8 Queues

In this chapter, we explore the queue, another ADT that has widespread use in computer science. We consider both a prototype and a professional version of the queue ADT. There are several implementation strategies for queues, some based on arrays and others based on linked structures. To illustrate the application of a queue, we develop a Case Study that simulates a supermarket checkout line. We close the chapter with an examination of a special kind of queue, known as a ***priority queue,*** and show how it is used in a second Case Study.

8.1 Overview of Queues

Like stacks, queues are linear collections. However, insertions are restricted to one end, called the ***rear,*** and removals to the other end, called the ***front.*** A queue thus supports a first-in first-out protocol (FIFO). Queues are omnipresent in everyday life and occur in any situation where people or things are lined up for processing on a first-come, first-served basis. Checkout lines in stores, highway tollbooth lines, and airport baggage check-in lines are familiar examples of queues.

Queues have two fundamental operations: ***enqueue,*** which adds an item to the rear of a queue and ***dequeue,*** which removes an item from the front. Figure 8.1 shows a queue as it might appear at various stages in its lifetime. In the figure, the queue's front is on the left and its rear is on the right.

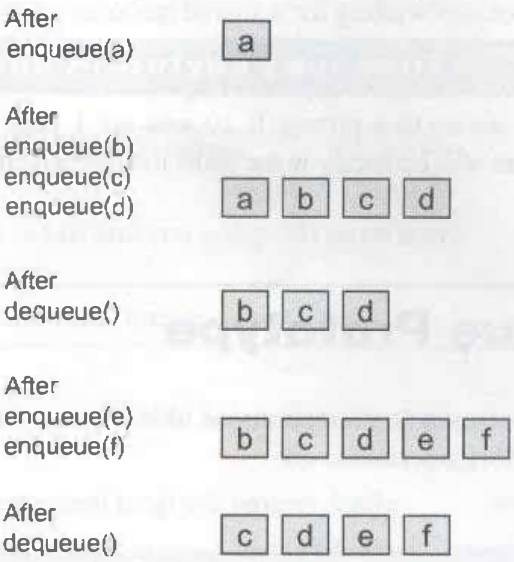

Figure 8.1 The states in the lifetime of a queue

Initially, the queue is empty. Then an item called a is enqueued. Next three more items called b, c, and d are enqueued, after which an item is dequeued, and so forth.

Related to queues is an ADT called a priority queue. In a queue, the item dequeued or served next is always the item that has been waiting the longest. But in some circumstances, this restriction is too rigid, and we would like to combine the idea of waiting with a notion of priority. The result is a priority queue, in which higher priority items are dequeued before those of lower priority, and items of equal priority are dequeued in FIFO order. Consider, for example, the manner in which passengers board an aircraft. The first-class passengers line up and board first, and the lower priority coach-class passengers line up and board second. However, this is not a true priority queue because once the first-class queue has emptied and the coach-class queue starts boarding, late arriving first-class passengers usually go to the end of the second queue. In a true priority queue, they would immediately jump ahead of all the coach-class passengers.

Most examples of queues in computer science involve scheduling access to shared resources, for instance:

CPU access	Processes are queued for access to a shared CPU.
Disk access	Processes are queued for access to a shared secondary storage device.
Printer access	Print jobs are queued for access to a shared laser printer.

Process scheduling can use either simple queues or priority queues. For example, processes involving keyboard input and screen output are often given higher priority access to the CPU than those that are computationally intensive. The result is that users, who tend to judge a computer's speed by its response time, are given the impression that the computer is fast.

Processes waiting for a shared resource can also be prioritized by their expected duration, with short processes given higher priority than longer ones, again with the intent of improving the apparent response time of a system. Imagine 20 print jobs queued up for access to a printer. If 19 jobs are 1 page long and 1 job is 200 pages long, more users will be happy if the short jobs are given higher priority and printed first.

8.2 A Queue Prototype

The fundamental operations of the queue ADT are `enqueue` and `dequeue`. The supporting operations are

`peek`	which returns the front item on a queue without removing it
`isEmpty`	which returns `true` is there are no more items on a queue
`isFull`	which returns `true` if a queue can hold no more items
`size`	which returns the number of items on a queue

As with stack operations, certain queue operations have preconditions which, if violated, result in exceptions. Here is a list of the exceptions thrown by the queue prototype:

`peek` and `dequeue`	throw an `IllegalStateException` if the queue is empty
`enqueue`	throws an `IllegalStateException` if the queue is full and an `IllegalArgumentException` if the item is `null`.

Table 8.1 defines the queue prototype's interface.

Table 8.1

The Interface for the Queue Prototype (QueuePT)

Fundamental Methods

`void` **`enqueue(Object item)`**
Adds an item to the rear of this queue. Throws an exception if the item is `null` or the queue is full.

`Object` **`dequeue()`**
Returns the item at the front of this queue and removes it from the queue. Throws an exception if the queue is empty.

Supporting Methods

`Object` **`peek()`**
Returns the item at the front of this queue without removing it from the queue. Throws an exception if the queue is empty.

Continues

Table 8.1 *(Continued)*

The Interface for the Queue Prototype (QueuePT)	
`boolean`	`isEmpty()` Returns `true` if this queue contains no items.
`boolean`	`isFull()` Returns `true` if this queue is full and can accept no more items.
`int`	`size()` Returns the number of items in this queue.

We then close the section with Table 8.2, a short illustration of the queue operations in action.

Table 8.2

The Effects of Queue Operations			
Operation	State of the Queue After the Operation	Value Returned	Comment
			Initially, the queue is empty.
`enqueue(a)`	`a`		The queue contains the single item `a`.
`enqueue (b)`	`a b`		`a` is at the front of the queue and `b` is at the rear.
`enqueue (c)`	`a b c`		`c` is added at the rear.
`isEmpty()`	`a b c`	`false`	The queue is not empty.
`size()`	`a b c`	`3`	The queue contains three items.
`peek()`	`a b c`	`a`	Return the front item on the queue without removing it.
`dequeue()`	`b c`	`a`	Remove the front item from the queue and return it. `b` is now the front item.
`dequeue ()`	`c`	`b`	Remove and return `b`.
`dequeue ()`		`c`	Remove and return `c`.
`isEmpty()`		`true`	The queue is empty.
`peek()`		`exception`	Peeking at an empty queue throws an exception.
`dequeue()`		`exception`	Trying to dequeue an empty queue throws an exception.
`enqueue(d)`	`d`		`d` is the front item.

8.3 Implementations of the Queue Prototype

Our approach to the implementation of queues is similar to the one we used for stacks. The structure of a queue lends itself to either an array implementation or a linked implementation. Because the linked implementation is somewhat more straightforward, we consider it first.

8.3.1 Linked Implementation

The linked implementations of stacks and queues have much in common. Both classes, `LinkedStackPT` and `LinkedQueuePT`, use a singly linked `Node` class to implement nodes. The operation `dequeue` is similar to `pop` in that it removes the first node from a linked list. However, `enqueue` and `push` differ. The operation `push` adds a node at the beginning of a linked list, whereas `enqueue` adds a node at the end. To provide fast access to both ends of a queue's linked list, there are pointers to both ends. Figure 8.2 shows a linked queue containing four items.

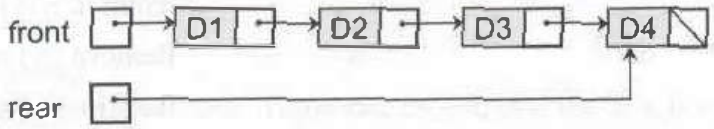

Figure 8.2 A linked queue with four items

Here is the code for the declarations of the instance variables `front` and `rear`:

```
private Node front;      // head node in the linked structure
private Node rear;       // tail node in the linked structure
```

During an enqueue operation, we create a new node, set the next pointer of the last node to the new node, and finally set the variable `rear` to the new node, as shown in Figure 8.3.

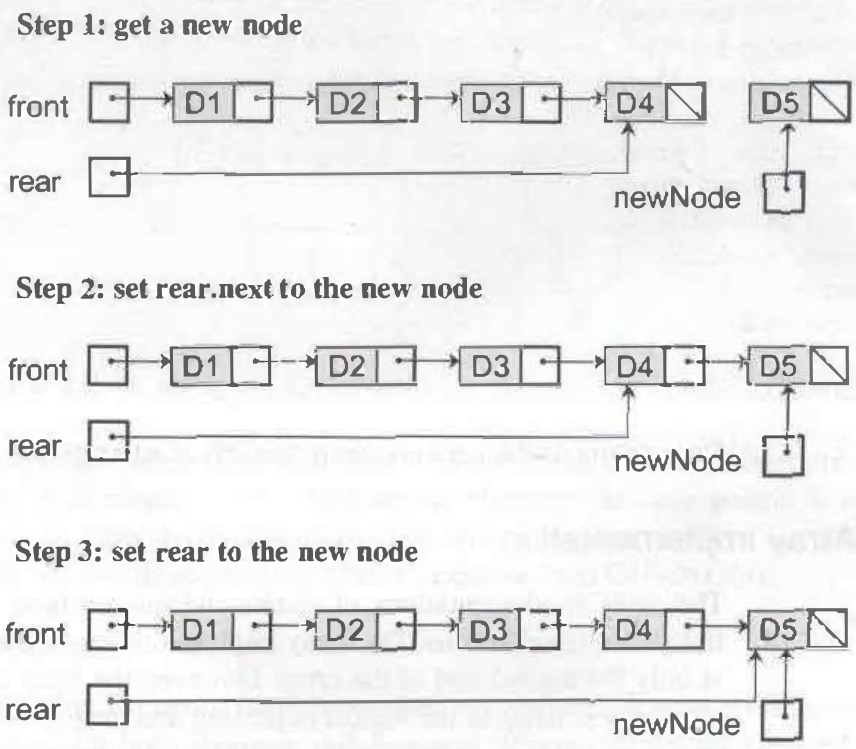

Figure 8.3 Adding an item to a linked queue

Here is code for the enqueue method:

```
public void enqueue(Object item){
   if (item == null)
      throw new IllegalArgumentException
               ("Trying to enqueue null onto the queue");

   Node node = new Node (item, null);
   if (isEmpty())
      front = node;
   else
      rear.next = node;
   rear = node;
   count++;
}
```

As mentioned earlier, dequeue is similar to pop. However, if the queue becomes empty after a dequeue operation, the front and rear pointers must both be set to null. Here is the code:

```
public Object dequeue(){
   if (isEmpty())
      throw new IllegalStateException ("Trying to dequeue an empty queue");

   Object item = front.value;
   front = front.next;
   if (front == null)
      rear = null;
   count--;
   return item;
}
```

Completion of the LinkedQueuePT is left as an exercise.

8.3.2 Array Implementation

The array implementations of stacks and queues have less in common than the linked implementations. The array implementation of a stack needs to access items at only the logical end of the array. However, the array implementation of a queue must access items at the logical beginning and the logical end. Doing this in a computationally effective manner is complex, so we approach the problem in a sequence of three attempts.

A First Attempt

Our first attempt at implementing a queue fixes the front of the queue at index position 0 and maintains an index variable, called rear, that points to the last item at position $n - 1$, where n is the number of items in the queue. A picture of such a queue, with four items in an array of six cells, is shown in Figure 8.4.

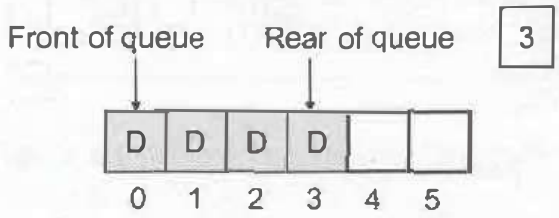

Figure 8.4 An array implementation of a queue with four items

For this implementation, enqueue operations are efficient; however, dequeue operations entail shifting all but the first item in the array to the left, which is an $O(n)$ process.

A Second Attempt

We can avoid `dequeue`'s linear behavior by not shifting items left each time the operation is applied. The modified implementation maintains a second index, called `front`, that points to the item at the front of the queue. The `front` pointer starts at 0 and advances through the array as items are dequeued. Figure 8.5 shows such a queue after five enqueue and two dequeue operations.

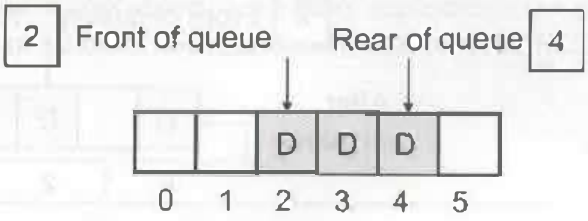

Figure 8.5 An array implementation of a queue with a front pointer

Notice that, in this scheme, cells to the left of the queue's `front` pointer are unused until we shift all elements left, which we do whenever the `rear` pointer is about to run off the end. Now the maximum running time of `dequeue` is $O(1)$, but at the cost of boosting the maximum running time of `enqueue` from $O(1)$ to $O(n)$.

A Third Attempt

By using a ***circular array implementation,*** we can simultaneously achieve good running times for both `enqueue` and `dequeue`. The implementation resembles the previous one in two respects:

1. The rear pointer starts at –1 and the front pointer starts at 0.
2. The front pointer chases the rear pointer through the array. During `enqueue`, the rear pointer moves further ahead of the front pointer, and during `dequeue`, the front pointer catches up by one position.

However, when either pointer is about to run off the end of the array, it is reset to 0. This has the effect of wrapping the queue around to the beginning of the array without the cost of moving any items.

As an example, let us assume that an array implementation uses six cells, that six items have been enqueued, and that two items have then been dequeued. According to our scheme, the next enqueue resets the rear pointer to 0. Figure 8.6 shows the state of the array before and after the rear pointer is reset to zero by the last enqueue operation.

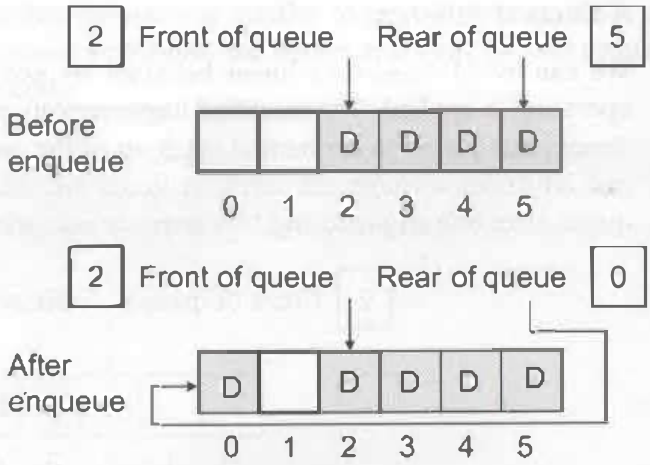

Figure 8.6 Wrapping data around a circular array implementation of a queue

The rear pointer now appears to chase the front pointer until the front pointer reaches the end of the array, at which point it too is reset to 0. As you can readily see, the maximum running times of both enqueue and dequeue are now O(1).

The alert reader will naturally wonder what happens when the queue becomes full and how the implementation can detect this condition. By maintaining a count of the items in the queue, we can determine if the queue is full or empty. When this count equals the size of the array, we know it's time to resize, assuming that we are using a dynamic array implementation.

After resizing, we would like the queue to occupy the initial segment of the array, with the front pointer set to 0. To achieve this, we consider two cases at the beginning of resizing process:

1. The front pointer is less than the rear pointer. In this case, we loop from `front` to `rear` in the original array and copy to positions 0 through `count - 1` in the new array.
2. The rear pointer is less than the front pointer. In this case, we loop from `front` to `count - 1` in the original array and copy to positions 0 through `count - front` in the new array. We then loop from 0 to `rear` in the original array and copy to positions `count - front + 1` to `count - 1` in the new array.

The resizing code for enqueue and dequeue is more complicated than the code for push and pop, but the process is still linear. Completion of the various implementations of class `ArrayQueuePT` is left as an exercise.

8.3.3 Time and Space Analysis for the Two Implementations

The time and space analysis for the two queue prototypes parallels that for the corresponding stack prototypes, so we do not dwell on the details. Consider first the

linked implementation of queues. The maximum running time of all methods is O(1), and the total space requirement is $2n + 2$, where n is the size of the queue.

For the circular array implementation of queues, if the array is static, then the maximum running time of all methods is O(1). If the array is dynamic, enqueue and dequeue jump to $O(n)$ anytime the array is resized, but retain an average running time of O(1). Space utilization for the array implementation again depends on the load factor. For load factors above ½, an array implementation makes more efficient use of memory than a linked implementation, and for load factors below ½, use is less efficient.

8.4 Two Applications of Queues

We now look briefly at two applications of queues: one involving computer simulations and the other round-robin CPU scheduling.

8.4.1 Simulations

Computer simulations are used to study the behavior of real-world systems, especially when it is impractical or dangerous to experiment with these systems directly. For example, a computer simulation could mimic traffic flows on a busy highway. Urban planners could then experiment with factors affecting traffic flows, such as the number and types of vehicles on the highway, the speed limits for different types of vehicles, the number of lanes in the highway, the frequency of tollbooths, etc. Outputs from such a simulation might include the total number of vehicles able to

move between designated points in a designated period and the average duration of a trip. By running the simulation with many combinations of inputs, the planners could determine how best to upgrade sections of the highway subject to the ever-present constraints of time, space, and money.

As a second example, consider the problem faced by the manager of a supermarket when trying to determine the number of checkout clerks to schedule at various times of the day. Some important factors in this situation are:

- the frequency with which new customers arrive
- the number of checkout clerks available
- the number of items in a customer's shopping cart
- the period of time considered

These factors could be inputs to a simulation program, which would then determine the total number of customers processed, the average time each customer waits for service, and the number of customers left standing in line at the end of the simulated time period. By varying the inputs, particularly the frequency of customer arrivals and the number of available checkout clerks, a simulation program could help the manager make effective staffing decisions for busy and slow times of the day. By adding an input that quantifies the efficiency of different checkout equipment, the manager can even decide whether it is cheaper to add more clerks or buy better equipment.

A common characteristic of both examples, and of simulation problems in general, is the moment-by-moment variability of essential factors. Consider the frequency of customer arrivals at checkout stations. If customers arrived at precise intervals, each with exactly the same number of items, it would be easy to determine how many clerks to have on duty. However, such regularity does not reflect the reality of a supermarket. Sometimes several customers show up at practically the same instant, and at other times no new customers arrive for several minutes. In addition, the number of items varies from customer to customer, and therefore, so does the amount of service required by each customer. All this variability makes it impossible to devise formulas to answer simple questions about the system, such as how customer waiting time varies with the number of clerks on duty. A simulation program, on the other hand, avoids the need for formulas by imitating the actual situation and collecting pertinent statistics.

Simulation programs use a simple technique to mimic variability. For instance, suppose new customers are expected to arrive on average once every 4 minutes. Then during each minute of simulated time, a program can generate a random number between 0 and 1. If the number is less than ¼, the program adds a new customer to a checkout line; otherwise, it does not. More sophisticated schemes based on probability distribution functions produce even more realistic results. Obviously, each time the program runs the results change slightly, but this only adds to the realism of the simulation.

Now let us discuss the common role played by queues in these examples. Both examples involve service providers and service consumers. In the first example, service providers include tollbooths and traffic lanes, and service consumers are the vehicles waiting at the tollbooths and driving in the traffic lanes. In the second example, clerks provide a service that is consumed by waiting customers. To emulate these conditions in a program, we associate each service provider with a queue of service consumers.

Simulations operate by manipulating these queues. At each tick of an imaginary clock, a simulation adds varying numbers of consumers to the queues and gives consumers at the head of each queue another unit of service. Once a consumer has received the needed quantity of service, it leaves the queue and the next consumer steps forward. During the simulation, the program accumulates statistics such as how many ticks each consumer waited in a queue and the percentage of time each provider is busy. The duration of a tick is chosen to match the problem being simulated. It could represent a millisecond, a minute, or a decade. In the program itself, a tick probably corresponds to one pass through the program's major processing loop.

Object-oriented languages are well suited to implementing simulation programs. For instance, in a supermarket simulation, each customer is an instance of a `Customer` class. A customer object keeps track of when the customer starts standing in line, when service is first received, and how much service is required. Likewise, a clerk is an instance of a `Clerk` class, and each clerk object contains a queue of customer objects. A simulator class coordinates the activities of the customers and clerks. At each clock tick, the simulation object

- generates new customer objects as appropriate
- assigns customers to cashiers
- tells each cashier to provide one unit of service to the customer at the bead of the queue

We develop a program based on these ideas in the chapter's first Case Study, and we will ask you to extend the program in a series of exercises.

8.4.2 Round-Robin CPU Scheduling

Most modern computers allow multiple processes to share a single CPU. There are various techniques for scheduling these processes. The most common, called round-robin scheduling, adds new processes to the end of a ready queue, which consists of processes waiting to use the CPU. Each process on the ready queue is dequeued in turn and given a slice of CPU time. When the time slice runs out, the process is returned to the rear of the queue, as shown in Figure 8.7.

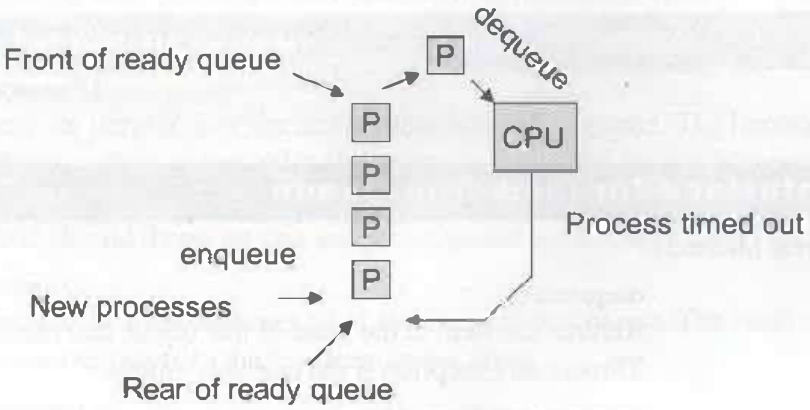

Figure 8.7 Scheduling processes for a CPU

Generally, not all processes need the CPU with equal urgency. For instance, user satisfaction with a computer is greatly influenced by the computer's response time to keyboard and mouse inputs; thus, it makes sense to give precedence to processes handling these inputs. Round-robin scheduling adapts to this requirement by using a priority queue and assigning each process an appropriate priority. As a follow-up to this discussion, the chapter's second Case Study shows how a priority queue can be used to schedule patients in an emergency room.

8.8 Priority Queues

As mentioned earlier, the queue ADT can be extended to the notion of a priority queue. When items are added to a priority queue, they are assigned a rank order. When they are dequeued, items of higher priority are removed before those of lower priority. Items of equal priority are dequeued in the usual FIFO order. `lamborne` includes a `PriorityQueue` interface that extends the `Queue` interface through the addition of a single method

```
void enqueue(Object item, int priority)
```

where `priority` is a positive integer representing the priority of the item being enqueued. Items can still be enqueued using the method

```
void enqueue(Object item)
```

and they are assigned a default priority of one.

The `lamborne` package includes two implementation of `PriorityQueue`. These are called `HeapPriorityQueue` and `LinkedPriorityQueue`. The first is dis-

cussed in Chapter 12, and the second is examined shortly. The constructor for LinkedPriorityQueue bas form

```
LinkedPriorityQueue(int maxPriority)
```

where maxPriority is a positive integer indicating the highest priority to be used with this priority queue.

The next code segment shows how to declare and use a priority queue.

```
// Create a new priority queue to hold items with priorities 1 and 2
PriorityQueue q = new LinkedPriorityQueue (2);

// Add the integers 1,2,3 to the queue with a priority of 1
for (int i = 1; i <= 3; i++)
   q.enqueue(new Integer(i), 1);

// Add the integers 10,11,12 to the queue with a priority of 2
for (i = 10; i <= 12; i++)
   q.enqueue(new Integer(i), 2);

// Dequeue and display all integers in the queue
while (! q.isEmpty())
   System.out.println(((Integer)q.dequeue()).intValue());
```

In this example, integers go onto the priority queue in the order 1, 2, 3, 10, 11, 12, but leave in the order 10, 11, 12, 1, 2, 3, thus demonstrating the role played by priorities.

8.8.1 Implementation of a Linked Priority Queue

We now discuss the implementation of a linked priority queue. For the sake of simplicity, we will do so in the context of a prototype. Table 8.9 gives the interface:

Table 8.9

The Interface for the Priority Queue Prototype (PriorityQueuePT)

void	**enqueue(Object item)** Adds an item to the tail of the subqueue with priority 1. Throws an exception if the item is null.
void	**enqueue(Object item, int priority)** Adds an item to the tail of the subqueue with the specified priority. Throws an exception if the item is null or the priority is not between 1 and the maximum priority allowed for this priority queue.

The linked implementation is based on an array of subqueues, and an item with priority n goes on the queue at index position $n - 1$. Figure 8.12 shows a priority queue with four priorities.

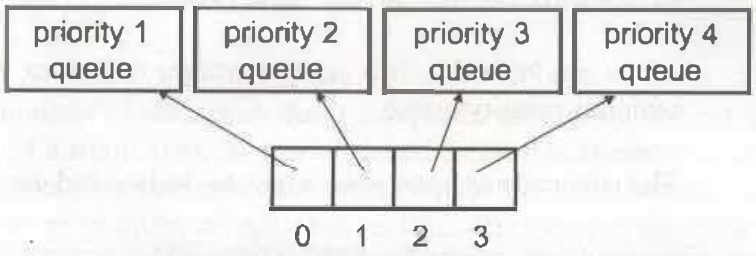

Figure 8.12 A linked priority queue with four priorities

We call this a linked implementation because each array element contains a reference to a queue. The completion of the linked implementation of priority queues is left as an exercise.