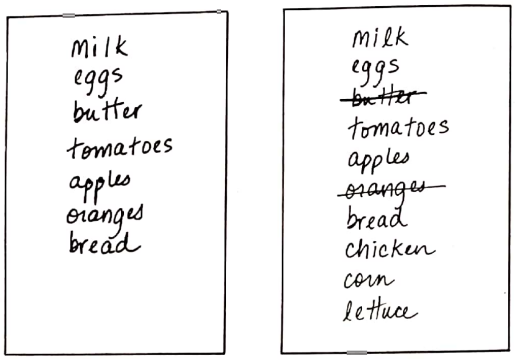
Chapter 5

Linked Lists

5.1 INTRODUCTION

The everyday usage of the term "list" refers to a linear collection of data items. Figure 5-1(*a*) shows a shopping list; it contains a first element, a second element, . . . , and a last element. Frequently, we want to add items to or delete items from a list. Figure 5-1(*b*) shows the shopping list after three items have been added at the end of the list and two others have been deleted (by being crossed out).



(*a*) (*b*)

Fig. 5-1

Data processing frequently involves storing and processing data organized into lists. One way to store such data is by means of arrays, discussed in Chap. 4. Recall that the linear relationship between the data elements of an array is reflected by the physical relationship of the data in memory, not by any information contained in the data elements themselves. This makes it easy to compute the address of an element in an array. On the other hand, arrays have certain disadvantages—e.g., it is relatively expensive to insert and delete elements in an array. Also, since an array usually occupies a block of memory space, one cannot simply double or triple the size of an array when additional space is required. (For this reason, arrays are called *dense lists* and are said to be *static* data structures.)

Another way of storing a list in memory is to have each element in the list contain a field, called a *link* or *pointer*, which contains the address of the next element in the list. Thus successive elements in the list need not occupy adjacent space in memory. This will make it easier to insert and delete elements in the list. Accordingly, if one were mainly interested in searching through data for inserting and deleting, as in word processing, one would not store the data in an array but rather in a list using pointers. This latter type of data structure is called a *linked list* and is the main subject matter of this chapter. We also discuss circular lists and two-way lists—which are natural generalizations of linked lists—and their advantages and disadvantages.

5.2 LINKED LISTS

A *linked list*, or *one-way list*, is a linear collection of data elements, called *nodes*, where the linear order is given by means of *pointers*. That is, each node is divided into two parts: the first part contains the information of the element, and the second part, called the *link field* or *nextpointer field*, contains the address of the next node in the list.

Figure 5-2 is a schematic diagram of a linked list with 6 nodes. Each node is pictured with two parts. The left part represents the information part of the node, which may contain an entire record of data items (e.g., NAME, ADDRESS, . . .). The right part represents the nextpointer field of the node; and there is an arrow drawn from it to the next node in the list. This follows the usual practice of drawing an arrow from a field to a node when the address of the node appears in the given field. The pointer of the last node contains a special value, called the *null* pointer, which is any invalid address.

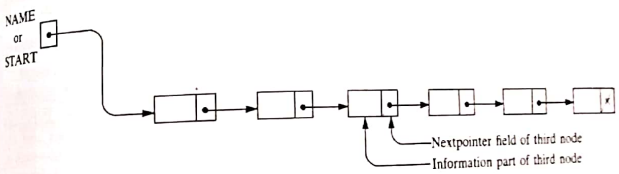


Fig. 5-2 Linked list with 6 nodes.

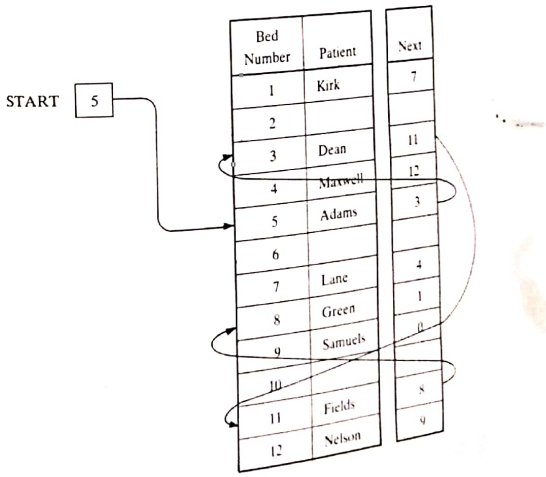


Fig. 5-3

(In actual practice, 0 or a negative number is used for the null pointer.) The null pointer, denoted by ∧ in the diagram, signals the end of the list. The linked list also contains a *list pointer variable*—called START or NAME—which contains the address of the first node in the list; hence there is an arrow drawn from START to the first node. Clearly, we need only this address in START to trace through the list. A special case is the list that has no nodes. Such a list is called the *null list* or *empty list* and is denoted by the null pointer in the variable START.

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.)

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.

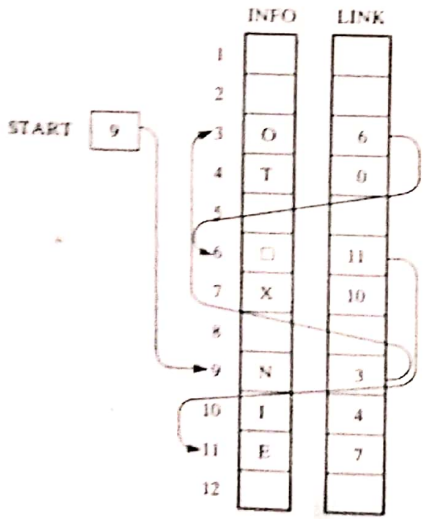


Fig. 5-4

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.

EXAMPLE 5.3

Figure 5-5 pictures how 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, 62

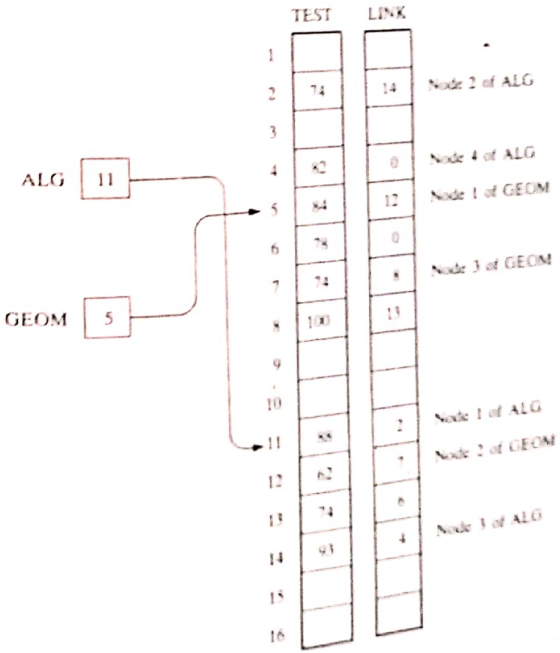


Fig. 5-5

and GEOM consists of the test scores

84, 62, 74, 100, 74, 78

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

EXAMPLE 5.4

Suppose a brokerage firm has four brokers and each broker has his own list of customers. Such data may be organized as in Fig. 5-6. That is, all four lists of customers appear in the same array CUSTOMER, and an array LINK contains the nextpointer fields of the nodes of the lists. There is also an array BROKER which contains the list of brokers, and a pointer array POINT such that POINT[K] points to the beginning of the list of customers of BROKER[K].

Accordingly, Bond's list of customers, as indicated by the arrows, consists of

Grant, Scott, Vito, Katz

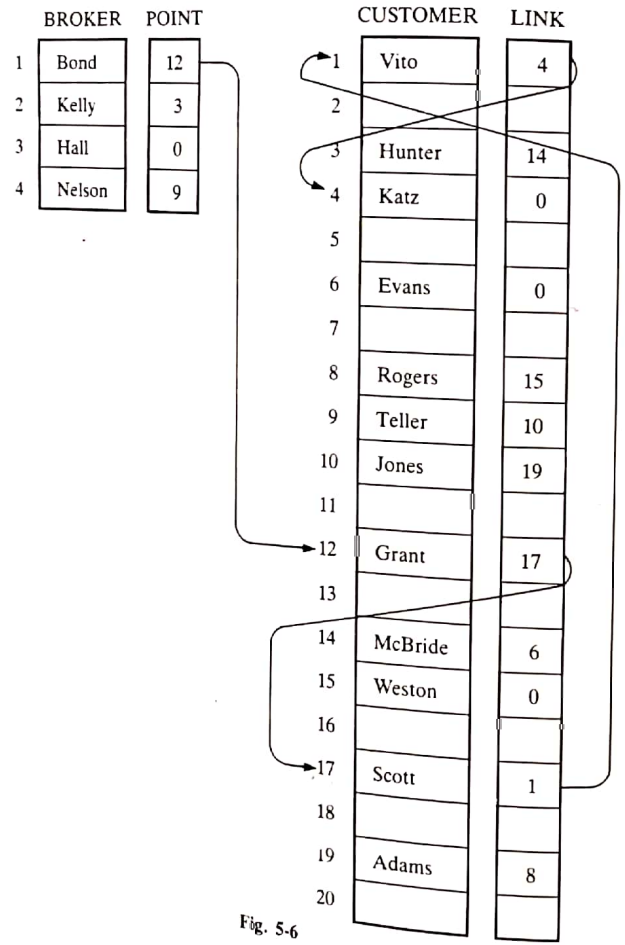


Fig. 5-6

Similarly, Kelly's list consists of

Hunter, McBride, Evans

and Nelson's list consists of

Teller, Jones, Adams, Rogers, Weston

Hall's list is the null list, since the null pointer 0 appears in POINT[3].

Generally speaking, the information part of a node may be a record with more than one data item. In such a case, the data must be stored in some type of record structure or in a collection of parallel arrays, such as that illustrated in the following example.

EXAMPLE 5.5

Suppose the personnel file of a small company contains the following data on its nine employees:

Name, Social Security Number, Sex, Monthly Salary

Normally, four parallel arrays, say NAME, SSN, SEX, SALARY, are required to store the data as discussed in Sec. 4.12. Figure 5-7 shows how the data may be stored as a sorted (alphabetically) linked list using only an additional array LINK for the nextpointer field of the list and the variable START to point to the first record in the list. Observe that 0 is used as the null pointer.

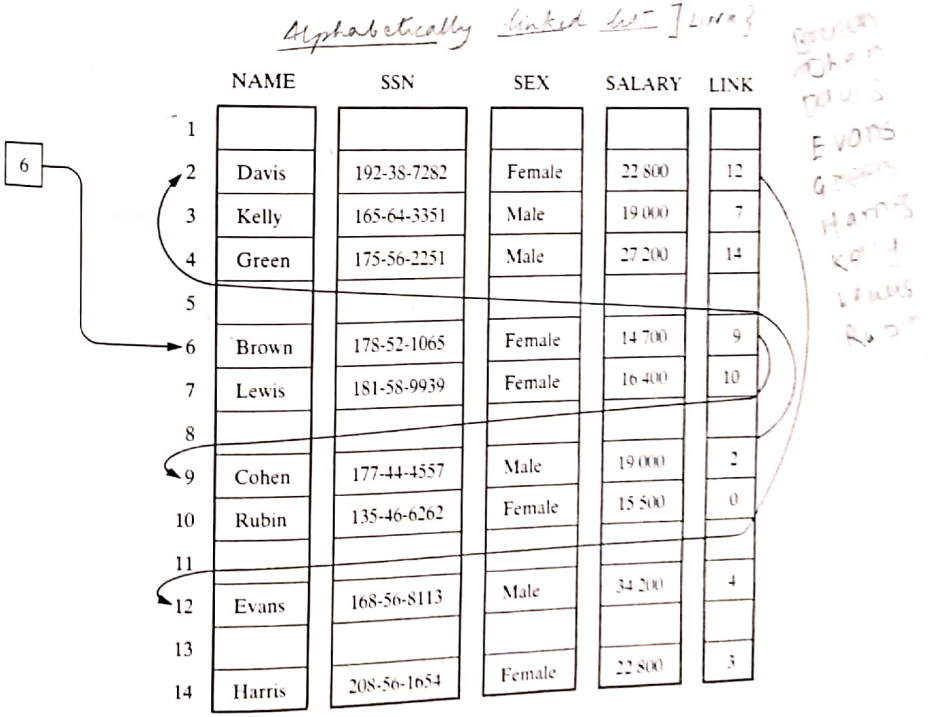


Fig. 5-7

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.

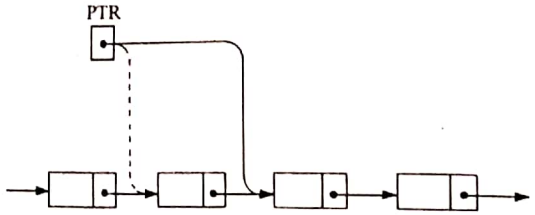


Fig. 5-8 PTR := LINK[PTR].

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.

EXAMPLE 5.6

The following procedure prints the information at each node of a linked list. Since the procedure must traverse the list, it will be very similar to Algorithm 5.1.

Procedure: PRINT(INFO, LINK, START)
This procedure prints the information at each node of the list.

1. Set PTR := START.
2. Repeat Steps 3 and 4 while PTR ≠ NULL:
3. Write: INFO[PTR].
4. Set PTR := LINK[PTR]. [Updates pointer.]
   [End of Step 2 loop.]
5. Return.

In other words, the procedure may be obtained by simply substituting the statement

Write: INFO[PTR]

for the processing step in Algorithm 5.1.

EXAMPLE 5.7

The following procedure finds the number NUM of elements in a linked list.

Procedure: COUNT(INFO, LINK, START, NUM)

1. Set NUM := 0. [Initializes counter.]
2. Set PTR := START. [Initializes pointer.]
3. Repeat Steps 4 and 5 while PTR ≠ NULL.
4. Set NUM := NUM + 1. [Increases NUM by 1.]
5. Set PTR := LINK[PTR]. [Updates pointer.]
   [End of Step 3 loop.]
6. Return.

Observe that the procedure traverses the linked list in order to count the number of elements, hence the procedure is very similar to the above traversing algorithm, Algorithm 5.1. Here, however, we require an initialization step for the variable NUM before traversing the list. In other words, the procedure could have been written as follows:

Procedure: COUNT(INFO, LINK, START, NUM)

1. Set NUM := 0. [Initializes counter.]
2. Call Algorithm 5.1, replacing the processing step by:
   Set NUM := NUM + 1.
3. Return.

Most list processing procedures have this form. (See Prob. 5.3.)

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

$$\text{PTR} = \text{NULL}$$

If not, then we check to see whether

$$\text{INFO[PTR]} = \text{ITEM}$$

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 on page 123.

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 $n/2$.

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.

Recall that with a sorted linear array we can apply a binary search whose running time is proportional to $\log_2 n$. On the other hand, a *binary search algorithm cannot be applied to a sorted linked list, since there is no way of indexing the middle element in the list.* This property is one of the main drawbacks in using a linked list as a data structure.

EXAMPLE 5.9

Consider, again, the personnel file in Fig. 5-7. The following module reads the name EMP of an employee and then gives the employee a 5 percent increase in salary. (Compare with Example 5.8.)

1. Read: EMPNAME.
2. Call SRCHSL(NAME, LINK, START, EMPNAME, LOC).
3. If LOC ≠ NULL, then:
   Set SALARY[LOC] := SALARY[LOC] + 0.05 * SALARY[LOC].
   Else:
   Write: EMPNAME is not in list.
   [End of If structure.]
4. Return.

Observe that now we can use the second search algorithm, Algorithm 5.3, since the list is sorted alphabetically.

5.6 MEMORY ALLOCATION; GARBAGE COLLECTION

The maintenance of linked lists in memory assumes the possibility of inserting new nodes into the lists and hence requires some mechanism which provides unused memory space for the new nodes. Analogously, some mechanism is required whereby the memory space of deleted nodes becomes available for future use. These matters are discussed in this section, while the general discussion of the inserting and deleting of nodes is postponed until later sections.

Together with the linked lists in memory, a special list is maintained which consists of unused memory cells. This list, which has its own pointer, is called the *list of available space* or the *free-storage list* or the *free pool*.

Suppose our linked lists are implemented by parallel arrays as described in the preceding sections, and suppose insertions and deletions are to be performed on our linked lists. Then the unused memory cells in the arrays will also be linked together to form a linked list using AVAIL as its list pointer variable. (Hence this free-storage list will also be called the AVAIL list.) Such a data structure will frequently be denoted by writing

LIST(INFO, LINK, START, AVAIL)

EXAMPLE 5.10

Suppose the list of patients in Example 5.1 is stored in the linear arrays BED and LINK (so that the patient in bed K is assigned to BED[K]). Then the available space in the linear array BED may be linked as in Fig. 5-9. Observe that BED[10] is the first available bed, BED[2] is the next available bed, and BED[6] is the last available bed. Hence BED[6] has the null pointer in its nextpointer field; that is, LINK[6] = 0.

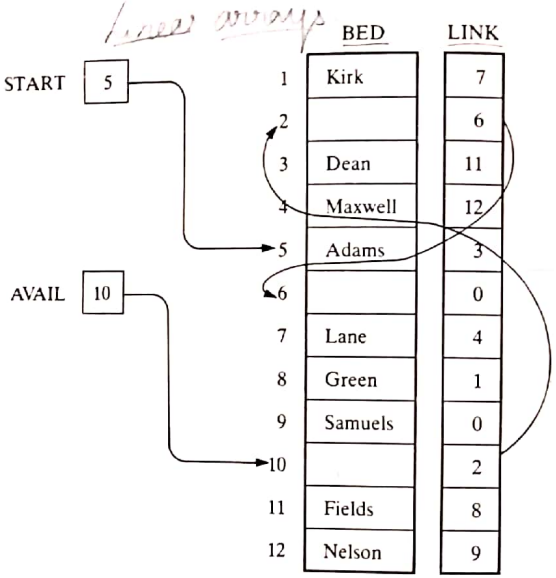


Fig. 5-9

EXAMPLE 5.11

(*a*) The available space in the linear array TEST in Fig. 5-5 may be linked as in Fig. 5-10. Observe that each of the lists ALG and GEOM may use the AVAIL list. Note that AVAIL = 9, so TEST[9] is the first free node in the AVAIL list. Since LINK[AVAIL] = LINK[9] = 10, TEST[10] is the second free node in the AVAIL list. And so on.

(*b*) Consider the personnel file in Fig. 5-7. The available space in the linear array NAME may be linked as in Fig. 5-11. Observe that the free-storage list in NAME consists of NAME[8], NAME[11], NAME[13], NAME[5] and NAME[1]. Moreover, observe that the values in LINK simultaneously list the free-storage space for the linear arrays SSN, SEX and SALARY.

(*c*) The available space in the array CUSTOMER in Fig. 5-6 may be linked as in Fig. 5-12. We emphasize that each of the four lists may use the AVAIL list for a new customer.

EXAMPLE 5.12

Suppose LIST(INFO, LINK, START, AVAIL) has memory space for $n = 10$ nodes. Furthermore, suppose LIST is initially empty. Figure 5-13 shows the values of LINK so that the AVAIL list consists of the sequence

INFO[1], INFO[2], . . . , INFO[10]

that is, so that the AVAIL list consists of the elements of INFO in the usual order. Observe that START = NULL since the list is empty.

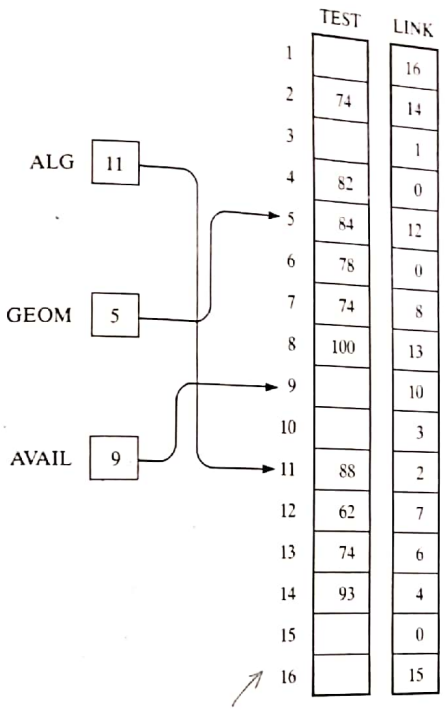


Fig. 5-10

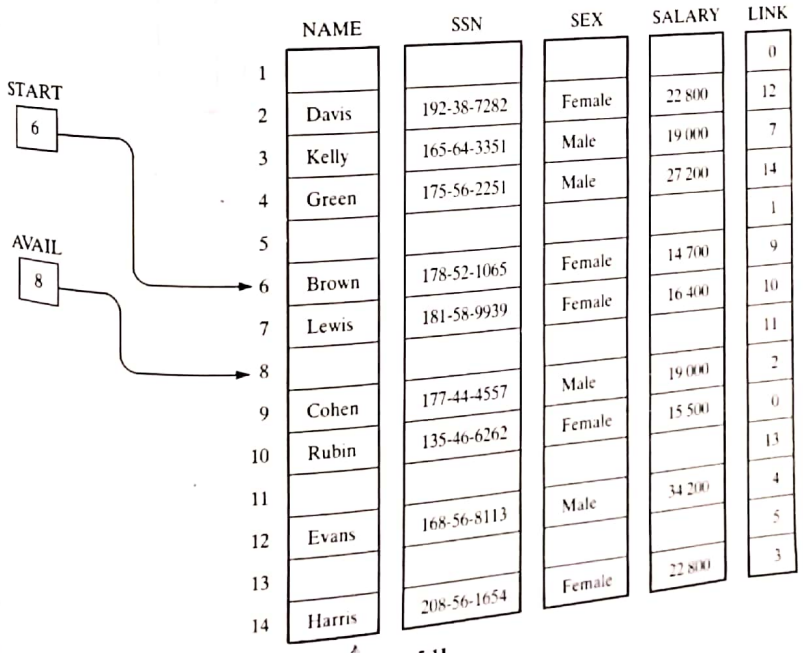


Fig. 5-11

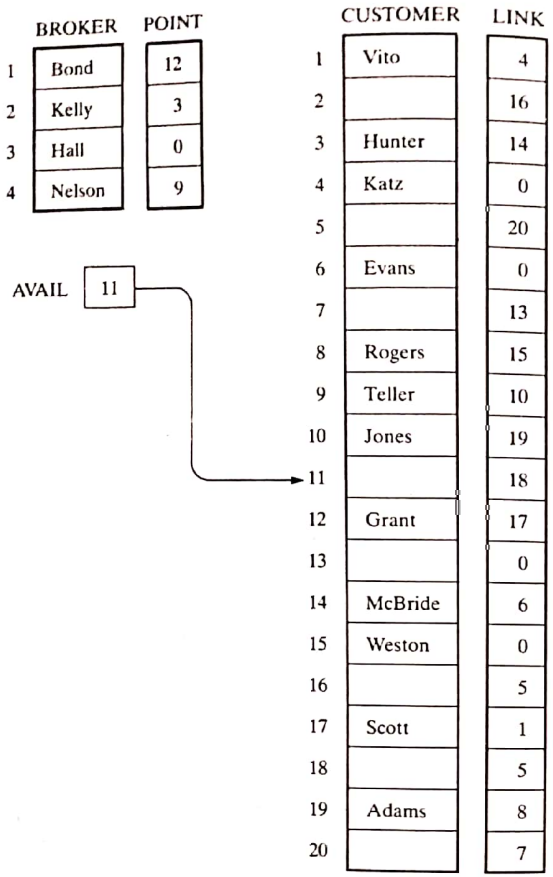


Fig. 5-12

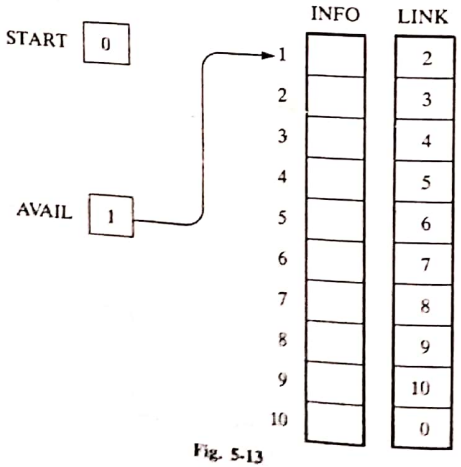


Fig. 5-13

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.

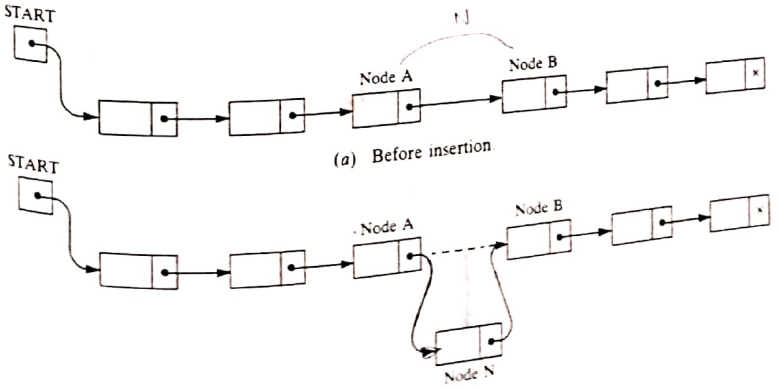
Overflow and Underflow

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.

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



(*a*) Before insertion

(*b*) After insertion

Fig. 5-14

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.

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 the 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.

(2) AVAIL now points to the second node in the free pool, to which node N previously pointed.

(3) The nextpointer field of node N now points to node B, to which node A previously pointed.

There are also two special cases. If the new node N is the first node in the list, then START will point to N; and if the new node N is the last node in the list, then N will contain the null pointer.

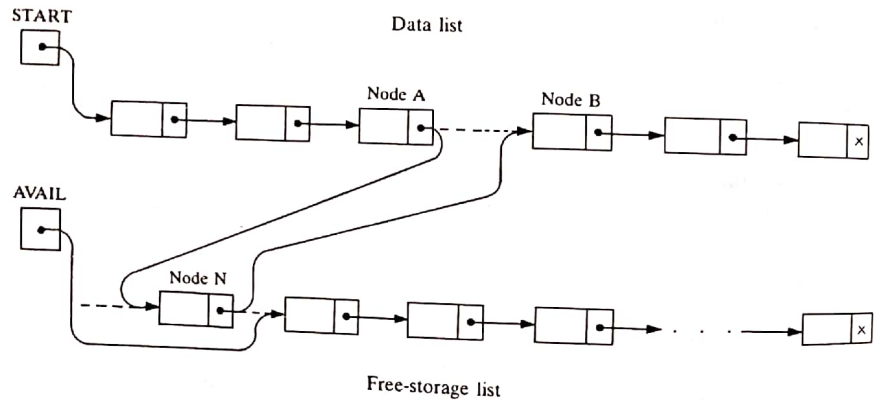


Fig. 5-15

EXAMPLE 5.13

(*a*) Consider Fig. 5-9, the alphabetical list of patients in a ward. Suppose a patient Hughes is admitted to the ward. Observe that

(i) Hughes is put in bed 10, the first available bed.

(ii) Hughes should be inserted into the list between Green and Kirk.

The three changes in the pointer fields follow.

1. LINK[8] = 10. [Now Green points to Hughes.]
2. LINK[10] = 1. [Now Hughes points to Kirk.]
3. AVAIL = 2. [Now AVAIL points to the next available bed.]

(*b*) Consider Fig. 5-12, the list of brokers and their customers. Since the customer lists are not sorted, we will assume that each new customer is added to the beginning of its list. Suppose Gordan is a new customer of Kelly. Observe that

(i) Gordan is assigned to CUSTOMER[11], the first available node.

(ii) Gordan is inserted before Hunter, the previous first customer of Kelly.

The three changes in the pointer fields follow:

1. POINT[2] = 11. [Now the list begins with Gordan.]
2. LINK[11] = 3. [Now Gordan points to Hunter.]
3. AVAIL = 18. [Now AVAIL points to the next available node.]

(*c*) Suppose the data elements A, B, C, D, E and F are inserted one after the other into the empty list in Fig. 5-13. Again we assume that each new node is inserted at the beginning of the list. Accordingly, after the six insertions, F will point to E, which points to D, which points to C, which points to B, which points to A; and A will contain the null pointer. Also, AVAIL = 7, the first available node after the six insertions, and START = 6, the location of the first node, F. Figure 5-16 shows the new list (where $n = 10$.)

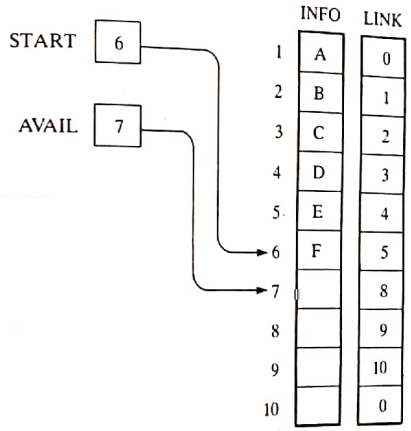


Fig. 5-16

Insertion Algorithms

Algorithms which insert nodes into linked lists come up in various situations. We discuss three of them here. The first one inserts a node at the beginning of the list, the second one inserts a node after the node with a given location, and the third one inserts a node into a sorted list. All our algorithms assume that the linked list is in memory in the form LIST(INFO, LINK, START, AVAIL) and that the variable ITEM contains the new information to be added to the list.

Since our insertion algorithms will use a node in the AVAIL list, all of the algorithms will include the following steps:

(*a*) Checking to see if space is available in the AVAIL list. If not, that is, if AVAIL = NULL, then the algorithm will print the message OVERFLOW.

(*b*) Removing the first node from the AVAIL list. Using the variable NEW to keep track of the location of the new node, this step can be implemented by the pair of assignments (in this order)

NEW := AVAIL, AVAIL := LINK[AVAIL]

(*c*) Copying new information into the new node. In other words,

INFO[NEW] := ITEM

The schematic diagram of the latter two steps is pictured in Fig. 5-17.

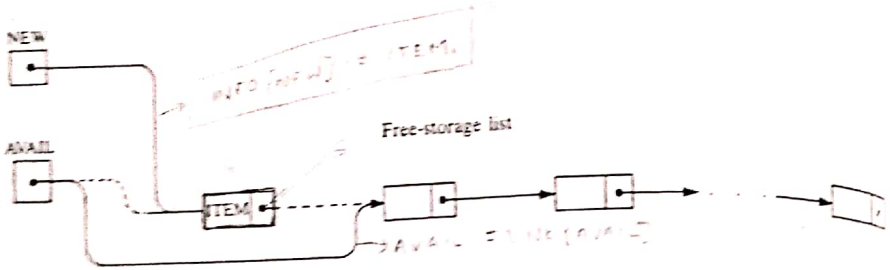


Fig. 5-17

Inserting at the Beginning of a List

Suppose our linked list is not necessarily sorted and there is no reason to insert a new node in any special place in the list. Then the easiest place to insert the node is at the begining of the list. An algorithm that does so follows.

Algorithm 5.4: INSFIRST(INFO, LINK, START, AVAIL, ITEM)
This algorithm inserts ITEM as the first node in the list.

1. [OVERFLOW?] 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. [Copies new data into new node.]
4. Set LINK[NEW] := START. [New node now points to original first node.]
5. Set START := NEW. [Changes START so it points to the new node.]
6. Exit.

Steps 1 to 3 have already been discussed, and the schematic diagram of Steps 2 and 3 appears in Fig. 5-17. The schematic diagram of Steps 4 and 5 appears in Fig. 5-18.

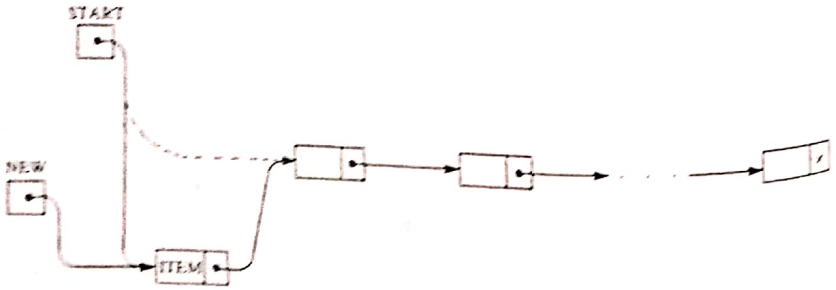


Fig. 5-18 Insertion at the beginning of a list.

EXAMPLE 5.14

Consider the lists of tests in Fig. 5-10. Suppose the test score 75 is to be added to the beginning of the geometry list. We simulate Algorithm 5.4. Observe that ITEM = 75, INFO = TEST and START = GEOM.

INSFIRST(TEST, LINK, GEOM, AVAIL, ITEM)

1. Since AVAIL = NULL, control is transferred to Step 2.
2. NEW = 9, then AVAIL = LINK[9] = 10.
3. TEST[9] = 75.
4. LINK[9] = 5.
5. GEOM = 9.
6. Exit.

Figure 5-19 shows the data structure after 75 is added to the geometry list. Observe that only three pointers are changed, AVAIL, GEOM and LINK[9].

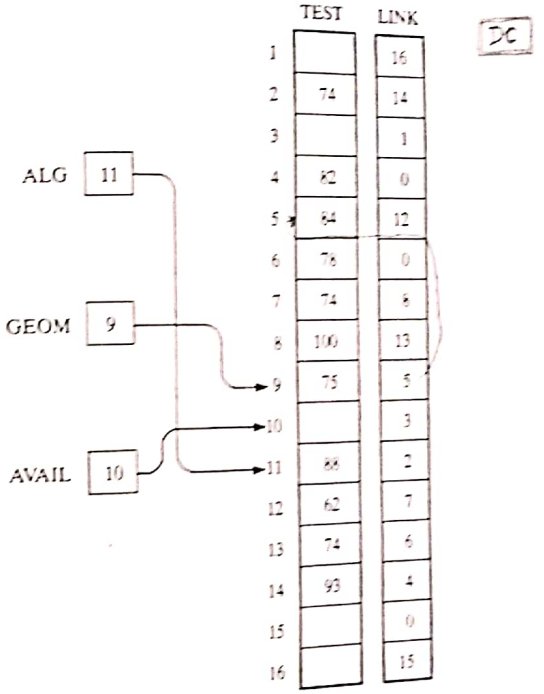


Fig. 5-19

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.

1. [OVERFLOW?] 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. [Copies new data into new node.]
4. If LOC = NULL, then: [Insert as first node.]
   Set LINK[NEW] := START and START := NEW.
   Else: [Insert after node with location LOC.]
   Set LINK[NEW] := LINK[LOC] and LINK[LOC] := NEW.
   [End of If structure.]
5. Exit.

Inserting into a Sorted Linked List

Suppose ITEM is to be inserted into a sorted linked LIST. Then ITEM must be inserted between nodes A and B so that

$$\text{INFO(A)} < \text{ITEM} \leq \text{INFO(B)}$$

The following is a procedure which finds the location LOC of node A, that is, which finds the location LOC of the last node in LIST whose value is less than ITEM.

Traverse the list, using a pointer variable PTR and comparing ITEM with INFO[PTR] at each node. While traversing, keep track of the location of the preceding node by using a pointer variable SAVE, as pictured in Fig. 5-20. Thus SAVE and PTR are updated by the assignments

$$\text{SAVE} := \text{PTR} \quad \text{and} \quad \text{PTR} := \text{LINK[PTR]}$$

The traversing continues as long as INFO[PTR] > ITEM, or in other words, the traversing stops as soon as ITEM ≤ INFO[PTR]. Then PTR points to node B, so SAVE will contain the location of the node A.

The formal statement of our procedure follows. The cases where the list is empty or where ITEM < INFO[START], so LOC = NULL, are treated separately, since they do not involve the variable SAVE.

Procedure 5.6: FINDA(INFO, LINK, START, ITEM, LOC)
This procedure finds the location LOC of the last node in a sorted list such that INFO[LOC] < ITEM, or sets LOC = NULL.

1. [List empty?] If START = NULL, then: Set LOC := NULL, and Return.
2. [Special case?] If ITEM < INFO[START], then: Set LOC := NULL, and Return.
3. Set SAVE := START and PTR := LINK[START]. [Initializes pointers.]
4. Repeat Steps 5 and 6 while PTR ≠ NULL.
5. If ITEM < INFO[PTR], then:
   Set LOC := SAVE, and Return.
   [End of If structure.]
6. Set SAVE := PTR and PTR := LINK[PTR]. [Updates pointers.]
   [End of Step 4 loop.]
7. Set LOC := SAVE.
8. Return.

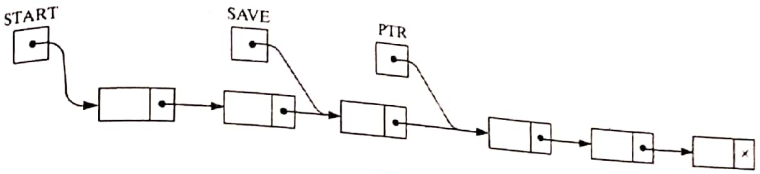


Fig. 5-20

Now we have all the components to present an algorithm which inserts ITEM into a linked list. The simplicity of the algorithm comes from using the previous two procedures.

Algorithm 5.7: INSSRT(INFO, LINK, START, AVAIL, ITEM)
This algorithm inserts ITEM into a sorted linked list.

1. [Use Procedure 5.6 to find the location of the node preceding ITEM.]
   Call FINDA(INFO, LINK, START, ITEM, LOC).
2. [Use Algorithm 5.5 to insert ITEM after the node with location LOC.]
   Call INSLOC(INFO, LINK, START, AVAIL, LOC, ITEM).
3. Exit.

EXAMPLE 5.15

Consider the alphabetized list of patients in Fig. 5-9. Suppose Jones is to be added to the list of patients. We simulate Algorithm 5.7, or more specifically, we simulate Procedure 5.6 and then Algorithm 5.5. Observe that ITEM = Jones and INFO = BED.

(*a*) FINDA(BED, LINK, START, ITEM, LOC)
1. Since START ≠ NULL, control is transferred to Step 2.
2. Since BED[5] = Adams < Jones, control is transferred to Step 3.
3. SAVE = 5 and PTR = LINK[5] = 3.
4. Steps 5 and 6 are repeated as follows:
   (*a*) BED[3] = Dean < Jones, so SAVE = 3 and PTR = LINK[3] = 11.
   (*b*) BED[11] = Fields < Jones, so SAVE = 11 and PTR = LINK[11] = 8.
   (*c*) BED[8] = Green < Jones, so SAVE = 8 and PTR = LINK[8] = 1.
   (*d*) Since BED[1] = Kirk > Jones, we have:
   LOC = SAVE = 8 and Return.

(*b*) INSLOC(BED, LINK, START, AVAIL, LOC, ITEM) [Here LOC = 8.]
1. Since AVAIL ≠ NULL, control is transferred to Step 2.
2. NEW = 10 and AVAIL = LINK[10] = 2.
3. BED[10] = Jones.
4. Since LOC ≠ NULL we have:
   LINK[10] = LINK[8] = 1 and LINK[8] = NEW = 10.
5. Exit.

Figure 5-21 shows the data structure after Jones is added to the patient list. We emphasize that only three pointers have been changed, AVAIL, LINK[10] and LINK[8].

Copying

Suppose we want to copy all or part of a given list, or suppose we want to form a new list that is the concatenation of two given lists. This can be done by defining a null list and then adding the appropriate elements to the list, one by one, by various insertion algorithms. A null list is defined by simply choosing a variable name or pointer for the list, such as NAME, and then setting NAME := NULL. These algorithms are covered in the problem sections.

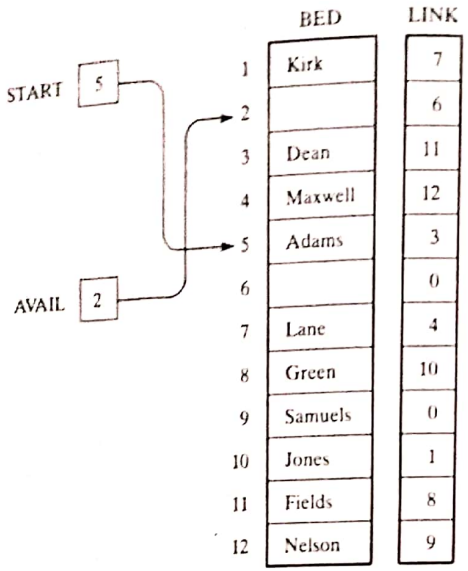


Fig. 5-21

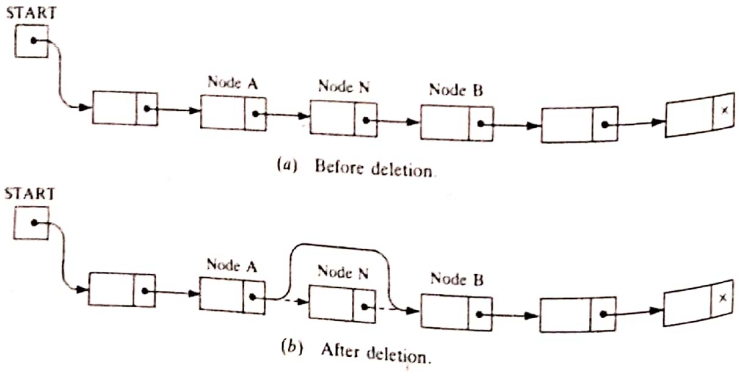
5.8 DELETION FROM A LINKED LIST

Let LIST be a linked list with a node N between nodes A and B, as pictured in Fig. 5-22(*a*). Suppose node N is to be deleted from the linked list. The schematic diagram of such a deletion appears in Fig. 5-22(*b*). The deletion occurs as soon as the nextpointer field of node A is changed so that it points to node B. (Accordingly, when performing deletions, one must keep track of the address of the node which immediately precedes the node that is to be deleted.)

Suppose our linked list is maintained in memory in the form

LIST(INFO, LINK, START, AVAIL)

Figure 5-22 does not take into account the fact that, when a node N is deleted from our list, we will



(*a*) Before deletion.

(*b*) After deletion.

Fig. 5-22

immediately return its memory space to the AVAIL list. Specifically, for easier processing, it will be returned to the beginning of the AVAIL list. Thus a more exact schematic diagram of such a deletion is the one in Fig. 5-23. Observe that three pointer fields are changed as follows:

(1) The nextpointer field of node A now points to node B, where node N previously pointed.

(2) The nextpointer field of N now points to the original first node in the free pool, where AVAIL previously pointed.

(3) AVAIL now points to the deleted node N.

There are also two special cases. If the deleted node N is the first node in the list, then START will point to node B; and if the deleted node N is the last node in the list, then node A will contain the NULL pointer.

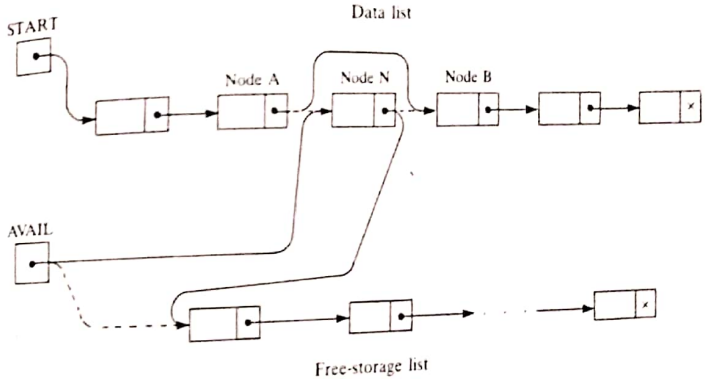


Fig. 5-23

EXAMPLE 5.16

(*a*) Consider Fig. 5-21, the list of patients in the hospital ward. Suppose Green is discharged, so that BED[8] is now empty. Then, in order to maintain the linked list, the following three changes in the pointer fields must be executed:

LINK[11] = 10 LINK[8] = 2 AVAIL = 8

By the first change, Fields, who originally preceded Green, now points to Jones, who originally followed Green. The second and third changes add the new empty bed to the AVAIL list. We emphasize that, before making the deletion, we had to find the node BED[11], which originally pointed to the deleted node BED[8].

(*b*) Consider Fig. 5-12, the list of brokers and their customers. Suppose Teller, the first customer of Nelson, is deleted from the list of customers. Then, in order to maintain the linked lists, the following three changes in the pointer fields must be executed:

POINT[4] = 10 LINK[9] = 11 AVAIL = 9

By the first change, Nelson now points to his original second customer, Jones. The second and third changes add the new empty node to the AVAIL list.

(*c*) Suppose the data elements E, B and C are deleted, one after the other, from the list in Fig. 5-16. The new list is pictured in Fig. 5-24. Observe that now the first three available nodes are:

INFO[3], which originally contained C
INFO[2], which originally contained B
INFO[5], which originally contained E

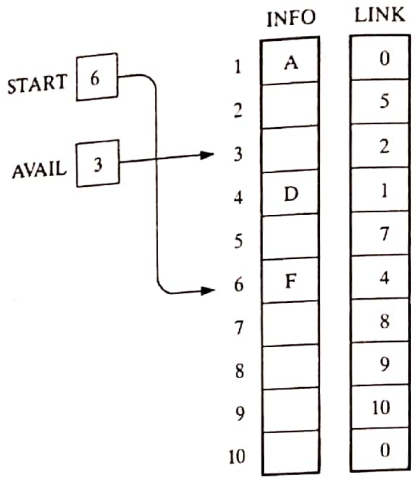


Fig. 5-24

Observe that the order of the nodes in the AVAIL list is the reverse of the order in which the nodes have been deleted from the list.

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:

$$\text{LINK[LOC]} := \text{AVAIL} \quad \text{and then} \quad \text{AVAIL} := \text{LOC}$$

These two operations are pictured in Fig. 5-25.

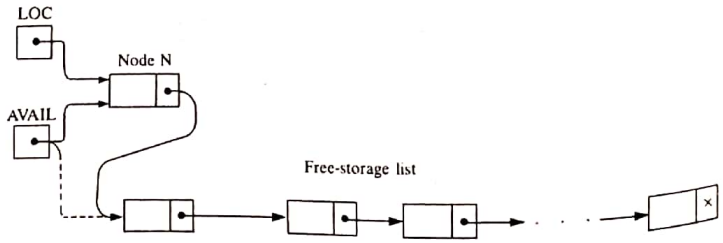


Fig. 5-25 LINK[LOC] := AVAIL and AVAIL := LOC.

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.

Figure 5-26 is the schematic diagram of the assignment

$$\text{START} := \text{LINK[START]}$$

which effectively deletes the first node from the list. This covers the case when N is the first node.

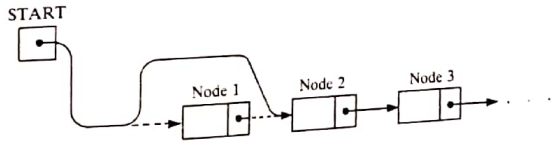


Fig. 5-26 START := LINK[START].

Figure 5-27 is the schematic diagram of the assignment

$$\text{LINK[LOCP]} := \text{LINK[LOC]}$$

which effectively deletes the node N when N is not the first node.

The simplicity of the algorithm comes from the fact that we are already given the location LOCP of the node which precedes node N. In many applications, we must first find LOCP.

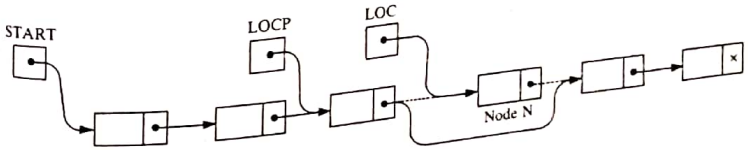


Fig. 5-27 LINK[LOCP] = LINK[LOC]

Deleting the Node with a Given ITEM of Information

Let LIST be a linked list in memory. Suppose we are given an ITEM of information and we want to delete from the LIST the first node N which contains ITEM. (If ITEM is a key value, then only one node can contain ITEM.) Recall that before we can delete N from the list, we need to know the location of the node preceding N. Accordingly, first we give a procedure which finds the location LOC of the node N containing ITEM and the location LOCP of the node preceding node N. If N is the first node, we set LOCP = NULL; and if ITEM does not appear in LIST, we set LOC = NULL. (The procedure is similar to Procedure 5.6.)

Traverse the list, using a pointer variable PTR and comparing ITEM with INFO[PTR] at each node. While traversing, keep track of the location of the preceding node by using a pointer variable SAVE, as pictured in Fig. 5-20. Thus SAVE and PTR are updated by the assignments

$$SAVE := PTR \quad \text{and} \quad PTR := LINK[PTR]$$

The traversing continues as long as INFO[PTR] ≠ ITEM, or in other words, the traversing stops as soon as ITEM = INFO[PTR]. Then PTR contains the location LOC of node N and SAVE contains the location LOCP of the node preceding N.

The formal statement of our procedure follows. The cases where the list is empty or where INFO[START] = ITEM (i.e., where node N is the first node) are treated separately, since they do not involve the variable SAVE.

Procedure 5.9: FINDB(INFO, LINK, START, ITEM, LOC, LOCP)
This procedure finds the location LOC of the first node N which contains ITEM and the location LOCP of the node preceding N. If ITEM does not appear in the list, then the procedure sets LOC = NULL; and if ITEM appears in the first node, then it sets LOCP = NULL.

1. [List empty?] If START = NULL, then:
   Set LOC := NULL and LOCP := NULL, and Return.
   [End of If structure.]
2. [ITEM in first node?] If INFO[START] = ITEM, then:
   Set LOC := START and LOCP := NULL, and Return.
   [End of If structure.]
3. Set SAVE := START and PTR := LINK[START]. [Initializes pointers.]
4. Repeat Steps 5 and 6 while PTR ≠ NULL:
5. If INFO[PTR] = ITEM, then:
   Set LOC := PTR and LOCP := SAVE, and Return.
   [End of If structure.]
6. Set SAVE := PTR and PTR := LINK[PTR]. [Updates pointers.]
   [End of Step 4 loop.]
7. Set LOC := NULL. [Search unsuccessful.]
8. Return.

Now we can easily present an algorithm to delete the first node N from a linked list which contains a given ITEM of information. The simplicity of the algorithm comes from the fact that the task of finding the location of N and the location of its preceding node has already been done in Procedure 5.9.

Algorithm 5.10: DELETE(INFO, LINK, START, AVAIL, ITEM)
This algorithm deletes from a linked list the first node N which contains the given ITEM of information.

1. [Use Procedure 5.9 to find the location of N and its preceding node.]
   Call FINDB(INFO, LINK, START, ITEM, LOC, LOCP).
2. If LOC = NULL, then: Write: ITEM not in list, and Exit.
3. [Delete node.]
   If LOCP = NULL, then:
      Set START := LINK[START]. [Deletes first node.]
   Else:
      Set LINK[LOCP] := LINK[LOC].
   [End of If structure.]
4. [Return deleted node to the AVAIL list.]
   Set LINK[LOC] := AVAIL and AVAIL := LOC.
5. Exit.

Remark: The reader may have noticed that Steps 3 and 4 in Algorithm 5.10 already appear in Algorithm 5.8. In other words, we could replace the steps by the following Call statement:

Call DEL(INFO, LINK, START, AVAIL, LOC, LOCP)

This would conform to the usual programming style of modularity.

EXAMPLE 5.17

Consider the list of patients in Fig. 5-21. Suppose the patient Green is discharged. We simulate Procedure 5.9 to find the location LOC of Green and the location LOCP of the patient preceding Green. Then we simulate Algorithm 5.10 to delete Green from the list. Here ITEM = Green, INFO = BED, START = 5 and AVAIL = 2.

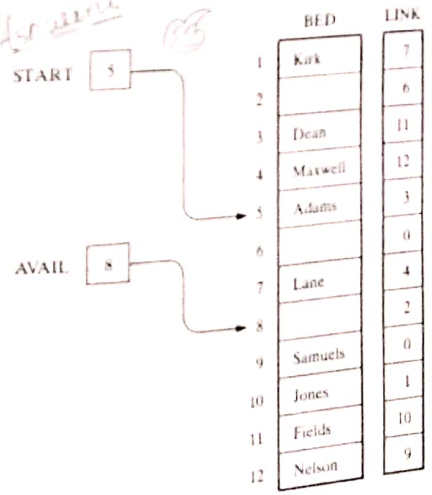


Fig. 5-28

Deleting the Node with a Given ITEM of Information

Let LIST be a linked list in memory. Suppose we are given an ITEM of information and we want to delete from the LIST the first node N which contains ITEM. (If ITEM is a key value, then only one node can contain ITEM.) Recall that before we can delete N from the list, we need to know the location of the node preceding N. Accordingly, first we give a procedure which finds the location LOC of the node N containing ITEM and the location LOCP of the node preceding node N. If N is the first node, we set LOCP = NULL, and if ITEM does not appear in LIST, we set LOC = NULL. (This procedure is similar to Procedure 5.6.)

Traverse the list, using a pointer variable PTR and comparing ITEM with INFO[PTR] at each node. While traversing, keep track of the location of the preceding node by using a pointer variable SAVE, as pictured in Fig. 5-20. Thus SAVE and PTR are updated by the assignments

SAVE := PTR and PTR := LINK[PTR]

The traversing continues as long as INFO[PTR] ≠ ITEM, or in other words, the traversing stops as soon as ITEM = INFO[PTR]. Then PTR contains the location LOC of node N and SAVE contains the location LOCP of the node preceding N.

The formal statement of our procedure follows. The cases where the list is empty or where INFO[START] = ITEM (i.e., where node N is the first node) are treated separately, since they do not involve the variable SAVE.

Procedure 5.9: FINDB(INFO, LINK, START, ITEM, LOC, LOCP)
This procedure finds the location LOC of the first node N which contains ITEM and the location LOCP of the node preceding N. If ITEM does not appear in the list, then the procedure sets LOC = NULL; and if ITEM appears in the first node, then it sets LOCP = NULL.

1. [List empty?] If START = NULL, then:
   Set LOC := NULL and LOCP := NULL, and Return.
   [End of If structure.]
2. [ITEM in first node?] If INFO[START] = ITEM, then:
   Set LOC := START and LOCP = NULL, and Return.
   [End of If structure.]
3. Set SAVE := START and PTR := LINK[START]. [Initializes pointers.]
4. Repeat Steps 5 and 6 while PTR ≠ NULL.
5. If INFO[PTR] = ITEM, then:
   Set LOC := PTR and LOCP := SAVE, and Return.
   [End of If structure.]
6. Set SAVE := PTR and PTR := LINK[PTR]. [Updates pointers.]
   [End of Step 4 loop.]
7. Set LOC := NULL. [Search unsuccessful.]
8. Return.

Now we can easily present an algorithm to delete the first node N from a linked list which contains a given ITEM of information. The simplicity of the algorithm comes from the fact that the task of finding the location of N and the location of its preceding node has already been done in Procedure 5.9.

Algorithm 5.10: DELETE(INFO, LINK, START, AVAIL, ITEM)
This algorithm deletes from a linked list the first node N which contains the given ITEM of information.

1. [Use Procedure 5.9 to find the location of N and its preceding node.]
   Call FINDB(INFO, LINK, START, ITEM, LOC, LOCP)
2. If LOC = NULL, then: Write: ITEM not in list, and Exit
3. [Delete node.]
   If LOCP = NULL, then:
      Set START := LINK[START]. [Deletes first node.]
   Else:
      Set LINK[LOCP] := LINK[LOC].
   [End of If structure.]
4. [Return deleted node to the AVAIL list.]
   Set LINK[LOC] := AVAIL and AVAIL := LOC.
5. Exit.

Remark: The reader may have noticed that Steps 3 and 4 in Algorithm 5.10 already appear in Algorithm 5.8. In other words, we could replace the steps by the following Call statement:

Call DEL(INFO, LINK, START, AVAIL, LOC, LOCP)

This would conform to the usual programming style of modularity.

EXAMPLE 5.17

Consider the list of patients in Fig. 5-21. Suppose the patient Green is discharged. We simulate Procedure 5.9 to find the location LOC of Green and the location LOCP of the patient preceding Green. Then we simulate Algorithm 5.10 to delete Green from the list. Here ITEM = Green, INFO = BED, START = 5 and AVAIL = 2.

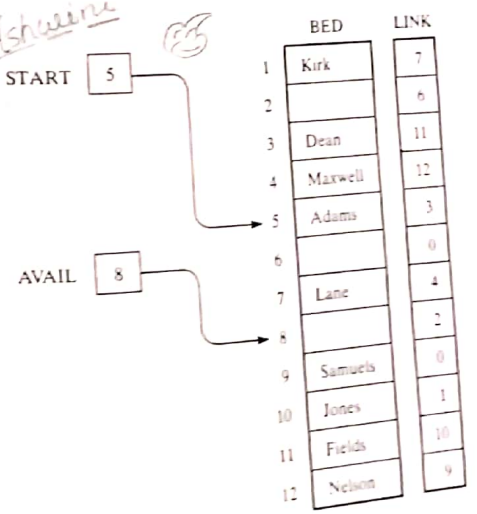


Fig. 5-28

(*a*) FINDB(BED, LINK, START, ITEM, LOC, LOCP)
1. Since START ≠ NULL, control is transferred to Step 2.
2. Since BED[5] = Adams ≠ Green, control is transferred to Step 3.
3. SAVE = 5 and PTR = LINK[5] = 3.
4. Steps 5 and 6 are repeated as follows:
   (*a*) BED[3] = Dean ≠ Green, so SAVE = 3 and PTR = LINK[3] = 11.
   (*b*) BED[11] = Fields ≠ Green, so SAVE = 11 and PTR = LINK[11] = 8.
   (*c*) BED[8] = Green, so we have:
   LOC = PTR = 8 and LOCP = SAVE = 11, and Return.

(*b*) DELLOC(BED, LINK, START, AVAIL, ITEM)
1. Call FINDB(BED, LINK, START, ITEM, LOC, LOCP). [Hence LOC = 8 and LOCP = 11.]
2. Since LOC ≠ NULL, control is transferred to Step 3.
3. Since LOCP ≠ NULL, we have:
   LINK[11] = LINK[8] = 10.
4. LINK[8] = 2 and AVAIL = 8.
5. Exit.

Figure 5-28 shows the data structure after Green is removed from the patient list. We emphasize that only three pointers have been changed, LINK[11], LINK[8] and AVAIL.

5.9 HEADER LINKED LISTS

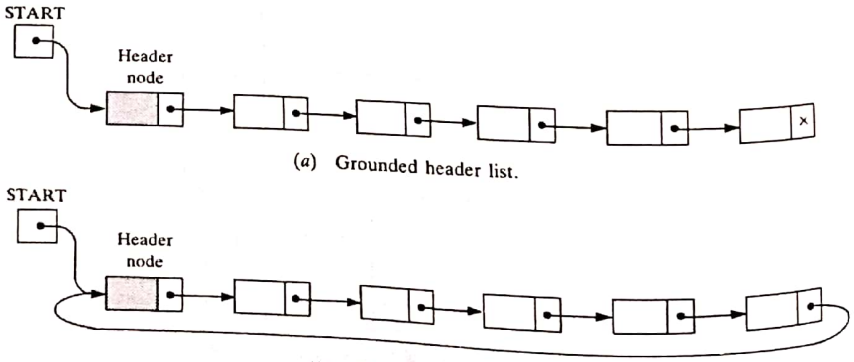
A *header* linked list is a linked list which always contains a special node, called the *header node*, at the beginning of the list. The following are two kinds of widely used header lists:

(1) A *grounded header list* is a header list where the last node contains the null pointer. (The term "grounded" comes from the fact that many texts use the electrical ground symbol to indicate the null pointer.)

(2) A *circular header list* is a header list where the last node points back to the header node

Figure 5-29 contains schematic diagrams of these header lists. Unless otherwise stated or implied, our header lists will always be circular. Accordingly, in such a case, the header node also acts as a sentinel indicating the end of the list.

Observe that the list pointer START always points to the header node. Accordingly, LINK[START] = NULL indicates that a grounded header list is empty, and LINK[START] = START indicates that a circular header list is empty.



(*a*) Grounded header list.

(*b*) Circular header list.

Fig. 5-29

Although our data may be maintained by header lists in memory, the AVAIL list will always be maintained as an ordinary linked list.

EXAMPLE 5.18

Consider the personnel file in Fig. 5-11. The data may be organized as a header list as in Fig. 5-30. Observe that LOC = 5 is now the location of the header record. Therefore, START = 5, and since Rubin is the last employee, LINK[10] = 5. The header record may also be used to store information about the entire file. For example, we let SSN[5] = 9 indicate the number of employees, and we let SALARY[5] = 191 600 indicate the total salary paid to the employees.

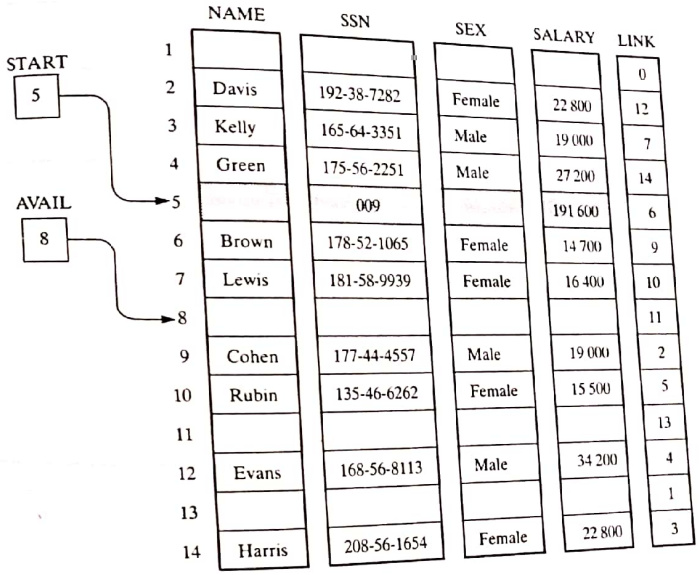


Fig. 5-30

The term "node," by itself, normally refers to an ordinary node, not the header node, when used with header lists. Thus the *first node* in a header list is the node following the header node, and the location of the first node is LINK[START], not START, as with ordinary linked lists.

Algorithm 5.11, which uses a pointer variable PTR to traverse a circular header list, is essentially the same as Algorithm 5.1, which traverses an ordinary linked list, except that now the algorithm (1) begins with PTR = LINK[START] (not PTR = START) and (2) ends when PTR = START (not PTR = NULL).

Circular header lists are frequently used instead of ordinary linked lists because many operations are much easier to state and implement using header lists. This comes from the following two properties of circular header lists:

(1) The null pointer is not used, and hence all pointers contain valid addresses.

(2) Every (ordinary) node has a predecessor, so the first node may not require a special case.

The next example illustrates the usefulness of these properties.

Algorithm 5.11: (Traversing a Circular Header List) Let LIST be a circular header list in memory. This algorithm traverses LIST, applying an operation PROCESS to each node of LIST.

1. Set PTR := LINK[START]. [Initializes the pointer PTR.]
2. Repeat Steps 3 and 4 while PTR ≠ START:
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.

EXAMPLE 5.19

Suppose LIST is a linked list in memory, and suppose a specific ITEM of information is given.

(*a*) Algorithm 5.2 finds the location LOC of the first node in LIST which contains ITEM when LIST is an ordinary linked list. The following is such an algorithm when LIST is a circular header list.

Algorithm 5.12: SRCHHL(INFO, LINK, START, ITEM, LOC)
LIST is a circular header 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 := LINK[START].
2. Repeat while INFO[PTR] ≠ ITEM and PTR ≠ START:
   Set PTR := LINK[PTR]. [PTR now points to the next node.]
   [End of loop.]
3. If INFO[PTR] = ITEM, then:
   Set LOC := PTR.
   Else:
   Set LOC := NULL.
   [End of If structure.]
4. Exit.

The two tests which control the searching loop (Step 2 in Algorithm 5.12) were not performed at the same time in the algorithm for ordinary linked lists; that is, we did not let Algorithm 5.2 use the analogous statement

Repeat while INFO[PTR] ≠ ITEM and PTR ≠ NULL:

because for ordinary linked lists INFO[PTR] is not defined when PTR = NULL.

(*b*) Procedure 5.9 finds the location LOC of the first node N which contains ITEM and also the location LOCP of the node preceding N when LIST is an ordinary linked list. The following is such a procedure when LIST is a circular header list.

Procedure 5.13: FINDBHL(INFO, LINK, START, ITEM, LOC, LOCP)

1. Set SAVE := START and PTR := LINK[START]. [Initializes pointers.]
2. Repeat while INFO[PTR] ≠ ITEM and PTR ≠ START.
   Set SAVE := PTR and PTR := LINK[PTR]. [Updates pointers.]
   [End of loop.]
3. If INFO[PTR] = ITEM, then:
   Set LOC := PTR and LOCP := SAVE.
   Else:
   Set LOC := NULL and LOCP := SAVE.
   [End of If structure.]
4. Exit.

Observe the simplicity of this procedure compared with Procedure 5.9. Here we did not have to consider the special case when ITEM appears in the first node, and here we can perform at the same time the two tests which control the loop.

(*c*) Algorithm 5.10 deletes the first node N which contains ITEM when LIST is an ordinary linked list. The following is such an algorithm when LIST is a circular header list.

Algorithm 5.14: DELLOCHL(INFO, LINK, START, AVAIL, ITEM)

1. [Use Procedure 5.13 to find the location of N and its preceding node.]
   Call LINDBHL(INFO, LINK, START, ITEM, LOC, LOCP).
2. If LOC = NULL, then: Write: ITEM not in list, and Exit.
3. Set LINK[LOCP] := LINK[LOC]. [Deletes node.]
4. [Return deleted node to the AVAIL list.]
   Set LINK[LOC] := AVAIL and AVAIL := LOC.
5. Exit.

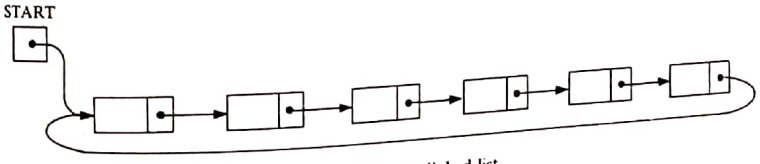
Again we did not have to consider the special case when ITEM appears in the first node, as we did in Algorithm 5.10.

Remark: There are two other variations of linked lists which sometimes appear in the literature:

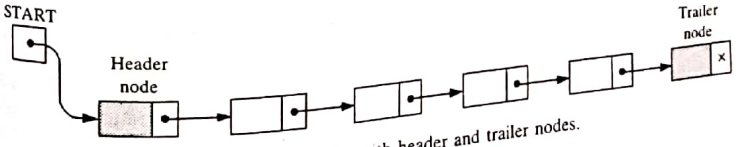
(1) A linked list whose last node points back to the first node instead of containing the null pointer, called a *circular list*

(2) A linked list which contains both a special header node at the beginning of the list and a special trailer node at the end of the list

Figure 5-31 contains schematic diagrams of these lists.



(*a*) Circular linked list.



(*b*) Linked list with header and trailer nodes.

Fig. 5-31

Polynomials

Header linked lists are frequently used for maintaining polynomials in memory. The header node plays an important part in this representation, since it is needed to represent the zero polynomial. This representation of polynomials will be presented in the context of a specific example.

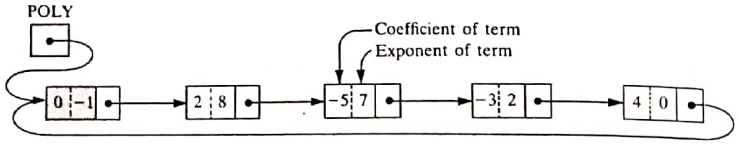
EXAMPLE 5.20

Let $p(x)$ denote the following polynomial in one variable (containing four nonzero terms):

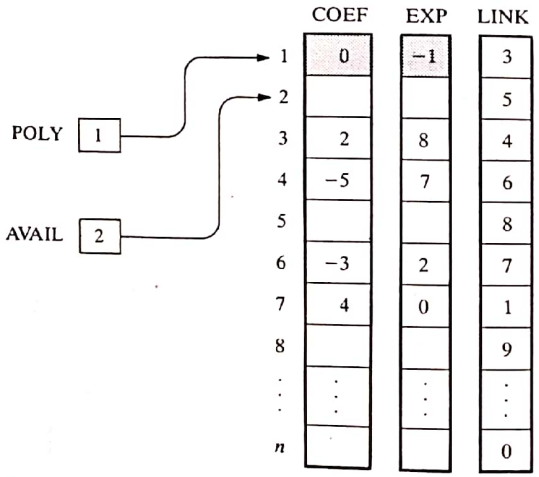
$$p(x) = 2x^8 - 5x^7 - 3x^2 + 4$$

Then $p(x)$ may be represented by the header list pictured in Fig. 5-32(a), where each node corresponds to a nonzero term of $p(x)$. Specifically, the information part of the node is divided into two fields representing, respectively, the coefficient and the exponent of the corresponding term, and the nodes are linked according to decreasing degree.

Observe that the list pointer variable POLY points to the header node, whose exponent field is assigned a negative number, in this case −1. Here the array representation of the list will require three linear arrays, which we will call COEF, EXP and LINK. One such representation appears in Fig. 5-32(b).



(a)



(b)

Fig. 5-32 $p(x) = 2x^8 - 5x^7 - 3x^2 + 4$.

5.10 TWO-WAY LISTS

Each list discussed above is called a one-way list, since there is only one way that the list can be traversed. That is, beginning with the list pointer variable START, which points to the first node or the header node, and using the next pointer field LINK to point to the next node in the list, we can traverse the list in only one direction. Furthermore, given the location LOC of a node N in such a list, one has immediate access to the next node in the list (by evaluating LINK[LOC]), but one does not have access to the preceding node without traversing part of the list. This means, in particular, that one must traverse that part of the list preceding N in order to delete N from the list.

This section introduces a new list structure, called a two-way list, which can be traversed in two directions: in the usual forward direction from the beginning of the list to the end, or in the backward direction from the end of the list to the beginning. Furthermore, given the location LOC of a node N in the list, one now has immediate access to both the next node and the preceding node in the list. This means, in particular, that one is able to delete N from the list without traversing any part of the list.

A *two-way list* is a linear collection of data elements, called *nodes*, where each node N is divided into three parts:

(1) An information field INFO which contains the data of N
(2) A pointer field FORW which contains the location of the next node in the list
(3) A pointer field BACK which contains the location of the preceding node in the list

The list also requires two list pointer variables: FIRST, which points to the first node in the list, and LAST, which points to the last node in the list. Figure 5-33 contains a schematic diagram of such a list. Observe that the null pointer appears in the FORW field of the last node in the list and also in the BACK field of the first node in the list.

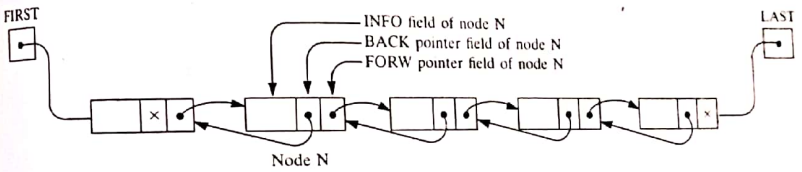


Fig. 5-33 Two-way list.

Observe that, using the variable FIRST and the pointer field FORW, we can traverse a two-way list in the forward direction as before. On the other hand, using the variable LAST and the pointer field BACK, we can also traverse the list in the backward direction.

Suppose LOCA and LOCB are the locations, respectively, of nodes A and B in a two-way list. Then the way that the pointers FORW and BACK are defined gives us the following:

Pointer property: FORW[LOCA] = LOCB if and only if BACK[LOCB] = LOCA

In other words, the statement that node B follows node A is equivalent to the statement that node A precedes node B.

Two-way lists may be maintained in memory by means of linear arrays in the same way as one-way lists except that now we require two pointer arrays, FORW and BACK, instead of one pointer array LINK, and we require two list pointer variables, FIRST and LAST, instead of one list pointer variable START. On the other hand, the list AVAIL of available space in the arrays will still be maintained as a one-way list—using FORW as the pointer field—since we delete and insert nodes only at the beginning of the AVAIL list.

EXAMPLE 5.21

Consider again the data in Fig. 5-9, the 9 patients in a ward with 12 beds. Figure 5-34 shows how the alphabetical listing of the patients can be organized into a two-way list. Observe that the values of FIRST and the pointer field FORW are the same, respectively, as the values of START and the array LINK; hence the list can be traversed alphabetically as before. On the other hand, using LAST and the pointer array BACK, the list can also be traversed in reverse alphabetical order. That is, LAST points to Samuels, the pointer field BACK of Samuels points to Nelson, the pointer field BACK of Nelson points to Maxwell, and so on.

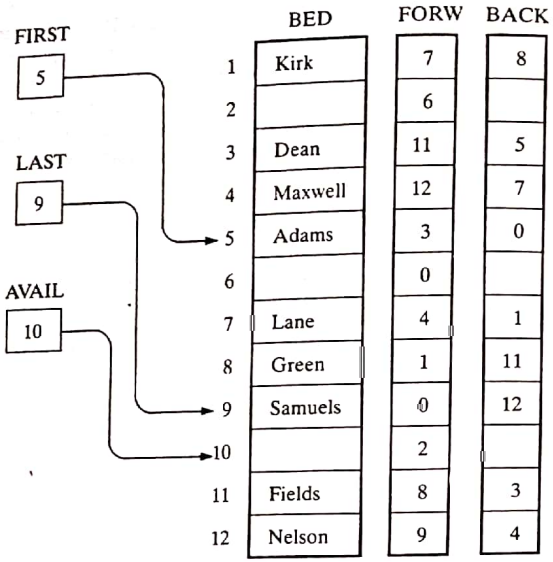


Fig. 5-34

Two-Way Header Lists

The advantages of a two-way list and a circular header list may be combined into a two-way circular header list as pictured in Fig. 5-35. The list is circular because the two end nodes point back to the header node. Observe that such a two-way list requires only one list pointer variable START, which points to the header node. This is because the two pointers in the header node point to the two ends of the list.

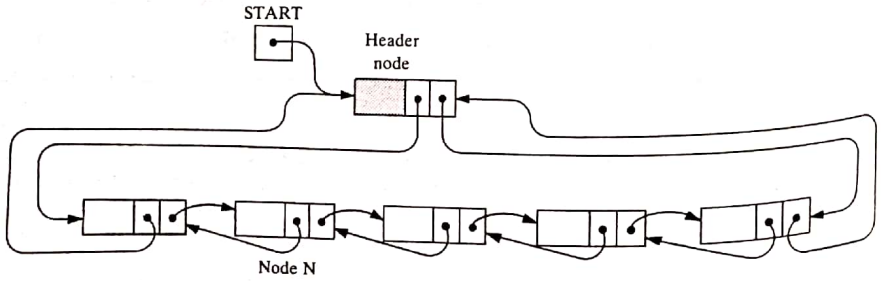


Fig. 5-35 Two-way circular header list.

EXAMPLE 5.22

Consider the personnel file in Fig. 5-30, which is organized as a circular header list. The data may be organized into a two-way circular header list by simply adding another array BACK which gives the locations of preceding nodes. Such a structure is pictured in Fig. 5-36, where LINK has been renamed FORW. Again the AVAIL list is maintained only as a one-way list.

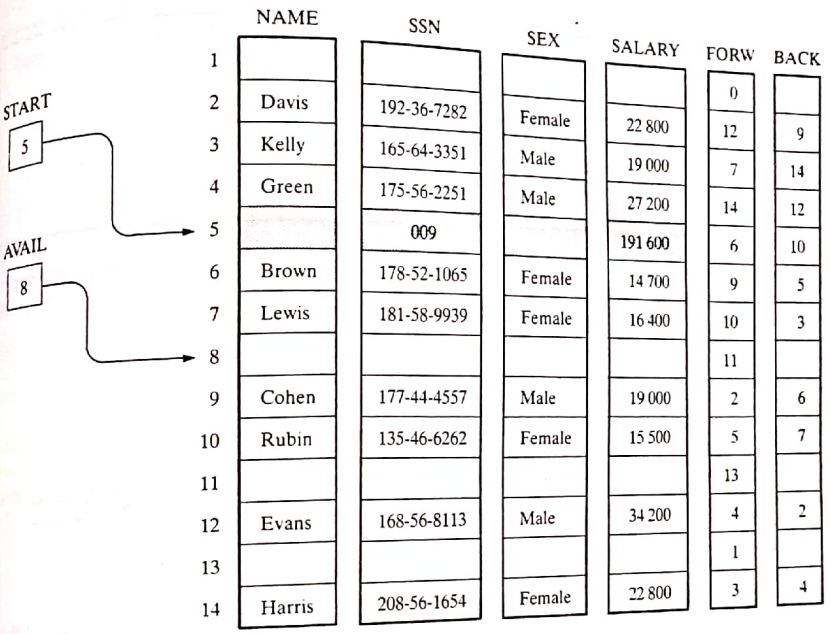


Fig. 5-36

Operations on Two-Way Lists

Suppose LIST is a two-way list in memory. This subsection discusses a number of operations on LIST.

Traversing. Suppose we want to traverse LIST in order to process each node exactly once. Then we can use Algorithm 5.1 if LIST is an ordinary two-way list, or we can use Algorithm 5.11 if LIST contains a header node. Here it is of no advantage that the data are organized as a two-way list rather than as a one-way list.

Searching. Suppose we are given an ITEM of information—a key value—and we want to find the location LOC of ITEM in LIST. Then we can use Algorithm 5.2 if LIST is an ordinary two-way list, or we can use Algorithm 5.12 if LIST has a header node. Here the main advantage is that we can search for ITEM in the backward direction if we have reason to suspect that ITEM appears near the end of the list. For example, suppose LIST is a list of names sorted alphabetically. If ITEM = Smith, then we would search LIST in the backward direction, but if ITEM = Davis, then we would search LIST in the forward direction.

Deleting. Suppose we are given the location LOC of a node N in LIST, and suppose we want to delete N from the list. We assume that LIST is a two-way circular header list. Note that BACK[LOC] and FORW[LOC] are the locations, respectively, of the nodes which precede and follow node N. Accordingly, as pictured in Fig. 5-37, N is deleted from the list by changing the following pair of pointers:

$$\text{FORW}[\text{BACK}[\text{LOC}]] := \text{FORW}[\text{LOC}] \quad \text{and} \quad \text{BACK}[\text{FORW}[\text{LOC}]] := \text{BACK}[\text{LOC}]$$

The deleted node N is then returned to the AVAIL list by the assignments:

$$\text{FORW}[\text{LOC}] := \text{AVAIL} \quad \text{and} \quad \text{AVAIL} := \text{LOC}$$

The formal statement of the algorithm follows.

Algorithm 5.15: DELTWL(INFO, FORW, BACK, START, AVAIL, LOC)

1. [Delete node.]
   Set FORW[BACK[LOC]] := FORW[LOC] and
   BACK[FORW[LOC]] := BACK[LOC].
2. [Return node to AVAIL list.]
   Set FORW[LOC] := AVAIL and AVAIL := LOC.
3. Exit.

Here we see one main advantage of a two-way list: If the data were organized as a one-way list, then, in order to delete N, we would have to traverse the one-way list to find the location of the node preceding N.

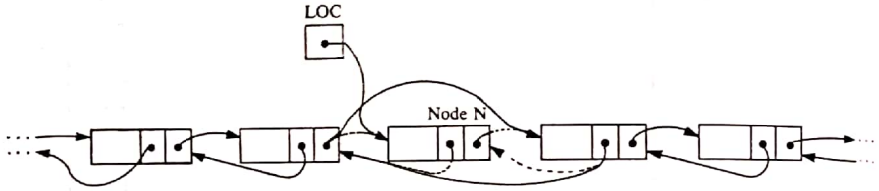


Fig. 5-37 Deleting node N.

Inserting. Suppose we are given the locations LOCA and LOCB of adjacent nodes A and B in LIST, and suppose we want to insert a given ITEM of information between nodes A and B. As with a one-way list, first we remove the first node N from the AVAIL list, using the variable NEW to keep track of its location, and then we copy the data ITEM into the node N; that is, we set:

NEW := AVAIL, AVAIL := FORW[AVAIL], INFO[NEW] := ITEM

Now, as pictured in Fig. 5-38, the node N with contents ITEM is inserted into the list by changing the following four pointers:

FORW[LOCA] := NEW, FORW[NEW] := LOCB
BACK[LOCB] := NEW, BACK[NEW] := LOCA

The formal statement of our algorithm follows.

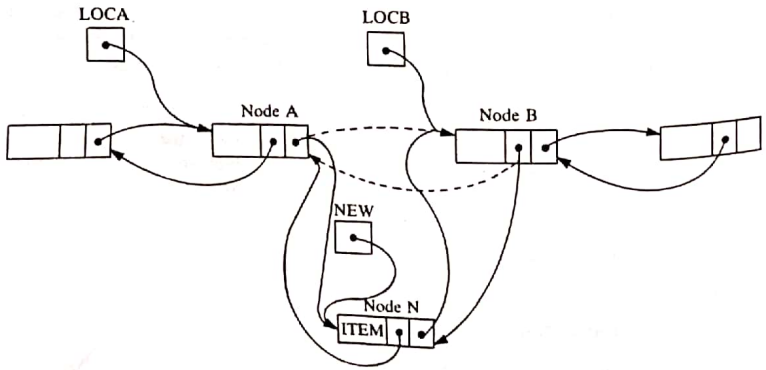


Fig. 5-38 Inserting node N.

Algorithm 5.16: INSTWL(INFO, FORW, BACK, START, AVAIL, LOCA, LOCB, ITEM)

1. [OVERFLOW?] If AVAIL = NULL, then: Write: OVERFLOW, and Exit.
2. [Remove node from AVAIL list and copy new data into node.]
   Set NEW := AVAIL, AVAIL := FORW[AVAIL], INFO[NEW] := ITEM.
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.

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.