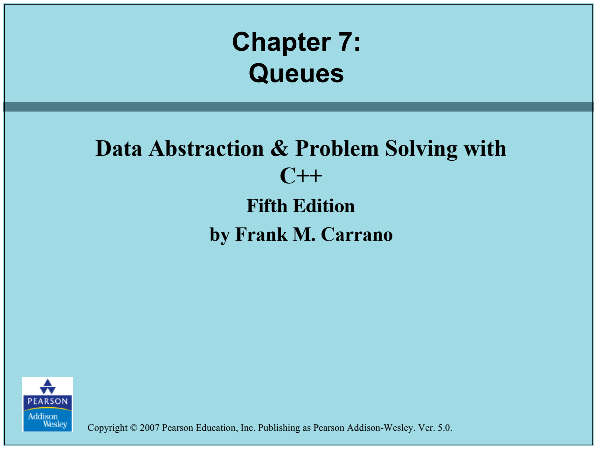
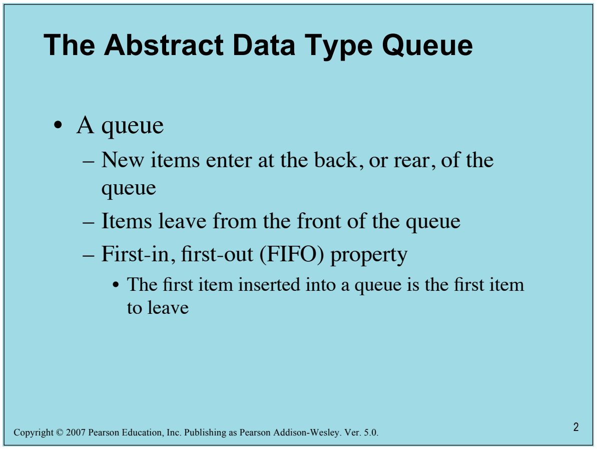


Chapter 7: Queues

Data Abstraction & Problem Solving with C++

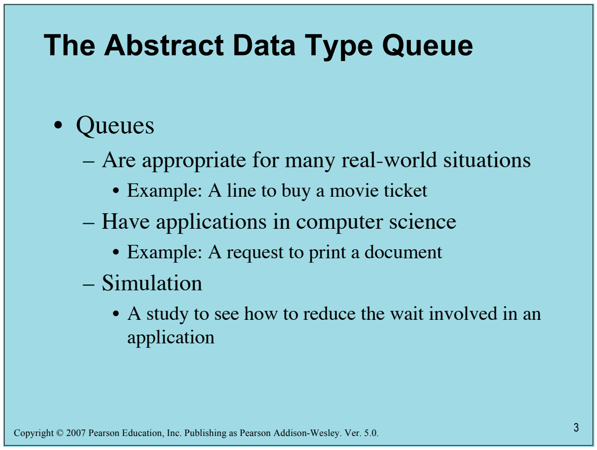
Fifth Edition

by Frank M. Carrano



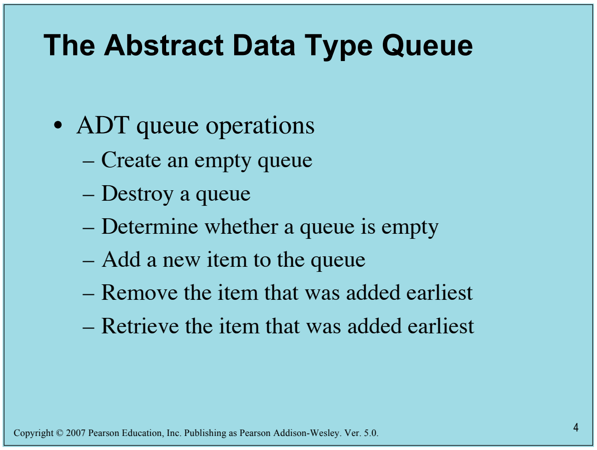
The Abstract Data Type Queue

- A queue
  - New items enter at the back, or rear, of the queue
  - Items leave from the front of the queue
  - First-in, first-out (FIFO) property
    - The first item inserted into a queue is the first item to leave



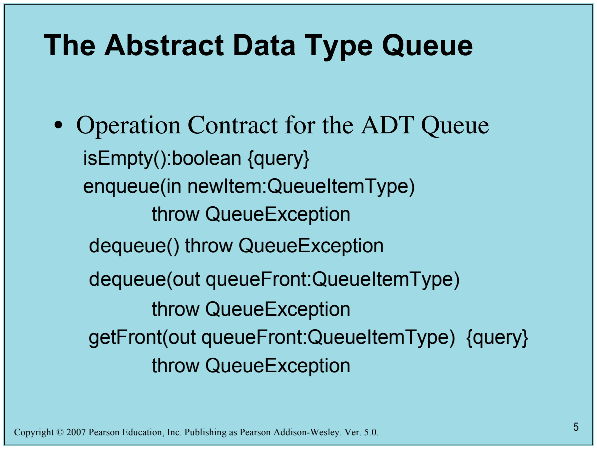
The Abstract Data Type Queue

- Queues
  - Are appropriate for many real-world situations
    - Example: A line to buy a movie ticket
  - Have applications in computer science
    - Example: A request to print a document
  - Simulation
    - A study to see how to reduce the wait involved in an application



The Abstract Data Type Queue

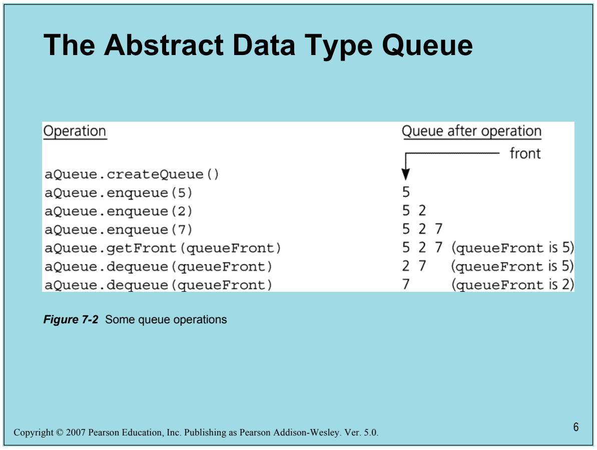
- ADT queue operations
  - Create an empty queue
  - Destroy a queue
  - Determine whether a queue is empty
  - Add a new item to the queue
  - Remove the item that was added earliest
  - Retrieve the item that was added earliest



The Abstract Data Type Queue

- Operation Contract for the ADT Queue

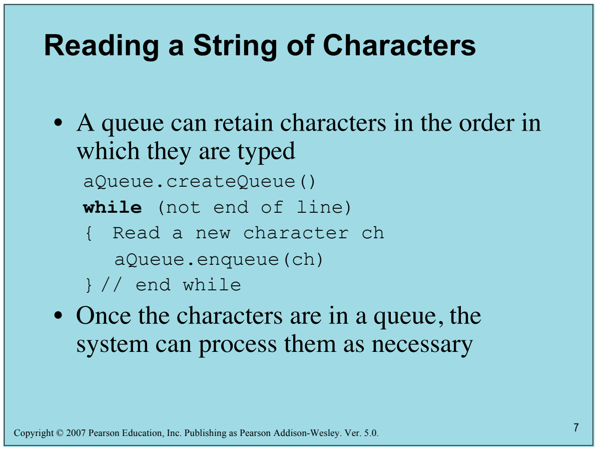
```
isEmpty():boolean {query}
enqueue(in newItem:QueueItemType)
        throw QueueException
dequeue() throw QueueException
dequeue(out queueFront:QueueItemType)
        throw QueueException
getFront(out queueFront:QueueItemType)  {query}
        throw QueueException
```



The Abstract Data Type Queue

```
Operation                          Queue after operation
                                   ┌──────────── front
aQueue.createQueue()               ▼
aQueue.enqueue(5)                  5
aQueue.enqueue(2)                  5 2
aQueue.enqueue(7)                  5 2 7
aQueue.getFront(queueFront)        5 2 7 (queueFront is 5)
aQueue.dequeue(queueFront)         2 7   (queueFront is 5)
aQueue.dequeue(queueFront)         7     (queueFront is 2)
```

Figure 7-2 Some queue operations

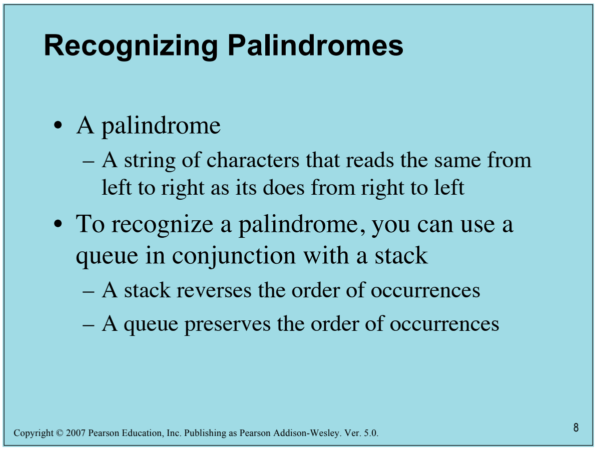


Reading a String of Characters

- A queue can retain characters in the order in which they are typed

```
aQueue.createQueue()
while (not end of line)
{  Read a new character ch
   aQueue.enqueue(ch)
} // end while
```

- Once the characters are in a queue, the system can process them as necessary



Recognizing Palindromes

- A palindrome
  - A string of characters that reads the same from left to right as its does from right to left
- To recognize a palindrome, you can use a queue in conjunction with a stack
  - A stack reverses the order of occurrences
  - A queue preserves the order of occurrences

Recognizing Palindromes

- A nonrecursive recognition algorithm for palindromes
  - As you traverse the character string from left to right, insert each character into both a queue and a stack
  - Compare the characters at the front of the queue and the top of the stack

Recognizing Palindromes

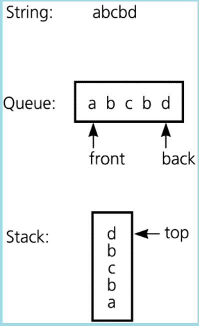


Figure 7-3
The results of inserting a string into both a queue and a stack

Implementations of the ADT Queue

- An array-based implementation
- Possible implementations of a pointer-based queue
  - A linear linked list with two external references
    - A reference to the front
    - A reference to the back
  - A circular linked list with one external reference
    - A reference to the back

A Pointer-Based Implementation

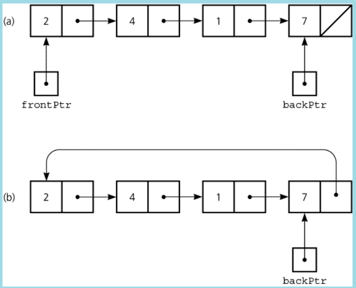
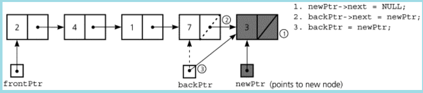


Figure 7-4 A pointer-based implementation of a queue: (a) a linear linked list with two external pointers; (b) a circular linear linked list with one external pointer

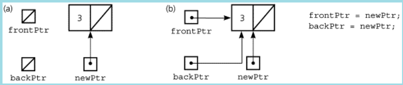
A Pointer-Based Implementation

Figure 7-5 Inserting an item into a nonempty queue



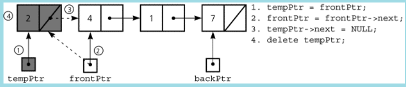
```
1. newPtr->next = NULL;
2. backPtr->next = newPtr;
3. backPtr = newPtr;
```

Figure 7-6 Inserting an item into an empty queue: (a) before insertion; (b) after insertion



```
frontPtr = newPtr;
backPtr = newPtr;
```

Figure 7-7 Deleting an item from a queue of more than one item



```
1. tempPtr = frontPtr;
2. frontPtr = frontPtr->next;
3. tempPtr->next = NULL;
4. delete tempPtr;
```

An Array-Based Implementation

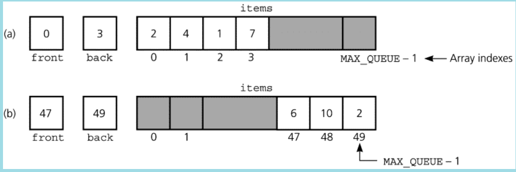


Figure 7-8

a) A naive array-based implementation of a queue; (b) rightward drift can cause the queue to appear full

An Array-Based Implementation

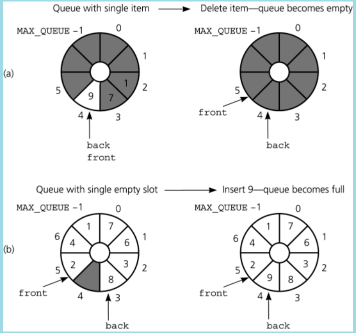
- A circular array
  - Eliminates the problem of rightward drift
  - BUT `front` and `back` cannot be used to distinguish between queue-full and queue-empty conditions

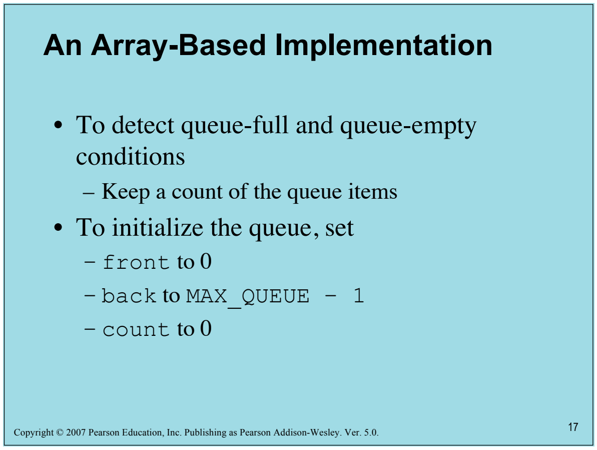
An Array-Based Implementation

Figure 7-11

(a) *front* passes *back* when the queue becomes empty;

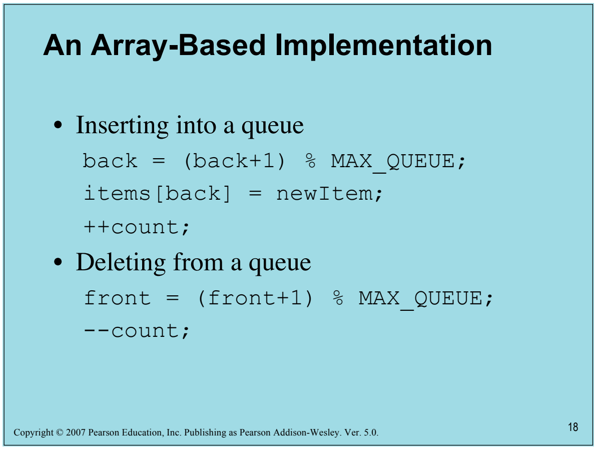
(b) *back* catches up to *front* when the queue becomes full





An Array-Based Implementation

- To detect queue-full and queue-empty conditions
  - Keep a count of the queue items
- To initialize the queue, set
  - `front` to 0
  - `back` to `MAX_QUEUE - 1`
  - `count` to 0



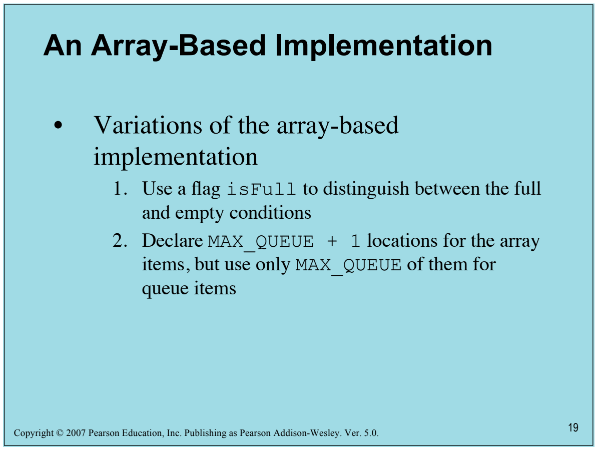
An Array-Based Implementation

- Inserting into a queue

```
back = (back+1) % MAX_QUEUE;
items[back] = newItem;
++count;
```

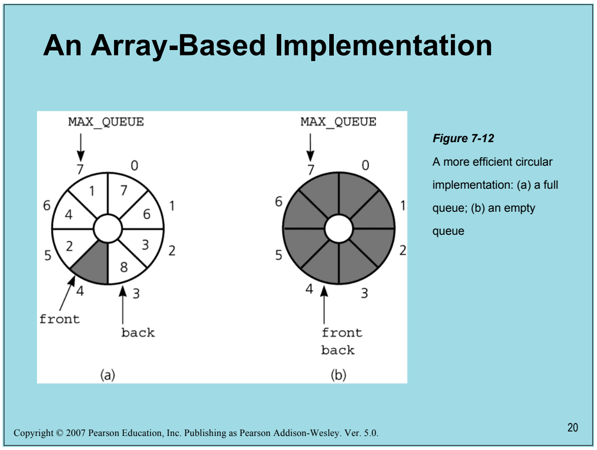
- Deleting from a queue

```
front = (front+1) % MAX_QUEUE;
--count;
```



An Array-Based Implementation

- Variations of the array-based implementation
  1. Use a flag `isFull` to distinguish between the full and empty conditions
  2. Declare `MAX_QUEUE + 1` locations for the array items, but use only `MAX_QUEUE` of them for queue items



An Array-Based Implementation

Figure 7-12

A more efficient circular implementation: (a) a full queue; (b) an empty queue

An Implementation That Uses the ADT List

The front of the queue is at position 1 of the list;
The back of the queue is at the end of the list

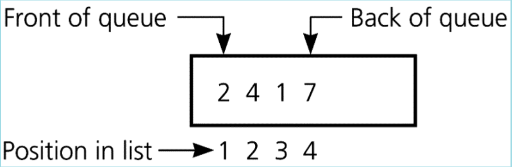


Figure 7-13
An implementation that uses the ADT list

An Implementation That Uses the ADT List

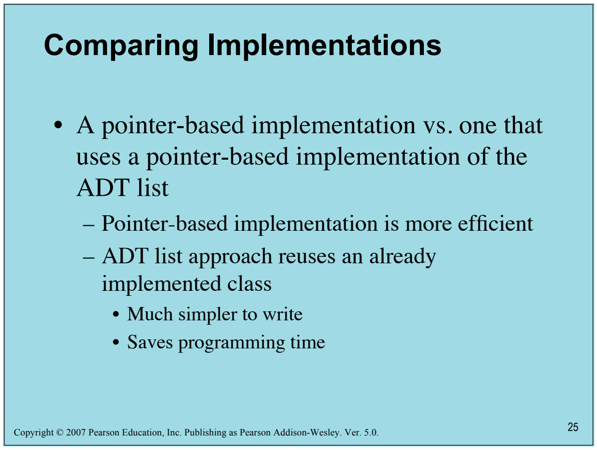
- `aList enqueue()`
  ```
  aList.insert(aList.getLength()+1,
               newItem)
  ```
- `dequeue()`
  ```
  aList.remove(1)
  ```
- `getFront(queueFront)`
  ```
  aList.retrieve(1, queueFront)
  ```

The Standard Template Library Class *queue*

- Some operations in the STL queue
  - Enqueue and dequeue operations are called `push` and `pop`, respectively, as for a stack
  - The `back` method returns a reference to the last item
  - The `size` method returns the number of items
- An adaptor container
  - Implemented using a more basic container type
  - The default queue container class is `deque`

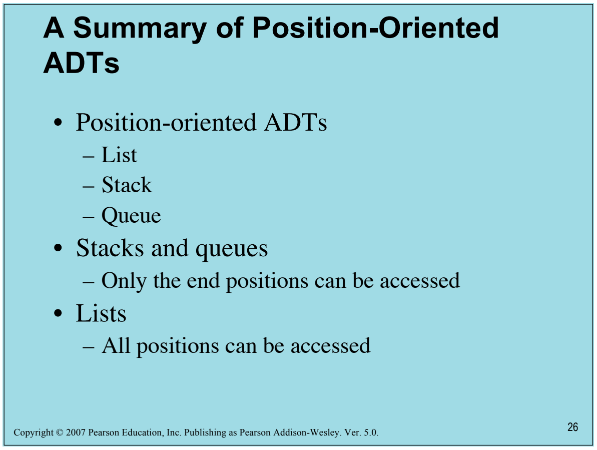
Comparing Implementations

- Fixed size versus dynamic size
  - A statically allocated array-based implementation
    - Fixed-size queue that can get full
    - Prevents the `enqueue` operation from adding an item to the queue, if the array is full
  - A dynamically allocated array-based implementation or a pointer-based implementation
    - No size restriction on the queue



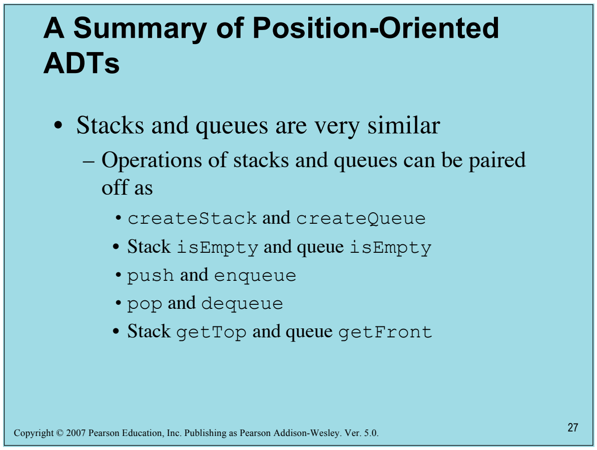
Comparing Implementations

- A pointer-based implementation vs. one that uses a pointer-based implementation of the ADT list
  - Pointer-based implementation is more efficient
  - ADT list approach reuses an already implemented class
    - Much simpler to write
    - Saves programming time



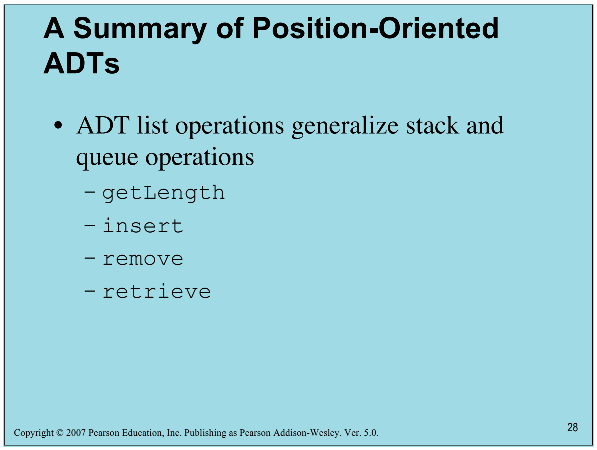
A Summary of Position-Oriented ADTs

- Position-oriented ADTs
  - List
  - Stack
  - Queue
- Stacks and queues
  - Only the end positions can be accessed
- Lists
  - All positions can be accessed



A Summary of Position-Oriented ADTs

- Stacks and queues are very similar
  - Operations of stacks and queues can be paired off as
    - `createStack` and `createQueue`
    - Stack `isEmpty` and queue `isEmpty`
    - `push` and `enqueue`
    - `pop` and `dequeue`
    - Stack `getTop` and queue `getFront`



A Summary of Position-Oriented ADTs

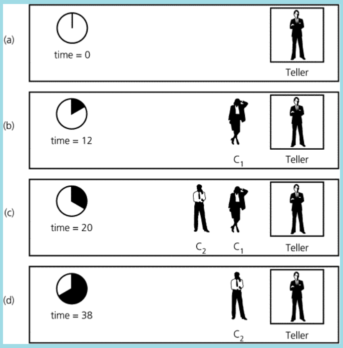
- ADT list operations generalize stack and queue operations
  - `getLength`
  - `insert`
  - `remove`
  - `retrieve`

Application: Simulation

- Simulation
  - – A technique for modeling the behavior of both natural and human-made systems
  - – Goal
    - Generate statistics that summarize the performance of an existing system
    - Predict the performance of a proposed system
  - – Example
    - A simulation of the behavior of a bank

Application: Simulation

Figure 7-14 A bank line at time (a) 0; (b) 12; (c) 20; (d) 38



Application: Simulation

- A time-driven simulation
  - – Simulated time advances by one time unit
  - – The time of an event is determined randomly and compared with the simulated time

Application: Simulation

- An event-driven simulation
  - – Simulated time advances to time of next event
  - – Events are generated by using a mathematical model based on statistics and probability

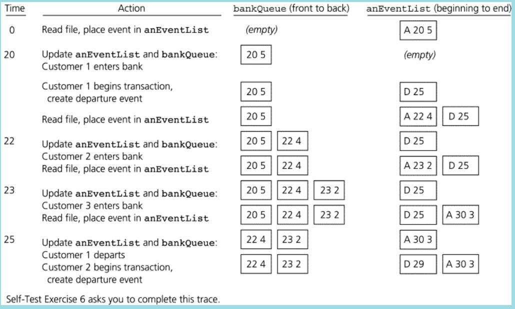
Application: Simulation

- The bank simulation is concerned with
  - Arrival events
    - External events: the input file specifies the times at which the arrival events occur
  - Departure events
    - Internal events: the simulation determines the times at which the departure events occur

Application: Simulation

- Bank simulation is event-driven and uses an event list
  - Keeps track of arrival and departure events that will occur but have not occurred yet
  - Contains at most one arrival event and one departure event

Application: Simulation



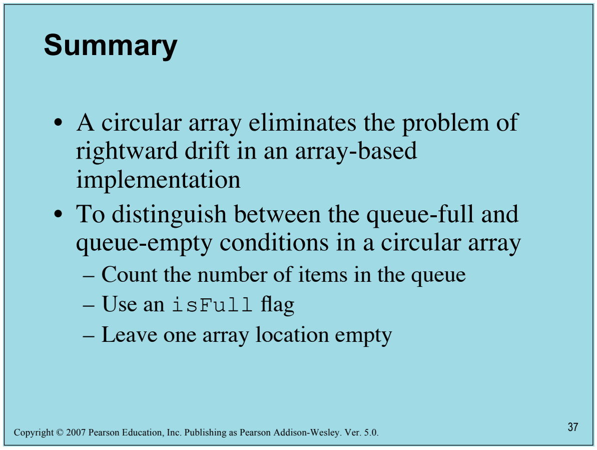
Time	Action	bankQueue (front to back)			anEventList (beginning to end)	
0	Read file, place event in **anEventList**	*(empty)*			A 20 5	
20	Update **anEventList** and **bankQueue**: Customer 1 enters bank	20 5			*(empty)*	
	Customer 1 begins transaction, create departure event	20 5			D 25	
	Read file, place event in **anEventList**	20 5			A 22 4	D 25
22	Update **anEventList** and **bankQueue**: Customer 2 enters bank	20 5	22 4		D 25	
	Read file, place event in **anEventList**	20 5	22 4		A 23 2	D 25
23	Update **anEventList** and **bankQueue**: Customer 3 enters bank	20 5	22 4	23 2	D 25	
	Read file, place event in **anEventList**	20 5	22 4	23 2	D 25	A 30 3
25	Update **anEventList** and **bankQueue**: Customer 1 departs	22 4	23 2		A 30 3	
	Customer 2 begins transaction, create departure event	22 4	23 2		D 29	A 30 3

Self-Test Exercise 6 asks you to complete this trace.

Figure 7-16 A partial trace of the bank simulation for the data 20 5, 22 4, 23 2, 30 3

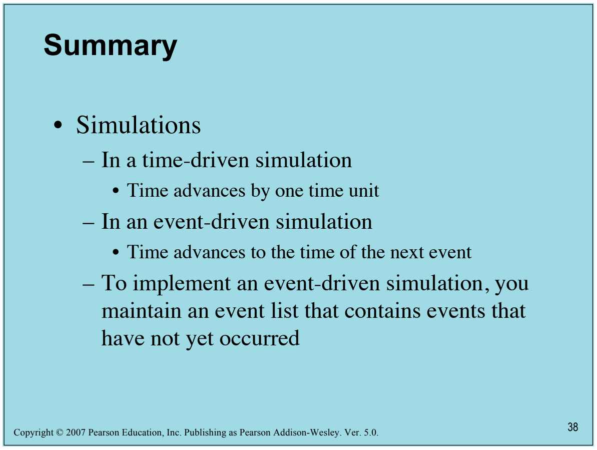
Summary

- The definition of the queue operations gives the ADT queue first-in, first-out (FIFO) behavior
- A pointer-based implementation of a queue uses either
  - A circular linked list
  - A linear linked list with both a head pointer and a tail pointer



Summary

- A circular array eliminates the problem of rightward drift in an array-based implementation
- To distinguish between the queue-full and queue-empty conditions in a circular array
  - Count the number of items in the queue
  - Use an `isFull` flag
  - Leave one array location empty



Summary

- Simulations
  - In a time-driven simulation
    - Time advances by one time unit
  - In an event-driven simulation
    - Time advances to the time of the next event
  - To implement an event-driven simulation, you maintain an event list that contains events that have not yet occurred