

**COURSE NAME: DATA STRUCTURES THROUGH C**  
**COURSE CODE: 1005172105**

**UNIT-I: Arrays and Linked Lists**

Abstract Data Types ADTs, Dynamic allocation of Arrays, Structures and unions, Polynomials, Spares Matrices Representation of multidimensional Arrays.

Single Linked List and Chains, Representing Chains in C, Polynomials and Polynomial Representation- Adding Polynomials- Circular List Representation of Polynomials, Sparse Matrices, Sparse Matrix Representation. Doubly Linked Lists.

**Unit Objectives:**

- Exposure to the concept of Arrays and Linked List
- Able to know Applications of arrays and linked lists

**Unit Outcomes:**

After learning the unit-1 the students should be able:

1. Differentiate primitive and non primitive data structures.
2. Design and apply appropriate data structures for solving computing problems.
3. Real time applications of arrays and Linked Lists.

S.NO	NAME OF THE TOPIC	PAGENO
1.	Abstract Data Types ADTs	2
2.	Dynamic allocation of Arrays	2
3.	Structures and unions	6
4.	Polynomials	11
5.	Spares Matrices Representation of multidimensional Arrays	19
6.	Single Linked List and Chains	21
7.	Polynomials and Polynomial Representation - Adding Polynomials	29
8.	Circular List Representation of Polynomials	35
9.	Sparse Matrices Sparse Matrix Representation	43
10.	Doubly Linked Lists	53

There are 4 library functions defined under `<stdlib.h>` makes dynamic memory allocation in C programming. They are `malloc()`, `calloc()`, `realloc()` and `free()`.

### malloc():

Syntax :

Pointer=(typecast\*) malloc (number-of-elements\*size-of-each-element);

- It is used to allocate memory dynamically at initial state.
- It creates memory in the form of bytes.
- If the memory is allocated successfully then this function returns first byte address.
- If memory is not allocated it returns NULL
- It places garbage values in memory locations after creation of memory.
- It requires only one argument.

### Example:

```
int *buf;  
  
buf=(int*)malloc(200*sizeof(int));  
  
If(buf == NULL) printf("Not enough memory");
```

It allocates 200 integers. Multiply 200 by sizeof(int).

### free():

Syntax: free(void \*block);

free de-allocates a memory block allocated previously by calloc(), malloc() or realloc().

Dynamically created memory will be stored in heap area of the data segment & life time is entire program.

For example:

```
int *ary;  
  
ary=(int*)malloc(10*sizeof(int));  
  
...  
  
free(ary);
```

### calloc():

Syntax: void (type\*)calloc (size\_t nitems size\_t size);

- It is used to allocate memory dynamically at initial state.
- It creates memory in the form of blocks.
- If the memory is allocated successfully then this function returns first byte address.
- If memory is not allocated it returns NULL.
- It places zeros in memory locations after creation of memory.
- It requires only two arguments.

For example, to allocate 10 integers, the following statement can be used:

```
ary = (int*) calloc (n, sizeof(int));
```

### realloc():

Syntax: realloc(oldpointer, newsize);

- By using realloc () function we can create the memory dynamically at middle stage
- (i.e. reallocating the memory).
- It requires two arguments of type void\* and new size.
- Void \* give previous block of old pointer.
- realloc() creates memory in bytes and initial values are garbage values.

Example:

```
int    *buf;

buf=(int*)malloc(5*sizeof(int));

--

--

buf=(int*)realloc(buf,sizeof(int)*10);

---

free(buf);
```

### **Example: Dynamic allocation for One dimensional array**

```
#include<stdio.h>

#include<conio.h>

void main()

{

int n,*ptr,i;
```

```
clrscr();

printf("enter size of array:");

scanf("%d",&n);

ptr=(int*)malloc(n*sizeof(int));

if(ptr==NULL)

{

printf("not allocated mem");

}

for(i=0;i<n;i++)

{

scanf("%d",ptr+i);

}

printf("elements are:");

for(i=0;i<n;i++)

{

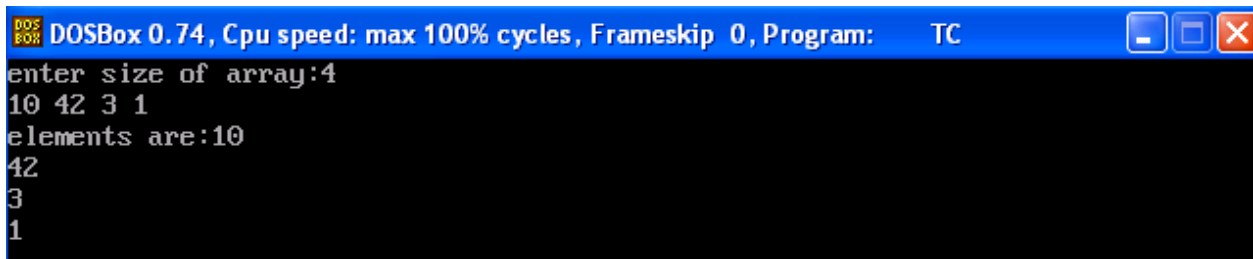
printf("%d\n",*(ptr+i));

}

getch();

}
```

OUTPUT:



DOSBox 0.74, Cpu speed: max 100% cycles, Frameskip 0, Program: TC

```
enter size of array:4
10 42 3 1
elements are:10
42
3
1
```

## STRUCTURES AND UNIONS:

A structure can be considered as a template used for defining a collection of variables under a single name. Structures help programmers to group elements of different data types into a single logical unit . This usually occurs just after the main() statement in a file.

we can define a structure called date with three elements day, month and year.

The syntax of this structure is as follows:

```
struct    tag_name
{
data type member1;
data type member2;
...
...
} ;
```

Example:

```
struct lib_books
{
char title[20];
char author[15];
int pages;
float price;
}; struct lib_books, book1, book2, book3;
```

structures do not occupy any memory until it is associated with the structure variable such as book1. the template is terminated with a semicolon. While the entire declaration is considered as a statement, each member is declared independently for its name and type in a separate statement inside the template.

The link between a member and a variable is established using the member operator ‘.’ Which is known as dot operator or period operator.

Accessing Structures:

For example define a complex type structure as follows.

```
struct complex {
    double real ;
    double imaginary ;           // Note that a variable may also be
    } cplx ;                     // defined at structure definition time
```

The elements of the structure are accessed using the dot operator, . , as follows

```
cplx.real = 10.0 ;  
  
    cplx.imag = 20.23 ;  
  
    scanf ( "%lf", &cplx.real ) ;
```

Example:

```
#include <stdio.h>  
  
#include <string.h>  
  
struct Books  
{  
    char title[50];  
    char author[50];  
    char subject[100];  
    int book_id;  
};  
  
int main( )  
{  
    struct Books Book1;    /* Declare Book1 of type Book */  
    struct Books Book2;    /* Declare Book2 of type Book */  
  
    /* book 1 specification */  
    strcpy( Book1.title, "C Programming");  
    strcpy( Book1.author, "Nuha Ali");  
    strcpy( Book1.subject, "C Programming Tutorial");  
    Book1.book_id = 6495407;  
  
    /* book 2 specification */
```

```
strcpy( Book2.title, "Telecom Billing");  
strcpy( Book2.author, "Zara Ali");  
strcpy( Book2.subject, "Telecom Billing Tutorial");  
Book2.book_id = 6495700;
```

```
/* print Book1 info */  
printf( "Book 1 title : %s\n", Book1.title);  
printf( "Book 1 author : %s\n", Book1.author);  
printf( "Book 1 subject : %s\n", Book1.subject);  
printf( "Book 1 book_id : %d\n", Book1.book_id);  
/* print Book2 info */  
printf( "Book 2 title : %s\n", Book2.title);  
printf( "Book 2 author : %s\n", Book2.author);  
printf( "Book 2 subject : %s\n", Book2.subject);  
printf( "Book 2 book_id : %d\n", Book2.book_id);  
return 0;  
}
```

Output:

Book 1 title : C Programming

Book 1 author : Nuha Ali

Book 1 subject : C Programming Tutorial

Book 1 book\_id : 6495407

Book 2 title : Telecom Billing

Book 2 author : Zara Ali

Book 2 subject : Telecom Billing Tutorial

Book 2 book\_id : 6495700



### Unions:

A union is a special data type available in C that allows to store different data types in the same memory location. You can define a union with many members, but only one member can contain a value at any given time. Unions provide an efficient way of using the same memory location for multiple-purpose.

Syntax:

```
union [union tag]
{
    member definition;
    member definition;
    ...
    member definition;
} [one or more union variables];
```

Example :-

```
Union Data {
    int i;
    float f;
    char str[20];
} data;
```

### Accessing Union Members:

To access any member of a union, we use the member access operator (.). The member access operator is coded as a period between the union variable name and the union member that we wish to access. You would use the keyword union to define variables of union type.

Example:

```
#include <stdio.h>
#include <string.h>
```

```
union Data {
```

```
int i;

float f;

char str[20];

};

int main( ) {

    union Data data;

    data.i = 10;

    data.f = 220.5;

    strcpy( data.str, "C Programming");

    printf( "data.i : %d\n", data.i);

    printf( "data.f : %f\n", data.f);

    printf( "data.str : %s\n", data.str);

    return 0;

}
```

Output:

data.i : 1917853763

data.f : 4122360580327794860452759994368.000000

data.str : C Programming

Difference between structure and union in C	
structure	union
Keyword struct defines a structure.	Keyword union defines a union.
Example structure declaration:  struct s_tag  {	Example union declaration:  union u_tag  {

<pre>int ival;  float fval;  char *cptr;  }s;</pre>	<pre>int ival;  float fval;  char *cptr;  }u;</pre>
<p>Within a structure all members gets memory allocated and members have addresses that increase as the declarators are read left-to-right. That is, the members of a structure all begin at different offsets from the base of the structure. The offset of a particular member corresponds to the order of its declaration; the first member is at offset 0. The total size of a structure is the sum of the size of all the members or more because of appropriate alignment.</p>	<p>For a union compiler allocates the memory for the largest of all members and in a union all members have offset zero from the base, the container is big enough to hold the <b>WIDEST</b> member, and the alignment is appropriate for all of the types in the union. When the storage space allocated to the union contains a smaller member, the extra space between the end of the smaller member and the end of the allocated memory remains unaltered.</p>
<p>Within a structure all members gets memory allocated; therefore any member can be retrieved at any time.</p>	<p>While retrieving data from a union the type that is being retrieved must be the type most recently stored. It is the programmer's responsibility to keep track of which type is currently stored in a union; the results are implementation-dependent if something is stored as one type and extracted as another.</p>
<p>One or more members of a structure can be initialized at once.</p>	<p>A union may only be initialized with a value of the type of its first member; thus union u described above (during example declaration) can only be initialized with an integer value</p>

### Polynomials:

Definition: A Polynomial is an expression that contains more than two terms. A term is made up of coefficient and exponent. For example of polynomial equation is  $P(x)=4x^3+6x^2+9x+7$ . In this example 4,6,9,7 are the coefficients and 3,2,1,0 are the exponents.

### Polynomial as an Abstract Data Type :

Abstract Data Type *Polynomial*

{

Instance: The resultant data is stored in the form of equation i.e. Exponent and coefficient.

Operations:

Addition(poly1,poly2) : It performs addition of two given polynomials poly1,poly2 and result will be stored in another polynomial.

Subtraction(poly1,poly2) : It performs subtraction of two given polynomials poly1,poly2 and result will be stored in another polynomial.

Multiply(poly1,poly2) : It performs multiplication of two given polynomials poly1,poly2 and result will be stored in another polynomial.

Divide(poly1,poly2) : It performs division of two given polynomials poly1,poly2 and result will be stored in another polynomial.

}

Polynomial equation can represent in two ways:

1. Array Representation
2. Linked Representation

### Array Representation:

- Polynomial Representation using an Array: (Method-1)

Polynomial equation can be stored in an array. But here the question is how to store in an array. An array representation assumes that the exponents of the given expression are arranged from 0 to the highest degree, which is represented by the subscript of an array beginning with 0. The coefficients of any exponent are stored inside the array at a particular index. i.e. The coefficient of exponent 2 is stored in 2<sup>nd</sup> index in an array.

For example

$$P(x)=4x^3+6x^2+9x+7$$

An array representation of the above polynomial is

0	1	2	3
7	9	6	4

The coefficient for exponent 3 is 4 and it is placed in 3<sup>rd</sup> index of an array.

This method suffering with drawback of waste of space. So we move to second method.

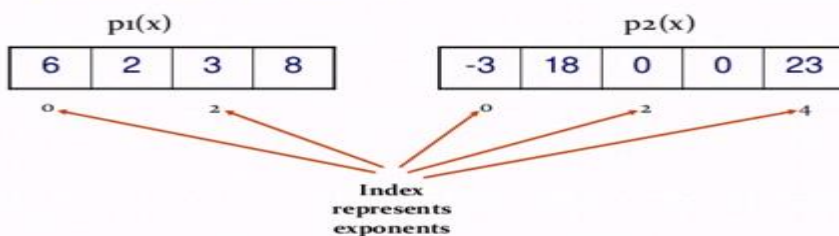
- Polynomial Representation using an Array: (Method-2)

Array representation using two polynomials

• **Array Implementation:**

•  $p_1(x) = 8x^3 + 3x^2 + 2x + 6$

•  $p_2(x) = 23x^4 + 18x - 3$



We can implement polynomials using arrays. So here is the program which demonstrates how to add 2 polynomials using arrays.

To add two polynomials

1. The first input array represents " $5 + 0x^1 + 10x^2 + 6x^3$ "
2. The second array represents " $1 + 2x^1 + 4x^2$ "
3. And Output is " $6 + 2x^1 + 14x^2 + 6x^3$ "

### Linked Representation:

Polynomial Representation Using Linked List:

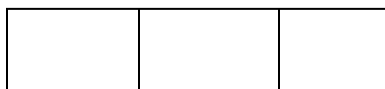
An important application of linked list is to represent polynomial and their manipulations. Main advantage of linked list for polynomial representation is that it can accommodate a number of polynomials of growing sizes so that combined size does not exceed the total memory available. The structure of a node in order to represent a term is as shown below:

```

struct polynomial
{
    int coefficient;
    int exponent;
    struct polynomial *next;
};
    
```

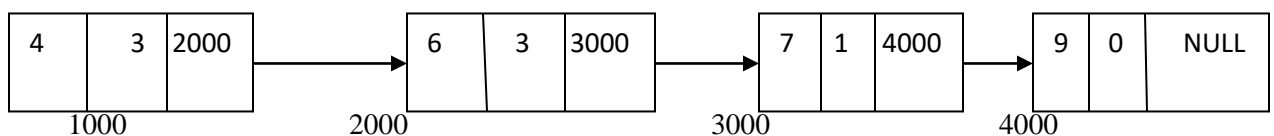
$P(x) = 4x^3 + 6x^2 + 7x + 9$

The above polynomial can be represented in linked list form as follows:



↓                      ↓                      ↓  
 Coefficient          exponent          next

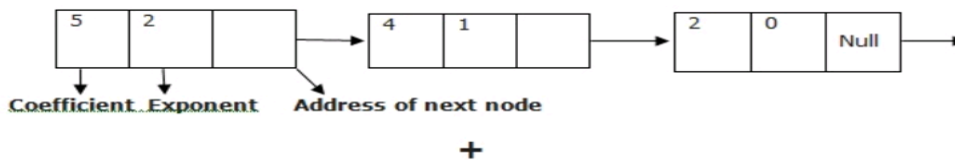
Example:



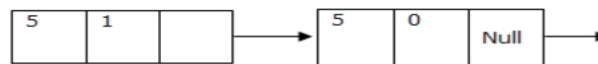
Addition of Two Polynomial :

For adding two polynomials using arrays is straightforward method, since both the arrays may be added up element wise beginning from 0 to n-1, resulting in addition of two polynomials. Addition of two polynomials using linked list requires comparing the exponents, and wherever the exponents are found to be same, the coefficients are added up. For terms with different exponents, the complete term is simply added to the result thereby making it a part of addition result.

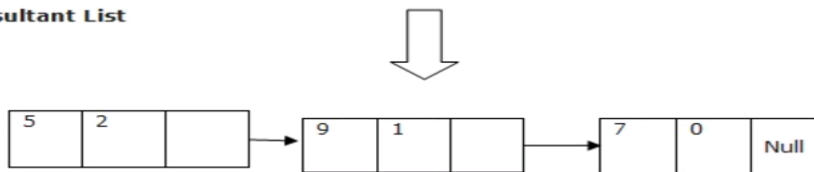
**List 1**



**List 2**



**Resultant List**



**Operations On polynomials:**

- Evaluation of polynomial
- Addition

### Evaluation of polynomial:

Evaluation of polynomial means , simplifying polynomial expression by the values of x.

Ex:

**Evaluate Polynomial**

$$7x^2 - 12x + 13 \quad \text{when } x=4$$

variable

**Solution :**

$$\begin{aligned} &= 7(4)^2 - 12(4) + 13 \\ &= 7(16) - 12(4) + 13 \\ &= 112 - 48 + 13 \\ &= 64 + 13 \\ &= 77 \end{aligned}$$

Program:

```
#include<stdio.h>
#include<math.h>
struct poly
{
    int coef;
    int exp;
};
void main()
{
    int x,n,s=0,I;
```



```
struct poly p[10];  
printf("enter no of elememts :");  
scanf("%d",&n);  
printf("enter coefficient & exponent values:");  
for(i=0;i<n;i++)  
{  
    scanf("%d%d",&p[i].coef,&p[i].exp);  
}  
printf("enter value of x");  
scanf("%d",&x);  
for(i=0;i<n;i++)  
{  
    s=s+p[i].coef*(pow(x,p[i].exp));  
}  
printf("evaluation of expression is %d",&s);  
}
```

### Polynomial Addition:

**Polynomial Addition**

Sum of two polynomials, we need only add the coefficient of equal powers. The constant terms should also be added. Adding polynomials is just a matter of combining like terms, with some order of operations considerations thrown in.

Eg:     Simplify  $(2x + 3) + (4x + 6)$   
               $= 2x + 3 + 4x + 6$   
               $= 2x+4x+9$   
               $= 6x+9$

For adding two polynomials compare exponent values polynomials.

Case 1: If both exponents are same then add both coefficients and copy it into resultant polynomial.

Case 2: If polynomial1 exponent is greater than polynomial2 then simply copy polynomial1 into resultant polynomial.



Case 3: If polynomial2 exponent is greater than polynomial1 then simply copy polynomial2 into resultant polynomial.

Case 4: If any value is left in polynomial1 then compare index value of that polynomial with no.of elements in that polynomial and copy it into resultant polynomial.

Case 5: If any value is left in polynomial2 then compare index value of that polynomial with no.of elements in that polynomial and copy it into resultant polynomial.

Algorithm:

1. Read two polynomials say A and B

2. Let M and N denote total terms in A and B respectively. Here, C is resultant polynomial.

3. Let  $i = j = k = 0$

$i$  is index of Polynomial1 i.e A

$j$  is index of Polynomial1 i.e B

$k$  is index of Polynomial1 i.e C

4. while ( $i < M$  and  $j < N$ ) do

begin // repeat till one of the polynomials is copied

if( $A[i].Exp = B[j].Exp$ )

begin

$C[k].Coef = A[i].Coef + B[j].Coef$

$C[k].Exp = A[i].Exp;$

$i = i + 1;$

$j = j + 1,$

$k = k + 1$

end

else if( $A[i].Exp > B[j].Exp$ )

begin

$C[k].Coef = A[i].Coef;$

$C[k].Exp = A[i].Exp;$

```
i = i + 1  
k = k + 1  
end  
else  
begin  
    C[k].Coef = B[j].Coef;  
    C[k].Exp = B[j].Exp;  
    j = j + 1  
    k = k + 1  
end  
end  
5.while(i < m) do  
    begin // copy remaining terms  
        C[k].Coef = A[i].Coef;  
        C[k].Exp = A[i].Exp;  
        i = i + 1  
        k = k + 1  
    end  
6.while (j < n) do  
    begin // copy remaining terms  
        C[k].Coef = B[j].Coef;  
        C[k].Exp = B[j].Exp;  
        j = j + 1  
        k = k + 1  
    end  
7. stop
```

## SPARSE MATRICES REPRESENTATION USING MULTI DIMENSIONAL ARRAYS:

**Definition:** In computer programming a matrix can be defined with a 2-dimensional array. Any array with m columns and n rows represents a  $m \times n$  matrix. There may be a situation in which a matrix contains more number of ZERO values than NON-ZERO values. Such matrix is known as Sparse Matrix. Sparse Matrix is matrix which contains very few non-zero elements.

A matrix is a two-dimensional data object made of m rows and n columns, therefore having total  $m \times n$  values. If most of the elements of the matrix have **0 value**, then it is called a sparse matrix.

### Why to use Sparse Matrix instead of simple matrix ?

**Storage:** There are lesser non-zero elements than zeros and thus lesser memory can be used to store only those elements.

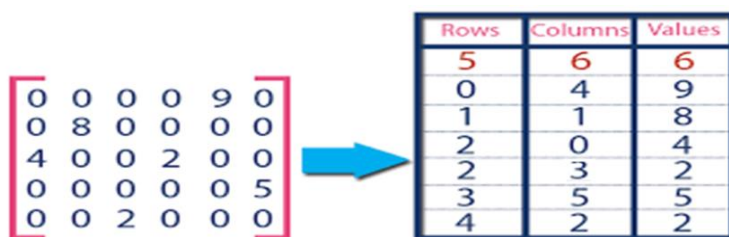
**Computing time:** Computing time can be saved by logically designing a data structure traversing only non-zero elements..

Sparse Matrix can be represented in two ways:

- Array Representation/Triplet Representation
- Linked Representation

### Triplet Representation of Sparse Matrix

In this representation, we consider only non-zero values along with their row and column index values. In this representation, the 0<sup>th</sup> row stores total rows, total columns and total non-zero values in the matrix.



Rows	Columns	Values
5	6	6
0	4	9
1	1	8
2	0	4
2	3	2
3	5	5
4	2	2

In above example matrix, there are only 6 non-zero elements (those are 9, 8, 4, 2, 5, 2) and matrix size is  $5 \times 6$ . we represent this matrix as shown in the above image. Here the first row in the right side table is filled with values 5, 6, 6 which indicates that it is a Sparse Matrix with 5 rows and 6 columns and 6 non-zero elements. Second row is filled with 0, 4, 9 which indicates the value in the matrix at 0<sup>th</sup> row and 4<sup>th</sup> column is 9. In the same way the remaining non-zero values also follows the similar pattern.

Sparse matrix program:

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

```
int main()
{
    int a[10][10],b[10][3],i,j,k=0,m,n;
    printf("enter row size and column size of array:");
    scanf("%d%d",&m,&n);
    for(i=0;i<m;i++)
    {
        for(j=0;j<n;j++)
        {
            scanf("%d",&a[i][j]);
        }
        printf("\n");
    }
    printf("before triplet form:\n");
    for(i=0;i<m;i<n)
    {
        for(j=0;j<n;j++)
        {
            printf("%d",a[i][j]);
        }
    }
    printf("\n");
    for(i=0;i<m;i++)
    {
        for(j=0;j<n;j++)
        {
```

```
if(a[i][j]!=0)
{
    b[k][0]=i;
    b[k][1]=j;
    b[k][2]=a[i][j];
    k++;
}
}
}
printf("Triplet form is :\n");
for(i=0;i<k;i++)
{
    for(j=0;j<3;j++)
    {
        printf("%d",b[i][j]);
    }
    printf("\n");
}
```

### SINGLE LINKED LIST AND CHAINS:

A **linked list** is a way to store a collection of elements. Like an array these can be character or integers. Each element in a linked list is stored in the form of a **node**.

A node is a collection of two sub-elements or parts. A **data** part that stores the element and a **next** part that stores the link to the next node.

Linked List:



Creating structure for a node:

Struct node

```
{  
int data;  
Struct node *link;  
};
```

Creating memory for the node:

A block of memory can be allocated at runtime by using malloc function.

*Struct node \*ptr;*

*Ptr=(struct node\*) malloc (sizeof (struct node));*

Explanation:

**Sizeof:** it returns the number of bytes needed for the node.

**Malloc:** it allocates block of memory returns void pointer. In this block we can store any type of data.

(struct node\*): the created block is type-casted to pointer to be of type node.

The members of a structure can be accessed by using pointer variable through arrow operator. The symbol -> is a minus sign followed by the greater than symbol.

Operations on singly linked lists:

- Inserting node into the list
- Deleting node from the list
- Display the contents of list

**Insertion:**

Element can be **inserted at the beginning** of list or end of list or middle of the list. Create a new node and the address of this is stored in pointer variable Head. Store the starting address of a list into Head. So Head contains the address of the starting node of the list. **For end insertion** store the address of the new

node into the next field of last node. First **find the element position to be inserted** in the list. Then insert the link between the nodes.

#### **Deletion:**

- **Front deletion**

Store the address of second node in the pointer variable start. The address of first node is lost. When address is lost element is deleted. It is front deletion.

- **End Deletion**

Store the null value in the next field of second last node in the list. This is end deletion.

- **Middle deletion**

First find the element to be deleted in the list. Then remove the link between the nodes.

#### **Display:**

Display is nothing but printing the contents of the list. Start is a pointer variable, which always contains the address of Head node. Through this pointer variable we can access any element in the list. We take one temporary pointer variable, which also stores the address of a first node. Through this pointer variable will print the contents of each node by every time shifting the temp to the next node.

Program:

```
#include<stdio.h>

#include<conio.h>

#include<stdlib.h>

struct node
{
    int data;
    struct node *next;
}*start=NULL,*p,*q,*temp;

typedef struct node sll;

void insbeg();

void inspos();

void insend();
```

```
void delbeg();
void delpos();
void delend();
void display();
void main()
{
    int ch; clrscr();
    printf("1.insbeg..2.inspos..3.insend..4.delbeg..5.delpos..6.delend..7.display..8.exit..\n");
    while(1)
    {
        printf("enter your choice:");
        scanf("%d",&ch);

        switch(ch)
        {
            case 1:insbeg();break;
            case 2:inspos();break;
            case 3:insend();break;
            case 4:delbeg();break;
            case 5:delpos();break;
            case 6:delend();break;
            case 7:display();break;
            case 8:exit(1);break;
            default:printf("wrong choics entered\n");
        }
    }
}
```



```
}
```

```
void insbeg()
```

```
{
```

```
    int val;
```

```
    temp=(sll *)malloc(sizeof(sll ));
```

```
    printf("enter data into node:");
```

```
    scanf("%d",&val);
```

```
    temp->data=val;
```

```
    q=start;
```

```
        while(q->next!=NULL)
```

```
        {
```

```
            q=q->next;
```

```
        }
```

```
    temp->next=start;
```

```
    start=temp;
```

```
    q->next=NULL;
```

```
}
```

```
void insend()
```

```
{    int val;
```

```
    temp=(sll *)malloc(sizeof(sll));
```

```
    printf("enter data into last:");
```

```
    scanf("%d",&val);
```

```
    temp->data=val;
```

```
q=start;

while(q->next!=NULL)
{
    q=q->next;
}

q->next=temp;
temp->next=NULL;
}

void inspos()
{
    int i,pos,val;
    printf("enter position to insert:");
    scanf("%d",&pos);
    printf("enter data to insert in %d position",pos);
    scanf("%d",&val);
    temp=(sll*)malloc(sizeof(sll));
    temp->data=val;
    q=start;
    for(i=1;i<pos-1;i++)
    {
        q=q->next;
    }
    p=q->next;
    q->next=temp;
    temp->next=p;
}
```

```
void display()
{
    q=start;
    while(q!=NULL)
    {
        printf("%d->",q->data);
        q=q->next;
    }
    printf("NULL\n");
}

void delbeg()
{
    q=start;
    start=q->next;
    free(q);
}

void delpos()
{
    int pos,i;
    printf("enter position to delete:");
    scanf("%d",&pos);
    q=start;
    for(i=1;i<pos-1;i++)
    {
        q=q->next;
    }
    temp=q->next;
```

```
q->next=temp->next;
free(temp);
}
void delend()
{
    sll *last;
    q=start;
    while(q->next->next!=NULL)
    {
        q=q->next;
    }
    last=q->next;
    q->next=NULL;
    free(last);
}
```

OUTPUT:

```

DOSBox 0.74, Cpu speed: max 100% cycles, Frameskip 0, Program: TC
1.insbeg..2.inspos..3.insend..4.delbeg..5.delpos..6.delend..7.display..8.exit..
enter your choice:1
enter data into node:30
enter your choice:1
enter data into node:20
enter your choice:1
enter data into node:10
enter your choice:7
10->20->30->NULL
enter your choice:2
enter position to insert:3
enter data to insert in 3 position25
enter your choice:7
10->20->25->30->NULL
enter your choice:3
enter data into last:40
enter your choice:7
10->20->25->30->40->NULL
enter your choice:4
enter your choice:7
20->25->30->40->NULL
enter your choice:6
enter your choice:7
20->25->30->NULL
enter your choice:5
enter position to delete:2
enter your choice:7
20->30->NULL
enter your choice:8
  
```

### POLYNOMIALS REPRESENTATION USING LINKED LIST:

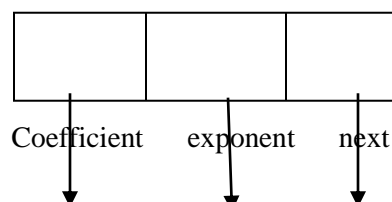
An important application of linked list is to represent polynomial and their manipulations. Main advantage of linked list for polynomial representation is that it can accommodate a number of polynomials of growing sizes so that combined size does not exceed the total memory available. The structure of a node in order to represent a term is as shown below:

```

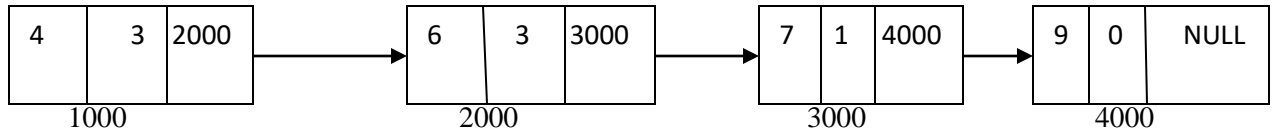
struct polynomial
{
    int coefficient;
    int exponent;
    struct polynomial *next;
};
  
```

$$P(x) = 4x^3 + 6x^2 + 7x + 9$$

The above polynomial can be represented in linked list form as follows:



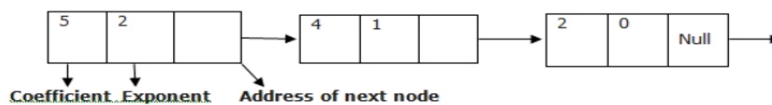
Example:



### Addition of Two Polynomial:

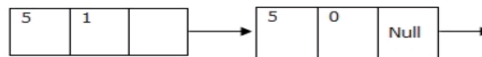
For adding two polynomials using arrays is straightforward method, since both the arrays may be added up element wise beginning from 0 to n-1, resulting in addition of two polynomials. Addition of two polynomials using linked list requires comparing the exponents, and wherever the exponents are found to be same, the coefficients are added up. For terms with different exponents, the complete term is simply added to the result thereby making it a part of addition result.

List 1

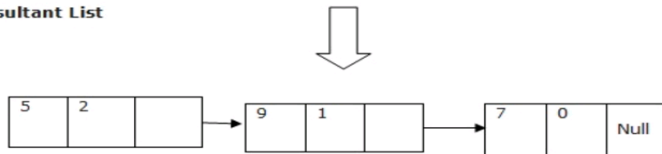


+

List 2



Resultant List



Program:

// using Linked Lists

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

```
struct poly
```

```
{
```

```
    int coeff;
```

```
    int exp;
```

```
    struct poly *next;
```

```
};
```

```
void create(int x, int y, struct poly *temp)
{
    struct poly *r, *z;
    z = temp;
    if(z == NULL)
    {
        r=(struct poly*)malloc(sizeof(struct poly));
        r->coeff = x;
        r->pow = y;
        *temp = r;
        r->next = (struct poly*)malloc(sizeof(struct poly));
        r = r->next;
        r->next = NULL;
    }
    else
    {
        r->coeff = x;
        r->pow = y;
        r->next = (struct Node*)malloc(sizeof(struct Node));
        r = r->next;
        r->next = NULL;
    }
}

void polyadd(struct Node *poly1, struct Node *poly2, struct Node *poly)
{

```

```
while(poly1->next && poly2->next)
{
    // If power of 1st polynomial is greater then 2nd, then store 1st as it is
    // and move its pointer
    if(poly1->pow > poly2->pow)
    {
        poly->pow = poly1->pow;
        poly->coeff = poly1->coeff;
        poly1 = poly1->next;
    }

    // If power of 2nd polynomial is greater then 1st, then store 2nd as it is
    // and move its pointer
    else if(poly1->pow < poly2->pow)
    {
        poly->pow = poly2->pow;
        poly->coeff = poly2->coeff;
        poly2 = poly2->next;
    }

    // If power of both polynomial numbers is same then add their coefficients
    else
    {
        poly->pow = poly1->pow;
        poly->coeff = poly1->coeff+poly2->coeff;
        poly1 = poly1->next;
```



```
poly2 = poly2->next;
}

// Dynamically create new node
poly->next = (struct Node *)malloc(sizeof(struct Node));
poly = poly->next;
poly->next = NULL;
}

while(poly1->next || poly2->next)
{
    if(poly1->next)
    {
        poly->pow = poly1->pow;
        poly->coeff = poly1->coeff;
        poly1 = poly1->next;
    }
    if(poly2->next)
    {
        poly->pow = poly2->pow;
        poly->coeff = poly2->coeff;
        poly2 = poly2->next;
    }
    poly->next = (struct Node *)malloc(sizeof(struct Node));
    poly = poly->next;
    poly->next = NULL;
}
```

```
}

// Display Linked list
void show(struct Node *node)
{
while(node->next != NULL)
{
printf("%dx^%d", node->coeff, node->pow);
node = node->next;
if(node->next != NULL)
printf(" + ");
}
}

int main()
{
struct poly *poly1 = NULL, *poly2 = NULL, *poly = NULL;

// Create first list of 5x^2 + 4x^1 + 2x^0
create(5,2,&poly1);
create(4,1,&poly1);
create(2,0,&poly1);
create (5,1,&poly2);
create (5,0,&poly2);
printf("1st Number: ");
show(poly1);
```

```
printf("\n2nd Number: ");  
  
show(poly2);  
  
poly = (struct Node *)malloc(sizeof(struct Node));  
  
polyadd(poly1, poly2, poly);  
  
// Display resultant List  
  
printf("\nAdded polynomial: ");  
  
show(poly);  
  
  
return 0;  
  
}
```

Output:

1st Number:  $5x^2 + 4x^1 + 2x^0$

2nd Number:  $5x^1 + 5x^0$

Added polynomial:  $5x^2 + 9x^1 + 7x^0$

## **CIRCULAR LIST REPRESENTATION OF POLYNOMIALS:**

### **Circular Linked list:**

A circular linked is one in which the next address of the last node is connected back to the first node.

In circular linked list there is no node called start and no node as last, because it is in a circular fashion. Circular linked list have certain advantages over singly linked lists.

The first of these is concerned with the accessibility of a node. In circular linked list every node is accessible from a given node.

Second advantage is deletion operation.

Disadvantages of circular linked list are, in circular list there is no particular start and end, so we have to take care. Otherwise it leads to infinite loop.

Stack and queue can be implemented using circular linked list.

Header node:

- In the list sometimes it is desirable to keep an extra node in the front of a list. Such a node does not represent an item in the list and is called a header or a list header.
- The info portion of this node could be used to keep global information about the entire list i.e. the info field of header node contains the number of nodes in the list. i.e. the info field of header node contains the number of nodes in the list.
- In other words, header node in the list is defined as a node, whose information field represents some special value like -1 or information about the entire list.

The following is the example for the linked list with header node.

Advantages:

- Each item in the list represents a component, where as the header node represents the entire assembly.
- By using header node we can obtain the number of items in the list without traversing the entire list.
- Another possibility for the use of the info portion of a list header is as a pointer to a current node in the list during a traversal process. This would eliminate the need for an external pointer during traversal.

Disadvantages

- In circular list there is no particular start and end, so we have to take care. Otherwise it leads to infinite loop.

*/\*Implementation of circular Linked List \*/*

```
struct node
{
int data;

struct node *link;

}s;

s *head,*tail,*p,*q,*k;
```

*/\*create function\*/*

```
void creation()
{
int x,n,i;

printf("enter number of elements");
```

```
scanf("%d",&n);

for(i=0;i<n;i++)
{
    P=(s *)malloc(sizeof(s));
    printf("enter data");
    scanf("%d",&x);
    p->data=x;
    if(i==0)
        head=tail=p;
    else
    {
        tail->link=p;
        tail=p;
        tail->link=head;
    }
}

/*Insertion function*/

void insertion()
{
    int ch,x;
    q=(s *)malloc(sizeof(s));
    printf("enter data");
    scanf("%d",&x);
    q->data=x;
    printf("1.insbeg 2.insmid 3.insend");
```

```
printf("enter choice");  
  
scanf("%d",&ch);  
  
switch(ch)  
{  
    case 1: insbeg();  
        break;  
    case 2: insmid();  
        break;  
    case 3: insend();  
        break;  
}  
}  
  
void insbeg()  
{  
    q->link=head;  
    head=q;  
    tail->link=head;  
}  
  
void insmid()  
{  
    int pos,i=1;  
    printf("enter position");  
    scanf("%d",&pos);  
    p=head;  
    while(i<=pos)  
    {
```

```
k=p;
p=p->link;
i++;
}
k->link=q;
q->link=p;
}
void insend()
{
    tail->link=q;
    tail=q;
    tail->link=head;
}
```

```
/*Deletion function*/
void deletion()
{
    int ch;
    printf("1.delbeg 2.delmid 3.delend");
    printf("enter choice");
    scanf("%d",&ch);
    switch(ch)
    {
        case 1: delbeg();
                break;
        case 2: delmid();
```

```
        break;

    case 3: delend();

        break;

    }

}

void delbeg()
{
    p=head;
    head=head->link;
    tail->link=head;
    printf("%d id deleted", p->data);
    free(p);
}

void delmid()
{
    int pos,i=1;
    printf("enter position");
    scanf("%d",&pos);

    p=head;
    while(i<=pos)
    {
        k=p;
        p=p->link;
        i++;
    }
    k->link=p->link;
```



```
printf("%d is deleted", p->data);  
  
free(p);  
  
}  
  
void delend()  
{  
    if(head->link!=head)  
    {  
        p=head;  
        while(p->link!=head)  
        {  
            k=p;  
            p=p->link;  
        }  
        tail=k;  
        tail->link=head;  
        printf("%d is deleted", p->data);  
        free(p);  
    }  
    else  
    {  
        p=head;  
        head=tail=null;  
        printf("%d is deleted", p->data);  
        free(p);  
    }  
}
```

```
/*Display function*/  
void display()  
{  
if(head==NULL)  
printf("List is empty");  
else  
for(p=head; p!=head;p=p->link)  
printf("%d", p->data);  
}  
/*Main program*/  
void main()  
{  
int ch;  
clrscr();  
do  
{  
printf("\n1->Creation");  
printf("\n2->Deletion");  
printf("\n3->display");  
printf("\n4->Exit");  
printf("\nEnter the Choice");  
scanf("%d",&ch);  
switch(ch)  
{  
case 1: creation();
```

```

break;

case 2: deletion();

break;

case 3: display();

break;

case 4: exit(0);

break;

}

}while(ch!=4);

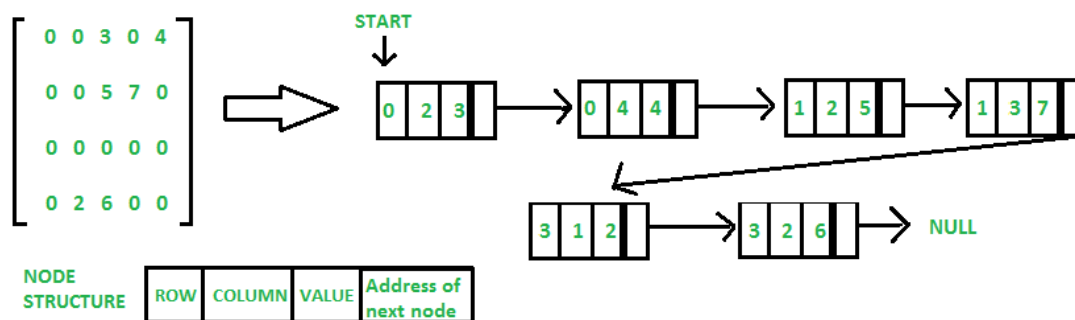
getch();

}
  
```

### SPARSE MATRICES REPRESENTATION USING LINKED LIST:

In linked list, each node has four fields. These four fields are defined as:

- Row: Index of row, where non-zero element is located
- Column: Index of column, where non-zero element is located
- Value: Value of the non zero element located at index – (row,column)
- Next node: Address of the next node



// C program for Sparse Matrix Representation

// using Linked Lists

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

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

```
// Node to represent sparse matrix
```

```
struct Node
```

```
{  
    int value;  
    int row_position;  
    int column_postion;  
    struct Node *next;  
};
```

```
// Function to create new node
```

```
void create_new_node(struct Node** start, int non_zero_element,  
                    int row_index, int column_index )  
{  
    struct Node *temp, *r;  
    temp = *start;  
    if (temp == NULL)  
    {  
        // Create new node dynamically  
        temp = (struct Node *) malloc (sizeof(struct Node));  
        temp->value = non_zero_element;  
        temp->row_position = row_index;  
        temp->column_postion = column_index;  
        temp->next = NULL;  
        *start = temp;  
    }
```

```
}  
else  
{  
    while (temp->next != NULL)  
        temp = temp->next;  
  
    // Create new node dynamically  
    r = (struct Node *) malloc (sizeof(struct Node));  
    r->value = non_zero_element;  
    r->row_position = row_index;  
    r->column_position = column_index;  
    r->next = NULL;  
    temp->next = r;  
  
}  
}  
  
// This function prints contents of linked list  
// starting from start  
void PrintList(struct Node* start)  
{  
    struct Node *temp, *r, *s;  
    temp = r = s = start;  
  
    printf("row_position: ");  
    while(temp != NULL)
```

```
{

    printf("%d ", temp->row_position);

    temp = temp->next;
}

printf("\n");

printf("column_postion: ");
while(r != NULL)
{
    printf("%d ", r->column_postion);

    r = r->next;
}

printf("\n");
printf("Value: ");
while(s != NULL)
{
    printf("%d ", s->value);

    s = s->next;
}

printf("\n");
}

// Driver of the program
int main()
{
    // Assume 4x5 sparse matrix
```

```
int sparseMatric[4][5] =
{
    {0, 0, 3, 0, 4},
    {0, 0, 5, 7, 0},
    {0, 0, 0, 0, 0},
    {0, 2, 6, 0, 0}
};

/* Start with the empty list */
struct Node* start = NULL;

for (int i = 0; i < 4; i++)
    for (int j = 0; j < 5; j++)
        // Pass only those values which are non - zero
        if (sparseMatric[i][j] != 0)
            create_new_node(&start, sparseMatric[i][j], i, j);

PrintList(start);

return 0;
}
```

Output:

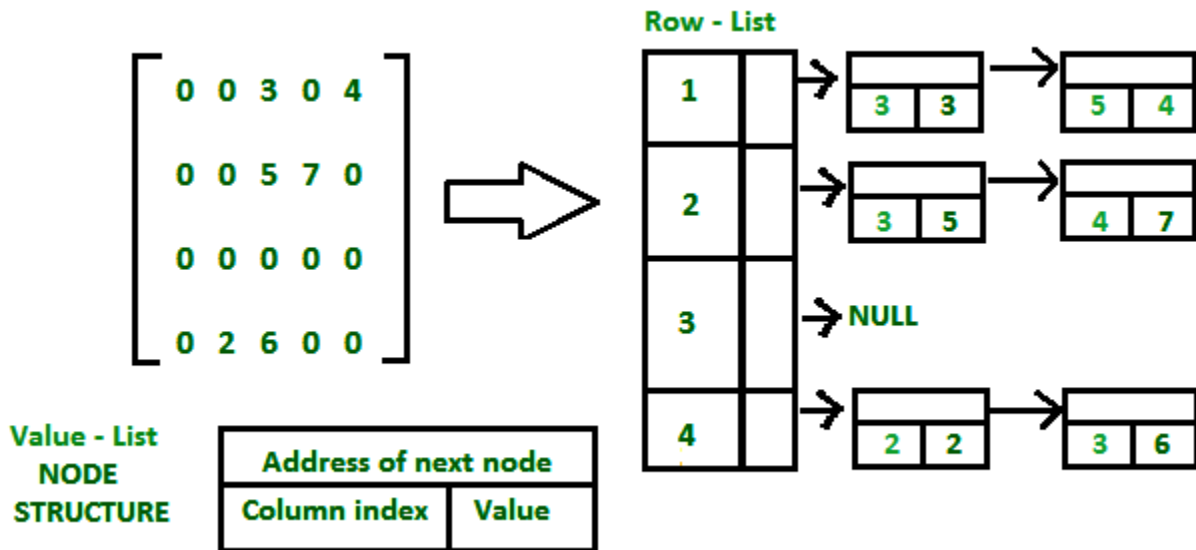
row\_position: 0 0 1 1 3 3

column\_postion: 2 4 2 3 1 2

Value: 3 4 5 7 2 6

### Sparse Matrix Input and Deleting a Sparse Matrix

One of the possible representations of sparse matrix is List of Lists (LIL). Where one list is used to represent the rows and each row contains the list of triples: Column index, Value(non – zero element) and address field, for non – zero elements. For the best performance both lists should be stored in order of ascending keys.



// C program for Sparse Matrix Representation

// using List Of Lists

#include<stdio.h>

#include<stdlib.h>

#define R 4

#define C 5

// Node to represent row - list

struct row\_list

{

int row\_number;

struct row\_list \*link\_down;

struct value\_list \*link\_right;

};

// Node to represent triples

struct value\_list



```
{
    int column_index;
    int value;
    struct value_list *next;
};

// Fuction to create node for non - zero elements
void create_value_node(int data, int j, struct row_list **z)
{
    struct value_list *temp, *d;

    // Create new node dynamically
    temp = (struct value_list*)malloc(sizeof(struct value_list));
    temp->column_index = j+1;
    temp->value = data;
    temp->next = NULL;

    // Connect with row list
    if ((*z)->link_right==NULL)
        (*z)->link_right = temp;
    else
    {
        // d points to data list node
        d = (*z)->link_right;
        while(d->next != NULL)
            d = d->next;
        d->next = temp;
    }
}
```

```
}  
  
// Function to create row list  
void create_row_list(struct row_list **start, int row,  
                    int column, int Sparse_Matrix[R][C])  
{  
    // For every row, node is created  
    for (int i = 0; i < row; i++)  
    {  
        struct row_list *z, *r;  
        // Create new node dynamically  
        z = (struct row_list*)malloc(sizeof(struct row_list));  
        z->row_number = i+1;  
        z->link_down = NULL;  
        z->link_right = NULL;  
        if (i==0)  
            *start = z;  
        else  
        {  
            r = *start;  
            while (r->link_down != NULL)  
                r = r->link_down;  
            r->link_down = z;  
        }  
  
        // Firstiy node for row is created,  
        // and then travering is done in that row
```

```
for (int j = 0; j < 5; j++)  
{  
    if (Sparse_Matrix[i][j] != 0)  
    {  
        create_value_node(Sparse_Matrix[i][j], j, &z);  
    }  
}  
}
```

//Function display data of LIL

void print\_LIL(struct row\_list \*start)

```
{  
    struct row_list *r;  
    struct value_list *z;  
    r = start;  
  
    // Traversing row list  
    while (r != NULL)  
    {  
        if (r->link_right != NULL)  
        {  
            printf("row=%d \n", r->row_number);  
            z = r->link_right;  
  
            // Traversing data list
```

```
while (z != NULL)
{
    printf("column=%d value=%d \n",
        z->column_index, z->value);
    z = z->next;
}
}
r = r->link_down;
}
}

//Driver of the program
int main()
{
    // Assume 4x5 sparse matrix
    int Sparse_Matrix[R][C] =
    {
        {0 , 0 , 3 , 0 , 4 },
        {0 , 0 , 5 , 7 , 0 },
        {0 , 0 , 0 , 0 , 0 },
        {0 , 2 , 6 , 0 , 0 }
    };

    // Start with the empty List of lists
    struct row_list* start = NULL;

    //Function creating List of Lists
    create_row_list(&start, R, C, Sparse_Matrix);
```

```
// Display data of List of lists  
print_LIL(start);  
return 0;  
}
```

Output:

```
row = 1  
column = 3 value = 3  
column = 5 value = 4  
row = 2  
column = 3 value = 5  
column = 4 value = 7  
row = 4  
column = 2 value = 2  
column = 3 value = 6
```

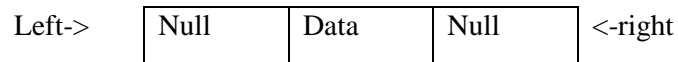
### **DOUBLY LINKED LIST:**

The drawback of singly linked list is we cannot traverse the list backwards.

To traverse the list in the reverse direction and forward direction the node should have two addresses and is stored in a two pointers.

So we define a structure for a node, which contains three fields, in which one field is data field and other two are address fields. One is left field and one more is right field.

A list containing this type of nodes is known as doubly linked list or two-way list.



The left link of the left most node and the right link of the right most node are both null indicating the end of the list of each direction.

The list can be traversed from the first node whose address is stored in a pointer variable start, to the last node in the forward direction. It can also be traversed from the last node whose address is stored in a pointer variable end, to the first node in backward direction.

#### Basic Operations

Insert Begin– Adds an element at the beginning of the list.

Delete begin– Deletes an element at the beginning of the list.

Insert Last – Adds an element at the end of the list.

Delete Last – Deletes an element from the end of the list.

Insert After/Position – Adds an element after an item of the list.

Delete **position**– Deletes an element from the list using the key.

Display-Display elements

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

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

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

```
struct node
```

```
{
```

```
    int data;
```

```
    struct node *next,*prev;
```

```
}*start=NULL,*p,*q,*temp;
```

```
typedef struct node dll;
```

```
void insbeg();
```

```
void inspos();
```

```
void insend();
```

```
void delbeg();
```

```
void delpos();
```

```
void delend();
```

```
void display();

void main()
{
    int ch; clrscr();

    printf("1.insbeg..2.inspos..3.insend..4.delbeg..5.delpos..6.delend..7.display..8.exit..\n");

    while(1)
    {
        printf("enter your choice:");
        scanf("%d",&ch);

        switch(ch)
        {
            case 1:insbeg();break;
            case 2:inspos();break;
            case 3:insend();break;
            case 4:delbeg();break;
            case 5:delpos();break;
            case 6:delend();break;
            case 7:display();break;
            case 8:exit(1);break;
            default:printf("wrong choics entered\n");
        }
    }
}

void insbeg()
```

```
{  
  
    int val;  
  
    temp=(dll *)malloc(sizeof(dll ));  
  
    printf("enter data into node:");  
  
    scanf("%d",&val);  
  
    temp->data=val;  
  
    if(start==NULL)  
    {  
  
        temp->prev=NULL;  
  
        temp->next=NULL;  
  
        start=temp;  
  
    }  
    else  
    {  
  
        temp->next=start;  
  
        temp->prev=NULL;  
  
        start=temp;  
  
    }  
}  
  
void insend()  
{  
    int val;  
  
    temp=(dll*)malloc(sizeof(dll));  
  
    printf("enter data into last:");  
  
    scanf("%d",&val);  
  
    temp->data=val;  
  
    q=start;
```



```
while(q->next!=NULL)
{
    q=q->next;
}
q->next=temp;
temp->prev=q;
temp->next=NULL;
}

void inspos()
{
    int i,pos,val;
    printf("enter position to insert:");
    scanf("%d",&pos);
    printf("enter data to insert in %d position",pos);
    scanf("%d",&val);
    temp=(dll*)malloc(sizeof(dll));
    temp->data=val;
    q=start;
    for(i=1;i<pos-1;i++)
    {
        q=q->next;
    }
    p=q->next;
    q->next=temp;
    temp->prev=q;
    p->prev=temp;
```

```
temp->next=p;
```

```
}
```

```
void display()
```

```
{
```

```
temp=start;
```

```
printf("NULL->");
```

```
while(temp!=NULL)
```

```
{
```

```
    printf("%d->",temp->data);
```

```
    temp=temp->next;
```

```
}
```

```
printf("NULL \n");
```

```
}
```

```
void delbeg()
```

```
{
```

```
q=start;
```

```
start=q->next;
```

```
start->prev=NULL;
```

```
free(q);
```

```
}
```

```
void delpos()
```

```
{
```

```
int pos,i;
```

```
printf("enter position to delete:");
```

```
scanf("%d",&pos);
```

```
q=start;
```

```
for(i=1;i<pos-1;i++)
```

```
{
```

```
    q=q->next;
```

```
}
```

```
temp=q->next;
```

```
p=temp->next;
```

```
q->next=p;
```

```
p->prev=q;
```

```
free(temp);
```

```
}
```

```
void delend()
```

```
{
```

```
q=start;
```

```
while(q->next->next!=NULL)
```

```
{
```

```
q=q->next;
```

```
}
```

```
temp=q->next;
```

```
q->next=NULL;
```

```
free(temp);
```

```
}
```

```
DOSBox 0.74, Cpu speed: max 100% cycles, Frameskip 0, Program: TC
1.insbeg..2.inspos..3.insend..4.delbeg..5.delpos..6.delend..7.display..8.exit..
enter your choice:1
enter data into node:10
enter your choice:3
enter data into last:20
enter your choice:3
enter data into last:30
enter your choice:7
NULL->10->20->30->NULL
enter your choice:2
enter position to insert:3
enter data to insert in 3 position25
enter your choice:7
NULL->10->20->25->30->NULL
enter your choice:4
enter your choice:7
NULL->20->25->30->NULL
enter your choice:6
enter your choice:7
NULL->20->25->NULL
enter your choice:5
enter position to delete:2
enter your choice:7
NULL->20->NULL
enter your choice:8_
```