CSE-213
(Data Structure)

Lecture on
Chapter-4: LINKED LISTS

Md. Jalal Uddin

Lecturer

Department of CSE

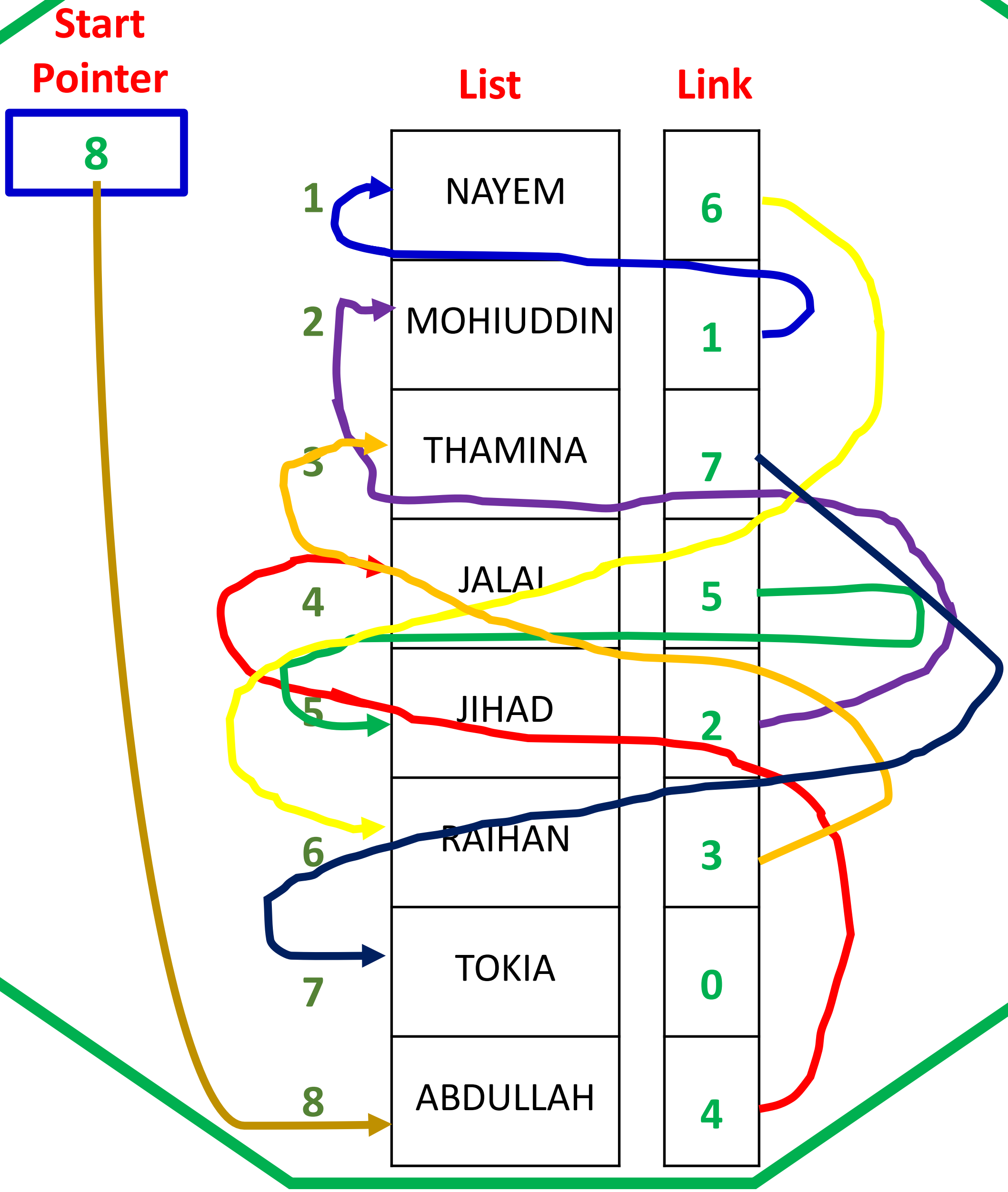
City University

Email: jalalruice@gmail.com

No: 01717011128 (Emergency Call)

Department of Computer Science & Engineering (CSE)
City University, Khagan, Birulia, Savar, Dhaka-1216, Bangladesh

LINKED LISTS:



Start Pointer

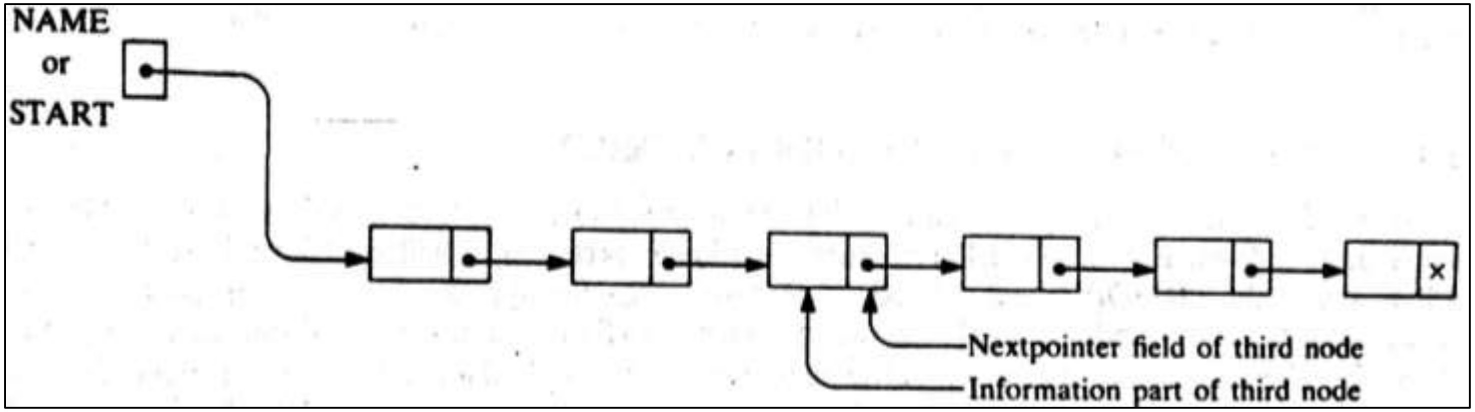
8

	List	Link
1	NAYEM	6
2	MOHIUDDIN	1
3	THAMINA	7
4	JALAL	5
5	JIHAD	2
6	RAIHAN	3
7	TOKIA	0
8	ABDULLAH	4

LINKED LIST

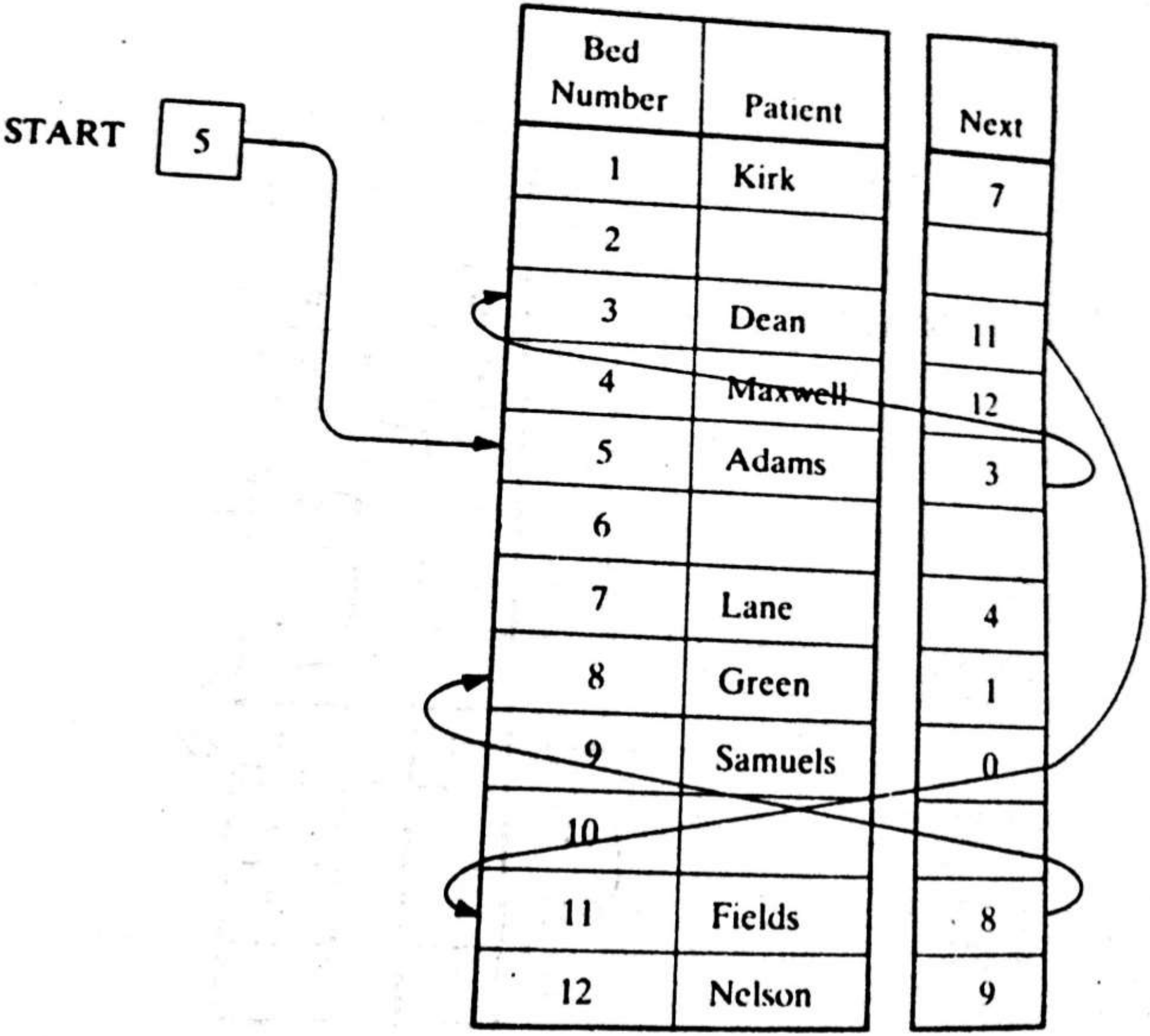
LINKED LISTS: Definition

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

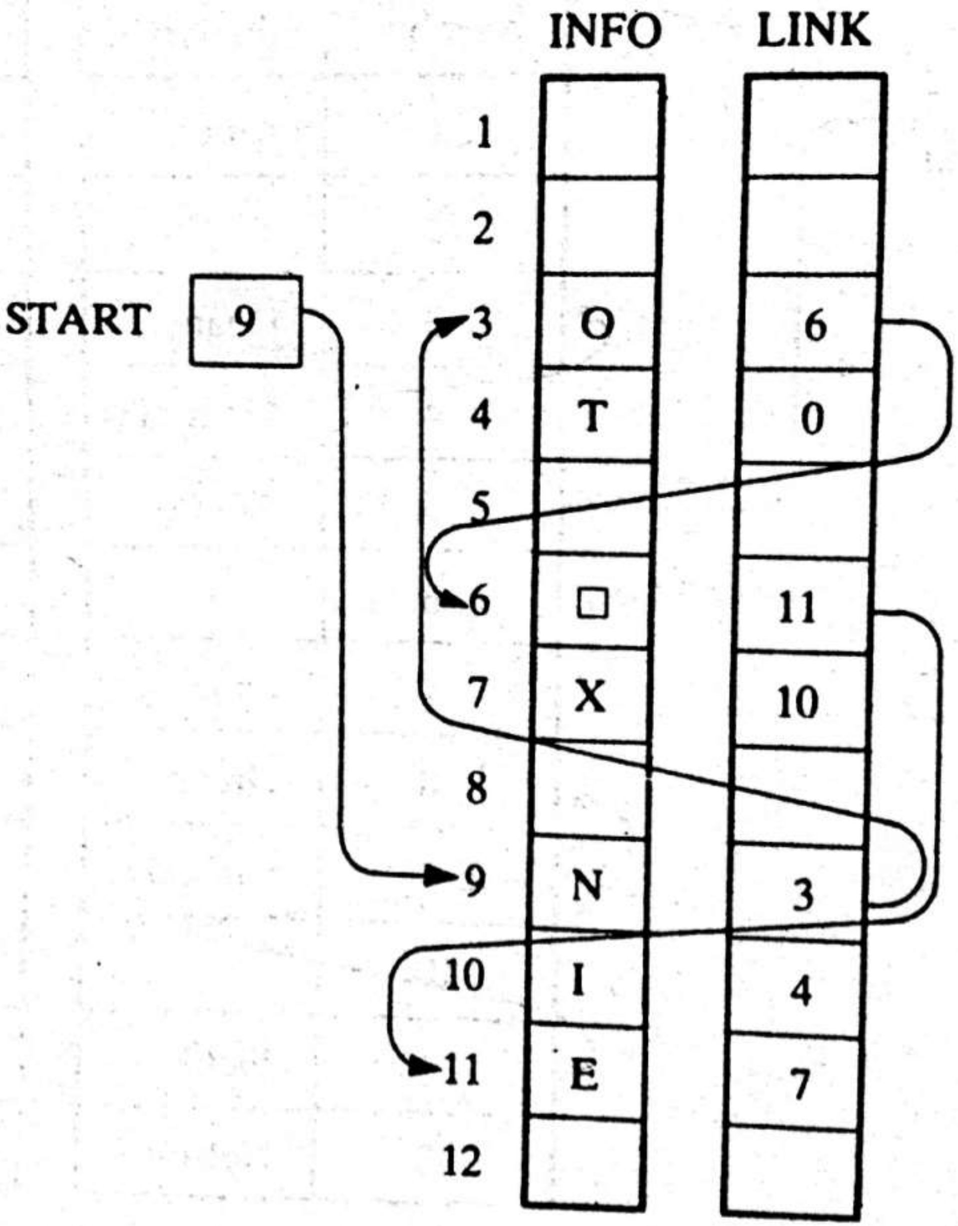
Let LIST be a linked list which will be maintained in the memory

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

	INFO	LINK
1		
2		
3	O	6
4	T	0
5		
6	□	11
7	X	10
8		
9	N	3
10	I	4
11	E	7
12		

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

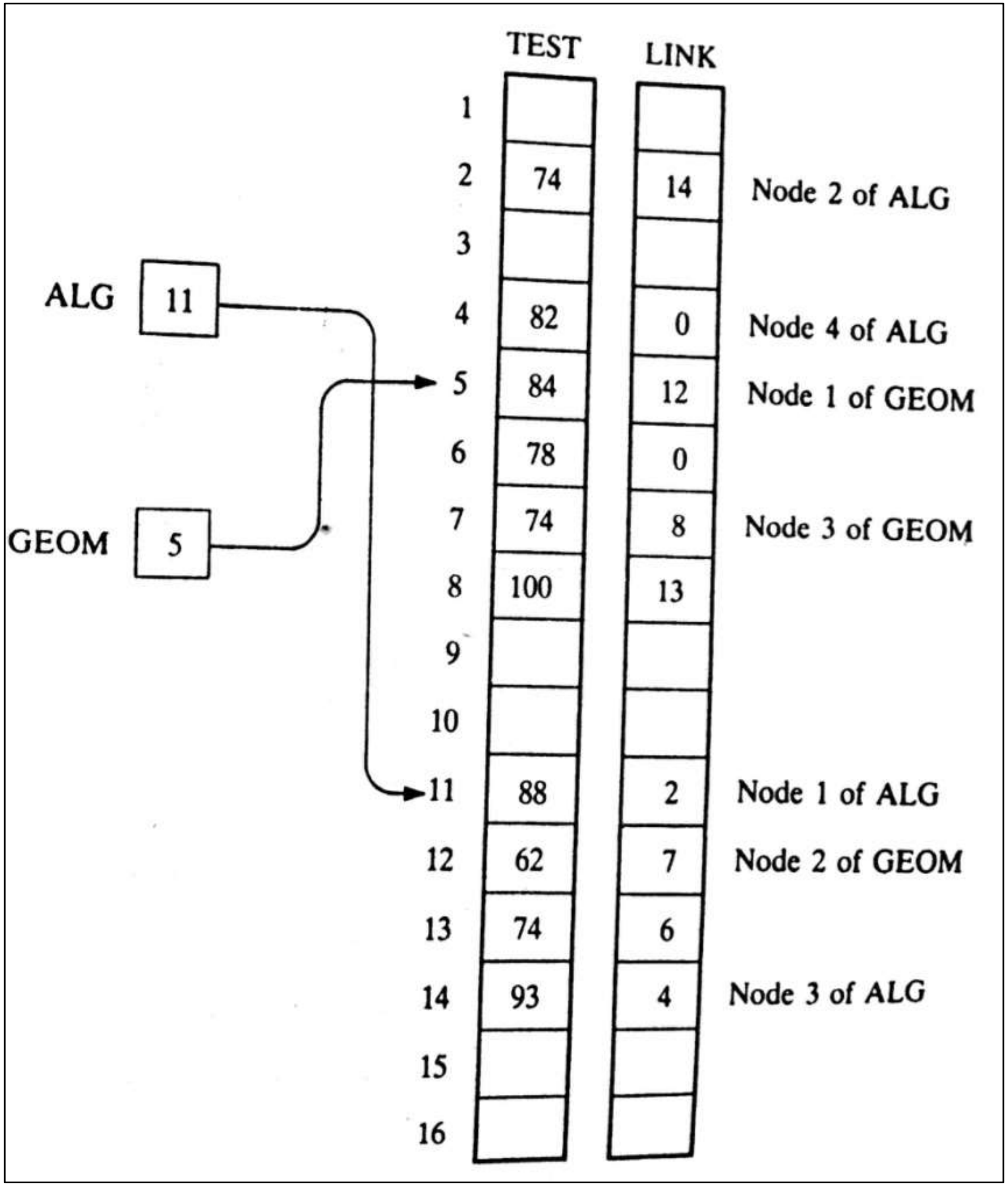
LINK[11]=7, so, INFO[7]= ? X

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			

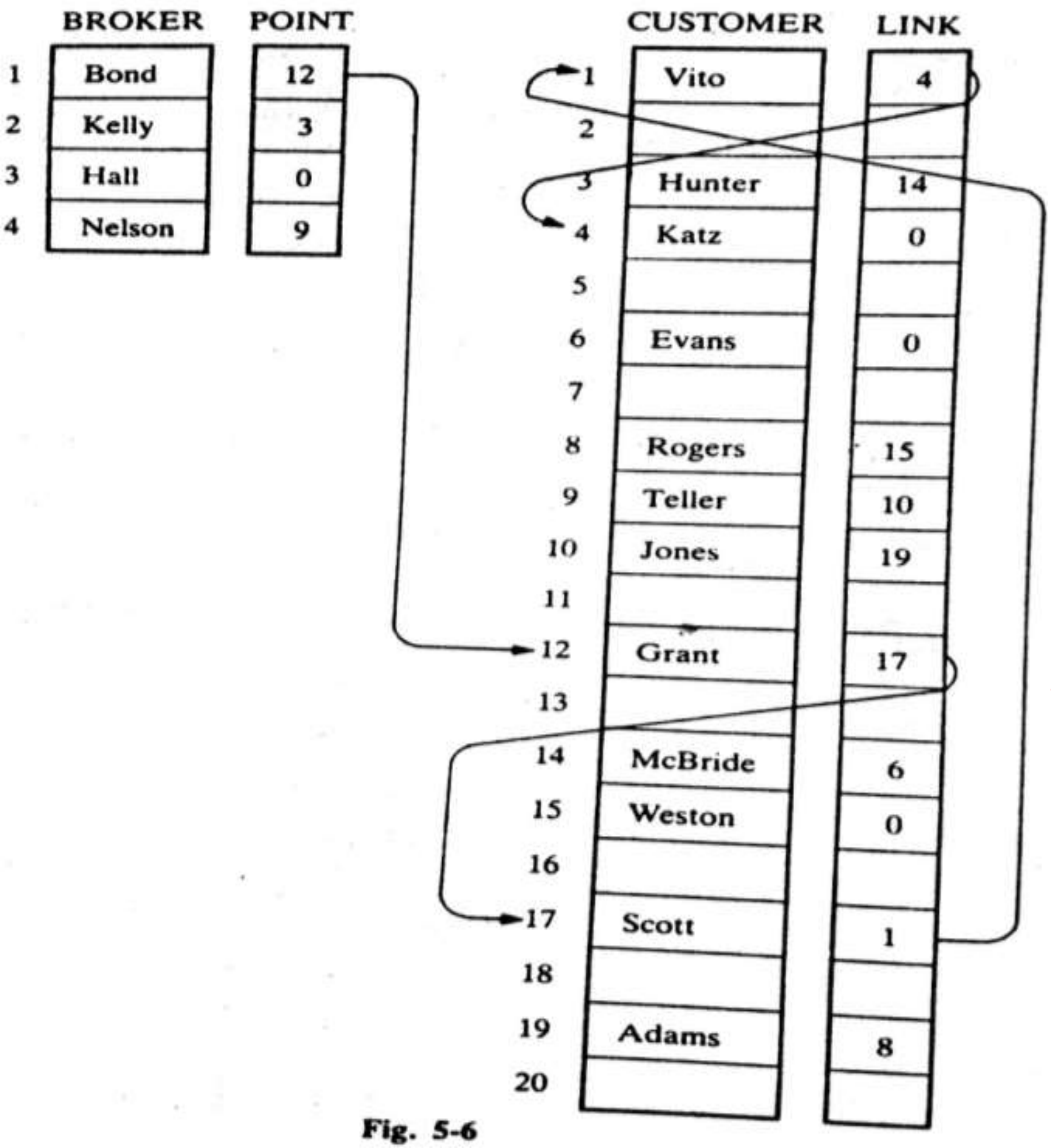
ALG consists of Test score:

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

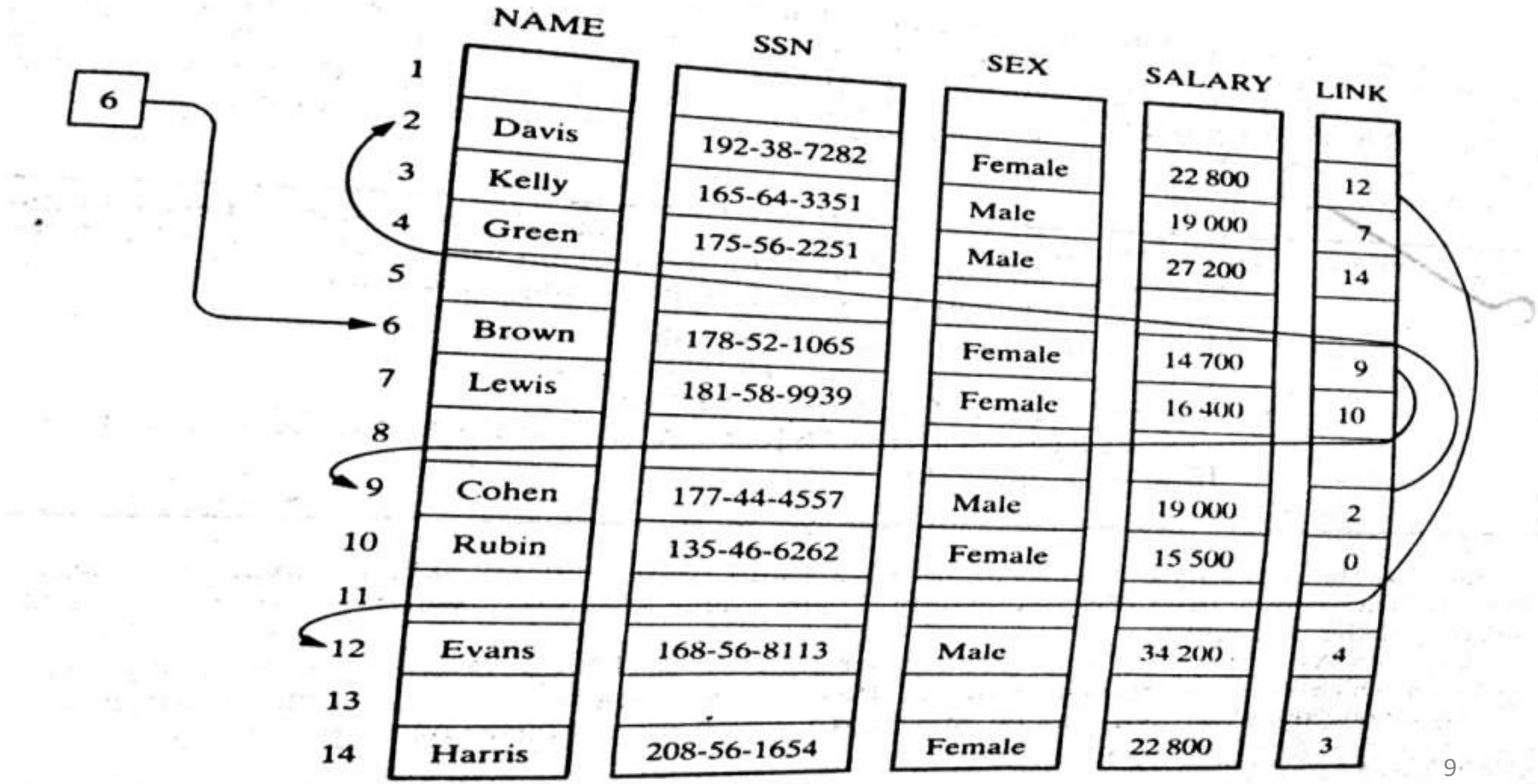
Kelly's list of customers:
Hunter, McBridge, Evans

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: **Types**

The following are the types of linked list:

- Singly Linked list
- Doubly Linked list
- Circular Linked list
- Doubly Circular Linked list

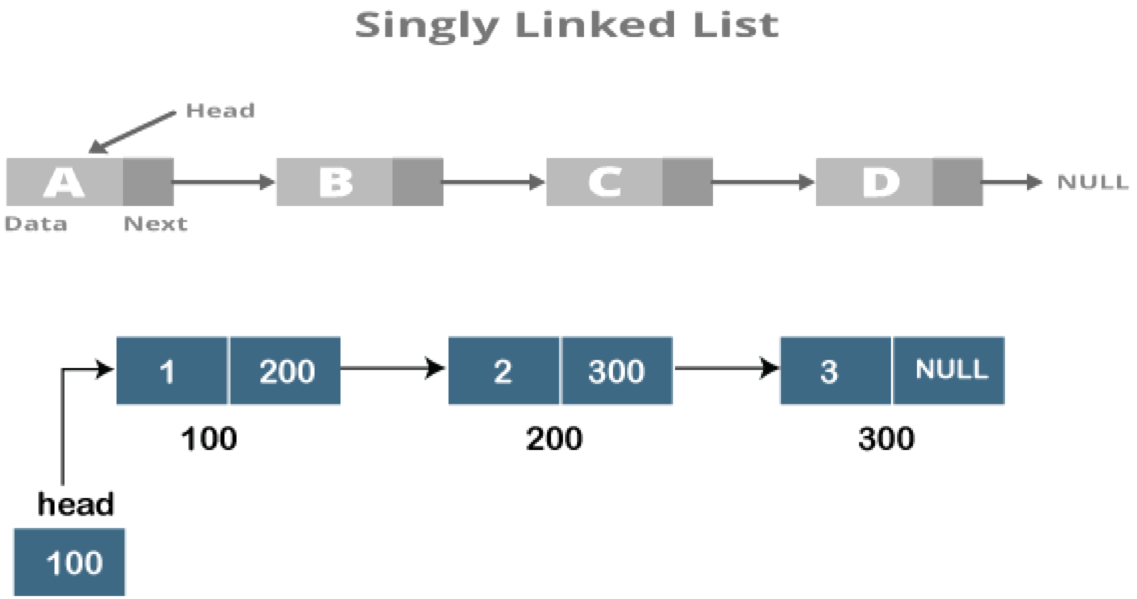
LINKED LISTS: Types

1. Singly Linked List

It is the simplest type of linked list in which every node contains some data and a pointer to the next node of the same data type.

The node contains a pointer to the next node means that the node stores the address of the next node in the sequence.

A single linked list allows the traversal of data only in one way. Below is the imag

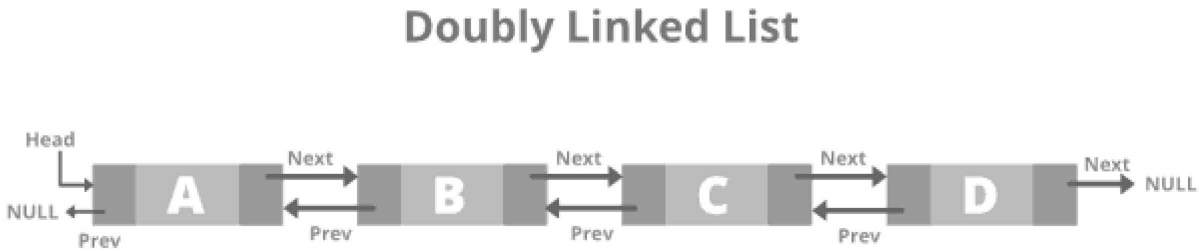


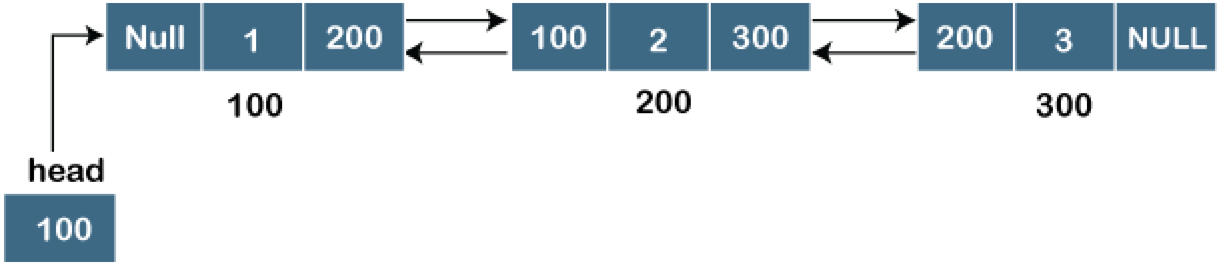
LINKED LISTS: Types

2. Doubly Linked list

A doubly linked list or a two-way linked list is a more complex type of linked list that contains a pointer to the next as well as the previous node in sequence.

Therefore, it contains three parts of data, a pointer to the next node, and a pointer to the previous node. This would enable us to traverse the list in the backward direction as well. Below is the image for the same:



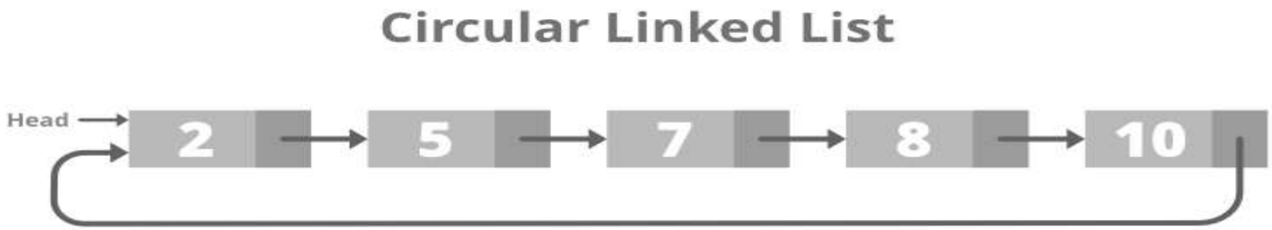


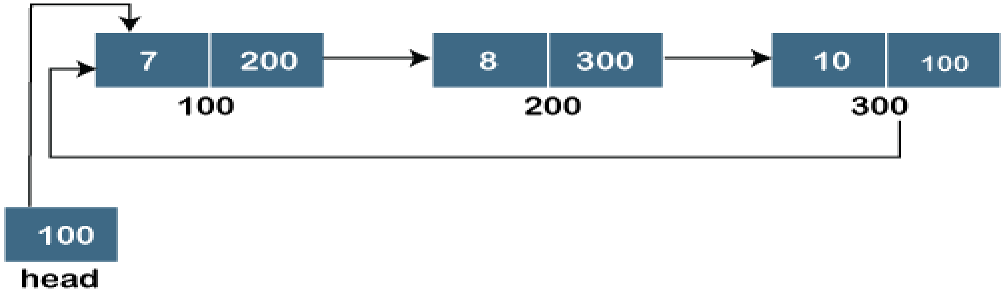
LINKED LISTS: Types

3. Circular Linked List

A circular linked list is that in which the last node contains the pointer to the first node of the list.

While traversing a circular linked list, we can begin at any node and traverse the list in any direction forward and backward until we reach the same node we started. Thus, a circular linked list has no beginning and no end. Below is the image for the same:

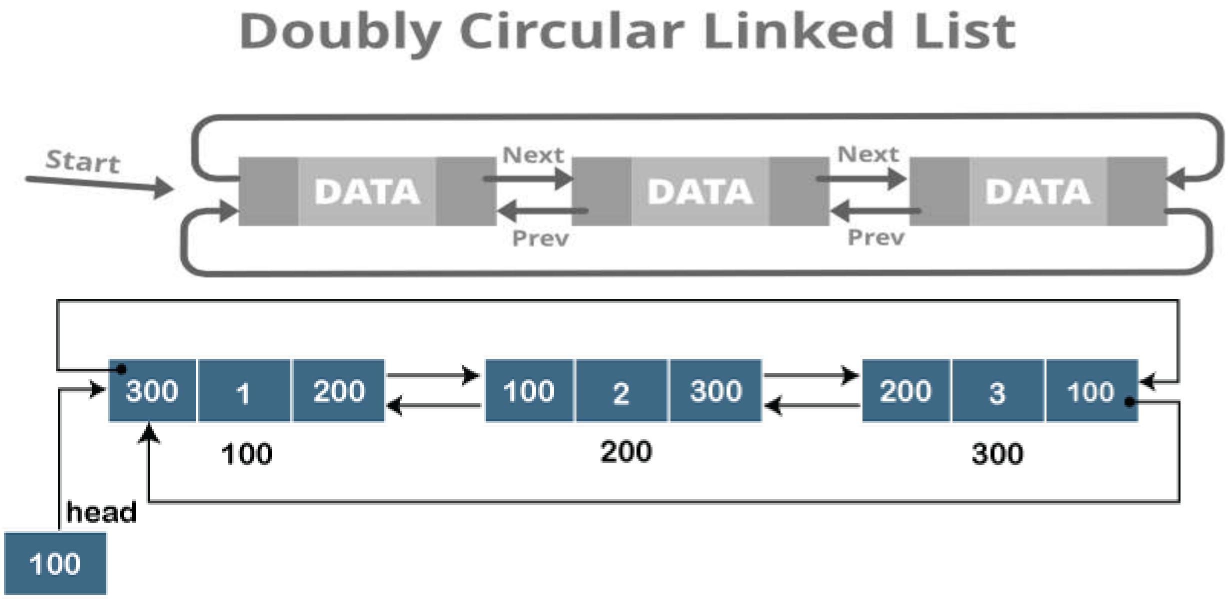




LINKED LISTS: Types

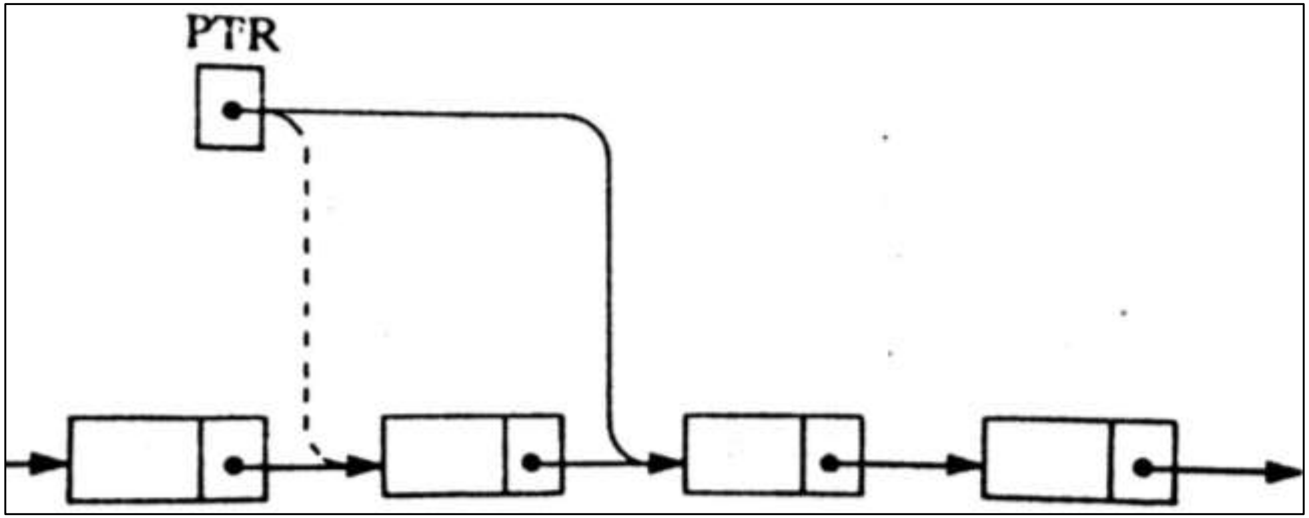
4. Doubly Circular linked list

A Doubly Circular linked list or a circular two-way linked list is a more complex type of linked list that contains a pointer to the next as well as the previous node in the sequence. The difference between the doubly linked and circular doubly list is the same as that between a singly linked list and a circular linked list. The circular doubly linked list does not contain null in the previous field of the first node. Below is the image for the same:



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 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.
```

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**

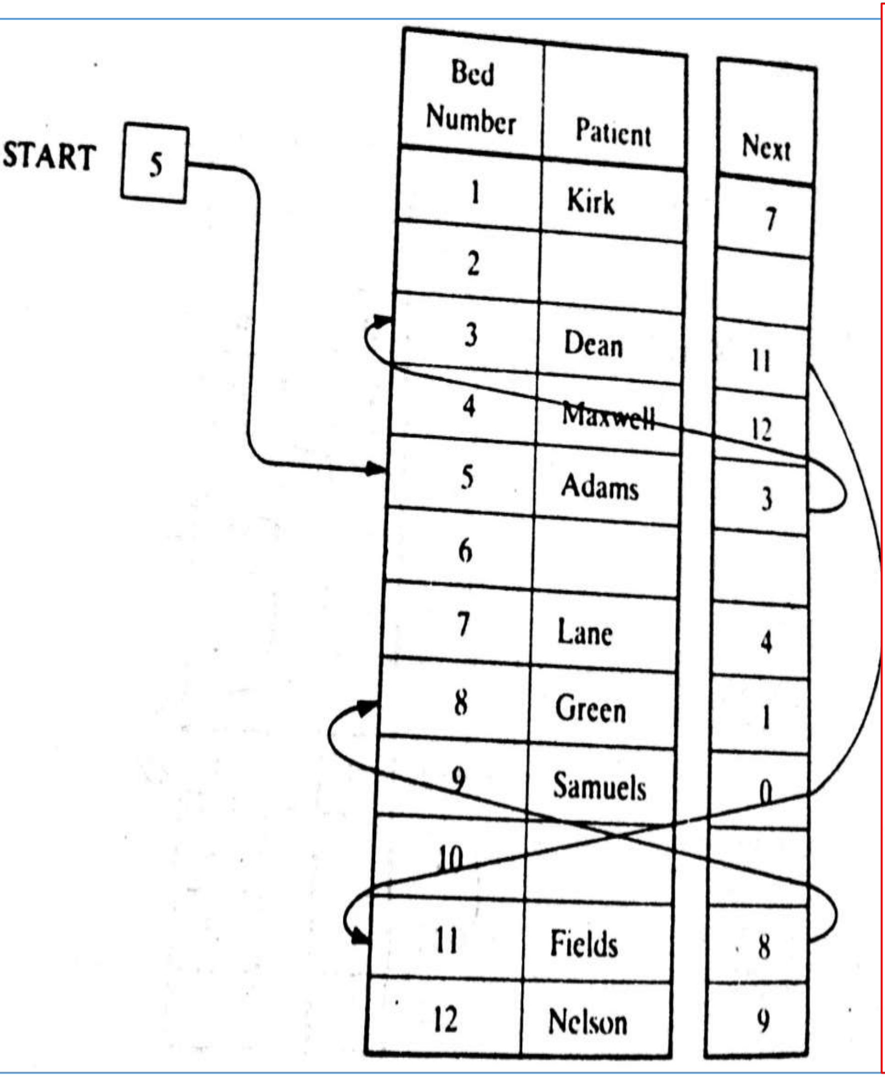
- ✓Together with the linked lists in memory, a special list is maintained which consists of unused memory cells.
- ✓This list, which has its own pointer, is called …
    - ✓***The list of available space*** or
    - ✓*the free-storage list* or
    - ✓*the free pool*.

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.

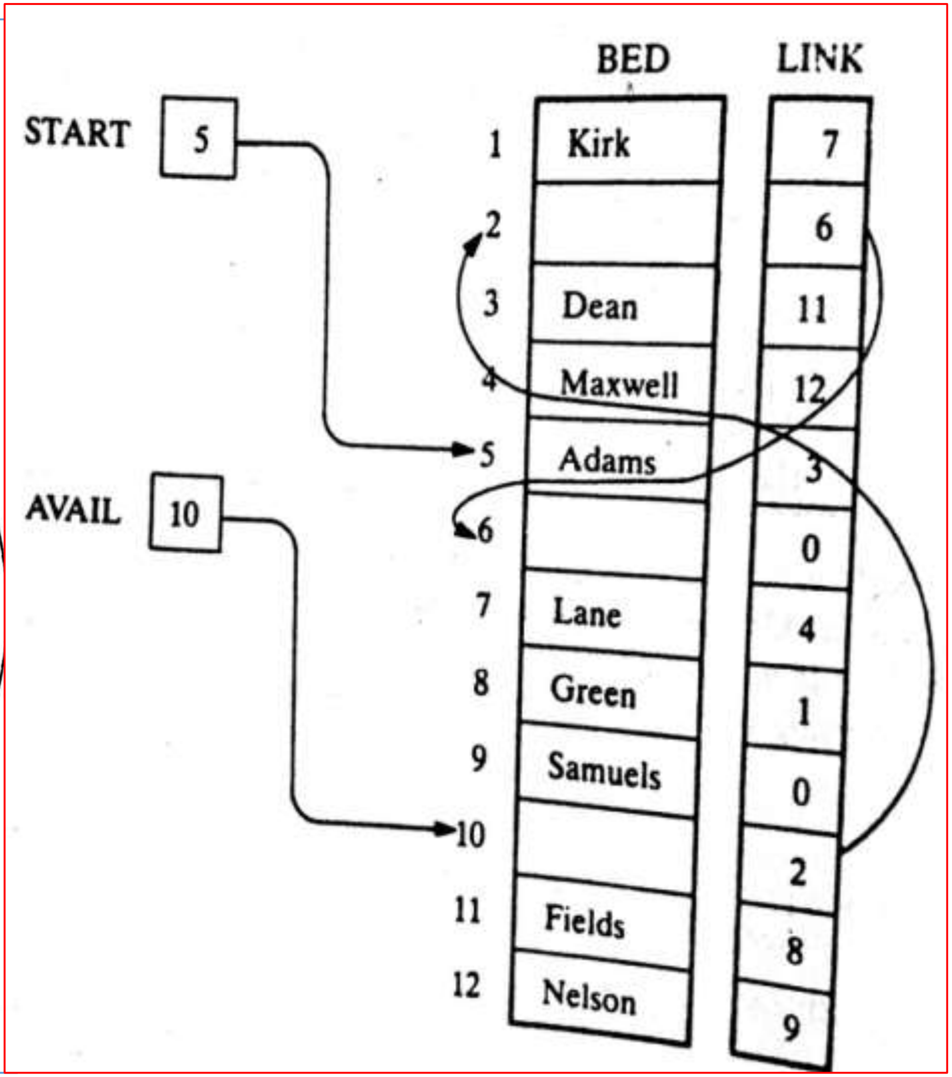
LIST(INFO, LINK, START, AVAIL)

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

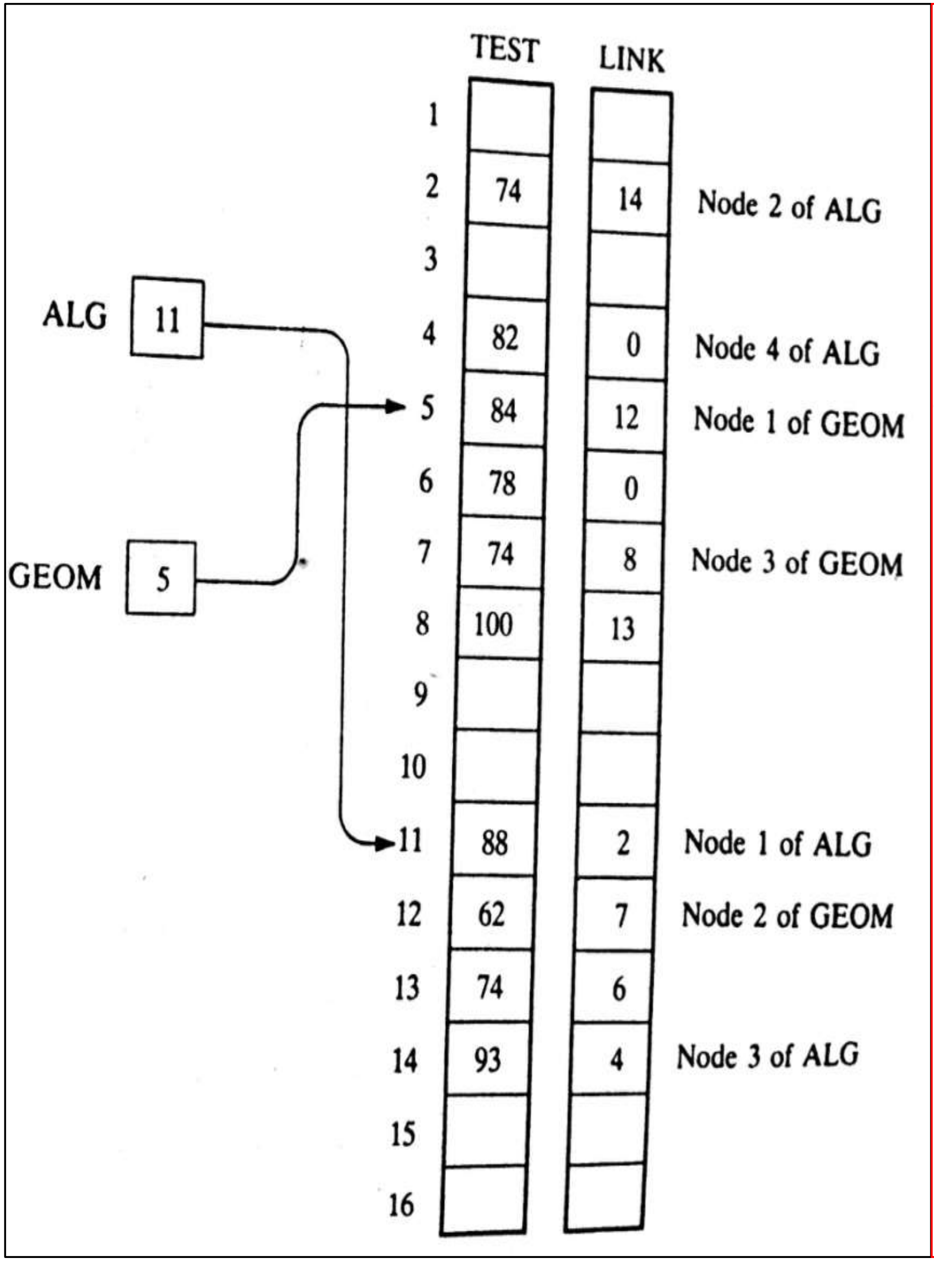


START 5

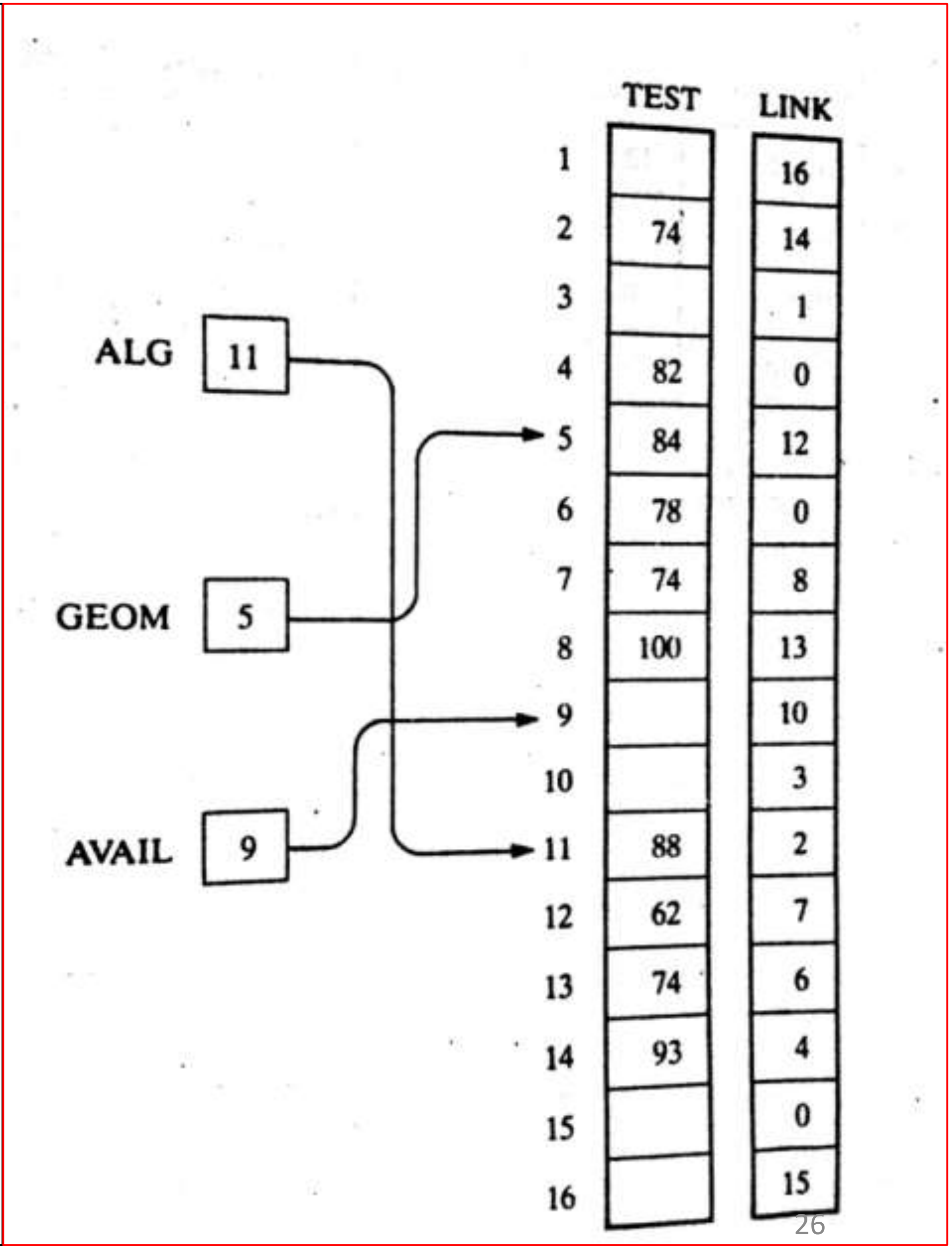
AVAIL 10

	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)

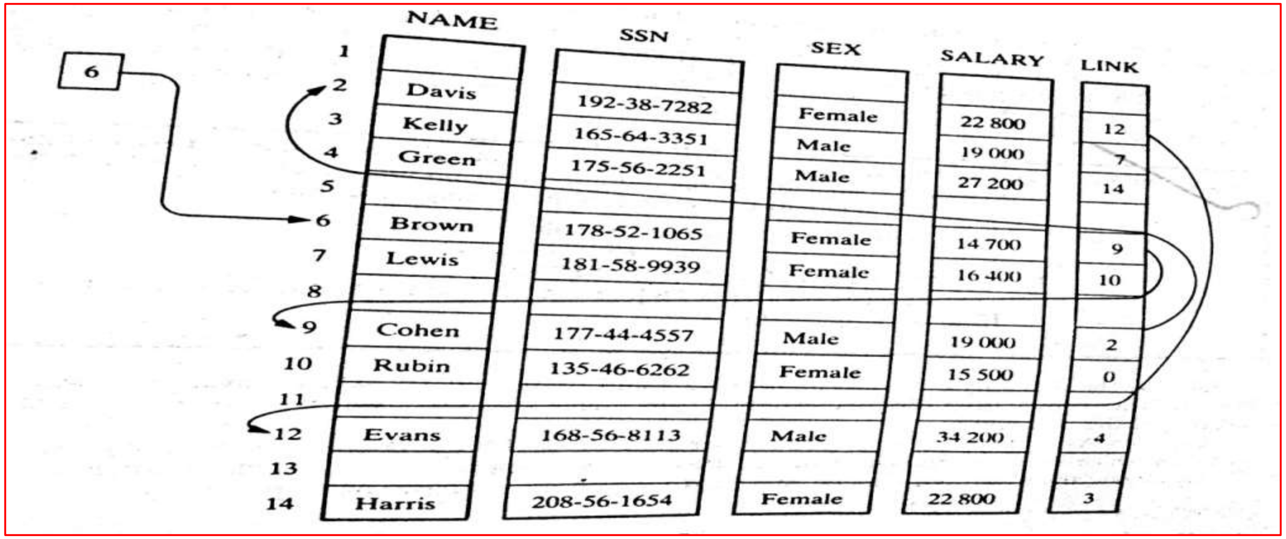


	TEST	LINK	
1			
2	74	14	Node 2 of ALG
3			
4	82	0	Node 4 of ALG
5	84	12	Node 1 of GEOM
6	78	0	
7	74	8	Node 3 of GEOM
8	100	13	
9			
10			
11	88	2	Node 1 of ALG
12	62	7	Node 2 of GEOM
13	74	6	
14	93	4	Node 3 of ALG
15			
16			

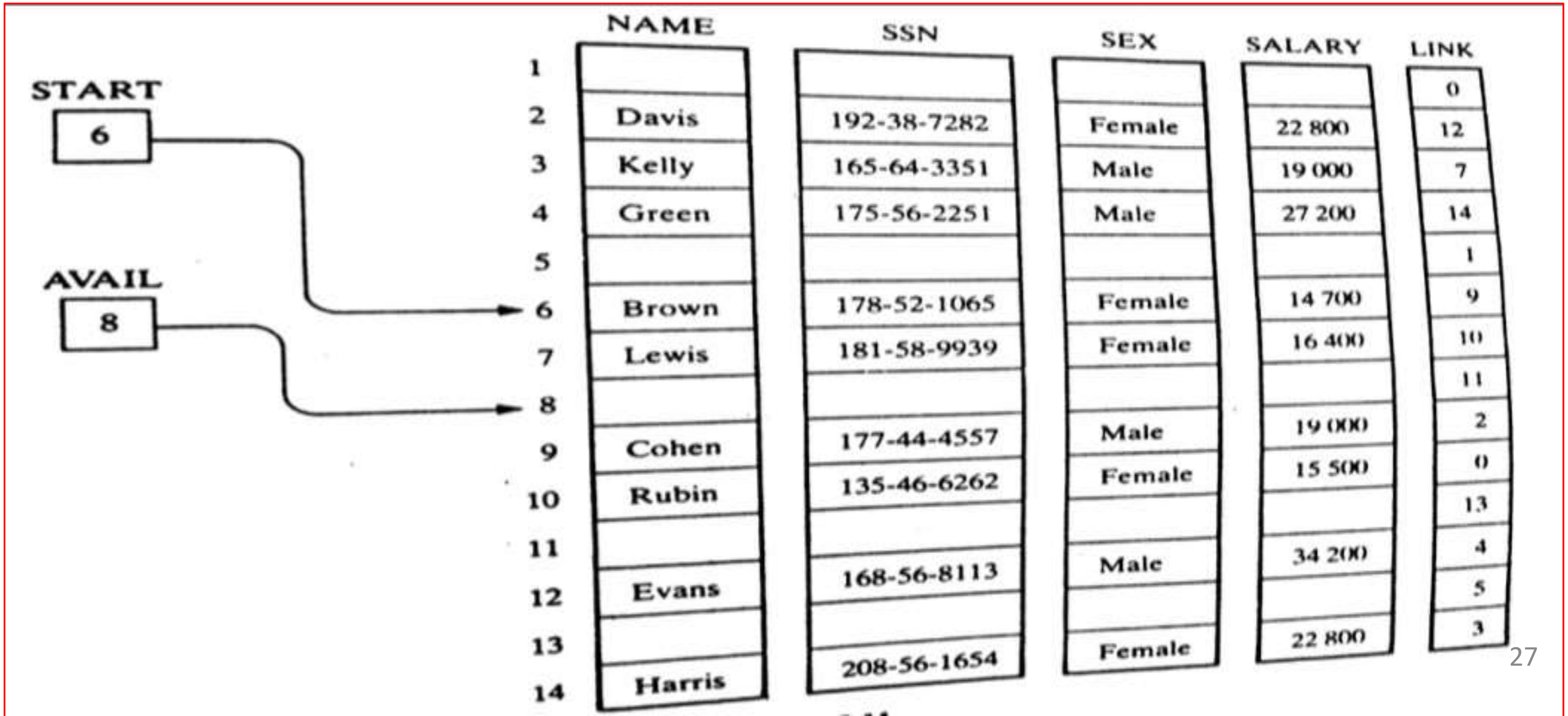


	TEST	LINK
1		16
2	74	14
3		1
4	82	0
5	84	12
6	78	0
7	74	8
8	100	13
9		10
10		3
11	88	2
12	62	7
13	74	6
14	93	4
15		0
16		15

LINKED LISTS: Example 5.11(b)

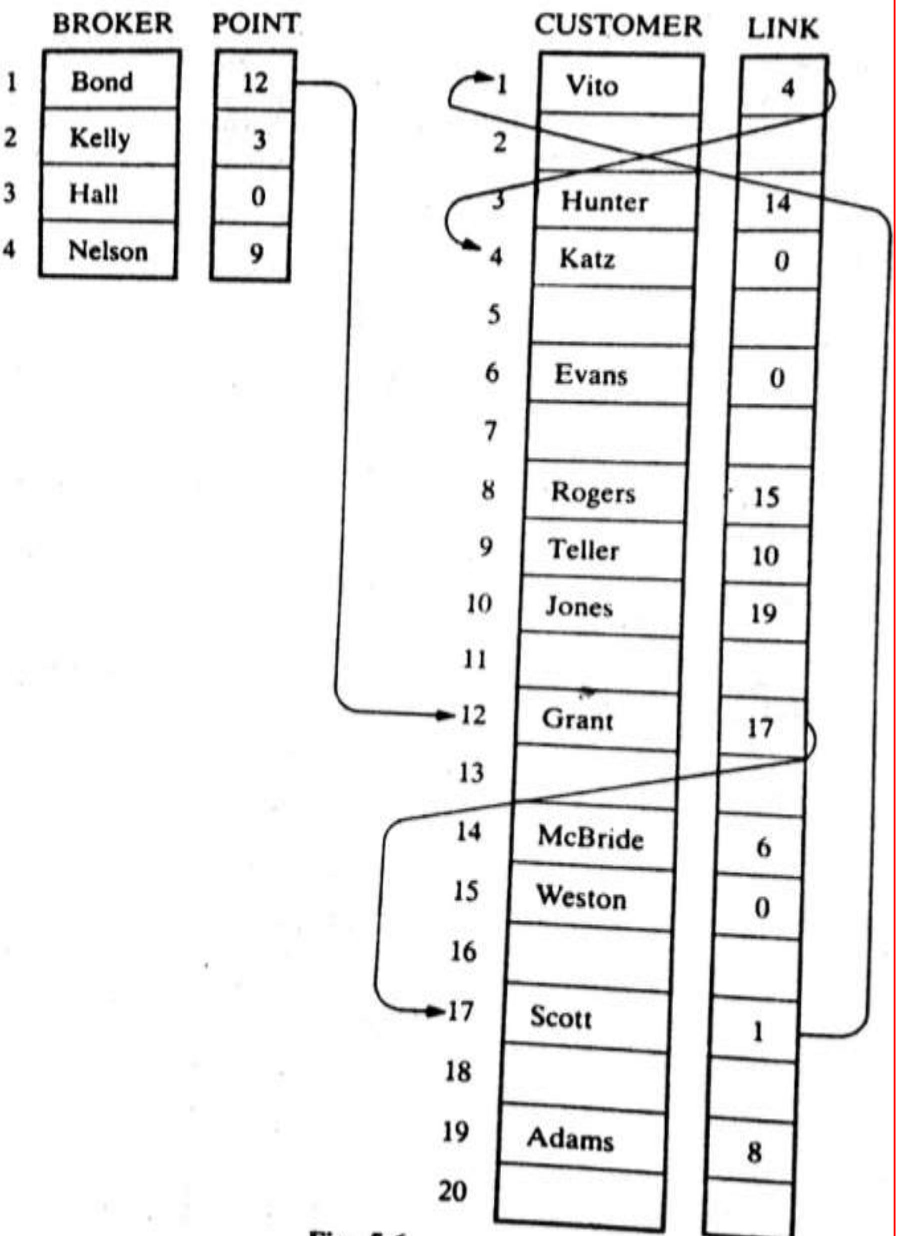


	NAME	SSN	SEX	SALARY	LINK
1					
2	Davis	192-38-7282	Female	22 800	12
3	Kelly	165-64-3351	Male	19 000	7
4	Green	175-56-2251	Male	27 200	14
5					
6	Brown	178-52-1065	Female	14 700	9
7	Lewis	181-58-9939	Female	16 400	10
8					
9	Cohen	177-44-4557	Male	19 000	2
10	Rubin	135-46-6262	Female	15 500	0
11		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

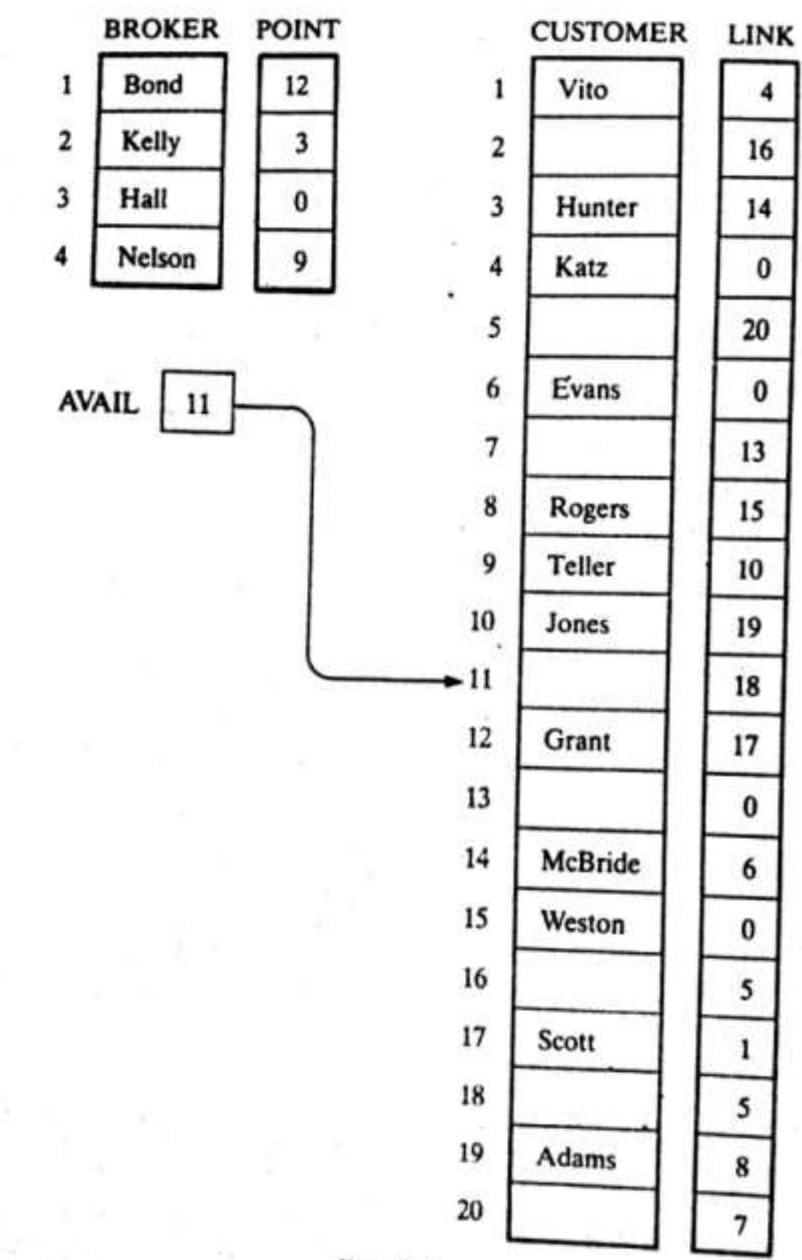
LINKED LISTS: Example 5.11(c)



	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

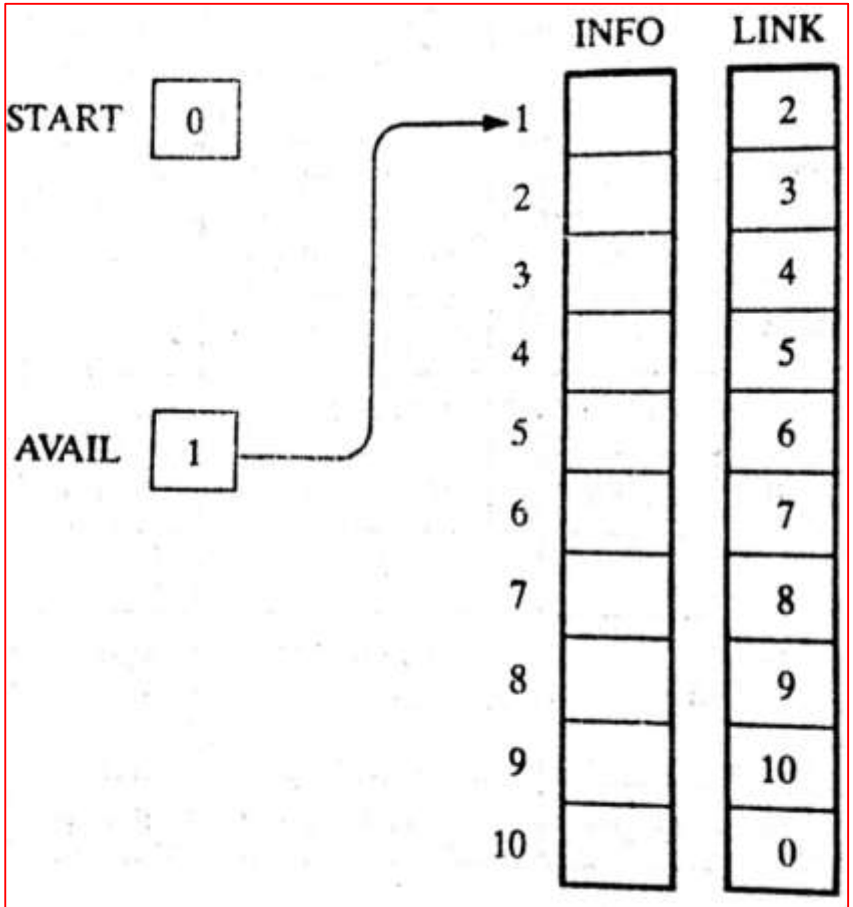
Fig. 5-12

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



	INFO	LINK
1		2
2		3
3		4
4		5
5		6
6		7
7		8
8		9
9		10
10		0

LINKED LISTS: Garbage Collection

Time Consuming Method for OS:

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 no 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.

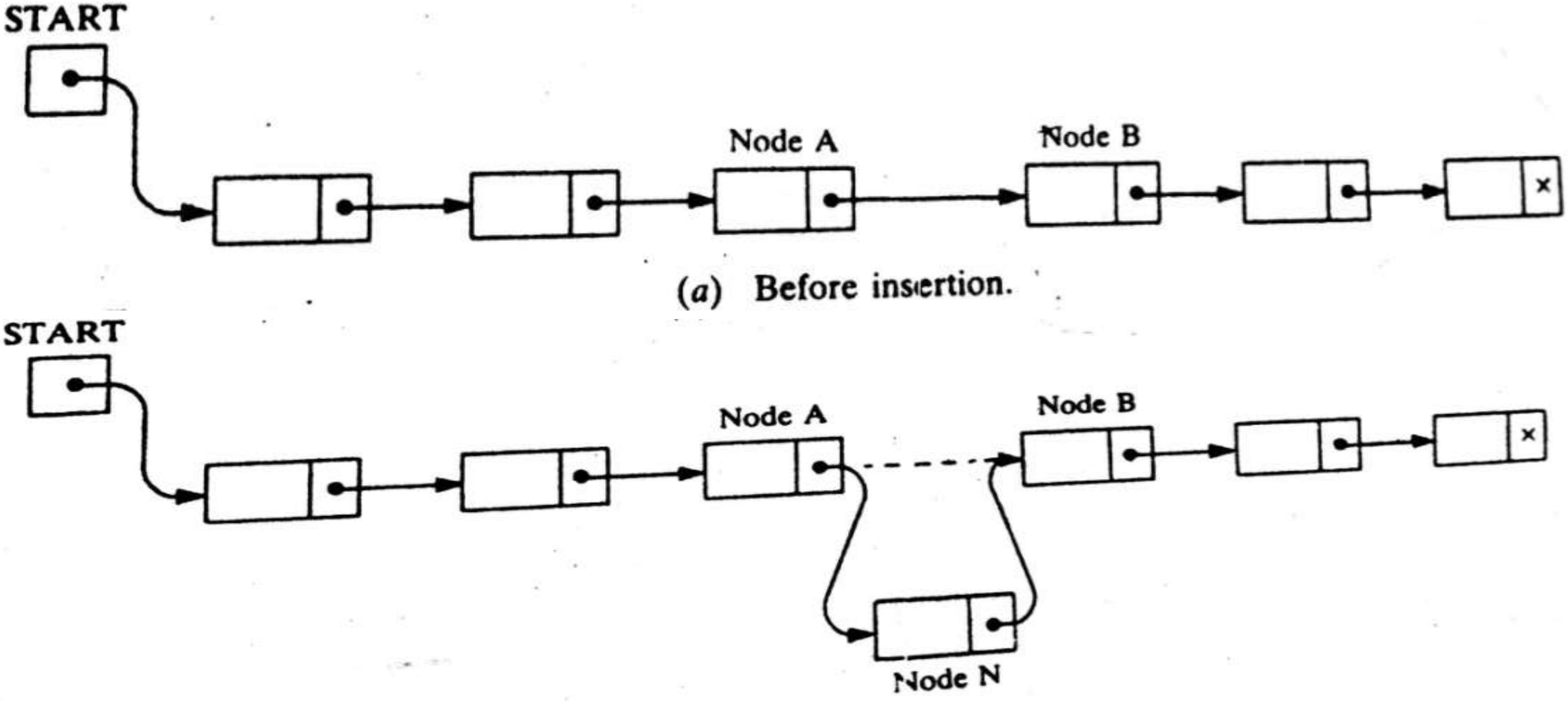
How we can handle Underflow situation?

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.

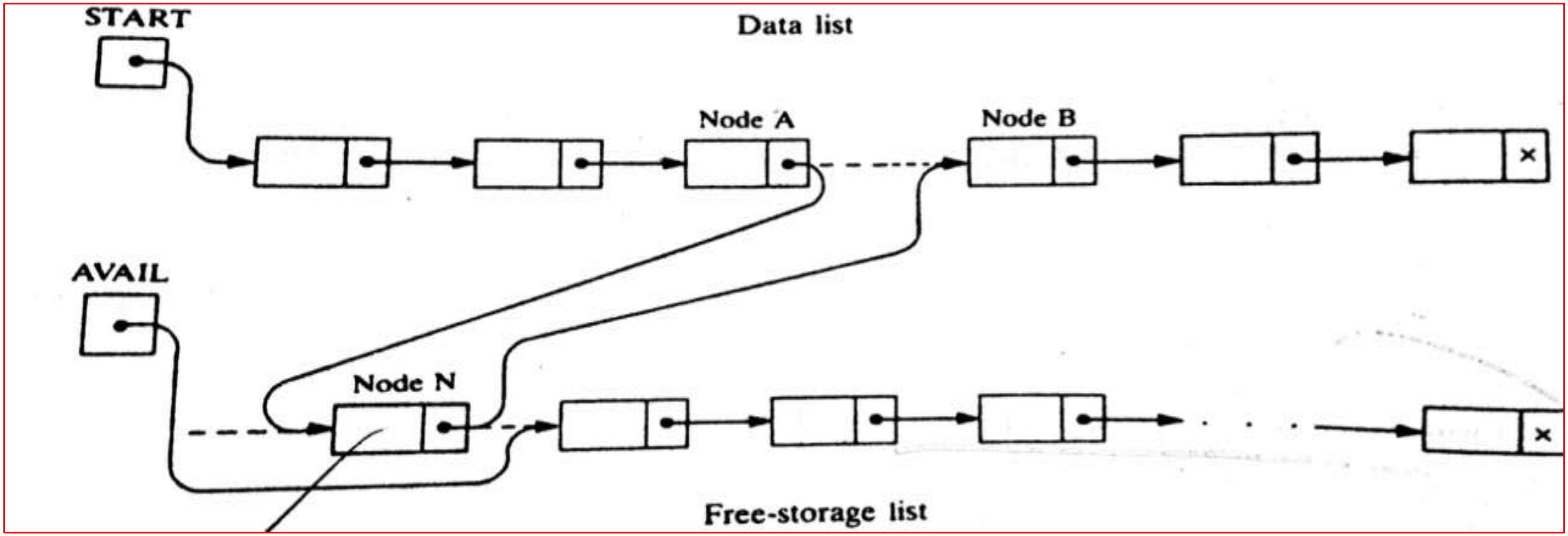
(*b*) After insertion.

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

- ✓ 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:

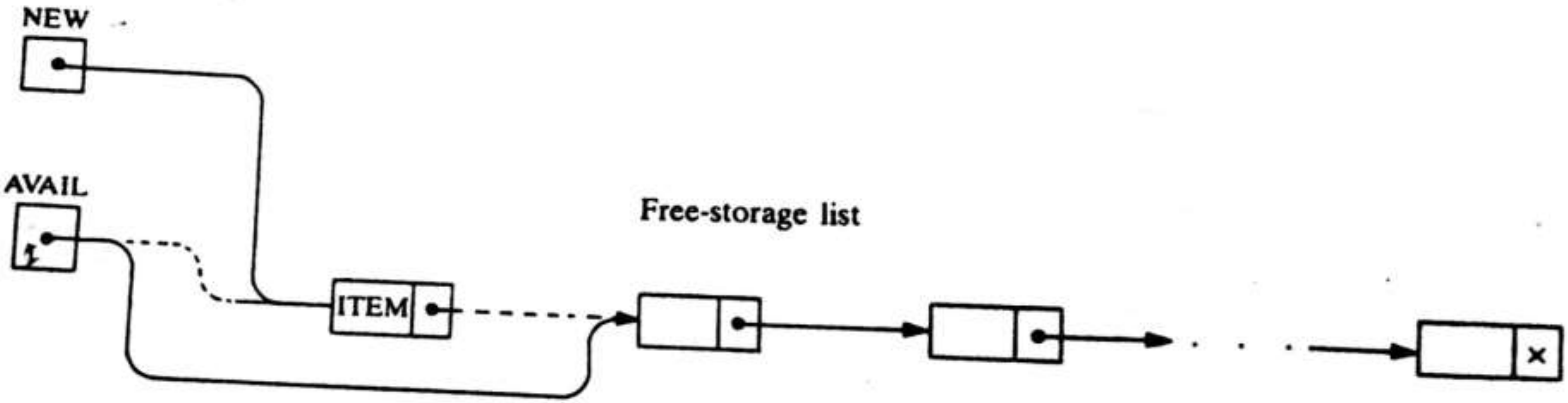
- Checking to see if space is available in the AVAIL list. If not, that is,
- If **AVAIL=NULL**, then the algorithm will print the message

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

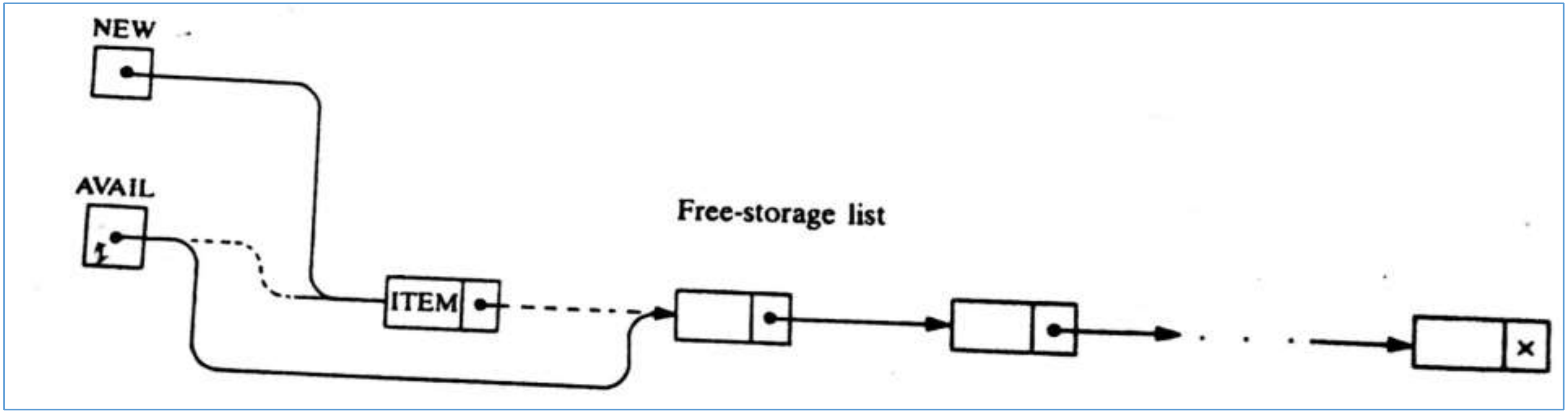
  NEW:=AVAIL, AVAIL:=LINK[AVAIL]
- Copying new information into the new node. i.e.,

  INFO[NEW]=ITEM

LINKED LISTS: Insertion Algorithms



LINKED LISTS: Inserting at the Beginning of a list



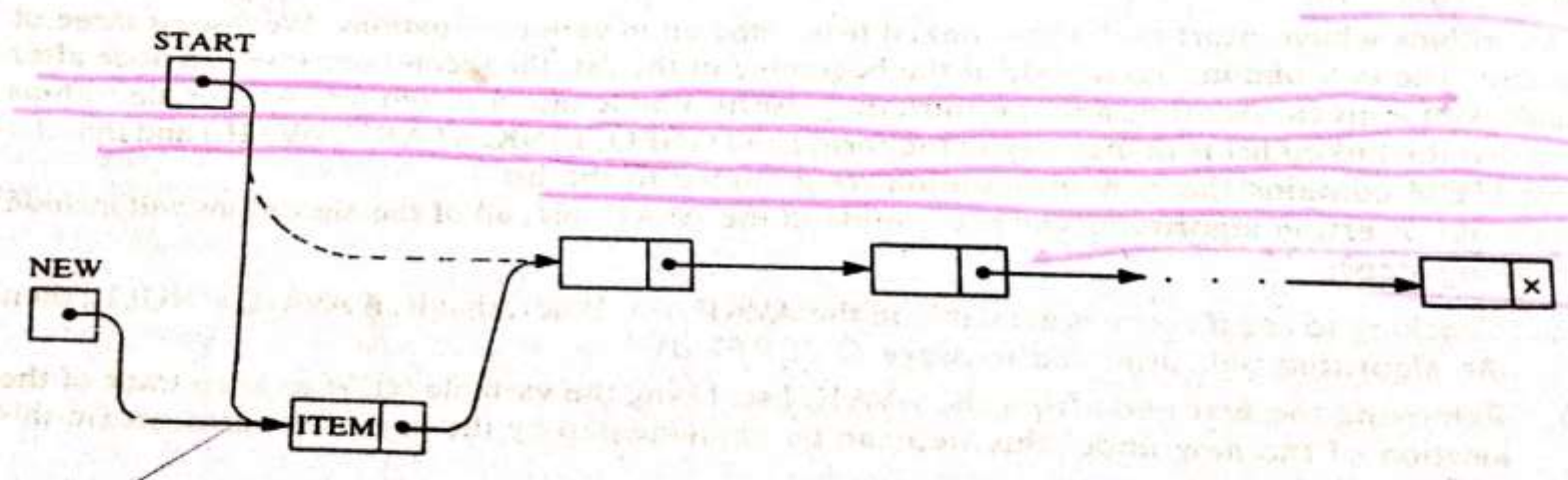


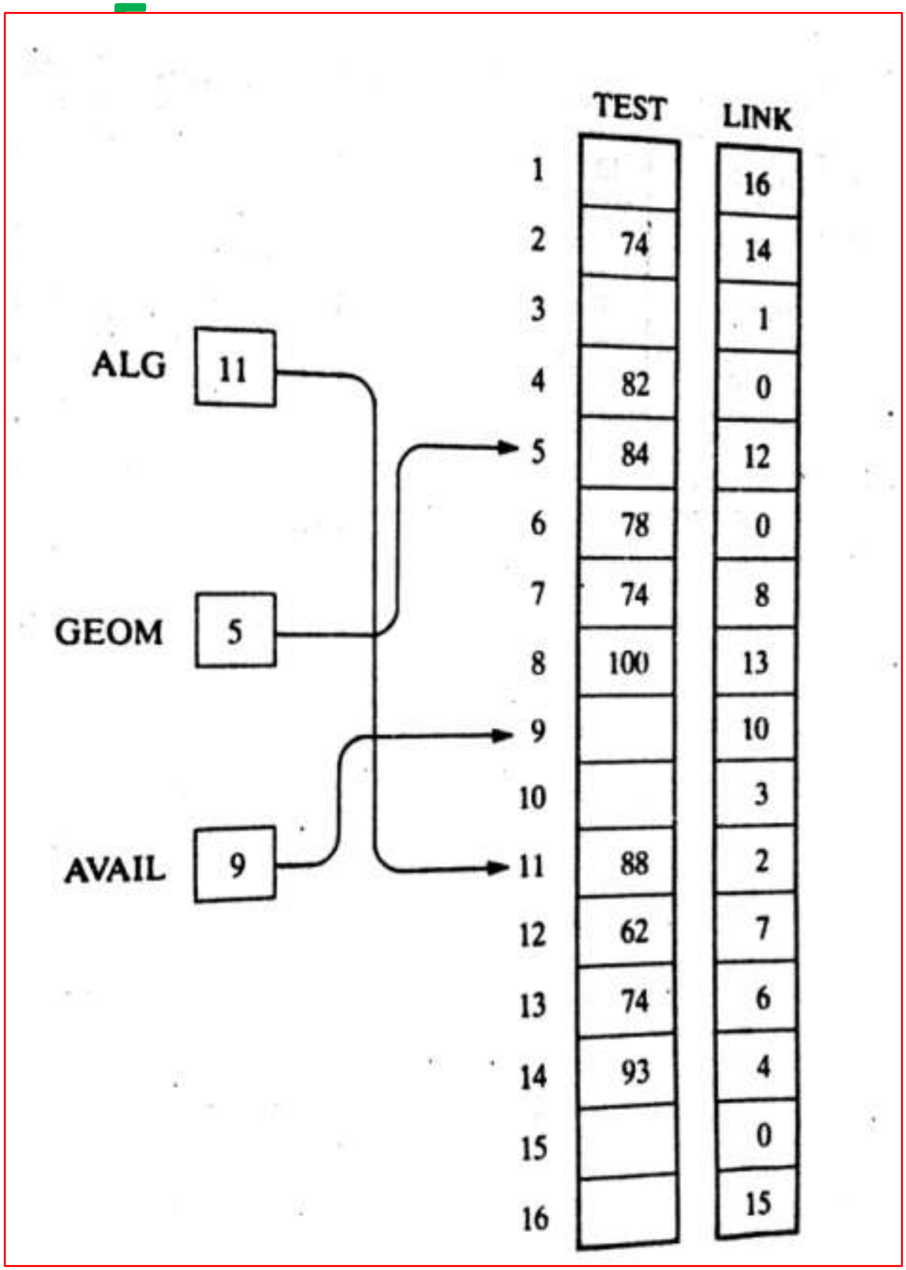
Fig. 5-18 Insertion at the beginning of a list.

LINKED LISTS: Inserting at the Beginning ….ALG

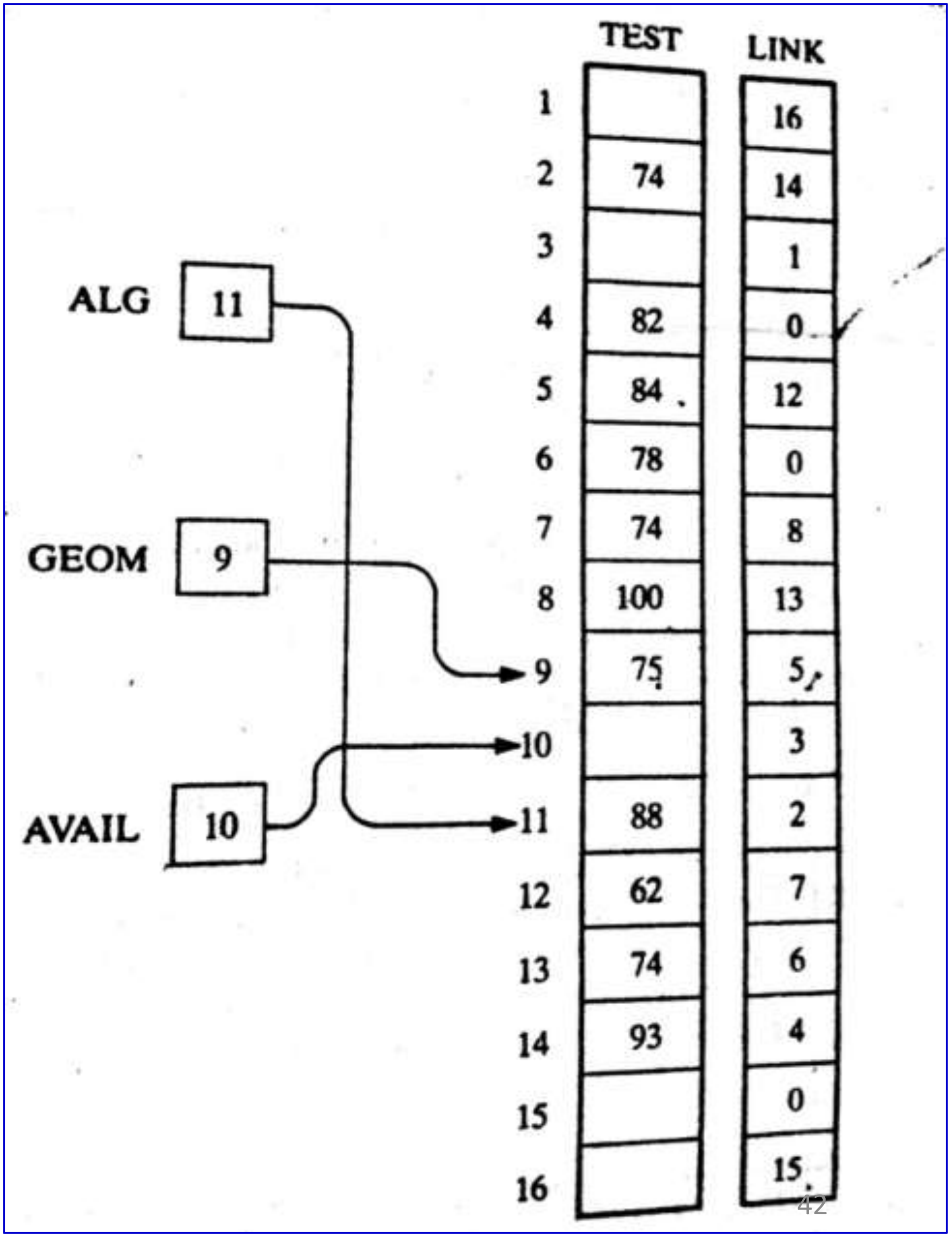
INSFIRST (INFO, LINK, START, AVAIL,ITEM)

```
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



	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

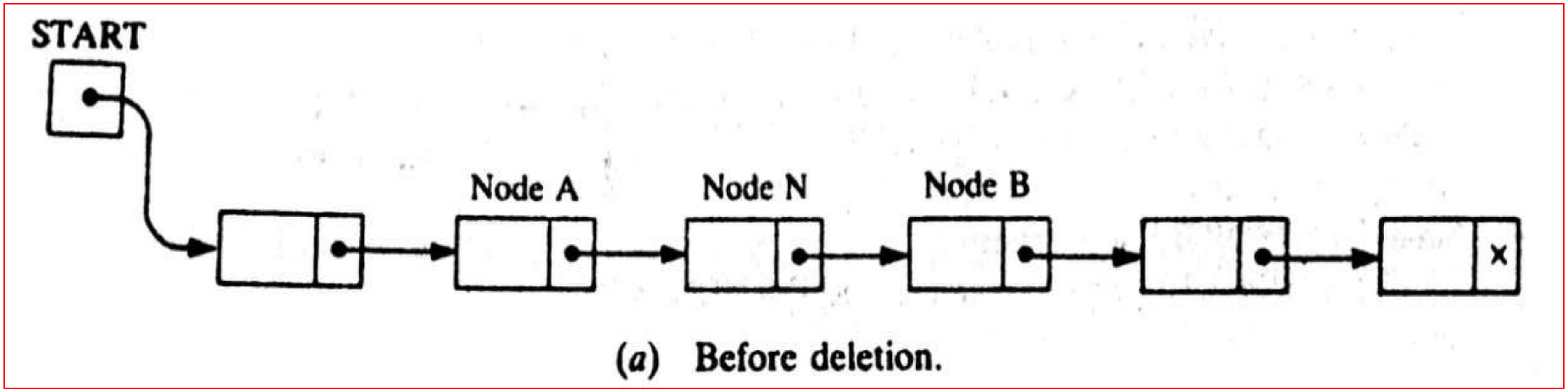
LINKED LISTS: Inserting after a given node

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

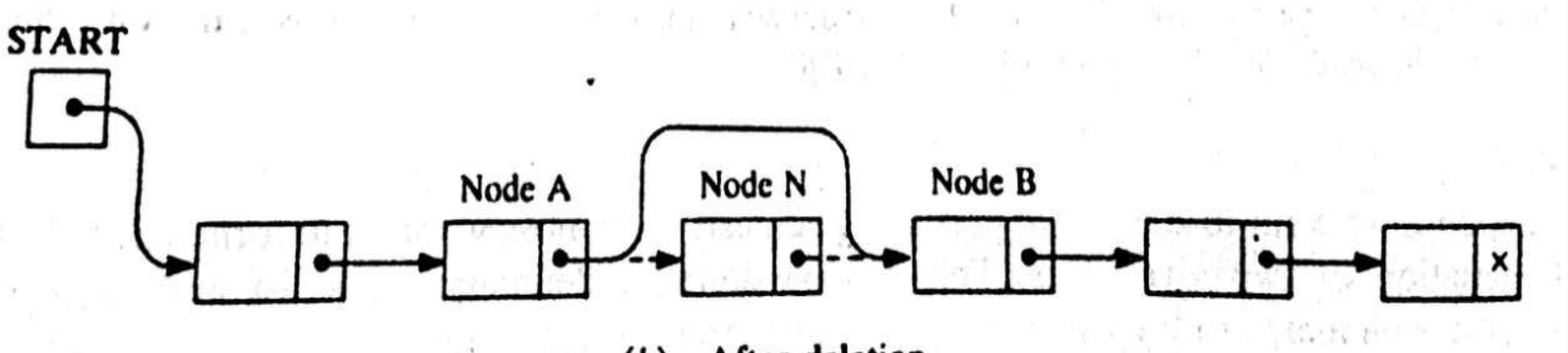
```
1.  [OVERFLOW?] If AVAIL=NULL, then Write: OVERFLOW, and Exit.
2.  [Remove first node from AVAIL list.]
     Set NEW:=AVAIL and AVAIL:=LINK[AVAIL]
3.  Set INFO[NEW]:=ITEM.[Copies new data into new node.]
4.  If LOC=NULL, then [Insert as first node.]
          Set LINK[NEW]:=START and START:=NEW.
    Else: [Insert after node with location LOC.]
          Set LINK[NEW]:=LINK[LOC] and LINK[LOC]:=NEW.
    [END if If structure.]
5.  Exit.
```

LINKED LISTS: Deletion from a Linked List

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



(*a*) Before deletion.

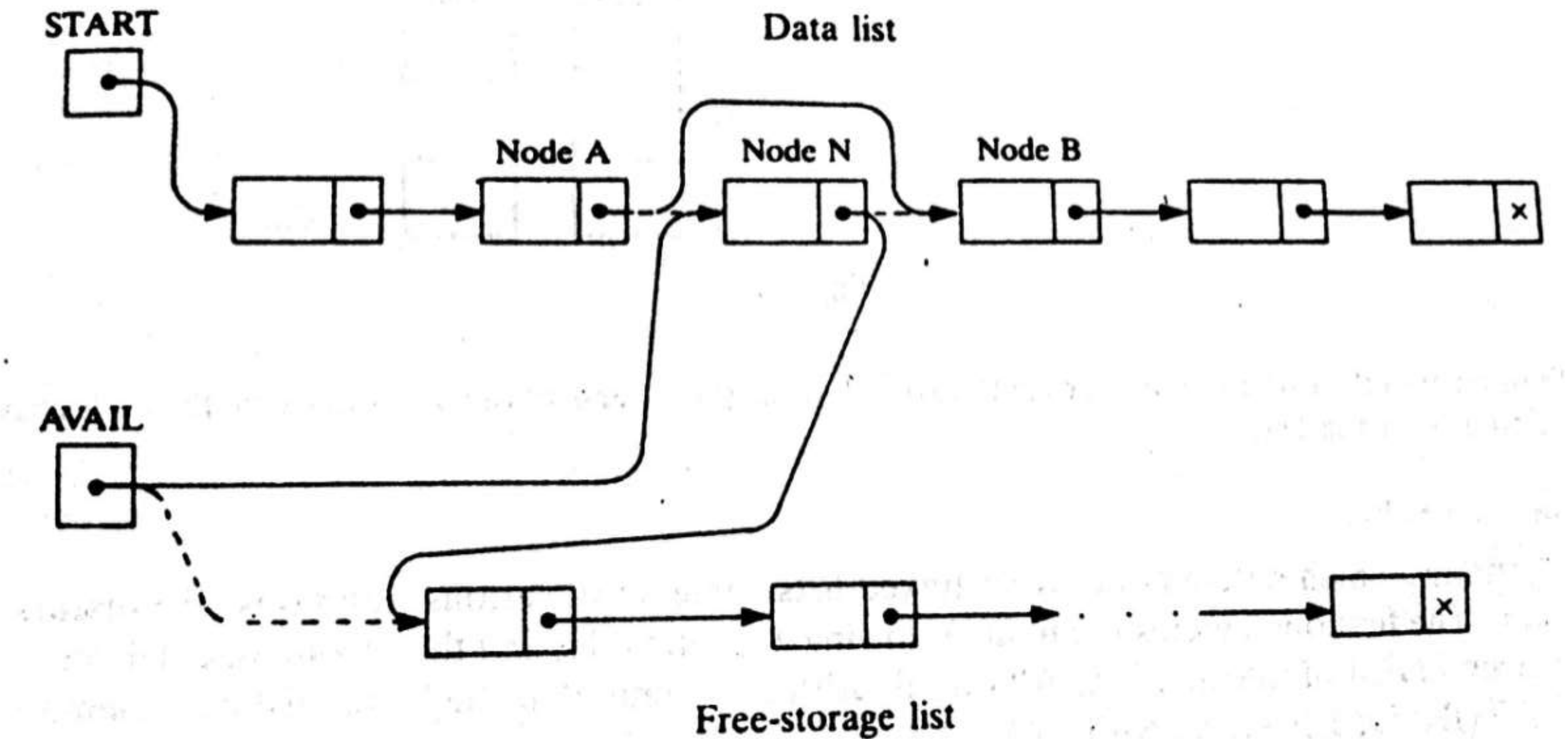


(*b*) After deletion.

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:

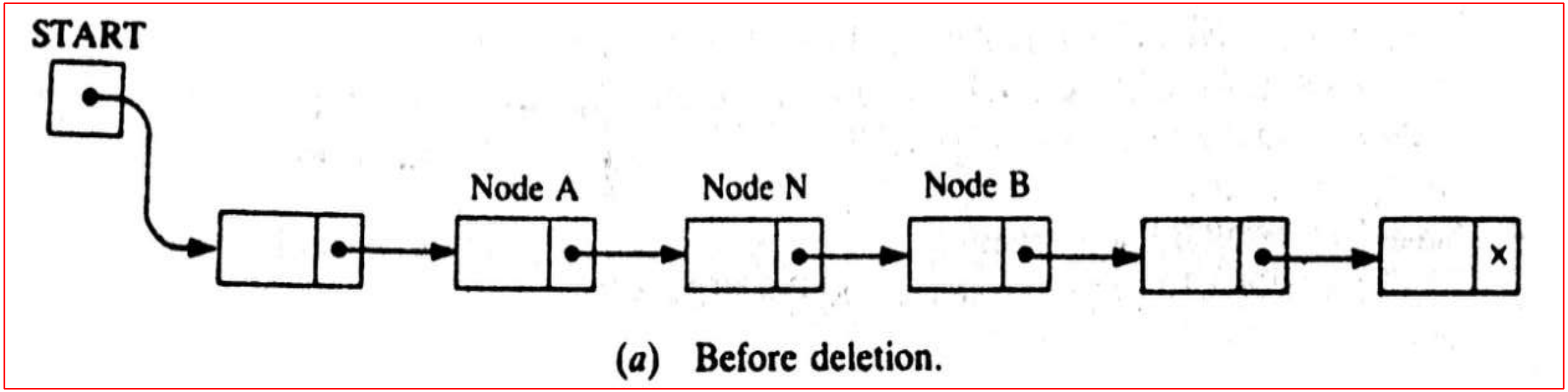


Three pointer fields are changed:

- 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.

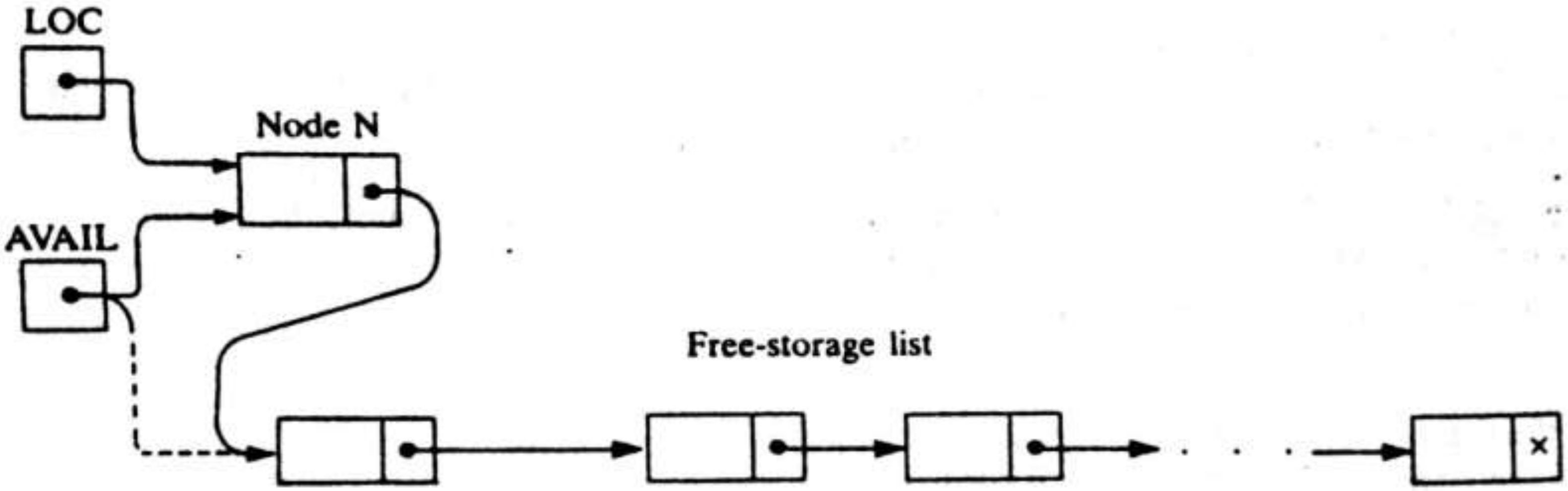


Fig. 5-25 LINK[LOC] := AVAIL and AVAIL := LOC.

LINKED LISTS:

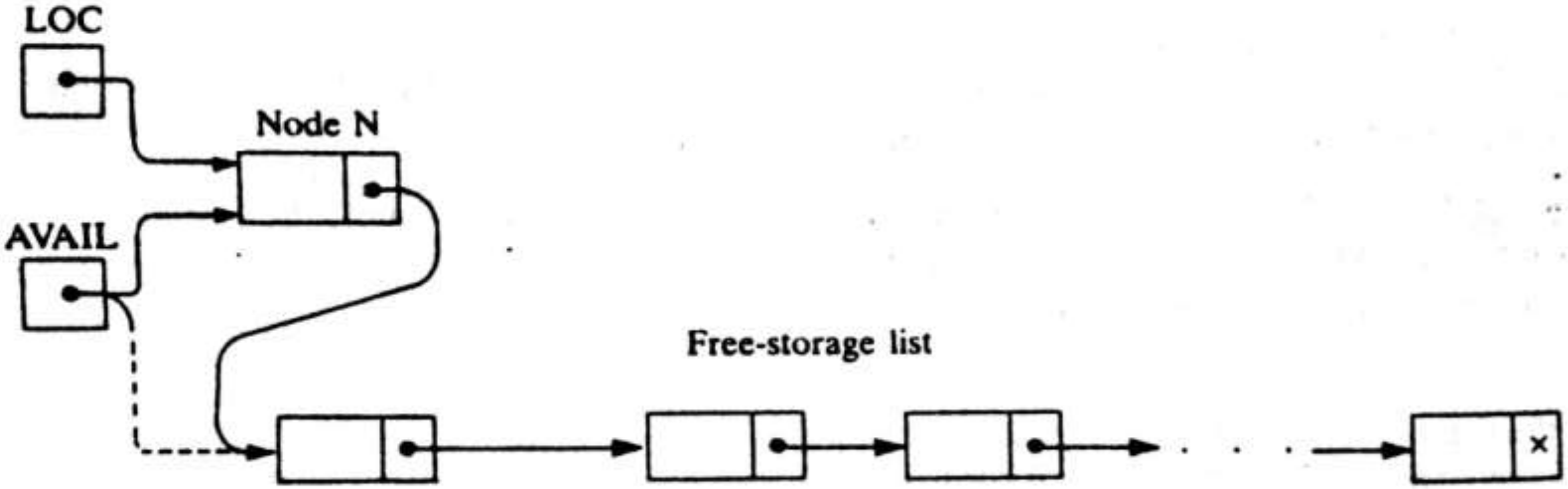


Fig. 5-25 LINK[LOC] := AVAIL and AVAIL := LOC.

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 Node

```
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.
```

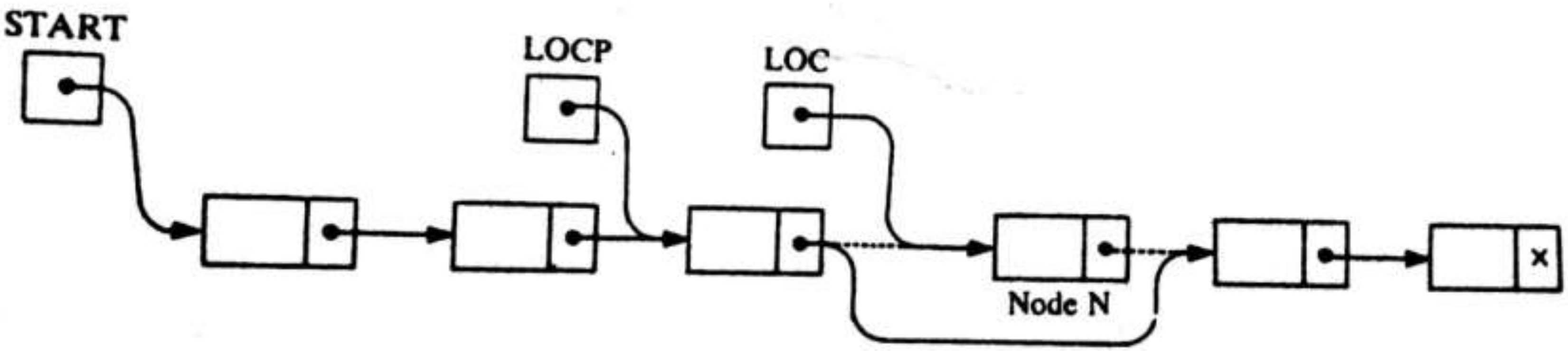


Fig. 5-27 LINK[LOCP]:= LINK[LOC].

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]