

UNIT-1

ARRAYS AND LINKED LIST

Abstract Data Types (ADTs)

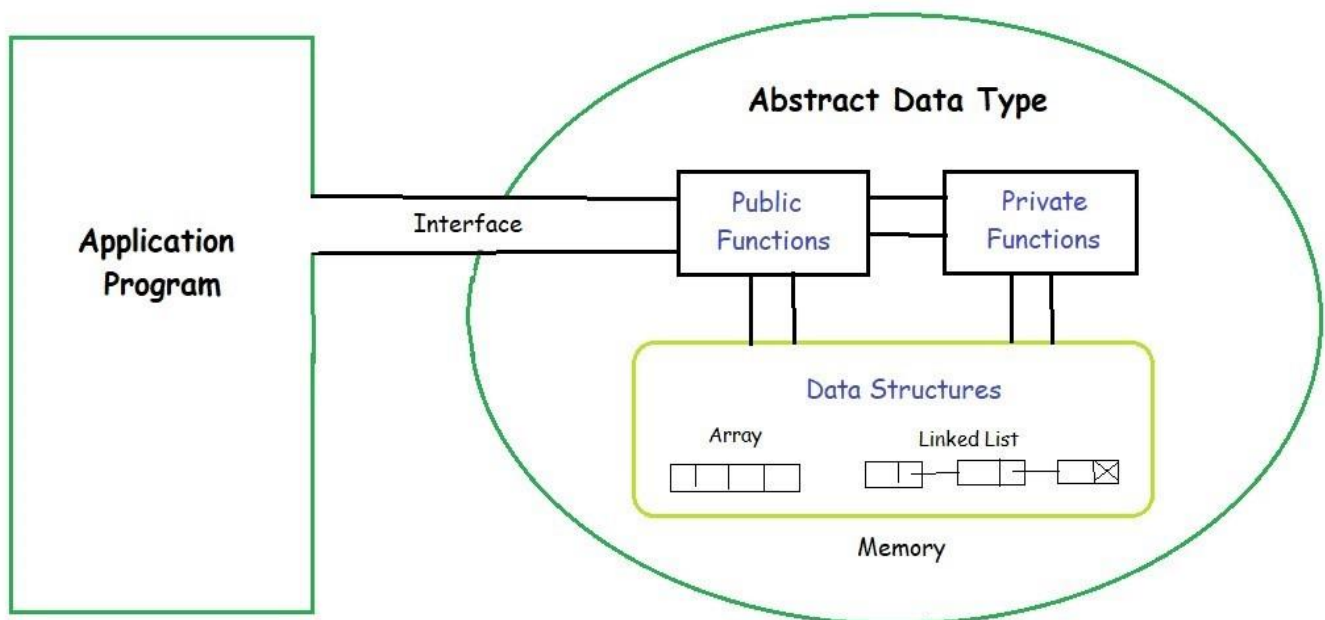
A **Data Type** refers to a named group of data which share similar properties or characteristics and which have common behavior among them.

Example: int, float, double etc.

Abstract data type: The **abstract datatype** is special kind of datatype, whose behaviour is defined by a set of values and set of operations. It is called "abstract" because it gives an implementation independent view. But how those operations are working that is totally hidden from the user. The ADT is made with primitive datatypes, but operation logics are hidden.

Examples of ADT: Stack, Queue, List etc.

Abstraction: Abstraction is the concept of simplifying a real-world concept into its essential elements. So, *Abstraction* refers to the act of representing essential features without including the background details or explanations.



Dynamic Allocation of Arrays

Arrays: An *array* is a collection of variables of the same type that are referenced by a common name. All arrays consist of *contiguous memory locations*.

Memory Allocation (Dynamic vs Static)

Each data element, stored in the memory, is given some memory. This process of giving memory is called *memory allocation*. The memory can be allocated in two manners: *dynamically* and *statically*.

STATIC MEMORY ALLOCATION

This memory allocation technique reserves fixed amount of memory before actual processing takes place, therefore, number of elements to be stored must be predetermined. Such type of memory allocation is called *static memory allocation*.

DYNAMIC MEMORY ALLOCATION

This memory allocation technique facilitates allocation of memory during the program execution itself, as and when required. This technique of memory allocation during run-time is called *dynamic memory allocation*. Dynamic memory allocation also facilitates release of memory, if memory is not required any more.

In C, there are 4 in-built functions under header file <stdlib.h> which performs dynamic memory allocation operations:

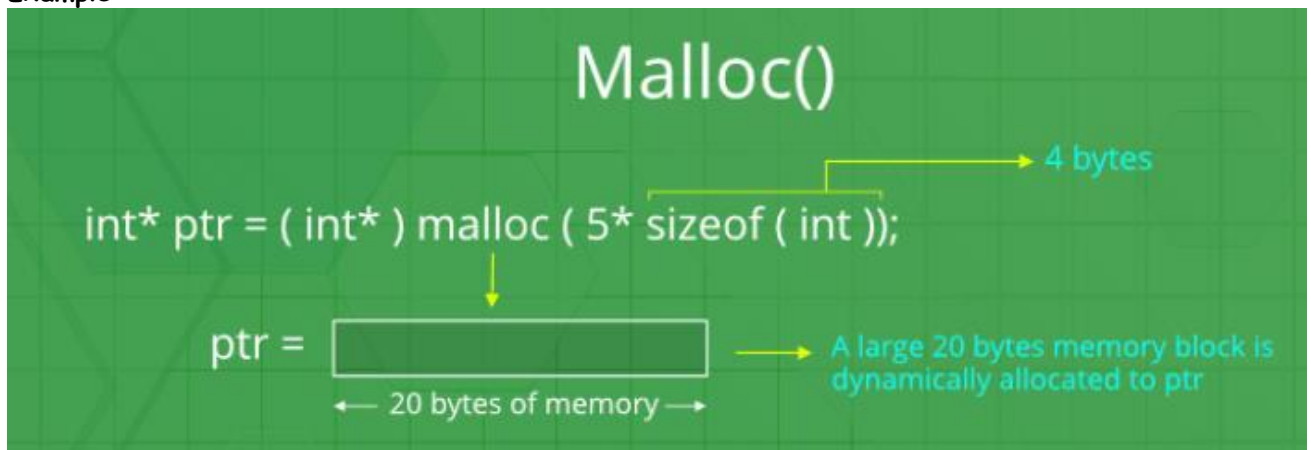
- i. malloc ()
- ii. calloc ()
- iii. realloc ()
- iv. free ()

I. malloc (): malloc refers to memory allocation in C, which is used to dynamically allocate a single large block of memory with the specified size. It returns a pointer of type void which can be cast into a pointer of any form. It initializes each block with default garbage value. If the memory is allocated successfully then the malloc () function will return first byte address.

Syntax:

```
ptr=(datatype*) malloc(no. of elements*sizeof(datatype));
```

Example:



If space is insufficient, allocation fails and returns a NULL pointer.

Sample Program:

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

```
int main()
{
    int* ptr;          //Declaration of pointer of int data-type
    int n,i;
    printf("\n Enter no. of elements : ");
    scanf("%d",&n);
    printf("\n Enter Elements : ");
    ptr=(int*)malloc(n*sizeof(int));          //dynamic memory allocation
    for(i=0;i<n;i++)
    {
        scanf("%d",&ptr[i]);
    }
    printf("\n The Elements are : ");
    for(i=0;i<n;i++)
    {
        printf("%d ",ptr[i]);
    }
    return 0;
}
```

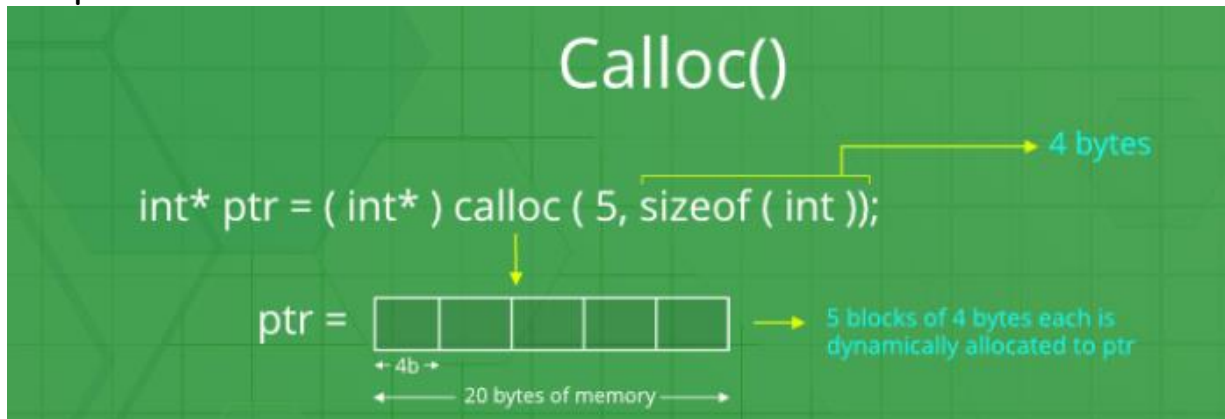
```
Enter no. of elements : 5
Enter Elements : 10 52 64 85 68
The Elements are : 10 52 64 85 68
```

II. calloc (): calloc refers to contiguous allocation method in C, which is used to dynamically allocate the specified number of blocks of memory of the specified type. It initializes each block with a default value zero. It also returns first byte address if the memory is allocated successfully. It also returns NULL value if the memory allocation fails.

Syntax:

```
ptr=(datatype*) calloc(no. of elements*sizeof(datatype));
```

Example:



Sample Program:

```
#include<stdio.h>
#include<stdlib.h>
#include<conio.h>
int main()
{
    int* ptr; //Pointer declaration
    int n,i;
    printf("\n Enter no. of elements : ");
    scanf("%d",&n);
    ptr=(int*)calloc(n,sizeof(int)); // Continuous memory allocation
    printf("\n Enter Elements: ");
    for(i=0;i<n;i++)
    {
        scanf("%d",&ptr[i]);
    }
    printf("\n The Elements are : ");
    for(i=0;i<n;i++)
        printf("%d ",ptr[i]);
    return 0;
}
```

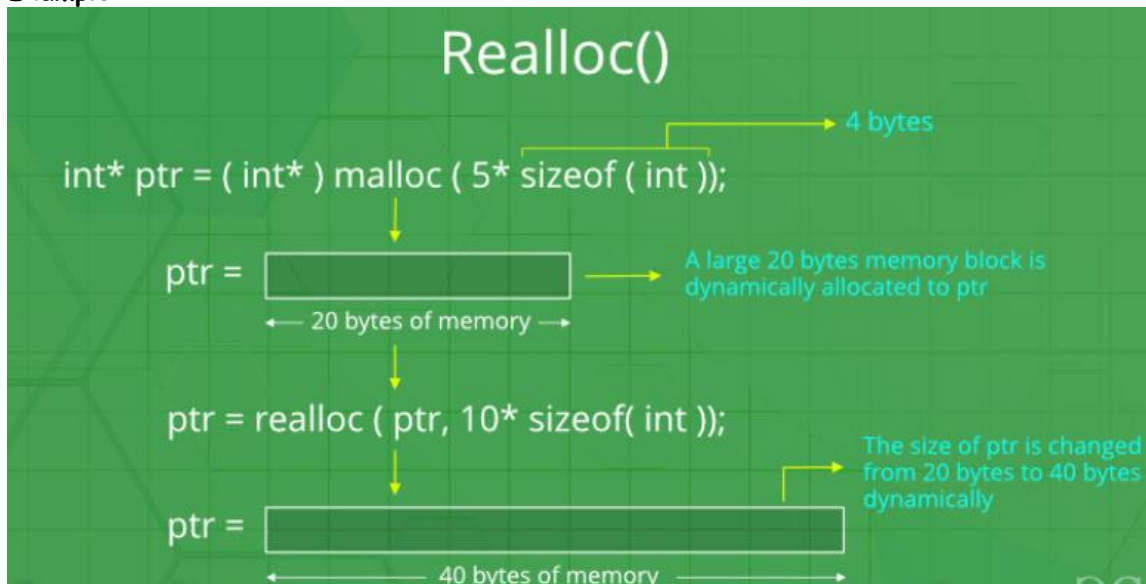
```
Enter no. of elements : 5
Enter Elements: 17 18 45 65 85
The Elements are : 17 18 45 65 85
```

- III. realloc ():** realloc refers to re-allocation in C, which is used to dynamically change the memory allocation of a previously allocated memory. In other words, if the memory previously allocated with the help of malloc or calloc is insufficient, realloc can be used to *dynamically re-allocate memory*. Re-allocation of memory maintains the already present value and new blocks will be initialized with default garbage value.

Syntax:

`ptr=(datatype*) realloc(ptr, no. of elements*sizeof(datatype));`

Example:



Sample Program:

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

int main()
{
    int* ptr; //pointer initialization
    int n,i,size;
    printf("\n Enter no. of elements : ");
    scanf("%d",&n);
    printf("\n Enter Elements : ");
    ptr=(int*)malloc(n*sizeof(int)); // Dynamic Memory allocation
    for(i=0;i<n;i++)
    {
        scanf("%d",&ptr[i]);
    }
    printf("\n The Elements are : ");
    for(i=0;i<n;i++)
    {
        printf("%d ",ptr[i]);
    }
    printf("\n\n Enter no. of elements to be appended: ");
    scanf("%d",&size);
    printf("\n Enter more Elements : ");
    ptr=(int*)realloc(ptr,(n+size)*sizeof(int)); //Re-allocation of memory
    for(i=n;i<size+n;i++)
    {
        scanf("%d",&ptr[i]);
    }
    printf("\n The Elements are : ");
    for(i=0;i<n+size;i++)
    {
        printf("%d ",ptr[i]);
    }
    return 0;
}
```

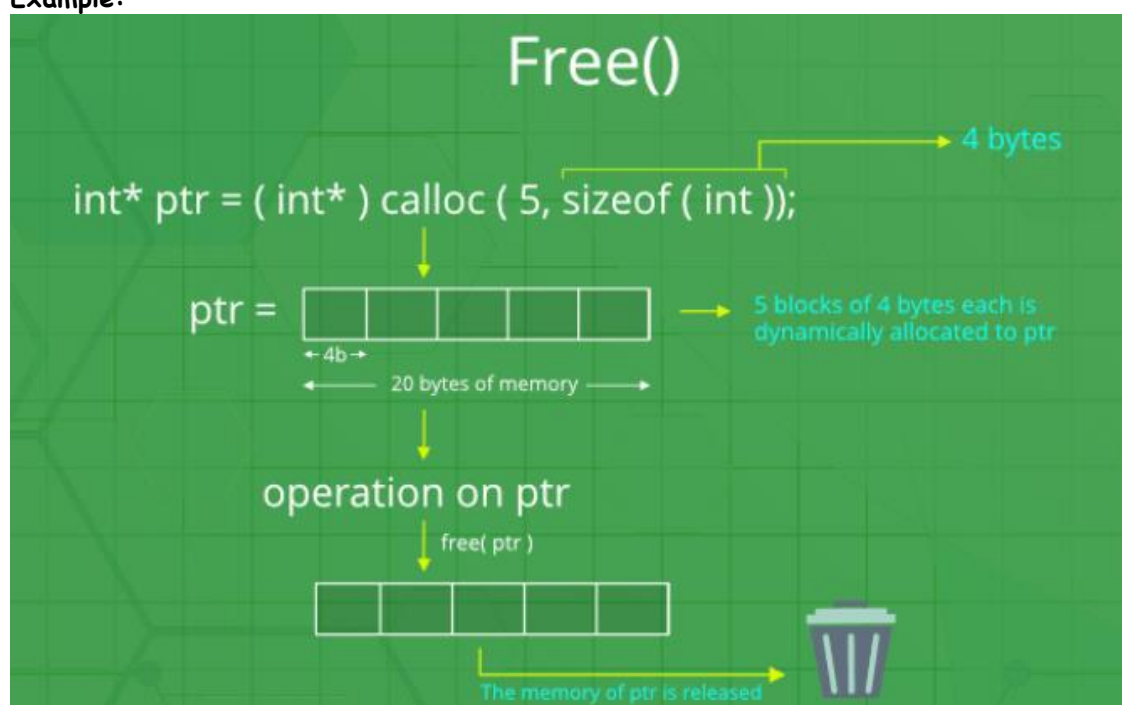
```
Enter no. of elements : 5
Enter Elements : 47 54 589 698 92
The Elements are : 47 54 589 698 92
Enter no. of elements to be appended: 4
Enter more Elements : 458 548 4747 10
The Elements are : 47 54 589 698 92 458 548 4747 10
```

- IV. **free ()**: free is a method in C which is used to dynamically *de-allocate* the memory. The memory allocated using functions malloc (), calloc (), realloc () is de-allocated using free in-built function. It helps to reduce wastage of memory by freeing it.

Syntax:

free(ptr);

Example:



Sample Program:

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

int main()
{
    int* ptr;
    int n,i;
    printf("\n Enter no. of elements : ");
    scanf("%d",&n);
    printf("\n Enter Elements : ");
    ptr=(int*)malloc(n*sizeof(int)); // Memory Allocated
    for(i=0;i<n;i++)
    {
        scanf("%d",&ptr[i]);
    }
    printf("\n The Elements are : ");
    for(i=0;i<n;i++)
    {
        printf("%d ",ptr[i]);
    }
    free(ptr); //De-allocation of memory
    return 0;
}
```

Enter no. of elements : 5

Enter Elements : 12 14 189 5487 89

The Elements are : 12 14 189 5487 89

Structures & Unions

Structures in C

A structure is a user-defined data type available in C that allows to combining data items of different kinds. Structures are used to represent a record. A C-style Structure is a collection of variables referenced under one name.

Defining a structure: To define a structure, you must use the **struct** keyword followed by statements. The *struct* keyword defines a new data type, with more than or equal to one member.

Syntax:

```
struct <structure-name>
{
    Members of Structures;
    .
    .
    .
};
```

Example:

```
struct student
{
    int roll;
    char name[20];
    float percentage;
};
```

Sample Program:

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

struct student
{
    int roll;
    char name[80];
    int marks;
};

void main()
{
    int i,j,n;
    struct student s[30];

    clrscr();

    printf("\n Enter no. of students : ");
    scanf("%d",&n);

    printf("\n Enter Details of Students : ");
    for(i=0;i<n;i++)
    {
        printf("\n\n Student %d : ",i+1);
        printf("\n Enter Roll No. : ");
        scanf("%d",&s[i].roll);
        printf("\n Enter Name : ");
        scanf("%s",s[i].name);
        printf("\n Enter Marks : ");
        scanf("%d",&s[i].marks);
    }

    printf("\n Students Details are :- \n");
    for(i=0;i<n;i++)
    {
        printf("\n\n Student %d : \n",i+1);
        printf("\n Roll No. : %d",s[i].roll);
        printf("\n Name : %s",s[i].name);
        printf("\n Marks : %d",s[i].marks);
    }

    getch();
}
```

```
Enter no. of students : 4
Enter Details of Students :
Student 1 :
Enter Roll No. : 101
Enter Name : Divyanshu
Enter Marks : 94
Student 2 :
Enter Roll No. : 102
Enter Name : Deepak
Enter Marks : 98
Student 3 :
Enter Roll No. : 103
Enter Name : Amit
Enter Marks : 96
Student 4 :
Enter Roll No. : 104
Enter Name : Ashish
Enter Marks : 94
Students Details are :-
Student 1 :
Roll No. : 101
Name : Divyanshu
Marks : 94
Student 2 :
Roll No. : 102
Name : Deepak
Marks : 98
Student 3 :
Roll No. : 103
Name : Amit
Marks : 96
Student 4 :
Roll No. : 104
Name : Ashish
Marks : 94
```


Unions in C

A union is a special data type available in C that allows storing 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 purposes.

Defining a Union: To define a union, you must use the **union** statement in the same way as you did while defining a structure. The union statement defines a new data type with more than one member for your program.

Syntax:

```
union [union name]
{
    member definition;
    member definition;
    ...
    member definition;
};
```

Example:

```
union student
{
    int roll;
    char name[20];
    float percentage;
};
```

Sample Program:

```
#include<stdio.h>

union student
{
    int roll;
    char name[80];
    int marks;
};

int main()
{
    int i,j,n;
    union student s[30];
    printf("\n Enter no. of students : ");
    scanf("%d",&n);
    printf("\n Enter Details of Students : ");
    for(i=0;i<n;i++)
    {
        printf("\n\n Student %d : ",i+1);
        printf("\n Enter Roll No. : ");
        scanf("%d",&s[i].roll);
        printf("\n Enter Name : ");
        scanf("%s",s[i].name);
        printf("\n Enter Marks : ");
        scanf("%d",&s[i].marks);
    }
    printf("\n Students Details are :- \n");
    for(i=0;i<n;i++)
    {
        printf("\n\n Student %d : \n",i+1);
        printf("\n Roll No. : %d",s[i].roll);
        printf("\n Name : %s",s[i].name);
        printf("\n Marks : %d",s[i].marks);
    }
    return 0;
}
```

Enter no. of students : 3

Enter Details of Students :

Student 1 :

Enter Roll No. : 1001

Enter Name : Raj

Enter Marks : 98

Student 2 :

Enter Roll No. : 1002

Enter Name : Amit

Enter Marks : 94

Student 3 :

Enter Roll No. : 1003

Enter Name : Aman

Enter Marks : 90

Students Details are :-

Student 1 :

Roll No. : 98

Name : b

Marks : 98

Student 2 :

Roll No. : 94

Name : ^

Marks : 94

Student 3 :

Roll No. : 90

Name : Z

Marks : 90

Similarities between Structure and Union

- Both are user-defined data types used to store data of different types as a single unit.
- Their members can be objects of any type, including other structures and unions or arrays. A member can also consist of a bit field.
- Both structures and unions support only assignment = and sizeof operators. The two structures or unions in the assignment must have the same members and member types.
- A structure or a union can be passed by value to functions and returned by value by functions. The argument must have the same type as the function parameter. A structure or union is passed by value just like a scalar variable as a corresponding parameter.
- '.' operator is used for accessing members.

Difference between Structure and Union

	STRUCTURE	UNION
Keyword	The keyword struct is used to define a structure	The keyword union is used to define a union.
Size	When a variable is associated with a structure, the compiler allocates the memory for each member. The size of structure is greater than or equal to the sum of sizes of its members.	when a variable is associated with a union, the compiler allocates the memory by considering the size of the largest memory. So, size of union is equal to the size of largest member.
Memory	Each member within a structure is assigned unique storage area of location.	Memory allocated is shared by individual members of union.
Value Altering	Altering the value of a member will not affect other members of the structure.	Altering the value of any of the member will alter other member values.
Accessing members	Individual member can be accessed at a time.	Only one member can be accessed at a time.
Initialization of Members	Several members of a structure can initialize at once.	Only the first member of a union can be initialized.

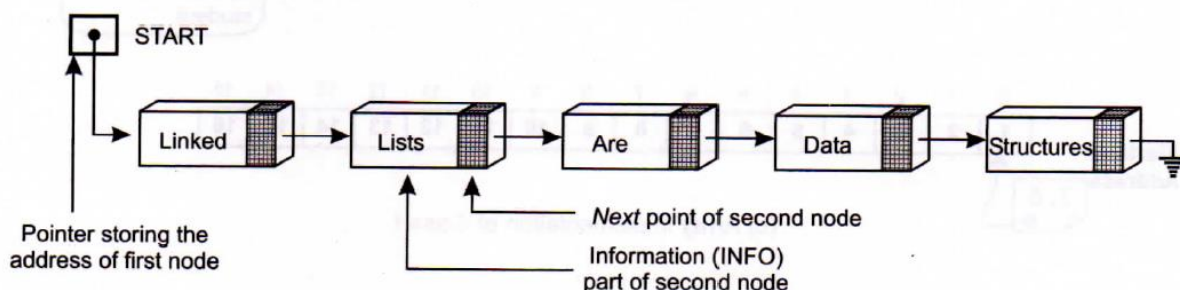
Single Linked List and Chains

The term 'list' refers to linear collection of data. One form of linear lists are arrays and another form is linked list. (Stack, Queue, etc.)

Linked List: It is a linear collection of data elements, called nodes pointing to the next nodes by means of pointers. Each node is divided into two parts: the first part containing the data of the element, and the second part is called the *link* or *next pointer* containing the address of the next node in the list.

Single Linked List:

The one-way implementation of a general list having elements : 'linked', 'lists', 'are', 'data', 'structures' can be done as shown below :



The pointer START is a special pointer which stores the very first address of a linked list. The *Next pointer* of the last node stores NULL value (shown as earth sign), which means this node is not pointing to any other node i.e., it is the last node in the list.

Creating structure for a node

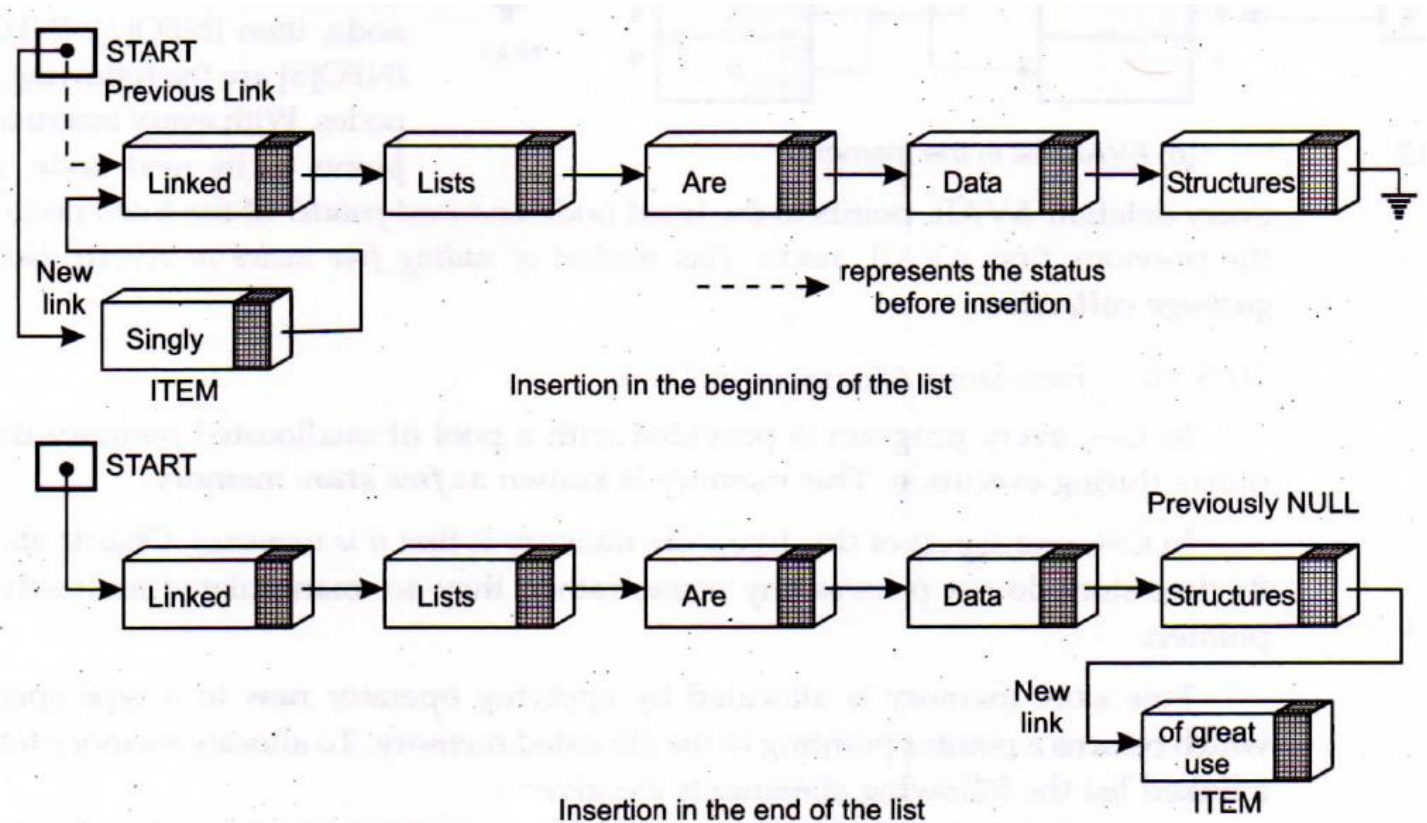
```
struct node
{
    int data;
    struct node *next;
};
```

Creating memory for the node

```
struct node *ptr;
ptr=(struct node*) malloc (sizeof (struct node));
```

INSERTION

In linked list, new node is either added in the beginning of the list or in the middle of the list or in the end of the list.



INSERTION AT THE BEGINNING

Algorithm:

Step 1: START

Step 2: Create a new node i.e. "newptr"

```
newptr= (struct node*) malloc (sizeof (struct node));
```

Step 3: Read data x

```
newptr->data=x;
```

```
newptr->next=NULL;
```

Step 4: newptr->next=start;

```
start=newptr;
```

Step 5: STOP

Sample Program:

In the following program, the pointer 'start', points to the beginning of the list. Function *Create_New_Node ()* takes one integer argument, allocates memory to create a new node and returns the pointer to the new node (return-type: node*). Function *Display ()* takes node* type pointer as argument and displays the list from this pointer till the end of the list. The below program firstly reads the new data dynamically, stores data in it and adds this node in the beginning of the linked list being pointed to by pointer start.

```

/* Insertion of data in the Beginning of the List. */

#include<stdio.h>
#include<conio.h>

struct node
{
    int data;
    struct node *next;
}*start,*newptr,*save,*temp;

struct node* Create_New_Node(int);
void Display(struct node*);

void main()
{
    int i,x,num,size;
    start=save=NULL;
    clrscr();
    printf("\n\t\t\t Creation of List \n");
    printf("\n Enter Size of the list : ");
    scanf("%d",&size);
    printf("\n Enter data in List : \n");
    for(i=0;i<size;i++)
    {
        scanf("%d",&x);
        newptr=Create_New_Node(x);
        if(start==NULL)
        {
            start=save=newptr;
        }
        else
        {
            save->next=newptr;
            save=newptr;
        }
    }
    printf("\n The List is : \n ");
    Display(start);
    printf("\n\t\t\t Insertion in the beginning of the list.");
    printf("\n\n Enter data (any integer): ");
    scanf("%d",&num);
    newptr=Create_New_Node(num);
    newptr->next=start;
    start=newptr;
    printf("\n Final List is : \n ");
    Display(start);
    getch();
}

struct node* Create_New_Node(int n)
{
    temp=(struct node*)malloc(sizeof(struct node));
    temp->data=n;
    temp->next=NULL;
    return temp;
}

void Display(struct node* np)
{
    while(np!=NULL)
    {
        printf("%d \t",np->data);
        np=np->next;
    }
}

```

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

File Edit Search Run Compile Debug Project Options

Output

Creation of List

Enter Size of the list : 4

Enter data in List : 12 343 653 21

The List is :
12 343 653 21

Insertion in the beginning of the list.

Enter data (any integer): 404

Final List is :
404 12 343 653 21 _

F1 Help ↑↓↔ Scroll

INSERTION AT THE MIDDLE/POSITION

Algorithm:

Step 1: START

Step 2: Create new node i.e. "newptr"
newptr=(struct node*) malloc (sizeof (struct node));

Step 3: Read data part in newptr node i.e. 'x'
newptr->data=x;
newptr->next=NULL;

Step 4: Read the position 'pos' (where to insert a node)
ptr=start;
for(i=0;i<pos-1; i++)
{
ptr=ptr->next;
}

Step 5: newptr->next=ptr->next;
ptr->next=newptr;

Step 6: STOP

Sample Program:

In the following program, the *start* pointer points to the beginning of the list, *pos* variable stores the required position where the new node to be inserted.

Function *Create_New_Node ()* takes one integer argument, allocates memory to create a new node and returns the pointer to the new node. (return-type: node*)

Function *Display ()* take node* type pointer as argument and displays the list from this pointer till the end of the list.

The below program firstly reads the new data, creates a new node dynamically, stores data in it and adds this node at the required position in the linked list.

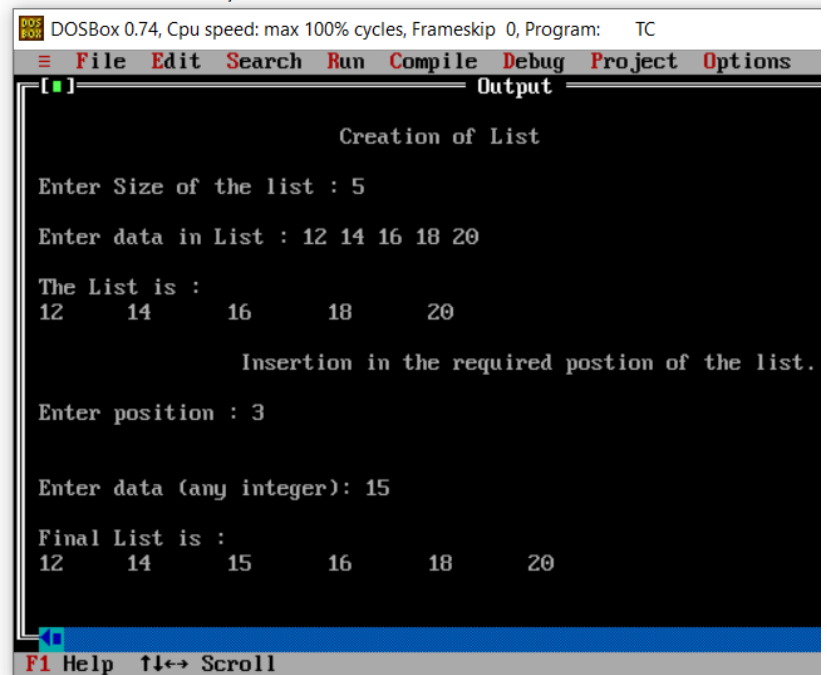
/* Insertion of data in the required position of the List. */

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

struct node
{
    int data;
    struct node *next;
}*start,*newptr,*save,*temp,*ptr;

struct node* Create_New_Node(int);
void Display(struct node*);

void main()
{
    int i,x,num,size,pos;
    start=save=NULL;
    clrscr();
    printf("\n\t\t\t\t\tCreation of List \n");
    printf("\n Enter Size of the list : ");
    scanf("%d",&size);
    printf("\n Enter data in List : \n");
    for(i=0;i<size;i++)
    {
        scanf("%d",&x);
        newptr=Create_New_Node(x);
        if(start==NULL)
        {
            start=save=newptr;
        }
        else
        {
            save->next=newptr;
            save=newptr;
        }
    }
}
```



```

    printf("\n The List is : \n ");
    Display(start);
    printf("\n\n\t\t Insertion in the required postion of the list.");
    printf("\n\n Enter position : ");
    scanf("%d",&pos);
    printf("\n\n Enter data (any integer): ");
    scanf("%d",&num);
    newptr=Create_New_Node(num);
    ptr=start;
    for(i=1;i<pos;i++)
    {
        ptr=ptr->next;
    }
    newptr->next=ptr->next;
    ptr->next=newptr;
    printf("\n Final List is : \n ");
    Display(start);
    getch();
}
struct node* Create_New_Node(int n)
{
    temp=(struct node*)malloc(sizeof(struct node));
    temp->data=n;
    temp->next=NULL;
    return temp;
}

void Display(struct node* np)
{
    while(np!=NULL)
    {
        printf("%d \t",np->data);
        np=np->next;
    }
}

```

INSERTION AT THE END

Algorithm:

Step 1: START

Step 2: Create new node i.e. "newptr"

newptr= (struct node*) malloc (sizeof (struct node));

Step 3: Read value i.e. 'x'

newptr->data=x;

newptr->next=NULL;

Step 4: q=start;

while(q->next!=NULL)

{

q=q->next;

}

Step 5: q->next=newptr;

q=newptr;

Step 6: STOP

Sample Program:

In the following program, the *start* pointer points to the beginning of the list, *save* pointer points to the last node.

Function *Create_New_Node ()* takes one integer argument, allocates memory to create a new node and returns the pointer to the new node. (return-type: node*)

Function *Display ()* take node* type pointer as argument and displays the list from this pointer till the end of the list.

The below program reads *data*, creates a new node dynamically, assigns read data to it and inserts this new node at the end of linked list being pointed to by pointer *save*.

```

/* Insertion of data in the End of the List. */

#include<stdio.h>
#include<conio.h>

struct node
{
    int data;
    struct node *next;
}*start,*newptr,*save,*temp;

struct node* Create_New_Node(int);
void Display(struct node*);

void main()
{
    int i,x,num,size;
    start=save=NULL;
    clrscr();
    printf("\n\t\t\t\t\t Creation of List \n");
    printf("\n Enter Size of the list : ");
    scanf("%d",&size);
    printf("\n Enter data in List : ");
    for(i=0;i<size;i++)
    {
        scanf("%d",&x);
        newptr=Create_New_Node(x);
        if(start==NULL)
        {
            start=save=newptr;
        }
        else
        {
            save->next=newptr;
            save=newptr;
        }
    }
    printf("\n The List is : \n ");
    Display(start);
    printf("\n\n\t\t\t\t\t Insertion in the end of the list.");
    printf("\n\n Enter data (any integer): ");
    scanf("%d",&num);
    newptr=Create_New_Node(num);
    save->next=newptr;
    save=newptr;
    printf("\n Final List is : \n ");
    Display(start);
    getch();
}

struct node* Create_New_Node(int n)
{
    temp=(struct node*)malloc(sizeof(struct node));
    temp->data=n;
    temp->next=NULL;
    return temp;
}

void Display(struct node* np)
{
    while(np!=NULL)
    {
        printf("%d \t",np->data);
        np=np->next;
    }
}

```

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

File Edit Search Run Compile Debug Project

Output

Creation of List

Enter Size of the list : 5

Enter data in List : 12 14 16 18 20

The List is :

12 14 16 18 20

Insertion in the end of the list.

Enter data (any integer): 22

Final List is :

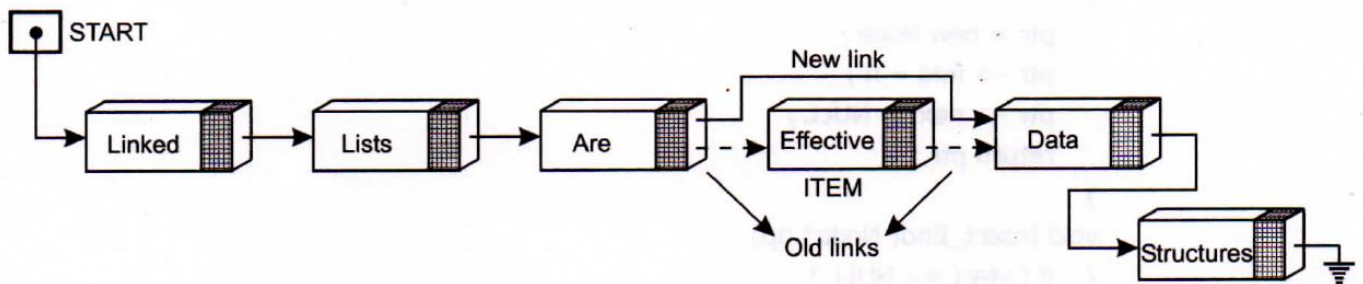
12 14 16 18 20 22

F1 Help ↑↓↔ Scroll

DELETION

Deletion of ITEM from a linked list involves

- (i) Search for ITEM in the list for availability
- (ii) If available, make its previous node point to its next node.



In deletion of a node, there are *three* possibilities :

- Case I** If the node to be deleted happens to be the first node, then START is made to point the second node in sequence and deleted node is added in the *free-store*.
i.e., **Save = START, START = START -> LINK** and then *delete Save*
- Case II** If the node happens to be in the middle of list, its previous pointer is saved in *Save* and if the node's pointer is *ptr* then **Save -> LINK = ptr -> LINK**. Also the deleted node is added in the *free-store* in the similar way as described in *case I*.
- Case III** If the node is at the end of the list, then its previous node (*save*) will point to NULL.
i.e., **Save -> LINK = NULL**

Here we are covering only the first case i.e., *deletion from the beginning*.

DELETION FROM THE BEGINNING

Algorithm:

- Step 1:** START
Step 2: temp=start
Step 3: start=start->next;
Step 4: free (temp);
Step 5: STOP

Sample Program:

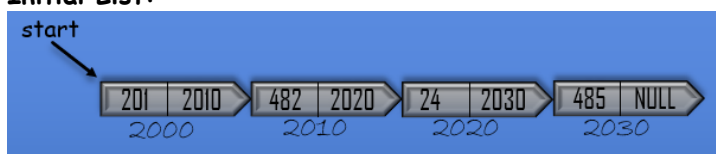
In the following program, the *start* pointer points to the beginning of the list, *save* pointer points to the last node.

Function *Create_New_Node ()* takes one integer argument, allocates memory to create a new node and returns the pointer to the new node. (return-type: node*)

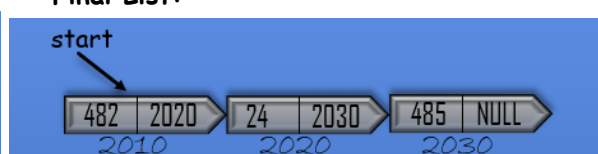
Function *Display ()* take node* type pointer as argument and displays the list from this pointer till the end of the list.

The below program deletes the node from the beginning of the list, this is done by storing starting node in a temporary node and performing start=start->next; then deleting temporary node with in-built function free(temp);

Initial List:



Final List:



```

/* Deletion of data from the Beginning of the List. */

#include<stdio.h>
#include<conio.h>

struct node
{
    int data;
    struct node *next;
}*start,*newptr,*save,*temp;

struct node* Create_New_Node(int);
void Display(struct node*);

void main()
{
    int i,x,num,size;
    start=save=NULL;
    clrscr();
    printf("\n\t\t\t\t\tCreation of List \n");
    printf("\n Enter Size of the list : ");
    scanf("%d",&size);
    printf("\n Enter data in List : ");
    for(i=0;i<size;i++)
    {
        scanf("%d",&x);
        newptr=Create_New_Node(x);
        if(start==NULL)
        {
            start=save=newptr;
        }
        else
        {
            save->next=newptr;
            save=newptr;
        }
    }
    printf("\n The List is : ");
    Display(start);
    printf("\n\n\t\t\t\t\tDeletion from the beginning of the list.");
    temp=start;
    start=start->next;
    free (temp);
    printf("\n\n Final List is : ");
    Display(start);
    getch();
}

struct node* Create_New_Node(int n)
{
    temp=(struct node*)malloc(sizeof(struct node));
    temp->data=n;
    temp->next=NULL;
    return temp;
}

void Display(struct node* np)
{
    while(np!=NULL)
    {
        printf("%d \t",np->data);
        np=np->next;
    }
}

```

```

DOSBox 0.74, Cpu speed: max 100% cycles, Frameskip 0, Program: TC
File Edit Search Run Compile Debug Project Options Wi
Output
Creation of List
Enter Size of the list : 7
Enter data in List : 10 20 30 40 50 60 70
The List is : 10 20 30 40 50 60 70
Deletion from the beginning of the list.
Final List is : 20 30 40 50 60 70
F1 Help ↑↔ Scroll

```

In the output we can notice that, the size of the linked list is 7 and the data entered are 10 20 30 40 50 60 70 and at the beginning the list contains 10 20 30 40 50 60 70 and after processing the deletion from the beginning of the list, the final list contains 20 30 40 50 60 70. This means the first node which contains value 10 got deleted with in-built function *free* ().

DELETION FROM THE POSITION/MIDDLE

Algorithm:

Step 1: START

Step 2: Read position i.e. 'pos'

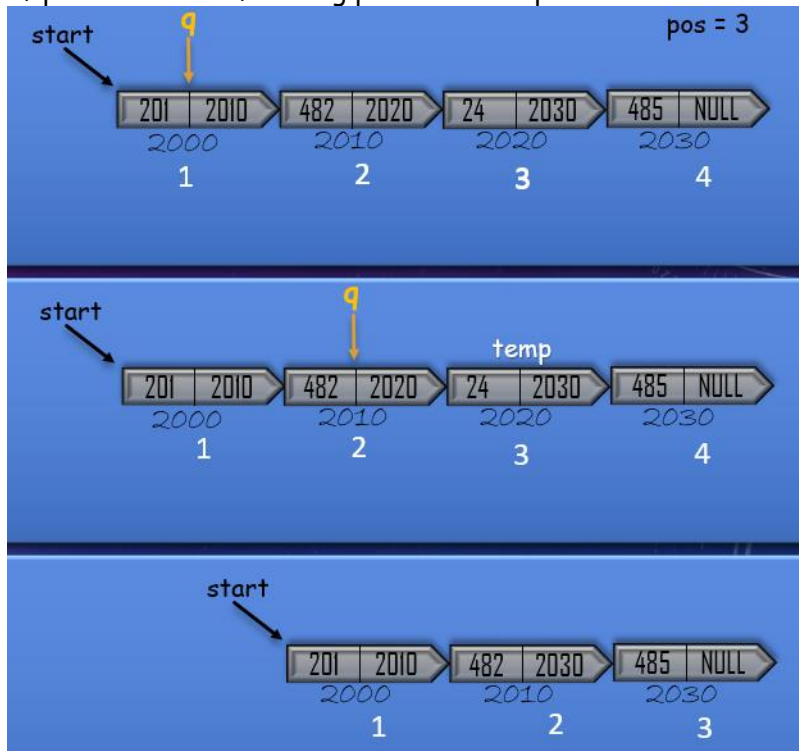
Step 3: Let q=start;
for (i=0;i<pos-1;i++)
{
 q=q->next;
}

Step 4: temp=q->next;
q->next=temp->next;

Step 5: free (temp);

Step 6: STOP

If pos=3 then the following process takes place:



Sample Program:

In the following program, the *start* pointer points to the beginning of the list, *save* pointer points to the last node.

Function *Create_New_Node()* takes one integer argument, allocates memory to create a new node and returns the pointer to the new node. (return-type: node*)

Function *Display()* take node* type pointer as argument and displays the list from this pointer till the end of the list.

```
/* Deletion of data from the Required Position of the List. */
```

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

struct node
{
    int data;
    struct node *next;
}*start,*newptr,*save,*temp,*q;

struct node* Create_New_Node(int);
void Display(struct node*);

void main()
{
    int i,x,pos,num,size;
    start=save=NULL;
    clrscr();
```

The screenshot shows a DOSBox window titled 'DOSBox 0.74, Cpu speed: max 100% cycles, Frameskip 0, Program: TC'. The window contains a menu bar with 'File', 'Edit', 'Search', 'Run', 'Compile', 'Debug', 'Project', 'Options', 'Window', and 'Help'. Below the menu bar is a command prompt window with the following text:

```
Creation of List
Enter Size of the list :7
Enter data in List : 10 20 30 40 50 60 70
The List is : 10      20      30      40      50      60      70
Deletion from the Required Position of the list.
Enter Required Position : 4
Final List is : 10      20      30      50      60      70
```

```

printf("\n\t\t\t\t Creation of List \n");
printf("\n Enter Size of the list : ");
scanf("%d",&size);
printf("\n Enter data in List : ");
for(i=0;i<size;i++)
{
    scanf("%d",&x);
    newptr=Create_New_Node(x);
    if(start==NULL)
    {
        start=save=newptr;
    }
    else
    {
        save->next=newptr;
        save=newptr;
    }
}
printf("\n The List is : ");
Display(start);
printf("\n\n\t\t\t Deletion from the Required Position of the list.");
printf("\n\n Enter Required Position : ");
scanf("%d",&pos);
q=start;
for(i=1;i<pos-1;i++)
{
    q=q->next;
}
temp=q->next;
q->next=temp->next;
free (temp);
printf("\n\n Final List is : ");
Display(start);
getch();
}
struct node* Create_New_Node(int n)
{
    temp=(struct node*)malloc(sizeof(struct node));
    temp->data=n;
    temp->next=NULL;
    return temp;
}

void Display(struct node* np)
{
    while(np!=NULL)
    {
        printf("%d \t",np->data);
        np=np->next;
    }
}

```

In the above program, the node at the required position is to be deleted. So, we will input required position from the user in variable 'pos'. Then we will run a for loop from the beginning of the list till the required position for storing the required nodes address in pointer 'q'. Then we will store it in temporary node and we will re-join previous node with the next node of temporary node. And finally, we will delete the temporary node with in-built function free ().

DELETION FROM THE END

Algorithm:

Step 1: START

Step 2: Let node q=start;

Step 3: while (q->next->next!=NULL)
 {
 q=q->next;
 }

Step 4: temp=q->next;
 q->next=NULL;

Step 5: free (temp);

Step 6: STOP

Sample Program:

/* Deletion of data from the End of the List. */

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

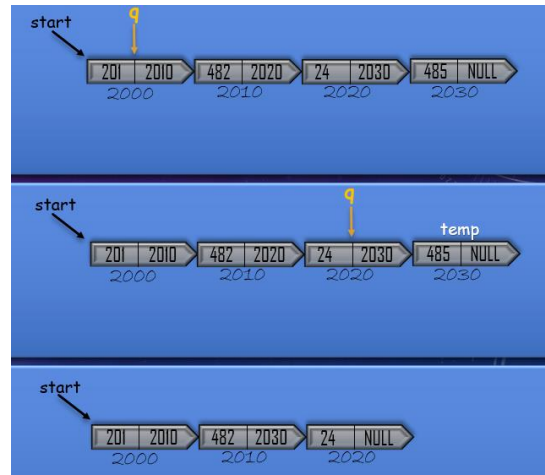
struct node
{
    int data;
    struct node *next;
}*start,*newptr,*save,*temp,*q;

struct node* Create_New_Node(int);
void Display(struct node*);

void main()
{
    int i,x,num,size;
    start=save=NULL;
    clrscr();
    printf("\n\t\t\t\t\tCreation of List \n");
    printf("\n Enter Size of the list : ");
    scanf("%d",&size);
    printf("\n Enter data in List : ");
    for(i=0;i<size;i++)
    {
        scanf("%d",&x);
        newptr=Create_New_Node(x);
        if(start==NULL)
        {
            start=save=newptr;
        }
        else
        {
            save->next=newptr;
            save=newptr;
        }
    }
    printf("\n The List is : ");
    Display(start);
    printf("\n\n\t\t\t\t\tDeletion from the END of the list.");
    q=start;
    while(q->next->next!=NULL)
    {
        q=q->next;
    }
    temp=q->next;
    q->next=NULL;
    free (temp);
    printf("\n\n Final List is : ");
    Display(start);
    getch();
}

struct node* Create_New_Node(int n)
{
    temp=(struct node*)malloc(sizeof(struct node));
    temp->data=n;
    temp->next=NULL;
    return temp;
}

void Display(struct node* np)
{
    while(np!=NULL)
    {
        printf("%d \t",np->data);
        np=np->next;
    }
}
```



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

File Edit Search Run Compile Debug Project Options Window Help

Output

```

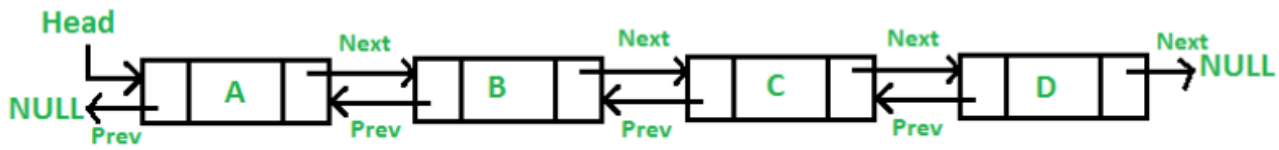
Creation of List
Enter Size of the list : 7
Enter data in List : 10 20 30 40 50 60 70
The List is : 10      20      30      40      50      60      70

Deletion from the END of the list.
Final List is : 10      20      30      40      50      60      _
  
```

F1 Help ↑↓↔ Scroll

Doubly Linked List

A Doubly Linked List (DLL) contains an extra pointer, typically called *previous pointer*, together with next pointer and data which are there in singly linked list.



Here, each node is divided into three parts: the first part contains address of previous node basically known as *previous link* or *previous pointer*, the second part containing the data of the node, and the third part is called the *link* or *next pointer* containing the address of the next node in the list.

Creating Structure of DLL node

```
struct node
{
    int data;
    struct node* next;    //Pointer to next node
    struct node* prev;    //Pointer to previous node
};
```

Creating memory for the node

```
struct node *ptr;
ptr=(struct node*) malloc (sizeof (struct node));
```

ADVANTAGES OVER SINGLE LINKED LIST

- 1) A DLL can be traversed in both forward and backward direction.
- 2) The delete operation in DLL is more efficient if pointer to the node to be deleted is given.
- 3) We can quickly insert a new node before a given node. In singly linked list, to delete a node, pointer to the previous node is needed. To get this previous node, sometimes the list is traversed. In DLL, we can get the previous node using previous pointer.

DISADVANTAGES OVER SINGLY LINKED LIST

- 1) Every node of DLL Require extra space for a previous pointer.
- 2) All operations require an extra pointer previous to be maintained. For example, in insertion, we need to modify previous pointers together with next pointers. For example, for insertions at different positions, we need 1 or 2 extra steps to set previous pointer.

INSERTION

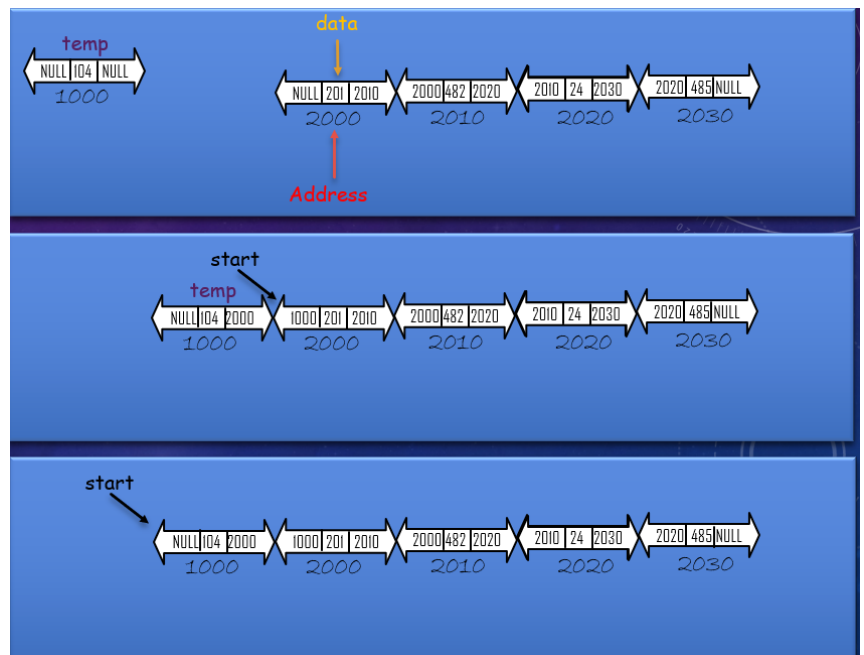
A node can be added in four ways:

- 1) At the front of the DLL
- 2) After a given node.
- 3) Before a given node.
- 4) At the end of the DLL

INSERTION AT THE BEGINNING

Algorithm:

- Step 1:** START
- Step 2:** Create a new node i.e. "temp"
temp=(struct node*)malloc(sizeof(node));
- Step 3:** Read data x
temp->data=x;
temp->prev=NULL;
temp->next=NULL;
- Step 4:** temp->next=start;
start->prev=temp;
start=temp;
- Step 5:** STOP



Sample Program:

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

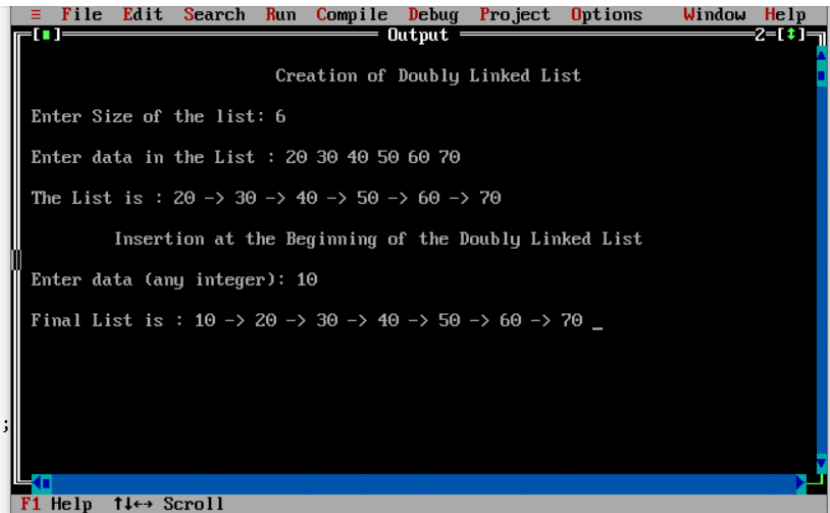
struct node
{
    int data;
    struct node *next;
    struct node *prev;
}*start,*save,*newptr,*temp;

struct node* Create_New_Node(int);
void Display(struct node*);

void main()
{
    int i,x,size;
    start=save=NULL;
    clrscr();
    printf("\n\t\t\t Creation of Doubly Linked List \n");
    printf("\n Enter Size of the list: ");
    scanf("%d",&size);
    printf("\n Enter data in the List : ");
    for(i=0;i<size;i++)
    {
        scanf("%d",&x);
        newptr=Create_New_Node(x);
        if(start==NULL)
        {
            start=save=newptr;
        }
        else
        {
            newptr->prev=save;
            save->next=newptr;
            save=newptr;
        }
    }
    printf("\n The List is : ");
    Display(start);
    printf("\n\n\t Insertion at the Beginning of the Doubly Linked List");
    printf("\n\n Enter data (any integer): ");
    scanf("%d",&x);
    newptr=Create_New_Node(x);
    newptr->prev=NULL;
    newptr->next=start;
    start->prev=newptr;
    start=newptr;
    printf("\n Final List is : ");
    Display(start);
    getch();
}

struct node* Create_New_Node(int n)
{
    temp=(struct node*)malloc(sizeof(struct node));
    temp->data=n;
    temp->next=NULL;
    temp->prev=NULL;
    return temp;
}

void Display(struct node* np)
{
    while(np->next!=NULL)
    {
        printf("%d -> ",np->data);
        np=np->next;
    }
    printf("%d ",np->data);
}
```



The screenshot shows a terminal window with a menu bar (File, Edit, Search, Run, Compile, Debug, Project, Options, Window, Help) and a title bar (Output, 2-[+]). The output text is as follows:

```
Creation of Doubly Linked List

Enter Size of the list: 6

Enter data in the List : 20 30 40 50 60 70

The List is : 20 -> 30 -> 40 -> 50 -> 60 -> 70

Insertion at the Beginning of the Doubly Linked List

Enter data (any integer): 10

Final List is : 10 -> 20 -> 30 -> 40 -> 50 -> 60 -> 70 _
```

In the above program, the *start* pointer points to the beginning of the list, *save* pointer points to the last node. Function *Create_New_Node ()* takes one integer argument, allocates memory to create a new node and returns the pointer to the new node. (return-type: node*)

Function *Display ()* take node* type pointer as argument and displays the list from this pointer till the end of the doubly linked list.

The above program reads *data*, creates a new node dynamically, assigns read data to it and inserts this new node at the beginning of linked list being pointed to by pointer *start*.

newptr->prev=NULL;

newptr->next=start;

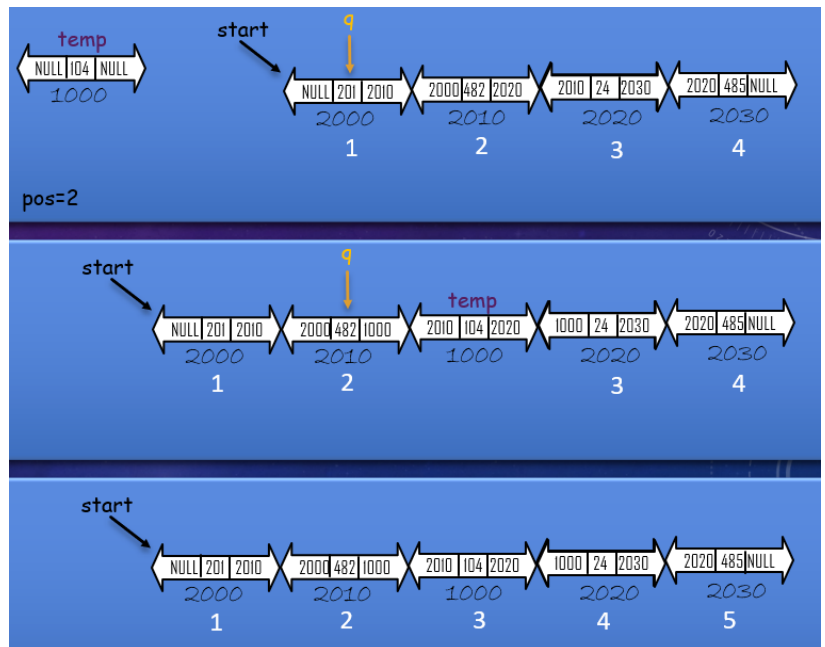
start->prev=newptr;

start=newptr;

INSERTION AFTER A GIVEN NODE

Algorithm:

- Step 1:** START
- Step 2:** Create a new node i.e. "temp"
temp=(struct node*)malloc(sizeof(node));
- Step 3:** Read data x, position 'pos'
q=start;
for(i=0;i<pos-1;i++)
{
 q=q->next;
}
- Step 4:** temp->data=x;
temp->prev=q;
temp->next=q->next;
q->next->prev=temp;
q->next=temp;
- Step 5:** STOP



Sample Program:

In the below program, the *start* pointer points to the beginning of the list, *save* pointer points to the last node. Function *Create_New_Node ()* takes one integer argument, allocates memory to create a new node and returns the pointer to the new node. (return-type: node*)

Function *Display ()* take node* type pointer as argument and displays the list from this pointer till the end of the doubly linked list.

The below program reads *data*, creates a new node dynamically, assigns read data to it and inserts this new node after the given node of linked list being pointed to by pointer '*q*'.

newptr->prev=q; newptr->next=q->next;

q->next->prev=newptr; q->next=newptr;

/* Insertion after a given node of the doubly linked list */

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

struct node
{
    int data;
    struct node *next;
    struct node *prev;
}*start,*save,*newptr,*temp,*q;

struct node* Create_New_Node(int);
void Display(struct node*);

void main()
{
    int i,x,size,pos;
    start=save=NULL;
    clrscr();
    printf("\n\t\t\t\t\t Creation of Doubly Linked List \n");
    printf("\n Enter Size of the list: ");
    scanf("%d",&size);
    printf("\n Enter data in the List : ");
    for(i=0;i<size;i++)
    {
        scanf("%d",&x);
        newptr=Create_New_Node(x);
        if(start==NULL)
        {
            start=save=newptr;
        }
        else
        {
            newptr->prev=save;
            save->next=newptr;
            save=newptr;
        }
    }
    printf("\n The List is : ");
    Display(start);
    printf("\n\n\t\t\t Insertion after a given node of the Doubly Linked List");
    printf("\n\n Enter data (any integer): ");
```

The screenshot shows the output of the C program. It displays the steps of creating a doubly linked list with 5 nodes (10, 20, 30, 40, 50) and then inserting a new node (35) at position 3. The final list is shown as 10 -> 20 -> 30 -> 35 -> 40 -> 50.

```
File Edit Search Run Compile Debug Project Options Window Help
Output
Creation of Doubly Linked List
Enter Size of the list: 5
Enter data in the List : 10 20 30 40 50
The List is : 10 -> 20 -> 30 -> 40 -> 50
Insertion after a given node of the Doubly Linked List
Enter data (any integer): 35
Enter the position : 3
Final List is : 10 -> 20 -> 30 -> 35 -> 40 -> 50
F1 Help F4 Scroll
```

```

scanf("%d",&x);
newptr=Create_New_Node(x);
printf("\n Enter the position : ");
scanf("%d",&pos);
q=start;
for(i=0;i<pos-1;i++)
{
    q=q->next;
}
newptr->prev=q;
newptr->next=q->next;
q->next->prev=newptr;
q->next=newptr;
printf("\n Final List is : ");
Display(start);
getch();
}

struct node* Create_New_Node(int n)
{
    temp=(struct node*)malloc(sizeof(struct node));
    temp->data=n;
    temp->next=NULL;
    temp->prev=NULL;
    return temp;
}

void Display(struct node* np)
{
    while(np->next!=NULL)
    {
        printf("%d -> ",np->data);
        np=np->next;
    }
    printf("%d ",np->data);
}

```

INSERTION BEFORE THE GIVEN NODE

Algorithm:

- Step 1:** START
- Step 2:** Create a new node i.e. "temp"
temp=(struct node*)malloc(sizeof(node));
- Step 3:** Read data x, position 'pos'
q=start;
for(i=0;i<pos-1;i++)
{
 q=q->next;
}
- Step 4:** temp->data=x;
temp->prev=q;
temp->next=q->next;
q->next=temp;
- Step 5:** STOP

Sample Program:

In the below program, the start pointer points to the beginning of the list, save pointer points to the last node. Function *Create_New_Node ()* takes one integer argument, allocates memory to create a new node and returns the pointer to the new node. (return-type: node*)

Function *Display ()* take node* type pointer as argument and displays the list from this pointer till the end of the doubly linked list.

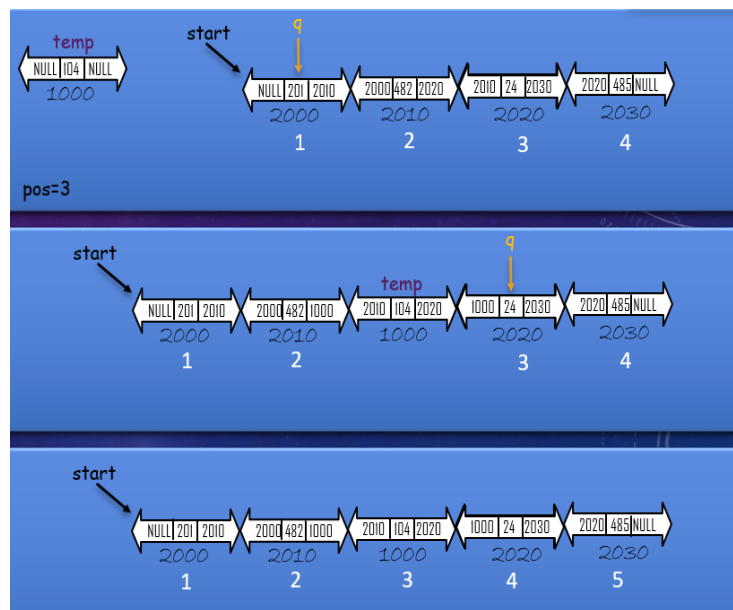
The below program reads data, creates a new node dynamically, assigns read data to it and inserts this new node before the given node of linked list being pointed to by pointer 'q'.

newptr->prev=q->prev;

newptr->next=q;

q->prev->next=newptr;

q->prev=newptr;



```

/* Insertion Before a given node of the doubly linked list */
#include<stdio.h>
#include<conio.h>

struct node
{
    int data;
    struct node *next;
    struct node *prev;
}*start,*save,*newptr,*temp,*q;

struct node* Create_New_Node(int);
void Display(struct node*);

void main()
{
    int i,x,size,pos;
    start=save=NULL;
    clrscr();
    printf("\n\t\t\t\t\t Creation of Doubly Linked List \n");
    printf("\n Enter Size of the list: ");
    scanf("%d",&size);
    printf("\n Enter data in the List : ");
    for(i=0;i<size;i++)
    {
        scanf("%d",&x);
        newptr=Create_New_Node(x);
        if(start==NULL)
        {
            start=save=newptr;
        }
        else
        {
            newptr->prev=save;
            save->next=newptr;
            save=newptr;
        }
    }
    printf("\n The List is : ");
    Display(start);
    printf("\n\t\t\t\t\t Insertion after a given node of the Doubly Linked List");
    printf("\n\n Enter data (any integer): ");
    scanf("%d",&x);
    newptr=Create_New_Node(x);
    printf("\n Enter the position : ");
    scanf("%d",&pos);
    q=start;
    for(i=0;i<pos-1;i++)
    {
        q=q->next;
    }
    newptr->prev=q->prev;
    newptr->next=q;
    q->prev->next=newptr;
    q->prev=newptr;
    printf("\n Final List is : ");
    Display(start);
    getch();
}

struct node* Create_New_Node(int n)
{
    temp=(struct node*)malloc(sizeof(struct node));
    temp->data=n;
    temp->next=NULL;
    temp->prev=NULL;
    return temp;
}

void Display(struct node* np)
{
    while(np->next!=NULL)
    {
        printf("%d -> ",np->data);
        np=np->next;
    }
    printf("%d ",np->data);
}

```

```

File Edit Search Run Compile Debug Project Options Window Help
Output
Creation of Doubly Linked List
Enter Size of the list: 5
Enter data in the List : 10 20 30 40 50
The List is : 10 -> 20 -> 30 -> 40 -> 50
Insertion after a given node of the Doubly Linked List
Enter data (any integer): 25
Enter the position : 3
Final List is : 10 -> 20 -> 25 -> 30 -> 40 -> 50
F1 Help F4 Scroll

```


INSERTION AT THE END

Algorithm:

- Step 1:** START
- Step 2:** Create a new node i.e. "temp"
temp=(struct node*)malloc(sizeof(node));
- Step 3:** q=start;
while(q->next!=NULL)
{
 q=q->next;
}
- Step 4:** q->next=temp;
temp->prev=q;
- Step 5:** STOP

Sample Program:

```
/* Insertion at the END of the doubly linked list */
```

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

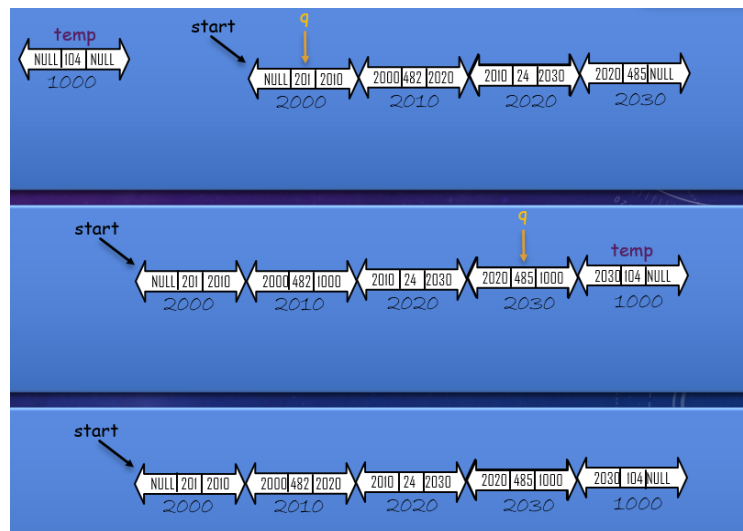
struct node
{
    int data;
    struct node *next;
    struct node *prev;
}*start,*save,*newptr,*temp,*q;

struct node* Create_New_Node(int);
void Display(struct node*);

void main()
{
    int i,x,size;
    start=save=NULL;
    clrscr();
    printf("\n\t\t\t\t\tCreation of Doubly Linked List \n");
    printf("\n Enter Size of the list: ");
    scanf("%d",&size);
    printf("\n Enter data in the List : ");
    for(i=0;i<size;i++)
    {
        scanf("%d",&x);
        newptr=Create_New_Node(x);
        if(start==NULL)
        {
            start=save=newptr;
        }
        else
        {
            newptr->prev=save;
            save->next=newptr;
            save=newptr;
        }
    }
    printf("\n The List is : ");
    Display(start);
    printf("\n\n\t\t\t\t\tInsertion after a given node of the Doubly Linked List");
    printf("\n\n Enter data (any integer): ");
    scanf("%d",&x);
    newptr=Create_New_Node(x);
    q=start;
    while(q->next!=NULL)
    {
        q=q->next;
    }
    q->next=newptr;
    newptr->prev=q;
    printf("\n Final List is : ");
    Display(start);
    getch();
}

struct node* Create_New_Node(int n)
{
    temp=(struct node*)malloc(sizeof(struct node));
    temp->data=n;
    temp->next=NULL;
    temp->prev=NULL;
    return temp;
}

void Display(struct node* np)
{
    while(np->next!=NULL)
    {
        printf("%d -> ",np->data);
        np=np->next;
    }
    printf("%d ",np->data);
}
```



```
File Edit Search Run Compile Debug Project Options Window Help
Output
5-11

Creation of Doubly Linked List

Enter Size of the list: 5
Enter data in the List : 10 20 30 40 50
The List is : 10 -> 20 -> 30 -> 40 -> 50

Insertion after a given node of the Doubly Linked List

Enter data (any integer): 60
Final List is : 10 -> 20 -> 30 -> 40 -> 50 -> 60 _
```

DELETION

A node can be deleted in three ways:

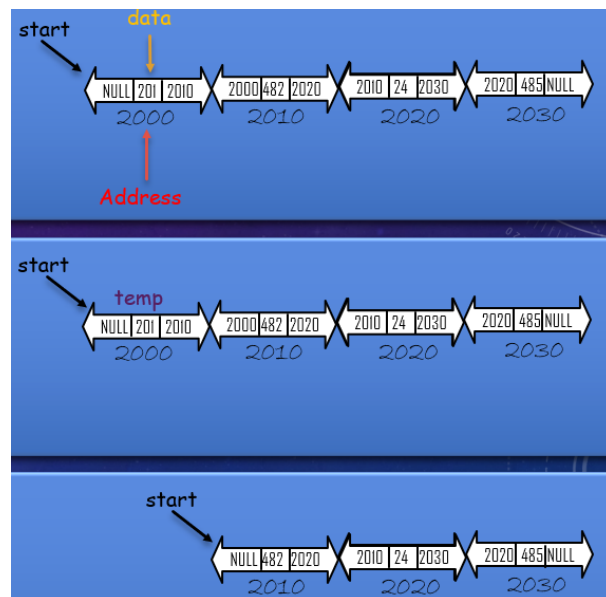
- 1) From the front of the DLL
- 2) From the middle/position
- 3) From the end

DELETION FROM THE BEGINNING

Algorithm:

- Step 1:** START
- Step 2:** temp=start;
- Step 3:** start=start->next;
start->prev=NULL;
- Step 4:** free(temp);
- Step 5:** STOP

Sample Program:



```
/* Deletion from the Beginning of the doubly linked list */
```

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

```
struct node
{
    int data;
    struct node *next;
    struct node *prev;
}*start,*save,*newptr,*temp;
```

```
struct node* Create_New_Node(int);
void Display(struct node*);
```

```
void main()
{
    int i,x,size;
    start=save=NULL;
    clrscr();
    printf("\n\t\t\t\t\t Creation of Doubly Linked List \n");
    printf("\n Enter Size of the list: ");
    scanf("%d",&size);
    printf("\n Enter data in the List : ");
    for(i=0;i<size;i++)
    {
        scanf("%d",&x);
        newptr=Create_New_Node(x);
        if(start==NULL)
        {
            start=save=newptr;
        }
        else
        {
            newptr->prev=save;
            save->next=newptr;
            save=newptr;
        }
    }
    printf("\n The List is : ");
    Display(start);
    printf("\n\n\t\t\t\t\t Deletion from the Beginning of the Doubly Linked List");
    temp=start;
```

```

    start=start->next;
    start->prev=NULL;
    free(temp);
    printf("\n Final List is : ");
    Display(start);
    getch();
}

```

```

struct node* Create_New_Node(int n)
{
    temp=(struct node*)malloc(sizeof(struct node));
    temp->data=n;
    temp->next=NULL;
    temp->prev=NULL;
    return temp;
}

```

```

void Display(struct node* np)
{
    while(np->next!=NULL)
    {
        printf("%d -> ",np->data);
        np=np->next;
    }
    printf("%d ",np->data);
}

```

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

File Edit Search Run Compile Debug Project Options

Output

Creation of Doubly Linked List

Enter Size of the list: 5

Enter data in the List : 10 20 30 40 50

The List is : 10 -> 20 -> 30 -> 40 -> 50

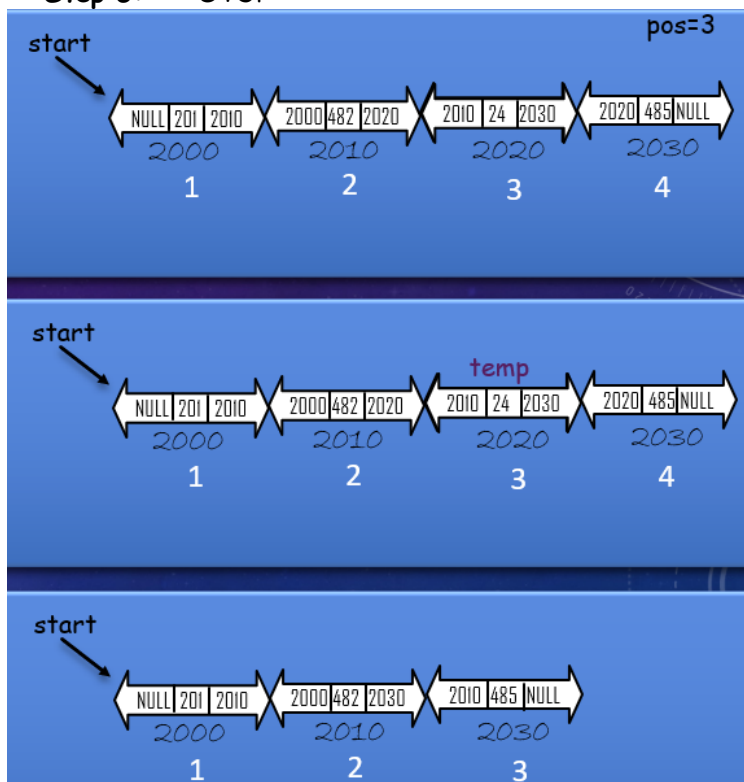
Deletion from the Beginning of the Doubly Linked List

Final List is : 20 -> 30 -> 40 -> 50 _

DELETION FROM THE MIDDLE/POSITION

Algorithm:

- Step 1: START
- Step 2: temp=start;
- Step 3: for(i=0;i<pos-1;i++)
 - {
 - temp=temp->next;
 - }
- temp->prev->next=temp->next;
- temp->next->prev=temp->prev;
- Step 4: free(temp);
- Step 5: STOP



Sample Program:

```
/* Deletion from the Middle/Position of the doubly linked list */

#include<stdio.h>
#include<conio.h>

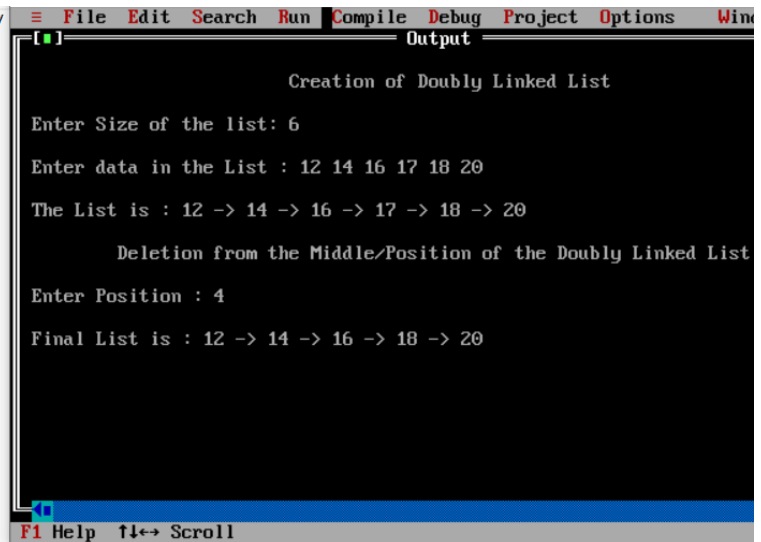
struct node
{
    int data;
    struct node *next;
    struct node *prev;
}*start,*save,*newptr,*temp;

struct node* Create_New_Node(int);
void Display(struct node*);

void main()
{
    int i,x,size,pos;
    start=save=NULL;
    clrscr();
    printf("\n\t\t\t\t\t Creation of Doubly Linked List \n");
    printf("\n Enter Size of the list: ");
    scanf("%d",&size);
    printf("\n Enter data in the List : ");
    for(i=0;i<size;i++)
    {
        scanf("%d",&x);
        newptr=Create_New_Node(x);
        if(start==NULL)
        {
            start=save=newptr;
        }
        else
        {
            newptr->prev=save;
            save->next=newptr;
            save=newptr;
        }
    }
    printf("\n The List is : ");
    Display(start);
    printf("\n\n\t\t\t\t\t Deletion from the Middle/Position of the Doubly Linked List");
    printf("\n\n Enter Position : ");
    scanf("%d",&pos);
    temp=start;
    for(i=0;i<pos-1;i++)
    {
        temp=temp->next;
    }
    temp->next->prev=temp->prev;
    temp->prev->next=temp->next;
    free(temp);
    printf("\n Final List is : ");
    Display(start);
    getch();
}

struct node* Create_New_Node(int n)
{
    temp=(struct node*)malloc(sizeof(struct node));
    temp->data=n;
    temp->next=NULL;
    temp->prev=NULL;
    return temp;
}

void Display(struct node* np)
{
    while(np->next!=NULL)
    {
        printf("%d -> ",np->data);
        np=np->next;
    }
    printf("%d ",np->data);
}
```



```
File Edit Search Run Compile Debug Project Options Win
Output

Creation of Doubly Linked List

Enter Size of the list: 6

Enter data in the List : 12 14 16 17 18 20

The List is : 12 -> 14 -> 16 -> 17 -> 18 -> 20

Deletion from the Middle/Position of the Doubly Linked List

Enter Position : 4

Final List is : 12 -> 14 -> 16 -> 18 -> 20

F1 Help F4 Scroll
```

DELETION FROM THE END

Algorithm:

- Step 1:** START
- Step 2:** temp=start;
- Step 3:** while(temp->next!=NULL)
 - {
 - temp=temp->next;
 - }
 - temp->prev->next=NULL;
- Step 4:** free(temp);
- Step 5:** STOP

Sample Program:

```
/* Deletion from the END of the doubly linked list */
```

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

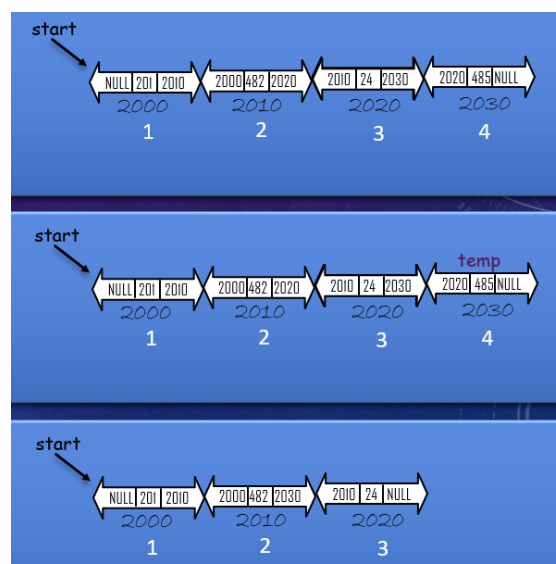
struct node
{
    int data;
    struct node *next;
    struct node *prev;
}*start,*save,*newptr,*temp;

struct node* Create_New_Node(int);
void Display(struct node*);

void main()
{
    int i,x,size;
    start=save=NULL;
    clrscr();
    printf("\n\t\t\t\t\t Creation of Doubly Linked List \n");
    printf("\n Enter Size of the list: ");
    scanf("%d",&size);
    printf("\n Enter data in the List : ");
    for(i=0;i<size;i++)
    {
        scanf("%d",&x);
        newptr=Create_New_Node(x);
        if(start==NULL)
        {
            start=save=newptr;
        }
        else
        {
            newptr->prev=save;
            save->next=newptr;
            save=newptr;
        }
    }
    printf("\n The List is : ");
    Display(start);
    printf("\n\n\t\t\t\t\t Deletion from the END of the Doubly Linked List");
    temp=start;
    while(temp->next!=NULL)
    {
        temp=temp->next;
    }
    temp->prev->next=NULL;
    free(temp);
    printf("\n Final List is : ");
    Display(start);
    getch();
}

struct node* Create_New_Node(int n)
{
    temp=(struct node*)malloc(sizeof(struct node));
    temp->data=n;
    temp->next=NULL;
    temp->prev=NULL;
    return temp;
}

void Display(struct node* np)
{
    while(np->next!=NULL)
    {
        printf("%d -> ",np->data);
        np=np->next;
    }
    printf("%d ",np->data);
}
```



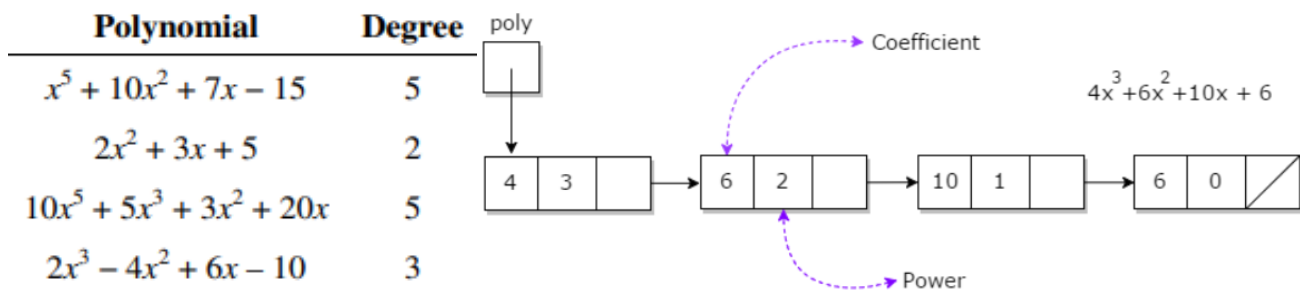
```
DOSBox 0.74, Cpu speed: max 100% cycles, Frameskip 0, Program: TC
File Edit Search Run Compile Debug Project Opt
Output
Creation of Doubly Linked List
Enter Size of the list: 5
Enter data in the List : 10 20 30 40 50
The List is : 10 -> 20 -> 30 -> 40 -> 50
Deletion from the END of the Doubly Linked List
Final List is : 10 -> 20 -> 30 -> 40 _
```


Polynomials

A polynomial is an expression that contains more than two terms. A term is made up of coefficient and exponent.

Example: $p(x) = 4x^3 + 8x^2 + 16x + 20$

Here, 4,8,16,20 are coefficients and 3,2,1,0 are exponents.



Polynomial Representation

Polynomial can be represented in 2 ways:

- Array representation
- Linked representation

REPRESENTATION OF POLYNOMIAL USING ARRAYS

Array representation assumes that the exponents of the given expression are arranged from 0 to the highest value (degree), which is represented by the subscript of the array beginning with 0. The coefficients of the respective exponent are placed at an appropriate index in the array.

Example: Consider the following expression,

$$p(x) = 4x^3 + 8x^2 + 16x + 20$$

The array representation of the above polynomial is:

Index	0	1	2	3
Coefficient	20	16	8	4

Drawbacks:

- If the polynomial is like $p(x) = 4^{1000} + 3^{500} + 120$, it requires very space starting from index number 0 till 1000 and hence wastage of memory is very large.
- If the polynomial expression is like $p(x) = 4^{10} + 3^5 + 12$, it has only three terms but the space allocated will the highest exponent value. Hence sometimes size of array is more than the no. of terms.

REPRESENTATION OF POLYNOMIAL USING ARRAY OF STRUCTURE

A polynomial may be represented using array of structure. A structure may be defined such that it contains two parts: one is the coefficient and second is the corresponding exponent. The structure definition may be given as shown below:

```
struct polynomial
{
    int coef;        //Coefficient
    int expo;        //Exponent
};
```

If we declare the following in main function:

struct polynomial p[50];

p[50] allocates 50 memory blocks with size 2bytes, so total size is 100 bytes of memory.

So, still there will be a wastage of memory, depicted in the picture.

So, to overcome the problem of memory wastage, Polynomial is represented using Linked List.

WASTE OF MEMORY

Index	Exponent	Coefficient
0	3	8
1	1	7
2	0	5
⋮	0	0
⋮	0	0
⋮	0	0
⋮	0	0
⋮	0	0
⋮	0	0
47	0	0
48	0	0
49	0	0

REPRESENTATION OF POLYNOMIAL USING LINKED LIST

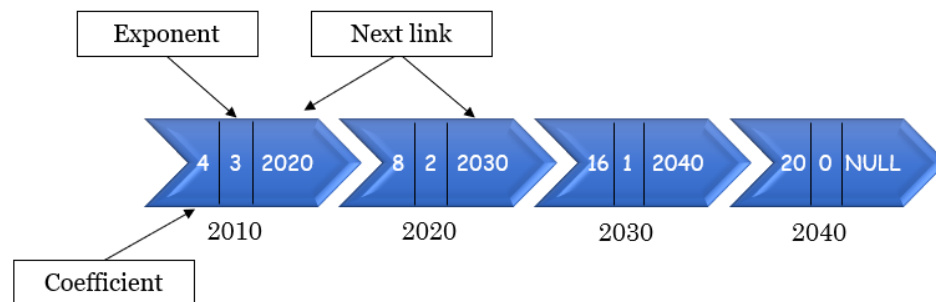
An important application of linked list is to represent polynomial and their manipulations. Main advantage of linked list for polynomial representation is that it can accommodate a number of polynomials of growing sizes so that combined size does not exceed the total memory available. The structure of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

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

Consider the following polynomial expression,

$$p(x) = 4x^3 + 8x^2 + 16x + 20$$

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



Operation on Polynomials

There are two types of polynomial operations in C,

- Evaluation (Solving the equation with given 'x' value)
- Addition of two polynomial expressions

Evaluation of Polynomial

Here, the value of 'x' is given and we have to solve the expression. Consider the following polynomial expression,

$$p(x) = 4x^3 + 8x^2 + 16x + 20$$

If the value of x is 2, then

$$\begin{aligned} p(x) &= 4(2)^3 + 8(2)^2 + 16(2) + 20 \\ &= 32 + 32 + 32 + 20 = 116 \end{aligned}$$

Program:

The below program is done using linked list. First, we have created a structure *poly* representing *polynomial*. Then, we have *coef*, *expo*, **next* representing coefficient, exponent, next node address respectively as structure members and we have created four variables of struct *poly* for the basic operations.

Functions used are:

Create_Node() for creating a node containing the values of coefficient, exponent and next node address.

```
struct poly* Create_Node(int C,int E)
{
    temp=(struct poly*)malloc(sizeof(struct poly));
    temp->coef=C;
    temp->expo=E;
    temp->next=NULL;
    return temp;
}
```

Display() for displaying the polynomial expression in equation form.

```
void Display(struct poly* np)
{
    while(np->next!=NULL)
    {
        printf("%dx^%d + ",np->coef,np->expo);
        np=np->next;
    }
    printf("%dx^%d ",np->coef,np->expo);
}
```

Evaluate() for evaluating the polynomial expression based on value of x. }

```
int Evaluate(struct poly* ptr,int X)
{
    int value=0,power,exval;
    while(ptr->next!=NULL)
    {
        exval=1;
        power=ptr->expo;
        while(power!=0)
        {
            exval*=X;
            power--;
        }
        value+=ptr->coef*exval;
        ptr=ptr->next;
    }
    value+=ptr->coef;
    return value;
}
```

```

/* Evaluation of a Polynomial Expression. */
#include<stdio.h>
#include<stdlib.h>
#include<math.h>

struct poly
{
    int coef;
    int expo;
    struct poly *next;
}*start,*rear,*newptr,*temp;

struct poly* Create_Node(int,int);
void Display(struct poly*);
int Evaluate(struct poly*,int);

int main()
{
    int i,c,e,x,ans,size;
    start=rear=NULL;
    printf("\n Enter no. of elements : ");
    scanf("%d",&size);
    for(i=0;i<size;i++)
    {
        printf("\n Enter coefficient : ");
        scanf("%d",&c);
        printf("\n Enter exponent : ");
        scanf("%d",&e);
        newptr=Create_Node(c,e);
        if(start==NULL)
        {
            start=rear=newptr;
        }
        else
        {
            rear->next=newptr;
            rear=newptr;
        }
    }
    printf("\n The Polynomial Expression is : ");
    Display(start);
    printf("\n Enter value of x : ");
    scanf("%d",&x);
    ans=Evaluate(start,x);
    printf("\n\t Answer = %d",ans);
    return 0;
}

struct poly* Create_Node(int C,int E)
{
    temp=(struct poly*)malloc(sizeof(struct poly));
    temp->coef=C;
    temp->expo=E;
    temp->next=NULL;
    return temp;
}

void Display(struct poly* np)
{
    while(np->next!=NULL)
    {
        printf("%dx^%d + ",np->coef,np->expo);
        np=np->next;
    }
    printf("%dx^%d ",np->coef,np->expo);
}

int Evaluate(struct poly* ptr,int X)
{
    int value=0,power,exval;
    while(ptr->next!=NULL)
    {
        exval=1;
        power=ptr->expo;
        while(power!=0)
        {
            exval*=X;
            power--;
        }
        value+=ptr->coef*exval;
        ptr=ptr->next;
    }
    value+=ptr->coef;
    return value;
}

```

```

Enter no. of elements : 4
Enter coefficient : 4
Enter exponent : 3
Enter coefficient : 8
Enter exponent : 2
Enter coefficient : 16
Enter exponent : 1
Enter coefficient : 20
Enter exponent : 0
The Polynomial Expression is : 4x^3 + 8x^2 + 16x^1 + 20x^0
Enter value of x : 2
Answer = 116

```

Addition of two Polynomial Expressions

In order to add two polynomials, say P and Q to get a resultant polynomial R , we have to compare their terms starting at the first nodes and moving towards the end one-by-one. Two pointers $Pptr$ and $Qptr$ are used to move along the terms of P and Q . There may be three cases during the comparison between terms in two polynomials.

Case 1: Exponents of two terms are equal. In this case coefficients in two nodes are to be added and a new term will be created with the values.

$$Rptr.coeff = Pptr.Coeff + Qptr.Coeff$$

And,

$$Rptr.Exp = Pptr.Exp$$

Case 2: $Pptr.Exp > Qptr.Exp$ i.e., the exponent of the current term in P is greater than the exponent of the current term in Q . Then, a duplicate of the current term in P is created and to be inserted in the polynomial R .

Case 3: $Pptr.Exp < Qptr.Exp$ i.e., the exponent of the current term in P is less than the exponent of the current term in Q . Then, a duplicate of the current term in Q is created and to be inserted in the polynomial R .

Example: Let's us consider $P(x) = 4x^{10} + 6x^5 + 8x^2 + 2$ and $Q(x) = 6x^8 + 4x^5 + 5x^2 + 10$ then the resultant polynomial will be $R(x) = 4x^{10} + 6x^8 + 10x^5 + 13x^2 + 12$

Sample Program:

The following program is illustrated using linked list and output for the above 3 cases are also given below. This program can also be written using arrays of structure. The following program uses functions `Create_Node()`, `Display()`, `Add()` for creating node, displaying polynomial expression, and performing addition of 2 polynomials respectively.

```
/* Addition of two Polynomial Expressions. */

#include<stdio.h>
#include<stdlib.h>

struct poly
{
    int coef;
    int expo;
    struct poly *next;
}*newptr,*rear,*Pptr,*Qptr,*Rptr,*temp;

struct poly* Create_Node(int,int);
void Display(struct poly*);
struct poly* Add(struct poly*,struct poly*);

int main()
{
    int i,c,e,x,ans,size;
    Pptr=Qptr=Rptr=newptr=rear=NULL;
    printf("\n Creating Polynomial Expression 1 : \n");
    printf("\n Enter no. of elements : ");
    scanf("%d",&size);
    for(i=0;i<size;i++)
    {
        printf("\n Enter coefficient : ");
        scanf("%d",&c);
        printf("\n Enter exponent : ");
        scanf("%d",&e);
        newptr=Create_Node(c,e);
        if(Pptr==NULL)
        {
            Pptr=rear=newptr;
        }
        else
        {
            rear->next=newptr;
            rear=newptr;
        }
    }
    newptr=rear=NULL;
```

Creating Polynomial Expression 1 :

Enter no. of elements : 4

Enter coefficient : 4

Enter exponent : 10

Enter coefficient : 6

Enter exponent : 5

Enter coefficient : 8

Enter exponent : 2

Enter coefficient : 2

Enter exponent : 0

Creating Polynomial Expression 2 :

Enter no. of elements : 4

Enter coefficient : 6

Enter exponent : 8

Enter coefficient : 4

Enter exponent : 5

Enter coefficient : 5

Enter exponent : 2

Enter coefficient : 10

Enter exponent : 0

The Polynomial Expressions are :

Expression 1 : $P(x) = 4x^{10} + 6x^5 + 8x^2 + 2x^0$

Expression 2 : $Q(x) = 6x^8 + 4x^5 + 5x^2 + 10x^0$

Resultant Polynomial Expression is :

$R(x) = 6x^8 + 10x^5 + 13x^2 + 12x^0 + 4x^{10}$

```

printf("\n Creating Polynomial Expression 2 : \n");
printf("\n Enter no. of elements : ");
scanf("%d",&size);
for(i=0;i<size;i++)
{
    printf("\n Enter coefficient : ");
    scanf("%d",&c);
    printf("\n Enter exponent : ");
    scanf("%d",&e);
    newptr=Create_Node(c,e);
    if(Qptr==NULL)
    {
        Qptr=rear=newptr;
    }
    else
    {
        rear->next=newptr;
        rear=newptr;
    }
}
printf("\n The Polynomial Expressions are : \n");
printf("\n\t Expression 1 : P(x) = ");
Display(Pptr);
printf("\n\t Expression 2 : Q(x) = ");
Display(Qptr);
Rptr=Add(Pptr,Qptr);
printf("\n\n Resultant Polynomial Expression is : \n ");
printf("\n\t R(x) = ");
Display(Rptr);
return 0;
}

struct poly* Create_Node(int C,int E)
{
    temp=(struct poly*)malloc(sizeof(struct poly));
    temp->coef=C;
    temp->expo=E;
    temp->next=NULL;
    return temp;
}

void Display(struct poly* np)
{
    while(np->next!=NULL)
    {
        printf("%dx^%d + ",np->coef,np->expo);
        np=np->next;
    }
    printf("%dx^%d ",np->coef,np->expo);
}

struct poly* Add(struct poly* pptr,struct poly* qptr)
{
    struct poly *rptr,*tptr,*tqptr;
    int flag,count1=0,count2=0;
    newptr=rear=rptr=NULL;
    tptr=pptr;
    tqptr=qptr;
    while(pptr->next!=NULL)
    {
        pptr=pptr->next;
        count1++;
    };
    while(qptr->next!=NULL)
    {
        qptr=qptr->next;
        count2++;
    };
    pptr=tptr;
    qptr=tqptr;
    if(count1>count2)
    {
        while(pptr!=NULL)
        {
            flag=0;
            while(qptr!=NULL)
            {
                if(pptr->expo==qptr->expo)
                {
                    newptr=Create_Node(pptr->coef+qptr->coef,pptr->expo);
                    flag=1;
                    qptr->expo=-1;
                    break;
                }
            }
        }
    }
}

```

```

Creating Polynomial Expression 1 :
Enter no. of elements : 4

Enter coefficient : 4
Enter exponent : 3

Enter coefficient : 3
Enter exponent : 2

Enter coefficient : 2
Enter exponent : 1

Enter coefficient : 1
Enter exponent : 0

Creating Polynomial Expression 2 :
Enter no. of elements : 5

Enter coefficient : 3
Enter exponent : 5

Enter coefficient : 7
Enter exponent : 4

Enter coefficient : 9
Enter exponent : 3

Enter coefficient : 8
Enter exponent : 1

Enter coefficient : 5
Enter exponent : 0

The Polynomial Expressions are :

Expression 1 : P(x) = 4x^3 + 3x^2 + 2x^1 + 1x^0
Expression 2 : Q(x) = 3x^5 + 7x^4 + 9x^3 + 8x^1 + 5x^0

Resultant Polynomial Expression is :

R(x) = 3x^5 + 7x^4 + 13x^3 + 10x^1 + 6x^0 + 3x^2
-----
Creating Polynomial Expression 1 :
Enter no. of elements : 4

Enter coefficient : 4
Enter exponent : 3

Enter coefficient : 3
Enter exponent : 2

Enter coefficient : 2
Enter exponent : 1

Enter coefficient : 1
Enter exponent : 0

Creating Polynomial Expression 2 :
Enter no. of elements : 3

Enter coefficient : 5
Enter exponent : 3

Enter coefficient : 7
Enter exponent : 2

Enter coefficient : 9
Enter exponent : 0

The Polynomial Expressions are :

Expression 1 : P(x) = 4x^3 + 3x^2 + 2x^1 + 1x^0
Expression 2 : Q(x) = 5x^3 + 7x^2 + 9x^0

Resultant Polynomial Expression is :

R(x) = 9x^3 + 10x^2 + 2x^1 + 10x^0

```

```

        qptr=qptr->next;
    }
    if(flag==0)
    {
        newptr=Create_Node(pptr->coef,pptr->expo);
    }
    if(rptr==NULL)
    {
        rptr=rear=newptr;
    }
    else
    {
        rear->next=newptr;
        rear=newptr;
    }
    pptr=pptr->next;
    qptr=tqptr;
}
while(qptr!=NULL)
{
    if(qptr->expo!=-1)
    {
        newptr=Create_Node(qptr->coef,qptr->expo);
        if(rptr==NULL)
        {
            rptr=rear=newptr;
        }
        else
        {
            rear->next=newptr;
            rear=newptr;
        }
    }
    qptr=qptr->next;
}
}
else
{
    while(qptr!=NULL)
    {
        flag=0;
        while(pptr!=NULL)
        {
            if(qptr->expo==pptr->expo)
            {
                newptr=Create_Node(qptr->coef+pptr->coef,qptr->expo);
                flag=1;
                pptr->expo=-1;
                break;
            }
            pptr=pptr->next;
        }
        if(flag==0)
        {
            newptr=Create_Node(qptr->coef,qptr->expo);
        }
        if(rptr==NULL)
        {
            rptr=rear=newptr;
        }
        else
        {
            rear->next=newptr;
            rear=newptr;
        }
        qptr=qptr->next;
        pptr=tpptr;
    }
    while(pptr!=NULL)
    {
        if(pptr->expo!=-1)
        {
            newptr=Create_Node(pptr->coef,pptr->expo);
            if(rptr==NULL)
            {
                rptr=rear=newptr;
            }
            else
            {
                rear->next=newptr;
                rear=newptr;
            }
        }
        pptr=pptr->next;
    }
}
return rptr;
}
}

```


Circular List Representation of Polynomials

possible to free all the nodes in a list more efficiently by employing a circular list

an efficient erase algorithm for circular lists

- maintain a chain of nodes(*av* list, available-space list) that have been "deleted"
- if *av* list is empty, then need to use command *new* to create a new node
- let *av* be a static class member of `CircList<Type>` of type `ListNode<Type>`
- use the functions `CircList::GetNode` and `CircList::RetNode` instead of *new* and *delete*

Getting a node

```
template <class Type>
ListNode <Type >* CircList ::GetNode()
// Provide a node for use.
{
    ListNode <Type >* x;
    if (!av) x = new ListNode <Type >;
    else { x = av; av = av->link; }
    return x;
}
```

Returning a node

```
template <class Type>
void CircList <Type >::RetNode(ListNode <Type >* x)
// Free the node pointed to by x.
{
    x->link = av;
    av = x;
}
```



erase a circular list by function `CircList<Type>::~~CircList`

Erasing a circular list

```
template <class KeyType>
void CircList <Type >::~~CircList()
// Erase the circular list pointed by first.
{
    if (first) {
        ListNode* second = first->link; // second node
        first->link = av; // first node linked to av
        av = second; //second node of list becomes front of av list
        first = 0;
    }
}
```

Equivalence Classes

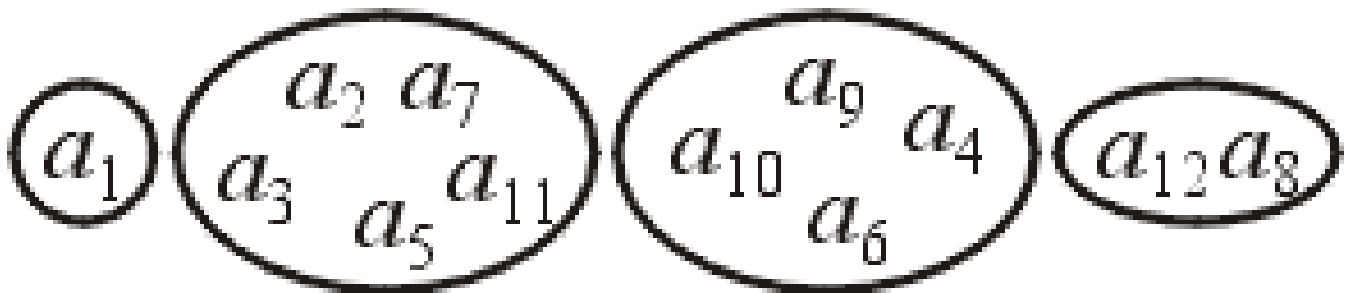
Local Definition

- An equivalence relation on a finite collection of objects may be described as follows: each object is related to itself and the relationship is symmetric; also, if two objects are related to the same third object, the two objects themselves must also be related.

Mathematical Definition

- A binary relationship \sim between two objects is said to be an equivalence relation if:
- Each object is related to itself: $a \sim a$
- The relationship is symmetric: $a \sim b$ if and only if $b \sim a$
- The relationship is transitive: $a \sim b$ and $b \sim c$ implies that $a \sim c$
- An equivalence relation allows one to partition a set of objects into *equivalence classes*: that is, a collection of subsets where all entries in a given equivalence class are related to all other elements within the same equivalence class.

Given n objects which satisfy an equivalence relation, we may partition them such that all objects in the same partition are related to each other as is shown:



- An equivalence relation is a relation operator that observes three properties:
- Reflexive: $(a R a)$, for all a
- Symmetric: $(a R b)$ if and only if $(b R a)$
- Transitive: $(a R b)$ and $(b R c)$ implies $(a R c)$
- Put another way, equivalence relations check if operands are in the same

Equivalence class:

The set of elements that are all related to each other via an equivalence relation Due to transitivity each member can only be a member of one equivalence class

Thus, equivalence classes are disjoint sets

Choose any distinct sets S and T , $S \cap T = \emptyset$ equivalence class

- Store elements in equivalence (general) trees
- Use the tree's root as equivalence class label
- find returns root of containing tree
- union merges tree
- Since all operations only search up the tree, we can store in an array

Implementation:

Index all objects from 0 to $N-1$

Store a parent array such that $s[i]$ is the index of i 's parent If i is a root, store the negative size of its tree*

find follows $s[i]$ until negative, returns index

union(x, y) points the root of x 's tree to the root of y 's tree



```
void equivalence()
```

```
{
    read n; // read in number of objects
    initialize seq to 0 and out to FALSE;
    while more pairs // input pairs
    {
        read the next pair (i,j);
        put j on the seq[i] list;
        put i on the seq[j] list;
    }
    for (i = 0 ; i < n ; i++)
        if (out[i] == FALSE) {
            out[i] = TRUE ;
            output the equivalence class that contains object i ;
        }
} // end of equivalence
```

Program 4.28: A more detailed version of equivalence algorithm

```
void equivalence()
```

```
{
    initialize;
    while more pairs
    {
        read the next pair (i,j);
        process this pair;
    }
    initialize for output;
    for (each object not yet output)
        output the equivalence class that contains this object;
} // end of equivalence
```

Program 4.27: First pass at equivalence algorithm

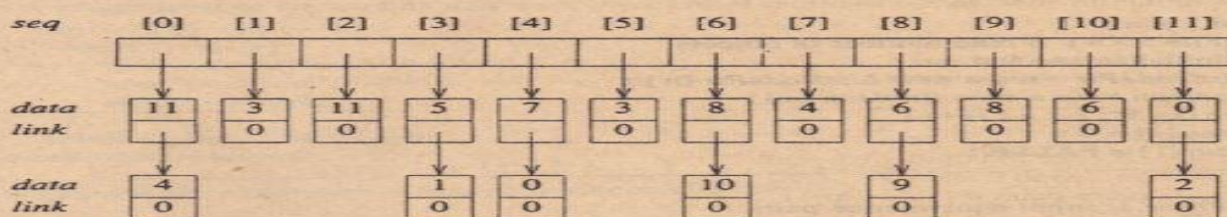


Figure 4.24: Lists after pairs have been input

Sparse Matrices

Sparse matrix is a matrix which contains very few non-zero elements. i.e. if a matrix contains a greater number of zeroes than the non-zero values. Such type of matrix is known as Sparse matrix.

Dense Matrix

1	2	31	2	9	7	34	22	11	5
11	92	4	3	2	2	3	3	2	1
3	9	13	8	21	17	4	2	1	4
8	32	1	2	34	18	7	78	10	7
9	22	3	9	8	71	12	22	17	3
13	21	21	9	2	47	1	81	21	9
21	12	53	12	91	24	81	8	91	2
61	8	33	82	19	87	16	3	1	55
54	4	78	24	18	11	4	2	99	5
13	22	32	42	9	15	9	22	1	21

Sparse Matrix

1	.	3	.	9	.	3	.	.	.
11	.	4	2	1
.	.	1	.	.	.	4	.	1	.
8	.	.	.	3	1
.	.	.	9	.	.	1	.	17	.
13	21	.	9	2	47	1	81	21	9
.
.	.	.	.	19	8	16	.	.	55
54	4	.	.	.	11
.	.	2	22	.	21

Advantages of Sparse Matrices

- **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 Matrices Representation

We can represent sparse matrix in memory in two different ways:

- Triplet representation (Array Representation or three columns terms)
- Linked representation

Triplet Representation

Suppose the elements are in the form of rows and columns i.e. in the matrix form. Let us consider the following data:

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

The location of non-zero values are:

(0,0) (0,1) (0,2) (0,3) **(0,4)** (0,5)
 (1,0) **(1,1)** (1,2) (1,3) (1,4) (1,5)
(2,0) (2,1) (2,2) **(2,3)** (2,4) (2,5)
 (3,0) (3,1) (3,2) (3,3) (3,4) **(3,5)**
 (4,0) (4,1) **(4,2)** (4,3) (4,4) (4,5)

Now the triplet representation will be in the form of three fields *rows*, *columns* and *values*. we consider only non-zero values along with their row and column index values. In this representation, the 0th row stores the total number of rows, total number of columns and the total number of non-zero values in the sparse matrix.

Linked Representation

In linked representation, we use a linked list data structure to represent a sparse matrix. In this linked list, we use two different nodes namely **header node** and **element node**.

Header Node Element Node

IndexValue	row	column	value
down	down/up	right	right