Example 5.1

A hospital ward contains 12 beds, of which 9 are occupied as shown in Fig. 5.3. Suppose we want an alphabetical listing of the patients. This listing may be given by the pointer field, called Next in the figure. We use the variable START to point to the first patient. Hence START contains 5, since the first patient, Adams, occupies bed 5. Also, Adams's pointer is equal to 3, since Dean, the next patient, occupies bed 3; Dean's pointer is 11, since Fields, the next patient, occupies bed 11; and so on. The entry for the last patient (Samuels) contains the null pointer, denoted by 0. (Some arrows have been drawn to indicate the listing of the first few patients.)

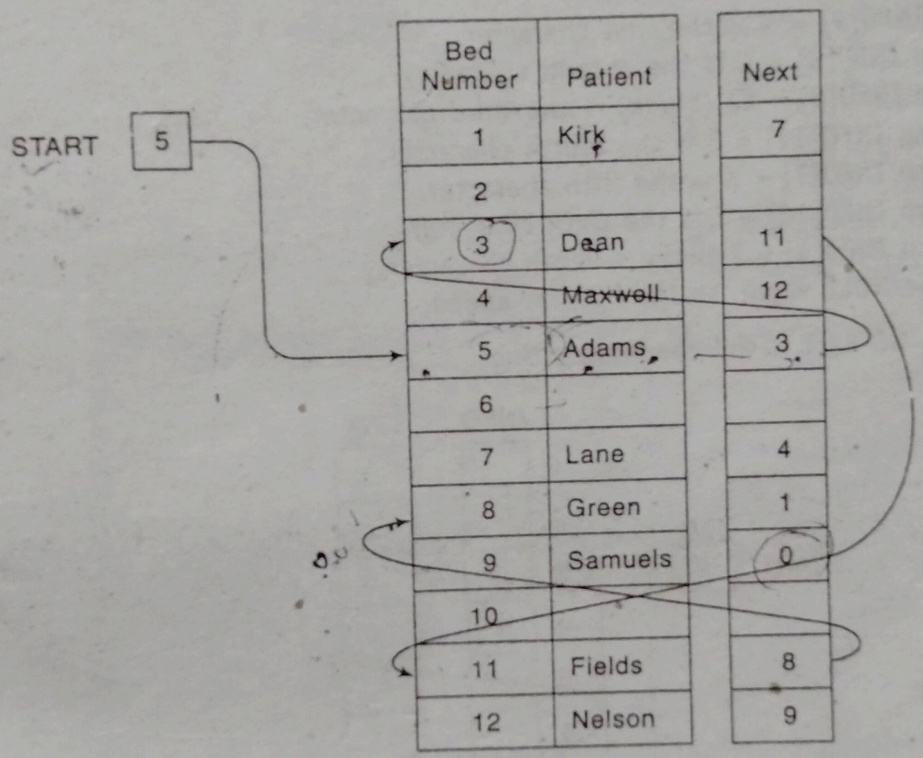


Fig. 5.3

5.3 REPRESENTATION OF LINKED LISTS IN MEMORY

Let LIST be a linked list. Then LIST will be maintained in memory, unless otherwise specified or implied, as follows. First of all, LIST requires two linear arrays—we will call them here INFO and LINK—such that INFO[K] and LINK[K] contain, respectively, the information part and the nextpointer field of a node of LIST. As noted above, LIST also requires a variable name—such as START—which contains the location of the beginning of the list, and a nextpointer sentinel—denoted by NULL—which indicates the end of the list. Since the subscripts of the arrays INFO and LINK will usually be positive, we will choose NULL = 0, unless otherwise stated.

The following examples of linked lists indicate that the nodes of a list need not occupy adjacent elements in the arrays INFO and LINK, and that more than one list may be maintained in the same linear arrays INFO and LINK. However, each list must have its own pointer variable giving the location of its first node.

Example 5.2

Figure 5.4 pictures a linked list in memory where each node of the list contains a single character. We can obtain the actual list of characters, or, in other words, the string, as follows:

START = 9, so INFO[9] = N is the first character.
LINK[9] = 3, so INFO[3] = O is the second character.
LINK[3] = 6, so INFO[6] = □ (blank) is the third character.
LINK[6] = 11, so INFO[11] = E is the fourth character.
LINK[11] = 7, so INFO[7] = X is the fifth character.
LINK[7] = 10, so INFO[10] = I is the sixth character.
LINK[10] = 4, so INFO[4] = T is the seventh character.
LINK[4] = 0, the NULL value, so the list has ended.

In other words, NO EXIT is the character string.

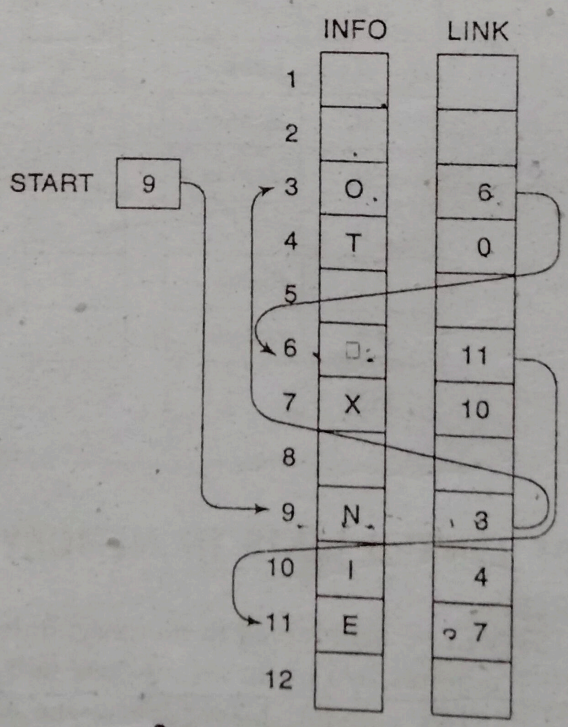


Fig. 5.4

Example 5.3

Figure 5.5 pictures show two lists of test scores, here ALG and GEOM, may be maintained in memory where the nodes of both lists are stored in the same linear arrays TEST and LINK. Observe that the names of the lists are also used as the list pointer variables. Here ALG contains 11, the location of its first node, and GEOM contains 5, the location of its first node. Following the pointers, we see that ALG consists of the test scores

88, 74, 93, 82

and GEOM consists of the test scores

84, 62, 74, 100, 74, 78

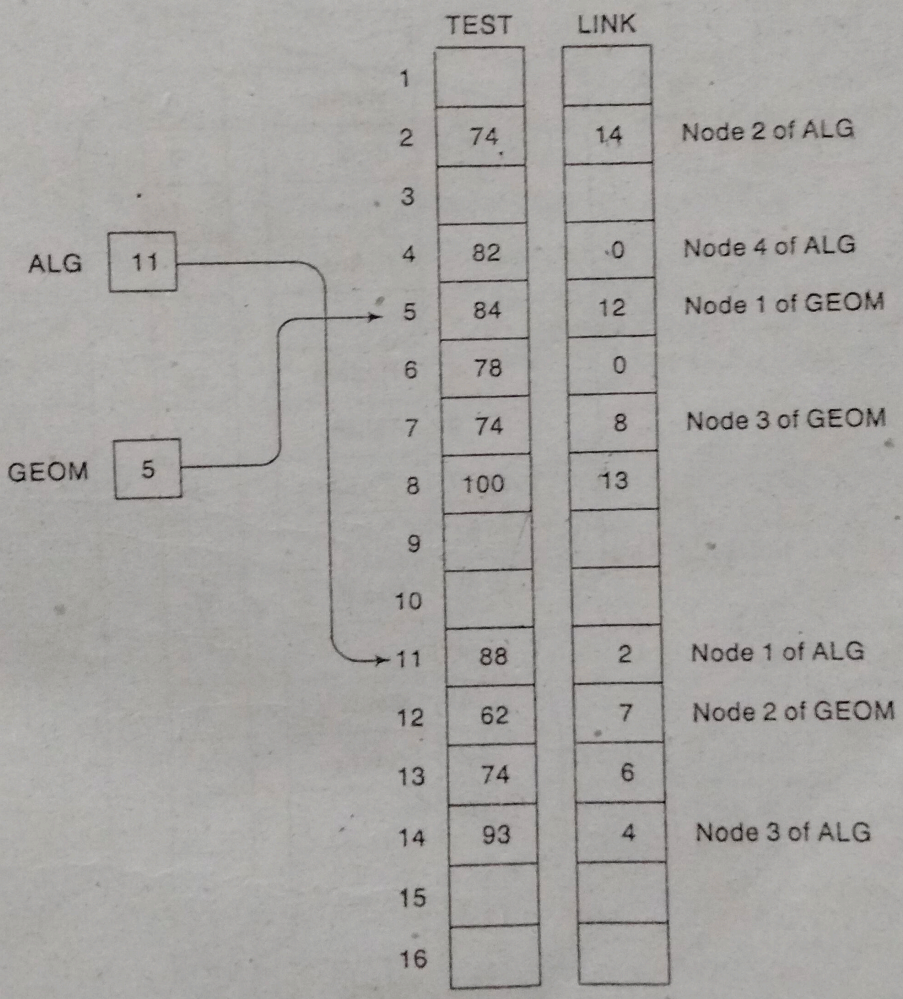


Fig. 5.5

(The nodes of ALG and some of the nodes of GEOM are explicitly labeled in the diagram.)

5.4 TRAVERSING A LINKED LIST

Let LIST be a linked list in memory stored in linear arrays INFO and LINK with START pointing to the first element and NULL indicating the end of LIST. Suppose we want to traverse LIST in order to process each node exactly once. This section presents an algorithm that does so and then uses the algorithm in some applications.

Our traversing algorithm uses a pointer variable PTR which points to the node that is currently being processed. Accordingly, LINK[PTR] points to the next node to be processed. Thus the assignment

PTR := LINK[PTR]

moves the pointer to the next node in the list, as pictured in Fig. 5.8.

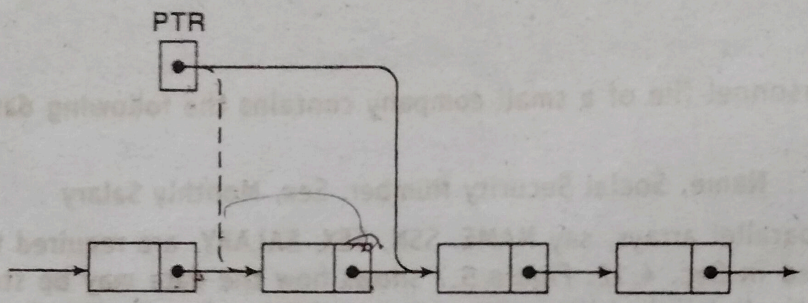


Fig. 5.8 *PTR : = LINK[PTR]*

The details of the algorithm are as follows. Initialize PTR or START. Then process INFO[PTR], the information at the first node. Update PTR by the assignment PTR := LINK[PTR], so that PTR points to the second node. Then process INFO[PTR], the information at the second node. Again update PTR by the assignment PTR := LINK[PTR], and then process INFO[PTR], the information at the third node. And so on. Continue until PTR = NULL, which signals the end of the list.

A formal presentation of the algorithm follows.

Algorithm 5.1: (Traversing a Linked List) Let LIST be a linked list in memory. This algorithm traverses LIST, applying an operation PROCESS to each element of LIST. The variable PTR points to the node currently being processed.

1. Set PTR := START. [Initializes pointer PTR.]
2. Repeat Steps 3 and 4 while PTR ≠ NULL.
3. Apply PROCESS to INFO[PTR].
4. Set PTR := LINK[PTR]. [PTR now points to the next node.]
   [End of Step 2 loop.]
5. Exit.

Observe the similarity between Algorithm 5.1 and Algorithm 4.1, which traverses a linear array. The similarity comes from the fact that both are linear structures which contain a natural linear ordering of the elements.

Caution: As with linear arrays, the operation PROCESS in Algorithm 5.1 may use certain variables which must be initialized before PROCESS is applied to any of the elements in LIST. Consequently, the algorithm may be preceded by such an initialization step.

5.5 SEARCHING A LINKED LIST

Let LIST be a linked list in memory, stored as in Secs. 5.3 and 5.4. Suppose a specific ITEM of information is given. This section discusses two searching algorithms for finding the location LOC of the node where ITEM first appears in LIST. The first algorithm does not assume that the data in LIST are sorted, whereas the second algorithm does assume that LIST is sorted.

If ITEM is actually a key value and we are searching through a file for the record containing ITEM, then ITEM can appear only once in LIST.

LIST Is Unsorted

Suppose the data in LIST are not necessarily sorted. Then one searches for ITEM in LIST by traversing through the list using a pointer variable PTR and comparing ITEM with the contents INFO[PTR] of each node, one by one, of LIST. Before we update the pointer PTR by

$$PTR := LINK[PTR]$$

we require two tests. First we have to check to see whether we have reached the end of the list; i.e., first we check to see whether

$$PTR = NULL$$

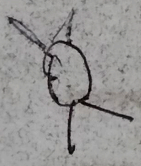
If not, then we check to see whether

$$INFO[PTR] = ITEM$$

The two tests cannot be performed at the same time, since INFO[PTR] is not defined when PTR = NULL. Accordingly, we use the first test to control the execution of a loop, and we let the second test take place inside the loop. The algorithm follows.

Algorithm 5.2 SEARCH(INFO, LINK, START, ITEM, LOC)
LIST is a linked list in memory. This algorithm finds the location LOC of the node where ITEM first appears in LIST, or sets LOC = NULL.

1. Set PTR := START.
2. Repeat Step 3 while PTR ≠ NULL:
3. If ITEM = INFO[PTR], then:
   Set LOC := PTR, and Exit.
   Else:
   Set PTR := LINK[PTR]. [PTR now points to the next node.]
   [End of If structure.]
   [End of Step 2 loop.]
4. [Search is unsuccessful.] Set LOC := NULL.
5. Exit.



The complexity of this algorithm is the same as that of the linear search algorithm for linear arrays discussed in Sec. 4.7. That is, the worst-case running time is proportional to the number n of elements in LIST, and the average-case running time is approximately proportional to $n/2$ (with the condition that ITEM appears once in LIST but with equal probability in any node of LIST).

Example 5.8

Consider the personnel file in Fig. 5.7. The following module reads the social security number NNN of an employee and then gives the employee a 5 percent increase in salary.

```
1. Read: NNN.
2. Call SEARCH(SSN, LINK, START, NNN, LOC).
3. If LOC ≠ NULL, then:
       Set SALARY[LOC] : = SALARY[LOC] + 0.05*SALARY[LOC],
   Else:
       Write: NNN is not in file.
   [End of If structure.]
4. Return.
```

(The module takes care of the case in which there is an error in inputting the social security number.)

LIST is Sorted

Suppose the data in LIST are sorted. Again we search for ITEM in LIST by traversing the list using a pointer variable PTR and comparing ITEM with the contents INFO[PTR] of each node, one by one, of LIST. Now, however, we can stop once ITEM exceeds INFO[PTR]. The algorithm follows.

Algorithm 5.3: SRCHSL(INFO, LINK, START, ITEM, LOC)
LIST is a sorted list in memory. This algorithm finds the location LOC of the node where ITEM first appears in LIST, or sets LOC = NULL.

```
1. Set PTR := START.
2. Repeat Step 3 while PTR ≠ NULL:
3.       If ITEM < INFO[PTR], then:
             Set PTR := LINK[PTR]. [PTR now points to next node.]
         Else if ITEM = INFO[PTR], then:
             Set LOC := PTR, and Exit. [Search is successful.]
         Else:
             Set LOC := NULL, and Exit. [ITEM now exceeds INFO[PTR].]
         [End of If structure.]
     [End of Step 2 loop.]
4. Set LOC := NULL.
5. Exit.
```

The complexity of this algorithm is still the same as that of other linear search algorithms; that is, the worst-case running time is proportional to the number n of elements in LIST, and the average-case running time is approximately proportional to $n/2$.

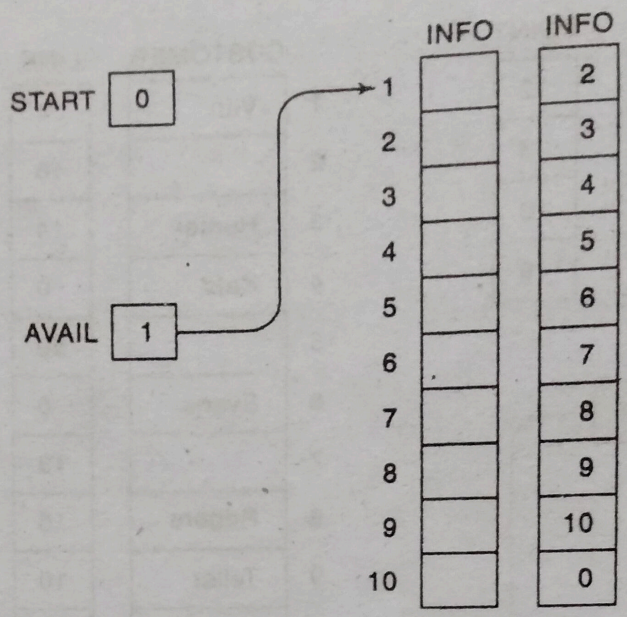


Fig. 5.13

Garbage Collection

Suppose some memory space becomes reusable because a node is deleted from a list or an entire list is deleted from a program. Clearly, we want the space to be available for future use. One way to bring this about is to immediately reinsert the space into the free-storage list. This is what we will do when we implement linked lists by means of linear arrays. However, this method may be too time-consuming for the operating system of a computer, which may choose an alternative method, as follows.

The operating system of a computer may periodically collect all the deleted space onto the free-storage list. Any technique which does this collection is called *garbage collection*. Garbage collection usually takes place in two steps. First the computer runs through all lists, tagging those cells which are currently in use, and then the computer runs through the memory, collecting all untagged space onto the free-storage list. The garbage collection may take place when there is only some minimum amount of space or no space at all left in the free-storage list, or when the CPU is idle and has time to do the collection. Generally speaking, the garbage collection is invisible to the programmer. Any further discussion about this topic of garbage collection lies beyond the scope of this text.

Overflow and Underflow

Sometimes new data are to be inserted into a data structure but there is no available space, i.e., the free-storage list is empty. This situation is usually called *overflow*. The programmer may handle overflow by printing the message OVERFLOW. In such a case, the programmer may then modify the program by adding space to the underlying arrays. Observe that overflow will occur with our linked lists when AVAIL = NULL and there is an insertion.

Analogously, the term *underflow* refers to the situation where one wants to delete data from a data structure that is empty. The programmer may handle underflow by printing the message UNDERFLOW. Observe that underflow will occur with our linked lists when START = NULL and there is a deletion.

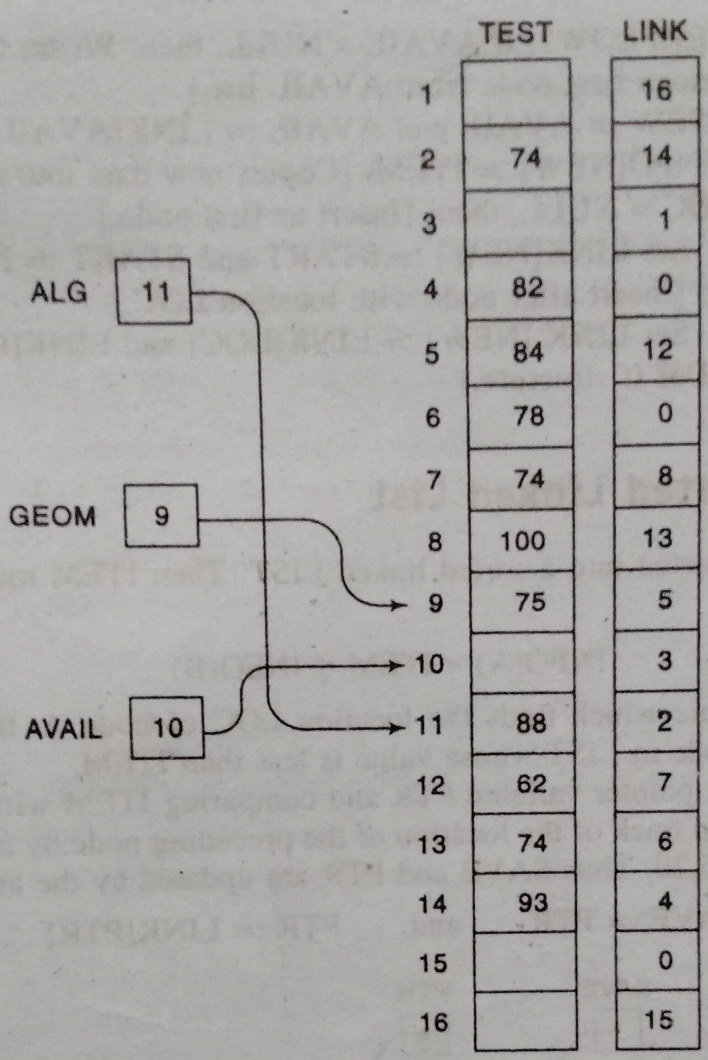


Fig. 5.19

Inserting after a Given Node

Suppose we are given the value of LOC where either LOC is the location of a node A in a linked LIST or LOC = NULL. The following is an algorithm which inserts ITEM into LIST so that ITEM follows node A or, when LOC = NULL, so that ITEM is the first node.

Let N denote the new node (whose location is NEW). If LOC = NULL, then N is inserted as the first node in LIST as in Algorithm 5.4. Otherwise, as pictured in Fig. 5.15, we let node N point to node B (which originally followed node A) by the assignment

LINK[NEW] := LINK[LOC]

and we let node A point to the new node N by the assignment

LINK[LOC] := NEW

A formal statement of the algorithm follows.

Algorithm 5.5: INSLOC(INFO, LINK, START, AVAIL, LOC, ITEM)
This algorithm inserts ITEM so that ITEM follows the node with location LOC or inserts ITEM as the first node when LOC = NULL.

5.7 INSERTION INTO A LINKED LIST

Let LIST be a linked list with successive nodes A and B, as pictured in Fig. 5.14(a). Suppose a node N is to be inserted into the list between nodes A and B. The schematic diagram of such an insertion appears in Fig. 5.14(b). That is, node A now points to the new node N, and node N points to node B, to which A previously pointed.

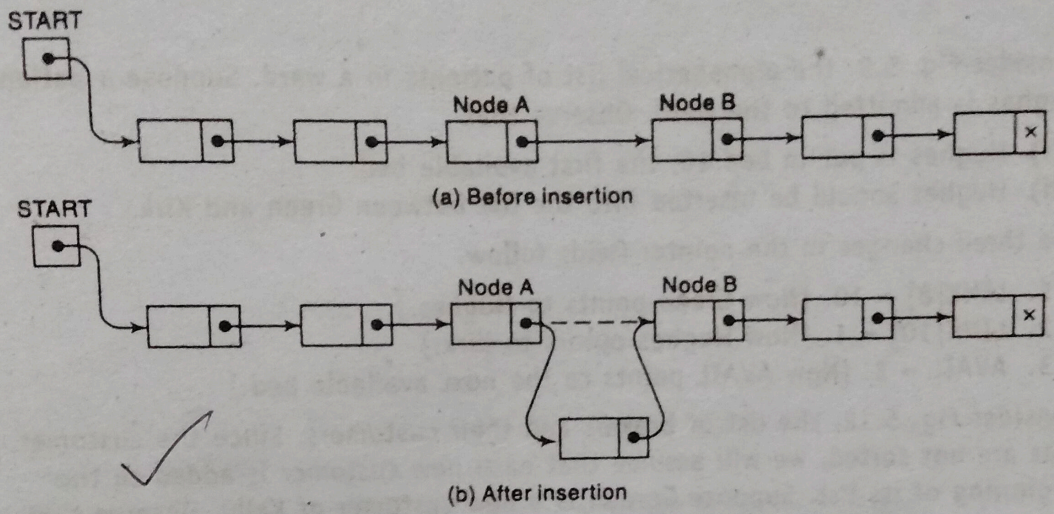


Fig. 5.14

Suppose our linked list is maintained in memory in the form

LIST(INFO, LINK, START, AVAIL)

Figure 5.14 does not take into account that the memory space for the new node N will come from thy AVAIL list. Specifically, for easier processing, the first node in the AVAIL list will be used for the new node N. Thus a more exact schematic diagram of such an insertion is that in Fig. 5.15. Observe that three pointer fields are changed as follows:

(1) The nextpointer field of node A now points to the new node N, to which AVAIL previously pointed.

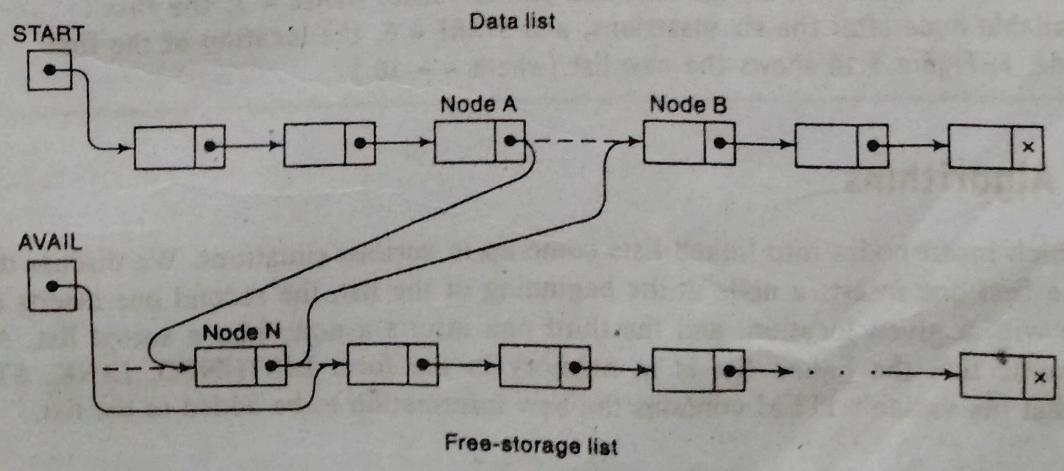


Fig. 5.15

Deletion Algorithms

Algorithms which delete nodes from linked lists come up in various situations. We discuss two of them here. The first one deletes the node following a given node, and the second one deletes the node with a given ITEM of information. All our algorithms assume that the linked list is in memory in the form LIST(INFO, LINK, START, AVAIL).

All of our deletion algorithms will return the memory space of the deleted node N to the beginning of the AVAIL list. Accordingly, all of our algorithms will include the following pair of assignments, where LOC is the location of the deleted node N:

LINK[LOC] := AVAIL and then AVAIL := LOC

These two operations are pictured in Fig. 5.25.

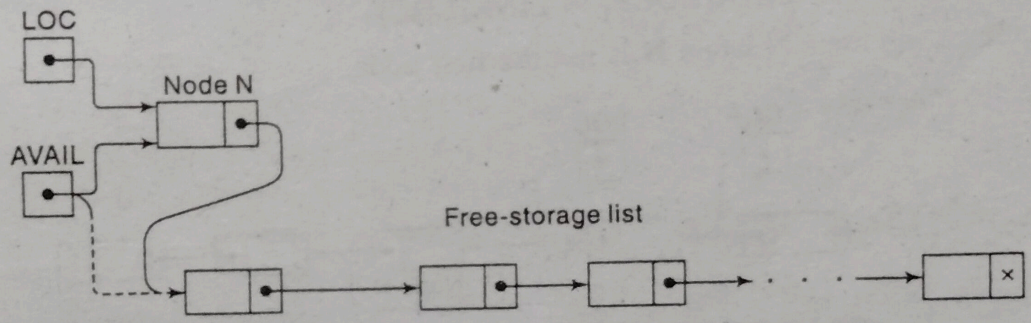


Fig. 5.25 *LINK[LOC] := AVAIL and AVAIL := LOC*

Some of our algorithms may want to delete either the first node or the last node from the list. An algorithm that does so must check to see if there is a node in the list. If not, i.e., if START = NULL, then the algorithm will print the message UNDERFLOW.

Deleting the Node Following a Given Node

Let LIST be a linked list in memory. Suppose we are given the location LOC of a node N in LIST. Furthermore, suppose we are given the location LOCP of the node preceding N or, when N is the first node, we are given LOCP = NULL. The following algorithm deletes N from the list.

Algorithm 5.8: DEL(INFO, LINK, START, AVAIL, LOC, LOCP)
This algorithm deletes the node N with location LOC. LOCP is the location of the node which precedes N or, when N is the first node, LOCP = NULL.

1. If LOCP = NULL, then:
   Set START := LINK[START]. [Deletes first node.]
   Else:
   Set LINK[LOCP] := LINK[LOC]. [Deletes node N.]
   [End of If structure.]
2. [Return deleted node to the AVAIL list.]
   Set LINK[LOC] := AVAIL and AVAIL := LOC.
3. Exit.

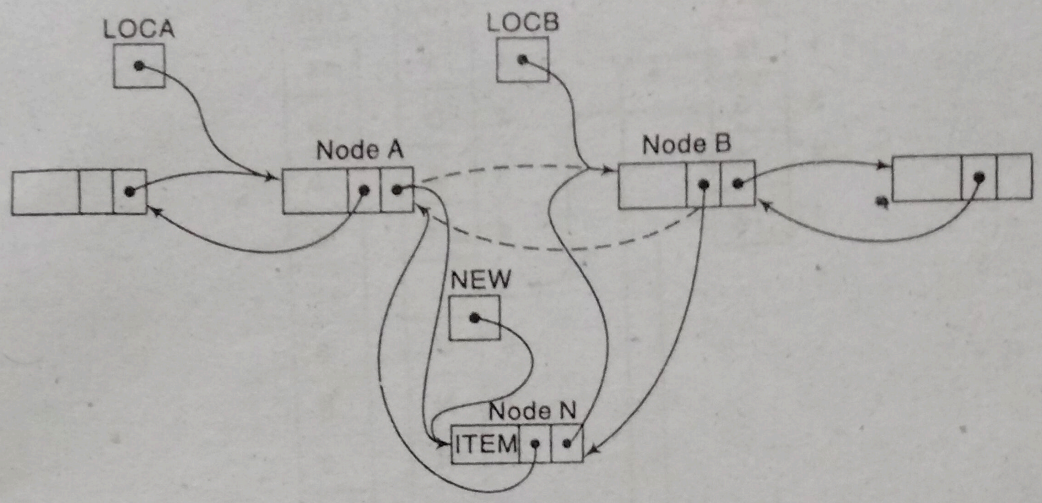


Fig. 5.38 *Inserting Node N*

3. [Insert node into list.]
   Set FORW[LOCA] := NEW, FORW[NEW] := LOCB,
   BACK[LOCB] := NEW, BACK[NEW] := LOCA.
4. Exit.

Algorithm 5.16 assumes that LIST contains a header node. Hence LOCA or LOCB may point to the header node, in which case N will be inserted as the first node or the last node. If LIST does not contain a header node, then we must consider the case that LOCA = NULL and N is inserted as the first node in the list, and the case that LOCB = NULL and N is inserted as the last node in the list.

Remark: Generally speaking, storing data as a two-way list, which requires extra space for the backward pointers and extra time to change the added pointers, rather than as a one-way list is not worth the expense unless one must frequently find the location of the node which precedes a given node N, as in the deletion above.

SOLVED PROBLEMS

Linked Lists

5.1 Find the character strings stored in the four linked lists in Fig. 5.39.

Here the four list pointers appear in an array CITY. Beginning with CITY[1], traverse the list, by following the pointers, to obtain the string PARIS. Beginning with CITY[2], traverse the list to obtain the string LONDON. Since NULL appears in CITY[3], the third list is empty, so it denotes Λ, the empty string. Beginning with CITY[4], traverse the list to obtain the string ROME. In other words, PARIS, LONDON, Λ and ROME are the four strings.

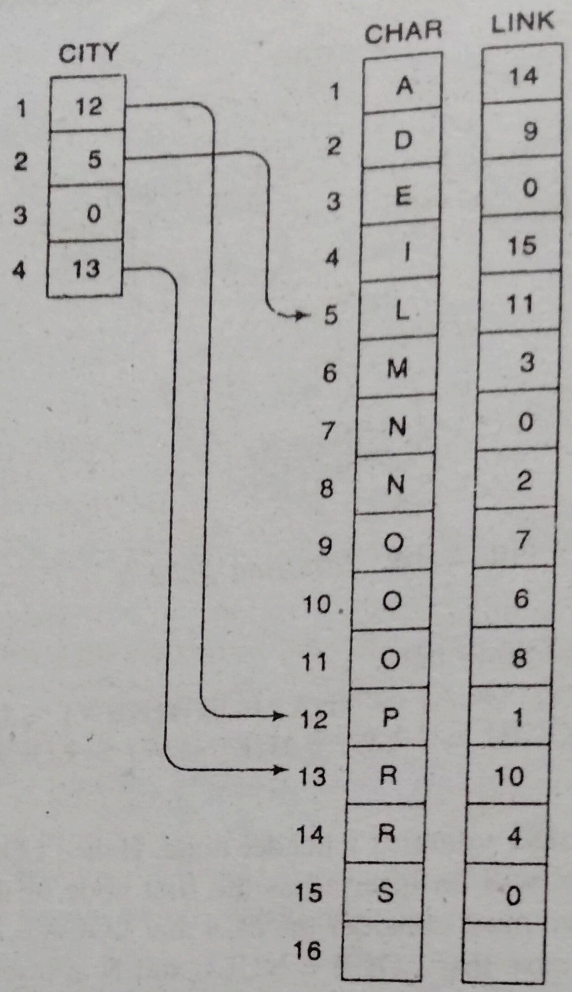


Fig. 5.39

5.2 The following list of names is assigned (in order) to a linear array INFO:

Mary, June, Barbara, Paula, Diana, Audrey, Karen, Nancy, Ruth, Eileen, Sandra, Helen

That is, INFO[1] = Mary, INFO[2] = June,..., INFO[12] = Helen. Assign values to an array LINK and a variable START so that INFO, LINK and START form an alphabetical listing of the names.

The alphabetical listing of the names follows:

Audrey, Barbara, Diana, Eileen, Helen, June, Karen, Mary, Nancy, Paula, Ruth, Sandra

The values of START and LINK are obtained as follows:

(a) INFO[6] = Audrey, so assign START = 6.
(b) INFO[3] = Barbara, so assign LINK[6] = 3.
(c) INFO[5] = Diana, so assign LINK[3] = 5.
(d) INFO[10] = Eileen, so assign LINK[5] = 10.

And so on. Since INFO[11] = Sandra is the last name, assign LINK[11] = NULL. Figure 5.40 shows the data structure where, assuming INFO has space for only 12 elements, we set AVAIL = NULL.