Linear List

- Elements in linear lists are in order by position (ordered set)
- Implemented using arrays
  - Linear relationships between elements of an array is reflected by physical relationships in memory
  - Advantage
    - address of an element can be calculated easily
  - Disadvantage
    - complex insertion and deletion
    - Array size is fixed (array size cannot be increased easily as it is stored in contiguous memory). For this reason arrays are called dense list or static data structure

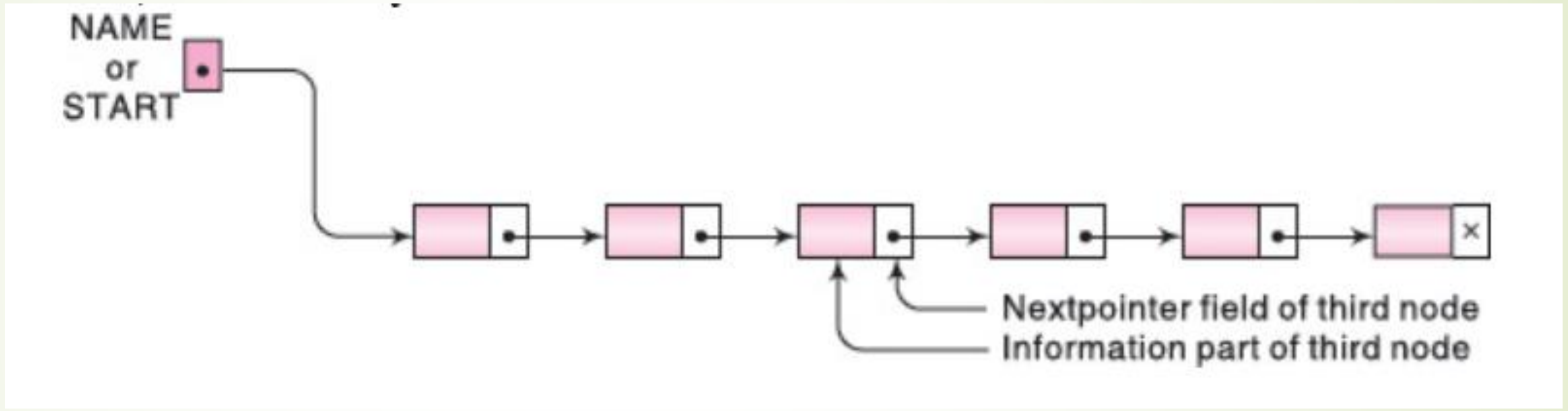
Linked Lists

- Another possible implementation of linear Lists
- List element are stored non contiguously in memory and special field is stored with each element (called link or pointer) that contain the address of the next element in the list
- Linear order is provided by means of pointers
- Each element is a node which consists of two parts
  - First part: value of element
  - Second part: pointer to (address of) the next node

Arrays Vs Linked Lists

Arrays	**Linked list**
Fixed size: Resizing is expensive	Dynamic size
Insertions and Deletions are inefficient: Elements are usually shifted	Insertions and Deletions are efficient: No shifting
Random access i.e., efficient indexing	No random access → Not suitable for operations requiring accessing elements by index such as sorting
No memory waste if the array is full or almost full; otherwise may result in much memory waste.	Since memory is allocated dynamically(acc. to our need) there is no waste of memory.
Sequential access is faster [Reason: Elements in contiguous memory locations]	Sequential access is slow [Reason: Elements not in contiguous memory locations]
Binary search can be used.	Binary search can not be used.

Schematic diagram of Linked List


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

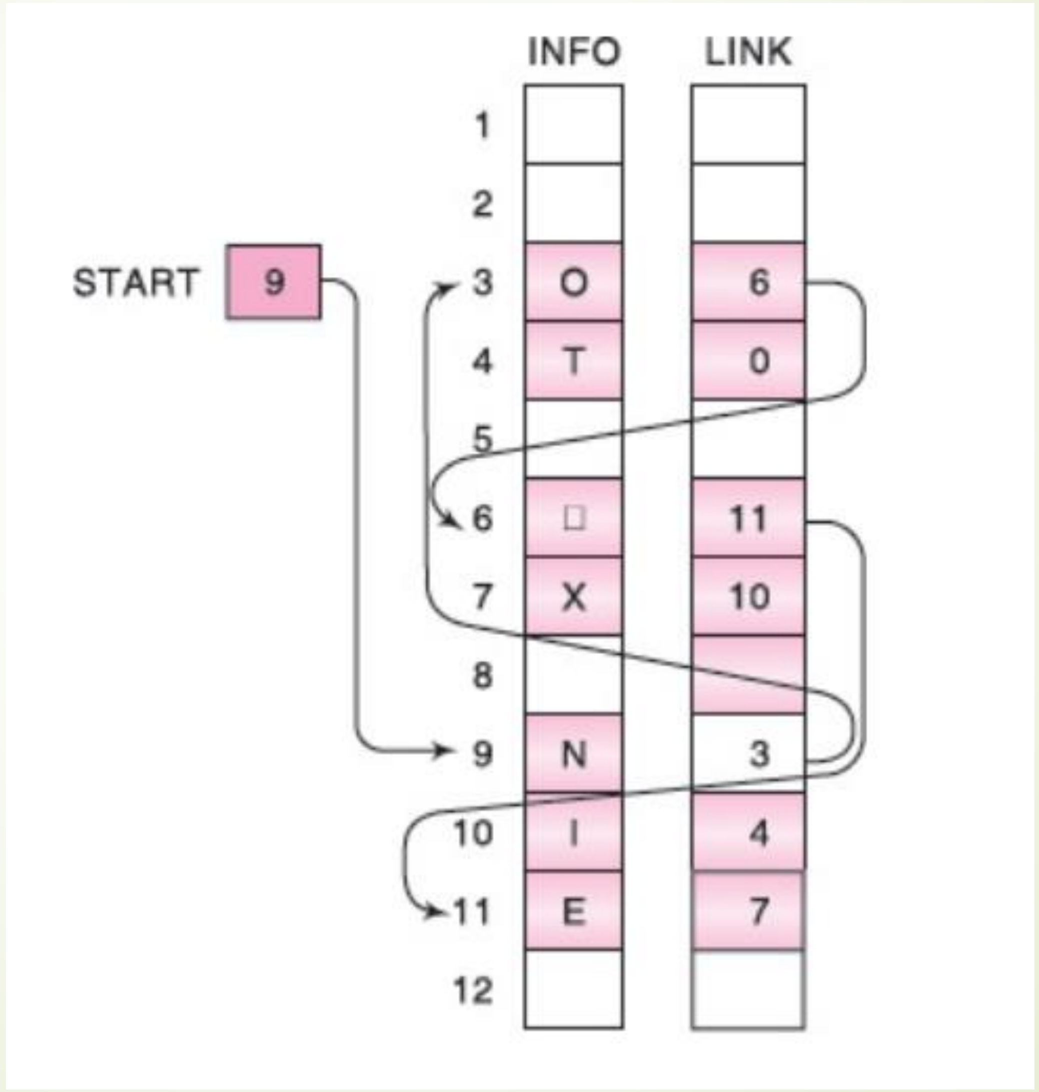
Null List: START pointer points to NULL

Basic Node Implementation using pointer

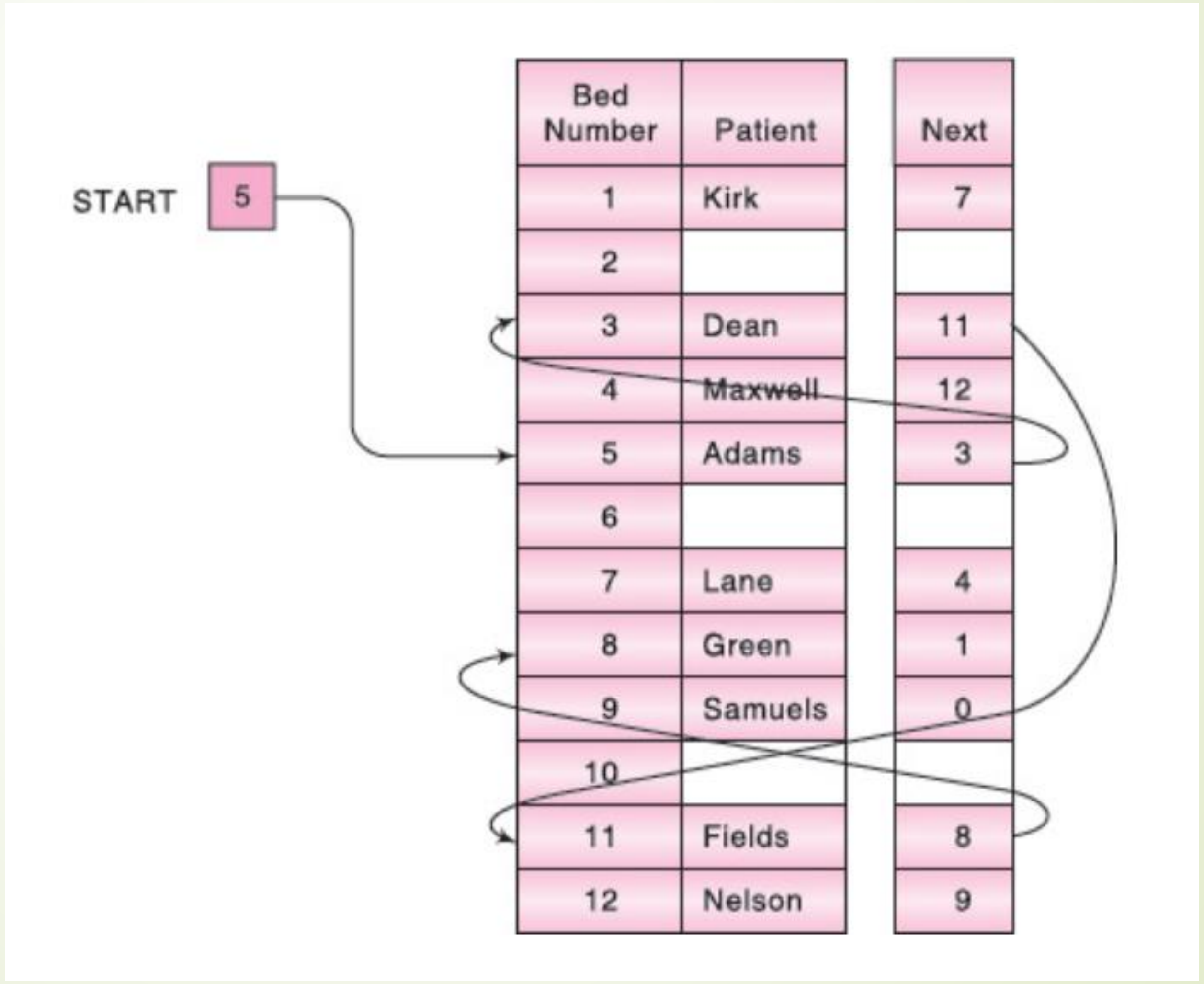
The following code is written in C:

```
Struct Node
{
    int data;           //any type of data could be another struct
    Node *next;         //this is an important piece of code “pointer”
};
```

Array Representation of Linked List in Memory

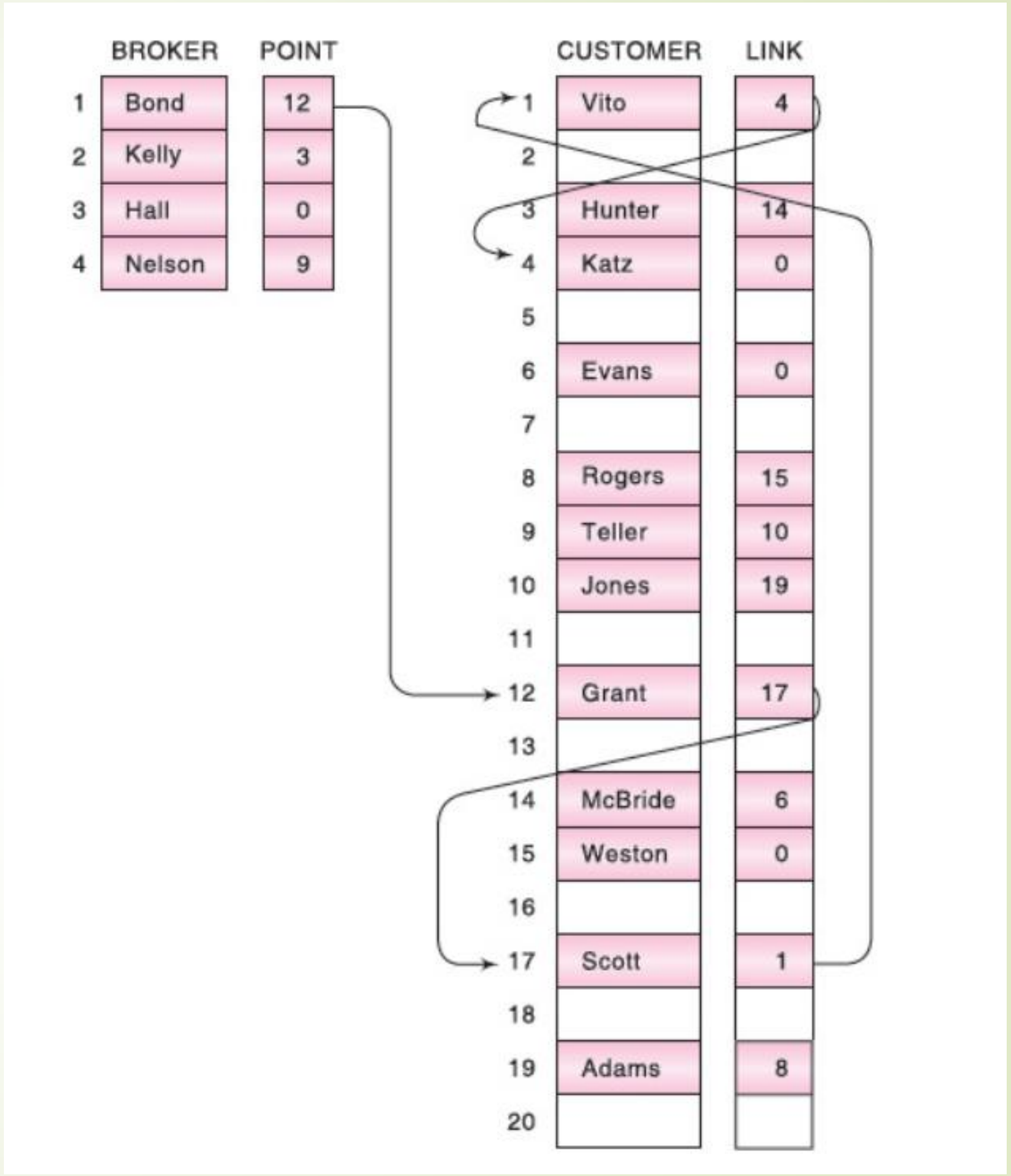


Array Representation of Linked List in Memory



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

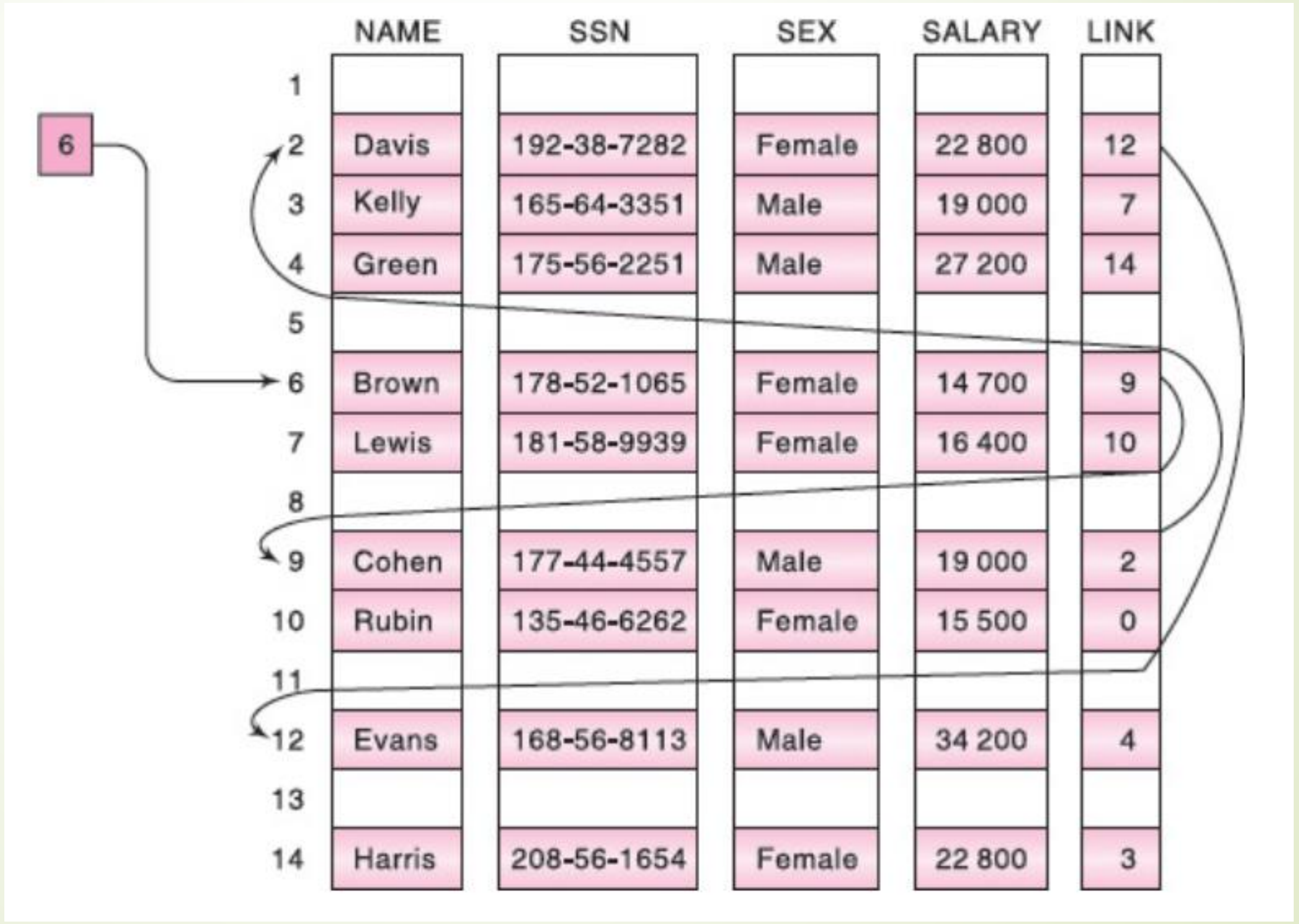
Multiple Linked Lists



	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		

Multiple items in INFO field

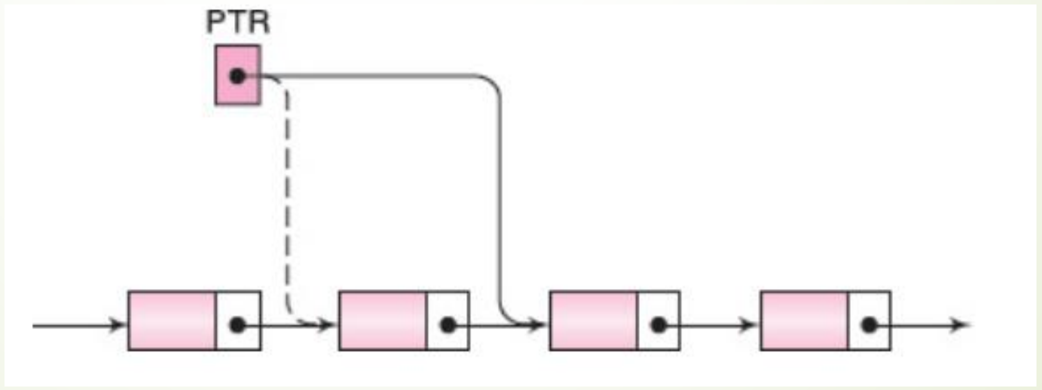


	NAME	SSN	SEX	SALARY	LINK
1					
2	Davis	192-38-7282	Female	22 800	12
3	Kelly	165-64-3351	Male	19 000	7
4	Green	175-56-2251	Male	27 200	14
5					
6	Brown	178-52-1065	Female	14 700	9
7	Lewis	181-58-9939	Female	16 400	10
8					
9	Cohen	177-44-4557	Male	19 000	2
10	Rubin	135-46-6262	Female	15 500	0
11		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

Traversal

- LIST be a linked list with two arrays INFO and LINK with START pointer
- NULL indicates end of list
- PTR points to current node, LINK[PTR] points to next node

PTR := LINK[PTR]



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.
```

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.
```

Code snippet in C and C++

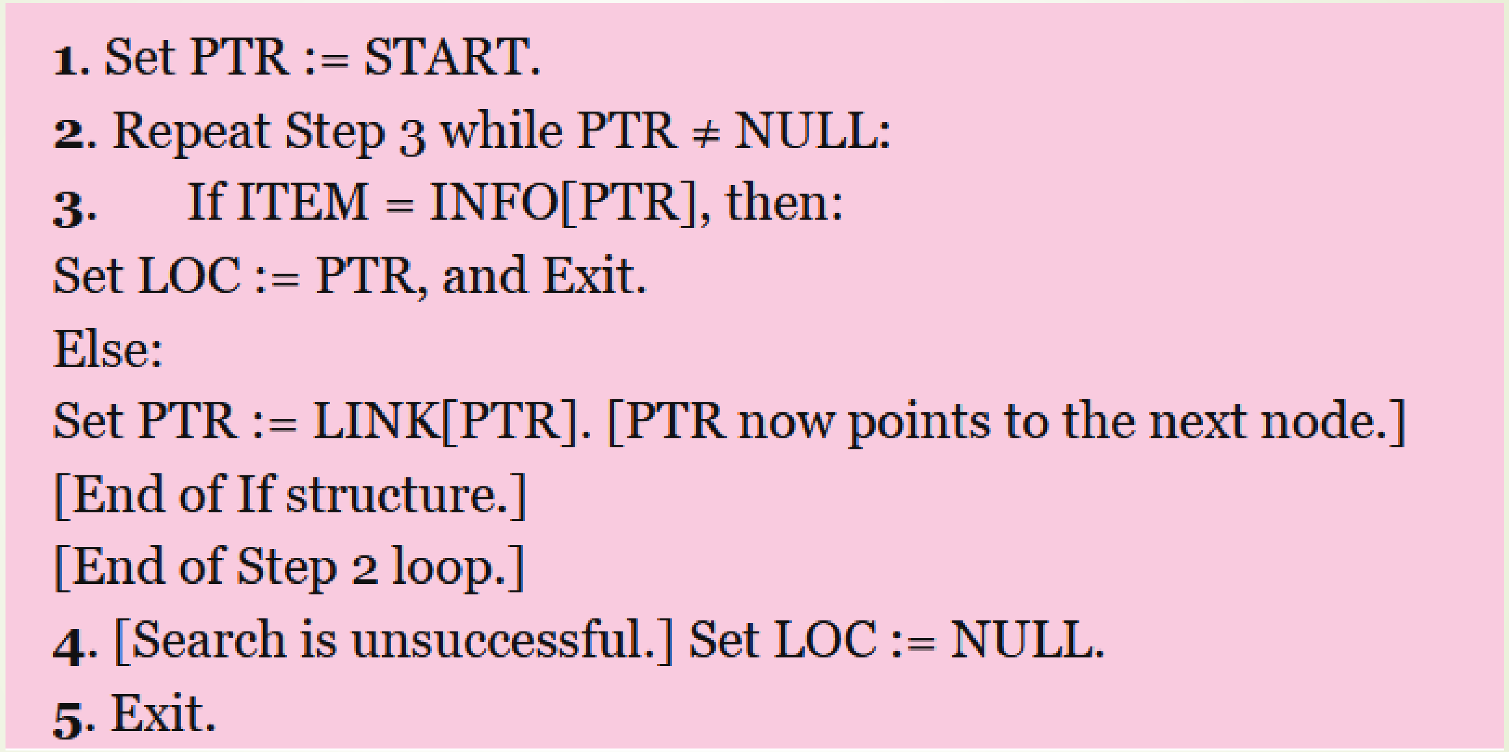
```c
struct Node {
    int data;
    struct Node* next;
};


void printList(struct Node* n)
{
while (n != NULL) {
        printf(" %d ", n->data);
        n = n->next;
    }
}
```

```cpp
class Node {
public:
    int data;
    Node* next;
};

void printList(Node* n)
{
while (n != NULL) {
        cout << n->data << " ";
        n = n->next;
  }
}
```

Searching



```
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.
```

Algorithm remain same if LIST is sorted

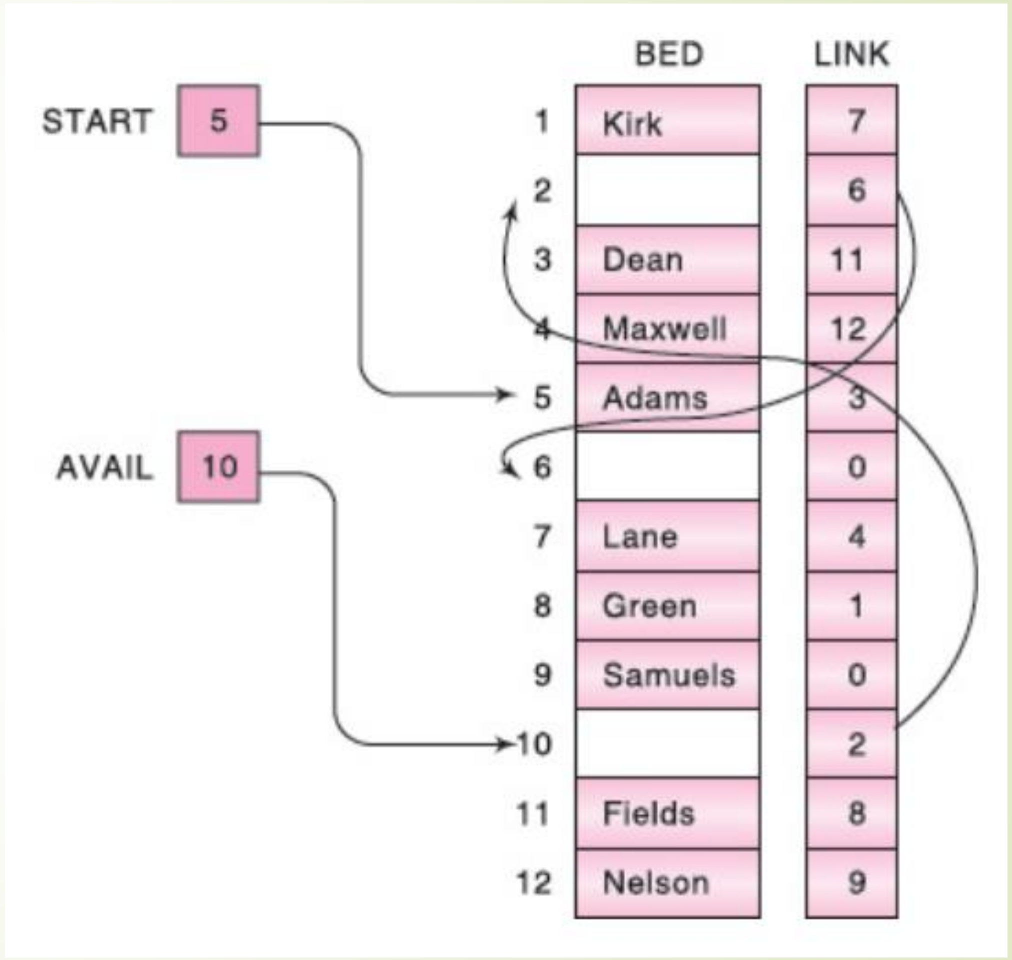
Binary Search cannot be applied: no way to go to mid element directly

Search Complexity= O(n/2)

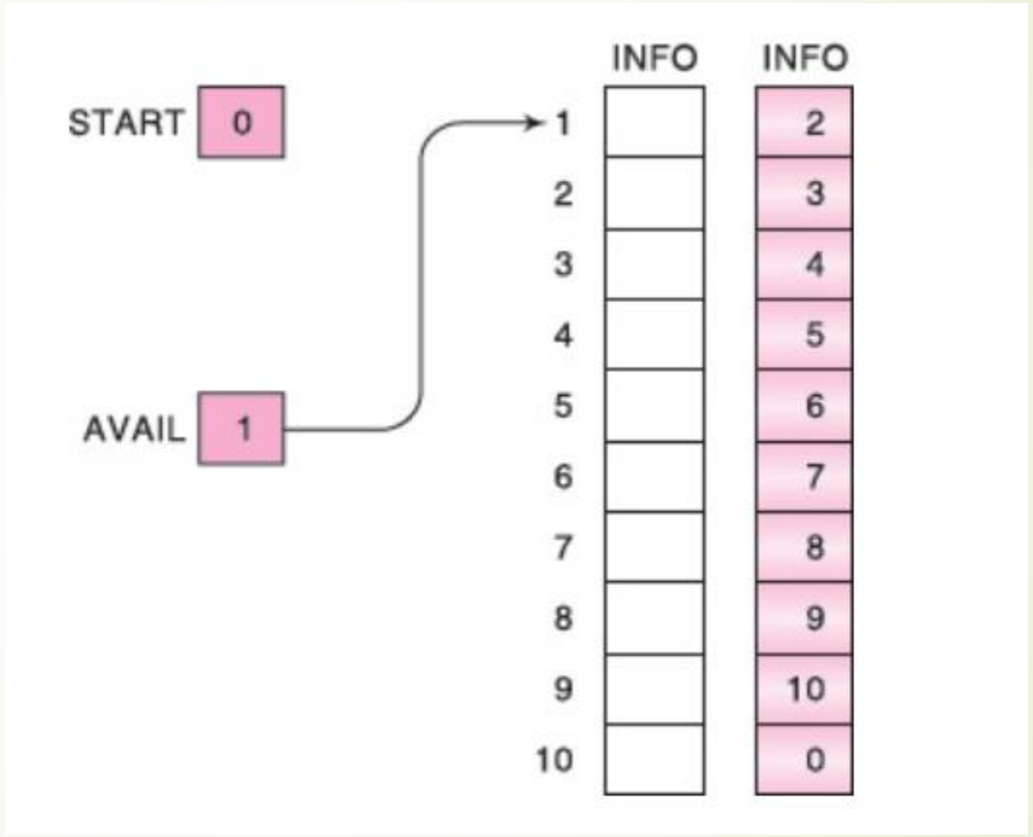
Memory Allocation

- Insertion requires memory for new nodes
- Deletion requires management of deleted nodes
- A special list is maintained which consists of unused memory cells called AVAIL

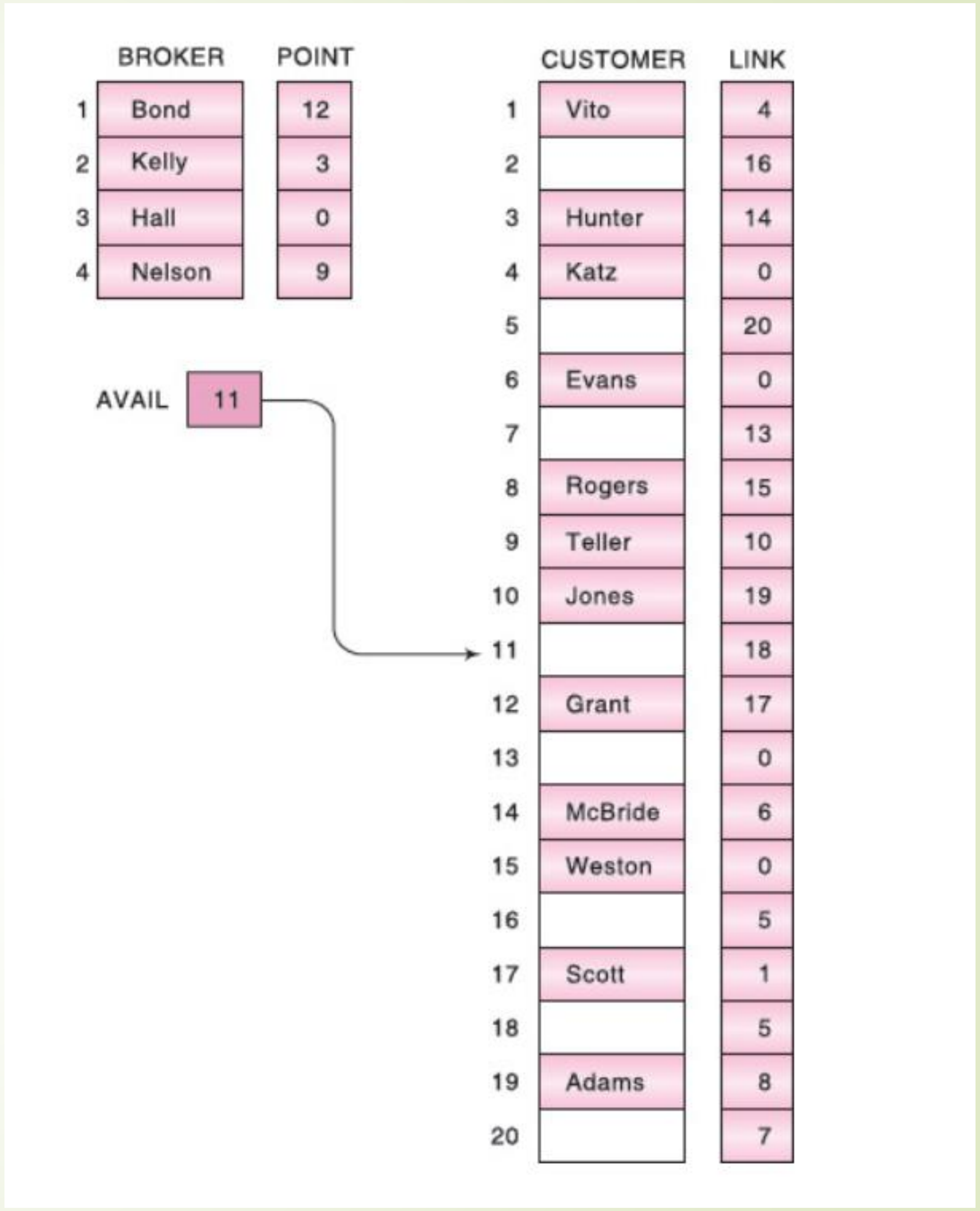
LIST(INFO, LINK, START, AVAIL)



Initial State: Example



Example



	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

Overflow and Underflow

- Overflow: Try to insert into LIST but free-storage list is empty
  - AVAIL = NULL
  - Increase the size of arrays
- Underflow: try to delete an element from LIST but it is empty
  - START = NULL

New node creation in C and C++

- C

```
(struct Node*)malloc(sizeof(struct Node))
```

- C++

```
new Node();
```

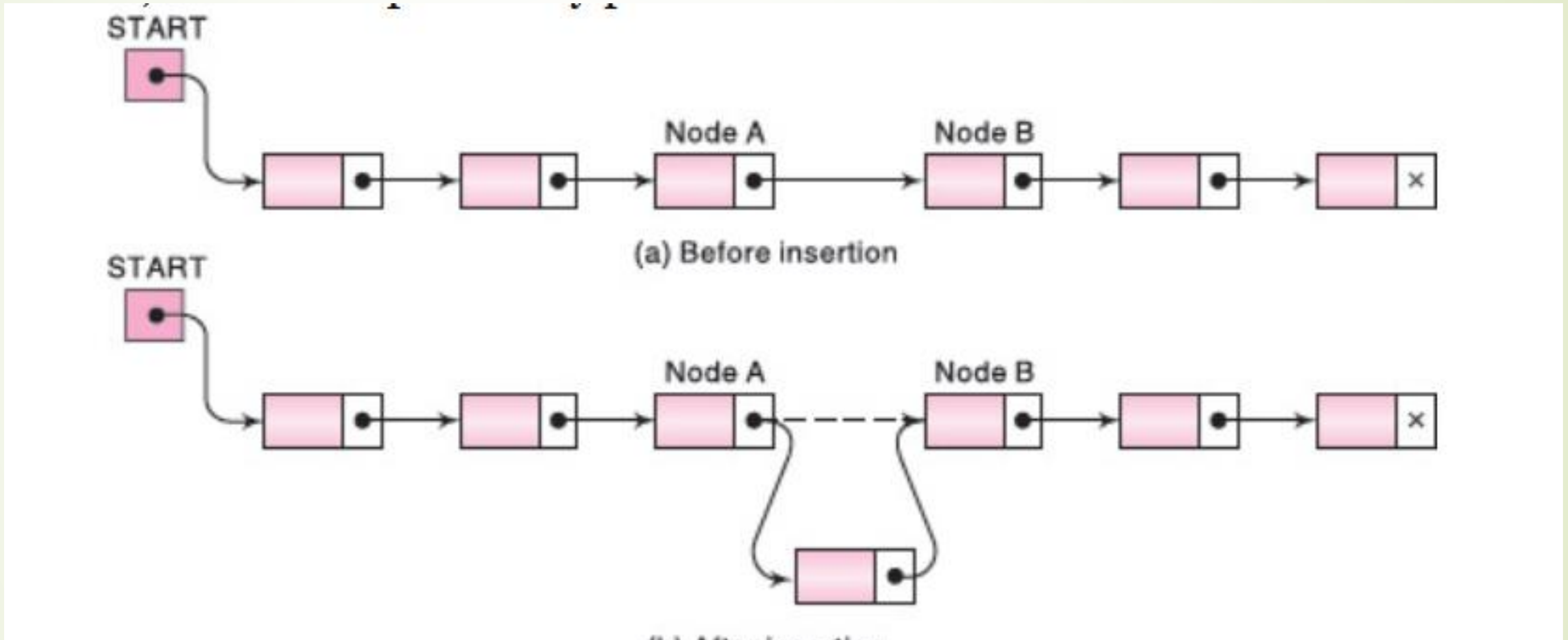
To delete (de-allocate the memory) a node

```
free(node)
```

Garbage Collection

- When a node is deleted, its space should be immediately available for future reuse
- However, this is too time-consuming for operating system
- Alternative method
  - Periodically collect all deleted space in a free-storage list
  - Two steps:
    - Run through all lists, tagging all cells in use
    - Run again to collect all untagged cells onto free-storage list
  - Garbage collection is done when there is few (or no) space available or the computer has time to run garbage collection

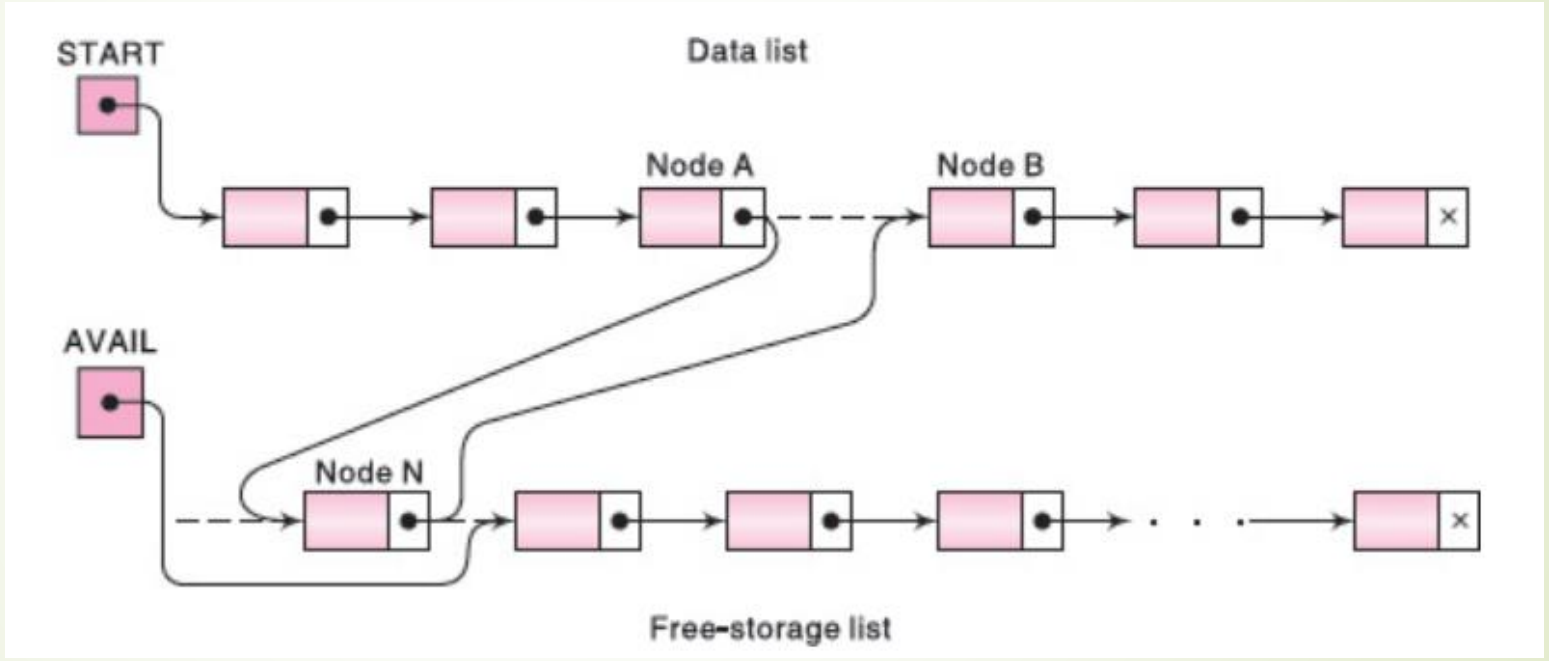
Insertion



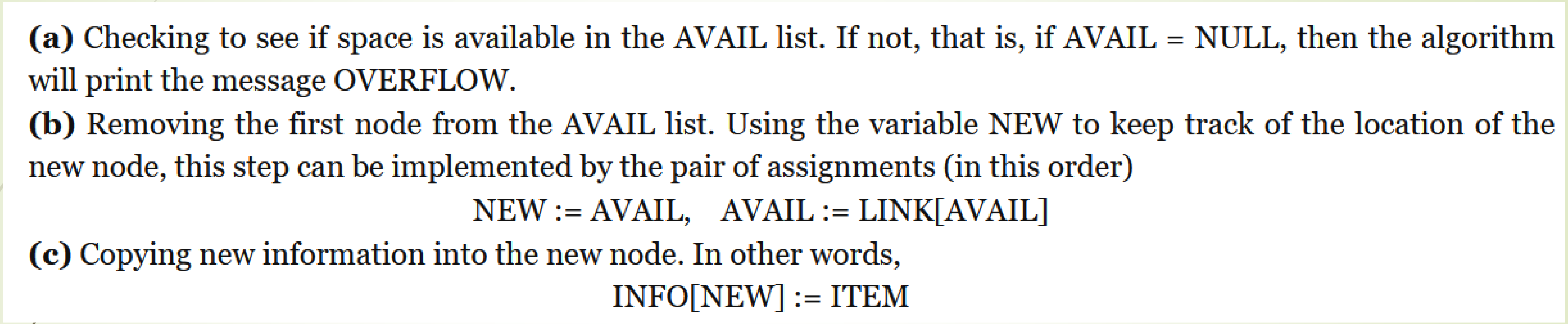
(a) Before insertion

(b) After insertion

Insertion: with AVAIL list



Steps to create a new node for insertion



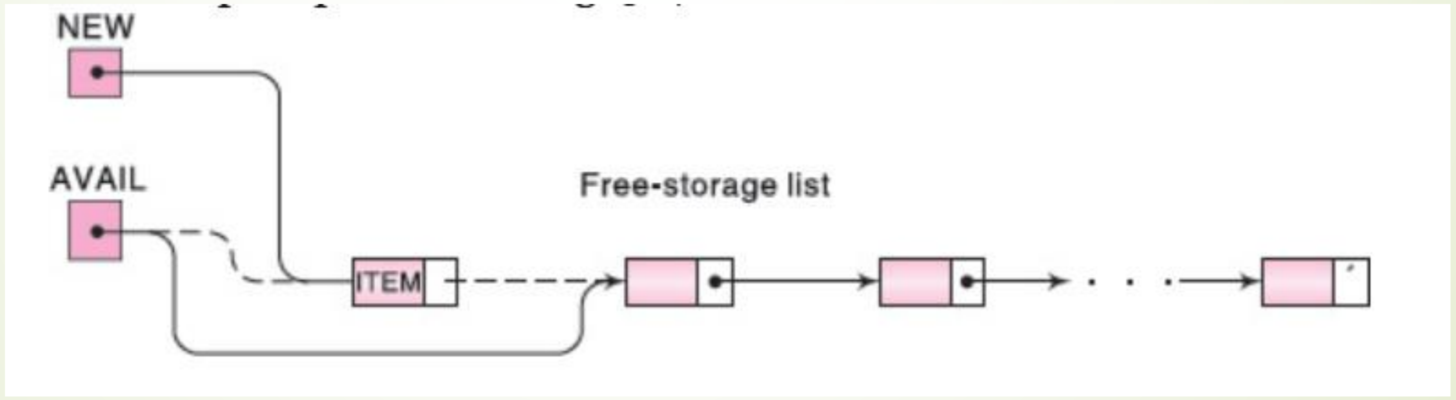
(a) Checking to see if space is available in the AVAIL list. If not, that is, if AVAIL = NULL, then the algorithm will print the message OVERFLOW.

(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



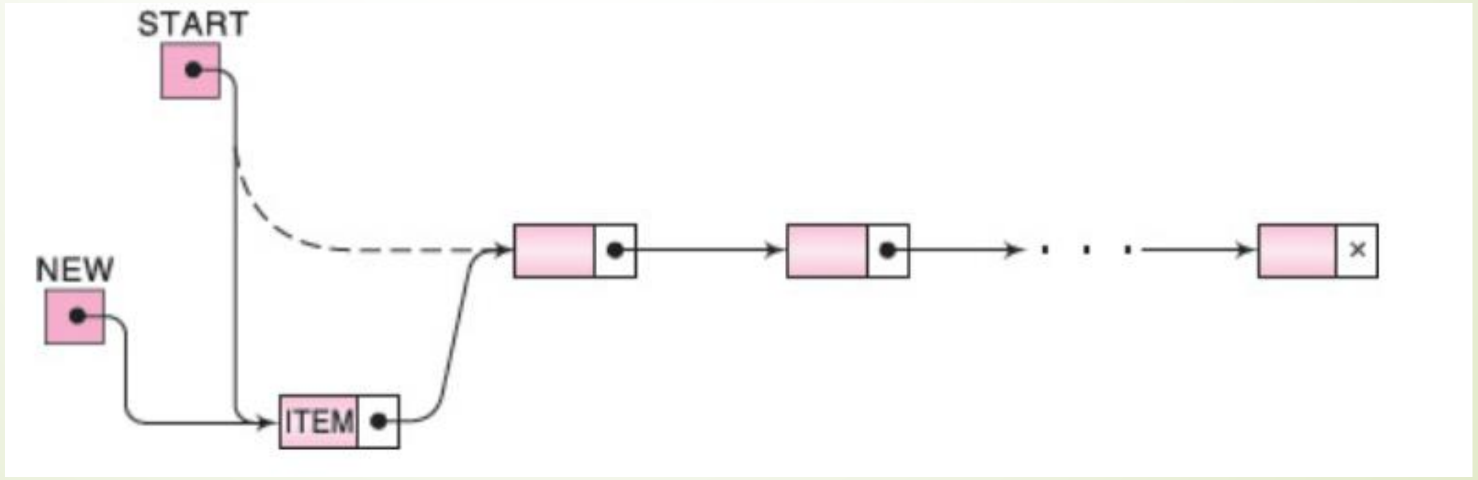
Inserting the node in a LL

There are 3 cases here:-

- Insertion at the beginning
- Insertion at the end
- Insertion after a particular node

Insertion at the beginning

```
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.
```



Insertion after a specific node

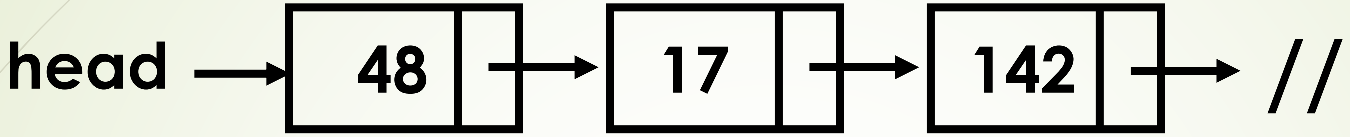
```
INSLOC(INFO, LINK, START, AVAIL, LOC, ITEM)
  1. [OVERFLOW?] If AVAIL = NULL, then: Write: OVERFLOW, and Exit.
  2. [Remove first node from AVAIL list.]
  Set NEW := AVAIL and AVAIL := LINK[AVAIL].
  3. Set INFO[NEW] := ITEM. [Copies new data into new node.]
  4. If LOC = NULL, then: [Insert as first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

Insertion at the end

Steps:

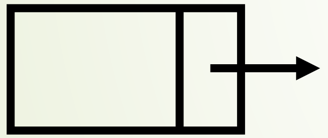
- Create a Node
- Set the node data Values and node pointer to NULL
- Goto the last node and make the next pointer of the last node point to the new node

Insertion at the end- Description

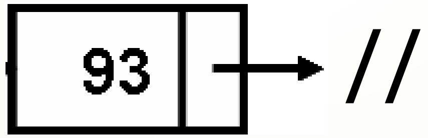


- Follow the previous steps and we get

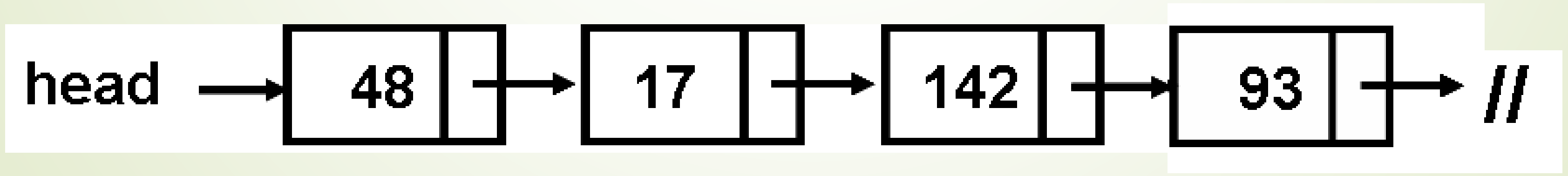
Step 1



Step 2



Step 3



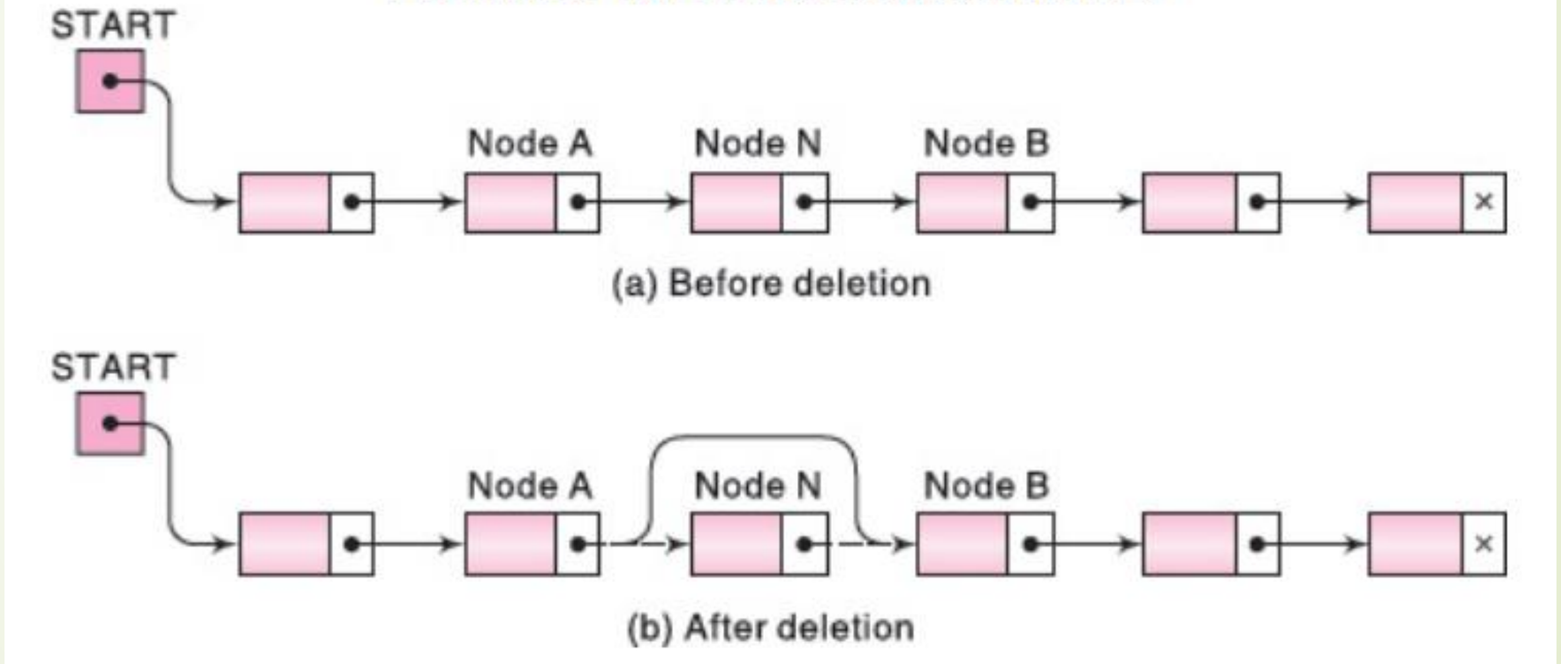
Insertion into sorted LL

```
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]. [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.
```

```
INSERT(INFO, LINK, START, AVAIL, ITEM)
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.
```

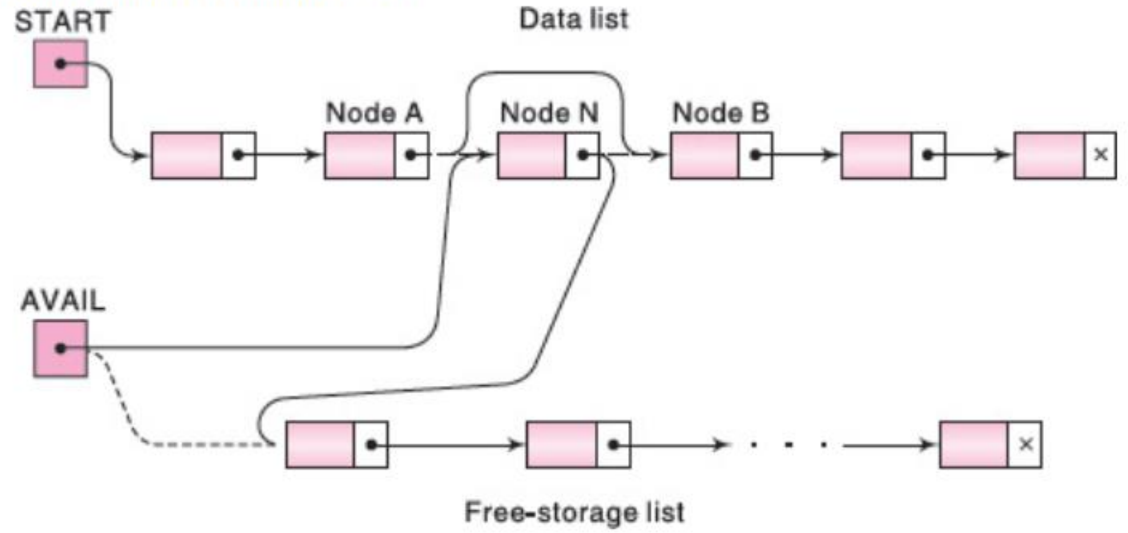
Deletion

LIST(INFO, LINK, START, AVAIL)



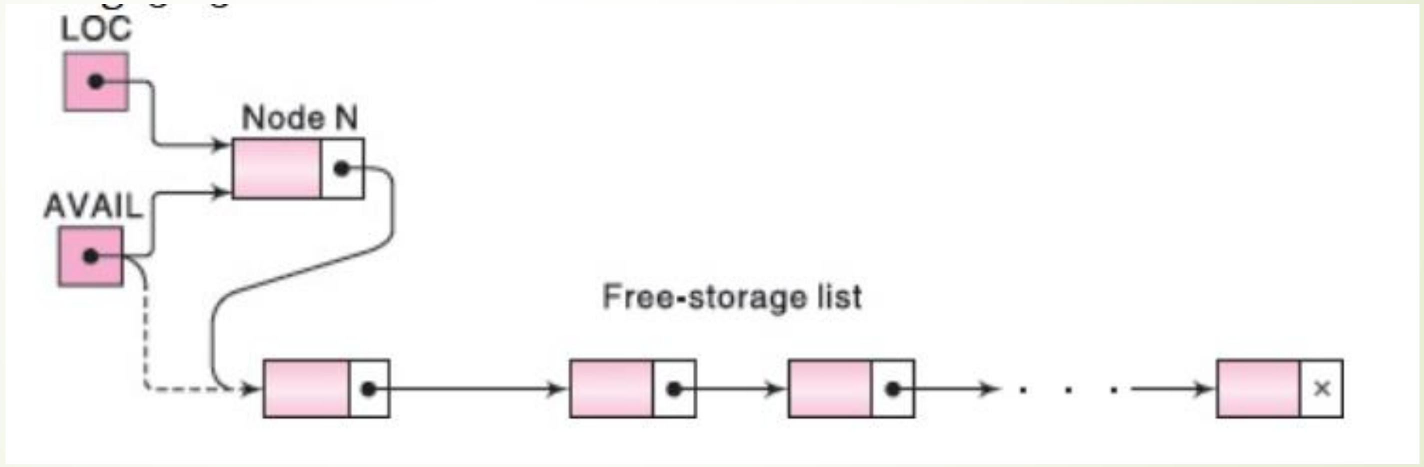
Deletion with AVAIL

(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.



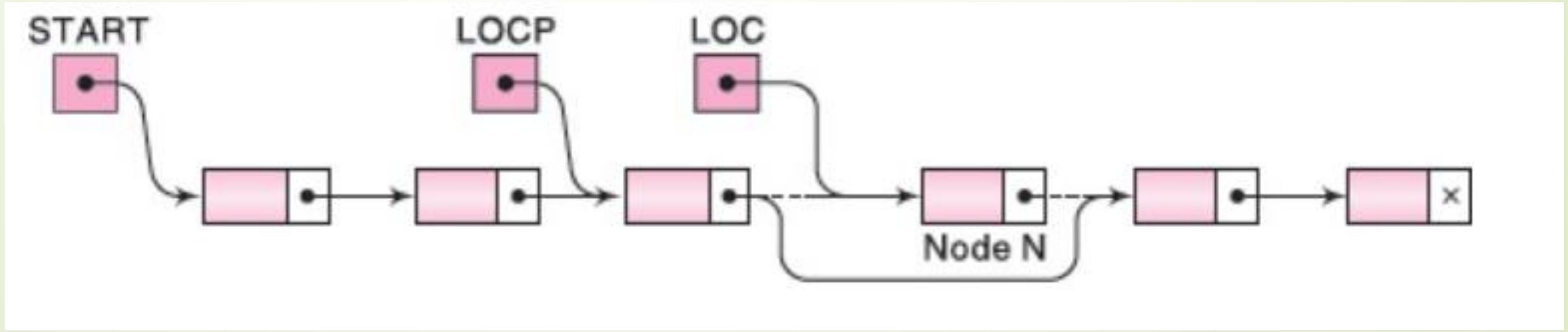
Collection of deleted node in AVAIL

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



Deletion following a given node

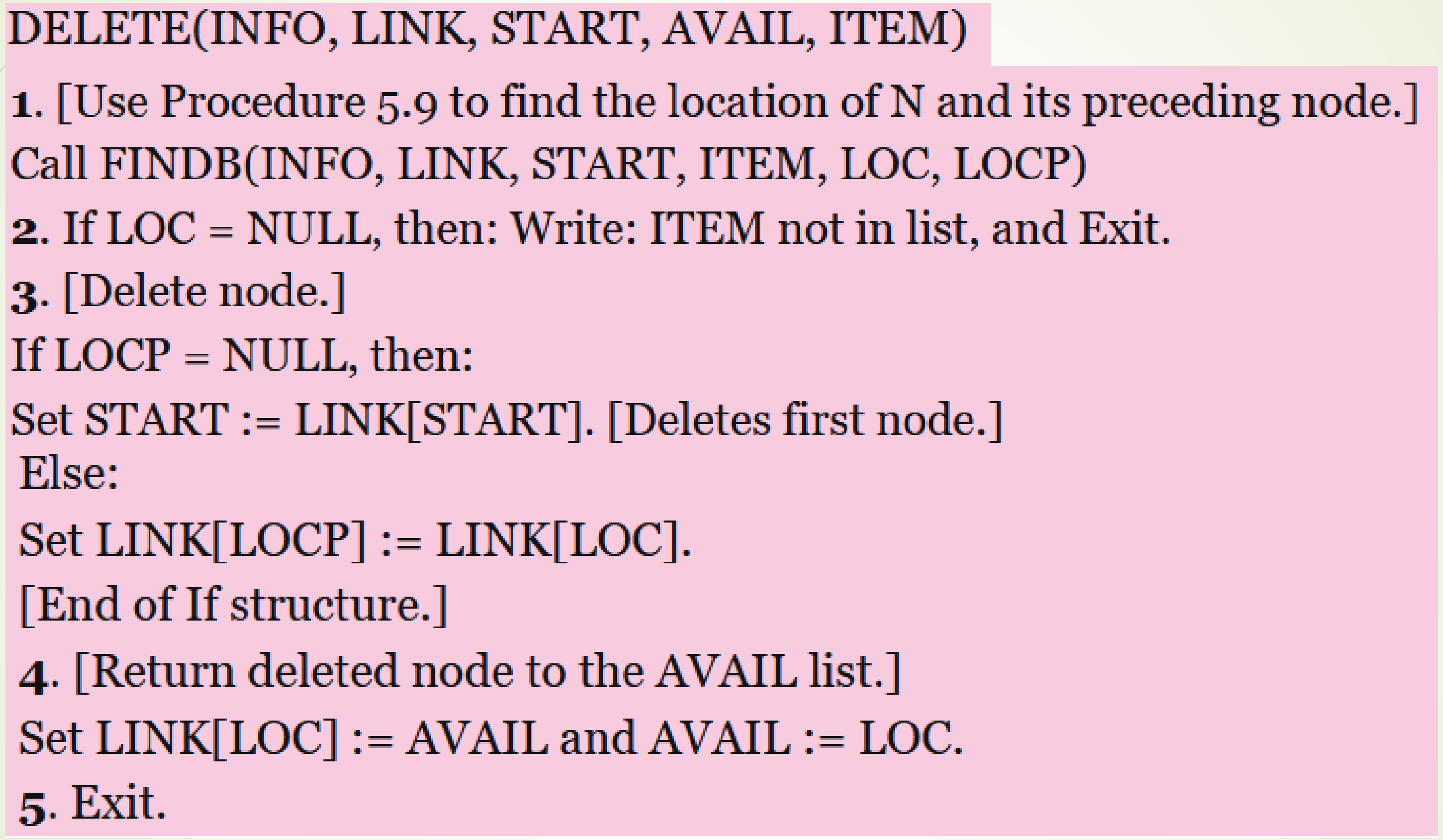
```
DEL(INFO, LINK, START, AVAIL, LOC, LOCP)
1. If LOCP = NULL, then:
   Set START := LINK[START]. [Deletes first node.]
   Else:
   Set LINK[LOCP] := LINK[LOC]. [Deletes node N.]
   [End of If structure.]
2. [Return deleted node to the AVAIL list.]
   Set LINK[LOC] := AVAIL and AVAIL := LOC.
3. Exit.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



Delete when ITEM is given

```
FINDB(INFO, LINK, START, ITEM, LOC, LOCP)
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.
```

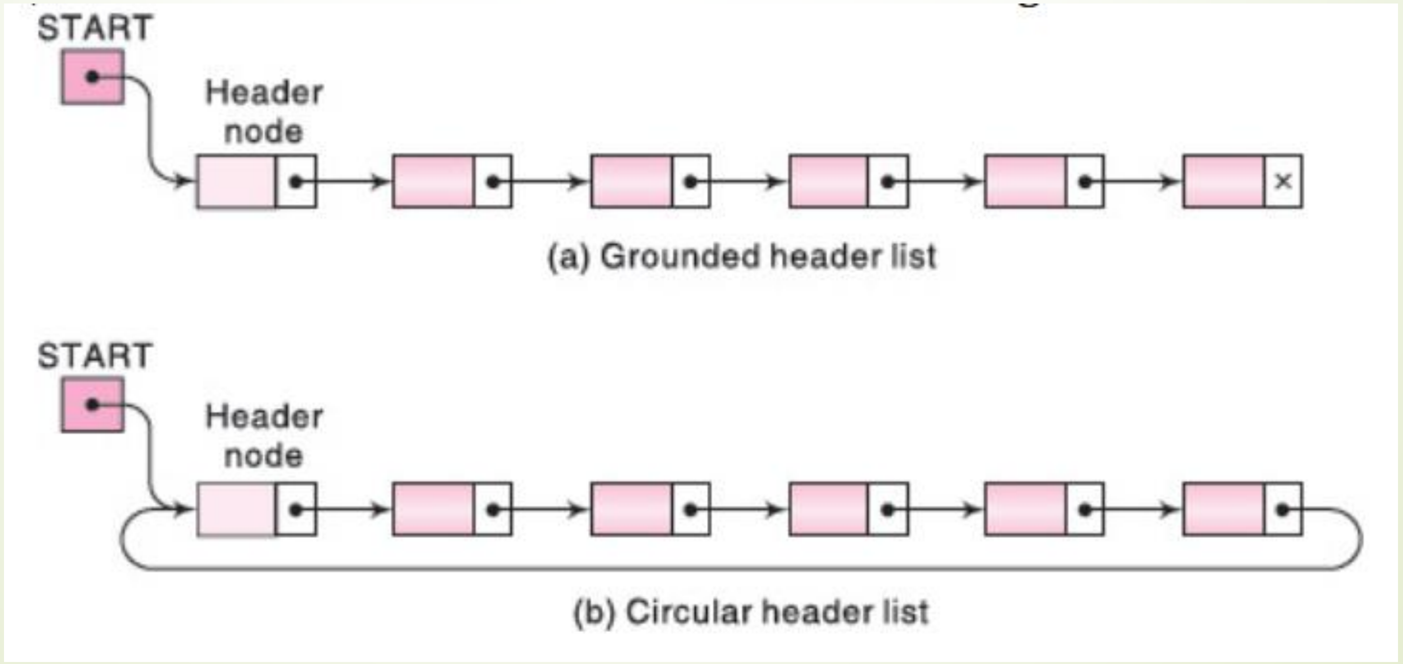
DELETE



```
DELETE(INFO, LINK, START, AVAIL, ITEM)
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.
```

Header Linked List

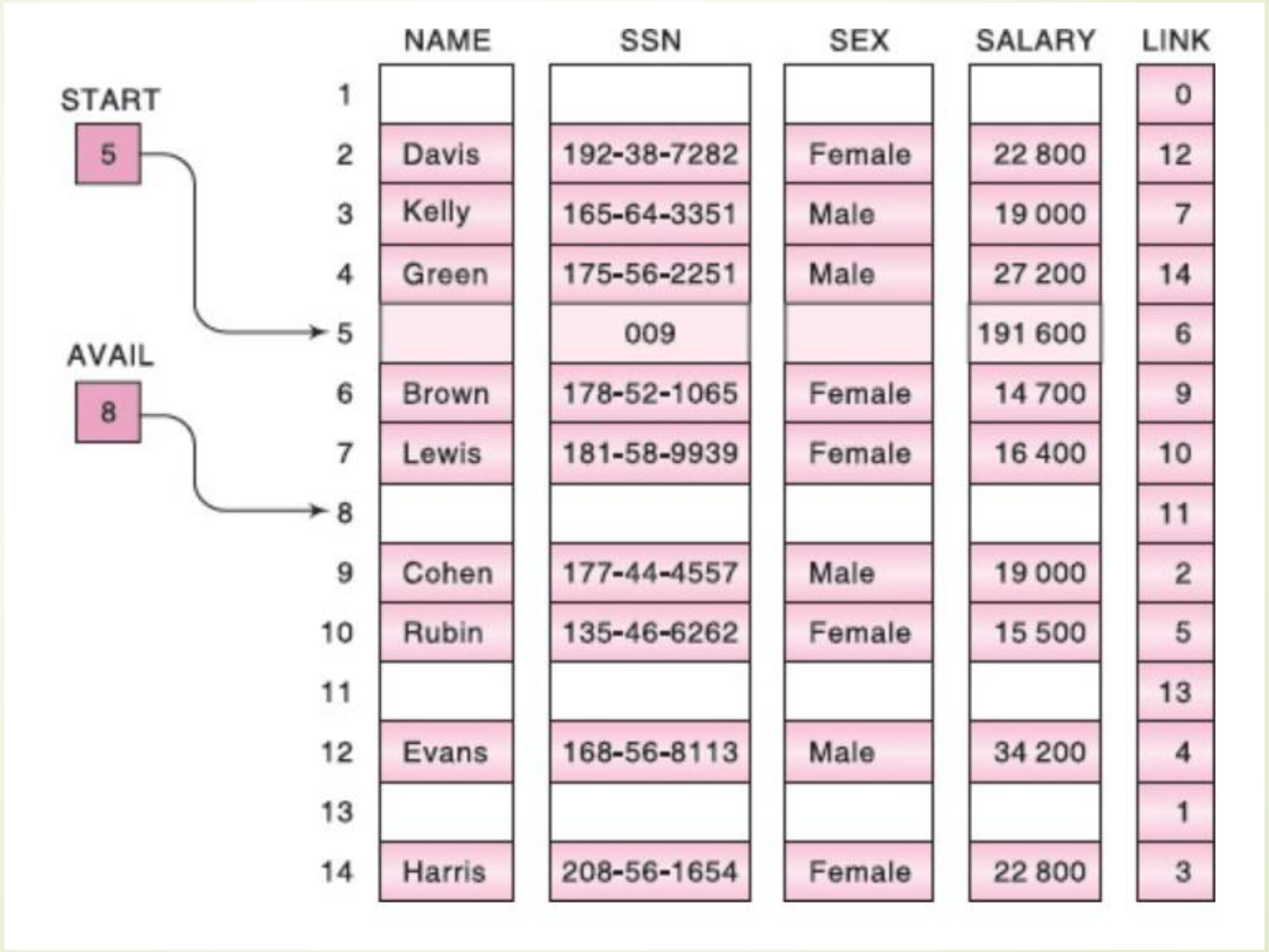
- Contain a special header node in the beginning
- Two kind of header lists
  - Grounded: last node contain NULL pointer
    - Empty if LINK[STRAT] = NULL
  - Circular: last node points back to header node
    - Emplty if LINK[START] = START



(a) Grounded header list

(b) Circular header list

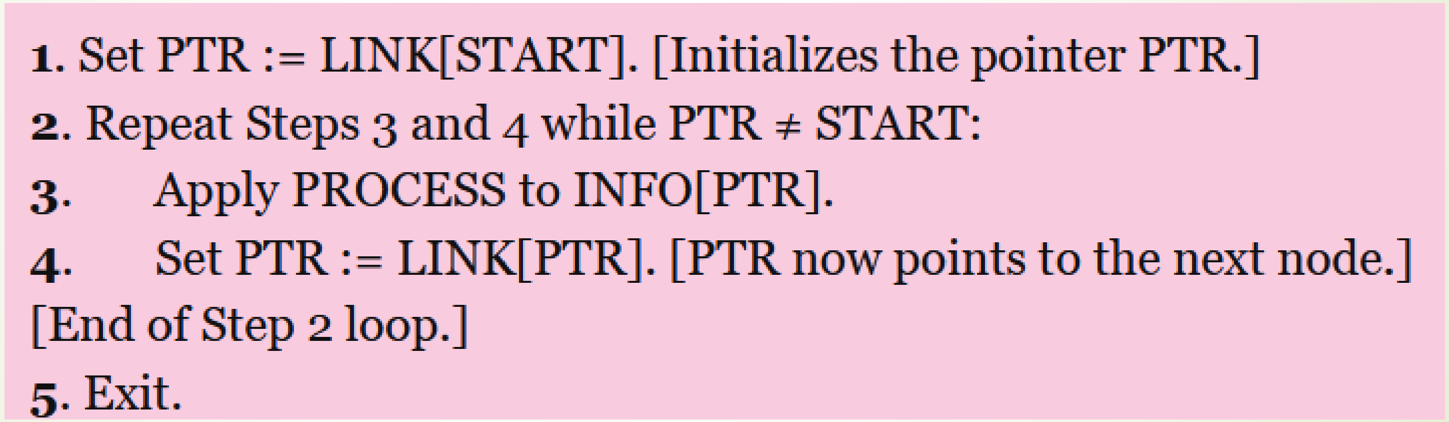
Example: Header List



	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		009		191 600	6
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	5
11					13
12	Evans	168-56-8113	Male	34 200	4
13					1
14	Harris	208-56-1654	Female	22 800	3

Advantage of circular LL

- NULL pointer is not used, every node has valid address
- Every node has a predecessor, first node does not require special case
- Traversal

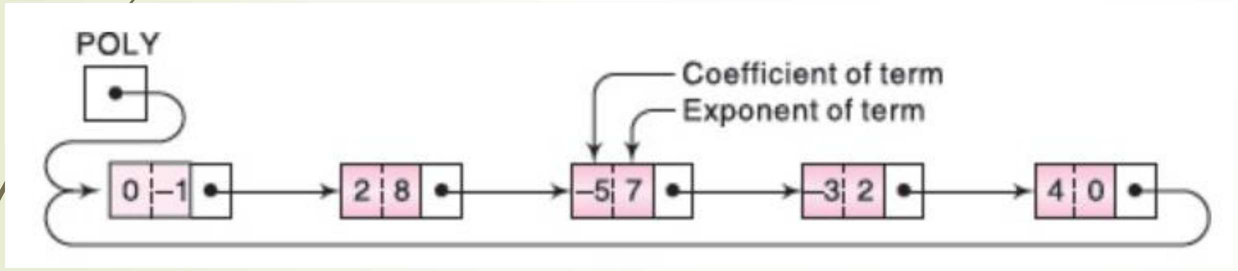


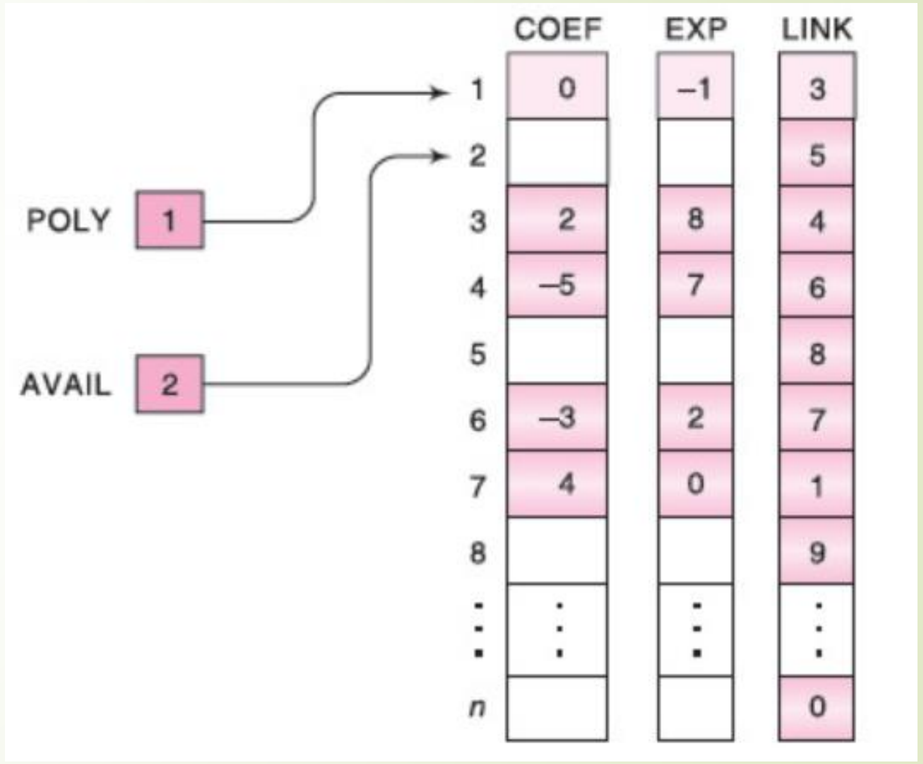
```
1. Set PTR := LINK[START]. [Initializes the pointer PTR.]
2. Repeat Steps 3 and 4 while PTR ≠ START:
3.      Apply PROCESS to INFO[PTR].
4.      Set PTR := LINK[PTR]. [PTR now points to the next node.]
   [End of Step 2 loop.]
5. Exit.
```

- Algorithms can be written for search, insertion and deletion operation for CLLs

Example of CHLL: polynomials

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

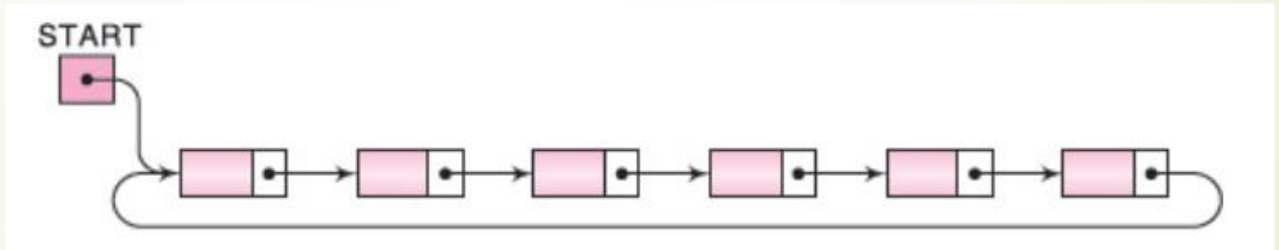




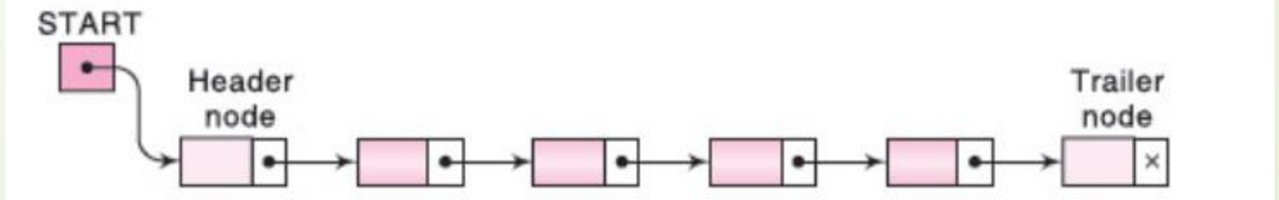
	COEF	EXP	LINK
1	0	–1	3
2			5
3	2	8	4
4	–5	7	6
5			8
6	–3	2	7
7	4	0	1
8			9
⋮	⋮	⋮	⋮
n			0

More variants of LLs

- Circular LL: in which last node points to the first (no header node)
- LL which contain a special header node in the beginning and a special trailer node in the end



(a) Circular linked list



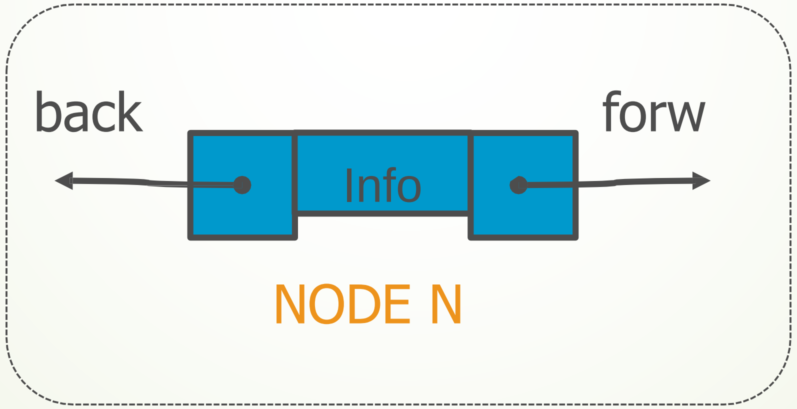
(b) Linked list with header and trailer nodes

▪ **Doubly Linked List (**Two-way LLs)

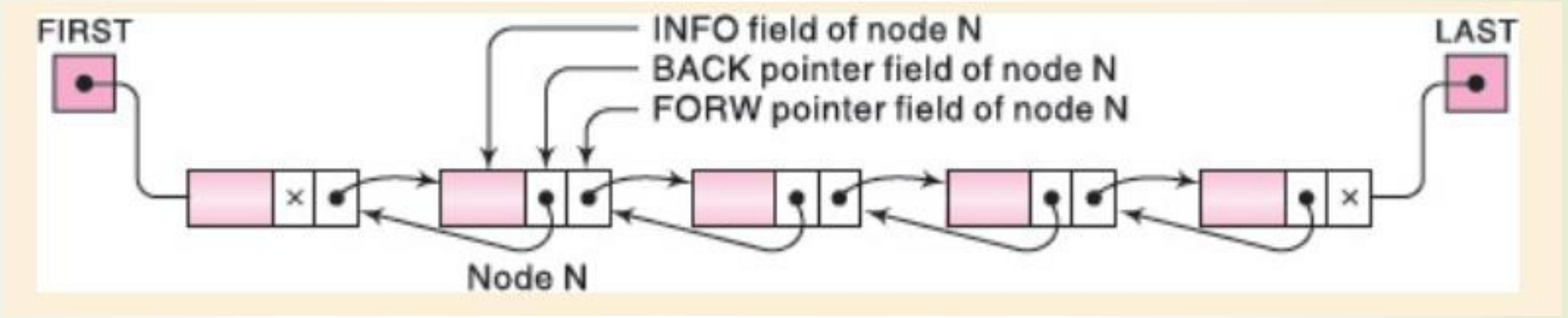
- Linked Lists or one-way LL or Singly LL
- In DLLs, Each node contain two pointers- FORW and BACK
- FORW points to the next node
- BACK points to the preceding node
- DLL is called two-way list because we can traverse the list in both the forward and backward direction

▪Node Data: three fields

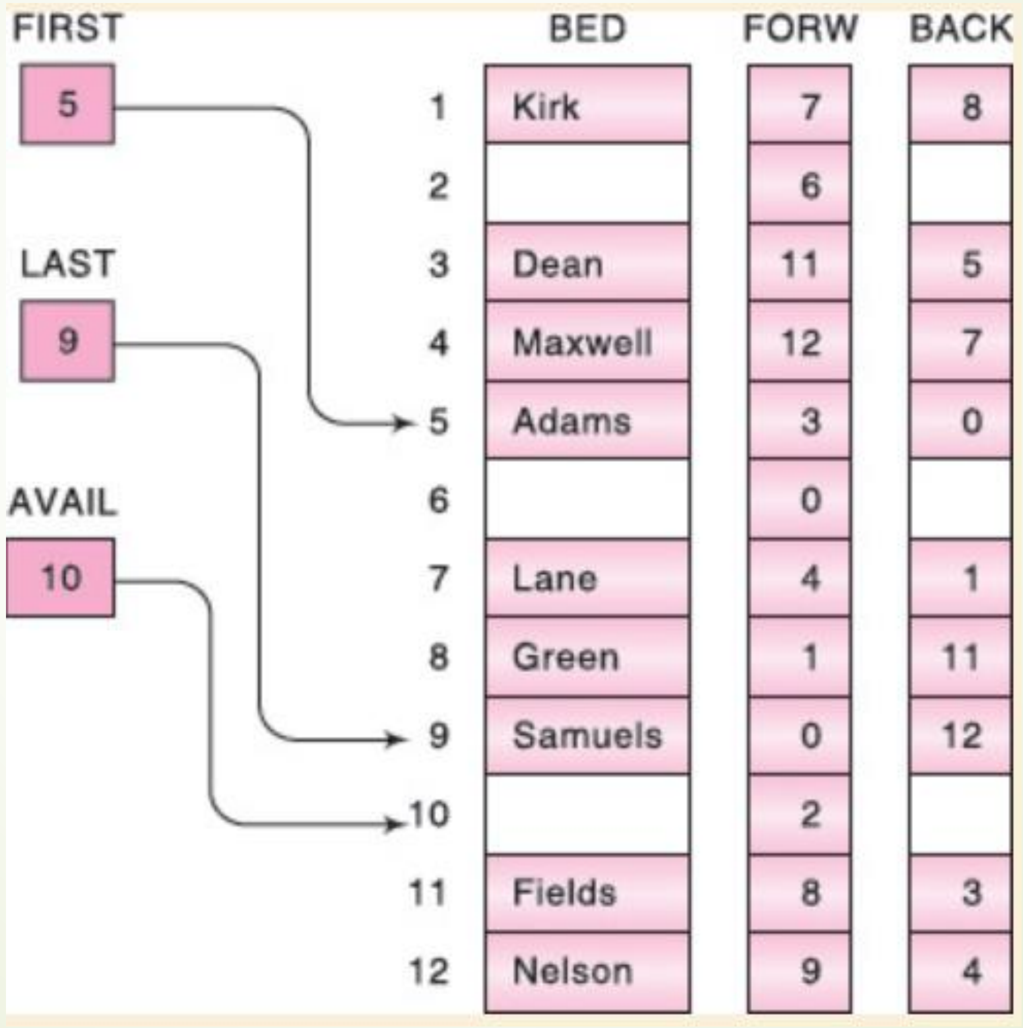
(1) An information field INFO which contains the data of N
(2) A pointer field FORW which contains the location of the next node in the list
(3) A pointer field BACK which contains the location of the preceding node in the list



Schematic Diagram



Array Representation



FIRST: 5

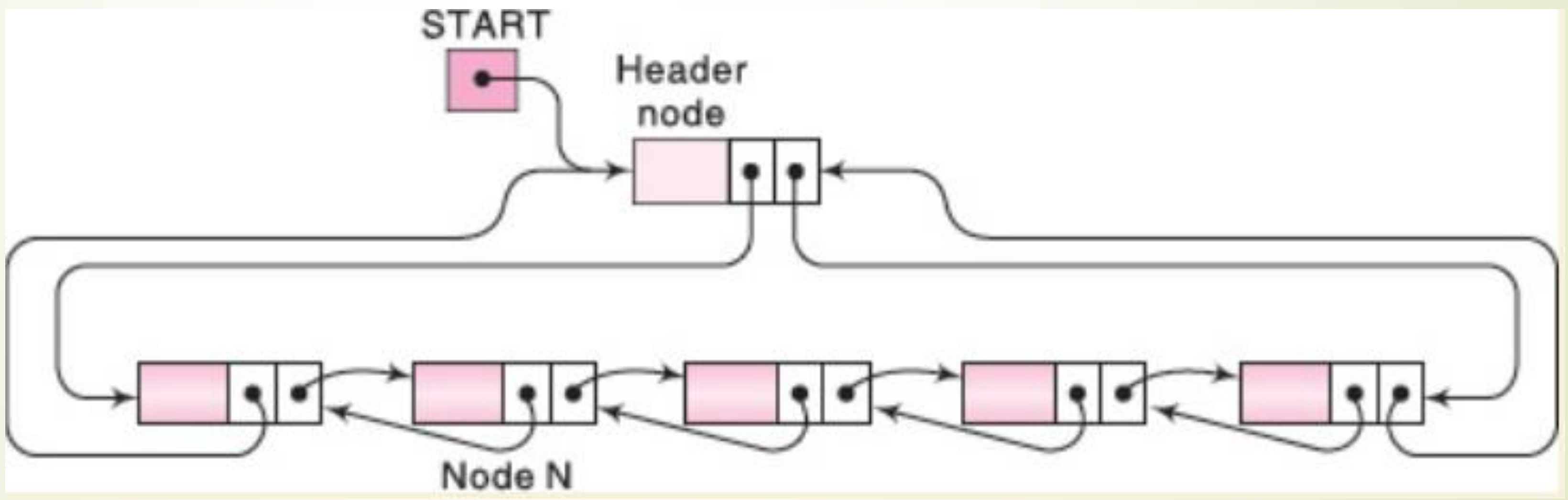
LAST: 9

AVAIL: 10

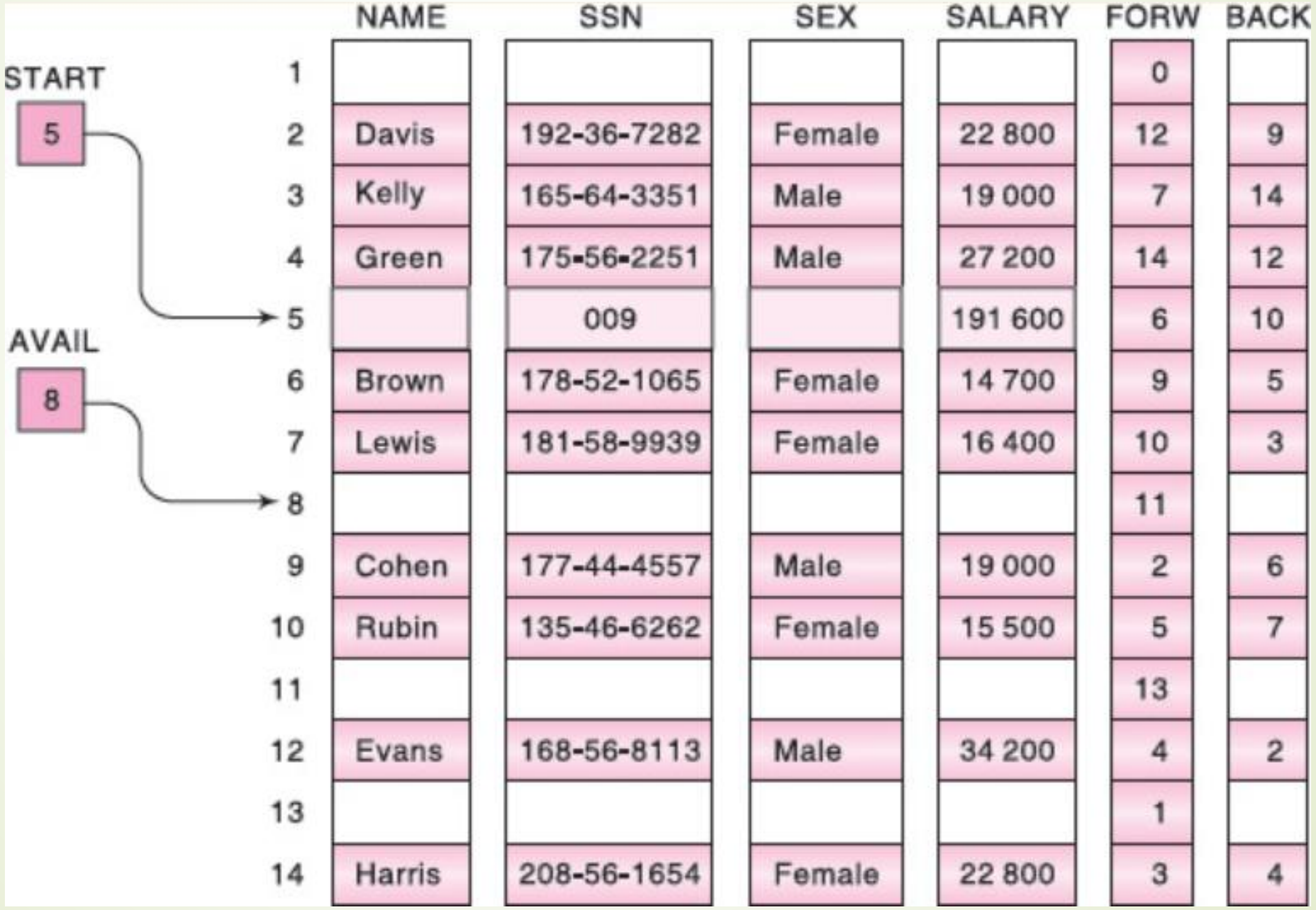
	BED	FORW	BACK
1	Kirk	7	8
2		6	
3	Dean	11	5
4	Maxwell	12	7
5	Adams	3	0
6		0	
7	Lane	4	1
8	Green	1	11
9	Samuels	0	12
10		2	
11	Fields	8	3
12	Nelson	9	4

Two Way Header List

- Advantages of both two way list and circular header list
- Only one START pointer is required



Example: Two Way Header List



START: 5

AVAIL: 8

	NAME	SSN	SEX	SALARY	FORW	BACK
1					0	
2	Davis	192-36-7282	Female	22 800	12	9
3	Kelly	165-64-3351	Male	19 000	7	14
4	Green	175-56-2251	Male	27 200	14	12
5		009		191 600	6	10
6	Brown	178-52-1065	Female	14 700	9	5
7	Lewis	181-58-9939	Female	16 400	10	3
8					11	
9	Cohen	177-44-4557	Male	19 000	2	6
10	Rubin	135-46-6262	Female	15 500	5	7
11					13	
12	Evans	168-56-8113	Male	34 200	4	2
13					1	
14	Harris	208-56-1654	Female	22 800	3	4

▪Operations on a Doubly linked list

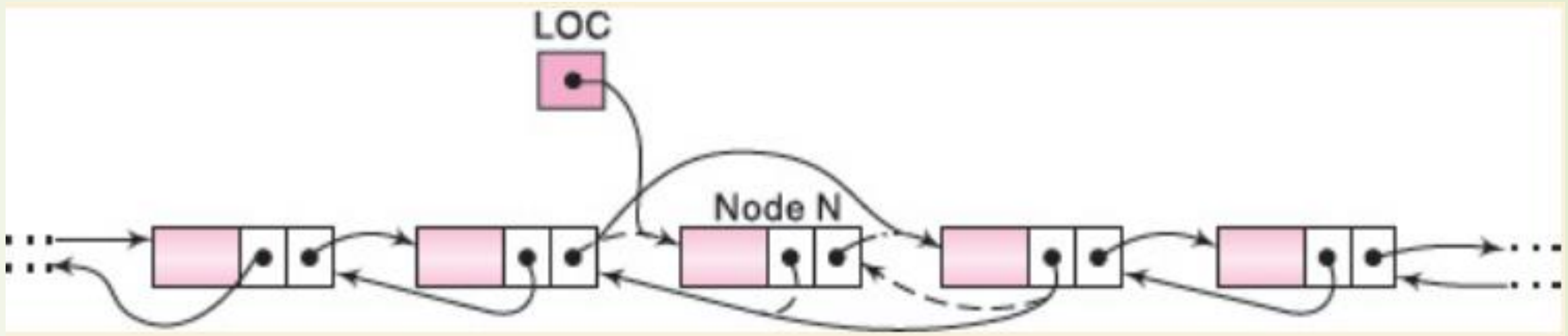
1) Create list
2) Searching
3) Traversal
4) Insert element at beginning in list.
5) Insert element at end in list.
6) Insert element at any place in list.
7) Delete element from the beginning of list.
8) Delete element from the end of list.
9) Delete element from any place from list.

Searching

There is advantage in DLL if it is sorted
We can start searching from the last node using back pointers if we suspect that information appears near the end of list.

Deletion

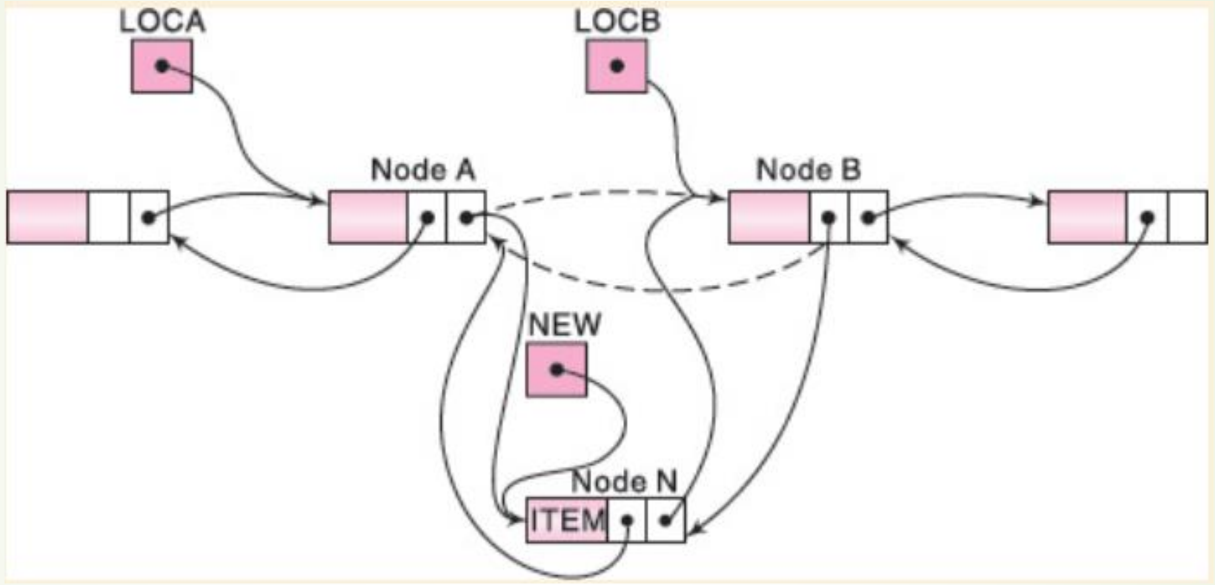
Only pointer to the node to be deleted is required



FORW[BACK[LOC]] := FORW[LOC] and BACK[FORW[LOC]] := BACK[LOC]

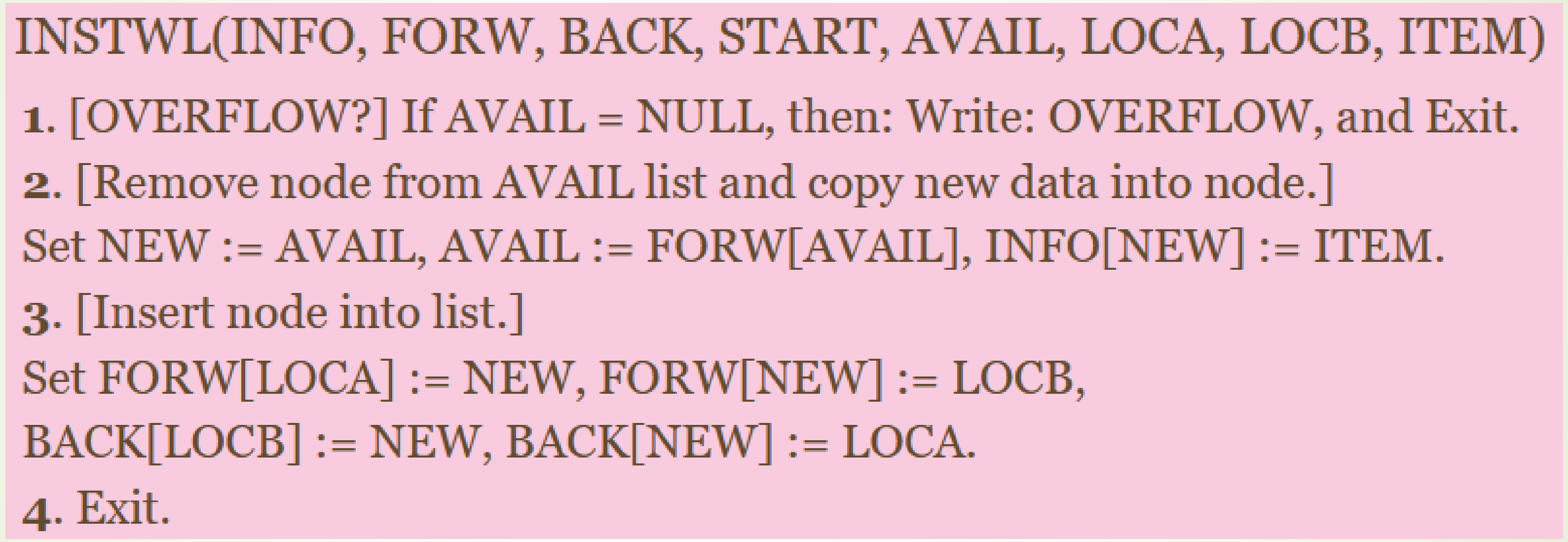
```
DELTWL(INFO, FORW, BACK, START, AVAIL, LOC)
 1. [Delete node.]
 Set FORW[BACK[LOC]] := FORW[LOC] and
 BACK[FORW[LOC]] := BACK[LOC].
 2. [Return node to AVAIL list.]
 Set FORW[LOC] := AVAIL and AVAIL := LOC.
 3. Exit.
```

Insertion between two nodes



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

Insertion



```
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.
```

Doubly Linked list

Advantages

1. We can traverse in both directions i.e. from starting to end and as well as from end to starting.
2. It is easy to reverse the linked list.
3. If we are at a node, then we can go to any node. But in linear linked list, it is not possible to reach the previous node.

Deletion is possible when only Node pointer is given.

Disadvantages

1. It requires more space per space per node because one extra field is required for pointer to previous node.
2. Insertion and deletion take more time than linear linked list because more pointer operations are required than linear linked list.