MODULE 3 : LINKED LISTS

Disadvantages Of Arrays :-

* The size is fixed. (during compilation time)
* The array items are stored continously next to each other.
* Insertion and deletion operation is a tedious (difficult) job.

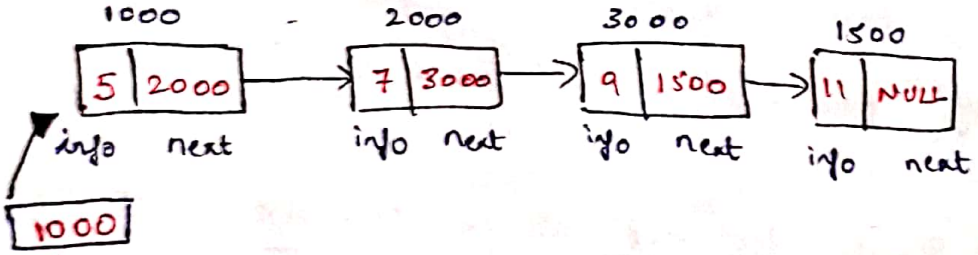
→ The above disadvantages can be overcome by using dynamic memory allocation ie; using linked lists.

LINKED LIST :-

Definition : A Linked List is a non-sequential collection of nodes. Each node has 2 fields:

i) info field - to store data or information to be manipulated.

(ii) next (or link) field - contains the address of the next node.



first/head → contains the address of 1st node.

Advantages Of Linked Lists :-

* Insertion and deletion operations can be done efficiently.
* NO wastage of memory. The memory can be allocated or deallocated as per requirement.

Representation (or Structure definition)

```
struct node
{
    int info;
    struct node *next;          // Self referential structure
};
```

* The info field can be integer, or character or any type of data, depending upon the type of information that the linked list is having.
* The next field is a pointer type. And the pointer should be a structure type. Because the next field holds the address of next node. And each node is basically a structure consisting of 'info' and 'next' field.

Garbage Collection :-

* The memory allocation for the objects (nodes) is done from the heap area of memory. If some objects is created and which is not been in use from a long time, then such an object is called garbage.
* The Garbage collection is a technique in which all such garbage (objects not used) is collected and recycled.
* The Garbage collection technique - cleans up the heap so that the memory occupied by the unused objects can be freed and can be used allocating for the new objects.
* The garbage collection algorithm works in 2 steps:
  1. Mark - In the marking process, all the live objects are located and marked them as non-collected objects.

② Sweep: Here, all unmarked objects are swept from the heap and the space that has been allocated by these objects can be used for allocating new objects.

Basic Operations on Linked List :-

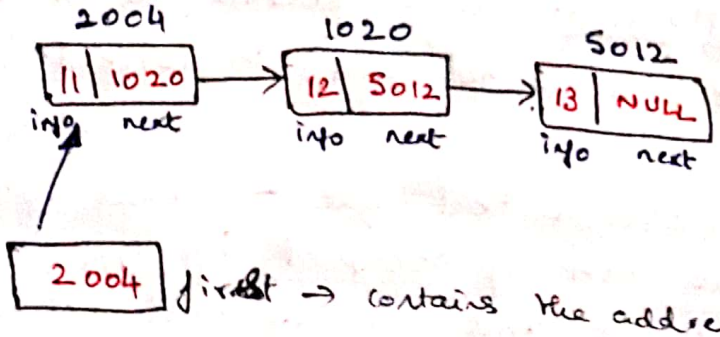
- Create (creates the linked list)
- Insert
- Delete
- Search.

TYPES OF LINKED LISTS :-

① Singly Linked Lists (SLL)
② Doubly Linked Lists (DLL)
③ Circular Singly Linked List (CSLL)
④ Circular Doubly Linked List (CDLL)

1) SINGLY LINKED LIST (SLL) :-

* It is called singly because this list consists of only one link, to point to next node.



first → contains the address of 1st node.

* The last field of the last node is NULL, means there is no further nodes.
* Initially, when no nodes are created, the variable first, will contain NULL (\0). This indicates the linked list does not exists.

'\0' first

Implementation of SLL:

Structure definition:

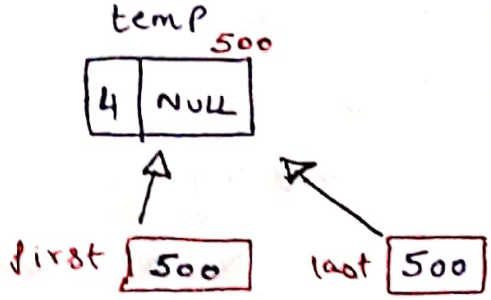
```
struct node
{
    int info;
    struct node *next;
} *first = NULL, *temp = NULL, *last = NULL;
```

Create a Node:

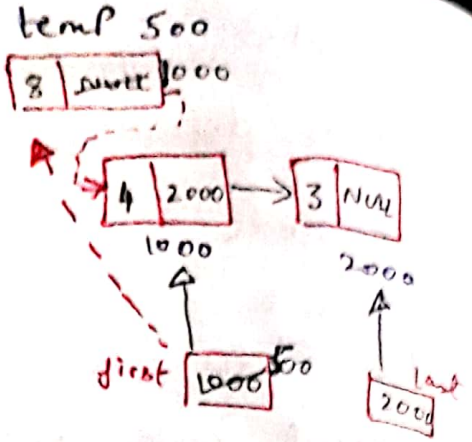
```
void create()
{
    temp = (struct node *) malloc(sizeof(struct node));
    printf("Enter the information to be stored \n");
    scanf("%d", &temp->info);
    temp->next = NULL;
}
```

Insertion at the beginning (front)

```
void insertbeg()
{
    if (first == NULL)          //List is empty
    {
        create();
        first = temp;
        last = first;
    }
    else
```

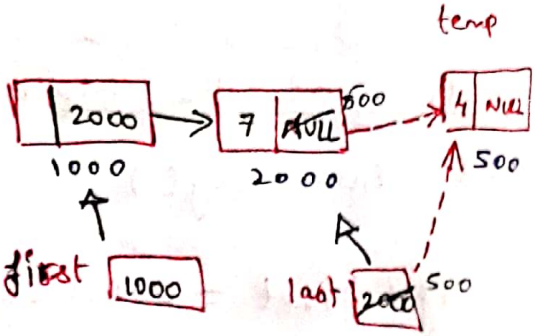


```
{
    create();
    temp→next = first;
     first = temp;
  }
}
```



Insertion at the end (Rear)

```
Void insertEnd()
{
   if(first==NULL)   //List is empty.
   {
      create();
      first = temp;
      last = first;
   }
   else
   {
      create();
     last→next = temp;
       last = temp;
   }
}
```



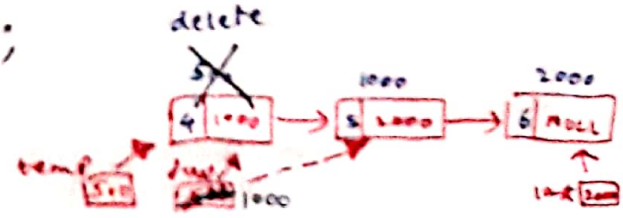
Display()

```
   void display()
   {   struct node *temp1;
```

```
temp1 = first;
if( temp1 == NULL)          // List is empty
{
      Printf(" List is empty ");
      return;
 }
Printf(" The contents of Linked List are: \n");
while(temp1 != NULL)        // Traverse through the list
  {
      Printf(" %d \n", temp1 -> info);
       temp1 = temp1 -> next;
   }
}
```

Delete a node at the beginning (Front)

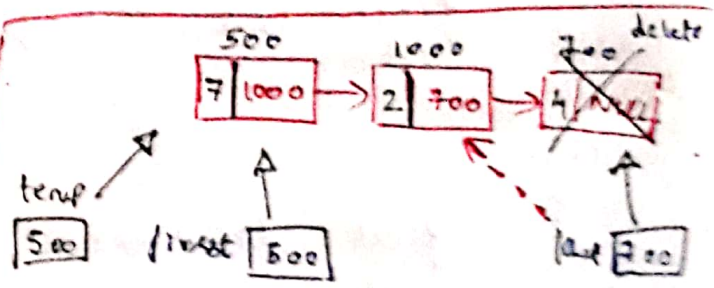
```
void deleteBeg()
{
   struct node *temp;
   temp = first;            // store address of 1st node in temp
   if( temp -> next == NULL) // only one node in list
   {
      printf(" The item deleted = %d \n", temp -> info);
       free(temp);
       first = last = NULL;
       return;
   }
  else
   {
      first = temp -> next;  // make 2nd node as 1st node
      Pf(" The item deleted: %d \n", temp -> info);
       free(temp);
   }
}
```



Deletion at the end (Rear)

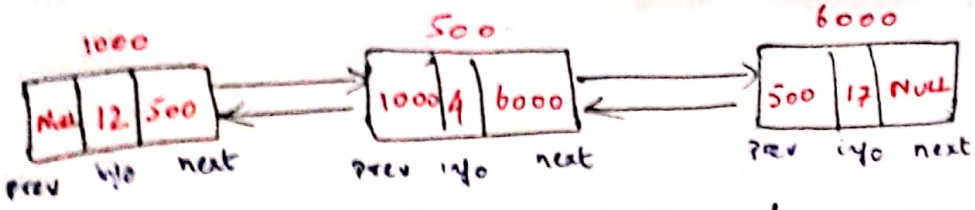
```
void deleteEnd()
{
    struct node *temp;
    temp = first;
    if(temp->next == NULL)
    {
        printf("The item deleted = %d\n", temp->info);
        free(temp);
        first = last = NULL;
        return;
    }
    else
    {
        while(temp->next != last)
            temp = temp->next;
        printf("The Item deleted = %d\n", last->info);
        free(last);
        temp->next = NULL;
        last = temp;
    }
}
```

```
while(temp->next != last)
      1000   700
```



8) DOUBLY LINKED LIST :-

* A doubly L.L. is a collection of nodes where each node is divided into 3 parts/fields:
  - info → where the information has to be stored
  - next → contains the address of the next node
  - Prev → contains the address of the previous node.



* In doubly linked list, traversing the list is possible in both directions.

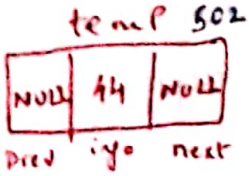
Implementation of D.LL:-

Structure definition

```
struct node
{
    int info;
    struct node *next;
    struct node *prev;
} *first = NULL, *last = NULL, *temp = NULL;
```

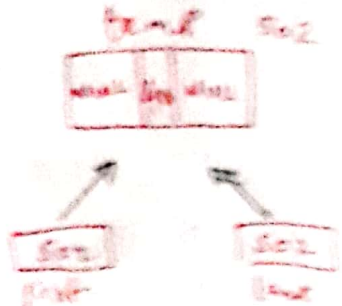
Create a node:

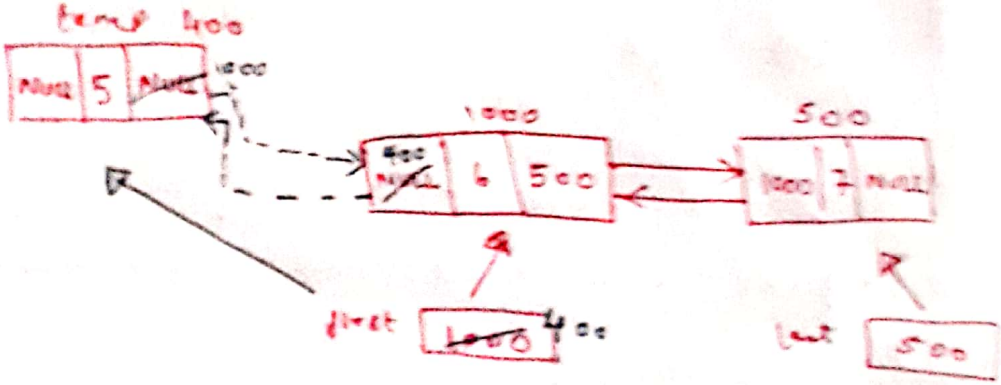
```
void create()
{
    temp = (struct node *) malloc(sizeof(struct node));
    printf("Enter the item\n");
    scanf("%d", &temp->info);
    temp->next = NULL;
    temp->prev = NULL;
}
```



Insertion at the beginning (First)

```
void InsertBeg()
{
    if(first == NULL)
    {
        create();
        first = last = temp;
    }
    else
    {
        temp -> next = first;
        first -> prev = temp;
        first = temp;
    }
}
```

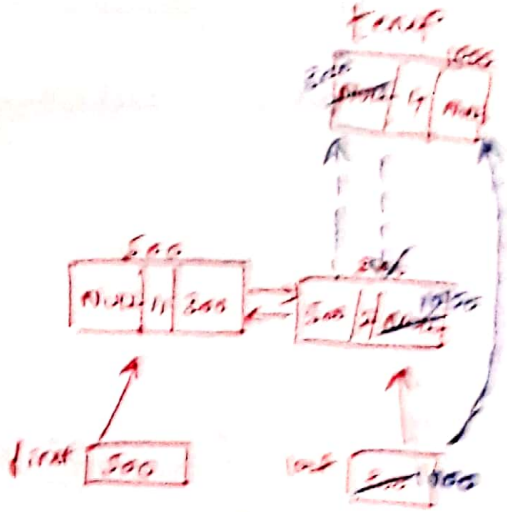




Insertion at the End (Rear)

```
void InsertEnd()
{
    if( first == NULL)
    {
        Create();
        first = last = temp;
    }
```

```
else
{
    last -> next = temp;
    temp -> prev = last;
    last = temp;
}
}
```



Display the contents of the list

```
void display()
{
    struct node *temp1;
    temp1 = first;
    if (temp1 == NULL)
    {
        printf("List is empty");
        return;
    }
    printf("The contents of linked lists are:\n");
    while (temp1 != NULL)
    {
        printf("%d\n", temp -> info);
        temp1 = temp1 -> next;
    }
}
```

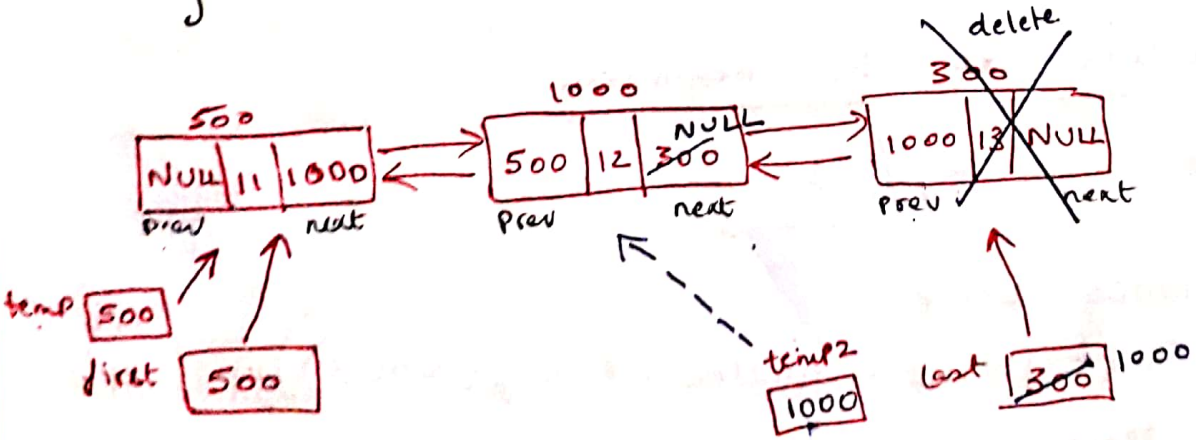
Delete a node at the beginning (Front)

```
void deleteBeg()
{
    struct node *temp;
    temp = first;
    if (first == NULL) // List is empty
    {
        Printf(" List is empty ");
        return;
    }
    if (temp -> next == NULL)    //only one node
    {
        Pf(" The item deleted = %d \n", temp -> info);
        free (temp);
        first = last = NULL;
    }
    else
    {
        first = first -> next;   // Make 2nd node as 1st node
        Printf(" The item deleted = %d \n", temp -> info);
        free (temp);
    }
}
```

Delete a node at the End :-

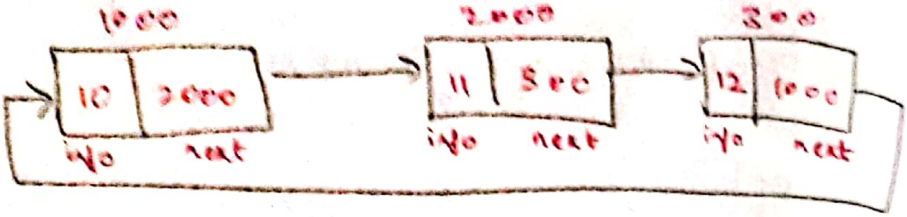
```
void deleteEnd()
{
    struct node *temp, *temp2;
```

```
temp = first;
if(first == NULL)
{
    Pf(" List is empty "); return;
}
if(temp -> next == NULL)
{
    Pf(" The item deleted = %d \n", temp -> info);
     free (temp);
     first = last = NULL;
 }
else
 {
      temp2 = last -> prev;
      temp2 -> next = NULL;
      Pf(" The item deleted = %d \n", last -> info);
      free (last);
       last = temp2;
  }
}
```



3.) CIRCULAR SINGLY LINKED LIST (CSLL)

* In circular S.L.L, the last node of the list will contain the address of the first node in the list.



Implementation:-

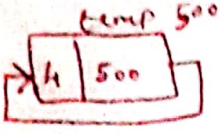
Structure definition:

```
struct node
{
    int info;
    struct node *next;
};

typedef struct node* NODE;
  NODE last;
  last = NULL;
```

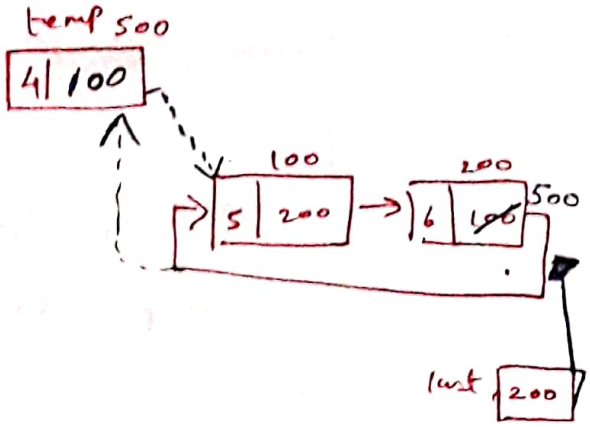
Insertion at the beginning

```
NODE insertBeg (int item, NODE last)
{
    NODE temp;
    temp = (NODE) malloc (sizeof (NODE));
    temp → info = item;
    if (last == NULL)        // list is empty
        last = temp;
```



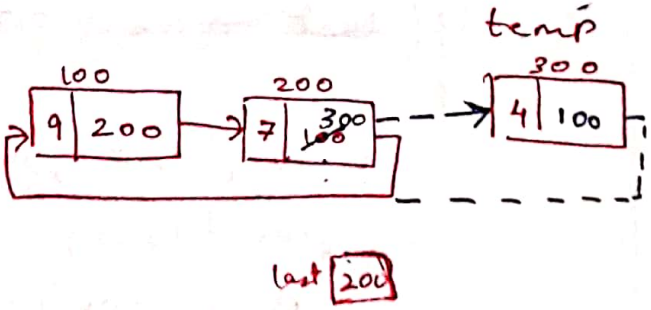
```
else
    temp -> next = last -> next;

last -> next = temp;
return last;
}
```



Insertion at the end :-

```
NODE insertEnd(int item, NODE last)
{
    NODE temp;
    temp = (NODE) malloc (sizeof(NODE));
    temp -> info = item;
    if (last == NULL)
        last = temp;
    else
        temp -> next = last -> next;
    last -> next = temp;
    return temp;
}
```



Display :-

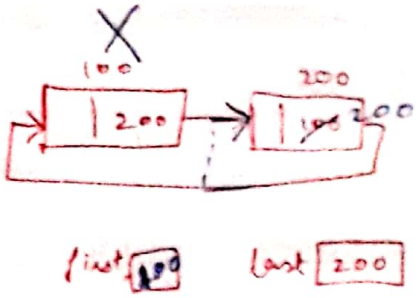
```
void display(NODE last)
{
    NODE temp;
    if(last == NULL) { pf("List is empty"); return; }
```

```
printf("contents of the list are: \n");
temp = last -> next;
while (temp != last)
{
    printf("%d\n", temp->info);
    temp = temp -> next;
}
printf("%d\n", temp->info);   //display last node
}
```

Deletion at the Beginning :-

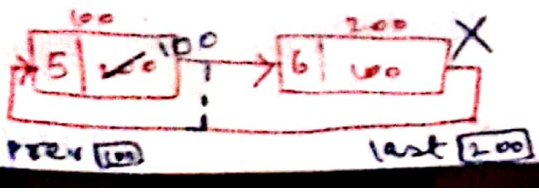
```
NODE deleteFront (NODE last)
{
    NODE first;
    if (last == NULL)
    {
        Pf("list is empty"); return;
    }
    if(last -> next == last)
    {
        printf("%d", last->info);
        free(last);
        return NULL;
    }
    first = last -> next;
    last -> next = first -> next;
```

```
    printf("%d", first->info);
    free(first);
    return last;
}
```



Deletion at the end :-

```
NODE deleteEnd(NODE last)
{
    NODE prev;
    if(last == NULL)
    { printf("List is empty"); return NULL;}
    if(last->next == last)
    {
        printf("Item deleted = %d\n", last->info);
        free(last);
        return NULL;
    }
    prev = last->next;          // obtain address of previous node
    while(prev->next != last)
    {
        prev = prev->next;
    }
    prev->next = last->next;    // Previous node is made last node
    printf("%d", last->info);
    free(last);                 // delete old last node
    return prev;                // return the new last node
}
```



Circular Singly L.L. with Header Nodes:

What is a header Node?

Definition: A header node is a special node whose next field always contains the address of the first node of the list.

* Using header node, any node in the list can be accessed.
* The info field of header node usually doesn't contain any information and such a node doesn't represent an item in the list.
* Useful information such as No. of nodes in the list can be stored in the info field.
* If the list is empty, then next field of header node contains NULL (\0).

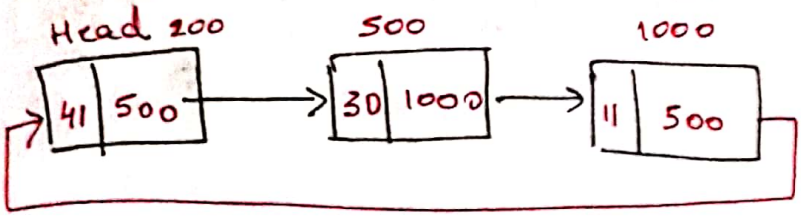
Advantages of header nodes:

1. Simplifies Insertion and deletion operations.
2. Avoids the usage of various cases such as "if only one node is present" what to do.
3. Design of program will be simple.

Circular S.L.L with header Nodes:

Example(1): An empty List using header node is pictorially represented as shown:

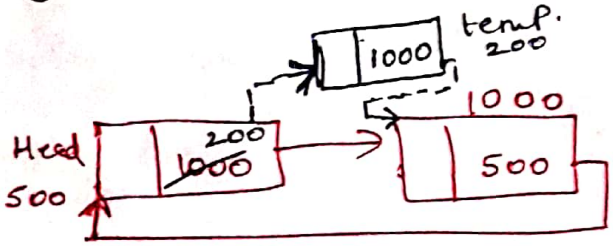
Example 2: The non-empty list using header node.



Operations:-

Insert a node at the beginning (Front End)

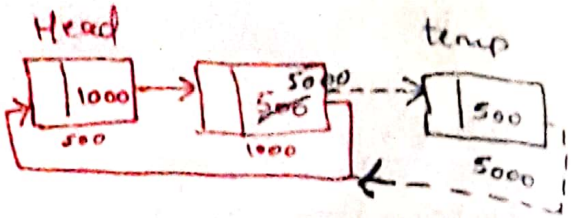
```
NODE insertBeg(int item, NODE head)
{
    NODE temp;
    temp = (NODE) malloc (sizeof(NODE));
    temp → info = item;
    temp → next = head → next;
    head → next = temp;
    return head;
}
```



Insert a node at the end:

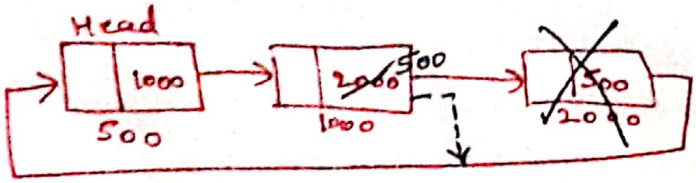
```
NODE insertEnd(int item, NODE head)
{
    NODE temp, cur;
    temp = (NODE) malloc (sizeof(NODE));
    temp → info = item;
    cur = head → next;  // obtain address of 1st node
    while(cur → next != head)  // obtain address of last node
    {
        cur = cur → next;
    }
```

```
    cur -> next = temp;
    temp -> next = head;
    return head;
}
```



Delete a node at rear end

```
NODE deleteEnd(NODE head)
{
    NODE prev, cur;
    if(head -> next == head)
    {
        pf("List is empty");
        return head;
    }
    cur = head -> next; // 1st node.
    prev = head;        // previous to 1st node.
    while(cur -> next != head)
    {
        prev = cur;
        cur = cur -> next;
    }
    prev -> next = head;
    pf("%d\n", cur -> info);
    free(cur);
    return head;
}
```



cur = 1000
prev = 500
while(2000 != 500)
{
prev = 1000
cur = 2000
}
while(500 != 500) ✗
prev -> next = 500

Delete at front End :-

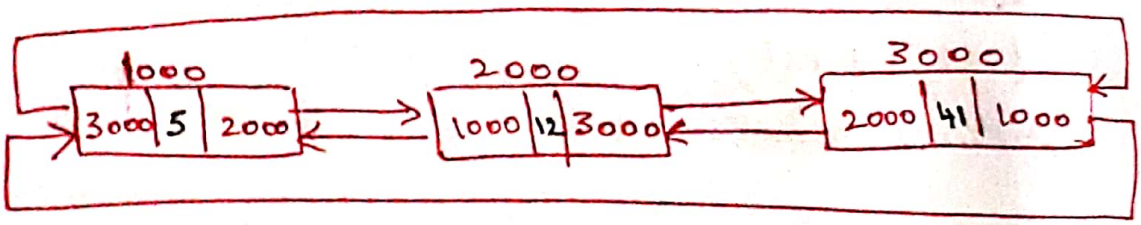
```
NODE deleteFront(NODE head)
{
    NODE first, second;
    if (head -> next == head)
    {
        Pf(" List is empty ");
        return head;
    }
    first = head -> next;      // obtain I st node of list
    second = first -> next;    // obtain 2nd node of list
    head -> next = second;     // Make 2nd node as I st node
    Pf(" Item deleted = %d \n", first -> info);
    free (first);
    return head;
}
```

Display the contents of CSLL :-

```
void display(NODE head)
{
    NODE temp;
    if(head -> next == head)
    {
        Pf(" List is empty");
        return;
    }
    printf(" The contents are : \n");
    temp = head -> next;
    while (temp != head)
    {
        Pf(" %d \n ", temp -> info);
        temp = temp -> next;
    }
```

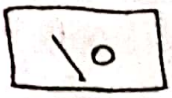
CIRCULAR DOUBLY LINKED LIST :-

* Here traversing is possible in forward, backward and also in circular fashion.
* Each node has 3 fields:
  - info - information to be stored.
  - next - contains the address of the next node
  - Prev - Contains the address of previous node.
* The Prev field of the I st node contains the address of the last node whereas the next field of the last node contains address of the first node.



Implementation:

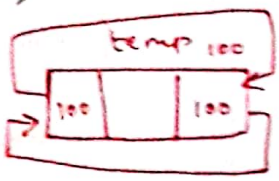
```
struct node
{
   int info;
   struct node *next;
   struct node *prev;
};
typedef struct node* NODE;
NODE first;
first = NULL;
```

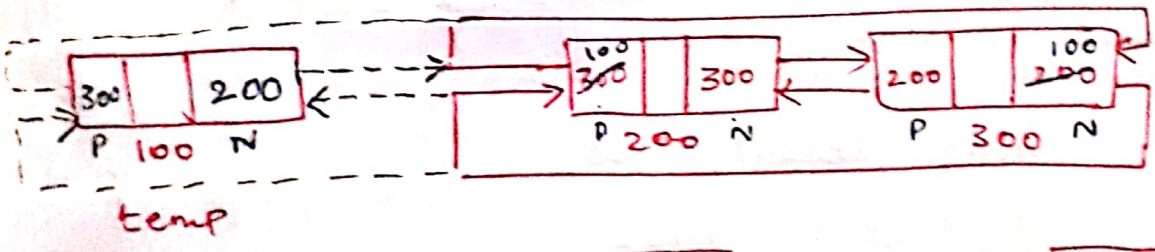


Initially (List is empty)

Insert a Node at the Beginning

```
NODE insertBeg(int item, NODE first)
{
    NODE temp, last;
    temp = (NODE)malloc (sizeof (NODE));
    temp → info = item;
    temp → prev = temp → next = temp;
    if (first == NULL) return temp;
    last = first → prev;        // Get address of last node
    temp → next = first;        // Link the 1st node with new node
    first → prev = temp;
    last → next = temp;         //Link the last node with new node
    temp → prev = last
    return temp;
}
```



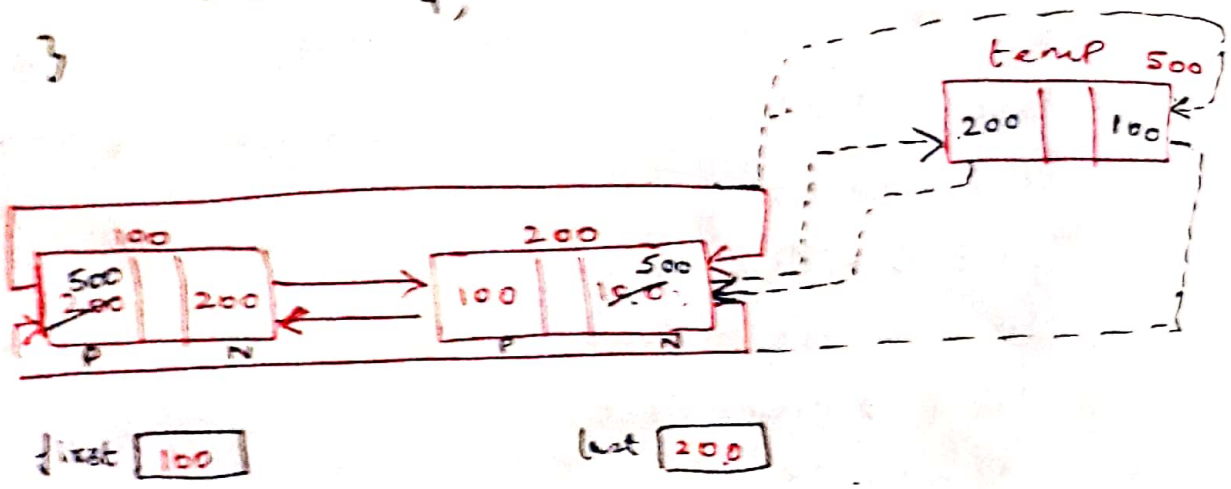


first 200 last 300

Insert a node at the rear End :-

```
NODE insertEnd(int item, NODE first)
{
    NODE temp, last;
    temp = (NODE) malloc (sizeof(NODE));
    temp → info = item;
    temp → prev = temp → next = temp;
```

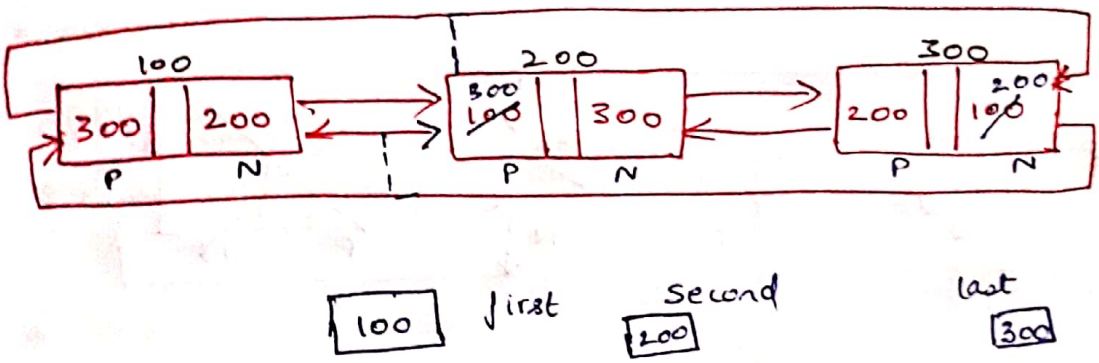
```
if (first == NULL)  return temp;
last = first -> prev;        // Get address of last node
last -> next = temp;         // Link last node with new node
temp -> prev = last;
first -> prev = temp;        // Link the first node with new node
temp -> next = first;
return first;
}
```



Delete a node at the Beginning

```
NODE deleteBeg(NODE first)
{
    NODE second, last;
    if (first == NULL)
    {
        Pf("List is empty");
        return NULL;
    }
    if (first -> next == first)  // Only one node
    {
        Pf("Item deleted = %d\n", first -> info);
        free(first);
        return NULL;
    }
```

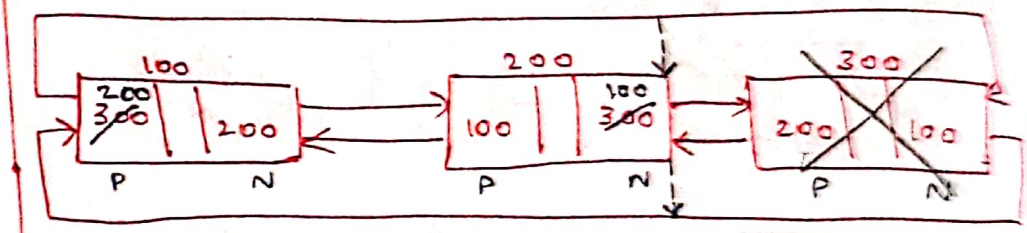
```
second = first -> next;
last = first -> prev;
second -> prev = last;
last -> next = second;
printf("Item deleted = %d\n", first->info);
free(first);
return second;
}
```



Delete a Node from the rear End

```
NODE deleteEnd(NODE first)
{
    NODE last, temp;
    if(first == NULL)
    {
        pf("List is empty"); return; NULL;
    }
    if(first -> next == first)
    {
        pf("Item deleted = %d\n", first->info);
        free(first);
        return NULL;
    }
```

```
last = first -> prev;
temp = last -> prev;
temp -> next = first;
first -> prev = temp;
Pf("Item deleted = %d\n", last -> info)
free(last);
return first;
}
```



first 100 last 300

temp 200

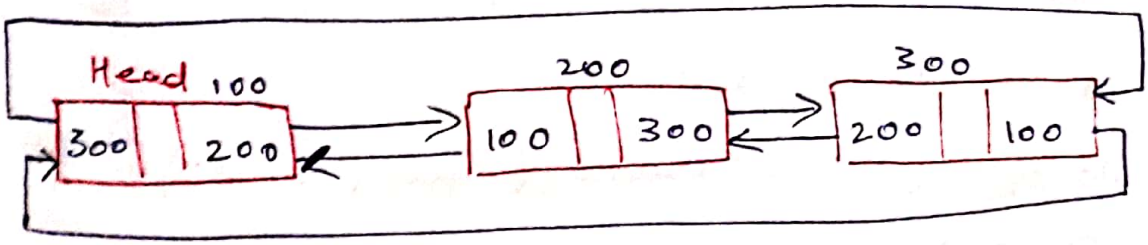
Display Circular D.L.L

```
Void display(NODE first)
{
    NODE cur, last;
    if(first == NULL)
    {
        Pf("List empty"); return;
    }
    Pf("Contents are :\n");
    cur = first;
    last = first -> prev;
```

```
while (cur != last)
{
    Pf(" %d\n", cur -> info);
    cur = cur -> next;
}
Pf(" %d", cur -> info);
}
```

Circular Doubly L.L. with Header Node :-

* A circular D.L.L. is a variation of doubly linked with a header node in which:
  - Prev field of header contains the address of the last node in the list.
  - The next field of header contains the address of the first node in the list.
  - The prev field of last node contains the address of its previous node.
  - The next field of the last node contains the address of header node of the list.

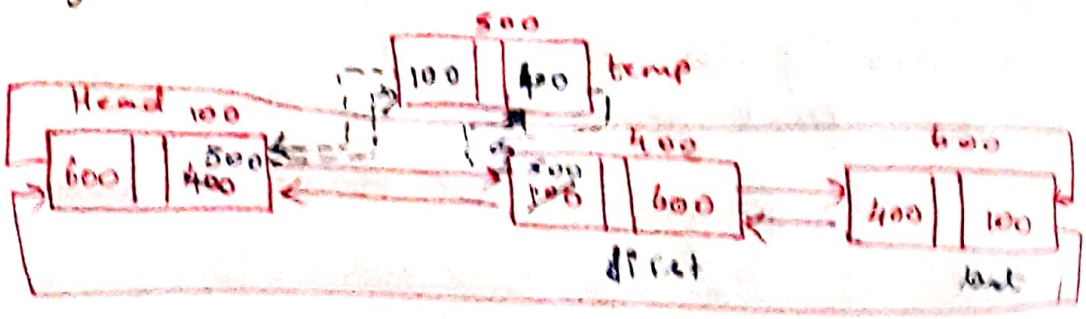


Implementation :

```
struct node
{
    int info;
    struct node *prev, *next;
};
typedef struct node * NODE;
```

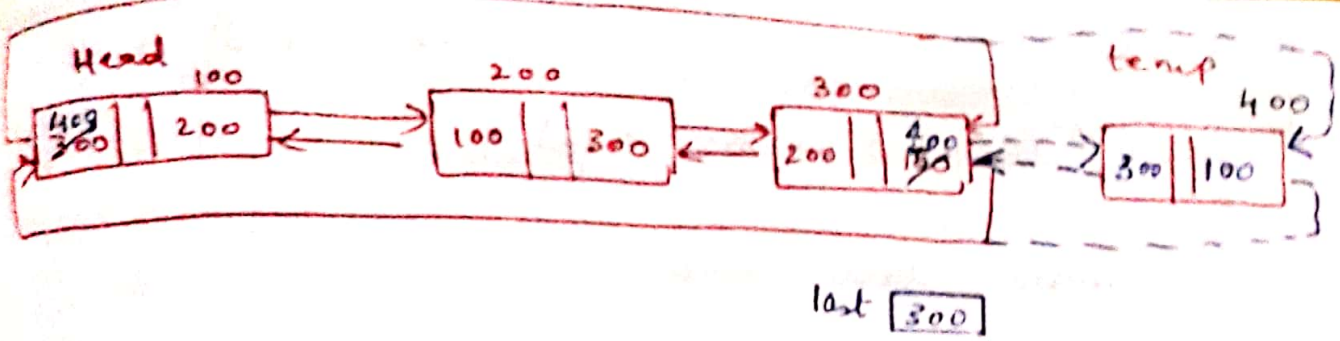
Insertion at the Beginning :-

```
NODE insertBeg(NODE head, int item)
{
    NODE temp, first;
    temp = (NODE) malloc (sizeof (NODE));
    temp -> info = item;
    first = head -> next;
    temp -> prev = head;
    head -> next = temp;
    temp -> next = first;
    first -> prev = temp;
    return head;
}
```



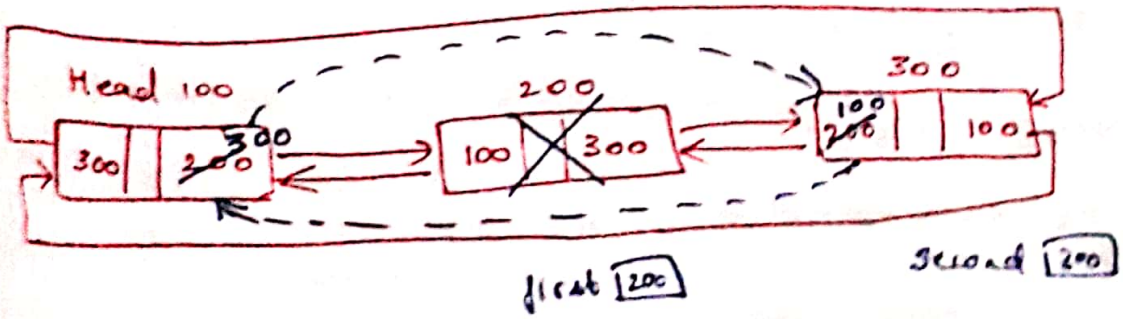
Insertion at the rear End

```
NODE insertEnd(NODE head, int item)
{
    NODE temp, last;
    temp = (NODE) malloc (sizeof (NODE));
    temp -> info = item;
    last = head -> prev;
    last -> next = temp;
    temp -> prev = last;
    temp -> next = head;
    head -> prev = temp;
    return head;
}
```



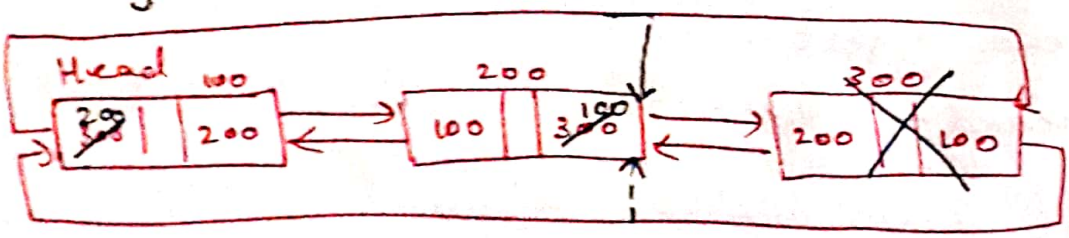
Delete a node from the Front End

```
NODE deleteBeg(NODE head)
{
    NODE first, second;
    if(head -> next == head)
    {
        Pf("List is empty");
        return;
    }
    first = head -> next;
    second = first -> next;
    head -> next = second;
    second -> prev = head;
    Pf("Item deleted = %d \n", first -> info);
    free(first);
    return head;
}
```



Delete a node from the rear end

```
NODE deleteEnd (NODE head)
{
    NODE last, temp;
    if (head -> next == head)
    {
        pf(" List is empty");
        return head;
    }
    last = head -> prev;
    temp = last -> prev;
    head -> prev = temp;
    temp -> next = head;
    printf (" Item deleted = %d ", last -> info);
    free ( last);
    return head;
}
```



temp 200 last 300

Display contents of C.DLL with header node

```
void display (NODE head)
{
    NODE cur;
    if (head -> next == head)
    {
        pf(" List empty "); return;
    }
```

```
printf(" contents of L.L are :\n");
cur = head -> next;
while ( cur != head)
{
    pf("%d\n", cur -> info);
    cur = cur -> next;
}
}
```

Additional List Operations (S.L.L) :-

1.) Write a Function to length of the list or to count the number of nodes present.

⇒
```
int length(NODE first)
{
    NODE cur;
    int count=0;
    if( first == NULL) return 0;
    cur = first ;
    while ( cur != NULL) // Traverse through list
    {
        count ++;
        cur = cur -> next;
    }
    return count;
}
```

2.) Function to search for an item in a list

```
Void search(int key, NODE first)
{
    NODE cur;
    if(first==NULL) { pf(" List Empty"); return; }
```

```
cur = first;
while (cur != NULL)
{
    if (key == cur->info) break;
    cur = cur->next;
}
if(cur == NULL)
{
    pf(" search unsuccessful\n");
    return;
}
Printf("search is successful\n");
}
```

3.) Function to delete a Node whose information (info) field is specified:

```
NODE deleteinfo(int key, NODE first)
{
    NODE prev, cur;
    if (first == NULL)
    {
        pf(" List is Empty");
        return NULL;
    }
    if(key == first->info) // If key is present in 1st nd
    {
        cur = first
        first = first->next;
        free (cur);
        return first;
    }
```

```
/* search for the node to be deleted */
prev = NULL;
cur = first;
while( cur != NULL)
{
    if( key == cur -> info) break;
    prev = cur;
    cur = cur -> next;
}
if( cur == NULL)    // If end of LList, key not found.
{   pf(" search unsuccessful ");
    return first;
}
/* search successful, then delete */
prev -> next = cur -> next;
free (cur);
return first;
}
```

4.) To Concatenate Two lists:

```
NODE concat(NODE first, NODE sec)
{
    NODE cur;
    if(first == NULL) return sec;
    if( sec == NULL) return first;
    cur = first;
    while(cur -> next != NULL)
        cur = cur -> next;
    cur -> next = sec;   // attach 1st node of 2nd list to end of 1st list
    return first;
}
```

5.) To reverse a list (Invert)

```
NODE reverse(NODE first)
{
    NODE cur, temp;
    cur = NULL;
    while (first != NULL)
    {
        temp = first -> next;
        first -> next = cur;
        cur = first;
        first = temp;
    }
    return cur;
}
```

6.) To create an Ordered (sorted) List:-

```
NODE insert(int item, NODE first)
{
    NODE temp, prev, cur;
    temp = (NODE) malloc (sizeof (NODE));
    temp -> info = item;
    temp -> next = NULL;
    if (first == NULL) return temp;
    if (item <= first -> info)
    {
        temp -> next = first;
        return temp;
    }
    PREV = NULL;
    cur = first;
```

```
while (cur != NULL && item > cur->info)
{
        prev = cur;
        cur = cur->next;
}

prev->next = temp;
temp->next = cur;
return first;
}
```

Stacks using Linked Lists :-

* W.K.T. stack is a special type of data structure where elements are inserted at one end and elements are deleted from the same end. That is, if an element is inserted at front end, then an element has to be deleted from front end. (or) If an element is inserted at rear end, then an element has to be deleted from the rear end.
* Thus, a stack can be implemented using the following functions :

insertBeg() deleteBeg() display()	OR	insertEnd() deleteEnd() display()

Queues using LinkedLists :-

A queue is a special type of data structure, where insertion takes place at one end and elements are deleted at the other end.

* Thus, a queue can be implemented using :

insertBeg() deleteEnd() display()	OR	insertEnd() deleteBeg() display()

* For double ended Queues using Linked Lists :
All functions ie; insertBeg(), deleteBeg(), insertEnd(), deleteEnd() and display() should be included.

Applications Of Linked Lists :-

1.) Evaluation of polynomials

2.) Addition of polynomials

3.) MultiLinked data structure (Sparse matrices)

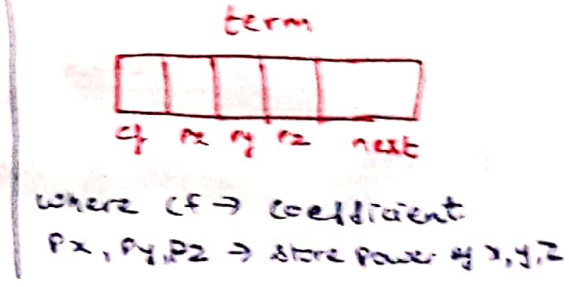
4.) Arithmetic operations on long positive numbers

5.) In symbol table construction (compiler design)

1.) Polynomial Representation and Evaluation :-

Representation using L.L

```
struct node
{
   int cf, Px, Py, Pz;
   struct node *next;
};
typedef struct node* NODE;
```

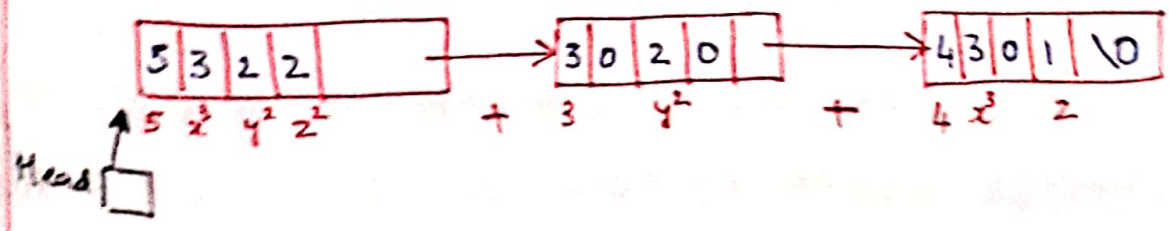


where cf → coefficient
Px, Py, Pz → store power of x, y, z

→ Using the above structure, the polynomial :

$$5x^3y^2z^2 + 3y^2 + 4x^3z$$

can be represented using singly linked list as shown:



* Function to insert at the rear end of the list (CSLL)

```
NODE insertrear(int cf, int px, int py, int pz, NODE head)
{
    NODE temp, cur;
    temp = (NODE) malloc (sizeof(NODE));
```

```
temp -> cf = cf;
temp -> px = Px;
temp -> Py = Py;
temp -> Pz = Pz;
Cur = head -> next;
while (Cur -> next != head)
{
    Cur = Cur -> next;
}
Cur -> next = temp;        // insert the node at the end
temp -> next = head;
return head;
}
```

⇒ How to read a polynomial consisting of 'n' terms

```
NODE readPoly(NODE head)
{
    int i, n;
    int cf, Px, Py, Pz;
    printf("Enter the no. of terms in the poly\n");
    scanf("%d", &n);
    for(i=1; i<=n; i++)
    {
        printf("Enter term:%d\n", i);
        printf("cf, Px Py Pz = ");
        scanf("%d %d %d %d", &cf, &Px, &Py, &Pz);
        head = insertrear(cf, Px, Py, Pz, head);
    }
    return head;
}
```

→ Display a Polynomial

```
void display (NODE head)
{
    NODE temp;
    if ( → next == head)
    {
        Printf(" Polynomial doesn't exist "); return;
    }
    temp = head → next;
    while (temp != head)
    {
        if (temp → cf < 0)
            Pf(" %d", temp → cf);
        else
            Pf("+%d", temp → cf);
        if (temp → px != 0) Pf(" x^%d ", temp → px);
        if (temp → py != 0) Pf(" y^%d ", temp → py);
        if (temp → pz != 0) Pf(" z^%d ", temp → pz);
        temp = temp → next;
    }
    Printf(" \n");
}
```

→ Evaluate a Polynomial :- } Refer Lab. Program 9

→ Add 2 Polynomials :- }

Note: Algorithm for evaluating a polynomial

Step 1 : Read the values of x, y, z

Step 2 : For each term, substitute x, y, z

Step 3 : Add all the terms, the final sum is obtained.

Algorithm to add 2 polynomials

For each term of polynomial 1

Step 1 : Access each term of poly1

Step 2 : Search for power of above term in poly2

Step 3 : If found in Poly 2

Add coefficients and add sum to poly 3

Else

Add the term of poly1 to poly 3

end for

Add Remaining terms of poly 2 to poly3.

2) Sparse Matrix Representation :-

* A spare matrix is a matrix that has very few non-zero elements spread out thinly

Eg :

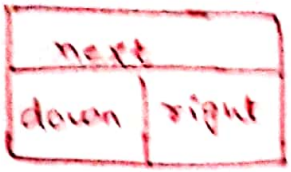
$$\begin{bmatrix} 10 & 0 & 0 & 40 \\ 11 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 50 \\ 0 & 15 & 0 & 25 \end{bmatrix}$$

* The Sparse matrix can be more efficiently represented using linked list. (rather than array

* Using Linked Lists :

* Each column of a sparse matrix is represented as circular linked list with a header node having three fields : down, right and node fields

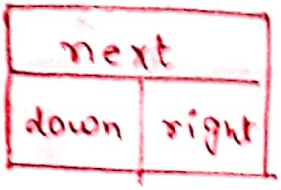
head



next → header of next column
right – not used
down – points to next item in the column.

* Each row of a sparse matrix is represented as a circularly linked list with a header node having 3 fields: down, right and next fields as shown:

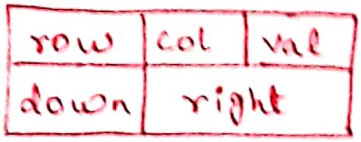
head



next – not used
right – points to next item in the row
down – not used.

* Each item is represented as a node having 5 fields as shown below:

item



right – points to next item in the row.
down – points to next item in the column.

Example: P.T.O

Note: The first node in the following list diagram is identified by the variable head, contains the size of the matrix where 5 represents the number of rows, 4 represents the number of columns and 6 represents non-zero elements to be manipulated.

Consider the following 5 x 4 matrix

	col[0]	col[1]	col[2]	col[3]
row[0]	3	0	0	0
row[1]	5	0	0	6
row[2]	0	0	0	0
row[3]	4	0	0	8
row[4]	0	0	9	0

* The above matrix can be representing using L.L as follows