The following pseudocode shows how you could use a priority queue to organize your assignments and other responsibilities so that you know which one to complete first:

```java
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()
```

### 13.4 Application: Simulation

**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.

**A problem to solve.** Ms. Simpson, president of the First City Bank of Springfield, has heard her customers complain about how long they have to wait for service at the branch located in a downtown grocery store. Because she fears losing those customers to another bank, she is considering whether to hire a second teller for that branch.

Before Ms. Simpson hires another teller, she would like an approximation of the average time a customer has to wait for service from that branch’s only teller. Ms. Simpson heard you were great at solving problems and has come to you for help. How can you obtain this information for her?

**Considerations.** You could stand with a stopwatch in the bank’s lobby all day, but that task is not particularly exciting. Besides, you should use an approach that also allows Ms. Simpson 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 then monitor the bank’s performance before making her final decision.
You conclude that the best way to obtain the information needed is to use a computer model to simulate the behavior of the 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? How long do the customers' transactions take?

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, such as predicting the effect of hiring more tellers at the bank. A large decrease in the time predicted for the average wait of a customer might justify the cost of hiring additional tellers.

After discussing the problem with Ms. Simpson, you decide that you want the simulation to determine

- The average time a customer waits before receiving service from the current single teller
- The decrease in customer wait time with each new teller added

**Simulation time and events.** 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 20, the bank's first customer arrives. Because there is no line, the customer goes directly to the teller and begins her transaction, which will take about 6 minutes to complete. At time 22, a second customer arrives. Because the first customer has not yet completed her transaction, the second customer must wait in line. At time 26, the first customer completes her transaction and the second customer can begin his. Figure 13-6 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 13-6, the first customer had to wait 0 minutes to begin a transaction and the second customer had to wait 4 minutes to begin a transaction—an average wait of 2 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 20 and the second at time 22? After studying real-world systems like our bank, mathematicians learned to model events such as the arrival of people by 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 our 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, we simply assume that we already have a list of events available for our use. In particular, for the bank problem, we 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

<table>
<thead>
<tr>
<th>Arrival time</th>
<th>Transaction length</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>6</td>
</tr>
<tr>
<td>22</td>
<td>4</td>
</tr>
<tr>
<td>23</td>
<td>2</td>
</tr>
<tr>
<td>30</td>
<td>3</td>
</tr>
</tbody>
</table>
FIGURE 13-6  A bank line at time (a) 0; (b) 20; (c) 22; (d) 26

indicates that the first customer arrives 20 minutes into the simulation and her transaction—once begun—requires 6 minutes; the second customer arrives 22 minutes into the simulation, and his transaction requires 4 minutes; and so on. Assume that the input file is ordered by arrival time.

The use of a data file with predetermined event information is common in simulations. It allows us to try many different scenarios or bank teller configurations with the same set of events to ensure a fair comparison.

Notice that the file does not contain departure events; the data does not specify when a customer will complete the transaction and leave. In fact, the departure time of a customer cannot be determined until the simulation is run, so the simulation must determine when departures occur. By using the arrival time and the transaction length, the simulation can easily determine
the time at which a customer departs. To compute the departure time, we add the length of the transaction to the time when the customer begins the transaction.

For example, if we run the simulation by hand with the previous data, we would compute the departure times as follows:

<table>
<thead>
<tr>
<th>Time</th>
<th>Event</th>
</tr>
</thead>
</table>
| 20   | Customer 1 enters bank and begins transaction  
   | Determine customer 1 departure event is at time 26 |
| 22   | Customer 2 enters bank and stands at end of line |
| 23   | Customer 3 enters bank and stands at end of line |
| 26   | Customer 1 departs; customer 2 begins transaction  
   | Determine customer 2 departure event is at time 30 |
| 30   | Customer 2 departs; customer 3 begins transaction  
   | Determine customer 3 departure event is at time 32 |
| 30   | Customer 4 enters bank and stands at end of line |
| 32   | Customer 3 departs; customer 4 begins transaction  
   | Determine customer 4 departure event is at time 35 |
| 35   | Customer 4 departs |

A customer's wait time is the elapsed time between arrival in the bank and the start of the transaction, that is, the amount of time the customer spends in line. 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 kinds of events:

**Note:** Kinds of events in an event-driven simulation

- Arrival 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 externally generated events. When a customer arrives at the bank, one of two things happens. If the teller is idle when the customer arrives, the customer goes to the teller 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 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. Thus, they are internally generated events. When a customer completes the transaction, he or she departs and the next person in line—if there is one—begins a transaction.

**Event loop.** The main tasks of an algorithm that performs a simulation are to repeatedly determine the times at which events occur and to process the events when they do occur. In simulation and gaming applications, this process is referred to as the event loop. Our algorithm is stated at a high level as follows:

```c
// 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
if (a departure event occurs at time currentTime)
   Process the departure event

/ / When an arrival event and departure event occur at the same time,
/ / arbitrarily process the arrival event first
currentTime++

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. Video games use this approach, since events can occur or need to be processed in almost every unit of time, which is typically a video frame. 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, you now have a file of predetermined 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:

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
   }

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

currentTime = time of next event

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

Managing and processing customers and events. As customers arrive, they go to the back of the line. The current customer, who was 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, bankLine, to represent the line of customers in the bank. For this problem, the only information that you must store in the queue about each customer is the time of arrival and the length of the transaction.

Arrival events and departure events are ordered by time, and we always want to remove and process the next event that should occur—the highest-priority event. The ADT priority queue is used in this way. Our events can be stored in the priority queue eventPriorityQueue. We can initialize eventPriorityQueue with the arrival events in the simulation data file and later add the departure events as they are generated.
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 departure = time service begins + length of transaction

Recall that the length of the customer’s transaction is in the event queue, 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 queue. Figure 13-7 illustrates a typical instance of an arrival event and a departure event used in this simulation.

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

<table>
<thead>
<tr>
<th>(a) Arrival event Type</th>
<th>Time</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>20</td>
<td>6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(b) Departure event Type</th>
<th>Time</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>28</td>
<td></td>
</tr>
</tbody>
</table>

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:

- Update the bank line: Add or remove customers.
- Update the event queue: Add or remove events.

To summarize, you process an arrival event as follows:

```
// TO PROCESS AN ARRIVAL EVENT
// Update the event queue
Remove the arrival event for customer C from the event queue

// Update the bank line
if (bank line is empty and teller is available)
{
    Departure time of customer C is current time + transaction length
    Add a departure event for customer C to the event queue
    Mark the teller as unavailable
}
else
    Add customer C to the bank line
```

When customer C arrives at the bank, if the line is empty and the teller is not serving another customer, customer C can go directly to the teller. The wait time is 0 and you insert a departure event into the event queue. If other customers are in line, or if the teller is assisting another customer, customer C must go to the end of the line.

You process a departure event as follows:

```
// TO PROCESS A DEPARTURE EVENT
// Update the event queue
Remove the departure event from the event queue
```
\[
\text{Update the bank line}
\]
\[
\text{if (bank line is not empty)}
\]
\[
\text{Remove customer } C \text{ from the front of the bank line}
\text{Customer } C \text{ begins transaction}
\text{Departure time of customer } C \text{ is current time + transaction length}
\text{Add a departure event for customer } C \text{ to the event queue}
\]
\[
\text{else}
\text{Mark the teller as available.}
\]

When a customer finishes a transaction and leaves the bank, if the bank line is not empty, the next customer \( C \) leaves the line and goes to the teller. You insert a departure event for customer \( C \) into the event queue.

You can now combine and refine the pieces of the solution into an algorithm that performs the simulation by using the ADTs queue and priority queue:

\[
\text{Performs the simulation.}
\]
\[
\text{simulate(): void}
\]
\[
\text{bankLine = a new empty queue} \quad // \text{Bank line}
\text{eventPriorityQueue = a new empty priority queue} \quad // \text{Event queue}
\text{tellerAvailable = true}
\]
\[
\text{Create and add arrival events to event queue}
\text{while (data file is not empty)}
\]
\[
\text{Get next arrival time } a \text{ and transaction time } t \text{ from file}
\text{newArrivalEvent = a new arrival event containing } a \text{ and } t
\text{eventPriorityQueue.enqueue(newArrivalEvent)}
\]
\[
\text{Event loop}
\text{while (eventPriorityQueue is not empty)}
\]
\[
\text{newEvent = eventPriorityQueue.peekFront()}
\]
\[
\text{Get current time}
\text{currentTime = time of newEvent}
\]
\[
\text{if (newEvent is an arrival event)}
\text{processArrival(newEvent, eventPriorityQueue, bankLine)}
\text{else}
\text{processDeparture(newEvent, eventPriorityQueue, bankLine)}
\]
\[
\text{Processes an arrival event.}
\text{processArrival(arrivalEvent: Event, eventPriorityQueue: PriorityQueue, bankLine: Queue)}
\]
\[
\text{Remove this event from the event queue}
\text{eventPriorityQueue.dequeue()}
\text{customer = customer referenced in arrivalEvent}
\text{if (bankLine.isEmpty() && tellerAvailable)}
\]
\[
\text{departureTime = currentTime + transaction time in arrivalEvent}
\text{newDepartureEvent = a new departure event with departureTime}
\text{eventPriorityQueue.enqueue(newDepartureEvent)}
\]
tellerAvailable = false
}
else
    bankLine.enqueue(customer)

// Processes a departure event.
+processDeparture(departureEvent: Event, eventPriorityQueue: PriorityQueue,
    bankLine: Queue)
{
    // Remove this event from the event queue
    eventPriorityQueue.dequeue()
    if (!bankLine.isEmpty())
    {
        // Customer at front of line begins transaction
        customer = bankLine.peekFront()
        bankLine.dequeue()
        departureTime = currentTime + transaction time in customer
        newDepartureEvent = a new departure event with departureTime
        eventPriorityQueue.enqueue(newDepartureEvent)
    }
    else
        tellerAvailable = true
}

Figure 13-8 traces this algorithm for the data given earlier and shows the changes to the queue and priority queue. Note that the notation C_i represents a customer who arrived at time i. There are several more implementation details that must be decided, such as how to represent customers and events. Programming Problem 6 at the end of this chapter asks you to complete the implementation of this simulation.

**FIGURE 13-8** A trace of the bank simulation algorithm for the data 20, 6, 22, 4, 23, 2, 30, 3. (The blue events are events that change or are created at each point in time.)

```
<table>
<thead>
<tr>
<th>Time</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>C_20</td>
<td></td>
<td>A 22 4</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>C_20</td>
<td></td>
<td>A 22 4</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>C_20</td>
<td></td>
<td>A 22 4</td>
<td>A 23 2</td>
</tr>
<tr>
<td>26</td>
<td>C_22</td>
<td>A 23 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>C_23</td>
<td>A 30 3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>C_30</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>35</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
Question 6 In the bank simulation problem, why is it impractical to read the entire input file and create a list of all the arrival and departure events before the simulation begins?

Question 7 Hand-trace the bank-line simulation using the following data:

<table>
<thead>
<tr>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>8</td>
<td>5</td>
</tr>
</tbody>
</table>

Each line of data contains an arrival time and a transaction time. Show the state of the queue and the priority queue at each step.

13.5 Position-Oriented and Value-Oriented ADTs

Of the abstract data types that we have seen so far, the stack, the list, and the queue have operations defined in terms of the positions of their data items. We call them position-oriented ADTs. Stacks and queues greatly restrict the positions that their operations can affect; only their end positions can be accessed. The list removes this restriction.

The ADT sorted list is an example of a value-oriented ADT. When a new item is added to a sorted list, the sorted list determines the item's placement according to its value. Unlike a list, a sorted list cannot be told where to place its items. The same is true of a priority queue, since it orders its entries according to their priorities, which are values that are a part of each entry.

Although stacks and queues order their entries, and you do not specify where a stack or a queue should place its items, the entries' positions are not determined according to their values. Thus, we do not classify stacks and queues as value oriented.

**Note:** The ADTs stack, queue, list, sorted list, and priority queue all order their entries. The stack, queue, and list are position oriented; the sorted list and priority queue are value oriented. The ADT bag does not order its entries; it is neither position oriented nor value oriented.

Stacks are really quite similar to queues. This similarity becomes apparent if you pair off their operations, as follows:

- Stack **isEmpty** and queue **isEmpty** see whether any items exist in the ADT.
- push and enqueue insert a new item into one end (the top and back, respectively) of the ADT.
- pop and dequeue: The pop operation removes the most recent item, which is at the top of the stack, and dequeue removes the first item, which is at the front of the queue.
- Stack **peek** and queue **peekFront**: The **peek** operation retrieves the most recent item, which is at the top of the stack, and **peekFront** retrieves the first item at the front of the queue.