Static and Dynamic List
Chapter 4

Presented by: Er. Aruna Chhatkuli

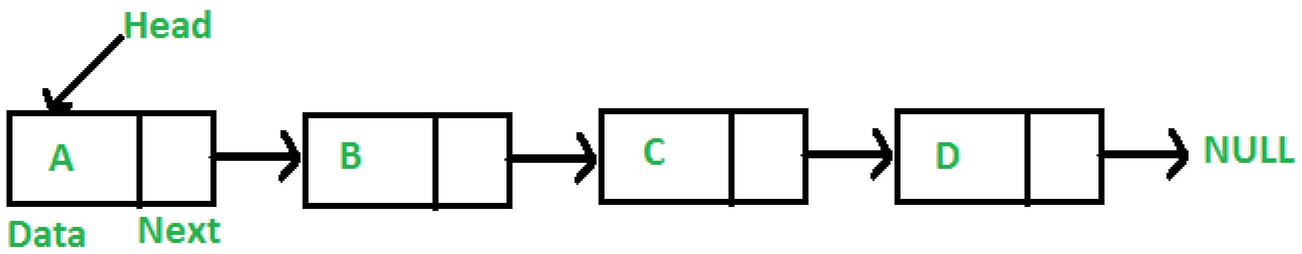
Nepal College of Information Technology,
Balkumari, Lalitpur

Introduction

List

- List is an ordered set consisting of number of elements to which addition, deletion operation can be done on the elements.

- The first element of the list is called the head of list whereas **the** last element is called tail of the list.

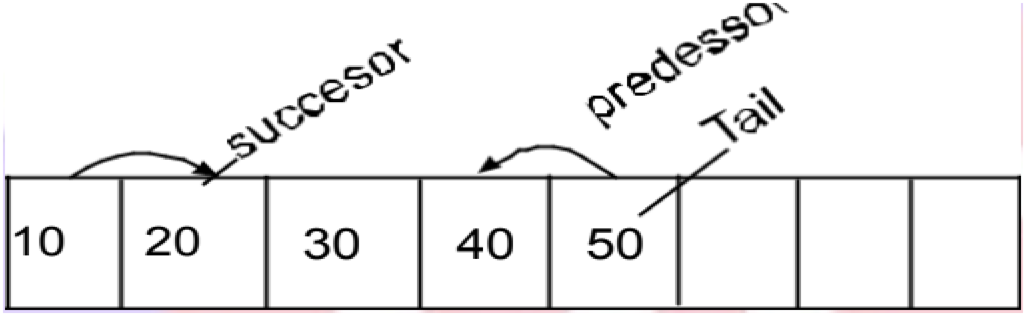


Introduction

List

- The next element of the head of the list is called successor and the previous element is called the predecessor.

- So a head doesn’t have a predecessor and a tail doesn’t have successor and the element in between has both one successor and predecessor.



Basic operation on List

1. Traversing an array list.
2. Searching an element in the list.
3. Insertion of an element in the list.
4. Deletion of an element in the list

Traversing an array List

- Take an array of size 10 which has 5 elements

10	**20**	**30**	**40**	**50**

Traversing an array list:-

- Here each element can be found through indeed no. of array & is incremented by 1.

  When, index =0.

Then, Arr[0] = 10 ⇒ determines 1st element in array.

Index = index +1.

Then, Arr[index]= 20.

- In this way we can transverse each element of array by incrementing the index by 1 until index is not greater than no. of elements.

Searching in an array list

Searching in an array list:-

- For searching an element, we first travers the array list and while traversing, we compare each element of array with the given element.

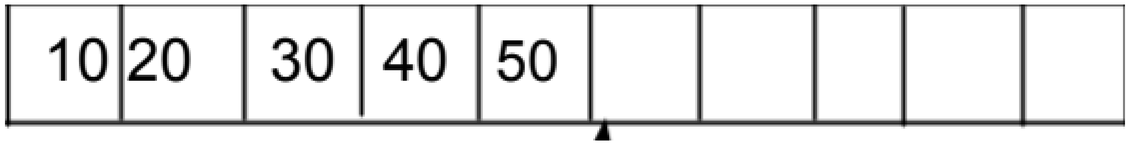
```
int i, item;
for (i=0;i<n:i++)
{
        if(item==arr[i])
        {
                printf("Element Found");
                return;
        }
}
```

Insertion of the element in list

Insertion can be done in two ways:

1. Insertion at end
2. Insertion in between

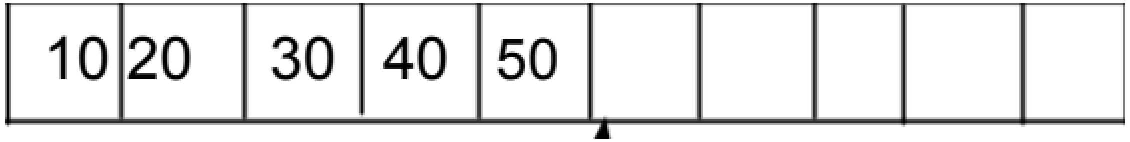
Insertion at end



- Set the array index to the total no. of elements in the array and then insert the element
- index= total no. of elements (i.e. 5)
- Arr[index] = value of inserted element

Insertion of the element in list

Insertion in between



➤ Shift right one position all array elements from last element to the array element before which we want to insert the element.

Insertion of the element in list

Variables we are using here:

1. arr : Name of the array.

2. size : Size of the array (i.e., total number of elements in the array)

3. i : Loop counter or counter variable for the for loop.

4. x : The data element to be insert.

5. pos : The position where we wish to insert the element.

Insertion of the element in list

The following algorithm inserts a data element x into a *given position* pos *(position specified by the programmer)* in a linear array arr.

Algorithm to Insert an element in an Array:

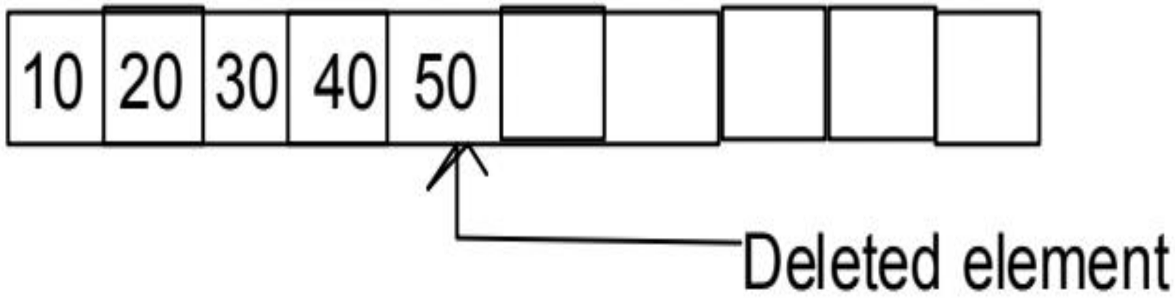
1. Start
2. [Reset size of the array.] set size = size + 1
3. [Initialize counter variable.] Set i = size – 1
4. For i = size - 1 to i >= pos – 1
   I. [Move i^{th} element forward.] set arr[i+1] = arr[i]
   II. [Decrease counter.] Set i = i – 1
   III. [End of step 4 loop.]
5. [Insert element.] Set arr[pos-1] = x
6. Stop.

Insertion of the element in list

```
int temp, item, pos,n;

if(pos==n){

printf("List overflow");

return;

}

if(pos > n+1)
{
printf("Enter position less than or
equal to %d", n+1);
        return;
}
if(pos == n+1){          // Insertion at end

        arr[n] = item;
        n = n+1
        return;
}
temp = n-1;   // Insertion in between

while(temp>=position-1) {

        arr[temp+1] = arr[temp];
        Temp- - ;
}
arr[position-1] = item;
n=n+1;
```

Deletion at end

Deletion of the last element:-



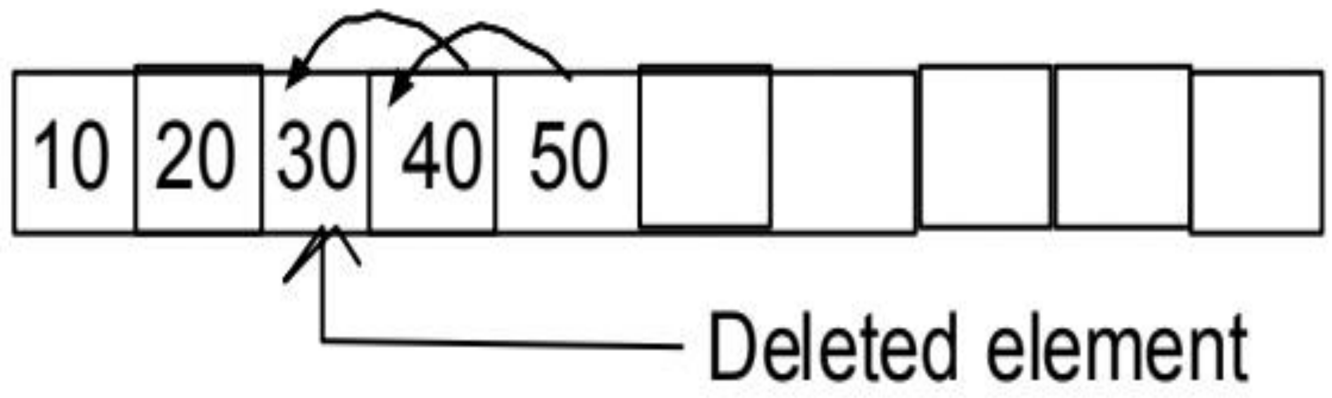
Traverse the array last and if the item is last item of the array, then delete that element & decrease the total no of element by 1,

Deletion in between

- First traverse the array list & compare array element with the element which is to be deleted.

- shift left one position from the next element to the last element of array

- decrease the total no. of elements by 1.



Deletion in between

```
int temp, position , item, n;
If (n= = 0)      // present or not element
{
printf("list underflow");
Return;
}
If (item == arr [n-1]) //deletion at the end.
{
n = n- 1;
return;
}
temp = position -1;
While (temp < = n -1)
{
Arr [temp] = arr [temp +1]
temp + + ;
}
n = n -1;
Return;
}
```

Advantages and Disadvantages of List

Advantages

Easy to compute the address of the array through the index.

Disadvantages of list

- Use of contiguous list which is time consuming.
- As array size is declared, we cannot take elements more than the array size.
- If the elements are less than the size of array, then there is wastage of memory.
- Too many shift operation on insertion and deletion.

STATIC IMPLEMENTATION OF LIST

- Static implementation can be implemented using arrays.
- It is very simple method but it has Static implementation.
- Once a size is declared, it cannot be change during the program execution. It is also not efficient for memory utilization.
- When array is declared, memory allocated is equal to the size of the array.

STATIC IMPLEMENTATION OF LIST

- The vacant space of array also occupiers the memory space.

- In both cases, if we store fewer arguments than declared, the memory is wasted and if more elements are stored than declared, array cannot be expanded.

- **It is suitable only when exact numbers of elements are to be stored.**

DYNAMIC IMPLEMENTATION OF LIST

- In static implementation of memory allocation, we cannot alter (increase or decrease) the size of an array and the memory allocation is fixed.

- There is a special data structure called linked list that provides a more flexible storage system and it does not required the use of array.

- **The advantage of a list over an array occurs when it is necessary to insert or delete an element in the middle of a group of other elements.**

LINKED LIST

- A **linked list** is a **linear collection** of **specially designed data structure, called nodes**, **linked** to one another by means of **pointer**.
- Each node is divided into 2 parts: the first part contains information of the element and the second part contains address of the next node in the link list.

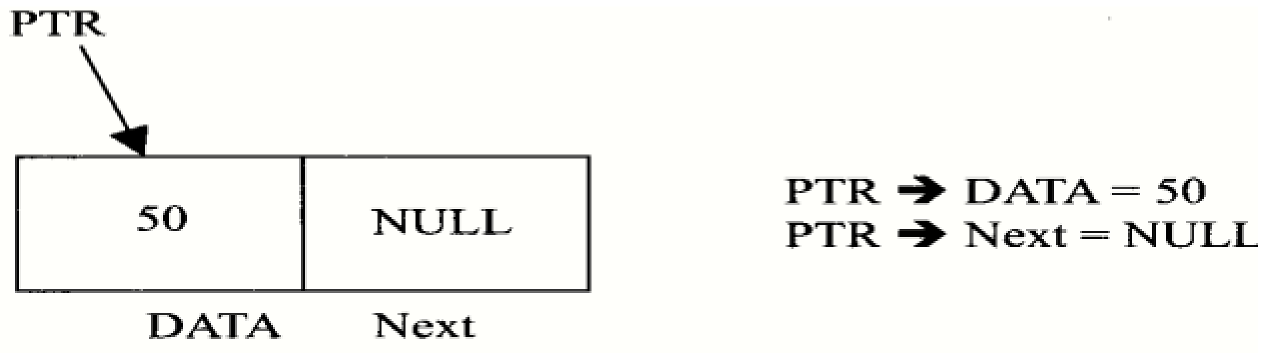


Fig1: Node

LINKED LIST

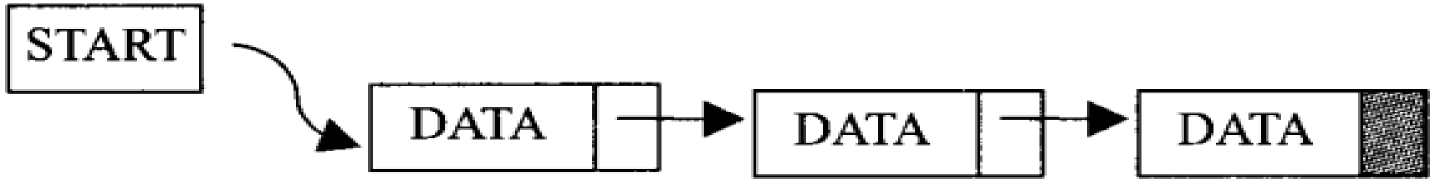


Fig2: Linked List

- Above Fig2 shows that the schematic diagram of a linked list with 3 nodes. Each node is divide with two parts.
- The left part of each node contains the data items and the right part represents the address of the next node.
- The next pointer of the **last node** contains a special value, called the NULL pointer, which does not point to any address of the node.
- That is NULL pointer indicates the end of the linked list.
- **START pointer** will hold the address of the 1st node in the list.
- **START =NULL if there is no list (i.e.; NULL list or empty list).**

Representation of Linked List

- List Suppose we want to store a list of integer numbers using linked list. Then it can be schematically represented as:

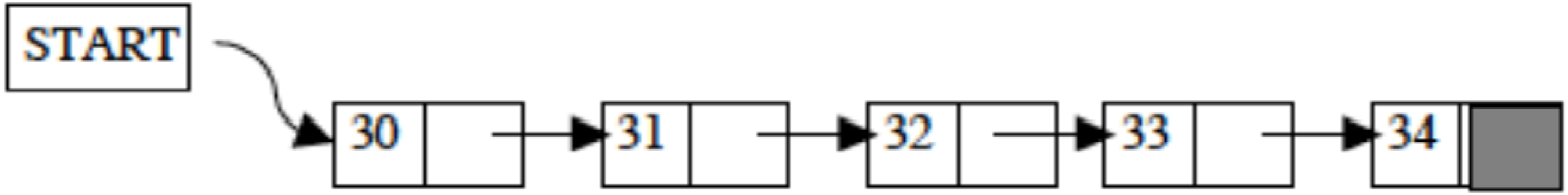


Fig3: Linked List Representation

➢ We can declare linear linked list as follows:

```
struct Node{
        int data;                    //instead of data we also use info
        struct Node *Next;           //instead of Next we also use link
}; typedef struct Node *NODE;
```

Advantages and Disadvantages of Linked List

Linked list have many **advantages** and some of them are:

1. Linked list are **dynamic data structure**. That is, they can grow or shrink during the execution of a program.
2. **Efficient memory utilization:** In linked list (or dynamic) representation, **memory is not pre allocated**. Memory is **allocated whenever it is required**. And it is **deallocated** (or removed) when it is not needed.
3. Insertion and deletion are easier and efficient. Linked list provides **flexibility** in **inserting a data item** at a **specified position** and **deletion** of a data item from the **given position**.
4. Many **complex applications** can be easily **carried out** with **linked list.**

Advantages and Disadvantages of Linked List

Linked list have many **disadvantages** and some of them are:

1. **More memory:** to store an integer number, a node with integer data and address field is allocated. That is more memory space is needed.
2. Access to an arbitrary data item is little bit cumbersome and also time consuming.

Operations on Linked List (Primitive Operations)

The primitive operations performed on Linked list is as follows:

1. Creation
2. Insertion
3. Deletion
4. Traversing
5. Searching
6. Find Previous
7. Concatenation

Operations on Linked List (Primitive Operations)

- **Creation** operation is used to create a linked list. Once a linked list is created with one node, insertion operation can be used to add more elements in a node.

```
Struct node{
int info;
Struct node *link;    //which is self referential structure also known as structure pointer
}
```

Operations on Linked List (Primitive Operations)

- **Insertion** operation is used to **insert a new node** at any **specified location** in the linked list.
- A new node may be inserted:
    i. At the beginning of the linked list
    ii. At the end of the linked list
    iii. At any specified position in between in a linked list

Operations on Linked List (Primitive Operations)

- **Deletion** operation is used to delete an item (or node) from the linked list.

- A node may be deleted from the
  i. Beginning of a linked list
  ii. End of a linked list
  iii. Specified location of the linked list

Operations on Linked List (Primitive Operations)

- **Traversing** is the process of going through all the nodes from one end to another end of a linked list.

- In a singly linked list we can **visit from left to right**, **forward traversing**, nodes only.

- But in doubly linked list **forward and backward** traversing is possible.

Operations on Linked List (Primitive Operations)

- **Concatenation** is the process of appending the second list to the end of the first list.
- Consider a list **A** having **n** nodes and **B** with **m** nodes.
- Then the operation concatenation will place the **1st node of B in the (n+1) the node in A.**
- After concatenation **A** will contain **(n+m)** nodes.

Types of Linked List

Following are the types of Linked list depending upon the arrangements of the nodes:

1. Singly Linked List
2. Doubly linked List
3. Circular Linked List
   i. Circular Singly Linked List
   ii. Circular Doubly Linked List

SINGLY LINKED LIST

- All the nodes in a singly linked list are arranged sequentially by linking with a pointer.
- A singly linked list can grow or shrink, because it is a dynamic data structure.
- The following figure explains the different operations on Singly Linked List.

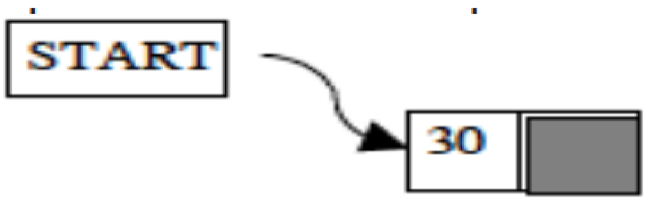


Fig4: Create a node with Data (30)

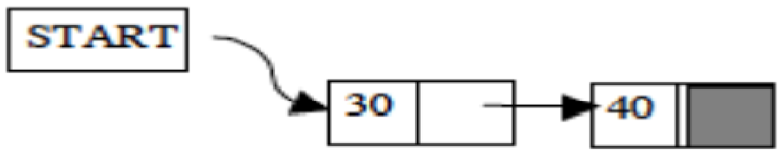
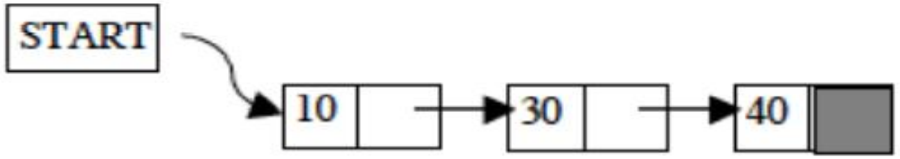


Fig5: Insert a node with Data (40) at the end

SINGLY LINKED LIST



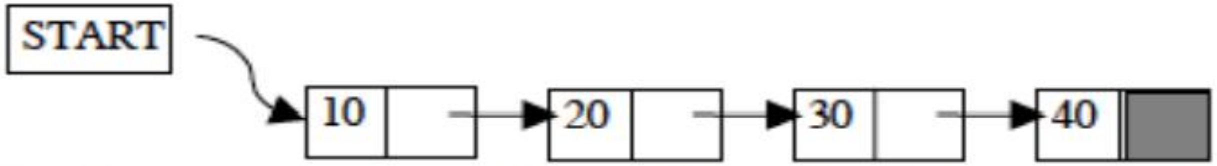
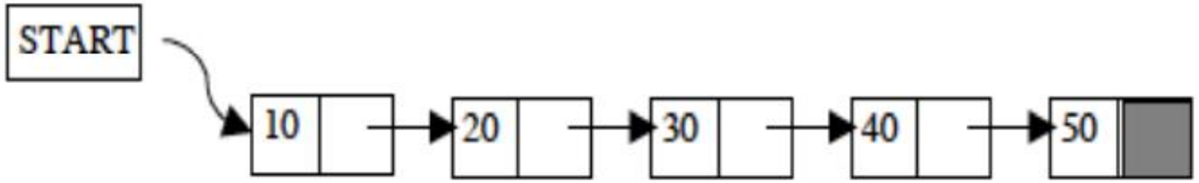


Fig6: Insert a node with Data (10) at the beginning   Fig7: Insert a node with Data (20) at 2nd position



Output → 10, 20, 30, 40, 50

Fig8: Insert a node with Data (50) at the end   Fig9: Traversing the nodes from left to right

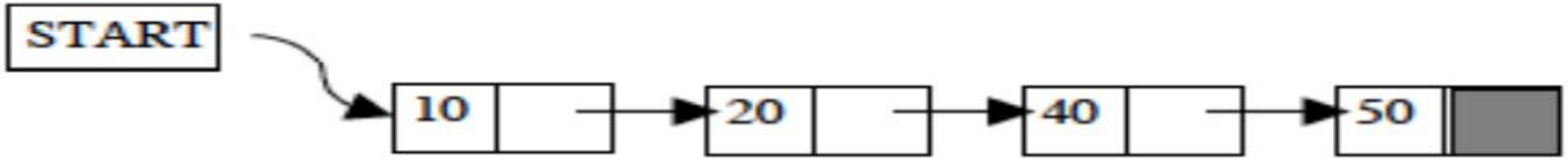


Fig10: Delete 3rd node from the

SINGLY LINKED LIST

Algorithm For Inserting A Node:

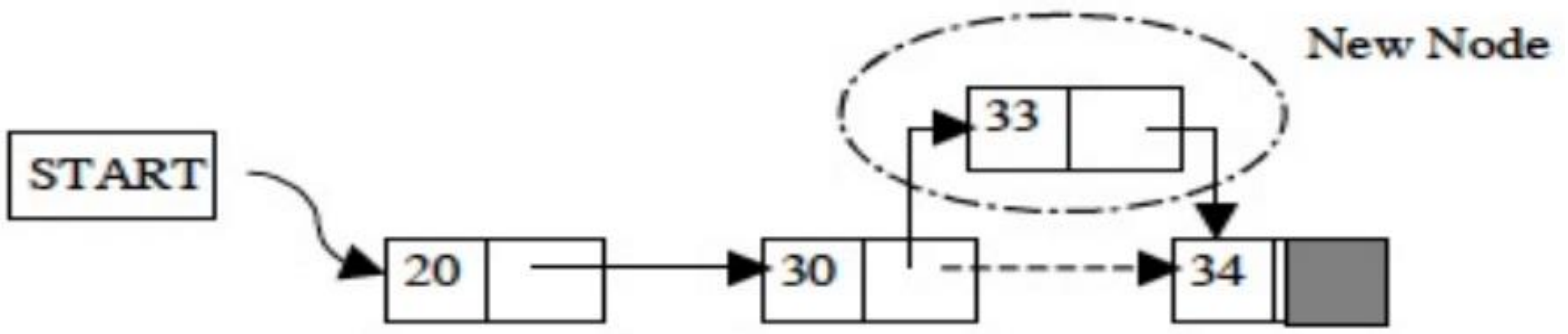


Fig11: Insertion of New Node at specific position

- ❖Suppose **START** is the first position in linked list. Let **DATA** be the element to be inserted in the new node.
- ❖**POS** is the position where the new node is to be inserted. **TEMP** is a temporary pointer to hold the node address.

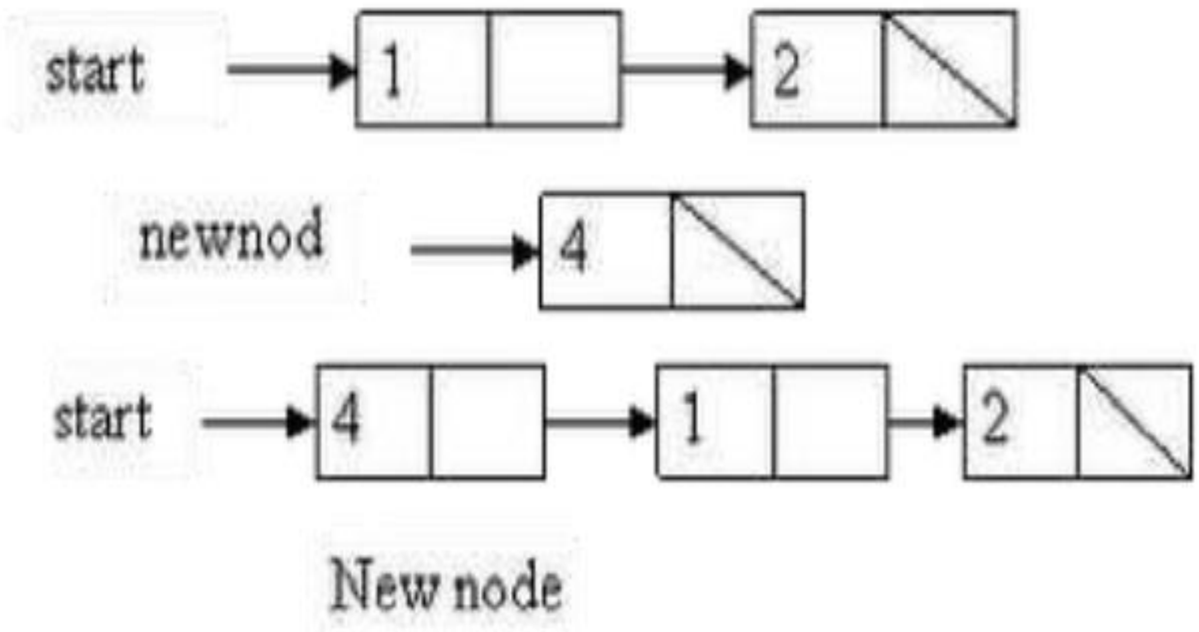
Insert a Node at the beginning of Linked List

1. Read DATA to be inserted
2. Create a NewNode
3. NewNode → DATA = DATA
4. If (START equal to NULL)
    i. NewNode → next = NULL
    ii. start = Newnode.
5. Else

i. NewNode → next = START

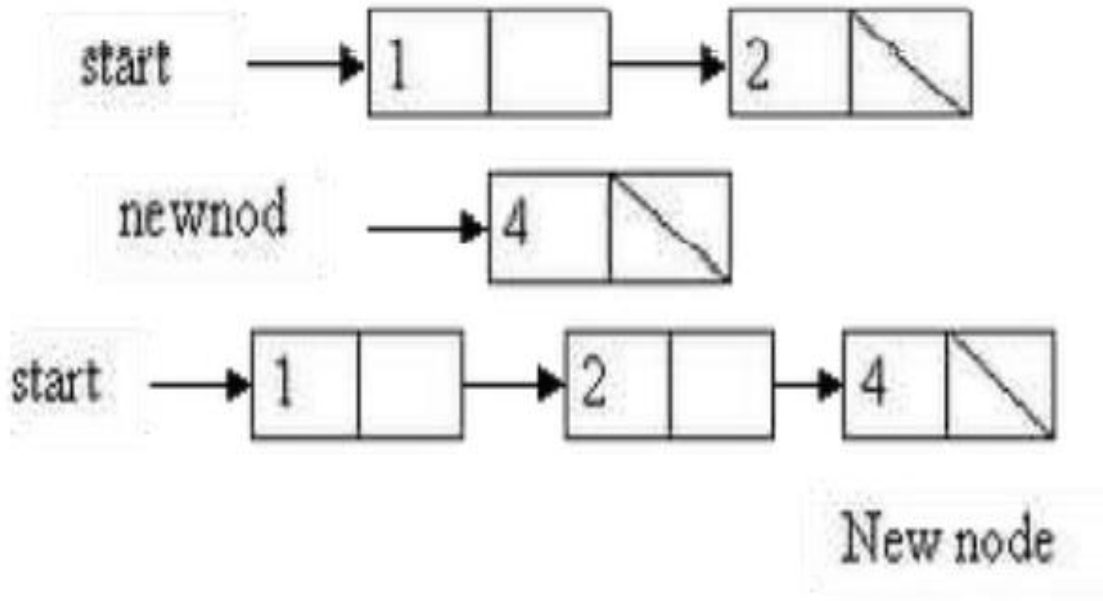
ii. START = NewNode

7. Exit



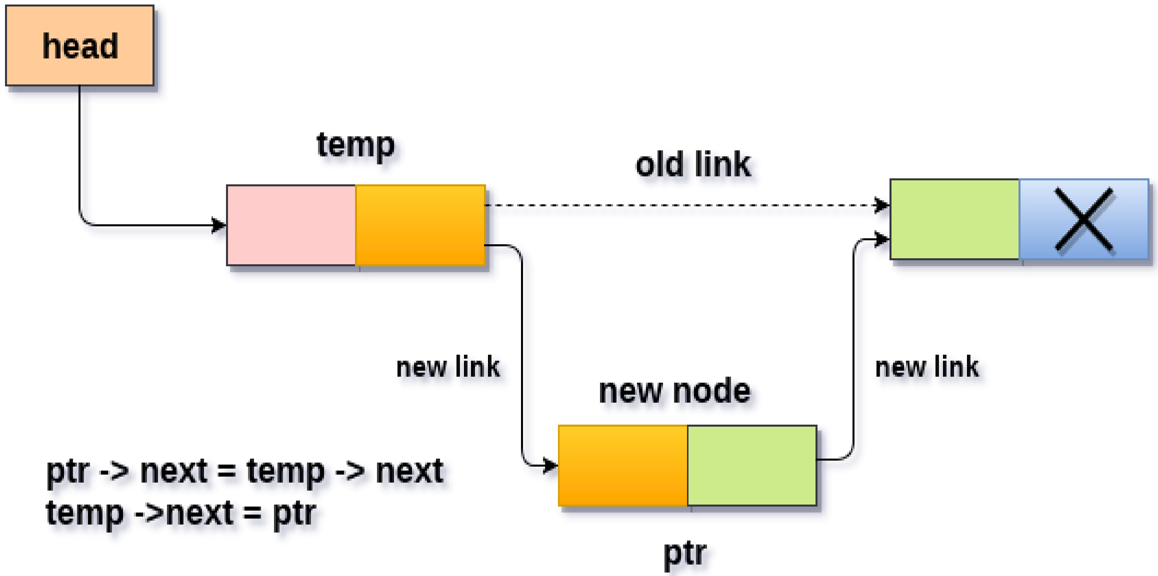
Insert a Node at the end of Linked List

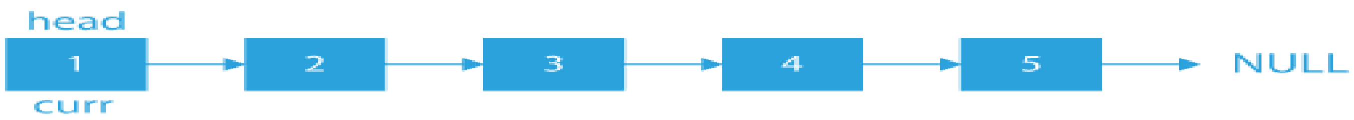
1. Read DATA to be inserted
2. Create a NewNode
3. NewNode → DATA = DATA
4. NewNode → Next = NULL
5. If (SATRT equal to NULL)
    a. START = NewNode
6. Else
    a. TEMP = START
    b. While (TEMP → Next not equal to NULL)
        i. TEMP = TEMP → Next
7. TEMP → Next = NewNode
8. Exit



Insert a Node at any specified position of Linked List

1. Read DATA and POS to be inserted.
2. Initialize TEMP = START; and count = 1.
3. Repeat the step 3 for (count = 1; count <pos-1; count++)
    a. TEMP = TEMP->Next
    b. If (TEMP ==NULL)
        i. Display “Node in the list less than the position”
        ii. Exit
4. Create a New Node
5. NewNode → DATA = DATA
6. NewNode → Next = TEMP → Next
7. TEMP → Next = NewNode
8. Exit

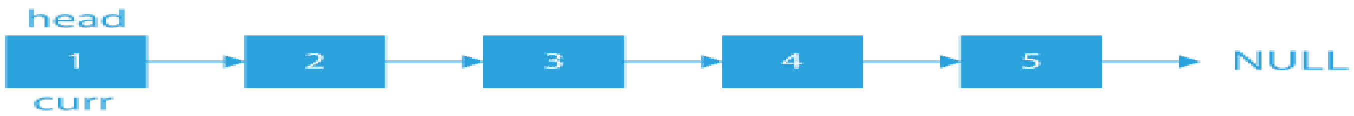




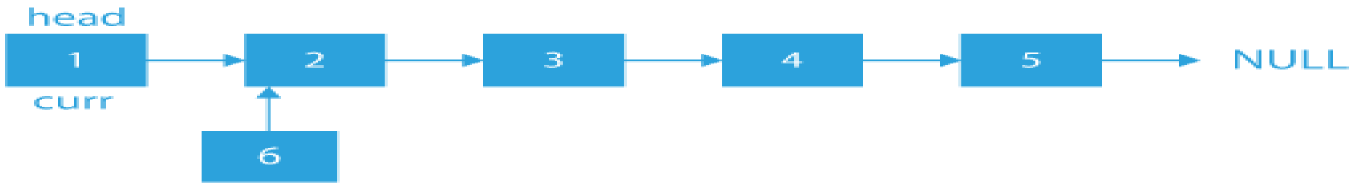
position = 2 Node to be inserted =


```
for (int i=1;i<(pos - 1); i++)
{ curr = curr → next; }
```

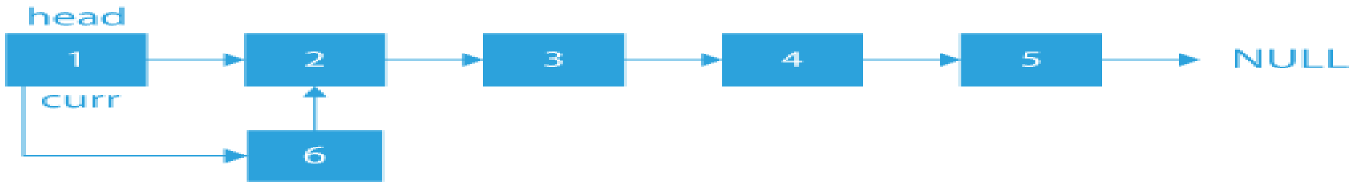
As i = 1 is not less than (position - 1) = 2 - 1,
loop will terminate



Make newNode → next = curr → next



Make curr → next = newnode



Final linked list

Algorithm For Display All Nodes

➢ Suppose START is the address of the first node in the linked list. Following algorithm will visit all nodes from the START node to the end.

1. Start.
2. If (START is equal to NULL)
   i. Display “The list is Empty”
   ii. Exit
3. Initialize TEMP = START
4. Repeat the step 4 and 5 until (TEMP->next !=NULL)
   i. Display “TEMP → DATA”
   ii. TEMP = TEMP → Next
5. Display TEMP-> DATA
6. Exit

SINGLY LINKED LIST

Algorithm For Deleting A Node

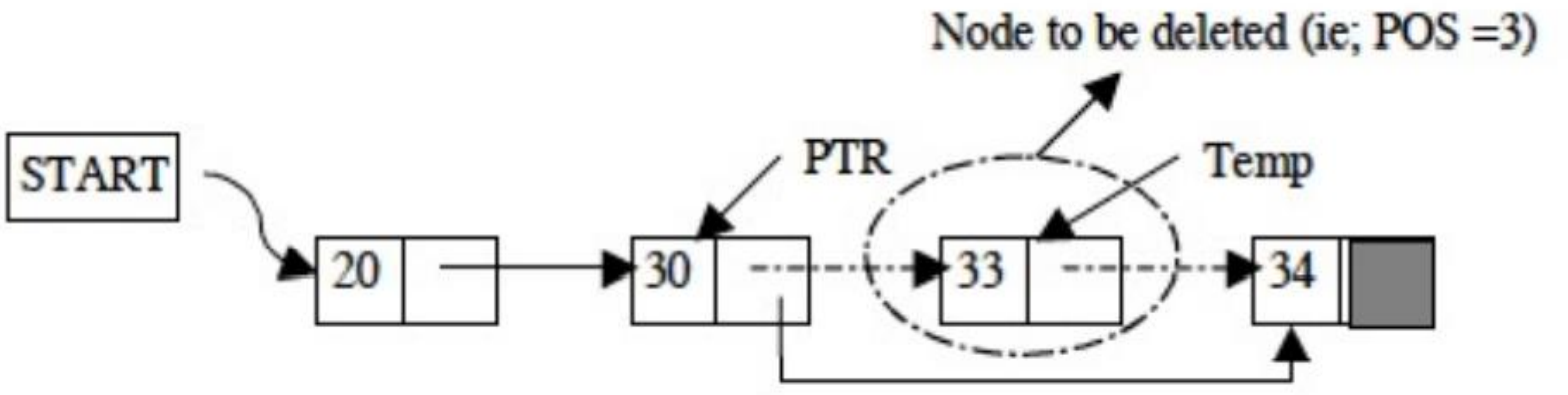


Fig12: Deletion of Node

- Suppose START is the first position in linked list.
- Let DATA be the element to be deleted.
- previous, current is a temporary pointer to hold the node address.

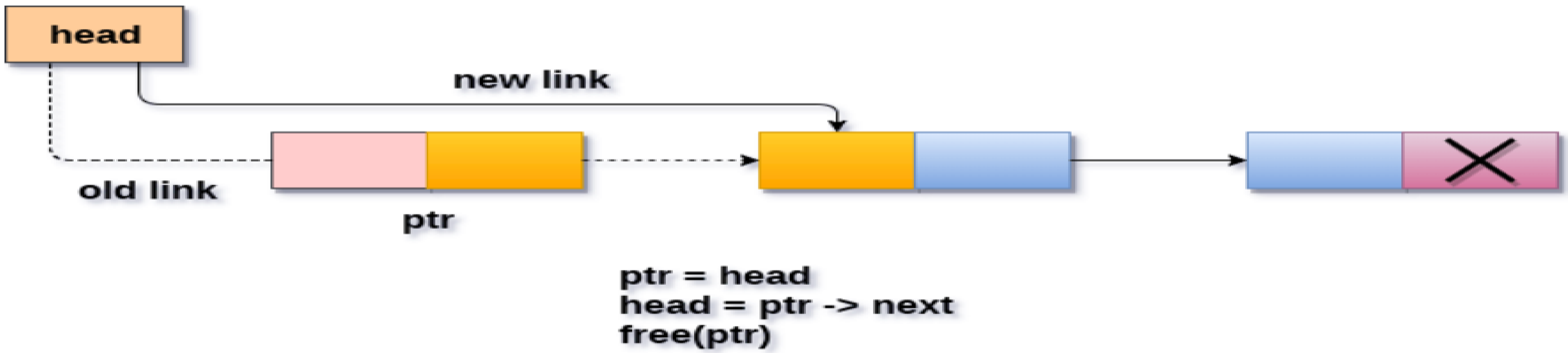
Deletion in singly linked list at beginning

Algorithm

- **Step 1:** IF HEAD = NULL

  Write UNDERFLOW
  Go to Step 5
  [END OF IF]

- **Step 2:** SET PTR = HEAD
- **Step 3:** SET HEAD = HEAD -> NEXT
- **Step 4:** FREE PTR
- **Step 5:** EXIT



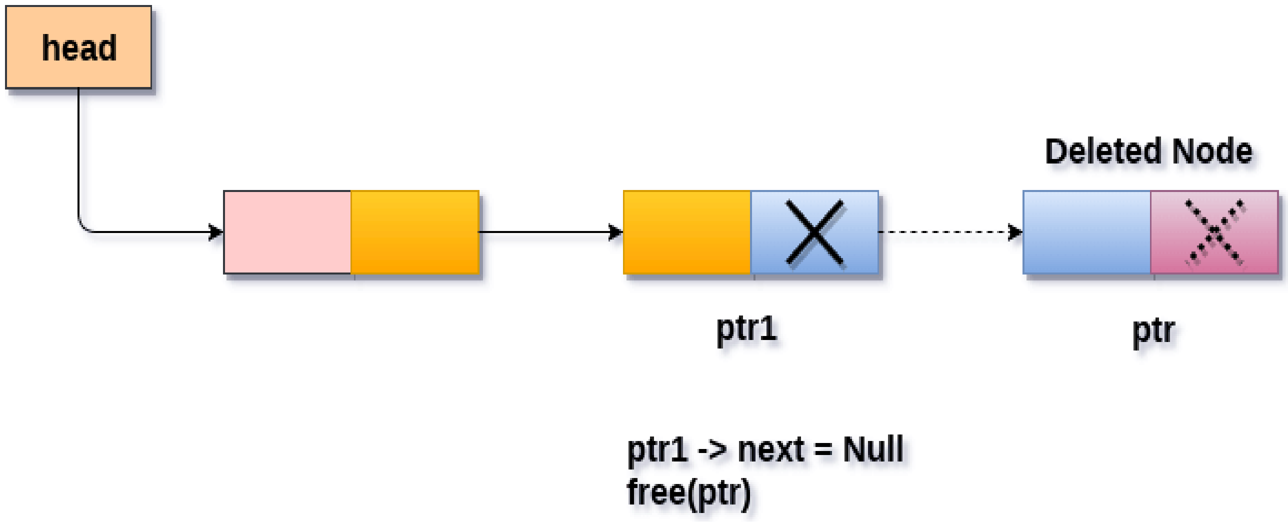
SINGLY LINKED LIST: Algorithm For Deleting A Last Node

- **Step 1:** IF HEAD == NULL

  Write UNDERFLOW

  Go to Step 8

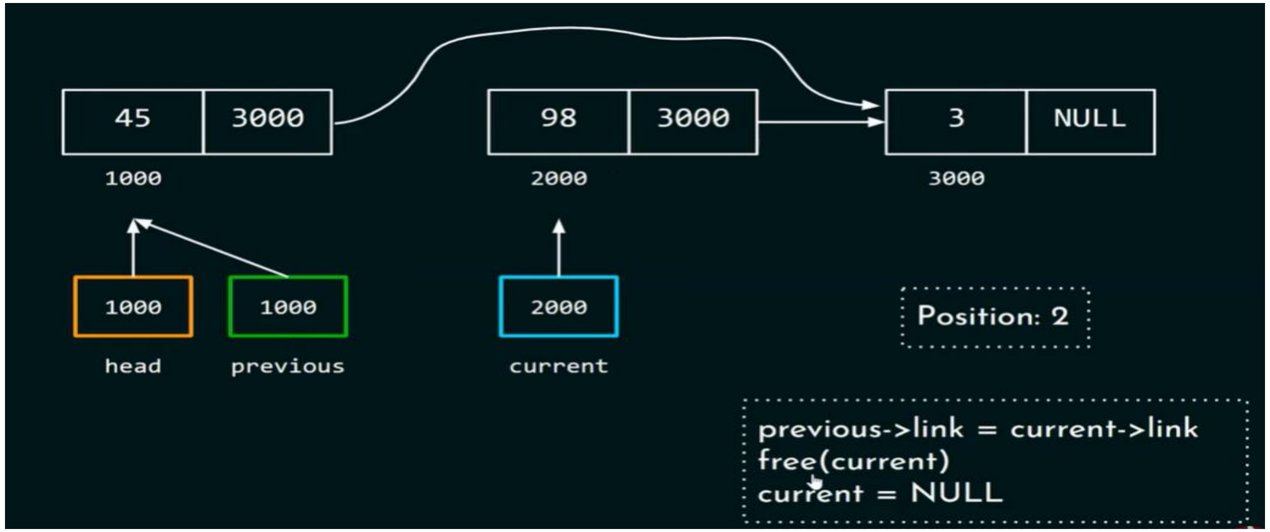
  [END OF IF]
- **Step 2:** SET PTR = HEAD
- **Step 3:** Repeat Steps 4 and 5 while (PTR -> NEXT!= NULL)
  - **Step 4:** SET PTR1 = PTR
  - **Step 5:** SET PTR = PTR -> NEXT
  - [END OF LOOP]
- **Step 6:** SET PTR1-> NEXT = NULL
- **Step 7:** FREE PTR
- **Step 8:** EXIT

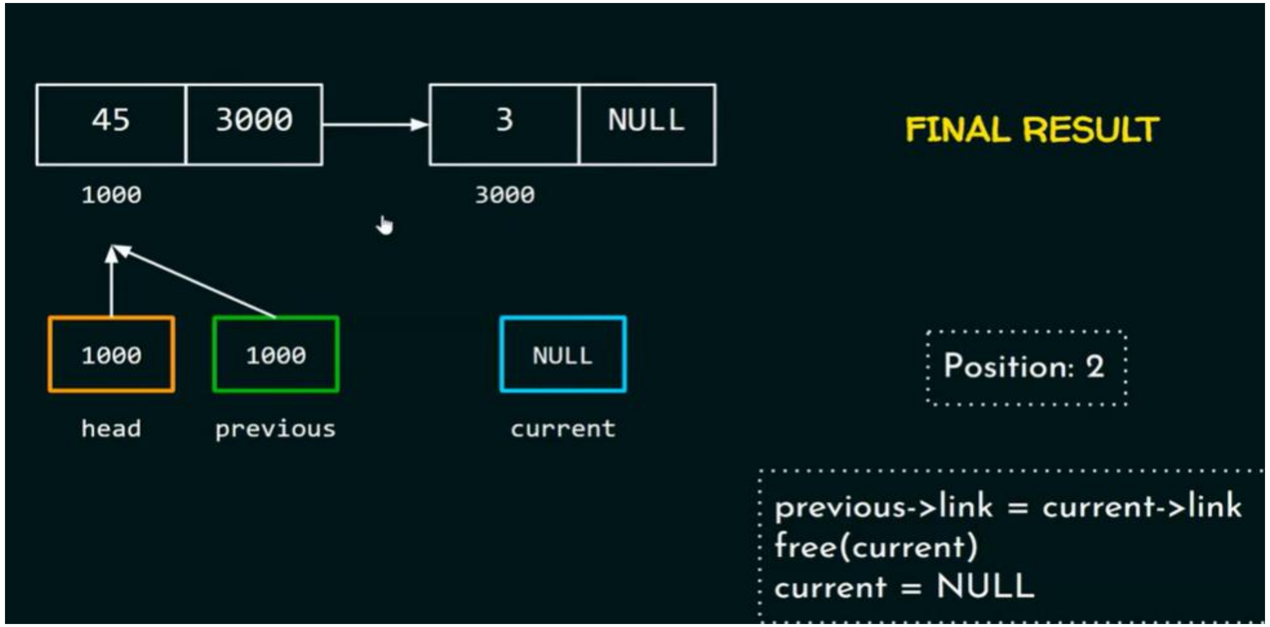


Deleting a node from the last

Algorithm For Deleting at specified position

1. Enter the position POS
2. Set previous = current=START
3. Set i=0
4. while (i<POS-1)
    i. Previous = current
    ii. current=current->next
    iii. i++
5. previous->next=current->next
6. Free(current)
7. Exit





Algorithm For Searching A Node

1. Read the DATA to be searched
2. Initialize TEMP = START; POS =1;
3. DO the step 4, 5 and 6.
4. If (TEMP → DATA is equal to DATA)
   i. Display "The data is found at POS"
   ii. Exit
5. TEMP = TEMP → Next
6. POS = POS+1
7. WHILE(TEMP->next!=NULL)
8. If (TEMP is equal to NULL)
   i. Display "The data is not found in the list"
9. Exit

Stack Using Linked List

The following figure shows that the implementation of stack using linked list:

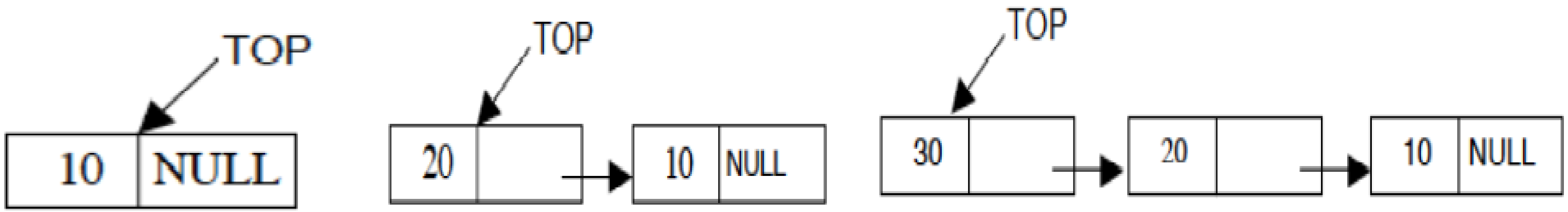


Fig13: Push (10) Fig14: Push (20) Fig15: Push (30)

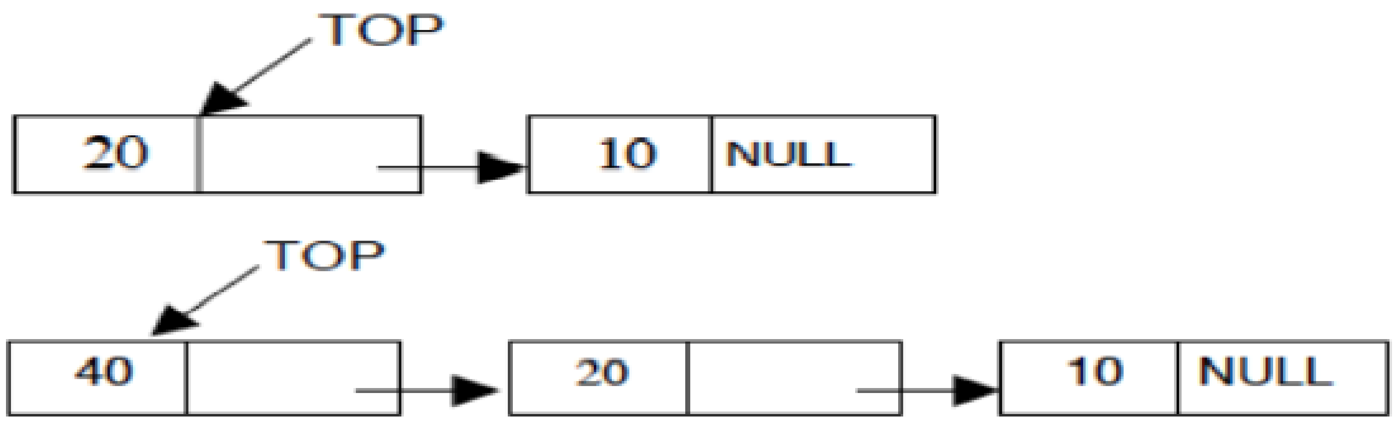


Fig16: X = Pop (i.e. X=30) Fig17: Push (40)

Algorithm For Push Operation

Suppose TOP is a pointer, which is pointing towards the topmost element of the stack. TOP is NULL when the stack is empty. DATA is the data item to be pushed.

1. Read the DATA to be pushed
2. Create a New Node
3. NewNode → DATA = DATA
4. If(TOP == NULL)
   i. TOP = NewNode
   ii. NewNode → Next = NULL
5. Else
   i. NewNode → Next = TOP
   ii. TOP = NewNode
6. Exit

Algorithm For Pop Operation

- ❖Algorithm For POP Operation Suppose TOP is a pointer, which is pointing towards the topmost element of the stack.
- ❖TOP is NULL when the stack is empty.
- ❖TEMP is pointer variable to hold any nodes address. DATA is the information on the node which is just deleted.

1. if (TOP is equal to NULL)
   a. Display “The stack is empty”
2. Else
   a. TEMP = TOP
   b. Display “The popped element TOP → DATA”
   c. TOP = TOP → Next
   d. TEMP → Next = NULL
   e. Free(TEMP)
3. Exit

Queue Using Linked List

The following figure shows that the implementation issues of Queue using linked list.

Fig18: Enqueue (10)

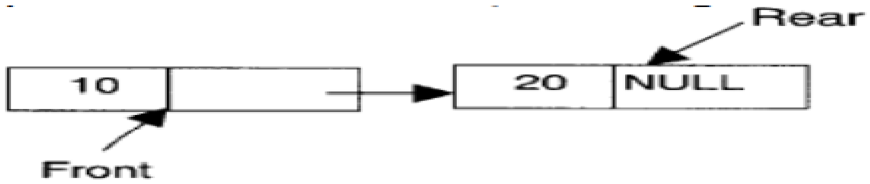


Fig19: Enqueue (20)

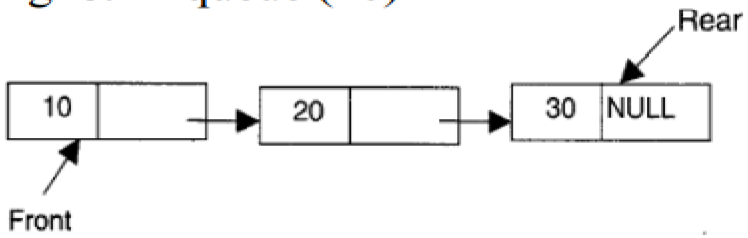


Fig20: Enqueue (30)

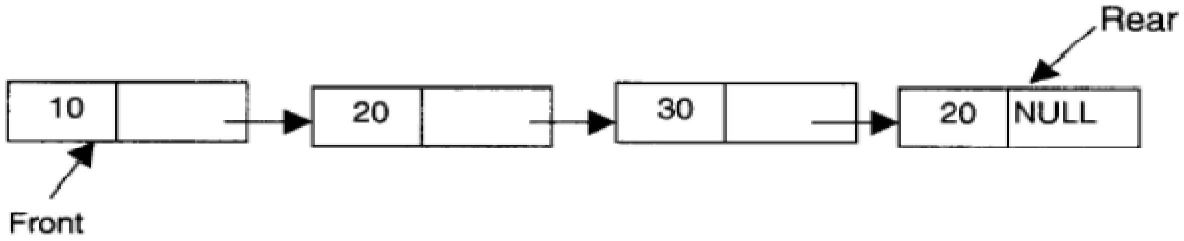


Fig21: Enqueue (20)

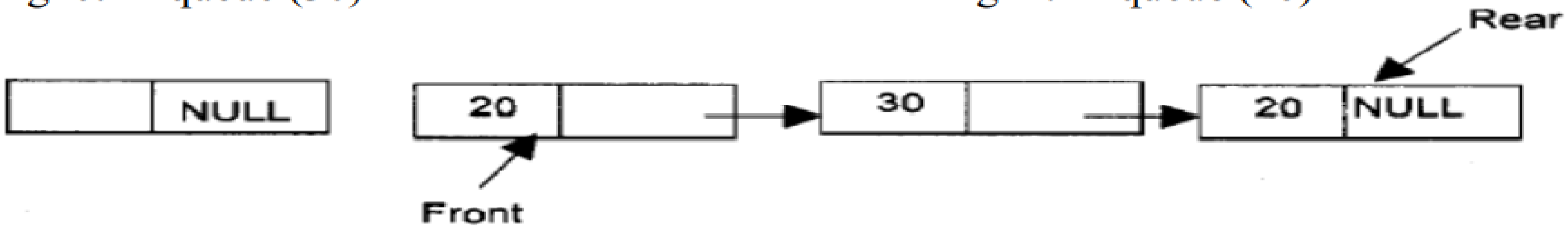


Fig22: X = Dequeue (i.e. X = 10)

Algorithm for insert an element into a Queue

- ❖REAR is a pointer in queue where the new elements are added.
- ❖FRONT is a pointer, which is pointing to the queue where the elements are popped. DATA is an element to be pushed.

1. Start.
2. Read the DATA element to be pushed.
3. Create a New Node.
4. NewNode → DATA = DATA
5. NewNode → Next = NULL
6. If(FRONT==NULL && REAR == NULL)
   i. FRONT = NewNode
   ii. REAR = NewNode
7. If(REAR not equal to NULL)
   i. REAR → next = NewNode;
8. REAR =NewNode;
9. Exit

Algorithms for delete an element from the queue

- ❖REAR is a pointer in queue where the new elements are added.
- ❖FRONT is a pointer, which is pointing to the queue where the elements are popped. DATA is an element popped from the queue.

1. Start.
2. If (FRONT is equal to NULL)
   i. Display "The Queue is empty"
3. Else
   i. Display "The popped element is FRONT → DATA"
   ii. If (FRONT is not equal to REAR)
      a. Temp = front
      b. FRONT = FRONT → Next
      c. Free(temp)
   iii. Else
      a. FRONT = NULL;
4. Exit

Advantages and Advantages of Singly Linked List

Advantages of Singly Linked List

1. Accessibility of a node in the forward direction is easier.
2. Insertion and deletion of node are easier.

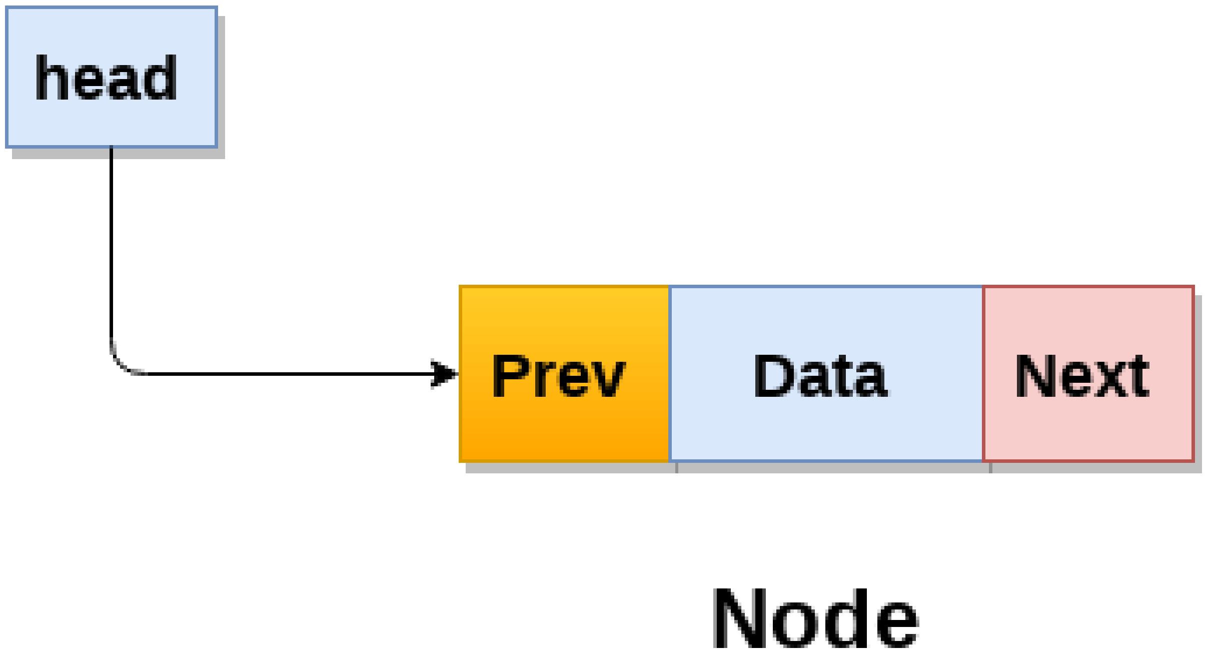
Disadvantages of Singly Linked List

1. Can insert only after a referenced node.
2. Removing node requires pointer to previous node.
3. Can traverse list only in the forward direction.

Doubly Link List

- ❖A doubly linked list is one in which all nodes are linked together by multiple links which help in accessing both the successor (next) and predecessor (previous) node for any arbitrary node within the list.

- ❖Every nodes in the doubly linked list has three fields: LeftPointer(Prev), RightPointer(Next) and DATA.



Doubly Link List

- Prev will point to the node in the left side (or previous node) i.e. Prev will hold the address of the previous node. Next will point to the node in the right side (or next node), i.e. Next will hold the address of next node.
- DATA will store the information of the node.

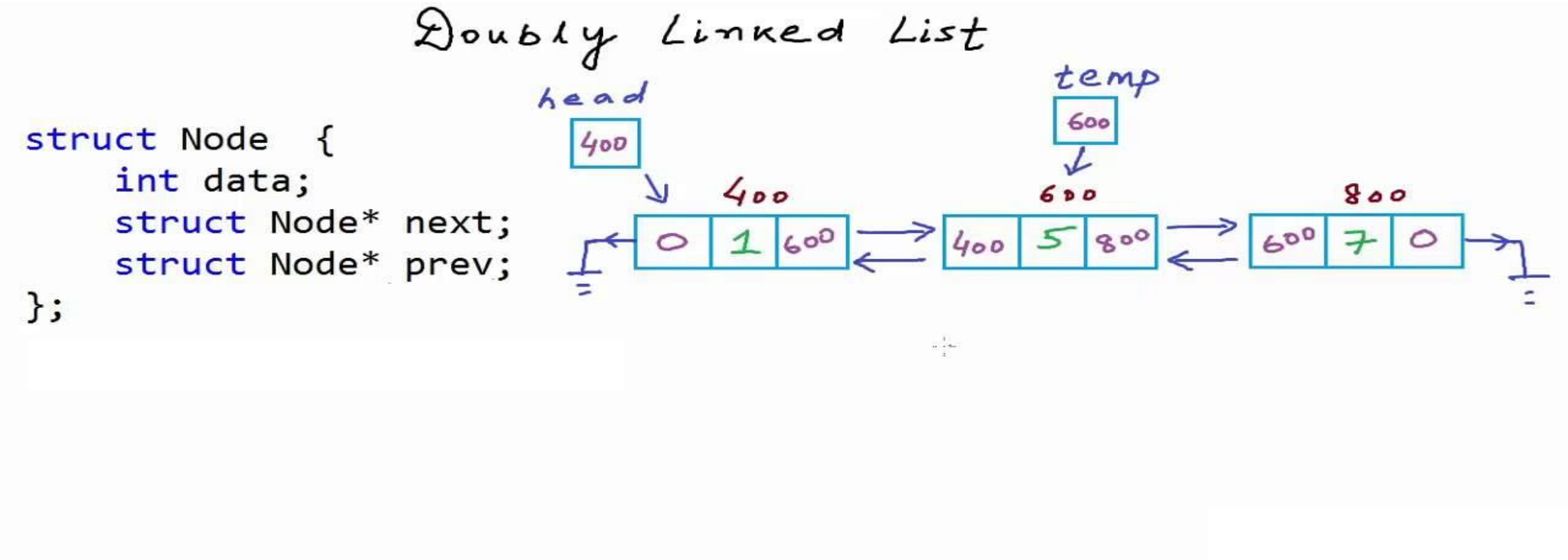


Fig23:Doubly Link List

Doubly Link List

- ❖Representation of Doubly Linked List Following declaration can represent doubly linked list

```
struct Node {
        int data;
        struct Node *Next;
        struct Node *Prev;
};
struct Node *NODE;
```

Doubly Linked List

❖All the primitive operations performed on singly linked list can also be performed on doubly linked list.

❖The following figures shows that the insertion and deletion of nodes.

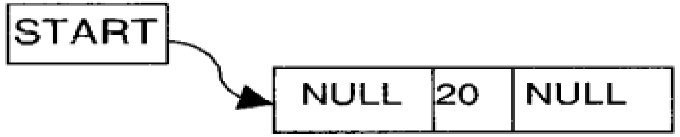


Fig25: Add (20)

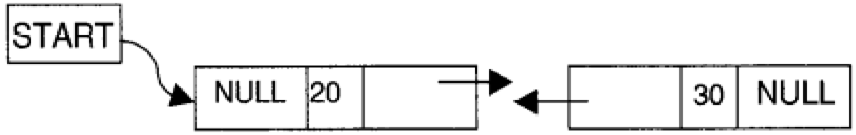


Fig26: Insert (30) at the end

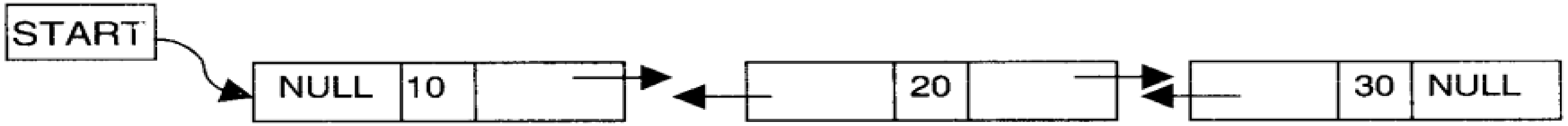


Fig27: Insert (10) at the Beginning

Fig28: Delete a node at the 2nd position (Delete 20 at 2nd position)

Algorithm for Inserting a Node in doubly linked list

❖Suppose START is the first position in linked list.

❖Let DATA be the element to be inserted in the new node.

❖POS is the position where the NewNode is to be inserted.

❖TEMP is a temporary pointer to hold the node address.

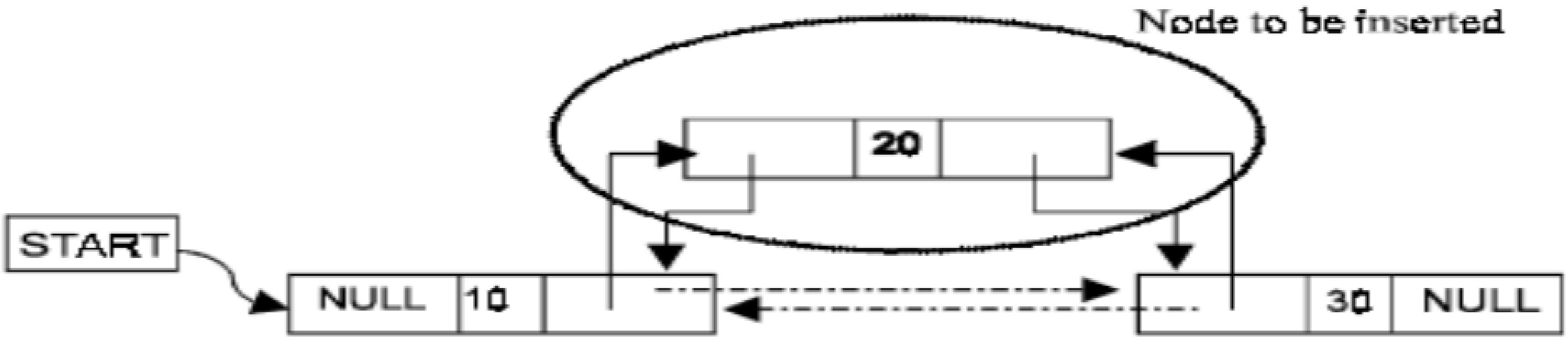
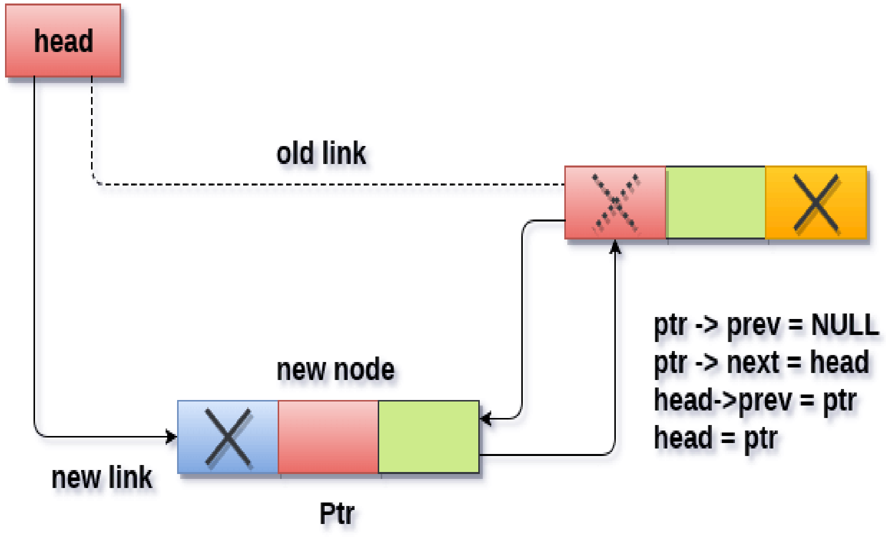


Fig29: Insert a node at the 2nd position

Insertion in doubly linked list at beginning

1. START
2. CREATE NEW_NODE
3. NEW_NODE->DATA = DATA
4. IF(HEAD == NULL)
   i. HEAD = NEW_NODE
   ii. NEW_NODE->PREV = NULL
   iii. NEW_NODE->NEXT = NULL
5. ELSE
   i. NEW_NODE -> PREV = NULL
   ii. NEW_NODE -> NEXT = HEAD
   iii. HEAD -> PREV = NEW_NODE
   iv. HEAD = NEW_NODE
6. EXIT



Insertion into doubly linked list at beginning

Insertion in doubly linked list at End

Step 1: CREATE NEW_NODE

Step 2: SET NEW_NODE -> DATA = DATA

Step 3: IF (HEAD == NULL)

SET HEAD = NEWNODE

NEW_NODE -> NEXT = NULL

NEW_NODE->PREV = NULL

Step 4 ELSE

SET TEMP = HEAD

while (TEMP -> NEXT != NULL)

SET TEMP = TEMP -> NEXT

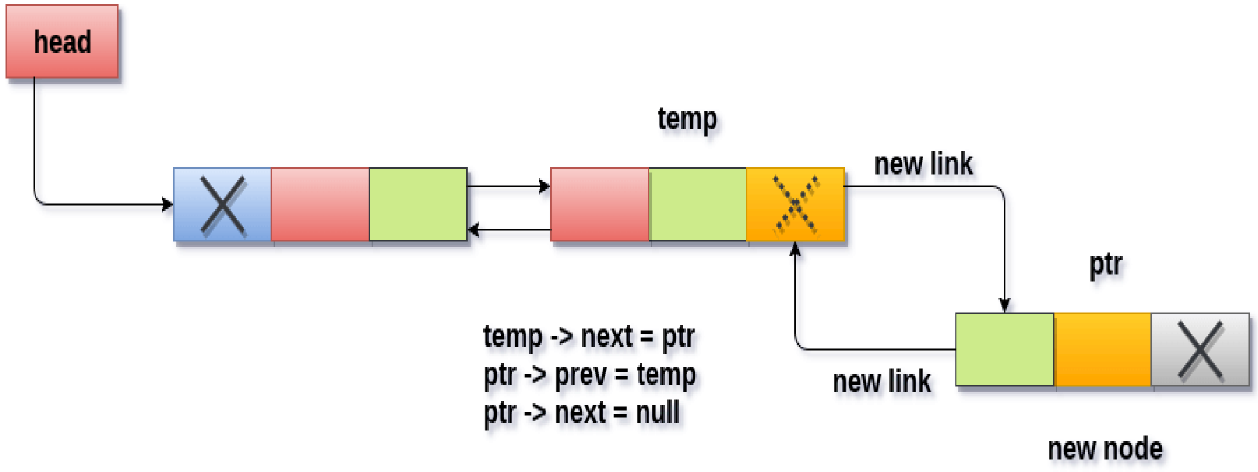
[END OF LOOP]

Step 5: SET TEMP -> NEXT = NEW_NODE

Step 6: SET NEW_NODE -> PREV = TEMP

Step 7: SET NEW_NODE ->NEXT = NULL

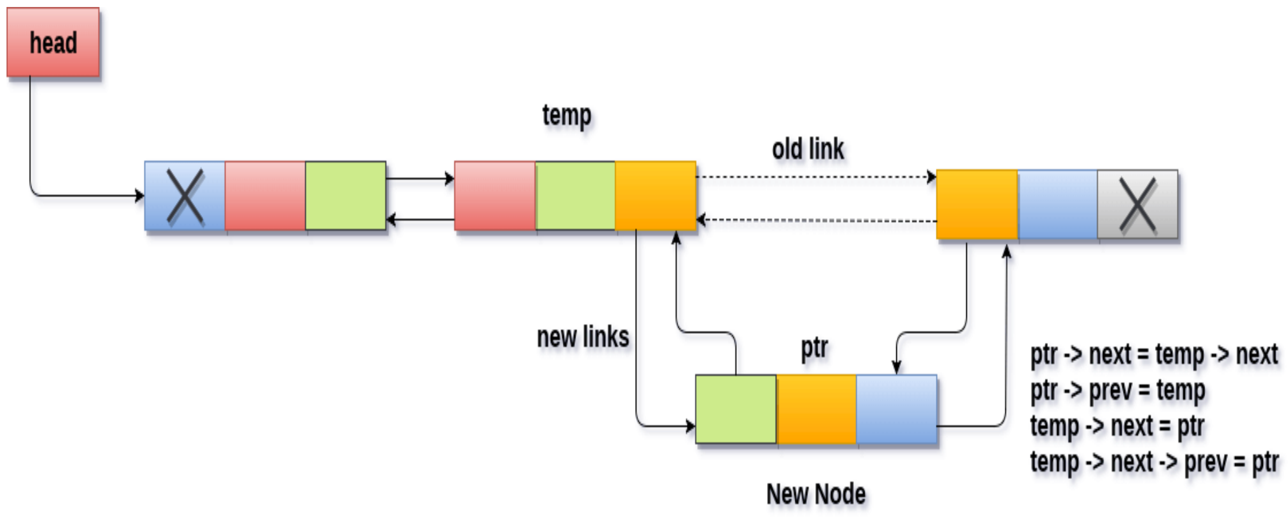
Step 7: EXIT



Insertion into doubly linked list at the end

Algorithm for Inserting a Node at specified position in doubly linked list

1. Start.
2. Read the DATA and POS
3. Initialize TEMP = START, i=1;
4. While((i<POS-1) and (TEMP is not equal to NULL))
    i. TEMP = TEMP → Next
    ii. i++
5. If (TEMP not equal to NULL) and (i equal to POS-1)
    i. Create a New Node
    ii. NewNode → DATA = DATA
    iii. NewNode → Next = TEMP → Next
    iv. NewNode → Prev = TEMP
    v. (TEMP → Next) → Prev = NewNode
    vi. TEMP → Next = New Node
6. Else
    a. Display “Position NOT found”
7. Exit.



Insertion into doubly linked list after specified node

Algorithm for Deleting a Node in doubly linked list

- ❖Suppose START is the address of the first node in the linked list.
- ❖Let POS is the position of the node to be deleted.
- ❖TEMP is the temporary pointer to hold the address of the node.
- ❖After deletion, DATA will contain the information on the deleted node.

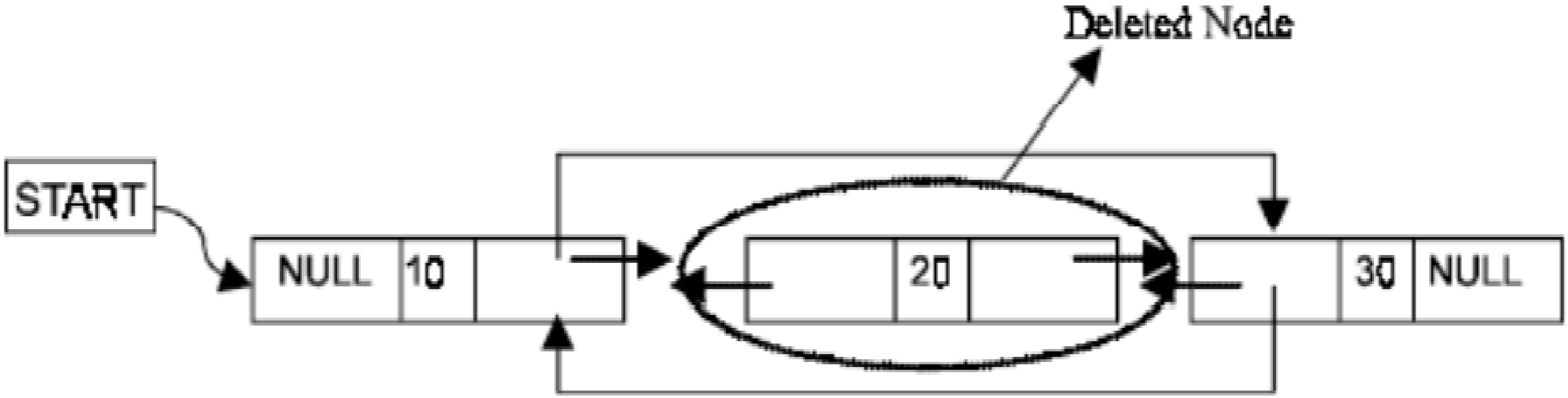
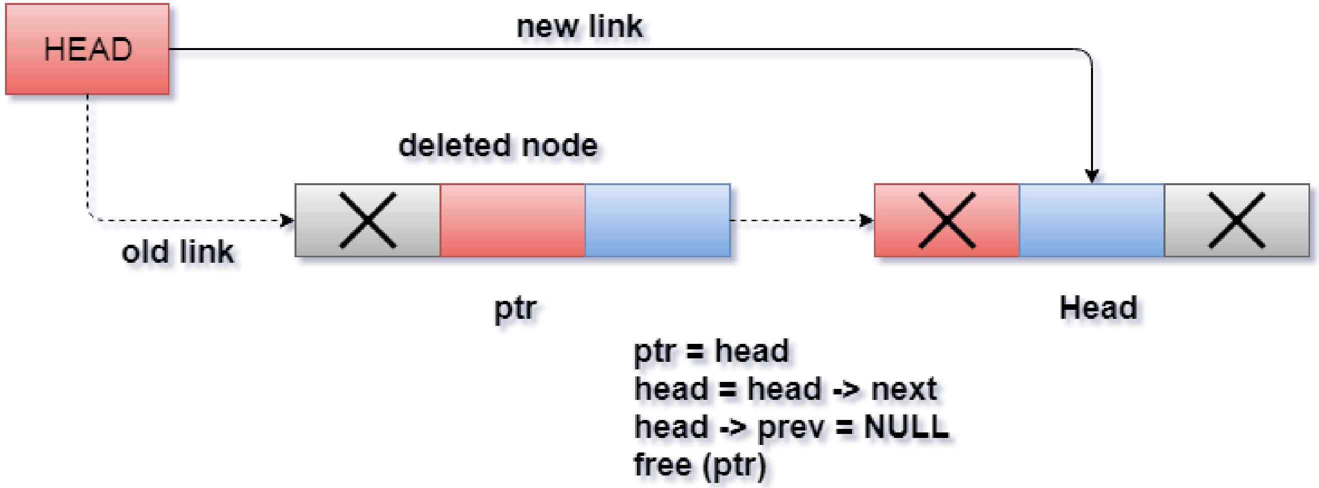


Fig30: Delete a node at the 2nd position

Algorithm for Deleting a Node at beginning in doubly linked list

- **STEP 1:** IF HEAD = NULL

  WRITE UNDERFLOW
  GOTO STEP 6
- **STEP 2:** SET PTR = HEAD
- **STEP 3:** SET HEAD = HEAD → NEXT
- **STEP 4:** SET HEAD → PREV = NULL
- **STEP 5:** FREE PTR
- **STEP 6:** EXIT



Deletion in doubly linked list from beginning

An algorithm to delete last element in double linked list

- **Step 1:** IF HEAD = NULL

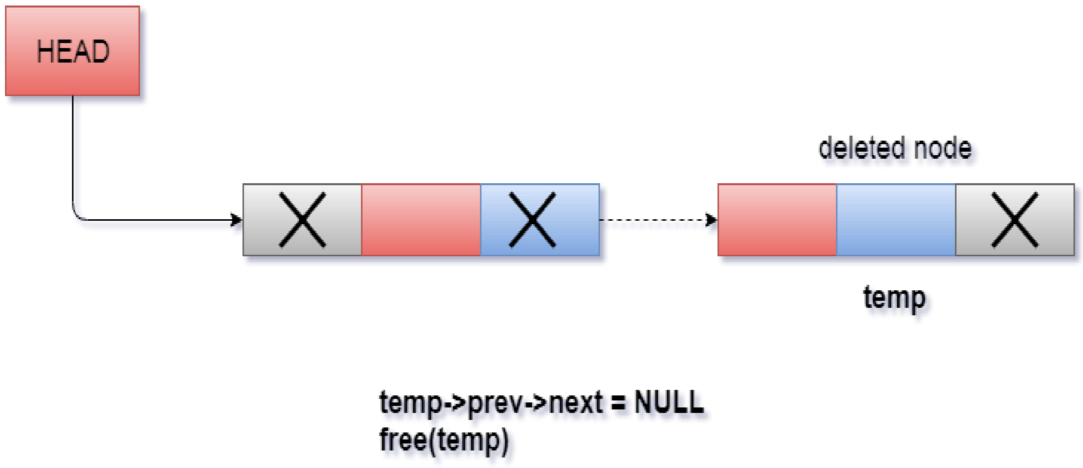
    Write UNDERFLOW
    Go to Step 6
    [END OF IF]

- **Step 2:** SET TEMP = HEAD
- **Step 3:** REPEAT STEP 4 WHILE TEMP->NEXT != NULL

    SET TEMP = TEMP->NEXT

    [END OF LOOP]

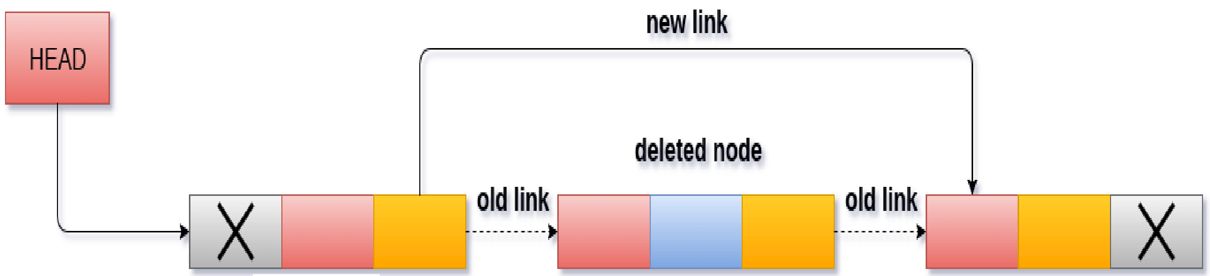
- **Step 4:** SET (TEMP ->PREV)-> NEXT = NULL
- **Step 5:** FREE TEMP
- **Step 6:** EXIT



Deletion in doubly linked list at the end

Algorithm for Deleting a Node in specified position

1. **Step 1:**IF HEAD = NULL
   Write UNDERFLOW
   Go to Step 7
   [END OF IF]
2. **Step 2:** SET TEMP = HEAD
3. **Step 3:** Repeat Step 4 while (TEMP -> DATA != ITEM)
   SET PTR = TEMP
   SET TEMP = TEMP->NEXT
   [END OF LOOP]
4. **Step 4:** SET PTR->NEXT = TEMP->NEXT
5. **Step 5:** SET (TEMP->NEXT)->PREV = PTR
6. **Step 6:** FREE TEMP
7. **Step 7:** EXIT



Deletion of a specified node in doubly linked list

CIRCULAR LINKED LIST

- A circular linked list is one, which has no beginning and no end.

- A singly linked list can be made a circular linked list by simply storing the address of the very first node in the linked field of the last node.

- Circular linked lists also make our implementation easier, because they eliminate the boundary conditions associated with the beginning and end of the list, thus eliminating the special case code required to handle these boundary conditions.

CIRCULAR LINKED LIST

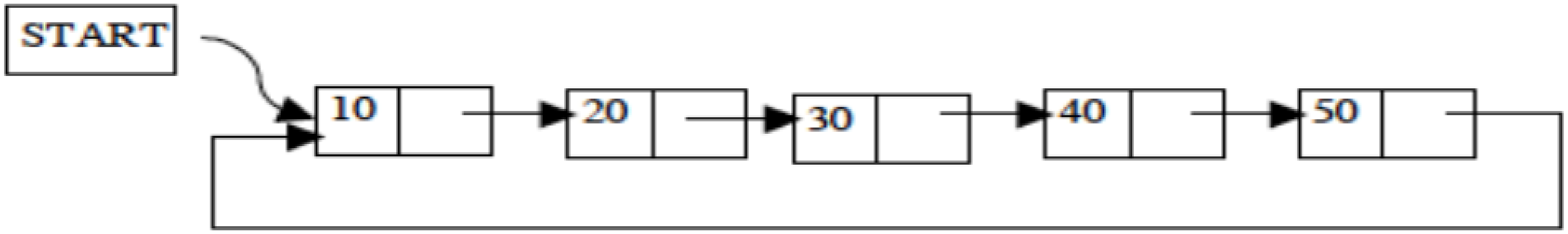


Fig31: Circular Linked List

A circular doubly linked list has both the successor pointer and predecessor pointer in circular manner as shown in the following Fig.

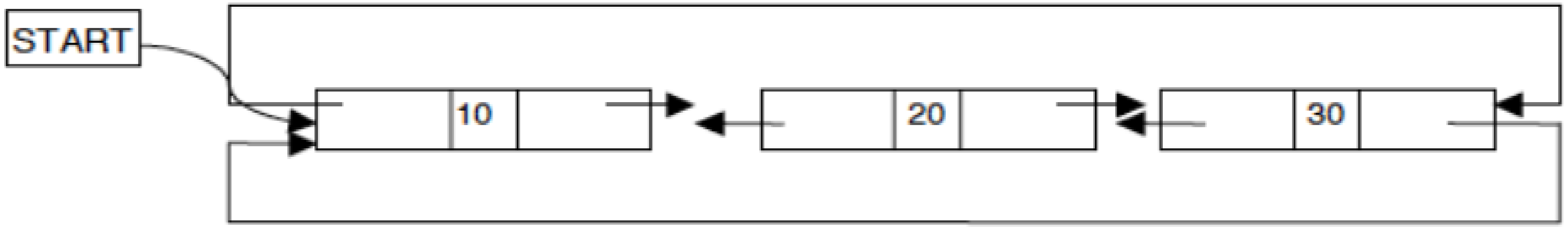


Fig32: Circular Doubly Linked List

CIRCULAR LINKED LIST ALGORITHM

Inserting a node at the beginning (Insert_First(START,Item)

1. start
2. Create a Newnode
3. IF START is equal to NULL then
    a. Set Newnode->data = item
    b. Set NewNode->next = Newnode
    c. Set START = Newnode
    d. Set Last = Newnode
    e. Exit
4. Else
    1. Set Newnode ->data = Item
    2. Set Newnode->next = Start
    3. Set START = Newnode
    4. Set last->next = Start.
5. Stop

Inserting a node at the End (Insert_End(START,Item)

1. Start.
2. Create a Newnode.
3. If START is equal to NULL, then
   a. Newnode->data = Item
   b. Newnode->next = Newnode
   c. START = Newnode
   d. LAST = Newnode
   e. Exit.
4. Else
   i. Newnode->data = Item
   ii. Last->next = Newnode
   iii. Last = Newnode
   iv. Last->next = START
5. Stop.

Deleting a node from beginning (Delete_First(START)

1. Start.
2. Declare a temporary node, ptr
3. If START is equal to NULL, then
   i. Display empty Circular queue
   ii. Exit
4. Else
   1. Ptr = START
   2. START = START->next
   3. Display, Element deleted is , ptr->data
   4. Last->next = START
   5. Free (ptr)
5. Stop.

Deleting a Node from End (Delete_End(START)

1. Start.
2. Declare a temporary node , ptr
3. If START is equal to NULL , then
   i. Print Circular list empty
   ii. Exit
4. ELSE
   i. Ptr = START
   ii. Repeat iii and iv while(ptr !=Last)
   iii. Ptr1 = ptr
   iv. Ptr = ptr->next
   v. Print element deleted is ptr->data
   vi. Ptr1->next = ptr->next
   vii. Last = ptr1
   viii. Free(ptr)
5. Stop.

Generalized Linked List

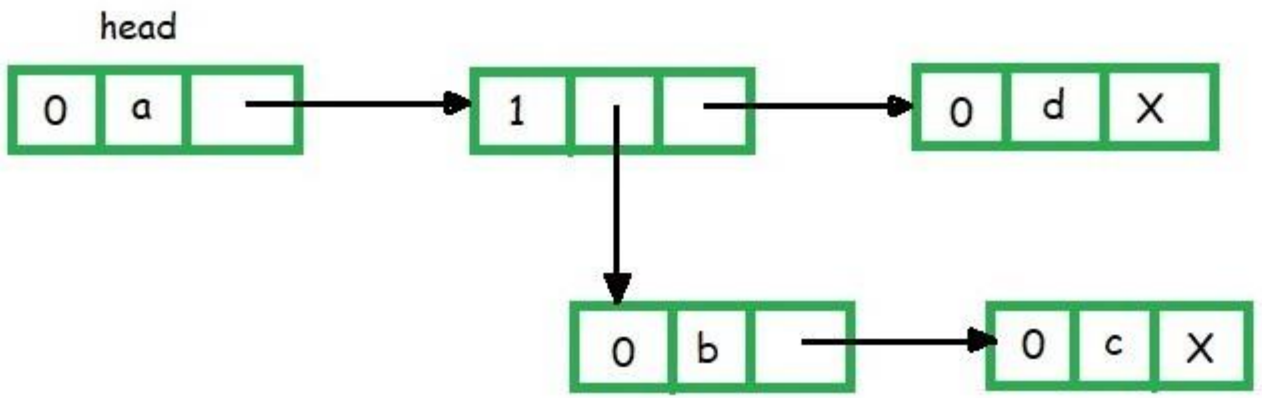
Why Generalized Linked List?

- Generalized linked lists are used because although the efficiency of polynomial operations using linked list is good but still, the disadvantage is that the linked list is unable to use *multiple variable polynomial equation* efficiently.
- It helps us to represent multi-variable polynomial along with the list of elements.

```
typedef struct node {
    char c;                          //Data
    int index;                       //Flag
    struct node *next, *down;         //Next & Down pointer
}GLL;
```

Generalized Linked List

Example of GLL {List representation} *(a, (b, c), d)*



- When first field is 0, it indicates that the second field is variable.
- If first field is 1 means the second field is a down pointer, means some list is starting.

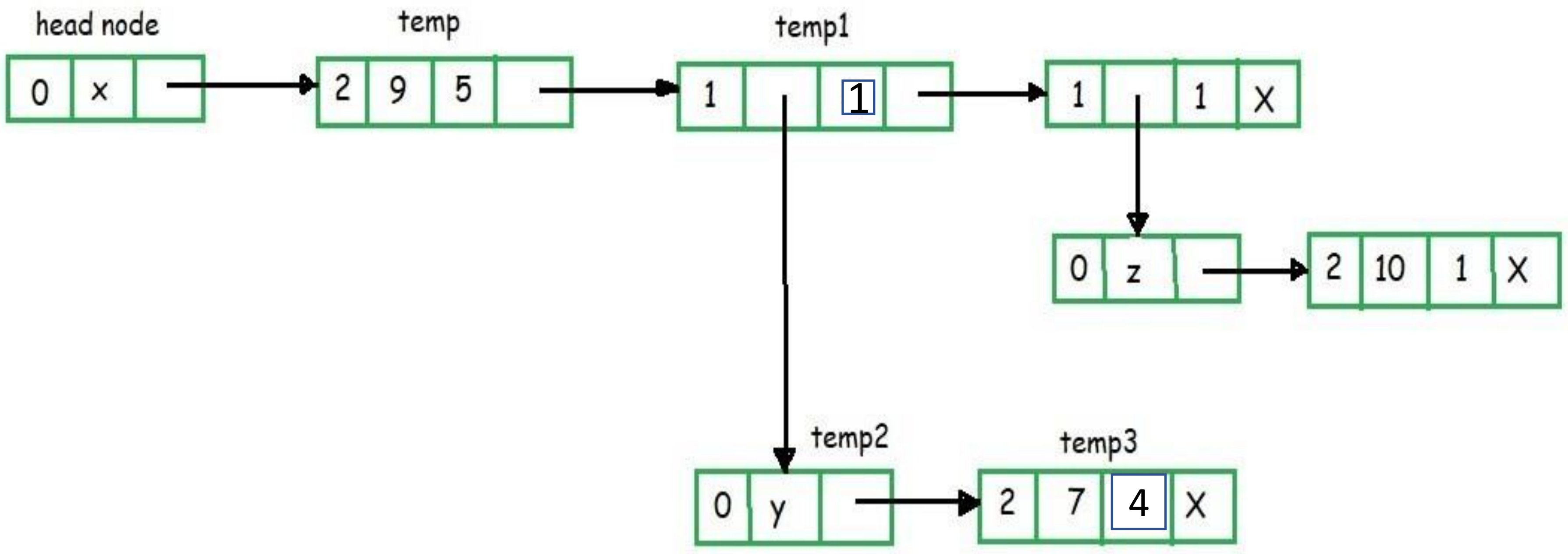
Polynomial Representation using Generalized Linked List

The typical node structure will be:

- Here Flag = 0 means *variable* is present
- Flag = 1 means *down pointer* is present
- Flag = 2 means *coefficient* and *exponent* is present

Generalized Linked List

Example: $9x^5 + 7xy^4 + 10xz$



Generalized Linked List

- For example, the abstract list represented by following figure may be represented as: list = (8,16,’g’, 99,’b’)

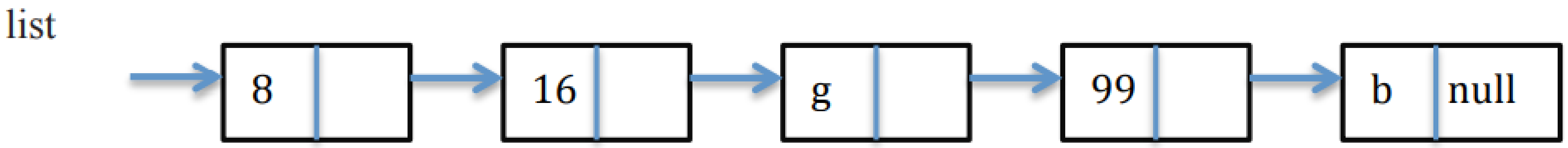


Fig35 (a): Generalized Linked List

- The null list is denoted by an empty parenthesis pair such as ( ).

Generalized Linked List

❖Thus the list of the following figure is represented as:

list = (8,(5,7,(18,3,6),( ),5),2,(6,8))

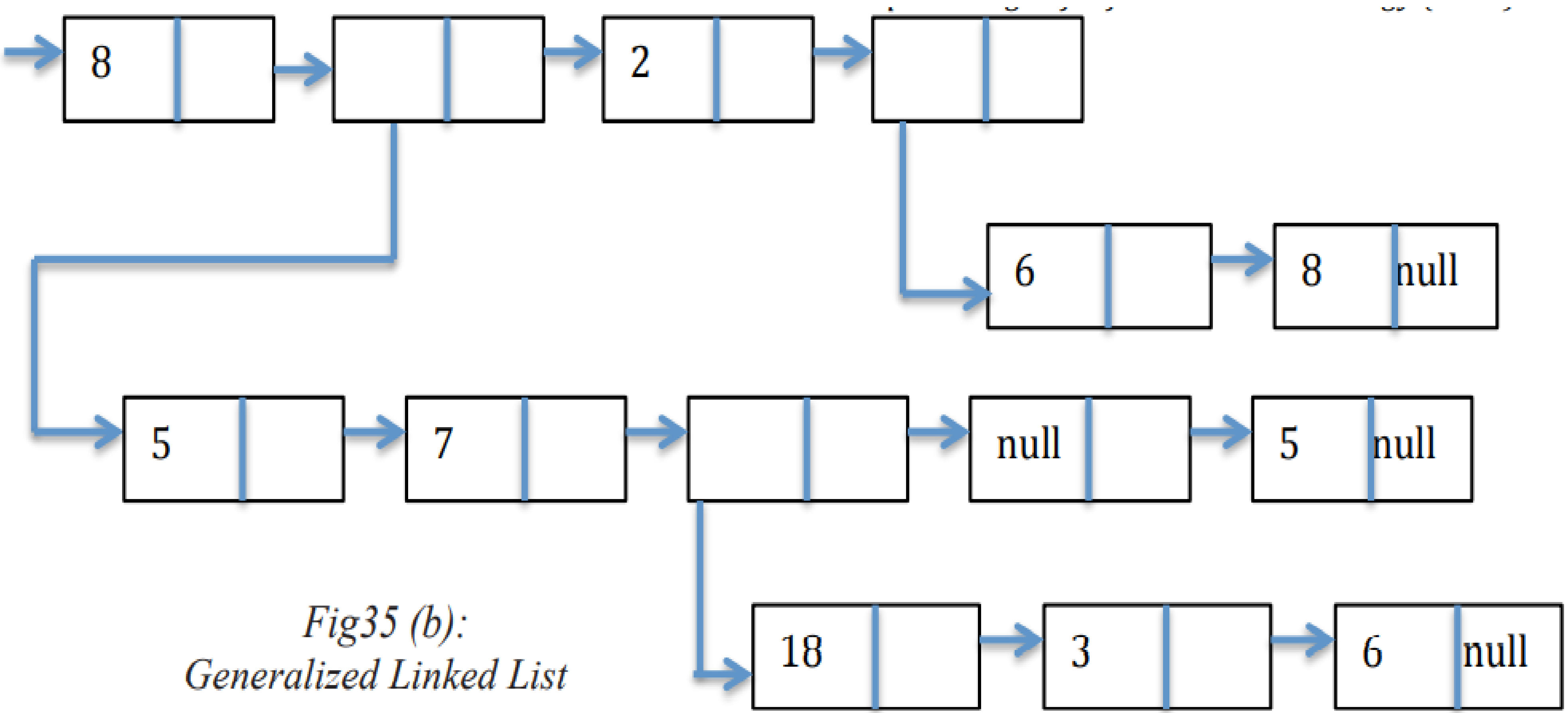


Fig35 (b):
Generalized Linked List

Polynomials using singly linked list (Application of Linked List)

- Different operations, such as addition, subtraction, division and multiplication of polynomials can be performed using linked list.
- Following example shows that the polynomial addition using linked list.
- In the linked representation of polynomials, each term is considered as a node.
- And such a node contains three fields: coefficient field, exponent field and Link field.

```
Struct polynode{
        int coeff;
        int expo;
        struct polynode *next;
};
```

Coefficent	Degree	Address of next node

Polynomials using singly linked list (Application of Linked List)

- Consider two polynomials f(x) and g(x); it can be represented using linked list as follows.

$$f(x) = ax^3 + bx + c$$
$$g(x) = mx^4 + nx^3 + ox^2 + px + q$$

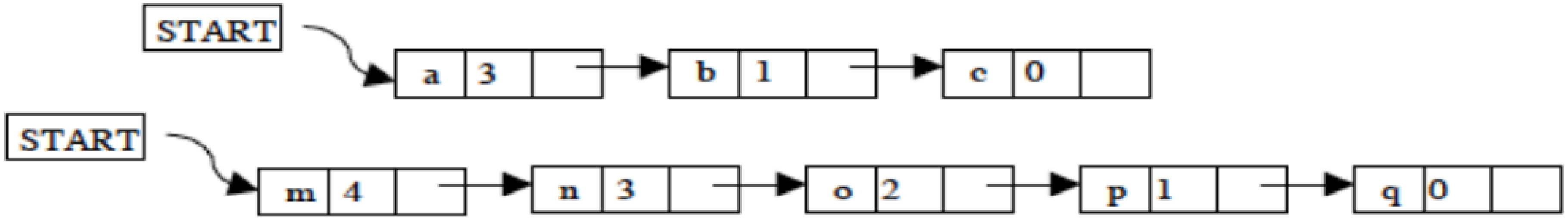


Fig33: Polynomial Representation in Linked List

These two polynomials can be added by h(x) = f(x) + g(x) = mx^4 + (a + n) x^3 + ox^2 + (b + p)x + (c + q) i.e.; adding the constants of the corresponding polynomials of the same exponentials. h(x) can be represented as

Fig34: Addition of polynomials

Thank you!!!!!