**International Islamic University Chittagong**
Department of Computer Science & Engineering
Spring - 2021

# Course Code: CSE-2321
# Course Title: Data Structures

**Mohammed Shamsul Alam**
Professor, Dept. of CSE, IIUC

# Lecture – 11

## Linked List

# Avail list

- 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 **Avail list** or the ***free storage list*** or the ***free pool***.

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.

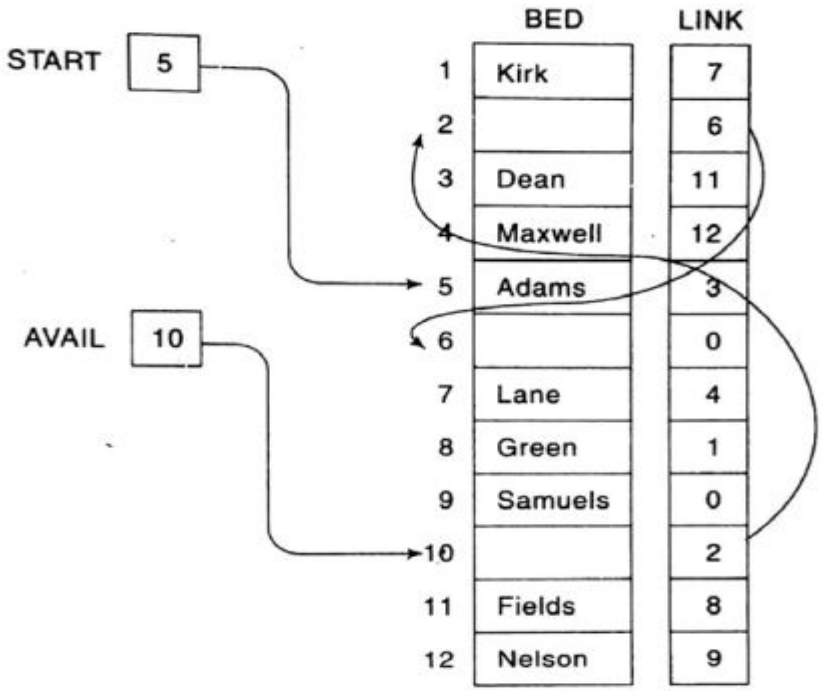



Fig. 5.9

# Garbage collection

- The operating system of a computer may periodically collect all the deleted space on to the free storage list. Any technique which does these collections is called **garbage collection**.
- The operating system will perform this operation whenever it finds the CPU idle or whenever the programs are falling short of memory space.

## 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.

# 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.

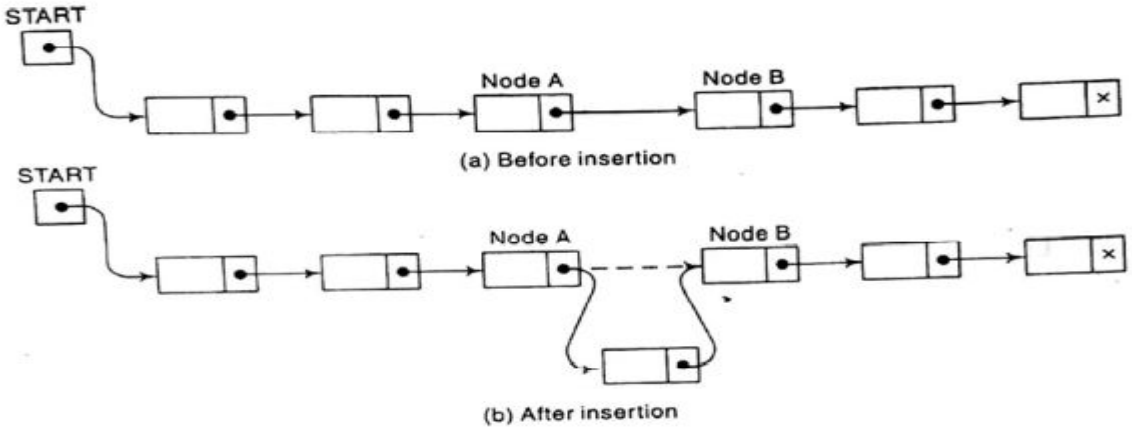



Fig. 5.14

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.

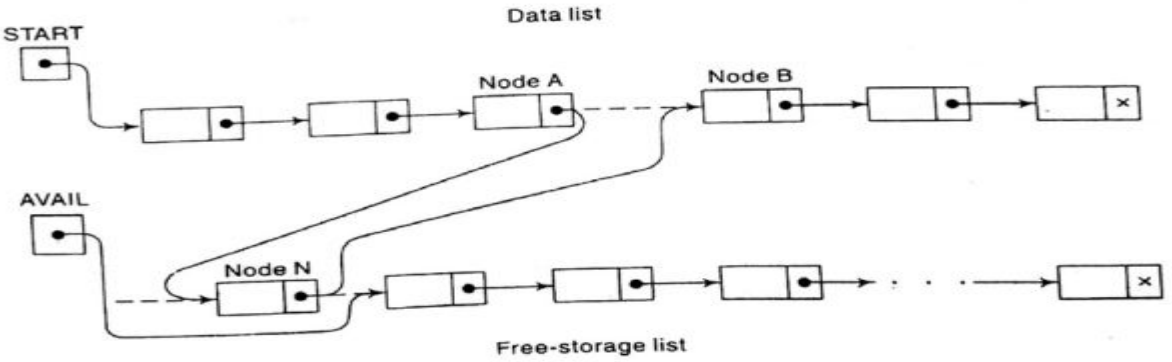



Fig. 5.15

# Inserting at the beginning of a Linked List

(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

**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.

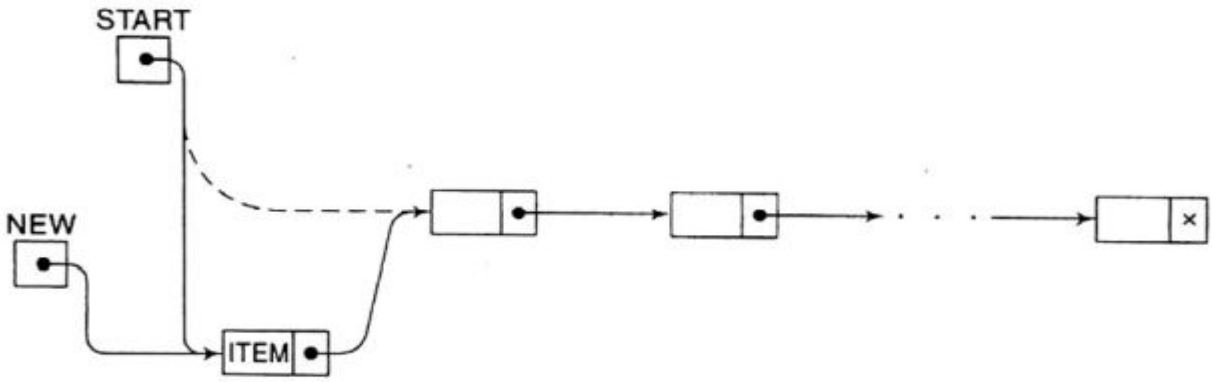



**Fig. 5.18** *Insertion at the Beginning of a List*

# Inserting after a given node

**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

**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.

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:** INSERT(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.

# Inserting into a sorted Linked list

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].

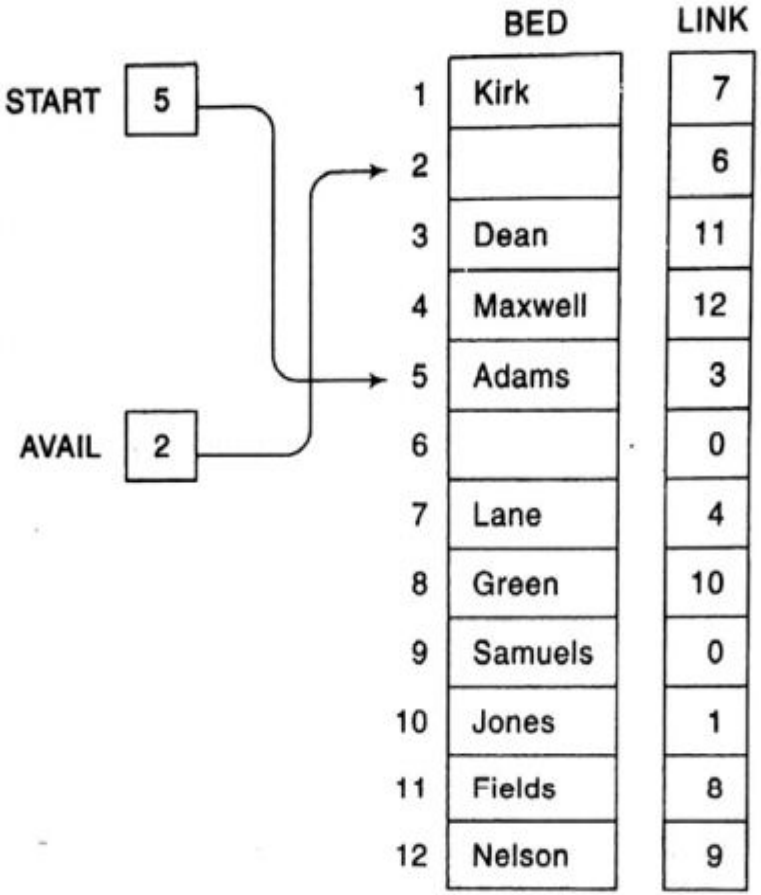



Fig. 5.21

# Deletion from a Linked List

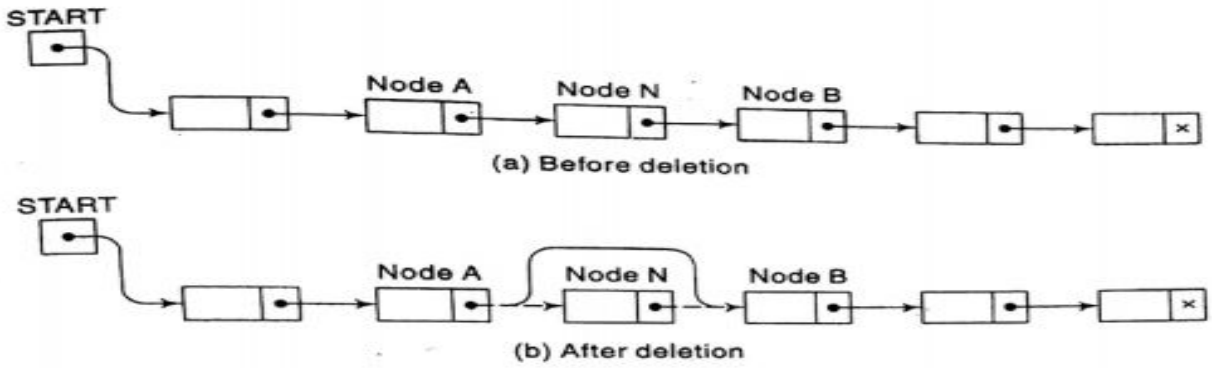



**Fig. 5.22**

Figure 5.22 does not take into account the fact that, when a node N is deleted from our list, we will 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.

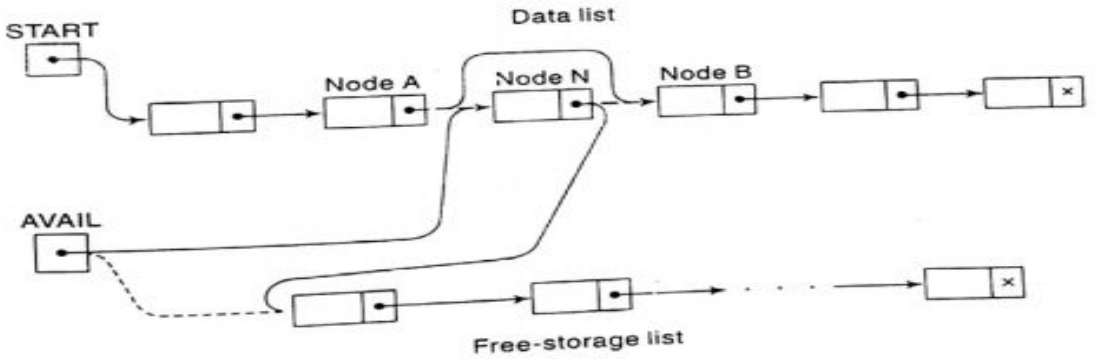



**Fig. 5.23**

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.

# Deleting the node following a given node

**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.

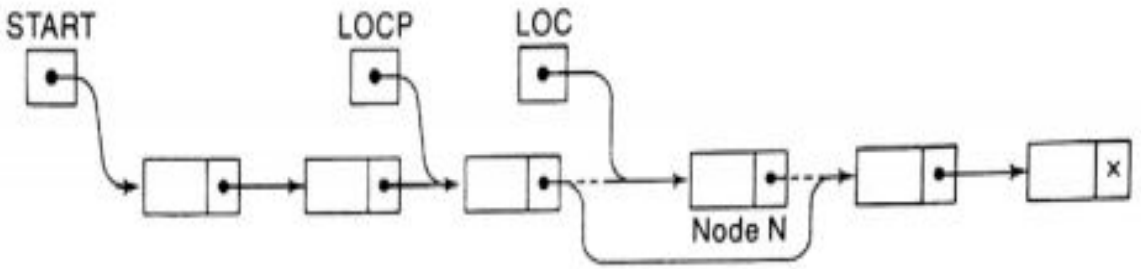



Fig. 5.27 *LINK[LOCP] := LINK[LOC]*

# Deleting the node with a given ITEM

**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.

**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.

# Deleting the node with a given ITEM

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.

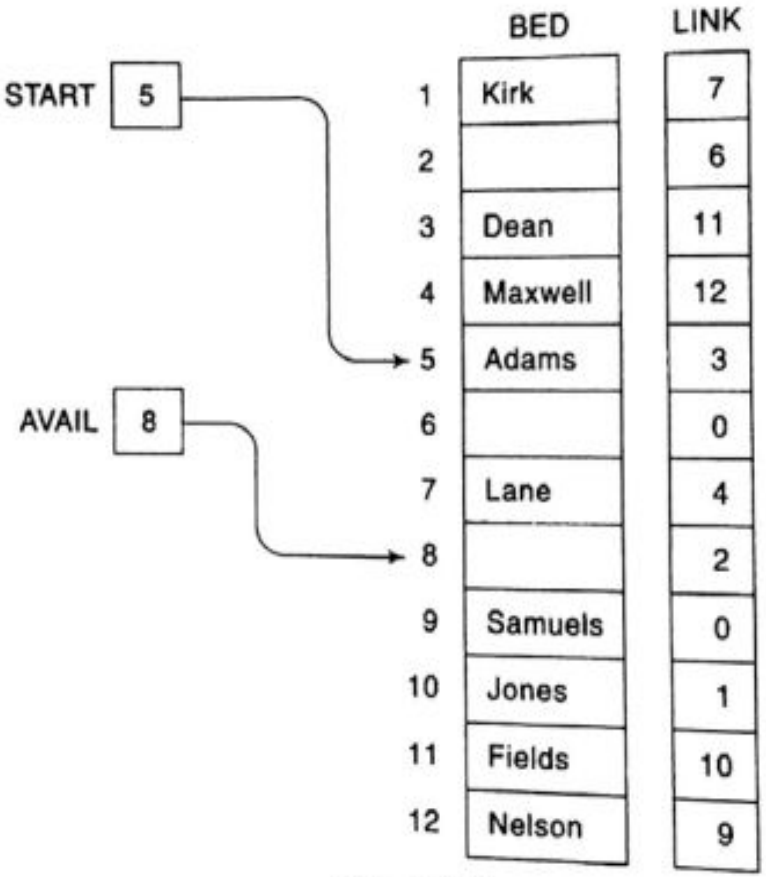



Fig. 5.28

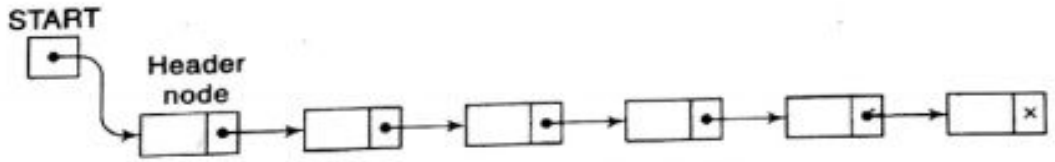
# Header Linked Lists

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:

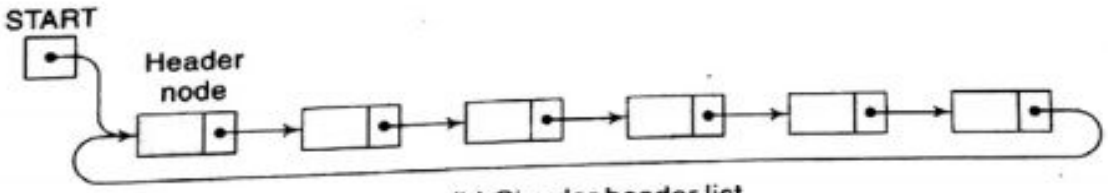
(1) A *grounded header list* is a header list where the last node contains the null pointer. (The term "grounded" comes from the fact that many texts use the electrical ground symbol to indicate the null pointer.)

(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.




(a) Grounded header list




(b) Circular header list

**Fig. 5.29**

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.

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

# Header Linked Lists

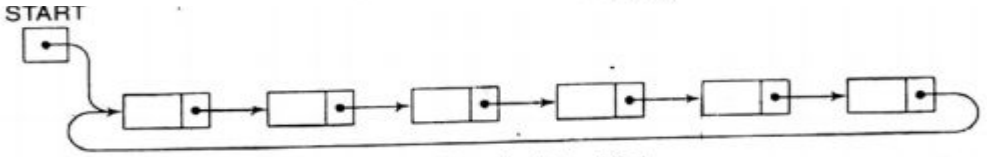
- Circular header list are frequently used instead of ordinary linked list because many operation are much easier to implement header list. This comes from the following two properties, all circular header lists.
  - The NULL pointer is not used & hence contains valid address.
  - Every ordinary node has a predecessor. So the first node may not required a special case.

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

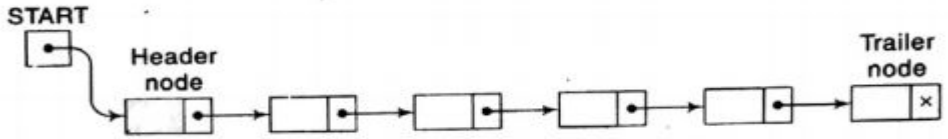
(1) A linked list whose last node points back to the first node instead of containing the null pointer, called a *circular list*

(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.

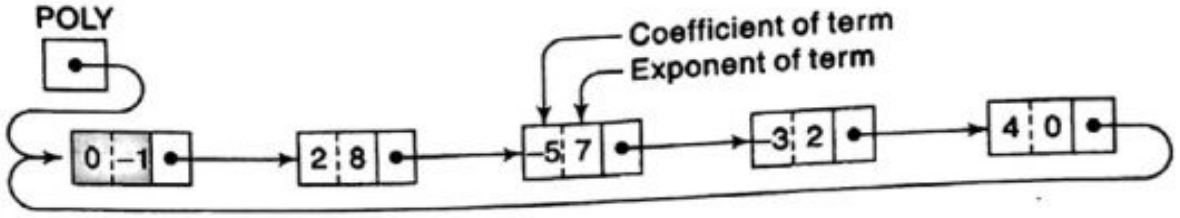
## Example 5.20

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)

# Two way Linked Lists

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.

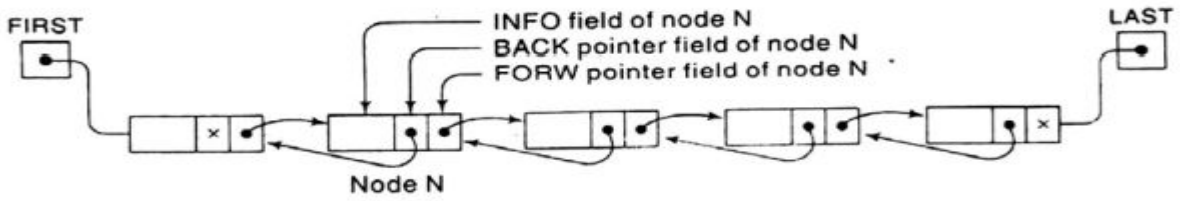



**Fig. 5.33** *Two-way List*

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 Linked Lists

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.

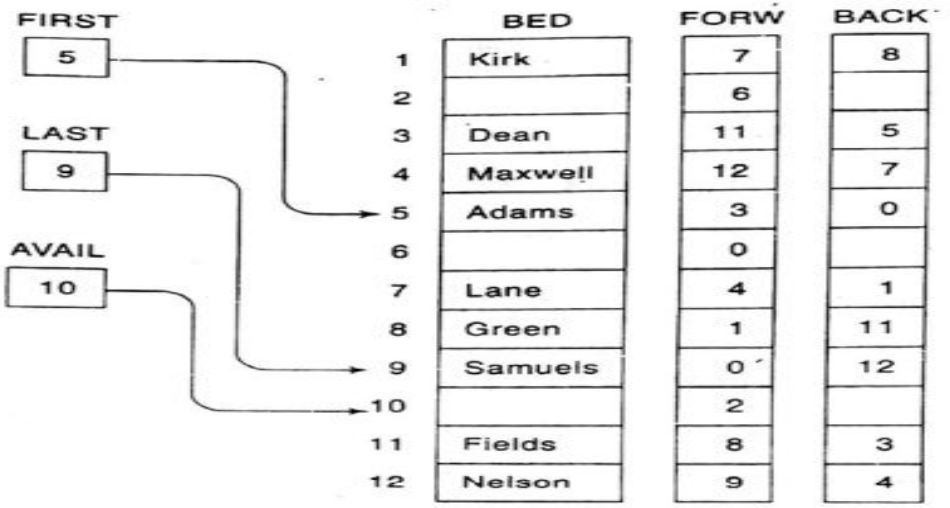



Fig. 5.34