Topic 2
Lecture 2b
Stack, Queue, and Deque

CSCI 240

Data Structures and Algorithms

Prof. Dominick Atanasio

Stacks

- FIGURE 5-1 Some familiar stacks

- Add item on top of stack
- Remove item that is topmost
  - Last In, First Out … LIFO

Specifications of the ADT Stack

Abstract Data Type: Stack

Data

- A collection of objects in reverse chronological order and having the same data type

Operations

Pseudocode	UML	Description
`push(newEntry)`	`+push(newEntry: T): void`	Task: Adds a new entry to the top of the stack. Input: `newEntry` is the new entry. Output: None.
`pop()`	`+pop(): T`	Task: Removes and returns the stack's top entry. Input: None. Output: Returns the stack's top entry. Throws an exception if the stack is empty before the operation.

Specifications of the ADT Stack

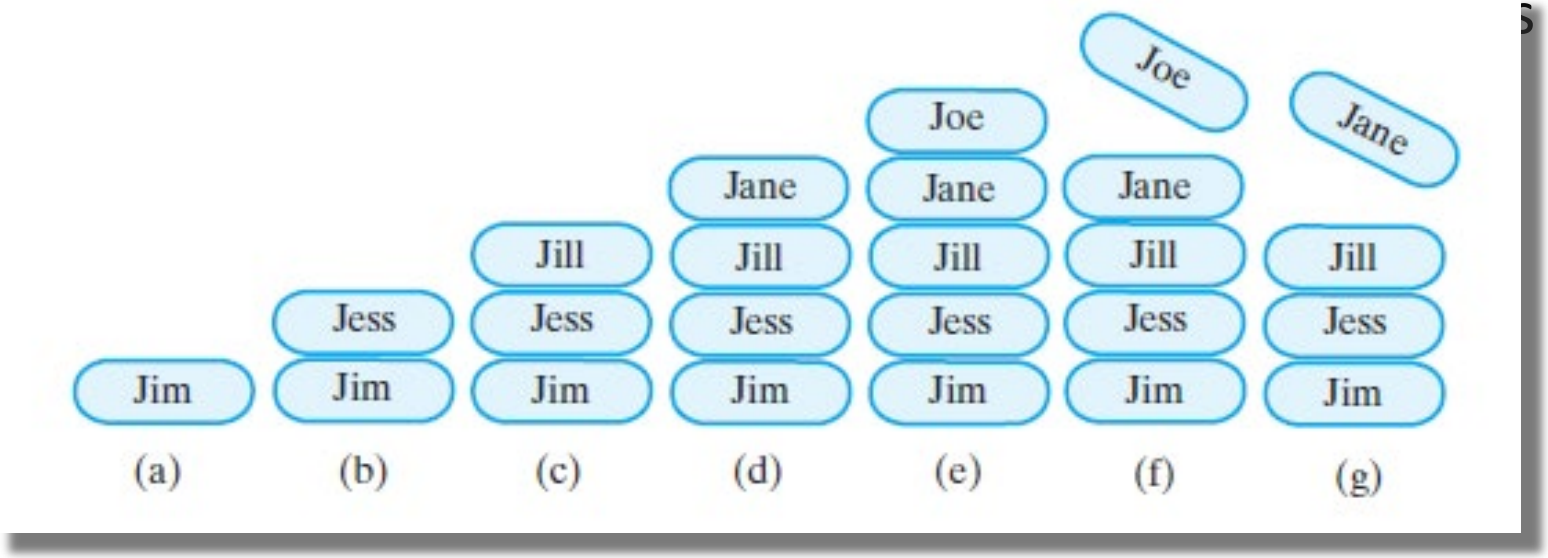
peek()	+peek(): T	Task: Retrieves the stack's top entry without changing the stack in any way. Input: None. Output: Returns the stack's top entry. Throws an exception if the stack is empty.
isEmpty()	+isEmpty(): boolean	Task: Detects whether the stack is empty. Input: None. Output: Returns true if the stack is empty.
clear()	+clear(): void	Task: Removes all entries from the stack. Input: None. Output: None.

Design Decision

- When stack is empty
  - What to do with pop and peek?
- Possible actions
  - Assume that the ADT is not empty;
  - Return null.
  - Throw an exception (which type?).

Example

- FIGURE 5-2 A stack of strings after (a) push adds Jim; (b) push adds Jess; (c) push adds Jill; (d) push … Joe; (g) pop retrieves and



Security Note

- Design guidelines
  - Use preconditions and postconditions to document assumptions.
  - Do not trust client to use public methods correctly.
  - Avoid ambiguous return values.
  - Prefer throwing exceptions instead of returning values to signal problem.

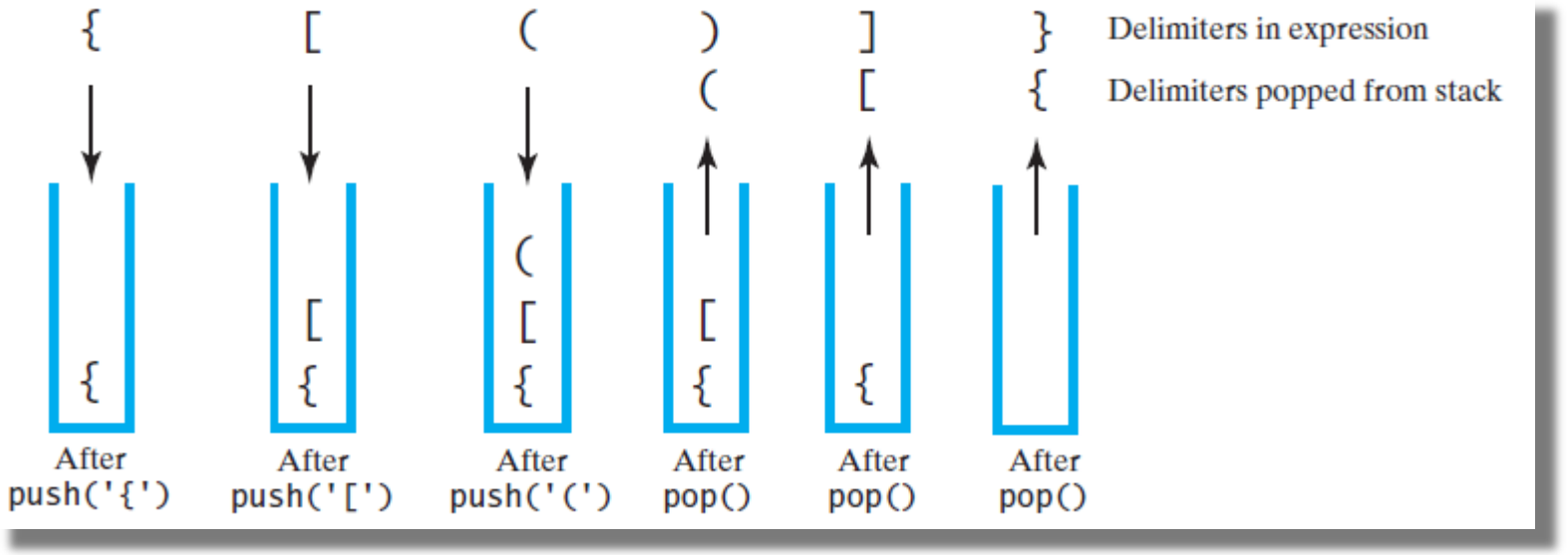
Usage of the Stack

Processing Algebraic Expressions

- Infix: each binary operator appears between its operands a + b
- Prefix: each binary operator appears before its operands + a b
- Postfix: each binary operator appears after its operands a b +
- Balanced expressions: delimiters paired correctly

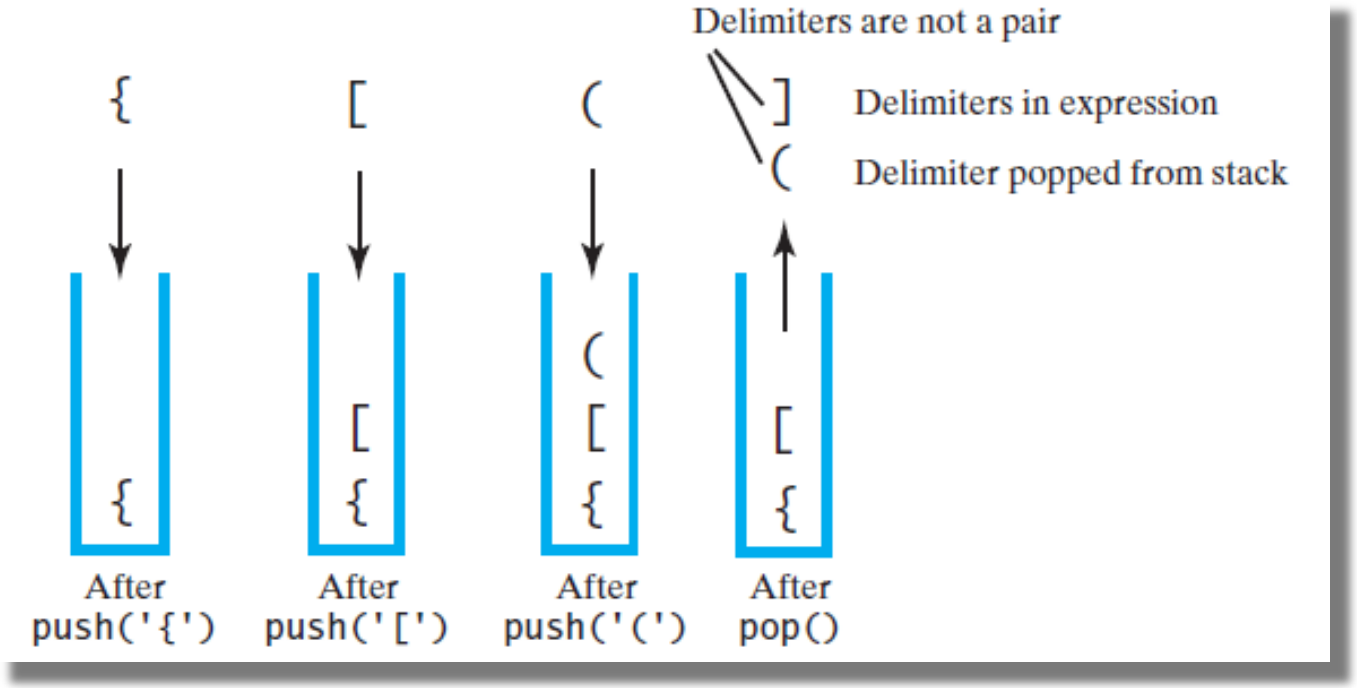
Processing Algebraic Expressions

- FIGURE 5-3 The contents of a stack during the scan of an expression that contains the balanced delimiters { [( )] }



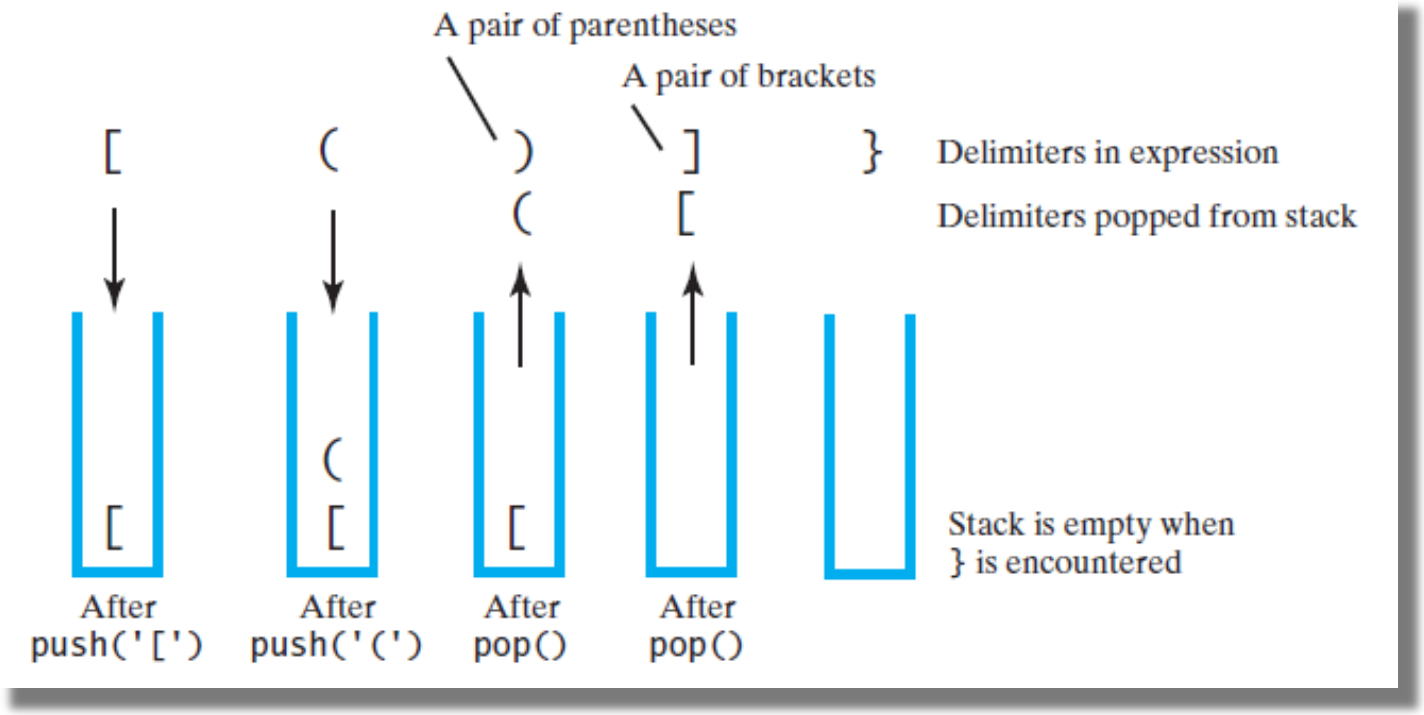
Processing Algebraic Expressions

- FIGURE 5-4 The contents of a stack during the scan of an expression that contains the unbalanced (



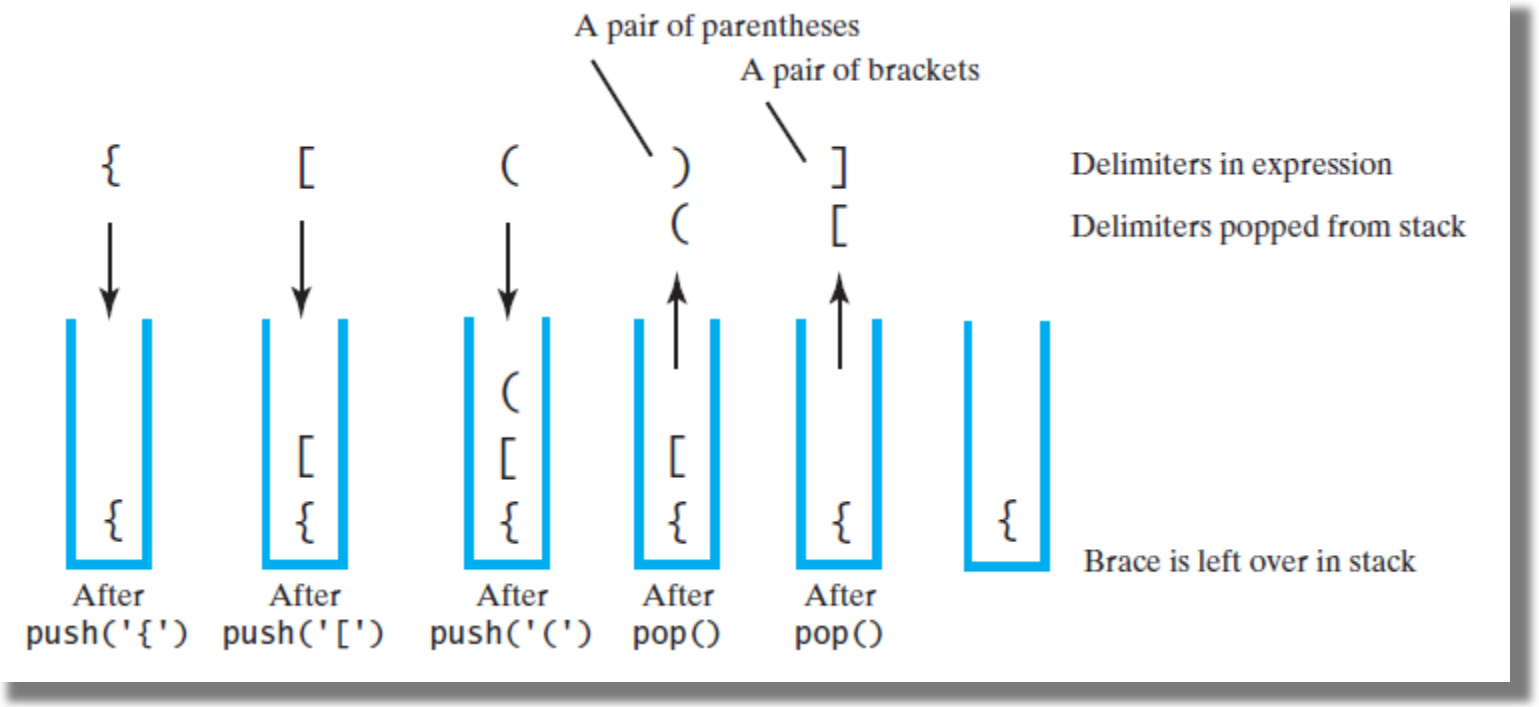
Processing Algebraic Expressions

- FIGURE 5-5 The contents of a stack during the scan of an expression that contains the unbalanced



Processing Algebraic Expressions

- FIGURE 5-6 The contents of a stack during the scan of an expression that contains the unbalanced delimiters { [( )]



Processing Algebraic Expressions

- Algorithm to process for balanced expression.

```
Algorithm checkBalance(expression)
// Returns true if the parentheses, brackets, and braces in an expression are paired correctly.

isBalanced = true
while ((isBalanced == true) and not at end of expression)
{
   nextCharacter = next character in expression
   switch (nextCharacter)
   {
      case '(': case '[': case '{':
         Push nextCharacter onto stack
         break

      case ')': case ']': case '}':
         if (stack is empty)
            isBalanced = false
         else
```

Processing Algebraic Expressions

- Algorithm to process for balanced expression.

```
            case ) : case ] : case } :
               if (stack is empty)
                  isBalanced = false
               else
               {
                  openDelimiter = top entry of stack
                  Pop stack
                  isBalanced = true or false according to whether openDelimiter and
                               nextCharacter are a pair of delimiters
               }
               break
         }
      }

      if (stack is not empty)
         isBalanced = false
      return isBalanced
```

Infix to Postfix

- FIGURE 5-7 Converting the infix expression a + b * c to postfix form

Next Character in Infix Expression	Postfix Form	Operator Stack (bottom to top)
a	*a*	
+	*a*	+
b	*a b*	+
*	*a b*	+ *
c	*a b c*	+ *
	a b c *	+
	a b c * +	

Successive Operators with Same Precedence

- FIGURE 5-8 Converting an infix expression to postfix form: (a) a - b + c;

Next Character in Infix Expression	Postfix Form	Operator Stack (bottom to top)
a	*a*	
−	*a*	−
b	*a b*	−
+	*a b* −	
	a b −	+
c	*a b* − *c*	+
	a b − *c* +	

Successive Operators with Same Precedence

- FIGURE 5-8 Converting an infix expression to postfix form: a ^ b ^ c

Next Character in Infix Expression	Postfix Form	Operator Stack (bottom to top)
a	*a*	
^	*a*	^
b	*a b*	^
^	*a b*	^ ^
c	*a b c*	^ ^
	a b c ^	^
	a b c ^ ^	

Infix-to-postfix Conversion

• Operand	Append each operand to the end of the output expression.
• Operator ^	Push ^ onto the stack.
• Operator +, -, *, or /	Pop operators from the stack, appending them to the output expression, until the stack is empty or its top entry has a lower precedence than the new operator. Then push the new operator onto the stack.
• Open parenthesis	Push (onto the stack.
• Close parenthesis	Pop operators from the stack and append them to the output expression until an open parenthesis is popped. Discard both parentheses.

http://www.cs.bilkent.edu.tr/~guvenir/courses/CS101/op_precedence.html

Infix-to-postfix Algorithm

```
Algorithm convertToPostfix(infix)
// Converts an infix expression to an equivalent postfix expression.

operatorStack = a new empty stack
postfix = a new empty string
while (infix has characters left to parse)
{
   nextCharacter = next nonblank character of infix
   switch (nextCharacter)
   {
      case variable:
         Append nextCharacter to postfix
         break

      case '^' :
         operatorStack.push(nextCharacter)
         break
```

Infix-to-postfix Algorithm

```
case '+' : case '-' : case '*' : case '/' :
   while (!operatorStack.isEmpty() and
          precedence of nextCharacter <= precedence of operatorStack.peek())
   {
      Append operatorStack.peek() to postfix
      operatorStack.pop()
   }
   operatorStack.push(nextCharacter)
   break

case '( ' :
   operatorStack.push(nextCharacter)
   break

case ')' : // Stack is not empty if infix expression is valid
   topOperator = operatorStack.pop()
   while (topOperator != '(')
   {
```

Infix-to-postfix Algorithm

```
                    Append topOperator to postfix
                    topOperator = operatorStack.pop()
                }
                break
            default: break // Ignore unexpected characters
        }
    }
    while (!operatorStack.isEmpty())
    {
        topOperator = operatorStack.pop()
        Append topOperator to postfix
    }
    return postfix
```

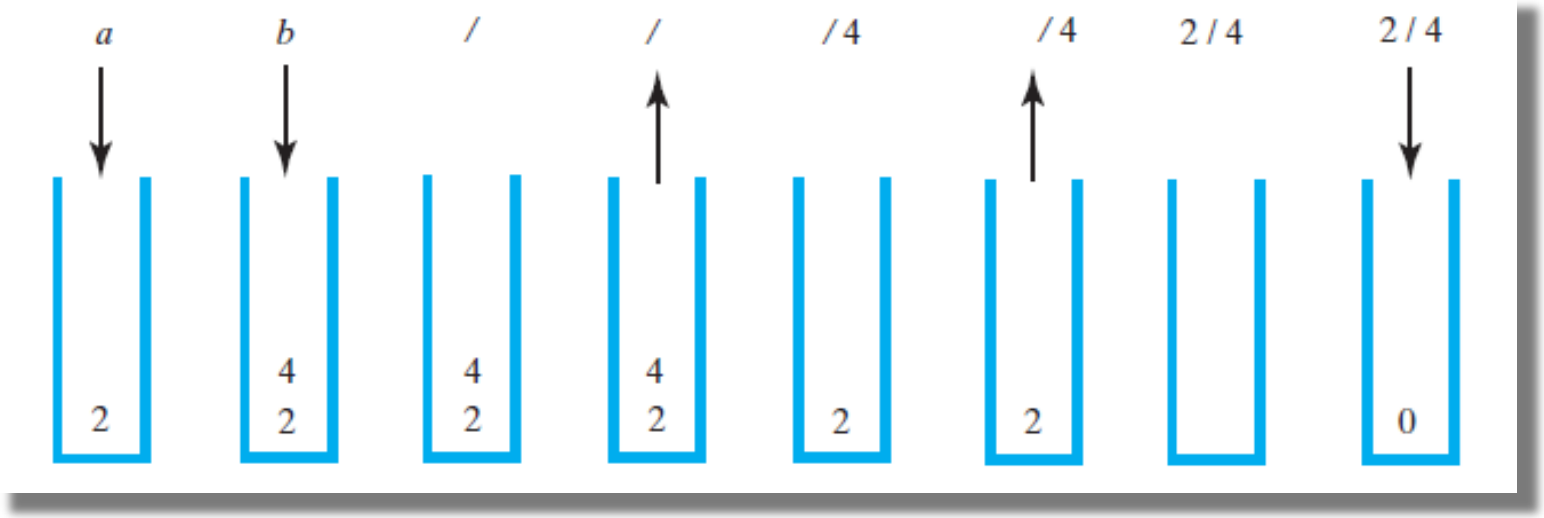
Infix to Postfix

- FIGURE 5-9 The steps in converting the infix expression a / b * (c + (d - e)) to postfix form

Next Character from Infix Expression	Postfix Form	Operator Stack (bottom to top)
a	*a*	
/	*a*	/
b	*a b*	/
*	*a b* /	
	a b /	*
(	*a b* /	* (
c	*a b* / *c*	* (
+	*a b* / *c*	* (+
(	*a b* / *c*	* (+ (
d	*a b* / *c d*	* (+ (
−	*a b* / *c d*	* (+ (−
e	*a b* / *c d e*	* (+ (−
)	*a b* / *c d e* −	* (+ (
	a b / *c d e* −	* (+
)	*a b* / *c d e* − +	* (
	a b / *c d e* − +	*
	a b / *c d e* − + *	

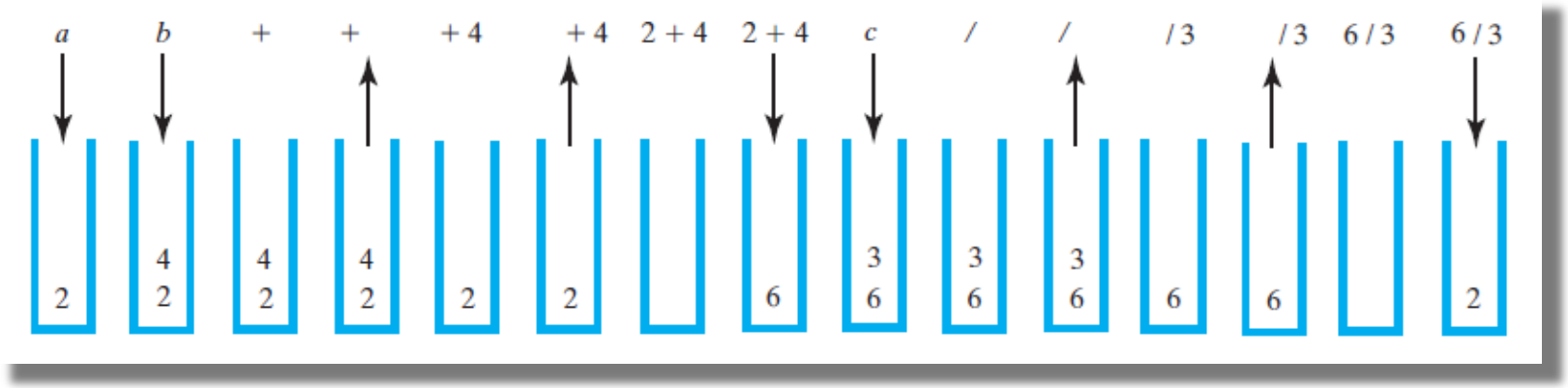
Evaluating Postfix Expressions

- FIGURE 5-10 The stack during the evaluation of the postfix expression a b / when a is 2 and b is 4



Evaluating Postfix Expressions

- FIGURE 5-11 The stack during the evaluation of the postfix expression a b + c / when a is 2, b is 4, and c is 3



Evaluating Postfix Expressions

- Algorithm for evaluating postfix expressions.

```
Algorithm evaluatePostfix(postfix)
// Evaluates a postfix expression.

valueStack = a new empty stack
while (postfix has characters left to parse)
{
    nextCharacter = next nonblank character of postfix
    switch (nextCharacter)
    {
      case variable:
         valueStack.push(value of the variable nextCharacter)
         break

      case '+' : case '-' : case '*' : case '/' : case '^' :
```

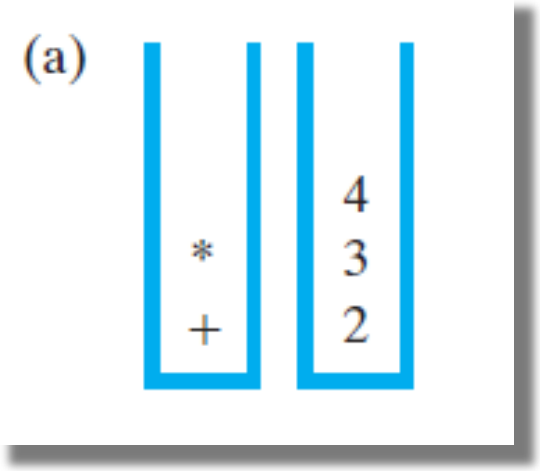
Evaluating Postfix Expressions

- Algorithm for evaluating postfix expressions.

```
            break
         case '+' : case '-' : case '*' : case '/' : case '^' :
            operandTwo = valueStack.pop()
            operandOne = valueStack.pop()
            result = the result of the operation in nextCharacter and its operands
                     operandOne and operandTwo
            valueStack.push(result)
            break
         default: break // Ignore unexpected characters
      }
   }
```

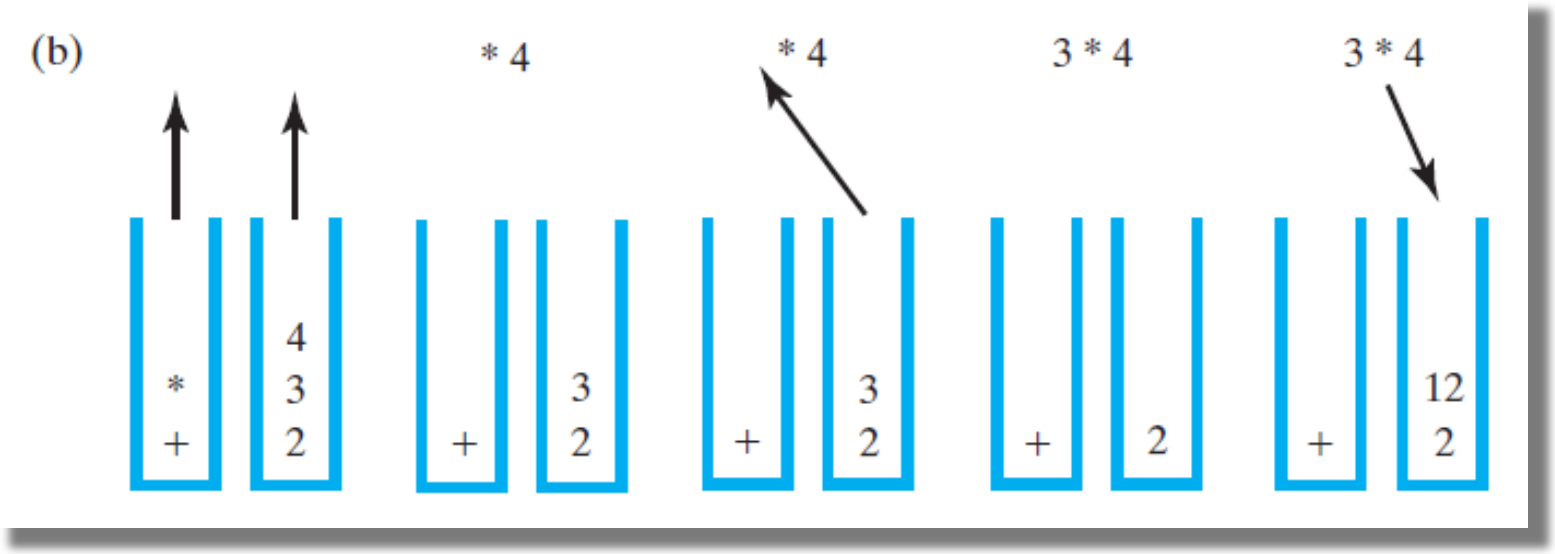
Evaluating Infix Expressions

- FIGURE 5-12 Two stacks during the evaluation of a + b * c when a is 2, b is 3, and c is 4:
- (a) after reaching the end of the expression;



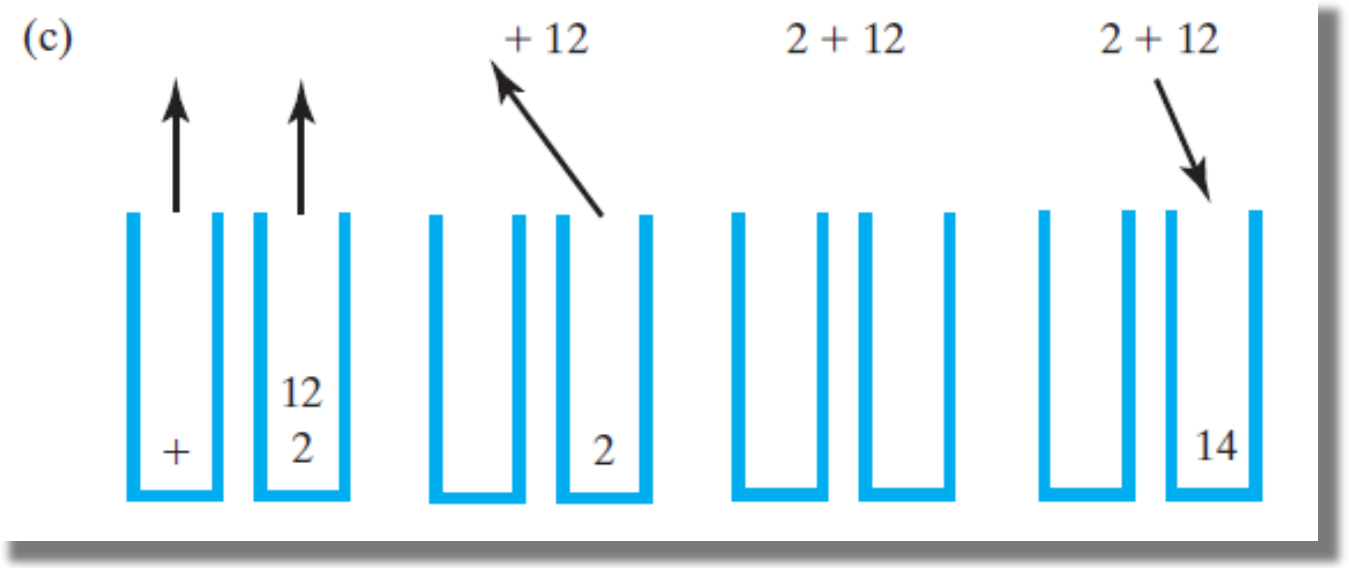
Evaluating Infix Expressions

- FIGURE 5-12 Two stacks during the evaluation of a + b * c when a is 2, b is 3, and c is 4:
  (b) while performing the multiplication;



Evaluating Infix Expressions

- FIGURE 5-12 Two stacks during the evaluation of a + b * c when a is 2, b is 3, and c is 4:
  (c) while performing the addition



Evaluating Infix Expressions

- Algorithm to evaluate infix expression.

```
Algorithm evaluateInfix(infix)
// Evaluates an infix expression.

operatorStack = a new empty stack
valueStack = a new empty stack
while (infix has characters left to process)
{
   nextCharacter = next nonblank character of infix
   switch (nextCharacter)
   {
      case variable:
         valueStack.push(value of the variable nextCharacter)
         break

      case '^' :
         operatorStack.push(nextCharacter)
         break

      case '+' : case '-' : case '*' : case '/' :
         while (!operatorStack.isEmpty() and
```

Evaluating Infix Expressions

- Algorithm to evaluate infix expression.

```
case '+' : case '-' : case '*' : case '/' :
   while (!operatorStack.isEmpty() and
          precedence of nextCharacter <= precedence of operatorStack.peek())
   {
      // Execute operator at top of operatorStack
      topOperator = operatorStack.pop()
      operandTwo = valueStack.pop()
      operandOne = valueStack.pop()
      result = the result of the operation in topOperator and its operands
               operandOne and operandTwo
      valueStack.push(result)
   }
   operatorStack.push(nextCharacter)
   break

case '(' :
   operatorStack.push(nextCharacter)
   break

case ')' : // Stack is not empty if infix expression is valid
```

Evaluating Infix Expressions

- Algorithm to evaluate infix expression.

```
case '(' :
   operatorStack.push(nextCharacter)
   break

case ')' : // Stack is not empty if infix expression is valid
   topOperator = operatorStack.pop()
   while (topOperator != '(')
   {
      operandTwo = valueStack.pop()
      operandOne = valueStack.pop()
      result = the result of the operation in topOperator and its operands
               operandOne and operandTwo
      valueStack.push(result)
      topOperator = operatorStack.pop()
   }
   break
```

Evaluating Infix Expressions

- Algorithm to evaluate infix expression.

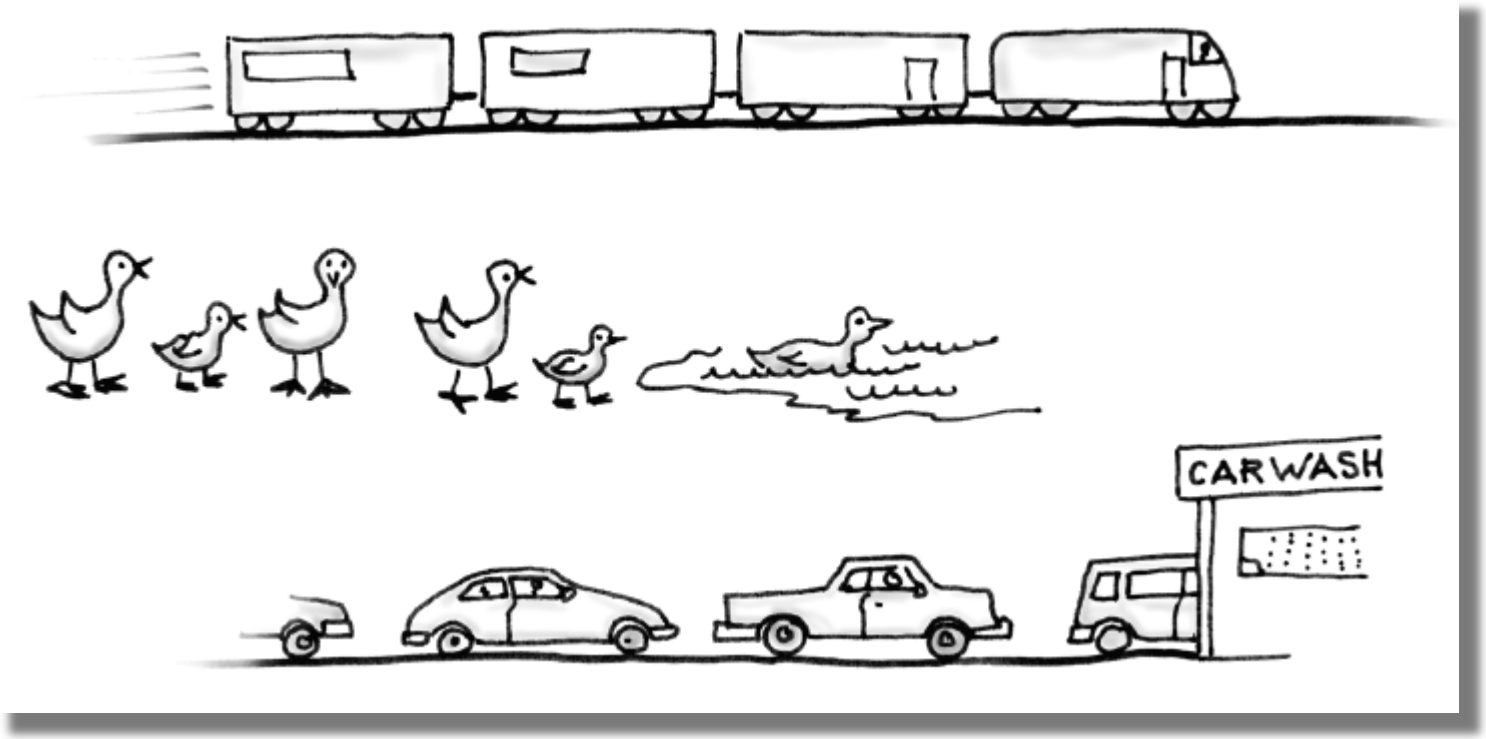
```
        default: break // Ignore unexpected characters
    }
}
while (!operatorStack.isEmpty())
{
  topOperator = operatorStack.pop()
  operandTwo = valueStack.pop()
  operandOne = valueStack.pop()
  result = the result of the operation in topOperator and its operands
           operandOne and operandTwo
  valueStack.push(result)
}
return valueStack.peek()
```

The ADT Queue

- A queue is another name for a waiting line
- Used within operating systems and to simulate real-world events
  - Come into play whenever processes or events must wait
- Entries organized first-in, first-out

The ADT Queue

- FIGURE 10-1 Some everyday queues



The ADT Queue

- Terminology
  - Item added first, or earliest, is at the front of the queue
  - Item added most recently is at the back of the queue
- Additions to a software queue must occur at its back
- Client can look at or remove only the entry at the front of the queue

The ADT Queue

Abstract Data Type: Queue

Data

- A collection of objects in chronological order and having the same data type

Operations

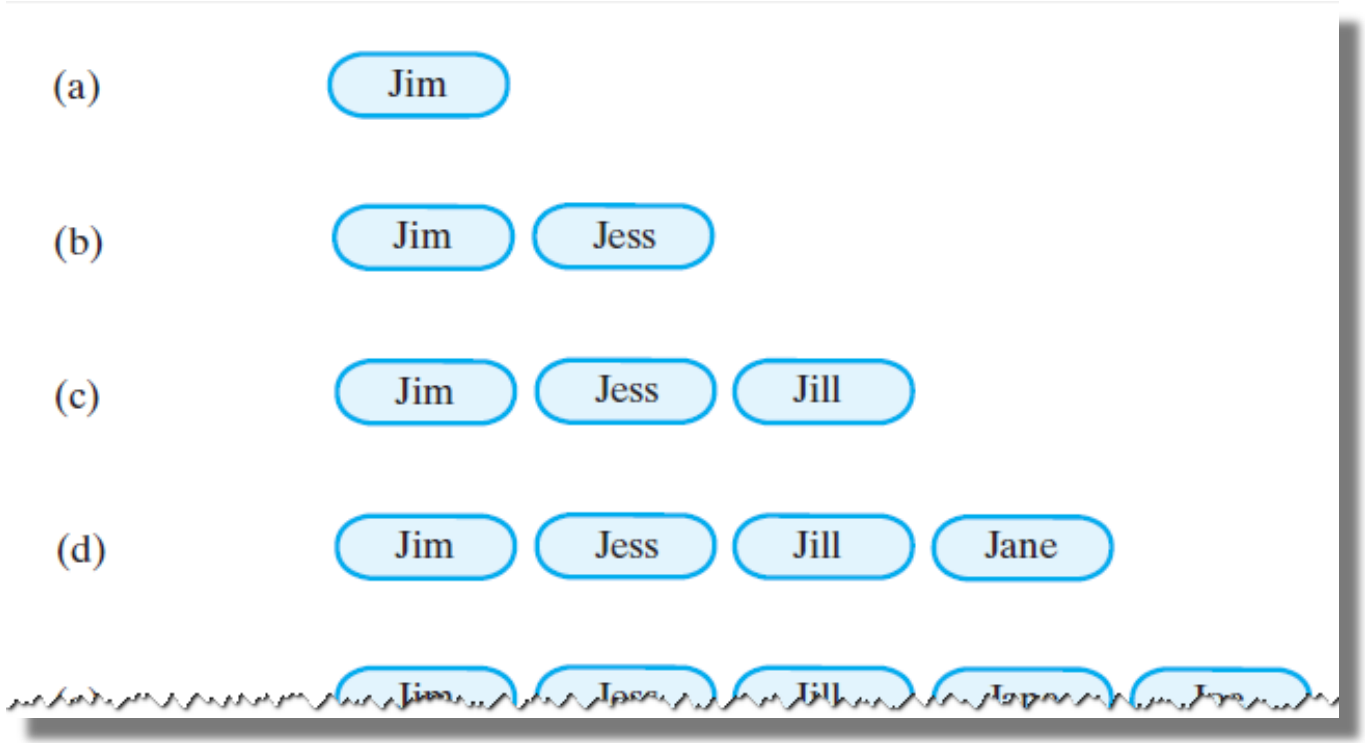
Pseudocode	UML	Description
`enqueue(newEntry)`	`+enqueue(newEntry: integer): void`	Task: Adds a new entry to the back of the queue. Input: `newEntry` is the new entry. Output: None.
`dequeue()`	`+dequeue(): T`	Task: Removes and returns the entry at the front of the queue. Input: None. Output: Returns the queue's front entry. Throws an exception if the queue is empty before the operation.

The ADT Queue

`getFront()`	`+getFront(): T`	Task: Retrieves the queue's front entry without changing the queue in any way. Input: None. Output: Returns the queue's front entry. Throws an exception if the queue is empty.
`isEmpty()`	`+isEmpty(): boolean`	Task: Detects whether the queue is empty. Input: None. Output: Returns true if the queue is empty.
`clear()`	`+clear(): void`	Task: Removes all entries from the queue. Input: None. Output: None.

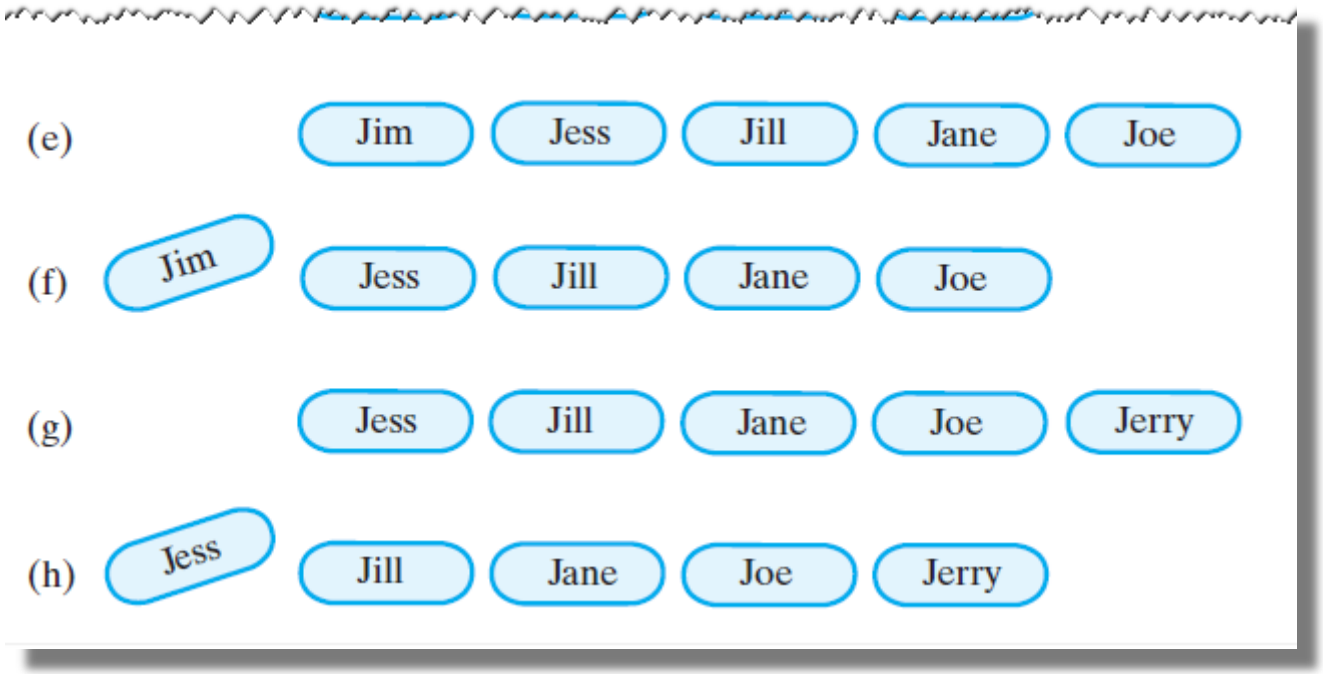
The ADT Queue

- FIGURE 10-2 A queue of strings after (a) enqueue adds Jim; (b) enqueue adds Jess; (c) enqueue adds Jill; (d) enqueue adds Jane;



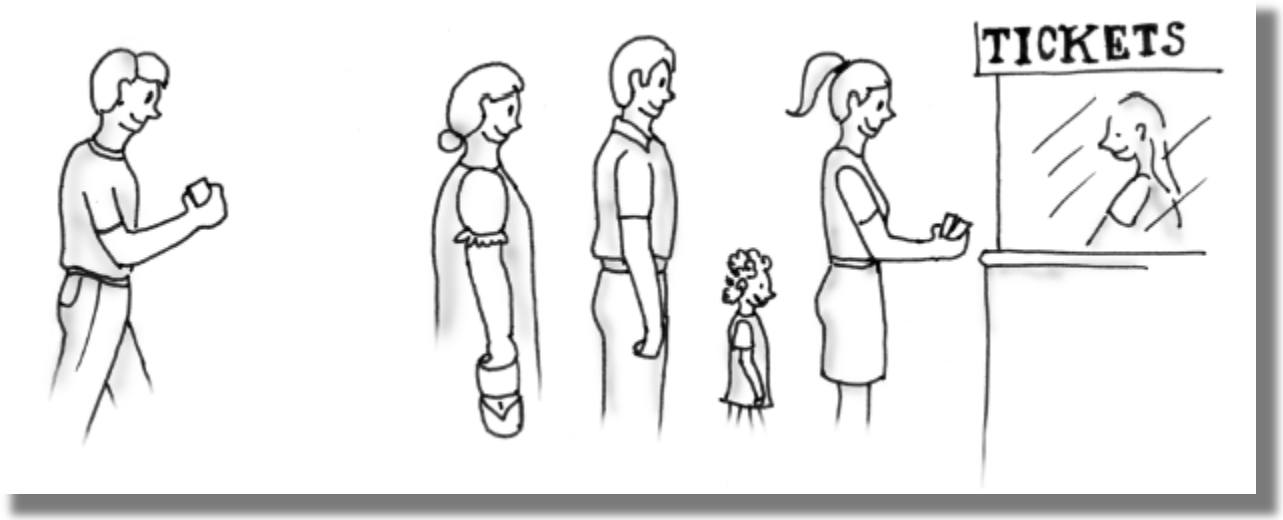
The ADT Queue

- FIGURE 10-2 A queue of strings after (e) enqueue adds Joe; (f) dequeue retrieves and removes Jim; (g) enqueue adds Jerry; (h) dequeue retrieves and removes Jess



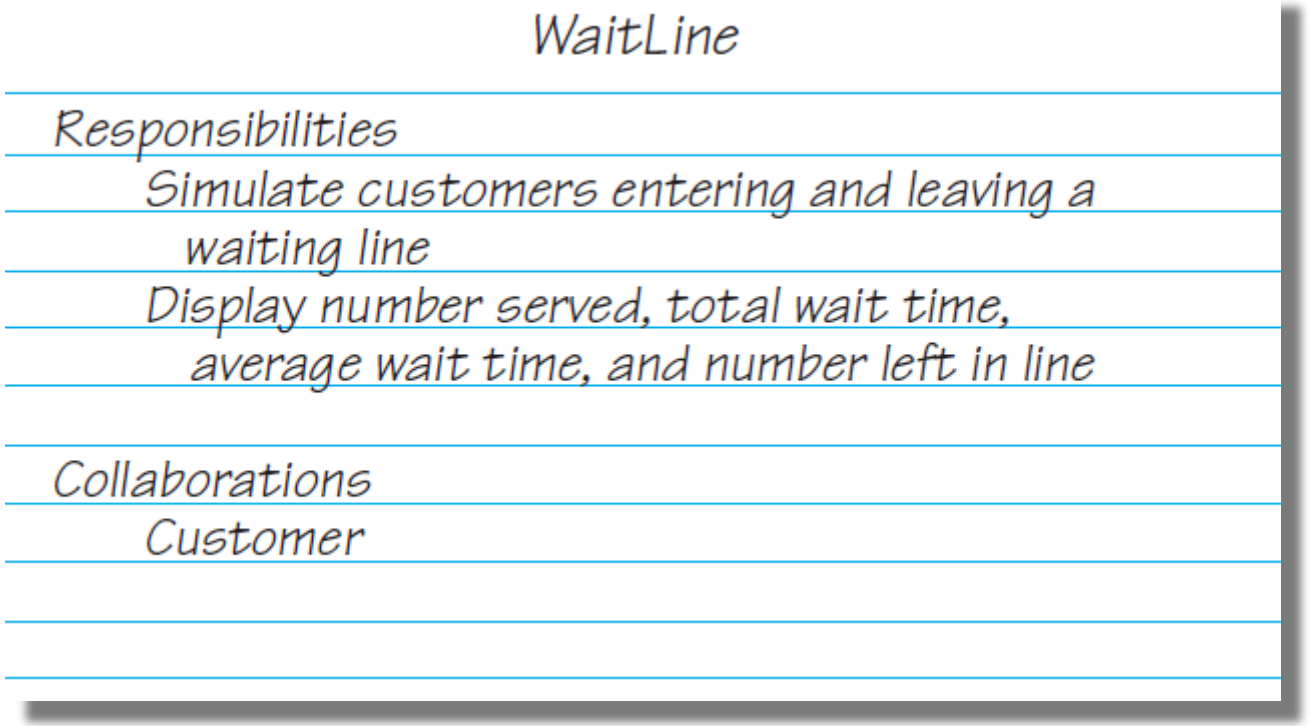
Simulating a Waiting Line

- FIGURE 10-3 A line, or queue, of people



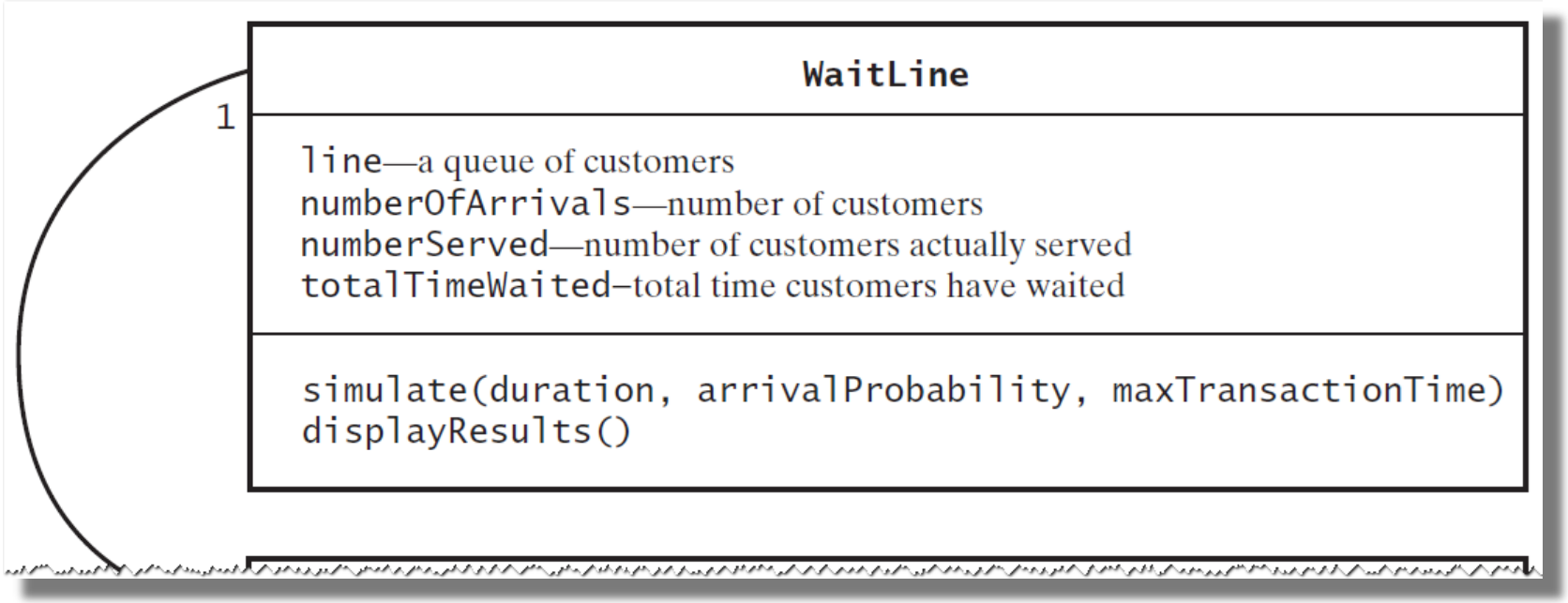
Simulating a Waiting Line

- FIGURE 10-4 A CRC card for the class WaitLine



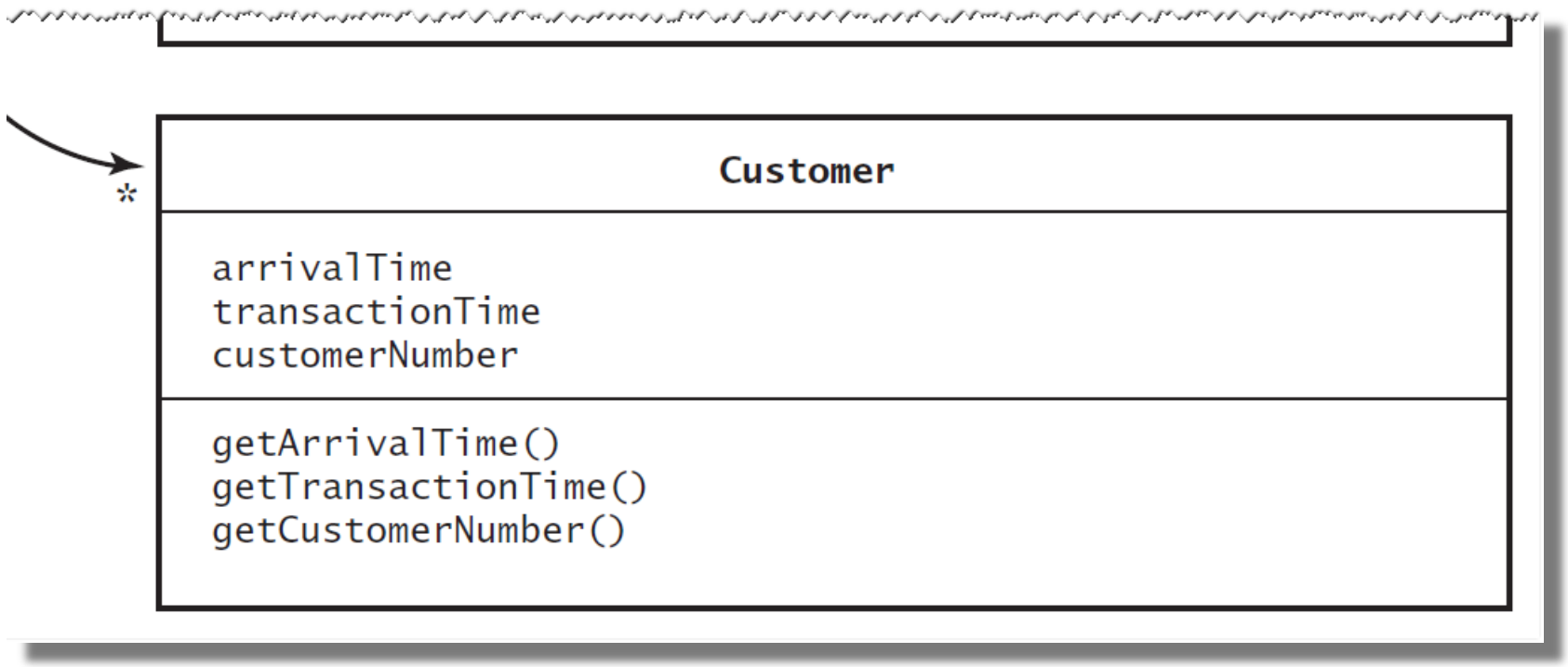
Simulating a Waiting Line

- FIGURE 10-5 A diagram of the classes WaitLine and Customer



Simulating a Waiting Line

- FIGURE 10-5 A diagram of the classes WaitLine and Customer



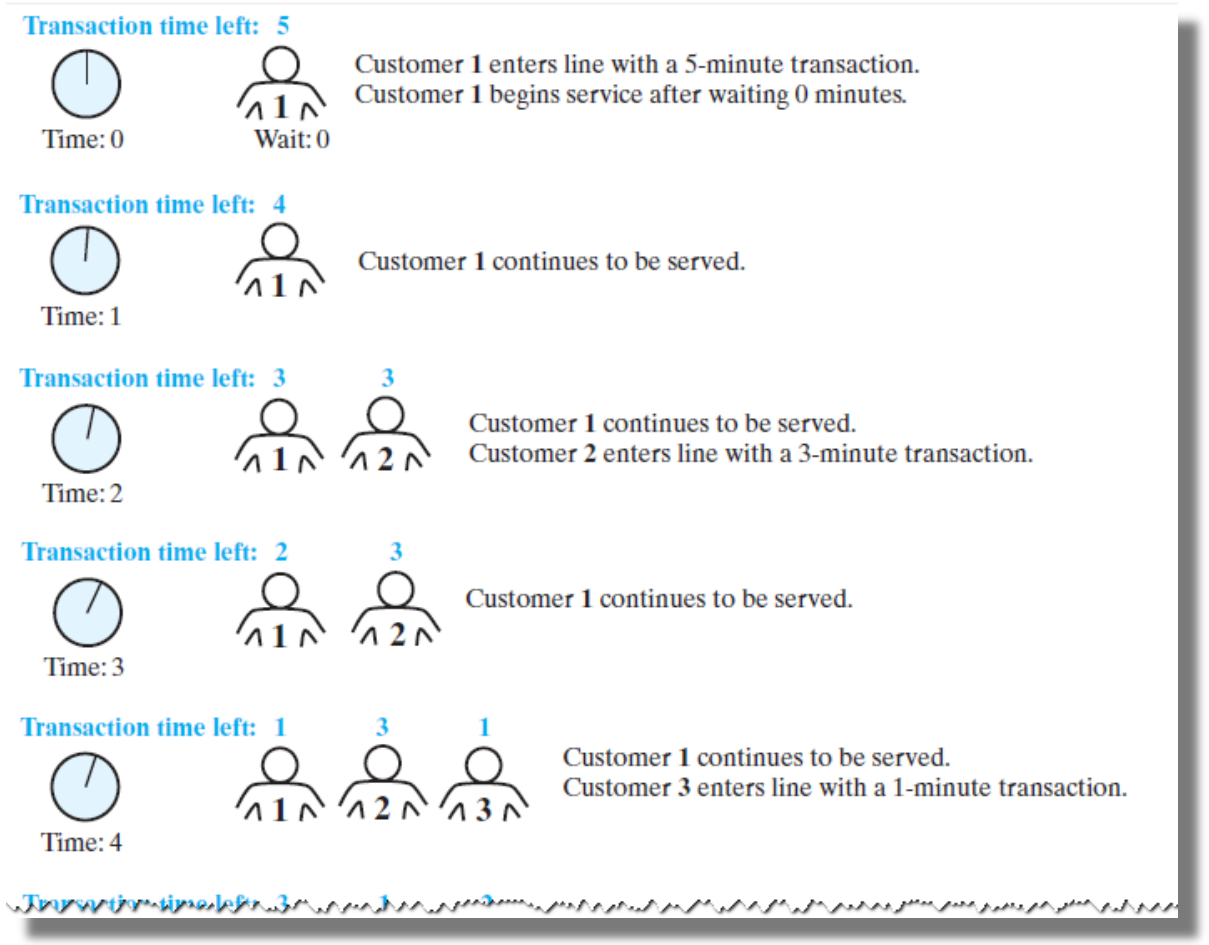
Simulating a Waiting Line

- Algorithm for simulate

```
Algorithm simulate(duration, arrivalProbability, maxTransactionTime)
transactionTimeLeft = 0
for (clock = 0; clock < duration; clock++)
{
   if (a new customer arrives)
   {
      numberOfArrivals++
      transactionTime = a random time that does not exceed maxTransactionTime
      nextArrival = a new customer containing clock, transactionTime, and
                    a customer number that is numberOfArrivals
      line.enqueue(nextArrival)
   }
   if (transactionTimeLeft > 0) // If present customer is still being served
      transactionTimeLeft--
   else if (!line.isEmpty())
   {
      nextCustomer = line.dequeue()
      transactionTimeLeft = nextCustomer.getTransactionTime() - 1
      timeWaited = clock - nextCustomer.getArrivalTime()
      totalTimeWaited = totalTimeWaited + timeWaited
      numberServed++
   }
}
```

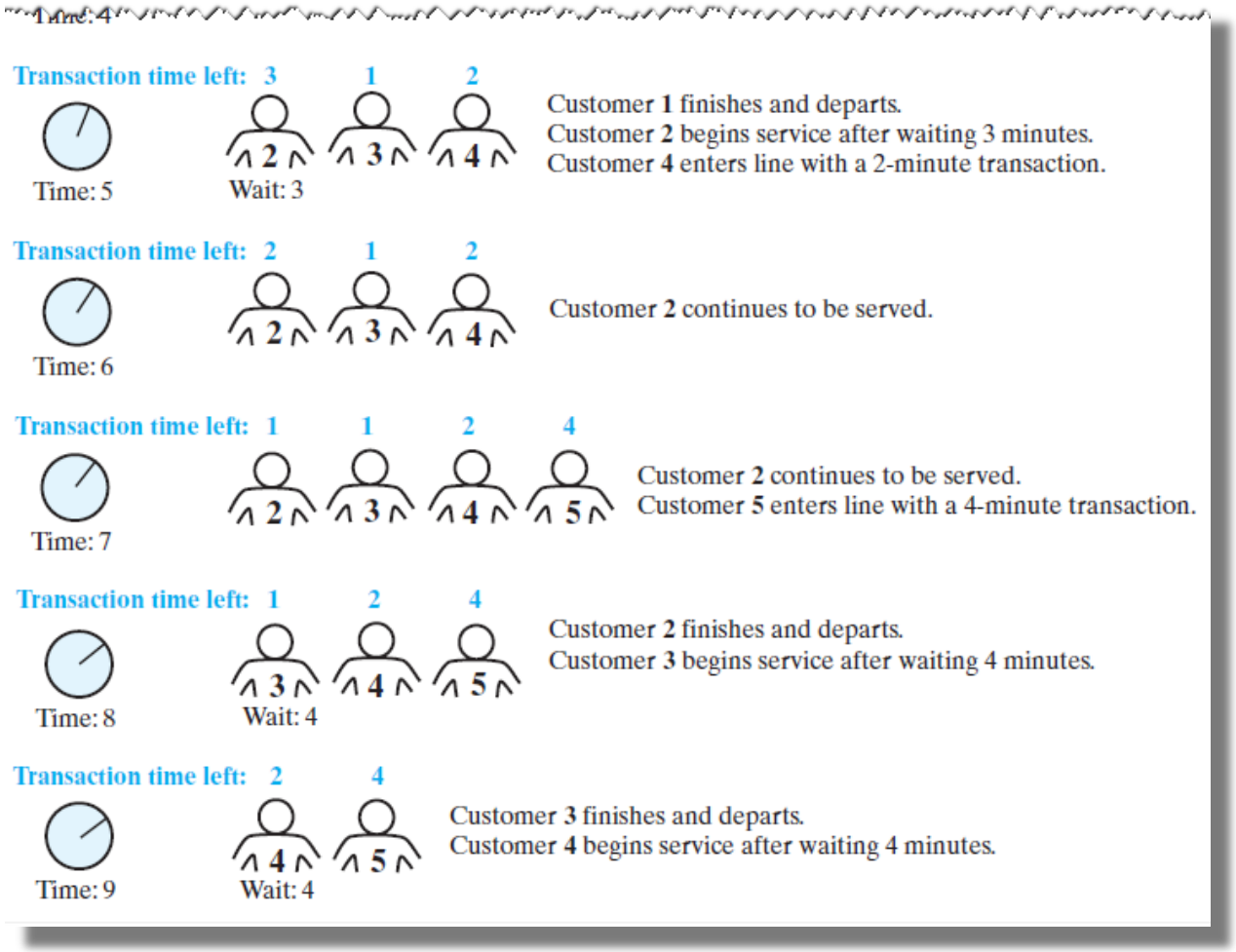
Simulating a Waiting Line

- FIGURE 10-6 A simulated waiting line



Simulating a Waiting Line

- FIGURE 10-6 A simulated waiting line

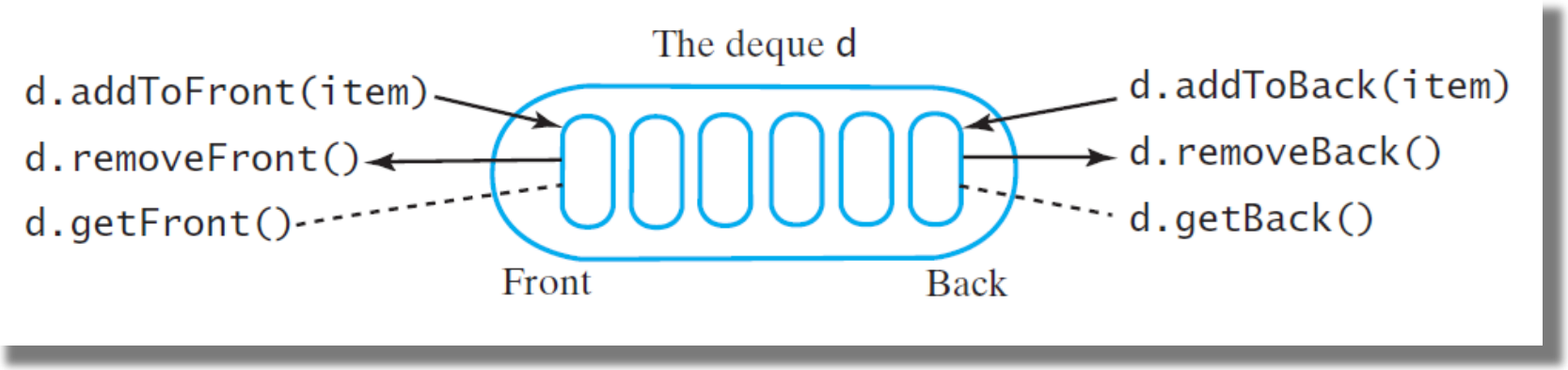


The ADT Deque

- A double ended queue
- Deque pronounced “deck”
- Has both queuelike operations and stacklike operations

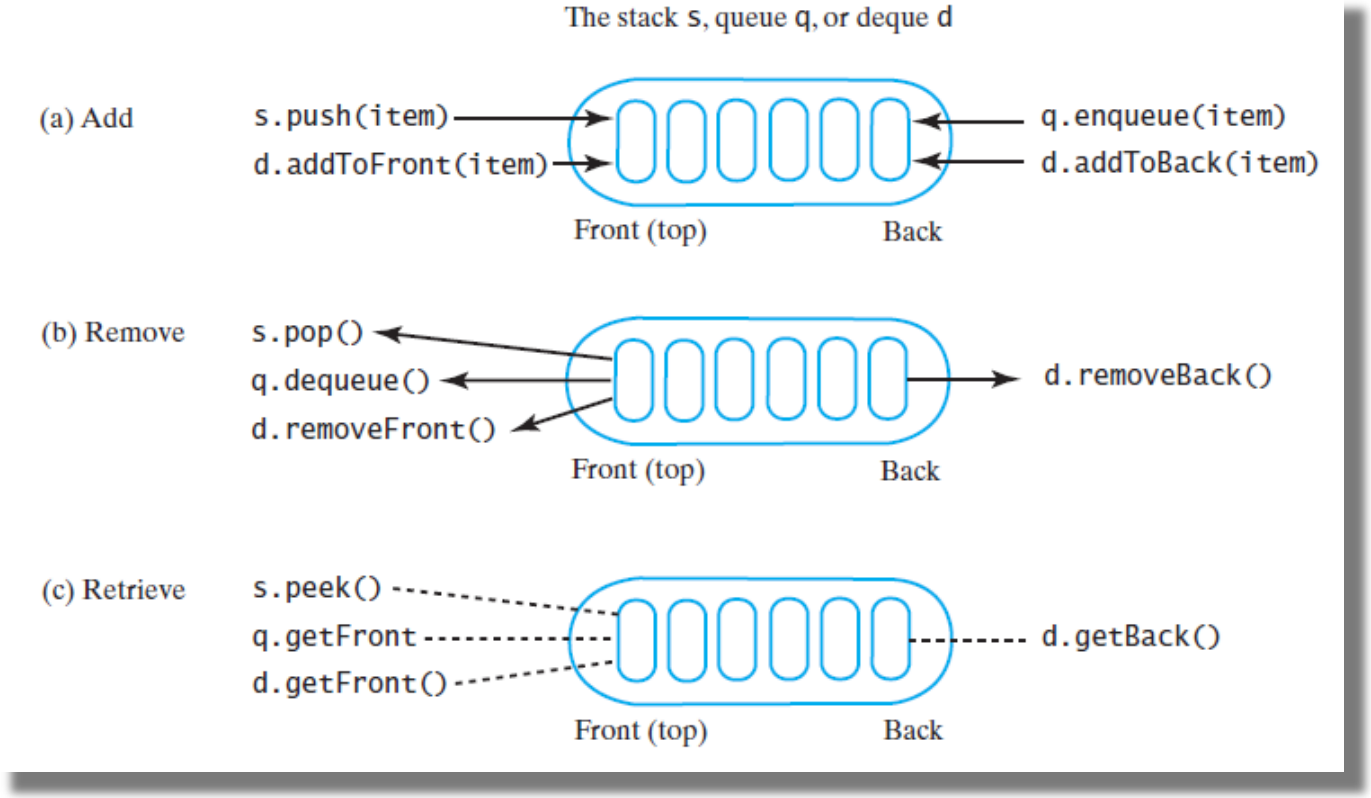
The ADT Deque

- FIGURE 10-10 An instance d of a deque



The ADT Deque

- FIGURE 10-11 A comparison of operations for a stack s, a queue q, and a deque d: (a) add; (b) remove; (c) retrieve



The ADT Deque

- Pseudocode that uses a deque to read and display a line of keyboard input

```
// Read a line
d = a new empty deque
while (not end of line)
{
   character = next character read
   if (character == ←)
      d.removeBack()
   else
      d.addToBack(character)
}
// Display the corrected line
while (!d.isEmpty())
   System.out.print(d.removeFront())
System.out.println()
```

ADT Priority Queue

- Consider how a hospital assigns a priority to each patient that overrides time at which patient arrived.
- ADT priority queue organizes objects according to their priorities
- Definition of “priority” depends on nature of the items in the queue