



STEVENS
INSTITUTE of TECHNOLOGY
THE INNOVATION UNIVERSITY®

CS 570: Data Structures

Queues

Instructor: Iraklis Tsekourakis

Email: itsekour@stevens.edu



CHAPTER 4 (PART 2)

Queues

Chapter Objectives

3

- ❑ To learn 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

Week 7

- Reading Assignment: Koffman and Wolfgang, Sections 4.5-4.8

Queue

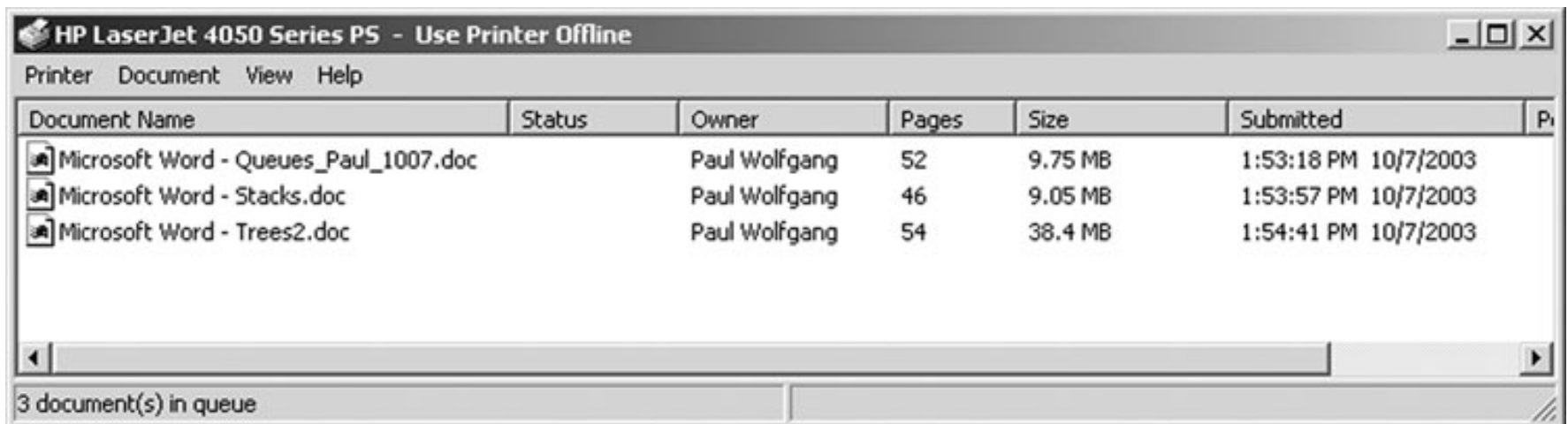
5

- 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*

Example: Print Queue

6

- 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

7

- ❑ 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

Specification for a Queue Interface

8

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> .

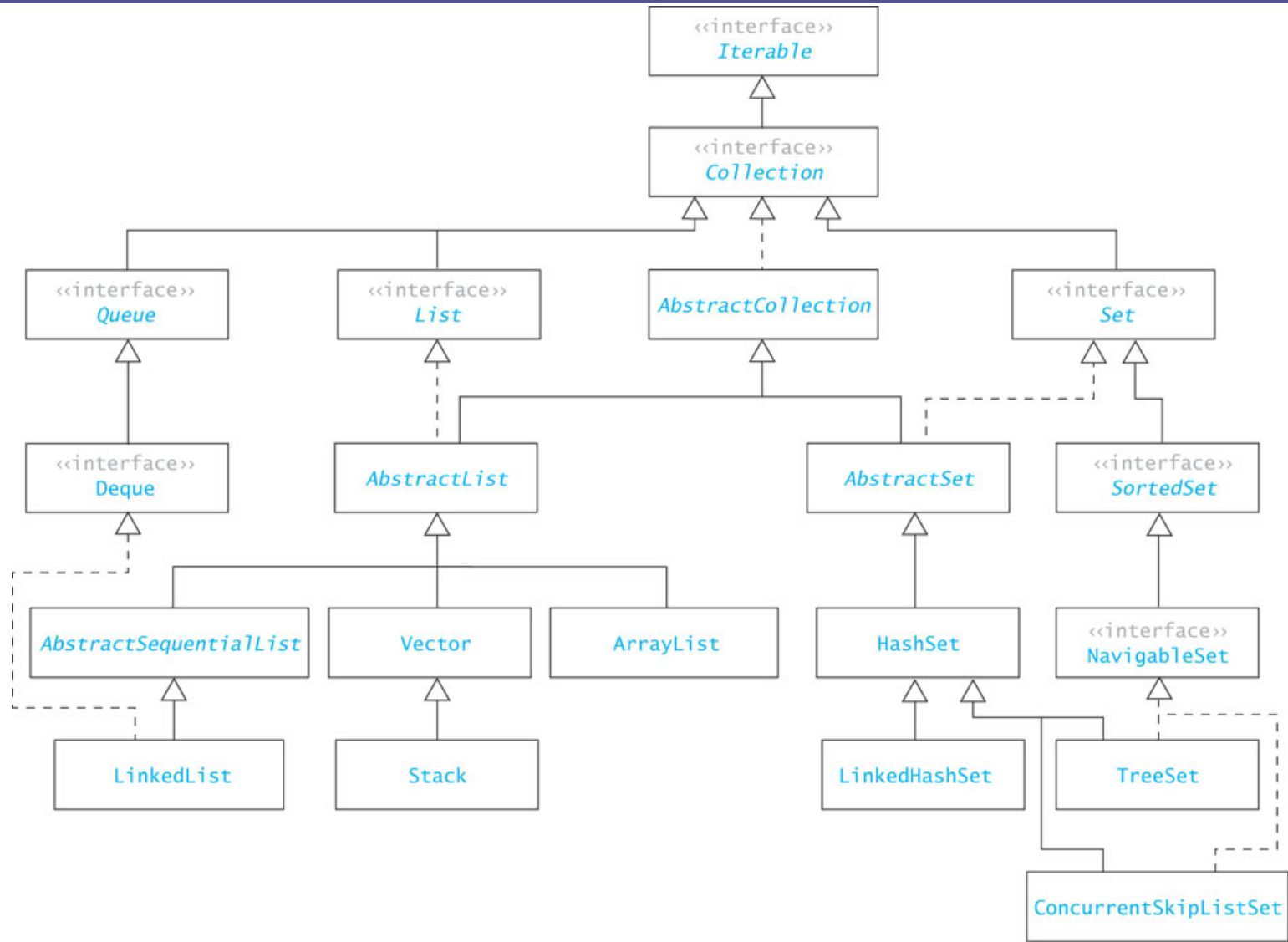
Class LinkedList Implements the Queue Interface

9

- 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 `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 Collection Framework

10



Implementing the Queue Interface

Section 4.7

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

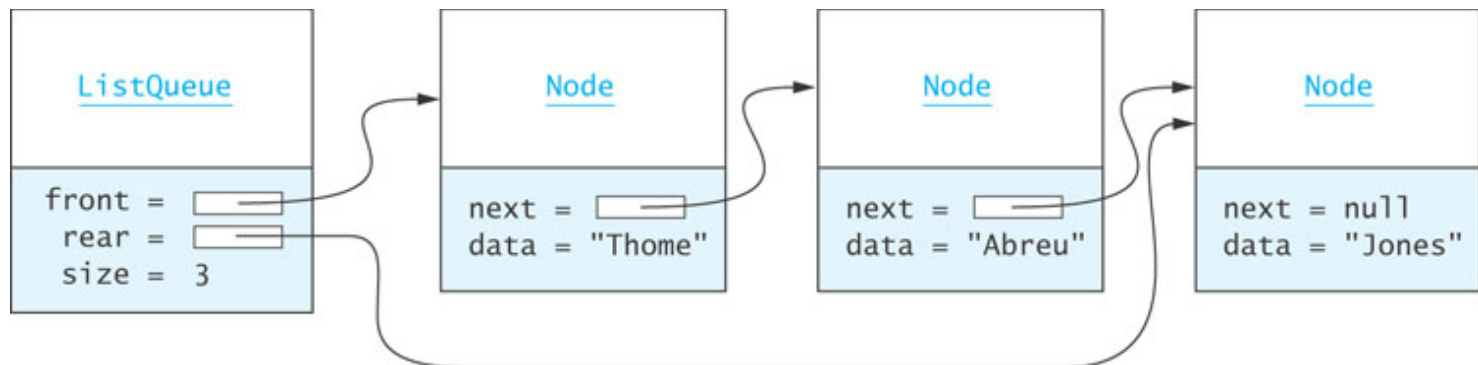
12

- 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

13

- ❑ 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



Using a Single-Linked List to Implement a Queue

14

```
import java.util.*;

/** Implements the Queue interface using a single-linked
    list.

    *   @author Koffman & Wolfgang * */
public class ListQueue < E >
    extends AbstractQueue < E >
    implements Queue < E > {

    // Data Fields
    /** Reference to front of queue. */
    private Node < E > front;

    /** Reference to rear of queue. */
    private Node < E > rear;
```

Using a Single-Linked List to Implement a Queue

15

```
/** Size of queue. */
private int size;

/** A Node is the building block for a single-linked list.
 */
private static class Node < E > {
    // Data Fields
    /** The reference to the data. */
    private E data;

    /** The reference to the next node. */
    private Node next;
```

Using a Single-Linked List to Implement a Queue

16

```
// Constructors
/** Creates a new node with a null next field.
    @param dataItem The data stored
 */
private Node(E dataItem) {
    data = dataItem;
    next = null;
}
/** Creates a new node that references another node.
    @param dataItem The data stored
    @param nodeRef The node referenced by new node
 */
private Node(E dataItem, Node < E > nodeRef) {
    data = dataItem;
    next = nodeRef;
}
} //end class Node
```


Using a Single-Linked List to Implement a Queue

17

```
// Methods

/** Insert an item at the rear of the queue.
    post: item is added to the rear of the queue.
    @param item The element to add
    @return true (always successful)    */
public boolean offer(E item) {
    // Check for empty queue.
    if (front == null) {
        rear = new Node < E > (item);
        front = rear;
    }
    else {
        ...
    }
}
```

Using a Single-Linked List to Implement a Queue

18

```
else {  
    // Allocate a new node at end, store item in  
    // it, and  
    // link it to old end of queue.  
    ??? ??  
}  
size++;  
return true;  
}
```

Using a Single-Linked List to Implement a Queue

19

```
/** Remove the entry at the front of the queue and return it
    if the queue is not empty.
    post: front references item that was second in the queue.
    @return The item removed if successful, or null if not
    */
public E poll() {
    E item = peek(); // Retrieve item at front.
    if (item == null)
        return null;
    // Remove item at front.
    front = front.next;
    size--;
    return item; // Return data at front of queue.
}
```

Using a Single-Linked List to Implement a Queue

20

```
/** Return the item at the front of the queue  
without removing it.
```

```
    @return The item at the front of the queue if  
    successful;
```

```
        return null if the queue is empty
```

```
*/
```

```
public E peek() {
```

```
    if (size == 0)
```

```
        return null;
```

```
    else
```

```
        return front.data;
```

```
}
```

```
}
```

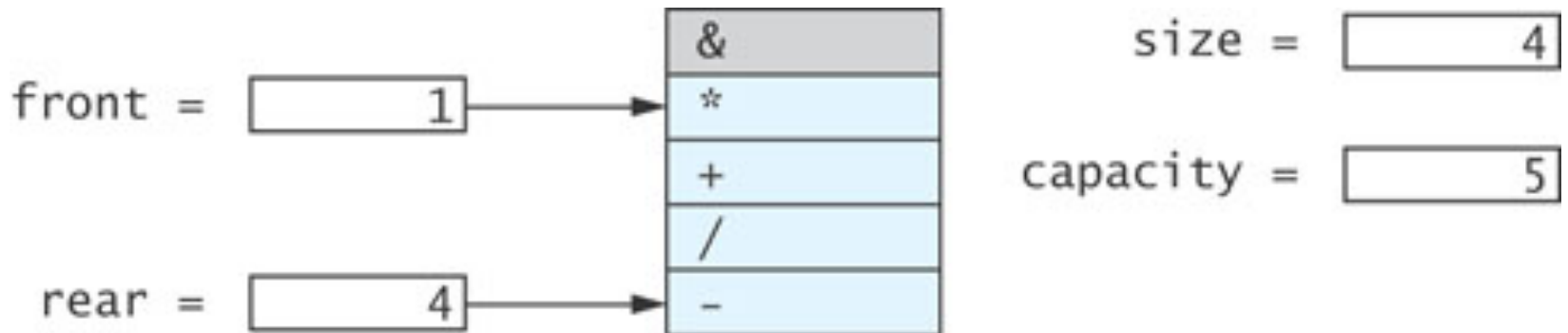
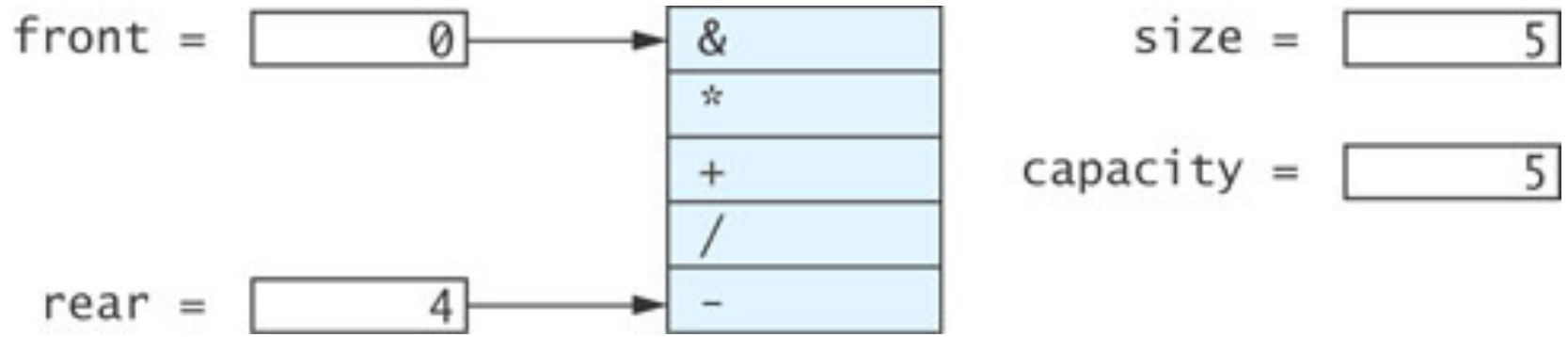
Implementing a Queue Using a Circular Array

21

- ❑ 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 can avoid these inefficiencies in an array

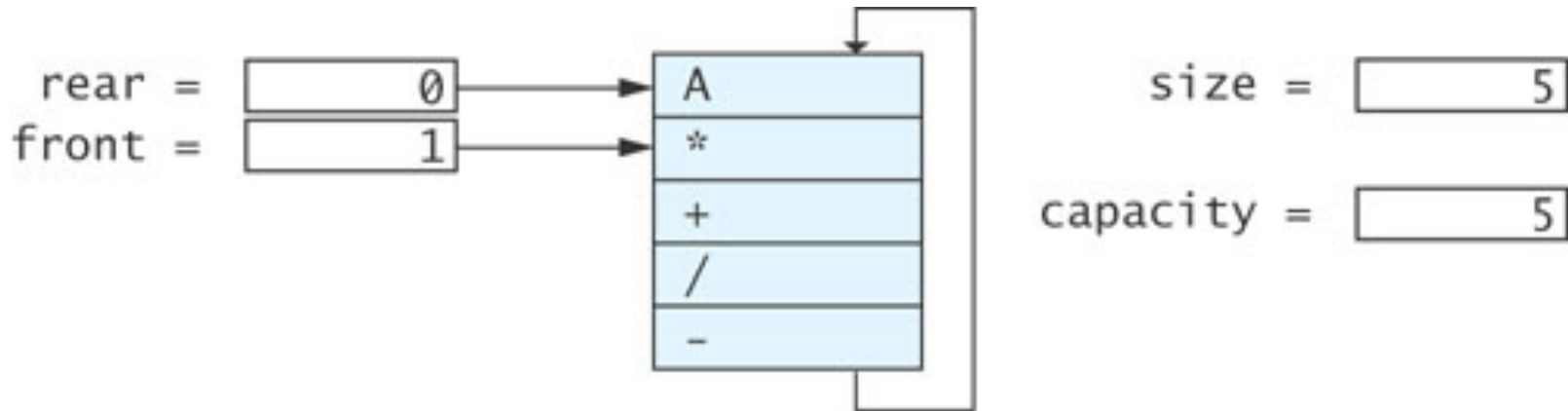
Implementing a Queue Using a Circular Array (cont.)

22



Implementing a Queue Using a Circular Array (cont.)

23



Implementing a Queue Using a Circular Array (cont.)

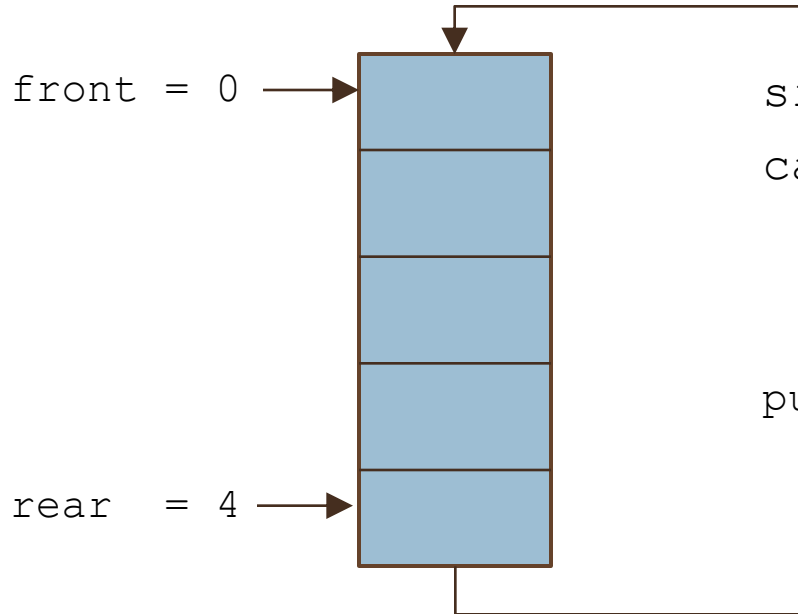
24

```
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.)

25

`q.offer('*');`

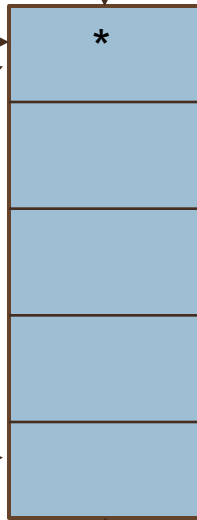
`front = 0` →

`rear = 0` →

`size = 1`

`capacity = 5`

`rear = 4` →



```
→ 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.)

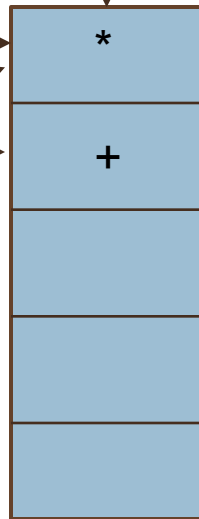
26

`q.offer('+');`

`front = 0` →

`rear = 0` →

`rear = 1` →



`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.)

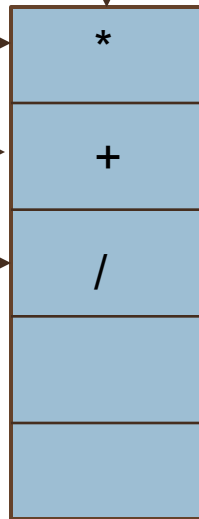
27

`q.offer('/');`

`front = 0` →

`rear = 1` →

`rear = 2` →



`size = 3`

`capacity = 5`

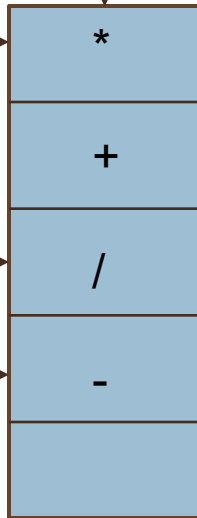
```
public boolean offer(E item) {  
    if (size == capacity) {  
        reallocate();  
    }  
    size++;  
    rear = (rear + 1) % capacity;  
    theData[rear] = item;  
    return true;  
}
```

Implementing a Queue Using a Circular Array (cont.)

28

`q.offer('-');`

`front = 0`



`size = 4`

`capacity = 5`

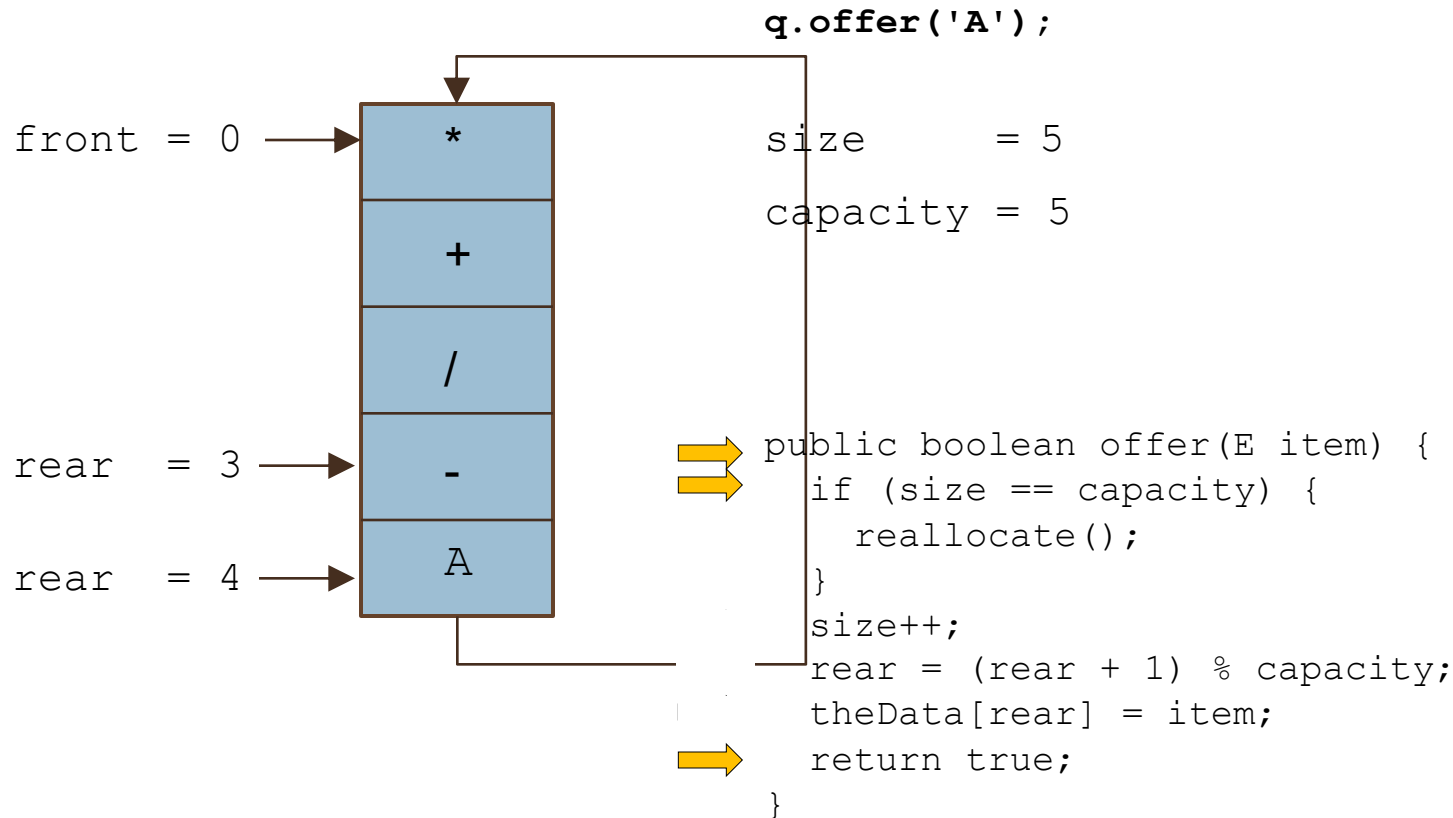
`rear = 2`

`rear = 3`

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

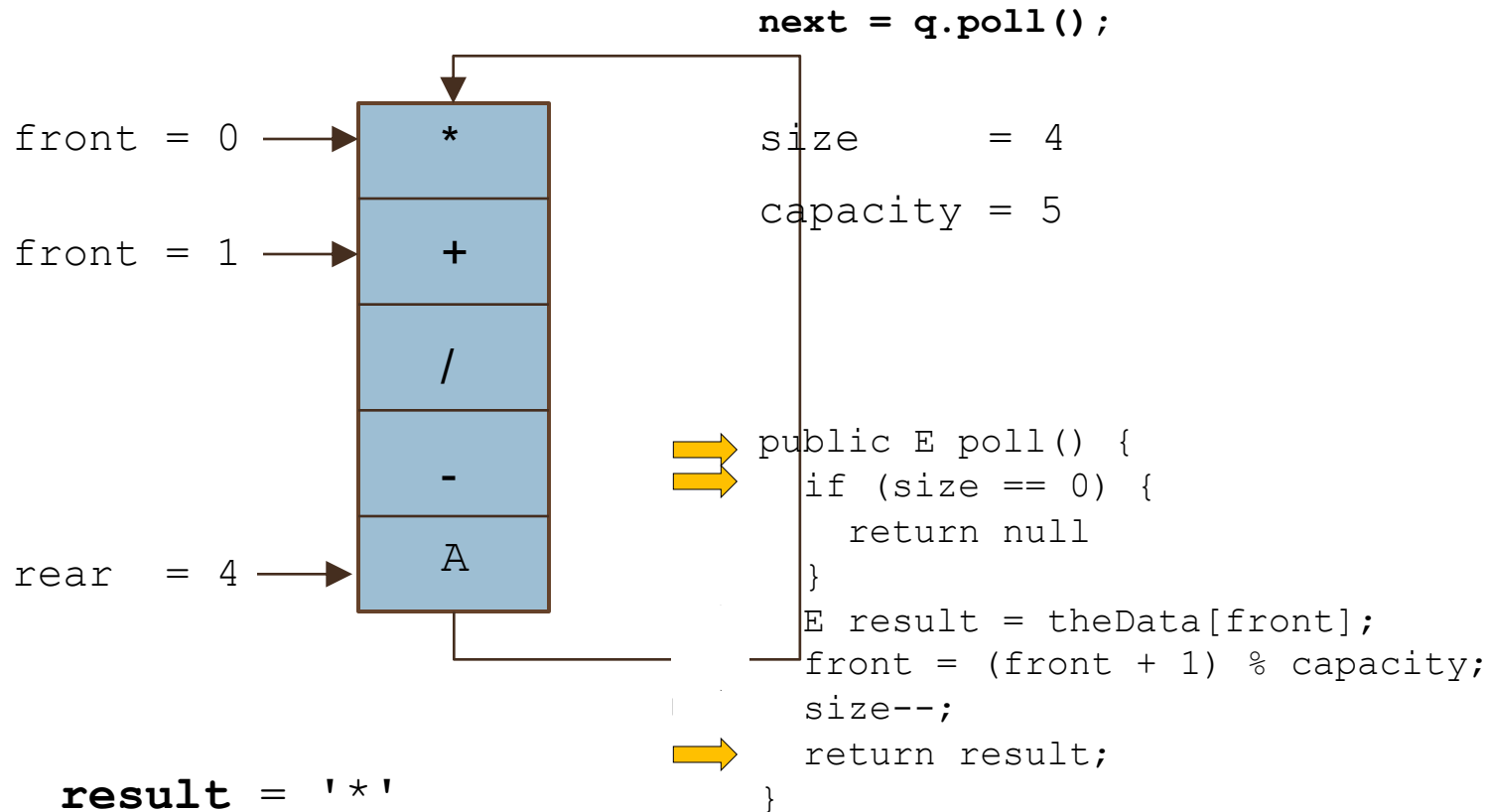
Implementing a Queue Using a Circular Array (cont.)

29



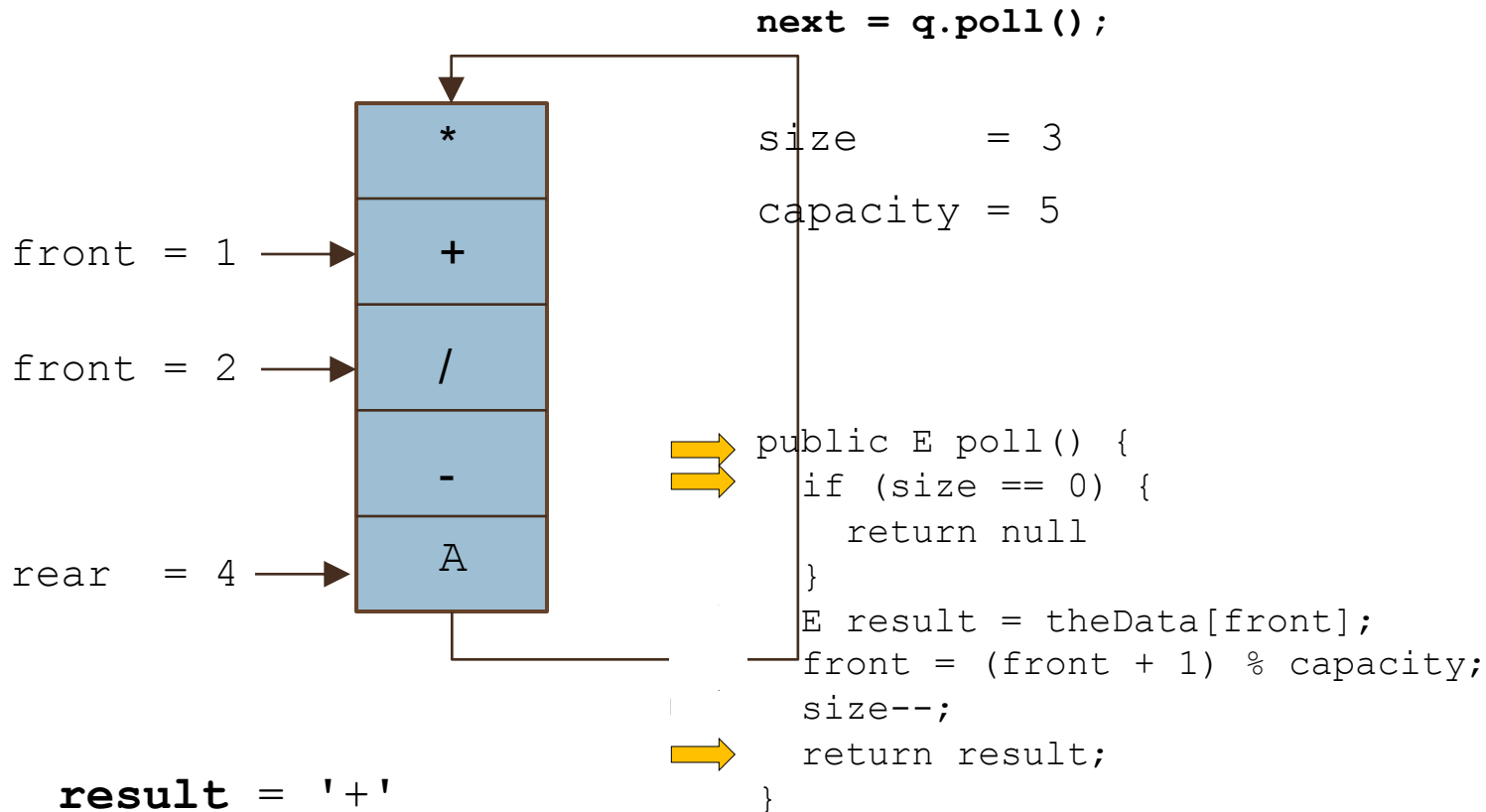
Implementing a Queue Using a Circular Array (cont.)

30



Implementing a Queue Using a Circular Array (cont.)

31

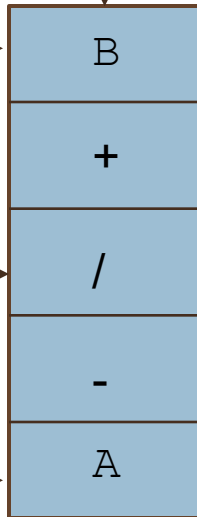


Implementing a Queue Using a Circular Array (cont.)

32

`q.offer('B');`

`rear = 0` →



`size = 4`

`capacity = 5`

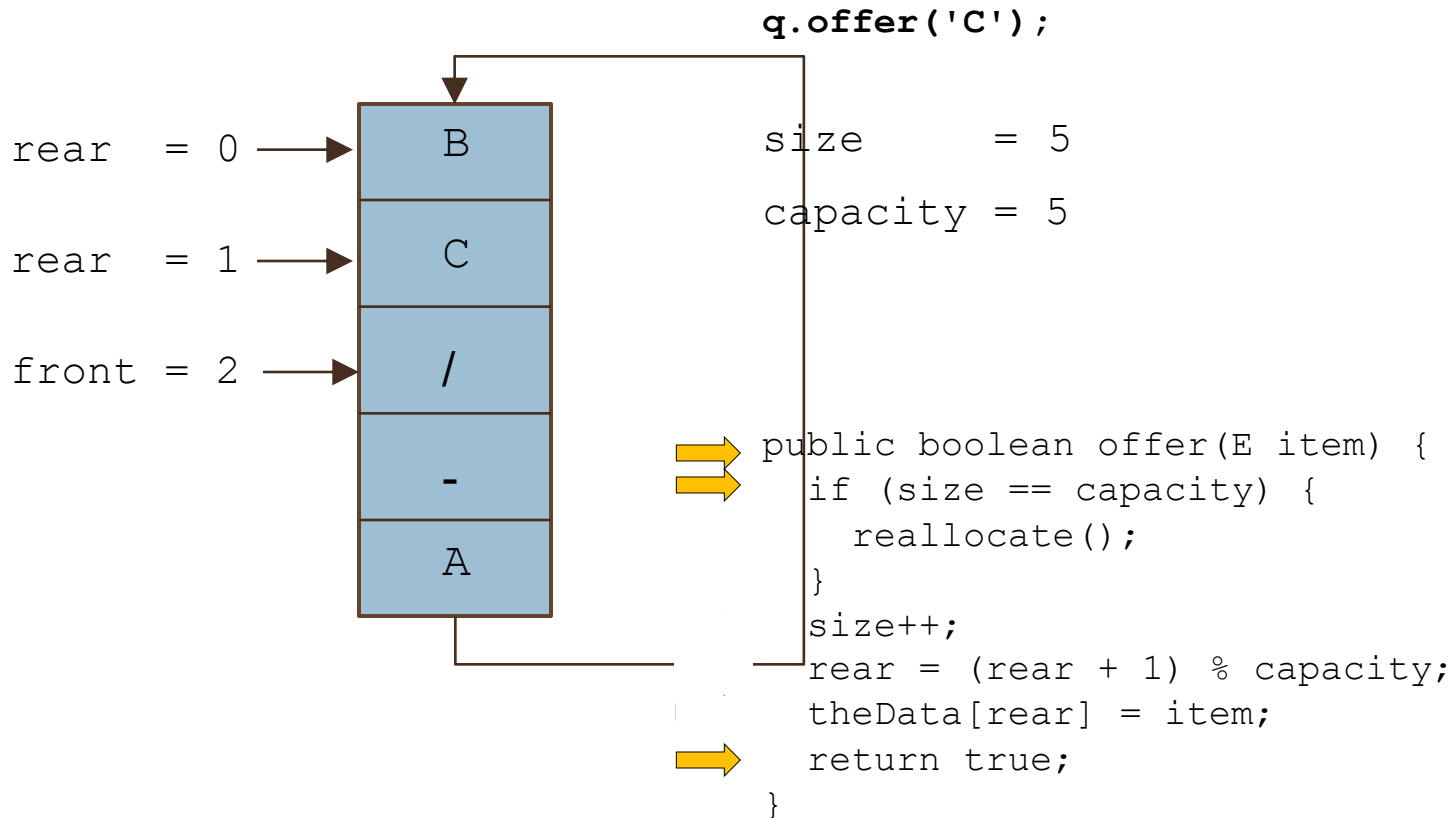
`front = 2` →

`rear = 4` →

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

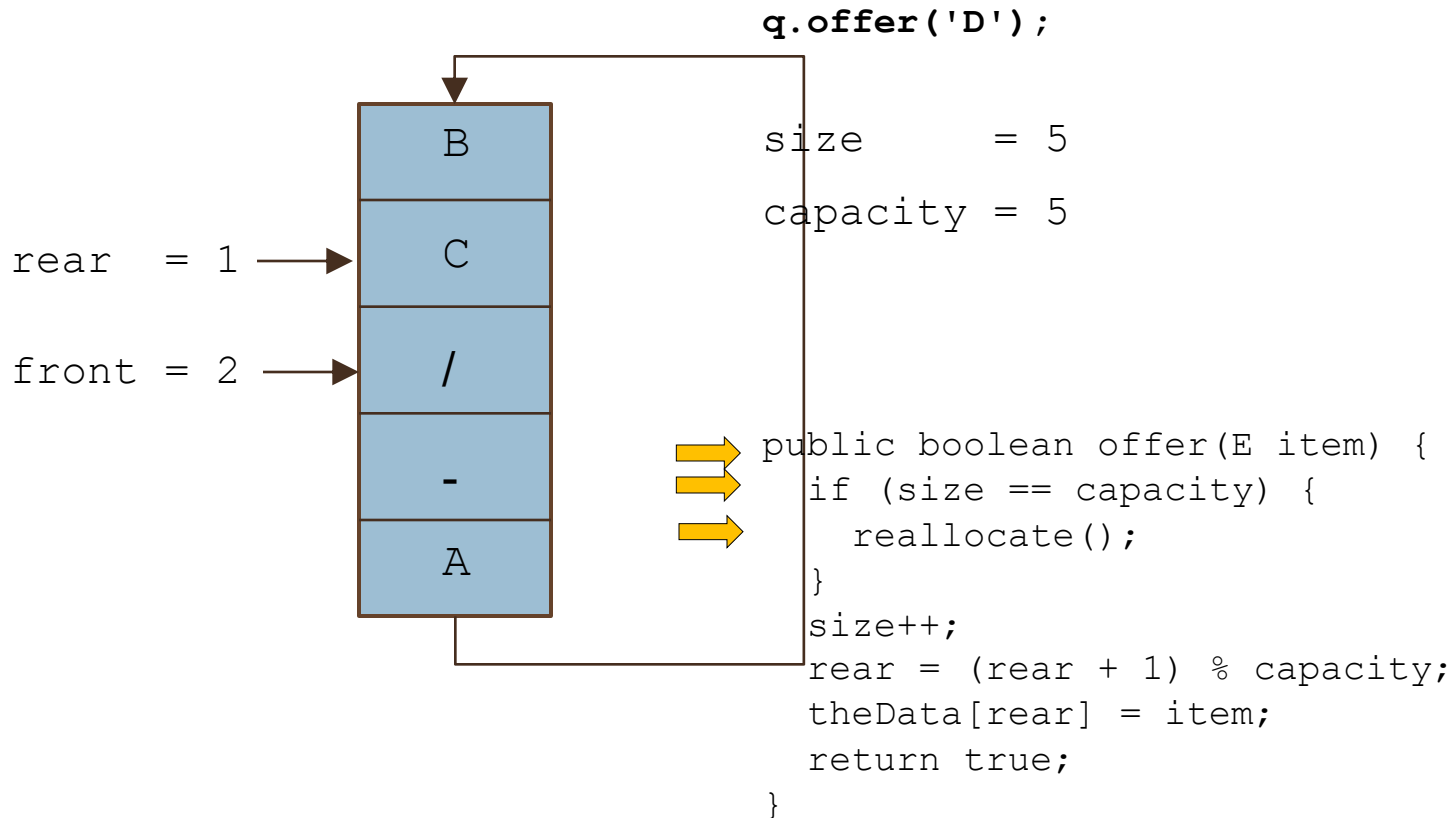

Implementing a Queue Using a Circular Array (cont.)

33

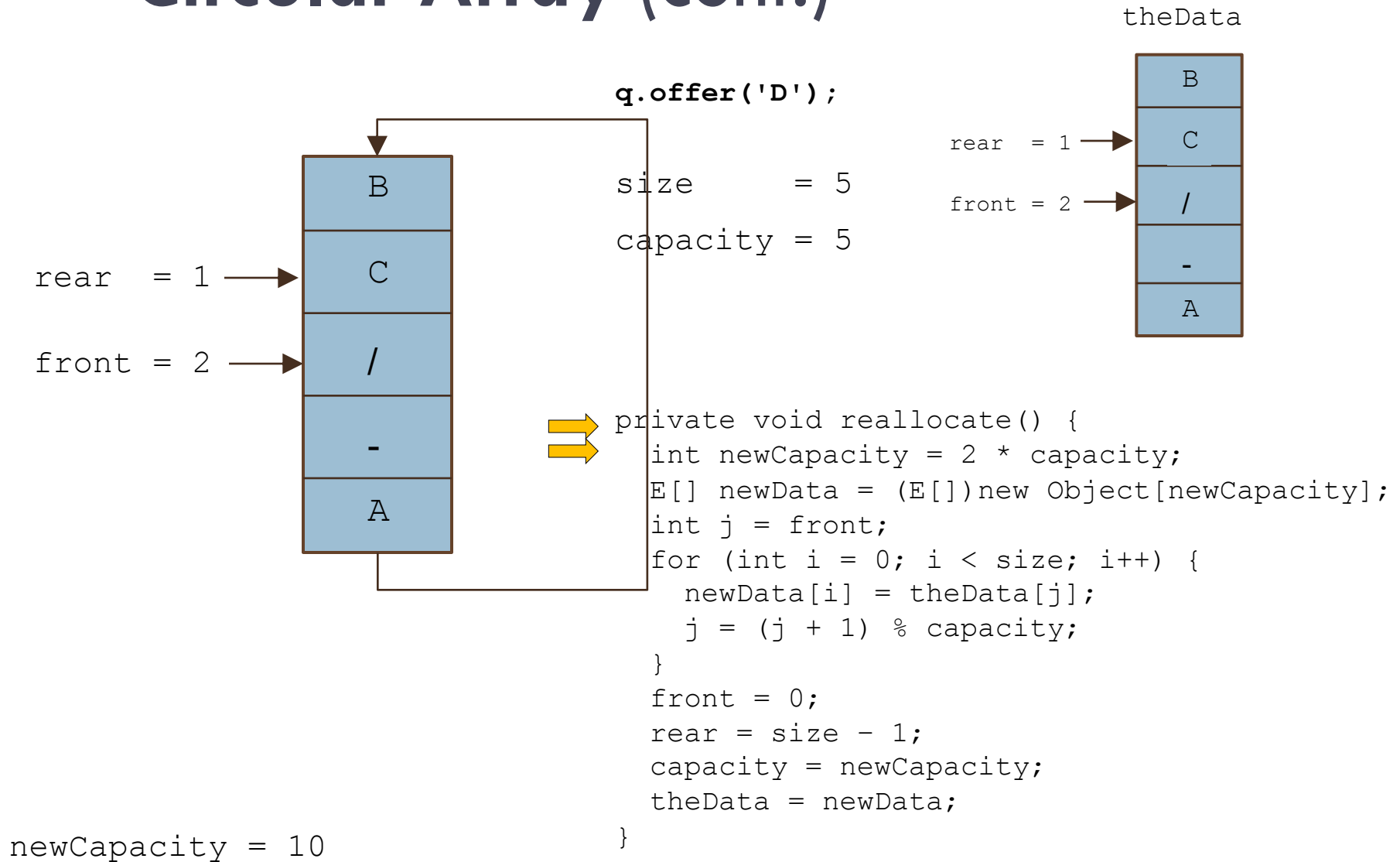


Implementing a Queue Using a Circular Array (cont.)

34

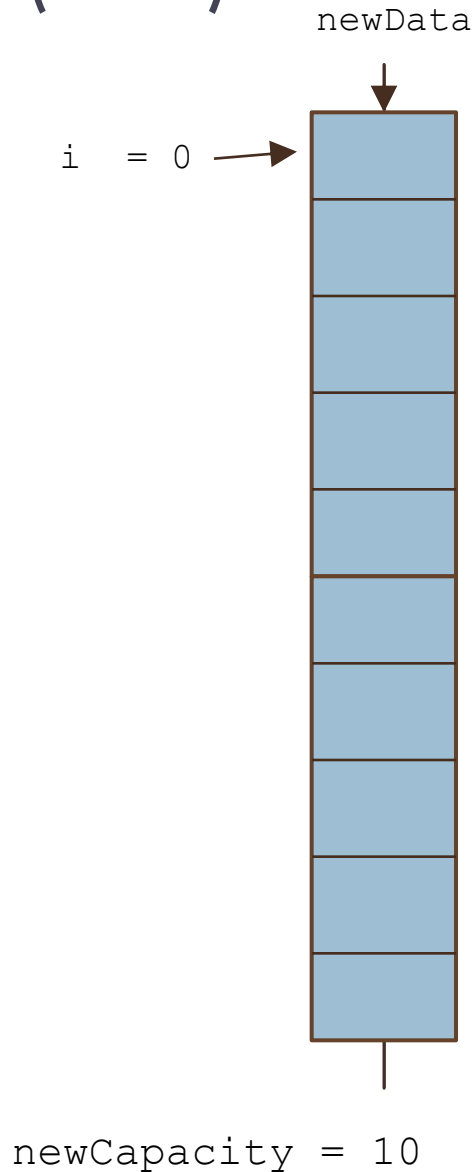


Implementing a Queue Using a Circular Array (cont.)



Implementing a Queue Using a Circular Array

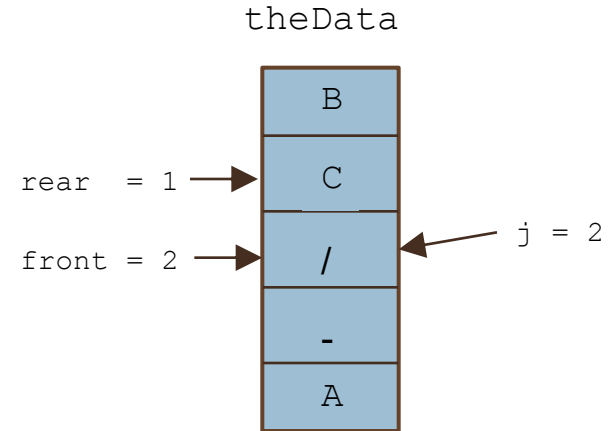
(cont.)



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

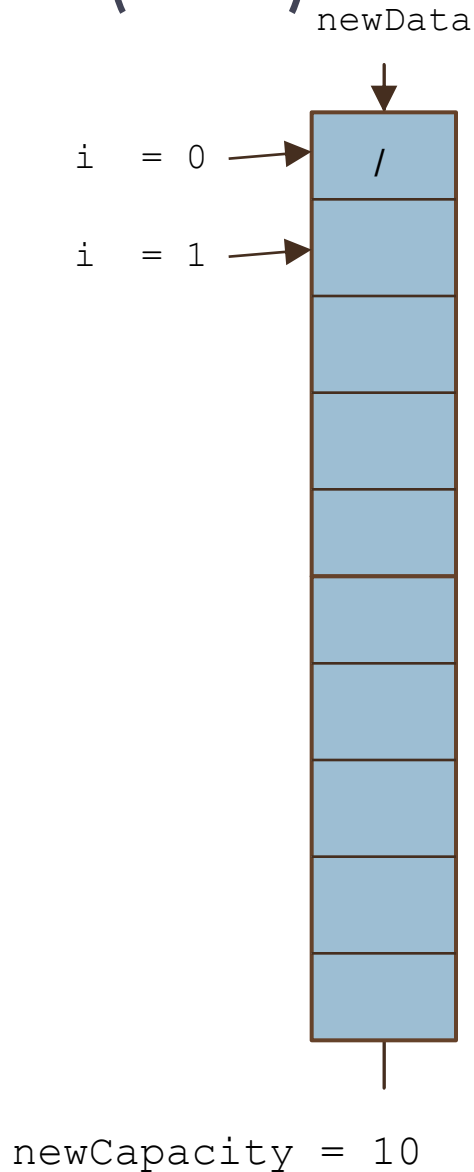
```
size      = 5
```

```
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;  
}
```

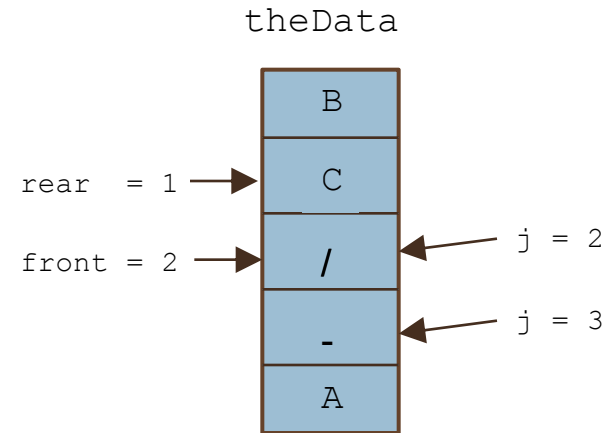
Implementing a Queue Using a Circular Array (cont.)



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

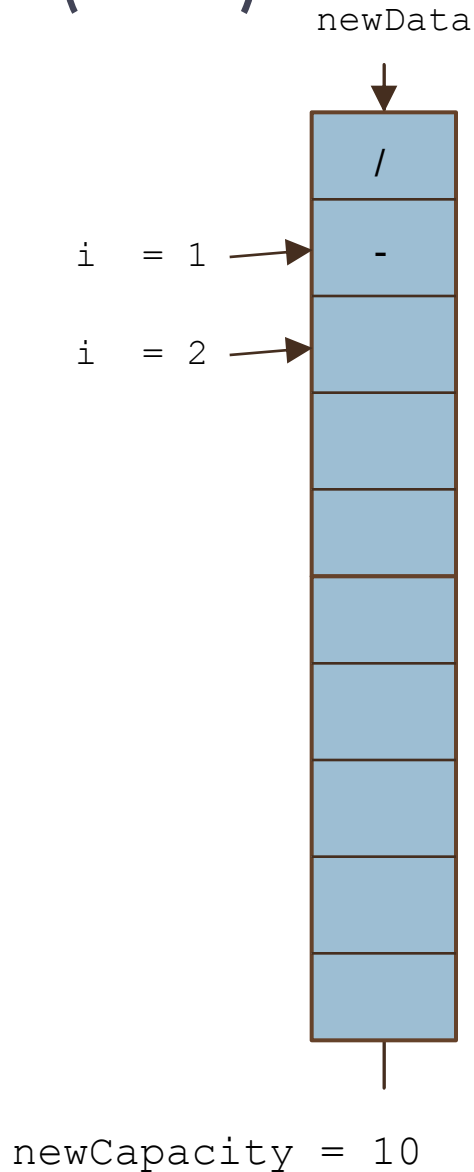
```
size = 5
```

```
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;  
}
```

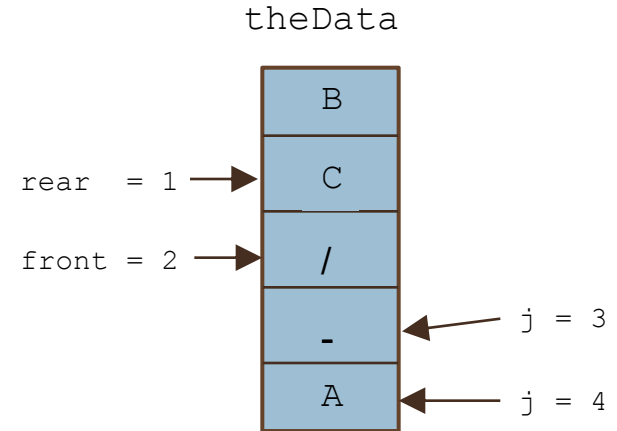
Implementing a Queue Using a Circular Array (cont.)



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

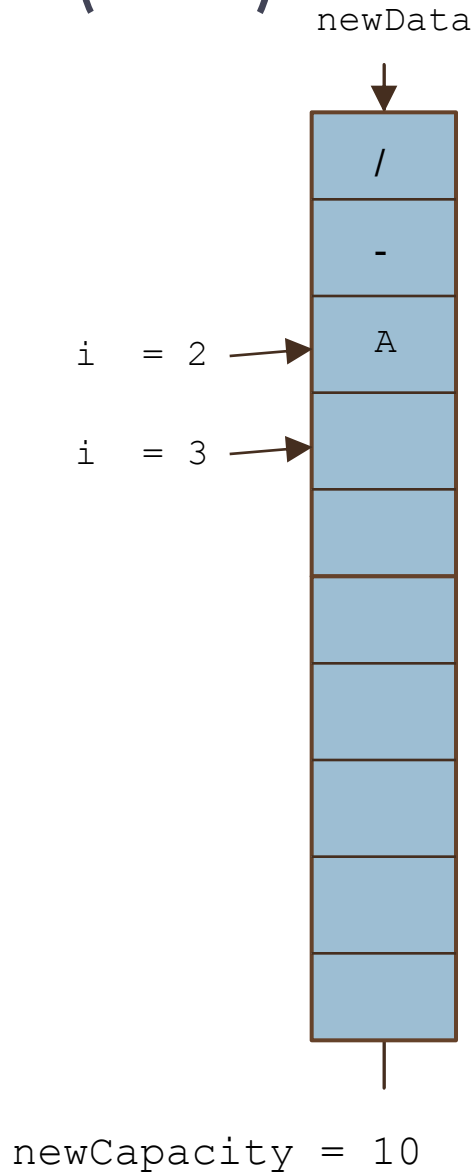
```
size      = 5
```

```
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;  
}
```

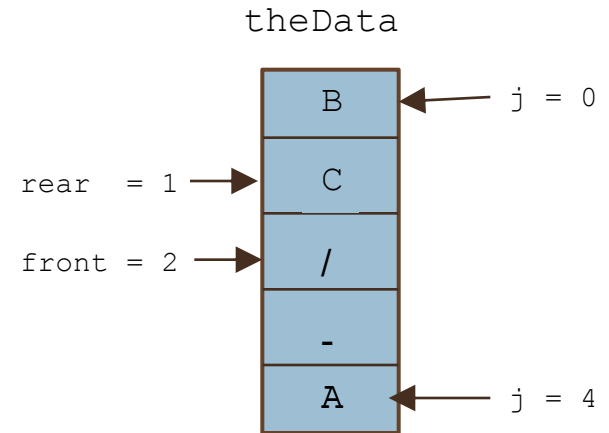
Implementing a Queue Using a Circular Array (cont.)



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

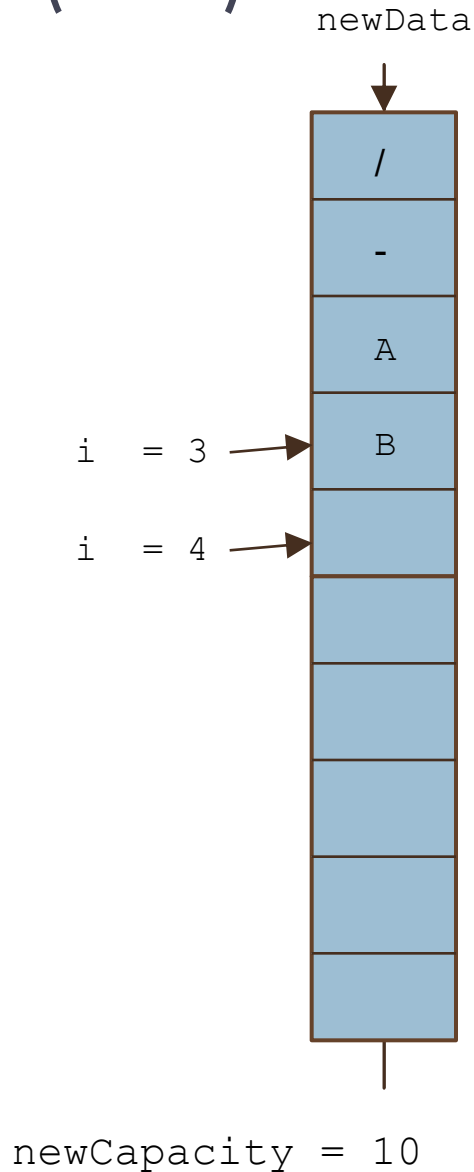
```
size      = 5
```

```
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;  
}
```

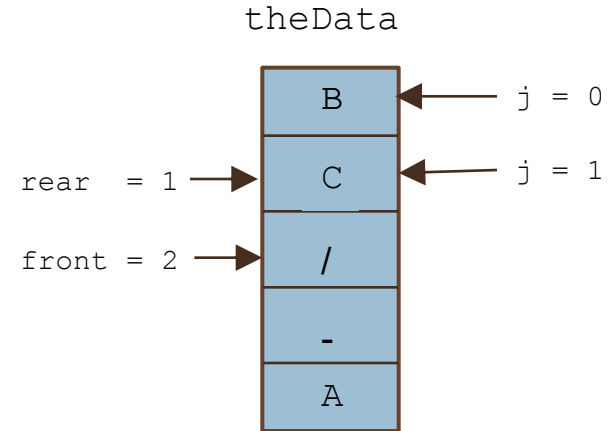
Implementing a Queue Using a Circular Array (cont.)



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

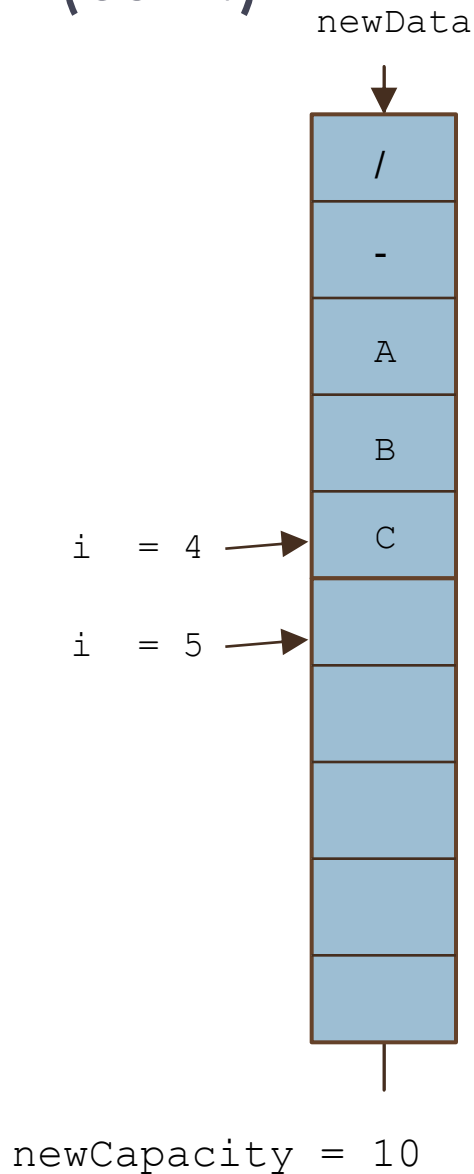
```
size      = 5
```

```
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;  
}
```

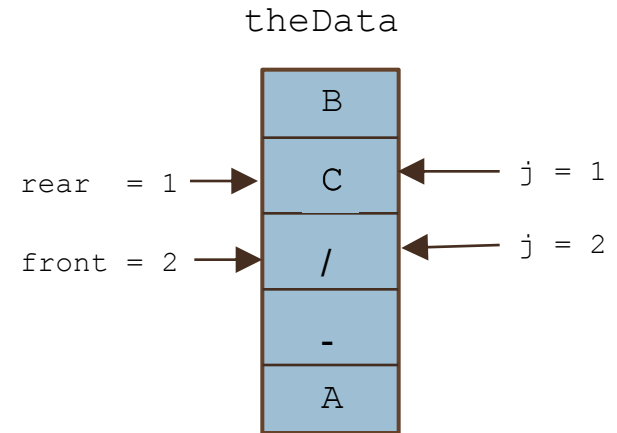

Implementing a Queue Using a Circular Array (cont.)



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

```
size      = 5
```

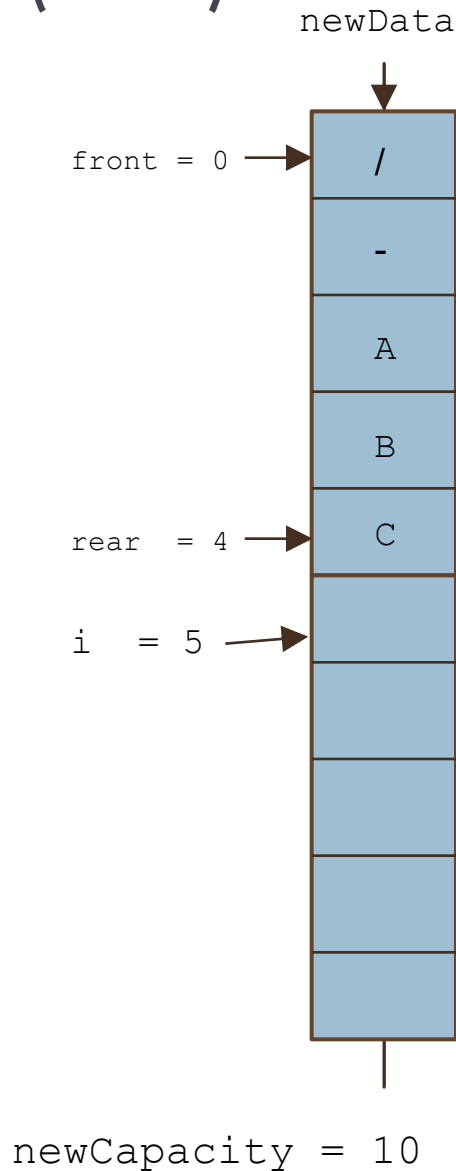
```
capacity = 5
```



```
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

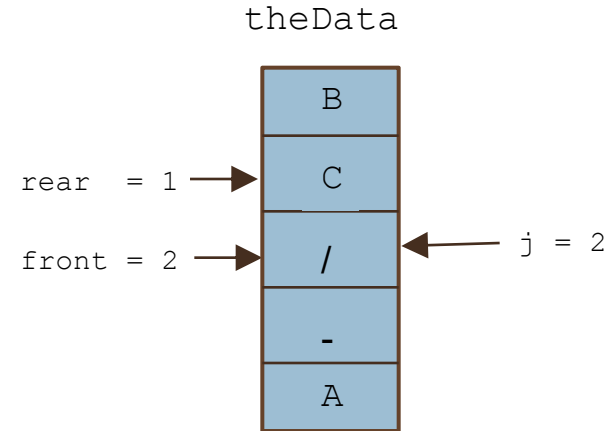
(cont.)



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

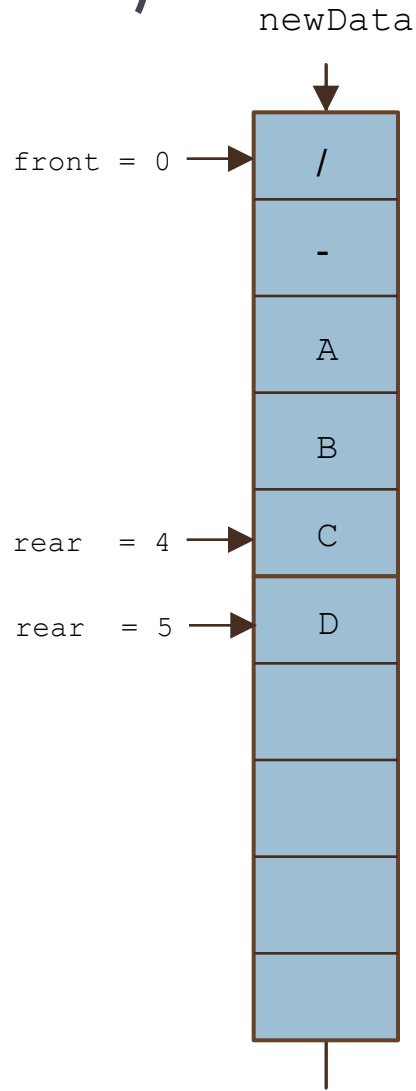
```
size      = 5
```

```
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.)

44

```
// Public Methods
/** Inserts an item at the rear of the queue.
    post: item is added to the rear of the queue.
    @param item The element to add
    @return true (always successful)    */
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.)

45

```
/** Removes the entry at the front of the queue and returns it
    if the queue is not empty.
    post: front references item that was second in the queue.
    @return The item removed if successful or null if not
 */
public E poll() {
    if (size == 0) {
        return null;
    }
    E result = theData[front];
    front = (front + 1) % capacity;
    size--;
    return result;
}
```

Comparing the Three Implementations

46

□ Computation time

- All three implementations (double-linked list, single-linked list, circular array) 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.)

47

□ 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 4.8

Deque **Interface**

49

- 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 recommended implementation

Deque Interface (cont.)

50

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**

51

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.)

52

- 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
push(e)	addFirst(e)
pop()	removeFirst()
peek()	peekFirst()
empty()	isEmpty()