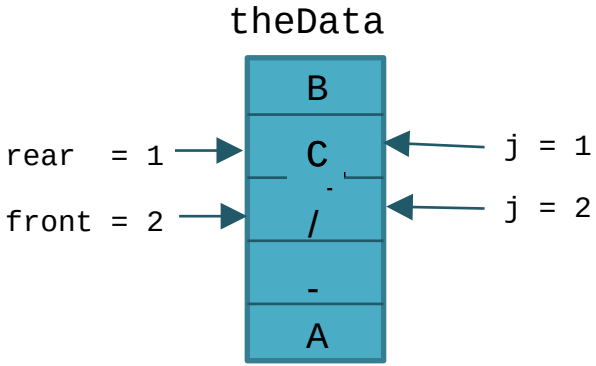
IMPLEMENTING A QUEUE USING A CIRCULAR ARRAY (CONT.)



newCapacity = 10

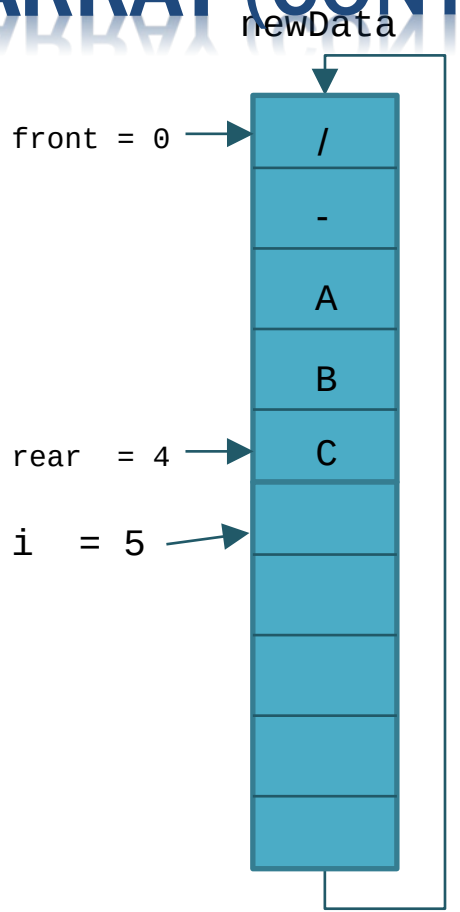
q.offer('D');

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.)

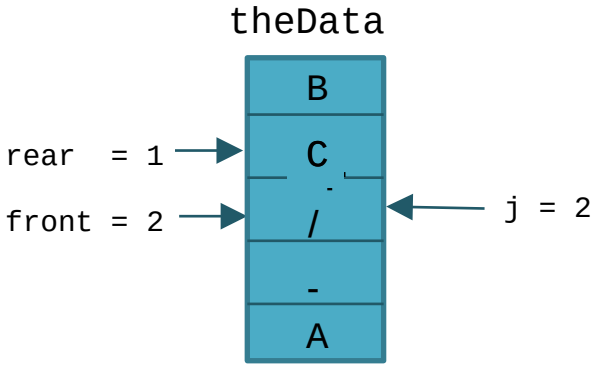


newCapacity = 10

q.offer('D');

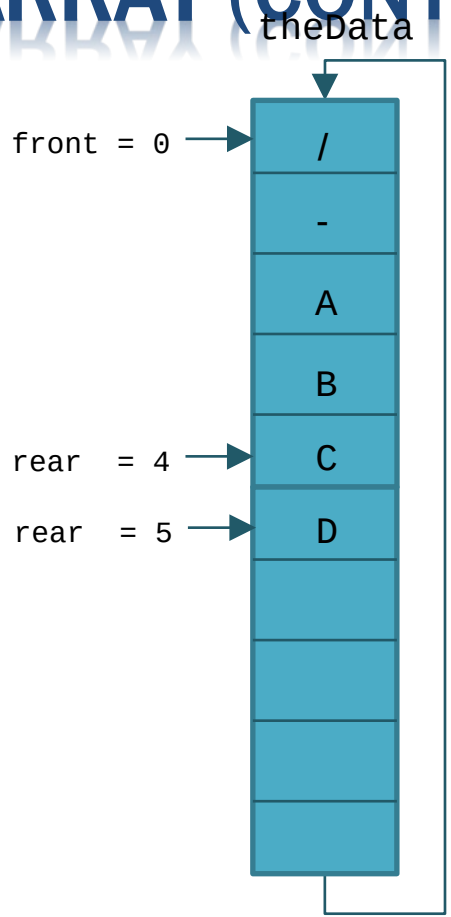
size = 5

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');
```

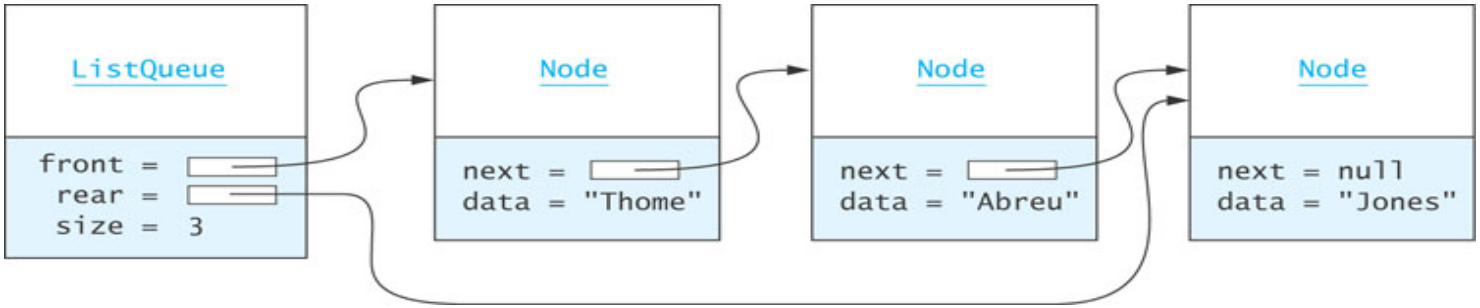
```
size     = 6
capacity = 10
```

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

IMPLEMENTING A QUEUE WITH A SINGLY LINKED LIST

- Supporting worst-case $O(1)$-time for all operations, and without any artificial limit on the capacity
- Orientation for Queue using singly linked list
  - Align the front of the queue with the front of the list,
  - Align the back of the queue with the tail of the list,

  (because the only update operation that singly linked lists support at the back end is an insertion)



IMPLEMENTATION OF A QUEUE USING A SINGLYLINKEDLIST.

- Each method of our LinkedQueue adaptation also runs in $O(1)$ worst-case time.

```
/** Realization of a FIFO queue as an adaptation of a SinglyLinkedList. */
public class LinkedQueue<E> implements Queue<E> {
  private SinglyLinkedList<E> list = new SinglyLinkedList<>();    // an empty list
  public LinkedQueue() { }                    // new queue relies on the initially empty list
  public int size() { return list.size(); }
  public boolean isEmpty() { return list.isEmpty(); }
  public void enqueue(E element) { list.addLast(element); }
  public E first() { return list.first(); }
  public E dequeue() { return list.removeFirst(); }
}
```

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

COMPARING THE IMPLEMENTATIONS

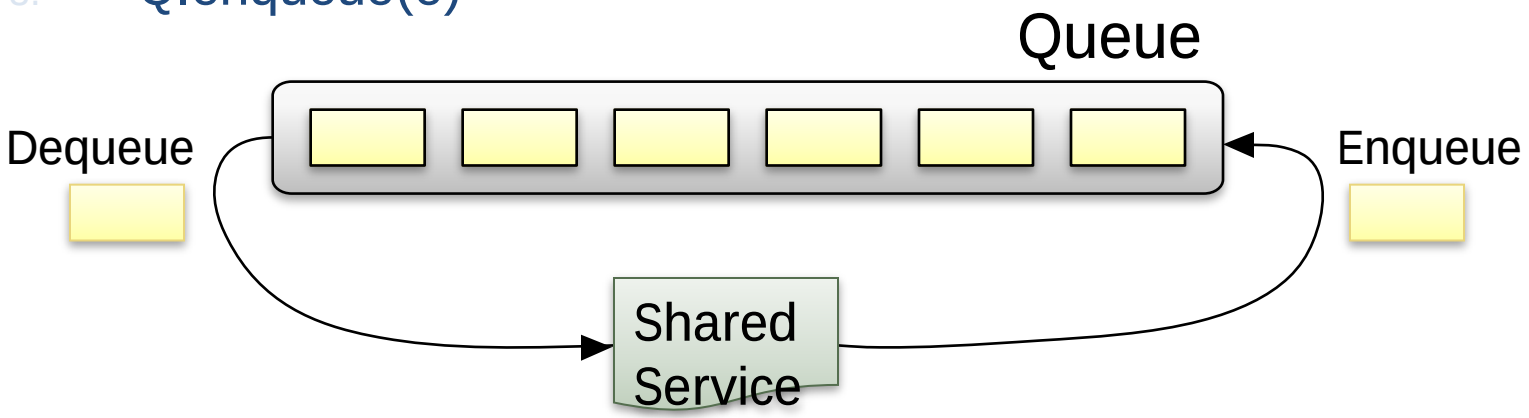
- Computation time
  - All four implementations are comparable in terms of computation time
  - All operations are O(1) regardless of implementation
  - CircularArray: Although reallocating an array is O(*n*), its is amortized over *n* items, so the cost per item is O(1)

COMPARING THE 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

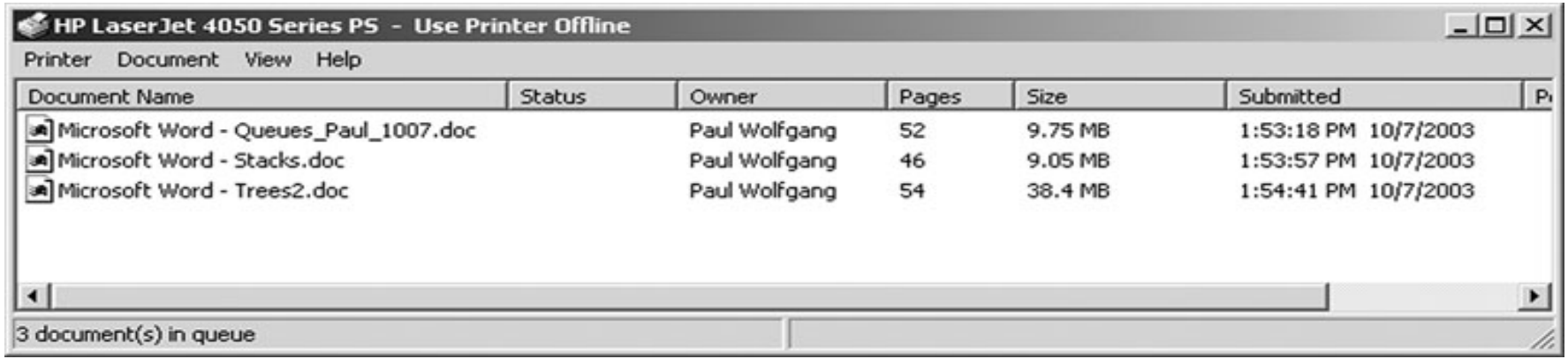
APPLICATION: ROUND ROBIN SCHEDULERS

- We can implement a round robin scheduler using a queue Q by repeatedly performing the following steps:
  1. e = Q.dequeue()
  2. Service element e
  3. Q.enqueue(e)



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



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

DOUBLE-ENDED QUEUE: DEQUE

- A deque (pronounced "deck") is short for double-ended queue
- A double-ended queue allows insertions and removals from both ends
- The deque abstract data type is more general than both the stack and the queue ADTs.

DEQUE ADT

update methods:

addFirst(e): Insert a new element e at the front of the deque.

addLast(e): Insert a new element e at the back of the deque.

removeFirst( ): Remove and return the first element of the deque (or null if the deque is empty).

removeLast( ): Remove and return the last element of the deque (or null if the deque is empty).

accessors

first( ): Returns the first element of the deque, without removing it (or null if the deque is empty).

last( ): Returns the last element of the deque, without removing it (or null if the deque is empty).

size( ): Returns the number of elements in the deque.

isEmpty( ): Returns a boolean indicating whether the deque is empty.

DEQUE INTERFACE

- 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

```
/**
 * Interface for a double-ended queue: a collection of elements that can be inserted
 * and removed at both ends; this interface is a simplified version of java.util.Deque.
 */
public interface Deque<E> {
  /** Returns the number of elements in the deque. */
  int size();
  /** Tests whether the deque is empty. */
  boolean isEmpty();
  /** Returns, but does not remove, the first element of the deque (null if empty). */
  E first();
  /** Returns, but does not remove, the last element of the deque (null if empty). */
  E last();
  /** Inserts an element at the front of the deque. */
  void addFirst(E e);
  /** Inserts an element at the back of the deque. */
  void addLast(E e);
  /** Removes and returns the first element of the deque (null if empty). */
  E removeFirst();
  /** Removes and returns the last element of the deque (null if empty). */
  E removeLast();
}
```

DEQUE **EXAMPLE**

Method	Return Value	D
addLast(5)	–	(5)
addFirst(3)	–	(3, 5)
addFirst(7)	–	(7, 3, 5)
first( )	7	(7, 3, 5)
removeLast( )	5	(7, 3)
size( )	2	(7, 3)
removeLast( )	3	(7)
removeFirst( )	7	( )
addFirst(6)	–	(6)
last( )	6	(6)
addFirst(8)	–	(8, 6)
isEmpty( )	false	(8, 6)
last( )	6	(8, 6)

series of operations and their effects on an initially empty deque *D* of integers.

IMPLEMENTING A DEQUE CIRCULAR ARRAY

- Representation similar to the ArrayQueue class
- One extra concern is avoiding use of negative values with the modulo operator:
- When an element is inserted at the front, the first index must effectively be decremented in circular fashion and it is a mistake to assign f = (f−1) % N (ex> f=0)

Solution: f = (f−1+N) % N.

IMPLEMENTING A DEQUE WITH A DOUBLY LINKED LIST

- Doubly linked list is most appropriate for implementing all operations efficiently.
  - DoublyLinkedList class implements the entire Deque interface;
  - "implements Deque<E>" to that class definition in order to use it as a deque.

Performance of the Deque Operations implemented with a doubly linked list

Method	Running Time
size, isEmpty	$O(1)$
first, last	$O(1)$
addFirst, addLast	$O(1)$
removeFirst, removeLast	$O(1)$

DEQUES IN THE JAVA COLLECTIONS FRAMEWORK

Interface: java.util.Deque

Implementations:

circular array: java.util.ArrayDeque

doubly linked list: java.util.LinkedList

Our Deque ADT	Interface java.util.Deque	
	throws exceptions	returns special value
first()	getFirst()	peekFirst()
last()	getLast()	peekLast()
addFirst(e)	addFirst(e)	offerFirst(e)
addLast(e)	addLast(e)	offerLast(e)
removeFirst()	removeFirst()	pollFirst()
removeLast()	removeLast()	pollLast()
size()	size()	
isEmpty()	isEmpty()	

Section 4.5

SIMULATING WAITING LINES USING QUEUES

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

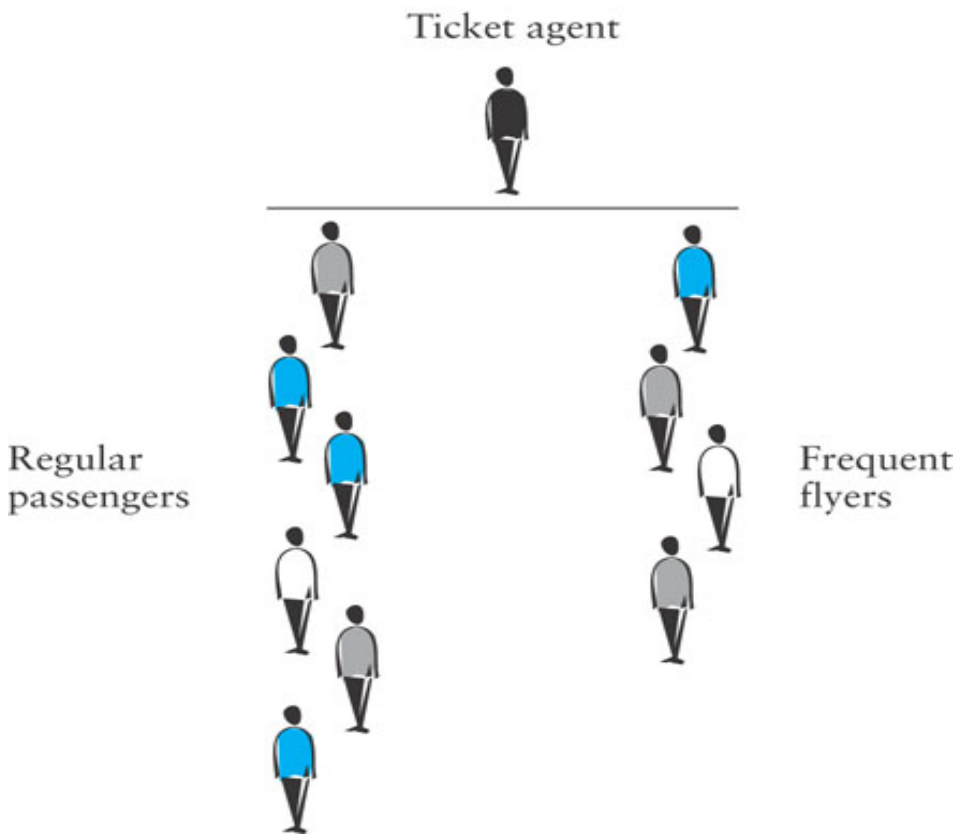
SIMULATING WAITING LINES USING QUEUES (CONT.)

- 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.)



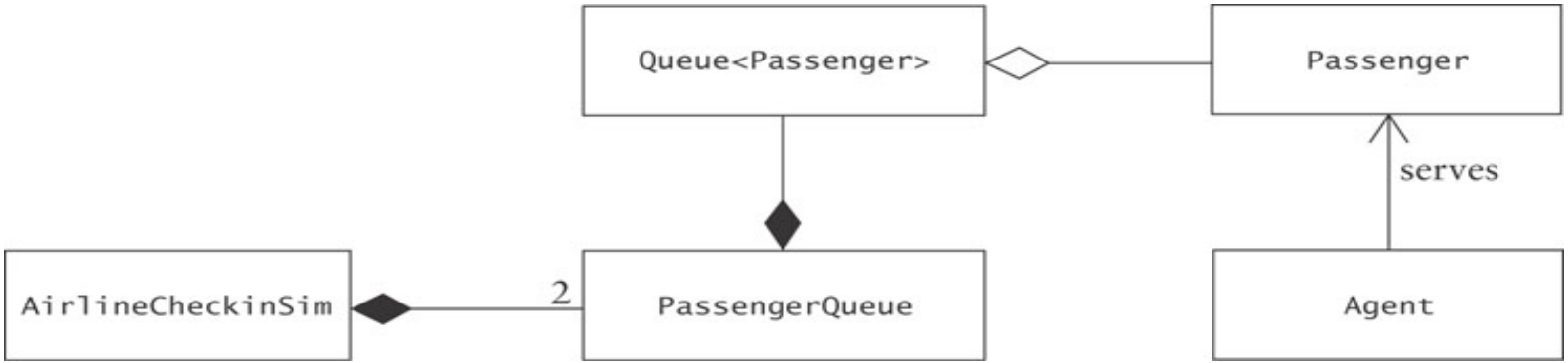
CASE STUDY: ANALYSIS

- 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

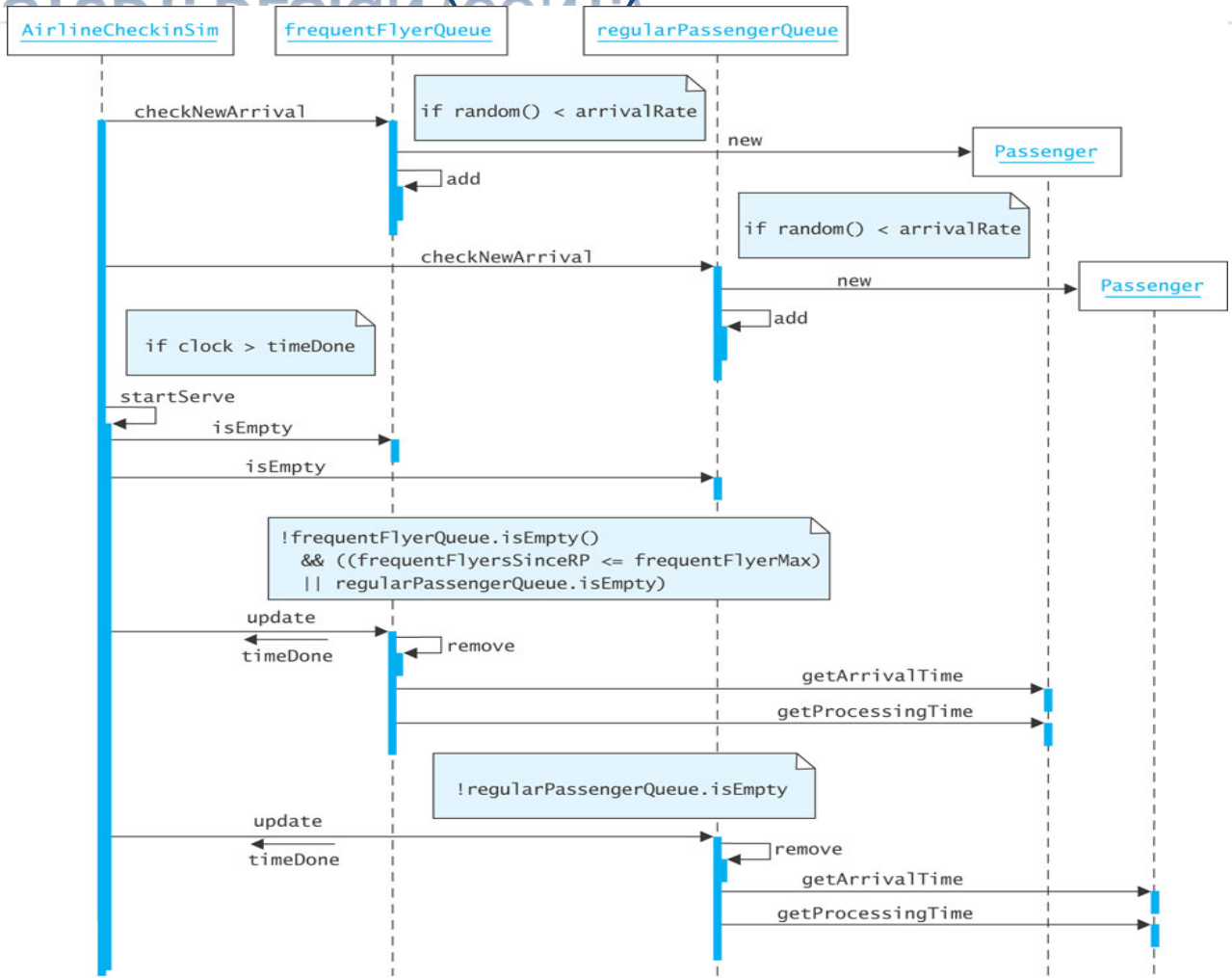
CASE STUDY: ANALYSIS (CONT.)

- 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

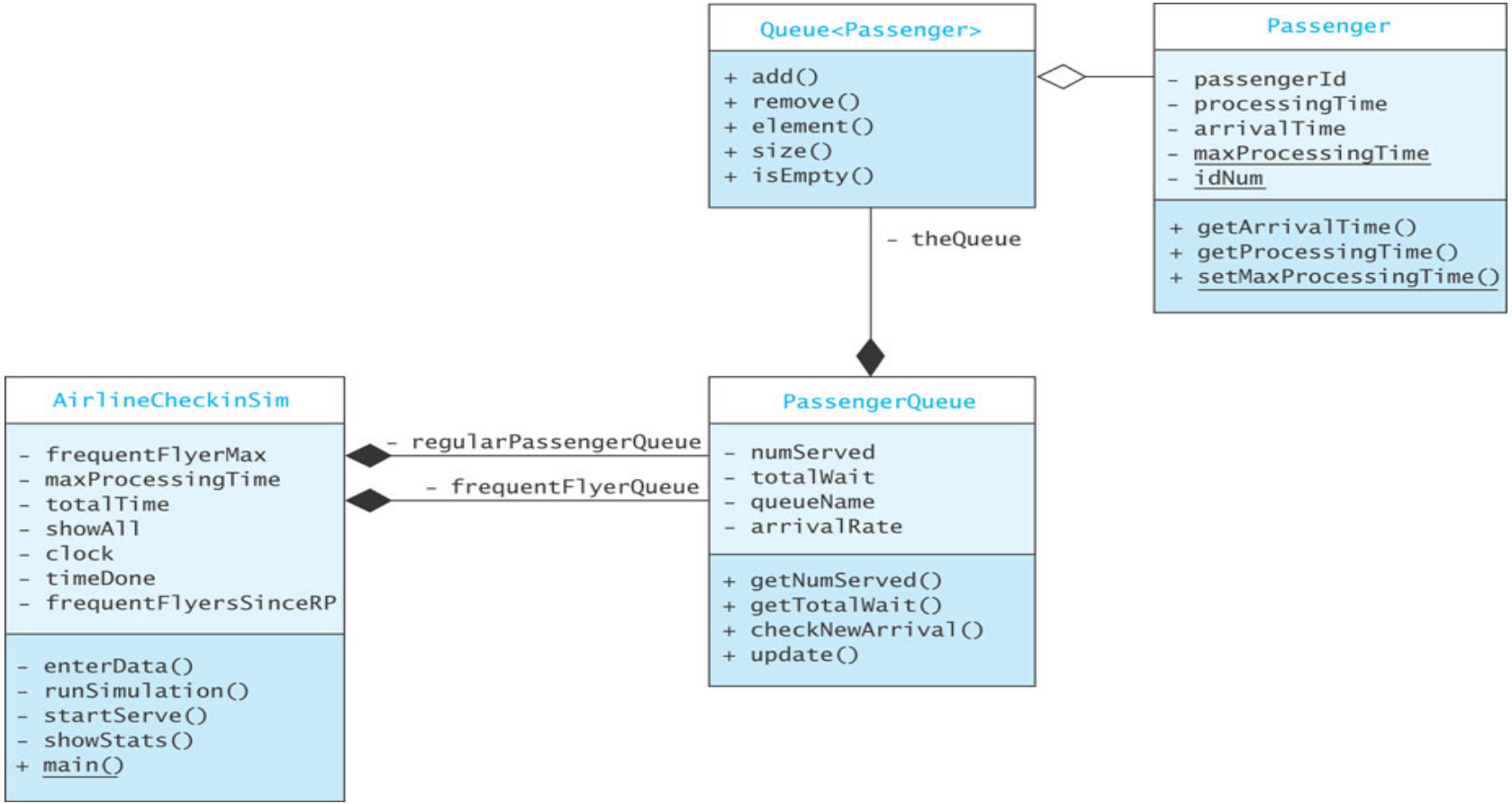
CASE STUDY: DESIGN (CONT.)



CASE STUDY: DESIGN (CONT.)



CASE STUDY: DESIGN (CONT.)



CASE STUDY: DESIGN (CONT.)

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

CASE STUDY: DESIGN (CONT.)

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

CASE STUDY: DESIGN (CONT.)

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