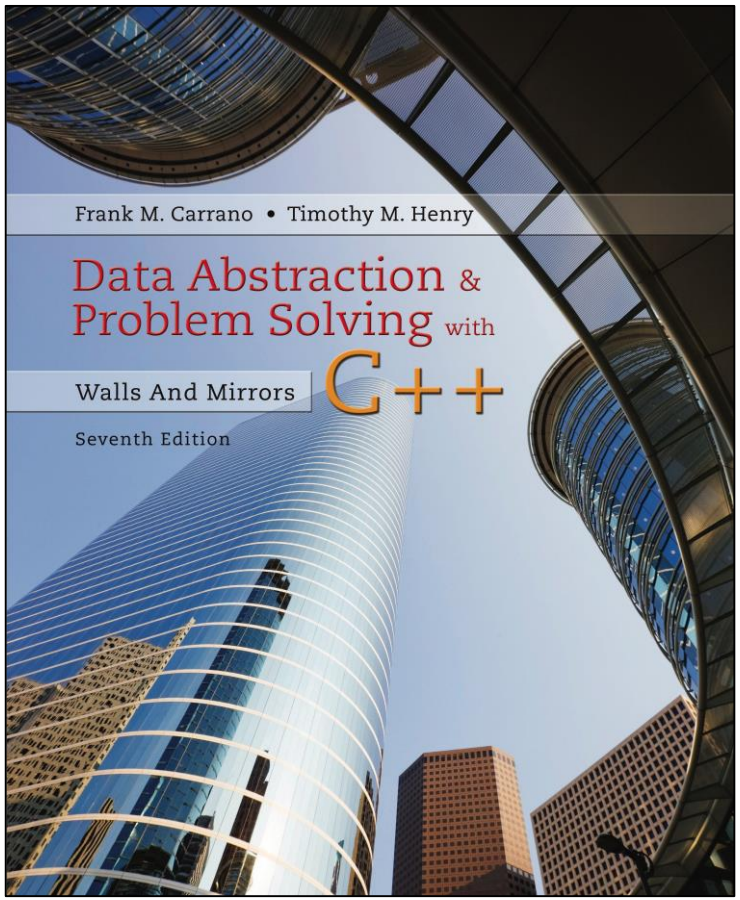
Data Abstraction & Problem Solving with C++: Walls and Mirrors

Seventh Edition



Chapter 13

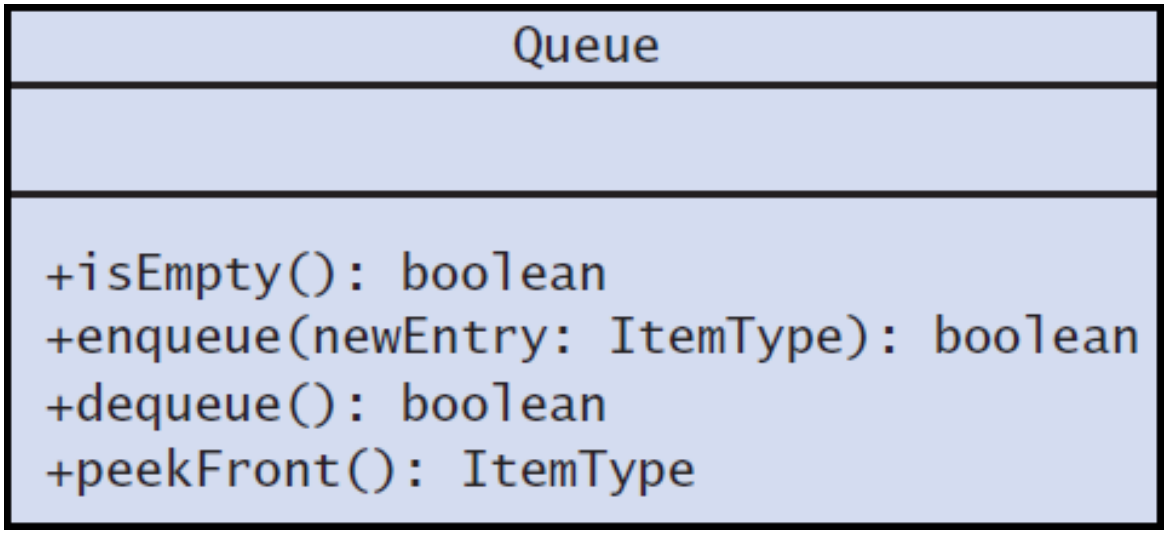
Queues and Priority Queues

The ADT Queue (1 of 5)

- Like a line of people
  - First person in line is first person served
  - New elements of queue enter at its back
  - Items leave the queue from its front
- Called FIFO behavior
  - **F**irst **I**n **F**irst **O**ut

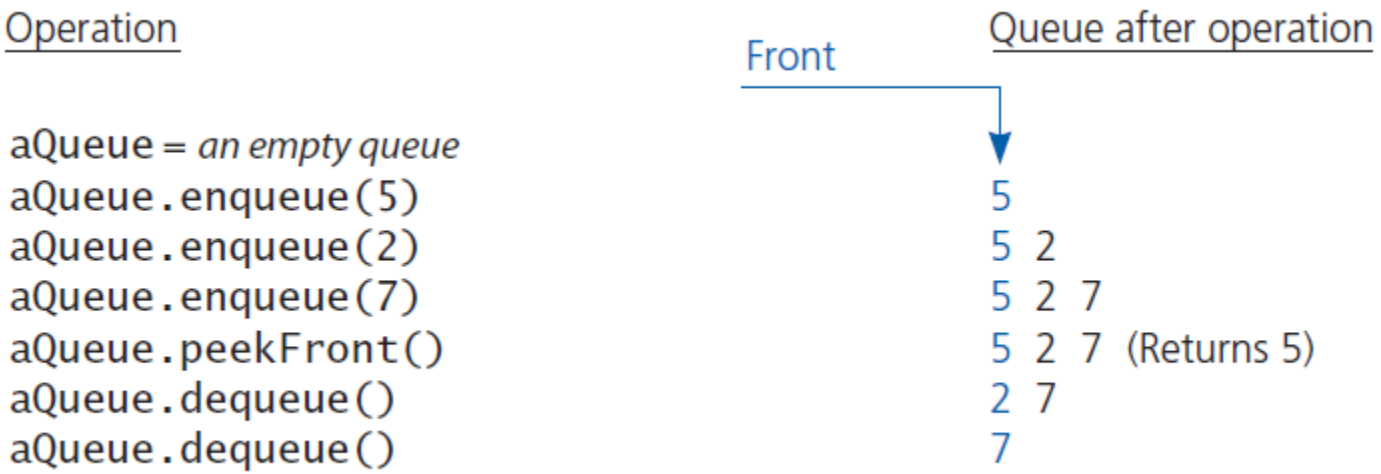
The ADT Queue (2 of 5)

Figure 13-1 UML diagram for the class Queue



The ADT Queue (3 of 5)

Figure 13-2 Some queue operations



The ADT Queue (4 of 5)

Listing 13-1 A C++ interface for queues

```
1  /** @file QueueInterface.h */
2  #ifndef QUEUE_INTERFACE_
3  #define QUEUE_INTERFACE_
4
5  template<class ItemType>
6  class QueueInterface
7  {
8  public:
9     /** Sees whether this queue is empty.
10     @return  True if the queue is empty, or false if not. */
11    virtual bool isEmpty() const = 0;
12
13    /** Adds a new entry to the back of this queue.
14     @post  If the operation was successful, newEntry is at the
15        back of the queue.
16     @param newEntry  The object to be added as a new entry.
17     @return  True if the addition is successful or false if not. */
18     virtual bool enqueue(const ItemType& newEntry) = 0;
```

The ADT Queue (5 of 5)

Listing 13-1 [Continued]

```
18      virtual bool enqueue(const ItemType& newEntry) = 0;
19
20      /** Removes the front of this queue.
21       @post  If the operation was successful, the front of the queue
22          has been removed.
23       @return  True if the removal is successful or false if not. */
24      virtual bool dequeue() = 0;
25
26      /** Returns the front of this queue.
27       @pre  The queue is not empty.
28       @post  The front of the queue has been returned, and the
29          queue is unchanged.
30       @return  The front of the queue. */
31      virtual ItemType peekFront() const = 0;
32
33      /** Destroys this queue and frees its memory. */
34      virtual ~QueueInterface() { }
35  }; // end QueueInterface
36  #endif
```

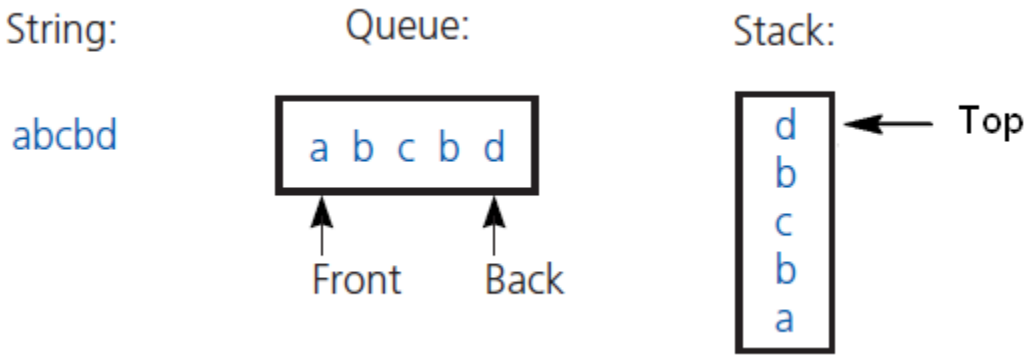
Applications Reading a String of Characters

Pseudocode to read a string of characters into a queue.

```
// Read a string of characters from a single line of input into a queue
aQueue = a new empty queue
while (not end of line)
{
    Read a new character into ch
    aQueue.enqueue(ch)
}
```

Applications Recognizing a Palindrome

Figure 13-3 The results of inserting the characters a, b, c, b, d into both a queue and a stack



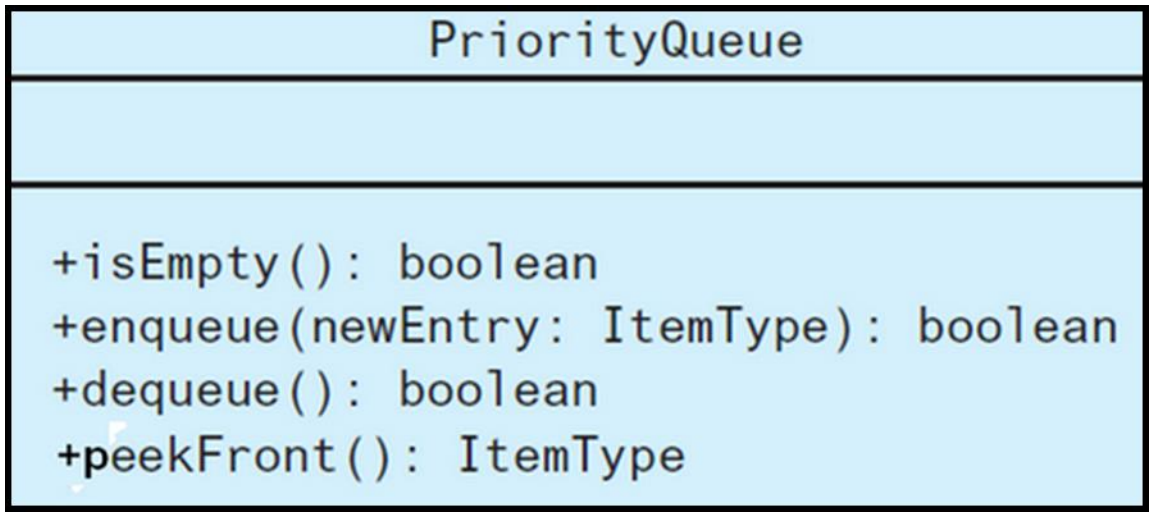
- Remove characters from front of queue, top of stack
- Compare each pair removed
- If all pairs match, string is a palindrome

The ADT Priority Queue (1 of 2)

- Organize data by priorities
  - Example: weekly “to do” list
- Priority value
  - We will say high value ⇨ high priority
- Operations
  - Test for empty
  - Add to queue in sorted position
  - Remove/get entry with highest priority

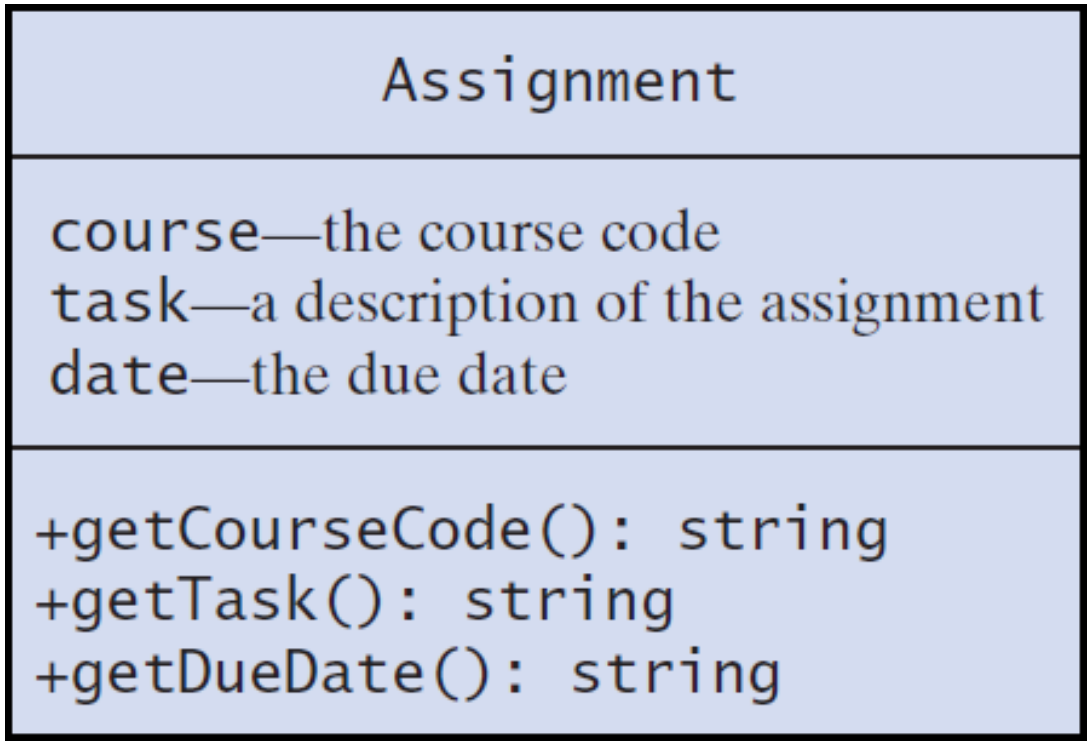
The ADT Priority Queue (2 of 2)

Figure 13-4 UML diagram for the class **PriorityQueue**



Tracking Your Assignments (1 of 2)

Figure 13-5 UML diagram for the class Assignment



Tracking Your Assignments (2 of 2)

Pseudocode to organize assignments, responsibilities

```
assignmentLog = a new priority queue using due date as the priority value
project = a new instance of Assignment
essay = a new instance of Assignment
quiz = a new instance of Assignment
errand = a new instance of Assignment
assignmentLog.enqueue(project)
assignmentLog.enqueue(essay)
assignmentLog.enqueue(quiz)
assignmentLog.enqueue(errand)
cout << "I should do the following first: "
cout << assignmentLog.peekFront()
```

Application: Simulation (1 of 9)

- Simulation models behavior of systems
- Problem to solve
  - Approximate average time bank customer must wait for service from a teller
  - Decrease in customer wait time with each new teller added

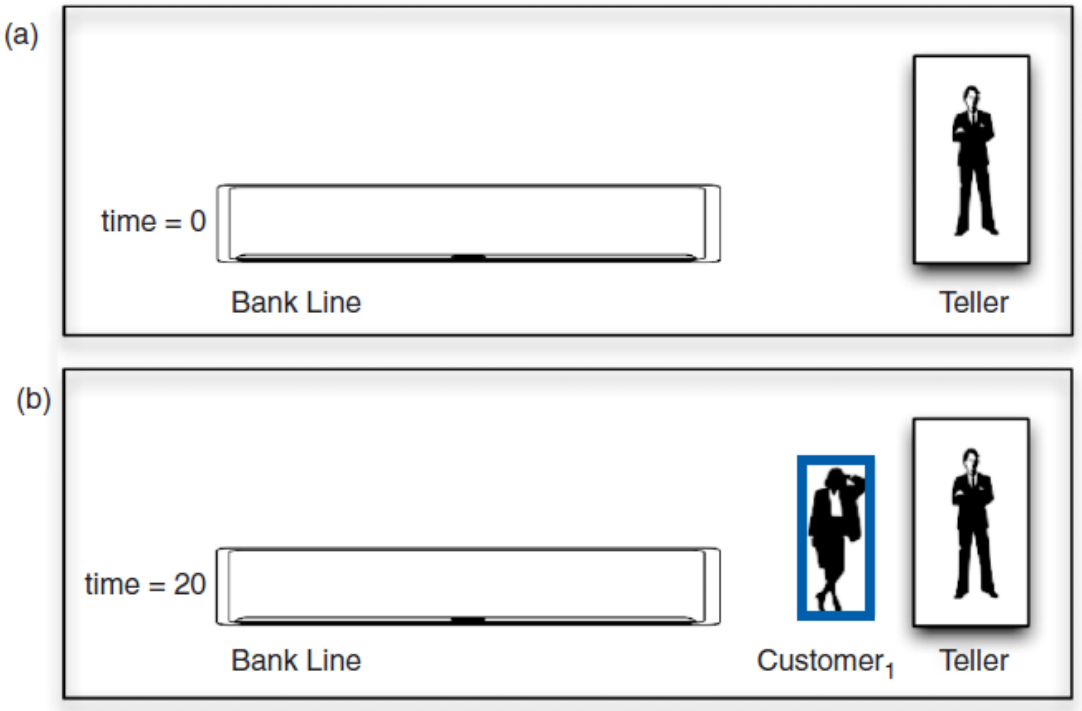
Application: Simulation (2 of 9)

Sample arrival and transaction times

Arrival time	Transaction length
20	6
22	4
23	2
30	3

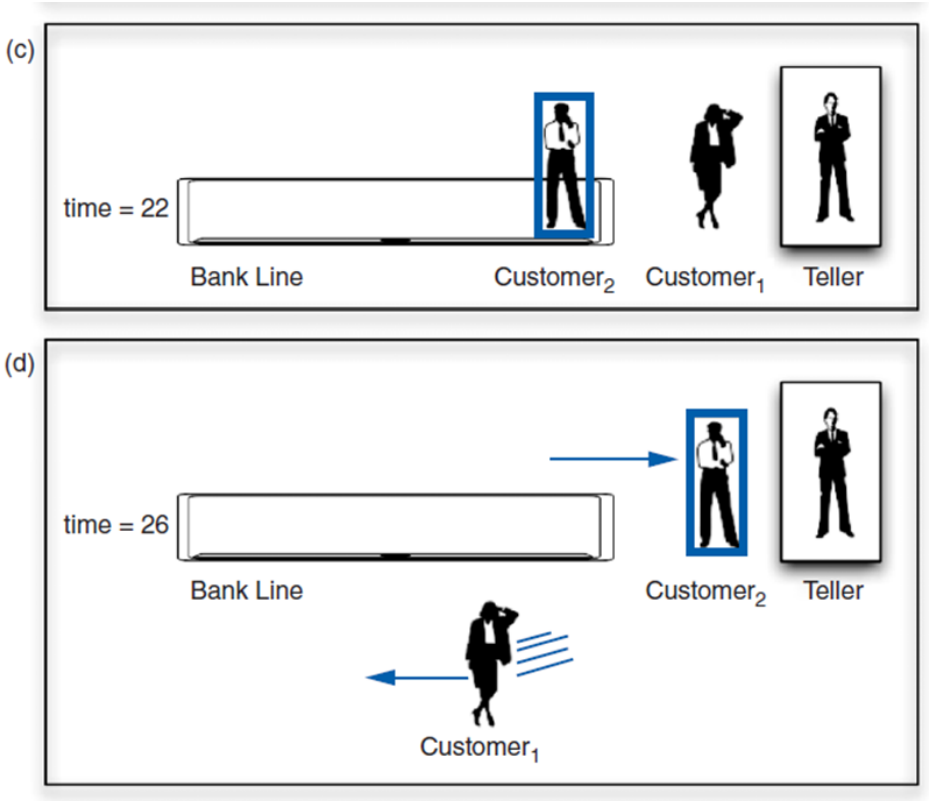
Application: Simulation (3 of 9)

Figure 13-6 A bank line at time (a) 0; (b) 20; (c) 22; (d) 26



Application: Simulation (4 of 9)

Figure 13-6 [Continued]



Application: Simulation (5 of 9)

Pseudocode for an event loop

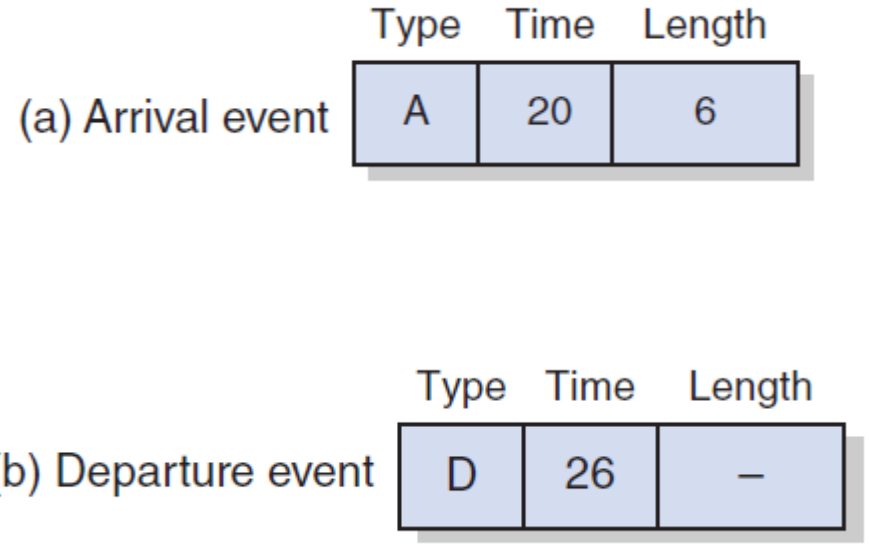
```
Initialize the line to "no customers"
while (events remain to be processed)
{
    currentTime = time of next event
    if (event is an arrival event)
        Process the arrival event
    else
        Process the departure event

    // When an arrival event and a departure event occur at the same time,
    // arbitrarily process the arrival event first
}
```

Application: Simulation (6 of 9)

- Time-driven simulation
  - Simulates the ticking of a clock
- Event-driven simulation considers
  - Only the times of certain events,
  - In this case, arrivals and departures
- Event list contains
  - All future arrival and departure events

Application: Simulation (7 of 9)

Figure 13-7 A typical instance of (a) an arrival event; (b) a departure event

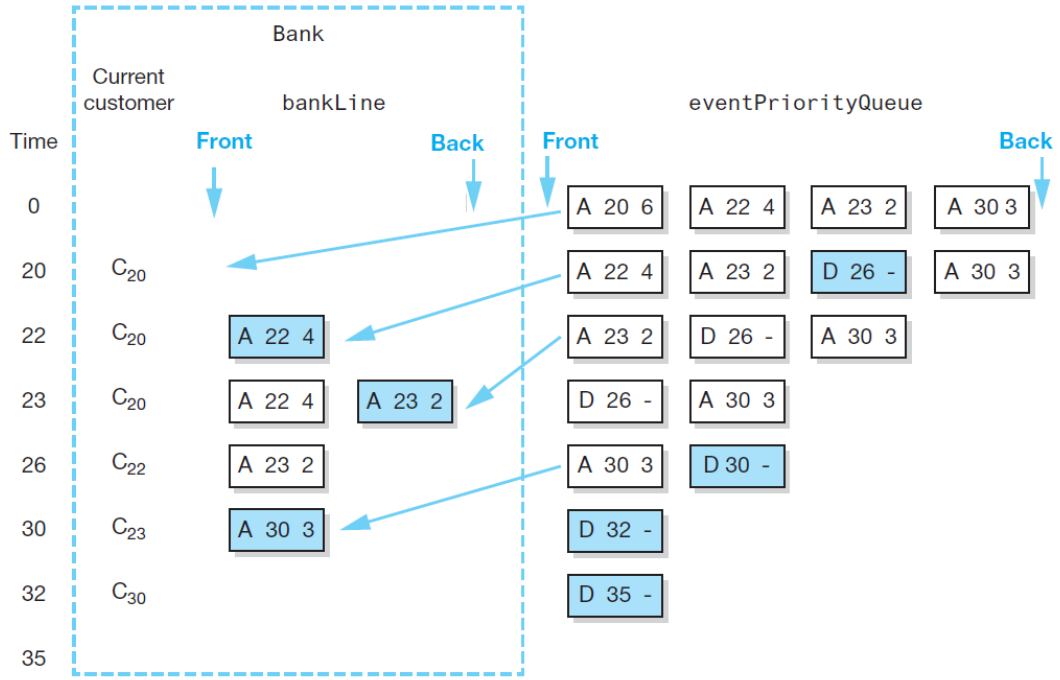


Application: Simulation (8 of 9)

- Two tasks required to process each event
  - Update the bank line: Add or remove customers.
  - Update the event queue: Add or remove events.
- New customer
  - Always enters bank line
  - Served while at the front of the line

Application: Simulation (9 of 9)

Figure 13-8 A trace of the bank simulation algorithm for the data (20, 6), (22, 4), (23, 2), (30, 3)



Position-Oriented and Value-Oriented ADTs (1 of 3)

- Position-oriented ADTs
  - List, stack, queue
- Value-oriented ADTs
  - Sorted list

Position-Oriented and Value-Oriented ADTs (2 of 3)

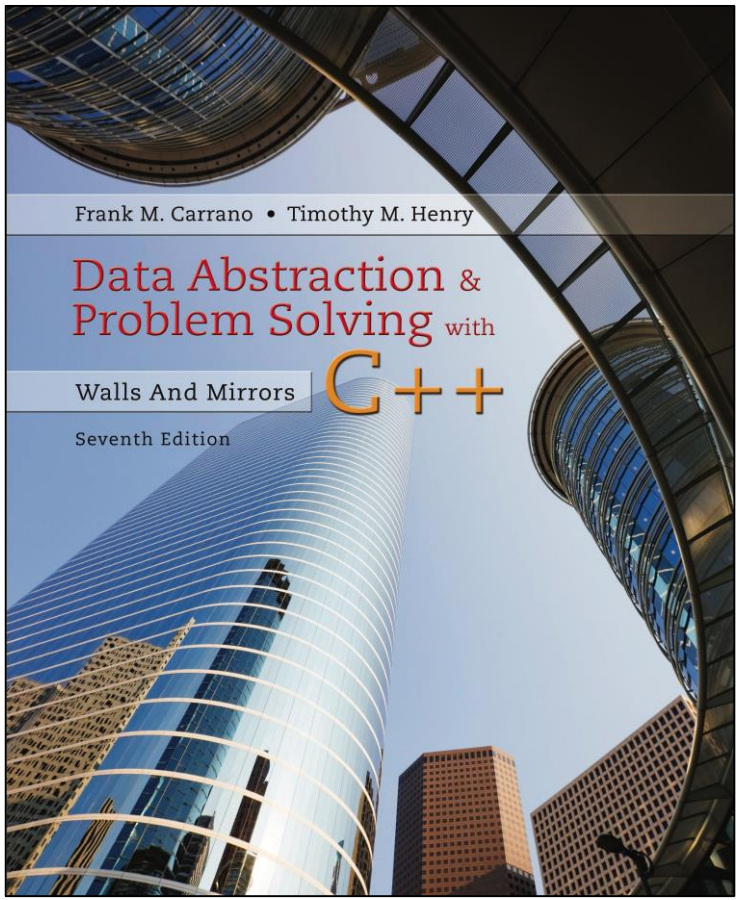
- Comparison of stack and queue operations
  - **isEmpty** for both
  - **pop** and **dequeue**
  - **peek** and **peekFront**

Position-Oriented and Value-Oriented ADTs (3 of 3)

- ADT list operations generalize stack and queue operations
  - **getLength**
  - **insert**
  - **remove**
  - **getEntry**

Data Abstraction & Problem Solving with C++: Walls and Mirrors

Seventh Edition



Chapter 14

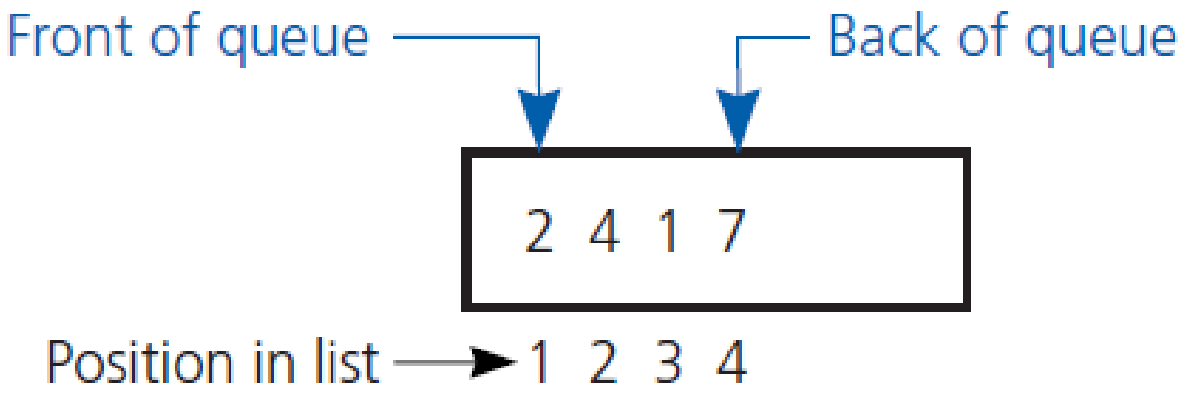
Queue and Priority Queue Implementations

Implementations of the ADT Queue

- Like stacks, queues can have
  - Array-based or
  - Link-based implementation.
- Can also use implementation of ADT list
  - Efficient to implement
  - Might not be most time efficient as possible

An Implementation That Uses the ADT List (1 of 6)

Figure 14-1 An implementation of the ADT queue that stores its entries in a list



An Implementation That Uses the ADT List (2 of 6)

Listing 14-1 The header file for the class **ListQueue**

```
1  /** ADT queue: ADT list implementation.
2   @file ListQueue.h */
3
4  #ifndef LIST_QUEUE_
5  #define LIST_QUEUE_
6
7  #include "QueueInterface.h"
8  #include "LinkedList.h"
9  #include "PrecondViolatedExcept.h"
10 #include <memory>
11
12 template<class ItemType>
13 class ListQueue : public QueueInterface<ItemType>
14 {
15 private:
```

An Implementation That Uses the ADT List (3 of 6)

Listing 14-1 [Continued]

```
16      std::unique_ptr<LinkedList<ItemType>> listPtr; // Pointer to list of queue items
17
18   public:
19      ListQueue();
20      ListQueue(const ListQueue& aQueue);
21      ~ListQueue();
22      bool isEmpty() const;
23      bool enqueue(const ItemType& newEntry);
24      bool dequeue();
25
26      /** @throw  PrecondViolatedExcept if this queue is empty. */
27      ItemType peekFront() const throw(PrecondViolatedExcept);
28   }; // end ListQueue
29   #include "ListQueue.cpp"
30   #endif
```

An Implementation That Uses the ADT List (4 of 6)

Listing 14-2 The implementation file for the class **ListQueue**

```
1  /** ADT queue: ADT list implementation.
2   @file ListQueue.cpp */
3  #include "ListQueue.h" // Header file
4  #include <memory>
5
6  template<class ItemType>
7  ListQueue<ItemType>::ListQueue()
8                    : listPtr(std::make_unique<LinkedList<ItemType>>())
9  {
10 }  // end default constructor
11
12 template<class ItemType>
13 ListQueue<ItemType>::ListQueue(const ListQueue& aQueue)
14                   : listPtr(aQueue.listPtr)
15 {
16 }  // end copy constructor
17
```

An Implementation That Uses the ADT List (5 of 6)

Listing 14-2 [Continued]

```
18  template<class ItemType>
19  ListQueue<ItemType>::~ListQueue()
20  {
21  }  // end destructor
22
23  template<class ItemType>
24  bool ListQueue<ItemType>::isEmpty() const
25  {
26     return listPtr->isEmpty();
27  }  // end isEmpty
28
29  template<class ItemType>
30  bool ListQueue<ItemType>::enqueue(const ItemType& newEntry)
31  {
32     return listPtr->insert(listPtr->getLength() + 1, newEntry);
33  }  // end enqueue
```

An Implementation That Uses the ADT List (6 of 6)

Listing 14-2 [Continued]

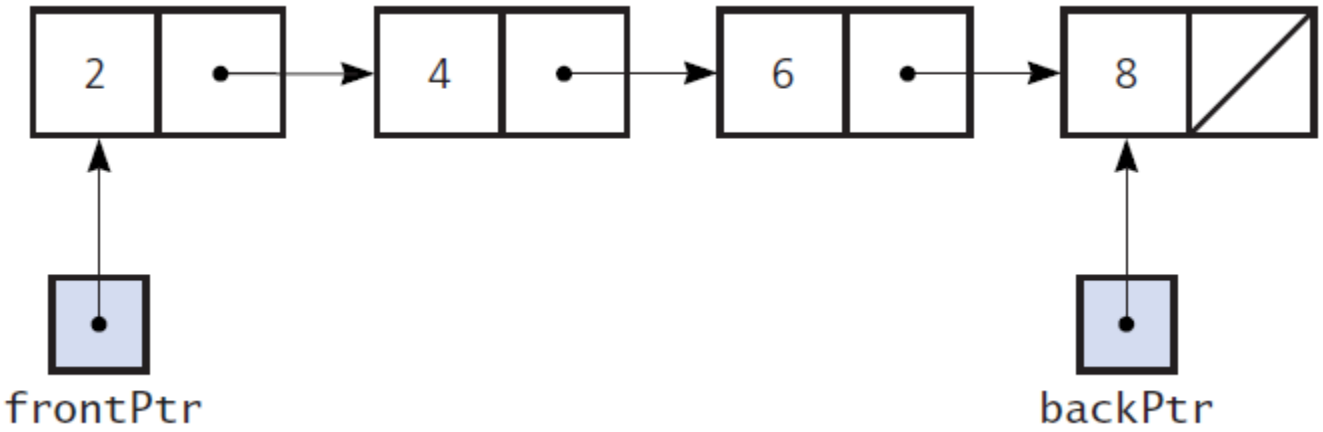
```
35  template<class ItemType>
36  bool ListQueue<ItemType>::dequeue()
37  {
38     return listPtr->remove(1);
39  }  // end dequeue
40
41  template<class ItemType>
42  ItemType ListQueue<ItemType>::peekFront() const throw(PrecondViolatedExcept)
43  {
44     if (isEmpty())
45        throw PrecondViolatedExcept("peekFront() called with empty queue.");
46
47     // Queue is not empty; return front
48     return listPtr->getEntry(1);
49  }  // end peekFront
50  // end of implementation file
```

A Link-Based Implementation (1 of 9)

- Similar to other link-based implementation
- One difference … Must be able to remove entries
  - From front
  - From back
- Requires a pointer to chain's last node
  - Called the "tail pointer"

A Link-Based Implementation (2 of 9)

Figure 14-2 A chain of linked nodes with head and tail pointers



A Link-Based Implementation (3 of 9)

Listing 14-3 The header file for the class **LinkedQueue**

```
/** ADT queue: Link-based implementation.
 @file LinkedQueue.h */

#ifndef LINKED_QUEUE_
#define LINKED_QUEUE_

#include "QueueInterface.h"
#include "Node.h"
#include "PrecondViolatedExcept.h"
#include <memory>

template<class ItemType>
class LinkedQueue : public QueueInterface<ItemType>
{
private:
   // The queue is implemented as a chain of linked nodes that has
   // two external pointers, a head pointer for the front of the queue
   // and a tail pointer for the back of the queue.
   std::shared_ptr<Node<ItemType>> frontPtr;
   std::shared_ptr<Node<ItemType>> backPtr;
```

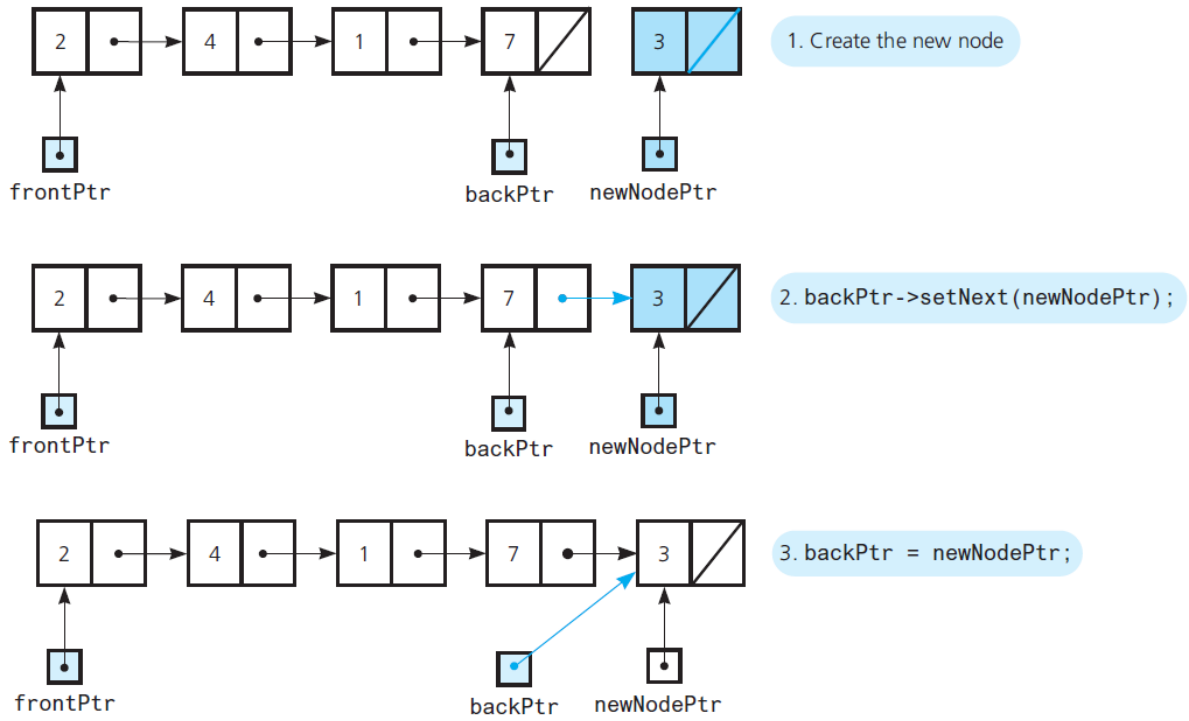
A Link-Based Implementation (4 of 9)

Listing 14-3 [Continued]

```
21
22  public:
23     LinkedQueue();
24     LinkedQueue(const LinkedQueue& aQueue);
25     ~LinkedQueue();
26
27     bool isEmpty() const;
28     bool enqueue(const ItemType& newEntry);
29     bool dequeue();
30
31     /** @throw  PrecondViolatedExcept if the queue is empty */
32     ItemType peekFront() const throw(PrecondViolatedExcept);
33  }; // end LinkedQueue
34  #include "LinkedQueue.cpp"
35  #endif
```

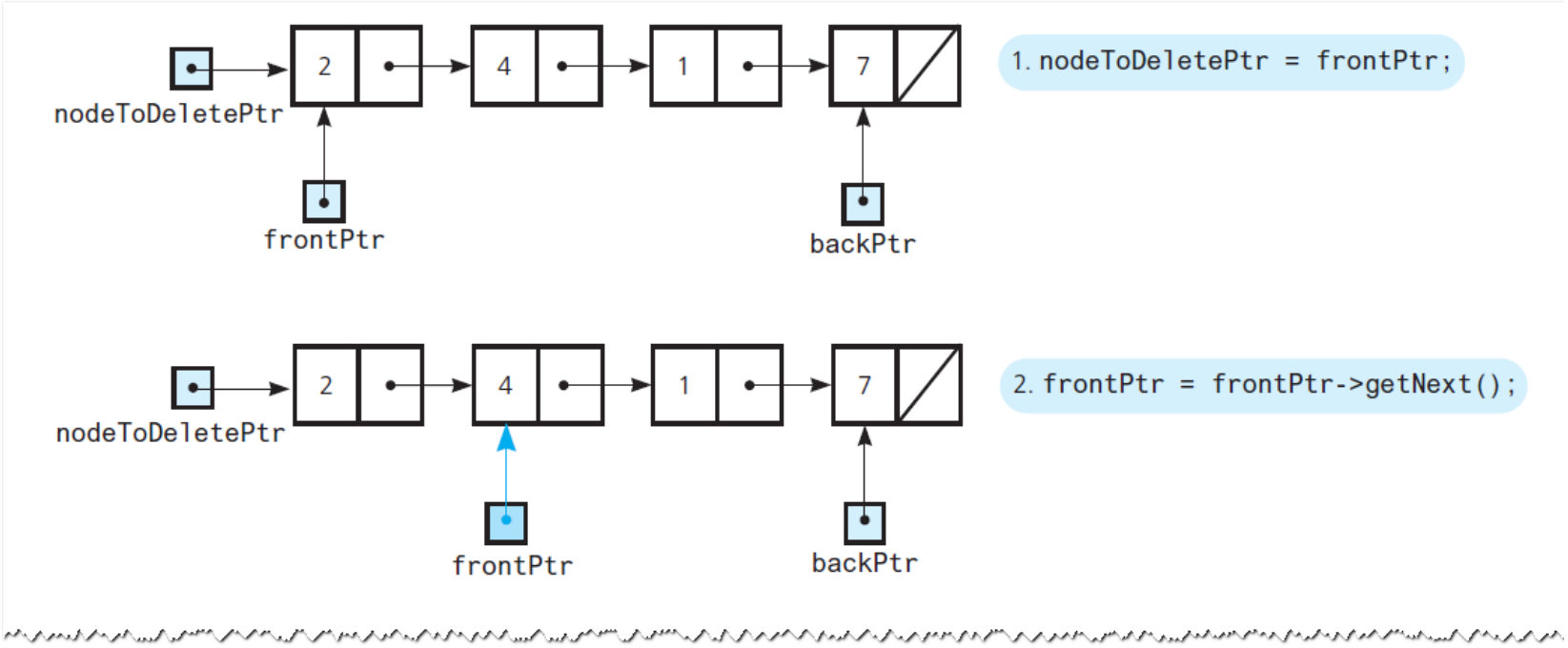
A Link-Based Implementation (5 of 9)

Figure 14-3 Adding an item to a nonempty queue



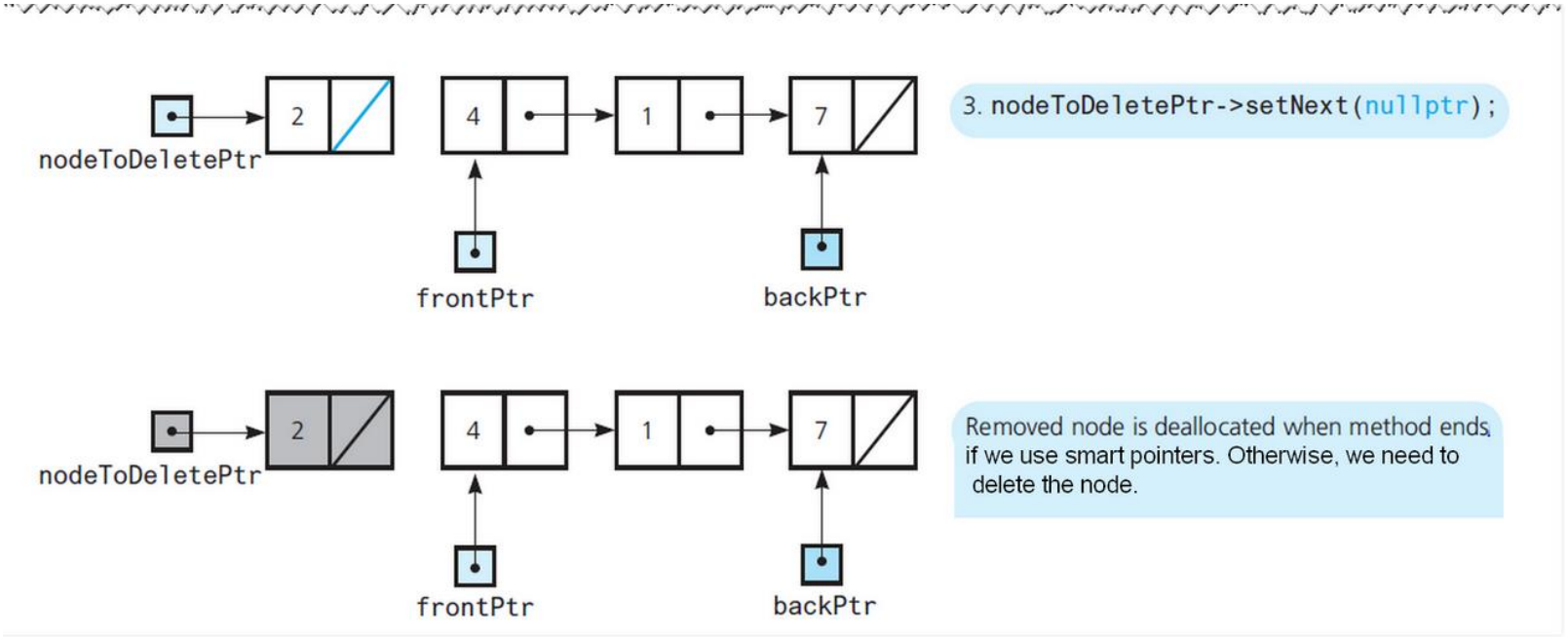
A Link-Based Implementation (6 of 9)

Figure 14-5 Removing an item from a queue of more than one item



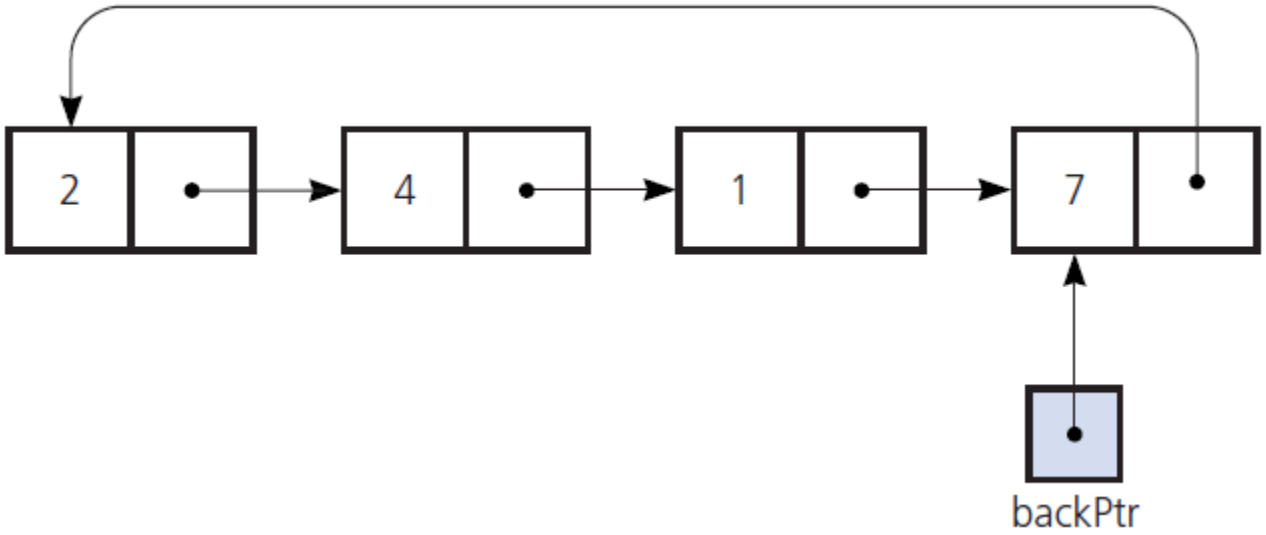
A Link-Based Implementation (8 of 9)

Figure 14-5 [Continued]



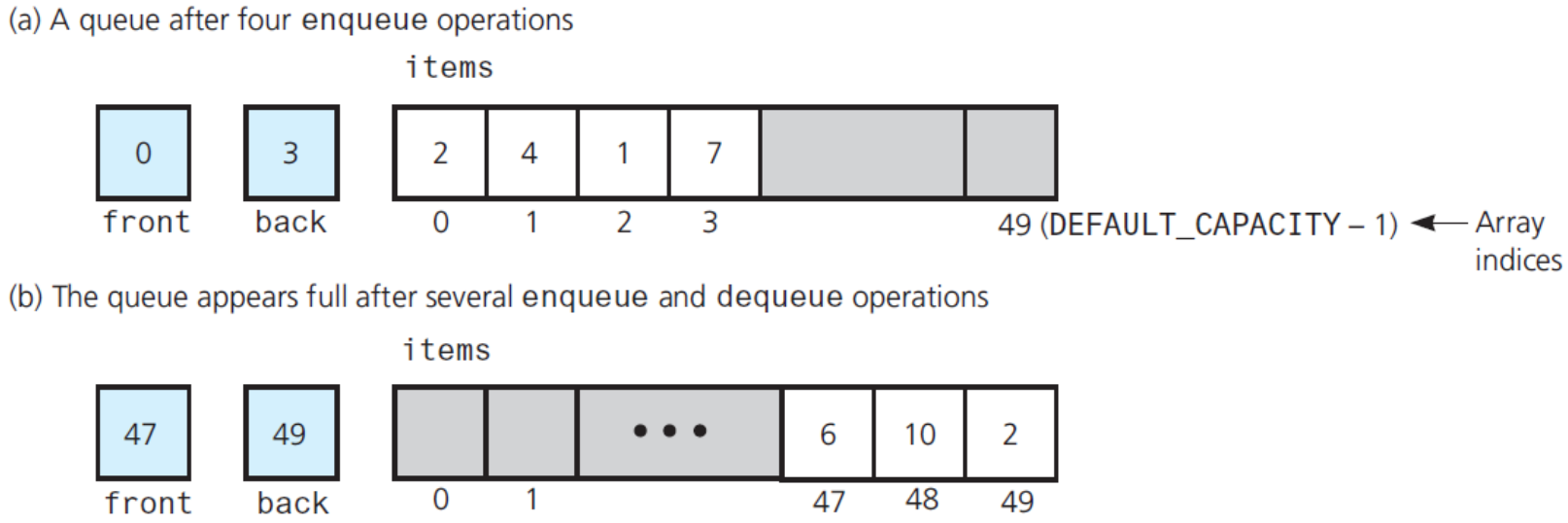
A Link-Based Implementation (9 of 9)

Figure 14-6 A circular chain of linked nodes with one external pointer



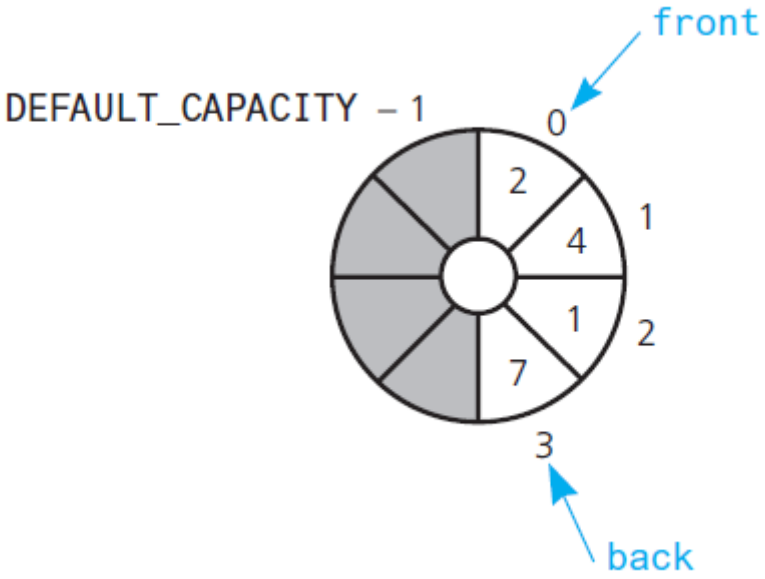
An Array-Based Implementation (1 of 11)

Figure 14-7 A naive array-based implementation of a queue for which rightward drift can cause the queue to appear full



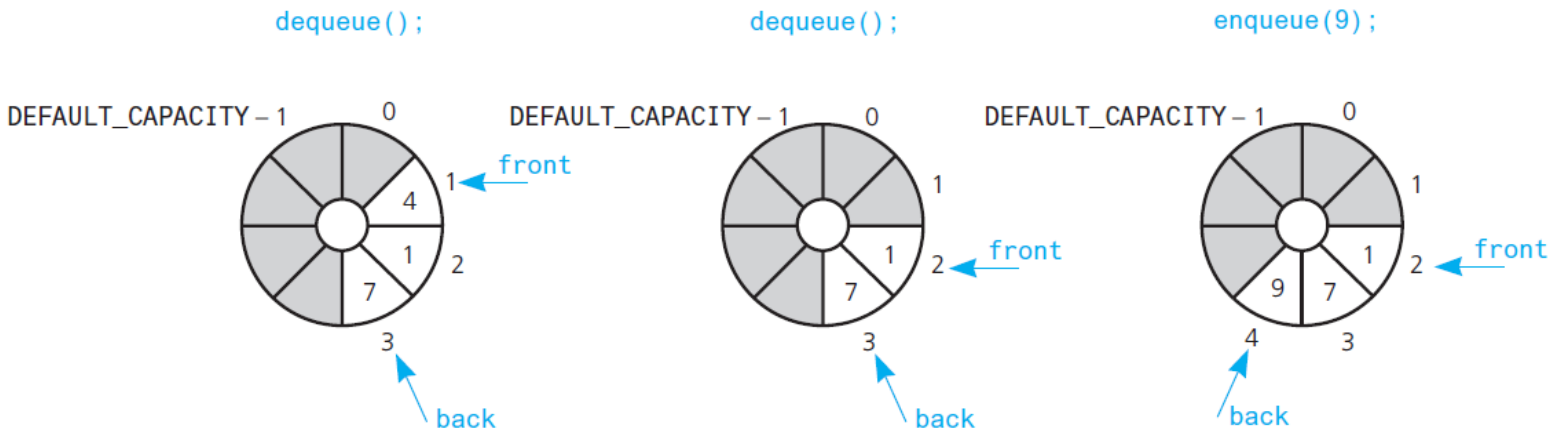
An Array-Based Implementation (2 of 11)

Figure 14-8 A circular array as an implementation of a queue



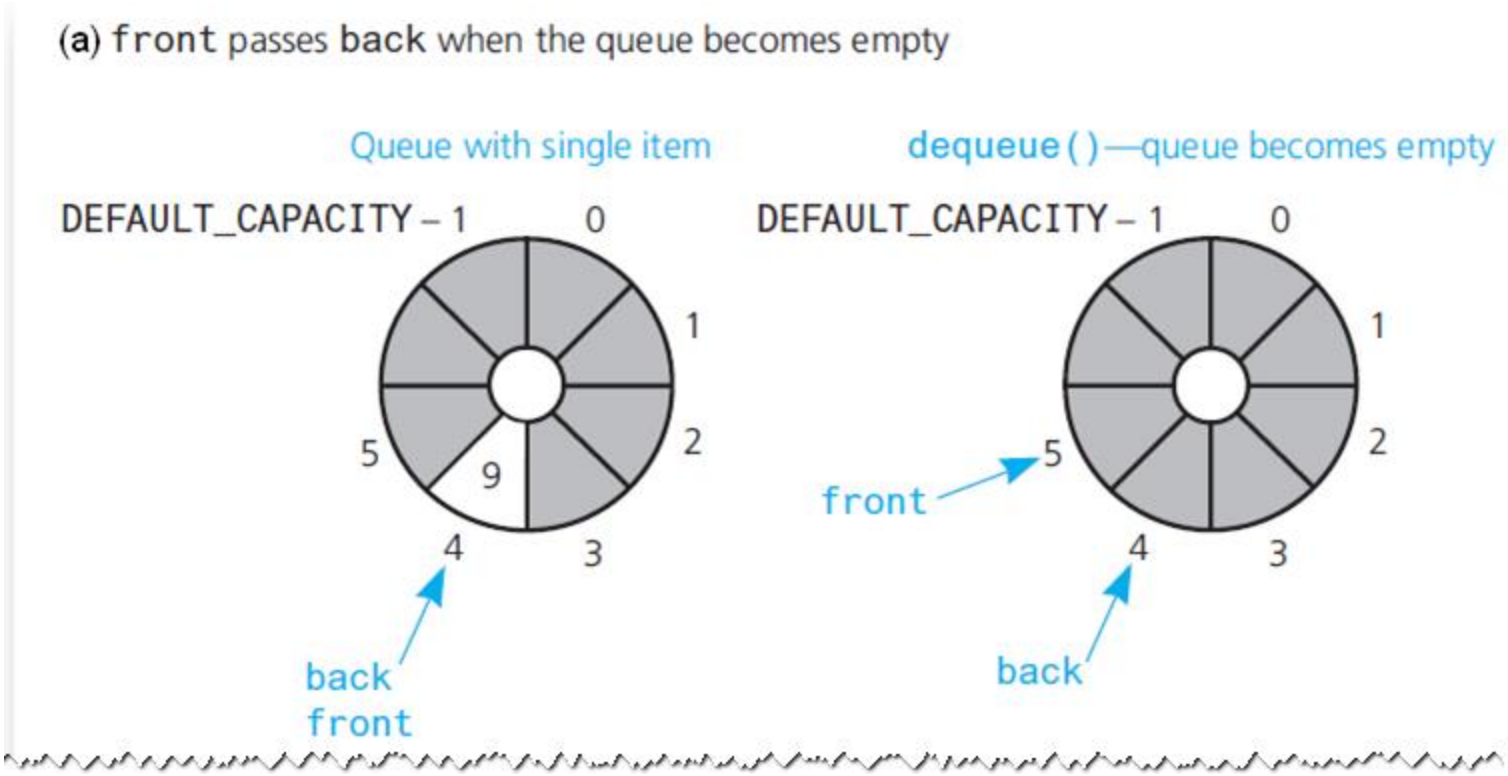
An Array-Based Implementation (3 of 11)

Figure 14-9 The effect of three consecutive operations on the queue in Figure 14-8



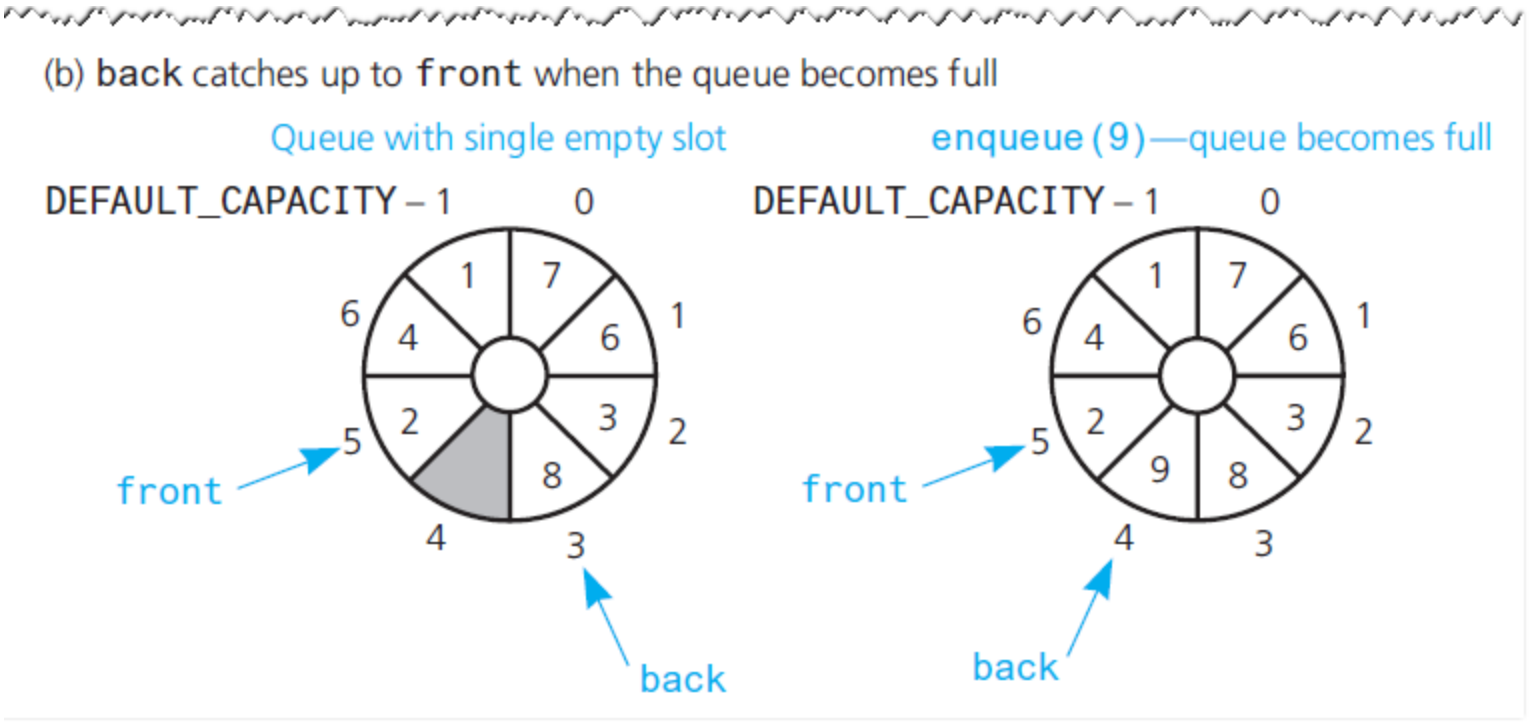
An Array-Based Implementation (4 of 11)

Figure 14-10 front and **back** as the queue becomes empty and as it becomes full



An Array-Based Implementation (5 of 11)

Figure 14-10 [Continued]



An Array-Based Implementation (6 of 11)

Listing 14-4 The header file for the class **ArrayQueue**

```
1  /** ADT queue: Circular array-based implementation.
2   @file ArrayQueue.h */
3  #ifndef ARRAY_QUEUE_
4  #define ARRAY_QUEUE_
5  #include "QueueInterface.h"
6  #include "PrecondViolatedExcept.h"
7
8  template<class ItemType>
9  class ArrayQueue : public QueueInterface<ItemType>
10 {
11 private:
12    static const int DEFAULAT_CAPACITY = 50;
13    ItemType items[DEFAULT_CAPACITY]; // Array of queue items
14    int      front;                   // Index to front of queue
15    int      back;                    // Index to back of queue
16    int      count;                   // Number of items currently in the queue
17
```

An Array-Based Implementation (7 of 11)

Listing 14-5 The implementation file for the class **ArrayQueue**

```
1  /** ADT queue: Circular array-based implementation.
2   @file ArrayQueue.cpp */
3  #include "ArrayQueue.h" // Header file
4
5  template<class ItemType>
6  ArrayQueue<ItemType>::ArrayQueue()
7                      : front(0), back(DEFAULT_CAPACITY - 1), count(0)
8  {
9  }  // end default constructor
10
11 template<class ItemType>
12 bool ArrayQueue<ItemType>::isEmpty() const
13 {
14    return count == 0;
15 }  // end isEmpty
16
```

An Array-Based Implementation (8 of 11)

Listing 14-5 [Continued]

```
16
17  template<class ItemType>
18  bool ArrayQueue<ItemType>::enqueue(const ItemType& newEntry)
19  {
20     bool result = false;
21     if (count < DEFAULT_CAPACITY)
22     {
23        // Queue has room for another item
24        back = (back + 1) % DEFAULT_CAPACITY;
25        items[back] = newEntry;
26        count++;
27        result = true;
28     } // end if
29
30     return result;
31  } // end enqueue
32
```

An Array-Based Implementation (9 of 11)

Listing 14-5 [Continued]

```
33  template<class ItemType>
34  bool ArrayQueue<ItemType>::dequeue()
35  {
36     bool result = false;
37     if (!isEmpty())
38     {
39        front = (front + 1) % DEFAULT_CAPACITY;
40        count--;
41        result = true;
42     }  // end if
43
44     return result;
45  }  // end dequeue
46
```

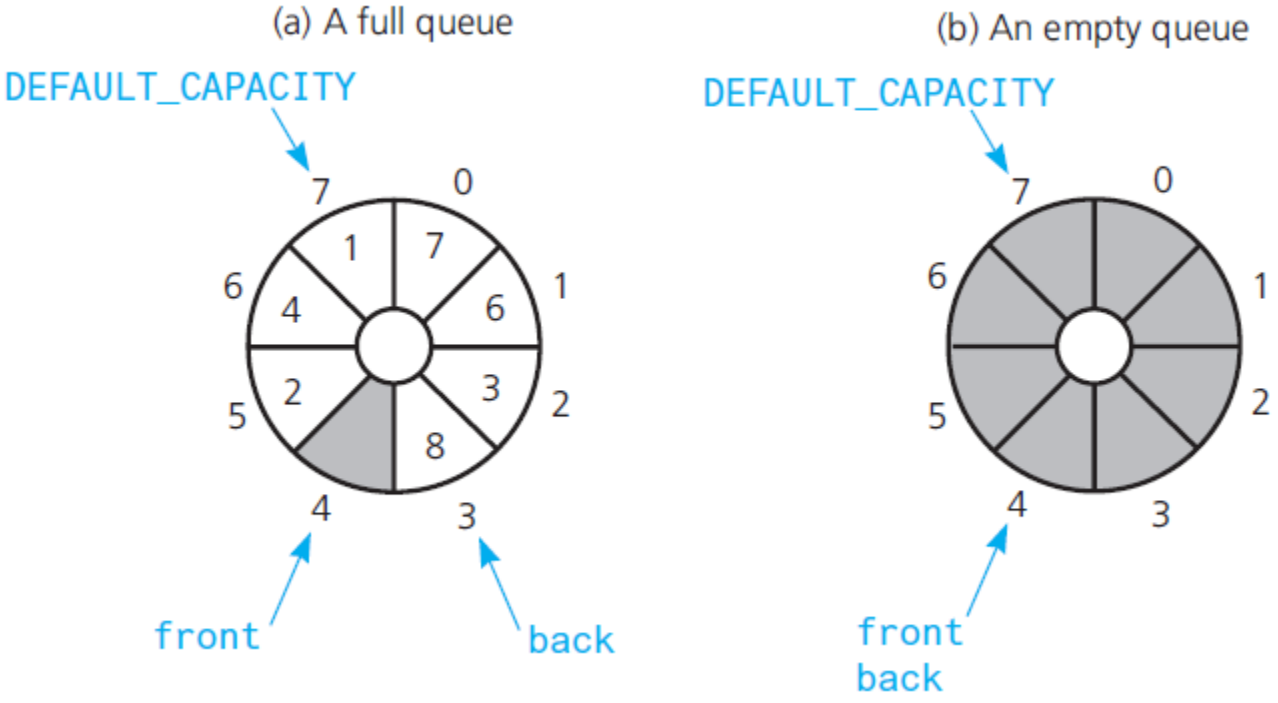
An Array-Based Implementation (10 of 11)

Listing 14-5 [Continued]

```
46
47  template<class ItemType>
48  ItemType ArrayQueue<ItemType>::peekFront() const throw(PrecondViolatedExcept)
49  {
50     // Enforce precondition
51     if (isEmpty())
52        throw PrecondViolatedExcept("peekFront() called with empty queue");
53
54     // Queue is not empty; return front
55     return items[front];
56  }  // end peekFront
57  // End of implementation file.
```

An Array-Based Implementation (11 of 11)

Figure 14-11 A circular array having one unused location as an implementation of a queue



Comparing Implementations

- Issues
  - Fixed size (array-based) versus dynamic size (link-based)
  - Reuse of already implemented class saves time

An Implementation of the ADT Priority Queue (1 of 2)

Listing 14-6 A header file for the class **SL_PriorityQueue.**

```
1  /** ADT priority queue: ADT sorted list implementation.
2   @file SL_PriorityQueue.h */
3  #ifndef PRIORITY_QUEUE_
4  #define PRIORITY_QUEUE_
5
6  #include "PriorityQueueInterface.h"
7  #include "LinkedSortedList.h"
8  #include "PrecondViolatedExcept.h"
9  #include <memory>
10
11 template<class ItemType>
12 class SL_PriorityQueue : public PriorityQueueInterface<ItemType>
13 {
14 private:
15    std::unique_ptr<LinkedSortedList<ItemType>> slistPtr; // Ptr to sorted list
16                                                          // of items
17
```

An Implementation of the ADT Priority Queue (2 of 2)

Listing 14-6 [Continued]

```
16
17
18  public:
19     SL_PriorityQueue();
20     SL_PriorityQueue(const SL_PriorityQueue& pq);
21     ~SL_PriorityQueue();
22
23     bool isEmpty() const;
24     bool enqueue(const ItemType& newEntry);
25     bool dequeue();
26
27     /** @throw  PrecondViolatedExcept if priority queue is empty. */
28     ItemType peekFront() const throw(PrecondViolatedExcept);
29  }; // end SL_PriorityQueue
30  #include "SL_PriorityQueue.cpp"
31  #endif
```

Copyright