

CHAPTER 4

Stacks & Queues

Chapter Objectives

- ❑ To learn about the stack data type and how to use its four methods
- ❑ To understand how Java implements a stack
- ❑ To learn how to implement a stack using an underlying array or linked list
- ❑ To see how to use a stack to perform various applications, including finding palindromes, testing for balanced (properly nested) parentheses, and evaluating arithmetic expressions
- ❑ To learn how to represent a waiting line (queue) and learn how to use the methods in the queue interface
- ❑ 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

Stack Abstract Data Type

Section 4.1

Stack Abstract Data Type

- A stack is one of the most commonly used data structures in computer science
- A stack can be compared to a Pez dispenser
 - ▣ Only the top item can be accessed
 - ▣ You can extract only one item at a time
- The top element in the stack is the last added to the stack (most recently)
- The stack's storage policy is *Last-In, First-Out*, or *LIFO*



Specification of the Stack Abstract Data Type

- Only the top element of a stack is visible; therefore the number of operations performed by a stack are few
- We need the ability to
 - ▣ test for an empty stack (`empty`)
 - ▣ inspect the top element (`peek`)
 - ▣ retrieve the top element (`pop`)
 - ▣ put a new element on the stack (`push`)

Methods	Behavior
<code>boolean empty()</code>	Returns true if the stack is empty; otherwise, returns false .
<code>E peek()</code>	Returns the object at the top of the stack without removing it.
<code>E pop()</code>	Returns the object at the top of the stack and removes it.
<code>E push(E obj)</code>	Pushes an item onto the top of the stack and returns the item pushed.

A Stack of Strings

Jonathan
Dustin
Robin
Debbie
Rich

(a)

Dustin
Robin
Debbie
Rich

(b)

Philip
Dustin
Robin
Debbie
Rich

(c)

- “Rich” is the oldest element on the stack and “Jonathan” is the youngest (Figure a)
- `String last = names.peek();` stores a reference to “Jonathan” in `last`
- `String temp = names.pop();` removes “Jonathan” and stores a reference to it in `temp` (Figure b)
- `names.push("Philip");` pushes “Philip” onto the stack (Figure c)

Stack Applications

Section 4.2

Finding Palindromes

- Palindrome: a string that reads identically in either direction, letter by letter (ignoring case)
 - ▣ kayak
 - ▣ "I saw I was I"
 - ▣ "Able was I ere I saw Elba"
 - ▣ "Level madam level"
- Problem: Write a program that reads a string and determines whether it is a palindrome

Finding Palindromes (cont.)

Data Fields	Attributes
<code>private String inputString</code>	The input string.
<code>private Stack<Character> charStack</code>	The stack where characters are stored.
Methods	Behavior
<code>public PalindromeFinder(String str)</code>	Initializes a new <code>PalindromeFinder</code> object, storing a reference to the parameter <code>str</code> in <code>inputString</code> and pushing each character onto the stack.
<code>private void fillStack()</code>	Fills the stack with the characters in <code>inputString</code> .
<code>private String buildReverse()</code>	Returns the string formed by popping each character from the stack and joining the characters. Empties the stack.
<code>public boolean isPalindrome()</code>	Returns true if <code>inputString</code> and the string built by <code>buildReverse</code> have the same contents, except for case. Otherwise, returns false .

Finding Palindromes (cont.)

```
import java.util.*;

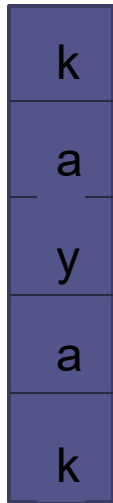
public class PalindromeFinder {
    private String inputString;
    private Stack<Character> charStack = new
                                                Stack<Character>();

    public PalindromeFinder(String str) {
        inputString = str;
        fillStack(); // fills the stack with the characters in
                     // inputString
    }
    ...
}
```

Finding Palindromes (cont.)

□ Solving using a stack:

- ▣ Push each string character, from left to right, onto a stack



k a y a k

```
private void fillStack() {  
    for(int i = 0; i < inputString.length(); i++) {  
        charStack.push(inputString.charAt(i));  
    }  
}
```

Finding Palindromes (cont.)

□ Solving using a stack:

- ▣ Pop each character off the stack, appending each to the `StringBuilder` result



k a y a k

```
private String buildReverse(){
    StringBuilder result = new StringBuilder();
    while(!charStack.empty()) {
        result.append(charStack.pop());
    }
    return result.toString();
}
```

Finding Palindromes (cont.)

...

```
public boolean isPalindrome() {  
    return inputString.equalsIgnoreCase(buildReverse());  
}  
}
```

Testing

- To test this class using the following inputs:
 - ▣ a single character (always a palindrome)
 - ▣ multiple characters in a word
 - ▣ multiple words
 - ▣ different cases
 - ▣ even-length strings
 - ▣ odd-length strings
 - ▣ the empty string (considered a palindrome)

Balanced Parentheses

- When analyzing arithmetic expressions, it is important to determine whether an expression is balanced with respect to parentheses

$(a + b * (c / (d - e))) + (d / e)$

- The problem is further complicated if braces or brackets are used in conjunction with parentheses
- The solution is to use stacks!

Balanced Parentheses (cont.)

Method	Behavior
<code>public static boolean isBalanced(String expression)</code>	Returns true if expression is balanced with respect to parentheses and false if it is not.
<code>private static boolean isOpen(char ch)</code>	Returns true if ch is an opening parenthesis.
<code>private static boolean isClose(char ch)</code>	Returns true if ch is a closing parenthesis.

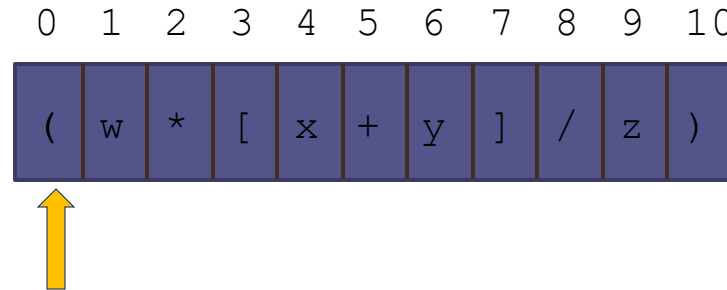
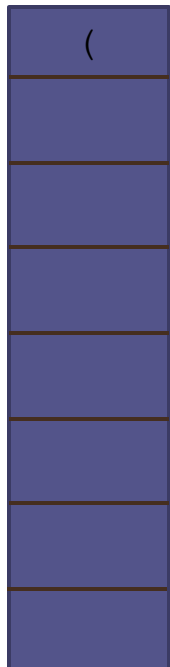
Balanced Parentheses (cont.)

Algorithm for method `isBalanced`

1. Create an empty stack of characters.
2. Assume that the expression is balanced (`balanced` is `true`).
3. Set `index` to 0.
4. `while` `balanced` is `true` and `index` < the expression's length
5. Get the next character in the data string.
6. if the next character is an opening parenthesis
7. Push it onto the stack.
8. else if the next character is a closing parenthesis
9. Pop the top of the stack.
10. if stack was empty or its top does not match the closing parenthesis
11. Set `balanced` to `false`.
12. Increment `index`.
13. Return `true` if `balanced` is `true` and the stack is empty.

Balanced Parentheses (cont.)

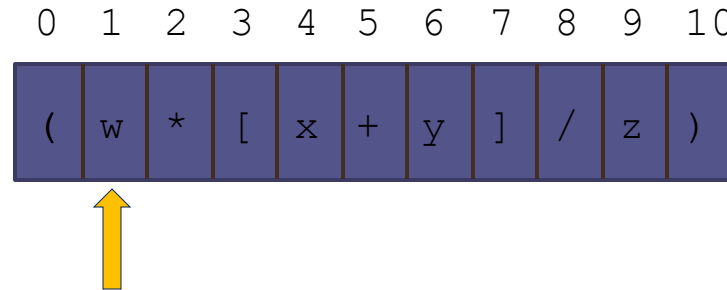
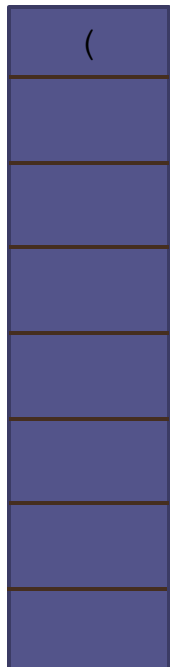
Expression: (w * [x + y] / z)



balanced : **true**
index : 0

Balanced Parentheses (cont.)

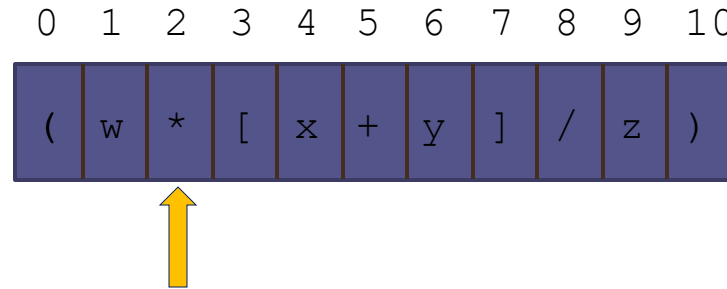
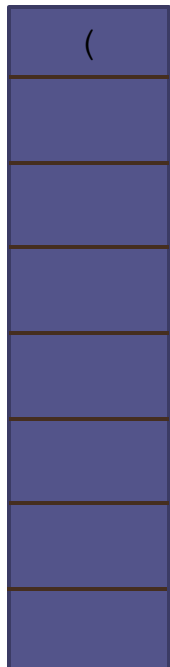
Expression: (w * [x + y] / z)



balanced : **true**
index : 1

Balanced Parentheses (cont.)

Expression: (w * [x + y] / z)

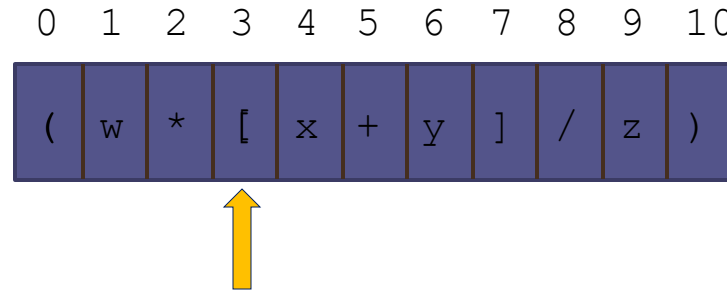
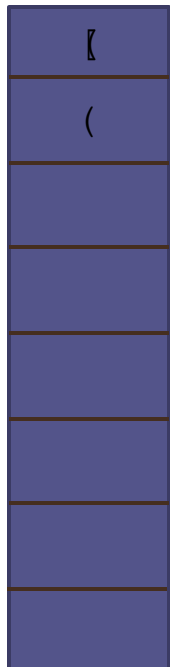


balanced : **true**

index : 2

Balanced Parentheses (cont.)

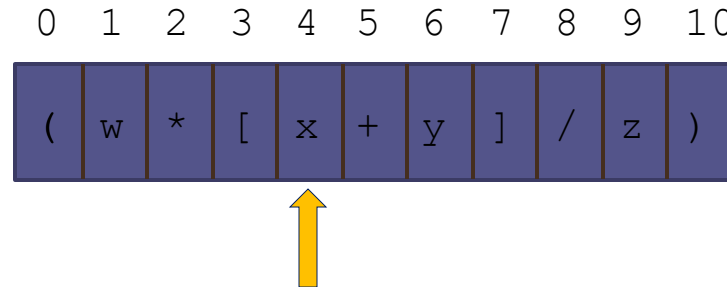
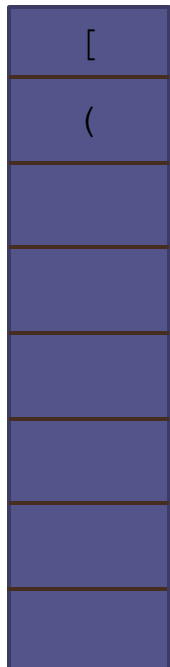
Expression: (w * [x + y] / z)



balanced : **true**
index : 3

Balanced Parentheses (cont.)

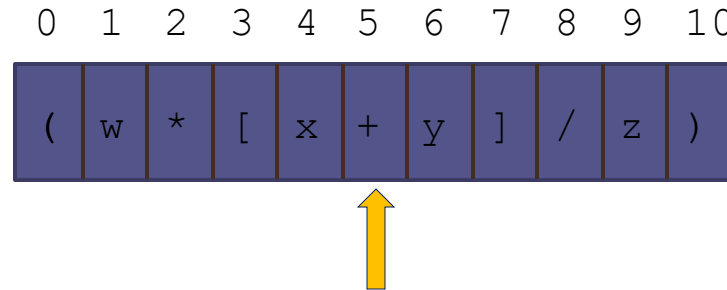
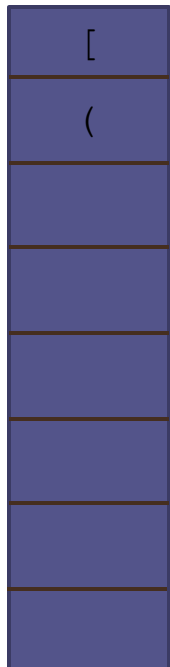
Expression: (w * [x + y] / z)



balanced : **true**
index : 4

Balanced Parentheses (cont.)

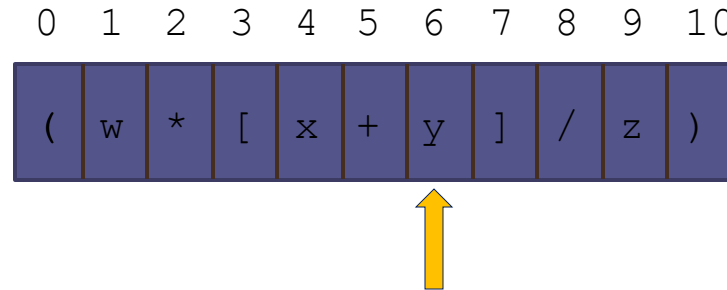
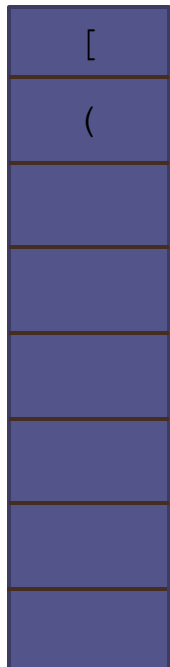
Expression: (w * [x + y] / z)



balanced : **true**
index : 5

Balanced Parentheses (cont.)

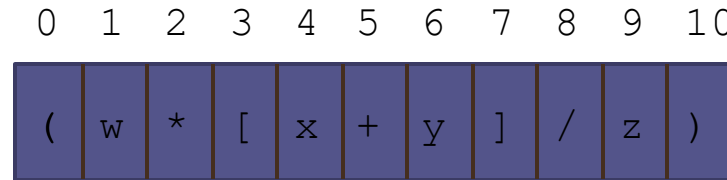
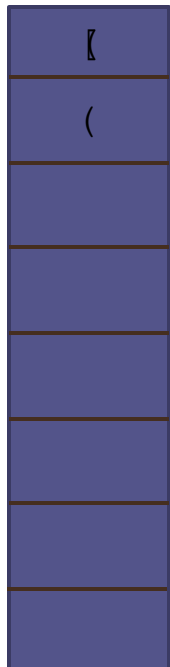
Expression: (w * [x + y] / z)



balanced : **true**
index : 6

Balanced Parentheses (cont.)

Expression: (w * [x + y] / z)

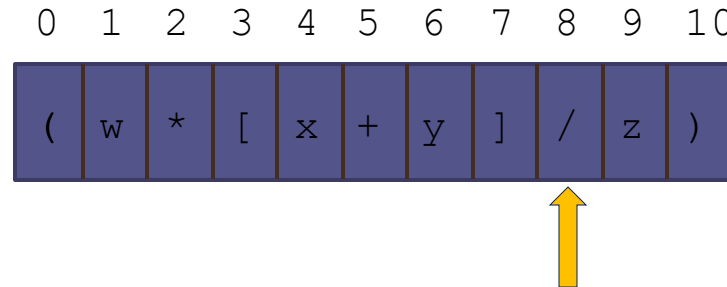
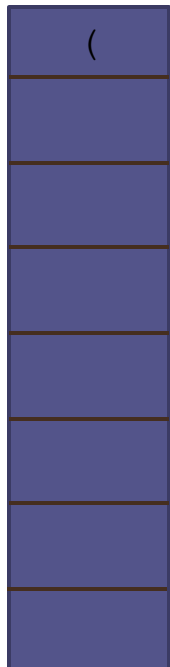


Matches!
Balanced still true

balanced : **true**
index : 7

Balanced Parentheses (cont.)

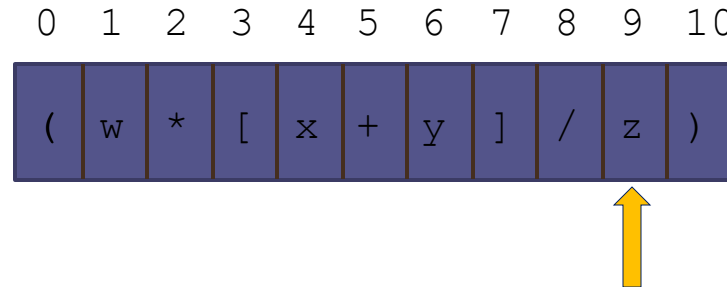
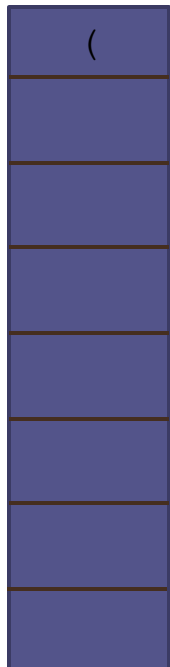
Expression: (w * [x + y] / z)



balanced : **true**
index : 8

Balanced Parentheses (cont.)

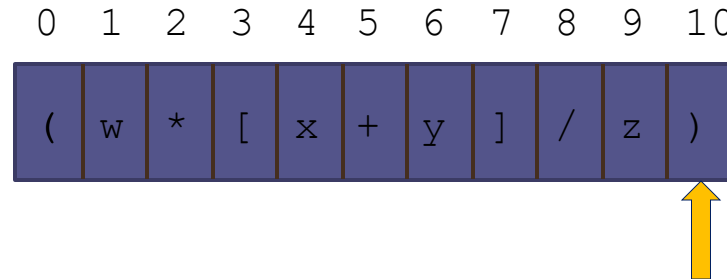
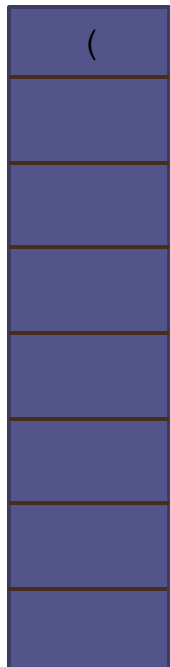
Expression: (w * [x + y] / z)



balanced : **true**
index : 9

Balanced Parentheses (cont.)

Expression: (w * [x + y] / z)



Matches!
Balanced **still** true

balanced : **true**
index : 10

Testing

- Provide a variety of input expressions displaying the result `true` **or** `false`
- Try several levels of nested parentheses
- Try nested parentheses where corresponding parentheses are not of the same type
- Try unbalanced parentheses
- No parentheses at all!
- PITFALL: attempting to pop an empty stack will throw an `EmptyStackException`. You can guard against this by either testing for an empty stack or catching the exception

Implementing a Stack

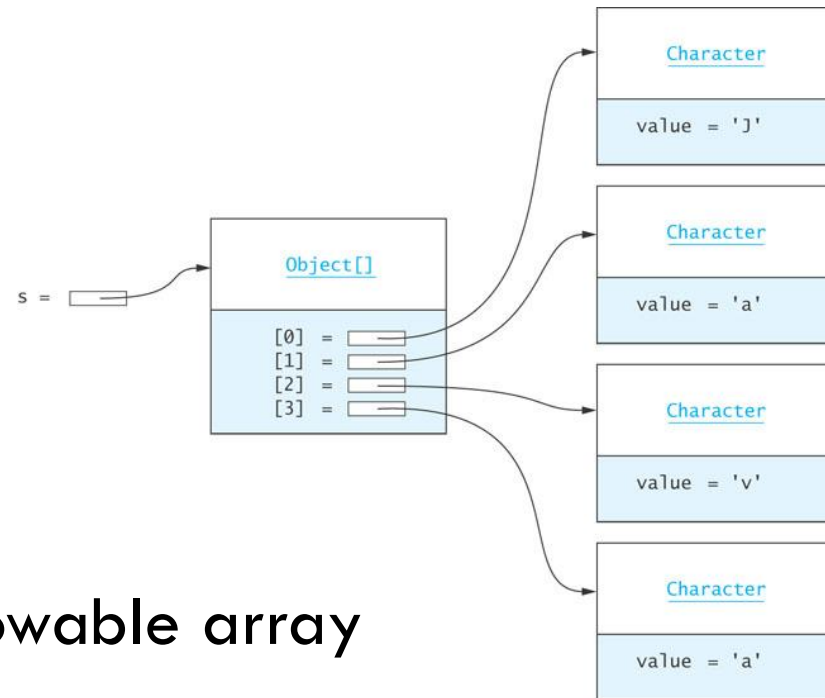
Section 4.3

Implementing a Stack as an Extension of Vector

- The Java API includes a `Stack` class as part of the package `java.util`:

```
public class Stack<E> extends Vector<E>
```

- The `Vector` class implements a growable array of objects
- Elements of a `Vector` can be accessed using an integer index and the size can grow or shrink as needed to accommodate the insertion and removal of elements



Implementing a Stack as an Extension of Vector (cont.)

- We can use Vector's add method to implement push:

```
public E push(obj E) {  
    add(obj);  
    return obj;  
}
```

- pop can be coded as

```
public E pop throws EmptyStackException {  
    try {  
        return remove (size() - 1);  
    } catch (ArrayIndexOutOfBoundsException ex) {  
        throw new EmptyStackException();  
    }  
}
```


Implementing a Stack as an Extension of Vector (cont.)

- Because a Stack *is a* Vector, all of Vector operations can be applied to a Stack (such as searches and access by index)
- But, since only the top element of a stack should be accessible, this violates the principle of information hiding

Implementing a Stack with a List Component

- As an alternative to a stack as an extension of `Vector`, we can write a class, `ListStack`, that has a `List` component (in the example below, `theData`)
- We can use either the `ArrayList`, `Vector`, or the `LinkedList` classes, as all implement the `List` interface. The `push` method, for example, can be coded as

```
public E push(E obj) {  
    theData.add(obj);  
    return obj;  
}
```

- A class which adapts methods of another class by giving different names to essentially the same methods (`push` instead of `add`) is called an *adapter class*
- Writing methods in this way is called *method delegation*

Implementing a Stack Using an Array

- If we implement a stack as an array we would need . . .

```
public class ArrayStack<E> implements Stack<E> {
    private E[] theData;
    int topOfStack = -1;
    private static final int INITIAL_CAPACITY = 10;

    @SupressWarnings("unchecked")
    public ArrayStack() {
        this(INITIAL_CAPACITY);
    }

    public ArrayStack(int capacity) {
        theData = (E[]) new Object[capacity];
    }
}
```

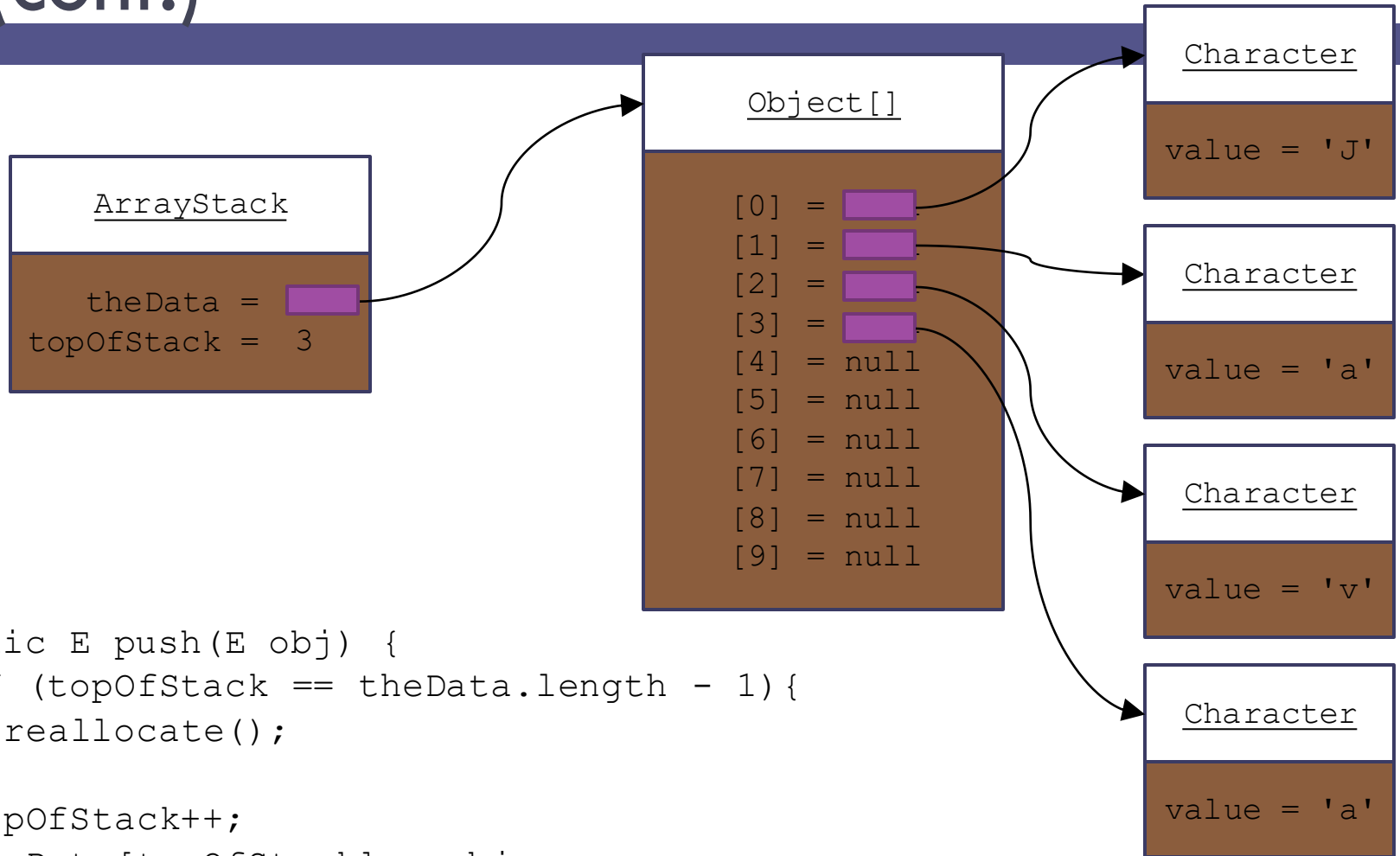
Allocate storage for an array with a default

Keep track of the top of the stack (subscript of the element at the top of the stack; for empty stack = -1)

There is no size variable or method

Implementing a Stack Using an Array

(cont.)



```
public E push(E obj) {  
    if (topOfStack == theData.length - 1) {  
        reallocate();  
    }  
    topOfStack++;  
    theData[topOfStack] = obj;  
    return obj;  
}
```

Implementing a Stack Using an Array

(cont.)

```
@Override
public E pop() {
    if (empty()) {
        throw new EmptyStackException();
    }
    return theData[topOfStack--];
}
```

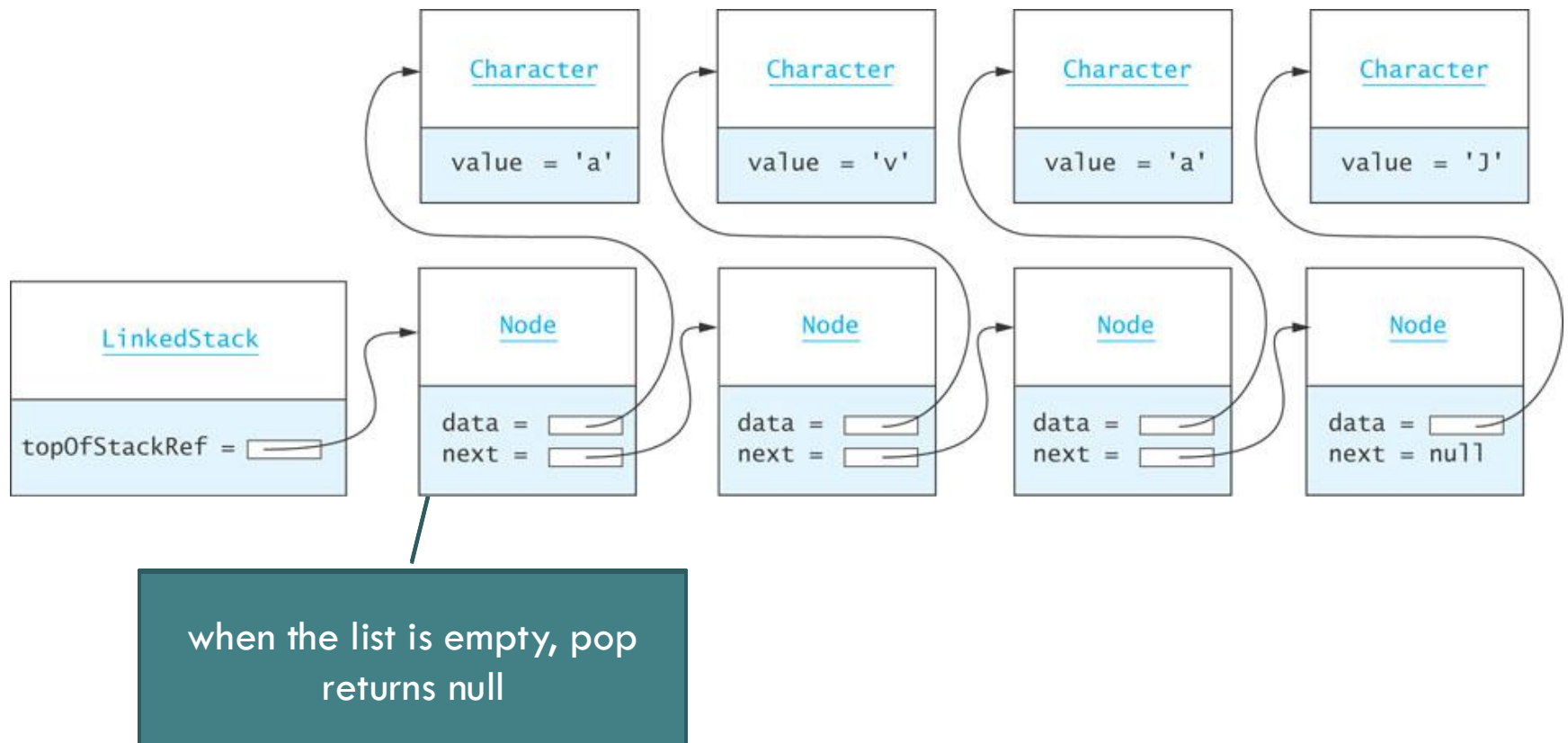
Implementing a Stack Using an Array

(cont.)

- This implementation is $O(1)$, in contrast to the Pez analogy and the “kayak” example, which are both $O(n)$

Implementing a Stack as a Linked Data Structure

- We can also implement a stack using a linked list of nodes



Implementing a Stack as a Linked Data Structure (cont.)

40

- Listing 4.5 (`LinkedStack.java`, pages ?)

Comparison of Stack Implementations

- ❑ Extending a `Vector` (as is done by Java) is a poor choice for stack implementation, since all `Vector` methods are accessible
- ❑ The easiest implementation uses a `List` component (`ArrayList` is the simplest) for storing data
 - An underlying array requires reallocation of space when the array becomes full, and
 - an underlying linked data structure requires allocating storage for links
 - As all insertions and deletions occur at one end, they are constant time, $O(1)$, regardless of the type of implementation used

Additional Stack Applications

Section 4.4

Additional Stack Applications

- Postfix and infix notation
 - Expressions normally are written in infix form, but
 - it easier to evaluate an expression in postfix form since there is no need to group sub-expressions in parentheses or worry about operator precedence

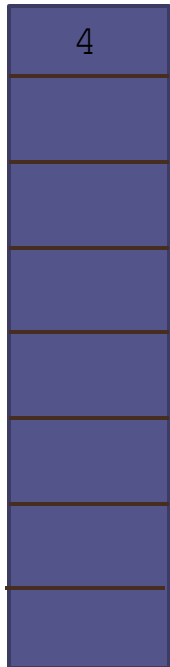
Postfix Expression	Infix Expression	Value
$\boxed{4 \ 7 \ *}$	$4 * 7$	28
$\boxed{4 \ \boxed{7 \ 2 \ +} \ *}$	$4 * (7 + 2)$	36
$\boxed{4 \ 7 \ *} \ 20 \ -$	$(4 * 7) - 20$	8
$\boxed{3 \ \boxed{4 \ 7 \ *} \ 2 \ /} \ +$	$3 + ((4 * 7) / 2)$	17

Evaluating Postfix Expressions

- Write a class that evaluates a postfix expression
- Use the space character as a delimiter between tokens

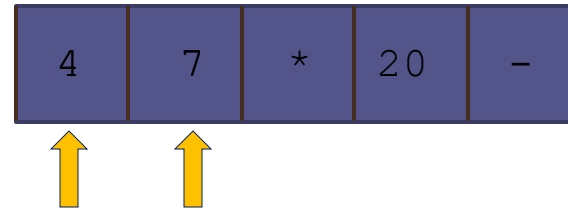
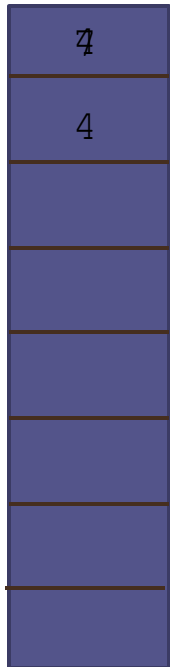
Data Field	Attribute
Stack<Integer> operandStack	The stack of operands (Integer objects).
Method	Behavior
public int eval(String expression)	Returns the value of expression.
private int evalOp(char op)	Pops two operands and applies operator op to its operands, returning the result.
private boolean isOperator(char ch)	Returns true if ch is an operator symbol.

Evaluating Postfix Expressions (cont.)



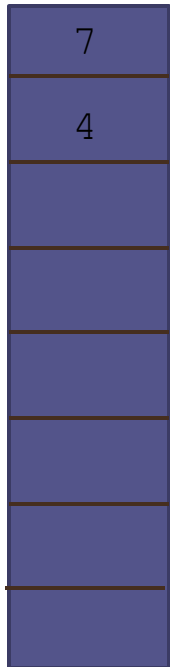
- ➔ 1. create an empty stack of integers
- ➔ 2. while there are more tokens
- ➔ 3. get the next token
- ➔ 4. if the first character of the token is a digit
- ➔ 5. push the token on the stack
- 6. else if the token is an operator
- 7. pop the right operand off the stack
- 8. pop the left operand off the stack
- 9. evaluate the operation
- 10. push the result onto the stack
- 11. pop the stack and return the result

Evaluating Postfix Expressions (cont.)

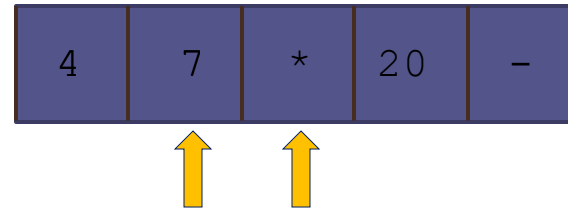


1. create an empty stack of integers
- 2. while there are more tokens
3. get the next token
- 4. if the first character of the token is a digit
- 5. push the token on the stack
6. else if the token is an operator
7. pop the right operand off the stack
8. pop the left operand off the stack
9. evaluate the operation
10. push the result onto the stack
11. pop the stack and return the result

Evaluating Postfix Expressions (cont.)

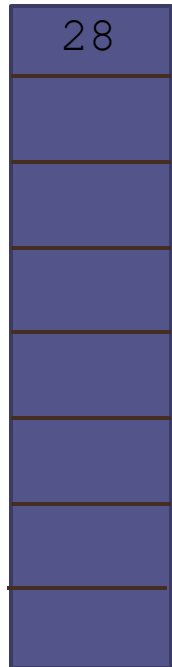


4 * 7

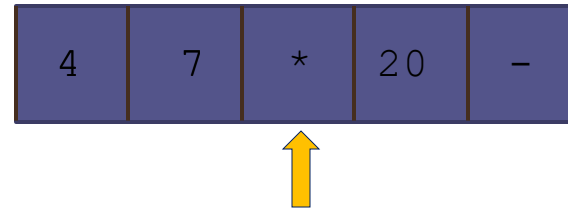


1. create an empty stack of integers
- 2. while there are more tokens
3. get the next token
- 4. if the first character of the token is a digit
5. push the token on the stack
- 6. else if the token is an operator
7. pop the right operand off the stack
8. pop the left operand off the stack
- 9. evaluate the operation
10. push the result onto the stack
11. pop the stack and return the result

Evaluating Postfix Expressions (cont.)

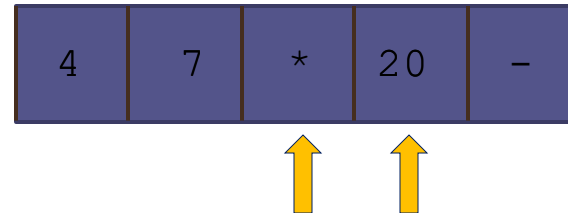
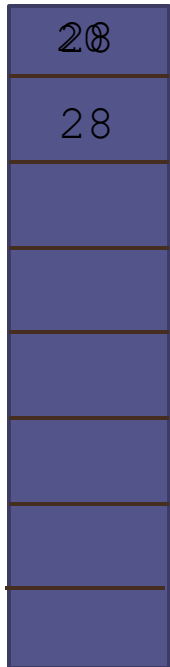


28



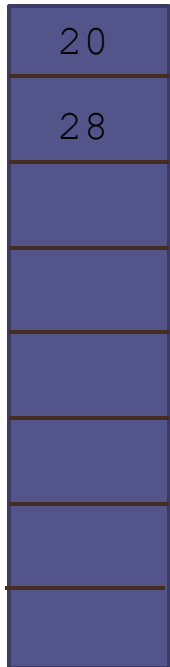
1. create an empty stack of integers
2. while there are more tokens
3. get the next token
4. if the first character of the token is a digit
5. push the token on the stack
6. else if the token is an operator
7. pop the right operand off the stack
8. pop the left operand off the stack
- 9. evaluate the operation
- 10. push the result onto the stack
11. pop the stack and return the result

Evaluating Postfix Expressions (cont.)

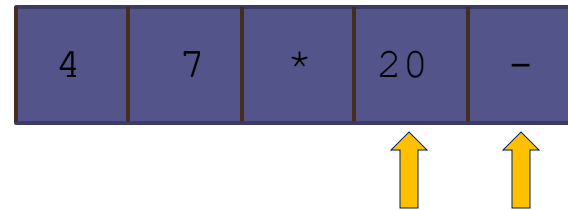


1. create an empty stack of integers
- 2. while there are more tokens
3. get the next token
- 4. if the first character of the token is a digit
- 5. push the token on the stack
6. else if the token is an operator
7. pop the right operand off the stack
8. pop the left operand off the stack
9. evaluate the operation
10. push the result onto the stack
11. pop the stack and return the result

Evaluating Postfix Expressions (cont.)

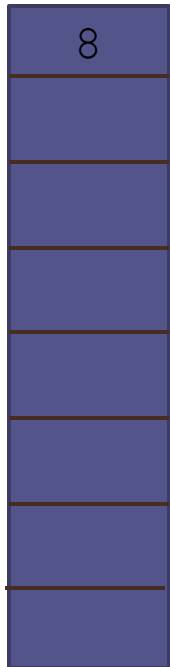


28 - 20

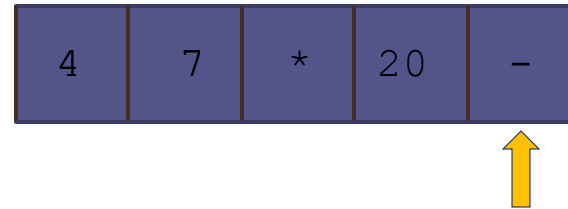


1. create an empty stack of integers
- 2. while there are more tokens
3. get the next token
- 4. if the first character of the token is a digit
5. push the token on the stack
- 6. else if the token is an operator
7. pop the right operand off the stack
8. pop the left operand off the stack
- 9. evaluate the operation
10. push the result onto the stack
11. pop the stack and return the result

Evaluating Postfix Expressions (cont.)

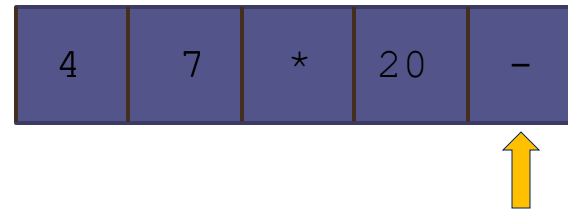
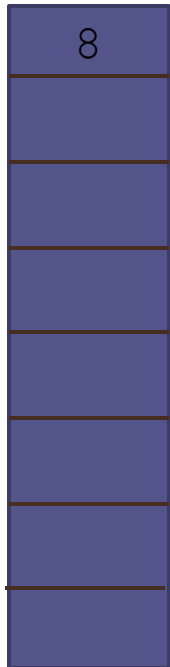


8



1. create an empty stack of integers
2. while there are more tokens
3. get the next token
4. if the first character of the token is a digit
5. push the token on the stack
6. else if the token is an operator
7. pop the right operand off the stack
8. pop the left operand off the stack
- 9. evaluate the operation
- 10. push the result onto the stack
11. pop the stack and return the result

Evaluating Postfix Expressions (cont.)



1. create an empty stack of integers
- 2. while there are more tokens
3. get the next token
4. if the first character of the token is a digit
5. push the token on the stack
6. else if the token is an operator
7. pop the right operand off the stack
8. pop the left operand off the stack
9. evaluate the operation
10. push the result onto the stack
- 11. pop the stack and return the result

Evaluating Postfix Expressions (cont.)

53

- Listing 4.6 (`PostfixEvaluator.java`, pages ?)

Evaluating Postfix Expressions (cont.)

- Testing: write a driver which
 - ▣ creates a `PostfixEvaluator` object
 - ▣ reads one or more expressions and report the result
 - ▣ catches `PostfixEvaluator.SyntaxErrorException`
 - ▣ exercises each path by using each operator
 - ▣ exercises each path through the method by trying different orderings and multiple occurrences of operators
 - ▣ tests for syntax errors:
 - an operator without any operands
 - a single operand
 - an extra operand
 - an extra operator
 - a variable name
 - the empty string

Converting from Infix to Postfix

- Convert infix expressions to postfix expressions
- Assume:
 - ▣ expressions consists of only spaces, operands, and operators
 - ▣ space is a delimiter character
 - ▣ all operands that are identifiers begin with a letter or underscore
 - ▣ all operands that are numbers begin with a digit

Data Field	Attribute
private Stack<Character> operatorStack	Stack of operators.
private StringBuilder postfix	The postfix string being formed.
Method	Behavior
public String convert(String infix)	Extracts and processes each token in infix and returns the equivalent postfix string.
private void processOperator(char op)	Processes operator op by updating operatorStack.
private int precedence(char op)	Returns the precedence of operator op.
private boolean isOperator(char ch)	Returns true if ch is an operator symbol.

Converting from Infix to Postfix

(cont.)

56

- Example: convert







$w - 5.1 / \text{sum} * 2$

to its postfix form

$w \ 5.1 \ \text{sum} \ / \ 2 \ * \ -$

Converting from Infix to Postfix

(cont.)

Next Token	Action	Effect on operatorStack	Effect on postfix
w	Append w to postfix.		w
-	The stack is empty Push - onto the stack		w
5.1	Append 5.1 to postfix		w 5.1
/	precedence(/) > precedence(-), Push / onto the stack		w 5.1
sum	Append sum to postfix		w 5.1 sum
*	precedence(*) equals precedence(/) Pop / off of stack and append to postfix		w 5.1 sum /

Converting from Infix to Postfix

(cont.)

Next Token	Action	Effect on operatorStack	Effect on postfix
*	precedence(*) > precedence(-), Push * onto the stack	<div>⌈ * - ⌋</div>	w 5.1 sum /
2	Append 2 to postfix	<div>⌈ * - ⌋</div>	w 5.1 sum / 2
End of input	Stack is not empty, Pop * off the stack and append to postfix	<div>⌈ - ⌋</div>	w 5.1 sum / 2 *
End of input	Stack is not empty, Pop - off the stack and append to postfix	<div>⌈ ⌋</div>	w 5.1 sum / 2 * -

Converting from Infix to Postfix

(cont.)

Algorithm for Method convert

1. Initialize postfix to an empty StringBuilder.
2. Initialize the operator stack to an empty stack.
3. **while** there are more tokens in the infix string
4. Get the next token.
5. **if** the next token is an operand
6. Append it to postfix.
7. **else if** the next token is an operator
8. Call processOperator to process the operator.
9. **else**
10. Indicate a syntax error.
11. Pop remaining operators off the operator stack and append them to postfix.

Converting from Infix to Postfix

(cont.)

Algorithm for Method processOperator

1. **if** the operator stack is empty
2. Push the current operator onto the stack.
- else**
3. Peek the operator stack and let topOp be the top operator.
4. **if** the precedence of the current operator is greater than the
 precedence of topOp
5. Push the current operator onto the stack.
- else**
6. **while** the stack is not empty and the precedence of the current
 operator is less than or equal to the precedence of topOp
7. Pop topOp off the stack and append it to postfix.
8. **if** the operator stack is not empty
9. Peek the operator stack and let topOp be the top
 operator.
10. Push the current operator onto the stack.

Converting from Infix to Postfix

(cont.)

61

- **Listing 4.7** (`InfixToPostfix.java`, pages ?)

Converting from Infix to Postfix

(cont.)

□ Testing

- Use enough test expressions to satisfy yourself that the conversions are correct for properly formed input expressions

- Use a driver to catch

`InfixToPostfix.SyntaxErrorException`

- **Listing 5.8** (`TestInfixToPostfix.java`, page ?)

Converting Expressions with Parentheses

- The ability to convert expressions with parentheses is an important (and necessary) addition
- Modify `processOperator` to push each opening parenthesis onto the stack as soon as it is scanned
- When a closing parenthesis is encountered, pop off operators until the opening parenthesis is encountered
- Listing 4.9 (`InfixToPostfixParens.java`, pages ?)

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 4.5

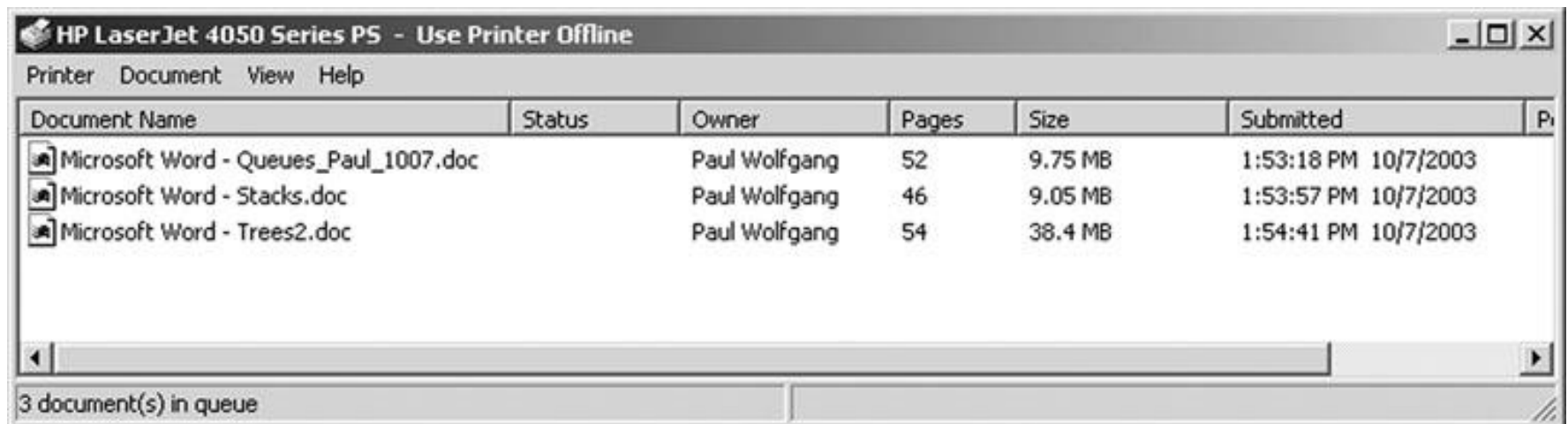
Queue Abstract Data Type

- 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

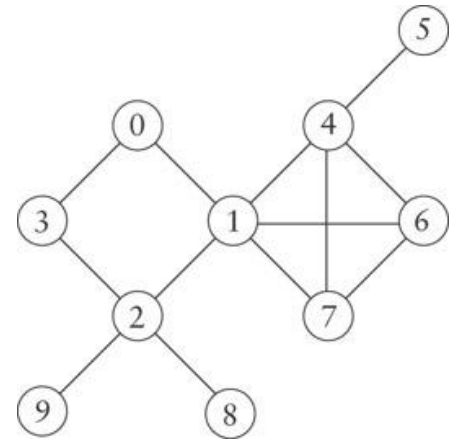


Unsuitability of a Print Stack

- ❑ 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



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 4.6

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 4.10(MaintainQueue, page ?)
- Listing 4.11 (method processCustomers in Class MaintainQueue, pages ?)

Implementing the Queue Interface

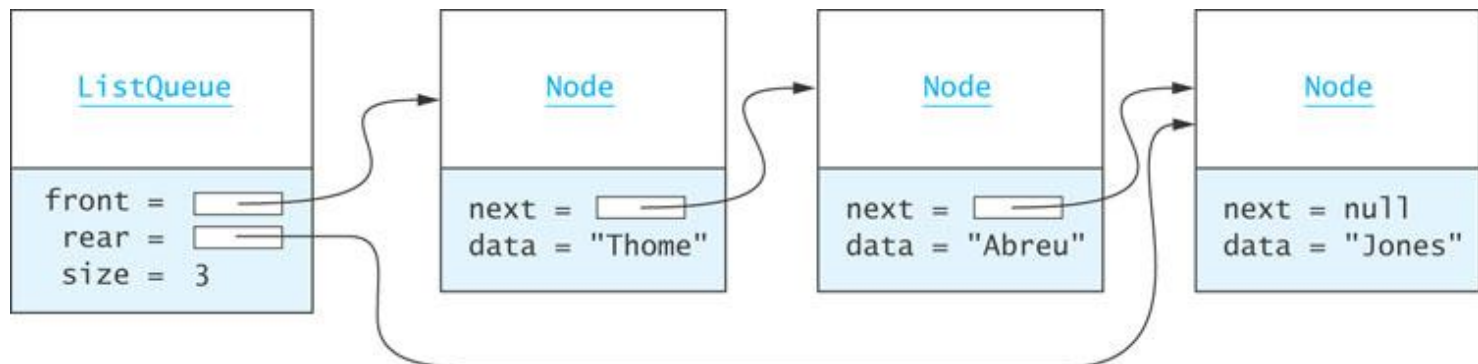
Section 4.7

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

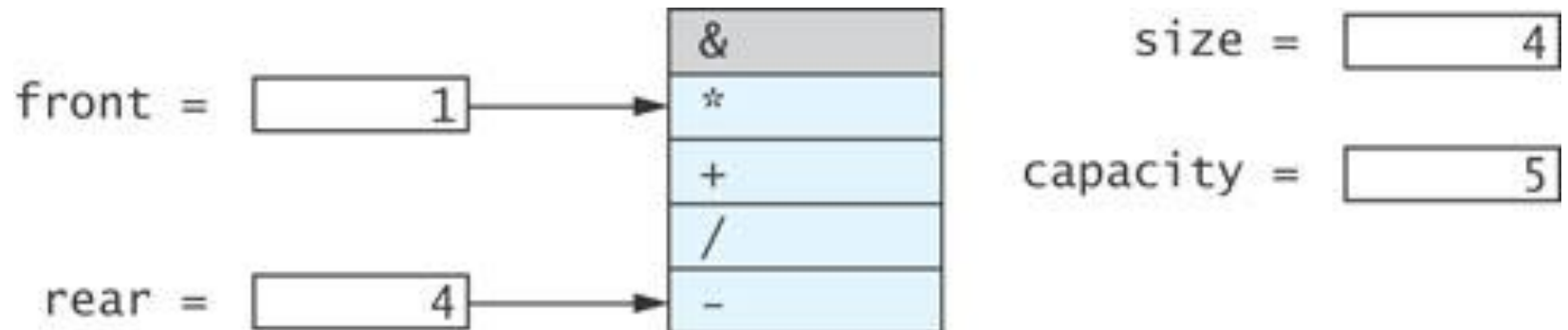
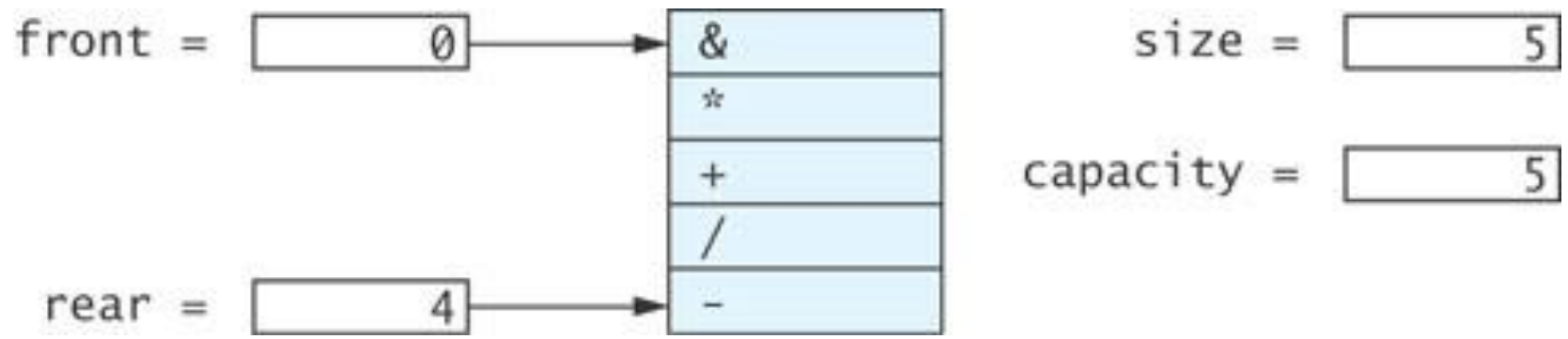


- ❑ Listing 4.3 (`ListQueue`, pages 208-209)

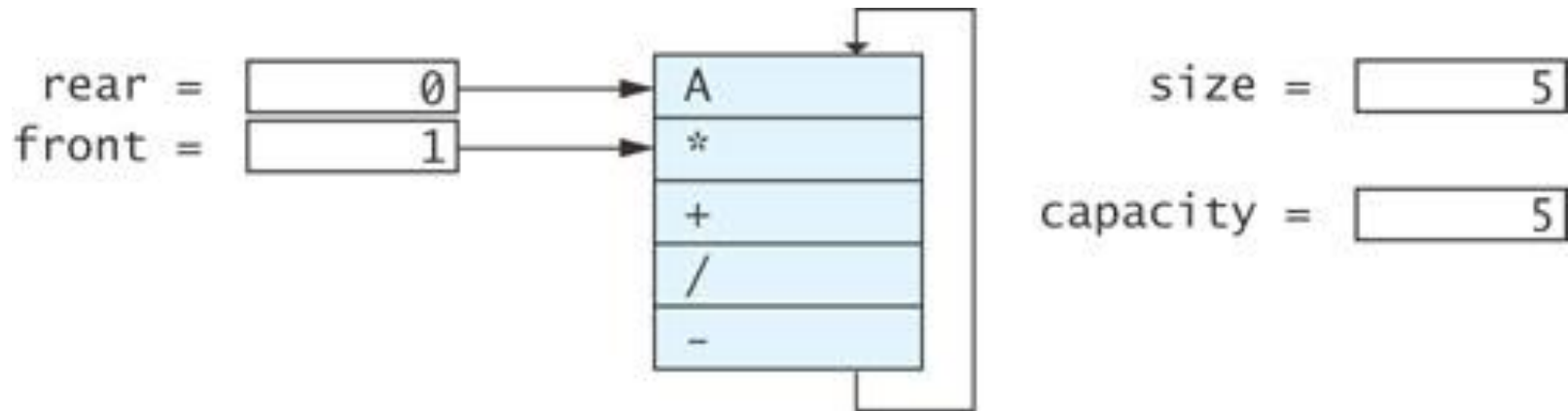
Implementing a Queue Using a Circular Array

- 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.)



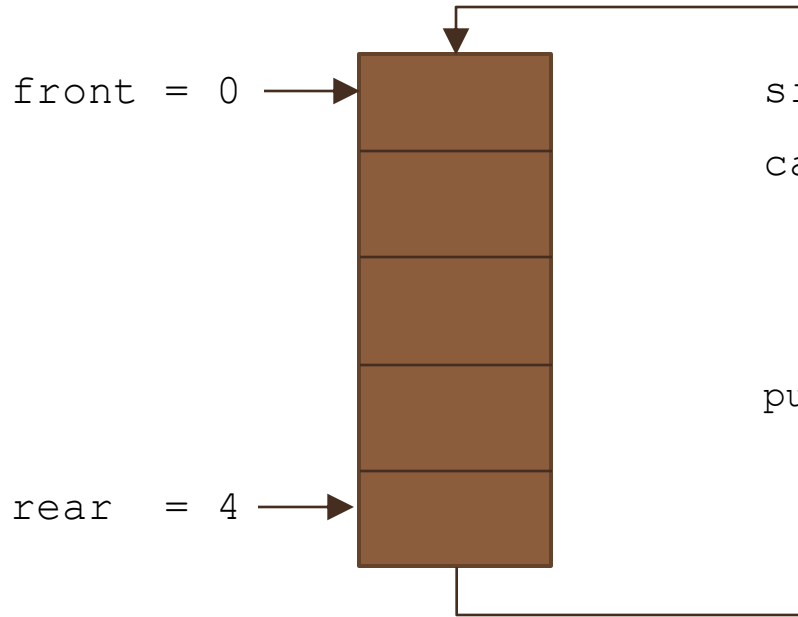
Implementing a Queue Using a Circular Array (cont.)

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

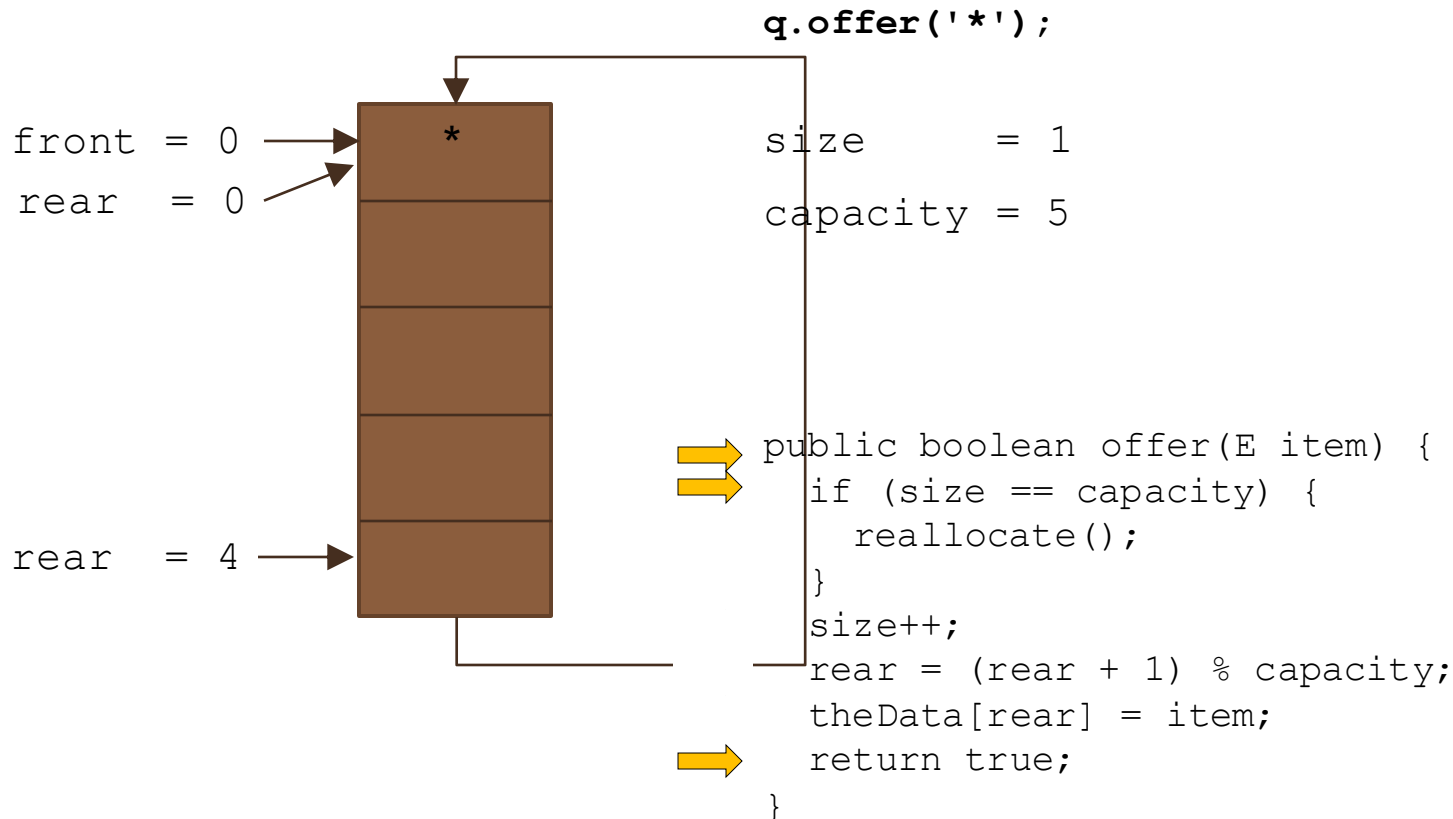
```
size      = 0
```

```
capacity = 5
```

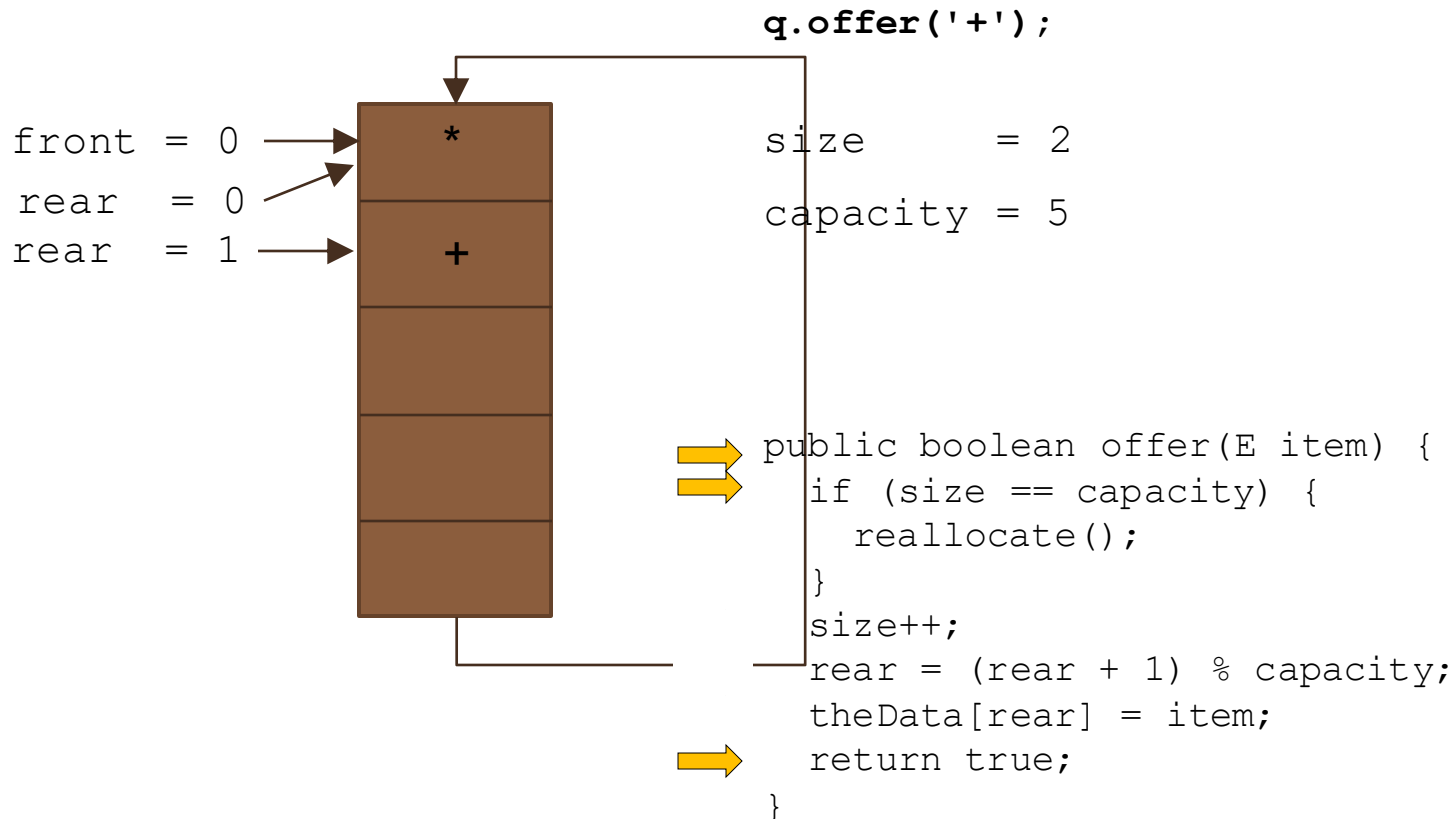
```
public ArrayQueue(int initCapacity) {  
    capacity = initCapacity;  
    theData = (E[])new Object[capacity];  
    front = 0;  
    rear = capacity - 1;  
    size = 0;  
}
```



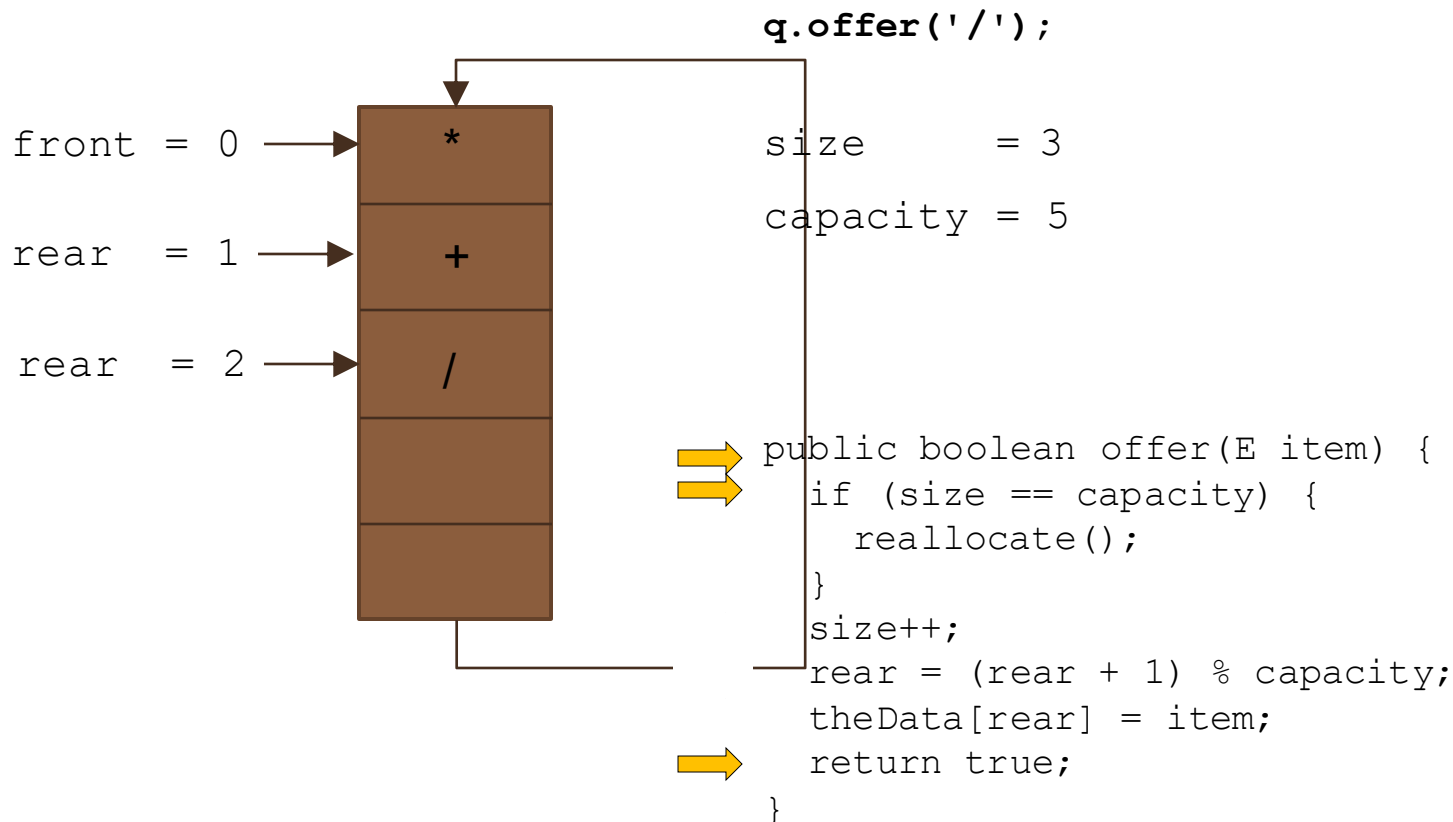
Implementing a Queue Using a Circular Array (cont.)



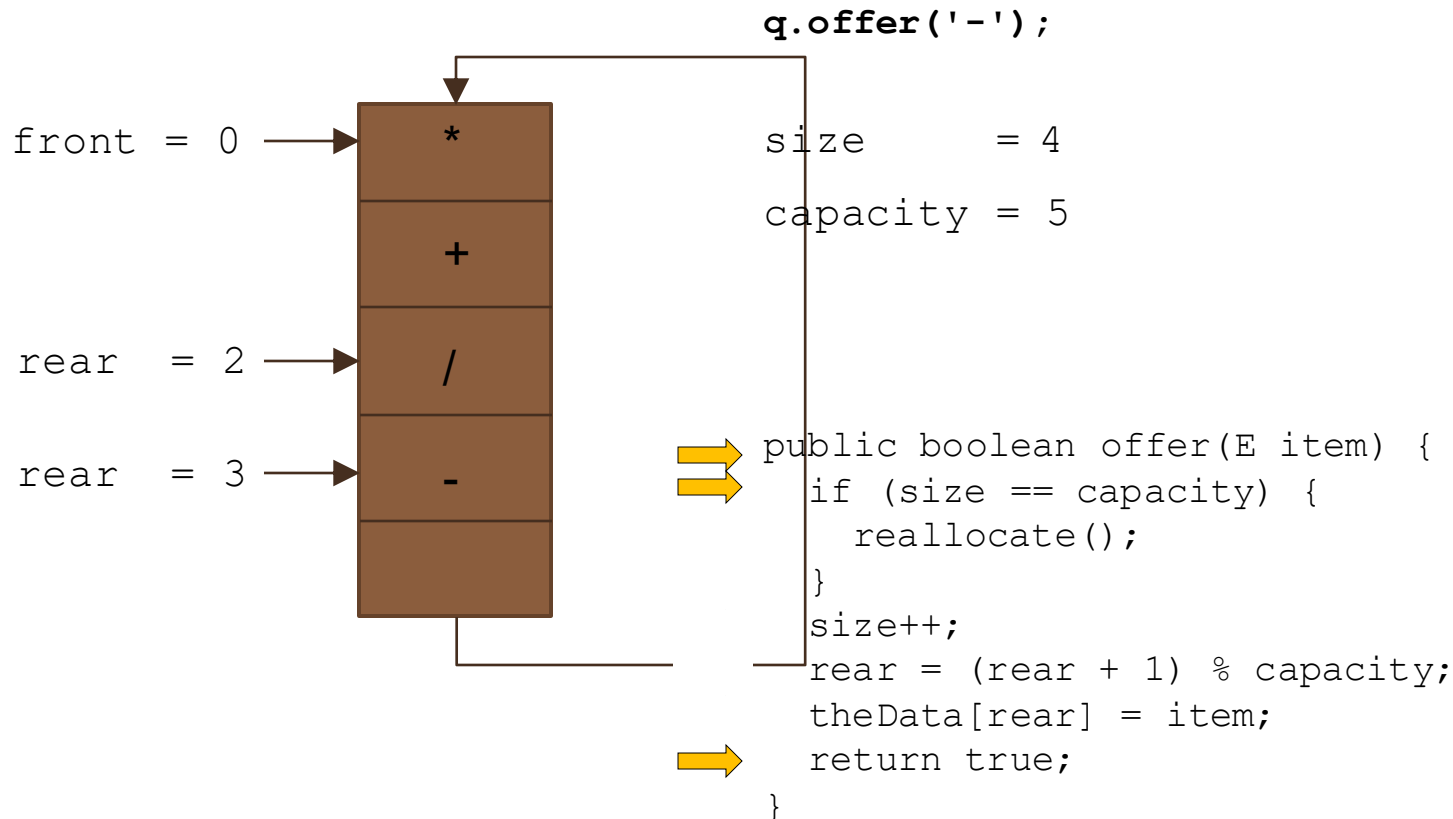
Implementing a Queue Using a Circular Array (cont.)



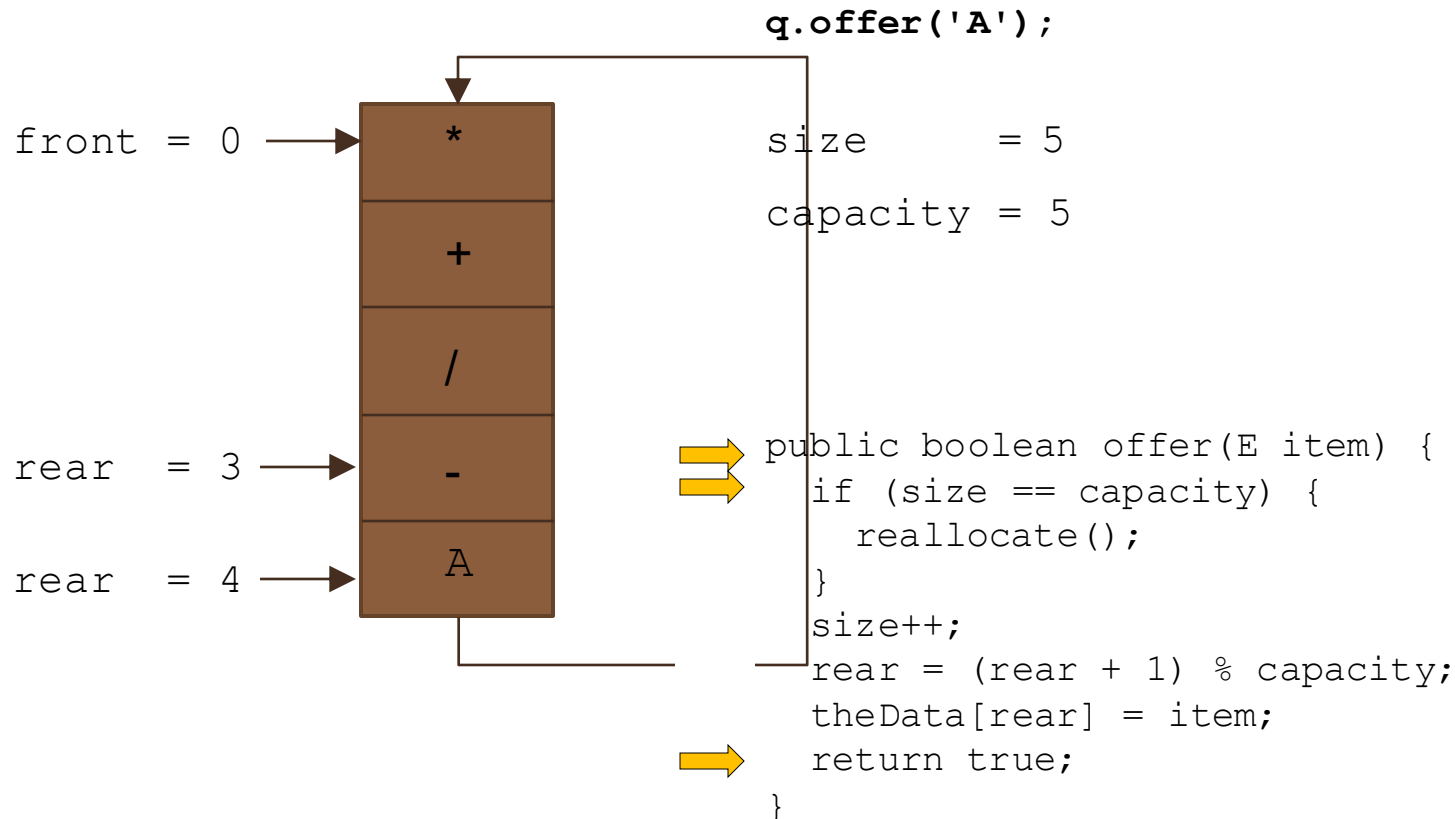
Implementing a Queue Using a Circular Array (cont.)



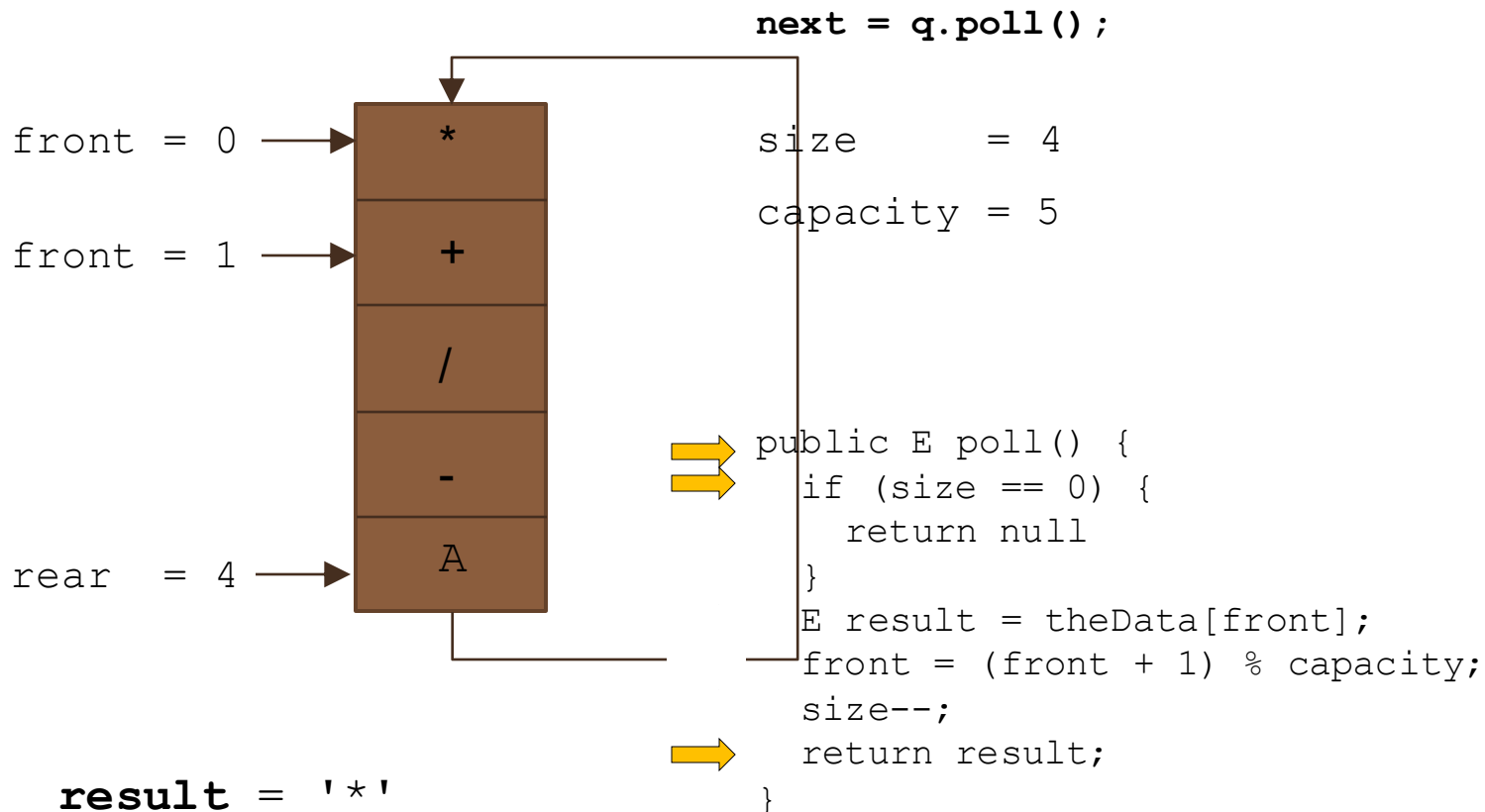
Implementing a Queue Using a Circular Array (cont.)



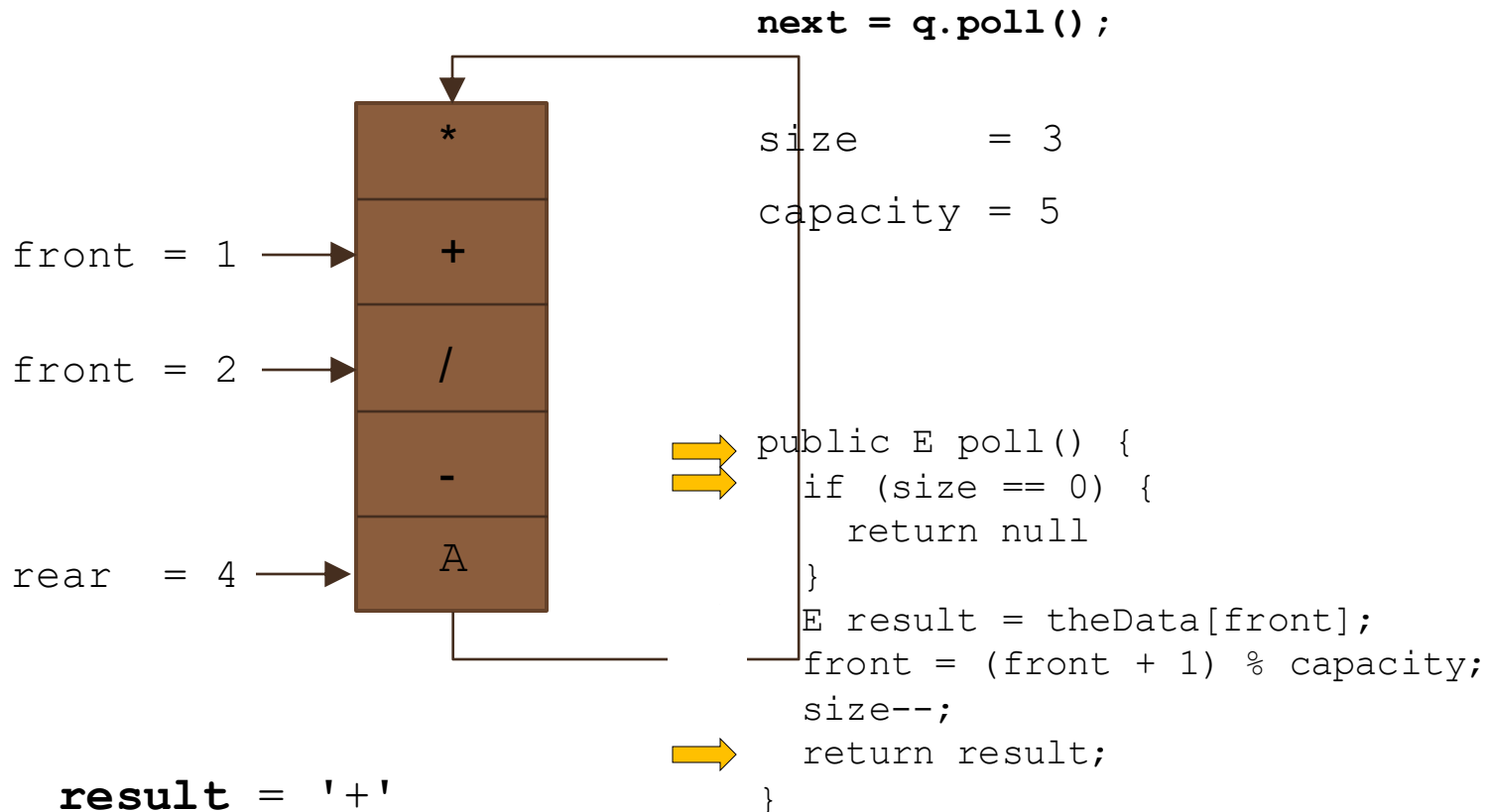
Implementing a Queue Using a Circular Array (cont.)



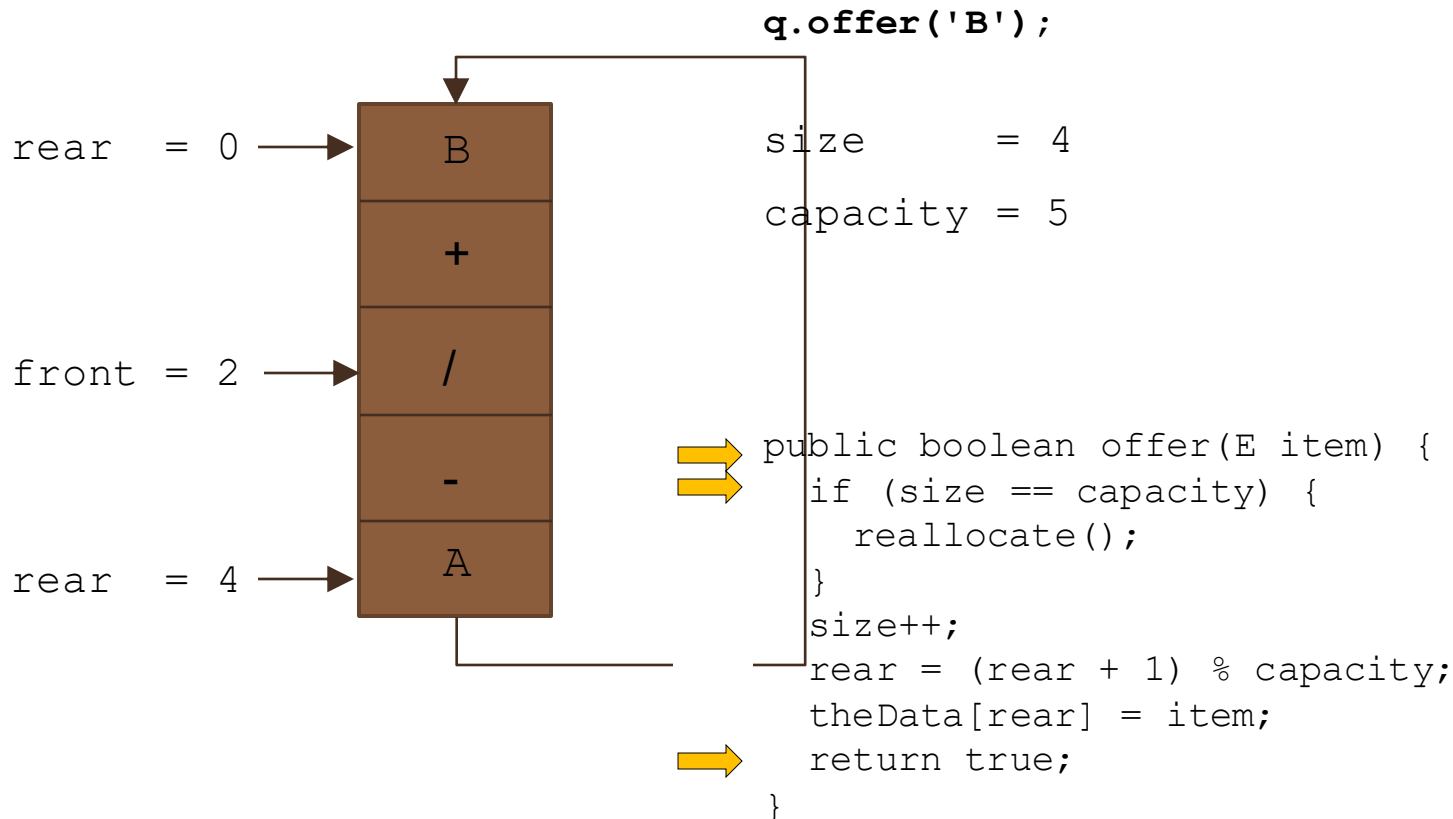
Implementing a Queue Using a Circular Array (cont.)



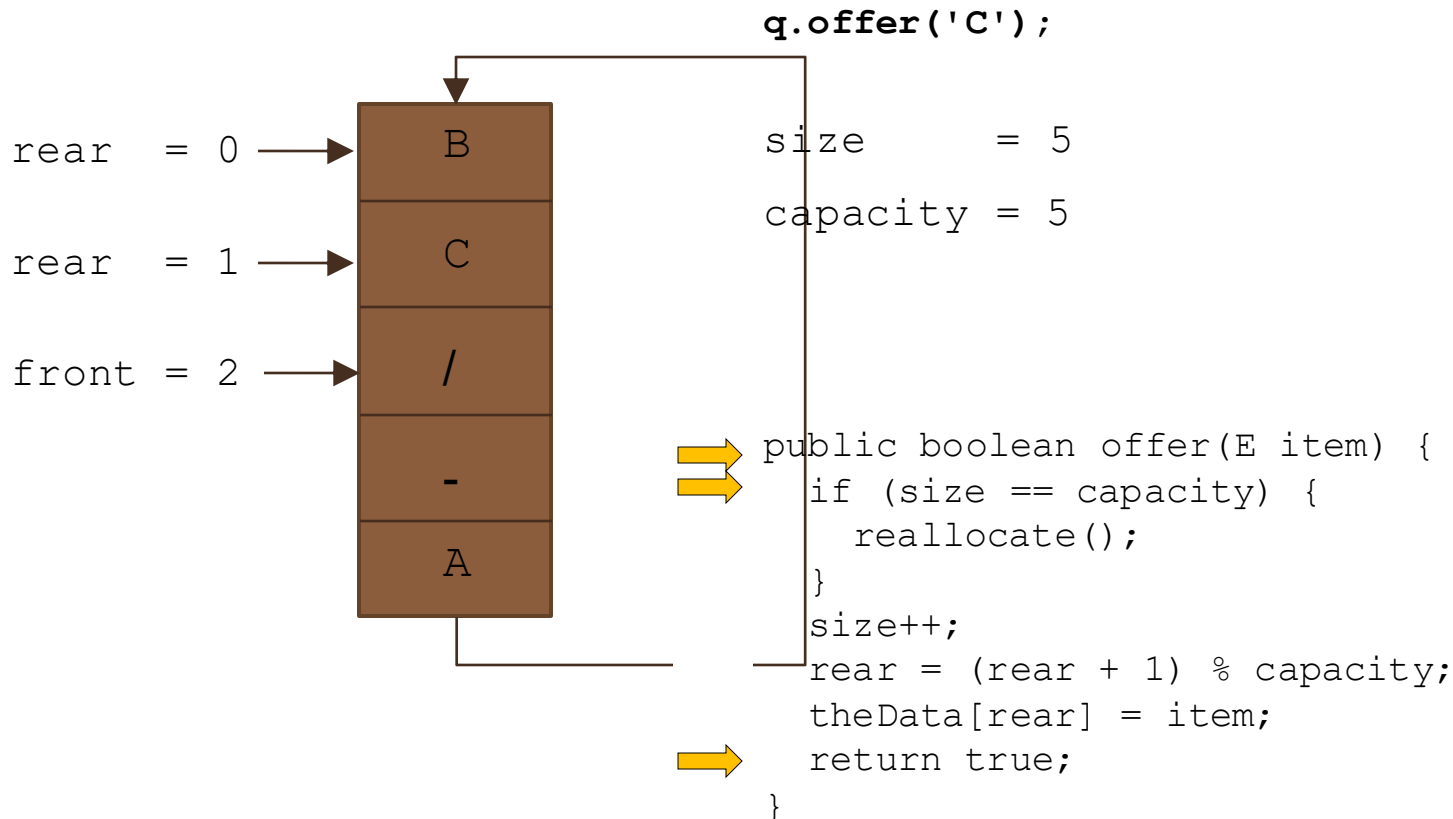
Implementing a Queue Using a Circular Array (cont.)



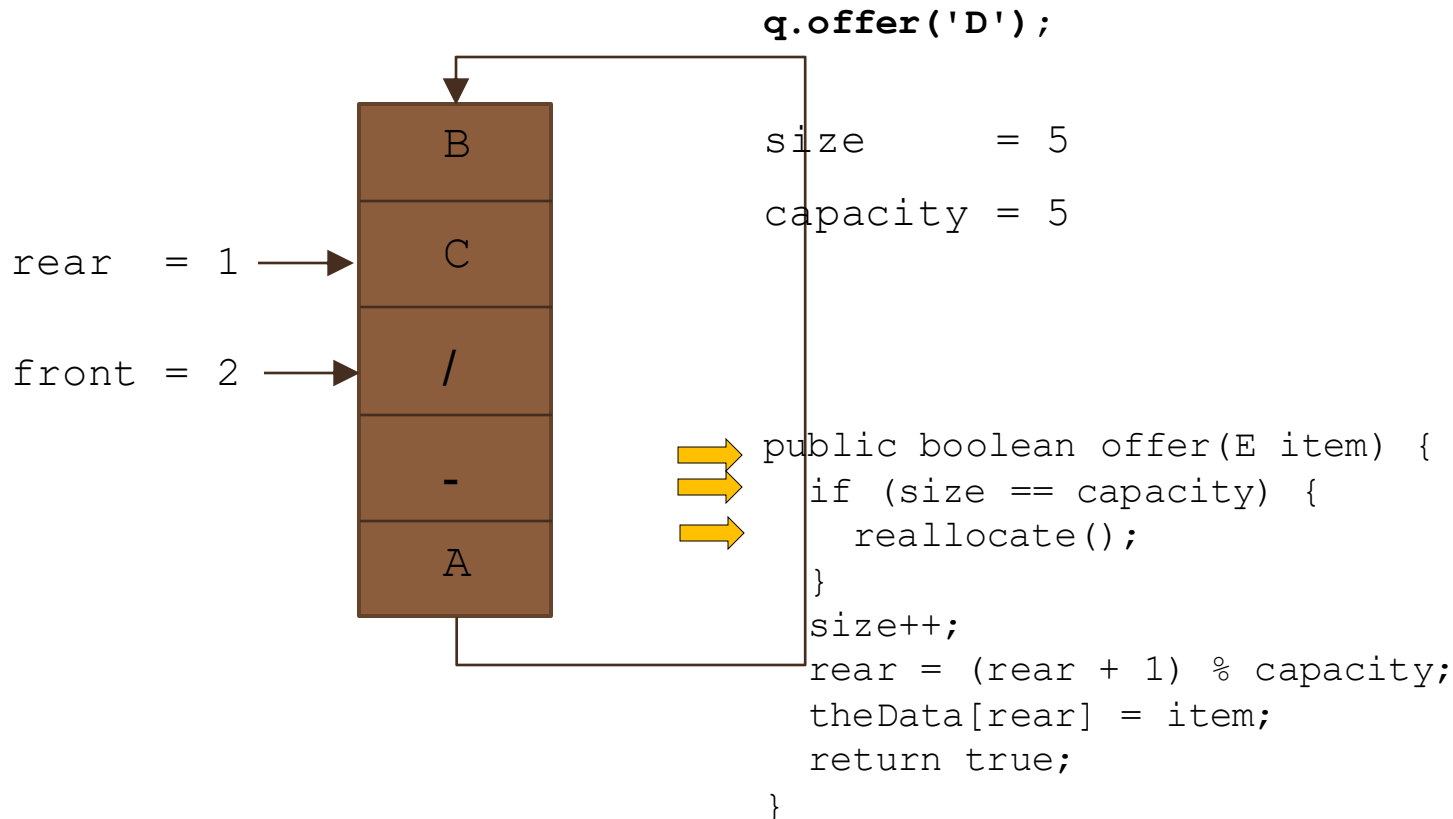
Implementing a Queue Using a Circular Array (cont.)



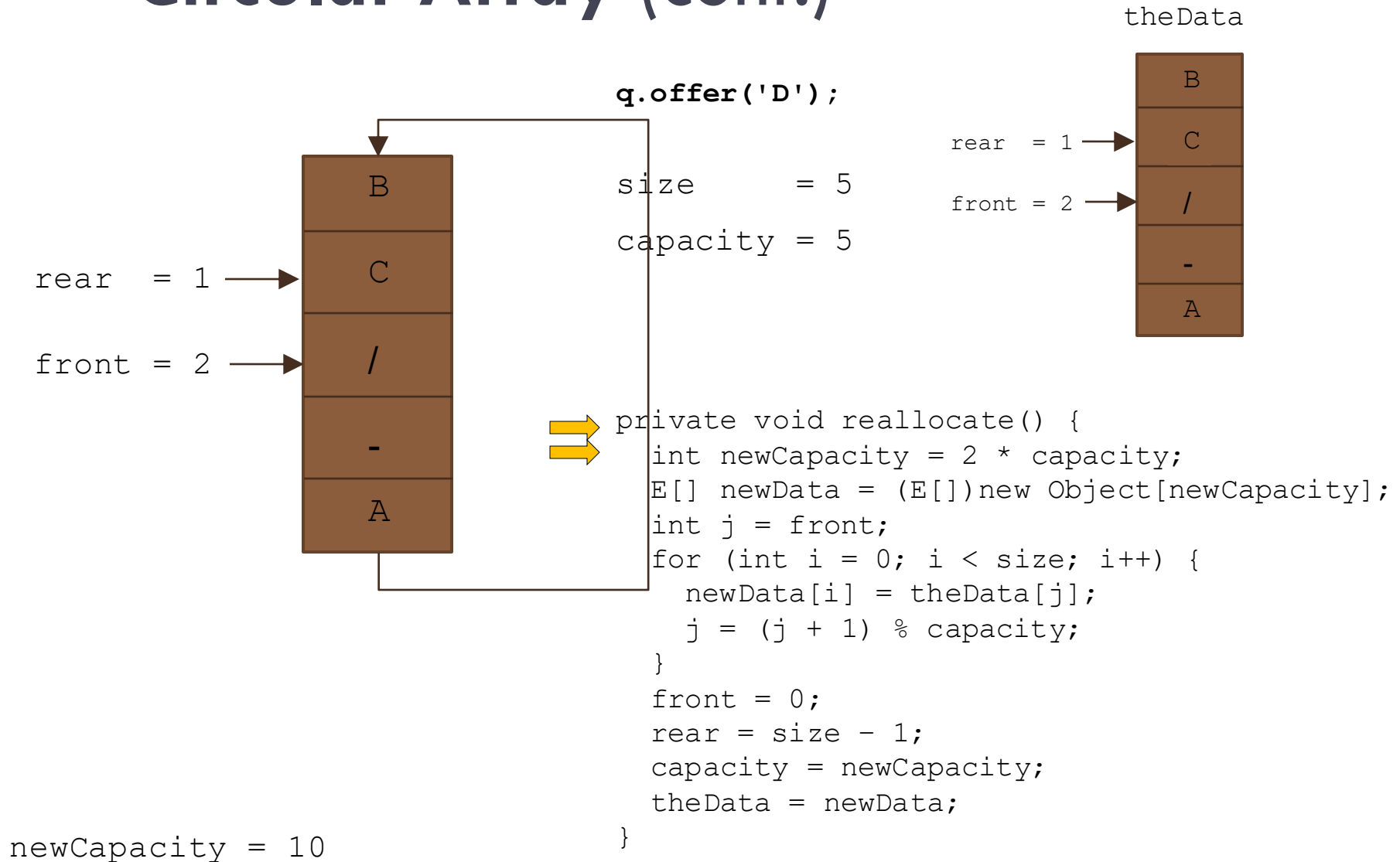
Implementing a Queue Using a Circular Array (cont.)



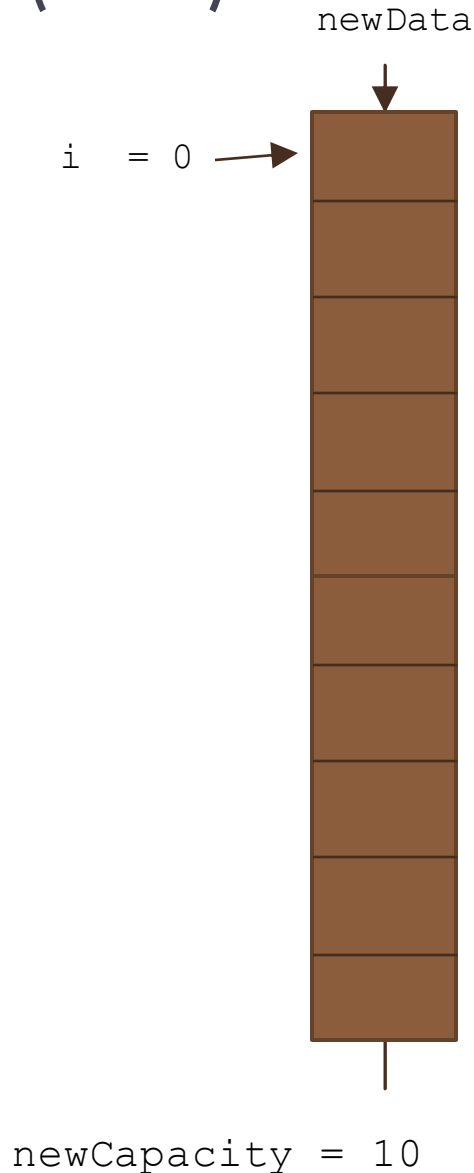
Implementing a Queue Using a Circular Array (cont.)



Implementing a Queue Using a Circular Array (cont.)



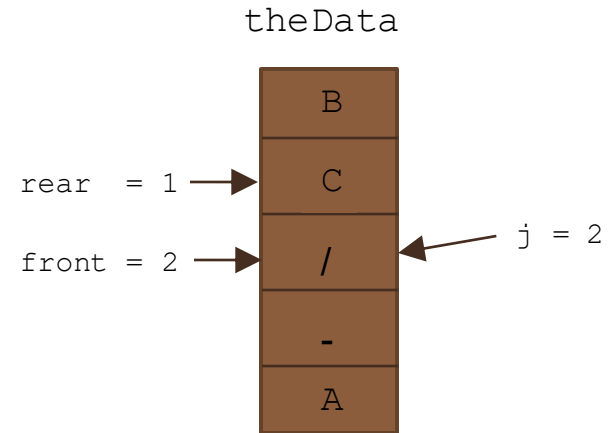
Implementing a Queue Using a Circular Array (cont.)



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

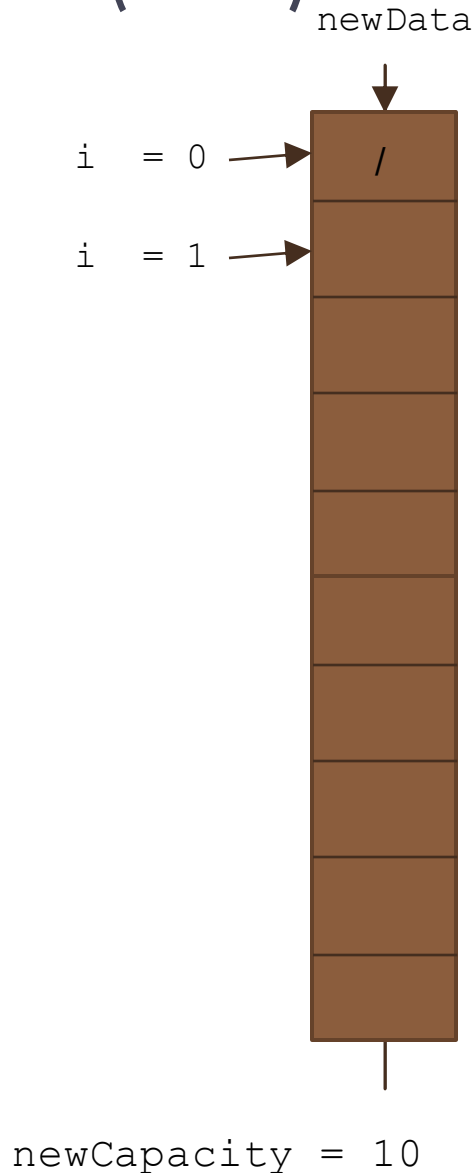
```
size      = 5
```

```
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;  
}
```

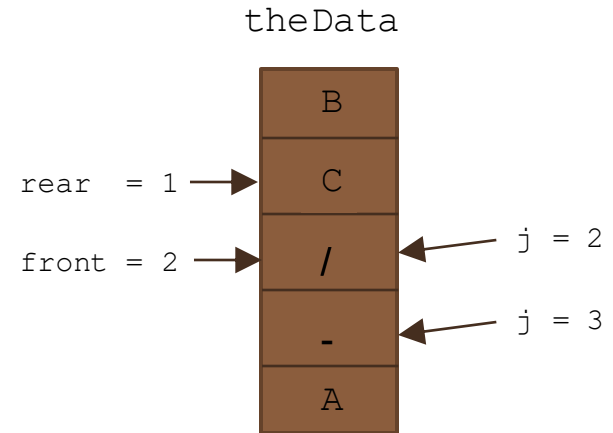
Implementing a Queue Using a Circular Array (cont.)



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

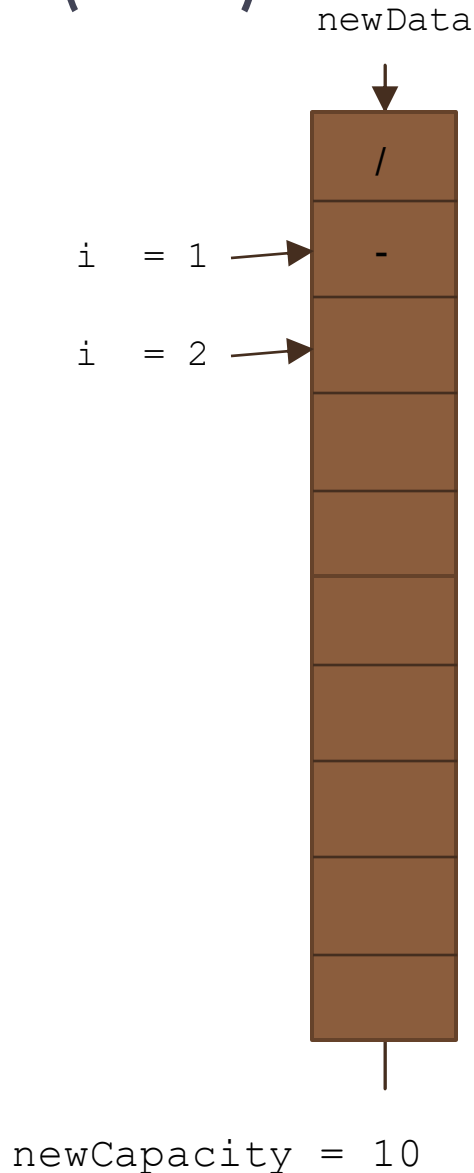
```
size      = 5
```

```
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;  
}
```

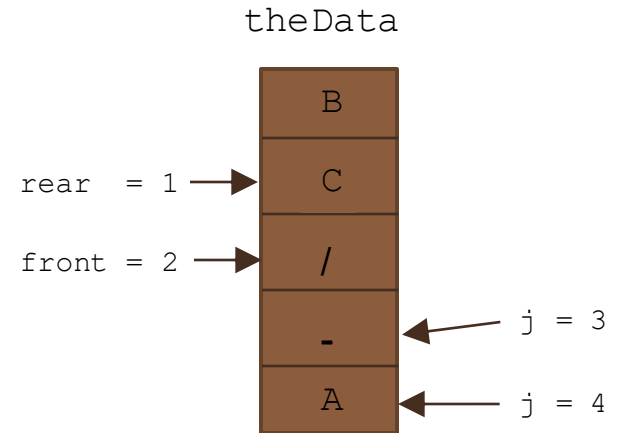

Implementing a Queue Using a Circular Array (cont.)



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

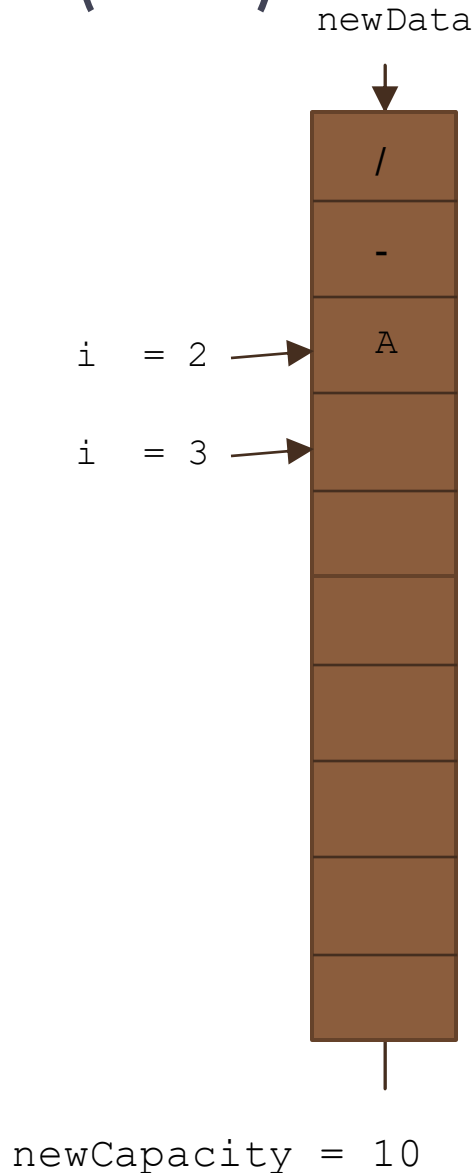
```
size      = 5
```

```
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;  
}
```

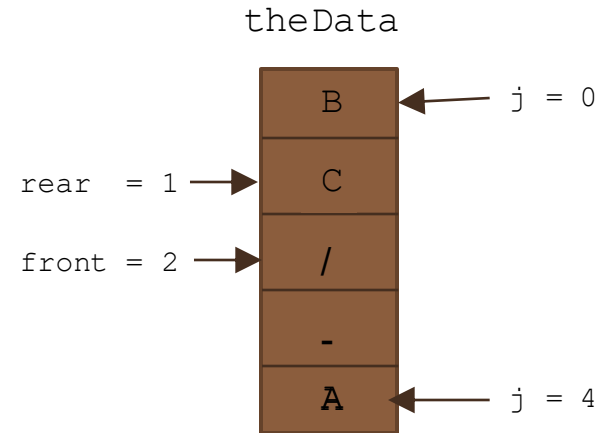
Implementing a Queue Using a Circular Array (cont.)



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

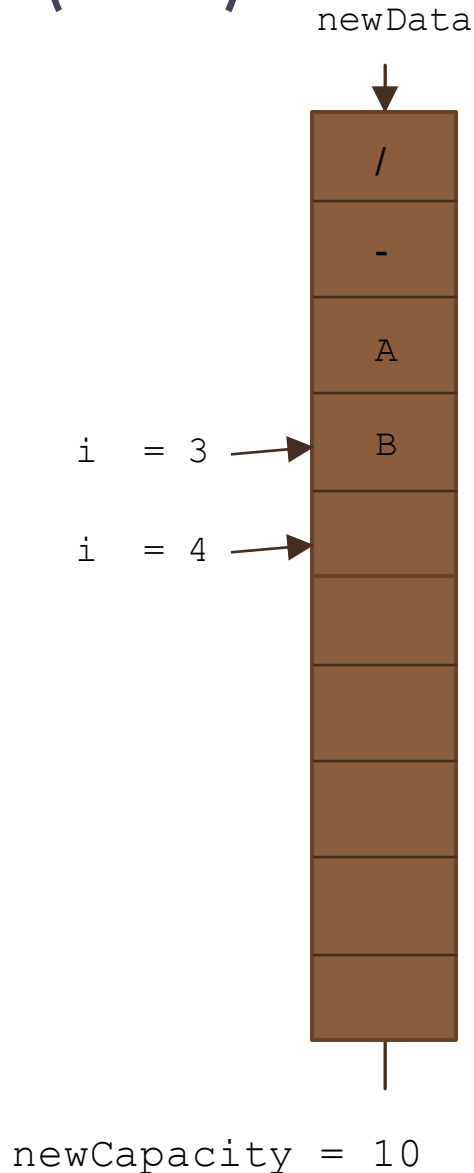
```
size      = 5
```

```
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;  
}
```

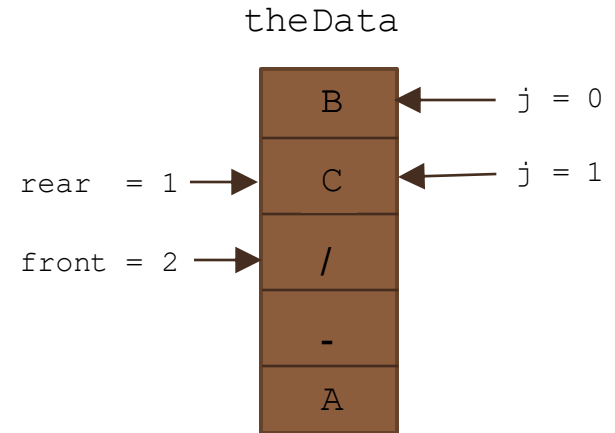
Implementing a Queue Using a Circular Array (cont.)



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

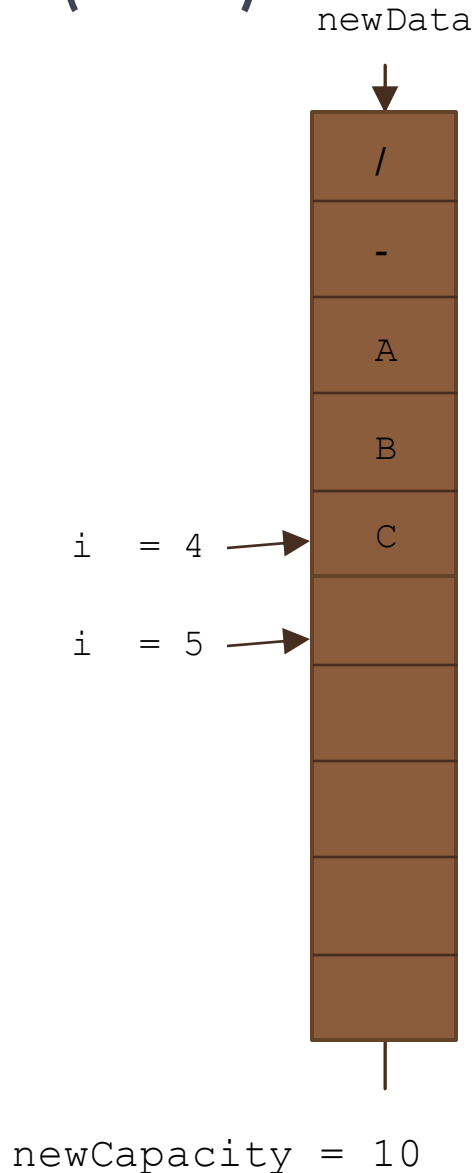
```
size      = 5
```

```
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;  
}
```

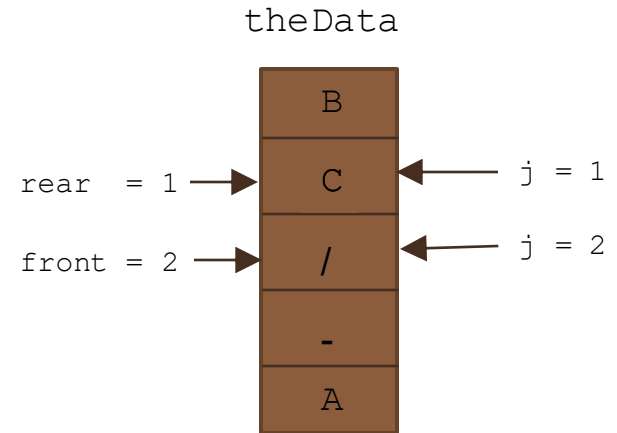
Implementing a Queue Using a Circular Array (cont.)



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

```
size      = 5
```

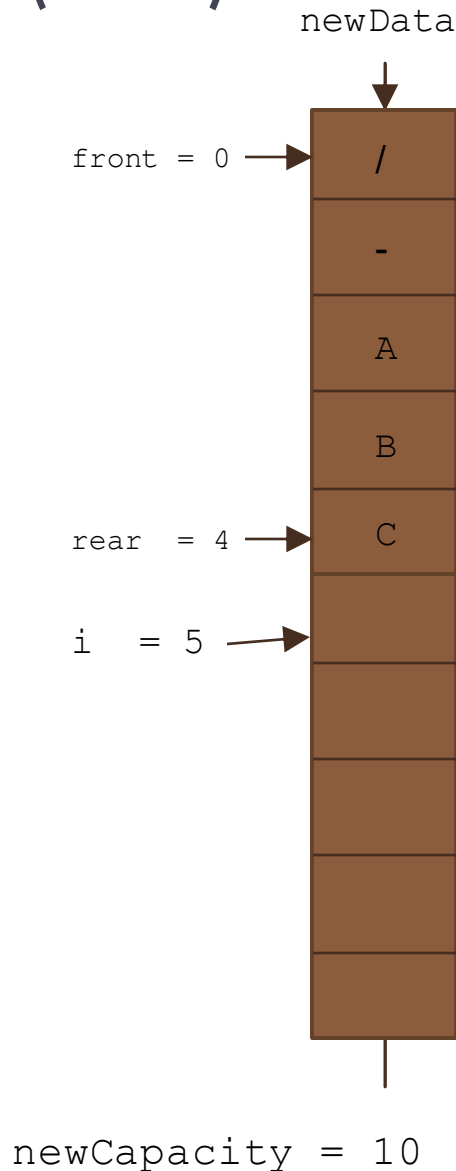
```
capacity = 5
```



```
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

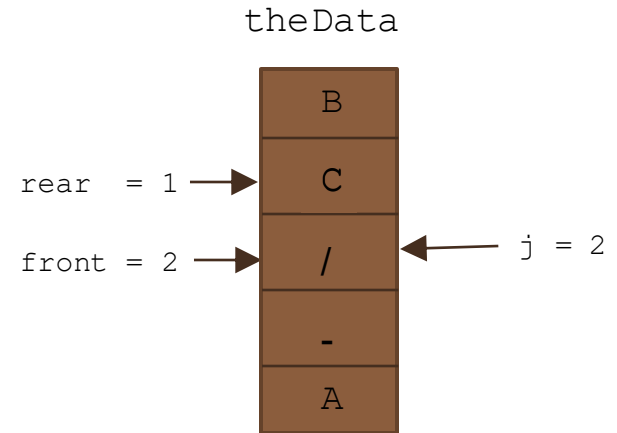
(cont.)



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

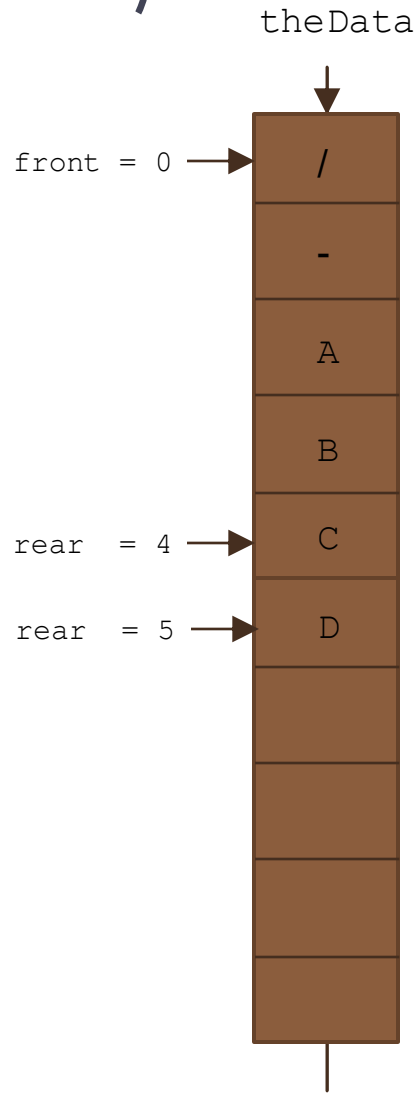
```
size      = 5
```

```
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.)

- Listing 4.12 (`ArrayQueue`, pages ?)

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;
    }
}
```

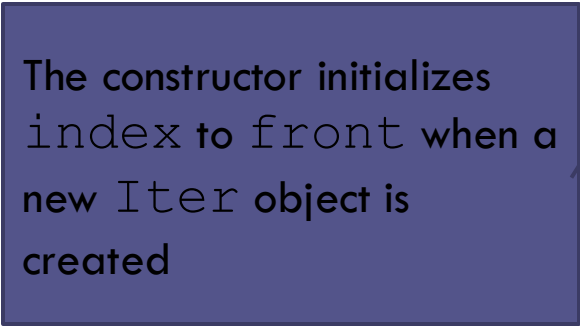
....

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;
}

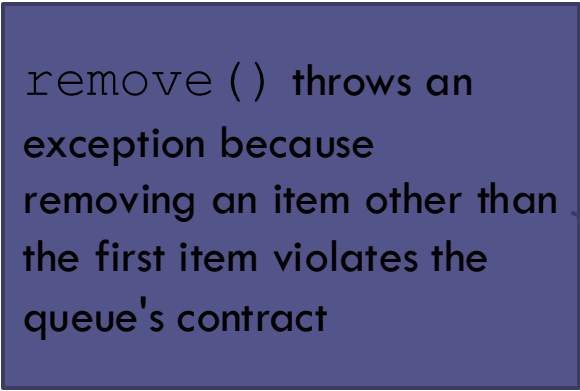
@Override
public void remove {
    throw new
        UnsupportedOperationException();
}
}
```

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 $O(n)$, it 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 4.8

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

Simulating Waiting Lines Using Queues

Section 4.9

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

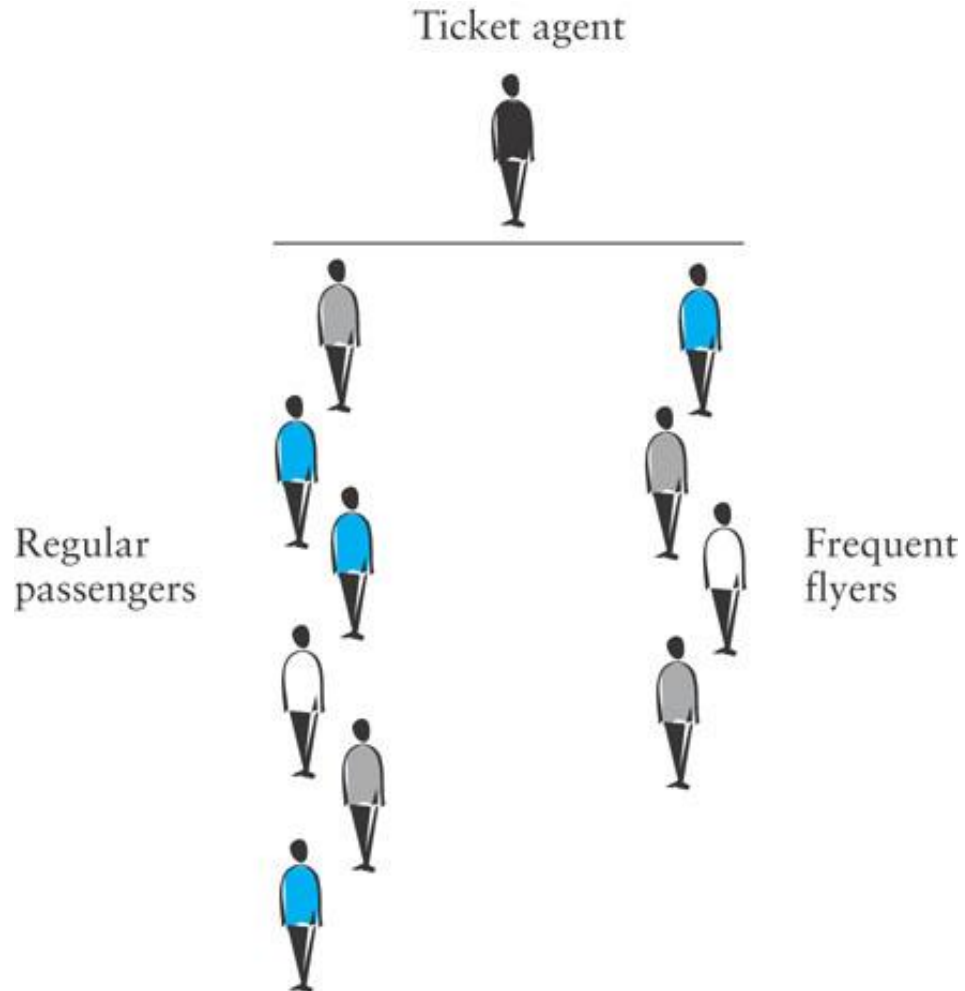
(cont.)

- 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.)



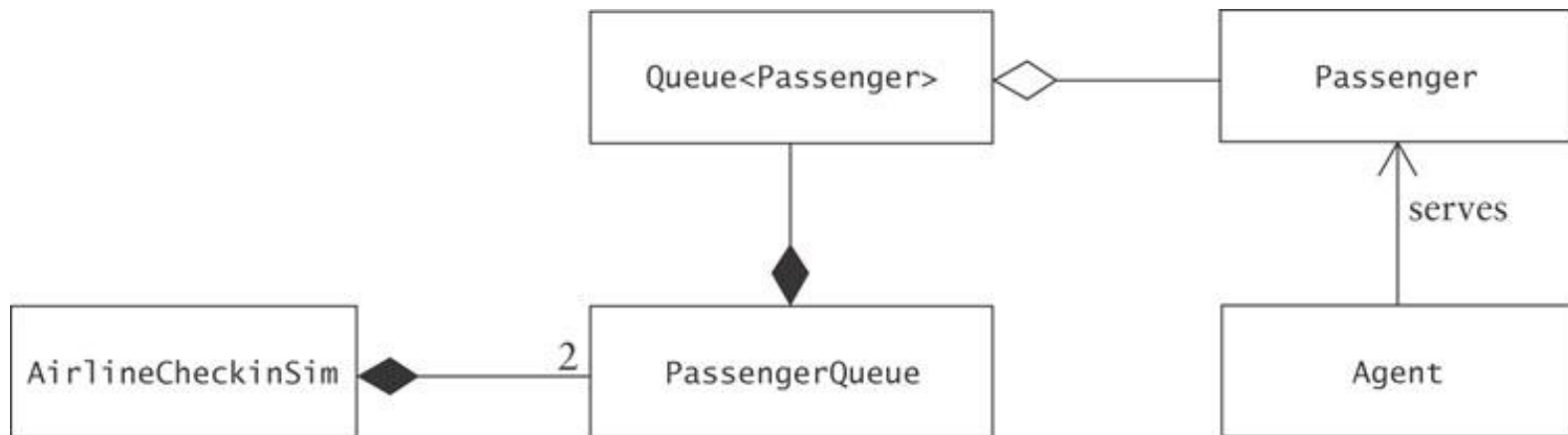
Case Study: Analysis

- 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

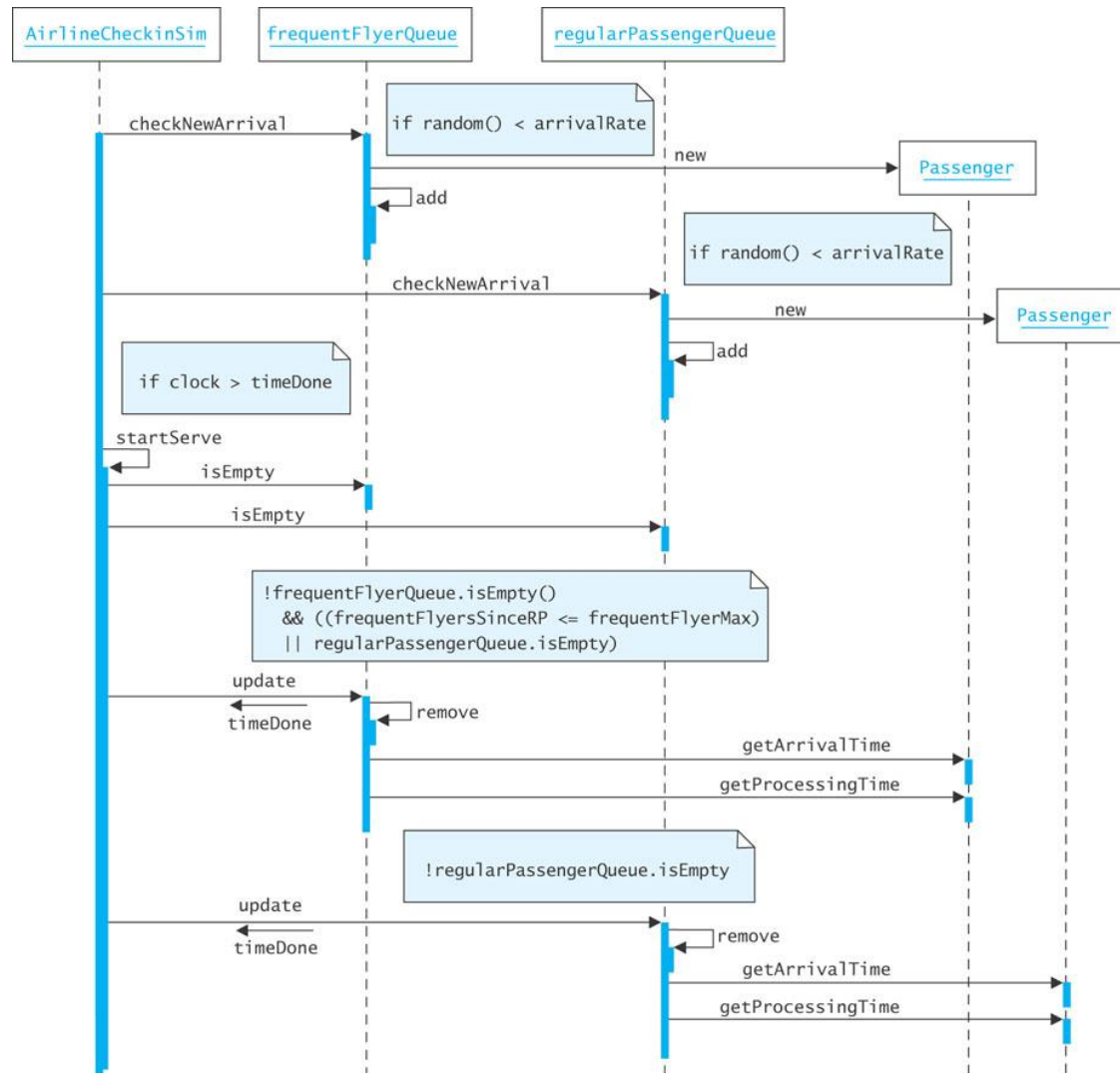
Case Study: Analysis (cont.)

- 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

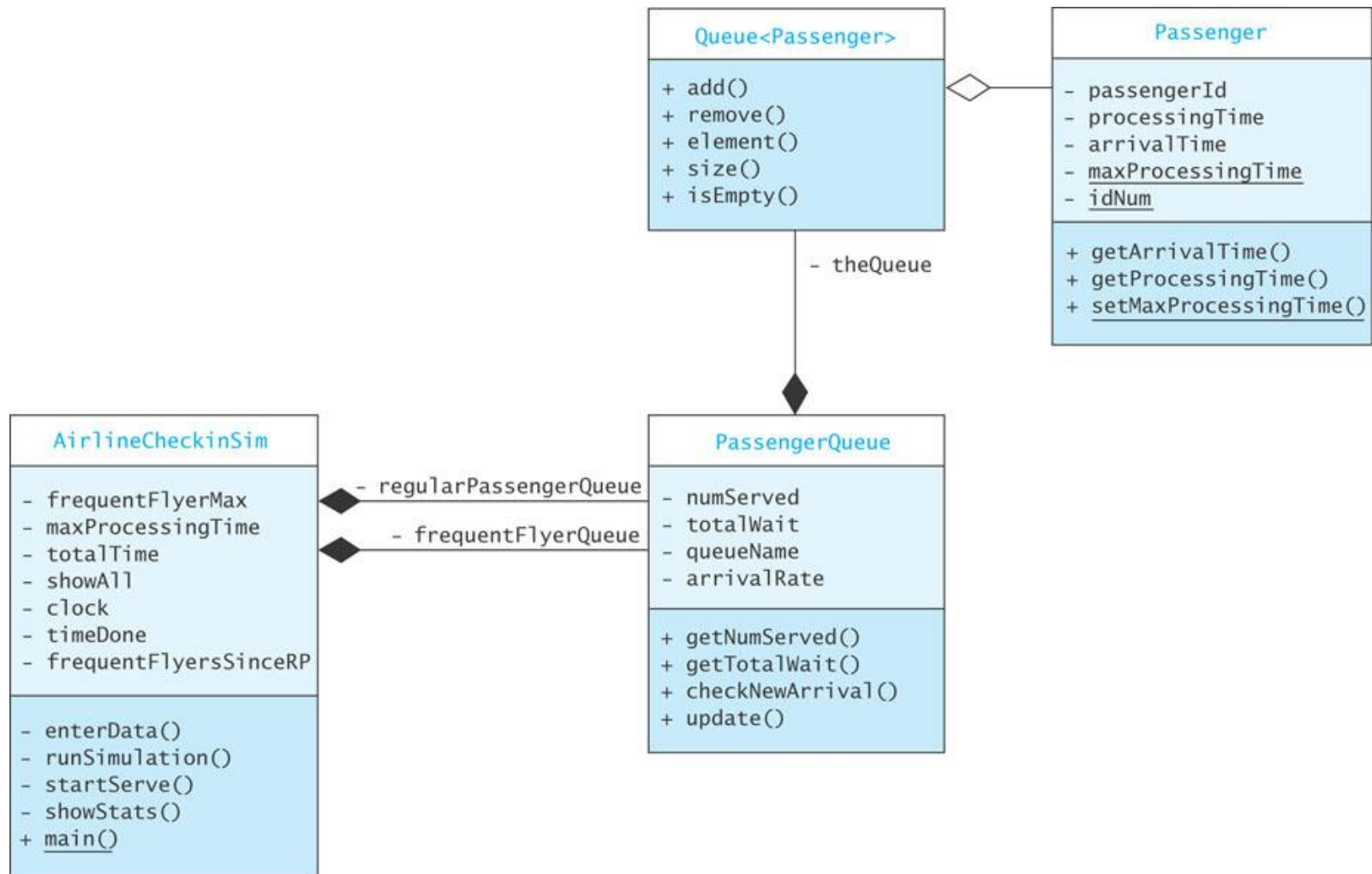
Case Study: Design (cont.)



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 <code>passenger</code> , 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 4.13 (`Passenger.java`, pages ?;
`PassengerQueue.java`, page ?,
`AirlineCheckinSim.java`, page ?)

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
```