
Cmpt-225

Queues

Queues

- A queue is a data structure that only allows items to be inserted at the end and removed from the front
 - Queues are **FIFO** (First In First Out) data structures
 - “fair” data structures
-

What Can You Use a Queue For?

- Processing inputs and outputs to screen (console)
 - Server requests
 - Instant messaging servers queue up incoming messages
 - Database requests
 - Print queues
 - One printer for dozens of computers
 - Operating systems use queues to schedule CPU jobs
 - Simulations
-

Queue Operations

- A queue should implement (at least) these operations:
 - **enqueue** – insert an item at the back of the queue
 - **dequeue** – remove an item from the front
 - **peek** – return the item at the front of the queue without removing it
 - Like stacks it is assumed that these operations will be implemented efficiently
 - That is, in constant time
-

Queue: Array Implementation

- First consider using an array as the underlying structure for a queue, one plan would be to
 - Make the back of the queue the current size of the queue (i.e., the number of elements stored)
 - Make the front of the queue index 0
 - Inserting an item can be performed in constant time
 - But removing an item would require shifting all elements in the queue to the left which is **too slow!**
 - **Therefore we need to find another way**
-

An Array-Based Implementation

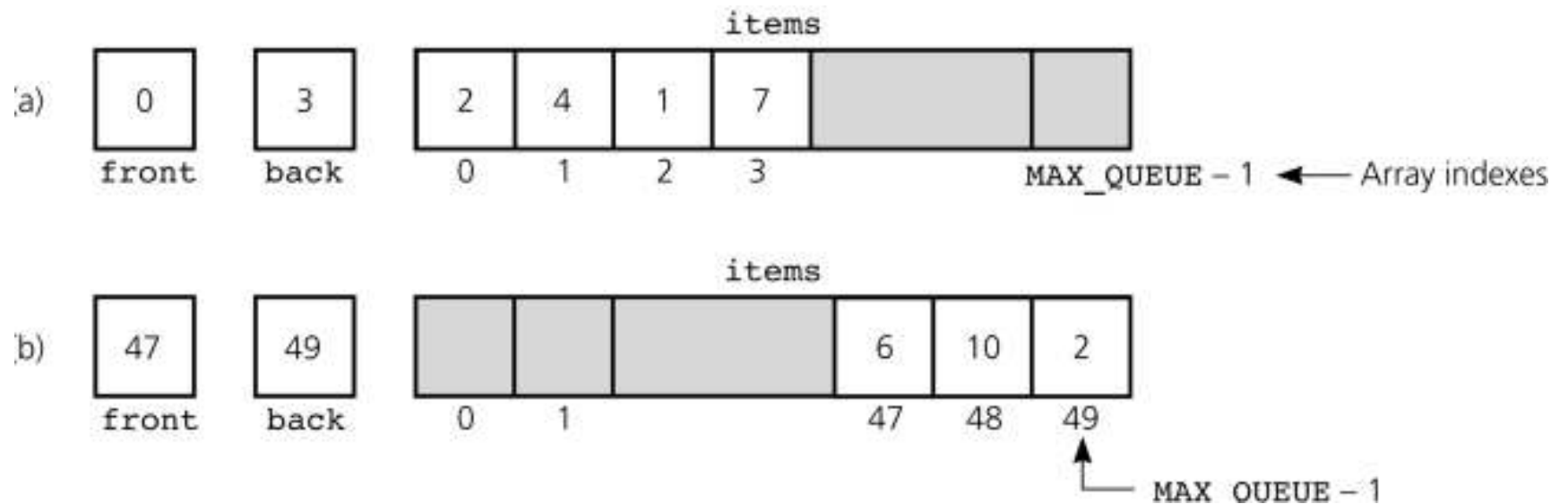
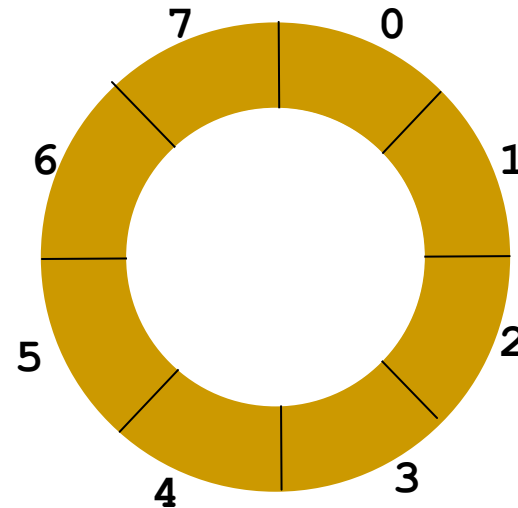


Figure 8-8

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

Circular Arrays

- **Neat trick:** use a ***circular array*** to insert and remove items from a queue in constant time
- The idea of a circular array is that the end of the array “wraps around” to the start of the array



The mod Operator

- The `mod` operator (%) is used to calculate remainders:
 - $1\%5 = 1$, $2\%5 = 2$, $5\%5 = 0$, $8\%5 = 3$
 - `mod` can be used to calculate the front and back positions in a circular array, therefore avoiding comparisons to the array size
 - The back of the queue is:
 - `(front + count - 1) % items.length`
 - where `count` is the number of items currently in the queue
 - After removing an item the front of the queue is:
 - `(front + 1) % items.length;`
-

Array Queue Example

front =	0
count =	1

6					
0	1	2	3	4	5

```
//Java Code
```

```
Queue q = new Queue();  
q.enqueue(6);
```

insert item at $(\text{front} + \text{count}) \% \text{items.length}$

Array Queue Example

front =	0
count =	5

6	4	7	3	8	
0	1	2	3	4	5

//Java Code

```
Queue q = new Queue();  
q.enqueue(6);  
q.enqueue(4);  
q.enqueue(7);  
q.enqueue(3);  
q.enqueue(8);
```

Array Queue Example

front =	2
count =	4

6	4	7	3	8	9
0	1	2	3	4	5

make front = $(0 + 1) \% 6 = 1$

make front = $(1 + 1) \% 6 = 2$

//Java Code

```
Queue q = new Queue();
```

```
q.enqueue(6);
```

```
q.enqueue(4);
```

```
q.enqueue(7);
```

```
q.enqueue(3);
```

```
q.enqueue(8);
```

```
q.dequeue(); //front = 1
```

```
q.dequeue(); //front = 2
```

```
q.enqueue(9);
```

Array Queue Example

front =	2
count =	5

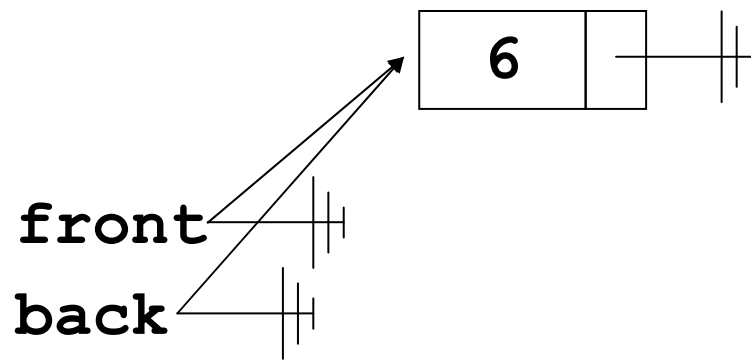
5		7	3	8	9
0	1	2	3	4	5

**insert at $(\text{front} + \text{count}) \% 6$
 $= (2 + 4) \% 6 = 0$**

//Java Code

```
Queue q = new Queue();  
q.enqueue(6);  
q.enqueue(4);  
q.enqueue(7);  
q.enqueue(3);  
q.enqueue(8);  
q.dequeue(); //front = 1  
q.dequeue(); //front = 2  
q.enqueue(9);  
q.enqueue(5);
```

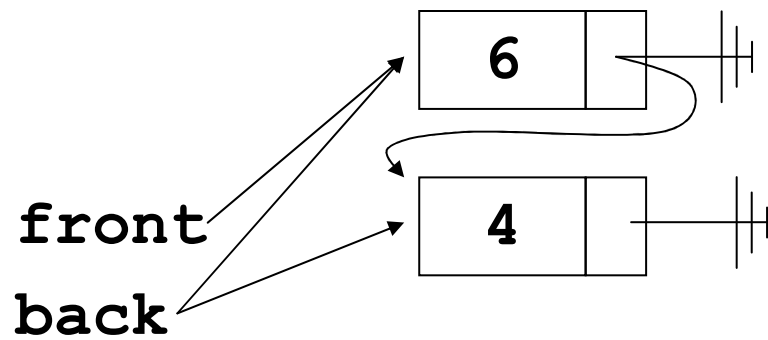
List Queue Example



//Java Code

```
Queue q = new Queue();  
q.enqueue(6);
```

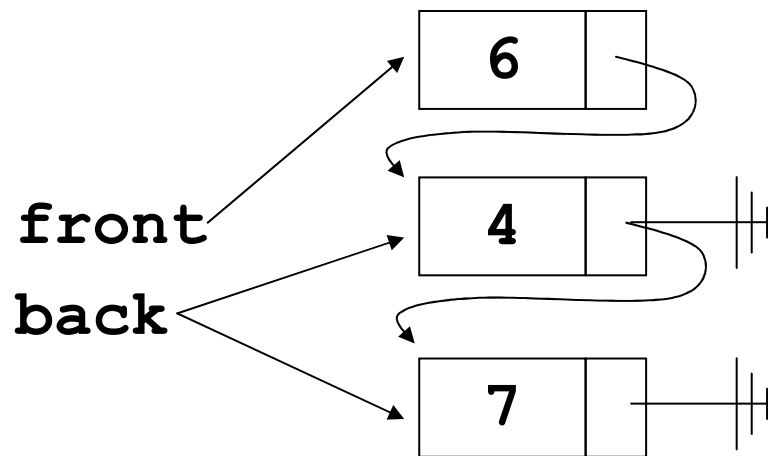
List Queue Example



//Java Code

```
Queue q = new Queue();  
q.enqueue(6);  
q.enqueue(4);
```

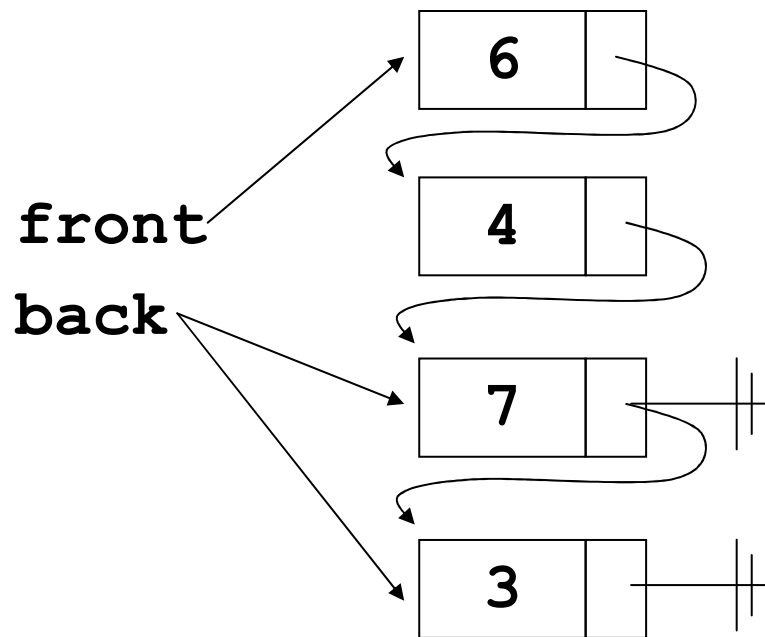
List Queue Example



//Java Code

```
Queue q = new Queue();  
q.enqueue(6);  
q.enqueue(4);  
q.enqueue(7);
```

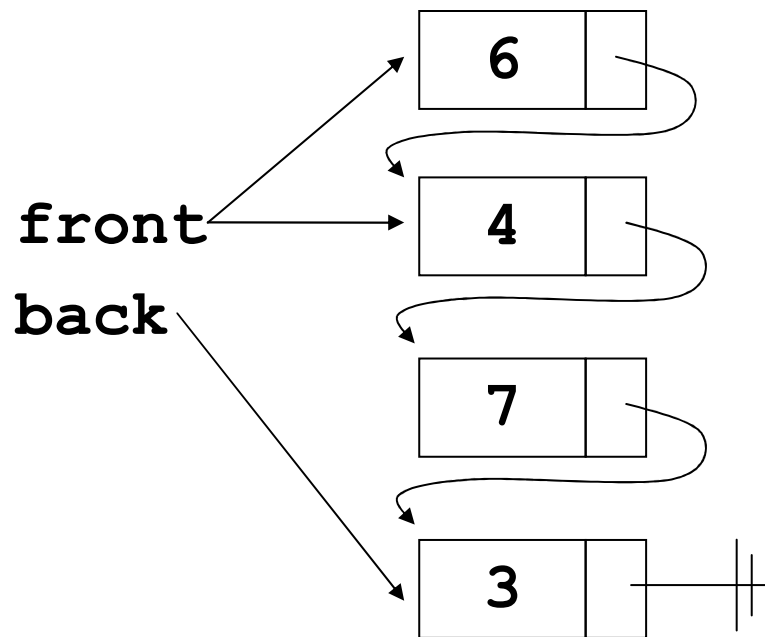
List Queue Example



//Java Code

```
Queue q = new Queue();  
q.enqueue(6);  
q.enqueue(4);  
q.enqueue(7);  
q.enqueue(3);
```

List Queue Example



//Java Code

```
Queue q = new Queue();  
q.enqueue(6);  
q.enqueue(4);  
q.enqueue(7);  
q.enqueue(3);  
q.dequeue();
```

Queue: Circular Linked List Implementation

- Possible implementations of a queue
 - A circular linked list with one external reference
 - A reference to the back

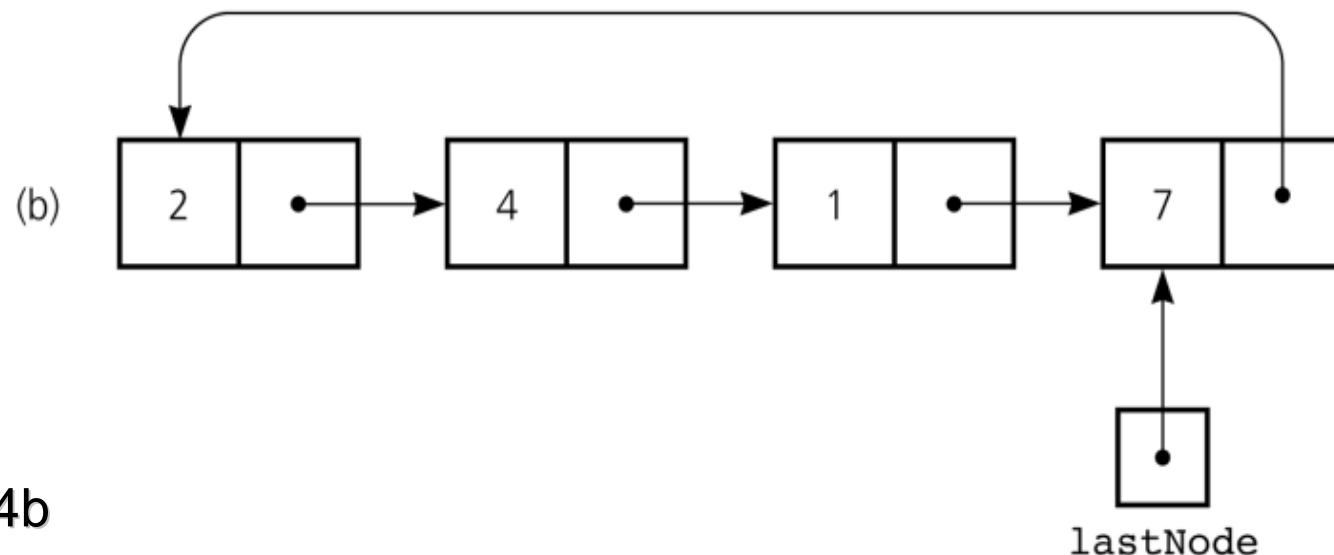


Figure 8-4b

A reference-based implementation of a queue: b) a circular linear linked list with one external reference

Queue: Circular Linked List Implementation

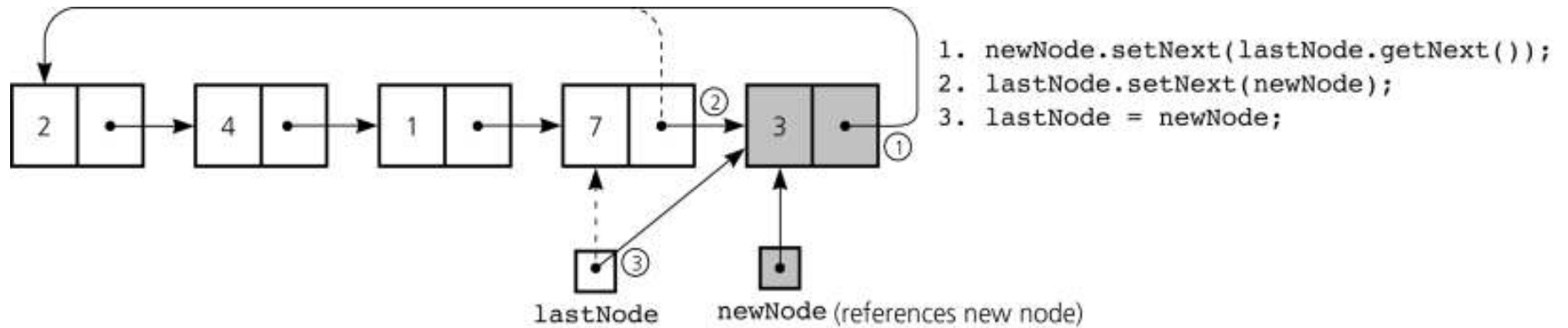


Figure 8-5

Inserting an item into a nonempty queue

Queue: Circular Linked List Implementation

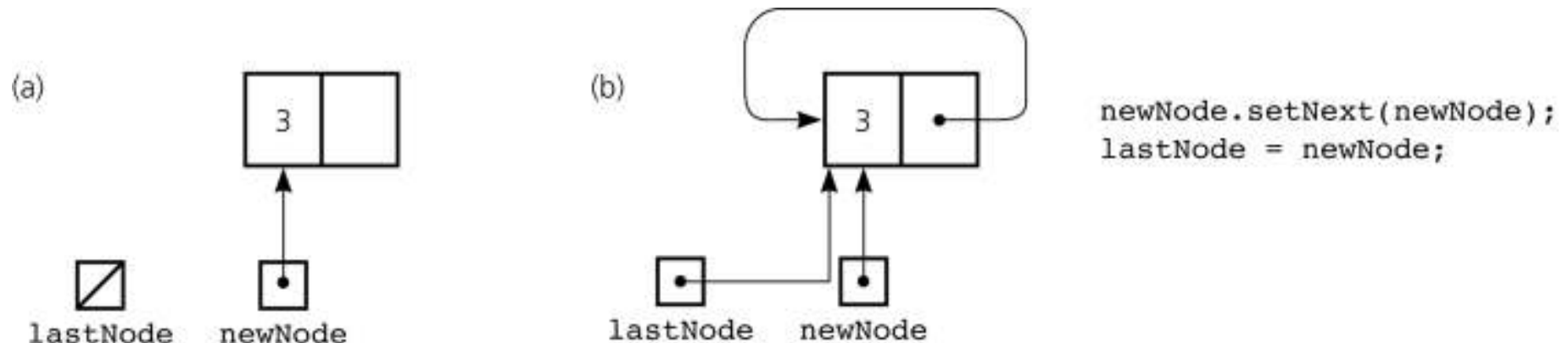


Figure 8-6

Inserting an item into an empty queue: a) before insertion; b) after insertion

Queue: Circular Linked List Implementation

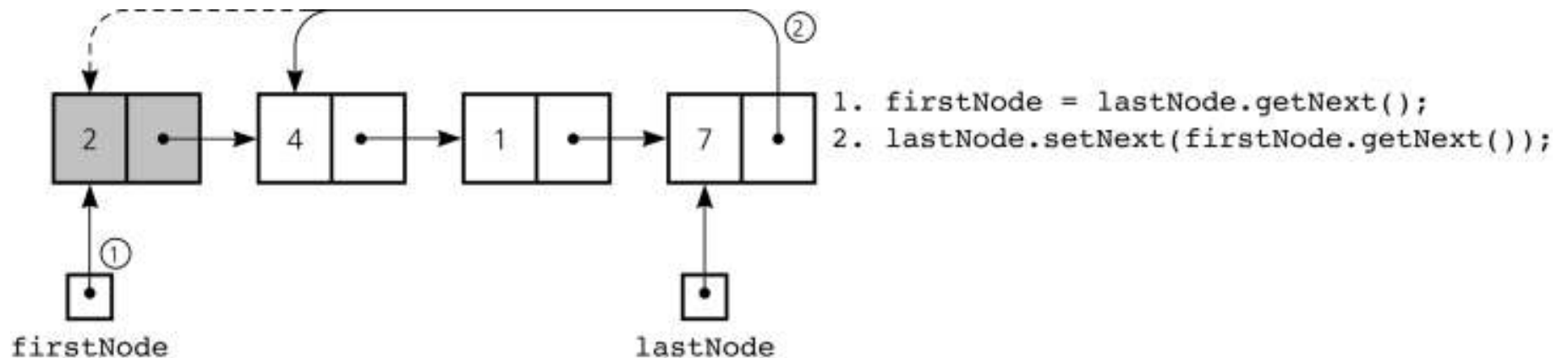


Figure 8-7

Deleting an item from a queue of more than one item

Queue: ADT List Implementation

```
public void enqueue(Object  
newItem) {  
    list.add(list.size()+1, newItem);  
} // end enqueue  
public Object dequeue() {  
    Object temp = list.get(1);  
    list.remove(1);  
    return temp;  
} // end dequeue
```

Queue: ADT List Implementation

- Efficiency depends on implementation of ADT List – in most common implementations, at least one of operations *enqueue()* and *dequeue()* is not efficient
 - On other hand: it was very fast to implement (code is easy, unlikely that errors were introduced when coding).
-

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

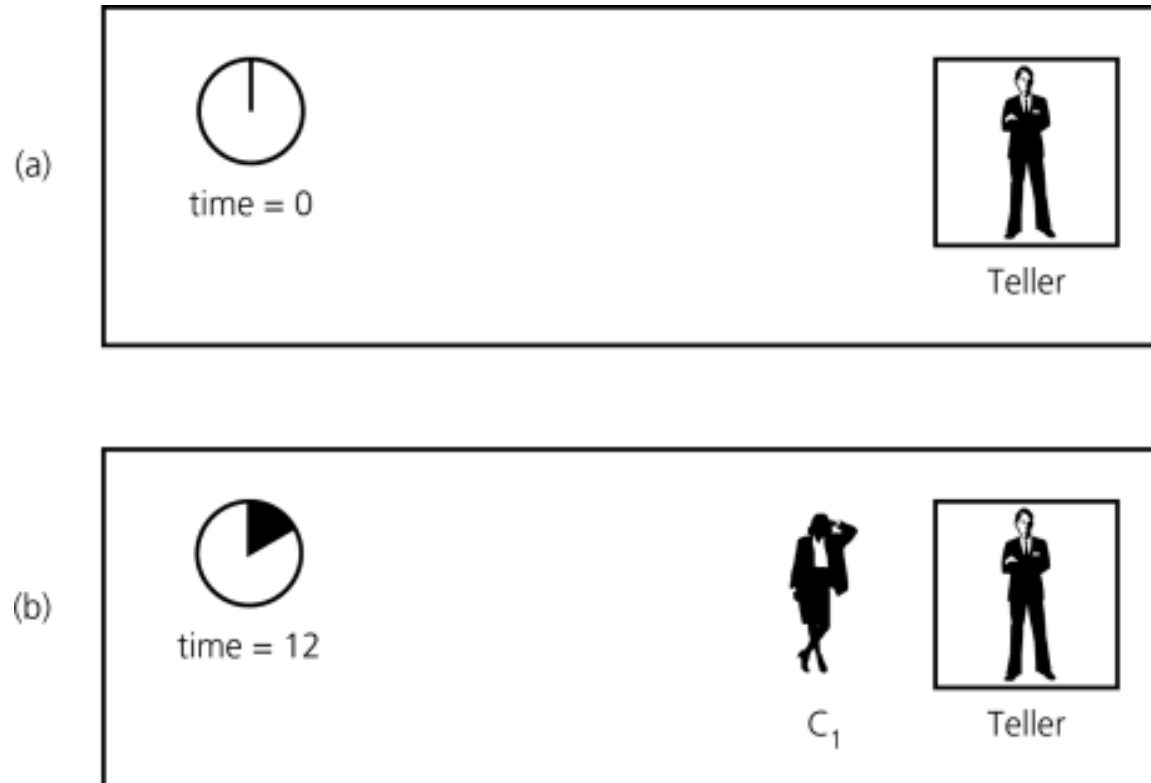


Figure 8-14a and 8-14b

A blank line at at time a) 0; b) 12

Application: Simulation

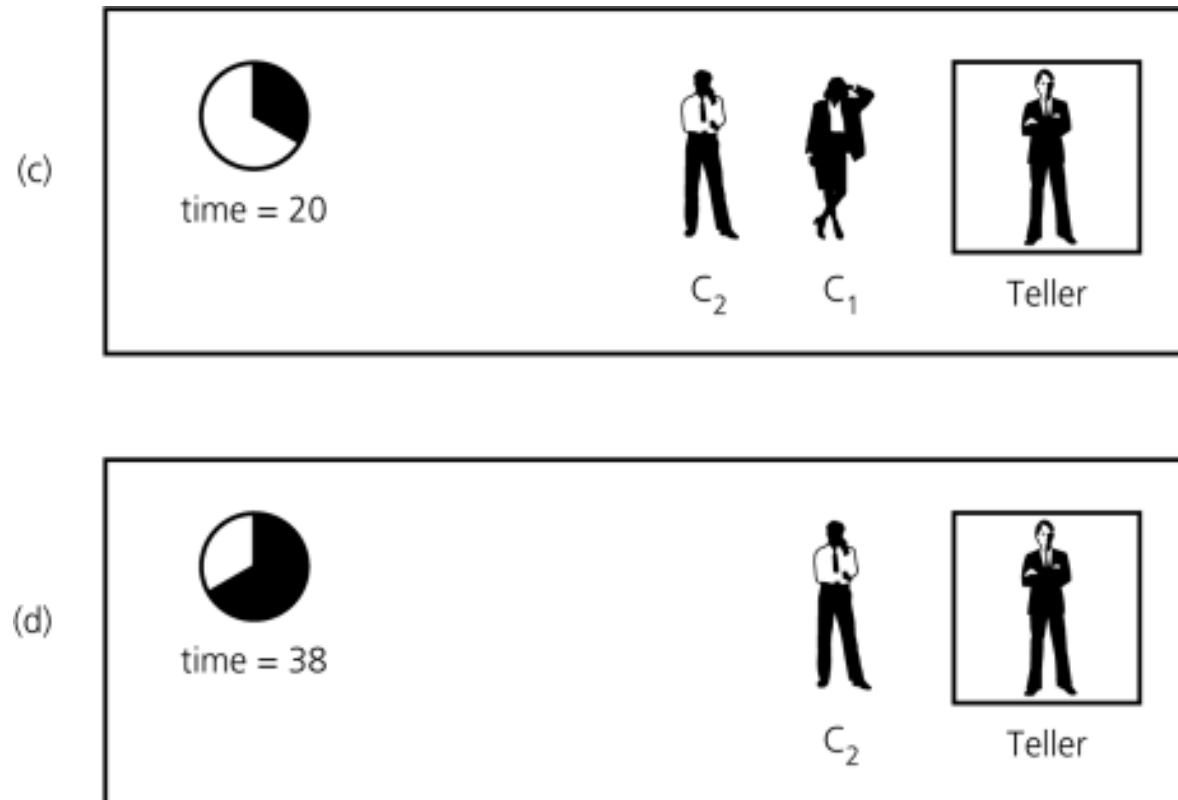


Figure 8-14c and 8-14d

A blank line at at time c) 20; d) 38

Application: Simulation

- An event-driven simulation
 - Simulated time is advanced to the time of the next event
 - Events are generated by a mathematical model that is based on statistics and probability
 - A time-driven simulation
 - Simulated time is advanced by a single time unit
 - The time of an event, such as an arrival or departure, is determined randomly and compared with a simulated clock
-

Application: Simulation

- The bank simulation is concerned with
 - Arrival events
 - Indicate the arrival at the bank of a new customer
 - External events: the input file specifies the times at which the arrival events occur
 - Departure events
 - Indicate the departure from the bank of a customer who has completed a transaction
 - Internal events: the simulation determines the times at which the departure events occur
-

Input file

Arrival	transaction length
20	5
22	4
23	2
30	3

```
while (events remain to be processed){  
    currentTime=time of the next event;  
    if(event is an arrival event)  
        process the arrival event  
    else  
        process the departure event  
}
```

Application: Simulation

- An event list is needed to implement an event-driven simulation
 - 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

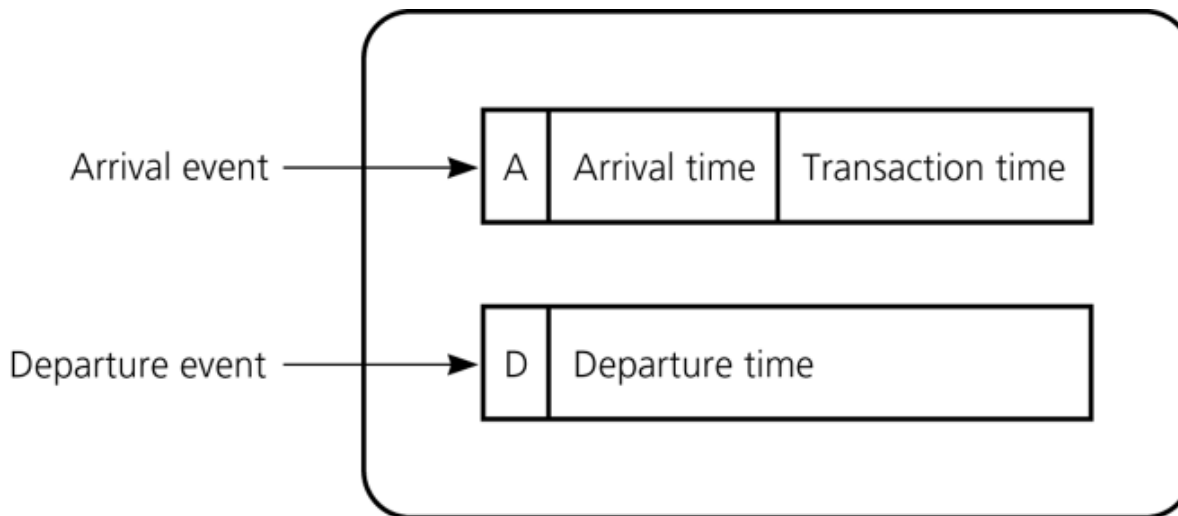


Figure 8-15
A typical instance
of the event list

Create an empty bankQueue
Create an empty eventList
Get the first arrival event from the input file.
Place the arrival event to the eventList.
while (eventList is not empty){
 newEvent = the first element in the eventList;
 if(newEvent is an arrival event)
 processArrival(newEvent);
 else
 processDeparture(newEvent);
}

```
process arrival (Event arrivalEvent){
    boolean atfront=bankQueue.isEmpty();
    bankQueue.enqueue(arrivalEvent);
    Delete arrivalEvent From the event list;
    if(atFront){
        Insert the departure event into the event list.
        time of departure=currentTime+transactionLength;
    }
    if(not at the end of input file)
        Get the next arrival and add to the eventList.
}
```

```
process Departure (Event departureEvent){  
    bankQueue.dequeue();  
    Delete departureEvent From the event list;  
    if(!bankQueue.isEmpty){  
        Insert into the event list a departure event.  
        time of departure=currentTime+transactionLength of the first  
        customer in the queue;  
    }  
}
```

Summary

- The definition of the queue operations gives the ADT queue first-in, first-out (FIFO) behavior
 - A reference-based implementation of a queue uses either
 - A circular linked list
 - A linear linked list with a head reference and a tail reference
 - An array-based implementation of a queue is prone to rightward drift
 - A circular array eliminates the problem of rightward drift
-

Summary

- To distinguish between the queue-full and queue-empty conditions in a queue implementation that uses a circular array, you can
 - Count the number of items in the queue
 - Use a `full` flag
 - Leave one array location empty
 - Models of real-world systems often use queues
 - The event-driven simulation in this chapter uses a queue to model a line of customers in a bank
-

Summary

■ Simulations

- Central to a simulation is the notion of simulated time
 - In a time-driven simulation
 - Simulated time is advanced by a single time unit
 - In an event-driven simulation
 - Simulated time is advanced 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
-