UNIT- IV

Linked Lists – II: Polynomial Representation- Adding Polynomials- Circular List Representation of Polynomials, Equivalence Classes, Sparse Matrices, Sparse Matrix Representation- Sparse Matrix Input- Deleting a Sparse Matrix, Doubly Linked Lists, Generalized Lists, Representation of Generalized Lists- Recursive Algorithms for Lists Reference Counts, Shared and Recursive Lists.

***

POLYNOMIALS

The general form of a polynomial equation with a simple variable is:

$$\mathbf{P(x) = a_{m-1}\, x^{em-1} + a_{m-2}\, x^{em-2} + - - - - - - -- + a_0\, x^{e0}}$$

Where,

a_i are nonzero coefficients and the e_i are nonnegative integer exponents.

The largest exponent of a polynomial is called it degree.

Example: P(x) = $3x^{14} - 7x^{8} + 1$

A polymial equation can be represented using arrays and linked lists.

POLYNOMIAL REPRESENTATIONS USING LINKED LIST

To represent the elements of polynomial equation in terms of a linked list, create nodes with three fields as:

coef	expo	link

Here,

coef field stores coefficient values a_i
expo field stores exponent values e_i
and link field stores address of the next element in the list.

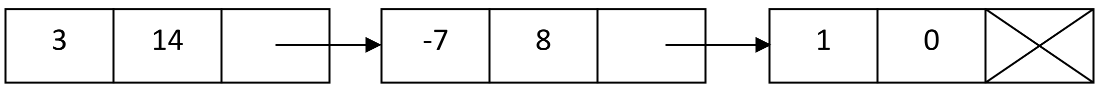
```
typedef struct list
{
    int coef,expo;
    struct list *link;
```

} PolyNode;

Example: Consider the polynomial equation as

P(x) = $3x^{14} - 7x^{8} + 1$

For this, linked representation can be shown as:



POLYNOMIAL ADDITION

Assume the given polynomial equations are P1 and P2 with the degrees D1 and D2. To add these two polynomial equations, first store the highest degree value D3 to form the result polynomial equation as P3. Initially set two pointer variables T1 at P1 and T2 and P2.

Here, addition process falls into three cases.

Case 1: If D3 > D1 then copy the node information of T2 into the Resultant polynomial equation and move the pointer T2 to next node.

Case 2: If D3 > D2 then copy the node information of T1 into the Resultant polynomial equation and move the pointer T1 to next node.

Case 3: If D3 = D1 and D3 = D2 then add the coefficient values of T1 and T2 and copy the value into the Resultant polynomial equation and move the two pointers T1 and T2 to next node.

Implementations

```
#include<iostream.h>
#include<alloc.h>
#include<conio.h>

class Polynomial
{
	typedef struct list
	{
```

```
        int coef,expo;
        struct list *link;
    }PolyNode;
    PolyNode *P1,*P2,*P3;
    int Degree1,Degree2,Degree3;
    public:Polynomial()
        {
            P1=P2=P3=NULL;
        }
        void AddData();
        void Display();
};
void Polynomial::AddData()
{
    PolyNode *NEW,*END1=NULL,*END2=NULL,*END3=NULL;
    int k;
    cout<<endl<<"First Polynomial Equation"<<endl;
    cout<<"Enter Degree of the Polynomial 1 =";
    cin>>Degree1;
    for(int i=Degree1;i>=0;i--)
    {
        cout<<"Enter Coefficient for Exponent ="<<i<<" =";
        cin>>k;
        NEW=(PolyNode*)malloc(sizeof(PolyNode));
        NEW->coef=k;
        NEW->expo=i;
        NEW->link=NULL;
        if(P1==NULL)
            P1=END1=NEW;
        else
        {
            END1->link=NEW;
            END1=NEW;
        }
    }
    cout<<endl<<"Second Polynomial Equation"<<endl;
    cout<<"Enter Degree of the Polynomial 2 =";
    cin>>Degree2;
    for(i=Degree2;i>=0;i--)
    {
        cout<<"Enter Coefficient for Exponent ="<<i<<" =";
        cin>>k;
        NEW=(PolyNode*)malloc(sizeof(PolyNode));
        NEW->coef=k;
        NEW->expo=i;
        NEW->link=NULL;
        if(P2==NULL)
            P2=END2=NEW;
        else
        {
```

```
				END2->link=NEW;
				END2=NEW;
			}
		}
		if(Degree1>Degree2)
			Degree3=Degree1;
		else
			Degree3=Degree2;

		PolyNode *T1=P1,*T2=P2;
		for(i=Degree3;i>=0;i--)
		{
			NEW=(PolyNode*)malloc(sizeof(PolyNode));
			if(i>Degree1)
			{
				NEW->coef=T2->coef;
				NEW->expo=T2->expo;
				NEW->link=NULL;
				if(P3==NULL)
					P3=END3=NEW;
				else
				{
					END3->link=NEW;
					END3=NEW;
				}
				T2=T2->link;
			}

			else if(i>Degree2)
			{
				NEW->coef=T1->coef;
				NEW->expo=T1->expo;
				NEW->link=NULL;
				if(P3==NULL)
					P3=END3=NEW;
				else
				{
					END3->link=NEW;
					END3=NEW;
				}
				T1=T1->link;
			}

			else
			{
				NEW->coef=T1->coef+T2->coef;
				NEW->expo=i;
				NEW->link=NULL;
				if(P3==NULL)
					P3=END3=NEW;
```

```
				else
				{
					END3->link=NEW;
					END3=NEW;
				}
				T1=T1->link;
				T2=T2->link;
			}
		}
}
void Polynomial::Display()
{
		PolyNode *Temp=P3;
		cout<<endl;
		while(Temp!=NULL)
		{
			if(Temp->coef!=0&&Temp->link!=NULL)
				cout<<Temp->coef<<" X ^ "<<Temp->expo<<" + ";
			 if(Temp->link==NULL)
				cout<<Temp->coef;
			Temp=Temp->link;
		}
}

void main()
{
		Polynomial obj;
		clrscr();
		obj.AddData();
		obj.Display();
}
```

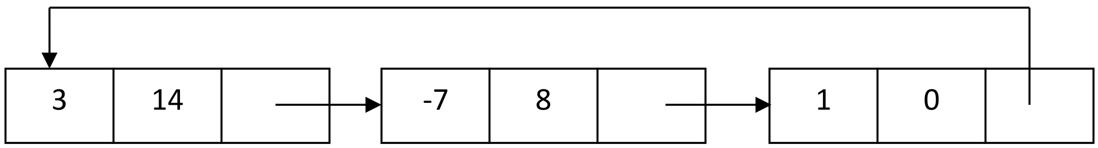
Circular list representation of Polynomials

In chain representation of a polynomial, link part of the last node consist NULL pointer. Whereas, in case of circular list representation link part of the last node is filled with address of the starting node.

Example: Consider the polynomial equation as

$P(x) \quad = \quad 3x^{14} – 7x^{8} + 1$

For this, circular list representation of the polynomial is:



EQUIVALENCE CLASSES

An **equivalence class** is a collection of **equivalence relation** partitions. A relation R over a set S is said to be **equivalence relation** over S if and only if it is reflexive, symmetric and transitive.

Reflexive: Let x ϵ S => (x , x) ϵ R

Symmetric: Let x, y ϵ S

IF x is related to y then y is related to x. i.e., If x R y => y R x

Transitive: Let x, y and z ϵ S

IF x is related to y and y is related to z then x is related to z.

i.e., If x R y and y R z => x R z

Example: Let S = {1, 2, 3, 4, 5, 6} and

R={(1,1), (1,5), (2,2), (2,3), (2,6), (3,2), (3,3), (3,6), (4,4), (5,1), (5,5), (6,2), (6,3), (6,6)}

Find equivalence classes for the above set.

Solution: Given Set S = {1, 2, 3, 4, 5, 6}

1	2	3	4	5	6

The elements related to 1 are: 1 , 5

	2	3	4		6

The elements related to 2 are: 2 , 3 , 6

			4		

The elements related to 4 are: 4

Thus, equivalence classes are = { {1, 5} , {2, 3, 6} , {4} }

Algorithm: The algorithm for equivalence classes is defined in two phases. In the first phase, the equivalence pair (i, j) are added and stored. In second phase, process starts with 1 and finds all pairs of (1, j). Then place the values of 1 and j are in the same class.

```
void equivalence()
{
        Initialize;
        while (there are no more pairs)
        {
                Read the next pair (i, j)
                Process the pair
        }
        Initialize the output
        do
        {
                Output a new equivalence class
        }while(Not done);
}
```

SPARSE MATRIX

A sparse matrix is a matrix representation in which number of zero elements is greater than the number of non-zero elements.

Example:

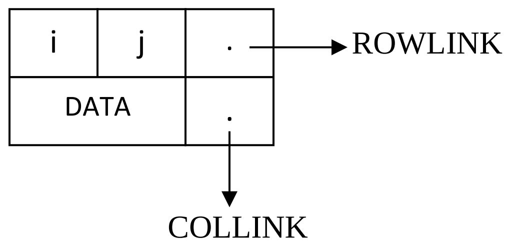
$$\begin{matrix} 0 & 0 & 0 & 0 & 9 & 0 \\ 0 & 8 & 0 & 0 & 0 & 0 \\ 4 & 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 5 \\ 0 & 0 & 2 & 0 & 0 & 0 \end{matrix}$$

Note: Sparse matrix can be represented using Triplet and Linked list.

Linked list Representation of a Sparse Matrix

Format – 1:

In linked implementation of the sparse matrix, each node is divided into different fields such as:



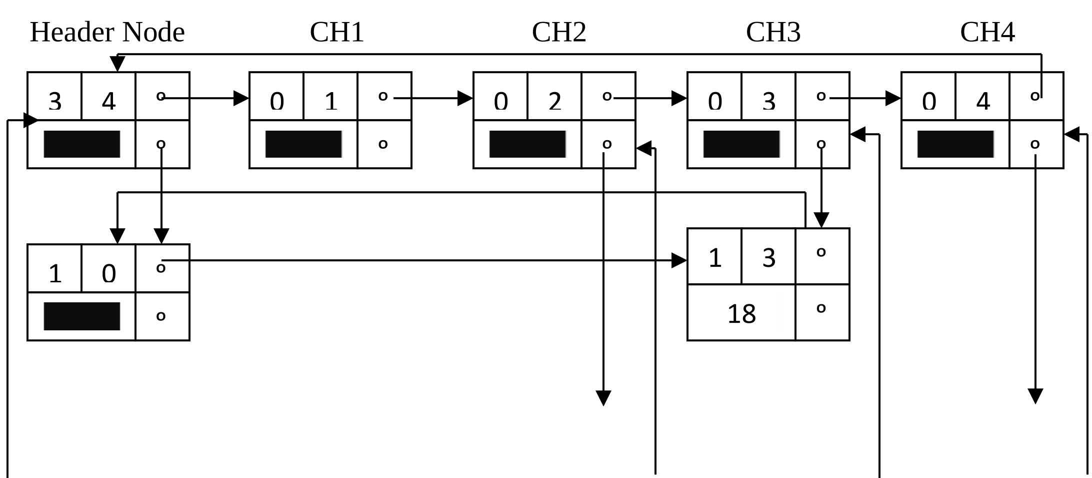
Where,

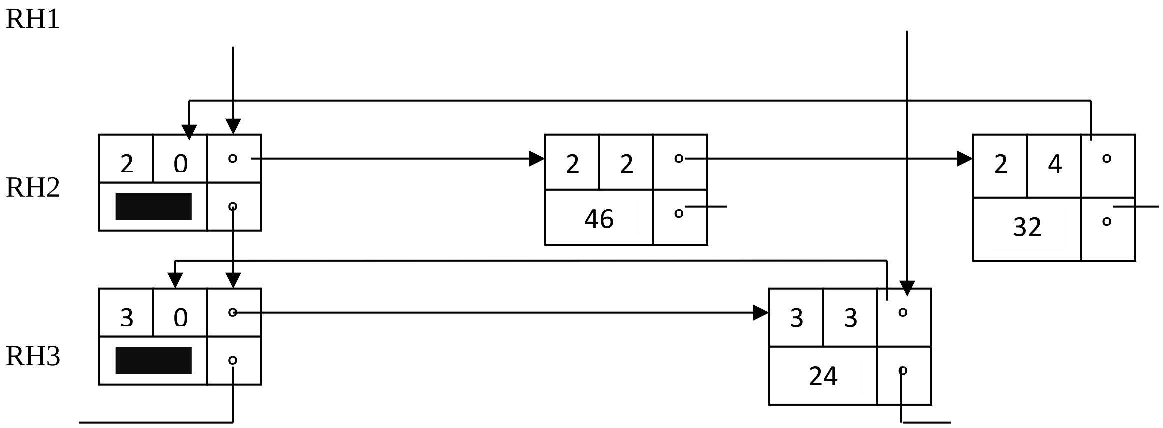
- i and j stores the row and column numbers of the matrix elements
- DATA field stores the matrix element
- ROWLINK points to the next node in the same row and COLLINK points to the next node in the same column of the list
- The basic principle is that, all nodes in a row (column) are circularly linked with each other and each row (column) maintains a header node.

Example: Given sparse matrix is:

0	0	18	0
0	46	0	32
0	0	24	0

For this, linked implementation is:





Here,

CH1, CH2, CH3 and CH4 are column header nodes
RH1, RH2 and RH3 are row header nodes.

Format – 2:

For programming implementation point of view, each node is divided into four fields such as:

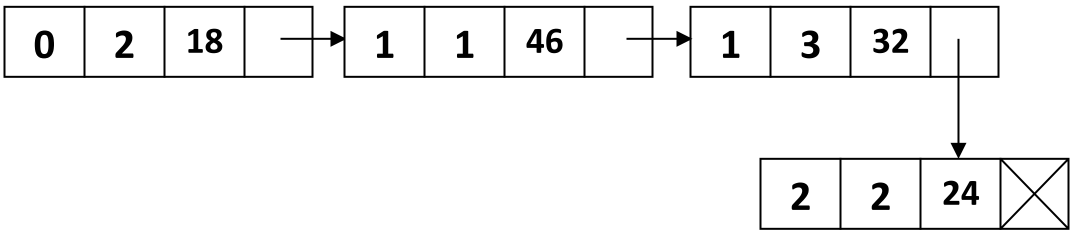
Row	Col	Data	Link

- **Row:** It represents the index of the row where the non-zero element is located.
- **Col:** It represents the index of the column where the non-zero element is located.
- **Data:** It is the value of the non-zero element that is located at the index.
- **Link:** It stores the address of the next node.

Example: Given sparse matrix is:

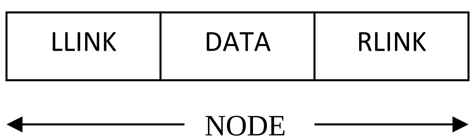
0	0	18	0
0	46	0	32
0	0	24	0

The linked list representation of the above matrix is:



DOUBLE LINKED LIST

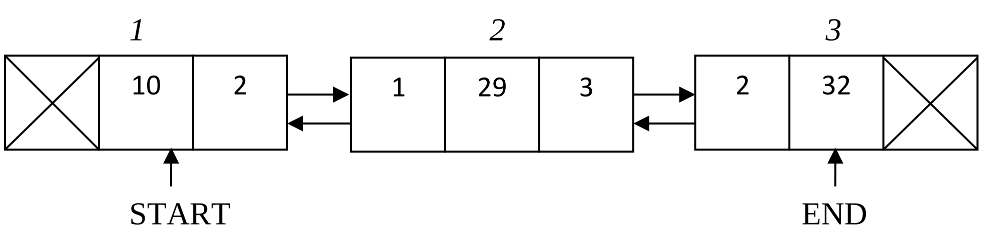
In double linked list, each node is divided into three parts as FLINK/LLINK, INFO/DATA and RLINK.



Here,

The first part FLINK/LLINK consist address of the left node of the list
The second part DATA/INFO consist information part of the element
The third part RLINK field consist address of the right node of the list.

Example:



Here, START and END are two pointer variables that points to beginning and ending nods of the list. The LLINK field of the first node and RLINK field of the last node filled with a special pointer called NULL pointer.

Abstract Data type for the Double linked list can be shown as:

AbstractDataType DList
{

Instances: Finite collection of zero or more elements linked by pointers.

Operations:

creation() : Create a double linked list with specified number of elements.
display() : Display elements of the double linked list.
insertion(): Insert a new element at the specified location.
deletion() : Remove an element from the double linked list.
count() : Returns number of elements of the double linked list.
search() : Search for the existing of a particular element.

}

To implement these operations, create a template class format as:

```
template<class T>
class DList
{
        typedef struct List
        {
                T DATA;
                struct List *LINK;
        }NODE;
        NODE *START,*END;
        public: DList()
                {
                        START=END=NULL;
                }
                void create();
                void display();
                void Finsertion(T);
                void Rinsertion(T);
                void Anyinsertion(T,int);
                T Fdeletion();
                T Rdeletion();
                T Anydeletion();
```

```
        int count();
        int search(T);
};
```

Basic operations performed on the double linked list are:

- i) Creating a list
- ii) Traversing the list
- iii) Insertion of a node into the list
- iv) Deletion of a node from the list
- v) Counting number of elements
- vi) Searching an element etc.,

In double linked list, nodes are created using self-referential structure as:

```
typedef struct list
{
        int DATA;
        struct list *LLINK,*RLINK;
}NODE;
```

Now, create new nodes with the format as: **NODE *NEW;**

i) Creating a list

Creating a list refers to the process of creating nodes of the list and arranges links in between the nodes of the list.

Initially no elements are available in the list. At this stage, set two pointer variables START and END to NULL pointer as:

NODE *START = NULL, *END = NULL;

Algorithm creation (): This procedure creates a double liked list with the specified number of elements.

```
Step 1:    Repeat WHILE TRUE
               READ an element as x
               IF x = -999 THEN
                   RETURN
               ELSE
                   Allocate memory for a NEW node
                   DATA(NEW) ← x
                   LLINK(NEW) ← NULL
                   RLINK(NEW) ← NULL
```

```
                IF START = NULL THEN
                        START ← END ← NEW
                ELSE
                        RLINK(END) ← NEW
                        LLINK(NEW) ← END
                        END ← NEW
                ENDIF
        ENDIF
EndRepeat
```

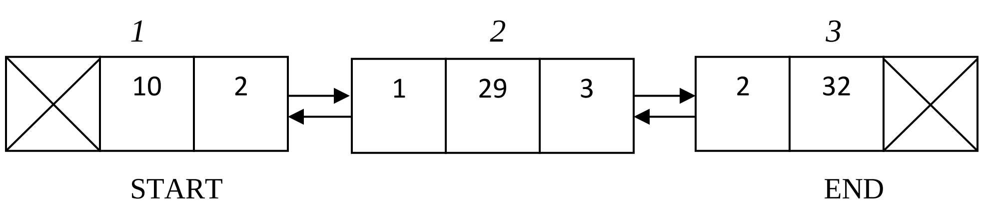
ii) Traversing the list

Traversing the list refers to the process of visiting every node of the list exactly once from the first node to the last node of the list.

Algorithm display(): This procedure is used to display the elements of the double linked list from the first node to the last node in forward direction and then last node to first node in backward direction.

```
Step 1:     IF START = NULL THEN
                    WRITE 'Double Linked List Empty'
            ELSE
                    TEMP ← START
                    Repeat WHILE   TEMP ≠ NULL
                            WRITE   DATA(TEMP)
                            TEMP ← RLINK(TEMP)
                    EndRepeat
                    TEMP ← END
                    Repeat WHILE   TEMP ≠ NULL
                            WRITE   DATA(TEMP)
                            TEMP ← LLINK(TEMP)
                    EndRepeat
            ENDIF
Step 2:     RETURN
```

Example:



Display(): Double linked list elements In

Forward Direction are: 10 29 32

Backward Direction are: 32 29 10

iii) Insertion of a node into the list

The process of inserting a node into the double linked list falls into three cases as:

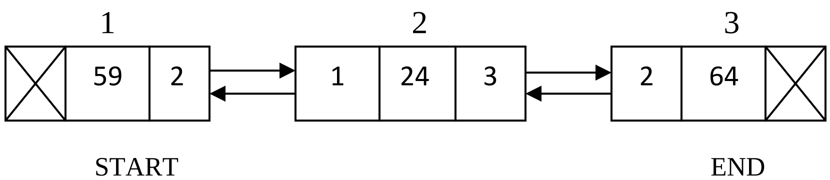
- Front insertion
- Rear insertion
- Any position insertion

Case 1: Front Insertion: In this case, a new node is inserted at front position of the double linked list.

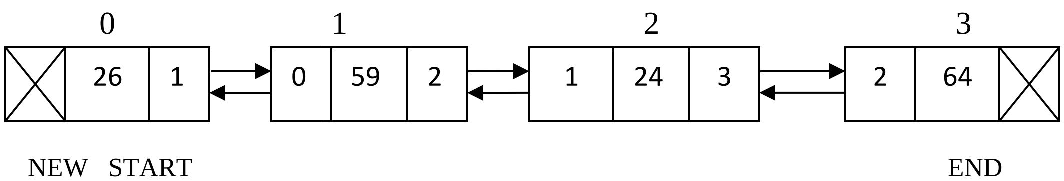
Algorithm Finsertion(x): This procedure inserts an element x at front end of the list.

```
Step 1:     Allocate memory for a NEW node
            DATA(NEW) ← x
            LLINK(NEW) ← NULL
            RLINK(NEW) ← NULL
Step 2:     IF START = NULL THEN
                  START ← END ← NEW
            ELSE
                  RLINK(NEW) ← START
                  LLINK(START) ← NEW
                  START ← NEW
            ENDIF
Step 3:     RETURN
```

Example: Assume initial status of the list as:



Finsertion (26):

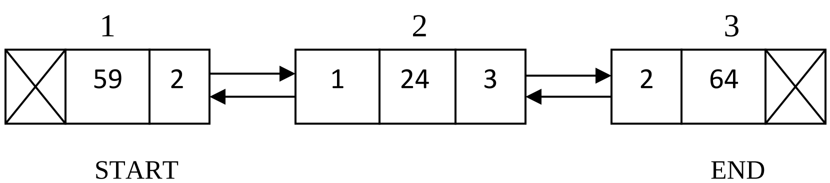


Case 2: Rear Insertion: In this case, a new node is inserted at rear position of the double linked list.

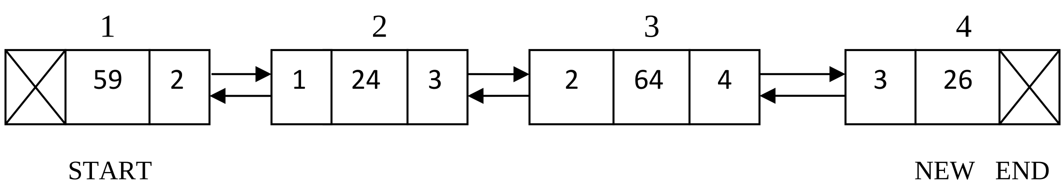
Algorithm Rinsertion(x): This procedure inserts an element x at rear end of the list.

```
Step 1:    Allocate memory for a NEW node
           DATA(NEW) ← x
           LLINK(NEW) ← NULL
           RLINK(NEW) ← NULL
Step 2:    IF END = NULL THEN
                START ← END ← NEW
           ELSE
                RLINK(END) ← NEW
                LLINK(NEW) ← END
                END ← NEW
           ENDIF
Step 3:    RETURN
```

Example: Assume initial status of the list as:



R*insertion (26):*



Case 3: Any Position Insertion: In this case, a new node is inserted at a specified position of the double linked list.

Algorithm Anyinsertion(x, pos): This procedure inserts an element x at the specified position pos of the list.

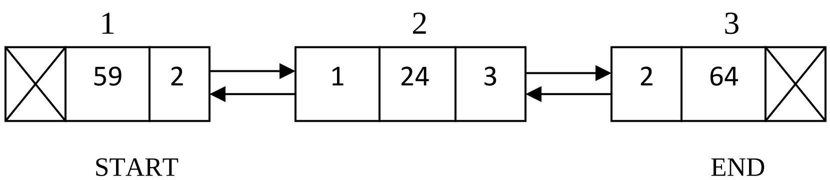
```
Step 1:    Allocate memory for a NEW node
           DATA(NEW) ← x
           LLINK(NEW) ← NULL
           RLINK(NEW) ← NULL
Step 2:    k ← 1
           PTR ← START
Step 3:    Repeat WHILE k < pos
                TEMP ← PTR
                PTR ← RLINK(PTR)
                k ← k+1
```

EndRepeat

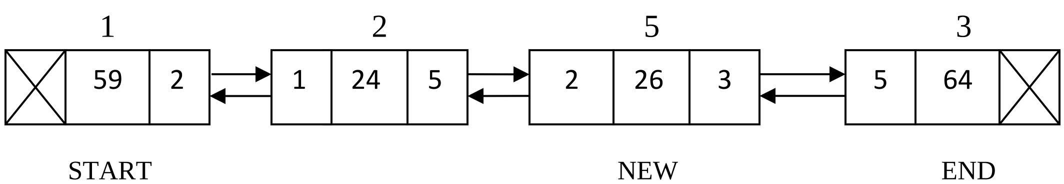
Step 4: RLINK(TEMP) ← NEW
LLINK(NEW) ← TEMP
RLINK(NEW) ← PTR
LLINK(PTR) ← NEW

Step 5: RETURN

Example: Assume initial status of the list as:



Any*insertion (26,3):*



iv) Deletion of a node from the list

The process of deleting an element from the double linked list falls into three categories as:

- Front deletion
- Rear deletion
- Any position deletion

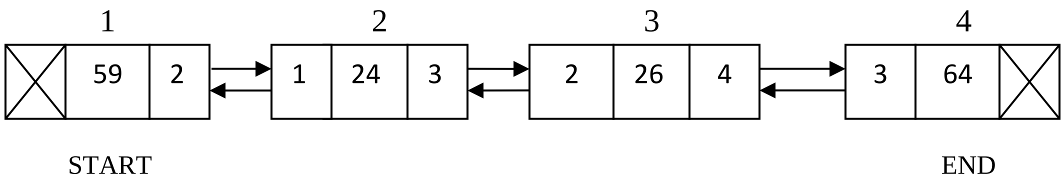
Case 1: Front Deletion: In this case, front node information is deleted from the double linked list.

Algorithm Fdeletion( ): This function deletes the front element of the list.

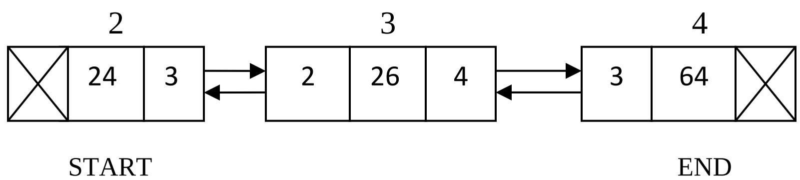
```
Step 1:     IF   START = NULL THEN
                    RETURN   -1
            ELSE
                    k← DATA(START)
                    IF    START = END   THEN
                            START ← END ← NULL
                    ELSE
                            TEMP ←START
                             START ← RLINK(START)
```

```
                LLINK(START) ← NULL
                RLINK(TEMP) ← NULL
                Release memory of TEMP
        ENDIF
        RETURN   k
ENDIF
```

Example: Assume initial status of the list as:



Fdeletion( ): Front Deleted Element = 59

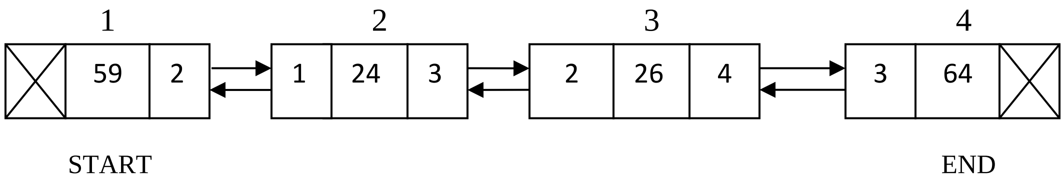


Case 2: Rear Deletion: In this case, rear node information is deleted from the double linked list.

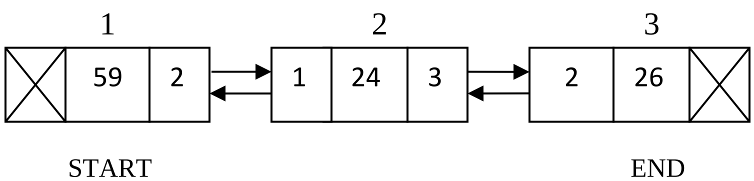
Algorithm Rdeletion( ): This function deletes the rear element of the list.

```
Step 1:     IF    END = NULL THEN
                    RETURN -1
            ELSE
                    k← DATA(END)
                    IF    START = END   THEN
                            START ← END ← NULL
                    ELSE
                            TEMP ← END
                            END ← LLINK(END)
                            RLINK(END) ← NULL
                            LLINK(TEMP) ← NULL
                            Release memory of TEMP
                    ENDIF
                    RETURN   k
            ENDIF
```

Example: Assume initial status of the list as:



Rdeletion( ): Rear Deleted Element = 64



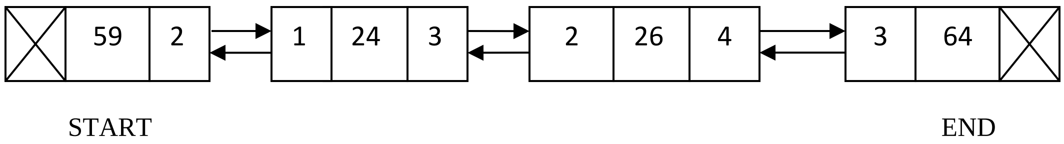
Case 3: Any Position Deletion: In this case, a specified position element is deleted from the double linked list.

Algorithm Anydeletion(pos): This function deletes a specified position 'pos' element of the list.

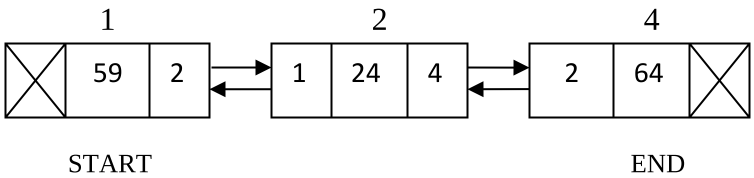
```
Step 1:     IF   START = NULL Then
                 RETURN  -1
            ELSE
                 PTR ← START
                 p ← 1

                 Repeat WHILE   p < pos
                      TEMP ← PTR
                      PTR ← RLINK(PTR)
                      p ← p+1
                 EndRepeat
                 k ← DATA(PTR)
                 RLINK(TEMP) ← RLINK(PTR)
                 LLINK(RLINK(PTR)) ← LLINK(PTR)
                 LLINK(PTR) ← NULL
                 RLINK(PTR) ← NULL
                 Release memory of PTR
                 RETURN   k
            ENDIF
```

Example: Assume initial status of the list as:

1 2 3 4



Anydeletion(3): Deleted Element = 26



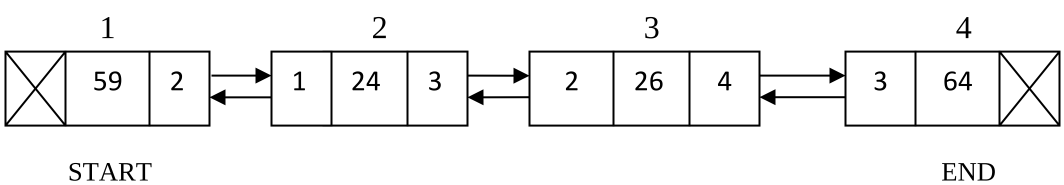
v) Counting number of nodes of the list

In this case, function counts number of nodes exists in the list.

Algorithm count(): This function is used to count number of elements of the list.

```
Step 1:     IF    START = NULL   THEN
                  RETURN   0
            ELSE
                  k ← 0
                  PTR ← START
                  Repeat WHILE    PTR ≠ NULL
                        k ← k+1
                        PTR ← RLINK(PTR)
                  EndRepeat
                  RETURN    k
            ENDIF
```

Example: Assume initial status of the list as:



count(): Number of nodes = 4

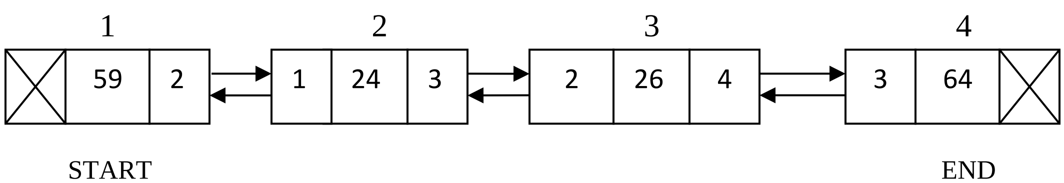
vi) Searching an element

In this case, function checks whether a key element is present in the list of elements or not. If the search element is found it refers to successful search; otherwise, it refers to unsuccessful search.

Algorithm search (key): This function checks whether an element 'key' present in the list of elements or not. It returns 1 if the search element key is found; otherwise, it returns 0.

```
Step 1:     IF    START = NULL   THEN
                  RETURN   0
            ELSE
                  PTR ← START
                  Repeat WHILE    PTR ≠ NULL
                        IF   DATA(PTR) = key   THEN
                              RETURN    1
                        ELSE
                              PTR ← RLINK(PTR)
                        ENDIF
                  EndRepeat
                  RETURN   0
            ENDIF
```

Example: Assume initial status of the list as:



search(64): Element Found

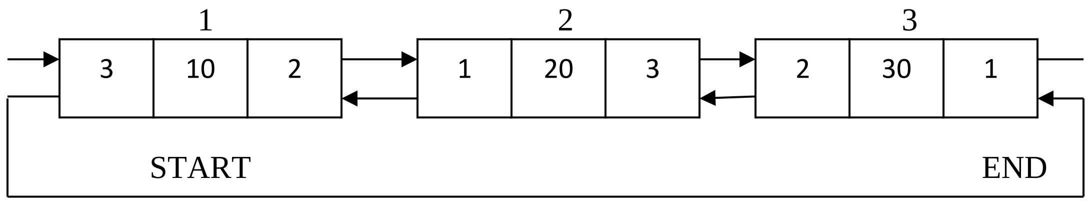
search(12): Element Not Found

Disadvantages:

In doubly linked list, forward and backward directions traversing are possible. But traversing from a specific location is not possible.

CIRCULAR DOUBLE LINKED LIST

In circular double linked list, the RLINK part of the last node contains address of the starting node and LLINK part of the first node contains address of the last node respectively. The diagrammatic representation of a circular double linked list is as follows:



<u>Operations:</u>

Algorithm creation(): This procedure creates a circular double linked with the specified number of elements.

```
Step 1:     Repeat WHILE TRUE
                READ an element as x
                IF x = -999 THEN
                    RETURN
                ELSE
                    Allocate memory for a NEW node
                    DATA(NEW) ← x
                    LLINK(NEW) ← NULL
                    RLINK(NEW) ← NULL

                    IF START = NULL THEN
                        START ← END ← NEW
                    ELSE
                        RLINK(END) ← NEW
                        LLINK(NEW) ← END
                        END ← NEW
                    ENDIF
                    RLINK(END) ← START
                    LLINK(START) ← END

                ENDIF
            EndRepeat
```

Algorithm display(): This procedure is used to display the elements of the circular double linked list from the first node to the last node.

```
Step 1:     IF START = NULL THEN
                WRITE 'Circular Double Linked List Empty'
            ELSE
                TEMP ← START
                Repeat WHILE   RLINK(TEMP) ≠ START
                    WRITE   DATA(TEMP)
                    TEMP ← RLINK(TEMP)
                EndRepeat
                WRITE  DATA(TEMP)
            ENDIF
Step 2:     RETURN
```

HEADER LINKED LIST

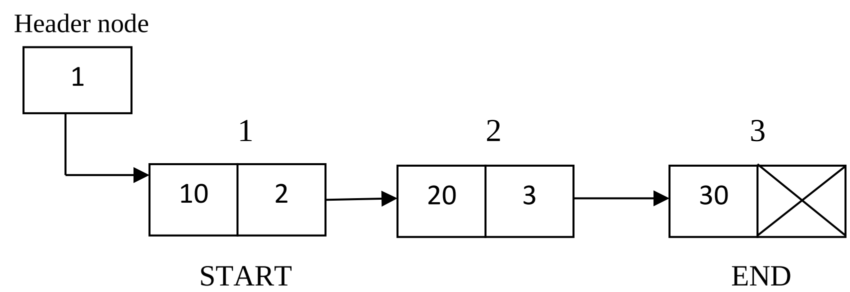
A header linked list is a linked list which always contains a special node called as the "**header node"** at beginning of the list that holds address of the starting node.

Commonly used header linked lists are:

- Grounded header linked list
- Circular header linked list

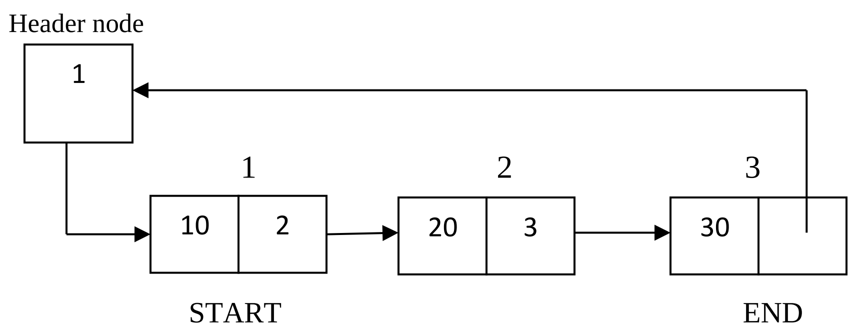
A **grounded header list** is a header linked list where the link part of the last node consists of NULL pointer.

Example:



A **circular header list** is a header linked list where the link part of the last node contains address of the header node i.e., it points back to the header node.

Example:



APPLICATION OF LINKED LISTS

Linked lists are used in different application areas such as sparse matrix manipulations, polynomial representations, stack implementations, queue implementations etc.

GENERALIZED LISTS

A generalized list L is a finite sequence of n elements ($n \geq 0$). The element ei is either an atom (single element) or another generalized list.

Example: Suppose L = ((A, B, C), ((D, E), F), G)

Here, L has three elements sub-list (A, B, C), sub-list ((D, E), F), and atom G. Again sub-list ((D, E), F) has two elements one sub-list (D, E) and atom F.

To represent it in terms of linked list, the node structure view can be shown as:

Flag	Data	DownPointer	Link

To represent a list of items there are certain assumptions about the node structure.

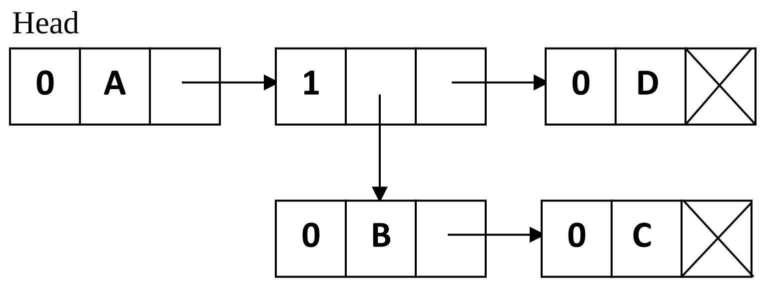
- ✓ Flag = 1 implies that *Down pointer* exists
- ✓ Flag = 0 implies that *Link pointer* exists
- ✓ Data means the atom
- ✓ Down pointer is the address of node which is down of the current node
- ✓ Link pointer is the address of node which is attached as the next node

Node structure format can be defined as:

```
typedef struct node
{
        char c;                     //Data

        int index;                  //Flag

        struct node *next, *down;   //Next & Down pointer

}GLL;
```

Example: Let L = (A, (B,C), D)

For this, list representation is:



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.

<u>Note:</u> 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.

<u>Polynomial Representation using Generalized Linked List</u>

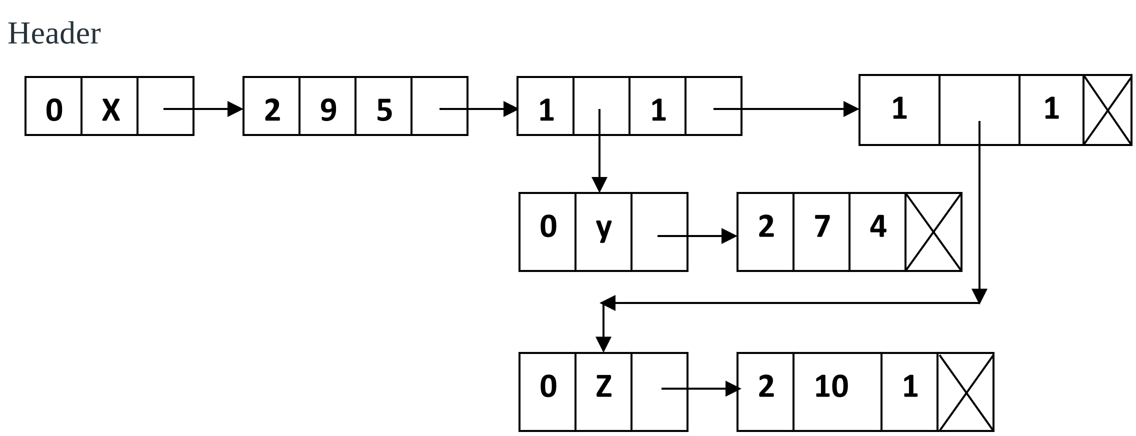
The node structure in this format is:

Flag	Data	DownPointer	Link

Here, Flag value is either 0, 1 or 2.

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

<u>Example:</u> Let $P(X,Y,Z) = 9\ X^5 + 7\ X\ Y^4 + 10\ X\ Z$



RECURSIVE ALGORITHMS FOR LISTS

Consider recursive functions for insertion and traversing operations of the single linked list.

```
typedef   struct List
{
        int DATA;
        struct List *LINK;
}NODE;

Node *  Process(int K)
{
        NODE *P = new NODE;
        P → DATA = K;
        P → LINK = NULL;
        return P;
}
```

Insertion Process

```
Node*    Insertion(NODE *Temp, int K)
{
        if (Temp = = NULL)
                return Process(K);
        else
                Temp → LINK = Insertion(Temp → LINK, K);
        return Temp;
}
```

Traverse Process

```
void Traverse(Node* Temp)
{
        if (Temp = =  NULL)
                return;
        cout << "      "<<Temp → DATA;
        Traverse(Temp → LINK);
}

void main( )
{
        NODE *HEAD = NULL;
        HEAD = Insertion(HEAD, 16);
```

```
        HEAD = Insertion(HEAD,42);
        cout<<" Elements Are = ";
        Traverse(HEAD);
}
```

REFERENCE COUNT

Reference counting is a programming technique of storing the number of references / pointers of an object.

In general number of objects is created at the time of implementation of the program. These objects are utilized at various stages of the program. The most common lifetime for an object is scope-based: the object exists for the duration of a function or bracketed block of code. Thus manual tracking of life time of an object is very difficult. This is particular true when the object is used widely through the program. It would be helpful to have a somewhat automatic lifetime management. Once everything is done using the object it should be deleted. Reference counting is one such technique.

Reference counting is a special mechanism that maintains the count of number of pointers pointed by the object. This method is simply keeping an extra counter along with each object that is created. The counter is the number of references that exist to the object. Anytime a pointer is copied increment the count, and anytime a pointer goes out of scope, decrement the count. When the count hits zero the object is deleted since nothing more is using it.

END