Queues

Queue ADT

Ref: "Data Structures and Algorithm Analysis", C. Shaffer, pp. 133–140.

- The Queue ADT stores arbitrary objects
- Insertions and deletions follow the first-in first-out scheme
- Insertions are at the rear of the queue and removals are at the front of the queue
- 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

Queue ADT

- Auxiliary queue operations:
  - object front(): 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
- Attempting the execution of dequeue or front on an empty queue throws an EmptyQueueException

Source : Dr. Calvin & textbook

Queue Applications

- Access to shared resources in an orderly fashion
  1. Printer queue
  2. CPU and I/O queues to allow multitasking in computers
- Simulate systems with waiting like banking, CPU, traffic lights etc. to assess their performance for various queueing strategies and resource constraints
- Efficient data structures for solving various graph search problems.

Source : Dr. Calvin & textbook

Specification for a Queue Interface

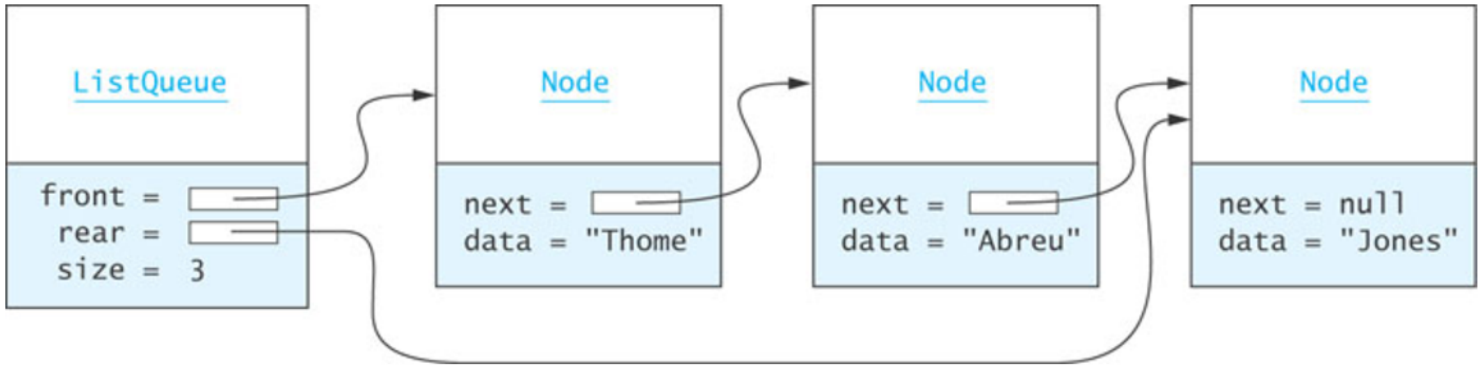
Method	Behavior
`boolean offer(E item)`	Inserts `item` at the rear of the queue. Returns **`true`** if successful; returns **`false`** if the item could not be inserted.
`E remove()`	Removes the entry at the front of the queue and returns it if the queue is not empty. If the queue is empty, throws a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

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

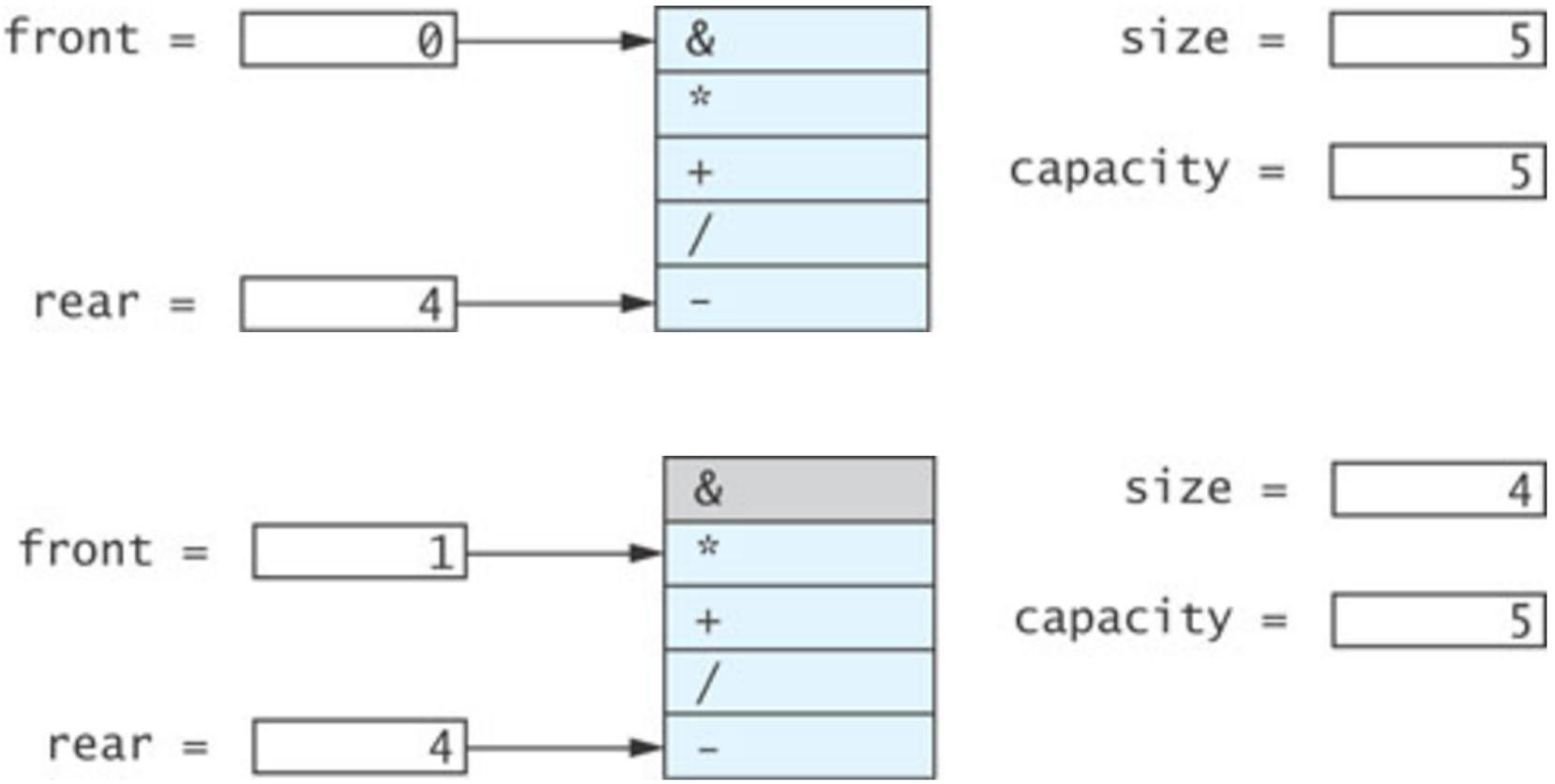
- Insertions are at the rear of a queue and removals are from the front
- We need a reference to the last list node so that insertions can be performed at O(1)
- The number of elements in the queue is changed by methods insert and remove



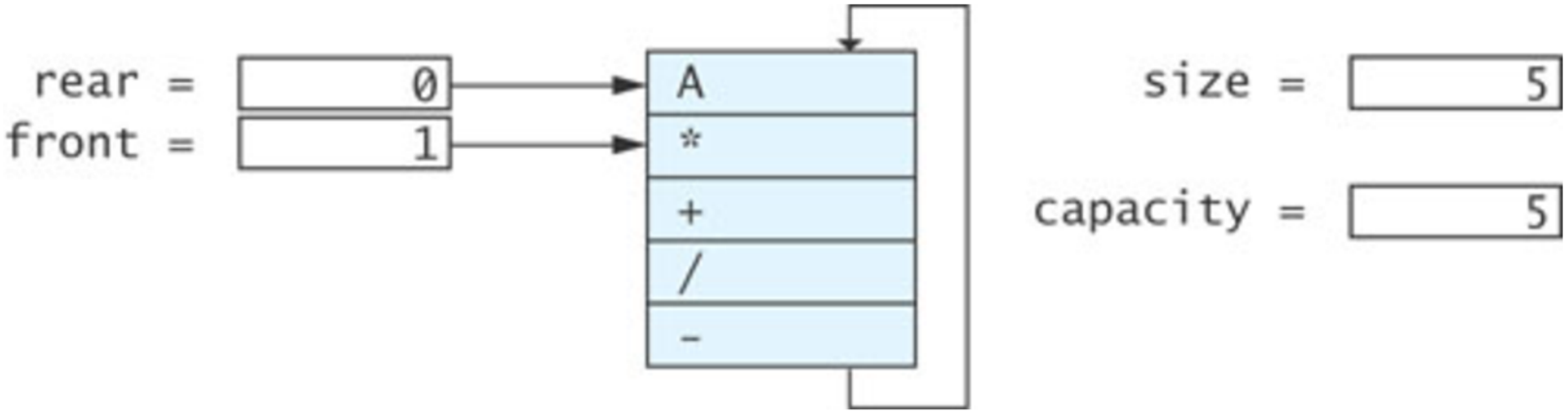
Implementing a Queue Using a Circular Array

- 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 can avoid these inefficiencies in an array

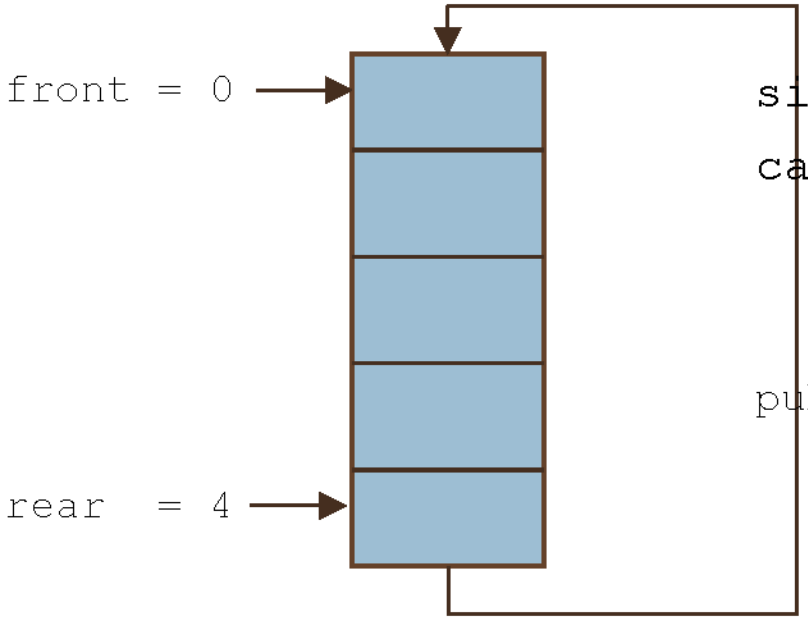
Implementing a Queue Using a Circular Array (cont.)



Implementing a Queue Using a Circular Array (cont.)



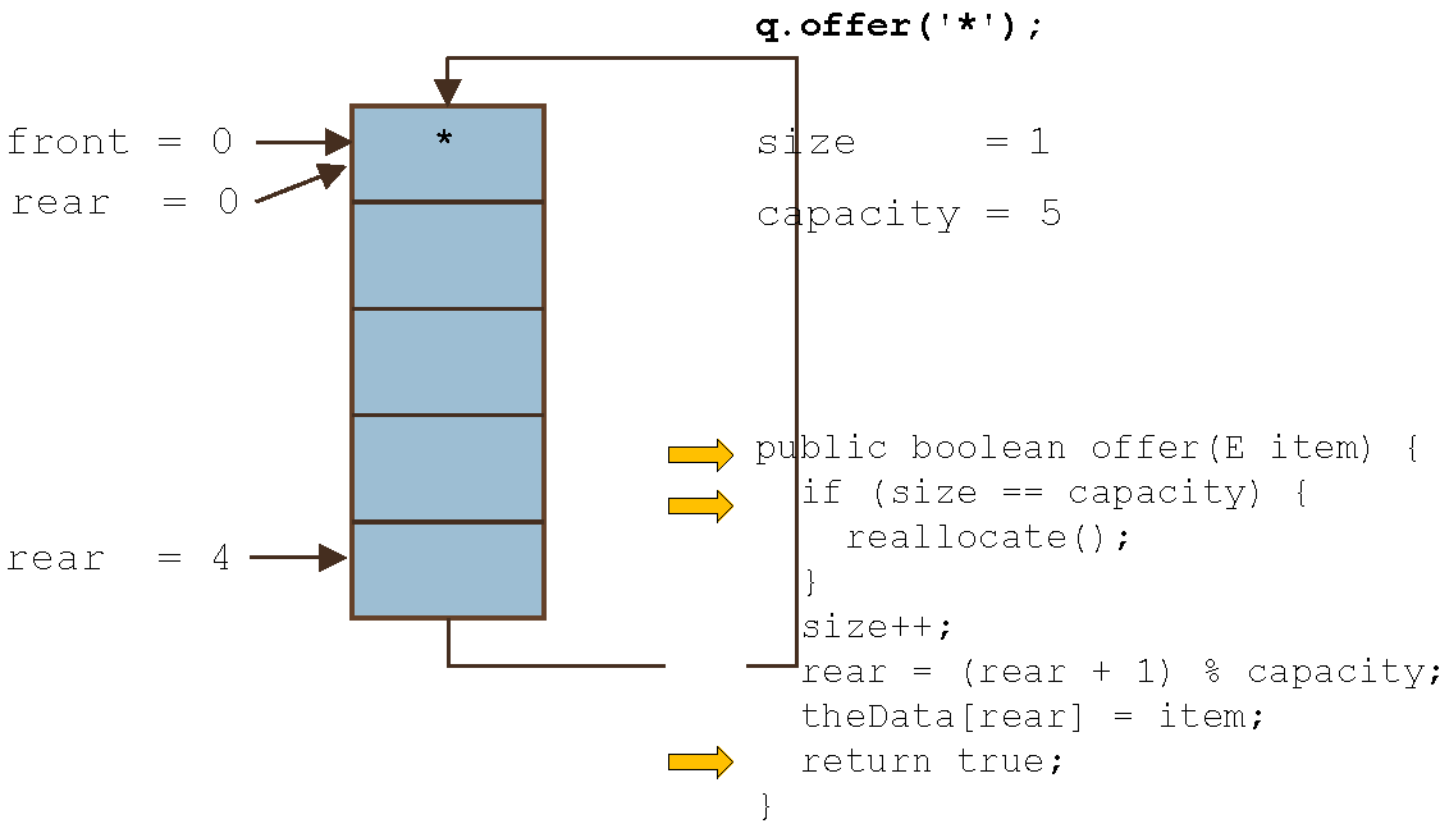
Implementing a Queue Using a Circular Array (cont.)



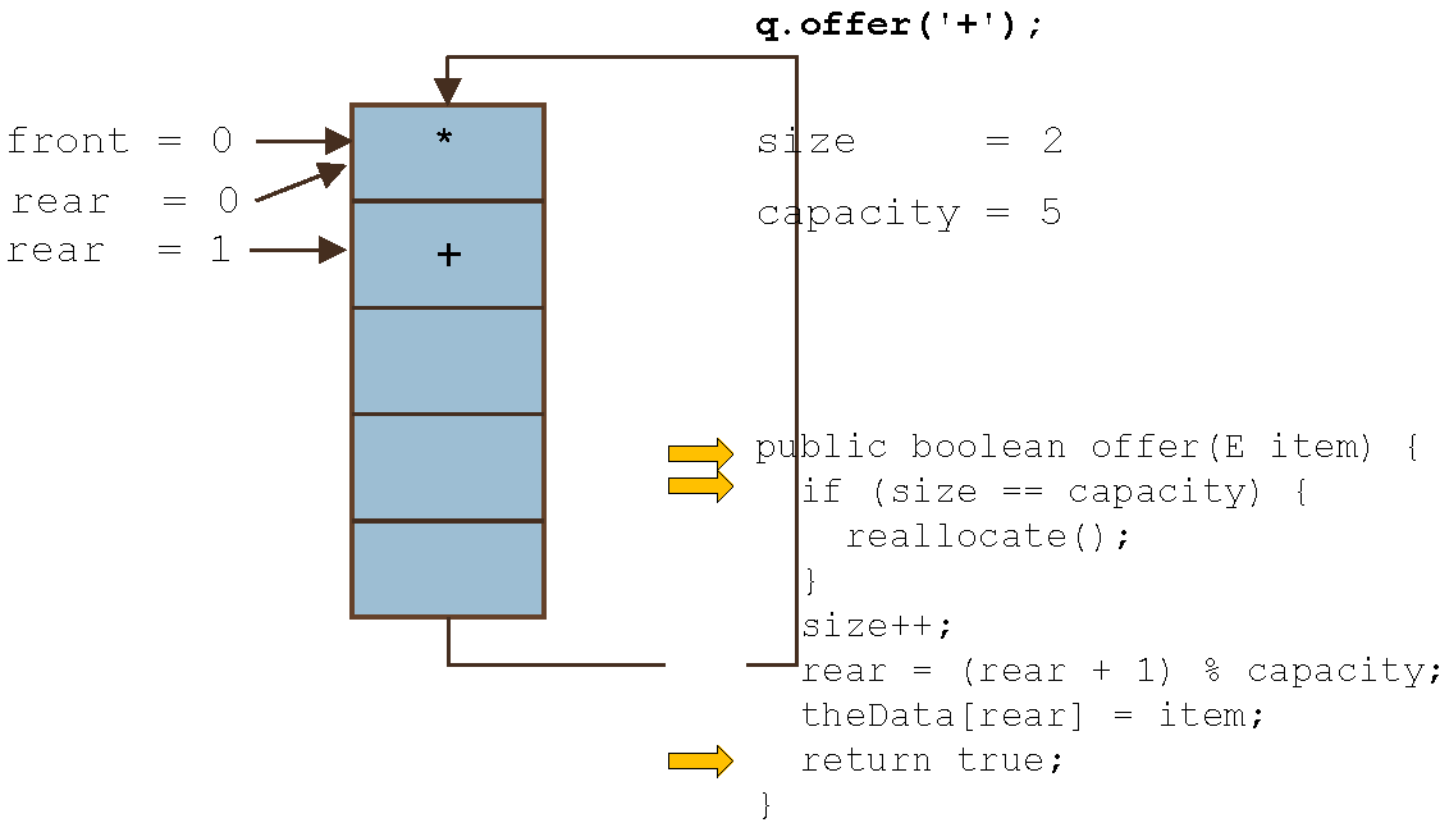

```
ArrayQueue q = new ArrayQueue(5);

size     = 0
capacity = 5
```

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

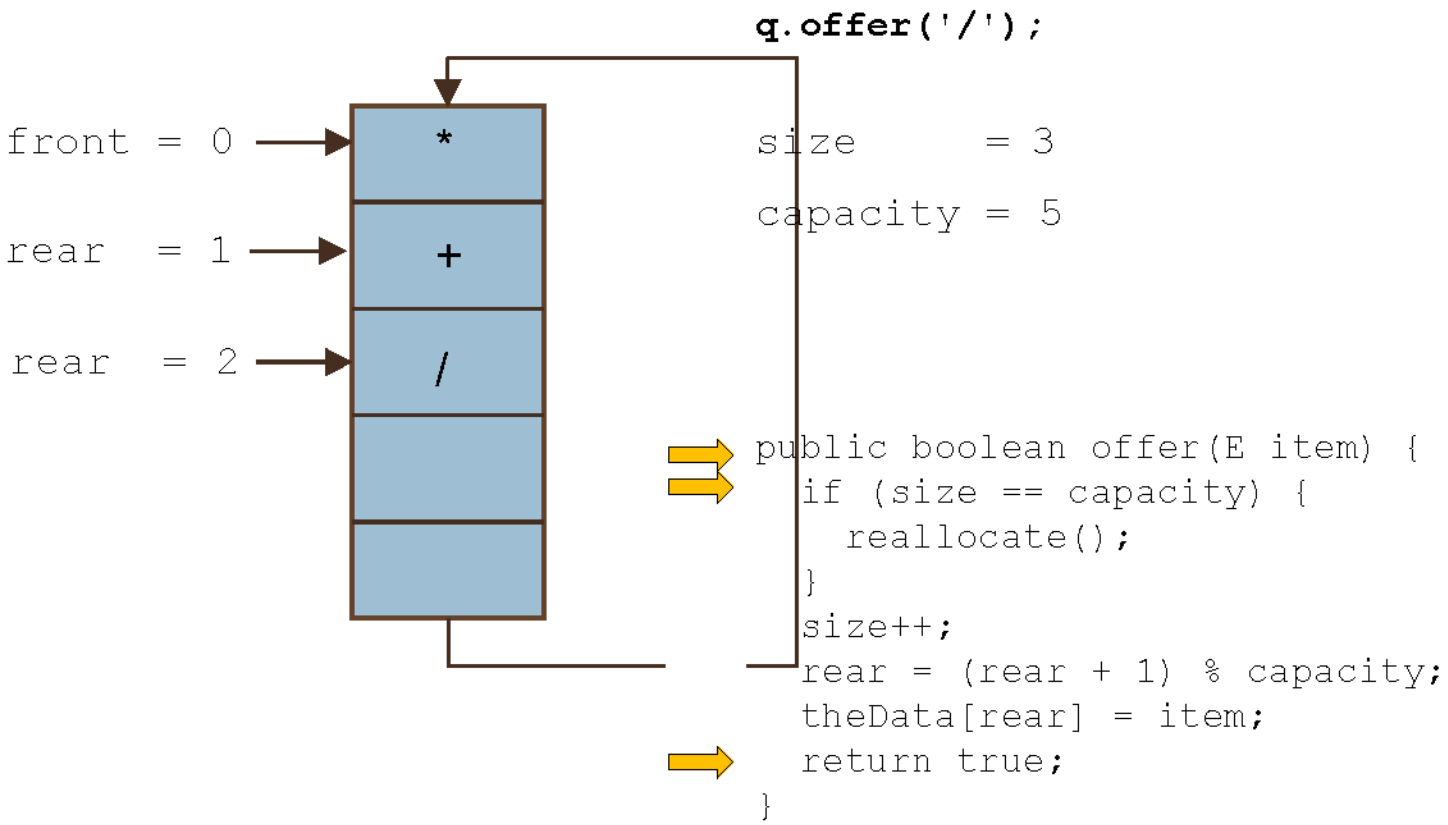
Implementing a Queue Using a Circular Array (cont.)



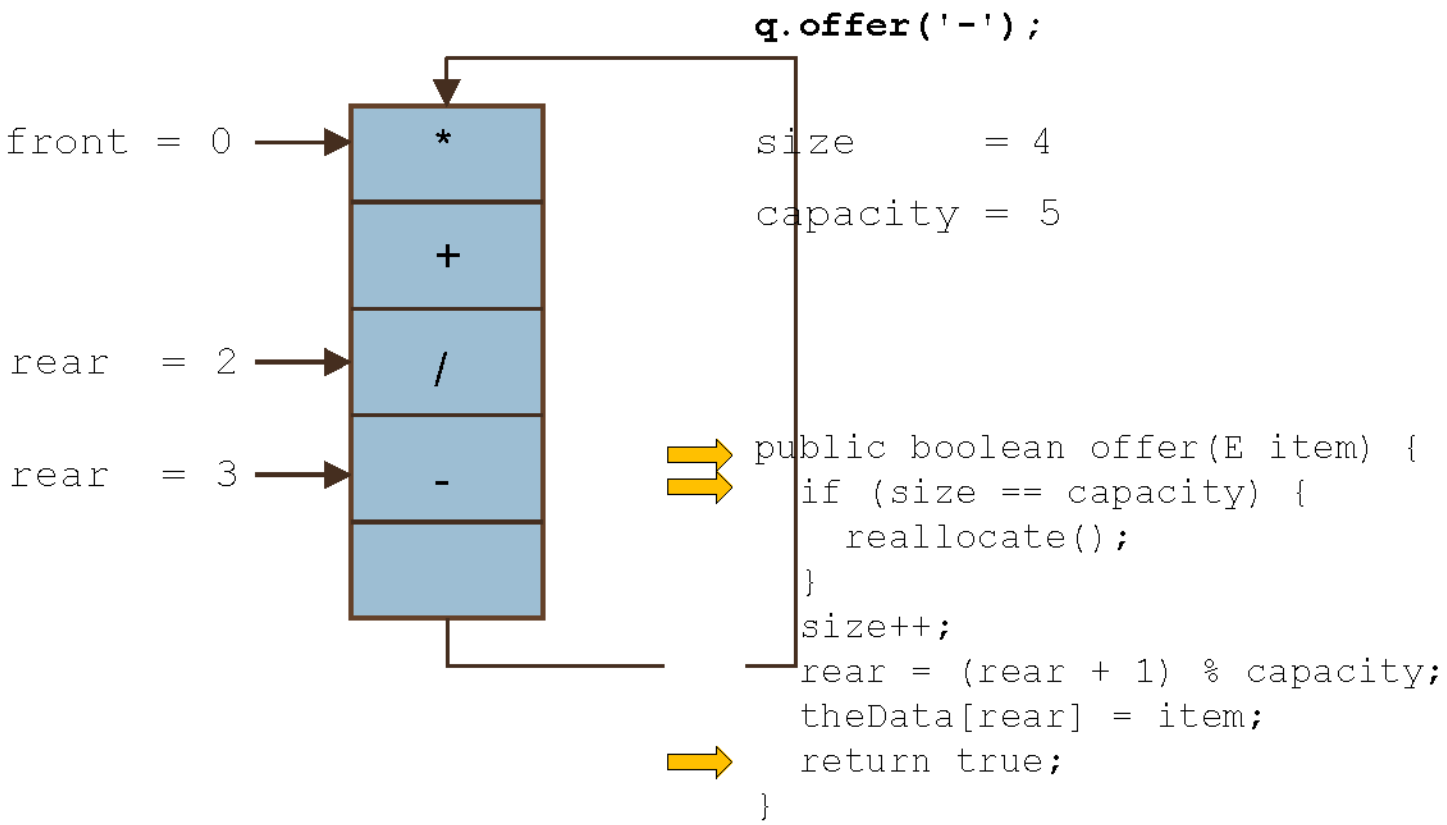
Implementing a Queue Using a Circular Array (cont.)



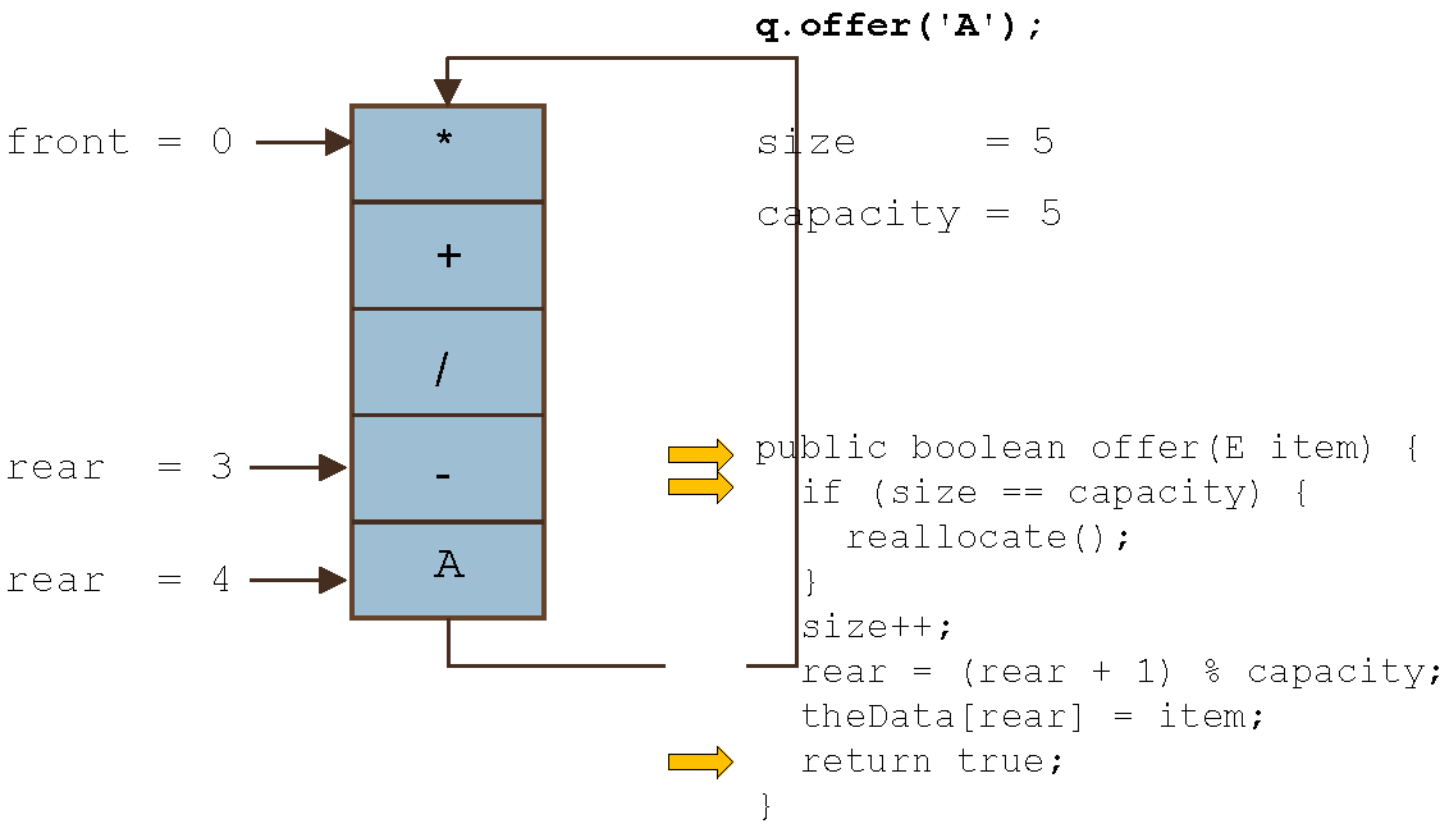
Implementing a Queue Using a Circular Array (cont.)



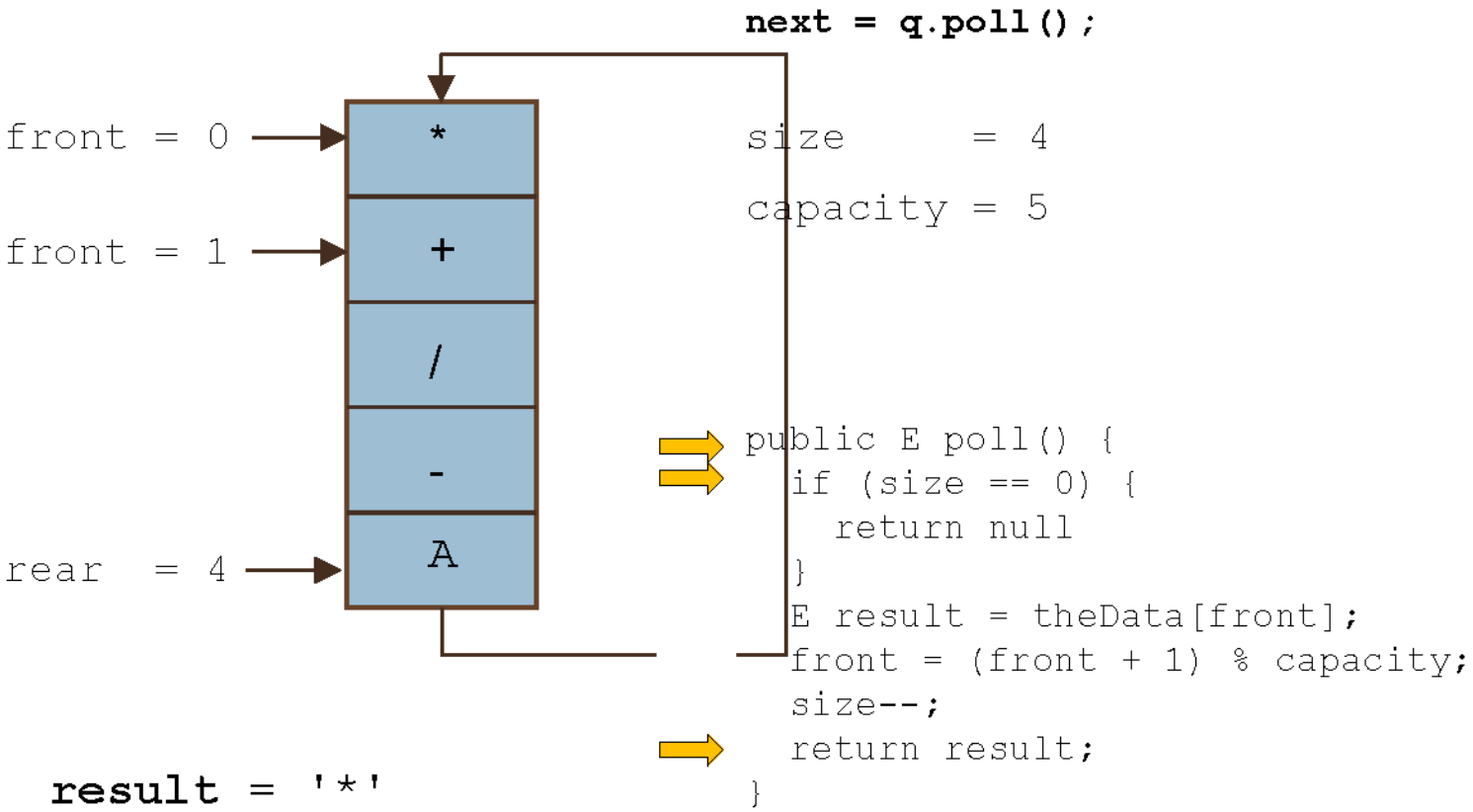
Implementing a Queue Using a Circular Array (cont.)



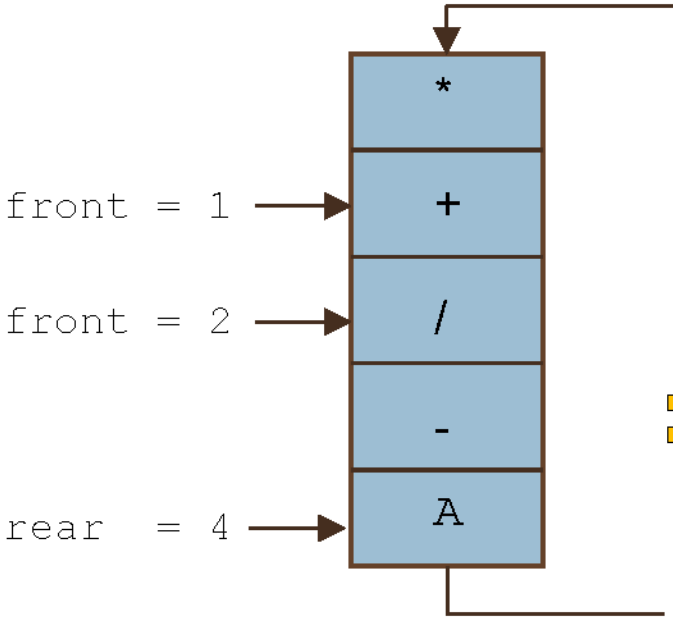
Implementing a Queue Using a Circular Array (cont.)



Implementing a Queue Using a Circular Array (cont.)



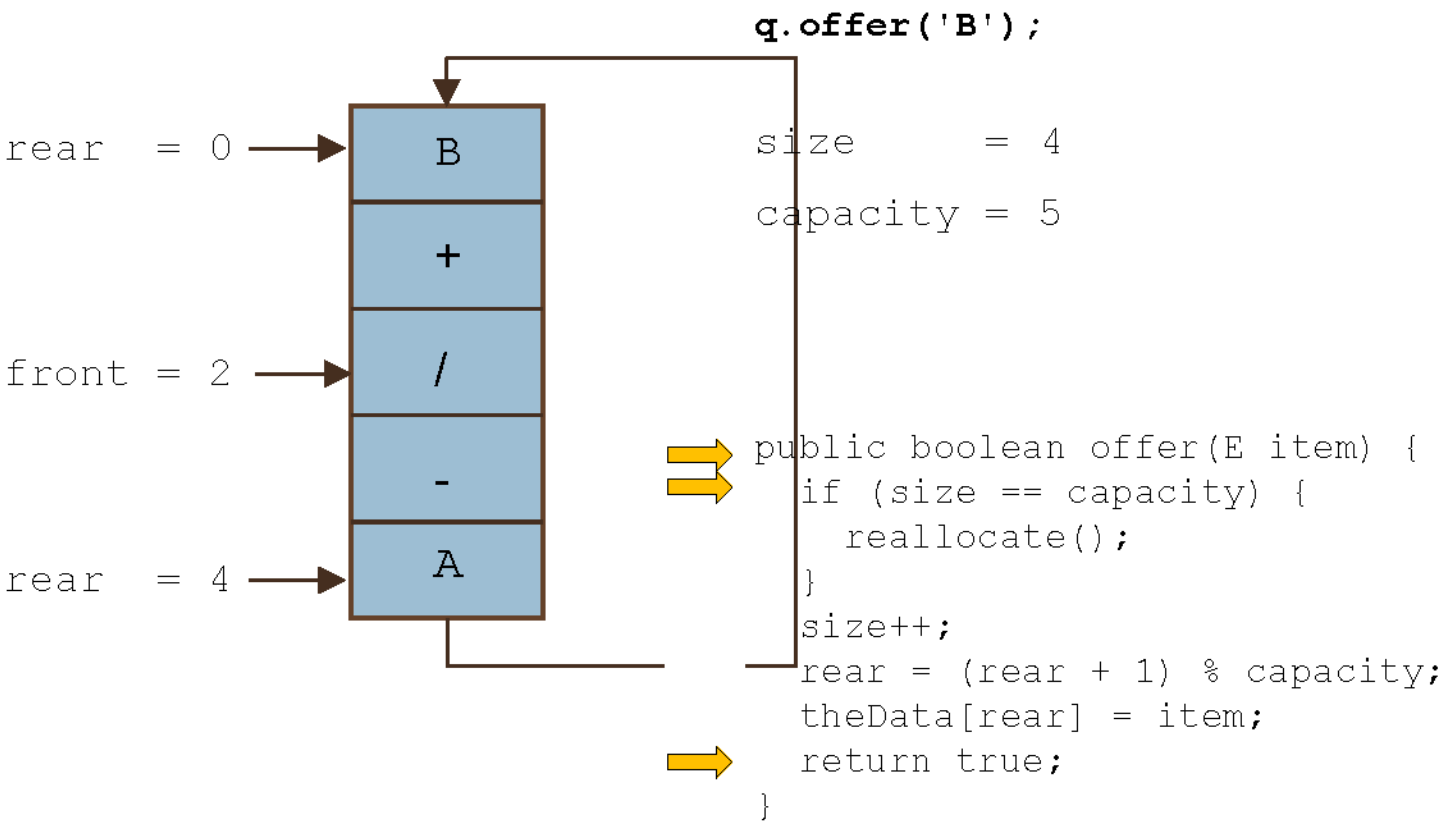
Implementing a Queue Using a Circular Array (cont.)



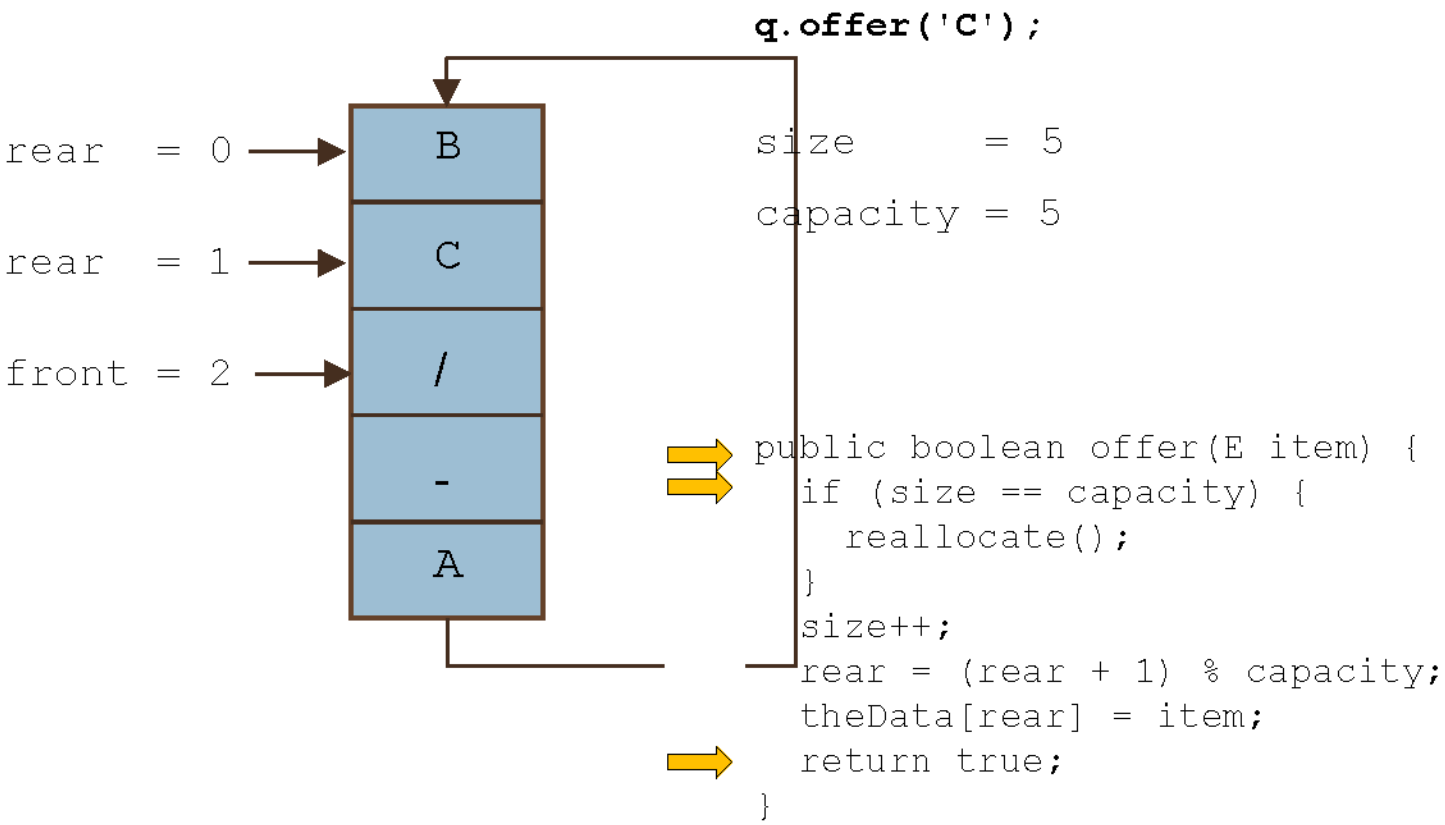

```
next = q.poll();

size     = 3
capacity = 5

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

result = '+'

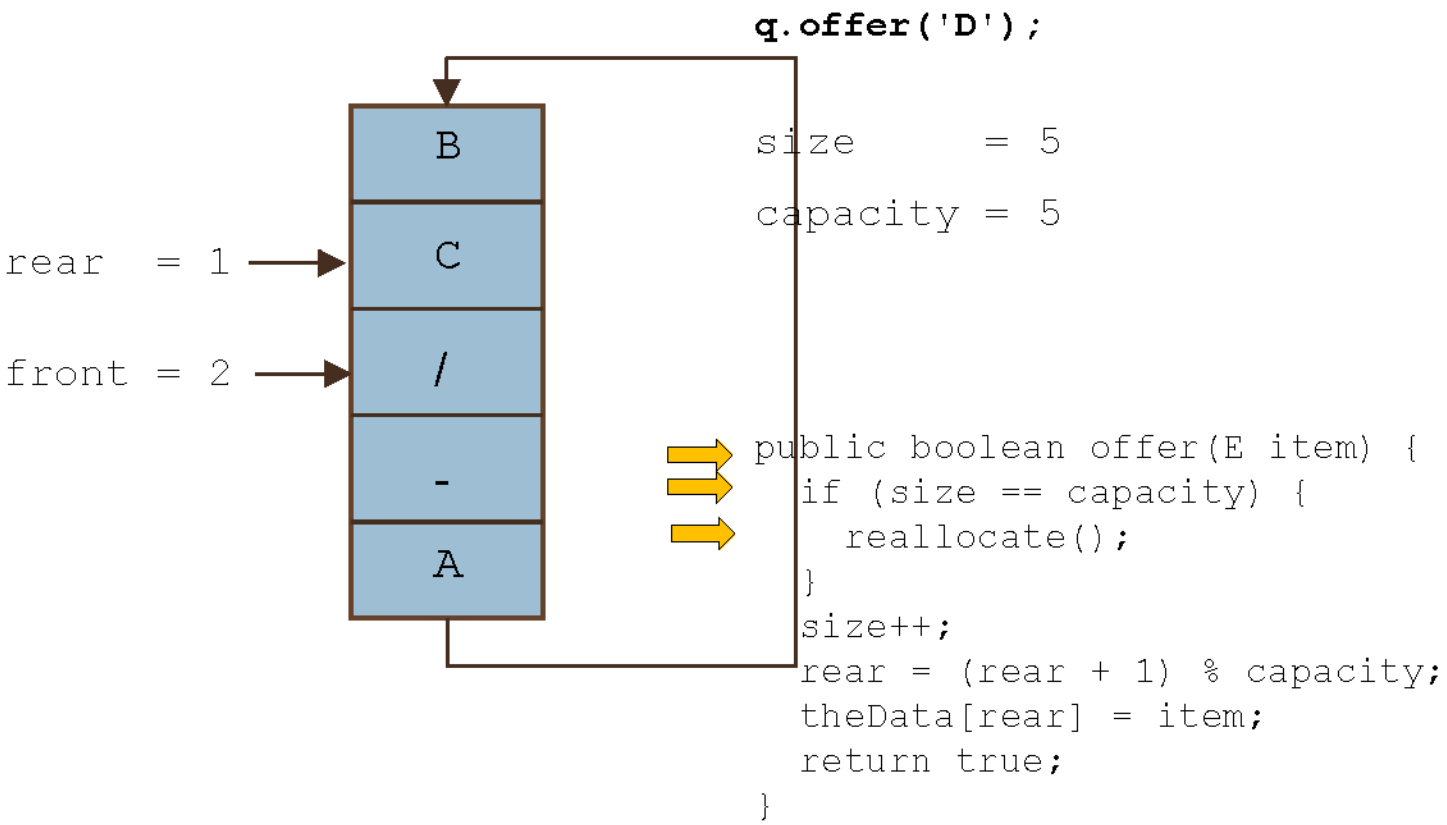
Implementing a Queue Using a Circular Array (cont.)



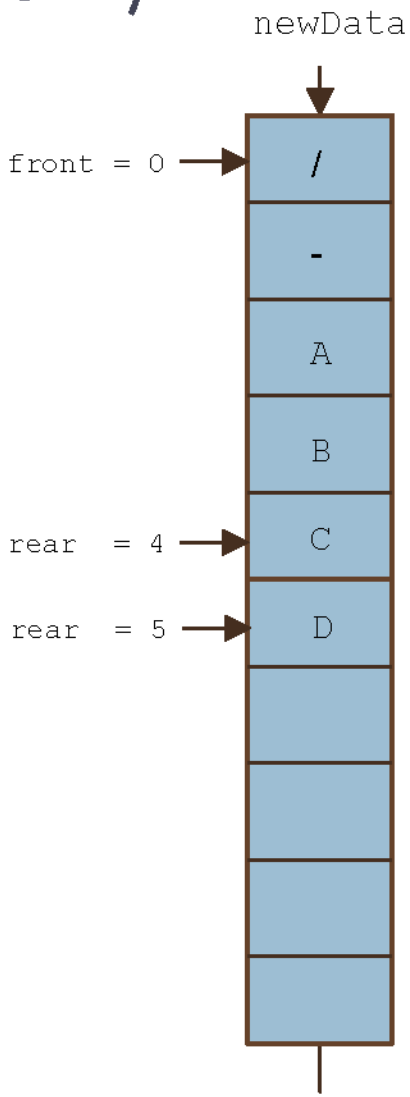
Implementing a Queue Using a Circular Array (cont.)



Implementing a Queue Using a Circular Array (cont.)



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.)


```
// 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.)


```
/** 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

- 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)