- ***get***. If you remove the restriction that the stack and queue versions of *peek* can retrieve items from only one position, you obtain an operation that can retrieve the item from any position of the list.

Because each of these three ADTs defines its operations in terms of an item's position in the ADT, this book has presented implementations for them that can provide easy access to specified positions. For example, the stack implementations allow the first position (top) to be accessed quickly, while the queue implementations allow the first position (front) and the last position (back) to be accessed quickly.

8.5 Application: Simulation

Simulation models the behavior of systems

Simulation—a major application area for computers—is a technique for modeling the behavior of both natural and human-made systems. Generally, the goal of a simulation is to generate statistics that summarize the performance of an existing system or to predict the performance of a proposed system. In this section, we will consider a simple example that illustrates one important type of simulation.

Consider the following problem. Ms. Simpson, president of the First City Bank of Springfield, has heard her customers complain about how long they have to wait for service. Because she fears that they may move their accounts to another bank, she is considering whether to hire a second teller.

Before Ms. Simpson hires another teller, she would like an approximation of the average time that a customer has to wait for service from First City's only teller. How can Ms. Simpson obtain this information? She could stand with a stopwatch in the bank's lobby all day, but she does not find this prospect particularly exciting. Besides, she would like to use a method that also allows her to predict how much improvement she could expect if the bank hired a given number of additional tellers. She certainly does not want to hire the tellers on a trial basis and monitor the bank's performance before making a final decision.

Ms. Simpson concludes that the best way to obtain the information she wants is to use a computer model to simulate the behavior of her bank. The first step in simulating a system such as a bank is to construct a mathematical model that captures the relevant information about the system. For example, how many tellers does the bank employ? How often do customers arrive? If the model accurately describes the real-world system, a simulation can derive accurate predictions about the system's overall performance. For example, a simulation could predict the average time a customer has to wait before receiving service. A simulation can also evaluate proposed changes to the real-world system. For example, it could predict the effect of hiring more tellers in the bank. A large decrease in the time predicted for the average wait of a customer might justify the cost of hiring additional tellers.

Simulated time

Central to a simulation is the concept of simulated time. Envision a stopwatch that measures time elapsed during a simulation. For example, suppose

that the model of the bank specifies only one teller. At time 0, which is the start of the banking day, the simulated system would be in its initial state with no customers. As the simulation runs, the stopwatch ticks away units of time—perhaps minutes—and certain events occur. At time 12, the bank's first customer arrives. Since there is no line, the customer goes directly to the teller and begins her transaction. At time 20, a second customer arrives. Because the first customer has not yet completed her transaction, the second customer must wait in line. At time 38, the first customer completes her transaction and the second customer can begin his. Figure 8-18 illustrates these four times in the simulation.

To gather the information you need, you run this simulation for a specified period of simulated time. During the course of the run, you need to keep track of certain statistics, such as the average time a customer has to wait for service. Notice that in the small example of Figure 8-18, the first customer had to wait 0 minutes to begin a transaction and the second customer had to wait 18 minutes to begin a transaction—an average wait of 9 minutes.

One point not addressed in the previous discussion is how to determine when certain events occur. For example, why did we say that the first customer arrived at time 12 and the second at time 20? By studying real-world systems like our bank, mathematicians have learned to model events such as the arrival of people, using techniques from probability theory. This statistical information is incorporated into the mathematical model of the system and is used to generate events in a way that reflects the real world. The simulation uses these events and is thus called an **event-driven simulation.** Note that the goal is to reflect the long-term average behavior of the system rather than to predict occurrences of specific events. This goal is sufficient for the needs of the simulation.

Although the techniques for generating events to reflect the real world are interesting and important, they require a good deal of mathematical sophistication. Therefore, simply assume that you already have a list of events available for your use. In particular, for the bank problem, assume that a file contains the time of each customer's arrival—an **arrival event**—and the duration of that customer's transaction once the customer reaches the teller. For example, the data

Sample arrival and transaction times

20	5
22	4
23	2
30	3

indicates that the first customer arrives 20 minutes into the simulation and that the transaction—once begun—requires 5 minutes; the second customer arrives 22 minutes into the simulation and the transaction requires 4 minutes; and so on. Assume that the input file is ordered by arrival time.

Notice that the file does not contain **departure events;** the data does not specify when a customer will complete the transaction and leave. Instead, the simulation must determine when departures occur. By using the arrival time

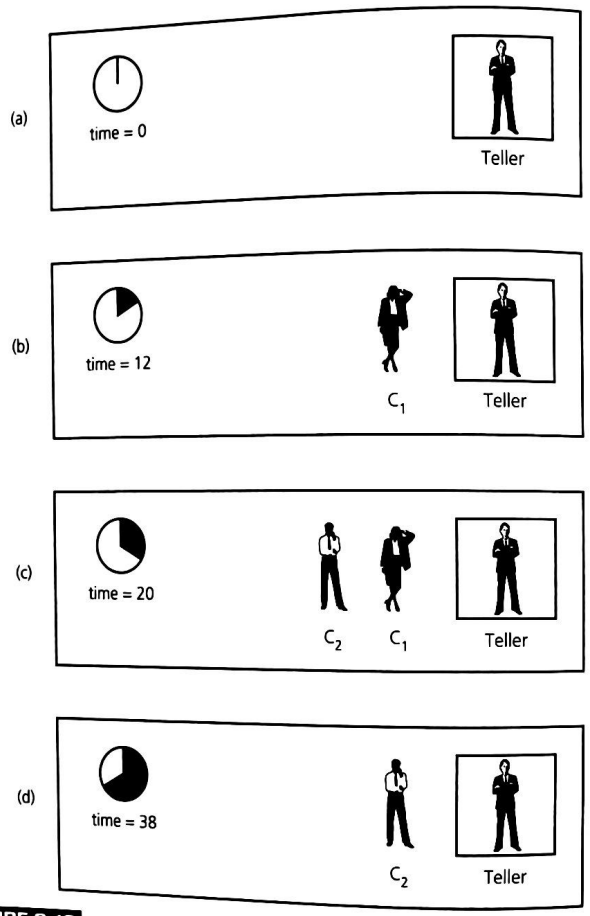

FIGURE 8-18
A bank line at time (a) 0; (b) 12; (c) 20; (d) 38

and the transaction length, the simulation can easily determine the time at which a customer departs. To see how to make this determination, you can conduct a simulation by hand with the previous data as follows:

The results of a simulation

Time	Event
20	Customer 1 enters bank and begins transaction
22	Customer 2 enters bank and stands at end of line
23	Customer 3 enters bank and stands at end of line
25	Customer 1 departs; customer 2 begins transaction
29	Customer 2 departs; customer 3 begins transaction
30	Customer 4 enters bank and stands at end of line
31	Customer 3 departs; customer 4 begins transaction
34	Customer 4 departs

A customer's wait time is the elapsed time between arrival in the bank and the start of the transaction. The average of this wait time over all the customers is the statistic that you want to obtain.

To summarize, this simulation is concerned with two types of events:

- **Arrival events.** These events indicate the arrival at the bank of a new customer. The input file specifies the times at which the arrival events occur. As such, they are **external events.** When a customer arrives at the bank, one of two things happens. If the teller is idle when the customer arrives, the customer enters the line and begins the transaction immediately. If the teller is busy, the new customer must stand at the end of the line and wait for service.
- **Departure events.** These events indicate the departure from the bank of a customer who has completed a transaction. The simulation determines the times at which the departure events occur. As such, they are **internal events.** When a customer completes the transaction, he or she departs and the next person in line—if there is one—begins a transaction.

The main tasks of an algorithm that performs the simulation are to determine the times at which the events occur and to process the events when they do occur. The algorithm is stated at a high level as follows:

A first attempt at a simulation algorithm

```
// initialize
currentTime = 0
Initialize the line to "no customers"

while (currentTime ≤ time of the final event) {

  if (an arrival event occurs at time currentTime) {
    Process the arrival event
  } // end if
  if (a departure event occurs at time currentTime) {
    Process the departure event
  } // end if
  // when an arrival event and departure event
  // occur at the same time, arbitrarily process
  // the arrival event first
```

```
    ++currentTime
  }  // end while
```

A time-driven simulation simulates the ticking of a clock

An event-driven simulation considers only times of certain events, in this case, arrivals and departures

But do you really want to increment *currentTime* by 1? You would for a **time-driven simulation,** where you would determine arrival and departure times at random and compare those times to *currentTime*. In such a case, you would increment *currentTime* by 1 to simulate the ticking of a clock. Recall, however, that this simulation is event driven, so you have a file of arrival times and transaction times. Because you are interested only in those times at which arrival and departure events occur and because no action is required between events, you can advance *currentTime* from the time of one event directly to the time of the next.

Thus, you can revise the pseudocode solution as follows:

First revision of the simulation algorithm

```
// 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
  }  // end if
  // when an arrival event and departure event
  // occur at the same time, arbitrarily process
  // the arrival event first
}  // end while
```

You must determine the time of the next arrival or departure event so that you can implement the statement

```
currentTime = time of next event
```

An event list contains all future events

To make this determination, you must maintain an **event list.** An event list contains all arrival and departure events that will occur but have not occurred yet. The times of the events in the event list are in ascending order, and thus the next event to be processed is always at the beginning of the list. The algorithm simply gets the event from the beginning of the list, advances to the time specified, and processes the event. The difficulty, then, lies in successfully managing the event list.

Since each arrival event generates exactly one departure event, you might think that you should read the entire input file and create an event list of all arrival and departure events sorted by time. Self-Test Exercise 5 asks you to explain why this approach is impractical. As you will see, you can instead

manage the event list for this particular problem so that it always contains at most one event of each kind.

This event list contains at most one arrival event and one departure event

Recall that the arrival events are specified in the input file in ascending time order. You thus never need to worry about an arrival event until you have processed all the arrival events that precede it in the file. You simply keep the earliest unprocessed arrival event in the event list. When you eventually process this event—that is, when it is time for this customer to arrive—you replace it in the event list with the next unprocessed arrival event, which is the next item in the input file.

Similarly, you need to place only the next departure event to occur on the event list. But how can you determine the times for the departure events? Observe that the next departure event always corresponds to the customer that the teller is currently serving. As soon as a customer begins service, the time of his or her departure is simply

time of next departure = time service begins + length of transaction

Recall that the length of the customer's transaction is in the input file, along with the arrival time. Thus, as soon as a customer begins service, you place a departure event corresponding to this customer in the event list. Figure 8-19 illustrates a typical instance of the event list for this simulation.

Now consider how you can process an event when it is time for the event to occur. You must perform two general types of actions:

Two tasks are required to process each event

- **Update the line:** Add or remove customers.
- **Update the event list:** Add or remove events.

As customers arrive, they go to the back of the line. The current customer, who is at the front of the line, is being served, and it is this customer that you remove from the system next. It is thus natural to use a queue to represent the line of customers in the bank. For this problem, the only information that you

A queue represents the customers in line

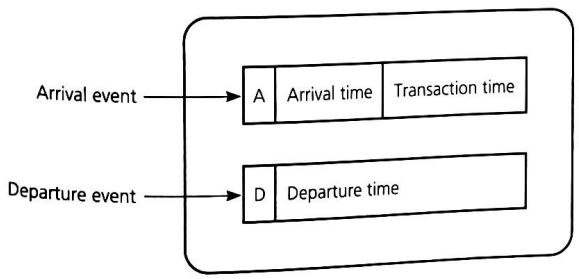

FIGURE 8-19
A typical instance of the event list

must store in the queue about each customer is the time of arrival and the length of the transaction. The event list, since it is sorted by time, is not a queue. We will examine it in more detail shortly.

The event list is not a queue

To summarize, you process an event as follows:

The algorithm for arrival events

```
TO PROCESS AN ARRIVAL EVENT

  // Update the event list
  Delete the arrival event for customer C from
      the event list

  if (new customer C begins transaction immediately) {
    Insert a departure event for customer C into the
        event list (time of event = current time +
        transaction length)
  } // end if
  if (not at the end of the input file) {
    Read a new arrival event and add it to the event list
        (time of event = time specified in file)
  } // end if
```

A new customer always enters the queue and is served while at the queue's front

Because a customer is served while at the front of the queue, a new customer always enters the queue, even if the queue is empty. You then delete the arrival event for the new customer from the event list. If the new customer is served immediately, you insert a departure event into the event list. Finally, you read a new arrival event into the event list. This arrival event can occur either before or after the departure event.

The algorithm for departure events

```
TO PROCESS A DEPARTURE EVENT

  // Update the line
  Delete the customer at the front of the queue
  if (the queue is not empty) {
     The current front customer begins transaction
  } // end if

  // Update the event list
  Delete the departure event from the event list
  if (the queue is not empty) {
     Insert into the event list the departure event for
        the customer now at the front of the queue
        (time of event = current time + transaction length)
  } // end if
```

After processing the departure event, you do not read another arrival event from the file. Assuming that the file has not been read completely, the event

list will contain an arrival event whose time is earlier than any arrival still in the input file.

Examining the event list more closely will help explain the workings of the algorithm. There is no typical form that an event list takes. For this simulation, however, the event list has four possible configurations:

Four configurations of the event list for this simulation

- Initially, the event list contains an arrival event A after you read the first arrival event from the input file but before you process it:

  Event list: A (initial state)

- Generally, the event list for this simulation contains exactly two events: one arrival event A and one departure event D. Either the departure event is first or the arrival event is first as follows:

  Event list: $D\ A$ (general case—next event is a departure)

  or

  Event list: $A\ D$ (general case—next event is an arrival)

- If the departure event is first and that event leaves the teller's line empty, a new departure event does not replace the just-processed event. Thus, in this case, the event list appears as

  Event list: A (a departure leaves the teller's line empty)

  Notice that this instance of the event list is the same as its initial state.

- If the arrival event is first and if, after it is processed, you are at the end of the input file, the event list contains only a departure event:

  Event list: D (the input has been exhausted)

Other situations result in an event list that has one of the previous four configurations.

You insert new events either at the beginning of the event list or at the end, depending on the relative times of the new event and the event currently in the event list. For example, suppose that the event list contains only an arrival event A and that another customer is now at the front of the line and beginning a transaction. You need to generate a departure event D for this customer. If the customer's departure time is before the time of the arrival event A, you must insert the departure event D before the event A in the event list. However, if the departure time is after the time of the arrival event, you must insert the departure event D after the arrival event A. In the case of a tie, you need a rule to determine which event should take precedence. In this solution, we arbitrarily choose to place the departure event after the arrival event.

You can now combine and refine the pieces of the solution into an algorithm that performs the simulation by using the ADT queue operations to manage the bank line:

The final pseudocode for the event-driven simulation

```
+simulate()
// Performs the simulation.

  Create an empty queue bankQueue to represent the bank line
  Create an empty event list eventList

  Get the first arrival event from the input file
  Place the arrival event in the event list

  while (the event list is not empty) {
    newEvent = the first event in the event list

    if (newEvent is an arrival event) {
      processArrival(newEvent, arrivalFile,
                     eventList, bankQueue)

    }
    else {
      processDeparture(newEvent, eventList, bankQueue)
    }  // end if
  }  // end while

+processArrival(in arrivalEvent:Event,
                in arrivalFile:File,
                inout anEventList:EventList,
                inout bankQueue:Queue)
// Processes an arrival event.

  atFront = bankQueue.isEmpty()  // present queue status

  // update the bankQueue by inserting the customer, as
  // described in arrivalEvent, into the queue
  bankQueue.enqueue(arrivalEvent)

  // update the event list
  Delete arrivalEvent from anEventList

  if (atFront) {
    // the line was empty, so new customer is at front
    // of line and begins transaction immediately
    Insert into the anEventList a departure event that
        corresponds to the new customer and has
    currentTime = currentTime + transaction length
  }  // end if

  if (not at end of input file) {
    Get the next arrival event from arrivalFile
    Add the event -- with time as specified in the input
        file -- to anEventList
  }  // end if

+processDeparture(in departureEvent:Event,
                  in anEventList:EventList,
                  inout bankQueue:Queue)
// processes a departure event.

  // update the line by deleting the front customer
  bankQueue.dequeue()

  // update the event list
  Delete departureEvent from anEventList
  if (!bankQueue.isEmpty())
    // customer at front of line begins transaction
    Insert into anEventList a departure event that
        corresponds to the customer now at the front of the
        line and has currentTime = currentTime
        + transaction length
  }  // end if
```


Figure 8-20 begins a trace of this algorithm for the data on page 436 and shows the changes to the queue and event list. Self-Test Exercise 6 at the end of this chapter asks you to complete the trace.

The event list is, in fact, an ADT. By examining the previous pseudocode, you can see that this ADT must include at least the following operations:

ADT event list operations

```
+createEventList()
// Creates an empty event list.

+isEmpty():boolean {query}
// Determines whether an event list is empty.

+insert(in anEvent:Event)
// Inserts anEvent into an event list so that events
// are ordered by time. If an arrival event and a
// departure event have the same time, the arrival
// event precedes the departure event.

+delete()
// Deletes the first event from an event list.

+retrieve():Event
// Retrieves the first event in an event list.
```

Time	Action	bankQueue (front to back)	anEventList (beginning to end)
		(empty)	A 20 5
0	Read file, place event in **anEventList**		*(empty)*
20	Update **anEventList** and **bankQueue**: Customer 1 enters bank	20 5	
	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 8-20

A partial trace of the bank simulation algorithm for the data
20 5
22 4
23 2
30 3

Programming Problem 8 at the end of this chapter asks you to complete the implementation of this simulation.

Summary

1. The definition of the queue operations gives the ADT queue first-in, first-out (FIFO) behavior.
2. The insertion and deletion operations for a queue require efficient access to both ends of the queue. Therefore, a reference-based implementation of a queue uses either a circular linked list or a linear linked list that has both a head reference and a tail reference.
3. An array-based implementation of a queue is prone to rightward drift. This phenomenon can make a queue look full when it really is not. Shifting the items in the array is one way to compensate for rightward drift. A more efficient solution uses a circular array.
4. If you use a circular array to implement a queue, you must be able to distinguish between the queue-full and queue-empty conditions. You can make this distinction