# CSE 2103
# (Data Structures)

## Lecture on

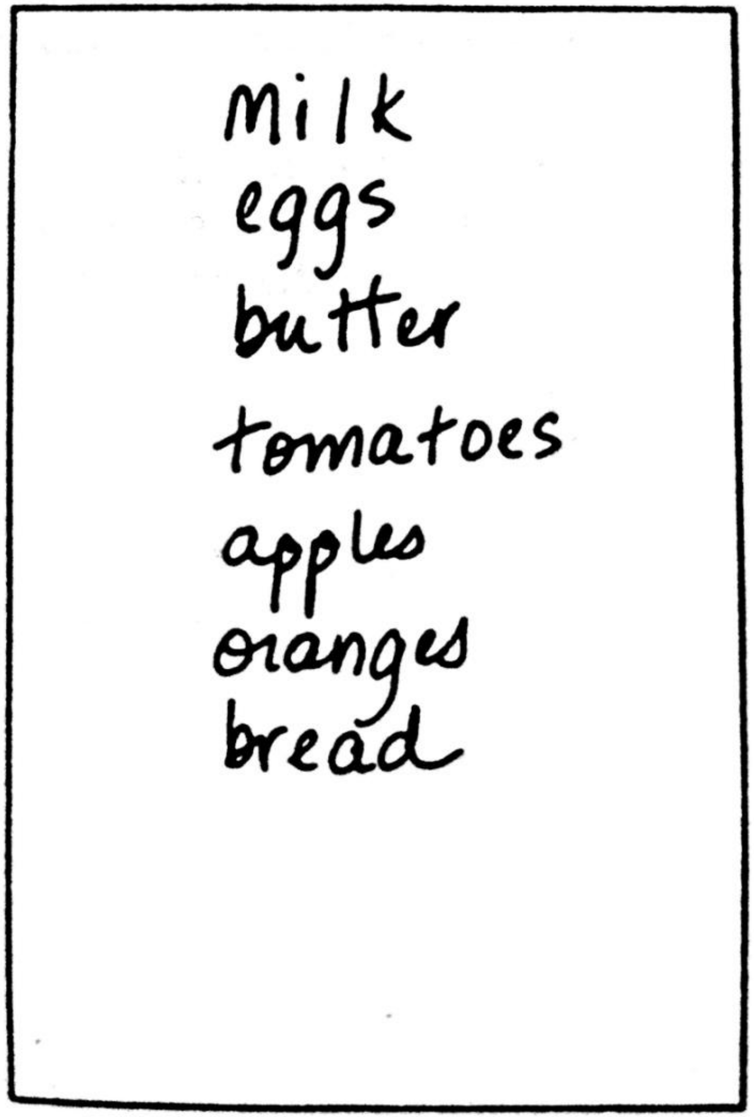
# Chapter-5: Linked LIst

By

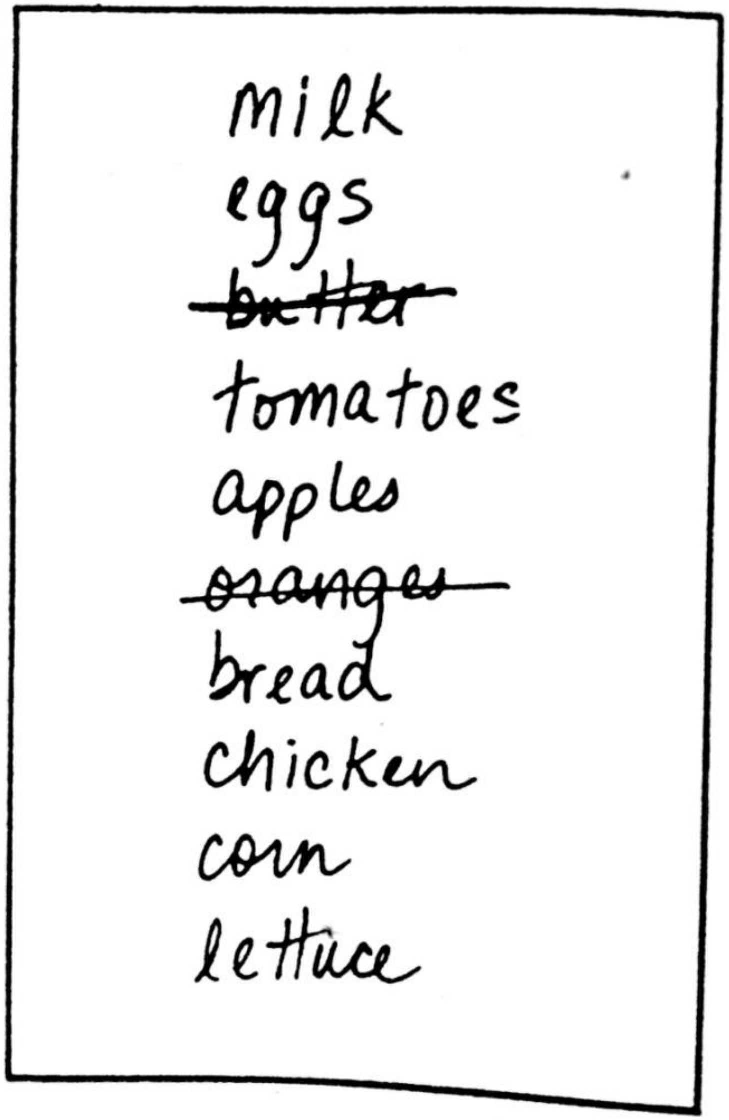
**Dr. M. Golam Rashed**
(golamrashed@ru.ac.bd)

**Department of Computer Science and Engineering(CSE)**
**Varendra University, Rajshahi, Bangladesh**

# LINKED LISTS: Introduction



milk
eggs
butter
tomatoes
apples
oranges
bread



milk
eggs
~~butter~~
tomatoes
apples
~~oranges~~
bread
chicken
corn
lettuce

# LINKED LISTS: Introduction

In linear array.......,

- ✓ 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.
- ✓ It is easy to compute the address of an element in an **Array**.

**Arrays have certain disadvantages:**

- ➢ It is relatively expensive to insert and delete elements in an array,
- ➢ One can not simply double or triple the size of an array when additional space is required.

For this reason,

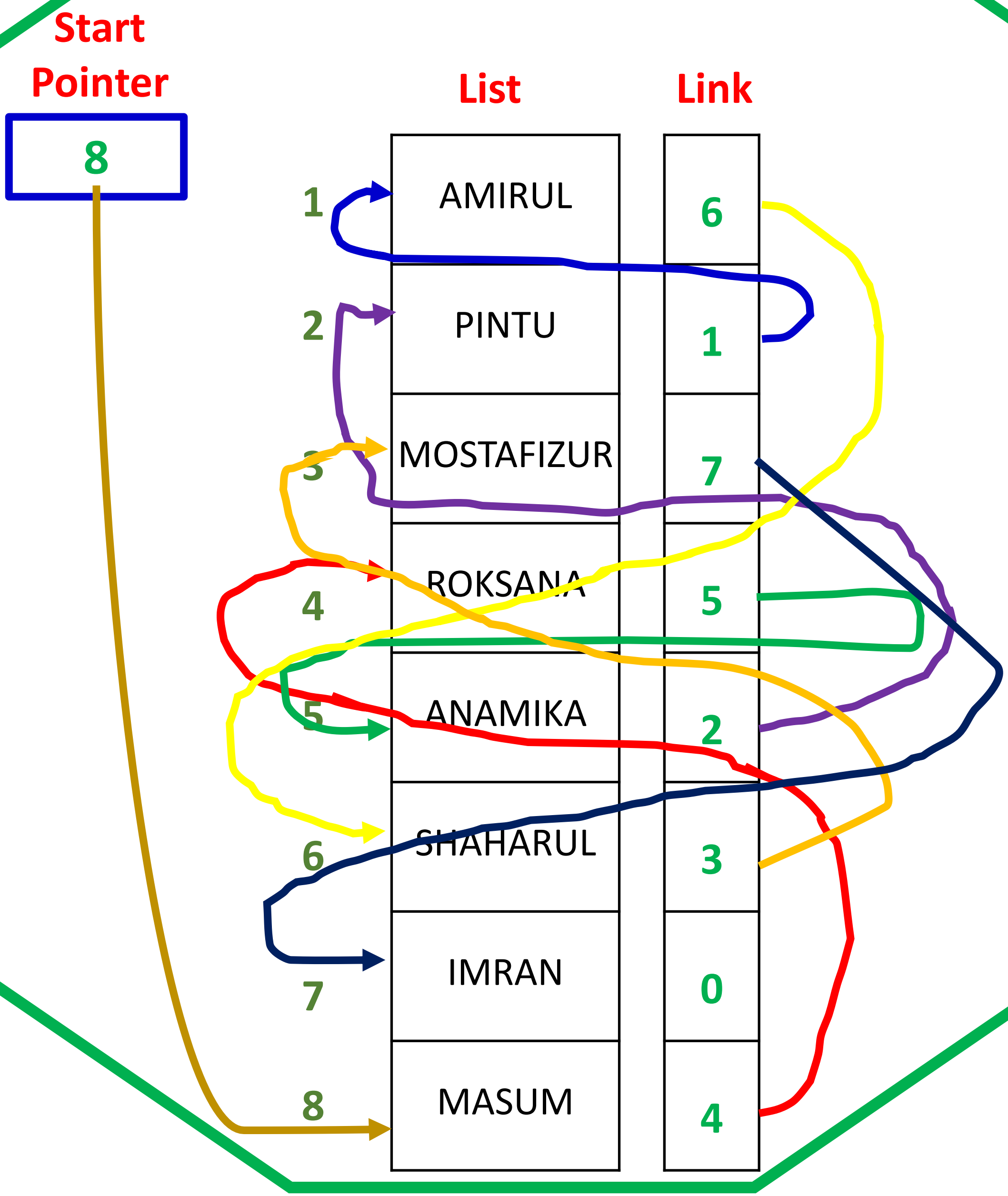
Arrays are

called *Dense lists*, and

said to be *Static* data structure.

# LINKED LISTS:

CSE 2103



**Start Pointer**

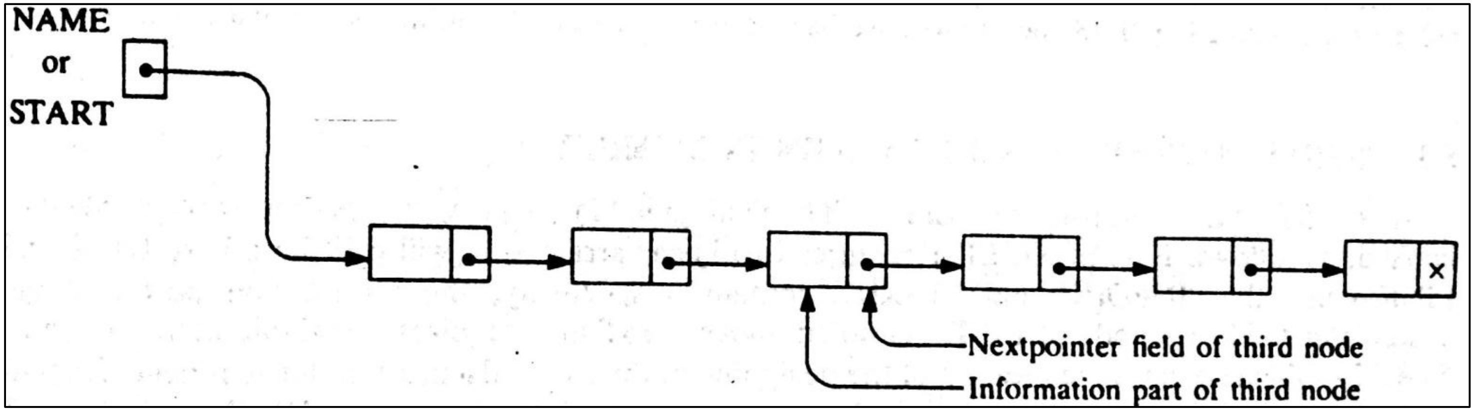
8

|  | List | Link |
|---|---|---|
| 1 | AMIRUL | 6 |
| 2 | PINTU | 1 |
| 3 | MOSTAFIZUR | 7 |
| 4 | ROKSANA | 5 |
| 5 | ANAMIKA | 2 |
| 6 | SHAHARUL | 3 |
| 7 | IMRAN | 0 |
| 8 | MASUM | 4 |

**LINKED LIST**

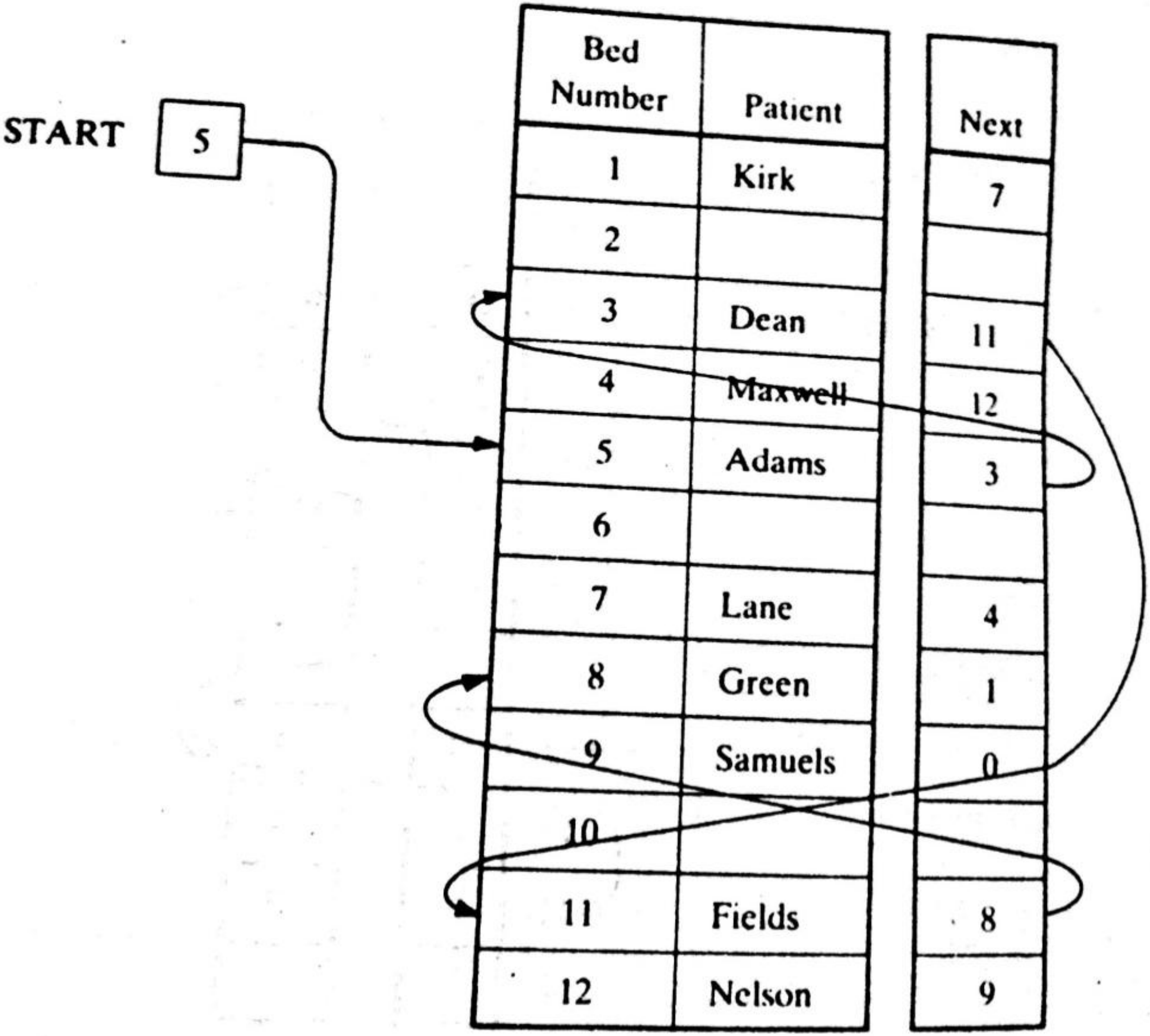
# LINKED LISTS: Definition

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*.


NAME
or
START
×
Nextpointer field of third node
Information part of third node

- Each node is divided into two parts:
    - The first part contains the information of the element,
    - The second part, called the *link field* or *nextpointer field*, contains the address of the next node in the list.
- There is an arrow drawn from a node to the next node in the list.
- The pointer of the last node contains a special value, called the *null* pointer, which is any invalid address (0 or a negative number), denoted by × in the diagram.
- The linked list also contain a *list pointer variable*-called START or NAME- which contain the address of the first node in the list.

# LINKED LISTS: Example



START 5

| Bed Number | Patient | Next |
|---|---|---|
| 1 | Kirk | 7 |
| 2 | | |
| 3 | Dean | 11 |
| 4 | Maxwell | 12 |
| 5 | Adams | 3 |
| 6 | | |
| 7 | Lane | 4 |
| 8 | Green | 1 |
| 9 | Samuels | 0 |
| 10 | | |
| 11 | Fields | 8 |
| 12 | Nelson | 9 |

# LINKED LISTS: Representation in Memory

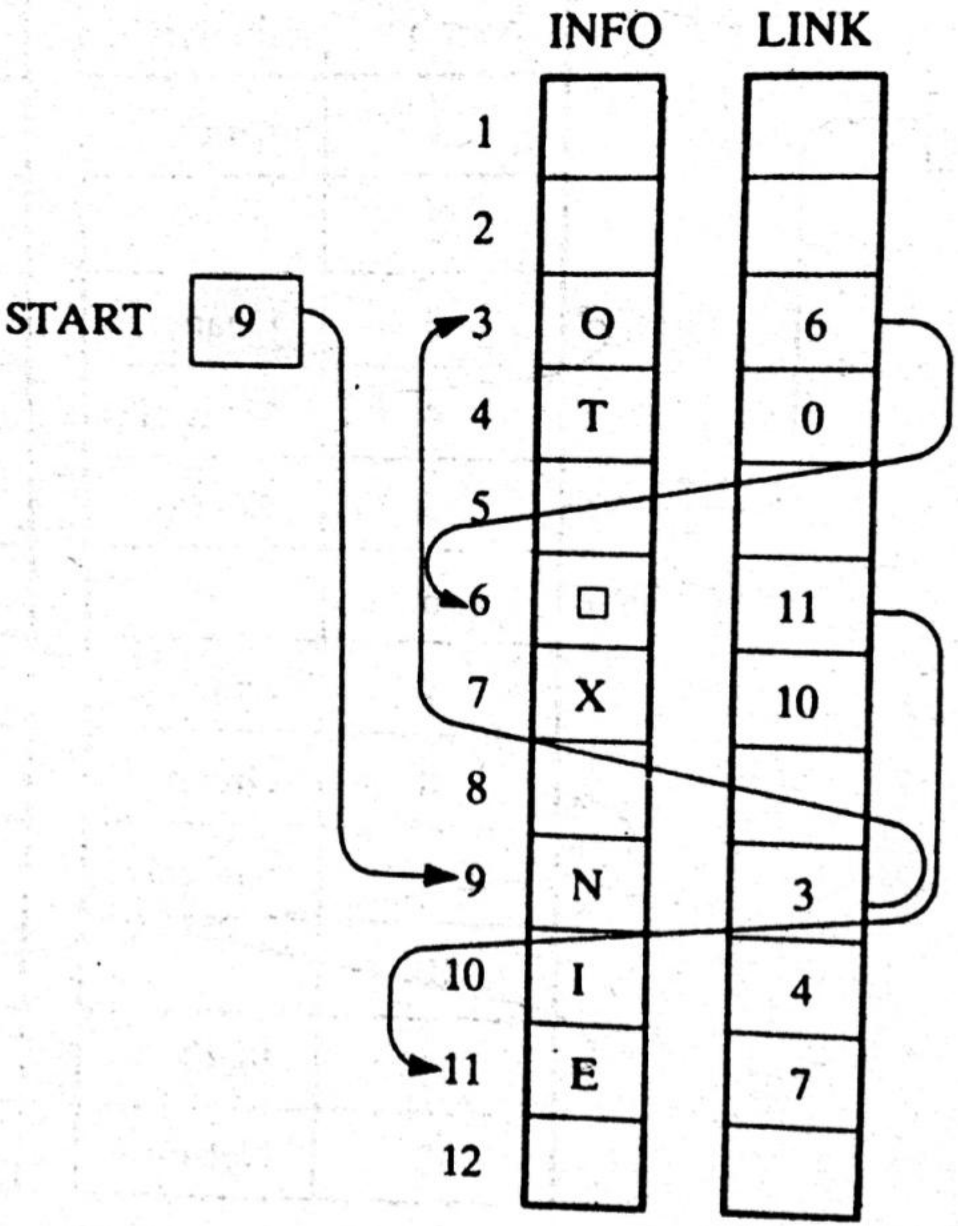
Let LIST be a linked list which will be maintained in the memory

LIST requires…..

- Two linear array,
  - INFO[K]-Information part, and
  - LINK[K]-nextpointer field of a node of LIST
- A variable name-START, indicating the beginning of the LIST.
- A nextpointer senitel-NULL, indicates the end of the LIST

❖ Since, the subscripts of the array INFO and LINK will usually be positive, NULL=0, unless otherwise stated

# LINKED LISTS: Example



START=9, so INFO[9]= ? N

LINK[9]=3, so, INFO[3]=? O

LINK[3]=6, so, INFO[6]=? ☐

LINK[6]=11, so, INFO[11]=? E

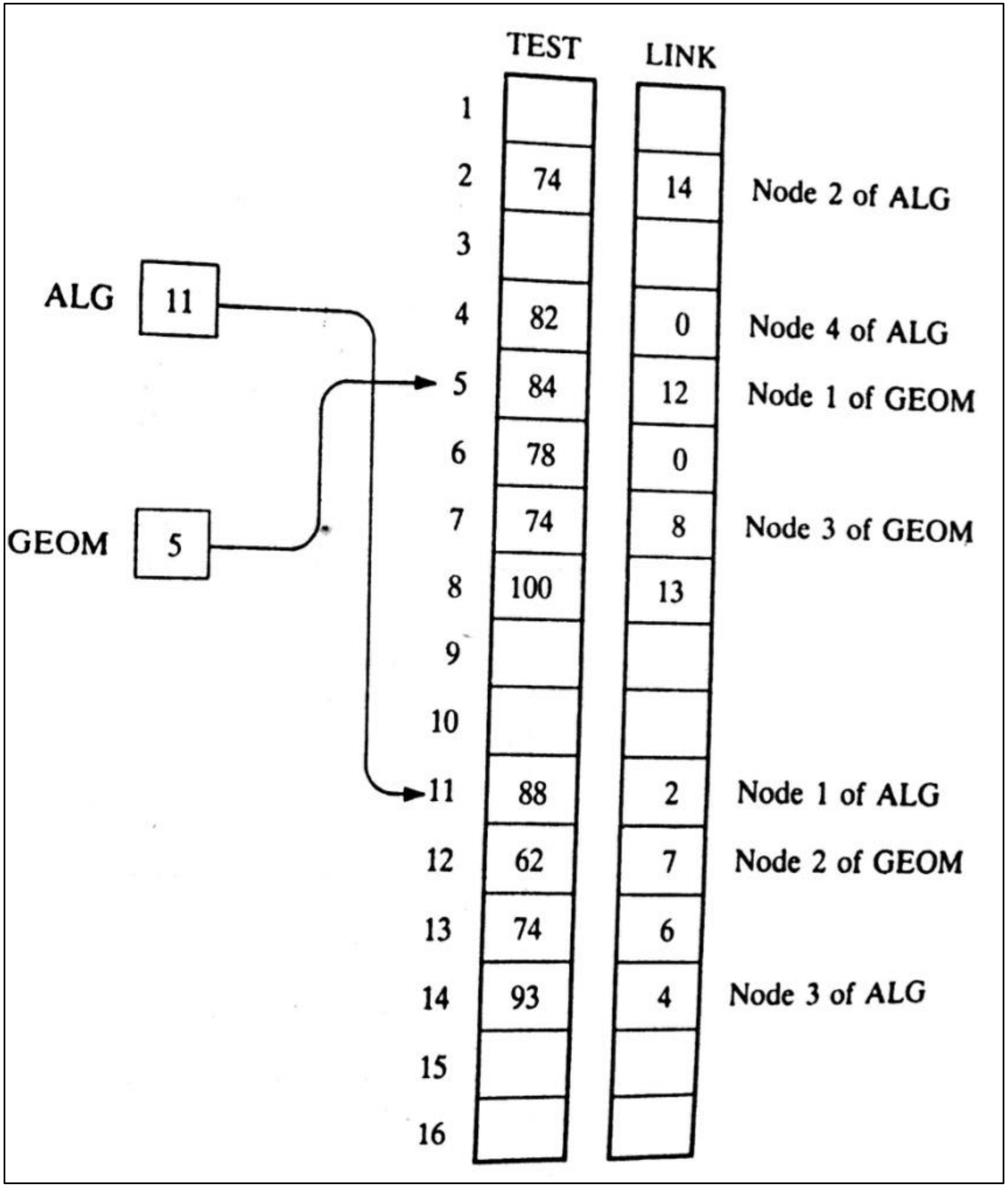
LINK[11]=7, so, INFO[7]= ? X

LINK[7]=10, so, INFO[10]= ? I

LINK[10]=4, so, INFO[4]= ? T

LINK[4]=0, so, INFO[0]= ? NULL

# LINKED LISTS: **Example**



| | TEST | LINK | |
|---|---|---|---|
| 1 | | | |
| 2 | 74 | 14 | Node 2 of ALG |
| 3 | | | |
| 4 | 82 | 0 | Node 4 of ALG |
| 5 | 84 | 12 | Node 1 of GEOM |
| 6 | 78 | 0 | |
| 7 | 74 | 8 | Node 3 of GEOM |
| 8 | 100 | 13 | |
| 9 | | | |
| 10 | | | |
| 11 | 88 | 2 | Node 1 of ALG |
| 12 | 62 | 7 | Node 2 of GEOM |
| 13 | 74 | 6 | |
| 14 | 93 | 4 | Node 3 of ALG |
| 15 | | | |
| 16 | | | |

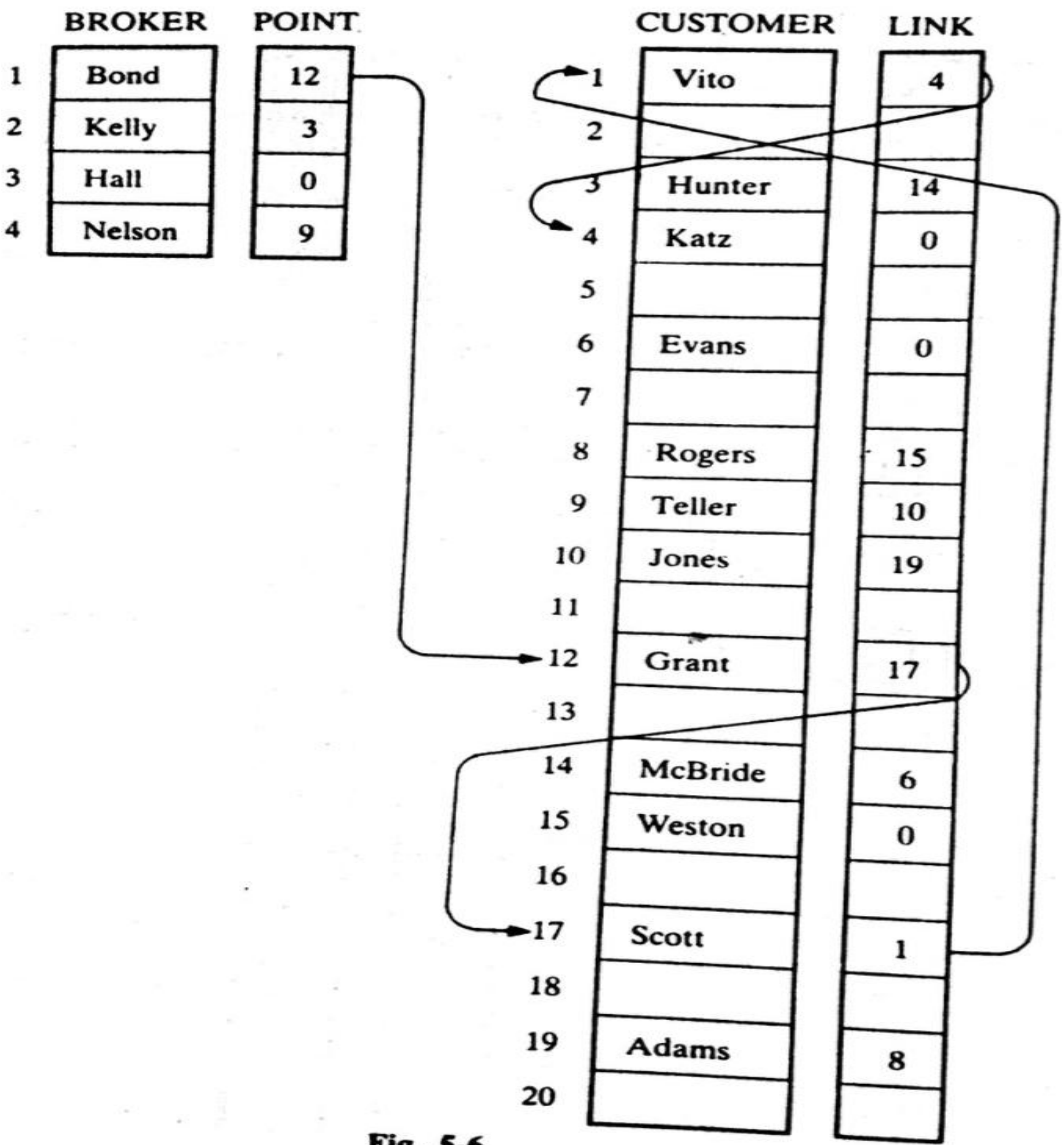
ALG consists of Test score:

88, 74, 93, 82

GEOM consists of Test score:

84, 62, 74, 100,74, 78

# LINKED LISTS: Example



|  | BROKER | POINT |
|---|---|---|
| 1 | Bond | 12 |
| 2 | Kelly | 3 |
| 3 | Hall | 0 |
| 4 | Nelson | 9 |

|  | CUSTOMER | LINK |
|---|---|---|
| 1 | Vito | 4 |
| 2 |  |  |
| 3 | Hunter | 14 |
| 4 | Katz | 0 |
| 5 |  |  |
| 6 | Evans | 0 |
| 7 |  |  |
| 8 | Rogers | 15 |
| 9 | Teller | 10 |
| 10 | Jones | 19 |
| 11 |  |  |
| 12 | Grant | 17 |
| 13 |  |  |
| 14 | McBride | 6 |
| 15 | Weston | 0 |
| 16 |  |  |
| 17 | Scott | 1 |
| 18 |  |  |
| 19 | Adams | 8 |
| 20 |  |  |

Fig. 5-6

Bond's list of customers:
Grant, Scott, Vito, Katz

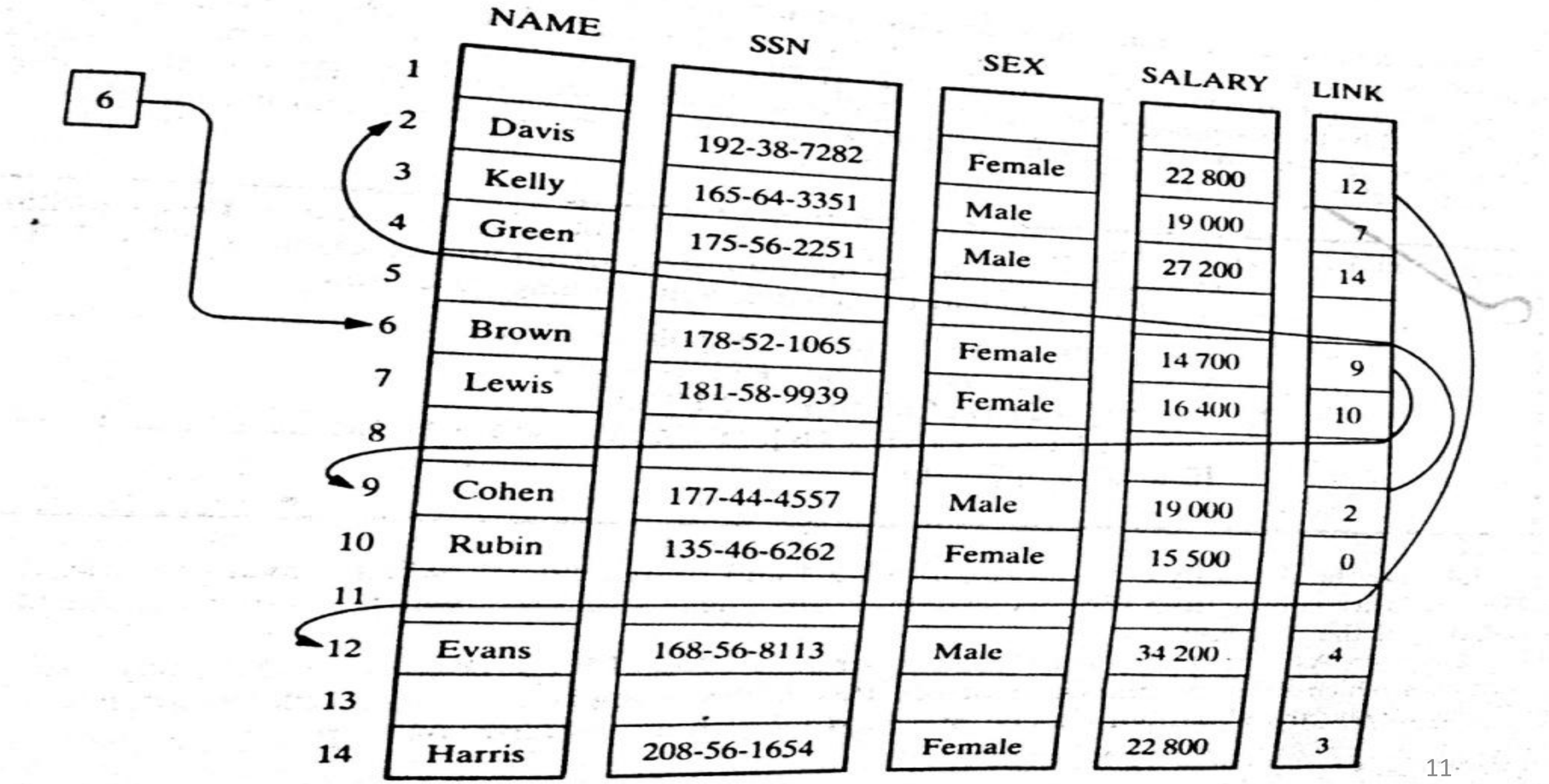
Kelly's list of customers:
Hunter, McBride, Evans

Hall's list of customers:
NULL / Empty List

Nelson's list of customers:
Teller, Jones, Adams, Rogers, Weston

# LINKED LISTS: **Example**

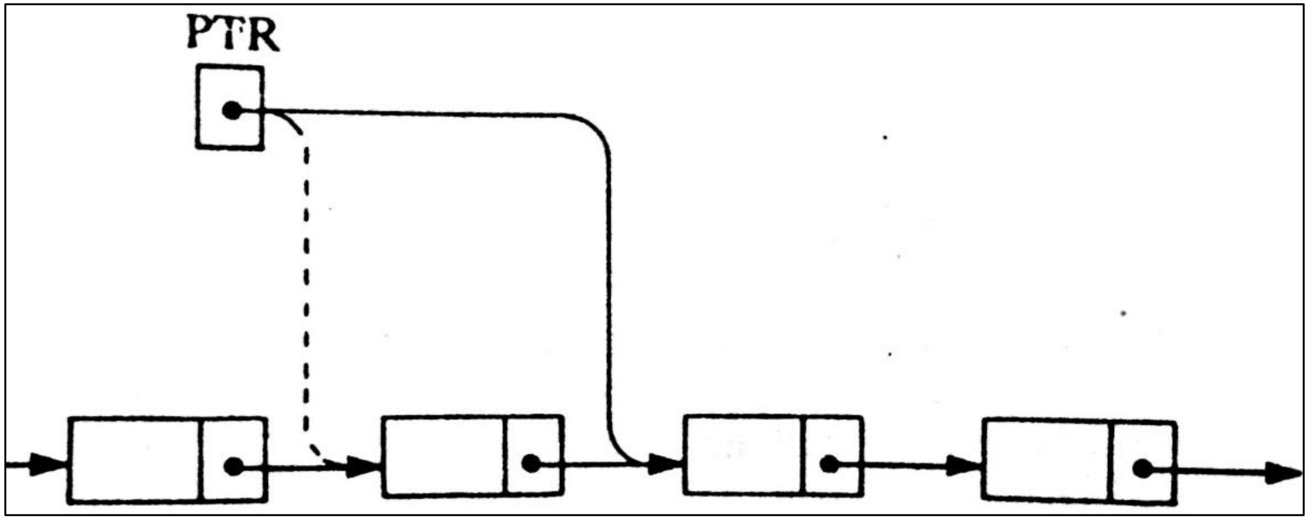
- In general, the information part of a node may be a record with more than one data item.
- In such a case, the data must be stored in some type of record structure or in a collection of parallel arrays.



| | NAME | SSN | SEX | SALARY | LINK |
|---|---|---|---|---|---|
| 1 | | | | | |
| 2 | Davis | 192-38-7282 | Female | 22 800 | 12 |
| 3 | Kelly | 165-64-3351 | Male | 19 000 | 7 |
| 4 | Green | 175-56-2251 | Male | 27 200 | 14 |
| 5 | | | | | |
| 6 | Brown | 178-52-1065 | Female | 14 700 | 9 |
| 7 | Lewis | 181-58-9939 | Female | 16 400 | 10 |
| 8 | | | | | |
| 9 | Cohen | 177-44-4557 | Male | 19 000 | 2 |
| 10 | Rubin | 135-46-6262 | Female | 15 500 | 0 |
| 11 | | | | | |
| 12 | Evans | 168-56-8113 | Male | 34 200 | 4 |
| 13 | | | | | |
| 14 | Harris | 208-56-1654 | Female | 22 800 | 3 |

# LINKED LISTS: **Traversing**

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**.



- ✓ 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,

  **PTR:=LINK[PTR],** moves the pointer to the next node in the list.

# LINKED LISTS: Traversing Algorithm

Let LIST be a linked list in memory.

```
LINKED_LIST_Traversing(INFO, LINK, START)
Step1.   Set PTR:=START.[Initialize pointer PTR.]
Step2.   Repeat Steps 3 and 4 while PTR ≠ NULL
Step3.           Apply PROCESS to INFO[PTR]
Step4.           Set PTR:=LINK[PTR]. [PTR now point to the next node.]
         [End of step 2 loop]
Step5.   Exit.
```

**Any difference with Linear array traversing ??**

```
LINEAR_ARRAY_Traversing
Step 1. [Initialize counter] Set K:=LB
Step 2. Repeat Step 3 and 4 while K<=UB [Repeat while loop]
Step 3.         [Visit element] Apply PROCESS to A[K].
Step 4.                 [Increase counter] Set K:=K+1.
        [End of Step 2 loop.]
Step 5. Exit.
```

# LINKED LISTS: Practice…

Teach yourself: Example 5.7 (Find the number of elements in a linked list)

# LINKED LISTS: **Searching**

Given:

- LIST- a linked list in memory (Unknown/Unseen)
- ITEM- a specific information.

Objective: Finding the location LOC of the node where ITEM first appears in LIST.

Here, TWO searching algorithm will be discussed for finding the location of LOC of the node where ITEM first appears in LIST.

- Algorithm-1 for LIST is Unsorted
- Algorithm-2 for LIST is Sorted.

# LINKED LISTS: Searching Algorithm-1

## SEARCH (INFO, LINK, START, ITEM, LOC) [when list is unsorted]

```
Step1.  Set PTR:=START
Step2.  Repeat Step 3 while PTR $\neq$ NULL
Step3.          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.]
Step4.  [Search is unsuccessful.] Set LOC:=NULL.
Step5.  Exit.
```

# LINKED LISTS: Searching Algorithm-2

## SEARCH (INFO, LINK, START, ITEM, LOC) [when list is sorted]

```
Step1.  Set PTR:=START
Step2.  Repeat Step 3 while PTR $\neq$ NULL
Step3.      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.]
Step4.  Set LOC:=NULL.
Step5.  Exit.
```

Descending Order Sorted LIST

Any change in algorithm for Ascending Order Sorted LIST?

# LINKED LISTS: Searching Algorithm

Limitations:

NB. 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.**

The main drawbacks in using a linked list as data structure

# LINKED LISTS: Memory Allocation

✓ The maintenance of linked lists in memory assumes the possibility of ......

  - Inserting new nodes into the lists, and
  - Deleting nodes from the list.

✓ Hence, requires some mechanism which provides:

  - Unused memory space for the new nodes.
  - Deleted nodes becomes available for future use.

**To meet the necessities........**

# LINKED LISTS: **Memory Allocation**

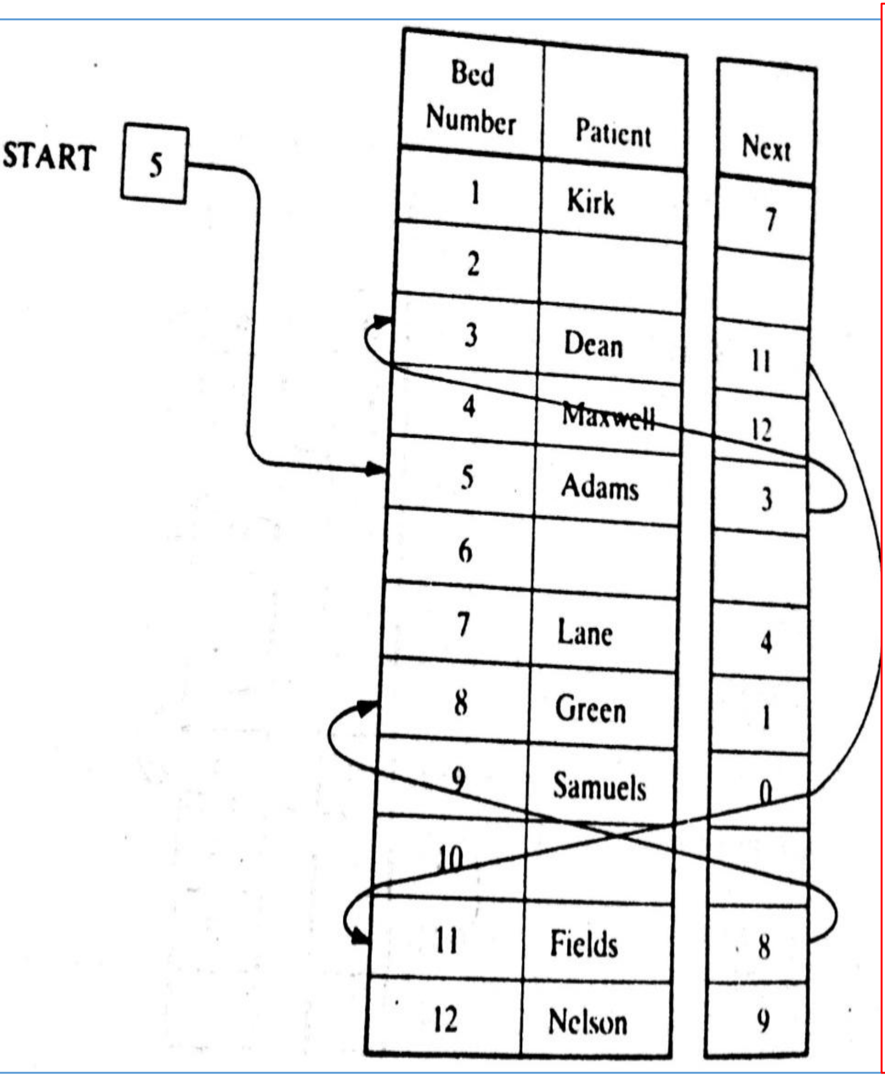
- ✓Together with the linked lists in memory, a special list is maintained which consists of unused memory cells.
- ✓This list, which has its own pointer, is called …
  - ✓***The list of available space*** or
  - ✓*the free-storage list* or
  - ✓*the free pool.*

# LINKED LISTS: Memory Allocation

- ✓Here, linked lists are implemented by parallel arrays.
- ✓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.

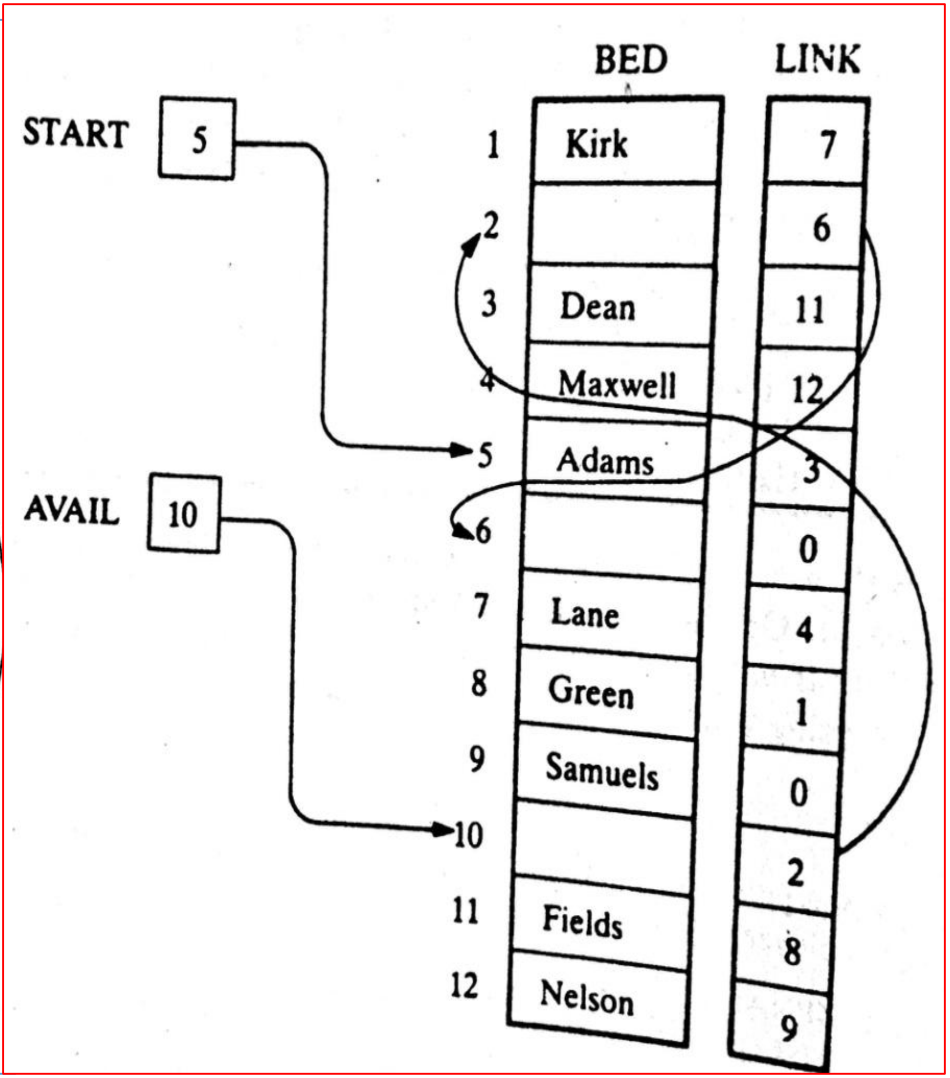
**LIST(INFO, LINK, START, AVAIL)**

# LINKED LISTS: Example 5.10



START 5

| Bed Number | Patient | Next |
|---|---|---|
| 1 | Kirk | 7 |
| 2 | | |
| 3 | Dean | 11 |
| 4 | Maxwell | 12 |
| 5 | Adams | 3 |
| 6 | | |
| 7 | Lane | 4 |
| 8 | Green | 1 |
| 9 | Samuels | 0 |
| 10 | | |
| 11 | Fields | 8 |
| 12 | Nelson | 9 |

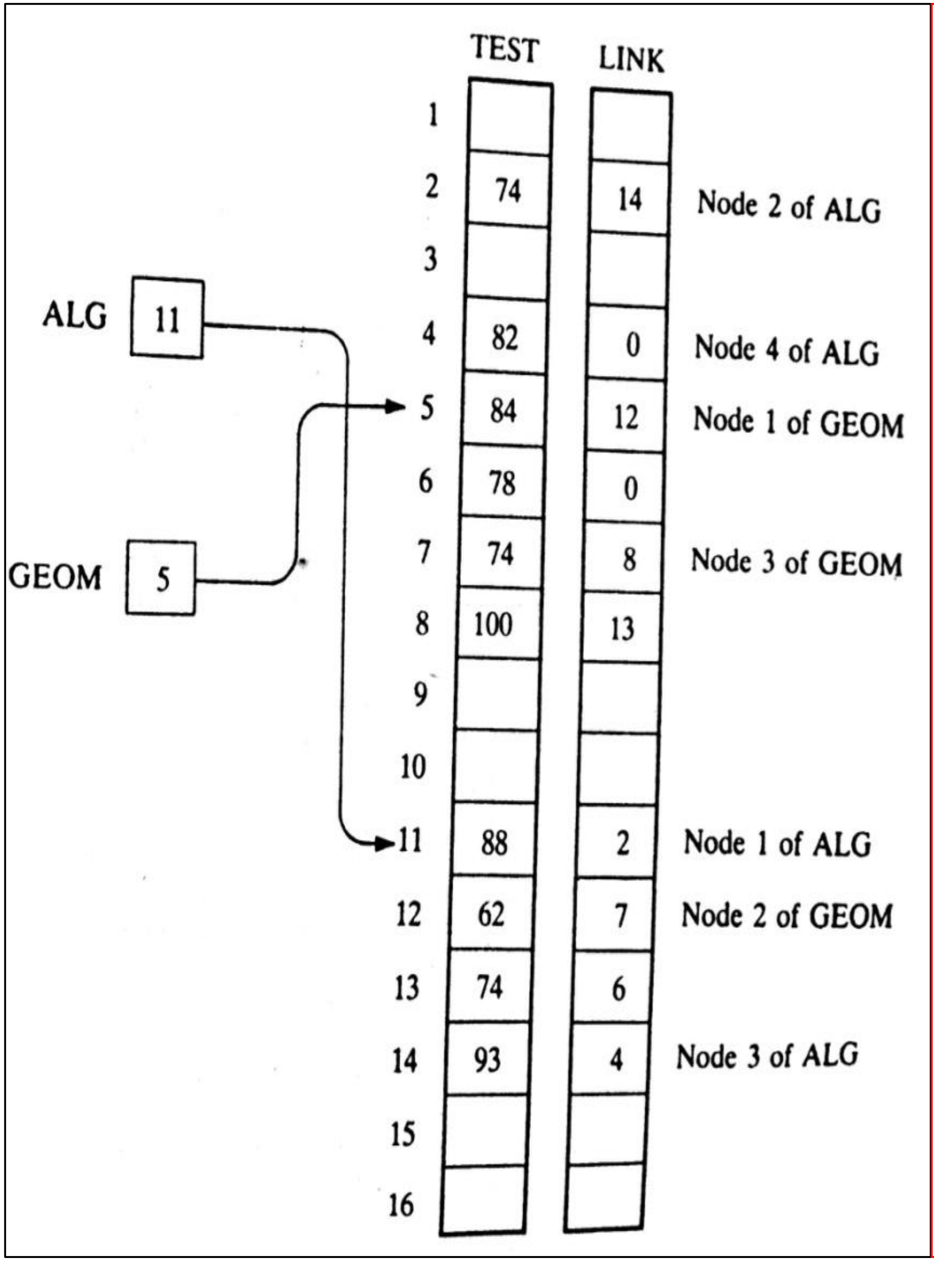


START 5

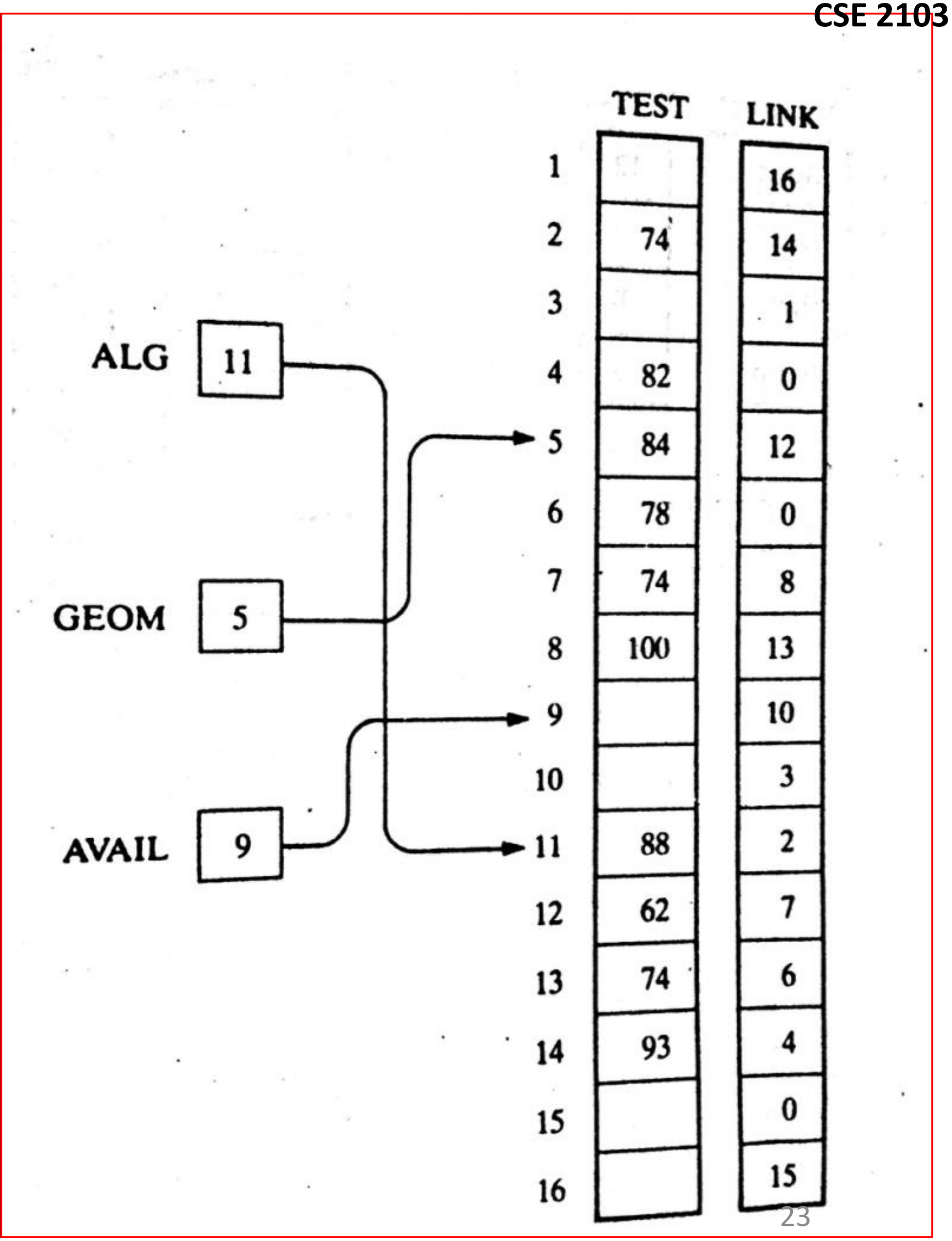
AVAIL 10

| | BED | LINK |
|---|---|---|
| 1 | Kirk | 7 |
| 2 | | 6 |
| 3 | Dean | 11 |
| 4 | Maxwell | 12 |
| 5 | Adams | 3 |
| 6 | | 0 |
| 7 | Lane | 4 |
| 8 | Green | 1 |
| 9 | Samuels | 0 |
| 10 | | 2 |
| 11 | Fields | 8 |
| 12 | Nelson | 9 |

# LINKED LISTS: Example 5.11(a)

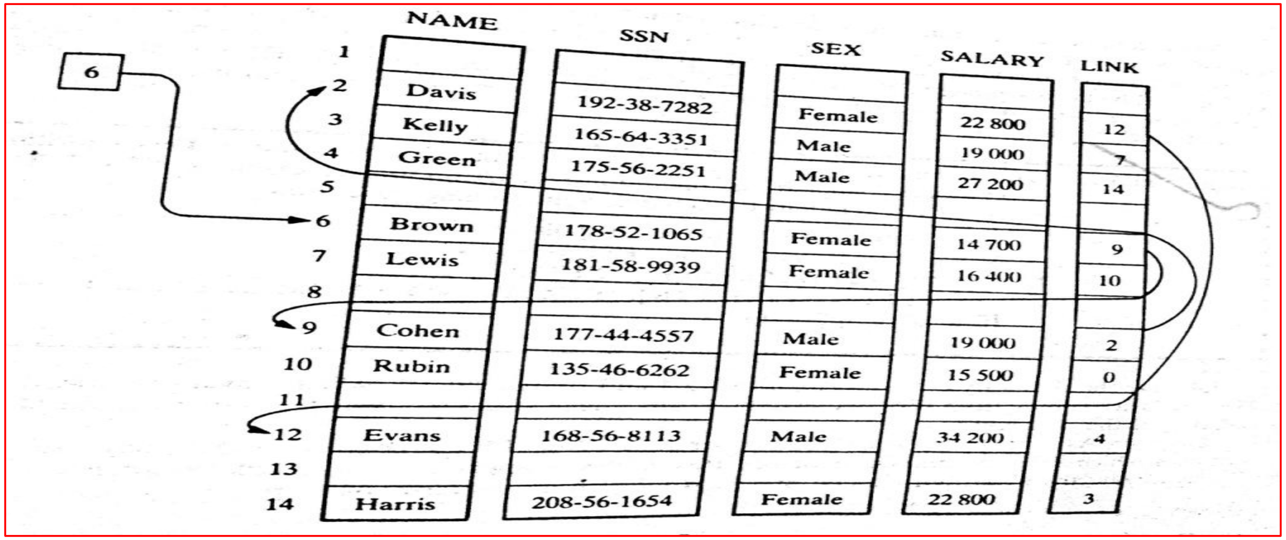


| | TEST | LINK | |
|---|---|---|---|
| 1 | | | |
| 2 | 74 | 14 | Node 2 of ALG |
| 3 | | | |
| 4 | 82 | 0 | Node 4 of ALG |
| 5 | 84 | 12 | Node 1 of GEOM |
| 6 | 78 | 0 | |
| 7 | 74 | 8 | Node 3 of GEOM |
| 8 | 100 | 13 | |
| 9 | | | |
| 10 | | | |
| 11 | 88 | 2 | Node 1 of ALG |
| 12 | 62 | 7 | Node 2 of GEOM |
| 13 | 74 | 6 | |
| 14 | 93 | 4 | Node 3 of ALG |
| 15 | | | |
| 16 | | | |

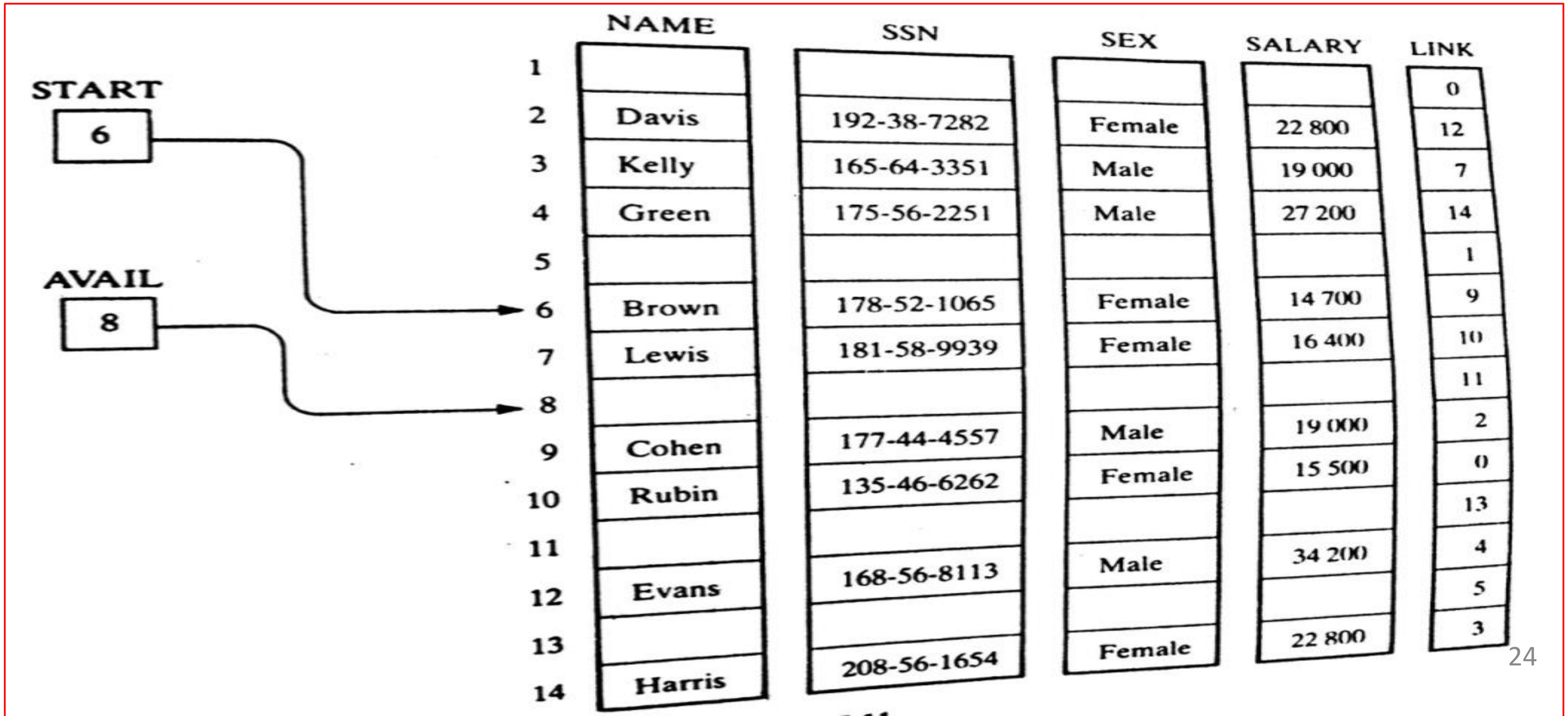


| | TEST | LINK |
|---|---|---|
| 1 | | 16 |
| 2 | 74 | 14 |
| 3 | | 1 |
| 4 | 82 | 0 |
| 5 | 84 | 12 |
| 6 | 78 | 0 |
| 7 | 74 | 8 |
| 8 | 100 | 13 |
| 9 | | 10 |
| 10 | | 3 |
| 11 | 88 | 2 |
| 12 | 62 | 7 |
| 13 | 74 | 6 |
| 14 | 93 | 4 |
| 15 | | 0 |
| 16 | | 15 |

# LINKED LISTS: Example 5.11(b)

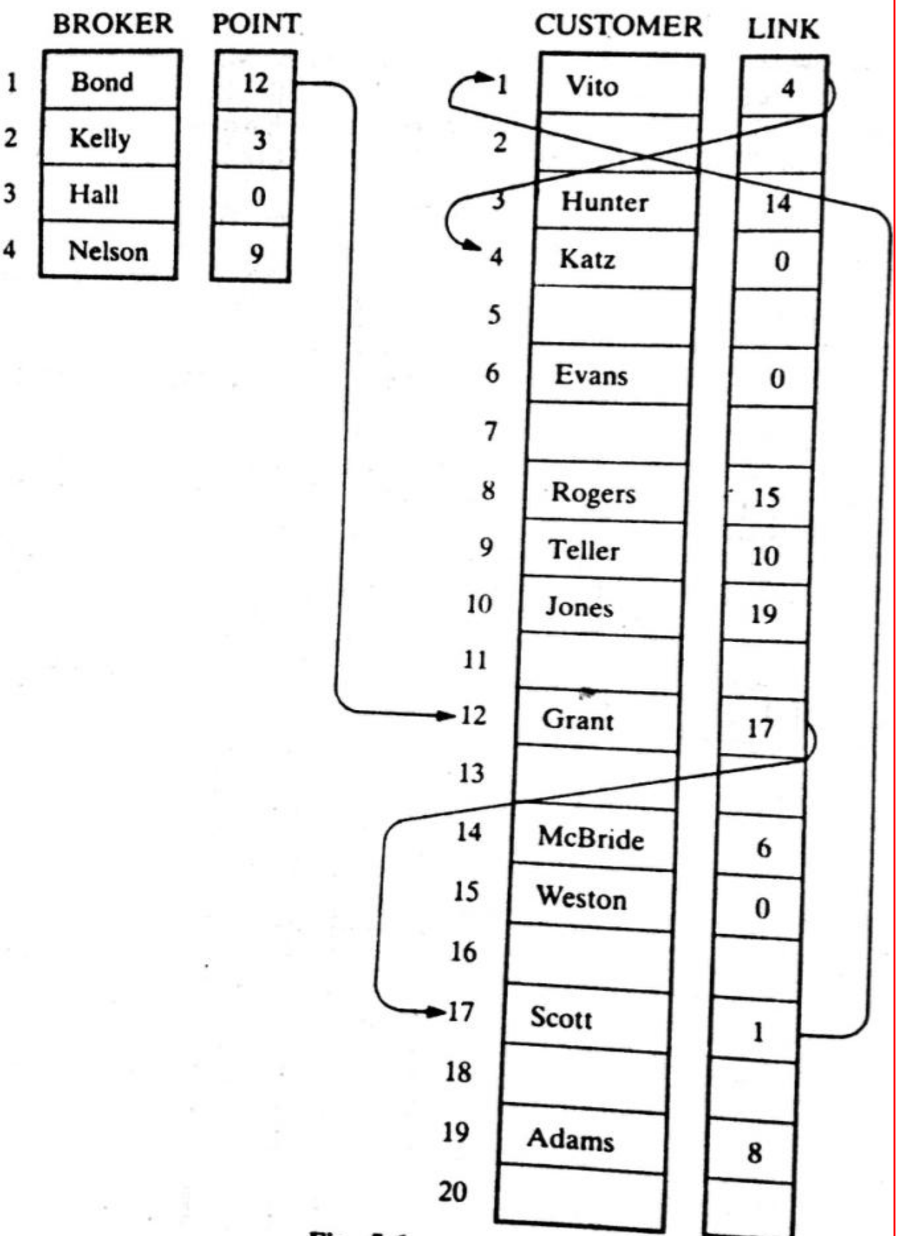


| | NAME | SSN | SEX | SALARY | LINK |
|---|---|---|---|---|---|
| 1 | | | | | |
| 2 | Davis | 192-38-7282 | Female | 22 800 | 12 |
| 3 | Kelly | 165-64-3351 | Male | 19 000 | 7 |
| 4 | Green | 175-56-2251 | Male | 27 200 | 14 |
| 5 | | | | | |
| 6 | Brown | 178-52-1065 | Female | 14 700 | 9 |
| 7 | Lewis | 181-58-9939 | Female | 16 400 | 10 |
| 8 | | | | | |
| 9 | Cohen | 177-44-4557 | Male | 19 000 | 2 |
| 10 | Rubin | 135-46-6262 | Female | 15 500 | 0 |
| 11 | | | | | |
| 12 | Evans | 168-56-8113 | Male | 34 200 | 4 |
| 13 | | | | | |
| 14 | Harris | 208-56-1654 | Female | 22 800 | 3 |



| | NAME | SSN | SEX | SALARY | LINK |
|---|---|---|---|---|---|
| 1 | | | | | 0 |
| 2 | Davis | 192-38-7282 | Female | 22 800 | 12 |
| 3 | Kelly | 165-64-3351 | Male | 19 000 | 7 |
| 4 | Green | 175-56-2251 | Male | 27 200 | 14 |
| 5 | | | | | 1 |
| 6 | Brown | 178-52-1065 | Female | 14 700 | 9 |
| 7 | Lewis | 181-58-9939 | Female | 16 400 | 10 |
| 8 | | | | | 11 |
| 9 | Cohen | 177-44-4557 | Male | 19 000 | 2 |
| 10 | Rubin | 135-46-6262 | Female | 15 500 | 0 |
| 11 | | | | | 13 |
| 12 | Evans | 168-56-8113 | Male | 34 200 | 4 |
| 13 | | | | | 5 |
| 14 | Harris | 208-56-1654 | Female | 22 800 | 3 |

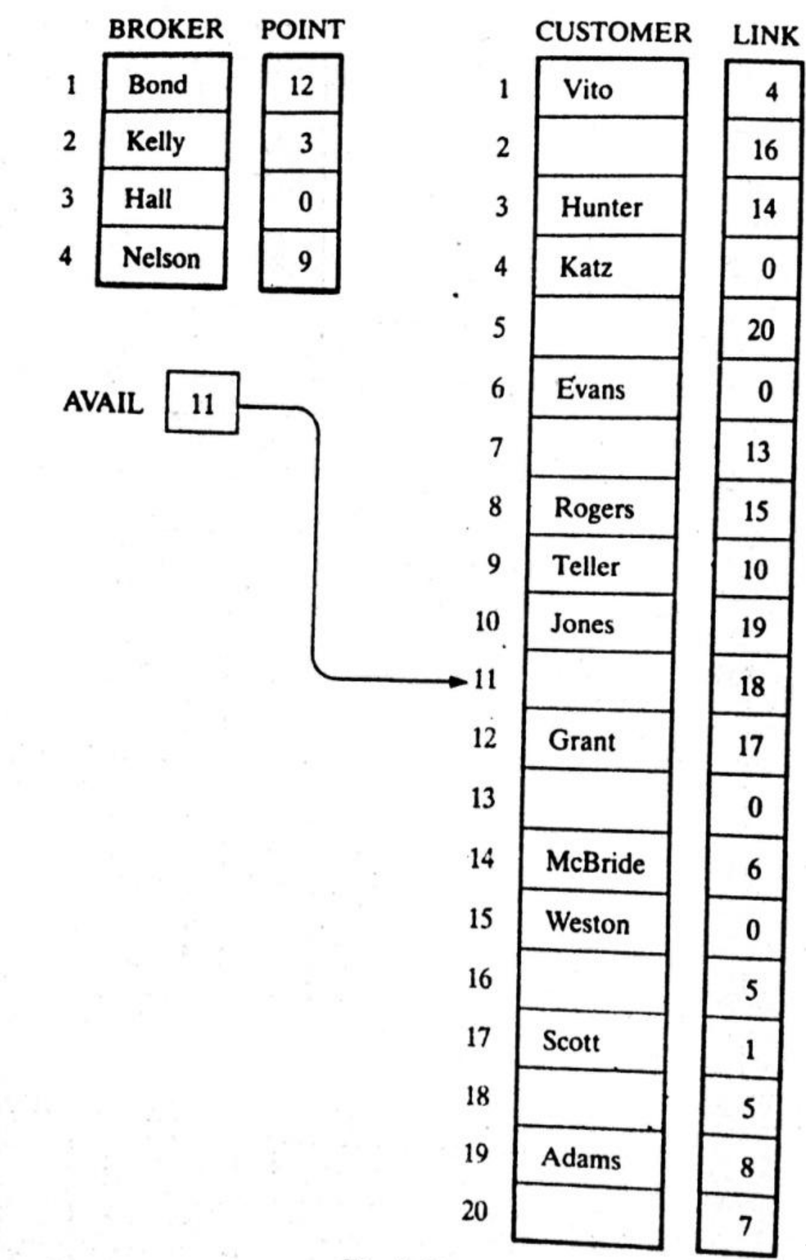
# LINKED LISTS: Example 5.11(c)



| | BROKER | POINT |
|---|---|---|
| 1 | Bond | 12 |
| 2 | Kelly | 3 |
| 3 | Hall | 0 |
| 4 | Nelson | 9 |

| | CUSTOMER | LINK |
|---|---|---|
| 1 | Vito | 4 |
| 2 | | |
| 3 | Hunter | 14 |
| 4 | Katz | 0 |
| 5 | | |
| 6 | Evans | 0 |
| 7 | | |
| 8 | Rogers | 15 |
| 9 | Teller | 10 |
| 10 | Jones | 19 |
| 11 | | |
| 12 | Grant | 17 |
| 13 | | |
| 14 | McBride | 6 |
| 15 | Weston | 0 |
| 16 | | |
| 17 | Scott | 1 |
| 18 | | |
| 19 | Adams | 8 |
| 20 | | |

Fig. 5-6



| | BROKER | POINT |
|---|---|---|
| 1 | Bond | 12 |
| 2 | Kelly | 3 |
| 3 | Hall | 0 |
| 4 | Nelson | 9 |

AVAIL 11

| | CUSTOMER | LINK |
|---|---|---|
| 1 | Vito | 4 |
| 2 | | 16 |
| 3 | Hunter | 14 |
| 4 | Katz | 0 |
| 5 | | 20 |
| 6 | Evans | 0 |
| 7 | | 13 |
| 8 | Rogers | 15 |
| 9 | Teller | 10 |
| 10 | Jones | 19 |
| 11 | | 18 |
| 12 | Grant | 17 |
| 13 | | 0 |
| 14 | McBride | 6 |
| 15 | Weston | 0 |
| 16 | | 5 |
| 17 | Scott | 1 |
| 18 | | 5 |
| 19 | Adams | 8 |
| 20 | | 7 |

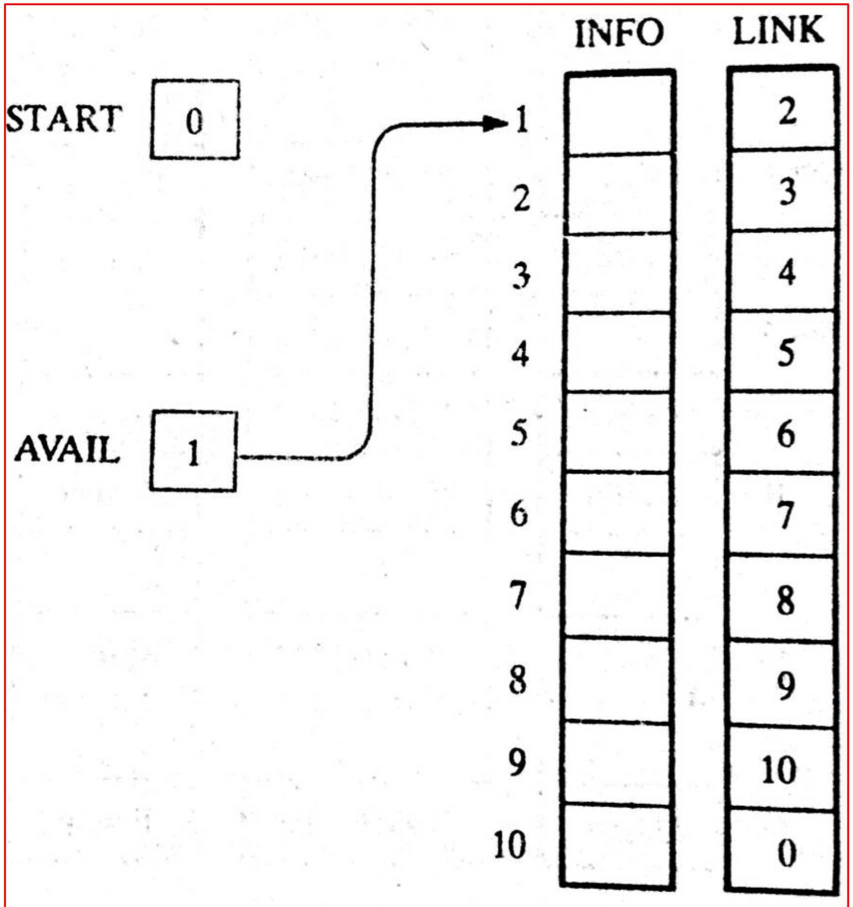
Fig. 5-12

# LINKED LISTS: **Example 5.12**

- Suppose LIST(INFO, LINK, START, AVAIL) has memory space for n=5 nodes,
- Furthermore, suppose LIST is initially empty.

## Your INSTATNT Task

**Show the LINKED LIST which consists of START, AVAIL, INFO, LINK**



# LINKED LISTS: Garbage Collection

## Time Consuming Method for OS:

- 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 the 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 OS of a computer.**

# LINKED LISTS: Garbage Collection

**Time Efficient Method for OS:**

The OS of a computer may periodically collect all the deleted space onto the free-space list. Any technique which does this collection is called *garbage collection*.

*Garbage Collection* usually takes place in **TWO** steps*:*

- ✓ Firstly, the Computer runs through the memory, tagging those cells which are currently in use, and
- ✓ Then, computer runs through the memory, collecting all untagged space onto the free-storage list.

# LINKED LISTS: Garbage Collection

## When garbage collection task executed by Computer?

The garbage collection may take place when …….

- There is only some minimum amount of space in the free-storage list, or
- There is no space at all left in the free-storage list, or
- When the CPU is idle and has time to do the collection

**The garbage collection is invisible to the programmer**

# LINKED LISTS: Overflow

Sometimes new data are to be inserted into a data structure but there is no available space (the free-storage list is empty). This situation is usually called ***Overflow.***

**Usually, Overflow will occur with our linked list when AVAIL=NULL and there is an insertion.**

***How we can handle Overflow situation?***

We may handle the *Overflow* situation by……….

- ✓ Printing the message **OVERFLOW**
- ✓ We may modify the program by adding space to the underlying arrays

# LINKED LISTS: **Underflow**

Sometime someone wants to delete data from a data structure where the data structure is empty. This situation is usually called **Underflow**.

Usually, underflow will occur with our linked lists when START=NULL and there is a deletion.

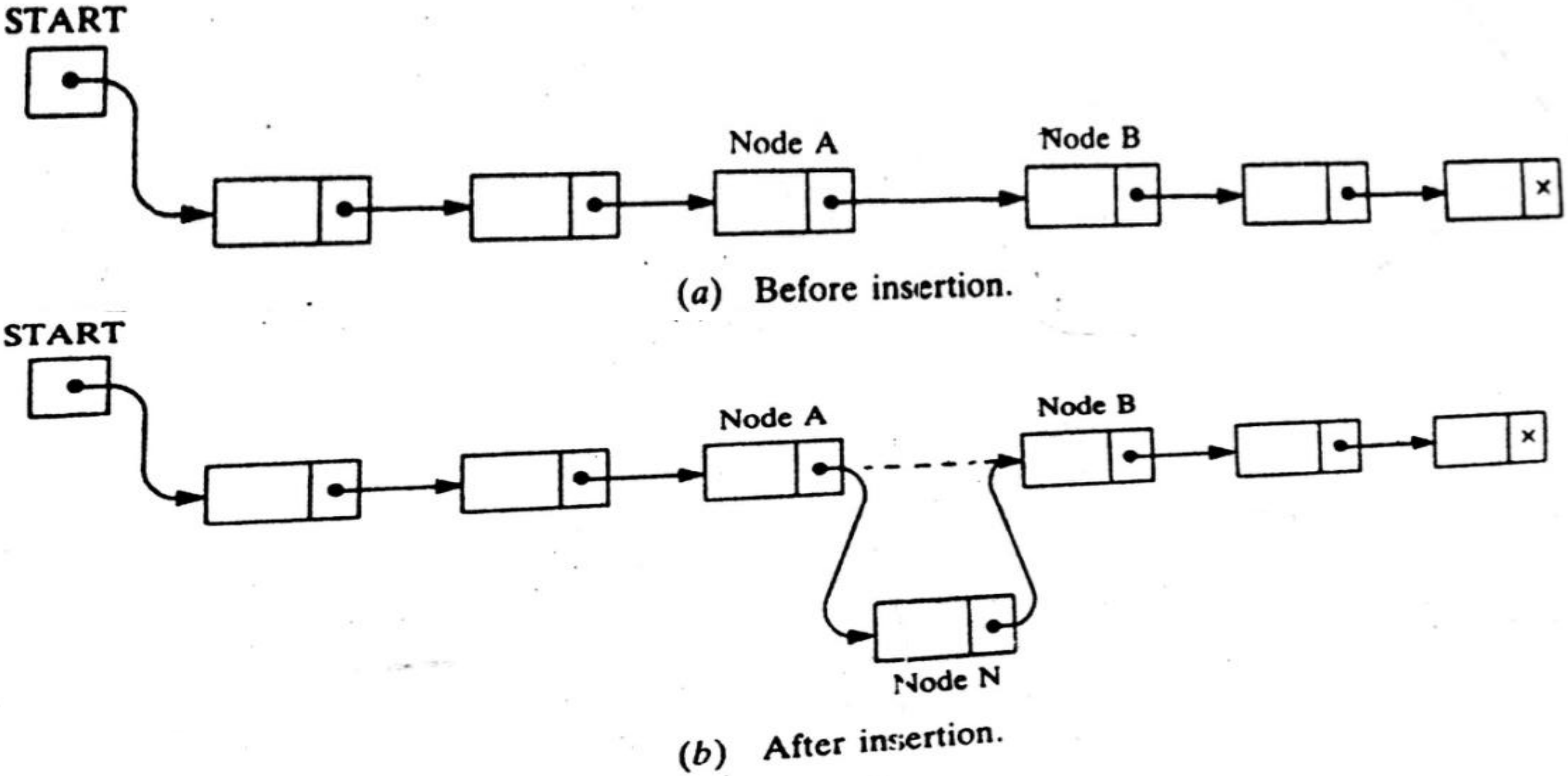
***How we can handle Underflow situation?***

We may handle the *Overflow* situation by.........

- ✓ Printing the message UNDERFLOW

# LINKED LISTS: **Insertion**

Let LIST be a linked list with successive nodes A and B. Suppose, a node N is to be inserted into the list between nodes A and B.



(a) Before insertion.

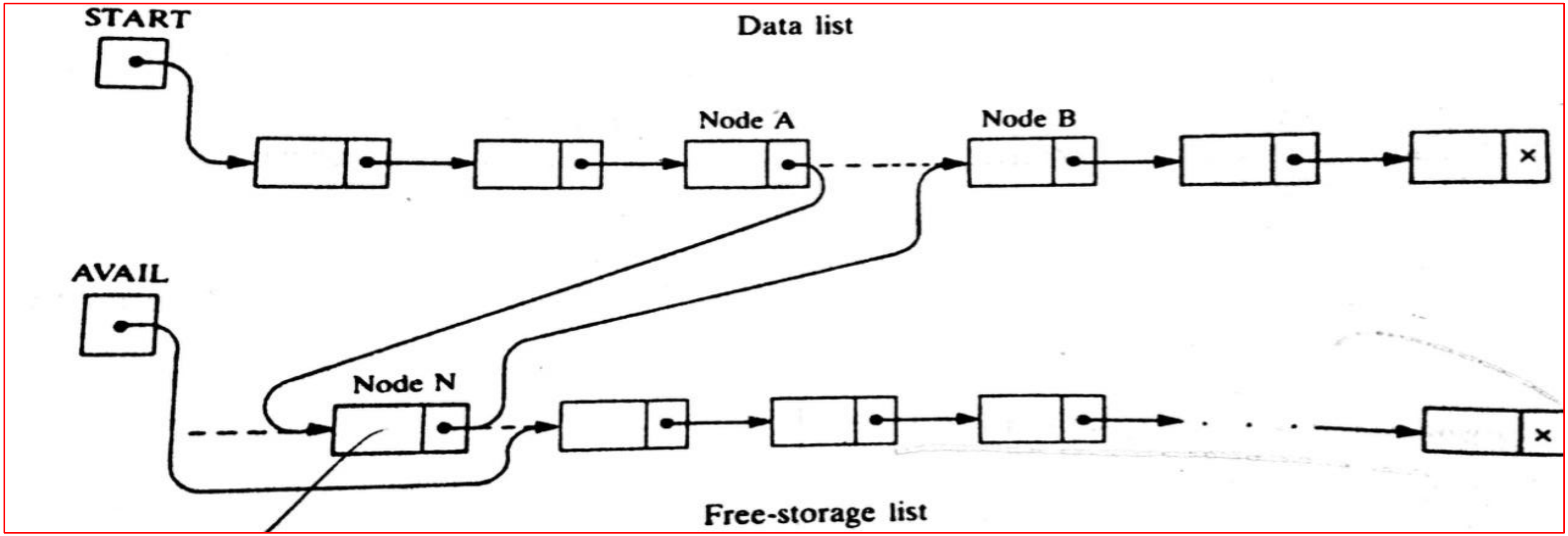
(b) After insertion.

Here, it does not take into account the memory space for the new node N will come from the AVAIL list.

But…..for easier processing, the first node in the AVAIL will be used for the new node N.

# LINKED LISTS: **Insertion**

## More exact Illustration of such an insertion



Here, THREE pointer fields are changed: ???

**Which are?**

- The nextpointer field of node A now points to the new node N.
- AVAIL now points to the second node in the free pool.
- The nextpointer field of node N now point to node B.

# LINKED LISTS: Insertion

**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 is the last node in the list, then N will contain the null pointer.

# LINKED LISTS: Insertion Algorithms

- **Insertion Algorithms in Linked List**
- **Inserting element at the Beginning of a Linked List**
- **Inserting element after a Given node of a Linked List**
- **Inserting element into a sorted Linked List.**

**Good Luck …!!**

# LINKED LISTS: Insertion Algorithms

- ✓ Algorithms which inserts nodes into linked lists come up in various situations :
  - ➢ Inserts a node at the beginning of the list,
  - ➢ Inserts a node after the node with a given location, and
  - ➢ Inserts a node into a sorted list.
- ✓ All the algorithms:
  - ➢ assume that linked list is in memory in the form

    **LIST(INFO, LINK, START, AVAIL)**
  - ➢ Have a variable **ITEM** which contains the new information to be added to the list.

# LINKED LISTS: Insertion Algorithms

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

- 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**.

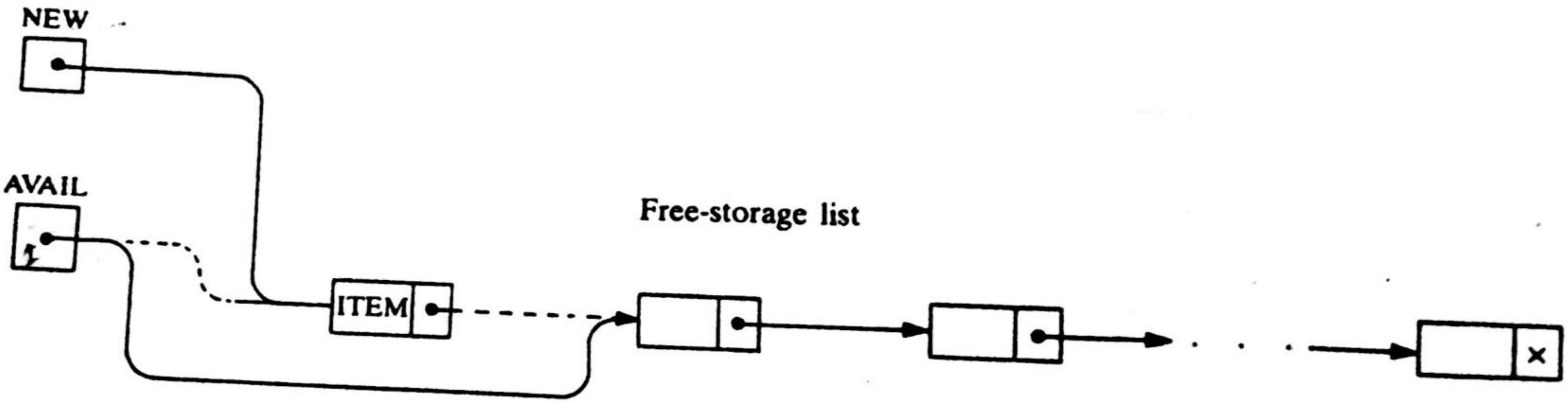
- Removing the first node from the AVAIL list. Using the variable NEW to keep track of the location of the new node.

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

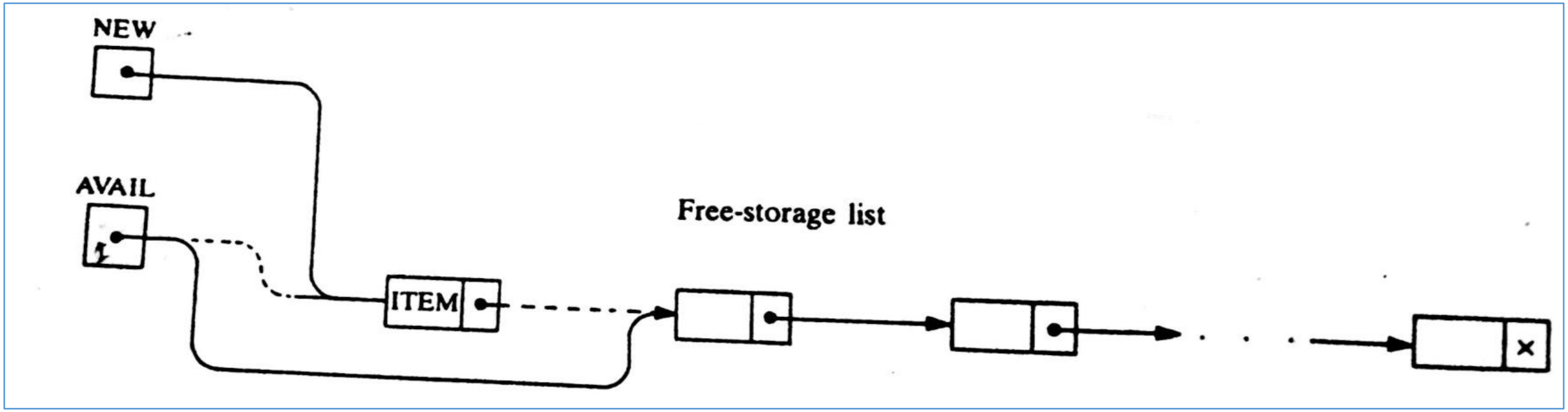
- Copying new information into the new node. i.e.,

  **INFO[NEW]=ITEM**

# LINKED LISTS: Insertion Algorithms



# LINKED LISTS: Inserting at the Beginning of a list



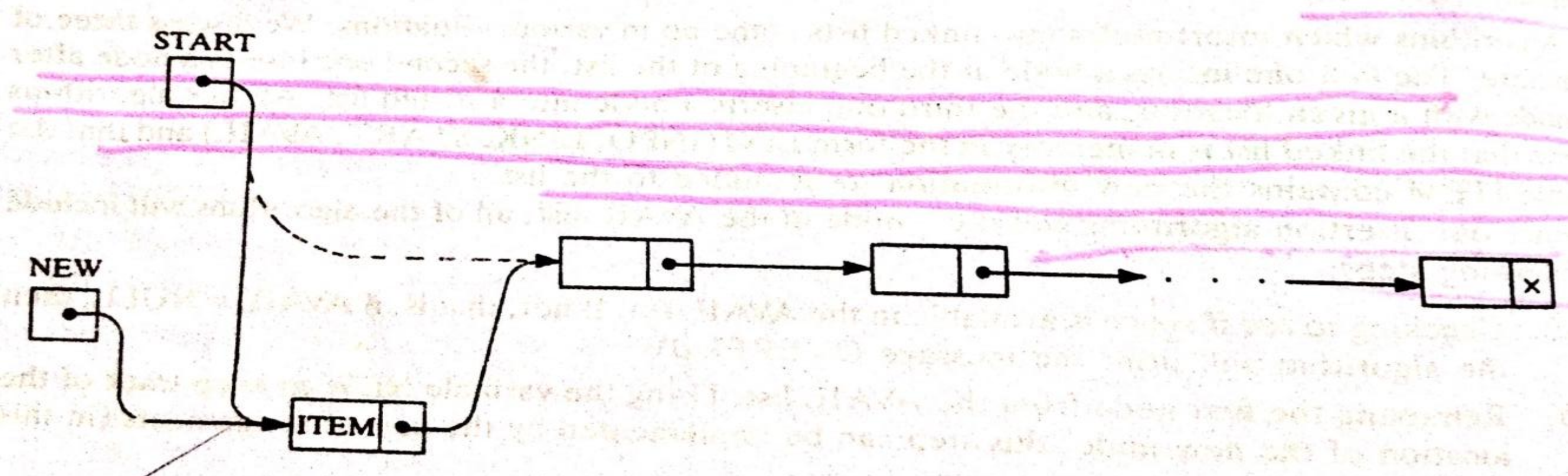


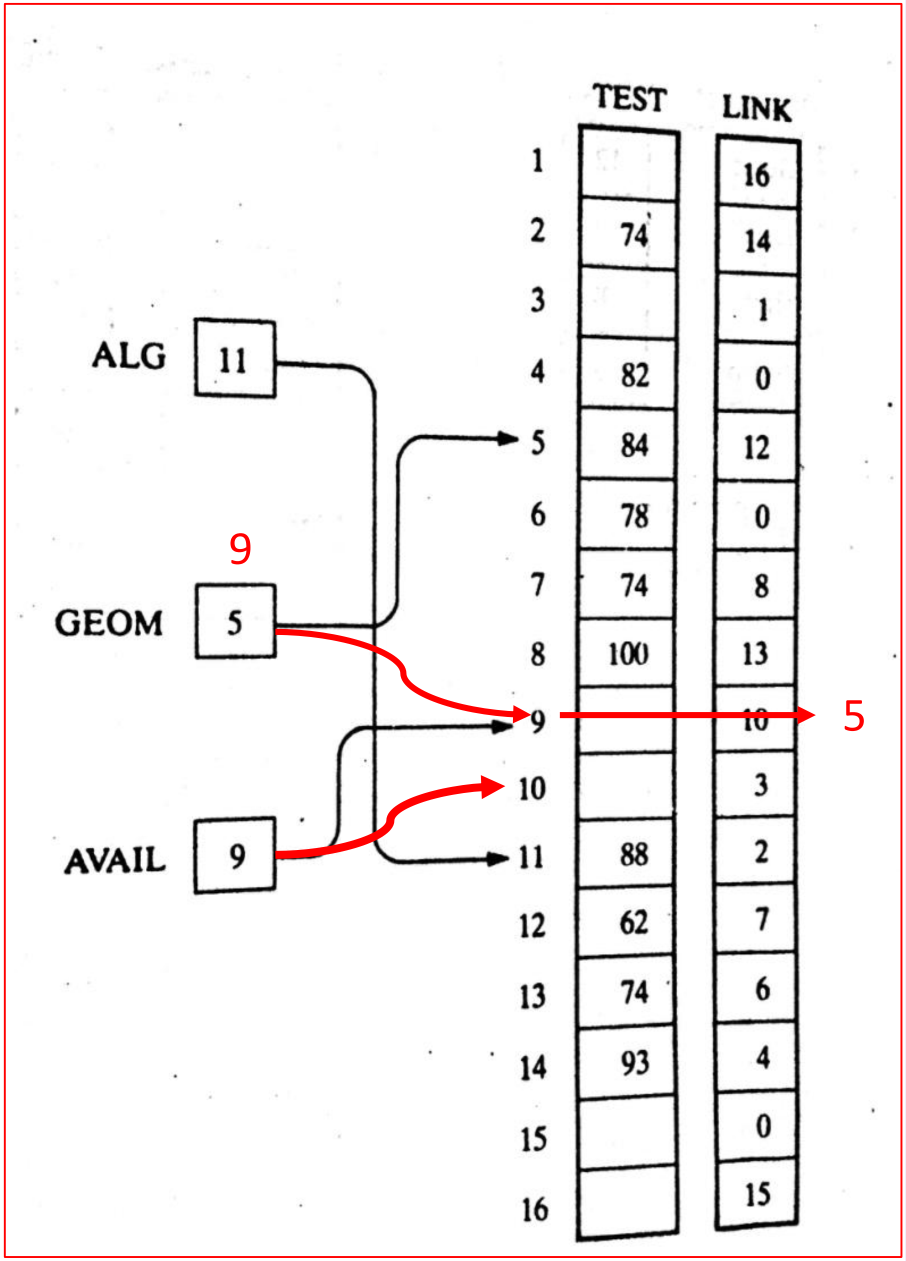
**Fig. 5-18** Insertion at the beginning of a list.

# LINKED LISTS: Inserting at the Beginning ….ALG

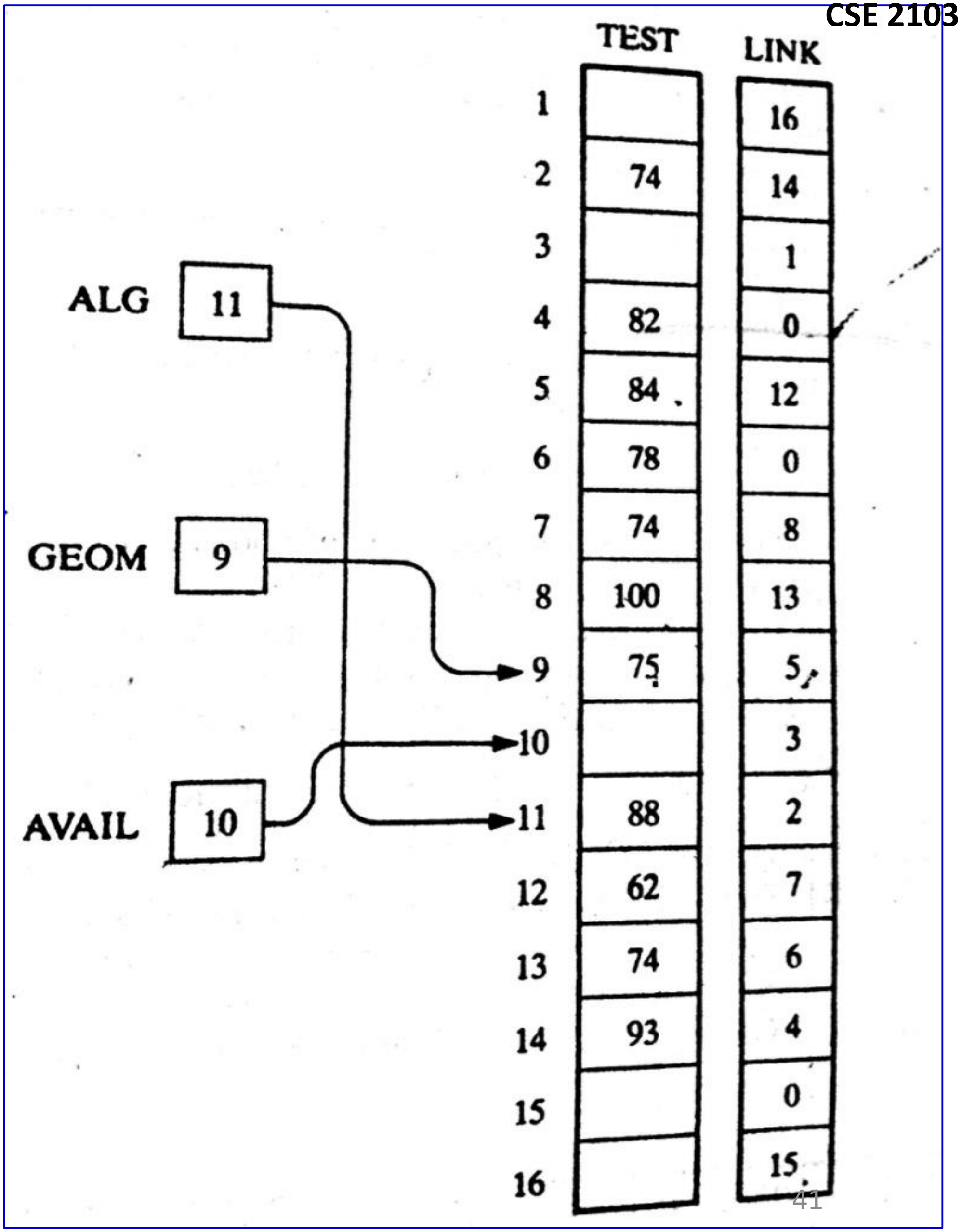
**INSFIRST (INFO, LINK, START, AVAIL,ITEM)**

```
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. [Change START so it points to the new node.]
6.  Exit.
```

# LINKED LISTS: Inserting at the Beginning ….Exmp



| | TEST | LINK |
|---|---|---|
| 1 | | 16 |
| 2 | 74 | 14 |
| 3 | | 1 |
| 4 | 82 | 0 |
| 5 | 84 | 12 |
| 6 | 78 | 0 |
| 7 | 74 | 8 |
| 8 | 100 | 13 |
| 9 | | 10 |
| 10 | | 3 |
| 11 | 88 | 2 |
| 12 | 62 | 7 |
| 13 | 74 | 6 |
| 14 | 93 | 4 |
| 15 | | 0 |
| 16 | | 15 |



| | TEST | LINK |
|---|---|---|
| 1 | | 16 |
| 2 | 74 | 14 |
| 3 | | 1 |
| 4 | 82 | 0 |
| 5 | 84 | 12 |
| 6 | 78 | 0 |
| 7 | 74 | 8 |
| 8 | 100 | 13 |
| 9 | 75 | 5 |
| 10 | | 3 |
| 11 | 88 | 2 |
| 12 | 62 | 7 |
| 13 | 74 | 6 |
| 14 | 93 | 4 |
| 15 | | 0 |
| 16 | | 15 |

# LINKED LISTS: Inserting after a given node

## INSLOC (INFO, LINK, START, AVAIL, LOC, ITEM)

```
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 if If structure.]
5.  Exit.
```

# LINKED LISTS: Inserting into a Sorted Linked List

Given Element → ITEM

ITEM must be inserted between nodes A and B so that

**INFO[A] < ITEM <= INFO[B]**

Thus,

- Firstly we should write an Algorithm to find the location, LOC, where is suitable to insert the **ITEM** in the sorted linked list.
- Secondly, we have to write the algorithm to insert to the found Location.

# LINKED LISTS: Find LOC in a Sorted Linked List

## FINDA(INFO, LINK, START, ITEM, LOC)

```
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].[Initialize 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]. [Update pointers.]
   [End of Step 4 loop.]
7. Set LOC:= SAVE.
8. Return
```

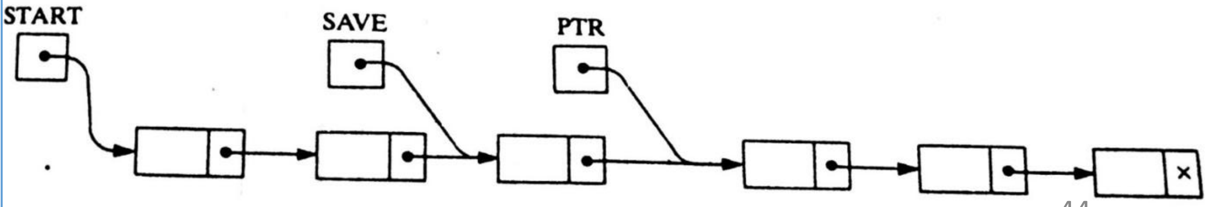


Fig. 5-20

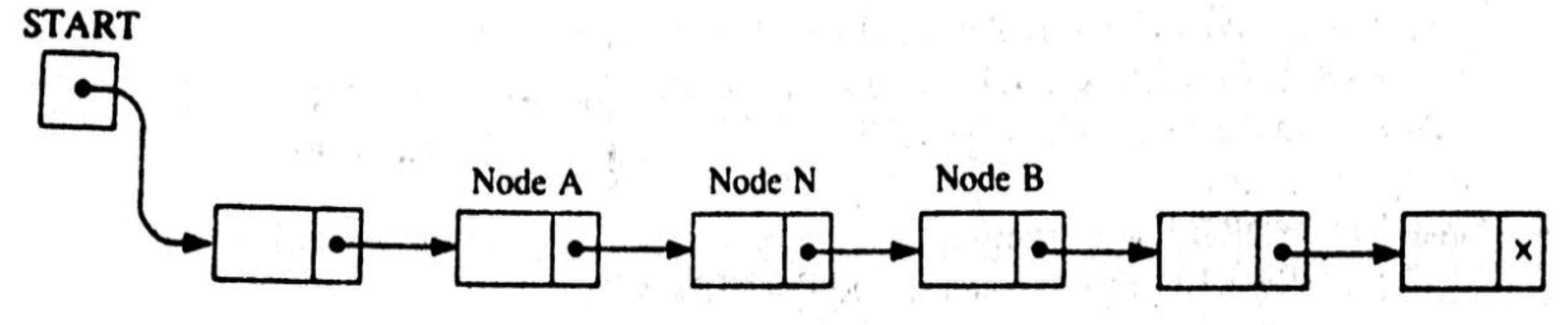
# LINKED LISTS: Insert ITEM into a Sorted Linked List

**INSSRT(INFO, LINK, START, AVAIL, ITEM)**

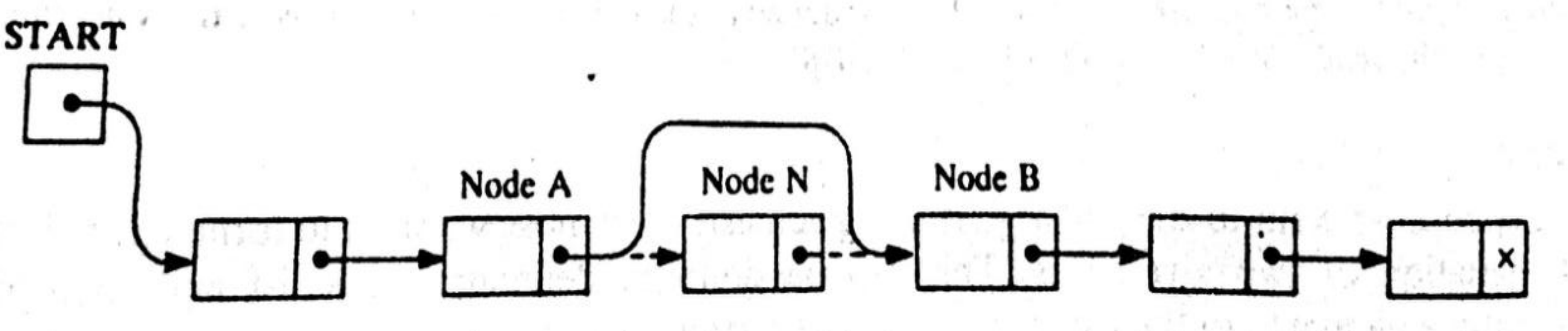
1. [Use **FINDA(INFO, LINK, START, ITEM, LOC)** to find the location of the proceeding ITEM]
2. Use **INSLOC (INFO, LINK, START, AVAIL, LOC, ITEM)** to insert ITEM after the node with location LOC.]
3. Exit.

# LINKED LISTS: Deletion from a Linked List

Let LIST be a linked list with a node N between nodes A and B.



(*a*) Before deletion.

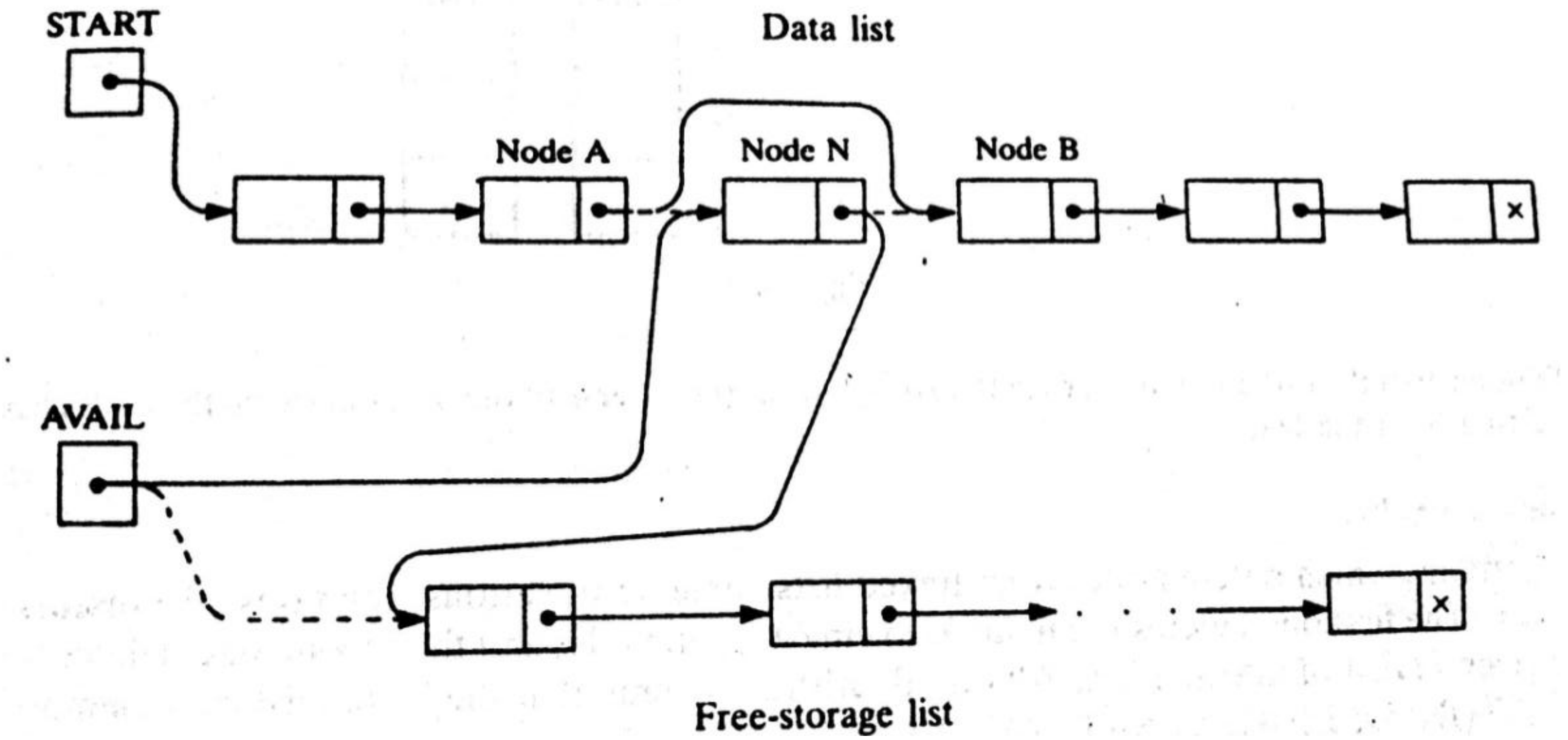


(*b*) After deletion.

Here we don’t care about the future of the deleted nodes of the linked list

# LINKED LISTS: Deletion from a Linked List

## More Exact procedure:

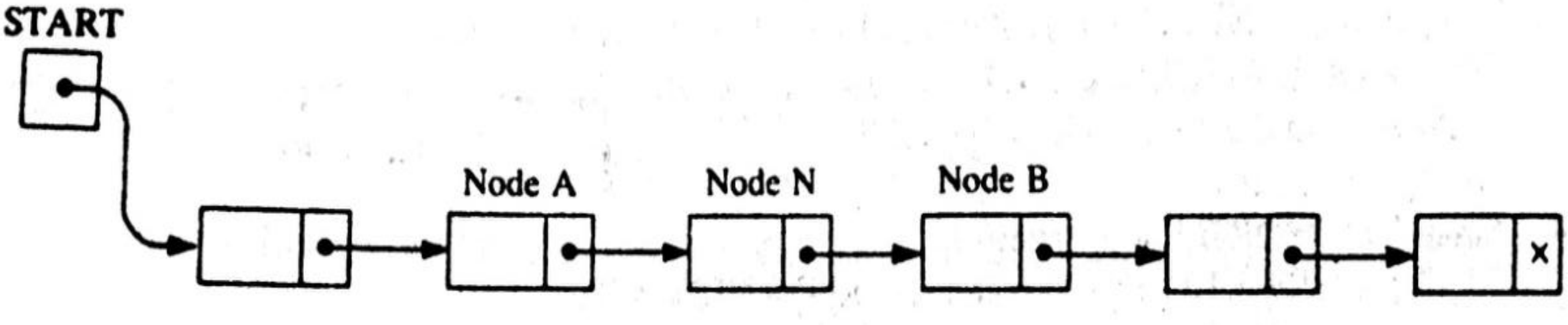


**Three pointer fields are changed:**

- The nextpointer field of node A now points to node B.
- The nextpointer field of N now points to the original first node in the free pool.
- AVAIL now points to the deleted node N.

# LINKED LISTS: Deletion from a Linked List

**There are also TWO special cases:**



(*a*) Before deletion.

- If the deleted node N is the first node in the list
  - ✓ **START will point to the node B**
- If the deleted node N is the last node in the list
  - ✓ **Node A will contain the NULL pointer**

# LINKED LISTS: Deletion Algorithms

Algorithms which delete nodes from linked lists come up in various situations:

- The first one deletes the node following a given node, and
- The second one deletes the node with a given ITEM of information

All algorithms assume that linked list is in memory in the form

LIST(INFO, LINK, START, AVAIL)

**All the algorithms will return the memory space of the deleted node N to the beginning of the AVAIL list.**

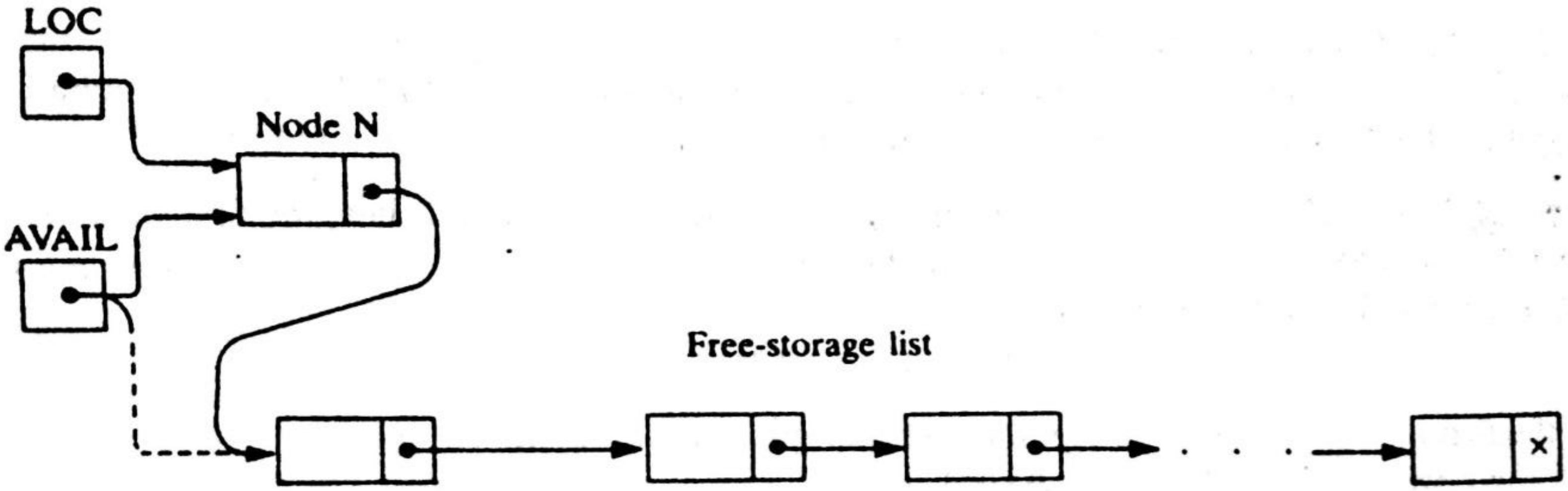


Fig. 5-25 LINK[LOC] := AVAIL and AVAIL := LOC.

# LINKED LISTS:

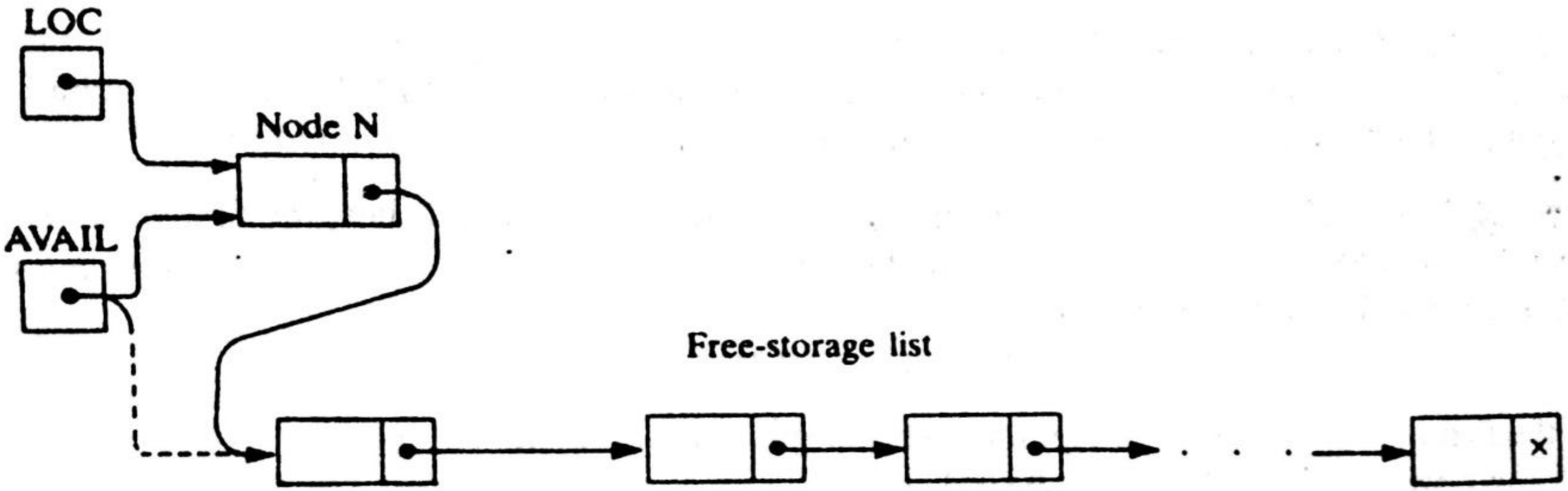


**Fig. 5-25** LINK[LOC] := AVAIL and AVAIL := LOC.

All algorithms will include the following pair of assignments where LOC is the location of the deleted node N:

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

# LINKED LISTS: Deleting the node following a Given Nod

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

**(it deletes the node N with location LOC, LOCP is the location of the node which precedes N)**

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

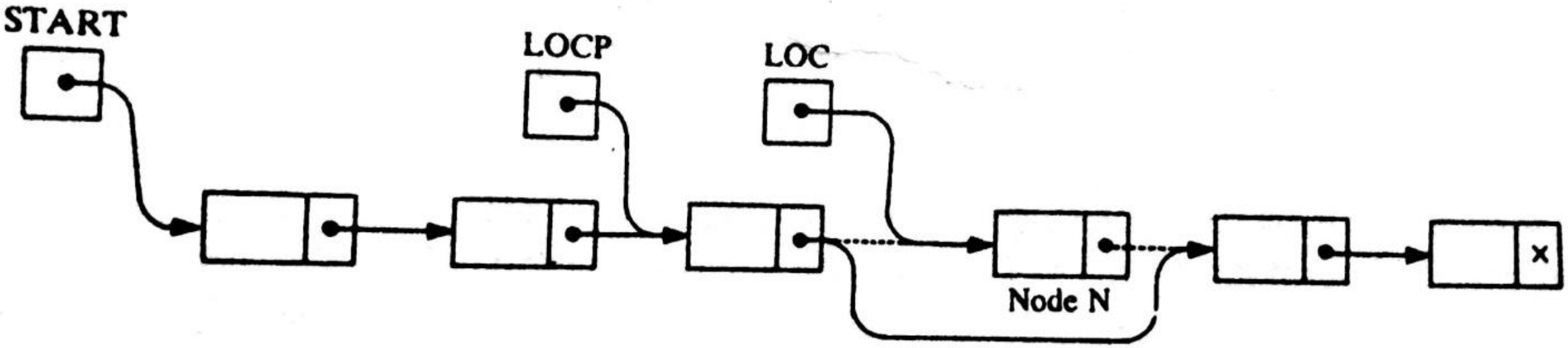


**Fig. 5-27** LINK[LOCP]:= LINK[LOC].

# LINKED LISTS: Deleting the node with a given ITEM

- 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 which contain ITEM.
- The algorithm has TWO **PARTS**

  - 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 with ITEM, we set LOCP=NULL
    - If ITEM does not appear in LIST, we set LOC=NULL

  - Traverse the list, using pointer variable PTR and comparing ITEM with INFO[PTR]at each node.
    - While track the location of the preceding node by using a pointer variable SAVE. Thus

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

# LINKED LISTS: Find LOC of the 1st node N which contains ITEM

## FINDB (INFO, LINK, START, ITEM, LOC, LOCP)

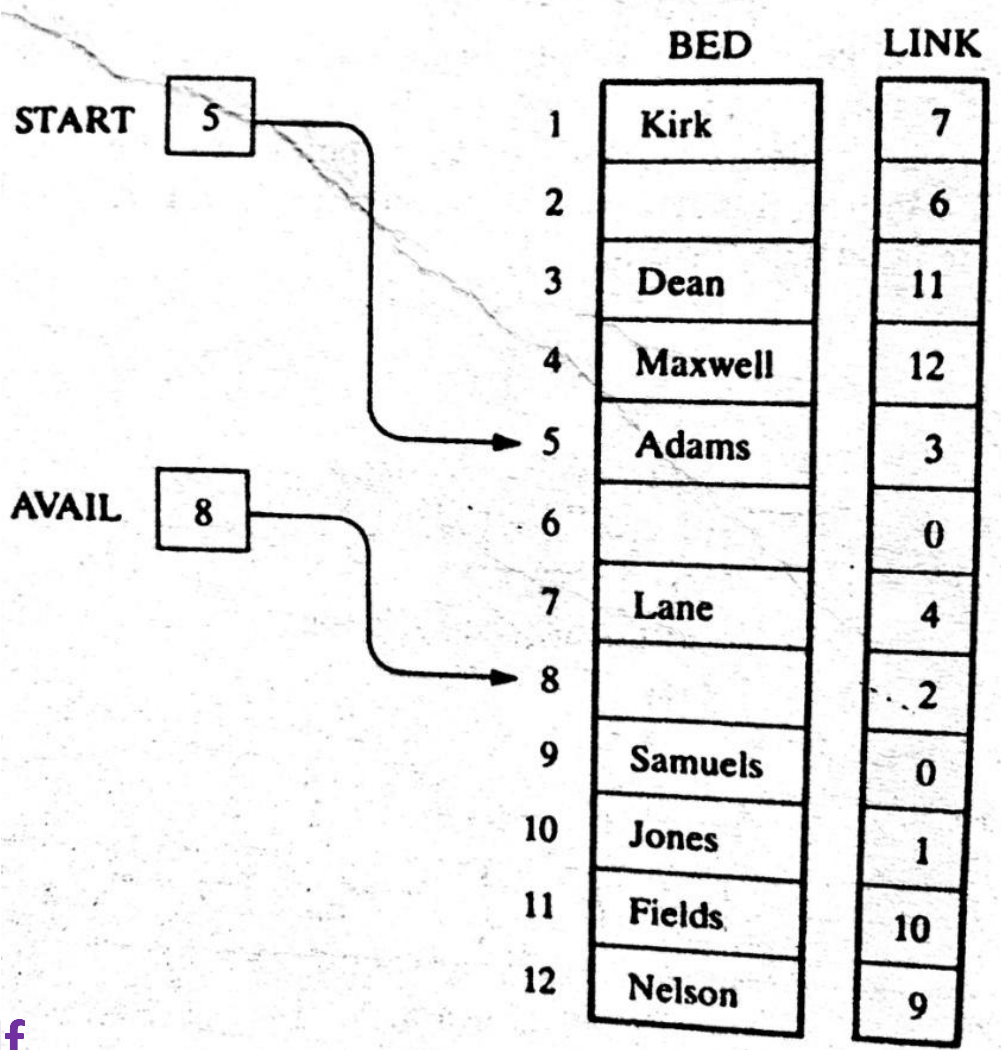
```
Step 1. [List Empty?] If Start=NULL, then:
                Set LOC:=NULL and LOCP:=NULL, and Return.
        [End of If Structure]
Step 2. [ITEM in first node?] If INFO[START]=ITEM, then:
                Set LOC:=START and LOCP=NULL, and Return.
        [End of If structure]
Step 3. Set SAVE:=START and PTR:=LINK[START]. [Initialize pointers]
Step 4. Repeat Steps 5 and 6 while PTR≠ NULL.
Step 5.         If INFO[PTR]=ITEM, then:
                        Set LOC:=PTR and LOCP:=SAVE, and Return.
                [End of If structure]
Step 6.         Set SAVE:=PTR and PTR:=LINK[PTR]. [Update pointers]
        [End of Step 4 loop]
STEP 7. Set LOC:=NULL. [Search Unsuccessful.]
Step 8. Return
```

# LINKED LISTS: Delete the 1st node which contains the given ITEM of Info

## DELETE (INFO, LINK, START, AVAIL, ITEM)

```
Step 1. Call FINDB(INFO, LINK, START, ITEM, LOC, LOCP)
Step 2. If LOC=NULL, then: Write: ITEM not in list, and Exit.
Step 3. [Delete node.]
        If LOCP=NULL, then
                Set Start:=LINK[START]. [Delete fist node]
        Else:
                Set LINK[LOCP]:=LINK[LOC].
        [End of If structure]
Step 4. [Return deleted node to the AVAIL list.]
        Set LINK[LOC]:=AVAIL and AVAIL:=LOC.
Step 5. Exit
```

# LINKED LISTS:



START 5

AVAIL 8

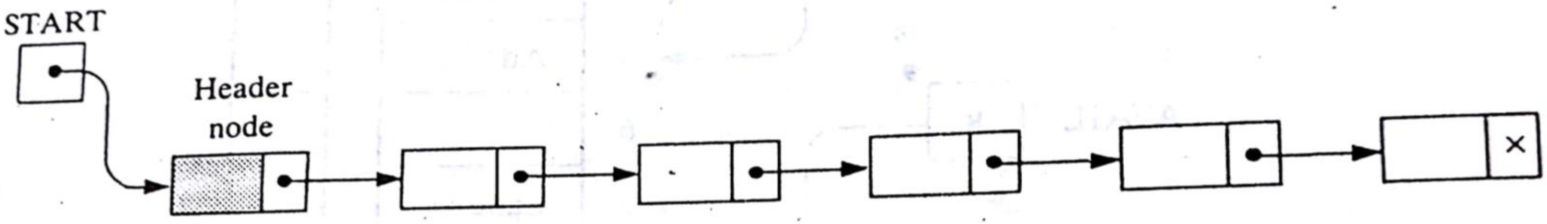
|  | BED | LINK |
|---|---|---|
| 1 | Kirk | 7 |
| 2 |  | 6 |
| 3 | Dean | 11 |
| 4 | Maxwell | 12 |
| 5 | Adams | 3 |
| 6 |  | 0 |
| 7 | Lane | 4 |
| 8 |  | 2 |
| 9 | Samuels | 0 |
| 10 | Jones | 1 |
| 11 | Fields | 10 |
| 12 | Nelson | 9 |

Fig. 5-28

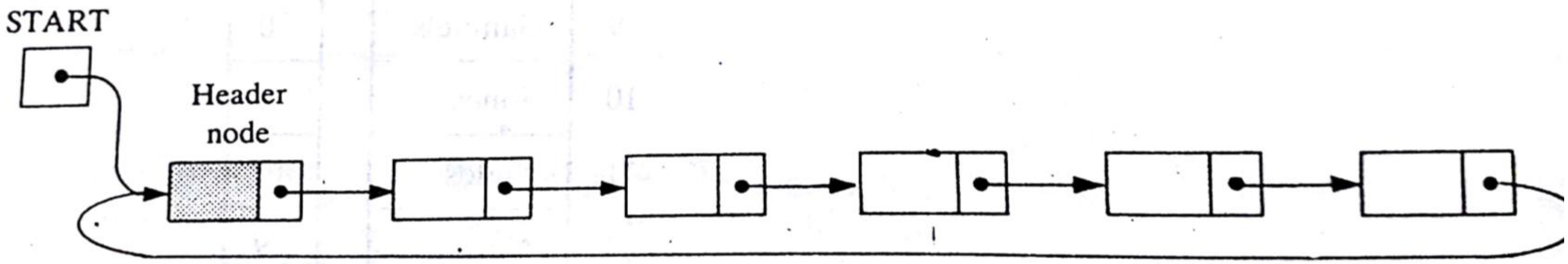
**Example 5.17 Teach Yourself**

# Header Linked LISTS

- A header lined 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:
  - **A Grounded Header List**:
  - **A Circular List:**



(*a*) Grounded header list.

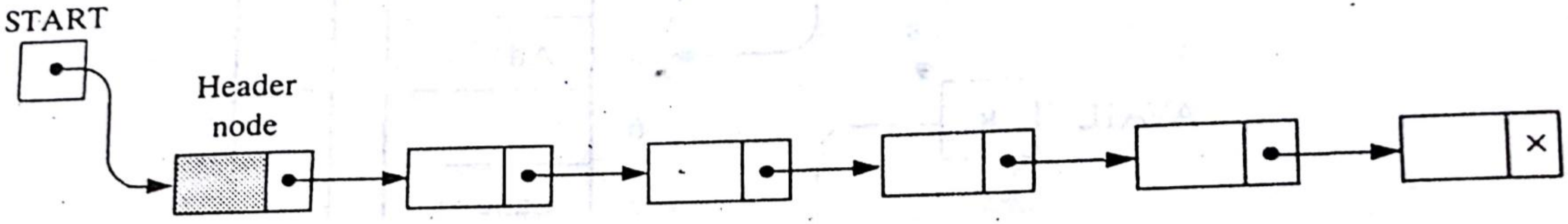


(*b*) Circular header list.

# Header Linked LISTS

**Grounded Header List**: is a header list where the last node contains the null pointer.

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

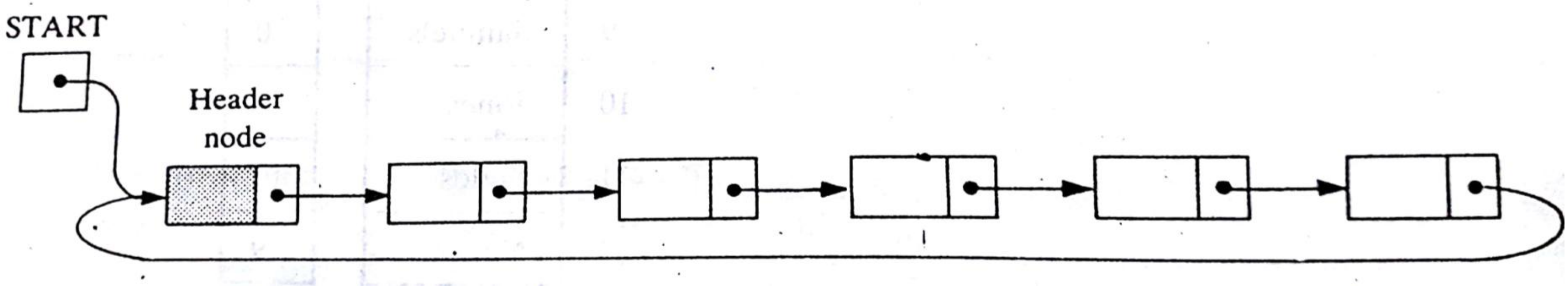


(*a*) Grounded header list.

# Header Linked LISTS

**A Circular Header List**: is a header list where the last node points back to the header node.

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



# Header Linked LISTS: Traversing

Traversing a Circular Header List(INFO, LINK, START)

```
Step1.  Set PTR:=LINK[START].[Initialize pointer PTR.]
Step2.  Repeat Steps 3 and 4 while PTR ≠ START
Step3.          Apply PROCESS to INFO[PTR]
Step4.          Set PTR:=LINK[PTR]. [PTR now point to the next node.]
        [End of step 2 loop]
Step5.  Exit.
```

LINKED_LIST_Traversing(INFO, LINK, START)

```
Step1.  Set PTR:=START.[Initialize pointer PTR.]
Step2.  Repeat Steps 3 and 4 while PTR ≠ NULL
Step3.          Apply PROCESS to INFO[PTR]
Step4.          Set PTR:=LINK[PTR]. [PTR now point to the next node.]
        [End of step 2 loop]
Step5.  Exit.
```

# Header Linked LISTS:

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

**SRCHHL(INFO, LINK, START, ITEM, LOC)**

[Finds the location of the first node in LIST which contains ITEM in a circular header list]

```
Step1. Set PTR:= LIST [START].
Step2. Repeat while INFO[PTR] ≠ITEM and PTR ≠ START:
        Set PRT := LINK [PTR]. [PTR now points to the next node.]
        [End of Loop]
Step3. If INFO[PTR]=ITEM, then:
                Set LOC:= PTR.
        ELSE:
        SET LOC:=NULL.
        [End of If structure]
Step4. Exit.
```

The two test which control the searching loop were not performed at the same time in the ordinary linked lists.

# Linked LISTS: Searching

**SEARCH (INFO, LINK, START, ITEM, LOC)**

```
Step1.  Set PTR:=START
Step2.  Repeat Step 3 while PTR ≠ NULL
Step3.      If ITEM=INFO [PTR], then:
                Set LOC:=PTR, and Exit.
            Else:
                Set LOC:=LINK[PTR]. [PTR now points to the next node.]
            [End of If structure.]
        [End of Step 2 loop.]
Step4.  [Search is unsuccessful.] Set LOC:=NULL.
Step5.  Exit.
```

# Circular Header Linked LISTS: Find Location

## FINDBHL (INFO, LINK, START, ITEM, LOC, LOCP)

**Finds the location LOC of the first node N which contains ITEM and also the location LOCP of the node preceding N.)**

```
Step1.   Set SAVE:=START and PTR:= LINK[START] [Initialize pointers]
Step2. Repeat while $INFO[PTR] \neq ITEM$ and $PTR \neq START$:
          Set SAVE := PTR and PTR:= LINK [PTR]. [Update pointers.]
          [End of Loop]
Step3. If INFO[PTR]=ITEM, then:
                    Set LOC:= PTR and LOCP:=SAVE.
          ELSE:
          SET LOC:=NULL and LOCP:=SAVE
          [End of If structure]
Step4. Exit.
```

# Circular Header Linked LISTS: Delete

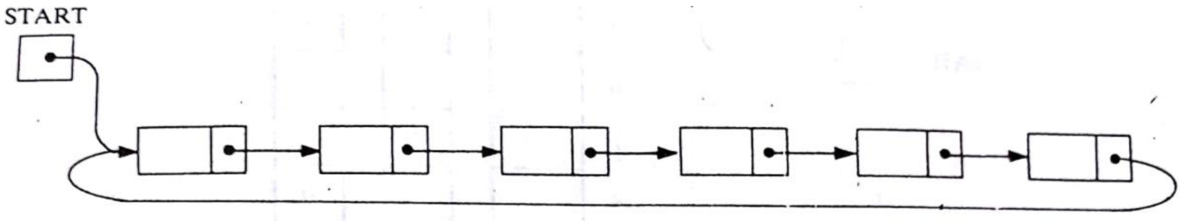
**DELLOCHL (INFO, LINK, START, AVAIL, ITEM)**

**(Deletes the first nodes N which contains ITEM when LIST is a Circular Header List.**

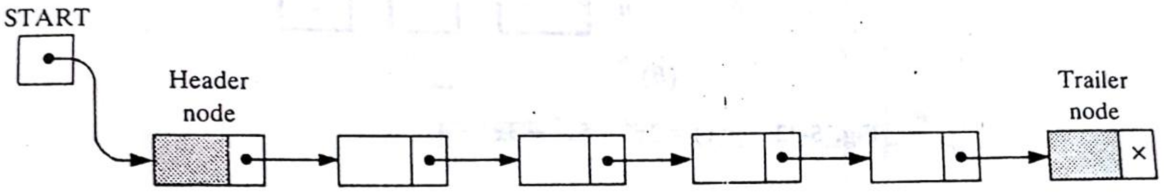
```
Step1.   Call FINDBHL (INFO, LINK. START, ITEM, LOC, LOCP)

Step2. If LOC=NULL, then: write: ITEM not in list, and Exit.

Step3. Set LINK[LOCP]:=LINK [LOC] [Delete Node.]

Step4. [Return deleted node to the AVAIL list.]

             Set LINK[LOC]:=AVAIL and AVAIL:=LOC.

Step5. Exit
```

# TWO other variations of LINKED LIST

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 contain both a special header node at the beginning of the list and a special trailer node at the end of the list.



(*a*) Circular linked list.



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

# TWO-WAY LISTS:

**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 contain the location of the preceding node in the list.

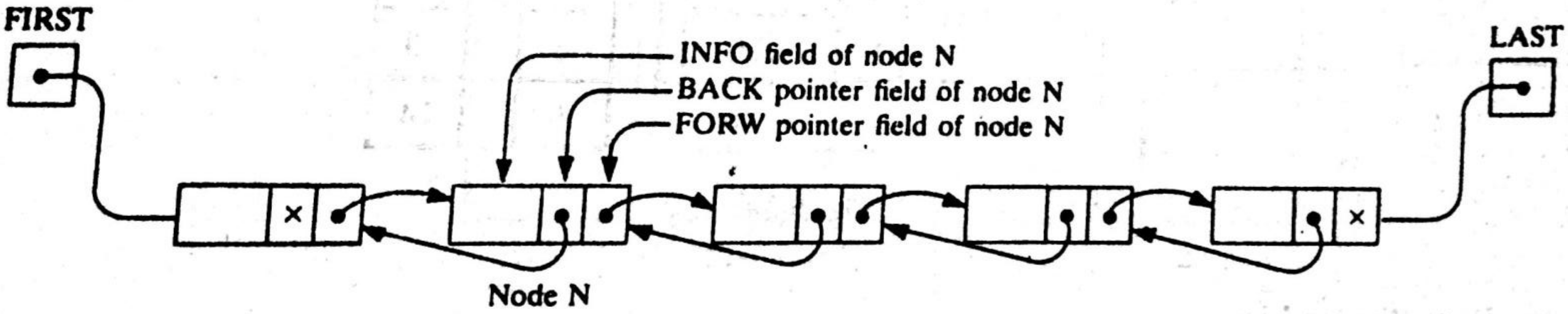


Fig. 5-33 Two-way list.

# TWO-WAY LISTS: Operations

## Deleting:

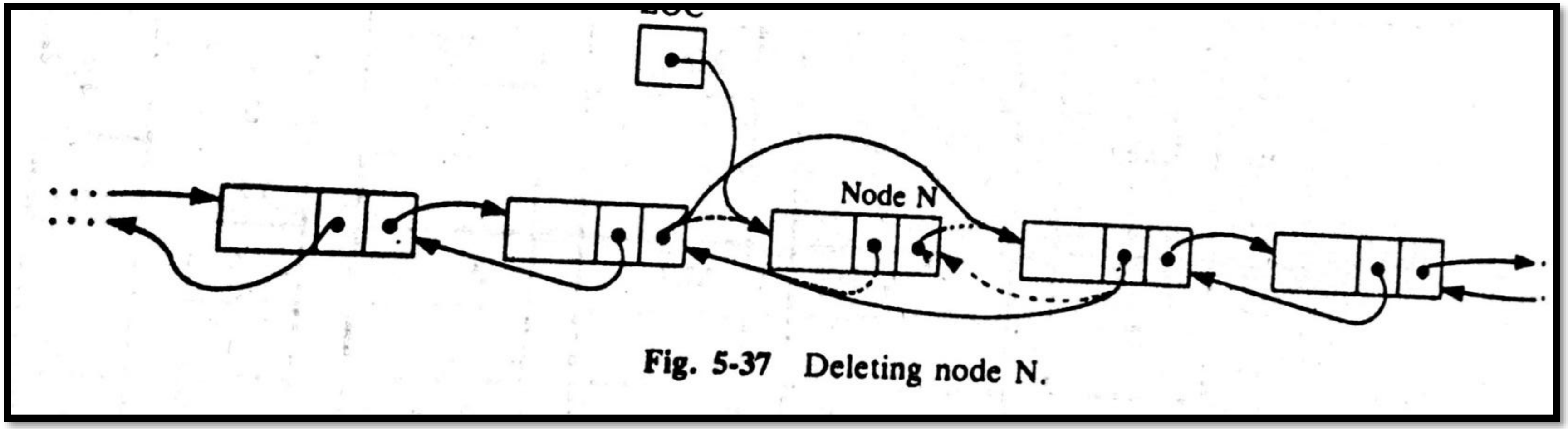


Fig. 5-37 Deleting node N.

## Inserting:

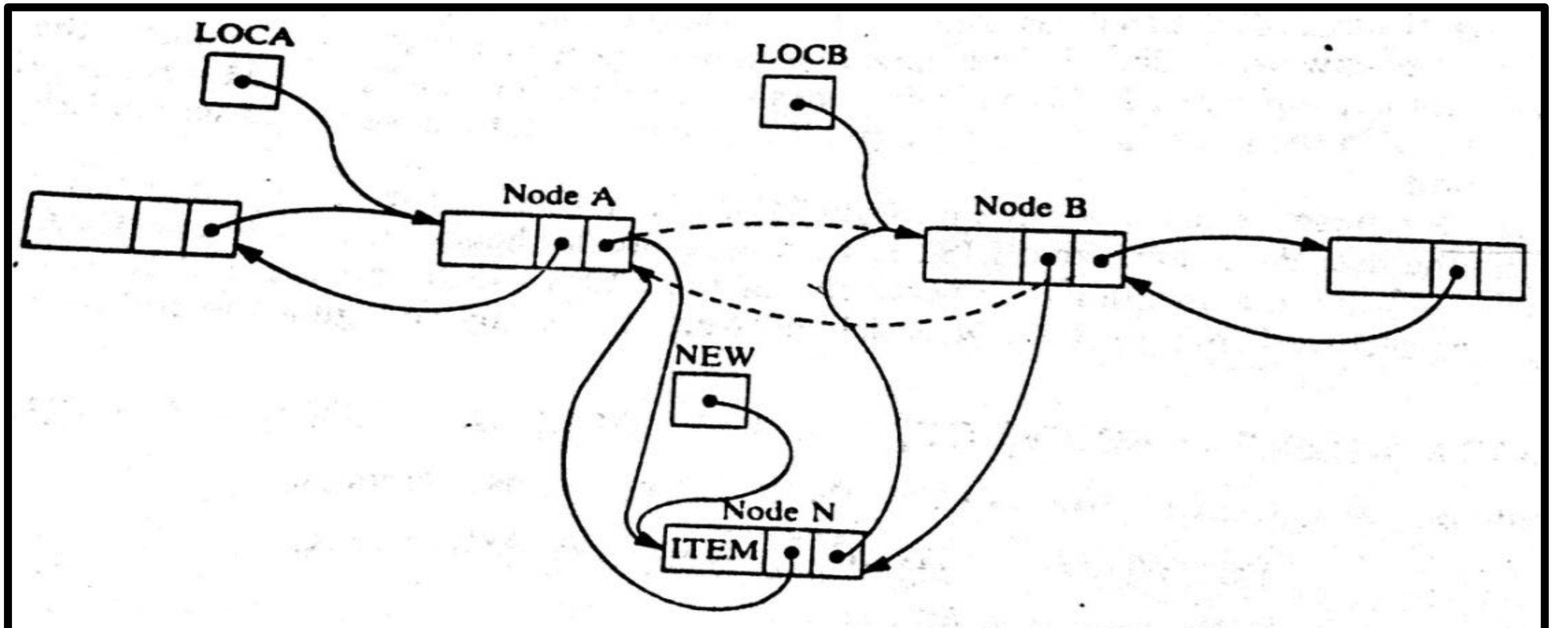


Fig. 5-38 Inserting node N.

# LINKED LISTS: Insertion Algorithms

**Prepare your NICE and INTERACTIVE presentation slides on the following topics individually:**

- **Insertion Algorithms in Linked List**
- **Inserting element at the Beginning of a Linked List**
- **Inserting element after a Given node of a Linked List**
- **Inserting element into a sorted Linked List.**

Presentation Schedule: Next Class Period
Presentation Duration: 7-8 minutes + 2-3 minutes QA.

**Good Luck ...!!**