PCCST303DATA STRUCTURES AND ALGORITHMS

PREPARED BY SHARIKA T R, ASSISTANT PROFESSOR, ASIET

Course Outcome

At the end of the course students should be able to:

Course Outcome		Bloom's Knowledge Level (KL)
CO1	Identify appropriate data structures for solving real world problems.	**K3**
CO2	Describe and implement linear data structures such as arrays, linked lists, stacks, and queues.	**K3**
CO3	Describe and Implement non linear data structures such as trees and graphs.	**K3**
CO4	Select appropriate searching and sorting algorithms to be used in specific circumstances.	**K3**

Note: *K1- Remember, K2- Understand, K3- Apply, K4- Analyse, K5- Evaluate, K6- Create*

Module 2- Linked List and Memory Management

Singly Linked List - Operations on Linked List, Stacks and Queues using Linked List, Polynomial representation using Linked List; Doubly Linked List; Circular Linked List;

Memory allocation - First-fit, Best-fit, and Worst-fit allocation schemes; Garbage collection and compaction.

ARRAY

A data structure where homogenous elements are stored in consecutive memory locations.

Size of the array must be defined in advance. Sometimes its not possible to predict the size.

- If size is too large wastage of memory
- If its too small requires resizing Inserting and deleting an item requires shifting of the remaining elements

Memory allocation:

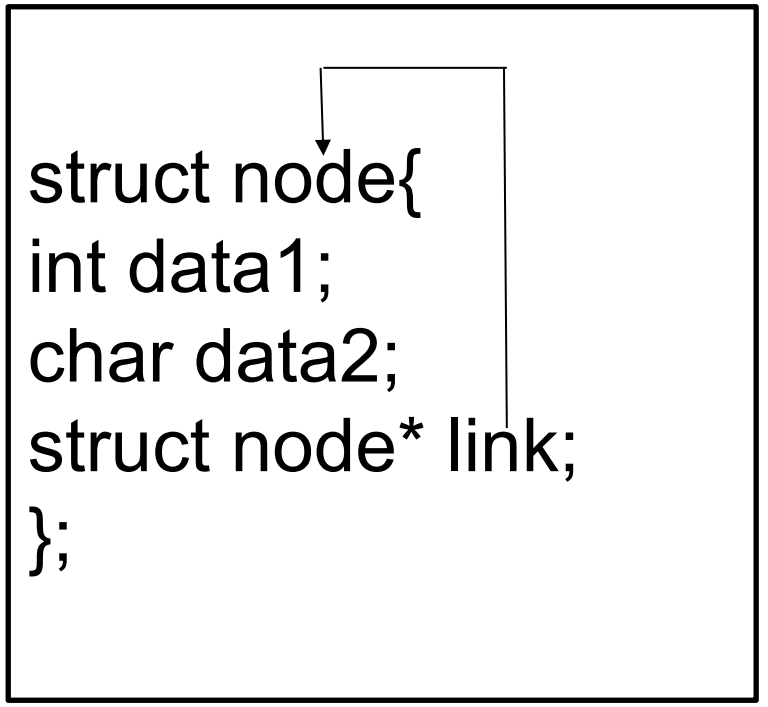
1. Static: Memory allocated at compile time Once memory is allocated it cannot be extended any more
2. Dynamic: Memory allocated at run time memory required can be varied during runtime, its is allocated based on the requirements only

Self Referential Structures

Self Referential structures are those structures that have one or more pointers which point to the same type of structure, as their member.

In other words, structures pointing to the same type of structures are self-referential in nature

One or more pointers points to the structure of same type



Example

```
struct node{
int data1;
char data2;
struct node* link;
};
void main()
{
struct node *ob;
}
```

- link' is a pointer to a structure of type 'node'.
- Hence, the structure 'node' is a self-referential structure with 'link' as the referencing pointer.
- An important point to consider is that the pointer should be initialized properly before accessing, as by default it contains garbage value.

Types of Self Referential Structures

1. Self Referential Structure with Single Link
2. Self Referential Structure with Multiple Links

Applications

Self referential structures are very useful in creation of other complex data structures like:

1. Linked Lists
2. Stacks
3. Queues
4. Trees
5. Graphs etc

Dynamic memory allocation

Creating and maintaining dynamic data structures requires dynamic memory allocation

Dynamic allocation allows a program to create space for a structure whose size isn't known until runtime.

Functions malloc and free, and operator sizeof, are essential to dynamic memory allocation.

The malloc function allocate memory and return a void pointer to it; NULL is returned if the requested allocation could not be performed

Function malloc is normally used with the sizeof operator.

```
struct Node {
int data;
struct Node *next;
};
void main() {
struct Node newnode = (struct Node*)malloc(sizeof(struct Node));
}
```

evaluates sizeof(struct Node) to determine the size in bytes of a structure of type struct Node,

allocates a new area in memory of that number of bytes and stores a pointer to the allocated memory in variable newnode.

Dynamic memory allocation

The free function deallocates memory.

- The memory is returned to the system so that it can be reallocated in the future.

To free memory dynamically allocated by the preceding malloc call, use the statement

free(newnode);

Memory Leaks

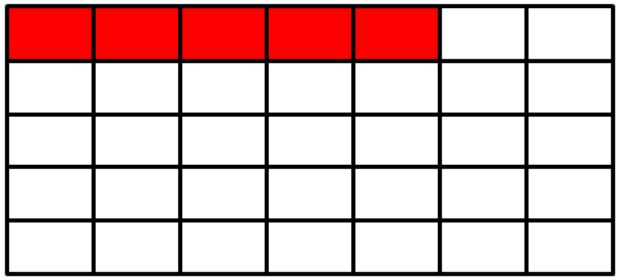
- Memory that has been dynamically allocated but has not been freed and is no longer in use.

Linked List

No need of consecutive memory location

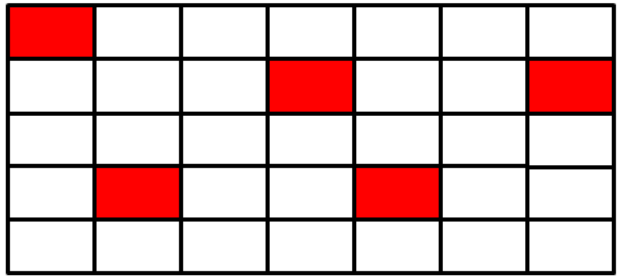
Can grow as long as memory is available

STATIC



Arrays need a contiguous Block of memory.

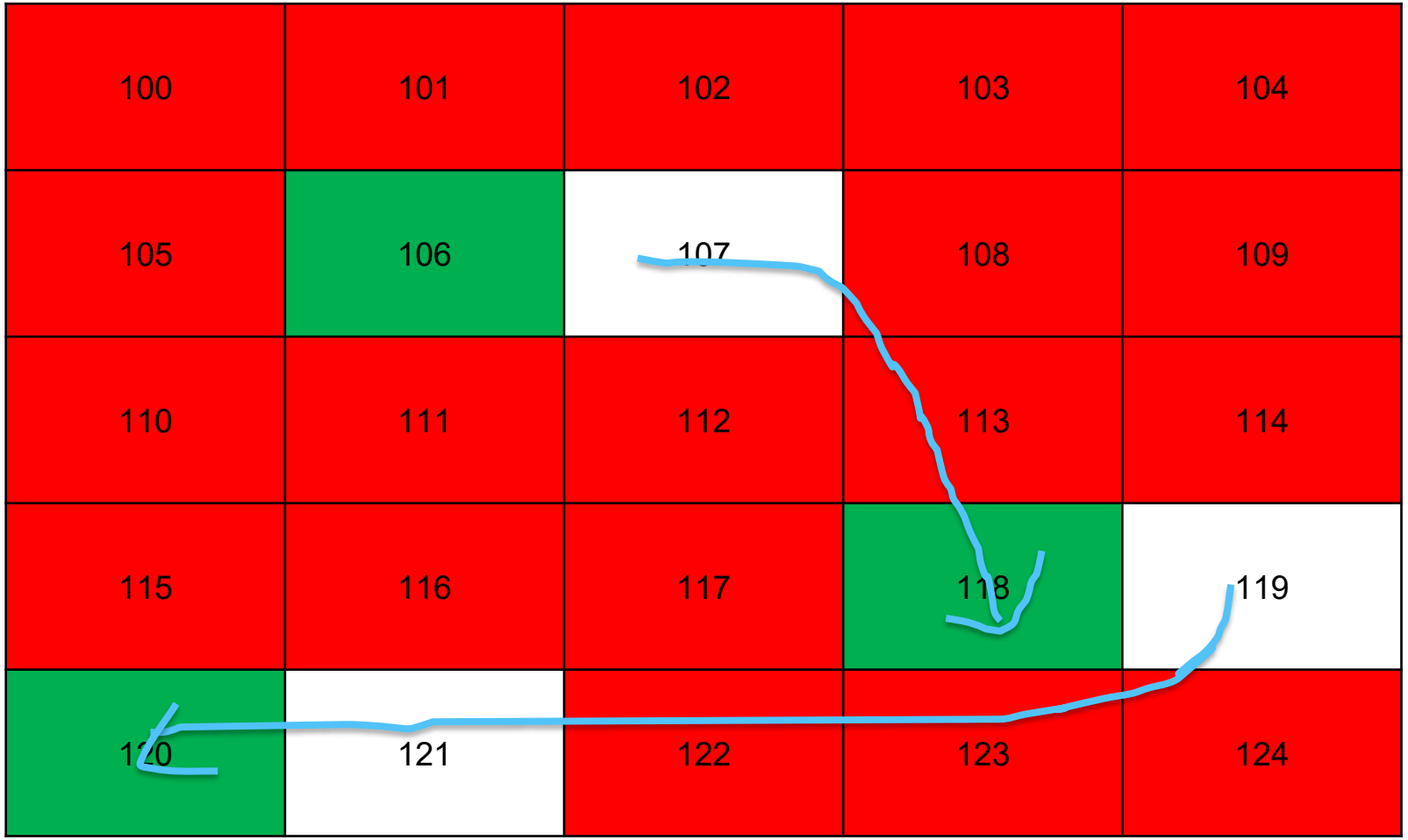
DYNAMIC

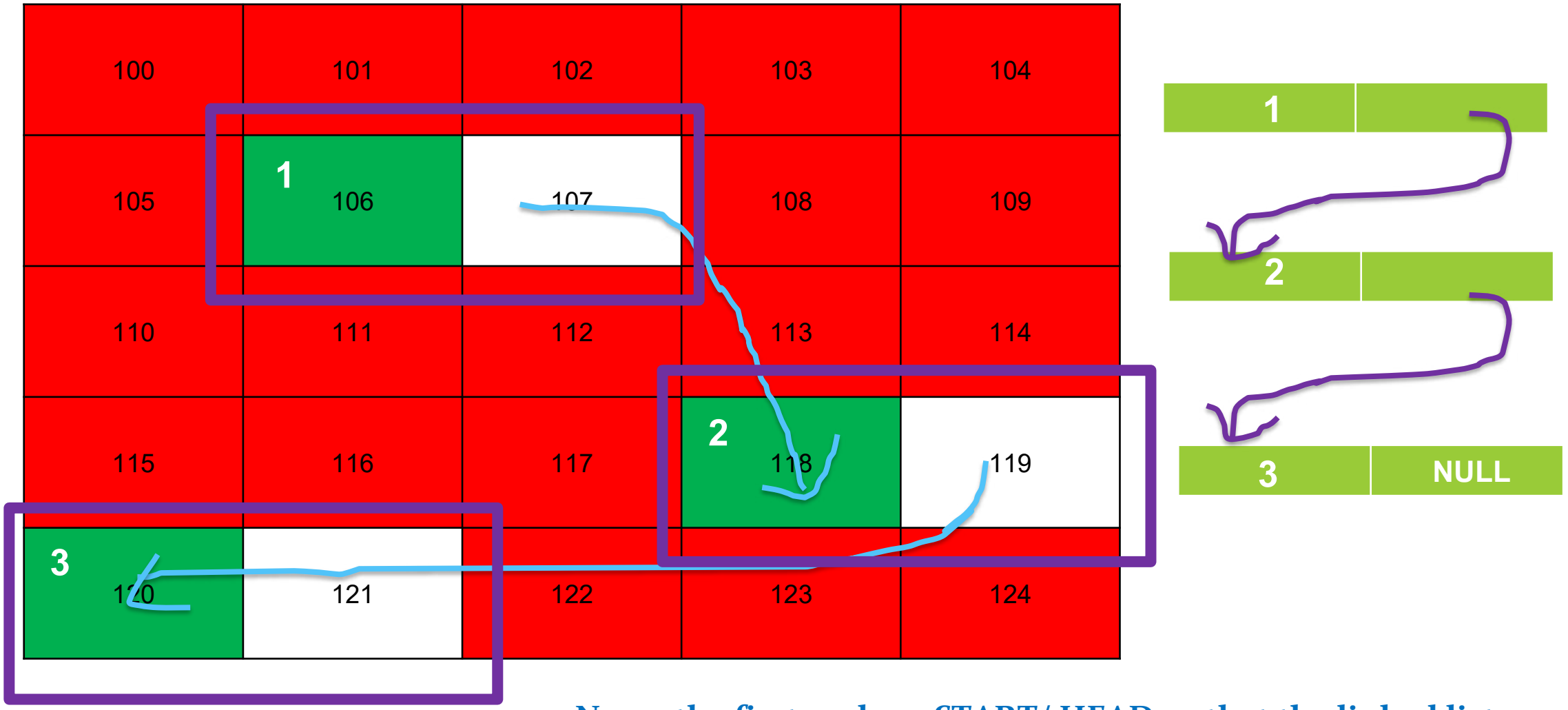


Linked Lists don't need to be contiguous In memory. They can grow Dynamically.

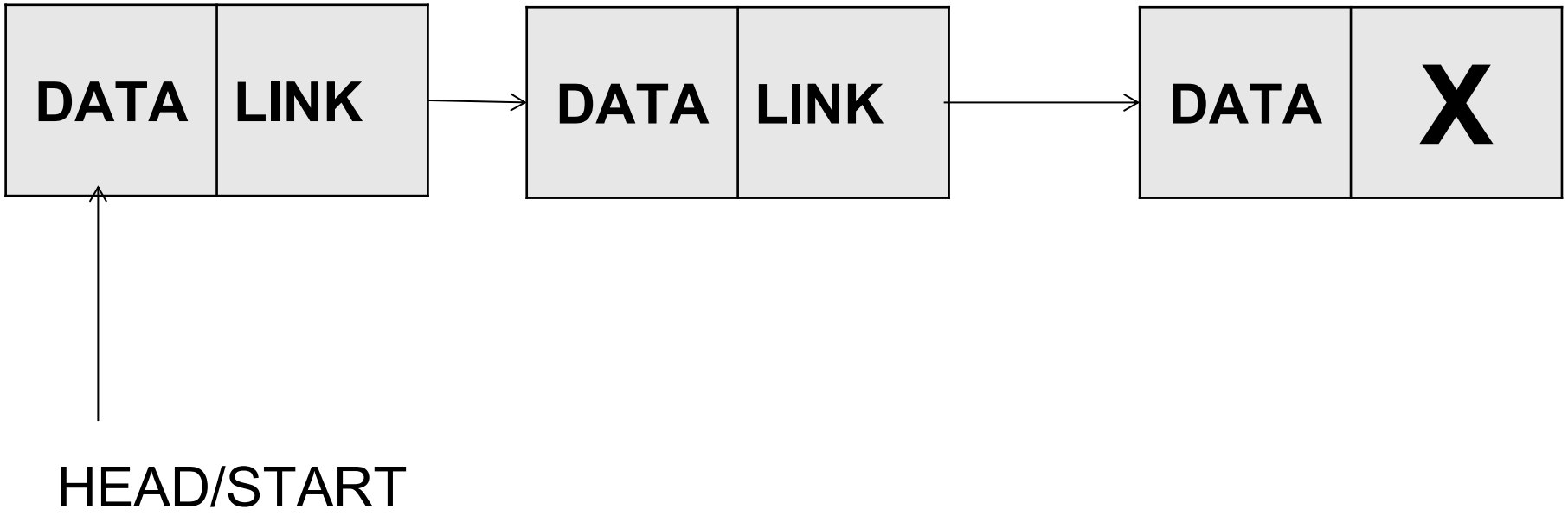
100	101	102	103	104
105	106	107	108	109
110	111	112	113	114
115	116	117	118	119
120	121	122	123	124

100	101	102	103	104
105	106	107	108	109
110	111	112	113	114
115	116	117	118	119
120	121	122	123	124


100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124



Name the first node as START/ HEAD so that the linked list can easily be accessed


DATA
LINK
DATA
LINK
DATA
X
HEAD/START

A Linked list is an ordered collection of finite, homogeneous data elements called nodes where linear order is maintained by means of links or pointers.

Nodes

- Linked list consist chain of elements, in which each element is referred to as a **node.**
- A node consists of two parts:
  - **Data**: Refers to the information held by the node
  - **Link**: Holds the address of the next node in the list

Head/Start

- Contains a pointer to the first data node in the list (or a null pointer if the list is empty).

Linked list Advantages & Disadvantages

Advantages

- Size of the list need not be determined in advance. A linked list can grow or shrink in size as the program runs. The nodes are dynamically allocated when needed and it is deallocated when it is no longer needed.
- Inserting or deleting a node from a linked list is very easy.

Disadvantage

- Extra space is needed to store the pointer associated with each element.
- Access to internal list elements may take more time than with an array, since to find an element we need to start searching from the first element.

Classification of linked List

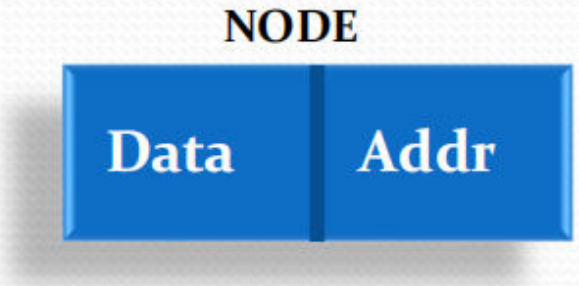
1. Single Linked List
2. Circular Linked List
3. Double linked List

Linked list / simple linked list / singly linked list

Representation of a node

- A node can be implemented by using the self-referential structure.
- Self referential structure:
  - one member of this structure is a pointer that points to the structure itself
- Structure can be defined as follows

```
struct node
{
    int data;
    struct node *nextaddr;
};
```



- The above structure definition has two parts, first part contains information part (data) and second part contains the address of the next node.

- A structure variable of type node can be created by using

```
struct node *newnode;
```

Single linked list

- Each node contains
  - One or more data
  - one link which points the subsequent node in the list.
- Therefore, every node in the linked list has a link field that stores the address of the next node in sequence.
- The last node in a linked list does not point to any other node. Therefore, it points to NULL.

Operations on a single Linked List

1. Insertion
2. Traversing
3. Deletion
4. Copy
5. Merging
6. Searching
7. Reversing
8. Sorting

SIMPLE LINKED LIST - INSERTION

- A linked list can be created by inserting elements at the end of a linked list
- Elements can be inserted at various positions in a linked list:
- The various positions are:
  - Insert at front(as a first element)
  - Insert at end(as a last element)
  - Insert at particular position
- Before insertion, it is required to make space for the node dynamically.
  - malloc() function make the space for the node variable

SIMPLE LINKED LIST - Insertion at end

- inserts a node at the end of the list if not null.
- If the list is empty it will form a list by creating a new node and that node will be initialized as the header of the list.

LL - Insertion at end

- Initially, list is empty
- The pointer HEAD points to NULL

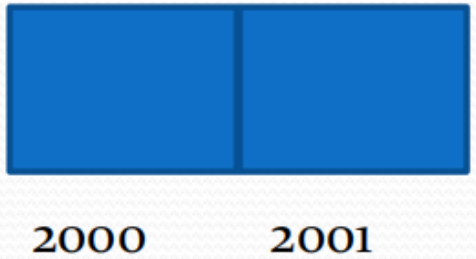
HEAD = NULL

LL - Insertion at end

Create first node.

1. Get space from memory

HEAD = NULL

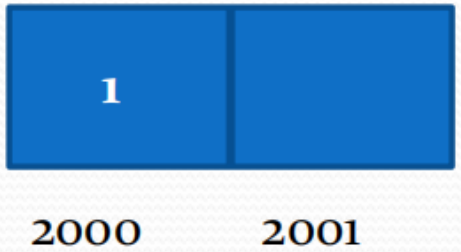


LL - Insertion at end

Create first node.

1. Get space from memory
2. Store the data in the node

HEAD = NULL



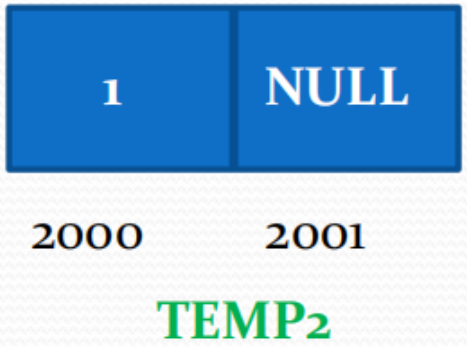
LL - Insertion at end

Temp2=malloc(node)
Temp2.data=item;
Temp2.nextaddr=NULL;

Create first node.

1. Get space from memory
2. Store the data in the node
3. Store NULL in the address field

HEAD = NULL

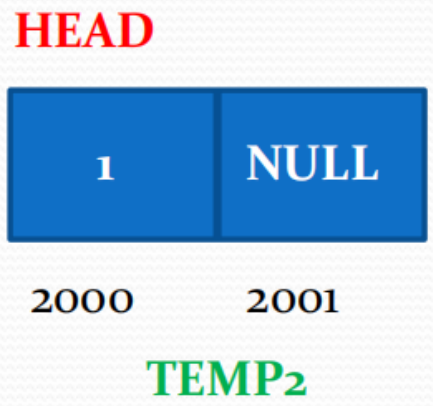


LL - Insertion at end

```
Temp2=malloc(node)
Temp2.data=item;
Temp2.nextaddr=NULL;
If(head=NULL) then
      head=temp2
```

Create first node.

1. Get space from memory
2. Store the data in the node
3. Store NULL in the address field
4. If list is empty – name that node as HEAD



LL - Insertion at end

Create next node.

1. **Get space from memory**
2. Store the data in the node
3. Store NULL in the address field
4. If list is empty – name that node as HEAD

LL - Insertion at end

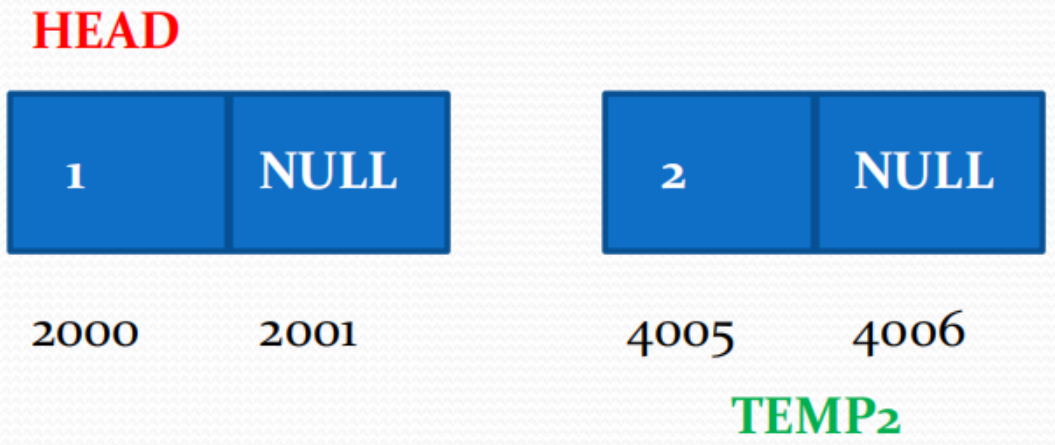
Create next node.

1. Get space from memory
2. **Store the data in the node**
3. Store NULL in the address field
4. If list is empty – name that node as HEAD

LL - Insertion at end

Create next node.

1. Get space from memory
2. Store the data in the node
3. **Store NULL in the address field**
4. If list is empty – name that node as HEAD



LL - Insertion at end

Create next node.

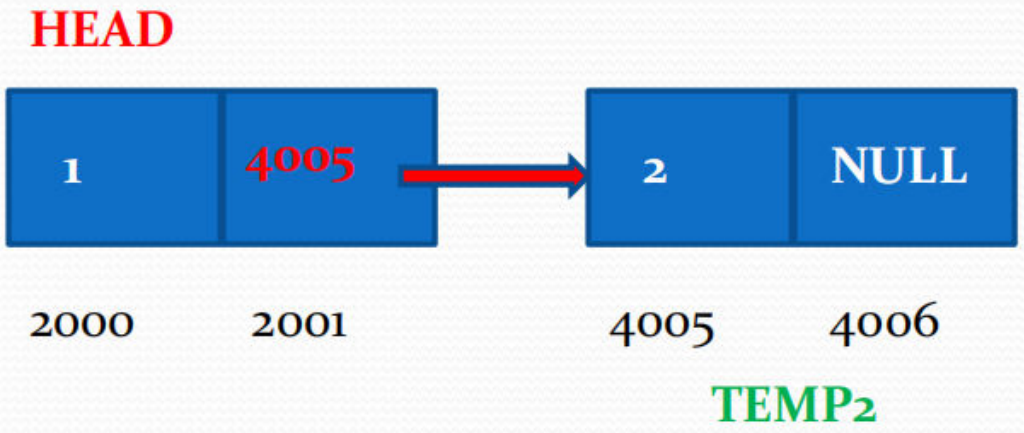
1. Get space from memory
2. Store the data in the node
3. Store NULL in the address field
4. If list is empty – name that node as HEAD
5. Else **store the address of this node in the address part of the previous node**

LL - Insertion at end

```
Temp2=malloc(node)
Temp2.data=item;
Temp2.nextaddr=NULL;
If(head=NULL) then
        Head=temp2
else
        Head.nextaddr=temp2
EndIf
```

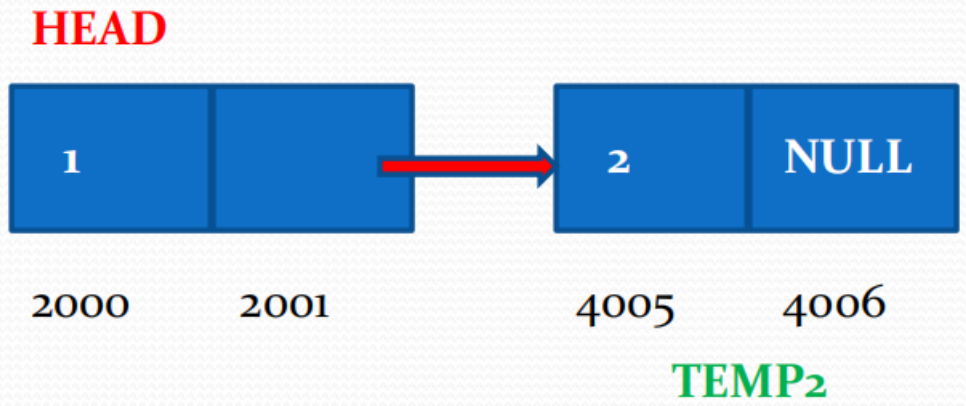
Create next node.

1. Get space from memory
2. Store the data in the node
3. Store NULL in the address field
4. If list is empty – name that node as HEAD
5. Else store the address of this node in the address part of the previous node



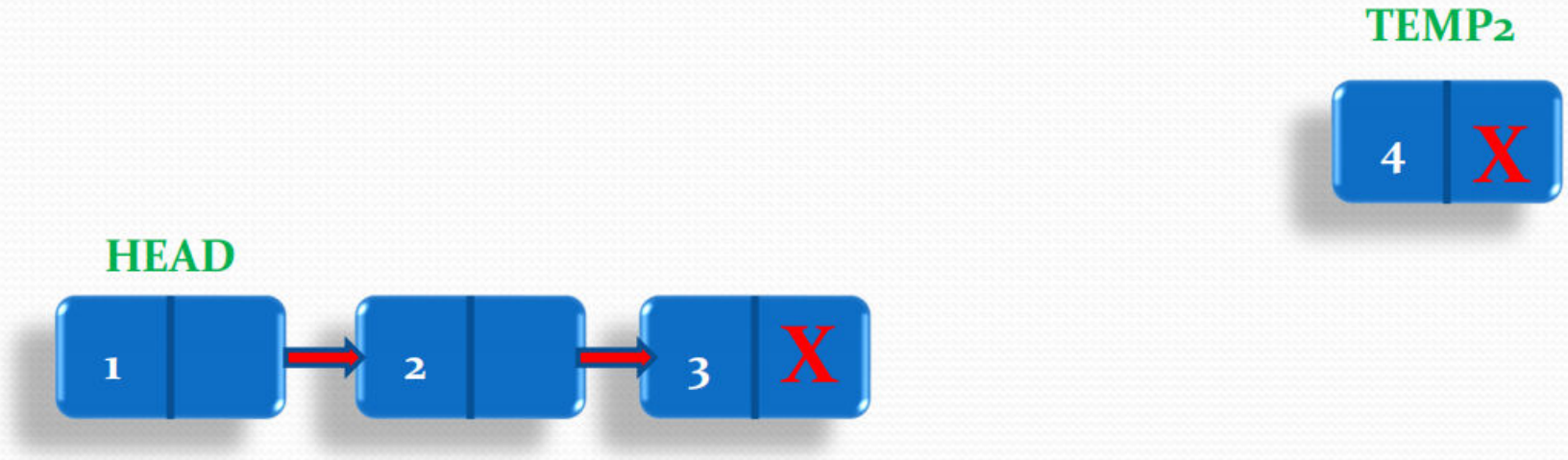
LL - Insertion at end

This process continues until the required number of nodes are created.



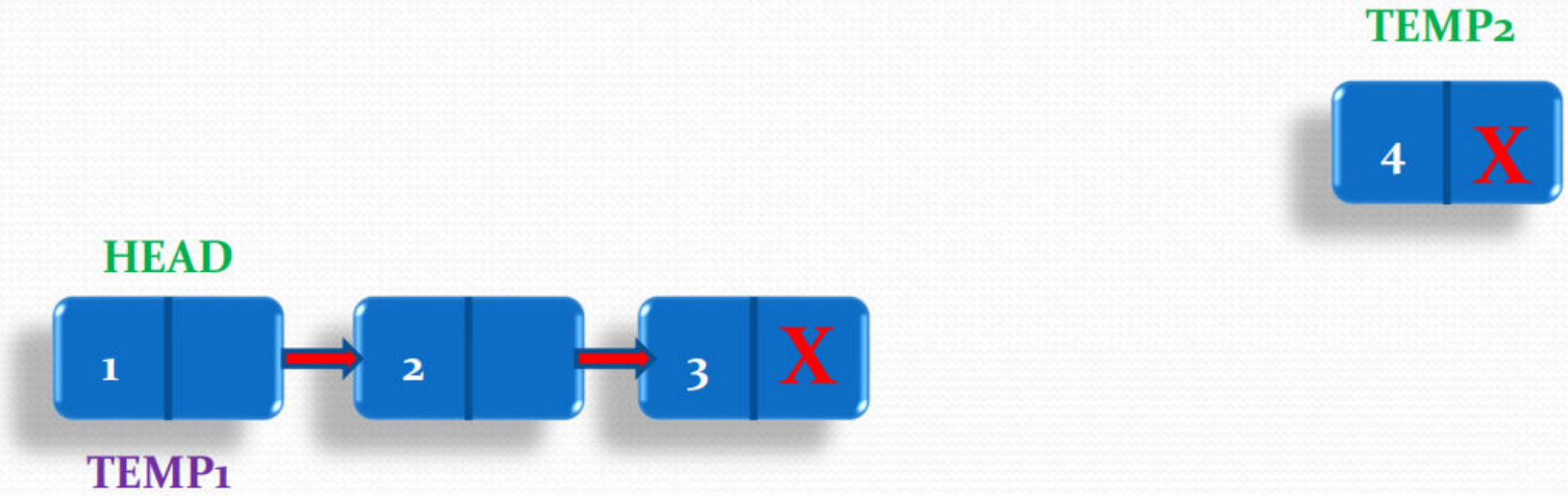
LL - Insertion at end

Insert 4 at the end of the list



LL - Insertion at end

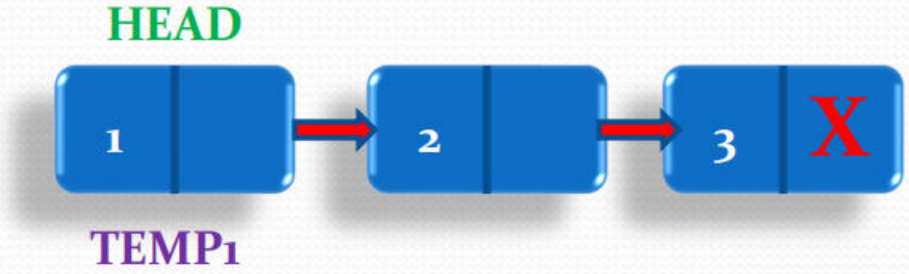
Set TEMP1= HEAD



LL - Insertion at end

While (TEMP1.nextaddr !=NULL)

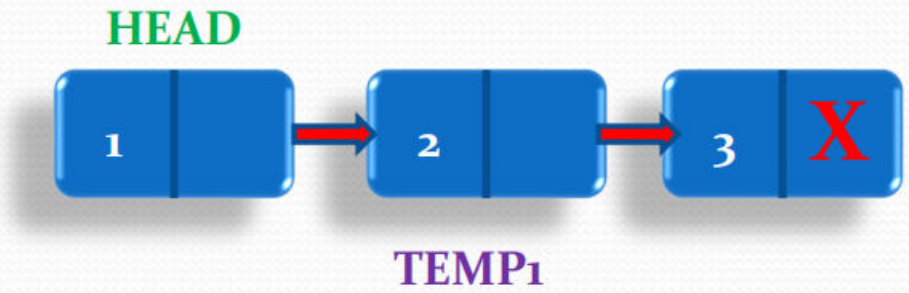
TEMP1=TEMP1.nextaddr



LL - Insertion at end

While (TEMP1.nextaddr !=NULL)

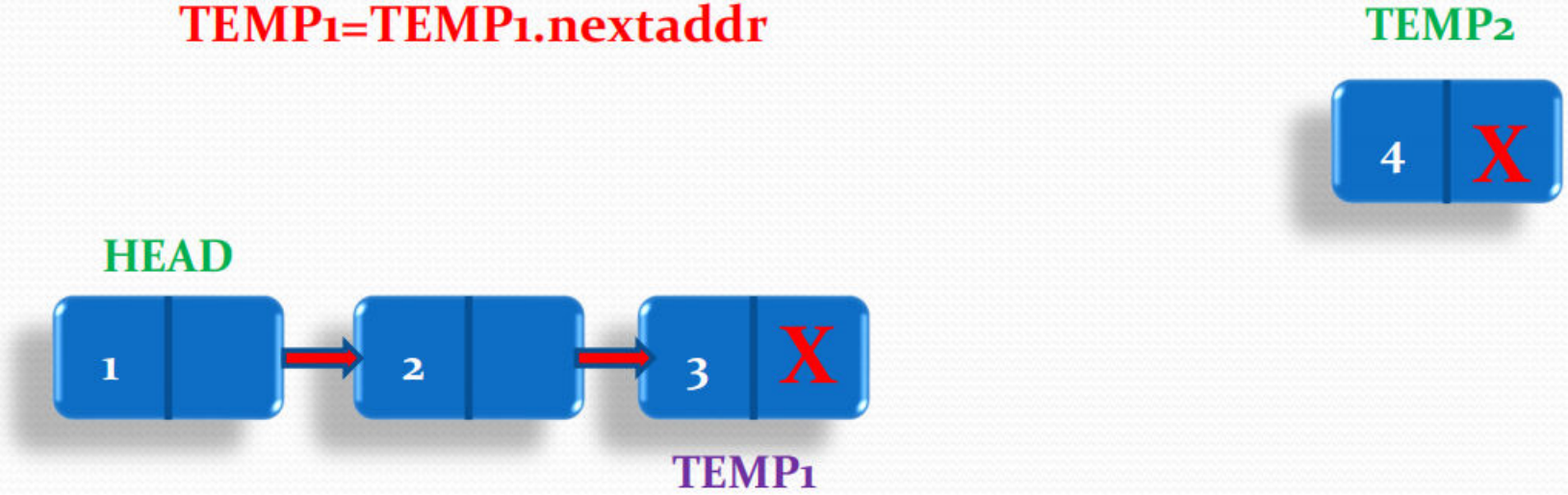
TEMP1=TEMP1.nextaddr



LL - Insertion at end

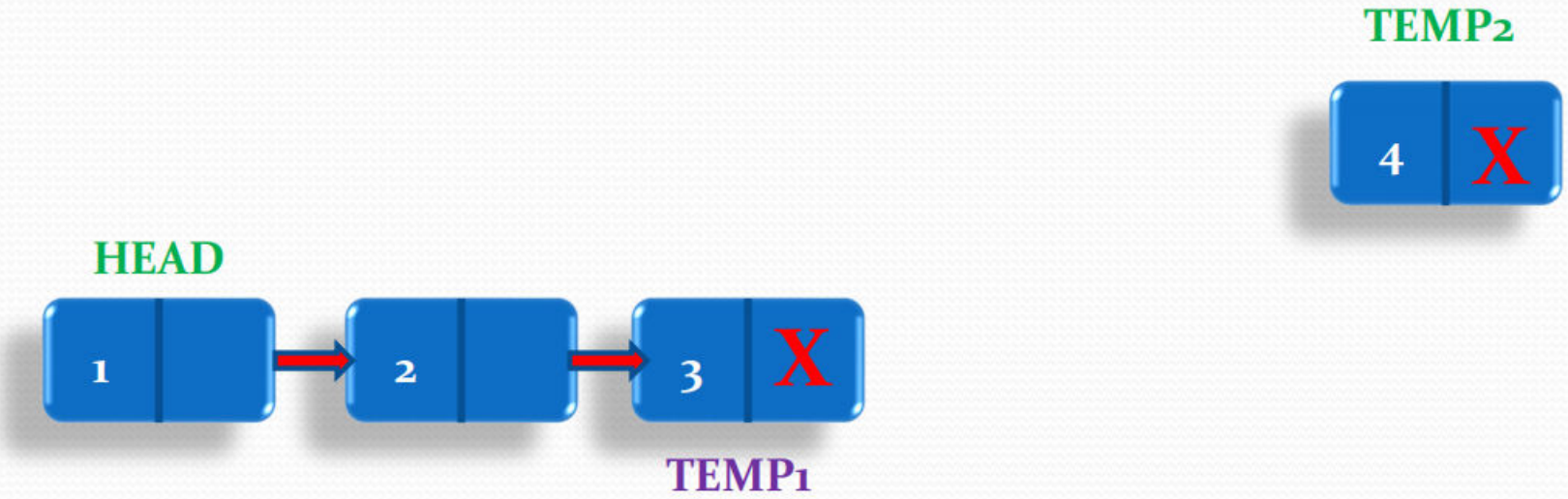
While (TEMP1.nextaddr !=NULL)

TEMP1=TEMP1.nextaddr



LL - Insertion at end

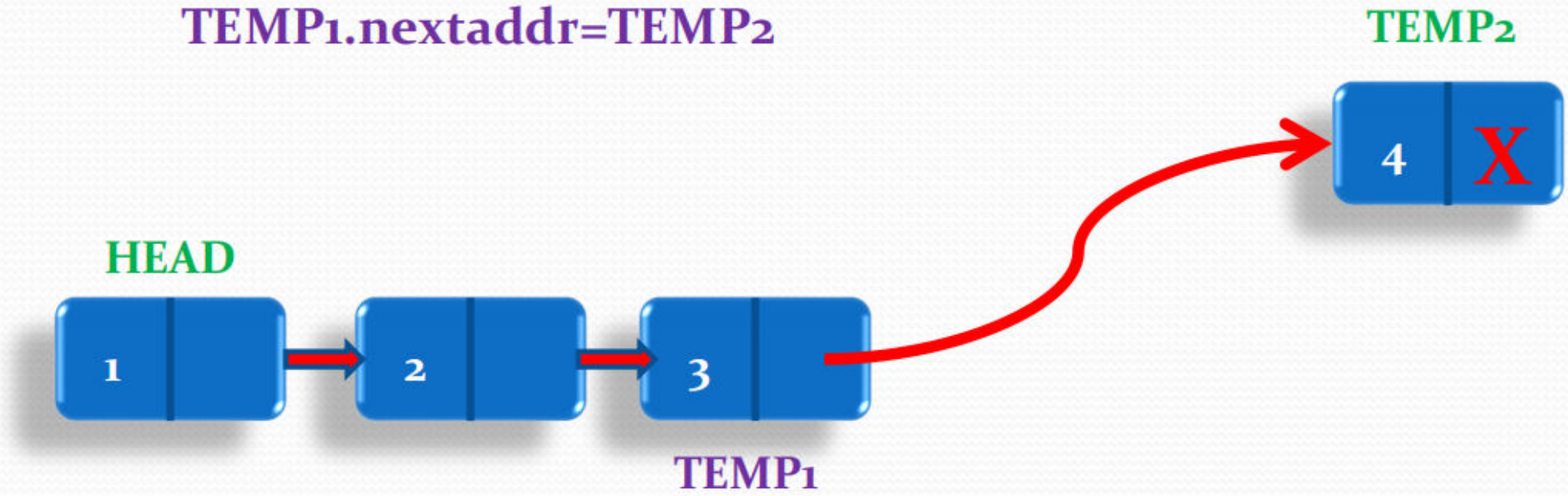
Now TEMP1.nextaddr =NULL



LL - Insertion at end

So add the new node, TEMP2 after TEMP1

TEMP1.nextaddr=TEMP2



LL - Insert At End Or Create List

Algorithm -INSERT AT END or CREATE LIST if list is null (header, item)

Input: head is the pointer to the header node and item is the element to be inserted.

Output: A node with element item is inserted at end of the list.

Data Structure: Linked list

Single linked list insertion at End

1. Start
2. Read item;
3. newnode=malloc(node);
4. newnode.data=item;
5. newnode.next=NULL
6. if(head==NULL) then // list empty
   a. HEAD=newnode
   b. Exit
7. Else
   a. temp=head
   b. while(temp.next!=NULL)
      i. temp=temp.next
   c. end while
   d. temp.next=newnode;
8. Endif
9. Stop

LL - Insertion at beginning of the list

inserts a node at the front of the list, if list is not null.

if the list is empty the new node is assigned as the HEAD

LL - Insertion at beginning of the list

Algorithm -INSERT AT FRONT (head, item)

Input: HEAD is the pointer to the header node and item is the element to be inserted.

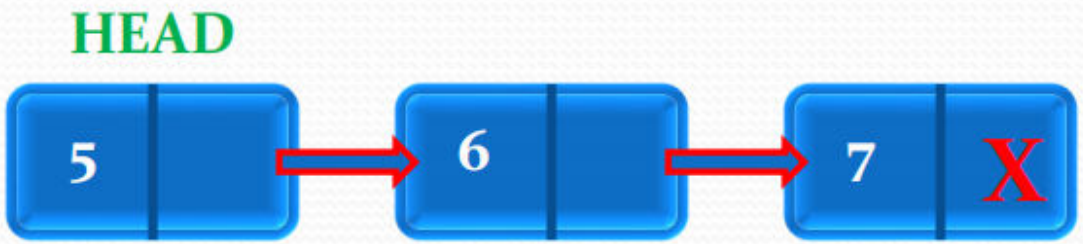
Output: A node with element item is inserted at front of the list if list is not null.

Data Structure: Linked list

1. Start
2. Read item;
3. newnode=malloc(node);
4. newnode.data=item;
5. newnode.next=NULL
6. if(head==NULL) then // list empty
   a. HEAD=newnode
   b. Exit
7. Else
   a. newnode.next=head;
   b. head= newnode
8. Endif
9. Stop

LL - Insertion at beginning of the list

1. Get space from memory
2. Store the data in the node
3. Store NULL in the address field

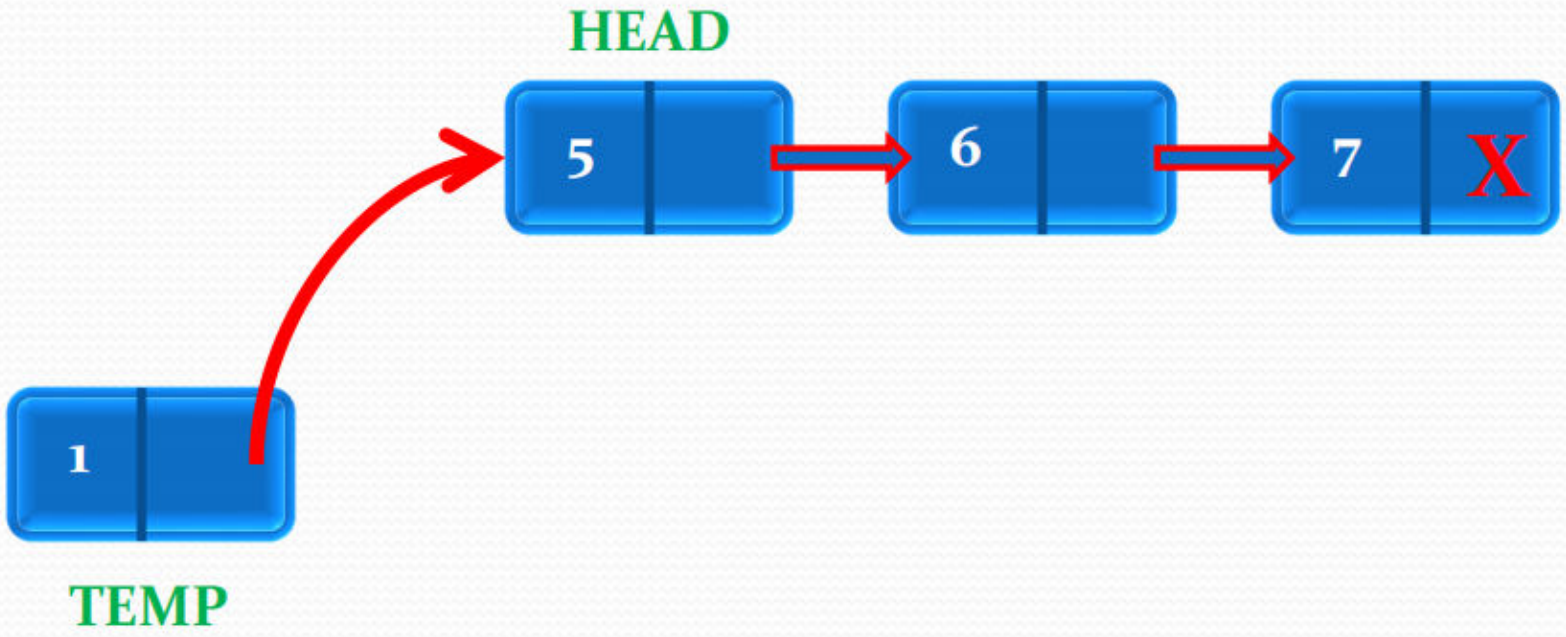


LL - Insertion at beginning of the list

Inorder to insert node 1 at the beginning, we just need to

1. Make the next node of the new node as the head

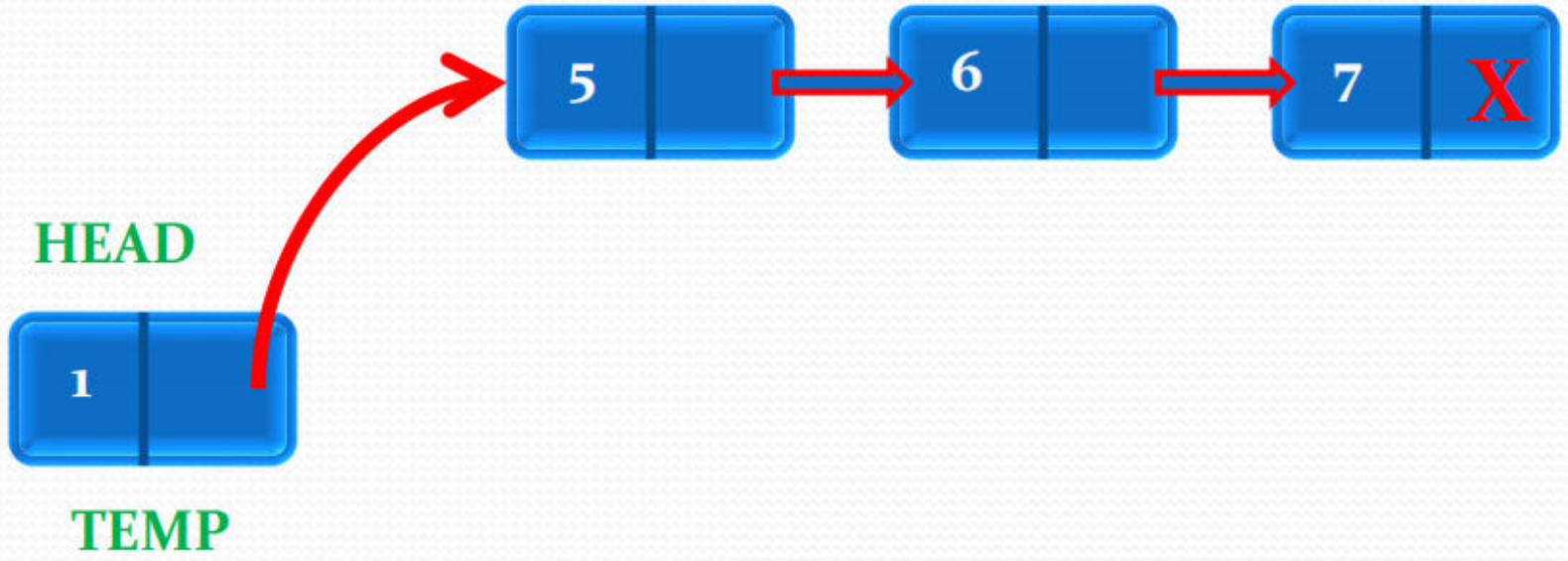
   i.e TEMP.nextaddr = HEAD



LL - Insertion at beginning of the list

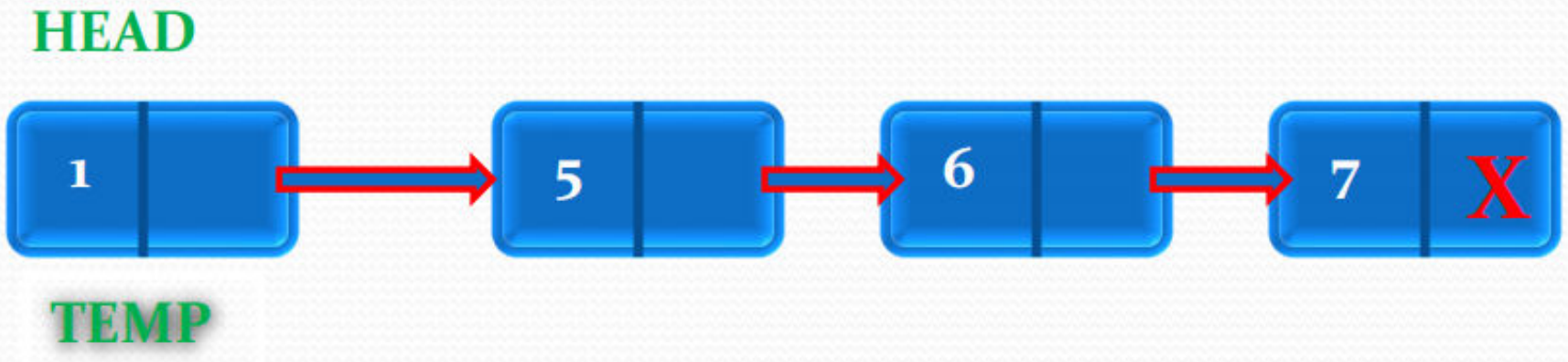
Inorder to insert node 1 at the beginning, we just need to

1. Make the next node of the new node as the head
   i.e TEMP.nextaddr = HEAD
2. Make the new node as head
   1. **i.e HEAD = TEMP**



LL - Insertion at beginning of the list

Now the linked list is



LL - Insertion at beginning of the list

What if the **linked list was empty ???**

HEAD = NULL

TEMP

LL - Insertion at beginning of the list

Create a new list by making the new node as HEAD

LL - Insertion at beginning of the list

Algorithm -INSERT AT FRONT (head, item)

Input: HEAD is the pointer to the header node and item is the element to be inserted.

Output: A node with element item is inserted at front of the list if list is not null.

Data Structure: Linked list

1. Start
2. Read item;
3. newnode=malloc(node);
4. newnode.data=item;
5. newnode.next=NULL
6. if(head==NULL) then // list empty
   a. HEAD=newnode
   b. Exit
7. Else
   a. newnode.next=head;
   b. head= newnode
8. Endif
9. Stop

LL - Insertion at particular position

- **inserts element at any position of the list**
- Say, insert element 3 after element 2

LL - Insertion at particular positior

Algorithm: INSERT AT ANY (header, item, key)

Input: head is the pointer to the header node and item is the element to be inserted and key being the data one node after which the item to be inserted.

Output: A node with element item is inserted at given position if the list is not null.

Data Structure: Linked list

1.Start
2.Read item;
3.newnode=malloc(node);
4.newnode.data=item;
5.newnode.next=NULL
6.if(head==NULL) then // list empty
    a.PRINT LIST EMPTY
    b.Exit

1. Else
   a. temp=HEAD
   b. while(temp.next!=NULL && temp.data!=key)
      i. temp=temp.next
   c. end while
   d. if(temp==NULL)
      i. PRINT(“KEY NOT FOUND”)
      ii. EXIT
   e. Endif
   f. newnode.next=temp.next;
   g. temp.next=newnode;
2. Stop

```c
#include<stdio.h>
#include<stdlib.h>
void beginsert(int);
struct node
{
    int data;
    struct node *next;
};
struct node *head;
```



```c
void main ()
{
    int choice,item;
    do
    {
        printf("\nEnter the item which you want to insert?\n");
        scanf("%d",&item);
        beginsert(item);
        printf("\nPress 0 to insert more ?\n");
        scanf("%d",&choice);
    }while(choice == 0);
}
```

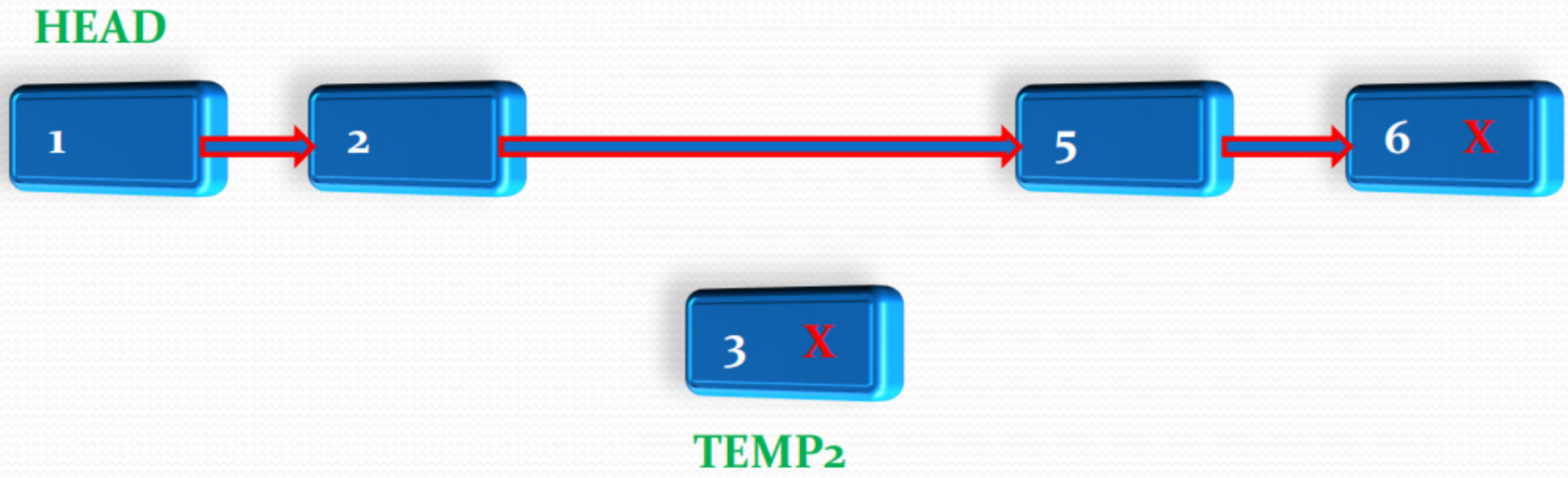
```
void beginsert(int item)
  {
     struct node *ptr = (struct node *)malloc(sizeof(struct node *));
     if(ptr == NULL)
     {
        printf("\nOVERFLOW\n");
     }
     else
     {
        ptr->data = item;
        ptr->next = head;
        head = ptr;
        printf("\nNode inserted\n");
     }

  }
```

LL - Insertion at particular position

- **inserts element at any position of the list**
- Say, insert element 3 after element 2

LL - Insertion at particular position

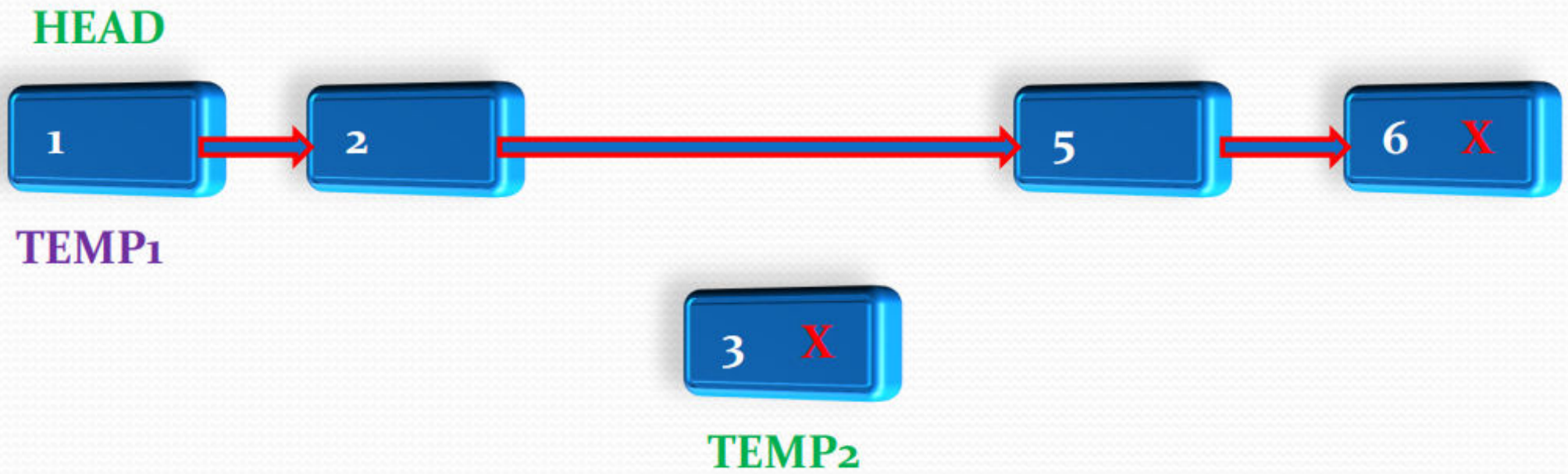
1. Get space from memory
2. Store the data in the node
3. Store NULL in the address field



LL - Insertion at particular position

1. Traverse the list until the desired key is found, say 2

   Set TEMP1 =HEAD

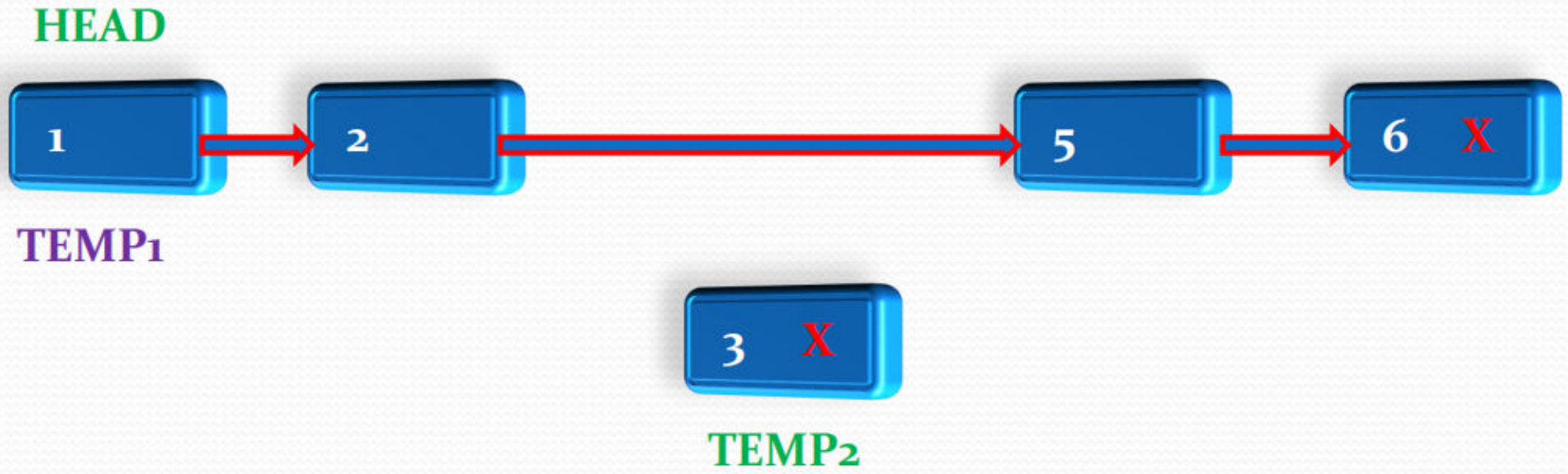


LL - Insertion at particular position

1. Traverse the list until the desired key is found, say 2

While(temp1!=NULL and temp1.data!=key)

Temp1=temp1.nextaddr

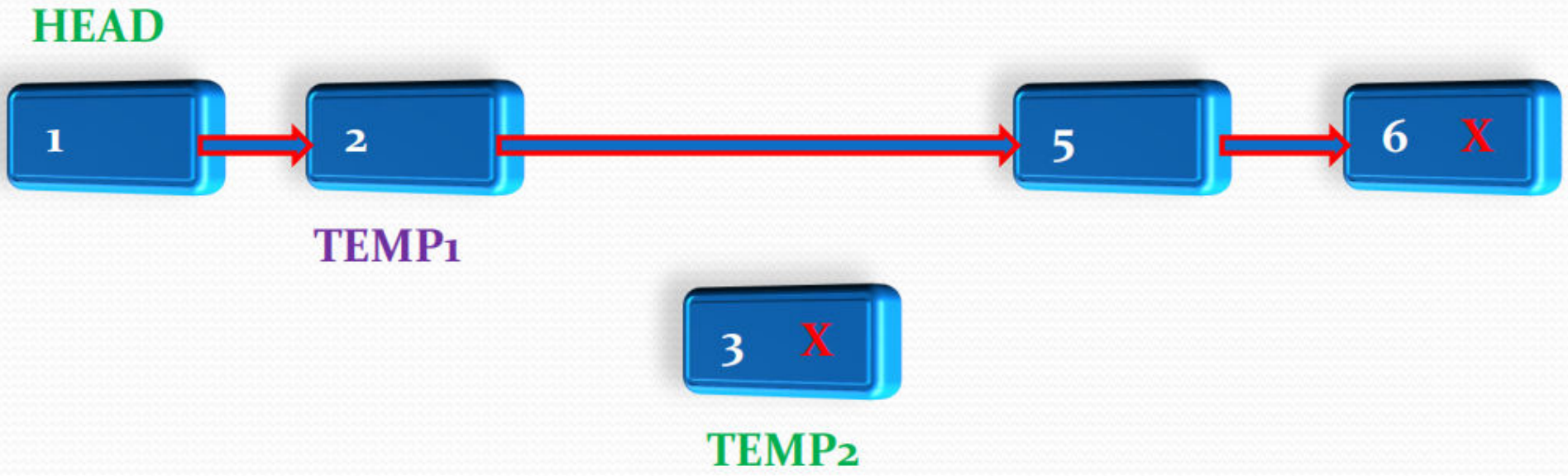


LL - Insertion at particular position

1. Traverse the list until the desired key is found, say 2

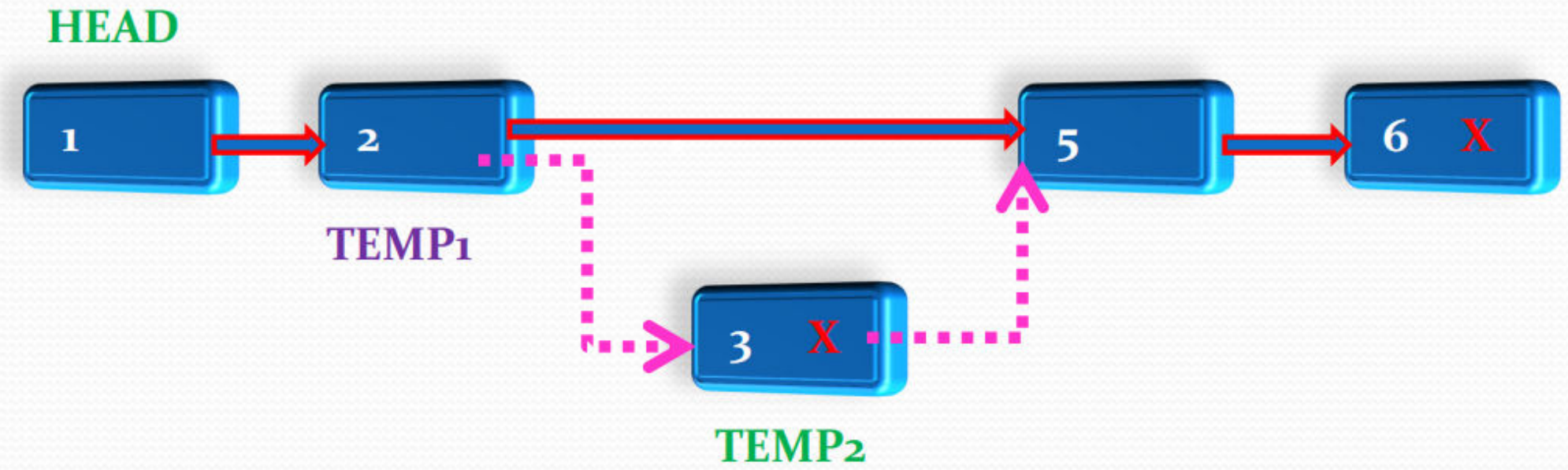
While(temp1!=NULL and temp1.data!=key)

Temp1=temp1.nextaddr



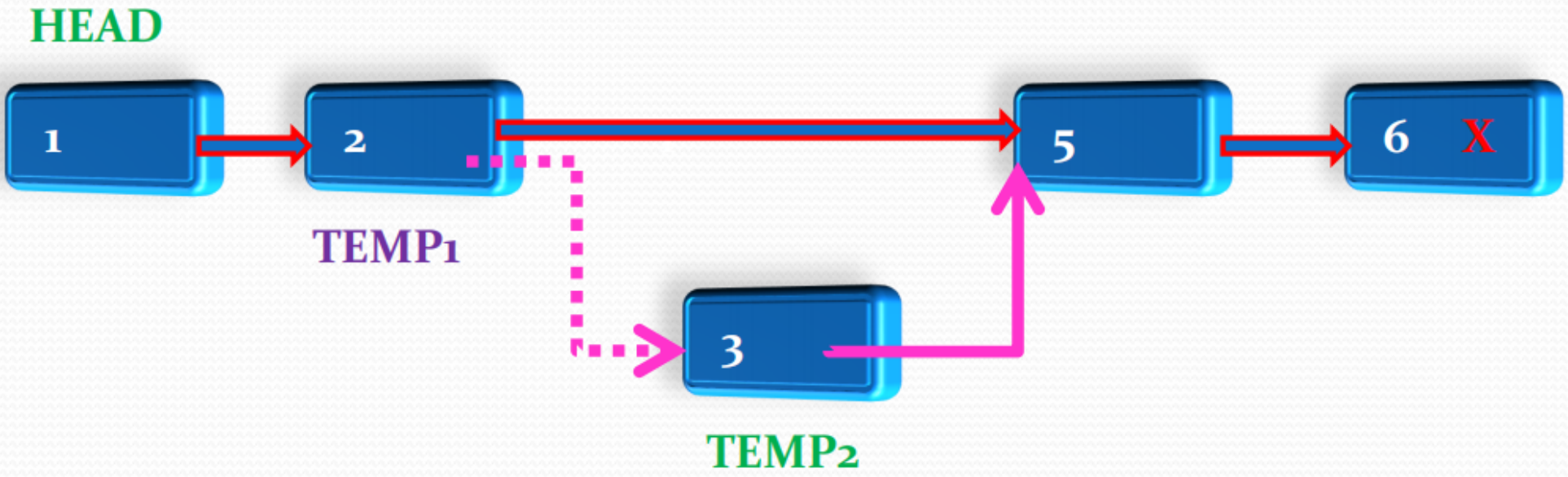
LL - Insertion at particular position

1. Traverse the list until the desired key is found, say 2



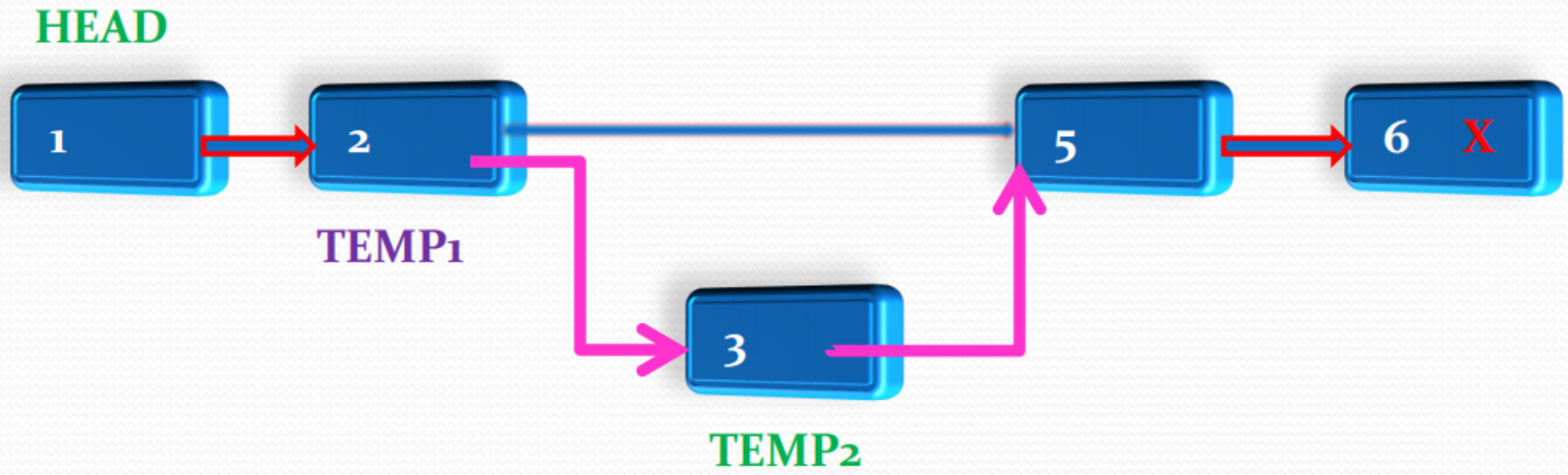
LL - Insertion at particular position

1. Traverse the list until the desired key is found, say 2
2. Set **TEMP2.nextaddr = TEMP1.nextaddr**



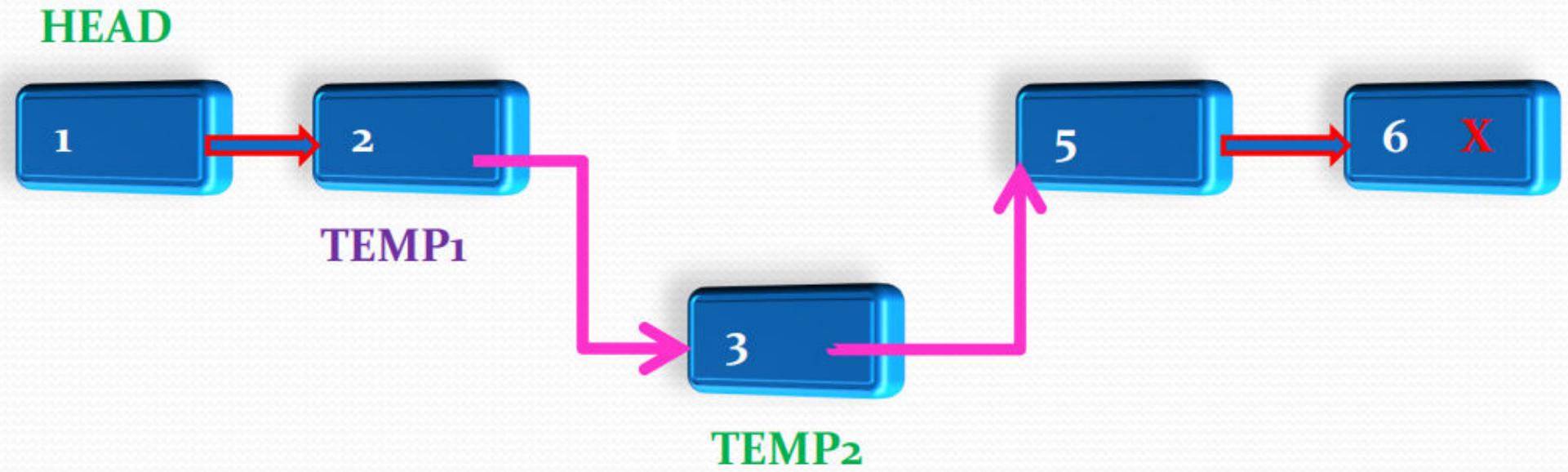
LL - Insertion at particular position

1. Traverse the list until the desired key is found, say 2
2. Set **TEMP1.nextaddr = TEMP2**



LL - Insertion at particular position

1. Traverse the list until the desired key is found, say 2
2. Set TEMP1.nextaddr = TEMP2



Traversing a linked list

Homework

Traversing a linked list

Algorithm TRAVERSE (header)

Input: Header is the pointer to the first node if not null.

Data Structure: A single linked list.

Steps:

1. If (head=NULL) then
    1. Print("List is empty")
    2. Exit
2. Else
    1. Temp=head
    2. While(temp!=NULL)
        1. Print(temp.data)
        2. Temp=temp.nextaddr
    3. EndWhile
3. stop

Linked List Deletion

Various cases of deletion

1. Deletion from the front of the list
2. Deletion from the end of the list
3. Deletion from any position of the list

Requires a special function FREE() to return the space of the deleted node to the free memory pool.

FREE () is just opposite of MALLOC().

LL Deletion from the front of the list

Algorithm DELETE FRONT(header)

Input: Header is the pointer to the header node of the linked list

Output: A single linked list eliminating the node at the front.

Data Structure: Linked list

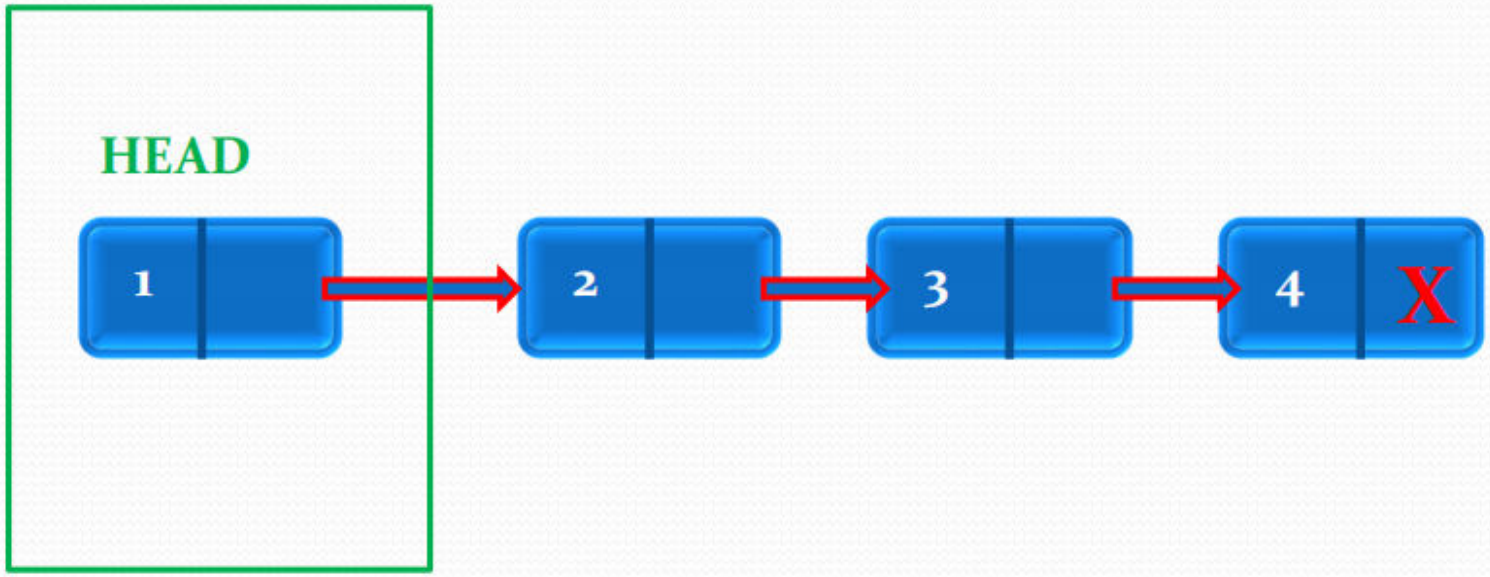
Steps:

1. If(head=NULL)
    1. Printf("List empty")
    2. Exit
2. EndIf
3. Temp=head
4. Head=head.nextaddr
5. FREE (temp)
6. Stop

Deletion from the FRONT of the list

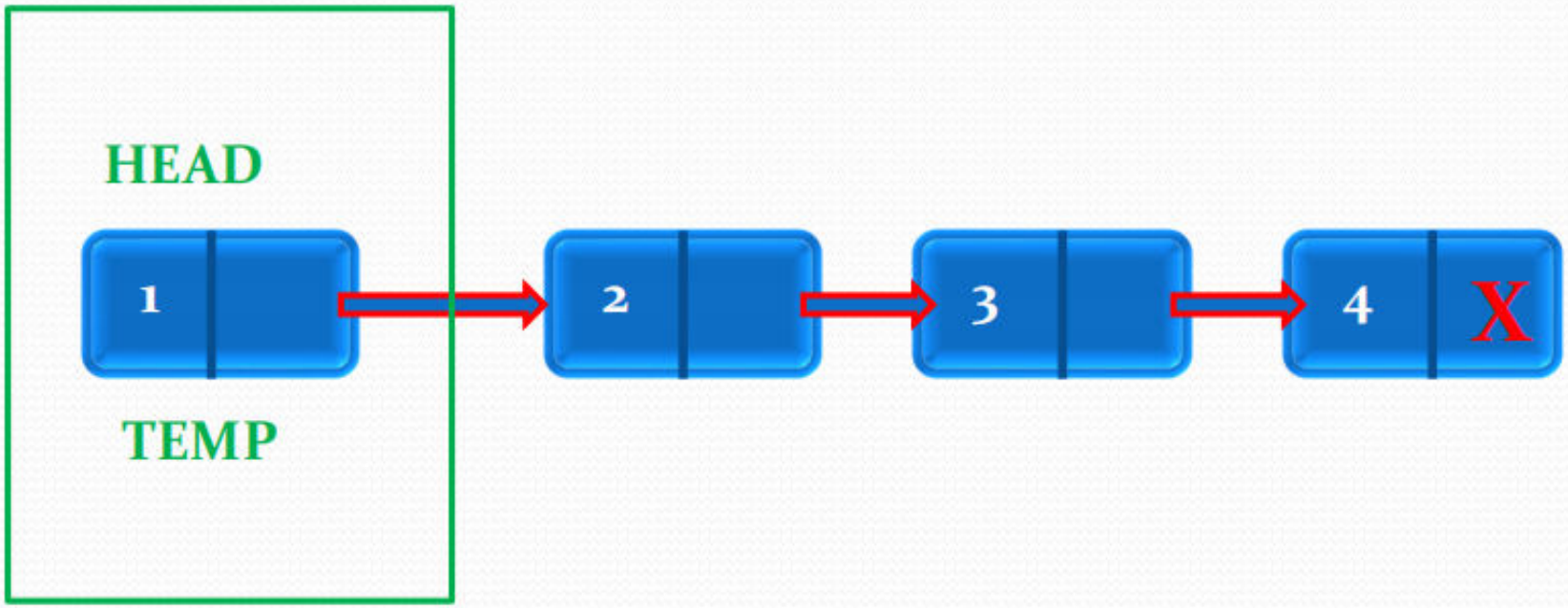
(1) LL- DELETION from the front of the list

- Delete the front node



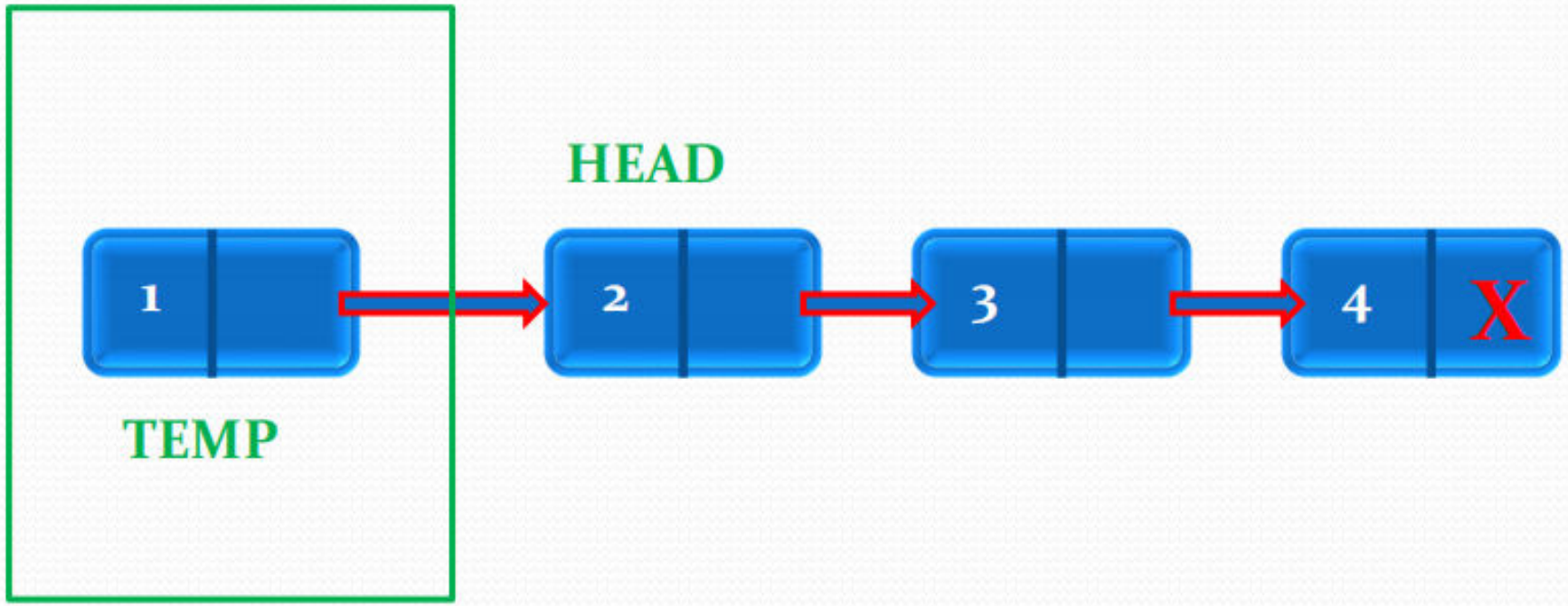
- Delete the front node

Set TEMP = HEAD



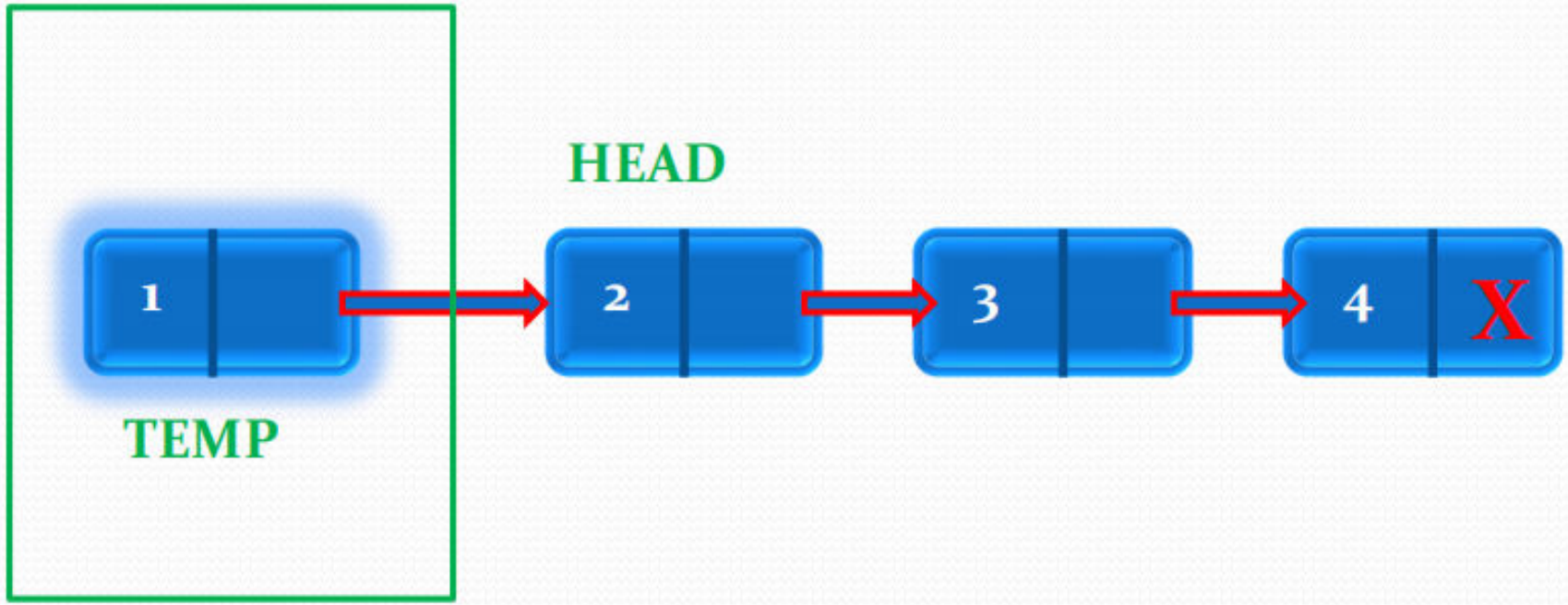
LL- DELETION from the front of the list

- No need to access the first node anymore . So **move the HEAD pointer** to the next node



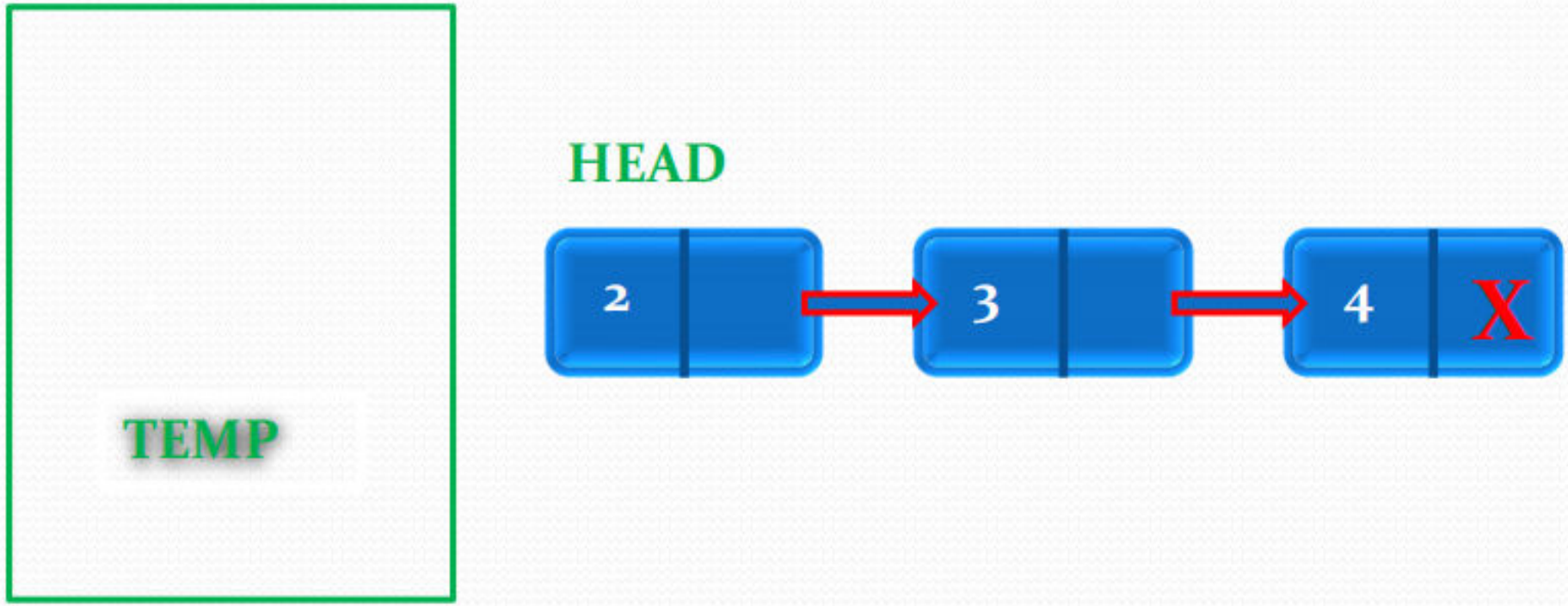
LL- DELETION from the front of the list

- Now free the memory area using FREE function
- i.e **FREE (TEMP)**



LL- DELETION from the front of the list

- Now free the memory area using FREE function
- i.e **FREE (TEMP)**



LL- DELETION from the end of the list

Deletion from the end of the list

The algorithm DELETE LAST is to delete a node at the end from the single linked list.

Algorithm: DELETE LAST (head)

Input: Head is the pointer to the header node of the linked list

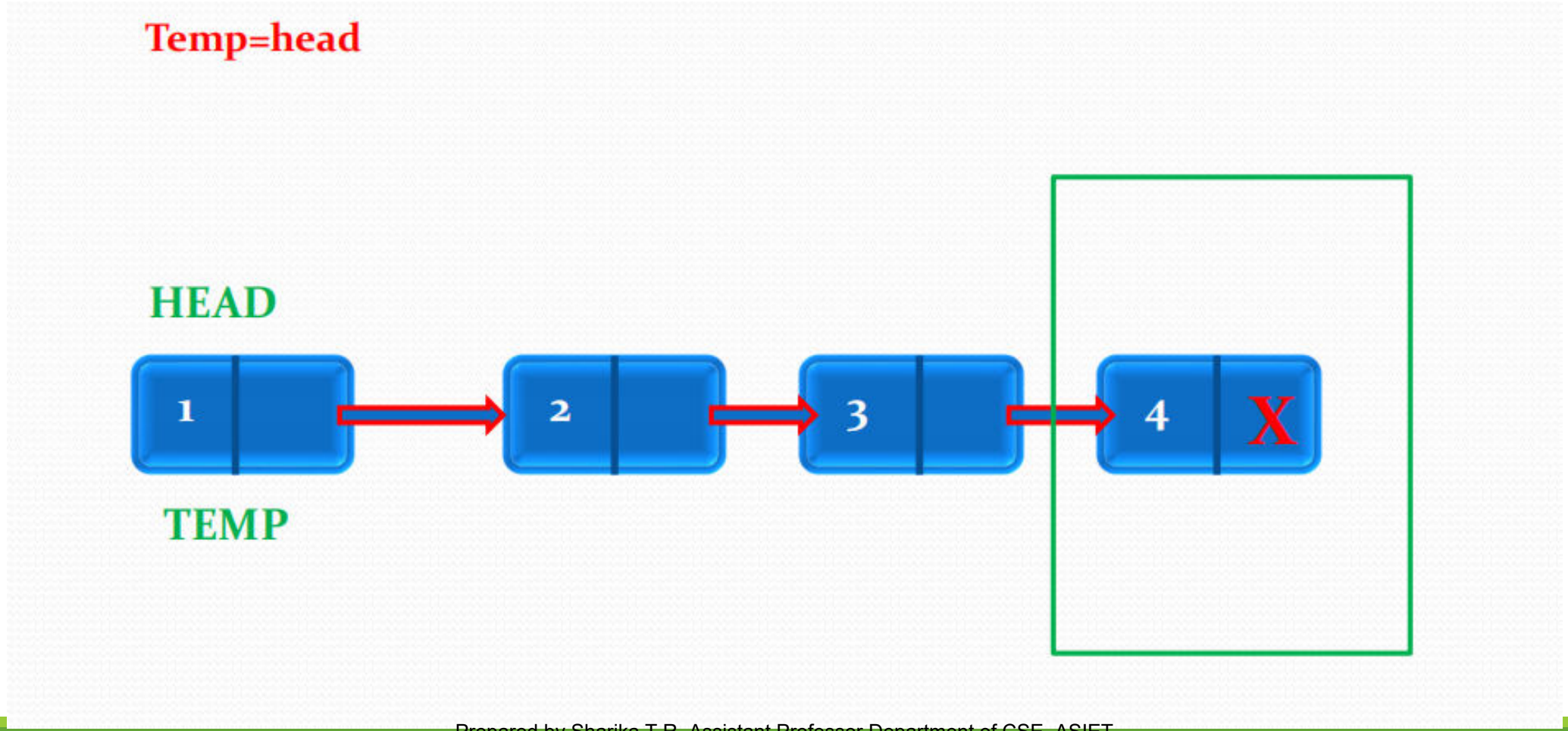
Output: A single linked list eliminating the node at the end.

Data Structure: Linked list

Steps:

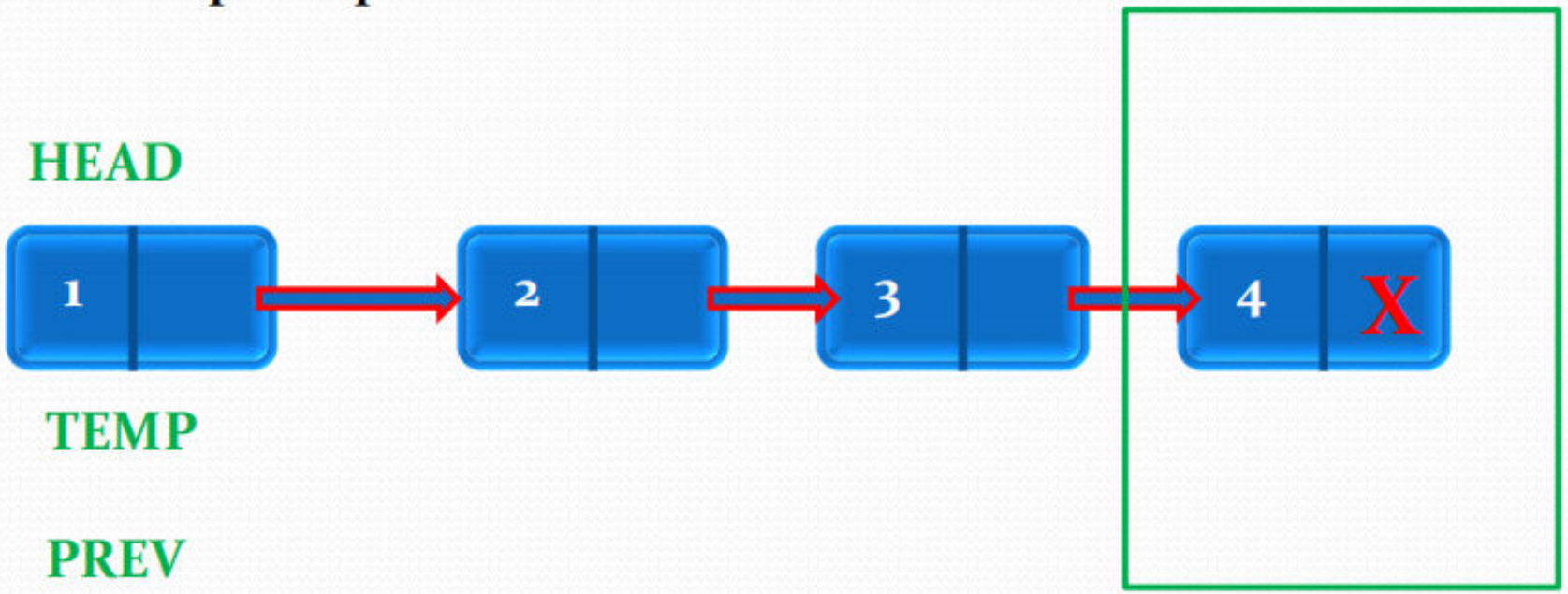
1. If(head=NULL)
    1. Printf("List is empty..")
    2. Exit
2. EndIf
3. Temp=head
4. If(**head.nextaddr=NULL**) *//only 1 element*
    1. **Head=NULL**
5. Else
    1. **While(temp.nextaddr!=NULL)**
        1. **Prev=temp;**
        2. **Temp=temp.nextaddr**
    2. EndWhile
    3. Prev.nextaddr=NULL
4. FREE (temp)
6. EndIf
7. Stop

Deletion from the END of the list



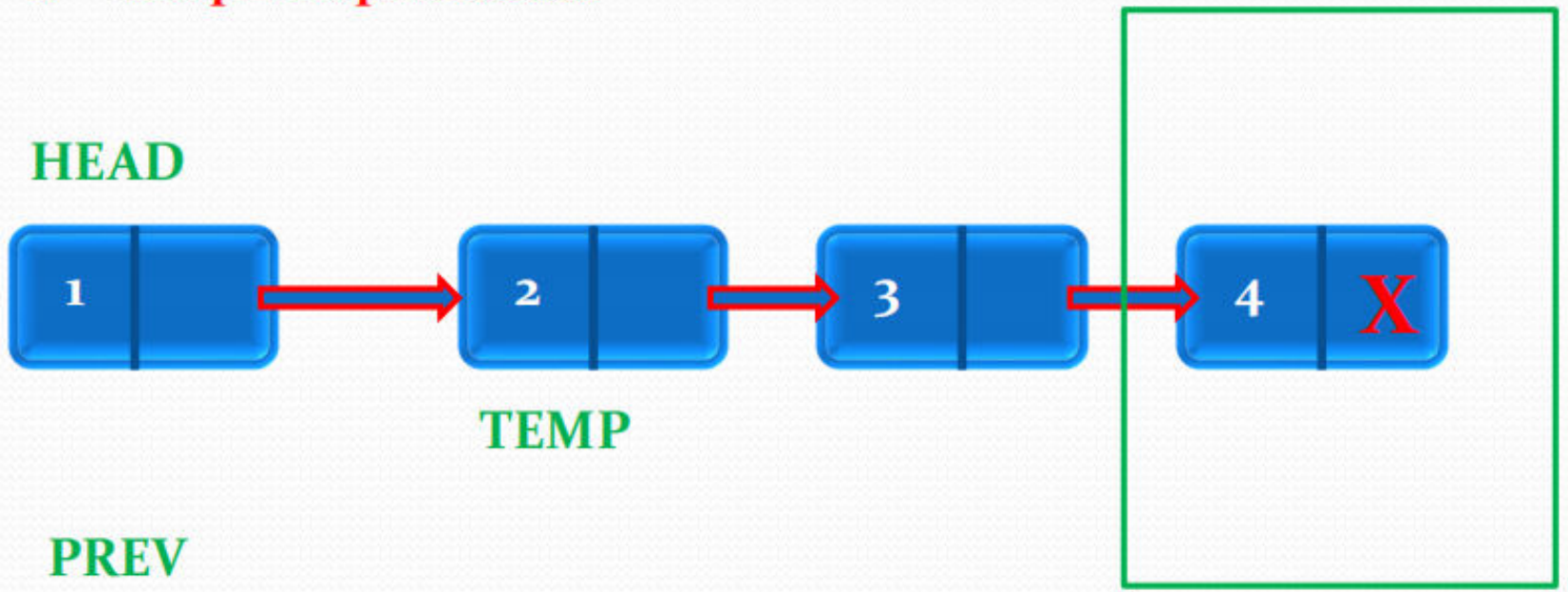
LL- DELETION from the end of the list

1. While(temp.nextaddr!=NULL)
    1. Prev=temp;
    2. Temp=temp.nextaddr



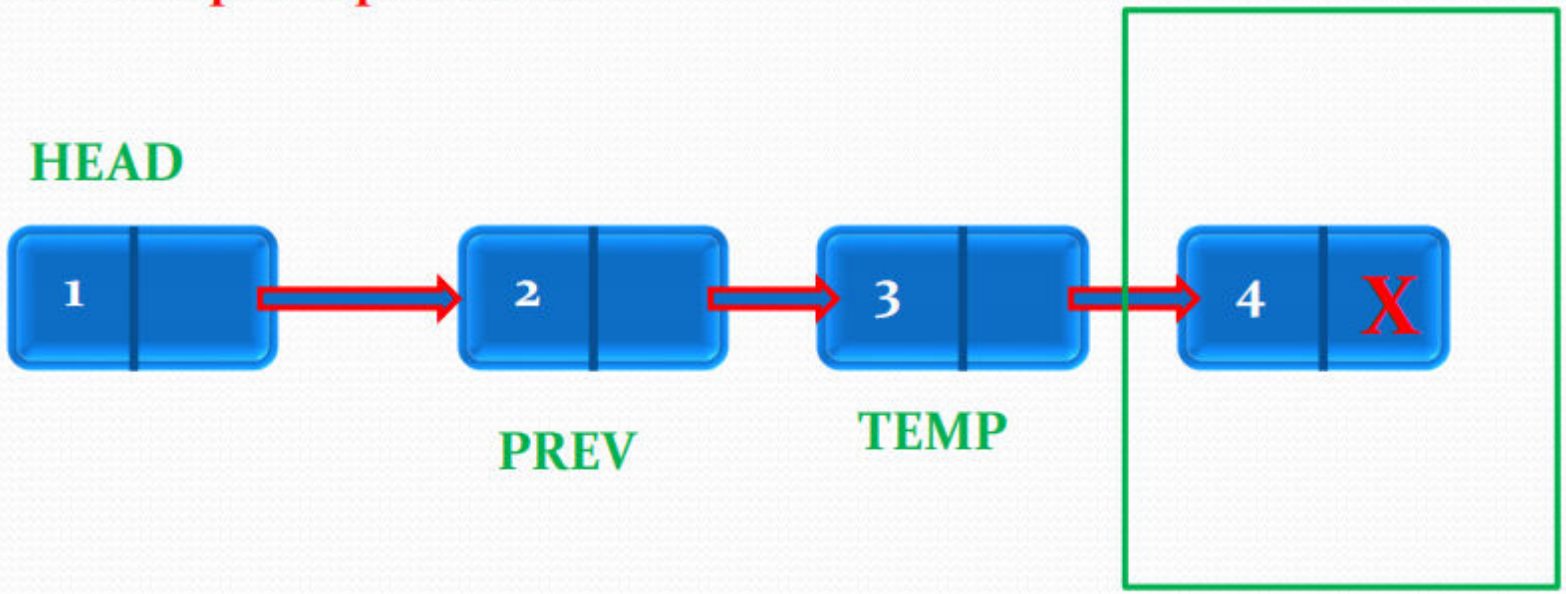
LL- DELETION from the end of the list

1. While(temp.nextaddr!=NULL)
    1. Prev=temp;
    2. Temp=temp.nextaddr



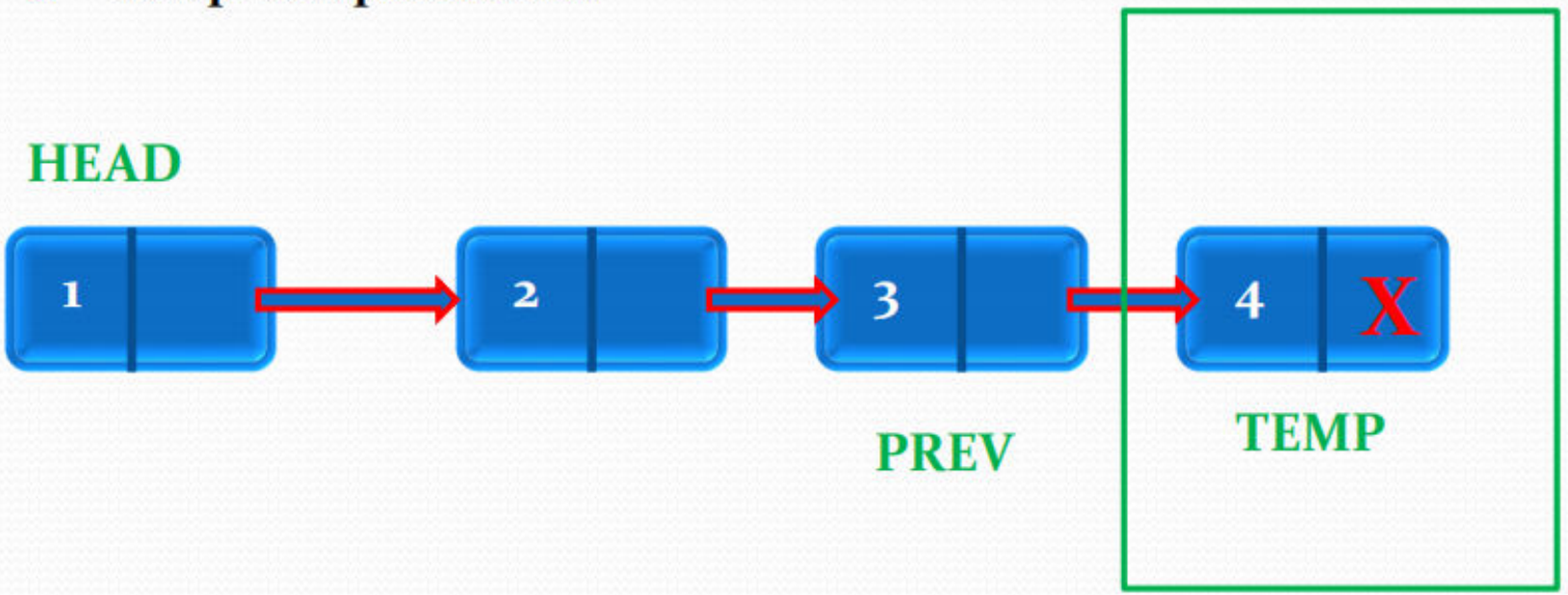
LL- DELETION from the end of the list

1. While(temp.nextaddr!=NULL)
    1. Prev=temp;
    2. Temp=temp.nextaddr



LL- DELETION from the end of the list

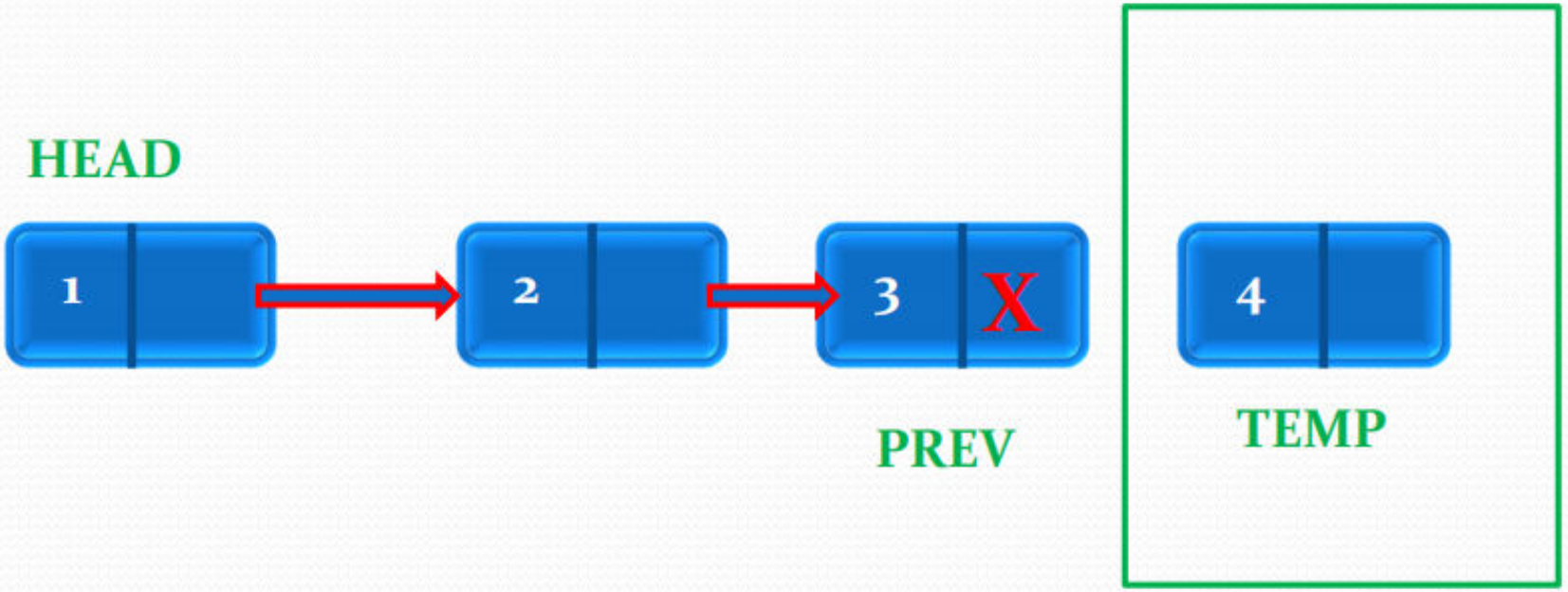
1. While(temp.nextaddr!=NULL)
    1. Prev=temp;
    2. Temp=temp.nextaddr



LL- DELETION from the end of the list

Prev.nextaddr=NULL

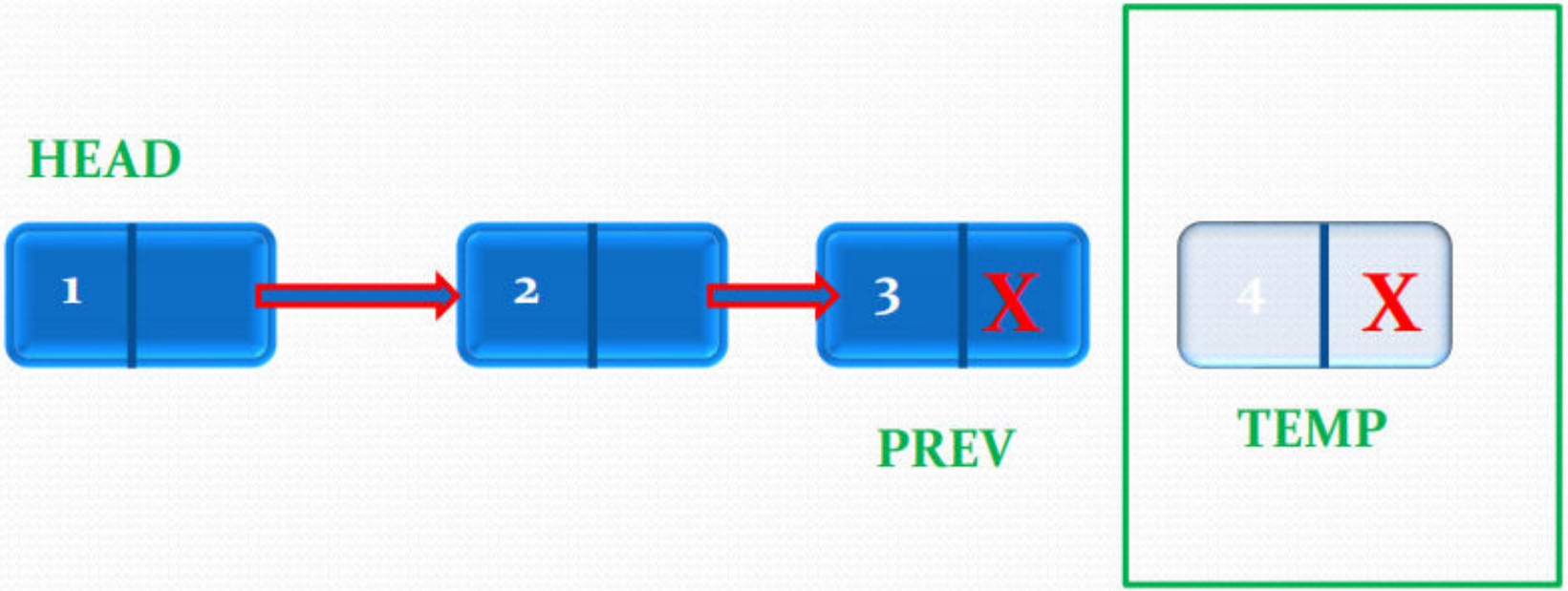
FREE (temp)



LL- DELETION from the end of the list

Prev.nextaddr=NULL

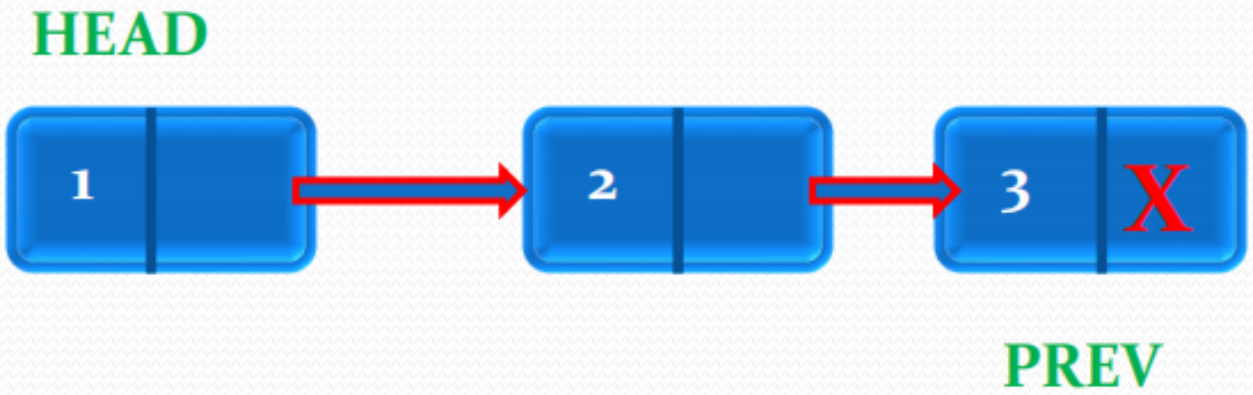
FREE (temp)



LL- DELETION from the end of the list

Prev.nextaddr=NULL

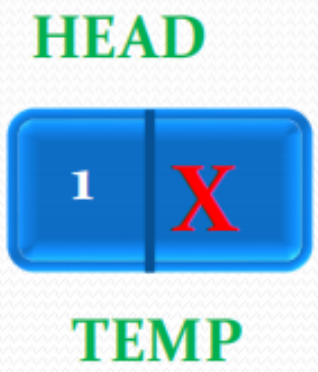
FREE (temp)



LL- DELETION from the end of the list

Special Case

1. Temp=head
2. If(temp.nextaddr=NULL) //only 1 element
    1. Head=NULL
    2. FREE (temp)



LL- DELETION from the end of the list

Special Case

1. If(head.nextaddr=NULL) //only 1 element
    1. Head=NULL
    2. FREE (temp)

HEAD = NULL

LL- DELETION from the end of the list

Special Case

1. If(**head.nextaddr=NULL**) *//only 1 element*
    1. **Head=NULL**
    2. FREE (temp)

HEAD = NULL

Algorithm: DELETE MIDDLE (header, key)

Input: Header is the pointer to the header node of the linked list, key is the data content of the node to be **deleted.**

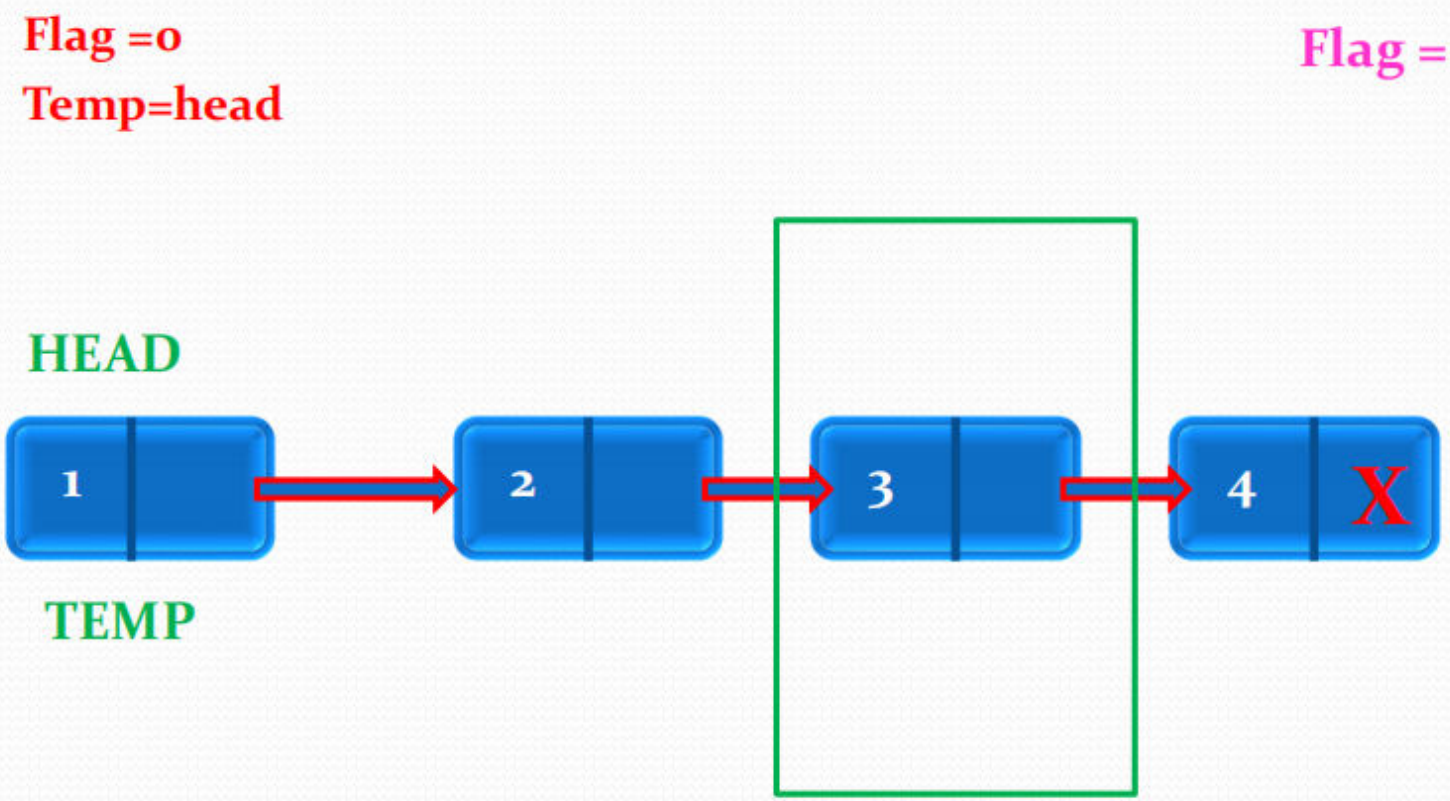
Output: A single linked list except the node with content as key.

Data Structure: Linked list.

Steps:

1. Flag=0
2. If(head=NULL) then
    1. Printf("List is Empty")
    2. Exit
3. Else If(header.data=key)
    1. Item=head.data
    2. Temp=head
    3. Head=head.link
    4. FREE (temp)
    5. Exit
4. Else
    1. Temp=head
    2. While(temp.link!=NULL)
        1. Prev=temp
        2. Temp=temp.link
        3. If(temp.data=key)
            1. Flag=1
            2. ExitWhile
        4. EndIf
    3. EndWhile
    4. If(flag=0)
        1. Print("Key not found)
        2. Exit
    5. Else
        1. Item=temp.data
        2. Prev.link=temp.link
        3. FREE (temp)
    6. EndIf
5. EndIf
6. Stop

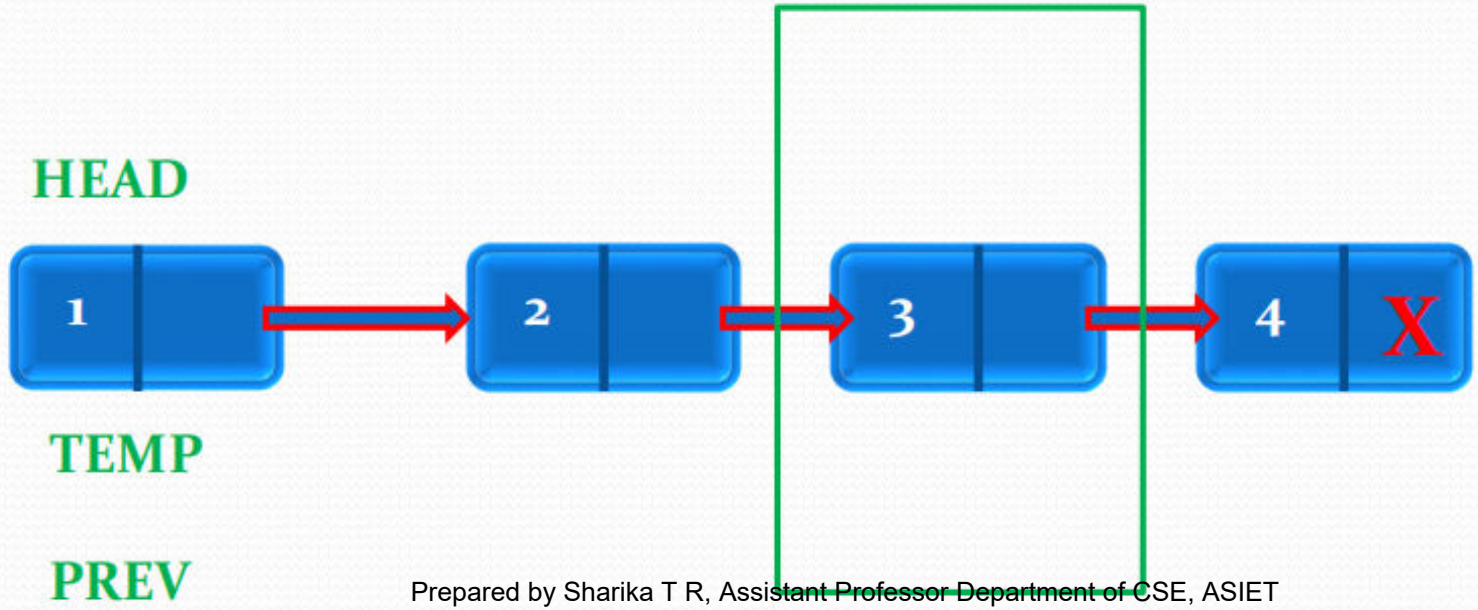
Deletion from MIDDLE of the list



LL- DELETION from middle of the list

```
While(temp.link!=NULL)
        Prev=temp
        Temp=temp.link
        If(temp.data=key)
                Flag=1
                ExitWhile
        EndIf
EndWhile
```

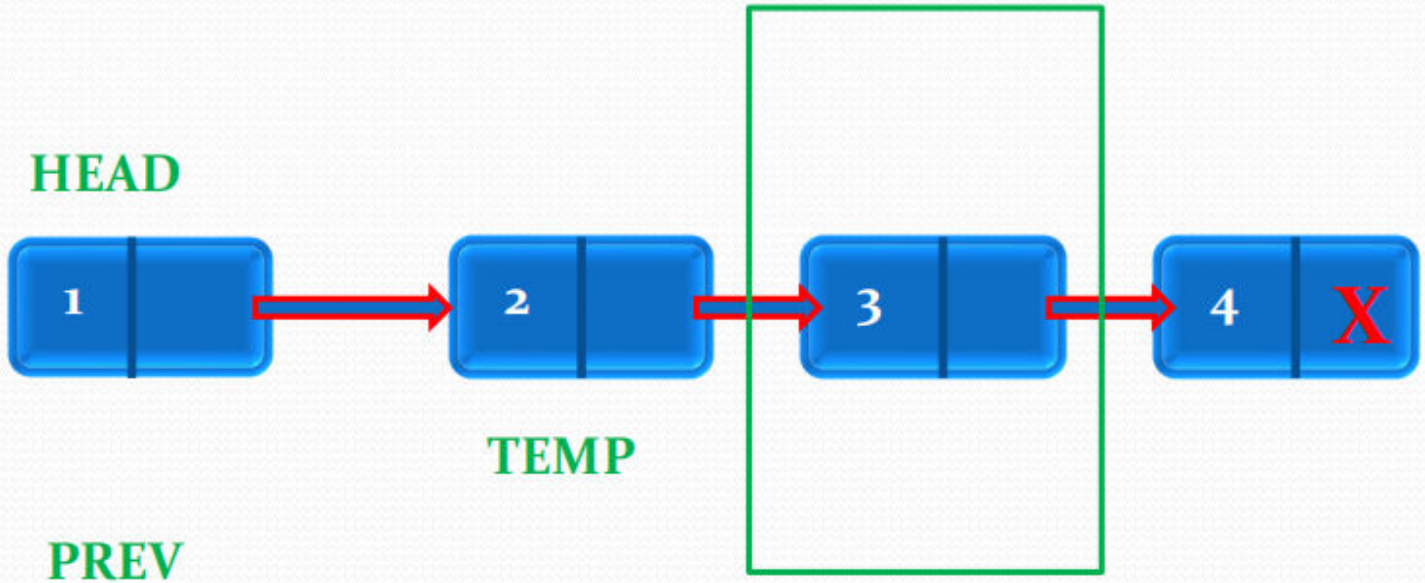
Flag = 0



LL- DELETION from middle of the list

```
While(temp.link!=NULL)
        Prev=temp
        Temp=temp.link
        If(temp.data=key)
                Flag=1
                ExitWhile
        EndIf
EndWhile
```

Flag = 0

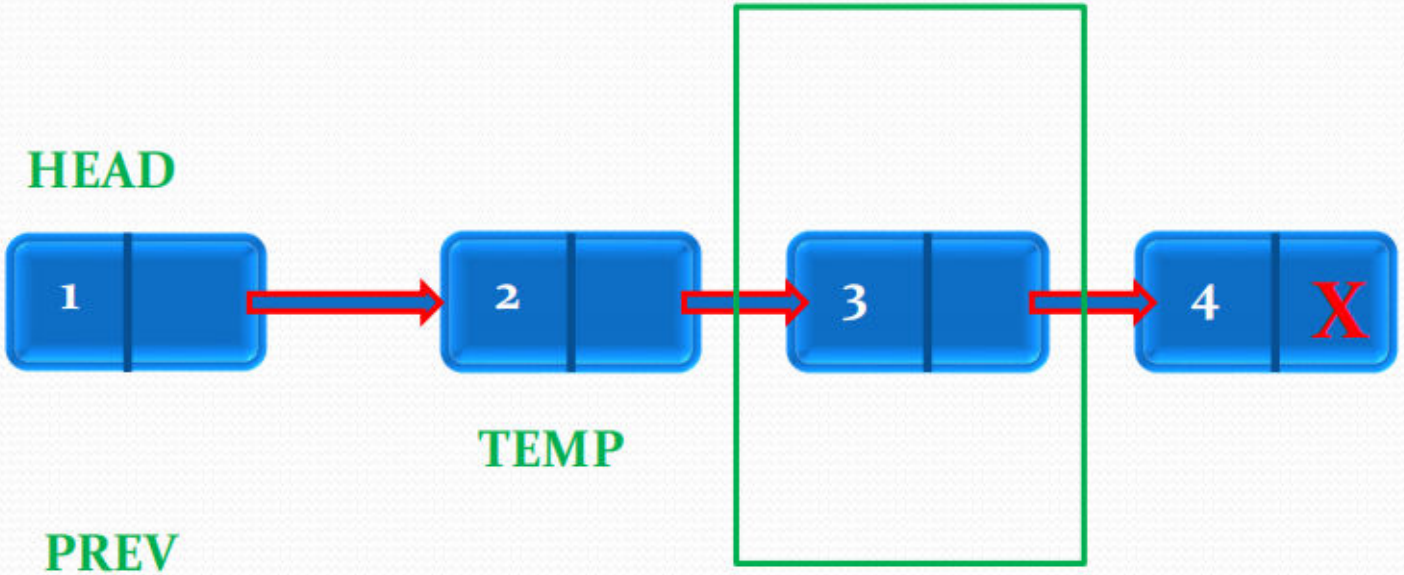


LL- DELETION from middle of the list


```
While(temp.link!=NULL)
        Prev=temp
        Temp=temp.link
        If(temp.data=key)
                Flag=1
                ExitWhile
        EndIf
EndWhile
```

NO

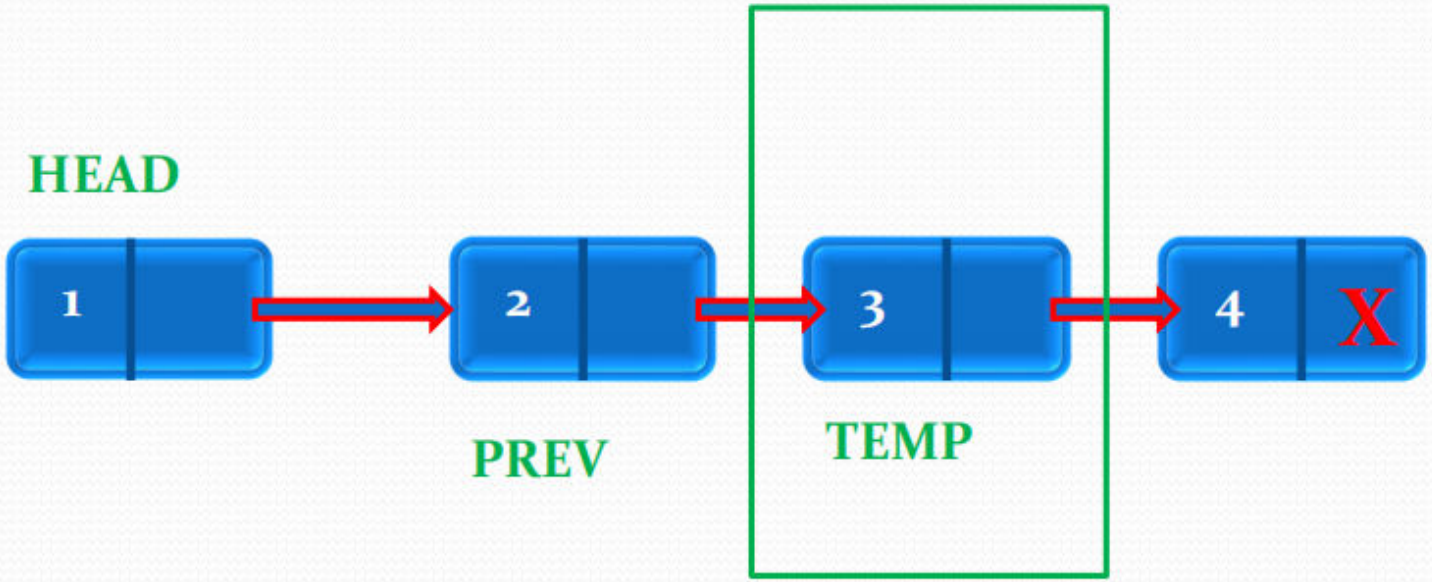
Flag = 0



LL- DELETION from middle of the list


```
While(temp.link!=NULL)
        Prev=temp
        Temp=temp.link
        If(temp.data=key)
                Flag=1
                ExitWhile
        EndIf
EndWhile
```

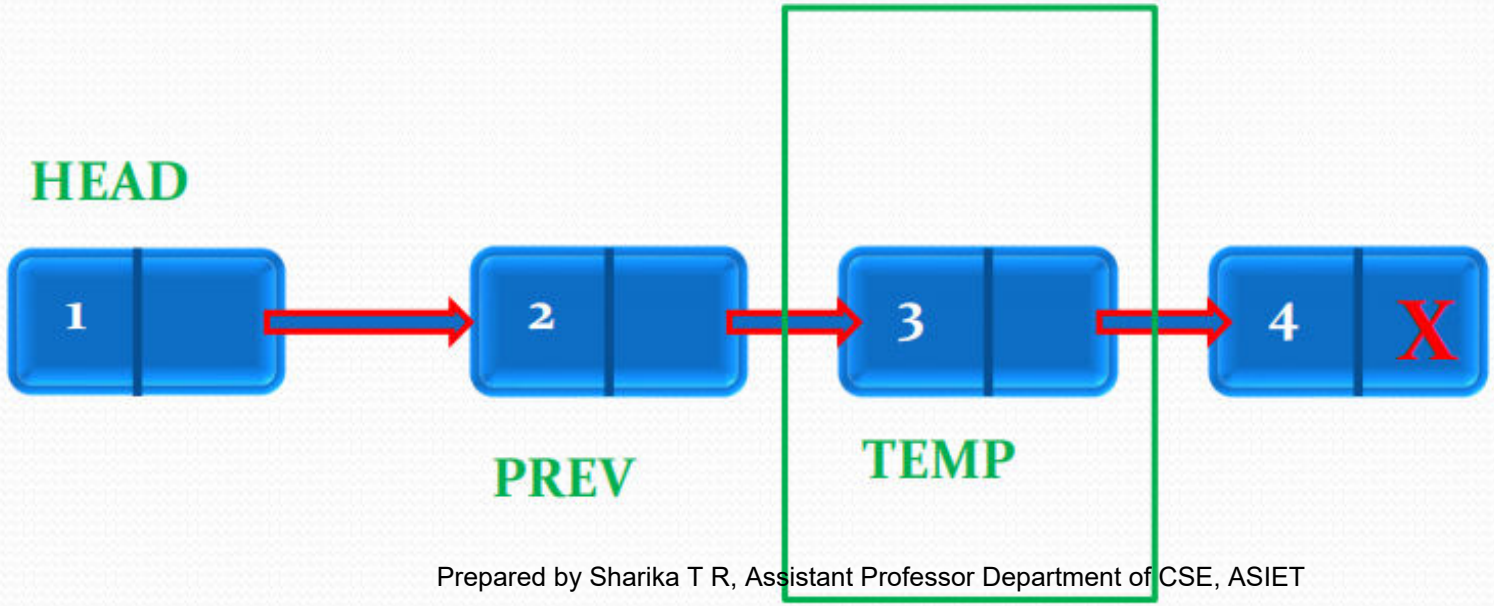
Flag = 0



LL- DELETION from middle of the list

```
While(temp.link!=NULL)
        Prev=temp
        Temp=temp.link
        If(temp.data=key)
                Flag=1
                ExitWhile
        EndIf
EndWhile
```

Flag = 1

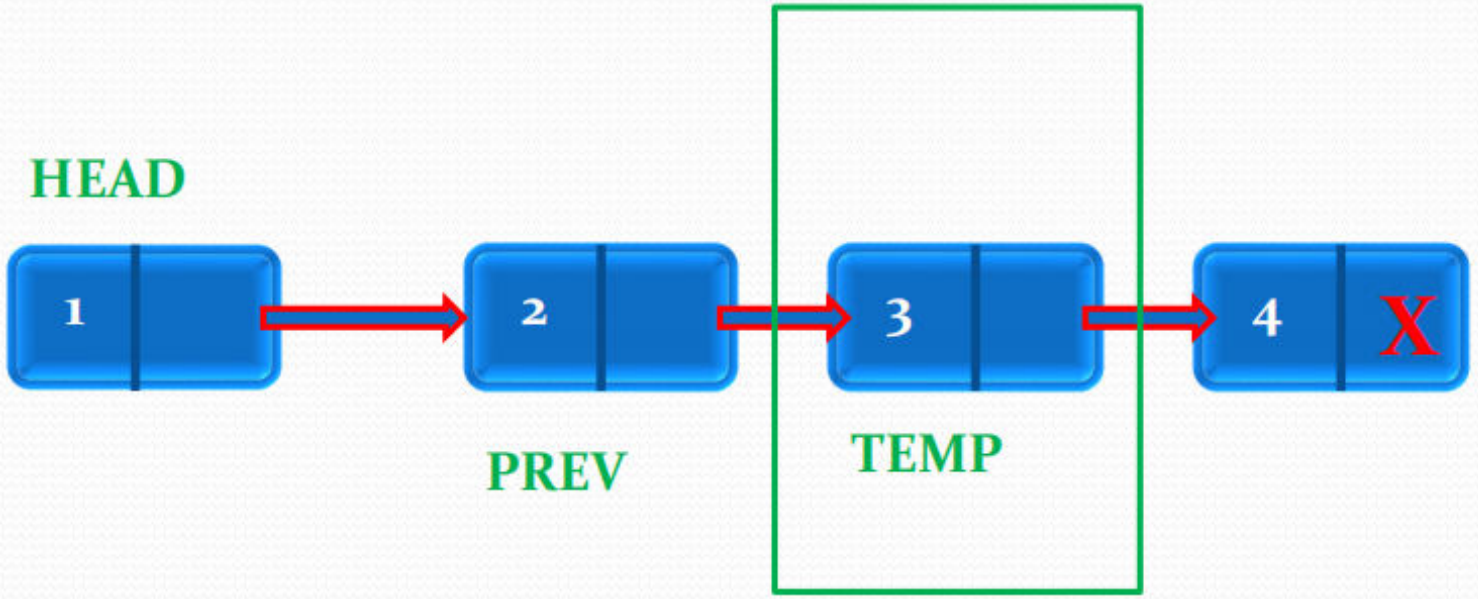


LL- DELETION from middle of the list

1. If(flag=o)
    1. Print("Key not found)
    2. Exit
2. Else
    1. Item=temp.data
    2. Prev.link=temp.link
    3. FREE (temp)

Flag = 1

Item = 3

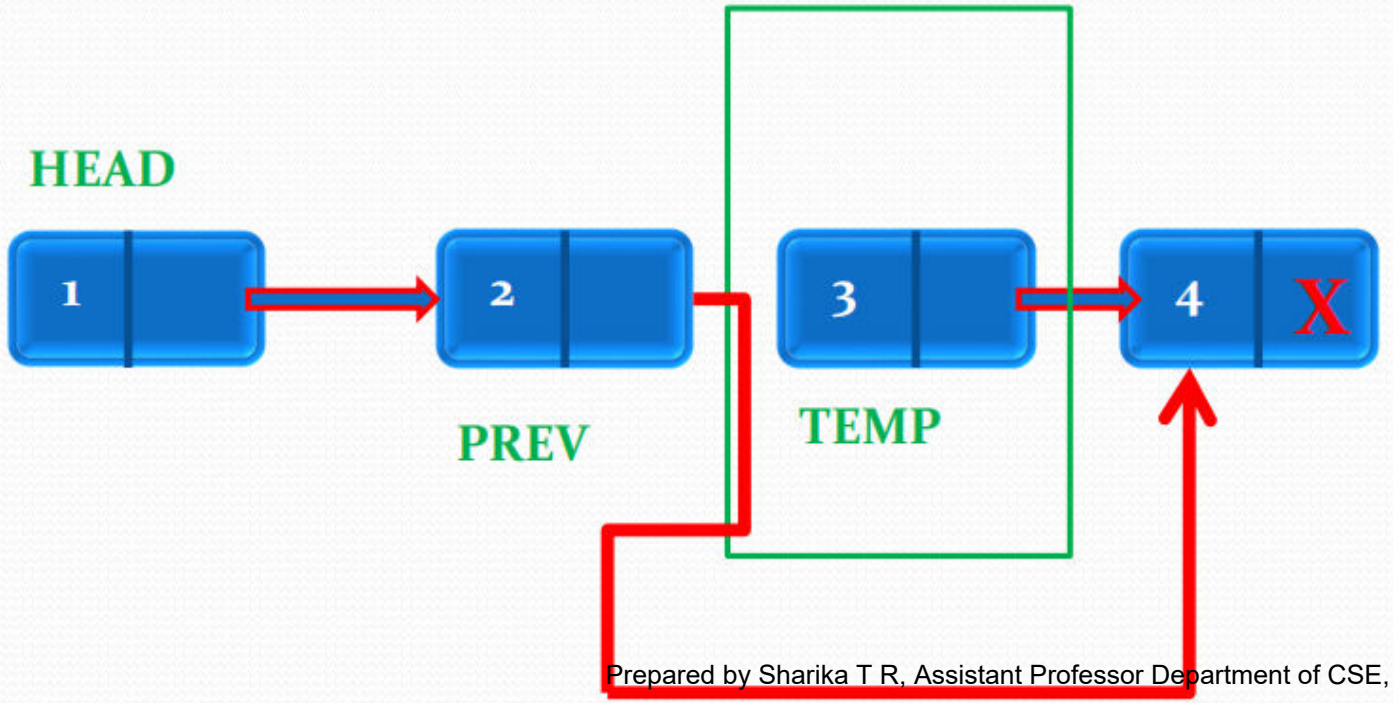


LL- DELETION from middle of the list

1. If(flag=o)
    1. Print("Key not found)
    2. Exit
2. Else
    1. Item=temp.data
    2. Prev.link=temp.link
    3. FREE (temp)

Flag = 1

Item = 3

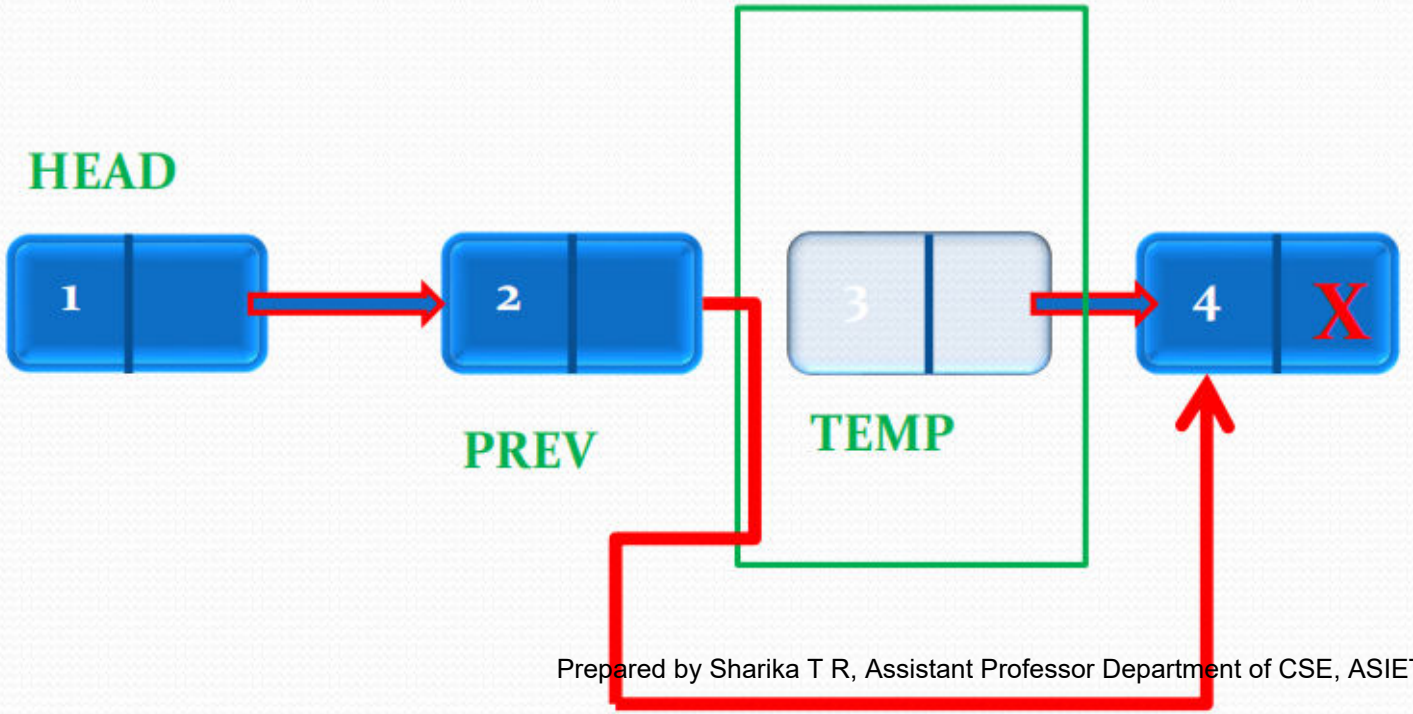


LL- DELETION from middle of the list

1. If(flag=o)
    1. Print("Key not found)
    2. Exit
2. Else
    1. Item=temp.data
    2. Prev.link=temp.link
    3. FREE (temp)

Flag = 1

Item = 3

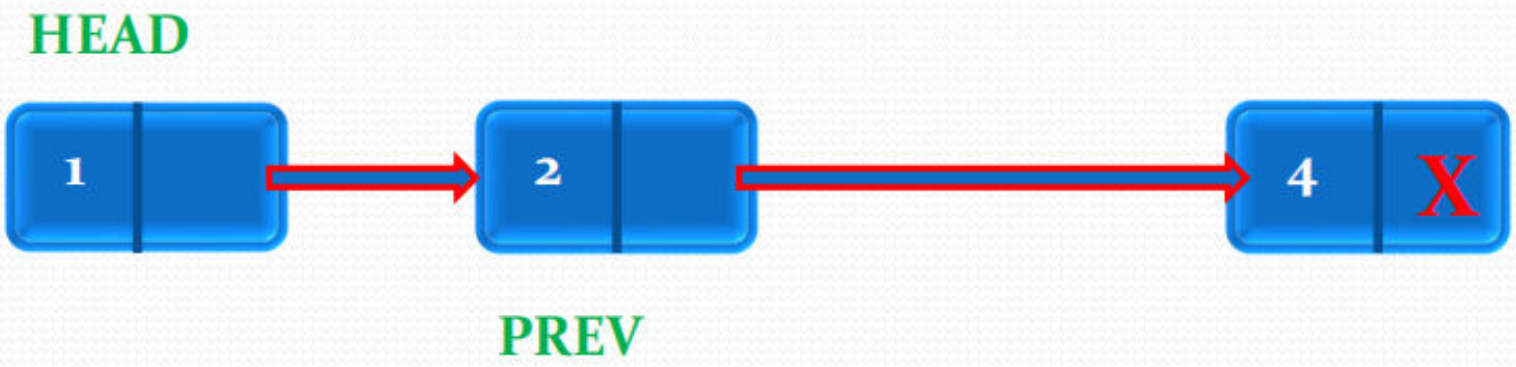


LL- DELETION from middle of the list

1. If(flag=o)
    1. Print("Key not found)
    2. Exit
2. Else
    1. Item=temp.data
    2. Prev.link=temp.link
    3. FREE (temp)

Flag = 1

Item = 3



Polynomial representation using linked lists

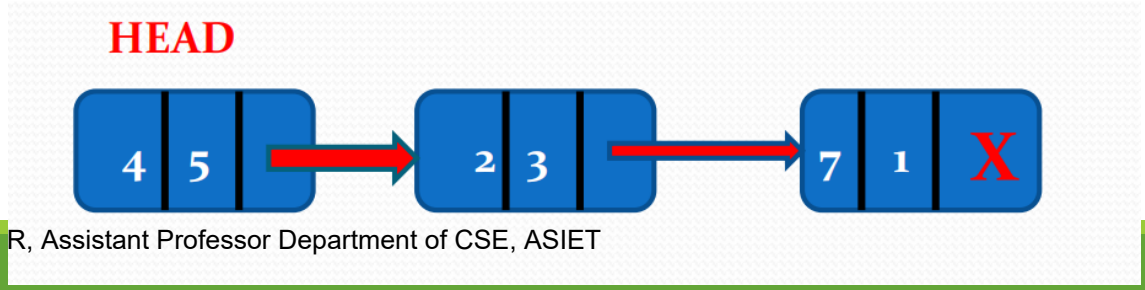
Each term in a polynomial expression comprise of occurrence of variable x is attached to two values:

- Coefficient
- Exponent

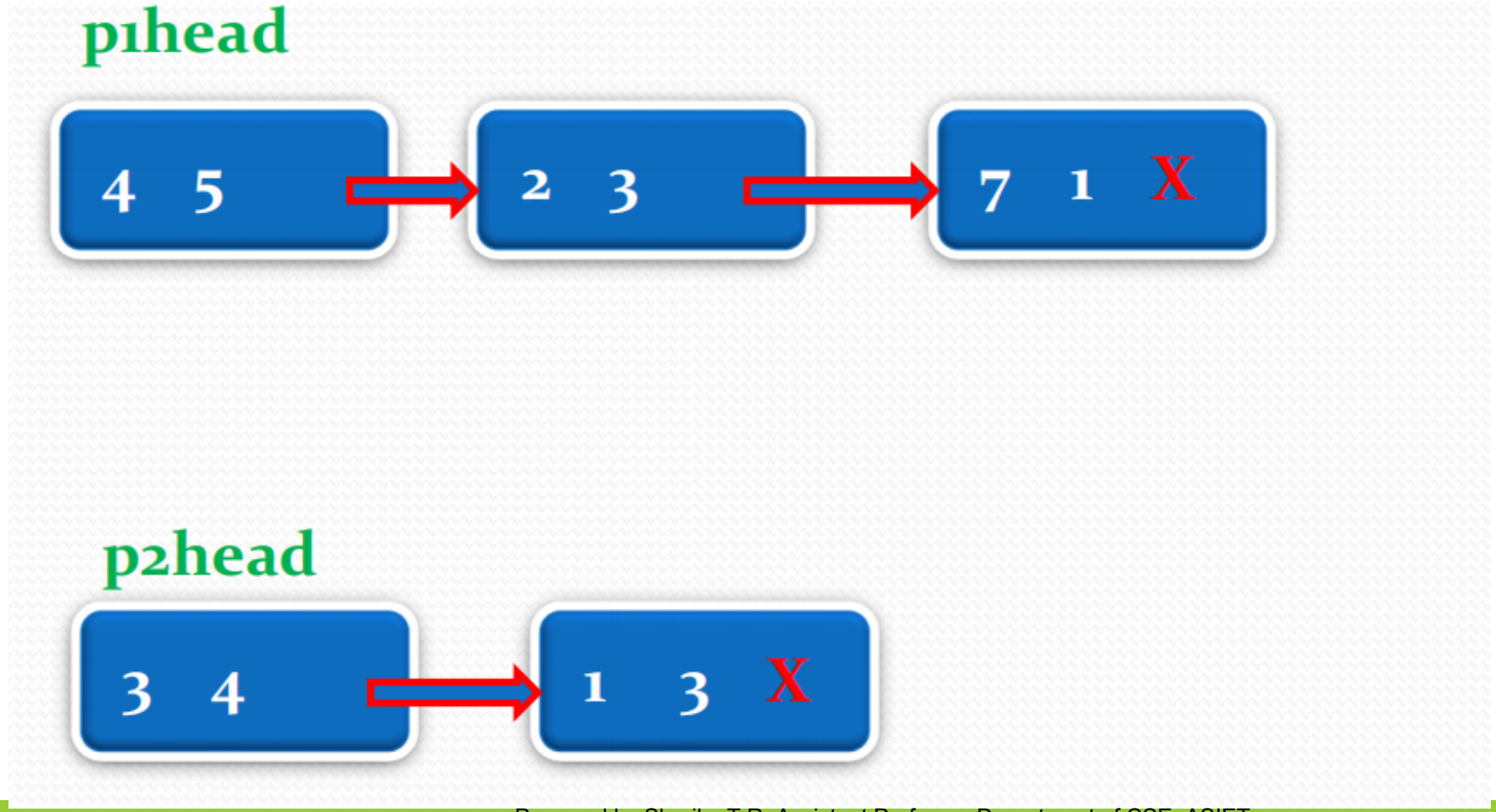
Each node in the linked list holds three pieces of information:

- Coefficient
- Exponent
- Address of the next node in sequence

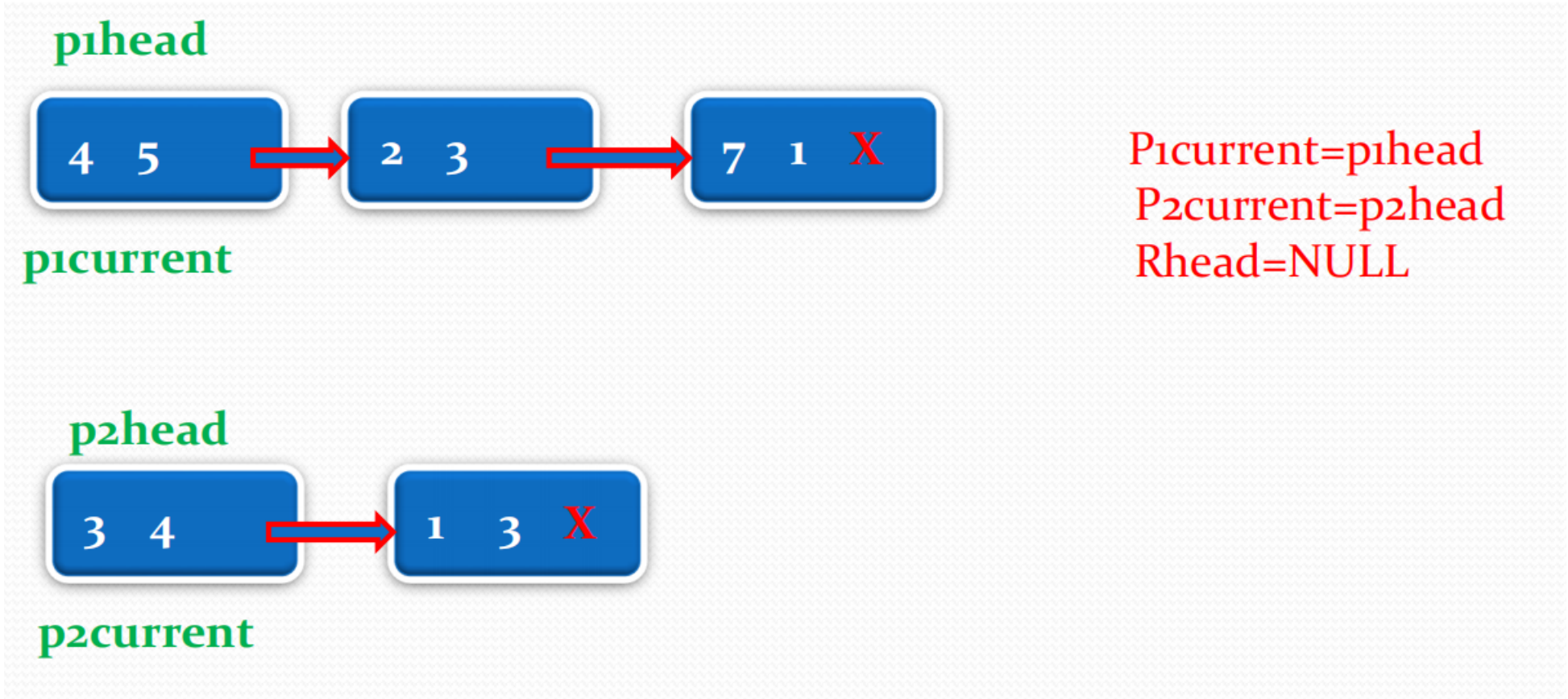
Eg: $4x^5 + 2x^3 + 7x$

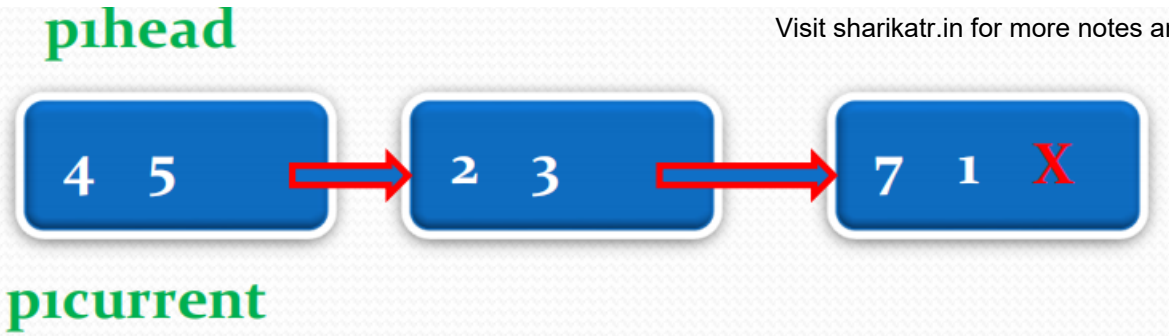


Add $4x^5 + 2x^3 + 7x$ and $3x^4 + 1x^3$

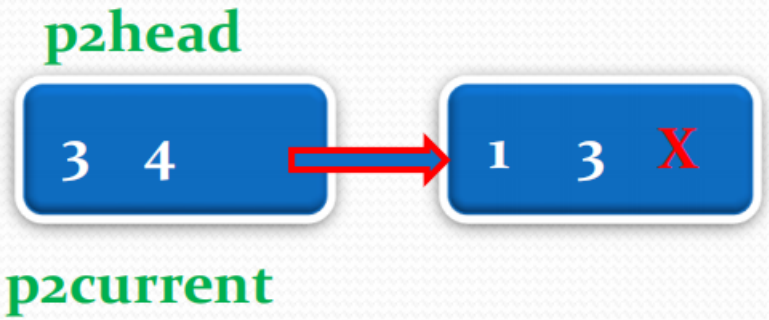


Add $4x^5 + 2x^3 + 7x$ and $3x^4 + 1x^3$

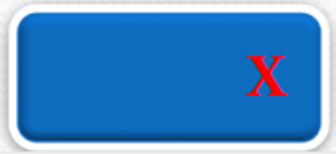


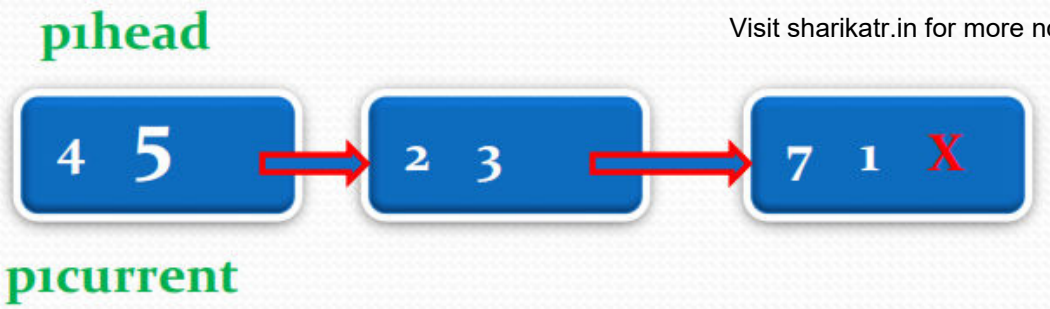


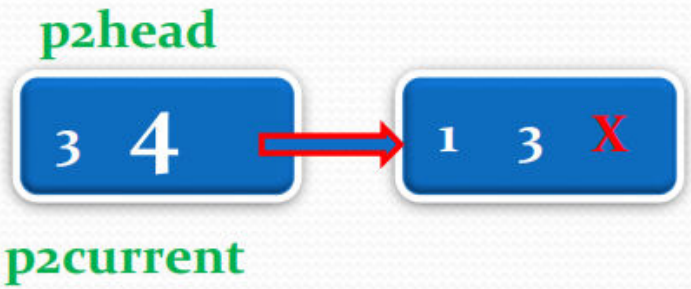

```
if(p1current!=NULL AND p2current!=NULL)
    Newnode=malloc(NODE)
    Newnode.link=NULL
```



Rhead=NULL

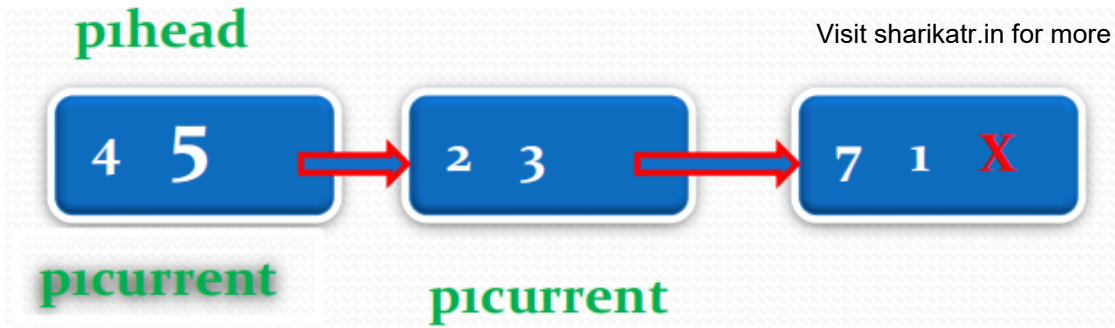


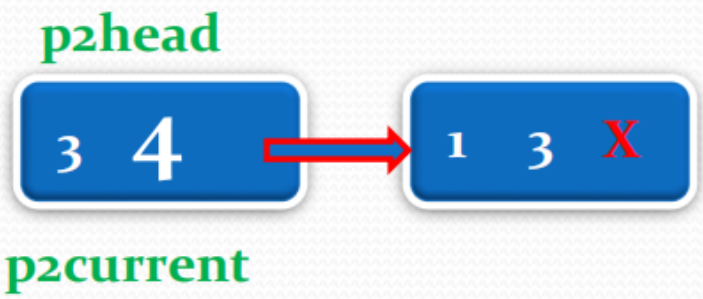





```
If(p1current.exp>p2current.exp)
    Newnode.coeff=p1current.coeff
    Newnode.exp=p1current.exp
    P1current=p1current.link
```

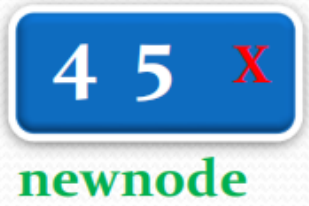
Rhead=NULL

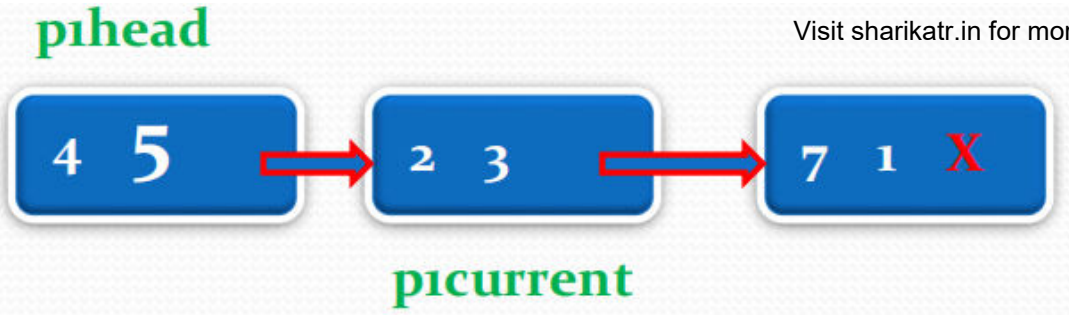




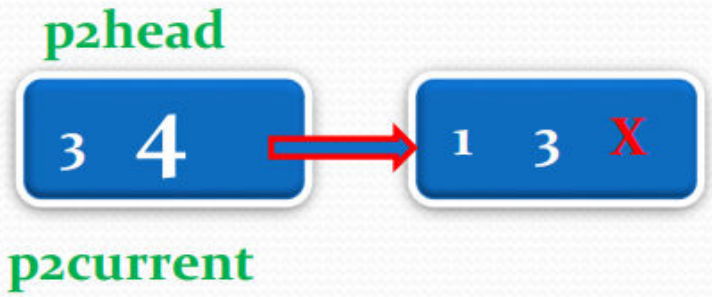

```
If(p1current.exp>p2current.exp)
    Newnode.coeff=p1current.coeff
    Newnode.exp=p1current.exp
    P1current=p1current.link
```

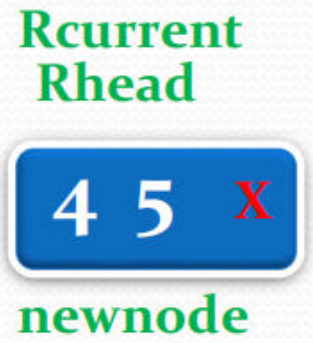
Rhead=NULL

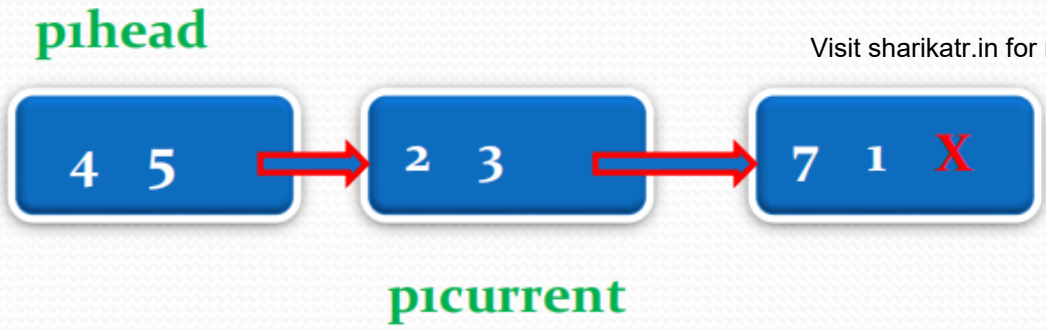




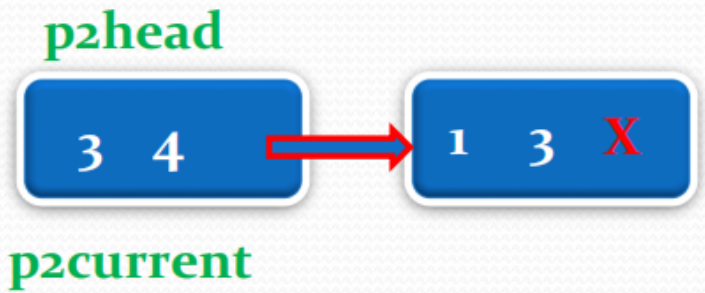

```
If(rhead=NULL)
    Rhead=newnode
    Rcurrent=Rhead
```

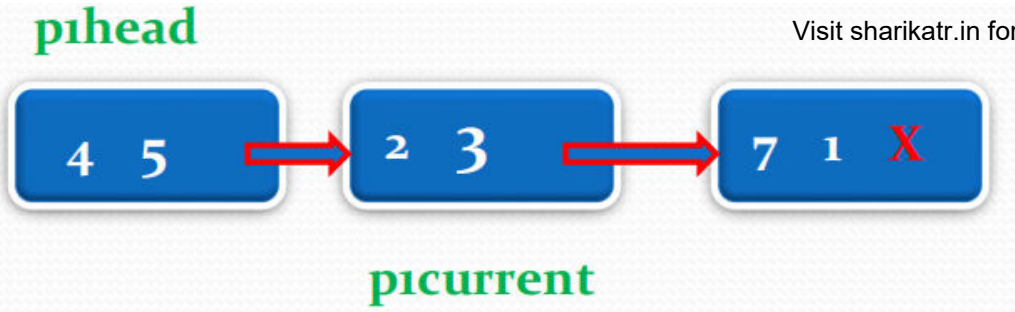


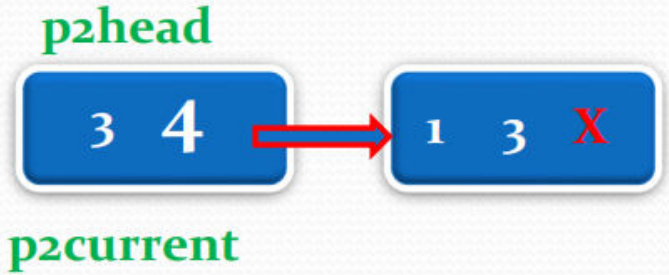




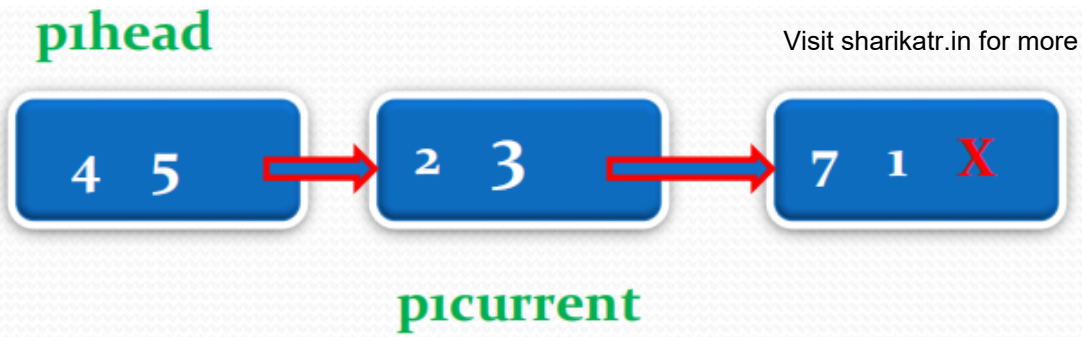

```
if(p1current!=NULL AND p2current!=NULL)
    Newnode=malloc (NODE)
    Newnode.link=NULL
```

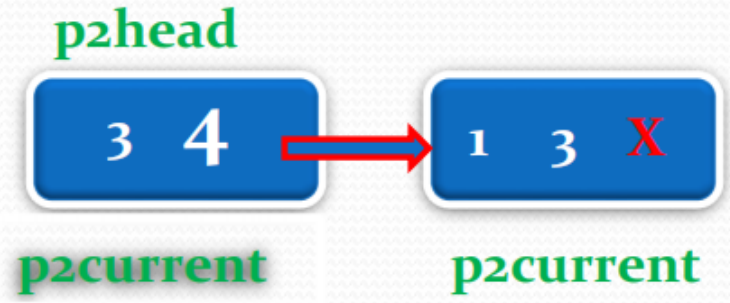




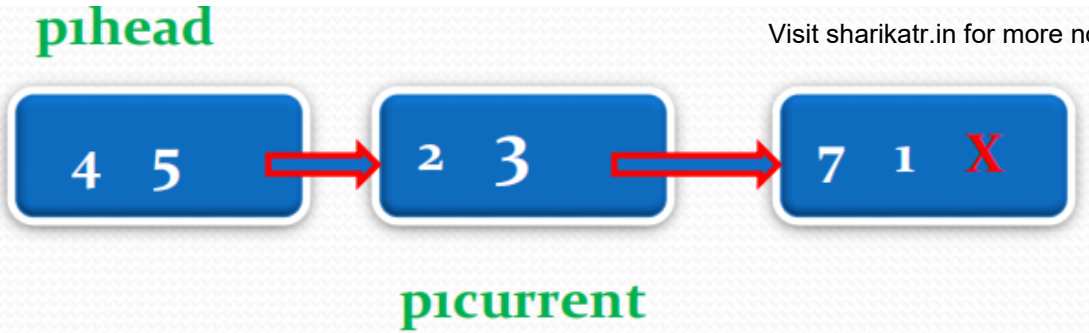


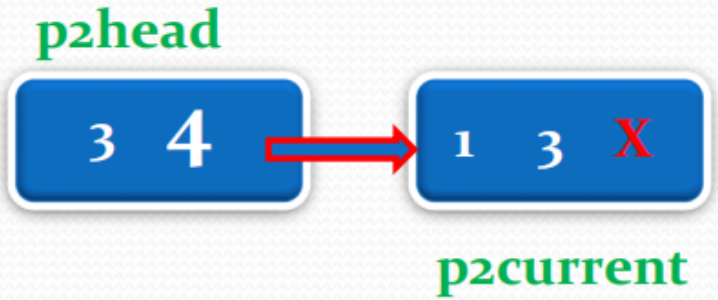

```
If(p2current.exp>p1current.exp)
    Newnode.coeff=p2current.coeff
    Newnode.exp=p2current.exp
    P2current=p2current.link
```



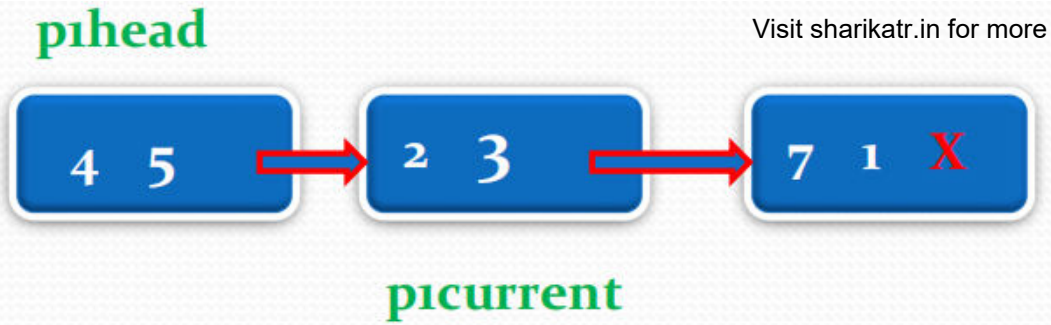


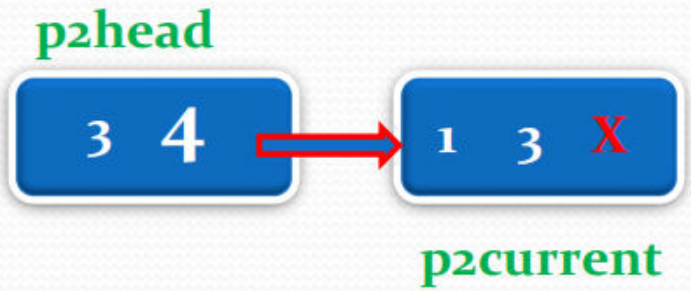

```
If(p2current.exp>p1current.exp)
    Newnode.coeff=p2current.coeff
    Newnode.exp=p2current.exp
    P2current=p2current.link
```



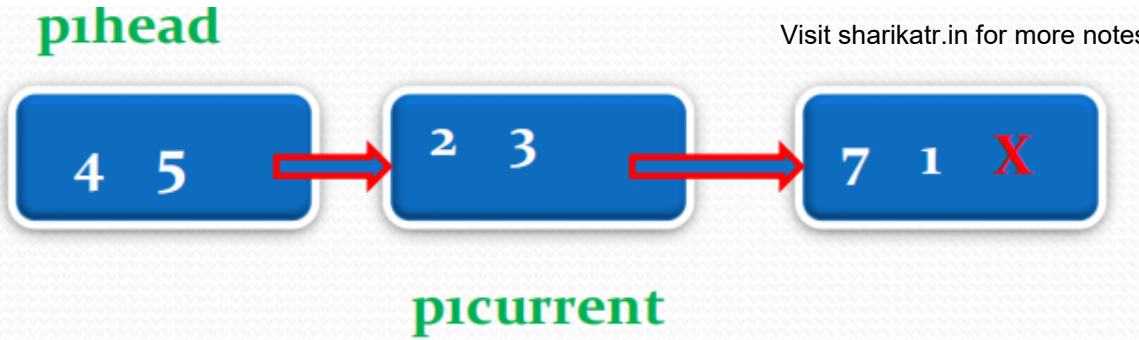



```
If(rhead !=NULL)
    Rcurrent.link=newnode
    Rcurrent=newnode
```

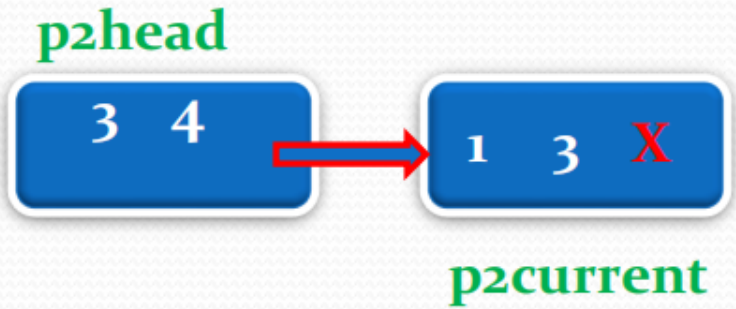


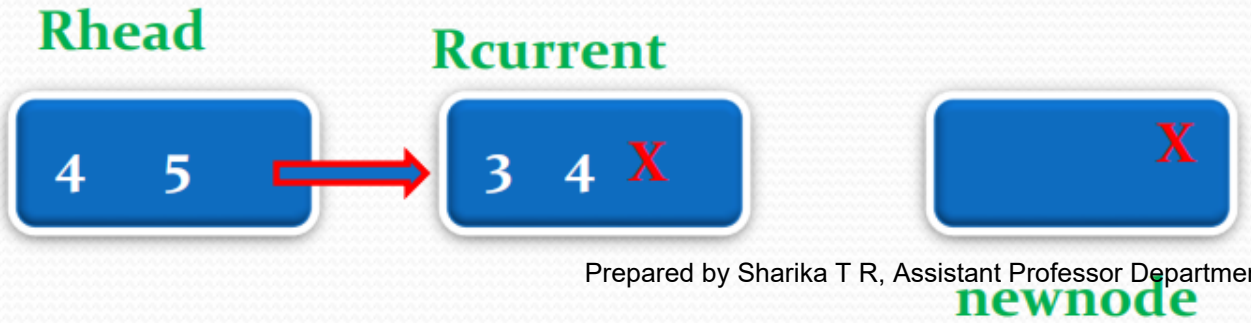


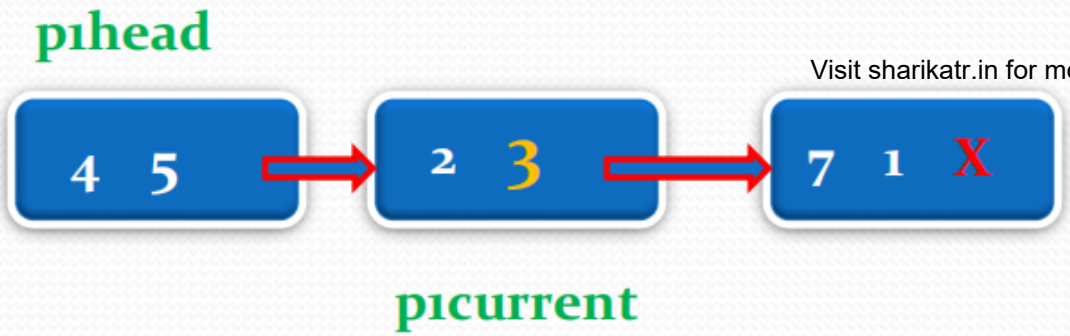

```
If(rhead !=NULL)
    Rcurrent.link=newnode
    Rcurrent=newnode
```

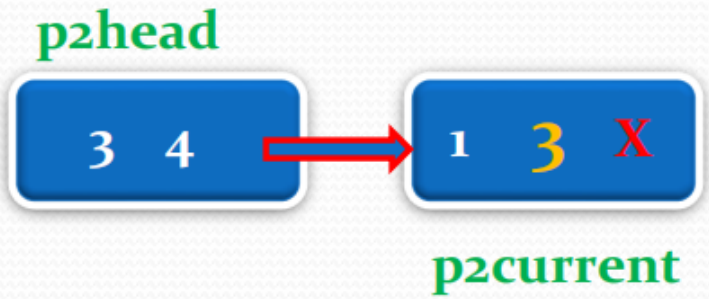



```
if(p1current!=NULL AND p2current!=NULL)
    Newnode=malloc (NODE)
    Newnode.link=NULL
```

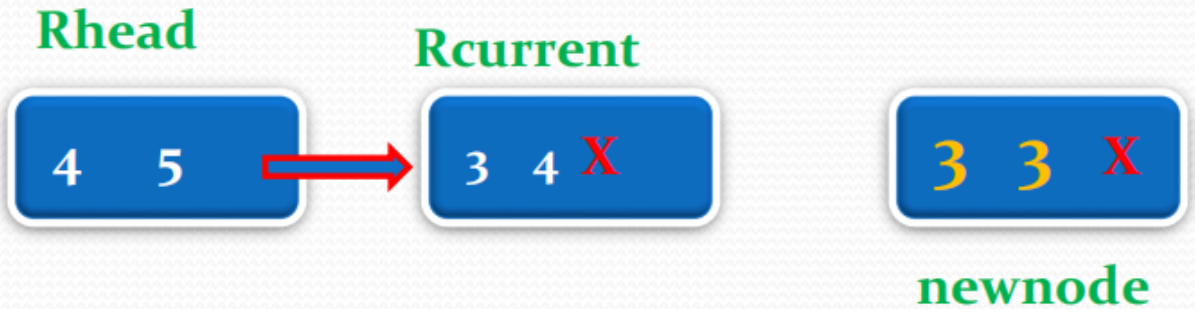


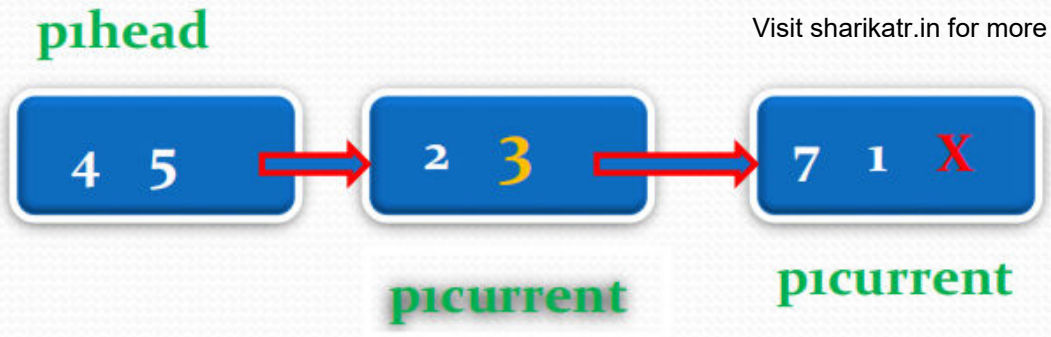




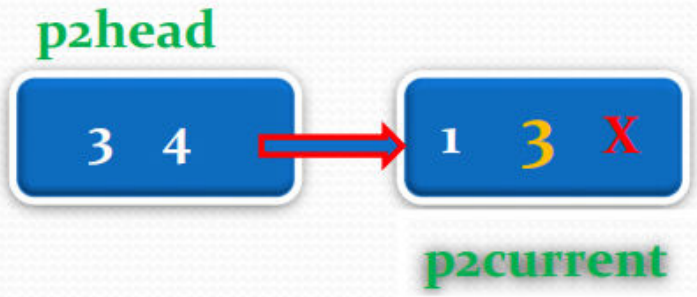



```
If(p1current.exp=p2current.exp)
    Newnode.coeff=p1current.coeff+p2current.coeff
    Newnode.exp=p1current.exp
    P1current=p1current.link
    P2current=p2current.link
```

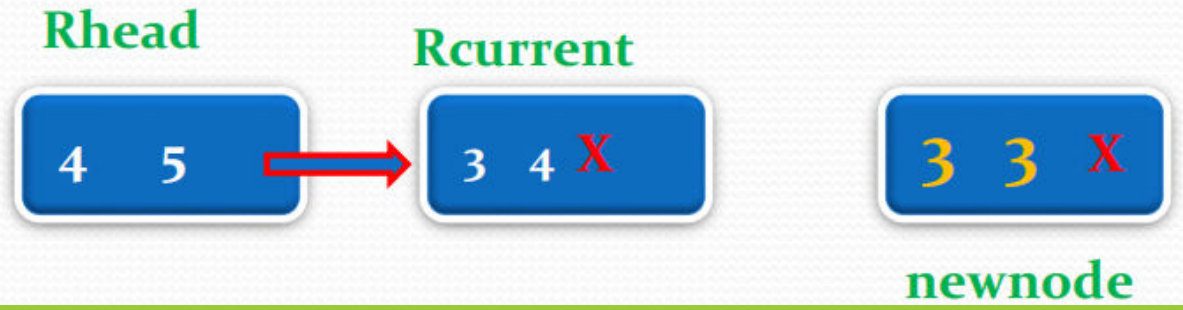


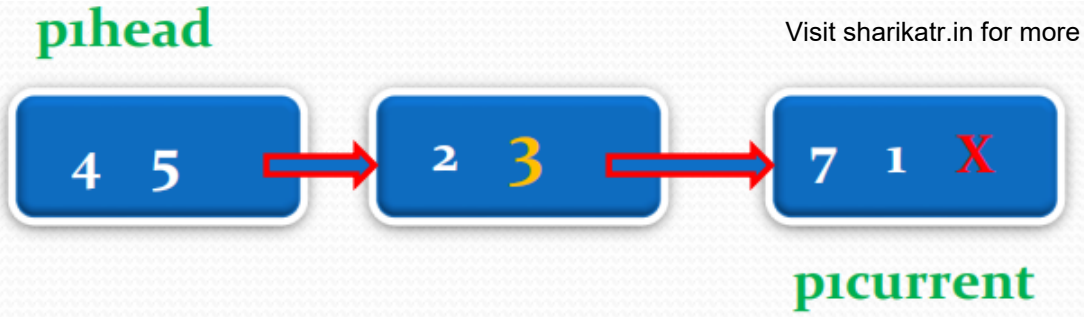



```
If(p1current.exp=p2current.exp)
    Newnode.coeff=p1current.coeff+p2current.coeff
    Newnode.exp=p1current.exp
    P1current=p1current.link
    P2current=p2current.link
```

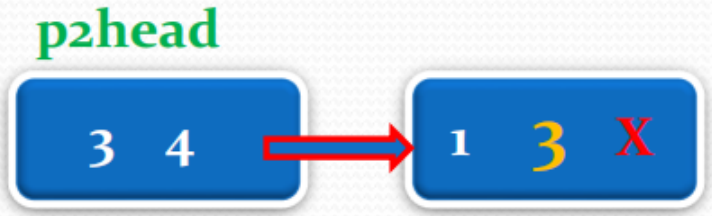


P2current=NULL

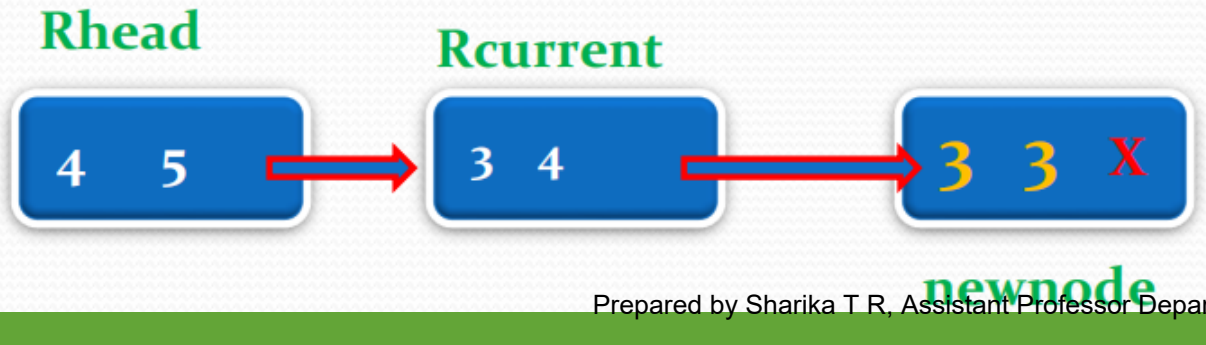


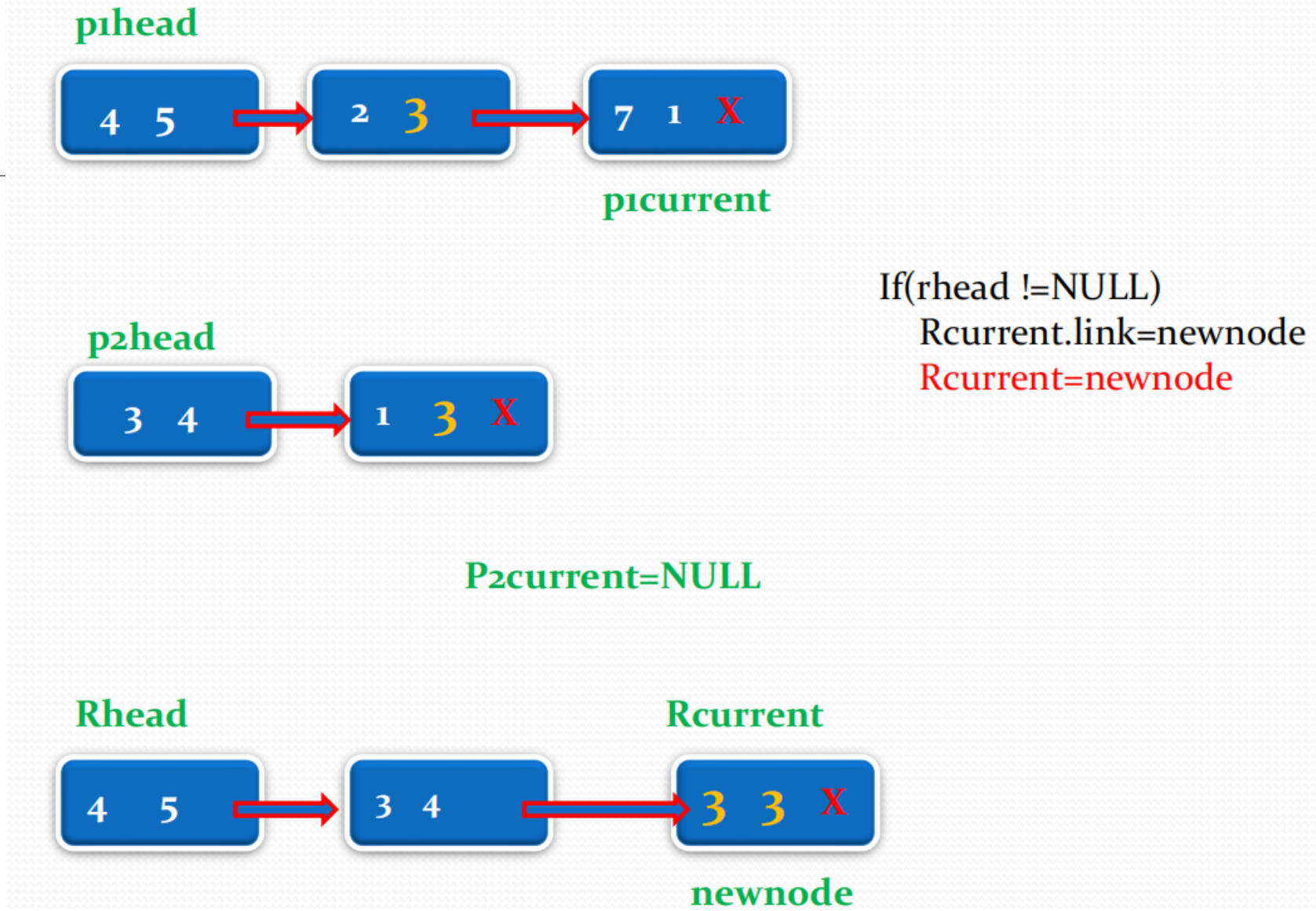


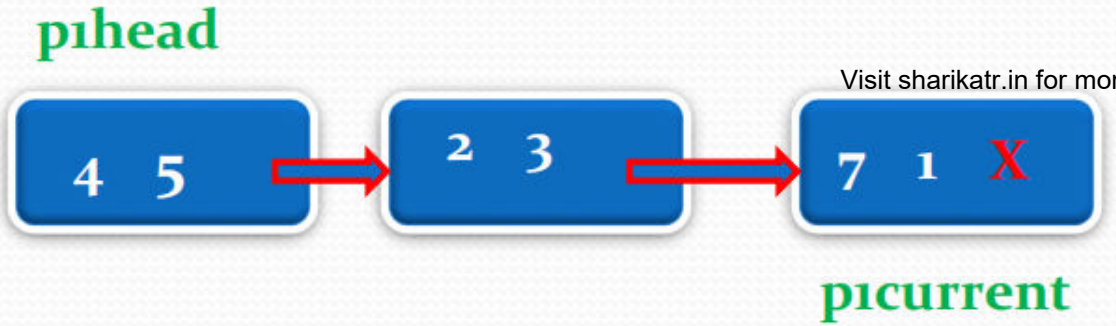

```
If(rhead !=NULL)
    Rcurrent.link=newnode
    Rcurrent=newnode
```



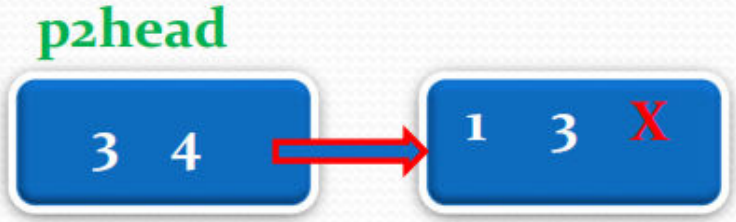
P2current=NULL



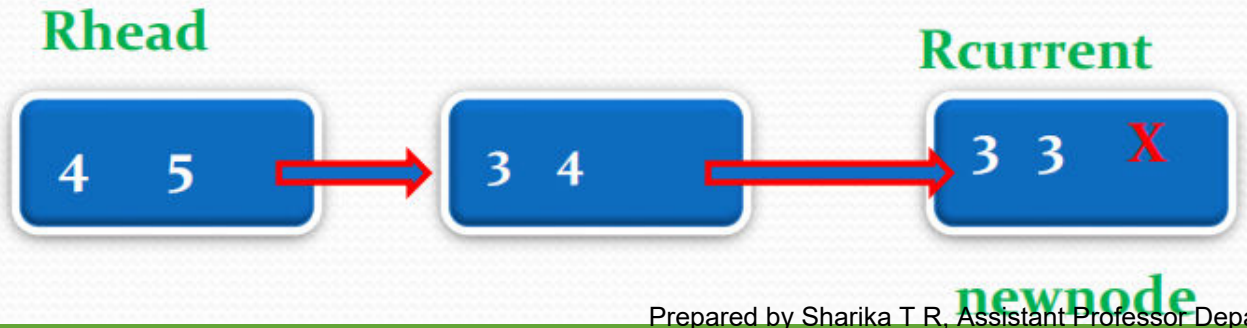

p1head
4 5
2 3
7 1 X
p1current
If(rhead !=NULL)
Rcurrent.link=newnode
Rcurrent=newnode
p2head
3 4
1 3 X
P2current=NULL
Rhead
Rcurrent
4 5
3 4
3 3 X
newnode

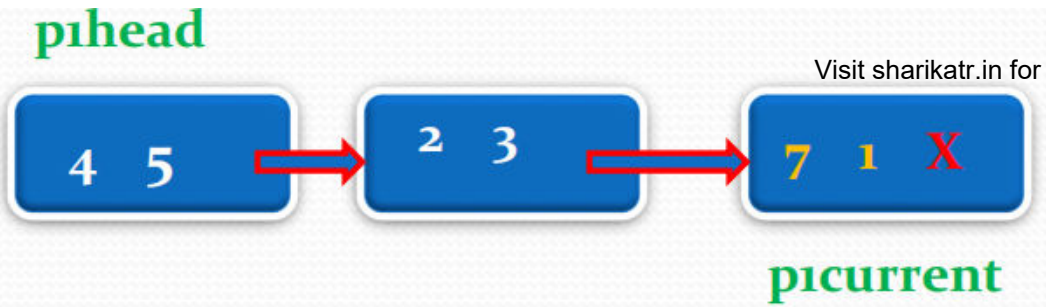



```
If (p1current!=NULL)
    Newnode=malloc (NODE)
    Newnode.coeff=p1current.coeff
    Newnode.exp=p1current.exp
    Newnode.link=NULL
    P1current=p1current.link
If(rhead=NULL)
        Rhead=newnode
        Rcurrent = rhead
Else
        Rcurrent.link=newnode
        Rcurrent-newnode
EndIf
```

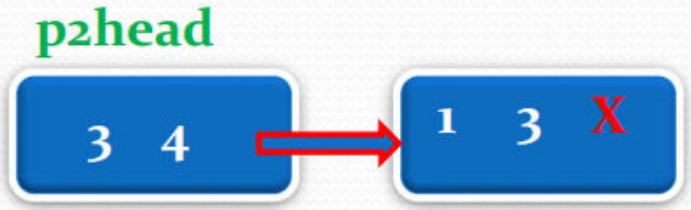


P2current=NULL

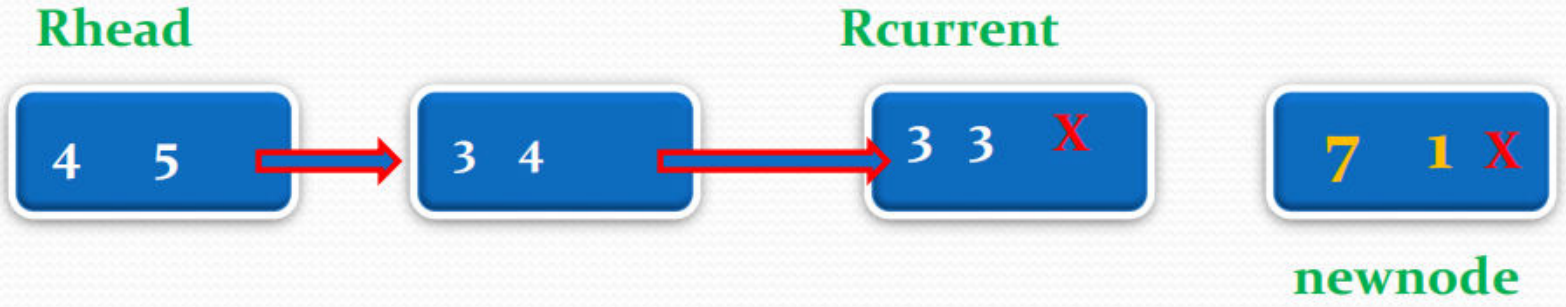


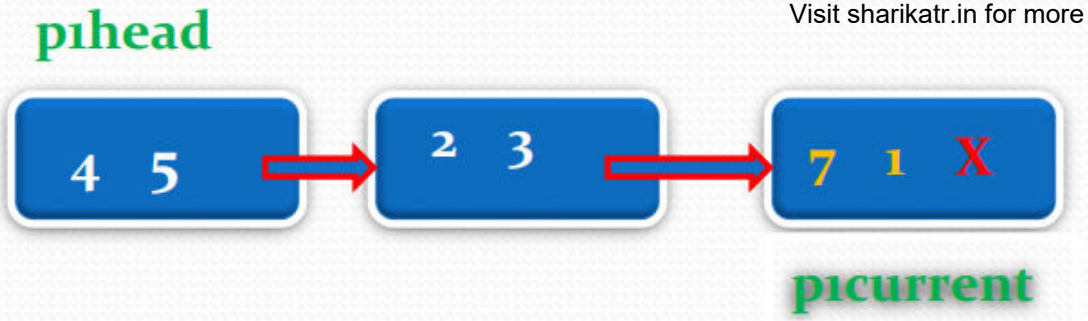



```
if(p1current!=NULL)
    Newnode=malloc (NODE)
    Newnode.coeff=p1current.coeff
    Newnode.exp=p1current.exp
    Newnode.link=NULL
    P1current=p1current.link
If(rhead=NULL)
        Rhead=newnode
        Rcurrent = rhead
Else
        Rcurrent.link=newnode
        Rcurrent=newnode
EndIf
```



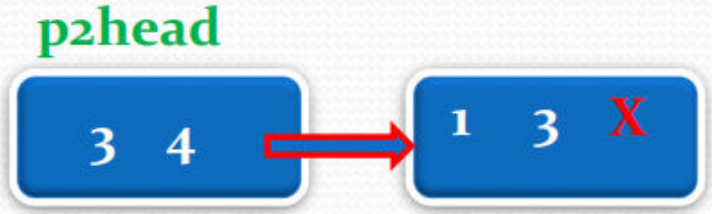
P2current=NULL



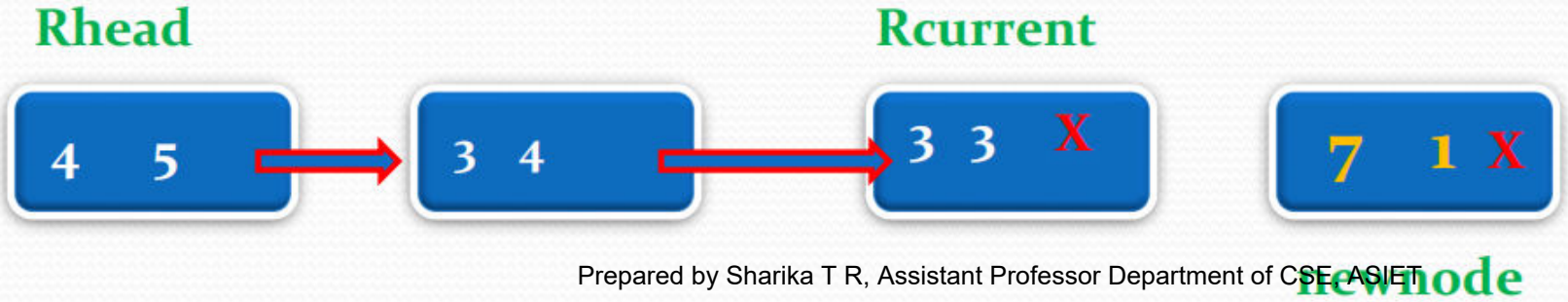


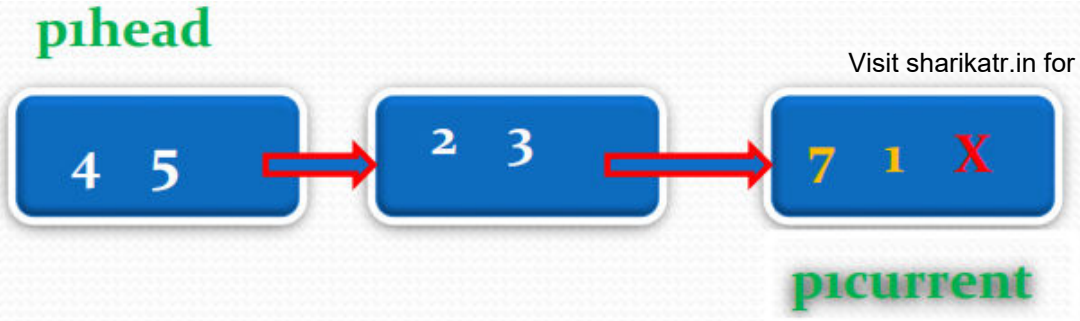

```
if(p1current!=NULL)
    Newnode=malloc (NODE)
    Newnode.coeff=p1current.coeff
    Newnode.exp=p1current.exp
    Newnode.link=NULL
    P1current=p1current.link
If(rhead=NULL)
        Rhead=newnode
        Rcurrent = rhead
Else
        Rcurrent.link=newnode
        Rcurrent=newnode
EndIf
```

P1current=NULL



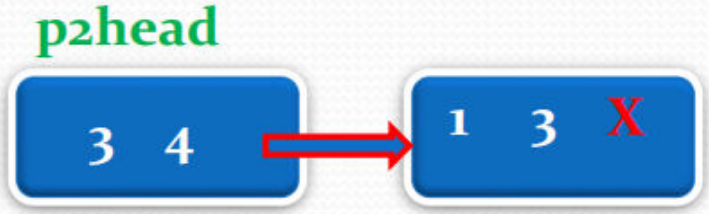
P2current=NULL



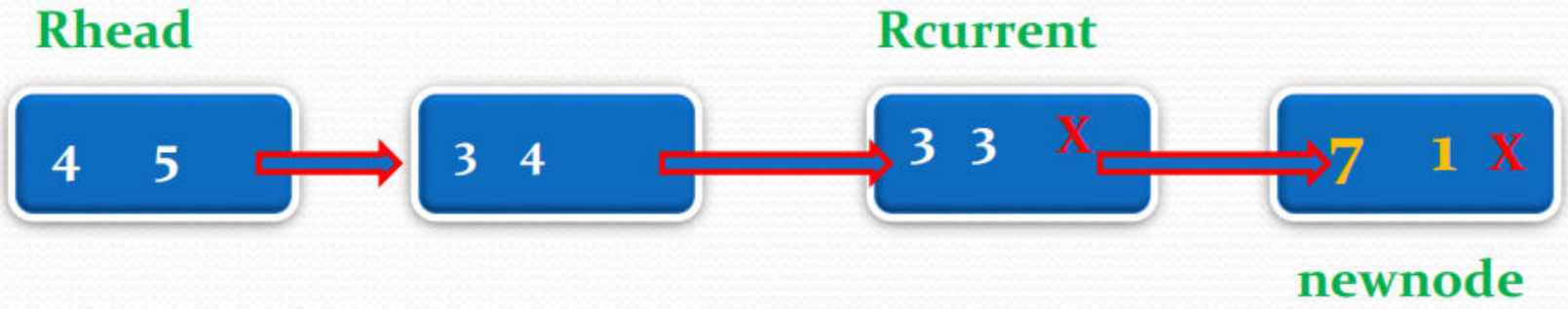



```
if(p1current!=NULL)
    Newnode=malloc (NODE)
    Newnode.coeff=p1current.coeff
    Newnode.exp=p1current.exp
    Newnode.link=NULL
    P1current=p1current.link
If(rhead=NULL)
        Rhead=newnode
        Rcurrent = rhead
Else
        Rcurrent.link=newnode
        Rcurrent=newnode
EndIf
```

P1current=NULL



P2current=NULL



Algorithm

Input:

1.Read two polynomials.

Output:

1.Displays the sum of the two polynomials given.

Algo:

Step 1: Start.

Step 2: Define user defined datatype node consisting of int coefficient and exponent

1. P1current=p1head
2. P2current=p2head
3. Rhead=NULL
4. While(p1current!=NULL OR p2current!=NULL)
    1. Newnode=malloc (NODE)
    2. Newnode.link=NULL
    3. If (p1current!=NULL AND p2current!=NULL)
        1. If(p1current.exp=p2current.exp)
            1. Newnode.coeff=p1current.coeff+p2current.coeff
            2. Newnode.exp=p1current.exp
            3. P1current=p1current.link
            4. P2current=p2current.link
        2. ElseIf(p1current.exp>p2current.exp)
            1. Newnode.coeff=p1current.coeff
            2. Newnode.exp=p1current.exp
            3. P1current=p1current.link
        3. ElseIf(p2current.exp>p1current.exp)
            1. Newnode.coeff=p2current.coeff
            2. Newnode.exp=p2current.exp
            3. P2current=p2current.link
    4. Else if (p1current!=NULL)
        1. Newnode.coeff=p1current.coeff
        2. Newnode.exp=p1current.exp
        3. P1current=p1current.link
    5. Else if (p2current!=NULL)
        1. Newnode.coeff=p2current.coeff
        2. Newnode.exp=p2current.exp
        3. P2current=p2current.link
    6. EndIf
    7. If(rhead=NULL)
        1. Rhead=newnode
        2. Rcurrent = rhead
    8. Else
        1. Rcurrent.link=newnode
        2. Rcurrent=newnode
    9. EndIf
4. EndWhile

Doubly Linked List

singly linked list - one way list

can move from the header node to any node in one direction only.

double linked list -two-way linked list

can move in either direction, either from left to right or from right to left.

Operations on a Double Linked List

All the operations a mentioned for a single linked list can be implemented more efficiently for the double linked list.


X
1
2
3
X

Doubly Linked list consists of chain of elements, in which each element is referred to as a **node.**

A node consists of three parts:

Data: Refers to the **information** held by the node

RightLink: Holds the address of the **next** node in the list

LeftLink: Holds the address of the **previous** node in the list

DLL – insert at end

Newnode=malloc(NODE)

Newnode.data=item

Newnode.leftaddr=NULL

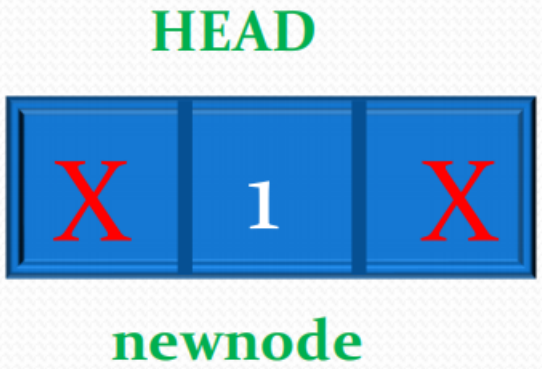
Newnode.rightaddr=NULL

HEAD= NULL

newnode

If(head=NULL)

Head=newnode

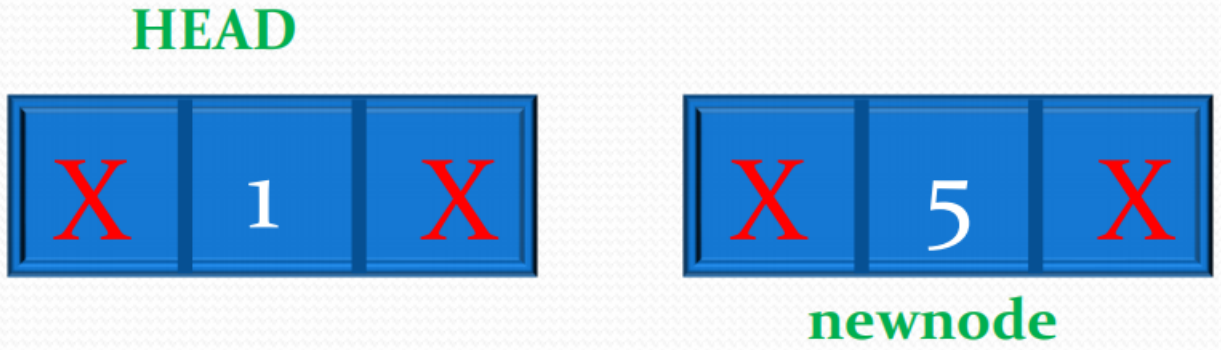


Newnode=malloc(NODE)

Newnode.data=item

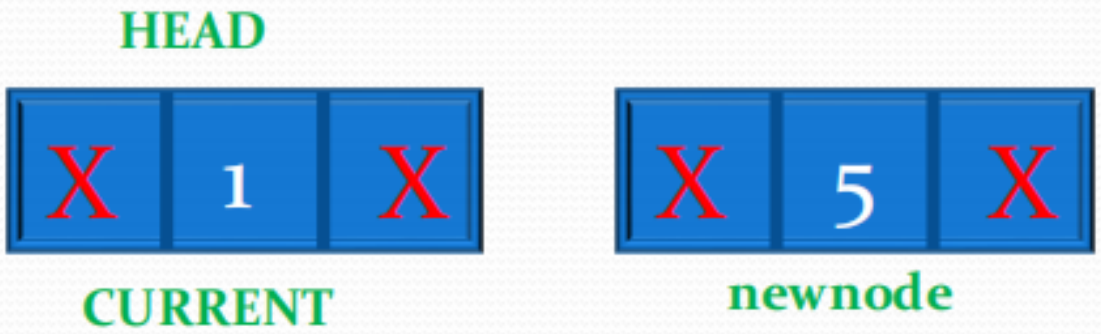
Newnode.leftaddr=NULL

Newnode.rightaddr=NULL



DLL – insert at end

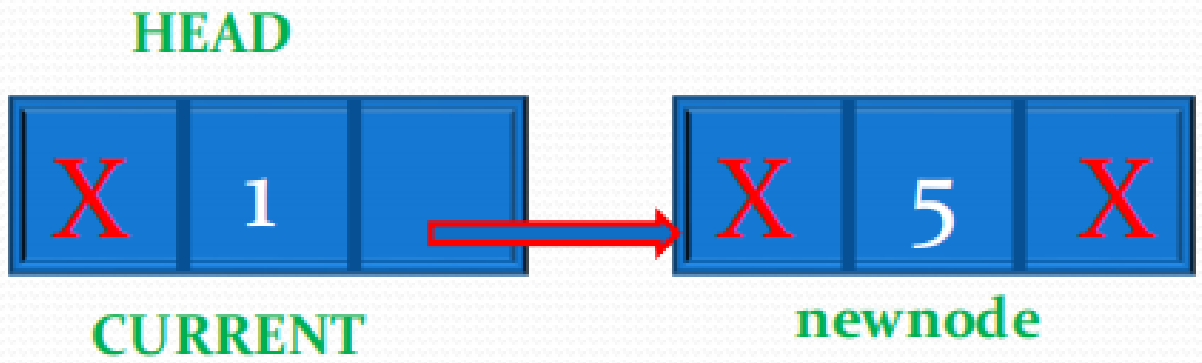
```
Current=head
While( current.rightaddr != NULL )
    Current=current.rightaddr
```



DLL – insert at end

Current.rightaddr=newnode

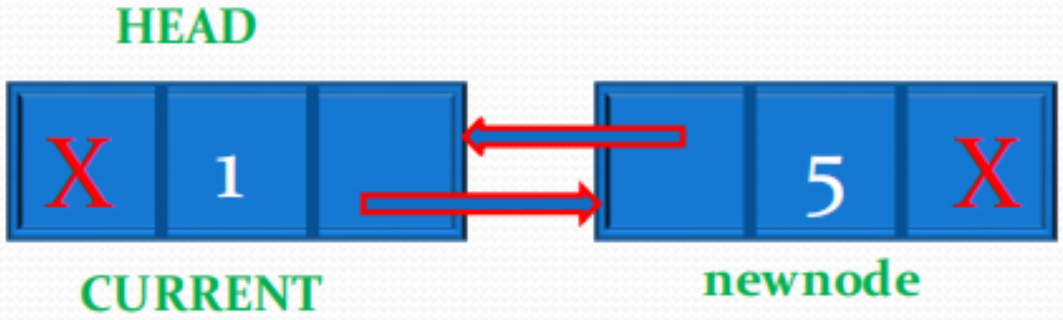
Newnode.leftaddr=current



DLL – insert at end

Current.rightaddr=newnode

Newnode.leftaddr=current



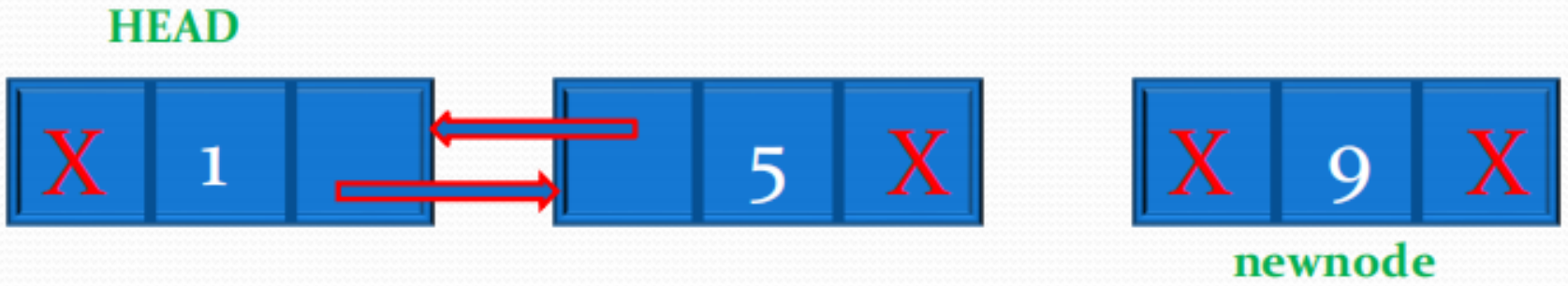
DLL – insert at end

Newnode=malloc(NODE)

Newnode.data=item

Newnode.leftaddr=NULL

Newnode.rightaddr=NULL

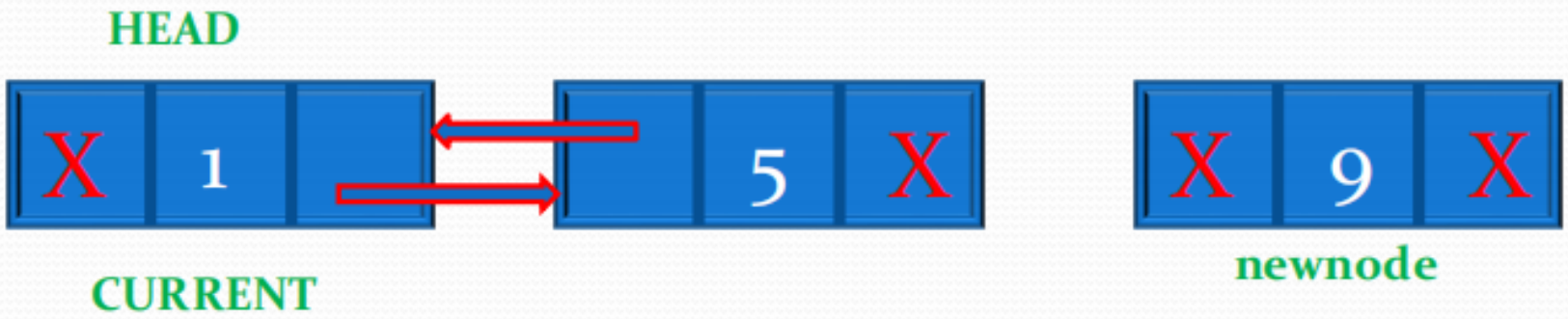


DLL – insert at end

Current=head

While(current.rightaddr!=NULL)

Current=current.rightaddr



DLL – insert at end


```
Current=head
While(current.rightaddr!=NULL)
   Current=current.rightaddr
```

DLL – insert at end

Current.rightaddr=newnode

Newnode.leftaddr=current

DLL – insert at end

Current.rightaddr=newnode

Newnode.leftaddr=current

DLL – insert at end

Algorithm: DL_INSERT_END(item)

Input: Item is the data content of the node to be inserted

Output: A double linked list enriched with the node containing data to be added at the end

Data Structure: Linked List

Steps:

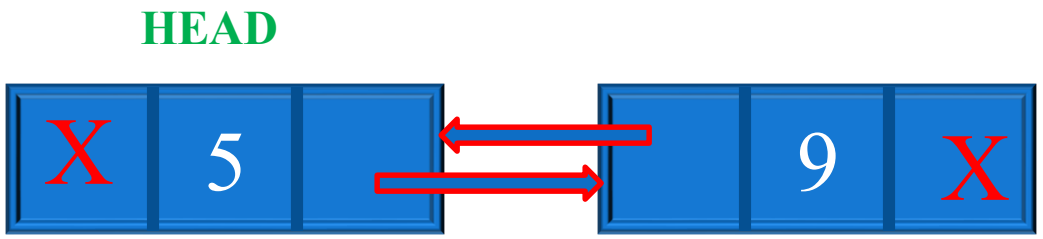
1. Newnode=malloc(NODE)
2. Newnode.data=item
3. Newnode.leftaddr=NULL
4. Newnode.rightaddr=NULL
5. If(head=NULL)
    1. Head=newnode
6. Else
    1. Current=head
    2. While(current.rightaddr!=NULL)
        1. Current=current.rightaddr
    3. EndWhile
    4. Current.rightaddr=newnode
    5. Newnode.leftaddr=current
7. EndIf
8. Stop

```c
#include <stdio.h>
#include <stdlib.h>

// Define the structure for a Node in the doubly linked list
typedef struct Node {
    int data;
    struct Node* leftaddr;
    struct Node* rightaddr;
} NODE;
```

```c
int main() {
    NODE* head = NULL;
    NODE* newnode;
    NODE* current;
    int item;

    // Insertion at the end - Example with 3 items
    for(int i = 0; i < 3; i++) {
        printf("Enter data for node %d: ", i+1);
        scanf("%d", &item);

        newnode = (NODE*)malloc(sizeof(NODE));
        newnode->data = item;
        newnode->leftaddr = NULL;
        newnode->rightaddr = NULL;
```

```
        if (head == NULL) {
            head = newnode;
        } else {
            current = head;
            while (current->rightaddr != NULL) {
                current = current->rightaddr;
            }
            current->rightaddr = newnode;
            newnode->leftaddr = current;
        }
    }

    // Displaying the list
    current = head; // Start from the head
    printf("Doubly Linked List: ");
    while (current != NULL) {
        printf("%d ", current->data);
        current = current->rightaddr;
    }
    printf("\n");

    return 0;
}
```

DLL – insert at FRONT



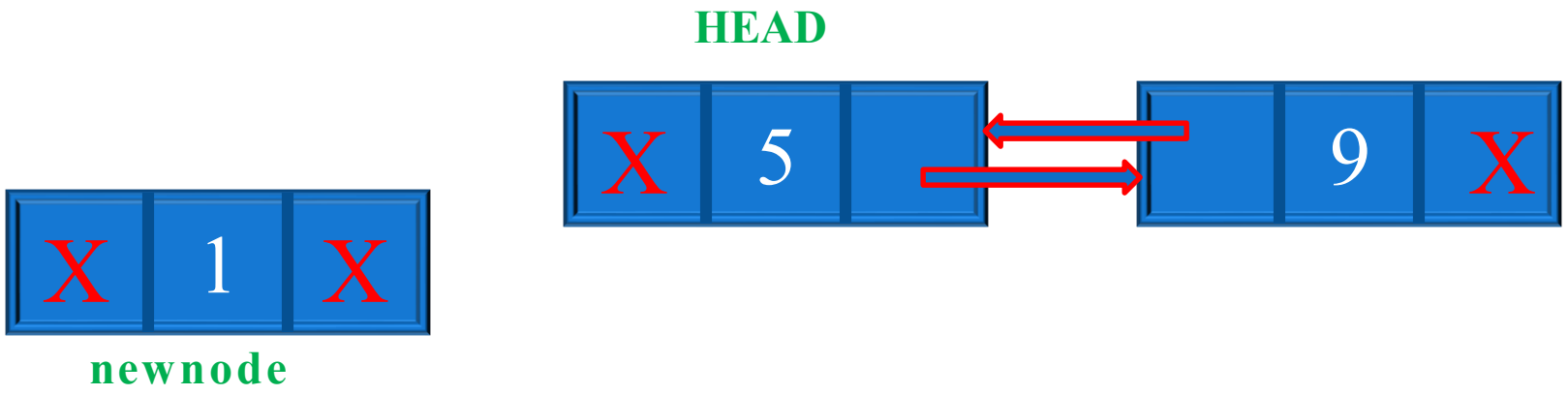
DLL – insert at FRONT

Newnode=malloc(NODE)

Newnode.data=item

Newnode.leftaddr=NULL

Newnode.rightaddr=NULL

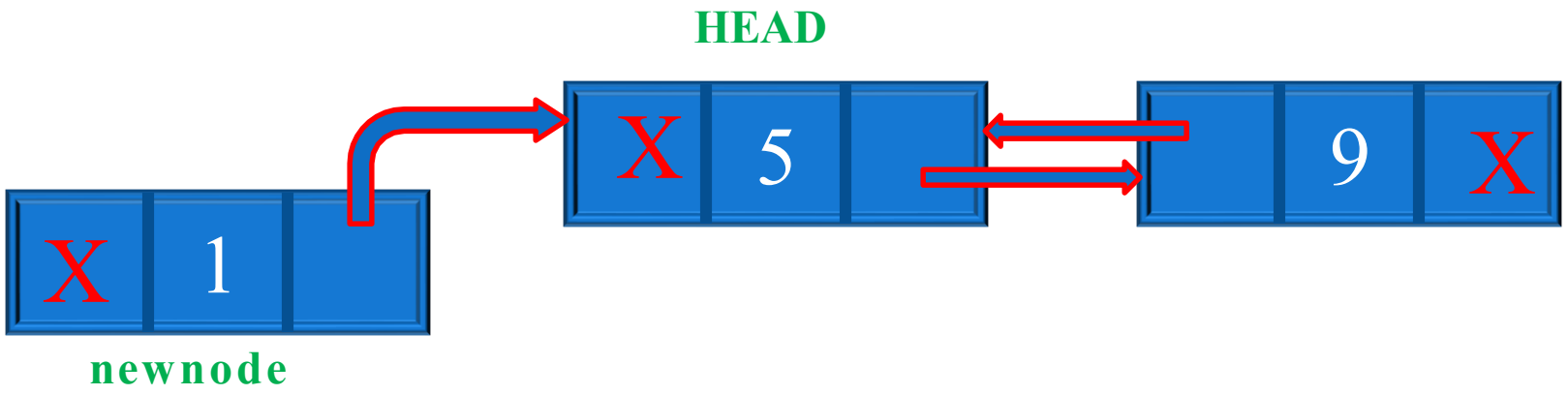


DLL – insert at FRONT

Newnode.rightaddr=head

Head.leftaddr=newnode

Head=newnode

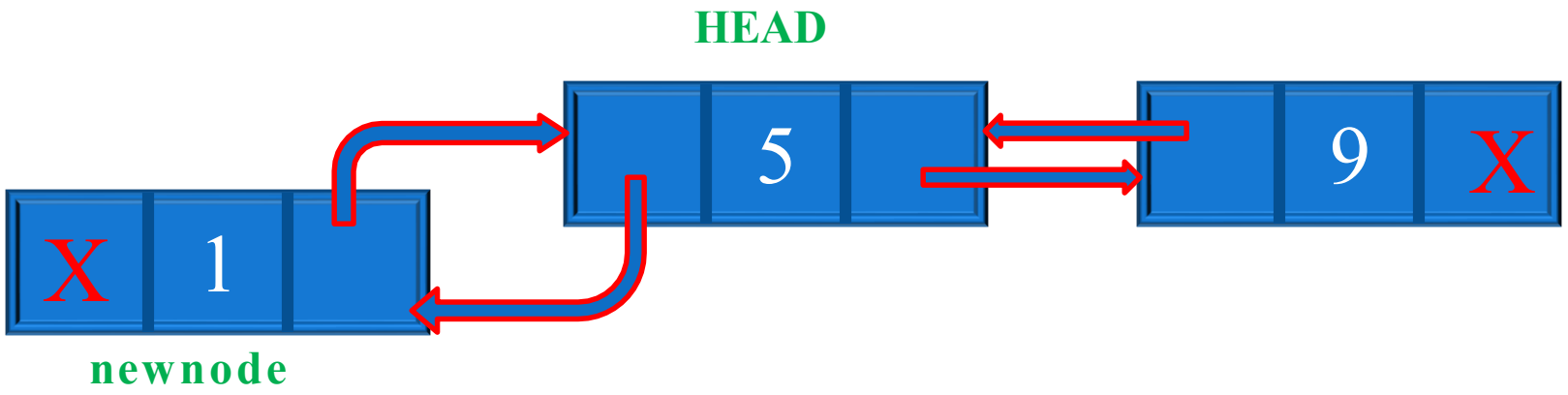


DLL – insert at FRONT

Newnode.rightaddr=head

Head.leftaddr=newnode

Head=newnode

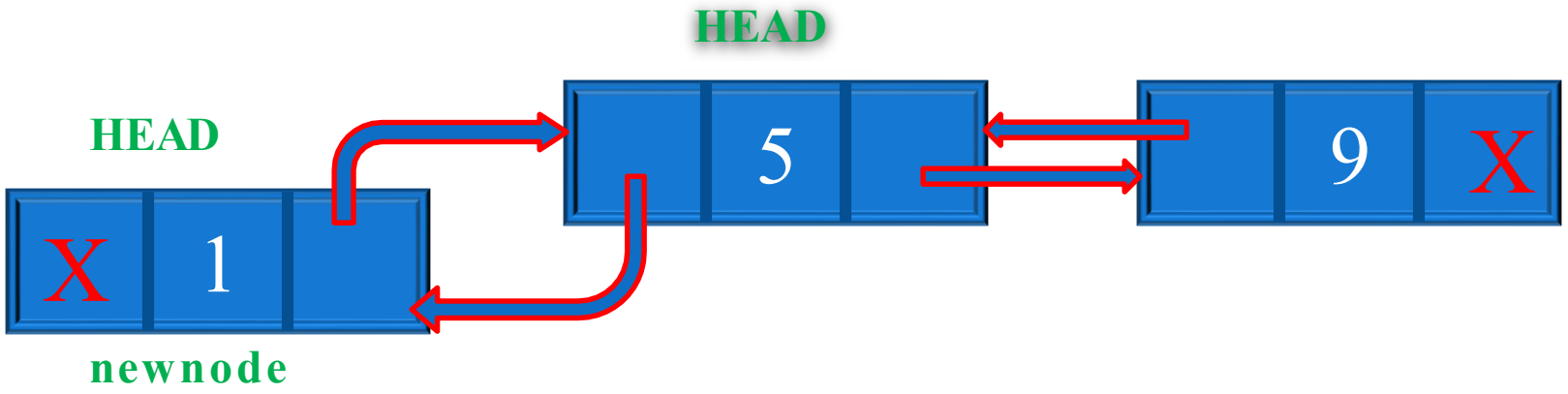


DLL – insert at FRONT

Newnode.rightaddr=head

Head.leftaddr=newnode

Head=newnode



DLL – insert at FRONT

Newnode.rightaddr=head

Head.leftaddr=newnode

Head=newnode

DLL – insert at front

Algorithm: DL_INSERT_FRONT(item)

Input: Item is the data content of the node to be inserted

Output: A double linked list enriched with the node containing data to be added at the front

Data Structure: Linked List

Steps:

1. Newnode=malloc(NODE)
2. Newnode.data=item
3. Newnode.leftaddr=NULL
4. Newnode.rightaddr=NULL
5. If(head=NULL)
    1. Head=newnode
6. Else
    1. Newnode.rightaddr=head
    2. Head.leftaddr=newnode
    3. Head=newnode
7. EndIf
8. Stop

DLL – insert after any position, say after 5

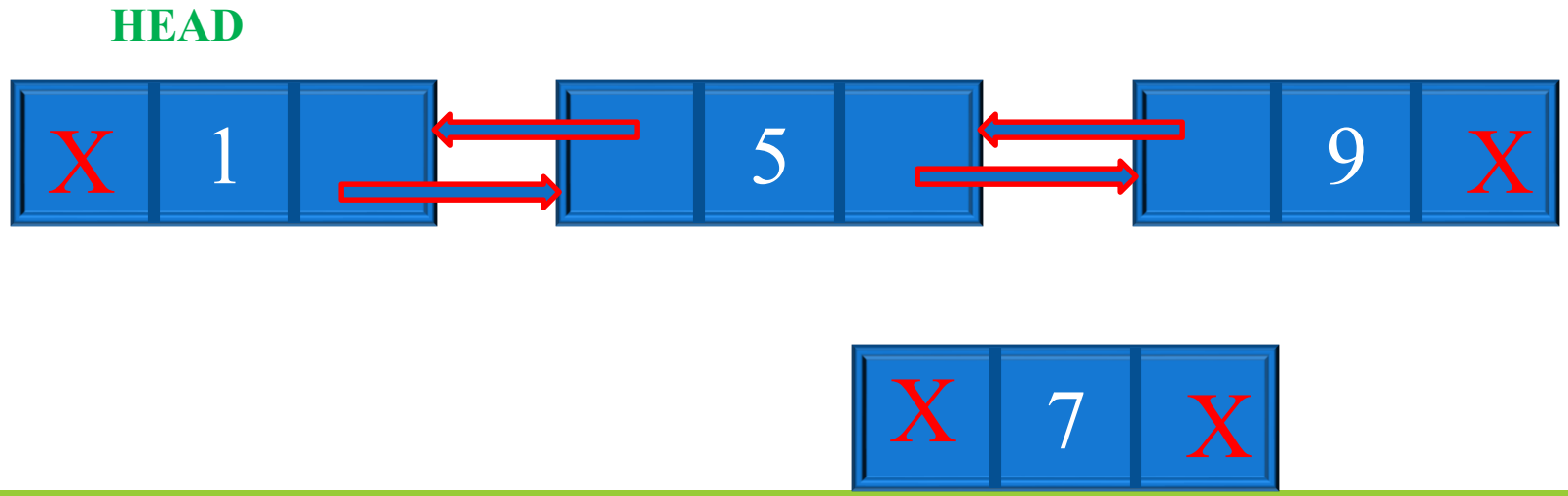
DLL – insert after any position, say after 5

Newnode=malloc(NODE)

Newnode.data=item

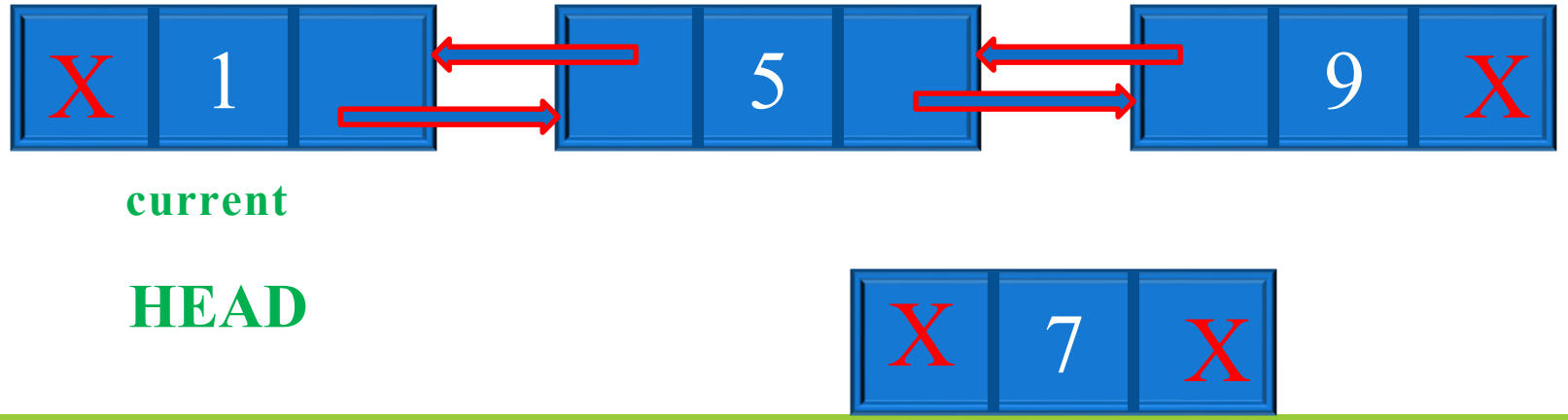
Newnode.leftaddr=NULL

Newnode.rightaddr=NULL



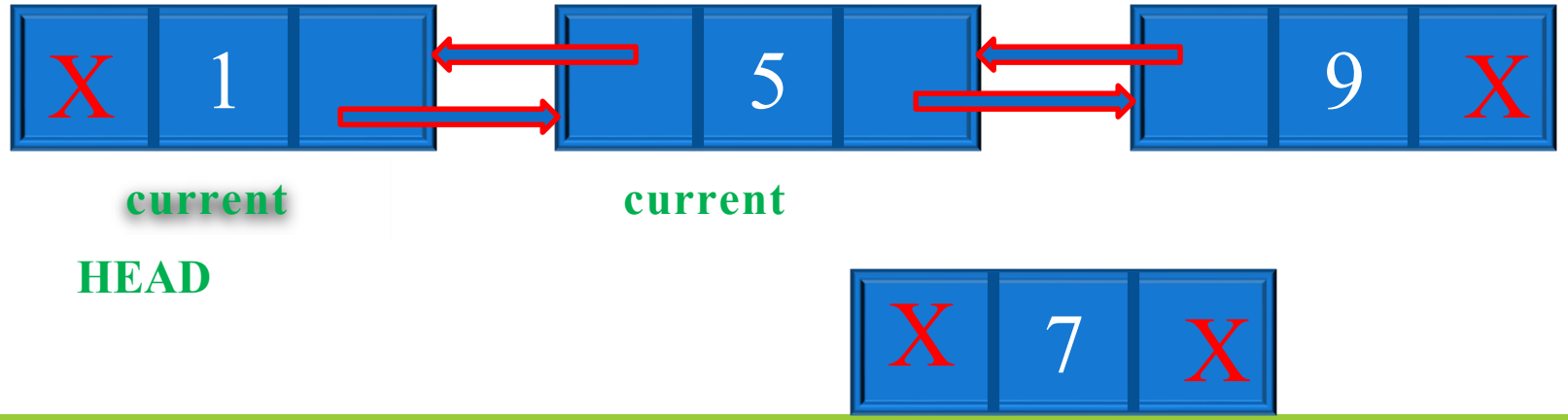
DLL – insert after any position, say after 5

```
Current=head
While(current!=NULL)
    If(current.data=key)
        Flag=1
    ExitWhile
    EndIf
    Current=current.rightaddr
EndWhile
```



DLL – insert after any position, say after 5

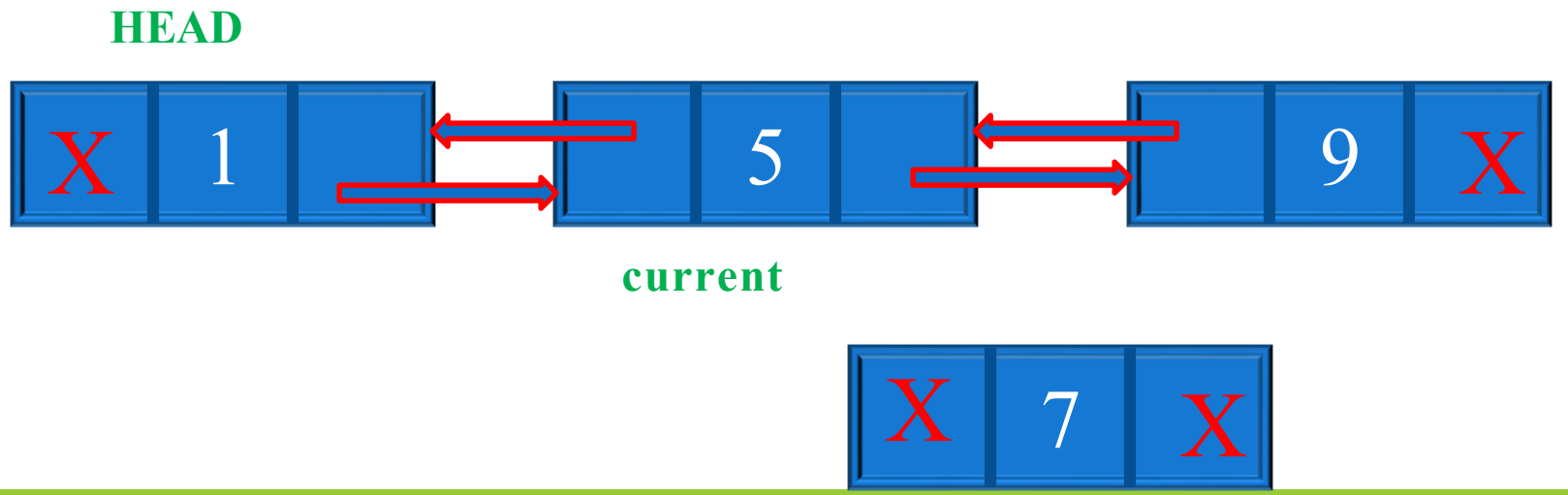

```
Current=head
While( current!=NULL )
    If( current.data=key )
        Flag=1
        ExitWhile
        EndIf
    Current=current.rightaddr
    EndWhile
```



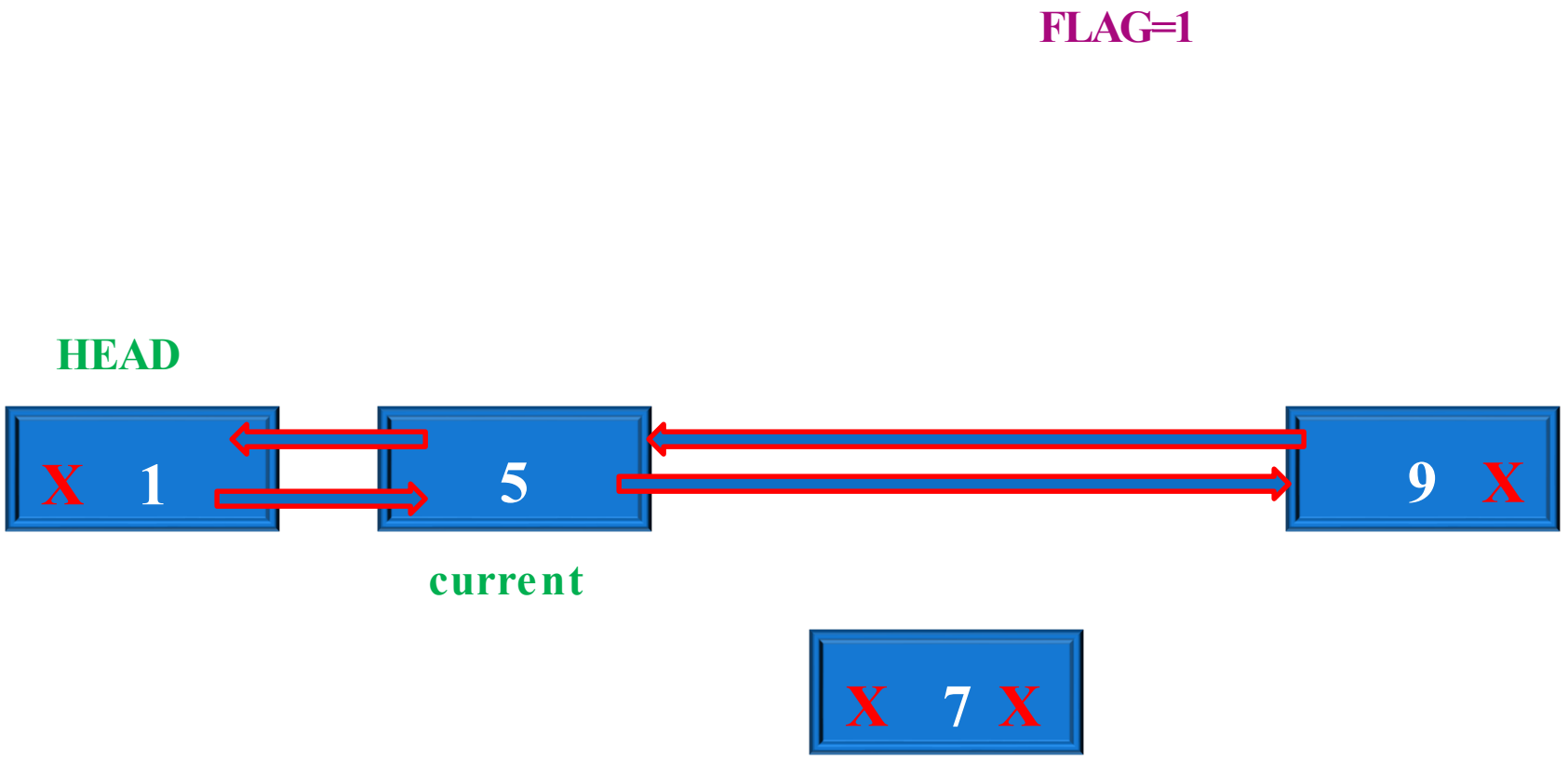
DLL – insert after any position, say after 5


```
Current=head
While( current!=NULL )
    If( current.data=key )
        Flag=1  ExitWhile
     EndIf
     Current=current.rightaddr
EndWhile
```

FLAG=1



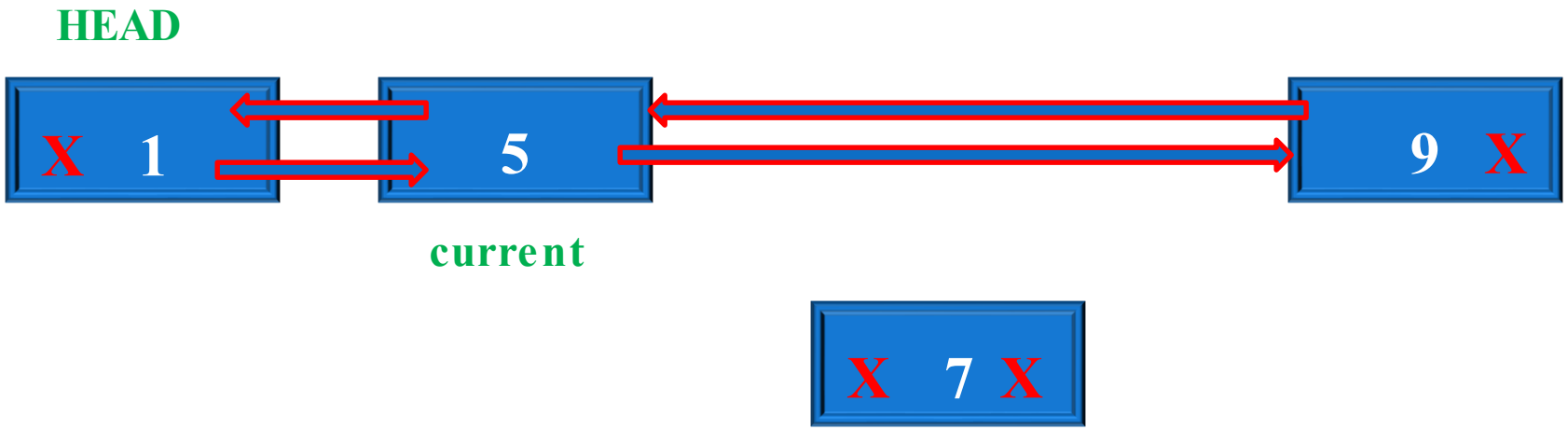
DLL – insert after any position, say after 5



DLL – insert after any position, say after 5

```
If(flag=1)
    If(current.rightaddr !=NULL)
        Temp=current.rightaddr
        Temp.leftaddr=newnode
        Newnode.rightaddr=temp
        Current.rightaddr=newnode
        Newnode.leftaddr=current
```

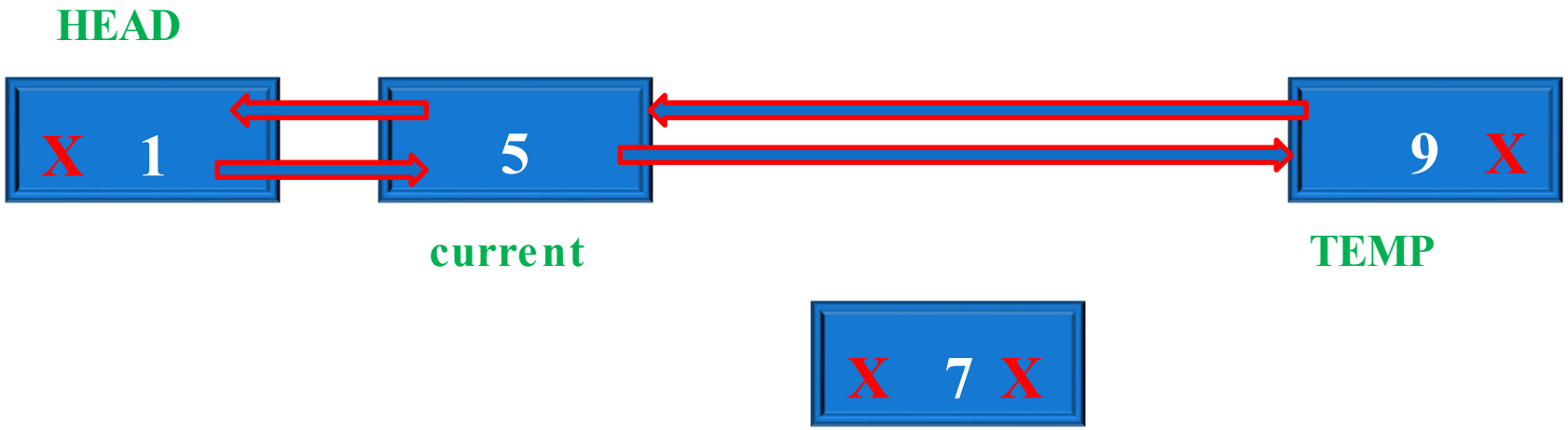
FLAG=1



DLL – insert after any position, say after 5

```
If(flag=1)
    If(current.rightaddr !=NULL)
        Temp=current.rightaddr
        Temp.leftaddr=newnode
        Newnode.rightaddr=temp
        Current.rightaddr=newnode
        Newnode.leftaddr=current
```

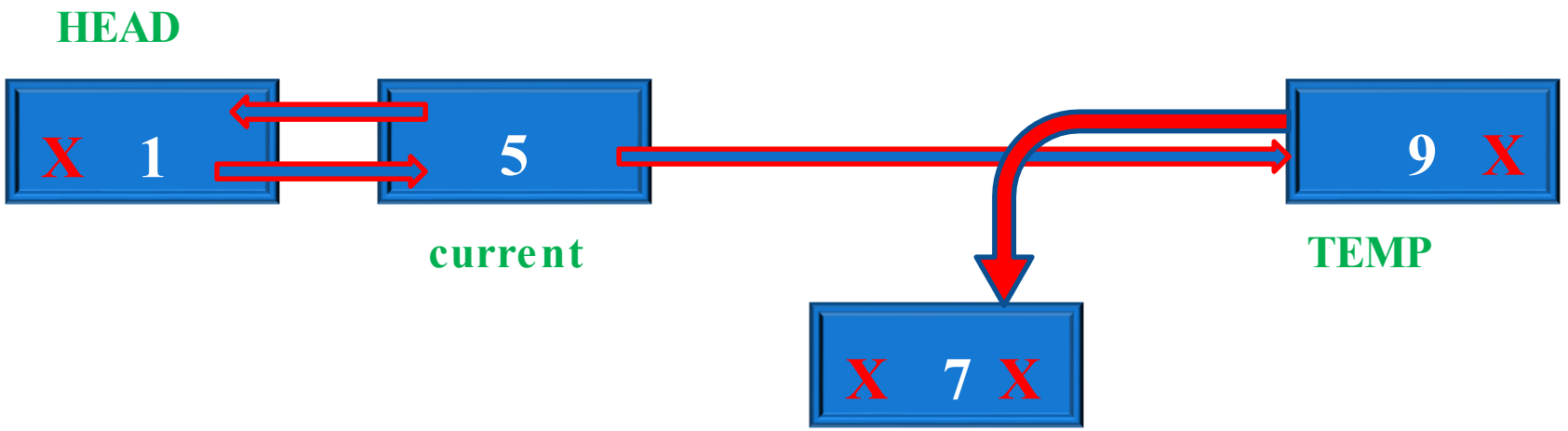
FLAG=1



DLL – insert after any position, say after 5

```
If(flag=1)
    If(current.rightaddr !=NULL)
        Temp=current.rightaddr
        Temp.leftaddr=newnode
        Newnode.rightaddr=temp
        Current.rightaddr=newnode
        Newnode.leftaddr=current
```

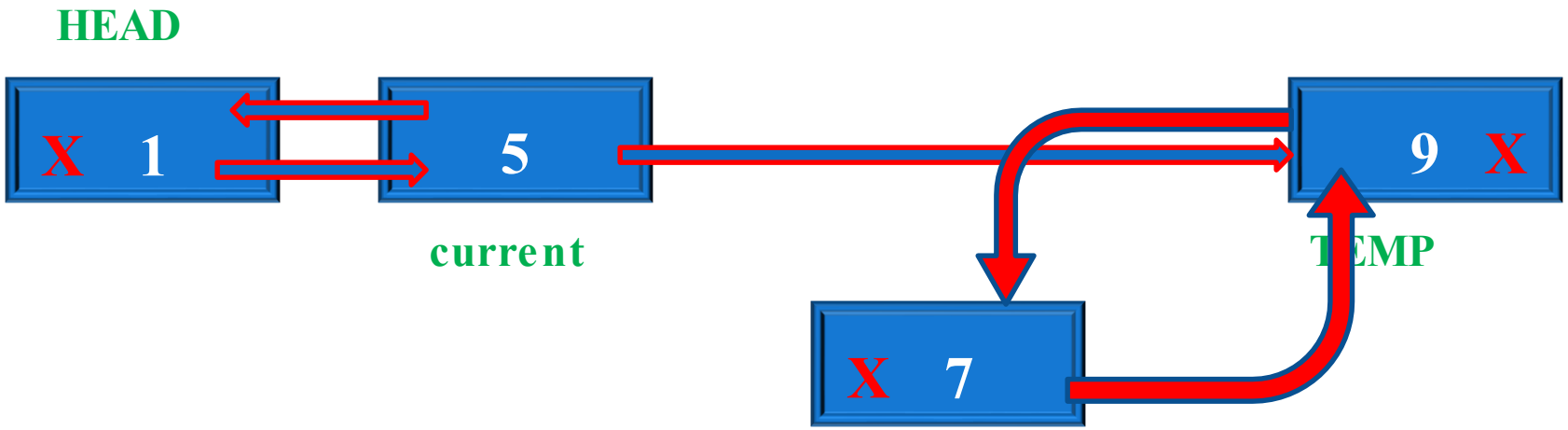
FLAG=1



DLL – insert after any position, say after 5

```
If(flag=1)
    If(current.rightaddr !=NULL)
        Temp=current.rightaddr
        Temp.leftaddr=newnode
        Newnode.rightaddr=temp
        Current.rightaddr=newnode
        Newnode.leftaddr=current
```

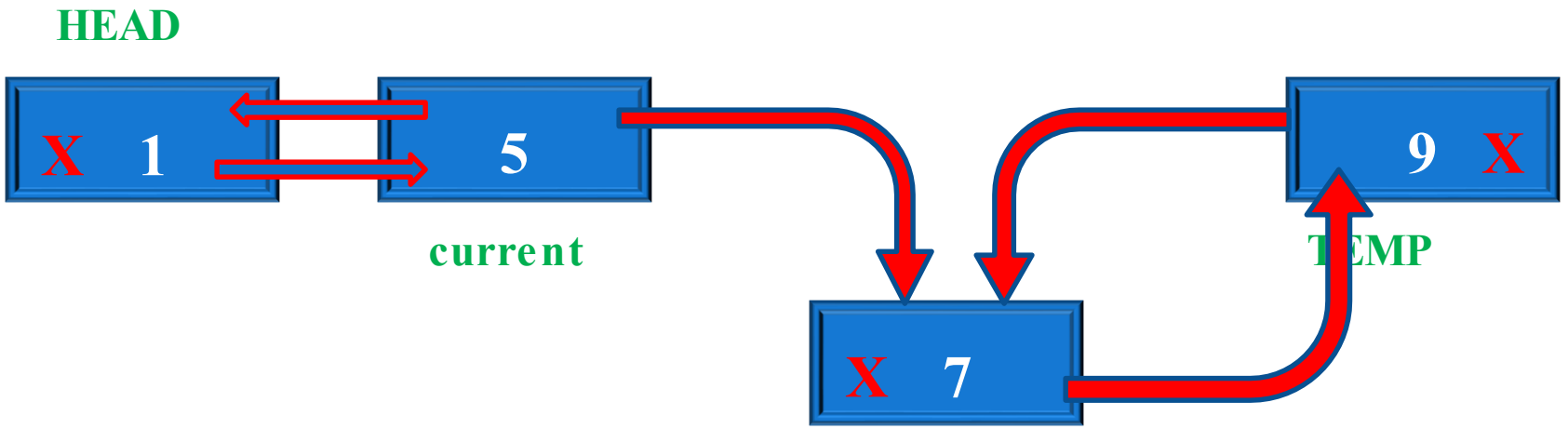
FLAG=1



DLL – insert after any position, say after 5

```
If(flag=1)
    If(current.rightaddr !=NULL)
        Temp=current.rightaddr
        Temp.leftaddr=newnode
        Newnode.rightaddr=temp
        Current.rightaddr=newnode
        Newnode.leftaddr=current
```

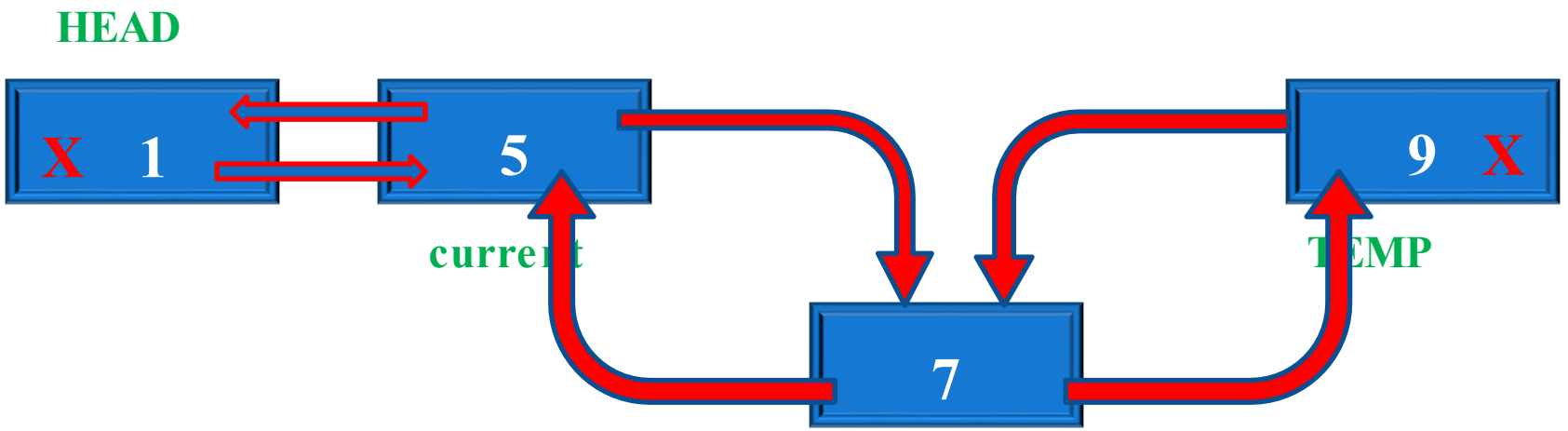
FLAG=1



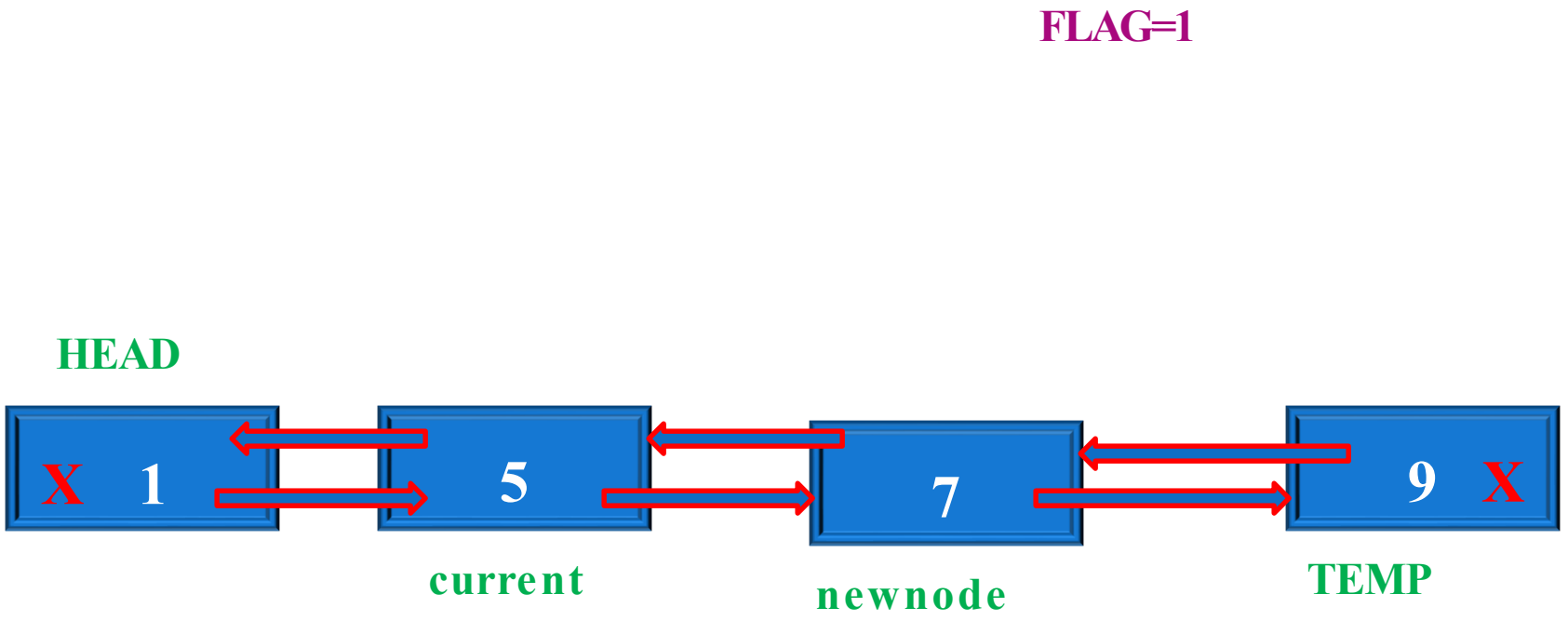
DLL – insert after any position, say after 5

```
If(flag=1)
    If(current.rightaddr !=NULL)
        Temp=current.rightaddr
         Temp.leftaddr=newnode
         Newnode.rightaddr=temp
         Current.rightaddr=newnode
         Newnode.leftaddr=current
```

FLAG=1



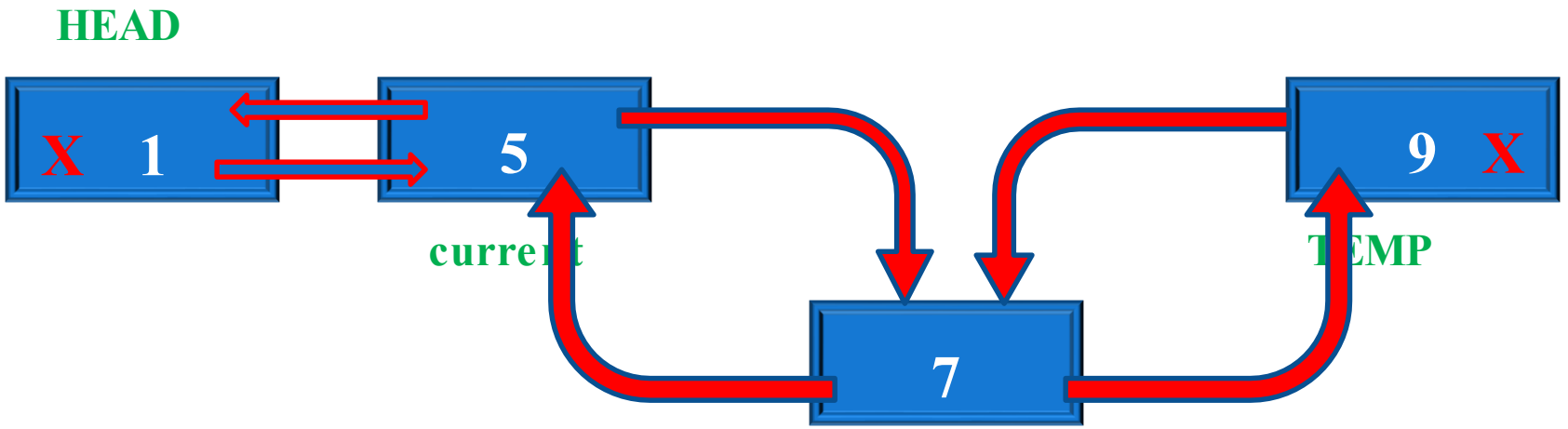
DLL – insert after any position, say after 5



DLL – insert after any position, say after 5

What will happen if 5 was the last node ??

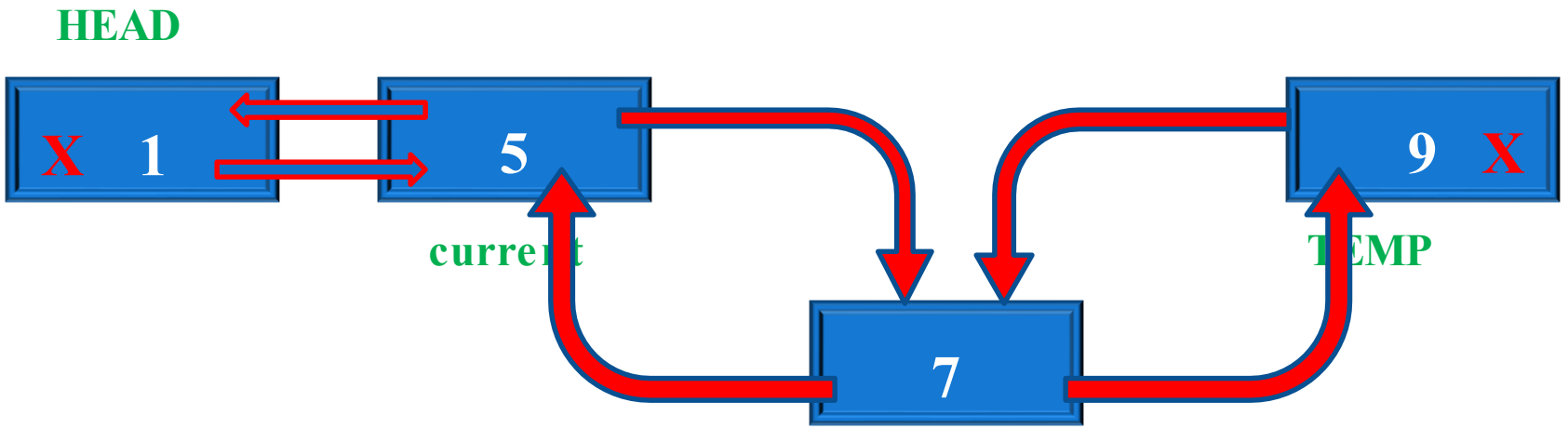
FLAG=1



DLL – insert after any position, say after 5

```
If(flag=1)
    If(current.rightaddr ==NULL)
        Temp=current.rightaddr
        Temp.leftaddr=newnode
        Newnode.rightaddr=temp
        Current.rightaddr=newnode
        Newnode.leftaddr=current
```

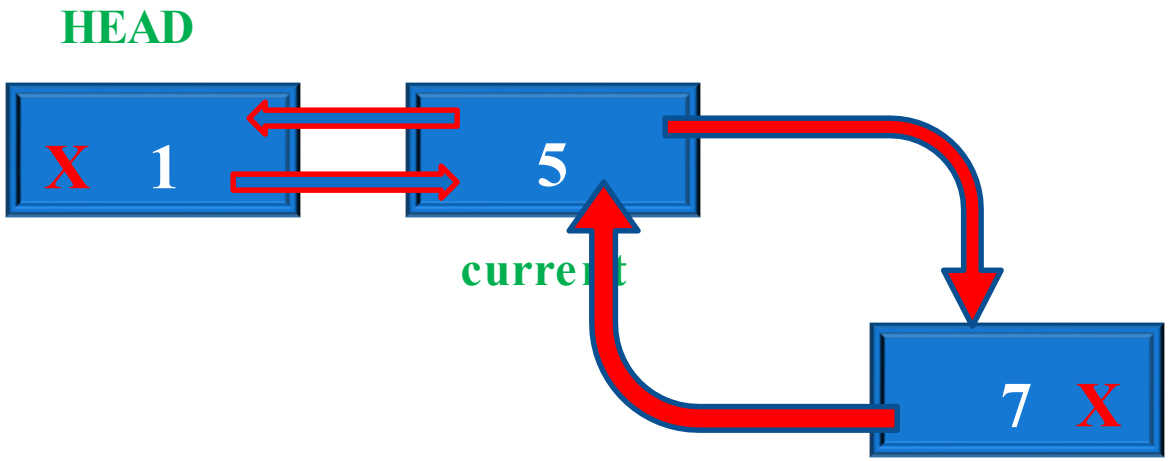
FLAG=1



DLL – insert after any position, say after 5

```
If(flag=1)
    If(current.rightaddr ==NULL)
        Temp=current.rightaddr
        Temp.leftaddr=newnode
        Newnode.rightaddr=temp
        Current.rightaddr=newnode
        Newnode.leftaddr=current
```

FLAG=1



Insert after any position

Steps:

1. Flag=0
2. Newnode=malloc(NODE)
3. Newnode.data=item
4. Newnode.leftaddr=NULL
5. Newnode.rightaddr=NULL
6. If(head==NULL)
    1. Print("List is Empty..")
7. Else
    1. Current=head
    2. While(current!=NULL)
        1. If(current.data=key)
            1. Flag=1
            2. ExitWhile
        2. EndIf
        3. Current=current.rightaddr
    3. EndWhile
    4. If(flag=1)
        1. If(current.rightaddr!=NULL)
            1. Temp=current.rightaddr
            2. Temp.leftaddr=newnode
            3. Newnode.rightaddr=temp
        2. End if
        3. Current.rightaddr=newnode
        4. Newnode.leftaddr=current
    5. Else
        1. Print("Key not Found..");
    6. EndIf
8. EndIf
9. Stop.

Deletion

DLL – delete from FRONT

DLL – delete from FRONT


```
If(head.rightaddr !=NULL)
    Current=head
    Item=current.data
    Head=head.rightaddr
    Head.leftaddr=NULL
    FREE(current)
```

DLL – delete from FRONT

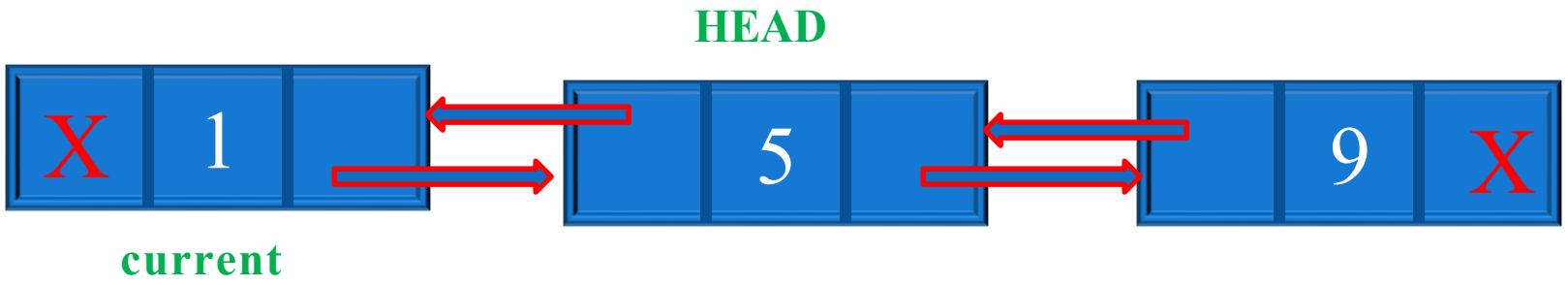

```
If(head.rightaddr !=NULL)
    Current=head
    Item=current.data
    Head=head.rightaddr
    Head.leftaddr=NULL
    FREE(current)
```

ITEM = 1

DLL – delete from FRONT


```
If(head.rightaddr !=NULL)
    Current=head
    Item=current.data
    Head=head.rightaddr
    Head.leftaddr=NULL
    FREE(current)
```

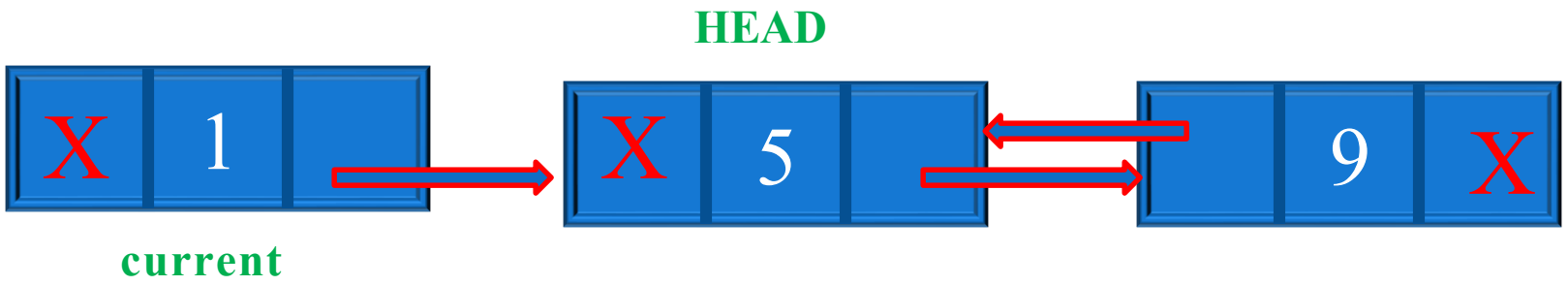
ITEM = 1



DLL – delete from FRONT


```
If(head.rightaddr !=NULL)
    Current=head
    Item=current.data
    Head=head.rightaddr
    Head.leftaddr=NULL
    FREE(current)
```

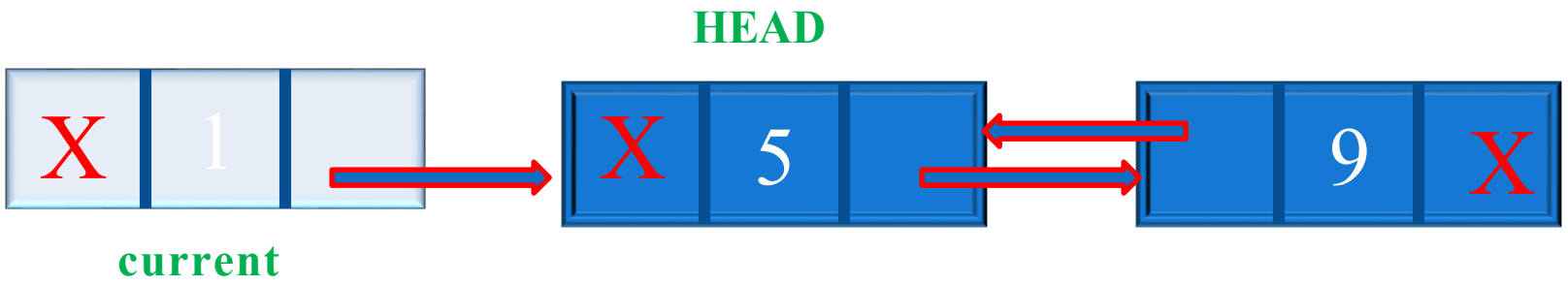
ITEM = 1



DLL – delete from FRONT


```
If(head.rightaddr !=NULL)
    Current=head
    Item=current.data
    Head=head.rightaddr
    Head.leftaddr=NULL
    FREE(current)
```

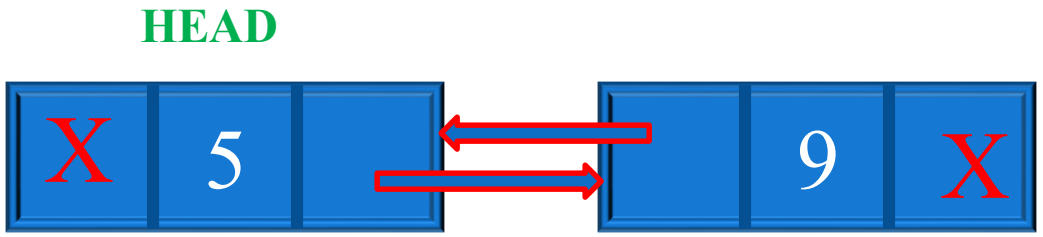
ITEM = 1



DLL – delete from FRONT


```
If(head.rightaddr !=NULL)
    Current=head
    Item=current.data
    Head=head.rightaddr
    Head.leftaddr=NULL
    FREE(current)
```

ITEM = 1



DLL – delete from FRONT

What if there is only one node ??

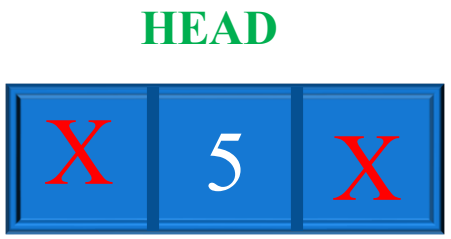
ITEM = 1

HEAD

DLL – delete from FRONT


```
If(head.rightaddr=NULL)
    Item=head.data
    FREE(head)
    Head=NULL
```

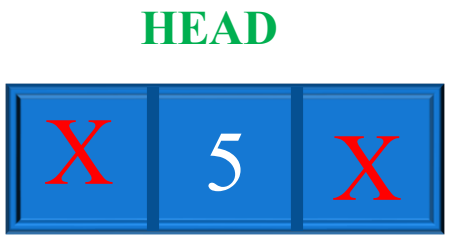
ITEM = 1



DLL – delete from FRONT


```
If(head.rightaddr=NULL)
    Item=head.data
    FREE(head)
    Head=NULL
```

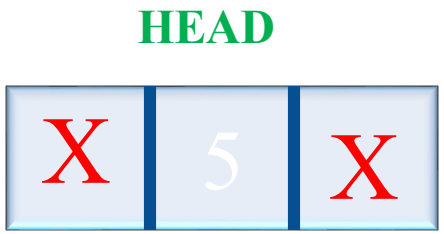
ITEM = 5



DLL – delete from FRONT


```
If(head.rightaddr=NULL)
    Item=head.data
    FREE(head)
    Head=NULL
```

ITEM = 5



DLL – delete from FRONT


```
If(head.rightaddr=NULL)
    Item=head.data
    FREE(head)
    Head=NULL
```

ITEM = 5

HEAD =NULL

Deletion from the front

Algorithm : DL_DELETION_FRONT()
Input: Double linked list with data
Output: List with front node removed
Data Structure: Linked List

Steps:

1. If (head=NULL) then
    1. Print("List is Empty…");
2. Else If(head.rightaddr=NULL)
    1. Item=head.data
    2. FREE(head)
    3. Head=NULL
3. Else
    1. Current=head
    2. Item=current.data
    3. Head=head.rightaddr
    4. Head.leftaddr=NULL
    5. FREE(current)
4. EndIf
5. Stop

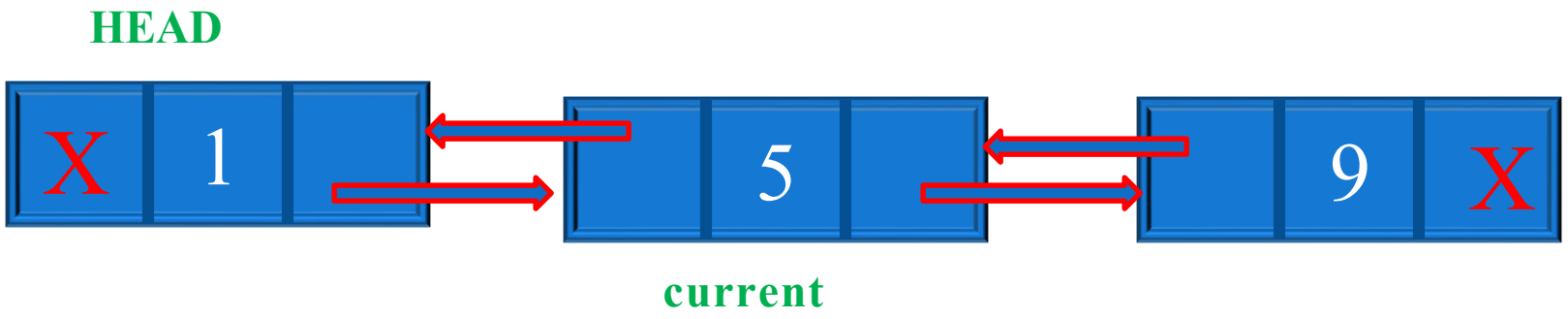
DLL – delete from END

DLL – delete from END

```
Current=head
While(current.rightaddr!=NULL)
    Current=current.rightaddr
```

DLL – delete from END

```
Current=head
While(current.rightaddr!=NULL)
    Current=current.rightaddr
```



DLL – delete from END

```
Current=head
While(current.rightaddr!=NULL)
    Current=current.rightaddr
```

DLL – delete from END

Item=current.data
Prev=current.leftaddr
Prev.rightaddr=NULL;
FREE(current)

ITEM = 9

DLL – delete from END

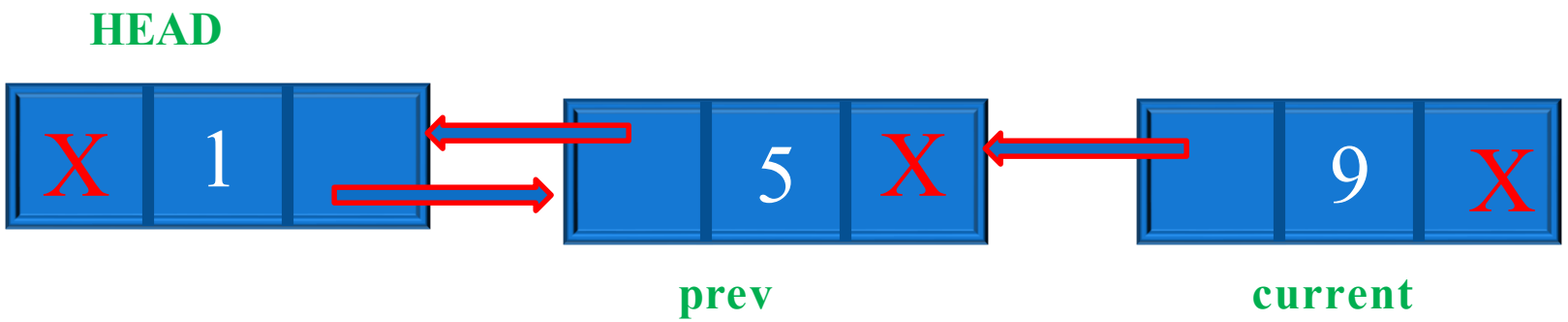
Item=current.data
Prev=current.leftaddr
Prev.rightaddr=NULL;
FREE(current)

ITEM = 9

DLL – delete from END

Item=current.data
Prev=current.leftaddr
Prev.rightaddr=NULL
FREE(current)

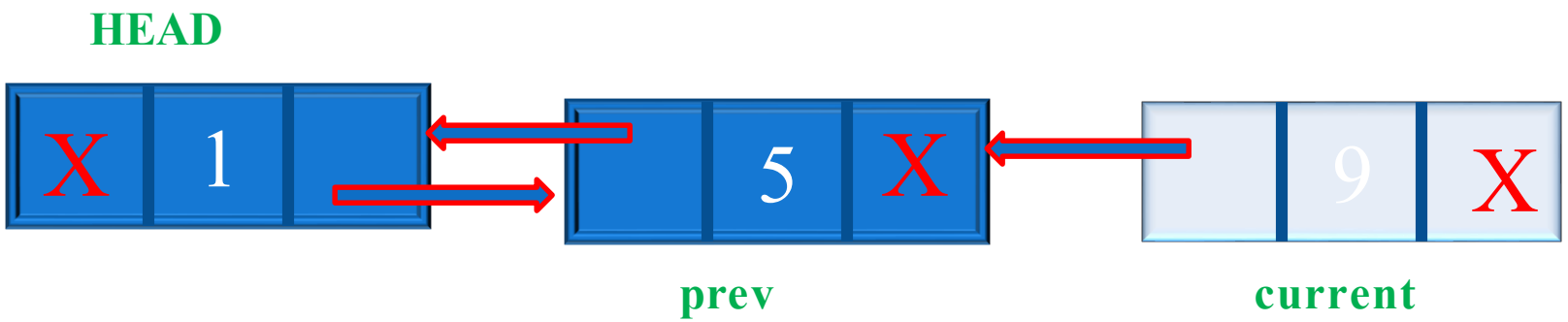
ITEM = 9



DLL – delete from END

Item=current.data
Prev=current.leftaddr
Prev.rightaddr=NULL
FREE(current)

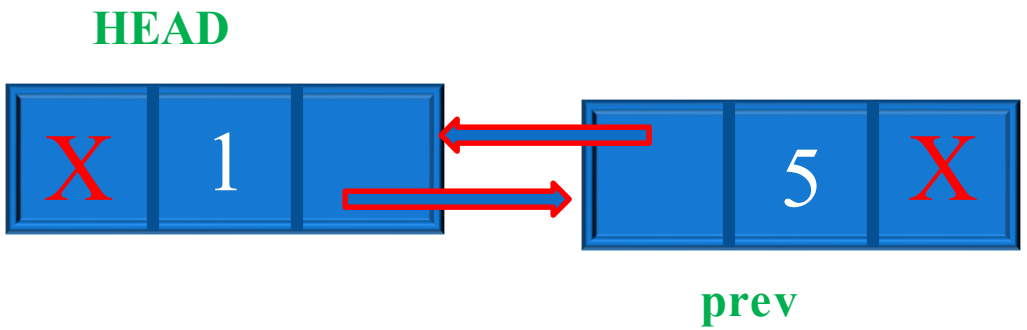
ITEM = 9



DLL – delete from END

Item=current.data
Prev=current.leftaddr
Prev.rightaddr=NULL
FREE(current)

ITEM = 9



Deletion from the end

Algorithm DL_DELETION_END()

Input: A double linked list with data

Output: A list with removed last node

Data Structure: Linked List

Steps:

1. If(head=NULL) then
    1. Print("List is Empty")
2. Else ~~if(head.rightaddr=NULL)~~
    1. Item=head.data
    2. FREE(head)
    3. Head=NULL
3. Else
    1. Current=head
    2. While(current.rightaddr!=NULL)
        1. Current=current.rightaddr
    3. Item=current.data
    4. Prev=current.leftaddr
    5. Prev.rightaddr=NULL;
    6. FREE(current)
1. EndIf
2. Stop

DLL – delete from any position

DLL – delete from any position, delete 5

```
Current=head
While(current!=NULL and current.data != key)
      Current=current.rightaddr
EndWhile
```

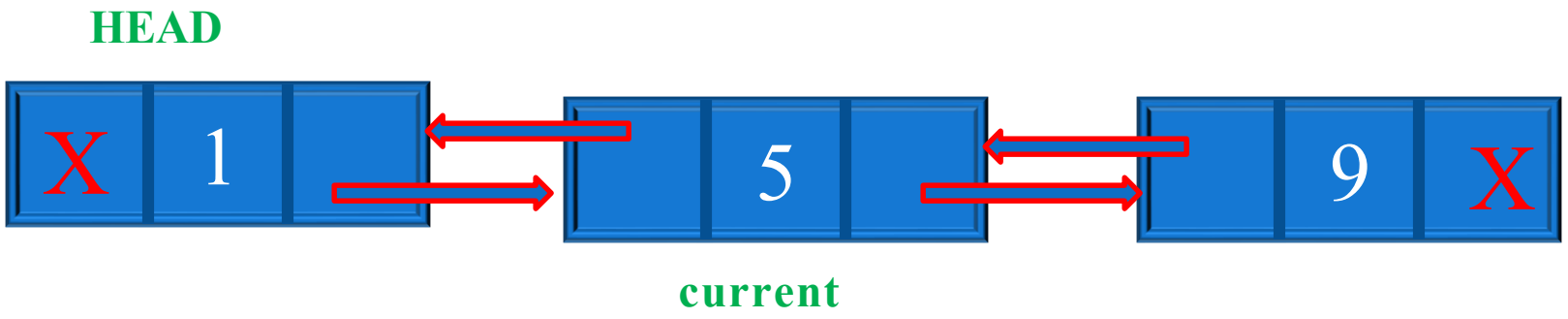
DLL – delete from any position, delete 5

Current=head

While(current!=NULL and current.data != key)

    Current=current.rightaddr

EndWhile

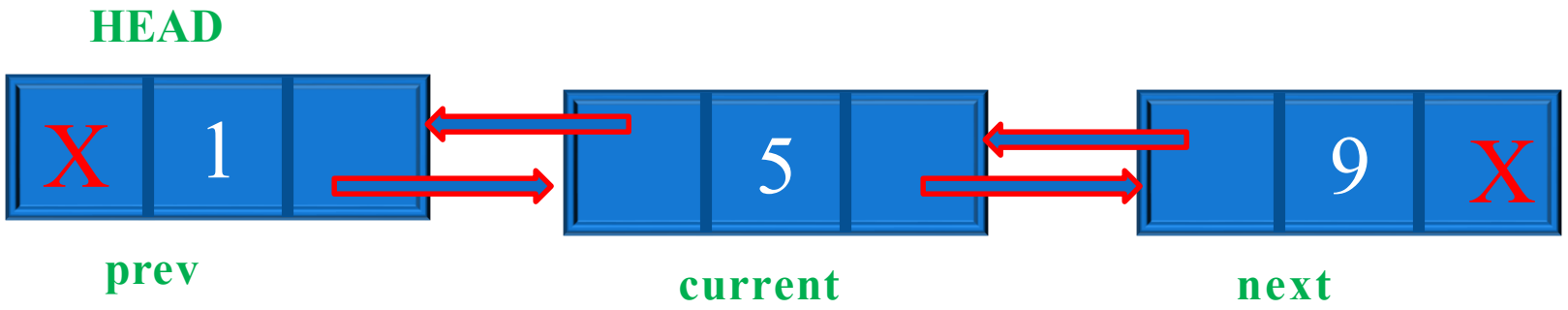


DLL – delete from any position, delete 5

Prev=current.leftaddr

next= current.rightaddr

```
If(current.rightaddr !=NULL)
      Next.leftaddr=prev
      Endif
Prev.rightaddr=next
FREE(current)
```



DLL – delete from any position, delete 5

Prev=current.leftaddr

next= current.rightaddr

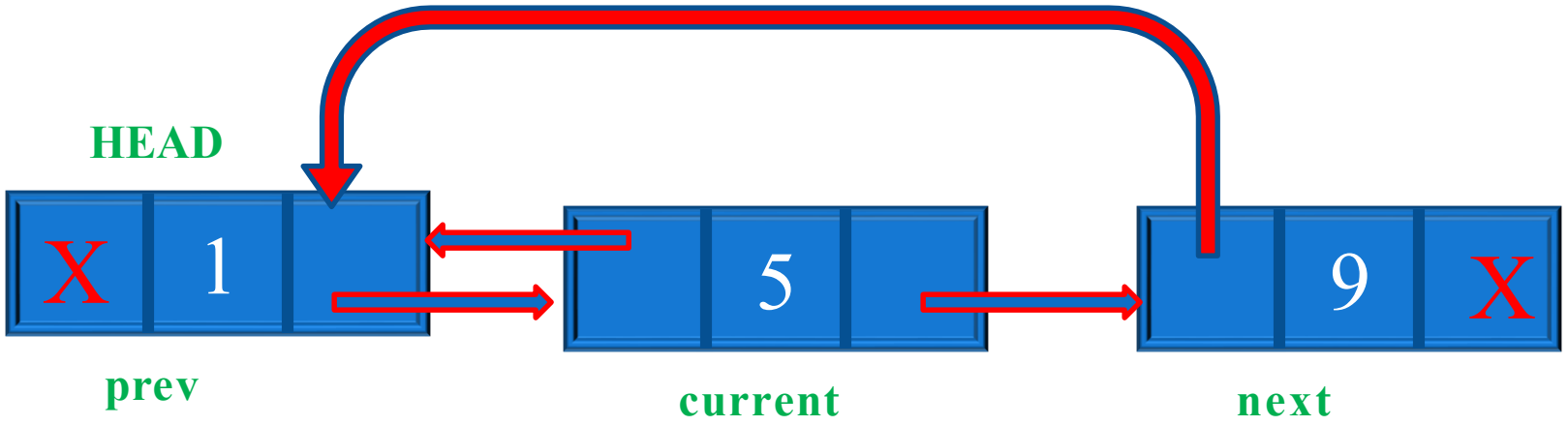
If(current.rightaddr !=NULL)

Next.leftaddr=prev

Endif

Prev.rightaddr=next

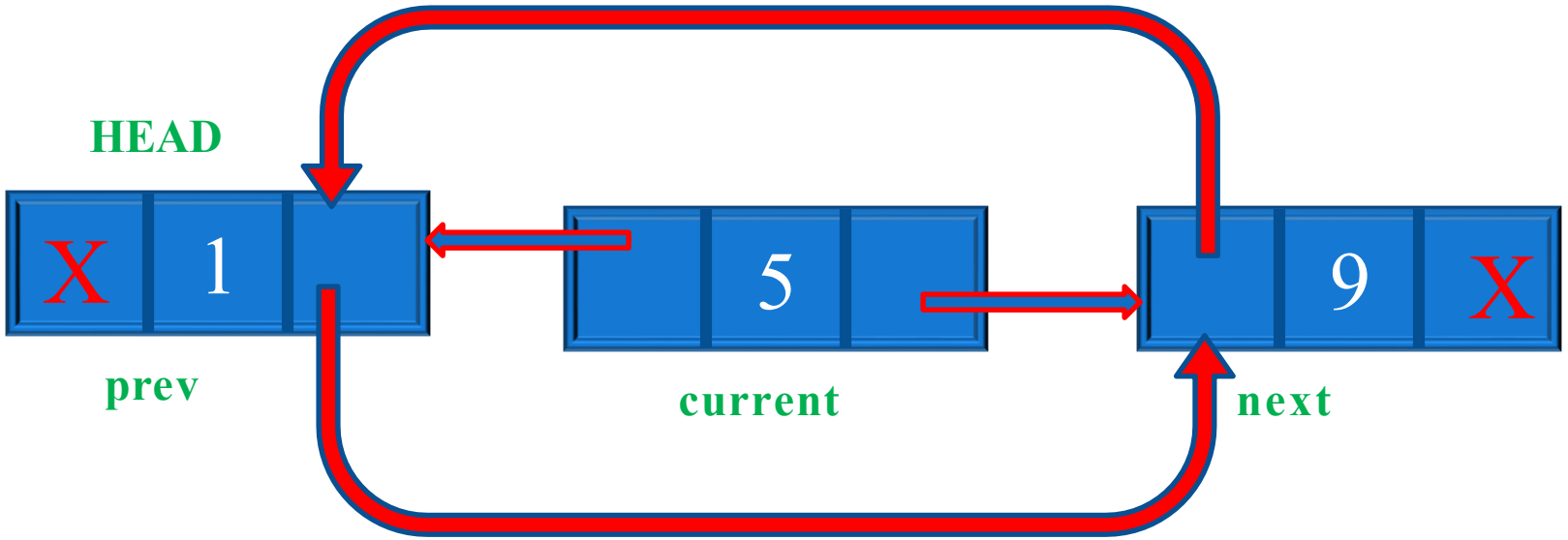
FREE(current)



DLL – delete from any position, delete 5

```
Prev=current.leftaddr
next= current.rightaddr
If(current.rightaddr !=NULL)
      Next.leftaddr=prev
      Endif
Prev.rightaddr=next
FREE(current)
```

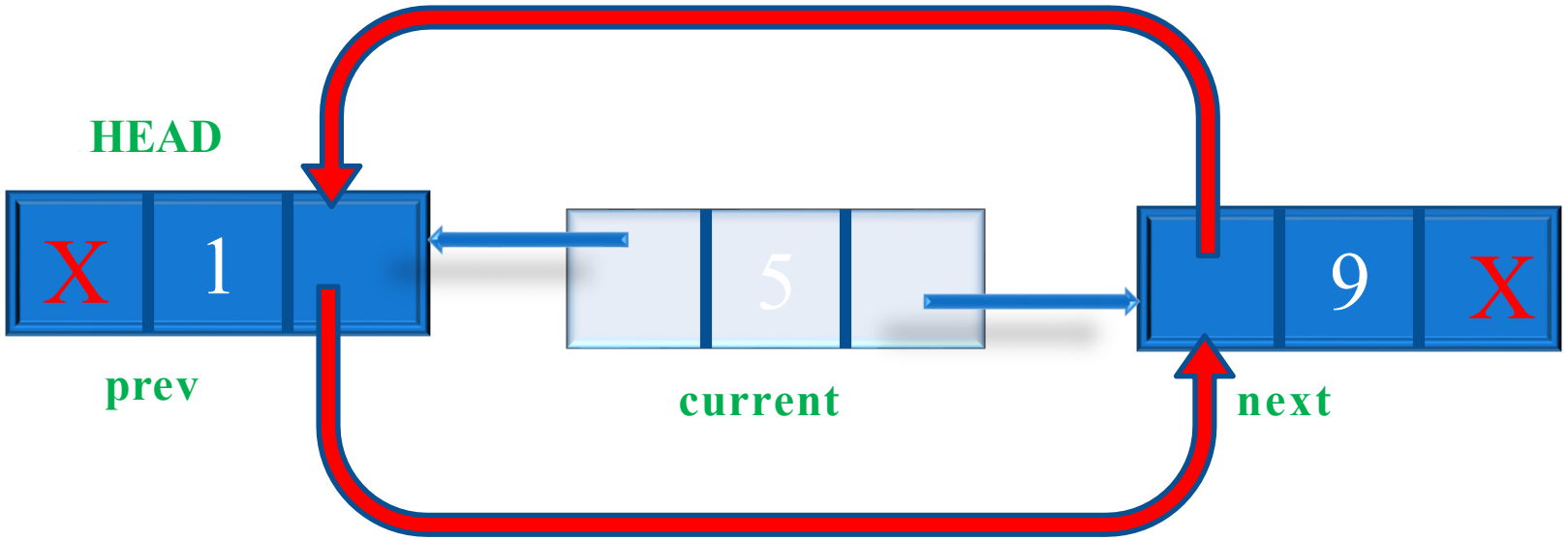
FLAG =1



DLL – delete from any position, delete 5

```
Prev=current.leftaddr
next= current.rightaddr
If(current.rightaddr !=NULL)
      Next.leftaddr=prev
      Endif
Prev.rightaddr=next
FREE(current)
```

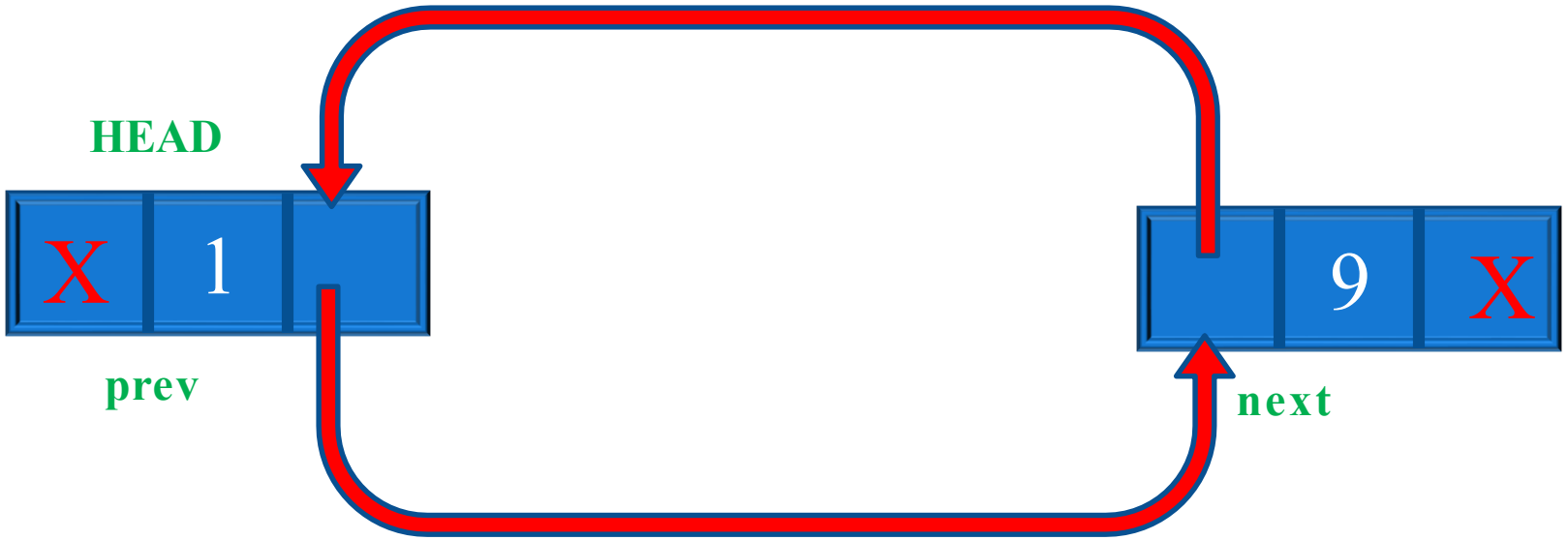
FLAG =1



DLL – delete from any position, delete 5

```
Prev=current.leftaddr
next= current.rightaddr
If(current.rightaddr !=NULL)
      Next.leftaddr=prev
      Endif
Prev.rightaddr=next
FREE(current)
```

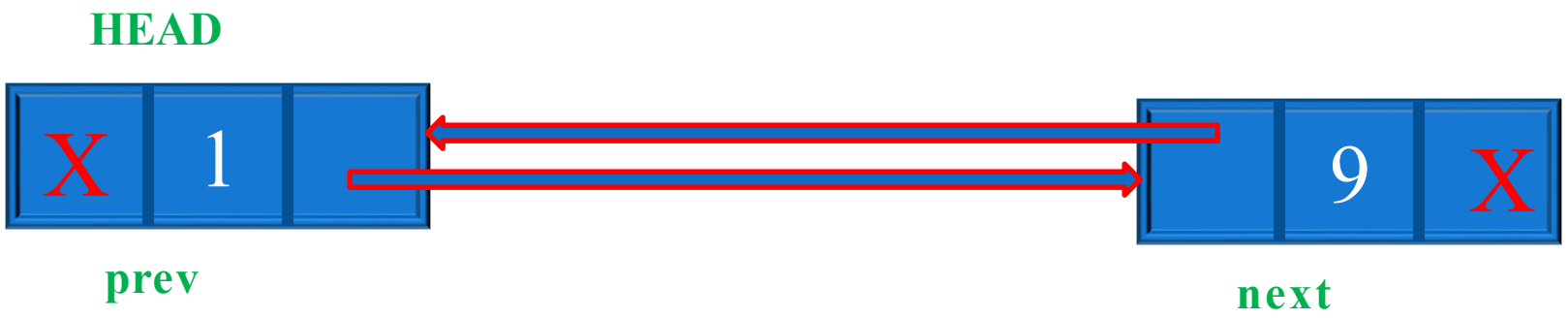
FLAG =1



DLL – delete from any position, delete 5

```
Prev=current.leftaddr
next= current.rightaddr
If(current.rightaddr !=NULL)
      Next.leftaddr=prev
      Endif
Prev.rightaddr=next
FREE(current)
```

FLAG =1



3.Deletion from any position

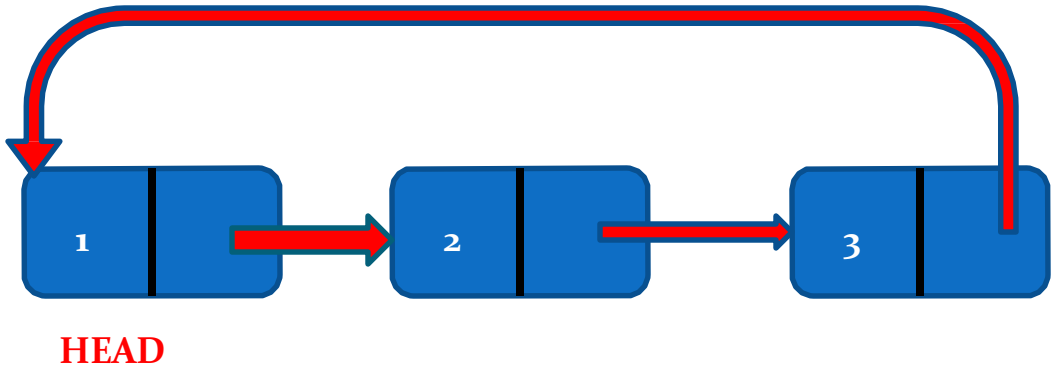
Algorithm DL_DELETE_ANY(item)
Input: A double linked list with pointer header
Output: A double linked list without a node having data content item
Data Structure: linked list

Steps:

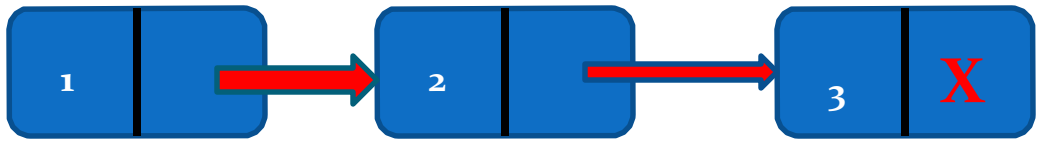
1. If(head=NULL)
    1. Print("List Empty..No Deletion)
2. Else
    1. If(head.data=item AND head.rightaddr=NULL)
        1. FREE(head)
        2. Head=NULL
    2. Else If(head.data=item AND head.rightaddr!=NULL)
        1. Current=head
        2. head=head.rightaddr
        3. Head.leftaddr=NULL
        4. FREE(current)
    3. Else
        1. Current=head
        2. While(current!=NULL and current.data !=key)
            1. Current=current.rightaddr
        3. EndWhile
        4. If(current!=NULL)
            1. Prev=current.leftaddr
            2. next= current.rightaddr
            3. If(current.rightaddr !=NULL)
                1. Next.leftaddr=prev
                2. Endif
            4. Prev.rightaddr=next
            5. FREE(current)
        5. Else print("Item not Found")
        6. EndIf
    4. EndIf
3. EndIf
4. Stop

Circular Linked List

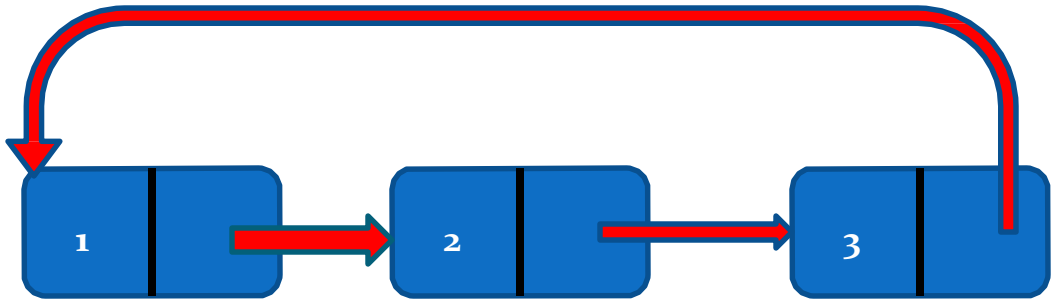
- single linked list - link field of the last node is null.
- circular linked list - link field of the last node holds the address of the first node.



Singly linked list

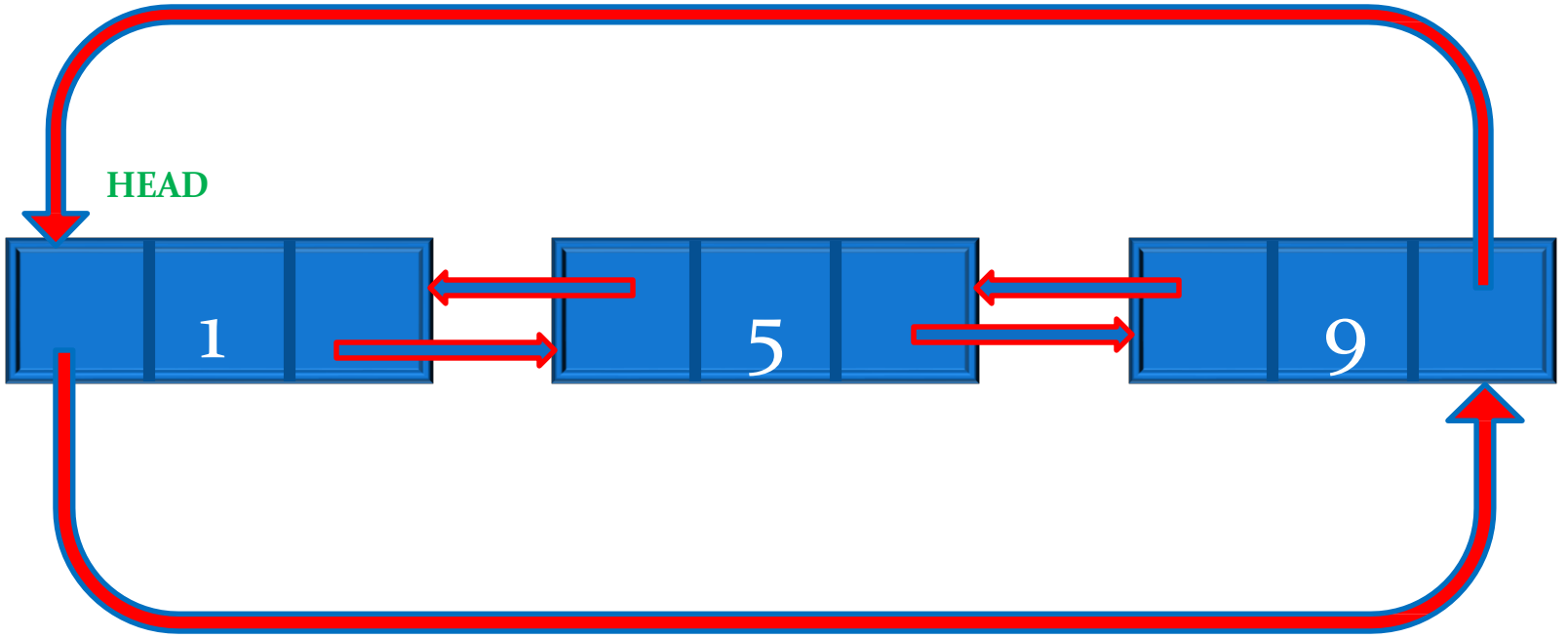


Singly Circular linked list



Doubly linked list

Doubly Circular linked list



insertion

Insertion in a circular linked list:

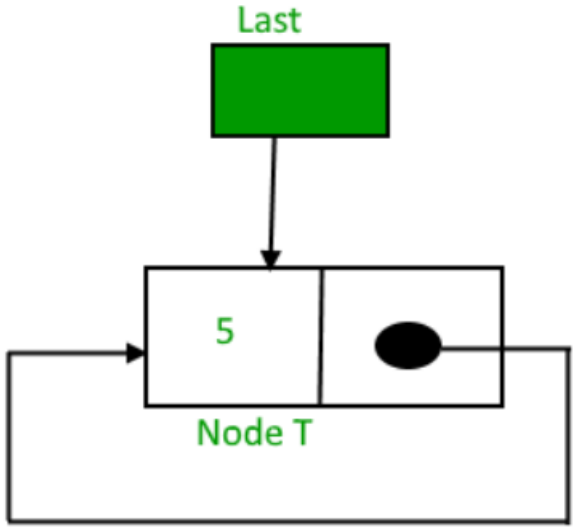
A node can be added in three ways:

- Insertion in an empty list
- Insertion at the beginning of the list
- Insertion at the end of the list
- Insertion in between the nodes

Insertion in an empty List:

To implement a circular singly linked list, we take an external pointer that points to the last node of the list. If we have a pointer last pointing to the last node, then last -> next will point to the first node.

After inserting node T,



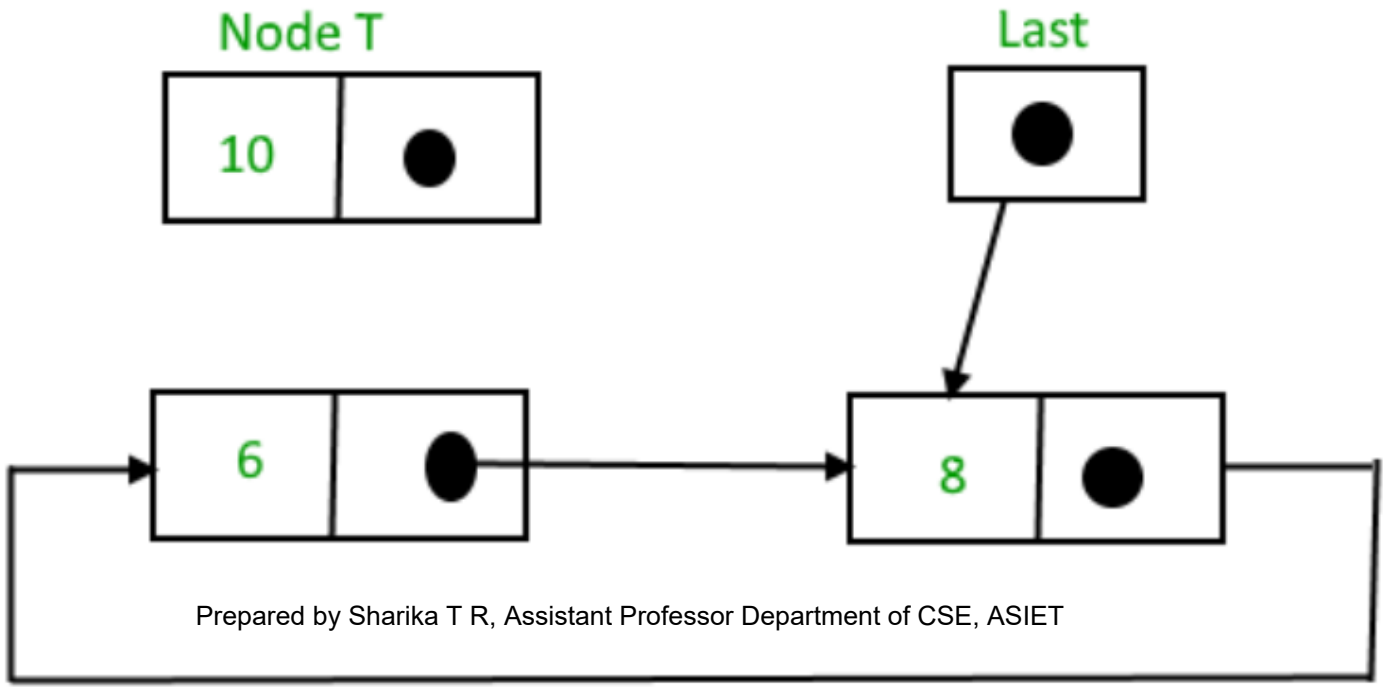
After insertion, T is the last node, so the pointer *last* points to node T. And Node T is the first and the last node, so T points to itself.

```
Newnode=malloc(node)
Newnode.data=item
Newnode.link=NULL
if (last != NULL)
        return last;
last = Newnode;
Newnode ->next = last;
```

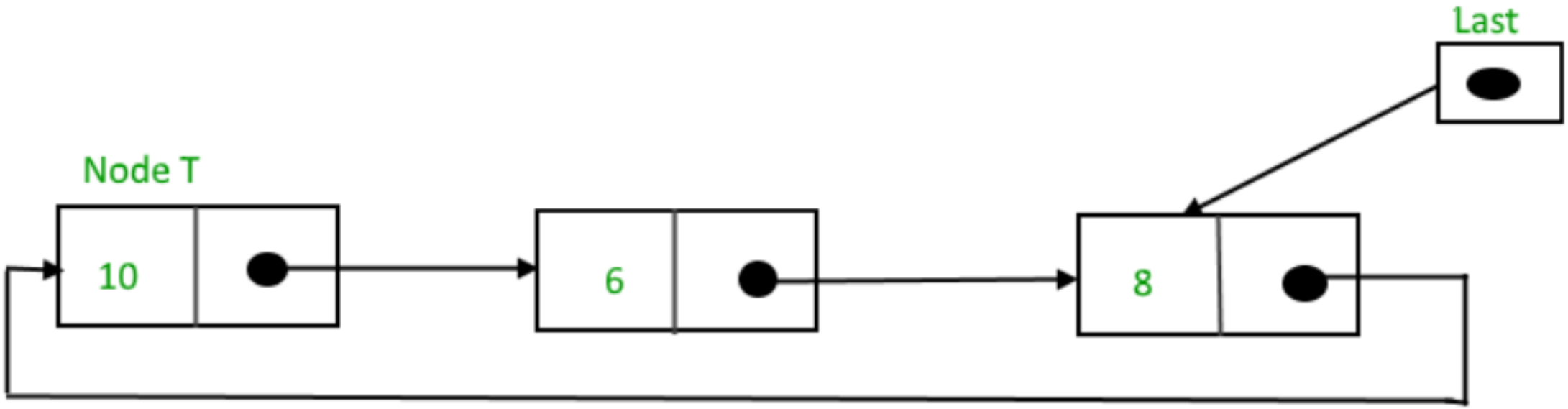
Insertion at the beginning of the list

To insert a node at the beginning of the list, follow these steps:

- Create a node, say T
- Make T -> next = last -> next
- last -> next = T



After insertion,

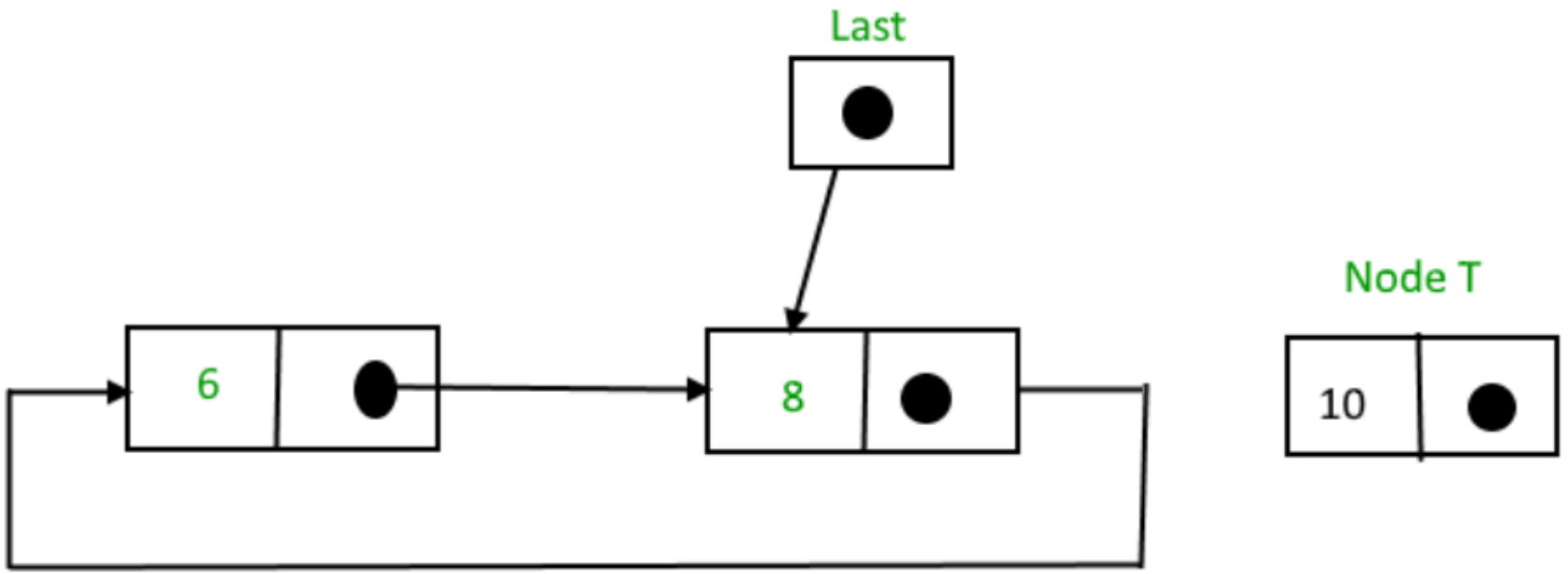


Newnode=malloc(node)
Newnode.data=item
Newnode.link=NULL

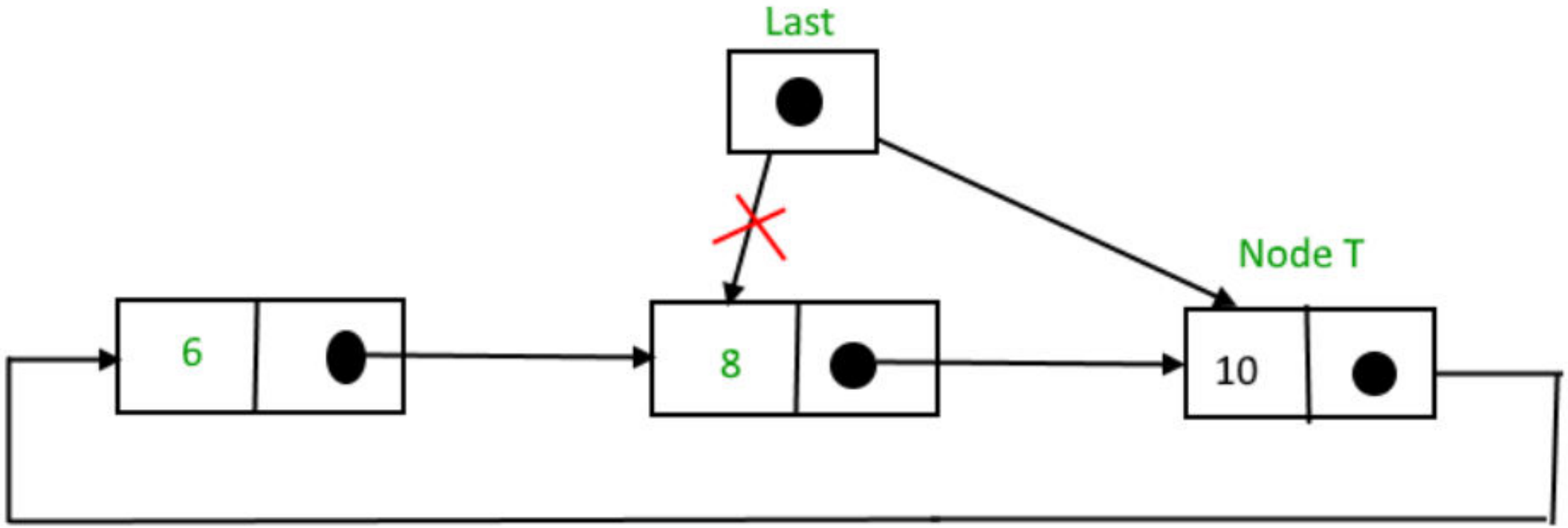
```
Newnode->next = last->next;
last->next = Newnode;
```

Insertion at the end of the list

- Create a node, say T
- Make T -> next = last -> next
- last -> next = T
- last = T



After insertion



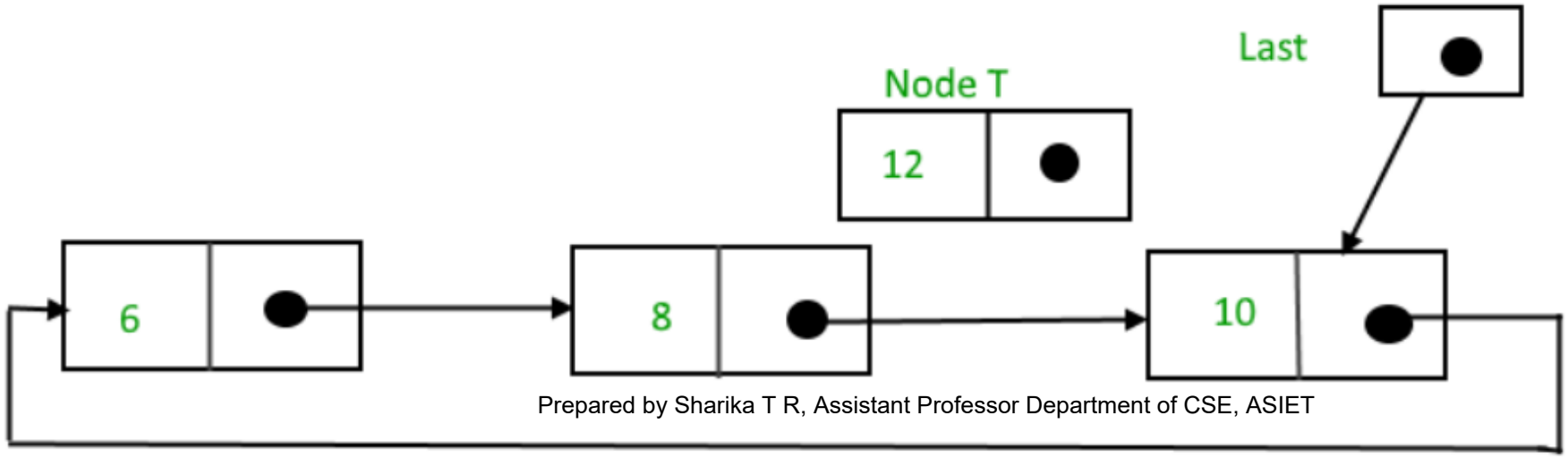
Newnode=malloc(node)
Newnode.data=item
Newnode.link=NULL

```
Newnode->next = last->next;
last->next = Newnode;
last=Newnode;
```

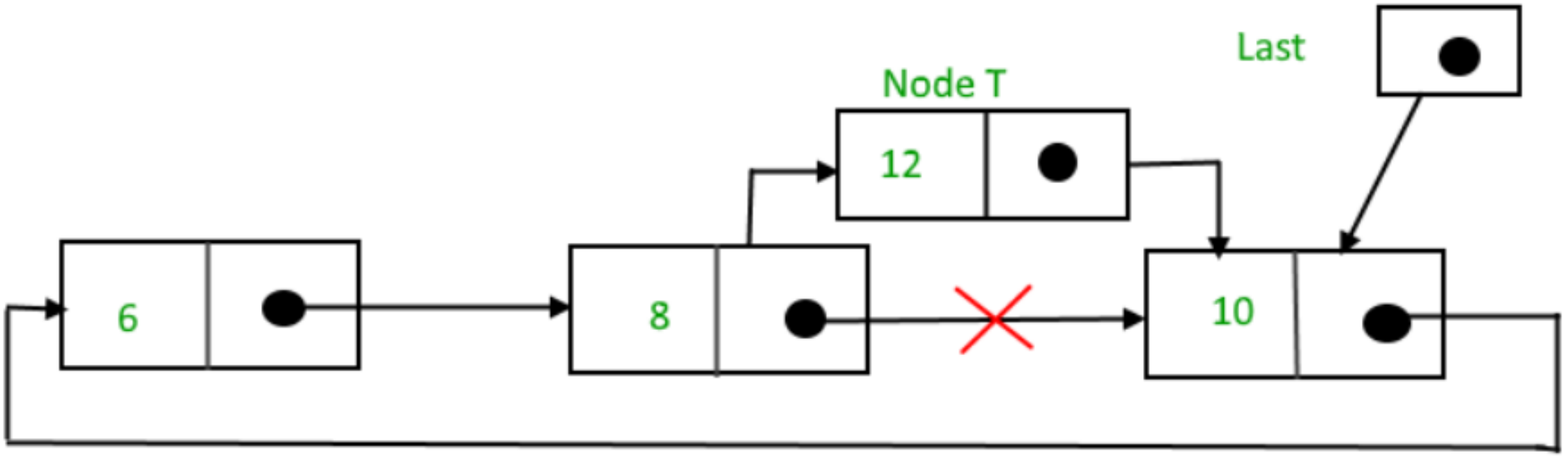
Insertion in between the nodes

- Create a node, say T.
- Search for the node after which T needs to be inserted, say that node is P.
- Make T -> next = P -> next;
- P -> next = T.

Suppose 12 needs to be inserted after the node that has the value 8,



After searching and insertion,



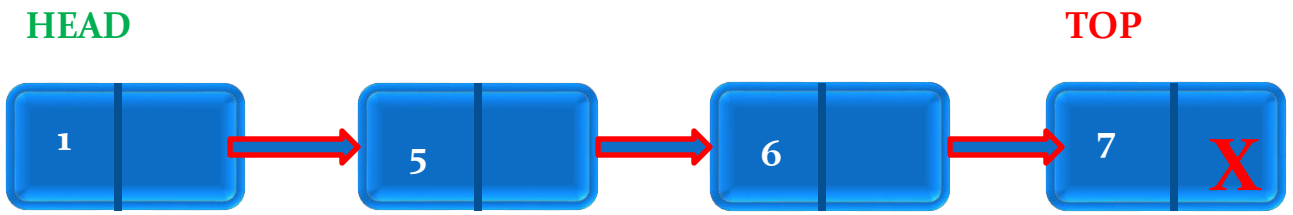
```
flag=0
If last is NULL Then
        Return NULL
End If
 Newnode = malloc(Node)
 Newnode.data = data
 Newnode.link = NULL
```

```
p = last.link
 If p.data is equal to item Then
        flag = 1
        Newnode.link = p.link
        p.link = Newnode
        last = Newnode
Else
      While p is not equal to last.link
        If p.data==item Then
          flag=1
          Newnode.link = p.link
          p.link = Newnode
           If p is equal to last Then
              last = Newnode
           End If
          Break
End If
```

```
    p = p.link
    If p is equal to last.link Then
            Break
     End If
   End While
EndIf

If flag=0
        Print item " not present in the list.“
End If
```

STACK

- Last in First out
- Insertion and deletion is performed at one end, top only.
- Keep an additional pointer, TOP to point to the last node
- Operations
  - PUSH - Insert at end
  - POP - Delete from end



STACK PUSH USING LINKED LIST

STEPS

1. newnode = malloc(sizeof(struct node))
2. newnode ->data=item
3. newnode ->link=TOP
4. TOP=newnode

STACK POP USING LINKED LIST

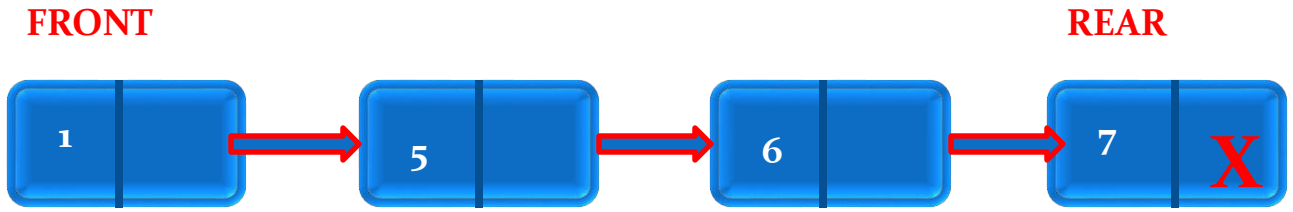
STEPS

1. If TOP=null
   - Print " Stack Empty"
   - Exit
2. Else
   - temp=TOP
   - Item=temp->data
   - TOP=temp->link
   - FREE(temp)

QUEUE

- First in First out
- Insertion at one end, rear and deletion is performed at the other end, front.
- Instead of Head we can use FRONT pointer
- And an additional pointer, REAR can be used to point to the last node.

- Operations
  - ENQUEUE - Insert at rear (end)
  - DEQUEUE - Delete from front



QUEUE ENQUEUE USING LINKED LIST

Steps

1. newnode = malloc(sizeof(struct node));
2. if (newnode == NULL)
   - return ERROR;
3. newnode->data = item;
4. newnode->link = NULL;
5. if (REAR == NULL)
   - FRONT = REAR = newnode;
6. Else
   - REAR->link = newnode
   - REAR = newnode

QUEUE DEQUEUE USING LINKED LIST

Steps

```
1.if (FRONT == NULL)
        return ERROR;
2. temp = FRONT;
3. item = temp->data;
4.FRONT = FRONT->link;
5.if (FRONT == NULL)
        REAR = NULL;
6.free(temp);
7.return item;
```

Arrays vs Linked lists

ARRAYS

1. Size of an array cannot be increased or decreased during execution
2. Insertion and deletion is difficult, as it requires shifting elements
3. It allows random access to elements
4. Memory space = number of elements
5. Memory allocated as contiguous blocks

LINKED LIST

1. LL can grow as long as memory is available. It can also shrink without wasting memory
2. Insertion and deletion is easy as it only requires changing a few pointers
3. It only allows sequential access
4. Extra memory space for a pointer is required with each element of the list.
5. No need to allocate contiguous blocks for the entire LL

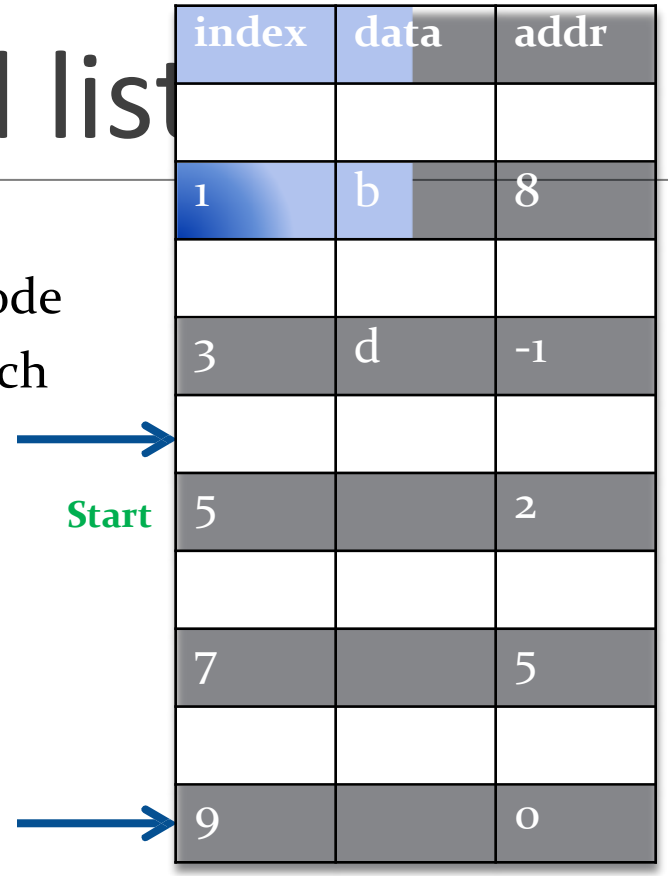
Implementation of linked list

1. **Static** or **sequential** or **array** implementation
2. **Dynamic** or **pointer** or **linked** implementation

Implementation of linked list

Static or sequential or array implementation

- Two 1D arrays are used.
  - One array to store the data part of each node
  - Other array to store the address part of each node
- Eg:



- A[1] = b (data part of second node)
- B[1] = 8 (address part of second node)

Array implementation is simple and efficient in randomly accessing the element of the list

But the size of the array should be declared in advance

Implementation of linked list

Dynamic or pointer or linked implementation

- Here nodes are created as and when required by using the dynamic memory allocation methods
  - In C – malloc, free
  - In C++ - new, delete

Comparison of linked list implementations

Array representation

1. LL has a fixed size
2. Only limited elements can be inserted
3. Deleting elements creates vacant spaces leading to wastage of memory
4. Insertion and deletion is difficult, as it requires shifting elements
5. It allows random access to elements

Pointer representation

1. LL can grow during execution
2. Unlimited elements can be inserted (depends on memory)
3. No memory wastage after deleting an element
4. Insertion and deletion is easy as it only requires changing a few pointers
5. It only allows sequential access

Memory management techniques

Two types of memory management-

1) static memory management- the net amount of memory required for various data for a program is allocated before the starting of the execution of the program. Once memory is allocated, it can neither be extended nor be returned to the memory block for the use of other programs at the same time.

2) dynamic memory management- allows the user to allocate and deallocate memory as per the necessity during the execution of programs. This dynamic memory management scheme is suitable in multiprogramming as well as single environment where generally more than one program reside in the memory and their memory requirement can be known only during their execution. The data structure for implementing such a scheme is linked list.

The heap is the region of the main memory from which portions of memory are dynamically allocated upon request of a program.

The maintenance of free memory blocks, assigning specific memory blocks to the user programs if necessary and cleaning memory from unneeded blocks to return them to the memory pool is performed by a part of the operating system called a **memory manager**.

Simple organization of memory requires a linked list of all memory blocks which is updated after a block is either requested or returned.

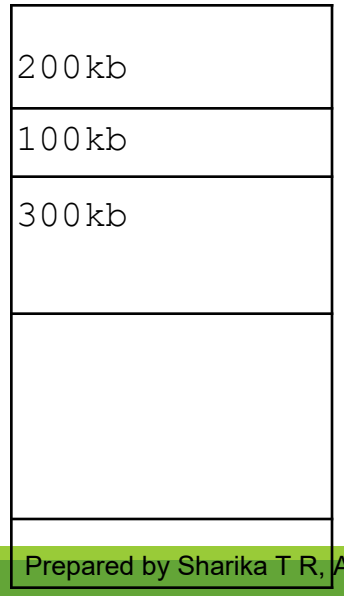
The blocks on such linked lists can be organized in a variety of ways, according to the block sizes or the block addresses.

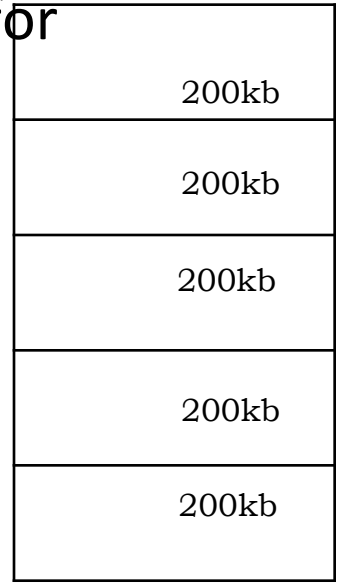
The size of blocks can be either fixed block or variable block.

Fixed block storage is simplest storage maintenance method. Here each block is of the same size.

Variable memory allocation

A program may require storage blocks in a large variety of sizes. In such cases, a memory management system must be able to process requests for variable length blocks.





Memory de-allocation

Garbage collection

A method of detecting and reclaiming free nodes

Nodes no longer in use remain allocated and undetected until all available storage has been allocated.

A subsequent request for allocation cannot be satisfied until nodes that have been allocated but are no longer in use are recovered.

When a request is made for additional nodes and there are none available, a system routine called **garbage collector** is called.

This routine searches through all nodes in the system, identifies those that are no longer accessible from an external pointer, and restores the inaccessible nodes to the available pool.

Compaction

The process of moving all used nodes or blocks to one end of the memory and all available(free) memory to the other end is called compaction.

	INTERNAL FRAGMENTATION	EXTERNAL FRAGMENTATION
Basic	It occurs when fixed sized memory blocks are allocated to the processes.	It occurs when variable size memory space are allocated to the processes dynamically.
Occurrence	When the memory assigned to the process is slightly larger than the memory requested by the process this creates free space in the allocated block causing internal fragmentation.	When the process is removed from the memory, it creates the free space in the memory causing external fragmentation.
Solution	The memory must be partitioned into variable sized blocks and assign the best fit block to the process.	Compaction, paging and segmentation.

Memory allocation schemes

First fit - allocate the first hole that's big enough.

Best fit - allocate smallest hole that's big enough.

Worst fit - allocate largest hole.

First Fit: The simplest algorithm is first fit. The process manager scans along the list of segments until it finds a hole that is big enough. The hole is then broken up into two pieces, one for the process and one for the unused memory, except in the statistically unlikely case of an exact fit. First fit is a fast algorithm because it searches as little as possible.

Next Fit: It works the same way as first fit, except that it keeps track of where it is whenever it finds a suitable hole. The next time it is called to find a hole, it starts searching the list from the place where it left off last time, instead of always at the beginning, as first fit does.

Best Fit: Best fit searches the entire list and takes the smallest hole that is adequate. Rather than breaking up a big hole that might be needed later, best fit tries to find a hole that is close to the actual size needed.

Worst Fit: Always take the largest available hole, so that the hole broken off will be big enough to be useful. Simulation has shown that worst fit is not a very good idea either.

Problem

Given five memory partitions of 100Kb, 500Kb, 200Kb, 300Kb, 600Kb (in order), how would the first-fit, best-fit, and worst-fit algorithms place processes of 212 Kb, 417 Kb, 112 Kb, and 426 Kb (in order)?
Which algorithm makes the most efficient use of memory?

First-fit:

First fit 212 Kb, 417 Kb, 112 Kb, and 426 Kb

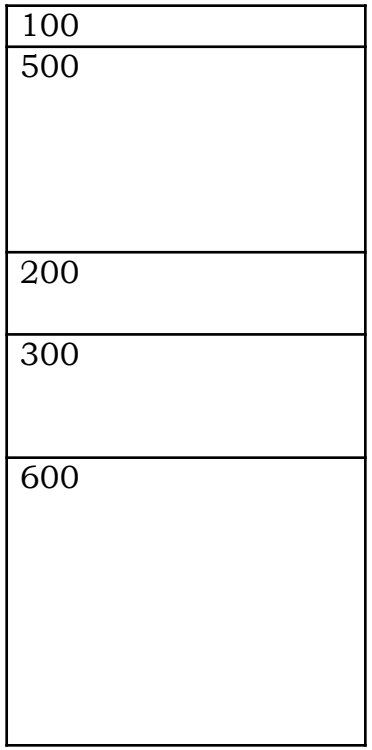
First-fit:

212K is put in 500K partition

417K is put in 600K partition

112K is put in 288K partition (new partition 288K = 500K - 212K)

426K must wait

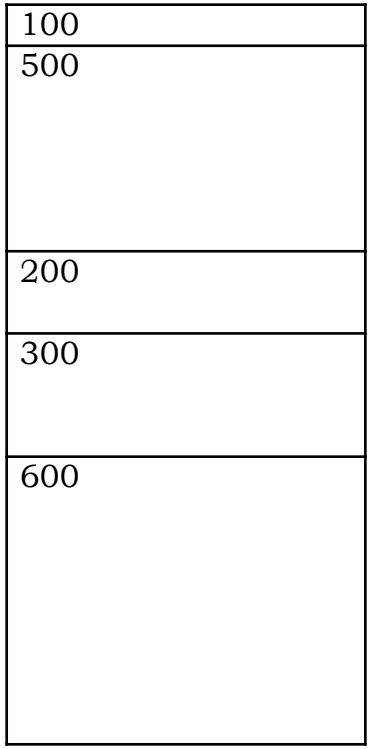


Best fit 212 Kb, 417 Kb, 112 Kb, and 426 Kb

Best-fit:

Best-fit:

212K is put in 300K partition 417K is put in 500K partition 112K is put in 200K partition 426K is put in 600K partition



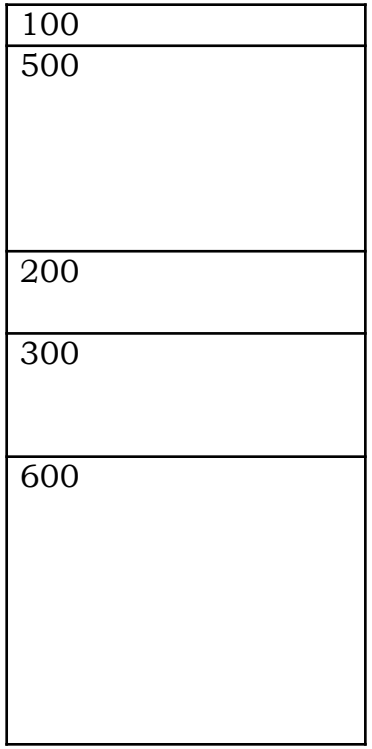
Worst fit 212 Kb, 417 Kb, 112 Kb, and 426 Kb

Worst-fit:

Worst-fit:

212K is put in 600K partition 417K is put in 500K partition 112K is put in 300K partition 426K must wait

In this example, best-fit turns out to be the best.



Problem

Free memory blocks of size 60K,25K,12K,20K,35K,45K and 40K are available in this order. Show the memory allocation for a sequence of job requests of size 22K,10K,42K, and 31K in first fit, best fit and worst fit allocation strategies.

First Fit

60K,25K,12K,20K,35K,45K,40K

60K,25K,12K,20K,35K,45K,40K

Processes□ 22K,10K,42K, and 31K

22K,

38K, 25K,12K,20K,35K,45,40K

10K,

28K, 25K,12K,20K,35K,45K,40K

……..,42K,……

……..,31K,42K,……

Next Fit

60K,25K,12K,20K,35K,45K,40K

60K,25K,12K,20K,35K,45K,40K

22K,

38K, 25K,12K,20K,35K,45K,40K

……..10K,……..42K,31K

Processes□ 22K,10K,42K, and 31K

Best Fit

60K,25K,12K,20K,35K,45K,40K

60K,25K,12K,20K,35K,45K,40K

……,22K,10K,……..,31K,42K……

Processes□ 22K,10K,42K, and 31K

Worst Fit

60K,25K,12K,20K,35K,45K,40K

60K,25K,12K,20K,35K,45K,40K

22K,……10K,31K

42K will wait

Processes□ 22K,10K,42K, and 31K

MODULE 2 ENDS