UNIT - III

LINKED LISTS

Syllabus :-

Linked Lists : Introduction, Representation of linked list in memory, single linked list, operations on single linked list, Reversing a single linked list, Applications of single linked list to represent polynomial expressions and sparses matrix manipulation, Advantages and Disadvantages of single linked list, circular linked List, Double linked list.

Assignment Questions :-

1) What is single linked list, write an algorithm to insert and delete a node in single linked list.?

2) What are the advantages & Disadvantages of single linked list

3) what are the differences between arrays & linked lists ?

4) write an algorithm for reversing single linked list elements ?

5) Explain applications of single linked list ?

6) what is double linked list ? write an algorithm for insert, delete and display the nodes in list ?

7) what is circular linked list ? Explain its operation?

Linked list :-

Linked list is a linear collection of data elements. these elements are called nodes. For each node we are having two fields

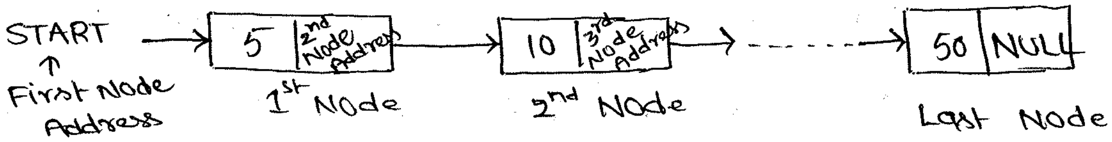
i) Data Field

ii) Address Field.

→ Data field used to store the element (information)

→ Address field used to store the address of next node (element). So it is a Pointer to store address.

- The last node not having next node, so the address field of last node is NULL.
- START-pointer, it is stores the first node address in the list.
- We can traverse entaire list by using START. To find the second node address we have the address in the first node.



- Using this technique the individual nodes of a list will form chain of nodes.
- If START = NULL, then the list is empty list.
- In order to form a linked list, we need a structure called node. which has two fields data and next.

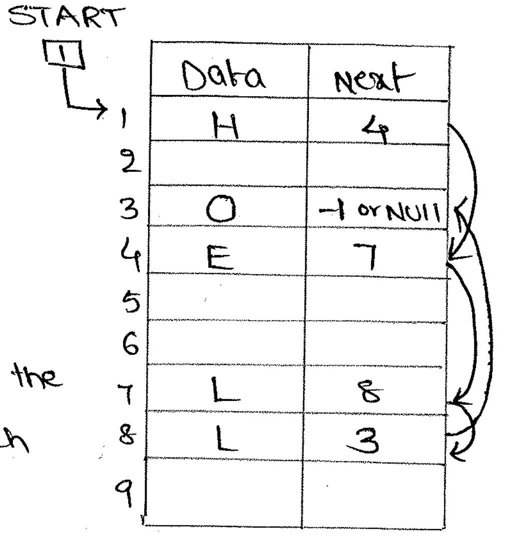
data → stores the information part

Next → Address of the next node.

```
struct node
  {
      int data;
      Struct node *next;
  };
```

Memory representation of linked list :-

- START is used to stores the address of the first node.
- In this example, Start = 1, So the first node stores at address 1, which element is H.
- The corresponding Next stores the address of the next node, which is 4.



	Data	Next
1	H	4
2		
3	O	-1 or NUll
4	E	7
5		
6		
7	L	8
8	L	3
9		

- So, we will look at address 4 to fetch the next data item.
- The second data element is obtained from address 4 is E, again wee see the corresponding Next to go to the next node.
- We repeat this procedure until we reach a position where the Next field contains -1 or NUll, then we denote last element in the list
- Remember that the nodes of a linked list need not be consecutive memory locations, here we are stores the elements at 1, 4, 7, 8, 3.

Memory Allocation & De-allocation :-

- If we want to add new node into already existing list in the memory, we first find free space ~~is~~ of memory then stores the information.
- computer will maintain a list of all free memory cells. The list of available space is called freepool.
- For pointing free space in memory we have a pointer called AVAIL. It stores the address of first free space in memory.

- After inserting new node into the list, the next available free space is Pointing the AVAIL.

- Deleting a node from the list, the space occupies by node is given back to free pool So the memory can reuse.

START 1

AVAIL 2

	Data	Next
1	H	4
2		5
3		-1
4	E	6
5		8
6	L	7
7	L	9
8		3
9	O	-1

- collecting all remaining space into free pools, this process called Garbage collection.

Dynamic Memory :-

- We are having 4 memory management functions, called calloc(), malloc(), realloc(), and free()
- All the functions are available in "stdlib.h"

1. malloc() :- Allocate required size of bytes and returns a pointer first byte of allocated space.

```
Variable = (Datatype *) malloc (sizeof (datatype));
Ex:-    Ptr = (int *) malloc (sizeof (int));
```

2. Calloc () :- Allocate space for array elements, initialize to zero and return a pointer to memory

```
Ex:-    variable = (Datatype *) calloc (N, sizeof (datatype));
        ptr = (int *) calloc (20, sizeof (int));
```

3. realloc () :- change the size of previously allocated space

```
variable = realloc (variable, new size);
```

4. free () :- Deallocate the previously allocated memory space

```
free (variable);

Ex:- free (ptr);
```

Arrays vs Linked list :-

1. Both are linear collection of data elements

2. → Arrays will allocate the memory in sequential order.
   → Linked list will allocate the memory for elements in random

3. → In Arrays insertion/Deletion is very difficult because if you delete first element, shifting all elements to previous locations
   → In Linked list insertion/Deletion can perform at any point, just by changing the next field of a node we can perform operations.

4. → In arrays we can add fixed size no.of elements
   → In Linked list we can add any number of elements.

5. → In arrays memory allocation at compile time, some times memory space wastage.
   → In Linked list memory allocation at Run-Time, by using dynamic memory allocation functions we can perform.

Single linked list :-

- Single linked list is the simple type of linked list in which every node contains some data and a pointer to the next node of the same data type.
- Traversal of linked list is only one way, from START to end of Node.
- Operations of single linked list

1. Traversing
2. Searching
3. Insertion
4. Deletion.

1. Traversing :-

- Traversing of linked list means accessing the nodes of the list inorder to perform some operations.
- Linked list contains the pointer START, which stores the address of the first node in the list.
- For the last node the next field address is NULL.
- We are taking one pointer PTR for accessing the nodes.

```
Algorithm Traversal ( )
{
    [Initialize]  set PTR := START;
    Repeat steps while  PTR != NULL
            Apply process  PTR → data ;
            Set  PTR = PTR → next ;
    End loop;
}
```

- For counting number of nodes in a list

```
Algorithm  countnodes( )
{
    [Initialize]   Set count := 0;
    [Initialize]   set PTR := START;
    Repeat Steps while  PTR != NULL
            set  count := Count +1 ;
            set  PTR := PTR → next ;
    End loop
    write  count ;
}
```

2. Searching :-

- Searching a list means to find particular element presented in the linked list or not
- There are two outcomes for searching, one is node address and Other is NULL
- The given Key element is presented in the list then it will return the node address, if it is not presented then it will return NULL.

```
Algorithm Search (item)
{
    [Initialize]   Set POS:= NULL;
    [Initialize]   Set PTR:: START;
    Repeat while   PTR != NULL
            if item == PTR → data then
                  Set POS := PTR;
            else
                  Set PTR:= PTR → next;
    return (POS);
}
```

3. Insertion :-

- If the list is already containing the nodes then we can insert a new node in following ways

1) At beginning of the list

2) At End of the list

3) At particular position in the list.

- If START = NULL then the list is empty
- If AVAIL = NULL then no free memory cells in the system.

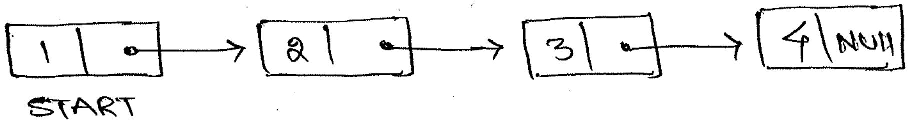
(i) Beginning of the list :-

- For inserting new node in to the list first check memory space is available or not
- If the memory is available (AVAIL != NULL) then create new node and AVAIL pointing to the next free space
- Now, insert the values for node, Directly insert data item into data field and in the Next field we can store the first node address
- Now, New node is the first node in the list so store the new node address into START.
- Finally, we are inserted new node at beginning of list.

```
Algorithm insert_beg (item)
{
        if AVAIL := NULL
                Write "No memory for creation";
                Go to Exit;
  New_node    { Set New_node := AVAIL;
  creation    { Set AVAIL := AVAIL -> next;
  values      { Set New_node -> data := item;
  insertion   { Set New_node -> next := START;
  into New_node
                Set START := New_node;
}
```

Example:

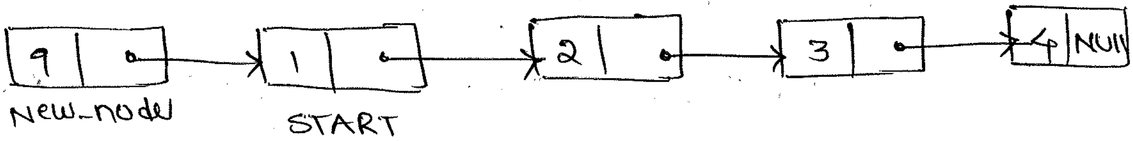
- Add new node containing data 9 into list.



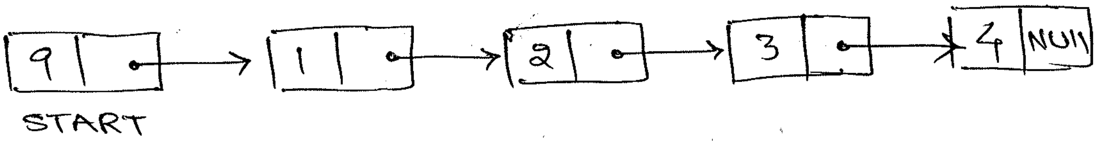
- Allocate memory for new_node containg data 9

New node

- Add new_node at beginning of the list by taking the next part of new_node containing address of START



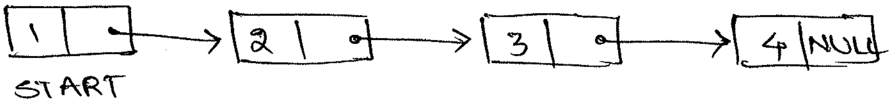
- Now make START to point to the first node of list.



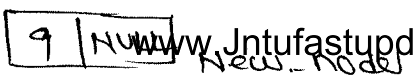
(ii) <u>Insert at End of list</u> :-

- Initially we are taking one pointer PTR for accessing the Nodes in the list and it is initialized with START.
- check free memory space is available for new_node.
- insert the field value of new_node, data part we can directly insert item and next part is NULL because the new_node is the last node in the list.
- Now, we are moving to the last node in the list by using PTR
- Add new node address to the next field of the last node then we are creating link between new node and previous last node
- Finally we are inserted new node at the end of list.

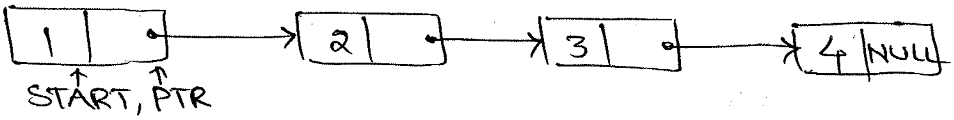
Example :- Add new node containing '9' at end of list.



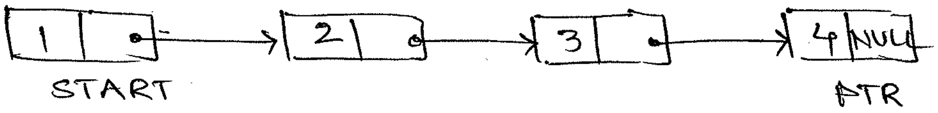
- Allocate memory for new_node containg data 9 and next address NULL



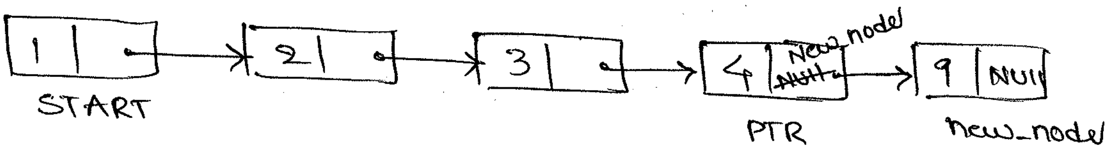
- Take a pointer variable PTR with value of START.

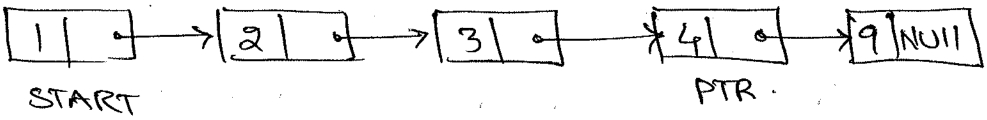


- move PTR to end of list



- Add new node at end of the list and change the Next field of PTR node by assigning new_node address





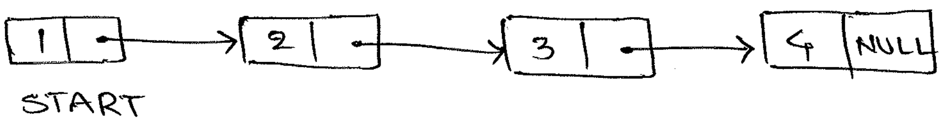
```
Algorithm insert_end (item)
{
    if AVAIL = NULL
        write "No memory for new node";
        Go to Exit;
    Set New_node := AVAIL;
    Set AVAIL := AVAIL → next;
    Set New_node → data := item;
    Set New_node → next := NULL;
    Set PTR := START;
    Repeat steps while PTR → next != NULL do
            Set PTR := PTR → next;
    End loop
    Set PTR → next := New_node;
}
```

(iii) Insert at particular position :-

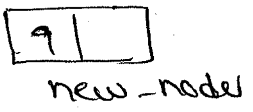
- Inserting new_node into the list first check memory is available or not, if it is available create new_node.
- Initially taking one pointer PTR assigned by START, then it is pointing to the first node
- Now we are moving from one node to other, upto given pos value.
- Now we are inserting the field value of new_node, data part by item and next field by Ptr → next.
- ANd now change the ptr next field address by new_node.
- Finally we are inserting new_node at given particular pos location.

```
Algorithm insert_pos( pos, item)
{
        if AVAIL = NULL
            Write "No memory for new_node";
            Goto Exit;
        new_node := AVAIL;
        AVAIL := AVAIL → Next;
        PTR := START;
        i := 1;
        While  i < POS - 1
                PTR = PTR → Next;
                i++;
        new_node → data := item;
        new_node → Next := PTR → Next;
        PTR → Next := new_node;
}
```

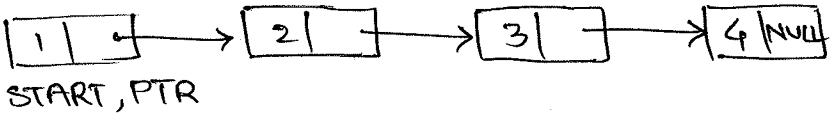
Example :- Add new node containing '9' at particular position.



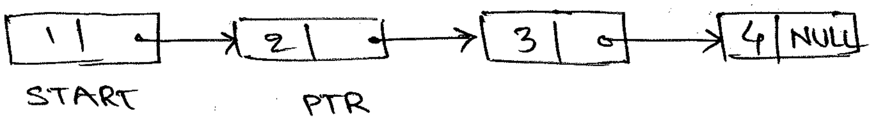
- Allocate memory for new_node containing data 9.



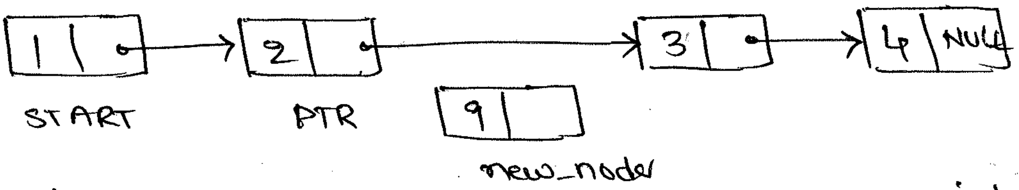
- Take a pointer variable PTR with value of START



- Move PTR to given POS = 3 ⇒ pos-1 position

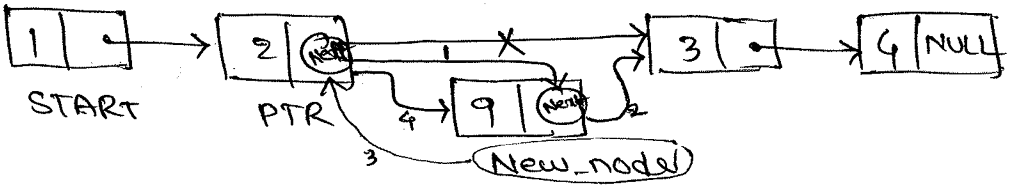


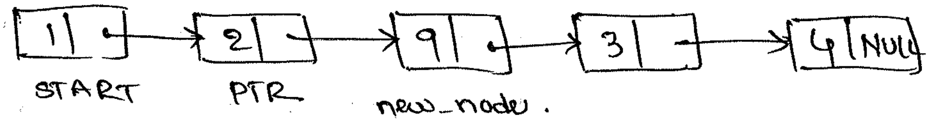
- Add new_node into the list after PTR node.



- Now copy the PTR → Next address and assigned into new_node → next, we are creating link between new_node and next node containing data 3

- Add change PTR next address, it points to new_node then finally we are inserting new_node at given position.





4. Deletion :-

- If we are already containing more than one node in the list, then we can perform different type of deletion operations on list

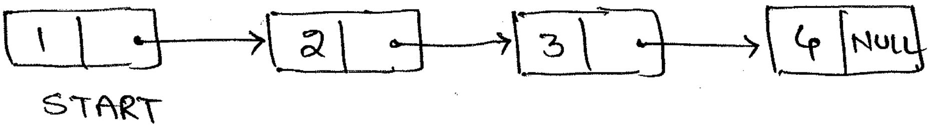
  (i) At Beginning of the list
  (ii) At End of the list
  (iii) At particular position of a node in list

(i) At Beginning of the list :-

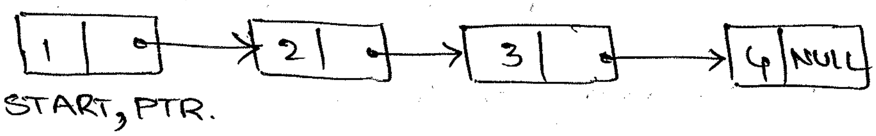
- First, we need to check wether the list is having the nodes or not, if nodes are presented in the list then only we can delete the node from the list otherwise not possible.
- We are taking a pointer variable PTR assigned by START.
- Now, we are changing the START position, because after deleting first node from the list second node is the starting node in the list
- Now, we can delete the PTR node from the list
- Finally we are deleting beginning node from the list

```
Algorithm del_beg ( )
{
    if START = NULL
        Write "No nodes in the list";
        Goto Exit;
    PTR := START;
    START := START → Next;
    free (Ptr);
}
```

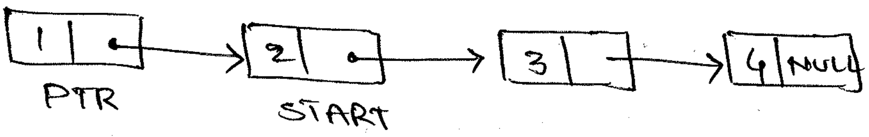
Example :- Deleting first node from the list containg data '1'.



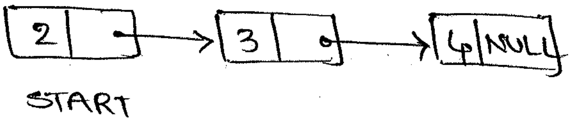
- Take one PTR variable containing START address, then it is pointing the first nodes.



- Now, change the START position to the next node by START → next, then it is pointing second nodes



- Now, delete the PTR node from the list then the final list is



- Finally we are deleting beginning node from the given list.

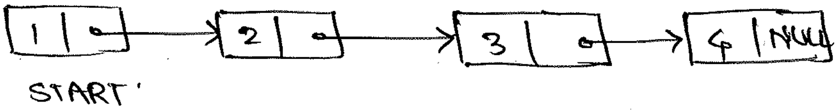
(ii) Deleting node at End of list :-

- First, we need to check wether the list is having the nodes or not, if nodes are presented in the list then only we can delete the node from the list otherwise it is not possible.
- Now, we are taking a PTR variable assining START address
- Now move PTR, pointing the last node in the list mean while we are taking PREPTR pointing the previous of last nodes.
- Now change the address of PREPTR to NULL
- Delete the PTR node from the list.

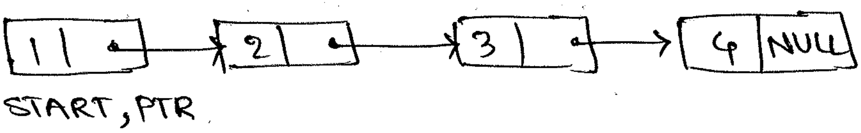
- Finally we are deleting the last node from list.

```
Algorithm del_end ( )
{
        if START = NULL
                Write " No nodes in the list ";
                Goto Exit ;
        PTR := START
        While PTR → Next != NULL do
                PREPTR := PTR ;
                PTR := PTR → Next ;
        PREPTR → Next := NULL;
        free ( PTR ) ;
}
```

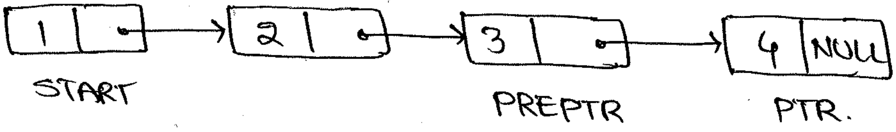
Example :- Delete the last node from list containing data '4'



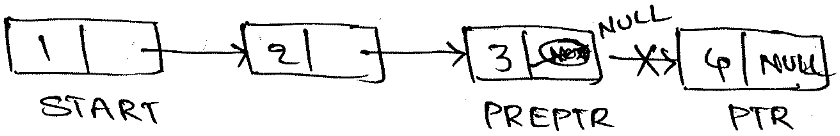
- Take a pointer variable PTR and it points to first Node.

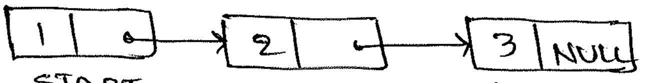


- Move PTR from starting to end of the list meanwhile take PREPTR pointing to the previous of PTR node.



- Now change the next value of PREPTR then we are breaking link between PREPTR and PTR and delete the PTR node from list



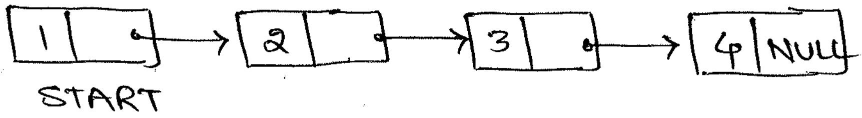


(iii) Deleting particular nodes from list :-

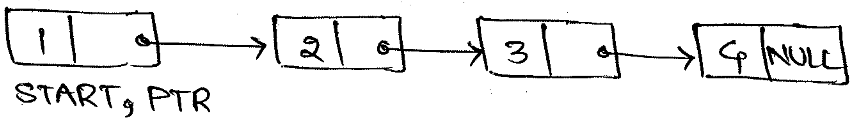
- First check the list containing nodes or not, if the nodes are presented then only we can delete the node from the list otherwise it is not possible
- Initally takes PTR variable containg START address.
- Now more PTR upto given POS value meanwhile take PREPTR, pointing previous of PTR nodes.
- Now change the address of PREPTR → Next by assigining a value of PTR → Next, now we are creating link between PREPTR and next nodes of PTR.
- Now we can delete PTR nodes from the list, finally we are deleting particular node from the list.

```
Algorithm del_pos (pos)
{
    if START = NULL
        Write " No nodes in the list";
        Goto Exit;
    PTR := START;
    i := 1;
    While i < POS do
        PREPTR := PTR;
        PTR := PTR → Next;
    PREPTR → Next := PTR → Next;
    free (PTR);
}
```

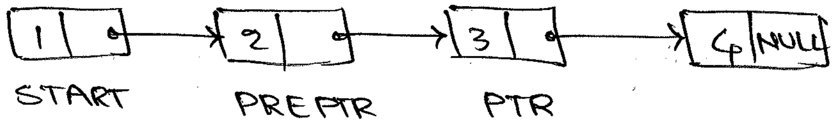
Example :- Delete a node at POS = 3 from list.



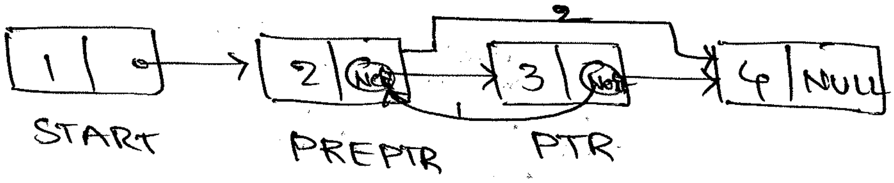
- Take PTR variable pointing to first node.



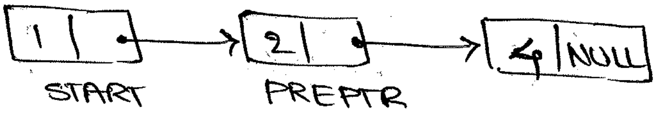
- Move PTR to POS = 3 location meanwhile take PREPTR pointing previous of PTR node.



- change the PREPTR next field address by PTR next field address then we are pointing or linking PREPTR node and next node of PTR



- Now delete PTR from the list then the final list as follows.



- Finally we are deleting particular node from the list.

Reversing A Single linked list :-

- First create a single linked list having nodes.
- taking two pointer variables PTR1 = NULL and PTR2
- Repeat the following process upto the last node
  1). Take first two nodes from the list and PTR2 is pointing to the second Node.

2) change the first node next field address assign the PTR1 value.

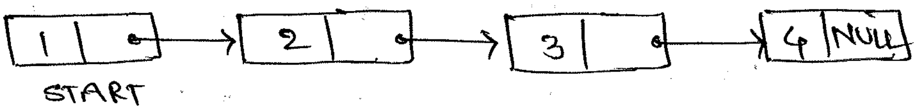
3) Now pointing the first node as PTR1

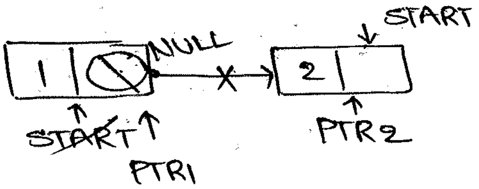
4) change the START, pointing to PTR2 because PTR2 is the starting node if the list is in reverse order.

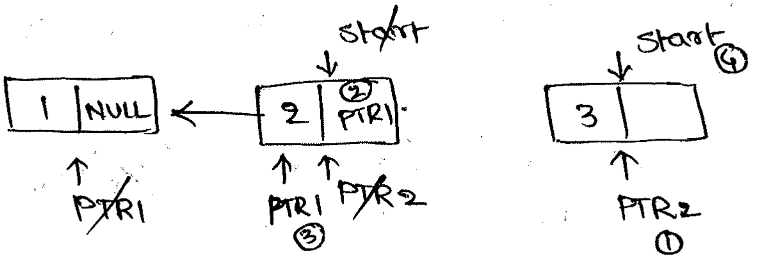
- After getting or moving to the last node change the next field of first node after reversing by PTR1

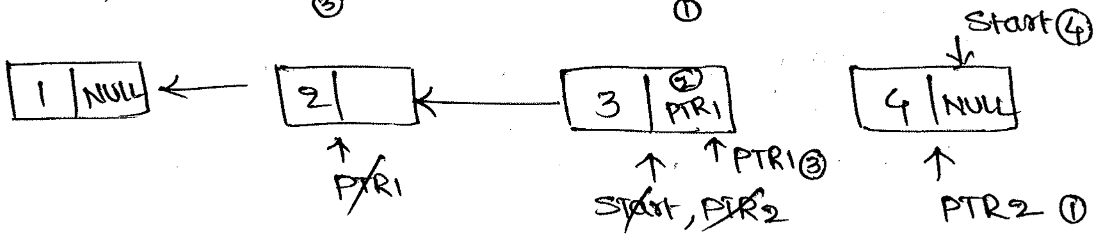
- Finally we are reversing the given single linked list.

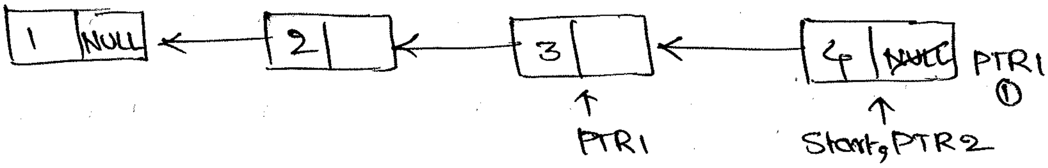
Example :-



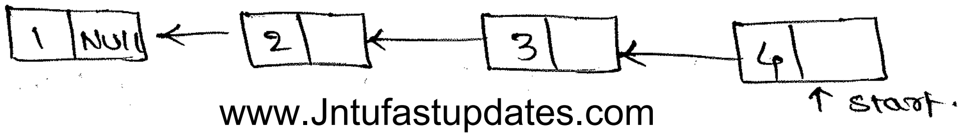
- 

- 

- 

- 

- Finally the reverse list is



Algorithm :-

```
Algorithm reverse ( )
{
    PTR1 := NULL , PTR2 ;
    While START do
    {
        PTR2 := START → Next ;
        START → Next := PTR1 ;
        PTR1 := START ;
        START := PTR2 ;
    }
    START → Next := PTR1 ;
}
```

Advantages & Disadvantages of Single linked list :-

Advantages :-

- 1) Insertions and Deletions can be done easily.
2) It doesn't need movement of elements for insertion and deletion
3) Size is not fixed so there is no space wastage
4) we can increase the sizes of the list according to our requirement.
5) Elements may or may not be stored in consecutive memory locations, even though we can store the data in computer
6) It is less expensive.

Disadvantages:-

1) It requires more space because pointers are also stores the information.

2) Different amount of time is required to access the elements in the list.

3) We can't traverse from last, only traverse from beginning.

4) It is not easy to sort the elements stored in the linear linked list.

Applications of linked list :-

- We are having two types of applications

1) Polynomial Representation
2) Sparse Matrix Manipulation.

1) Polynomial Representation :-

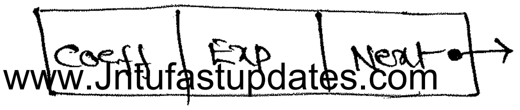
- Polynomials are the expressions containing number of terms with non-zero coefficients and exponents

$$P(x) = a_0 + a_1x^1 + a_2x^2 + ---- + a_{n-1}x^{n-1} + a_nx^n$$

where a_i is non-zero coefficient
n is non-negative integer.

- In the linked representation of Polynomials, each term is considered as node and the node containing 3 fields

1) coefficient field
2) Exponent field
3) Next node address field

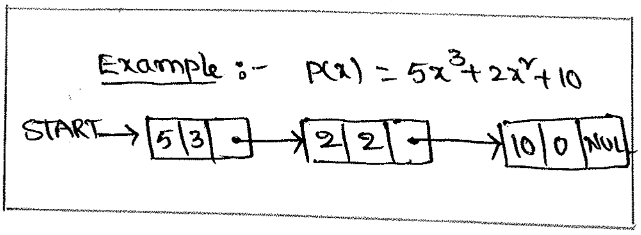


- The coefficient field holds the value of coefficient of a term and the exponent field contains the exponent value of the term.
- Next field contains the address of the next term in the polynomial.
- In C, the structure of polynomial nodes is

```
Struct Polynode
{
    int coeff;
    int exp;
    Struct Polynodes *Next;
};
```

- Algorithm for creation of polynomial equation in linkedlist

```
Algorithm create_poly (  )
{
    Read c, e;
    While coeff != 0 do
    {
        if START == NULL then
        {
            new_node := AVAIL;
            AVAIL := AVAIL → Next;
            new_nodes → coeff := c;
            new_nodes → exp := e;
            new_node → Next := NULL;
        }
        else
        {
            PTR := START;
            While PTR → Next != NULL do
                PTR := PTR → Next;
            new_node := AVAIL;
            AVAIL := AVAIL → Next;
            new_node → coeff := C ;
            new_nodes → exp := e ;
            new_nodes → Next := NULL ;
            PTR → Next := new_nodes ;
        }
        Write "Enter the coefficient and Exponent values" ;
        Read c,e ;
    } // closing of while
} // closer of function.
```


Example :- $P(x) = 5x^3+2x^2+10$

Operations of polynomial :-

- We are having 4 types of polynomial operations, those are

1) Evaluating polynomial at given value
2) Addition of two polynomials
3) Subtraction of two polynomials
4) Multiply two polynomials

Addition of two polynomials :-

- Initially take two polynomials P & Q and Resultant R
- We have to compare their starting terms from first node and moving towards end one by one
- Three PTR variables to represent their lists PPTR, QPTR and RPTR respective P, Q and R.
- there are 3 cases during the comparison between the terms of polynomials

1) The exponent of two terms are equal, the coefficients of two terms are added and a new term is created with the value

RPTR → coeff := PPTR → coeff + QPTR → coeff

and

RPTR → Exp := PPTR → Exp ;

2) If the exponent of P is greater than the exponent of Q then the duplicate of current term P is created and inserted in polynomial R

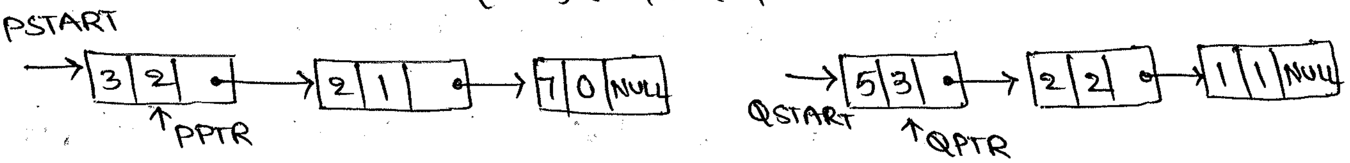
3) If the exponent of P is smaller than the exponent of Q then the duplicate of current term Q is created and inserted in polynomial R

- Append the remaining terms in the either polynomials to the resultant polynomial R.

Example :- Take two polynomials P, Q

Let $P = 3x^2 + 2x + 7$

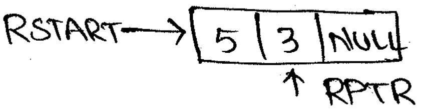
$Q = 5x^3 + 2x^2 + x$

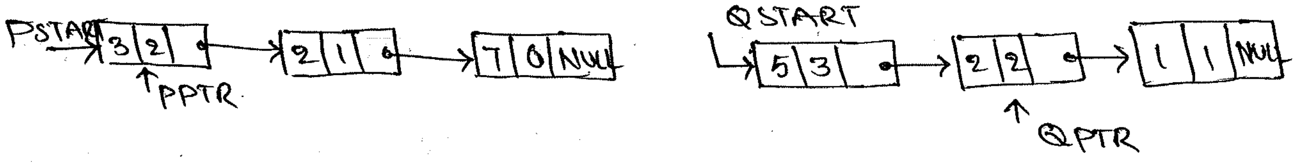


- Compare the exponents of P and Q

$Exp(P) < Exp(Q) \Rightarrow PPTR \to EXP < QPTR \to EXP$
$2 < 3$

- Add the Q term into the resultant polynomial R and move to next node.



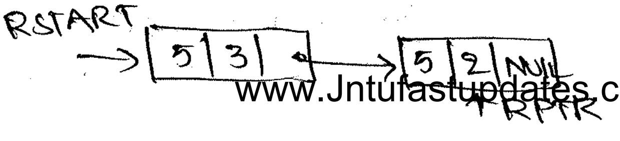


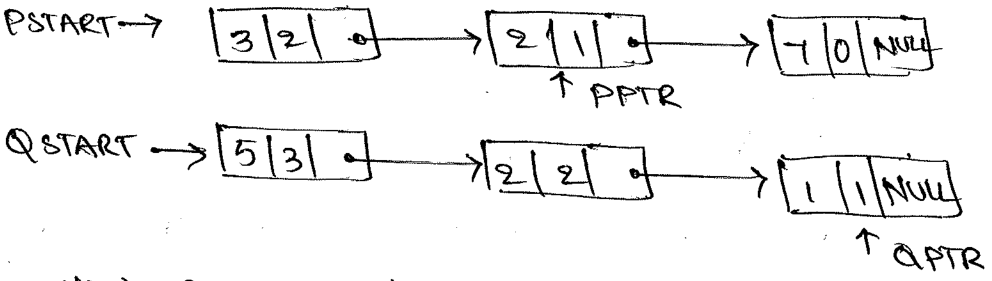
- Compare the exponents of P and Q

$Exp(P) = Exp(Q) \Rightarrow 2 = 2$ then

Add two coefficients $3 + 2 = 5$

- Now the resultant polynomial R is



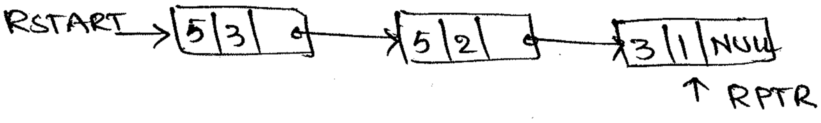


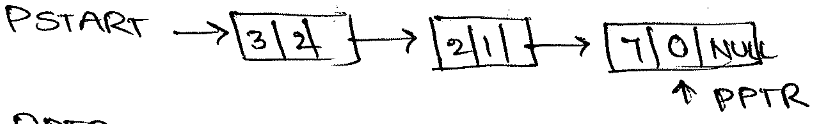
- Now compare the exponents of two polynomials

  exp(P) = exp(Q) => 1 = 1 then

  add the two coefficients 2+1 = 3

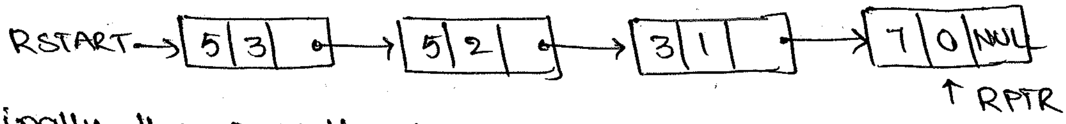
- Now the new node $\boxed{3|1|\ }$ added to resultant R and move PPTR and QPTR to the next nodes





- QPTR = NULL

  So, directly append the P remaining terms into the resutant R Polynomial.



- Finally the resultant polynomial R is

  $R(x) = 5x^3 + 5x^2 + 3x + 7.$

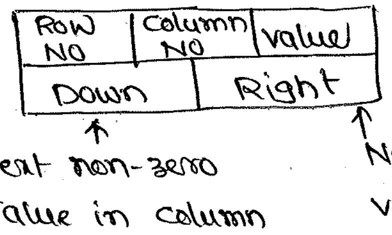
- Algorithm for addition of two polynomials

```
Algorithm Add_Poly ( )
{
    PPTR := PSTART , QPTR := QSTART , RPTR := RSTART;
    While PPTR != NULL and QPTR != NULL do
    {
        if PPTR → Exp = QPTR → Exp then
        {
            new_node := AVAIL ;
            AVAIL := AVAIL → Next ;

            RPTR := new_node;
        RPTR → coeff := PPTR → coeff + QPTR → coeff ;
        RPTR → Exp := PPTR → Exp ;
        RPTR → Next := NULL ;
        PPTR := PPTR → Next ;
        QPTR := QPTR → Next ;
      }
  if  PPTR → Exp > QPTR → Exp  then
    {
        new_node := AVAIL ;
        AVAIL := AVAIL → Next ;
        RPTR := new_node ;
        RPTR → coeff := PPTR → coeff ;
        RPTR → Exp := PPTR → Exp ;
        RPTR → Next := NULL ;
        PPTR ⇒ PPTR → Next ;
    }
  if  PPTR → Exp < QPTR → Exp  then
    {
        new_node := AVAIL ;
        AVAIL := AVAIL → Next ;
        RPTR := New_node ;
        RPTR → coeff := QPTR → coeff ;
        RPTR → Exp := QPTR → Exp ;
        RPTR → Next := NULL ;
        QPTR := QPTR → Next ;
    }
  } // End of while loop

  while  PPTR != NULL  do
    {
        new_node := AVAIL ;
        AVAIL := AVAIL → Next ;
        RPTR := New_node ;
        RPTR → coeff := PPTR → coeff ;
        RPTR → Exp := PPTR → Exp ;
        RPTR → Next := NULL ;
        PPTR := PPTR → Next ;
    }
```

```
while  QPTR != NULL  do
   {
      new_node : = AVAIL ;
      AVAIL := AVAIL → Next ;
      RPTR : = new_node ;
      RPTR → coeff  := QPTR → coeff ;
      RPTR → Exp := QPTR → Exp ;
      RPTR → Next := NULL ;
      QPTR := QPTR → Next ;
   }
} || End of function.
```

(2) Sparse Matrix Manipulation :-

- Sparse matrices are those matrices which have majority of their elements equal to zero.
- The Node representation of sparse matrix is



- In C, the structure of sparse matrix is

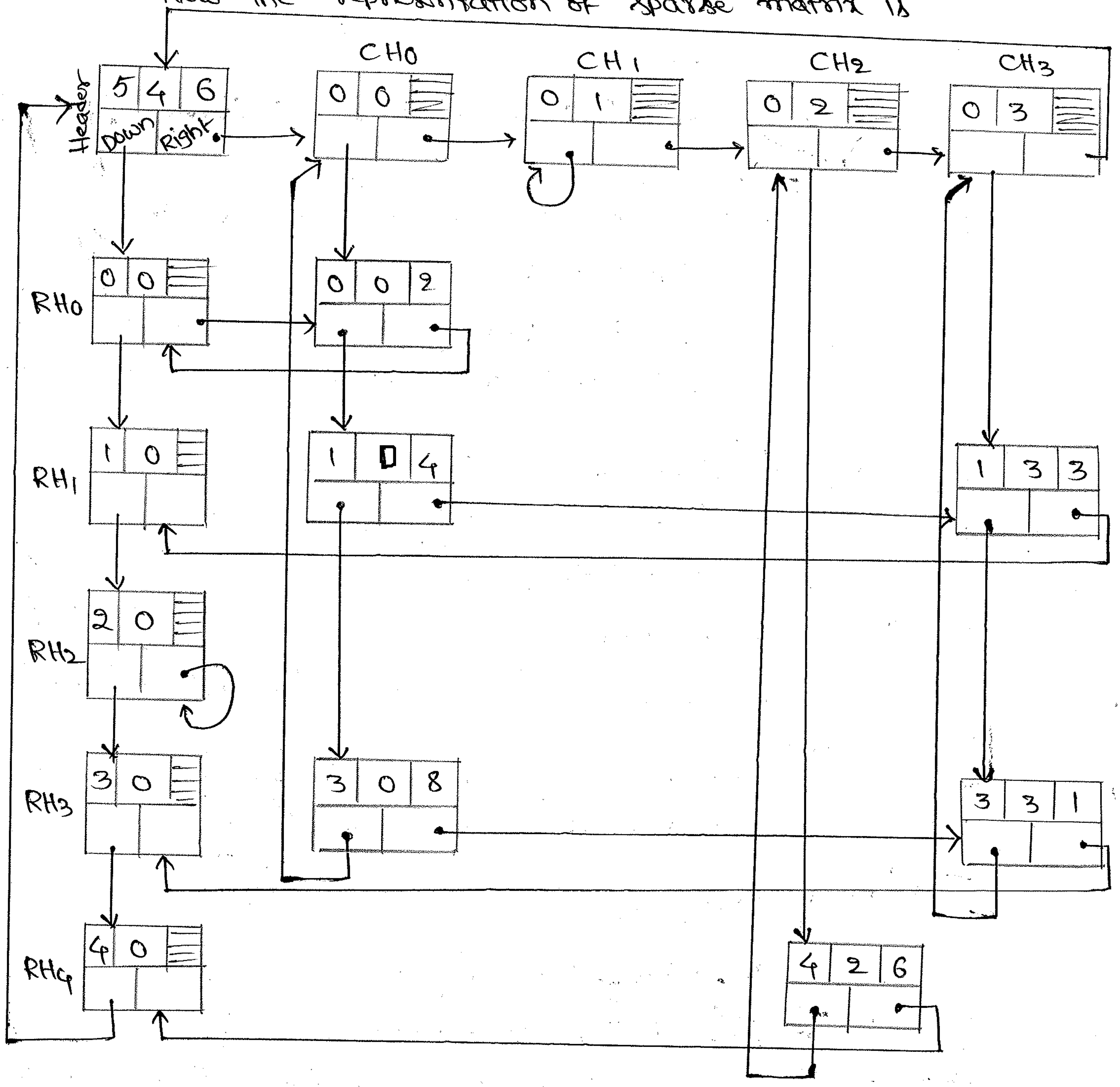
```
struct sparse_node
   {
      int row, column, value;
      struct sparse_node *down, *right ;
   }
```

- In dummy header we are mainting no.of rows and no. of columns and also maintain no. of non-zero elements in the matrix.

Example :- Let the matrix is

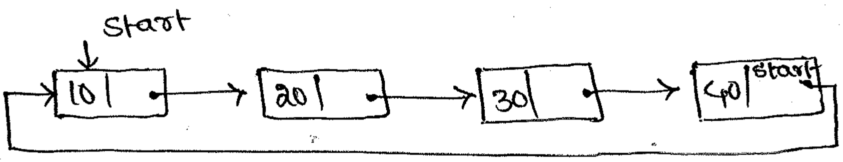
$$\begin{bmatrix} 2 & 0 & 0 & 0 \\ 4 & 0 & 0 & 3 \\ 0 & 0 & 0 & 0 \\ 8 & 0 & 0 & 1 \\ 0 & 0 & 6 & 0 \end{bmatrix}$$

- Let the row headers are represented as RH0, RH1, RH2 --- and the column headers are represented as CH0, CH1, CH2 ----
- Now the representation of sparse matrix is



Circular linked List :-

- A Linked list where the last node points the starting node is called the circular linked list.
- There is no beginning and ending of list.



- In C, the structure of circular linked list is

```
Struct node
  {  int data;
     struct node *Next;
  };
```

- The operations of circular linked list is
  1) Insertion of a node
  2) Deletion of a node.

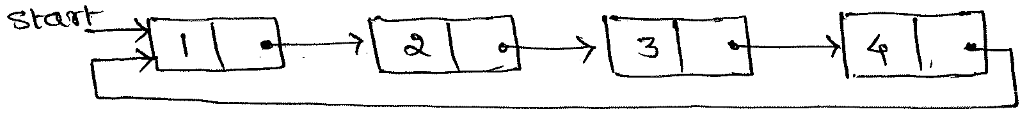
1) Insertion of a node :-

- We can insert a new node in a circular linked list in 2 ways
  (i) Inserting a node at beginning
  (ii) Inserting a node at ending

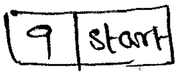
(i) At beginning :-

```
Algorithm insert_beg(item)
  {  if AVAIL = NULL
         Write "No memory for creating a node";
         Goto Exit;
     new_node := AVAIL;
     AVAIL := AVAIL → Next;
     new_node → data := item;
     new_node → Next := Start;
     Ptr := Start;
     While ptr → Next != Start do
          Set Ptr := Ptr → Next;
Ptr → Next := new_node;
Start := new_node;
}
```

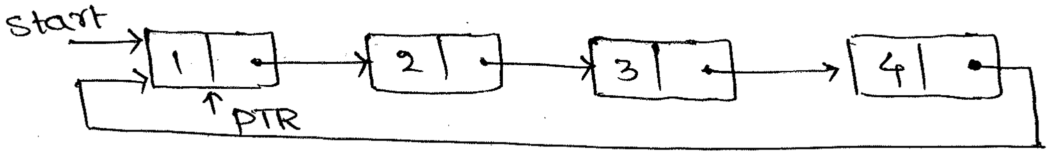
Example :- Inserting a newnode of data '9' into circular linked list

- 

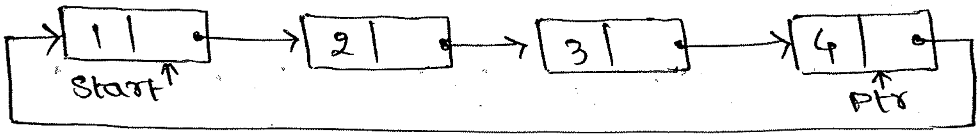
- Allocate memory for the new node and initialize its Data part 9 and Next by start



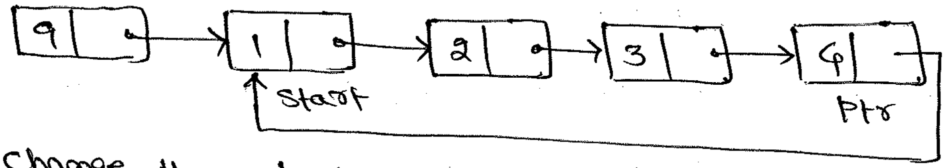
- Take a pointer variable PTR that points to START node



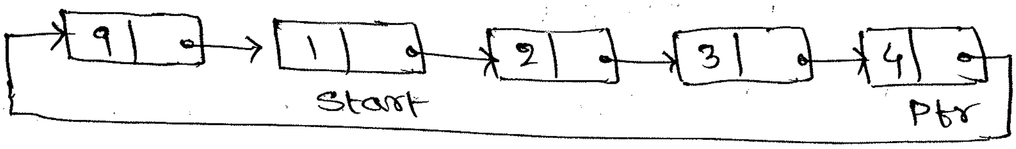
- move PTR so that its points to last node



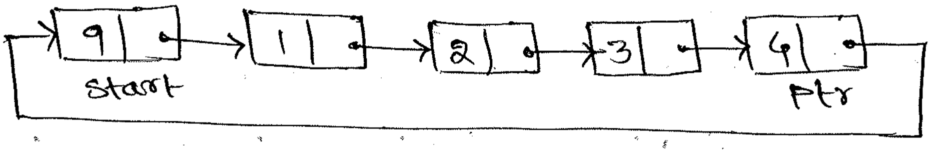
- Add the new node at beginning or starting of list



- Now change the last node next address to the new_node



- change the start to the new node

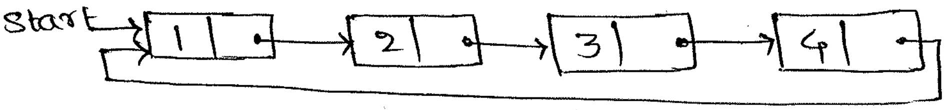


- Finally we are inserted new node at beginning of the circular linked list.

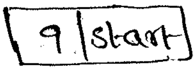
(ii) At Ending :-

```
Algorithm insert_end (item)
  {  if AVAIL = NULL then
          write " No memory for creating new_node";
          Goto Exit;
     new_node := AVAIL;
     AVAIL := AVAIL → Next ;
     new_node → data := item;
     new_node → Next := Start;
     Ptr := Start ;
     While Ptr → Next != Start do
            Ptr := ptr → Next ;
     Set Ptr → Next := new_node ;
  }
```

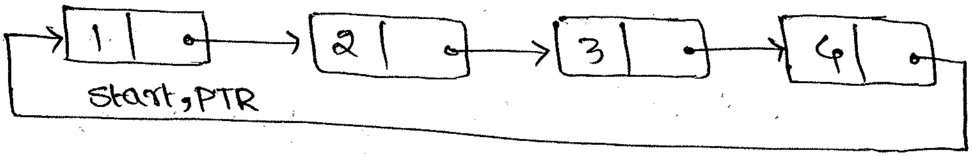
Example :- Insert new node into list having data 9



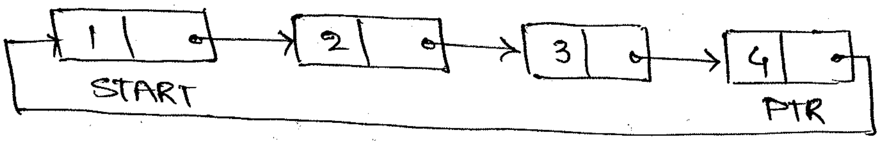
- Allocate new memory for new node and data 9, next start



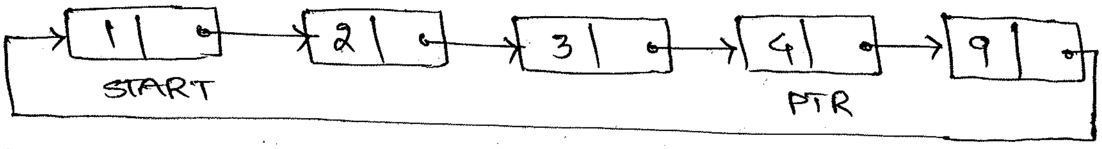
- Take a pointer variable PTR which initially points to START



- move PTR upto last node, PTR points to last node



- Add new node after the PTR node



- Finally we are inserted new node at end of the circular linked list.

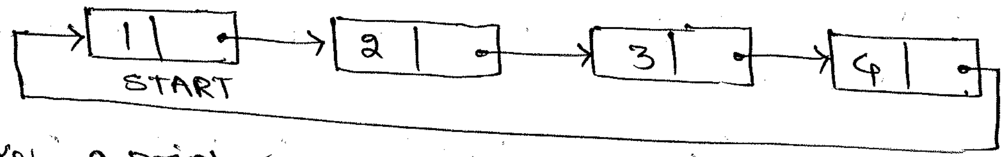
(2) Deletion of a node :-

- We can delete the node from circular linked list in two ways
  - (i) Deleting a node at beginning
  - (ii) Deleting a node at ending.

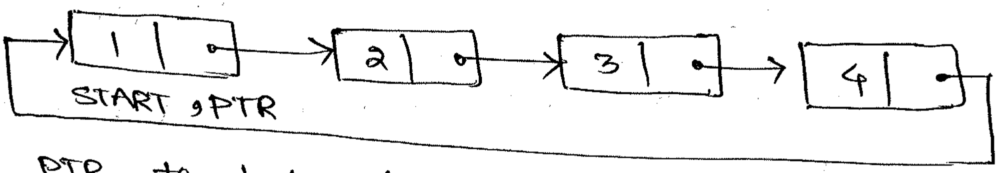
(i) At beginning :-

```
Algorithm  Delete_beg ( )
{  if  START = NULL
        Write "No nodes in the list";
        Goto Exit;
   Set  PTR :: = START;
   While  Ptr → Next != START  do
          PTR :: = PTR → Next ;
   Ptr → next  : = Start → Next ;
   free  ( START);
}  START  : = PTR → Next ;
```

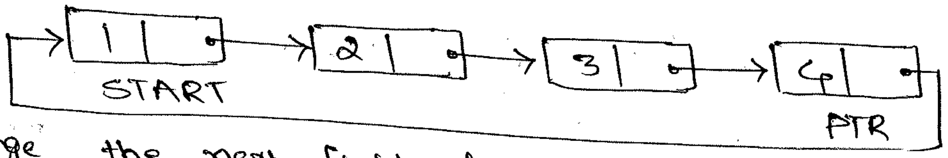
Example :- Deleting first node from the list



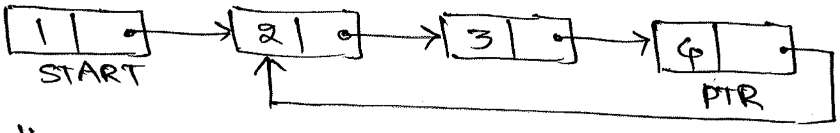
- Take a pointer PTR which points to the first node in list



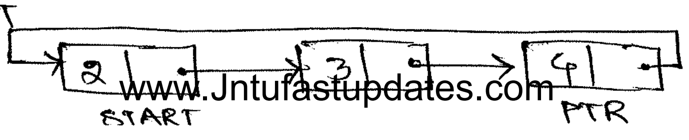
- move PTR to last node in the list



- change the next field of the PTR node to second node in list



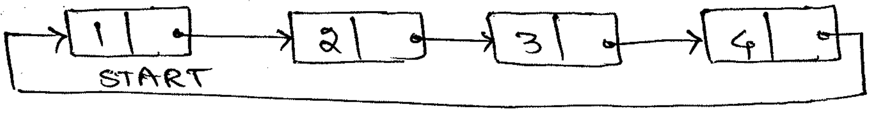
- Delete the START node from list and change to next node as START



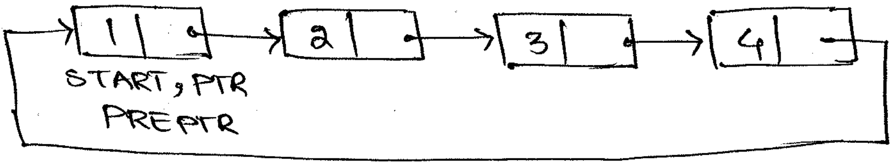
(ii) At Ending :-

```
Algorithm Delete_end ( )
{   if START = NULL
        write "No nodes in the list";
        Go to Exit;
    PTR := START;
    While PTR → Next != START do
        PREPTR := PTR
        PTR := PTR → Next ;
    PREPTR → Next := START ;
    free ( PTR) ;
}
```

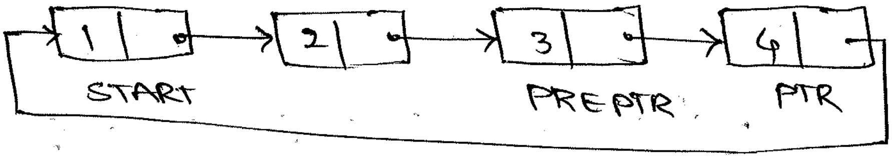
Example :- Deleting last node from the list.



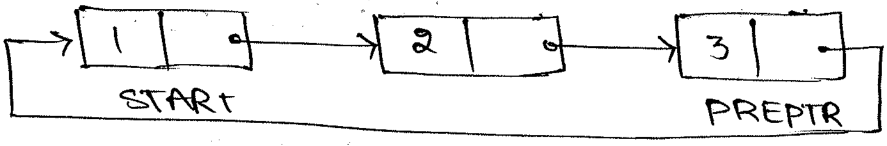
- Take two pointer variables PREPTR and PTR, initially points START



- move PTR to last node and PREPTR is points to previous of PTR node



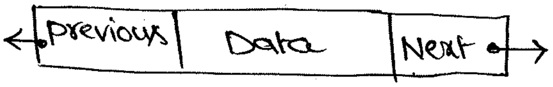
- Change the next field of PREPTR node and delete the PTR node from the list



- Finally we are deleting the last from the given Circular linked list.

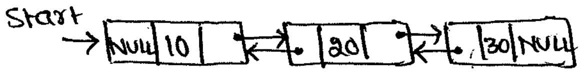
Double linked list :-

- A double linked list is more complex type of linked list which contains a pointer to the Next as well as Previous nodes in the sequence.
- In double linked list we can access both the successor nodes (Next nodes) and predecessor nodes (previous nodes) for any Orbitary node in the list
- Each node in double linked list



- In C, the structure of Double linked list is

```
Struct node
{
      int data;
      struct node *prev;
      Struct node *Next;
};
```



- If you are having single node in the list then prev and Next values are null.
- For the first node always prev is NULL and for the last node always Next is NULL
- There are 2 operations on double linked list, those are
  1) Insertion
  2) Deletion.

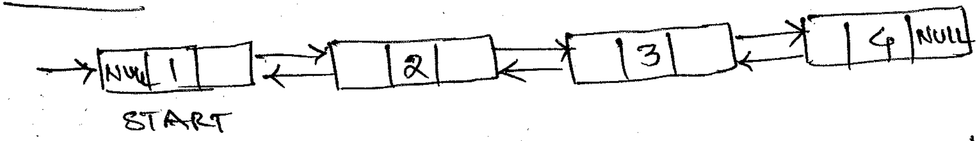
1) Insertion :-

- We can insert new node in double linked list in 3 ways
  (i) Inserting new node at beginning
  (ii) Inserting new node at Ending
  (iii) Inserting new node at given Position.

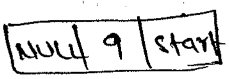
(i) At Beginning :-

```
Algorithm    insert_beg (item)
  {    if  AVAIL = NULL
           write "No memory for creation of new node";
           Goto Exit ;
       New_node := AVAIL;
       AVAIL := AVAIL → Next ;
       New_node → prev := NULL ;
       New_node → data := item ;
       New_node → Next := Start ;
       Start → prev := new_node ;
       Start := new_node ;
  }
```

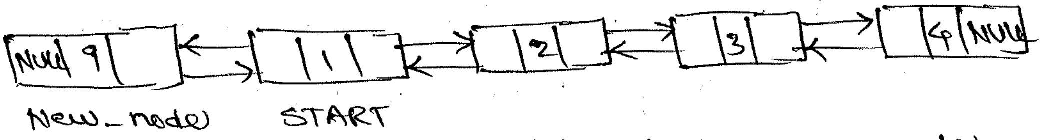
Example :- insert new node at beginning of list with data '9'



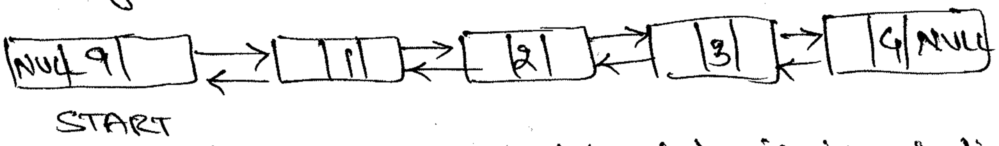
- Allocate new memory for node with data '9' and the field values



- Add the new node before the start node and change the Start previous field address to new node



- Now change START position to the new node

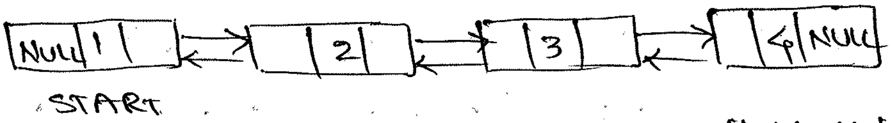


- Finally we are inserted new_node at beginning of list

(ii) At Ending of the list :-

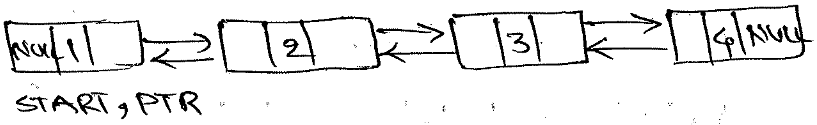
```
Algorithm    insert_end (item)
  {
       if AVAIL = NULL
           write "No memory for creation of new node";
           Goto Exit ;
       new_node := AVAIL ;
       AVAIL := AVAIL → Next ;
new_node → data := item;
new_node → Next := NULL;
PTR := START;
While PTR → Next != NULL do
        PTR := PTR → Next;
PTR → Next := new_node;
new_node → prev := PTR;
}
```

Example :- Insert new node at end of the list with data '9'

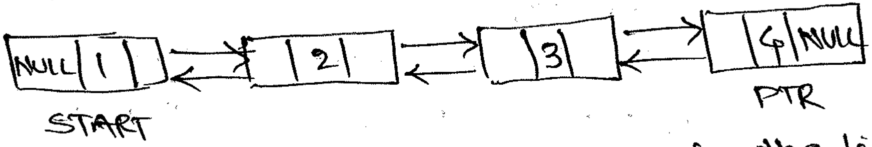


- Allocate memory for new node and field values

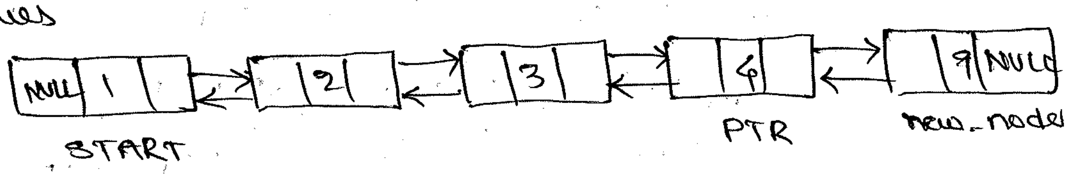
- Take PTR variable, initially its points to the first node



- move PTR to the end of list



- Add new_node after PTR node in the list and change the PTR next field and new_node prev field values

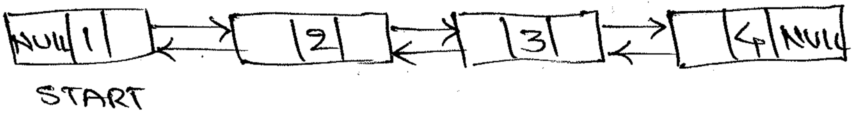


- Finally we are inserted new node at end of the list

(iii) At particular position :-

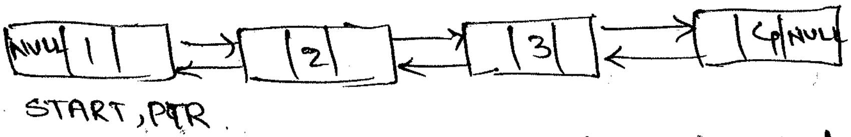
```
Algorithm insert_POS ( pos, item)
{
    if AVAIL = NULL.
        Write "No memory for new node";
        Goto Exit;
    new_node := AVAIL;
    AVAIL := AVAIL → Next;
    new_node → data := item;
    PTR := START ;    i=1;
    While i<POS-1    do
            PTR := PTR → Next ;
            i++ ;
     new_node → prev := PTR ;
     new_node → Next := PTR → Next ;
     PTR → next → prev := new_node ;
     PTR → next := new_node ;
}
```

Example :- insert new node of data 9 at POS = 4.

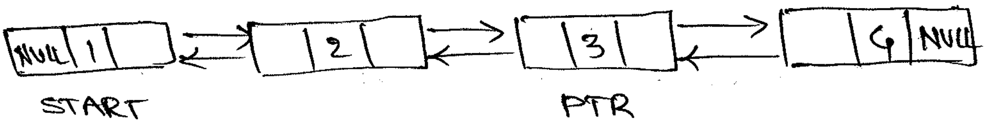


- Allocate memory for newnode of data 9

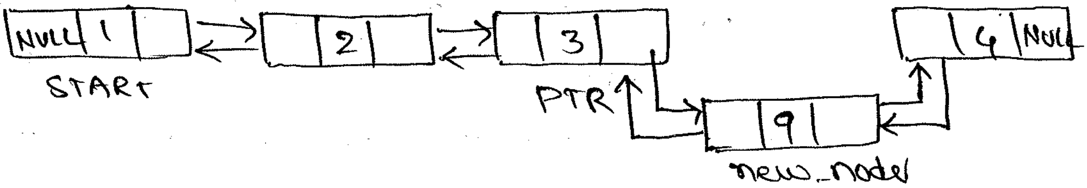
- Take PTR variable its points to the first node



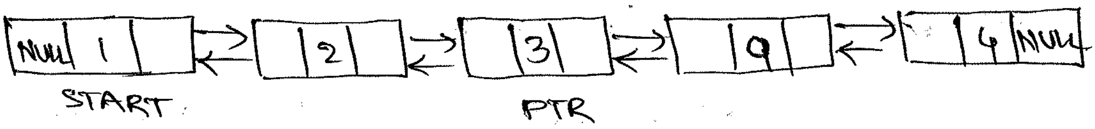
- Move PTR to the next node until POS-1 location



- Insert new node between PTR and its next node and change all the field of PTR next and new node fields



-



- Finally we are inserted new_node at particular position of '4'.

2. Deletion :-

- We can delete the node from double linked list in 3 ways

    (i) At beginning of the list

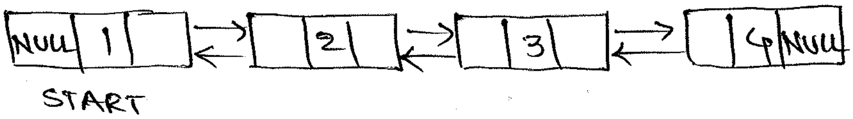
    (ii) At End of the list

    (iii) At particular position of the list.

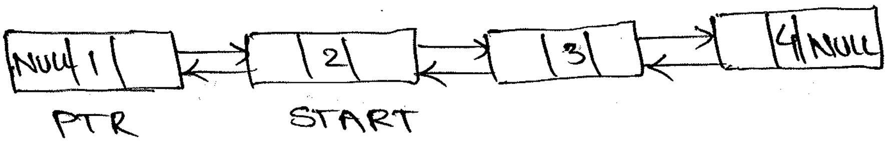
(i) Delete Node at beginning of the list :-

```
Algorithm delete_beg ( )
{
    if START = NULL
        write " No nodes in the list";
        Goto Exit;
    PTR := START;
    START := START ->Next ;
    START -> prev := NULL;
    free ( PTR ) ;
}
```

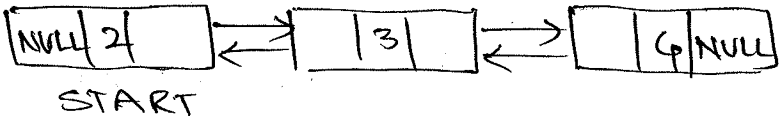
Example :- Delete the first nodes from double linked list.



- Takes PTR its points to the first nodes and change START value to the next node



- Delete PTR node from the list and change START previous field value to NULL.

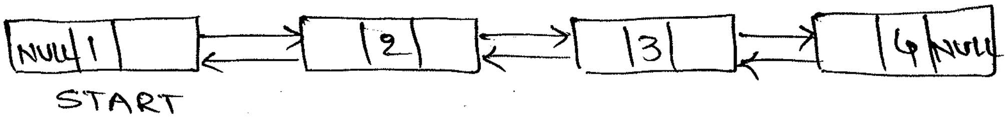


- Finally we are deleted the beginning of the node from the given double linked list.

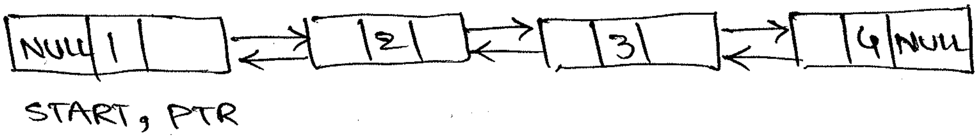
(ii) Delete Nodes at End of the list :-

```
Algorithm delete_End ( )
{
    if START = NULL
        write "No nodes in the list ";
        Goto Exit;
    PTR :=START ;
    while PTR -> Next != NULL do
        PTR := PTR -> Next ;
        PTR → prev → Next := NULL
        free (PTR);
    }
```

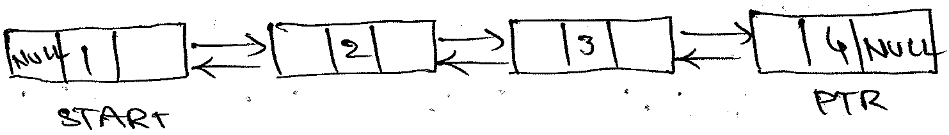
Example :- Delete nodes at last from the given list



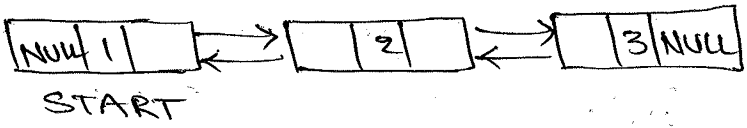
- Take PTR variable its pointing to the START node



- Move PTR to end of thelist.



- Change PTR previous nodes next field and delete PTR nodes from list

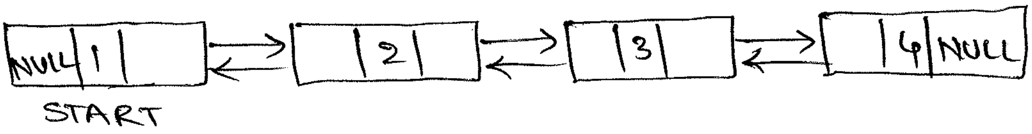


- Finally we are deleted the last node from list.

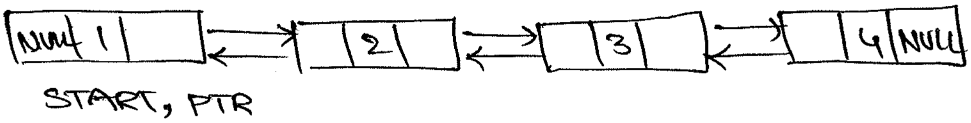
(iii) Delete node at given Position :-

```
Algorithm Delete_Pos( POS)
{
    if START = NULL
        Write "No nodes in the list";
        Goto Exit;
    PTR := START;
    i:= 1;
    While i < POS do
        PTR := PTR → Next;
    PTR → Prev → Next := PTR → Next;
    PTR → Next → prev := PTR → prev;
    free (PTR);
}
```

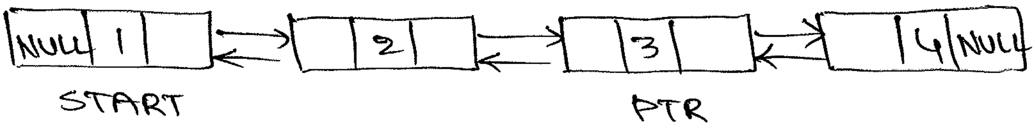
Example :- Delete a node at given position of '3' from the list



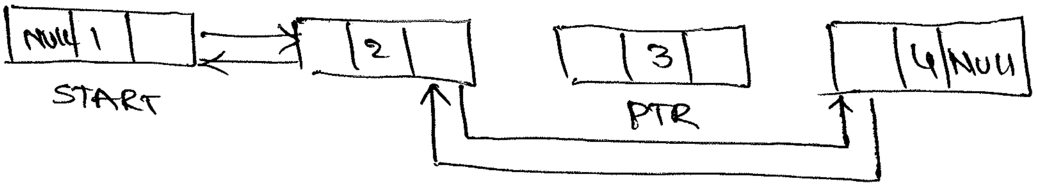
- Take PTR and it is pointing to START node



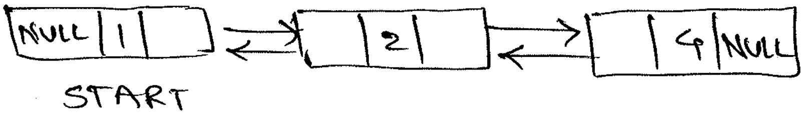
- move PTR to upto given position of '3'.



- Now change the PTR ~~node~~ previous node next field and PTR next node previous field values



- Now delete PTR node from the list



- Finally we are deleted the given position node from the list.