

# DATA STRUCTURE AND ALGORITHM

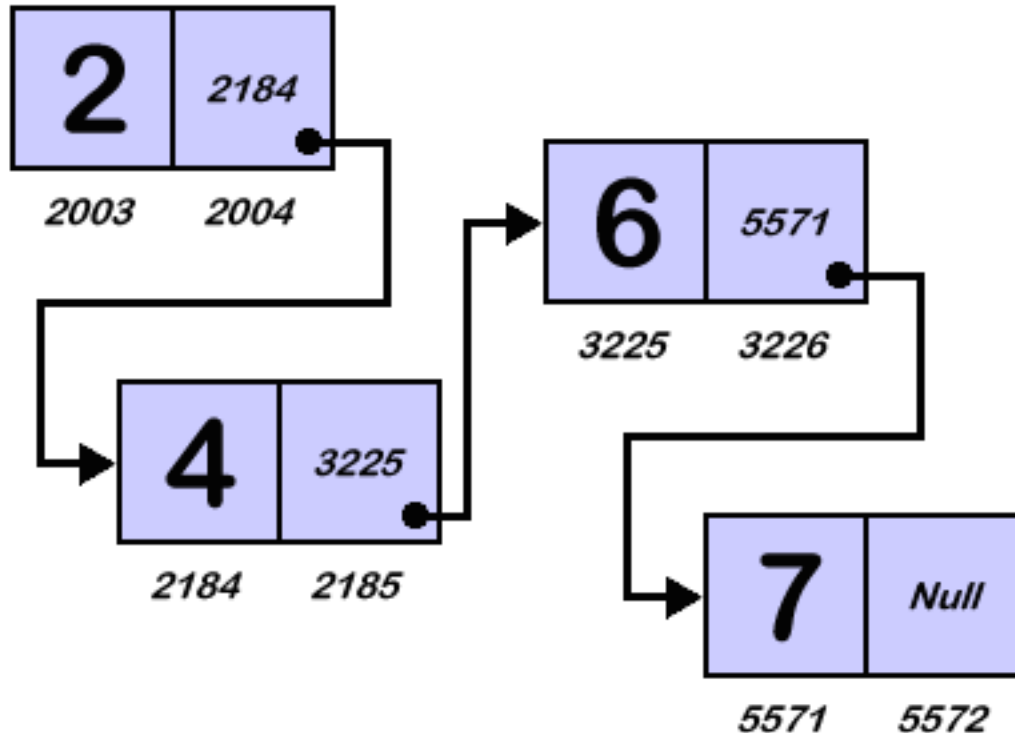
## Unit 2- Linked List

# Linked List

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



# Example of linked list



Let LIST be a Linear list.

LIST requires two arrays

INFO: information part

LINK: next pointer field of a node

START: contains the location of beginning of the list

NULL: next pointer sentinel which denotes the end of list

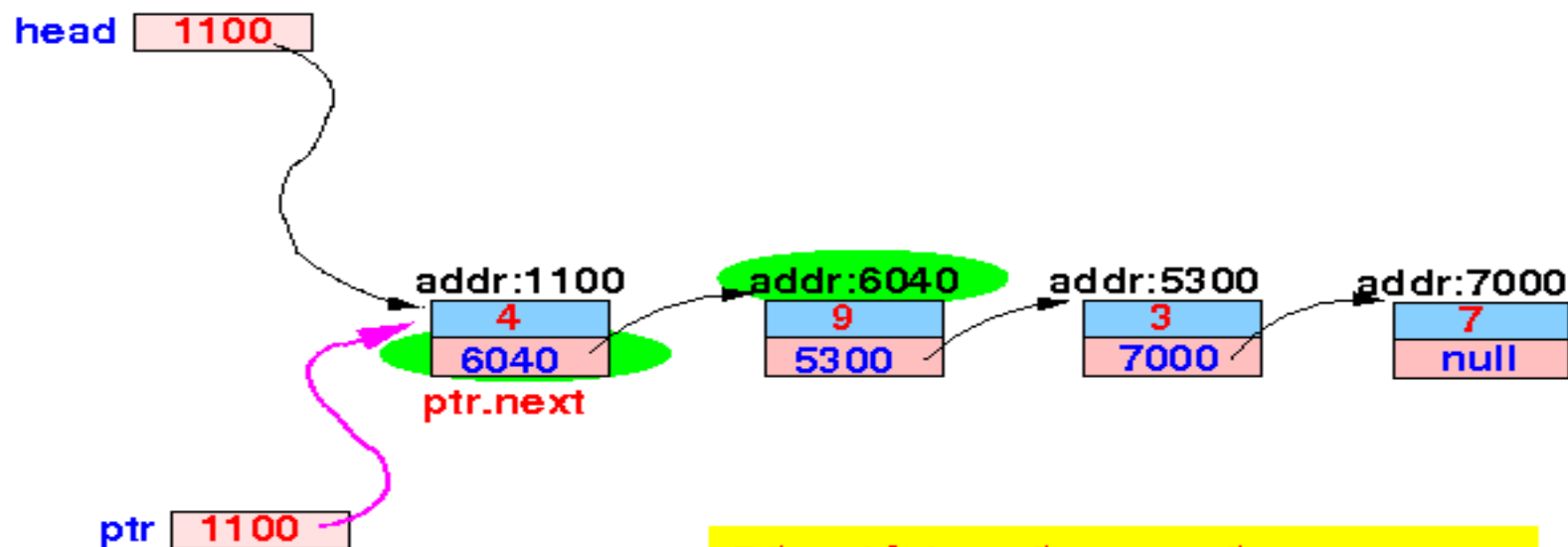
# Traversing a Linked List

Traverse : Process each node exactly once.

PTR: which points to the node that is currently being executed.

LINK[PTR]: points to the next node to be processed.

$\text{PTR} = \text{LINK}[\text{PTR}] \dots \dots \dots$  Moves the pointer to the next node



*Identifying the members:*

value 



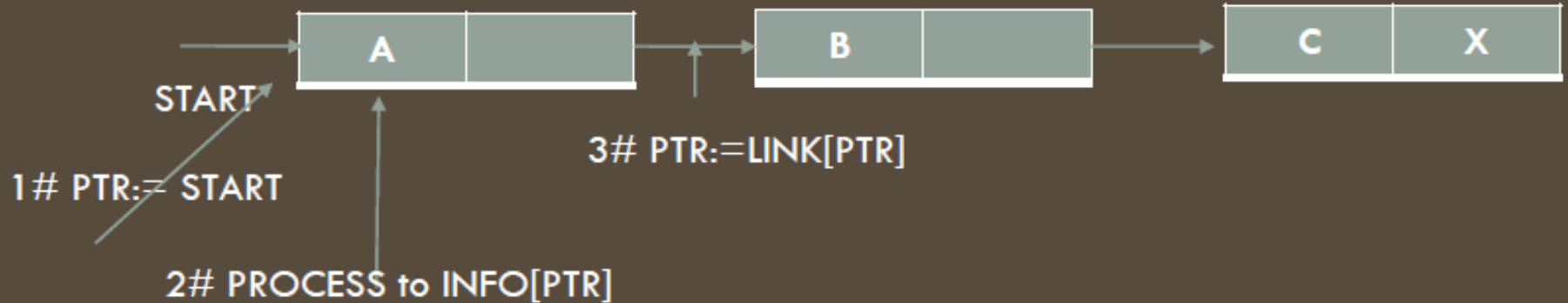
  
next

# Algorithm - Traverse

Traversing a Linked List: Let LIST be a Linked List in Memory . This algorithm traverses LIST, applying an operation PROCESS to each element of LIST. The variable PTR points to the node currently being processed.

1. Set  $PTR = START$ . [Initialize pointer PTR.]
2. Repeat step 3 and 4 while  $PTR \neq NULL$ .
3.     Apply PROCESS to  $INFO[PTR]$ .
4.     Set  $PTR = LINK[PTR]$ . [ PTR now points to the next node.]  
      [End of step 2 loop.]
5. Exit

# Algorithm - Traverse



# Searching a Linked List- List is Unsorted

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

LIST is a linked list in memory. This algorithm find the location LOC of the node where ITEM first appears in LIST, Set LOC=NULL.

1. Set PTR=START
2. Repeat Step 3 while PTR  $\neq$  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 Step 2 loop.]
4. [Search is unsuccessful.] Set LOC= NULL.
5. Exit



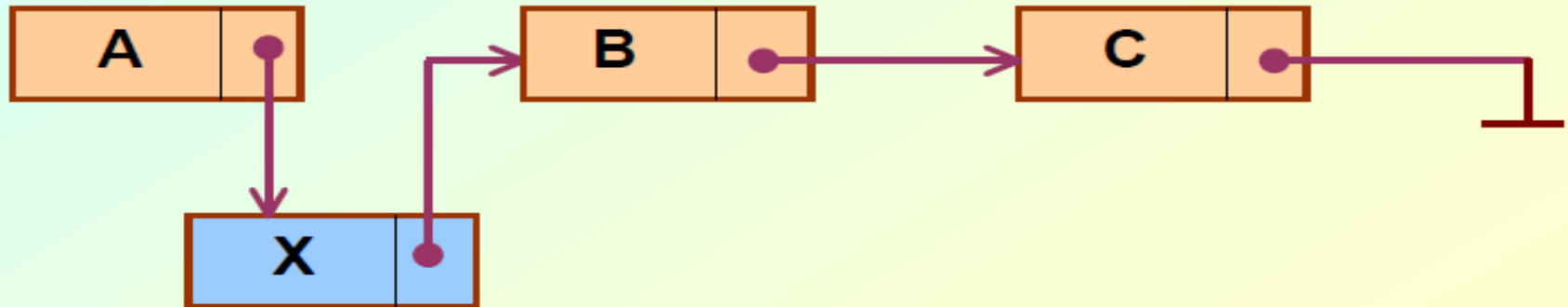
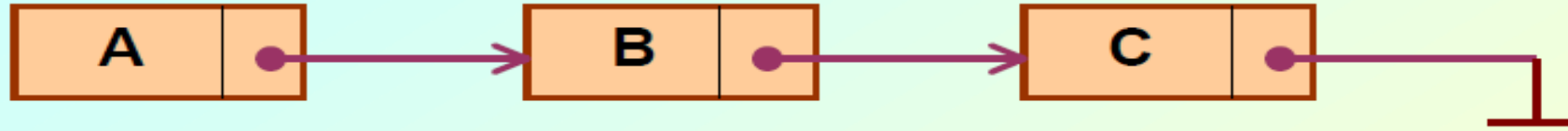
# LIST is Sorted

SRCHSL( INFO, LINK, START, ITEM, LOC)

LIST is a sorted list in memory .This algorithm finds the location LOC of the node where ITEM first appears in LIST, or sets LOC= NULL

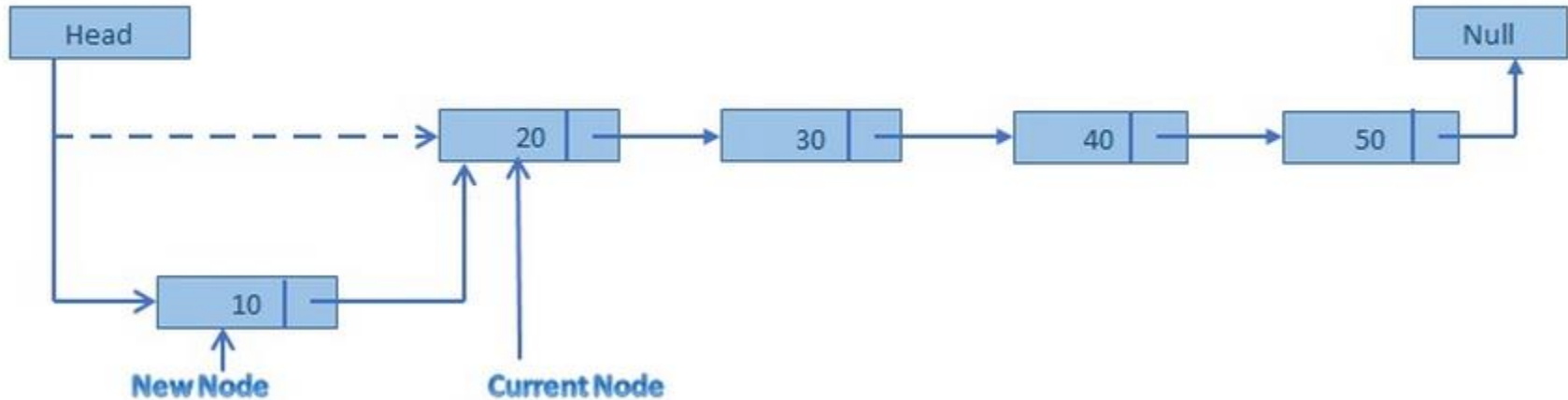
1. Set PTR= START
  2. Repeat step 3 while PTR  $\neq$  NULL
  3. if ITEM > INFO[PTR], then:
    - Set PTR= LINK[PTR]. [PTR now points to the next node]
    - Else if ITEM = INFO[PTR], then:
      - Set LOC= PTR, and Exit. [ Search is successful].
    - Else: Set LOC = NULL, and Exit. [INFO[PTR] now exceeds ITEM.]
- [End of IF Structure]
- [End of Step 2 loop.]
4. Set LOC = NULL.
  5. Exit.

# Insertion in Linked List



# Insertion – At the Front

Insertion at the front of list



# Insertion – At the Front/Beginning

## **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 the 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 given node



Suppose we are given the value of LOC where either LOC is the location of a node A in a linked list or  $LOC = \text{NULL}$ .

This insertion algorithm inserts ITEM into LIST so that ITEM follows node A or, when  $LOC = \text{NULL}$ , so that ITEM is the first node.

# Insertion – After a given node

**Let N denote the new node (whose location is NEW).**

If  $LOC = NULL$ , then N is inserted as the first node in LIST as in previous algorithm.

Otherwise, we let node N point to node B (which originally followed node A) by the assignment:

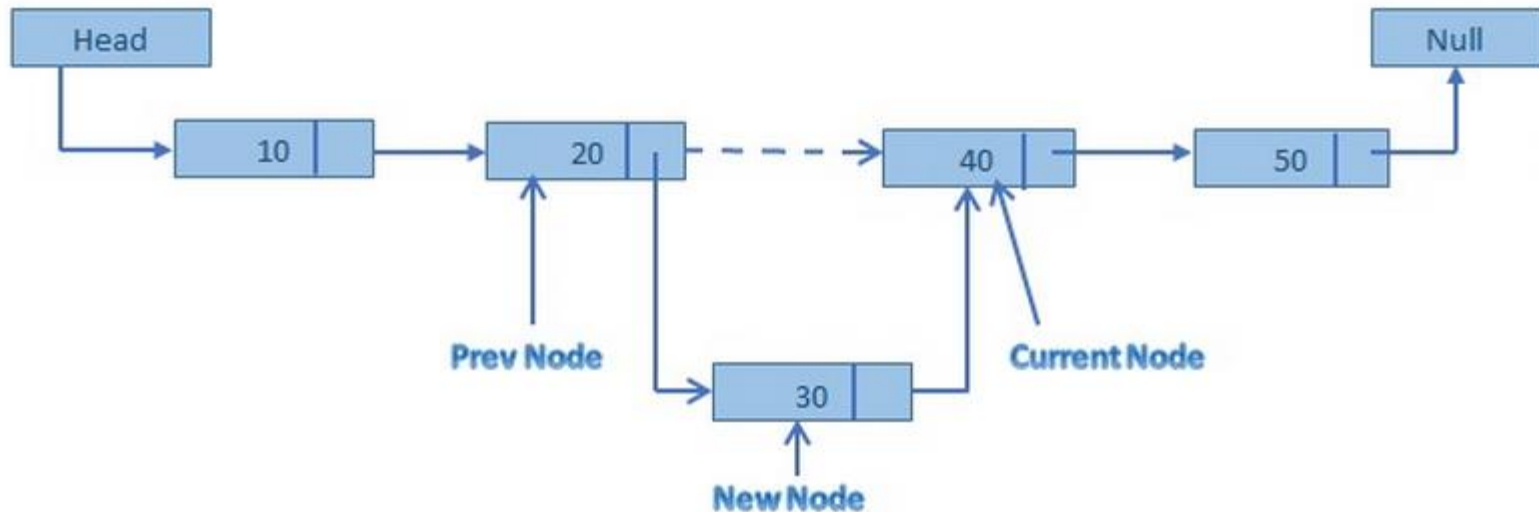
$$\text{LINK}[\text{NEW}] = \text{LINK}[\text{LOC}]$$

And we let node A point to the new node N by the assignment:

$$\text{LINK}[\text{LOC}] = \text{NEW}$$

# Insertion – After a given node

Insertion Node in given location Linked List



# Insertion – After a given node

## **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 the 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.

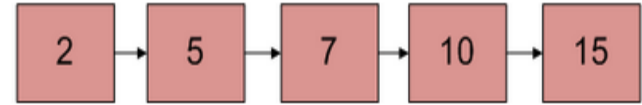


# Insertion – In a sorted list

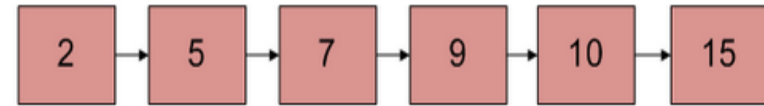
Let input linked list is sorted in increasing order.

- 1) If Linked list is empty then make the node as head and return it.
- 2) If value of the node to be inserted is smaller than value of head node then insert the node at start and make it head.
- 3) In a loop, find the appropriate node after which the input node (let 9) is to be inserted. To find the appropriate node start from head, keep moving until you reach a node GN (10 in the below diagram) who's value is greater than the input node. The node just before GN is the appropriate node (7).
- 4) Insert the node (9) after the appropriate node (7) found in step 3.

Initial Linked List



Linked List after insertion of 9



# Insertion – In a Sorted List

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

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

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

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

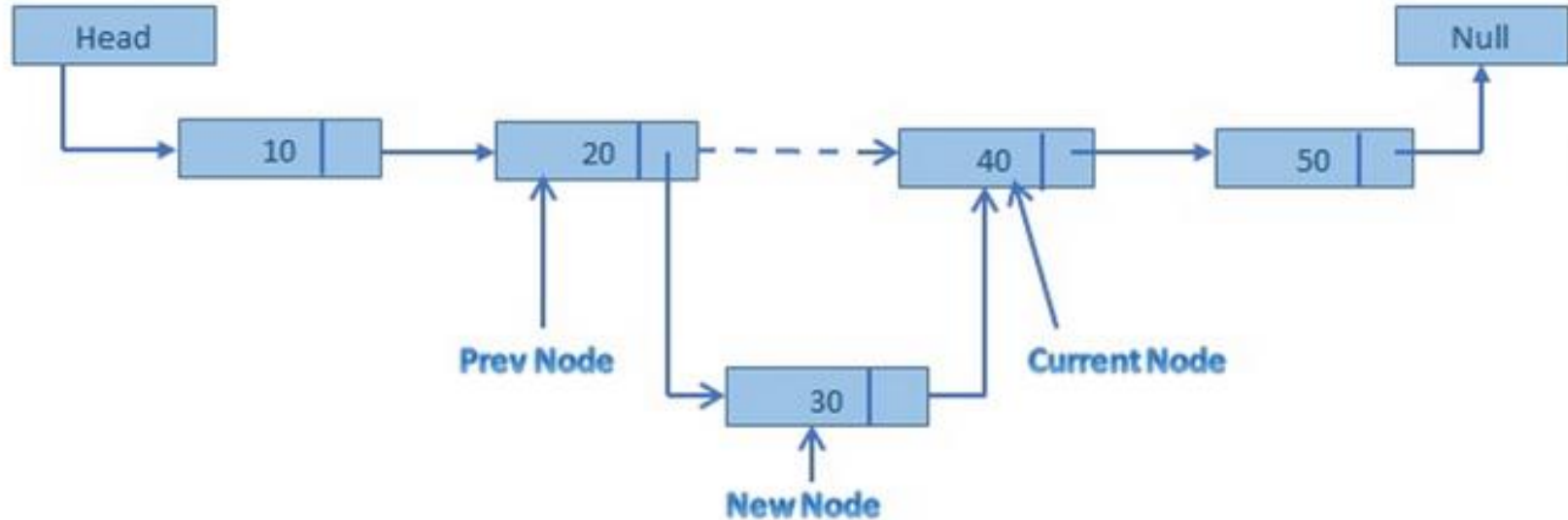
Thus SAVE and PTR are updated by the assignments:

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

# Insertion – In a Sorted List

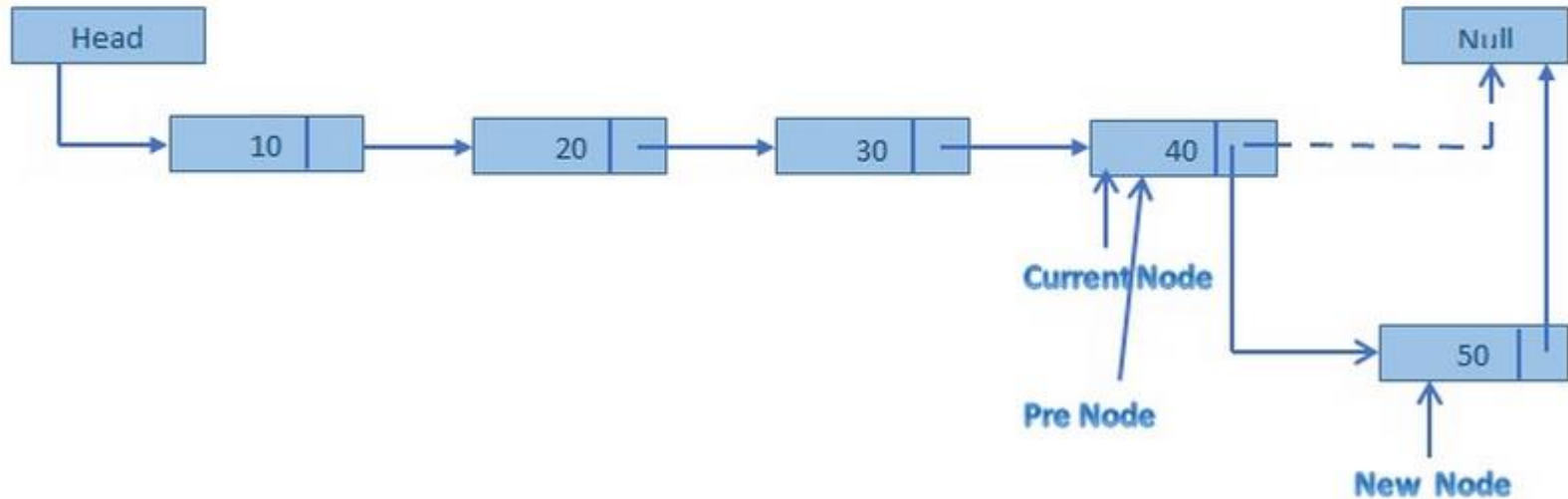
The traversing continues as long as **INFO[PTR] > ITEM**, or in other words, the traversing stops as soon as **ITEM ≤ INFO[PTR]**. Then PTR points to the node B, so SAVE will contain the location of the node A.

# Insertion – In a Sorted List



# Insertion – At the End

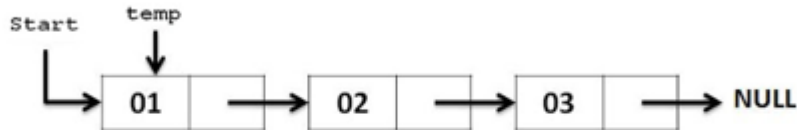
Insertion at the end of the list.



# Deletion in Linked List

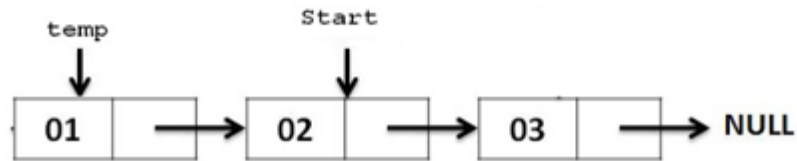
## Step 1 : Store Current Start in Another Temporary Pointer

```
temp = start;
```



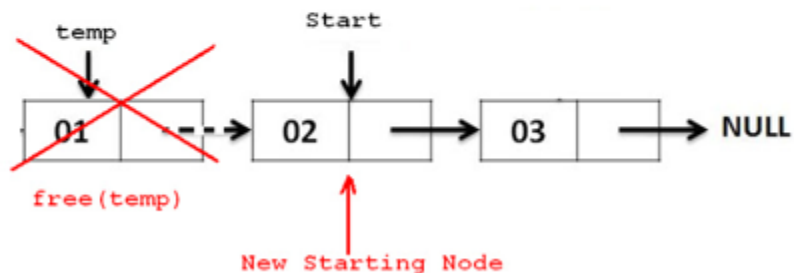
## Step 2 : Move Start Pointer One position Ahead

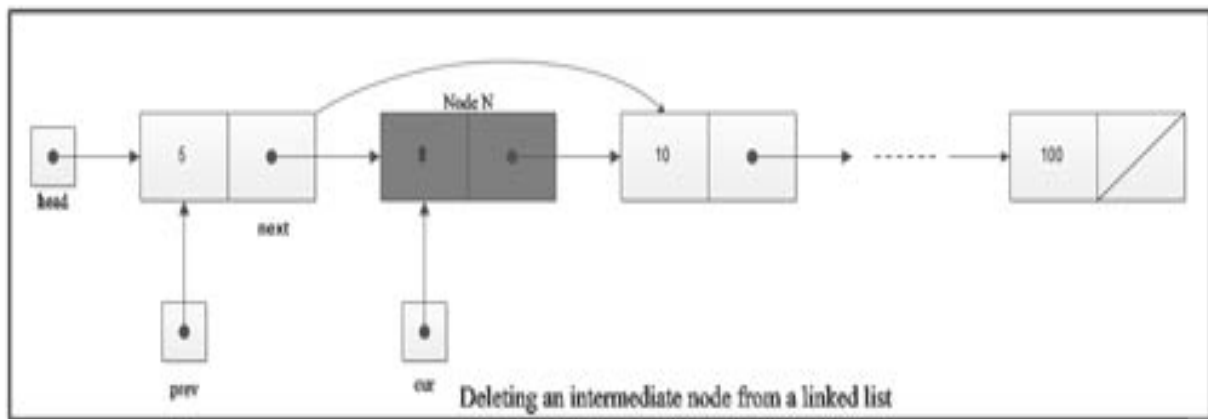
```
start = start->next;
```



**Step 3 :** Delete temp i.e Previous Starting Node as we have Updated Version of Start Pointer

```
free(temp);
```







# Algorithm for Deletion:

Algo:

DEL( INFO, LINK, START, AVAIL, CUR, PREV)

This algorithm deletes the node N with location CUR. PREV is the location of the node which precedes N or, when N is the first node, PREV= NULL

1. If PREV= NULL, then:

Set START = LINK[START].....Delete first Node

Else

Set LINK[PREV]=LINK[CUR].....Delete Node N.

End of if structure.

2. [Return deleted node to the AVAIL list]

Set LINK[CUR]= AVAIL and AVAIL= CUR.

3. Exit.

# Deleting a Node with a given ITEM of information

FINDB( INFO, LINK, START, ITEM, CUR, PREV)

This procedure finds the location CUR of the first node N which contains ITEM and the Location PREV of the node preceding N. if item does not appear in the list, then the procedure sets CUR= NULL, and if ITEM appears in the first node PREV= NULL

1. [List Empty?] if START= NULL, then:  
Set CUR= NULL and PREV= NULL and Return.  
END of if Structure
2. [ ITEM in the first Node?] if INFO[START]= ITEM, then:  
Set CUR= START and PREV=NULL and Return.  
[ End of if structure]
3. Set SAVE=START and PTR= LINK[START]. .....Initialize pointers
4. Repeat steps 5 and 6 while PTR  $\neq$  NULL.
5. if INFO[PTR]= ITEM, then:  
Set CUR=PTR and PREV=SAVE, and Return.  
End of if Structure
6. Set SAVE= PTR and PTR=LINK[PTR].....Update pointers  
END of Step 4 Loop
7. Set CUR= NULL.....Search Unsuccessful
8. Return



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

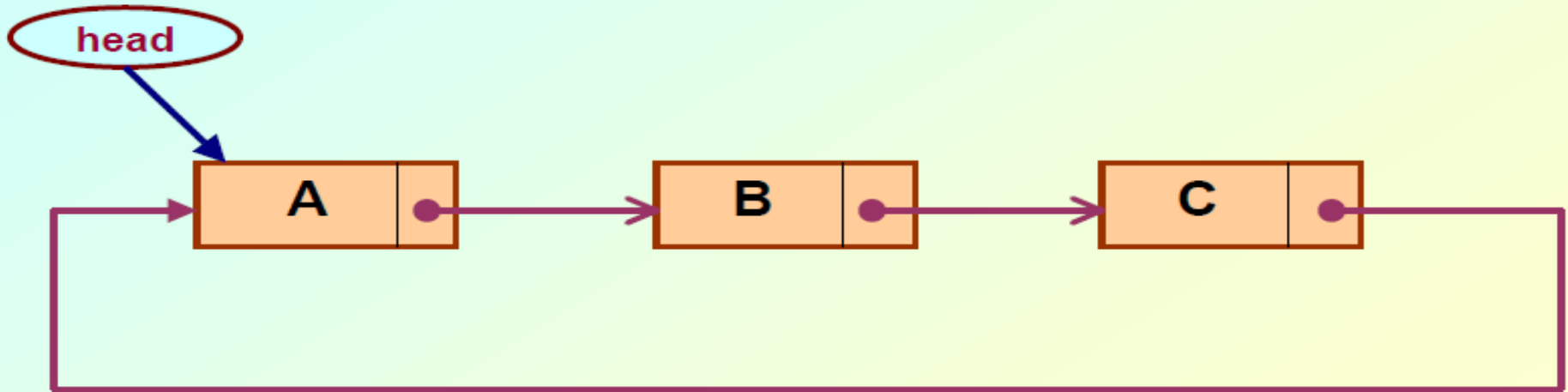
This algorithm deletes from a linked list the first node N which contains the given ITEM of information

1. [find the location of N and its preceding node.]  
    Call FINDB(INFO, LINK, START, ITEM, CUR, PREV)  
    If CUR= NULL, then write: ITEM not in the list, and EXIT.
2. If CUR= NULL , then write : ITEM not in the list, and Exit.
3. [Delete Node]  
    if PREV= NULL, then  
        SET START= LINK[START].....Delete first node  
    Else:  
        Set LINK[PREV]=LINK[CUR]  
    End of if structure
4. [ Return deleted node to the AVAIL list]  
    Set LINK[CUR]= AVAIL and AVAIL = CUR

# Circular Linked List

## – Circular linked list

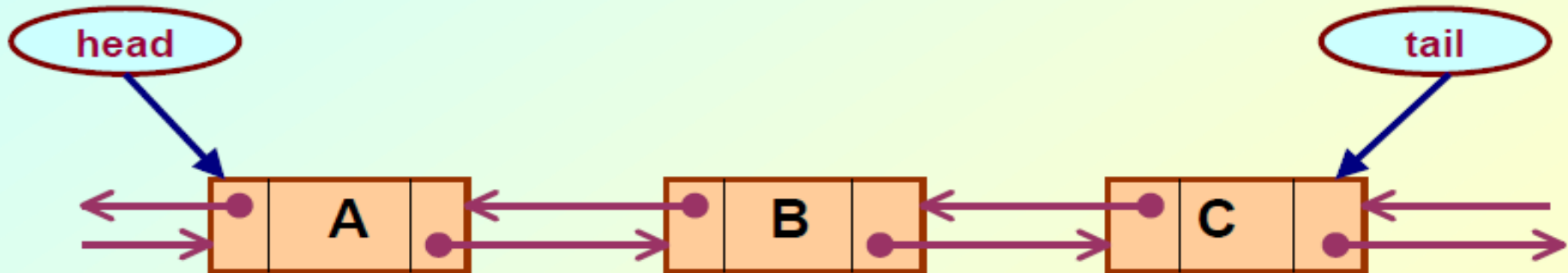
- The pointer from the last element in the list points back to the first element.



# Doubly Linked List

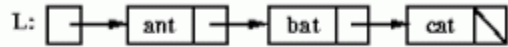
## – Doubly linked list

- Pointers exist between adjacent nodes in both directions.
- The list can be traversed either forward or backward.
- Usually two pointers are maintained to keep track of the list, *head* and *tail*.

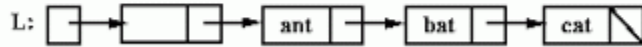


# Header Linked List

Without a header node:



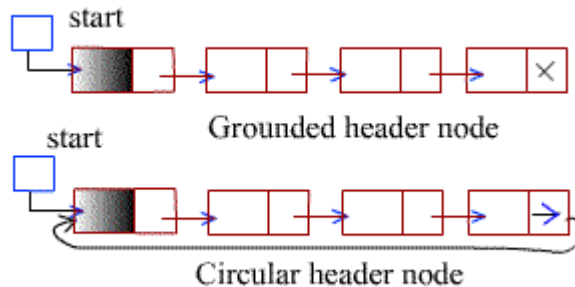
With a header node:




Header Linked List:

Which Contains a special node called the Header Node at the beginning of the list.

1. *Grounded Header List*: where the last node contains the NULL pointer
2. *Circular Header List*: is a header list where the last node points to the Header node.



- 
1. START always points to the header node
  2. LINK[START]=NULL indicated grounded header list is empty
  3. LINK[START]=START indicated that a circular header list is empty
  4. LINK[START] is first node.. Not START node
  4. Circular Header list is frequently used instead of ordinary Linked List because many operations are much easier to state and implement using Header List
  5. Two properties of Circular Header Lists:
    - The NULL pointer is not used and hence all pointers contain valid addresses
    - Every ordinary node has a predecessor, so the first node may not require a special case.