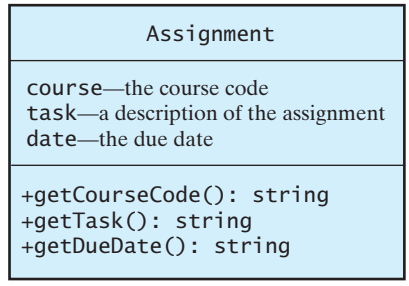
FIGURE 13-5 UML diagram for the class `Assignment`



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:

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

13.4 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.

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 Ms. Simpson?

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 to begin service from the current single teller
- The decrease in customer wait time with each new teller added

Simulated time

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.

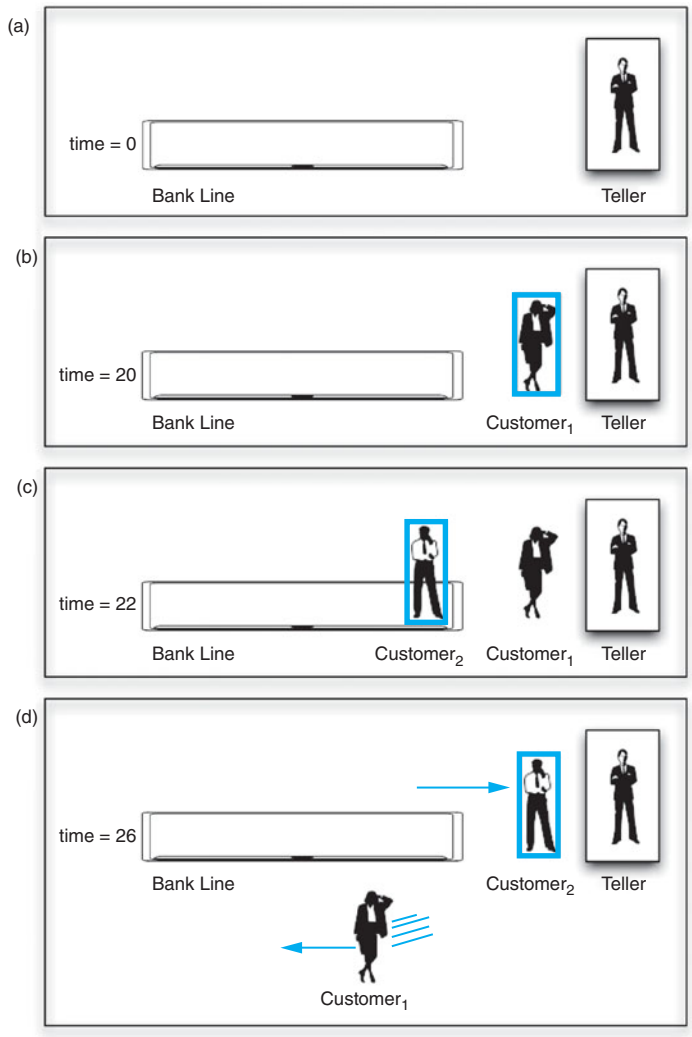
One point not addressed in the previous discussion is how to determine when certain events occur. For example, why did we say that the first customer arrived at time 20 and the second at time 22? After studying real-world systems like our bank, mathematicians learned to model events such as the arrival 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

Sample arrival and transaction times

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

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:

The results of a simulation

Time	Event
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**. 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
    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 frame. In such a case, you would increment `currentTime` by 1 to simulate the ticking of a clock.

A time-driven simulation simulates the ticking of a clock

Recall, however, that this simulation is event driven, so you 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.

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

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

    // 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 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.

An event list contains all future arrival events and departure events

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, `bankQueue`, 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.

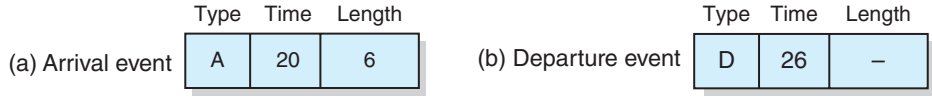
Arrival events and departure events are ordered by time, and we always want to remove and process the next event that should occur—the highest-priority event. The ADT priority queue is used in this way. Our events can be stored in the priority queue `eventListPQueue`. We can initialize `eventListPQueue` 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 list, 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 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



Two tasks are required to process each event

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 list: Add or remove events.

To summarize, you process an arrival event as follows:

The algorithm for arrival events

```
// TO PROCESS AN ARRIVAL EVENT

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

// 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 list
    Mark the teller as unavailable
}
else
    Add customer C to the bank line
```

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

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 list. 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:

The algorithm for departure events

```
// TO PROCESS A DEPARTURE EVENT

// Update the event list
Remove the departure event from the event list

// Update the bank line
if (bank line is not empty)
```

```
{
   Remove customer C from the front of the bank line
   Customer C begins transaction
   Departure time of customer C is current time + transaction length
   Add a departure event for customer C to the event list
}
else
   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 list.

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:

The final pseudocode for the event-driven simulation

```
// Performs the simulation.
simulate(): void

   Create an empty queue bankQueue to represent the bank line
   Create an empty priority queue eventListPQueue for the event list

   tellerAvailable = true

   // Create and add arrival events to event list
   while (data file is not empty)
   {
      Get next arrival time a and transaction time t from file
      newArrivalEvent = a new arrival event containing a and t
      eventListPQueue.add(newArrivalEvent)
   }

   // Event loop
   while (eventListPQueue is not empty)
   {
      newEvent = eventListPQueue.peek()

      // Get current time
      currentTime = time of newEvent

      if (newEvent is an arrival event)
         processArrival(newEvent, eventListPQueue, bankQueue)
      else
         processDeparture(newEvent, eventListPQueue, bankQueue)
   }


// Processes an arrival event.
processArrival(arrivalEvent: Event, eventListPQueue: PriorityQueue,
               bankQueue: Queue)

   // Remove this event from the event list
   eventListPQueue.remove()

   customer = customer referenced in arrivalEvent
   if (bankQueue.isEmpty() && tellerAvailable)
   {
      departureTime = currentTime + transaction time in arrivalEvent
      newDepartureEvent = a new departure event with departureTime
      eventListPQueue.add(newDepartureEvent)
      tellerAvailable = false
   }
   else
      bankQueue.enqueue(customer)
```

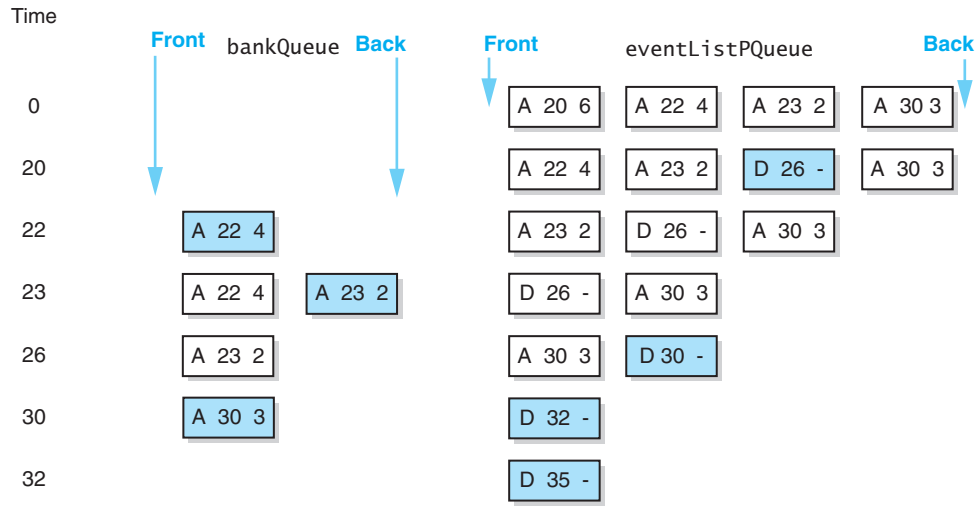
```
// Processes a departure event.
+processDeparture(departureEvent: Event, eventListPQueue: PriorityQueue,
                  bankQueue: Queue)

   // Remove this event from the event list
   eventListPQueue.remove()

   if (!bankQueue.isEmpty())
   {
      // Customer at front of line begins transaction
      customer = bankQueue.peek()
      bankQueue.dequeue()
      departureTime = currentTime + transaction time in customer
      newDepartureEvent = a new departure event with departureTime
      eventListPQueue.add(newDepartureEvent)
   }
   else
      tellerAvailable = true
```

Figure 13-8 begins a trace of this algorithm for the data given earlier and shows the changes to the queue and priority queue. Checkpoint Question 6 asks you to complete the trace. 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



Question 5 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?

2. Using the class `priority_queue` in the Standard Template Library, define and test the class `OurPriorityQueue` that is derived from `PriorityQueueInterface`, as developed in Exercise 8. The class `priority_queue` has the following methods that you can use to define the methods for `OurPriorityQueue`:

```
priority_queue();                       // Default constructor
bool empty() const;                     // Tests whether the priority queue is empty
void push(const ItemType& newEntry);    // Adds newEntry to the priority queue
void pop();                             // Removes the entry having the highest priority
ItemType& top();                        // Returns a reference to the entry having the
                                        // highest priority
```

To access `priority_queue`, use the following `include` statement:

```
#include <priority_queue>;
```

Whenever you need a queue or a priority queue for any of the following problems, use the classes `OurQueue` *and* `OurPriorityQueue` *that Programming Problems 1 and 2 ask you to write.*

---

3. Implement the palindrome-recognition algorithm described in Section 13.2.2.

4. Implement the recognition algorithm that you wrote to solve Exercise 2 using the classes `OurQueue`, as described in Programming Problem 1, and `OurStack` , as described in Programming Problem 1 of Chapter 6.

5. Implement the radix sort of an array by using a queue for each group. The radix sort is discussed in Section 11.2.3 of Chapter 11.

6. Implement the event-driven simulation of a bank that this chapter described. A queue of arrival events will represent the line of customers in the bank. Maintain the arrival events and departure events in a priority queue, sorted by the time of the event. Use a link-based implementation for the event list.

The input is a text file of arrival and transaction times. Each line of the file contains the arrival time and required transaction time for a customer. The arrival times are ordered by increasing time.

Your program must count customers and keep track of their cumulative waiting time. These statistics are sufficient to compute the average waiting time after the last event has been processed. Display a trace of the events executed and a summary of the computed statistics (the total number of arrivals and average time spent waiting in line). For example, the input file shown in the left columns of the following table should produce the output shown in the right column.

Input file		Output from processing file on left
1	5	Simulation Begins
2	5	Processing an arrival event at time: 1
4	5	Processing an arrival event at time: 2
20	5	Processing an arrival event at time: 4
22	5	Processing a departure event at time: 6
24	5	Processing a departure event at time: 11
26	5	Processing a departure event at time: 16
28	5	Processing an arrival event at time: 20
30	5	Processing an arrival event at time: 22
88	3	Processing an arrival event at time: 24
		Processing a departure event at time: 25
		Processing an arrival event at time: 26
		Processing an arrival event at time: 28
		Processing an arrival event at time: 30

```
Processing a departure event at time:  30
Processing a departure event at time:  35
Processing a departure event at time:  40
Processing a departure event at time:  45
Processing a departure event at time:  50
Processing an arrival event at time:   88
Processing a departure event at time:  91
Simulation Ends

Final Statistics:

   Total number of people processed: 10
   Average amount of time spent waiting: 5.6
```

7. Modify and expand the event-driven simulation program that you wrote in Programming Problem 6.

a. Add an operation that displays the event list, and use it to check your hand trace in Exercise 11.

b. Add some statistics to the simulation. For example, compute the maximum wait in line, the average length of the line, and the maximum length of the line.

c. Modify the simulation so that it accounts for three tellers, each with a distinct line. You should keep in mind that there should be

- Three queues, one for each teller
- A rule that chooses a line when processing an arrival event (for example, enter the shortest line)
- Three distinct departure events, one for each line
- Rules for breaking ties in the event list

Run both this simulation and the original simulation on several sets of input data. How do the statistics compare?

d. The bank is considering the following change: Instead of having three distinct lines (one for each teller), there will be a single line for the three tellers. The person at the front of the line will go to the first available teller. Modify the simulation of part *c* to account for this variation. Run both simulations on several sets of input data. How do the various statistics compare (averages and maximums)? What can you conclude about having a single line as opposed to having distinct lines?

8. The people who run the Motor Vehicle Department (MVD) have a problem. They are concerned that people do not spend enough time waiting in lines to appreciate the privilege of owning and driving an automobile. The current arrangement is as follows:

- When people walk in the door, they must wait in a line to sign in.
- Once they have signed in, they are told either to stand in line for registration renewal or to wait until they are called for license renewal.
- Once they have completed their desired transaction, they must go and wait in line for the cashier.
- When they finally get to the front of the cashier's line, if they expect to pay by check, they are told that all checks must get approved. To do this, it is necessary to go to the check-approver's table and then reenter the cashier's line at the end.