Hand-Writing:

2. What would be the value of queues Q1 and Q2, and stack S after the following algorithm segment:

```
 1 S  = createStack
 2 Q1 = createQueue
 3 Q2 = createQueue
 4 enqueue (Q1, 5)
 5 enqueue (Q1, 6)
 6 enqueue (Q1, 9)
 7 enqueue (Q1, 0)
 8 enqueue (Q1, 7)
 9 enqueue (Q1, 5)
10 enqueue (Q1, 0)
11 enqueue (Q1, 2)
12 enqueue (Q1, 6)
13 loop (not emptyQueue (Q1))
   1 dequeue (Q1, x)
   2 if (x == 0)
     1 z = 0
     2 loop (not emptyStack (S))
       1 popStack (S, y)
       2 z = z + y
     3 end loop
     4 enqueue (Q2, z)
   3 else
     1 pushStack (S, x)
   4 end if
14 end loop
```

4. What would be the contents of queue Q1 and queue Q2 after the following code is executed and the following data are entered?

```
1 Q1 = createQueue
2 Q2 = createQueue
3 loop (not end of file)
  1 read number
  2 enqueue (Q1, number)
  3 enqueue (Q2, number)
  4 loop (not empty Q1)
    1 dequeue (Q1, x)
    2 enqueue (Q2, x)
  5 end loop
4 end loop
```

The data are 5, 7, 12, 4, 0, 4, 6.

10. Write a C function to implement Problem 9.

(Problem 9. is only for reference, No need to do it!)

9. Using only the algorithms in the queue ADT, write an algorithm called `catQueue` that concatenates two queues together. The second queue should be put at the end of the first queue.

Programming:

22. Using the C ADT, write a program that simulates the operation of a telephone system that might be found in a small business, such as your local pizza parlor. Only one person can answer the phone (a single-server queue), but there can be an unlimited number of calls waiting to be answered.

    Queue analysis considers two primary elements, the length of time a requester waits for service (the queue wait time—in this case, the customer calling for pizza) and the service time (the time it takes the customer to place the order). Your program should simulate the operation of the telephone and gather statistics during the process.

    The program requires two inputs to run the simulation: (1) the length of time in hours that the service is provided and (2) the maximum time it takes for the operator to take an order (the maximum service time).

    Four elements are required to run the simulation: a timing loop, a call simulator, a call processor, and a start call function.

    a. Timing loop: This is simply the simulation loop. Every iteration of the loop is considered 1 minute in real time. The loop continues until the service has been in operation the requested amount of time (see input above). When the operating period is complete, however, any waiting calls must be answered before ending the simulation. The timing loop has the following subfunctions:

       - Determine whether a call was received (call simulator)
       - Process active call
       - Start new call

       This sequence allows a call to be completed and another call to be started in the same minute.

    b. Call simulator: The call simulator uses a random-number generator to determine whether a call has been received. Scale the random number to an appropriate range, such as 1 to 10.

       The random number should be compared with a defined constant. If the value is less than the constant, a call was received; if it is not, no call was received. For the simulation set the call level to 50%; that is, on the average, a call is received every 2 minutes. If a call is received, place it in a queue.

c. **Process active call:** If a call is active, test whether it has been completed. If completed, print the statistics for the current call and gather the necessary statistics for the end-of-job report.
d. **Start new call:** If there are no active calls, start a new call if there is one waiting in the queue. Note that starting a call must calculate the time the call has been waiting.

During the processing, print the data shown in Table 4-2 after each call is completed. (*Note*: you will not get the same results.)

Clock time	Call number	Arrival time	Wait time	Start time	Service time	Queue size
4	1	2	0	2	3	2
6	2	3	2	5	2	4

TABLE 4-2 Sample Output for Project 22

At the end of the simulation, print out the following statistics gathered during the processing. Be sure to use an appropriate format for each statistic, such as a float for averages.

a. Total calls: calls received during operating period
b. Total idle time: total time during which no calls were being serviced
c. Total wait time: sum of wait times for all calls
d. Total service time: sum of service time for all calls
e. Maximum queue size: greatest number of calls waiting during simulation
f. Average wait time: total wait time/number of calls
g. Average service time: total service time/number of calls

Run the simulator twice. Both runs should simulate 2 hours. In the first simulation, use a maximum service time of 2 minutes. In the second run, use a maximum service time of 5 minutes.

25. Queues are commonly used in network systems. For example, e-mail is placed in queues while it is waiting to be sent and after it arrives at the recipient's mailbox. A problem occurs, however, if the outgoing mail processor cannot send one or more of the messages in the queue. For example, a message might not be sent because the recipient's system is not available.

    Write an e-mail simulator that processes mail at an average of 40 messages per minute. As messages are received, they are placed in a queue. For the simulation assume that the messages arrive at an average rate of 30 messages per minute. Remember, the messages must arrive randomly, so you need to use a random-number generator to determine when messages are received (see "Queue Simulation," starting on page 175).

    Each minute, you can dequeue up to 40 messages and send them. Assume that 25% of the messages in the queue cannot be sent in any processing cycle. Again, you need to use a random number to determine whether a given message can be sent. If it can't be sent, put it back at the end of the queue (enqueue it).

    Run the simulation for 24 hours, tracking the number of times each message had to be requeued. At the end of the simulation, print the statistics that show:

    a. The total messages processed
    b. The average arrival rate
    c. The average number of messages sent per minute
    d. The average number of messages in the queue in a minute
    e. The number of messages sent on the first attempt, the number sent on the second attempt, and so forth
    f. The average number of times messages had to be requeued (do not include the messages sent the first time in this average)

Ref: (Queue Simulation)

Queue Simulation

An important application of queues is **queue simulation**, a modeling activity used to generate statistics about the performance of queues. Let's build a model of a single-server queue—for example, a saltwater taffy store on a beach boardwalk. The store has one window, and a clerk can service only one customer at a time. The store also ships boxes of taffy anywhere in the country. Because there are many flavors of saltwater taffy and because it takes longer to serve a

customer who requests that the taffy be mailed, the time to serve customers varies between 1 and 10 minutes.

We want to study the store's activity over a hypothetical day. The store is open 8 hours per day, 7 days a week. To simulate a day, we build a model that runs for 480 minutes (8 hours times 60 minutes per hour).

The simulation uses a digital clock that lets events start and stop in 1- minute intervals. In other words, customers arrive on the minute, wait an integral number of minutes in the queue, and require an integral number of minutes to be served. In each minute of operation, the simulation needs to check three events: the arrival of customers, the start of customer processing, and the completion of customer processing.

Events

The arrival of a new customer is determined in a module we name *new customer*. To determine the arrival rate, the store owner used a stopwatch and studied customer patterns over several days. The owner found that, on average, a customer arrives every 4 minutes. We simulate an arrival rate using a random-number generator that returns a value between 1 and 4. If the number is 4, a customer has arrived. If the number is 1, 2, or 3, no customer has arrived.

We start processing a customer when the server is idle. In each minute of the simulation, therefore, the simulator needs to determine whether the clerk (the server) is busy or idle. In the simulation this is done with a module called *server free*. If the clerk is idle, the next waiting customer in line (the queue) can be served. If the clerk is busy, the waiting customers remain in the queue.

Finally, at the end of each minute, the simulation determines whether it has completed the processing for the current customer. The processing time for the current customer is determined by a random-number generator when the customer processing is started. Then, for each customer, we loop the required number of minutes to complete the transaction. When the customer has been completely served, we gather statistics about the sale and set the server to an idle state.

Data Structures

Four data structures are required for the queue simulation: a queue head, a queue node, a current customer status, and a simulation statistics structure. These structures are described below and shown in Figure 4-14.

Queue Head

We use the standard head node structure for the queue. It contains two node pointers—front and rear—and a count of the number of elements currently in the queue.

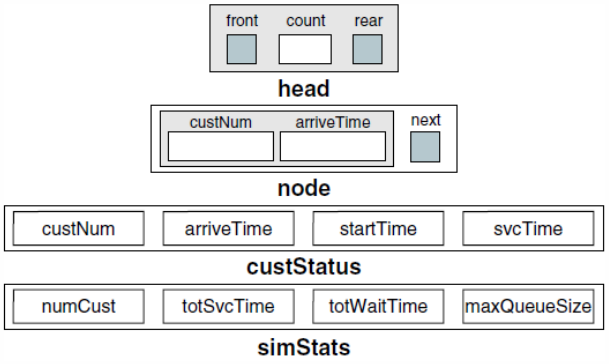

FIGURE 4-14 Queue Data Structures

Queue Node

The queue node contains only two elements: the customer data and a next node pointer. The customer data consist of a sequential customer number and the arrival time. The customer number is like the number you take when you go into a busy store with many customers waiting for service. The arrival time is the time the customer arrived at the store and got in line (was enqueued). The next node pointer is used to point to the next customer in line.

Customer Status

While we are processing the customer's order, we use a customer status structure to keep track of four pieces of data. First we store the customer's number and arrival time. Because we need to know what time we started processing the order, we store the start time. Finally, as we start the processing, we use a random-number generator to calculate and store the time it will take to fill the customer's order.

Simulation Statistics

At the conclusion of the simulation, we need to report the total number of customers processed in the simulation, the total and average service times, the total and average wait times, and the greatest number of customers in the queue at one time. We use a simulation statistics structure to store these data.

Output

At the conclusion of the simulation, we print the statistics gathered during the simulation along with the average queue wait time and average queue service time. To verify that the queue is working properly, we also print the following statistics each time a customer is completely served: arrival time, start time, wait time, service time, and the number of elements in the queue.

Simulation Algorithm

We are now ready to write the simulation algorithm. The design consists of a driver module that calls three processing modules, as shown in Figure 4-15. It parallels the requirements discussed earlier. New customer determines whether a customer has arrived and needs to be enqueued. Server free determines whether the server is idle, and if so, it starts serving the next customer, if one exists. Service complete determines whether the current customer has been completely served, and if so, collects the necessary statistical data and prints the data about the customer. At the end of the simulation, we print the statistics using the print stats module.

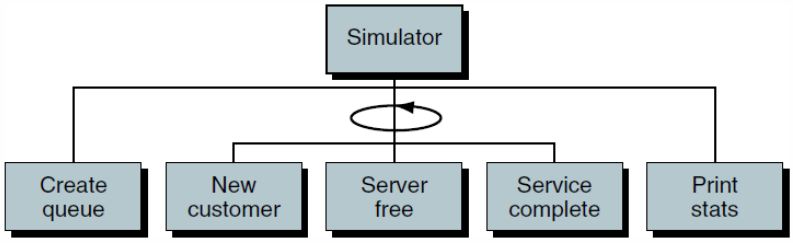

FIGURE 4-15 Design for Queue Simulation

Simulator

The pseudocode for the simulation driver is shown in Algorithm 4-11.

ALGORITHM 4-11 Queue Simulation: Driver

```
Algorithm taffySimulation
This program simulates a queue for a saltwater taffy store.
      Written by:
      Date:
 1 createQueue (queue)
 2 loop (clock <= endTime OR moreCusts)
   1  newCustomer  (queue, clock, custNum)
   2  serverFree   (queue, clock, custStatus, moreCusts)
   3  svcComplete  (queue, clock, custStatus,
                    runStats,     moreCusts)
   4  if (queue not empty)
      1  set moreCusts true
   5  end if
   6  increment clock
 3 end loop
 4 printStats (runStats)
end taffySimulation
```

Algorithm 4-11 Analysis

The driver creates the queue and then loops until the simulation is complete. Each loop tests to see whether a new customer needs to be enqueued, checks to see whether the server is idle so that a new customer can be started, and checks to see whether the current customer's service is complete.

To determine whether the simulation is complete, we need to test two conditions. We can only stop the simulation when we have run for the allocated time *and* there are no more customers. If there is more time left in the simulation, we are not finished. Even when we reach the end of the processing time, however, we are not finished if a customer is being served or if customers are waiting in the queue. We use a more customers flag to determine whether either of these conditions is true.

The server free logic sets the more customers flag to true when it starts a call (statement 2.2). The service complete logic sets it to false when it completes a call (statement 2.3). In the driver loop, we need to set it to true if there are calls waiting in the queue (statement 2.4). This test ensures that calls waiting in the queue are handled after the clock time has been exhausted.

At the end of the simulation, the driver calls a print algorithm that calculates the averages and prints all of the statistics.

New Customer

The pseudocode for new customer is shown in Algorithm 4-12.

ALGORITHM 4-12 Queue Simulation: New Customer

```
Algorithm  newCustomer (queue, clock, custNum)
This algorithm determines if a new customer has arrived.
  Pre    queue is a structure to a valid queue
         clock is the current clock minute
         custNum is the number of the last customer
  Post   if new customer has arrived, placed in queue
 1 randomly determine if customer has arrived
 2 if (new customer)
   1  increment custNum
   2  store custNum in custData
   3  store arrival time in custData
   4  enqueue (queue, custData)
 3 end if
end newCustomer
```

Algorithm 4-12 Analysis

It is important to note that we could not simply add a customer every fourth call. For the simulation study to work, the customers must arrive in a random fashion, not exactly every 4 minutes. By randomly determining when a customer has arrived (using a random-number generator), we may go several minutes without any customers and then have several customers arrive in succession. This random distribution is essential to queuing theory.

Server Free

The pseudocode to determine whether we can start serving a new customer is shown in Algorithm 4-13.

ALGORITHM 4-13 Queue Simulation: Server Free

```
Algorithm serverFree (queue, clock, status, moreCusts)
This algorithm determines if the server is idle and if so
starts serving a new customer.
  Pre   queue is a structure for a valid queue
        clock is the current clock minute
        status holds data about current/previous customer
  Post  moreCusts is set true if a call is started
1 if (clock > status startTime + status svcTime - 1)
    Server is idle.
  1  if (not emptyQueue (queue))
     1  dequeue (queue, custData)
     2  set status custNum    to custData number
     3  set status arriveTime to custData arriveTime
     4  set status startTime  to clock
     5  set status svcTime    to random service time
     6  set moreCusts true
  2  end if
2 end if
end serverFree
```

Algorithm 4-13 Analysis

The question in this algorithm is, "How can we tell if the server is free?" The customer status record holds the answer. It represents either the current customer or, if there is no current customer, the previous customer. If there is a current customer, we cannot start a new customer. We can tell if the record represents the new customer by comparing the clock time with the end time for the customer. The end time is the start time plus the required service time. If the clock time is greater than the calculated end time, then, if the queue is not empty, we start a new customer.

One question remains to be answered: "How do we start the first call?" When we create the customer status structure, we initialize everything to 0. Now, using the formula in statement 1, we see that a 0 start time plus a 0 service time minus 1 must be less than the clock time, so we can start the first customer.

Service Complete

The logic for service complete is shown in Algorithm 4-14.

ALGORITHM 4-14 Queue Simulation: Service Complete

```
Algorithm svcComplete (queue, clock, status,
                       stats, moreCusts)
This algorithm determines if the current customer's
processing is complete.
  Pre    queue is a structure for a valid queue
         clock is the current clock minute
         status holds data about current/previous customer
         stats contains data for complete simulation
```

continued

ALGORITHM 4-14 Queue Simulation: Service Complete *(continued)*

```
  Post   if service complete, data for current customer
         printed and simulation statistics updated
         moreCusts set to false if call completed
1 if (clock equal status startTime + status svcTime - 1)
  Current call complete
  1  set waitTime to status startTime - status arriveTime
  2  increment stats numCust
  3  set stats totSvcTime  to stats totSvcTime + status svcTime
  4  set stats totWaitTime to stats totWaitTime + waitTime
  5  set queueSize to queueCount (queue)
  6  if (stats maxQueueSize < queueSize)
     1  set stats maxQueueSize to queueSize
  7  end if
  8  print (status custNum   status arriveTime
            status startTime status svcTime
            waitTime         queueCount (queue))
  9  set    moreCusts to false
2 end if
end svcComplete
```

Algorithm 4-14 Analysis The question in this algorithm is similar to the server free question. To help analyze the logic, let's look at a few hypothetical service times for the simulation. They are shown in Table 4-1.

Start time	Service time	Time completed	Minutes served
1	2	2	1 and 2
3	1	3	3
4	3	6	4, 5, and 6
7	2	8	7 and 8

TABLE 4-1 Hypothetical Simulation Service Times

Because we test for a new customer in server free before we test for the customer being complete, it is possible to start and finish serving a customer who needs only 1 minute in the same loop. This is shown in minute 3 above. The correct calculation for the end of service time is therefore

```
start time + service time – 1
```

This calculation is borne out in each of the examples in Table 4-1. We therefore test for the clock time equal to the end of service time, and if they

are equal, we print the necessary data for the current customer and accumulate the statistics needed for the end of the simulation.

Print Stats

The last algorithm in the simulation prints the statistics for the entire simulation. The code is shown in Algorithm 4-15.

ALGORITHM 4-15 Queue Simulation: Print Statistics

```
Algorithm printStats (stats)
This algorithm prints the statistics for the simulation.
  Pre    stats contains the run statistics
  Post   statistics printed
 1 print (Simulation Statistics:)
 2 print ("Total customers: "    stats numCust)
 3 print ("Total service time: " stats totSvcTime)
 4 set avrgSvcTime to stats totSvcTime / stats numCust
 5 print ("Average service time: " avrgSvcTime)
 6 set avrgWaitTime to stats totWaitTime / stats numCust
 7 print ("Average wait time: " avrgWaitTime)
 8 print ("Maximum queue size: " stats maxQueueSize)
end printStats
```