Observe that the output consists of the following seven moves:

A → C, A → B, C → B, A → C, B → A, B → C, A → C,

This agrees with the solution in Fig. 6.15.

Summary

The Towers of Hanoi problem illustrates the power of recursion in the solution of various algorithmic problems. This section has shown how to implement recursion by means of stacks when using a programming language—notably FORTRAN or COBOL—which does not allow recursive programs. In fact, even when using a programming language—such as Pascal—which does support recursion, the programmer may want to use the nonrecursive solution, since it may be much less expensive than using the recursive solution.

6.10 QUEUES

A *queue* is a linear list of elements in which deletions can take place only at one end, called the *front*, and insertions can take place only at the other end, called the *rear*. The terms "front" and "rear" are used in describing a linear list only when it is implemented as a queue.

Queues are also called first-in first-out (FIFO) lists, since the first element in a queue will be the first element out of the queue. In other words, the order in which elements enter a queue is the order in which they leave. This contrasts with stacks, which are last-in first-out (LIFO) lists.

Queues abound in everyday life. The automobiles waiting to pass through an intersection form a queue, in which the first car in line is the first car through; the people waiting in line at a bank form a queue, where the first person in line is the first person to be waited on; and so on. An important example of a queue in computer science occurs in a timesharing system, in which programs with the same priority form a queue while waiting to be executed. (Another structure, called a priority queue, is discussed in Sec. 6.13.)

Example 6.10

Figure 6.19(a) is a schematic diagram of a queue with 4 elements; where AAA is the front element and DDD is the rear element. Observe that the front and rear elements of the queue are also, respectively, the first and last elements of the list. Suppose an element is deleted from the queue. Then it must be AAA. This yields the queue in Fig. 6.19(b), where BBB is now the front element. Next, suppose EEE is added to the queue and then FFF is added to the queue. Then they must be added at the rear of the queue, as pictured in Fig. 6.19(c). Note that FFF is now the rear element. Now suppose another element is deleted from the queue; then it must be BBB, to yield the queue in Fig. 6.19(d). And so on. Observe that in such a data structure, EEE will be deleted before FFF because it has been placed in the queue before FFF. However, EEE will have to wait until CCC and DDD are deleted.

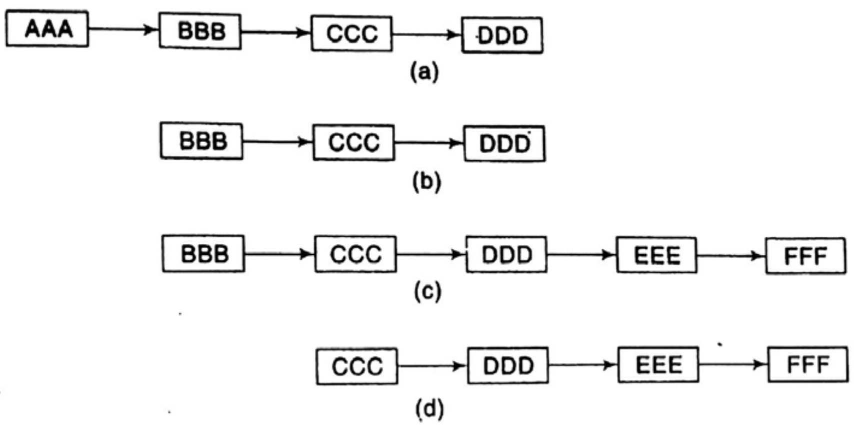


Fig. 6.19

Representation of Queues

Queues may be represented in the computer in various ways, usually by means at one-way lists or linear arrays. Unless otherwise stated or implied, each of our queues will be maintained by a linear array QUEUE and two pointer variables: FRONT, containing the location of the front element of the queue; and REAR, containing .the location of the rear element of the queue. The condition FRONT = NULL will indicate that the queue is empty.

Figure 6.20 shows the way the array in Fig. 6.19 will be stared in memory using an array QUEUE with N elements. Figure 6.20 also indicates the way elements will be deleted from the queue and the way new elements will be added to the queue. Observe that whenever an element is deleted from the queue, the value of FRONT is increased by 1; this can be implemented by the assignment

FRONT := FRONT + 1

Similarly, whenever an element is added to the queue, the value of REAR is increased by 1; this can be implemented by the assignment

REAR := REAR + 1

This means that after N insertions, the rear element of the queue will occupy QUEUE[N] or, in other words; eventually the queue will occupy the last part of the array. This occurs even though the queue itself may not contain many elements.

Suppose we want to insert an element ITEM into a queue at the time the queue does occupy the last part of the array, i.e., when REAR = N. One way to do this is to simply move the entire queue to the beginning of the array, changing FRONT and REAR accordingly, and then inserting ITEM as above. This procedure may be very expensive. The procedure we adopt is to assume that the array QUEUE is circular, that is, that QUEUE[1] comes after QUEUE[N] in the array. With this assumption, we insert ITEM into the queue by assigning ITEM to QUEUE[1]. Specifically, instead of increasing REAR to N + 1, we reset REAR = 1 and then assign

QUEUE[REAR] := ITEM

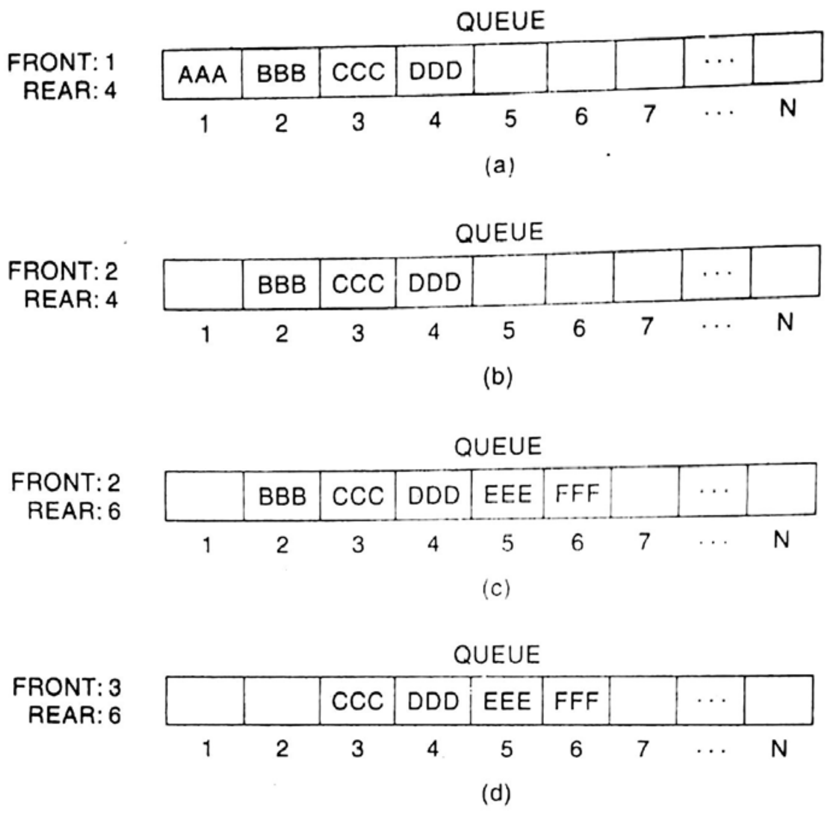


Fig. 6.20 *Array Representation of a Queue*

Similarly, if FRONT = N and an element of QUEUE is deleted, we reset FRONT = 1 instead of increasing FRONT to N + 1. (Some readers may recognize this as modular arithmetic, discussed in Sec. 2.2.)

Suppose that our queue contains only one element, i.e., suppose that

$$\text{FRONT} = \text{REAR} \neq \text{NULL}$$

and suppose that the element is deleted. Then we assign

$$\text{FRONT} := \text{NULL} \quad \text{and} \quad \text{REAR} := \text{NULL}$$

to indicate that the queue is empty.

Example 6.11

Figure 6.21 shows how a queue may be maintained by a circular array QUEUE with N = 5 memory locations. Observe that the queue always occupies consecutive locations except when it occupies locations at the beginning and at the end of the array. If the queue is viewed as a circular array, this means that it still occupies consecutive locations. Also, as indicated by Fig. 6.21(m), the queue will be empty only when FRONT = REAR and an element is deleted. For this reason, NULL is assigned to FRONT and REAR in Fig. 6.21(m).

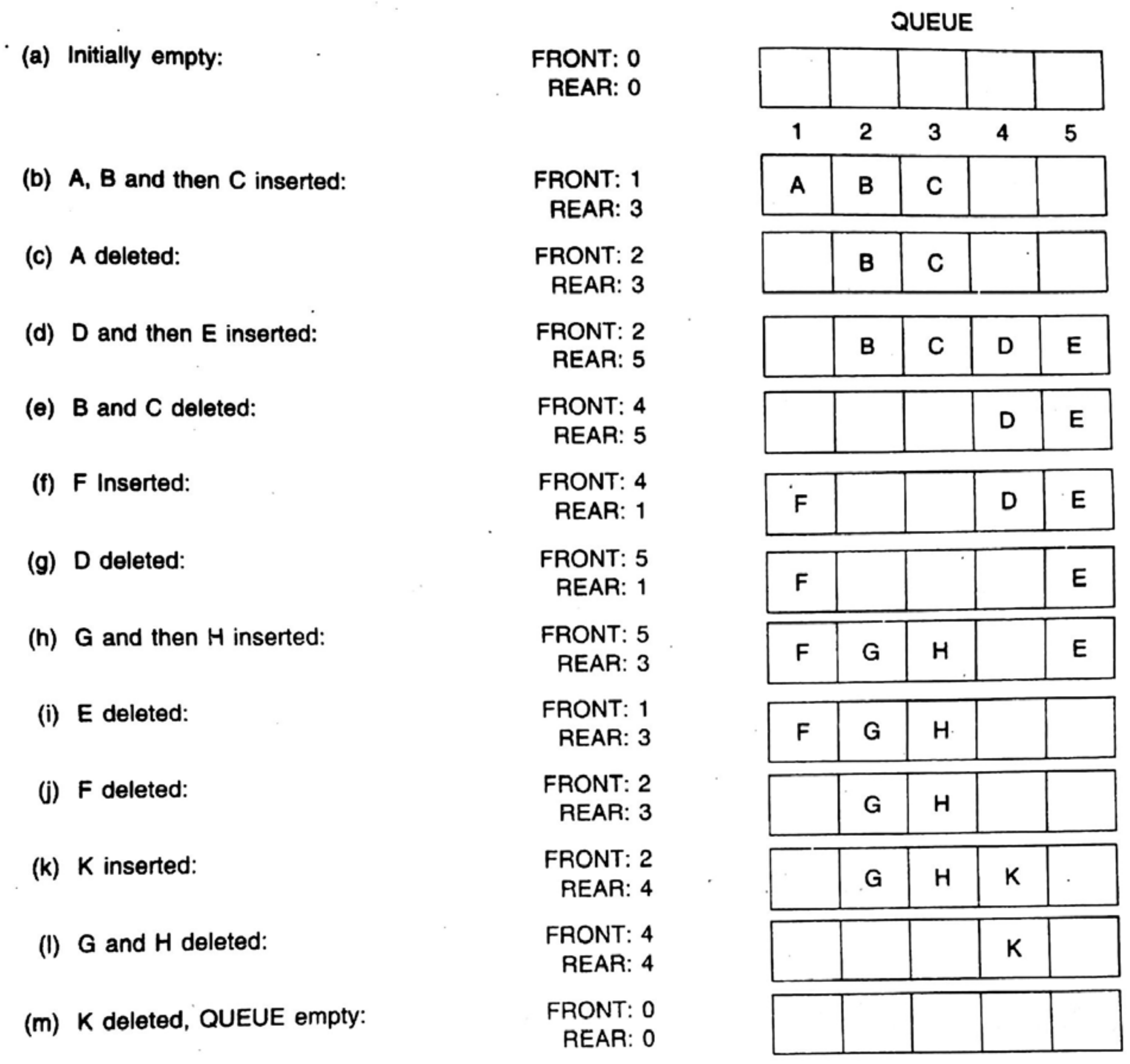

QUEUE
(a) Initially empty: FRONT: 0 REAR: 0
1 2 3 4 5
(b) A, B and then C inserted: FRONT: 1 REAR: 3
A B C
(c) A deleted: FRONT: 2 REAR: 3
B C
(d) D and then E inserted: FRONT: 2 REAR: 5
B C D E
(e) B and C deleted: FRONT: 4 REAR: 5
D E
(f) F Inserted: FRONT: 4 REAR: 1
F D E
(g) D deleted: FRONT: 5 REAR: 1
F E
(h) G and then H inserted: FRONT: 5 REAR: 3
F G H E
(i) E deleted: FRONT: 1 REAR: 3
F G H
(j) F deleted: FRONT: 2 REAR: 3
G H
(k) K inserted: FRONT: 2 REAR: 4
G H K
(l) G and H deleted: FRONT: 4 REAR: 4
K
(m) K deleted, QUEUE empty: FRONT: 0 REAR: 0

Fig. 6.21

We are now prepared to formally state our procedure QINSERT (Procedure 6.13), which inserts a data ITEM into a queue. The first thing we do in the procedure is to test for overflow, that is, to test whether or not the queue is filled.

Next we give a procedure QDELETE (Procedure 6.14), which deletes the first element from a queue, assigning it to the variable ITEM. The first thing we do is to test for underflow, i.e., to test whether or not the queue is empty.

```
Procedure 6.13: QINSERT(QUEUE, N, FRONT, REAR, ITEM)
    This procedure inserts an element ITEM into a queue.

    1. [Queue already filled?]
       If FRONT = 1 and REAR = N, or if FRONT = REAR + 1, then:
           Write: OVERFLOW, and Return.
```

```
2. [Find new value of REAR.]
   If FRONT := NULL, then: [Queue initially empty.]
       Set FRONT := 1 and REAR := 1.
   Else if REAR = N, then:
       Set REAR := 1.
   Else:
       Set REAR := REAR + 1.
   [End of If structure.]
3. Set QUEUE[REAR] := ITEM. [This inserts new element.]
4. Return.
```

Procedure 6.14: QDELETE(QUEUE, N, FRONT, REAR, ITEM)
This procedure deletes an element from a queue and assigns it to the variable ITEM.

```
1. [Queue already empty?]
   If FRONT := NULL, then: Write: UNDERFLOW, and Return.
2. Set ITEM := QUEUE[FRONT].
3. [Find new value of FRONT.]
     If FRONT = REAR, then: [Queue has only one element to start.]
         Set FRONT := NULL and REAR := NULL.
     Else if FRONT = N, then:
         Set FRONT := 1.
     Else:
         Set FRONT := FRONT + 1.
   [End of If structure.]
4. Return.
```

6.11 LINKED REPRESENTATION OF QUEUES

In this section we discuss the linked representation of a queue. A linked queue is a queue implemented as a linked list with two pointer variables FRONT and REAR pointing to the nodes which is in the FRONT and REAR of the queue. The INFO fields of the list hold the elements of the queue and the LINK fields hold pointers to the neighboring elements in the queue. Fig. 6.22 illustrates the linked representation of the queue shown in Fig. 6.16(a).

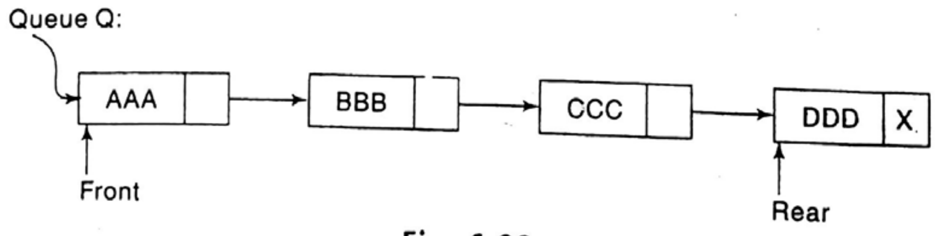


Fig. 6.22

In the case of insertion into a linked queue, a node borrowed from the AVAIL list and carrying the item to be inserted is added as the last node of the linked list representing the queue. The REAR pointer is updated to point to the last node just added to the list. In the case of deletion, the first node of the list pointed to by FRONT is deleted and the FRONT pointer is updated to point to the next node in the list. Fig. 6.23 and Fig. 6.24 illustrate the insert and delete operations on the queue shown in Fig. 6.22.

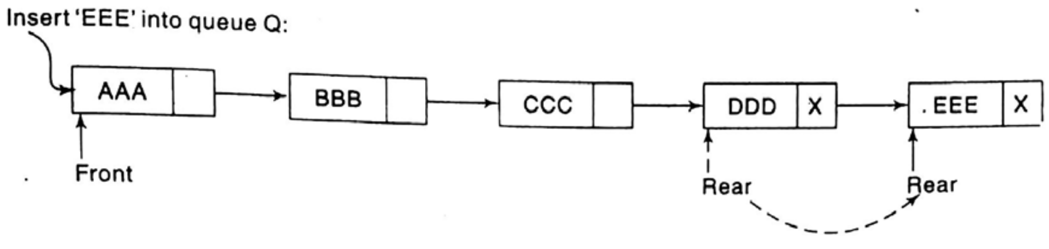


Fig. 6.23

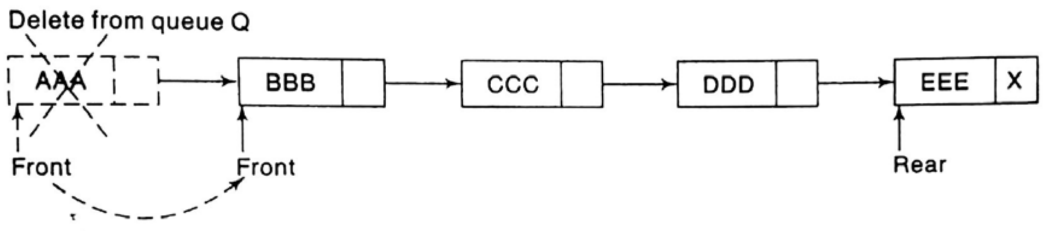


Fig. 6.24

The array representation of a queue suffers from the drawback of limited queue capacity. This in turn calls for the checking of OVERFLOW condition every time an insertion is made into the queue. Also, due to the inherent disadvantage of the array data structure in which data movement is expensive, the maintenance of the queue calls for its circular implementation.

In contrast, the linked queue is not limited in capacity and therefore as many nodes as the AVAIL list can provide, may be inserted into the queue. This dispenses with the need to check for the OVERFLOW condition during insertion. Also, unlike the array representation, the linked queue functions as a linear queue and there is no need to view it as 'circular' for efficient management of space.

Procedure 6.15: LINKQ_INSERT(INFO,LINK, FRONT, REAR,AVAIL,ITEM)
This procedure inserts an ITEM into a linked queue

```
1. [Available space?] If AVAIL = NULL, then Write
                      OVERFLOW and Exit
2. [Remove first node from AVAIL list]
       Set NEW := AVAIL and AVAIL := LINK[AVAIL]
3. Set INFO[NEW] := ITEM and LINK[NEW]=NULL
                      [Copies ITEM into new node]
```

```
4. If (FRONT = NULL) then FRONT = REAR = NEW
                  [If Q is empty then ITEM is the first element in the queue Q]
   else set LINK[REAR] := NEW and REAR = NEW
                  [REAR points to the new node appended to
                   the end of the list]
5. Exit.
```

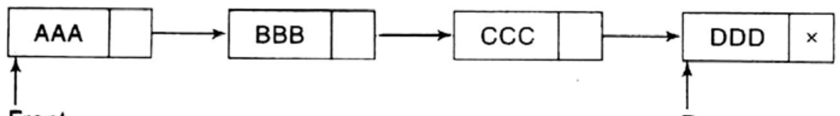
Procedure 6.16: LINKQ_DELETE (INFO, LINK, FRONT, REAR, AVAIL, ITEM)
This procedure deletes the front element of the linked queue and stores it in ITEM

```
1. [Linked queue empty?] if (FRONT = NULL) then Write: UNDERFLOW
                         and Exit
2. Set TEMP = FRONT [If linked queue is nonempty, remember FRONT in
                     a temporary variable TEMP]
3. ITEM = INFO (TEMP)
4. FRONT = LINK (TEMP) [Reset FRONT to point to the next element in
                        the queue]
5. LINK(TEMP) =AVAIL and AVAIL=TEMP [return the deleted node
                                     TEMP to the AVAIL list]
6. Exit.
```

Example 6.12

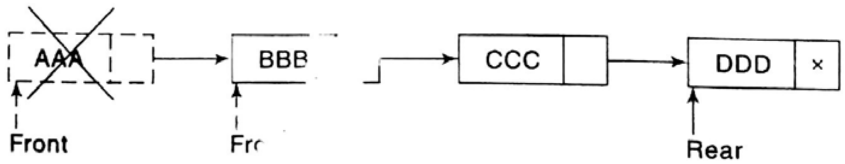
For the linked queue shown in Fig. 6.22, the snapshots of the queue structure after the execution of the following operations are shown in Fig. 6.25.
(i) Delete (ii) Delete (iii) Insert FFF

Original linked queue:



(a)

(i) Delete



(b)

(ii) Delete

(iii) Insert FFF

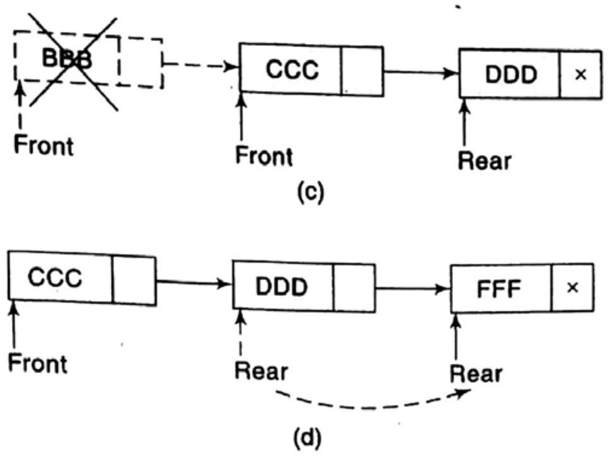


Fig. 6.25

6.12 DEQUES

A *deque* (pronounced either "deck" or "dequeue") is a linear list in which elements can be added or removed at either end but not in the middle. The term deque is a contraction of the name *double-ended queue.*

There are various ways of representing a deque in a computer. Unless it is otherwise stated or implied, we will assume our deque is maintained by a circular array DEQUE with pointers LEFT and RIGHT, which point to the two ends of the deque. We assume that the elements extend from the left end to the right end in the array. The term "circular" comes from the fact that we assume that DEQUE[1] comes after DEQUE[N] in the array. Figure 6.26 pictures two deques, each with 4 elements maintained in an array with N = 8 memory locations. The condition LEFT = NULL will be used to indicate that a deque is empty.

There are two variations of a deque—namely, an input-restricted deque and an output-restricted deque—which are intermediate between a deque and a queue. Specifically, an *input-restricted deque* is a deque which allows insertions at only one end of the list but allows deletions at both ends of the list; and an *output-restricted deque* is a deque which allows deletions at only one end of the list but allows insertions at both ends of the list.

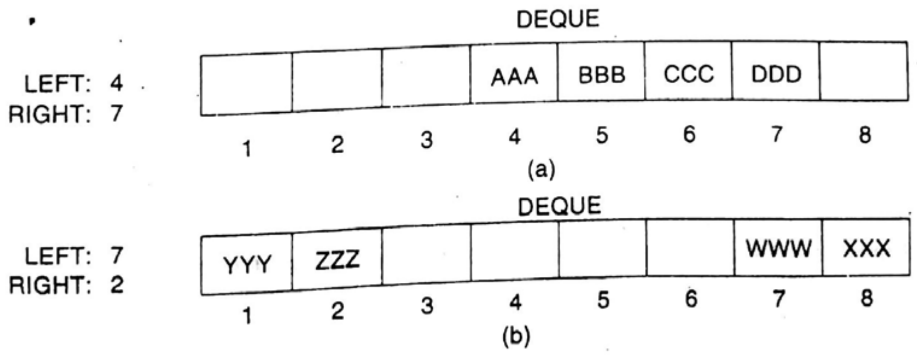


Fig. 6.26

The procedures which insert and delete elements in deques and the variations on those procedures are given as supplementary problems. As with queues, a complication may arise (a) when there is overflow, that is, when an element is to be inserted into a deque which is already full, or (b) when there is underflow, that is, when an element is to be deleted from a deque which is empty. The procedures must consider these possibilities.

6.13 PRIORITY QUEUES

A *priority queue* is a collection of elements such that each element has been assigned a priority and such that the order in which elements are deleted and processed comes from the following rules:

(1) An element of higher priority is processed before any element of lower priority.
(2) Two elements with the same priority are processed according to the order in which they were added to the queue.

A prototype of a priority queue is a timesharing system: programs of high priority are processed first, and programs with the same priority form a standard queue.

There are various ways of maintaining a priority queue in memory. We discuss two of them here: one uses a one-way list, and the other uses multiple queues. The ease or difficulty in adding elements to or deleting them from a priority queue clearly depends on the representation that one chooses.

One-Way List Representation of a Priority Queue

One way to maintain a priority queue in memory is by means of a one-way list, as follows:

(a) Each node in the list will contain three items of information: an information field INFO, a priority number PRN and a link number LINK.
(b) A node X precedes a node Y in the list (1) when X has higher priority than Y or (2) when both have the same priority but X was added to the list before Y. This means that the order in the one-way list corresponds to the order of the priority queue.

Priority numbers will operate in the usual way: the lower the priority number, the higher the priority.

Example 6.13

Figure 6.27 shows a schematic diagram of a priority queue with 7 elements. The diagram does not tell us whether BBB was added to the list before or after DDD. On the other hand, the diagram does tell us that BBB was inserted before CCC, because BBB and CCC have the same priority number and BBB appears before CCC in the list. Figure 6.28 shows the way the priority queue may appear in memory using linear arrays INFO, PRN and LINK. (See Sec. 5.2.)

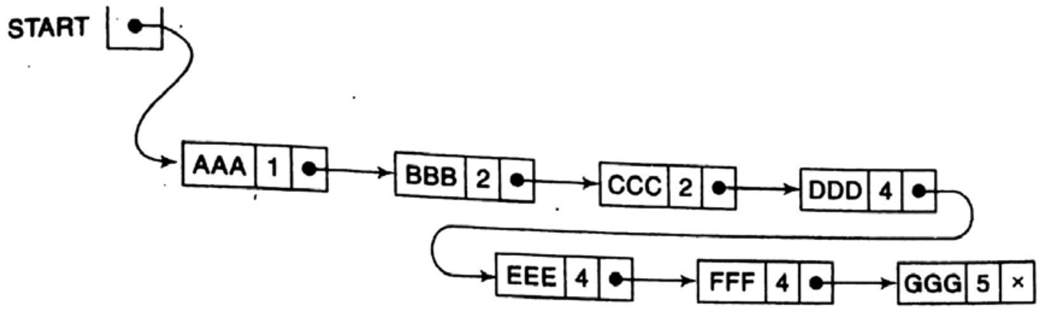
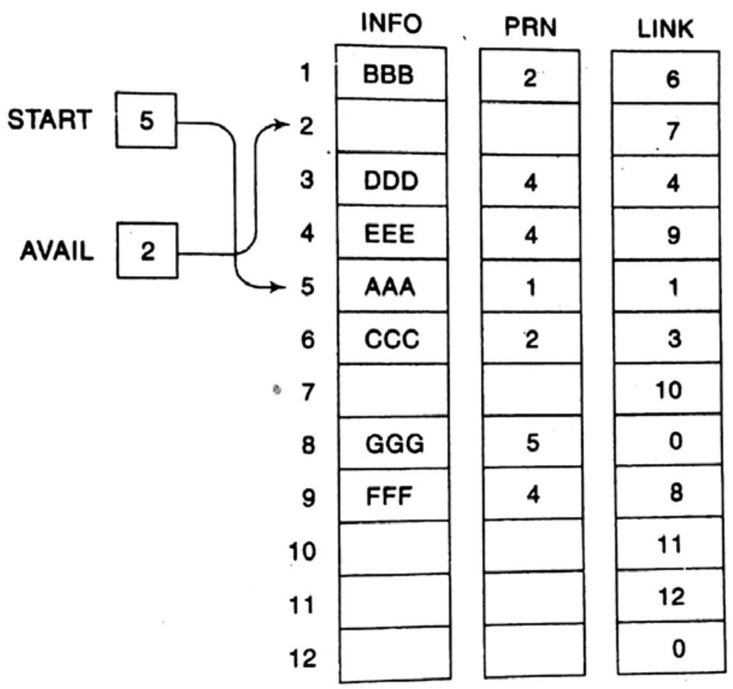


Fig. 6.27



	INFO	PRN	LINK
1	BBB	2	6
2			7
3	DDD	4	4
4	EEE	4	9
5	AAA	1	1
6	CCC	2	3
7			10
8	GGG	5	0
9	FFF	4	8
10			11
11			12
12			0

Fig. 6.28

The main property of the one-way list representation of a priority queue is that the element in the queue that should be processed first always appears at the beginning of the one-way list. Accordingly, it is a very simple matter to delete and process an element from our priority queue. The outline of the algorithm follows.

Algorithm 6.17: This algorithm deletes and processes the first element in a priority queue which appears in memory as a one-way list.

```
1. Set ITEM := INFO[START]. [This saves the data in the first node.]
2. Delete first node from the list.
3. Process ITEM.
4. Exit.
```

The details of the algorithm, including the possibility of underflow, are left as an exercise.

Adding an element to our priority queue is much more complicated than deleting an element from the queue, because we need to find the correct place to insert the element. An outline of the algorithm follows.

Algorithm 6.18: This algorithm adds an ITEM with priority number N to a priority queue which is maintained in memory as a one-way list.

(a) Traverse the one-way list until finding a node X whose priority number exceeds N. Insert ITEM in front of node X.

(b) If no such node is found, insert ITEM as the last element of the list.

The above insertion algorithm may be pictured as a weighted object "sinking" through layers of elements until it meets an element with a heavier weight.

The details of the above algorithm are left as an exercise. The main difficulty in the algorithm comes from the fact that ITEM is inserted before node X. This means that, while traversing the list, one must also keep track of the address of the node preceding the node being accessed.

Example 6.14

Consider the priority queue in Fig. 6.27. Suppose an item XXX with priority number 2 is to be inserted into the queue. We traverse the list, comparing priority numbers. Observe that DDD is the first element in the list whose priority number exceeds that of XXX. Hence XXX is inserted in the list in front of DDD, as pictured in Fig. 6.29. Observe that XXX comes after BBB and CCC, which have the same priority as XXX. Suppose now that an element is to be deleted from the queue. It will be AAA, the first element in the list. Assuming no other insertions, the next element to be deleted will be BBB, then CCC, then XXX, and so on.

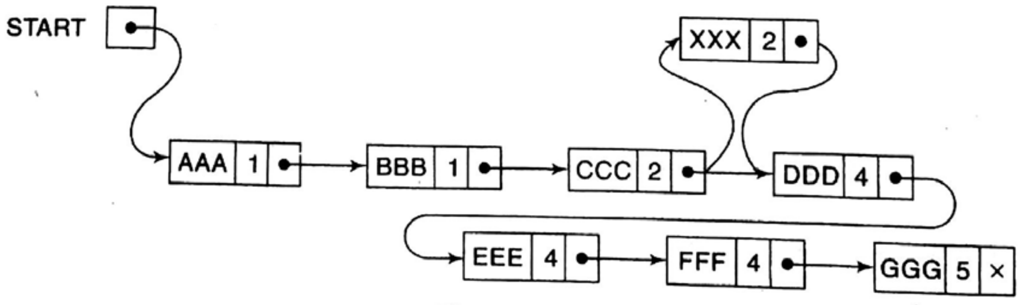


Fig. 6.29

Array Representation of a Priority Queue

Another way to maintain a priority queue in memory is to use a separate queue for each level of priority (or for each priority number). Each such queue will appear in its own circular array and must have its own pair of pointers, FRONT and REAR. In fact, if each queue is allocated the same amount of space, a two-dimensional array QUEUE can be used instead of the linear arrays. Figure 6.30 indicates this representation for the priority queue in Fig. 6.29. Observe that FRONT[K] and REAR[K] contain, respectively, the front and rear elements of row K of QUEUE, the row that maintains the queue of elements with priority number K.

	FRONT	REAR
1	2	2
2	1	3
3	0	0
4	5	1
5	4	4

$$\begin{array}{c|cccccc} & 1 & 2 & 3 & 4 & 5 & 6 \\ \hline 1 & & \text{AAA} & & & & \\ 2 & \text{BBB} & \text{CCC} & \text{XXX} & & & \\ 3 & & & & & & \\ 4 & \text{FFF} & & & & \text{DDD} & \text{EEE} \\ 5 & & & & \text{GGG} & & \end{array}$$

Fig. 6.30

The following are outlines of algorithms for deleting and inserting elements in a priority queue that is maintained in memory by a two-dimensional array QUEUE, as above. The details of the algorithms are left as exercises.

Algorithm 6.19: This algorithm deletes and processes the first element in a priority queue maintained by a two-dimensional array QUEUE.

```
1. [Find the first nonempty queue.]
   Find the smallest K such that FRONT[K] ≠ NULL.
2. Delete and process the front element in row K of QUEUE.
3. Exit.
```

Algorithm 6.20: This algorithm adds an ITEM with priority number M to a priority queue maintained by a two-dimensional array QUEUE.

```
1. Insert ITEM as the rear element in row M of QUEUE.
2. Exit.
```

Summary

Once again we see the time-space tradeoff when choosing between different data structures for a given problem. The array representation of a priority queue is more time-efficient than the one-way list. This is because when adding an element to a one-way list, one must perform a linear search on the list. On the other hand, the one-way list representation of the priority queue may be more space-efficient than the array representation. This is because in using the array representation, overflow occurs when the number of elements in any single priority level exceeds the capacity for that level, but in using the one-way list, overflow occurs only when the total number of elements exceeds the *total* capacity. Another alternative is to use a linked list for each priority level.