UNIT 1:-Algorithm Analysis:-

Mathematical Background, Model, Analysis and Run Time Calculations, Lists: Abstract Data Types, List using arrays and pointers, Singly Linked, Doubly Linked, Circular Linked Lists, Polynomial ADT.

Algorithm Definition: - An algorithm is a set of instructions that are used to solve a problem.

Properties of an algorithm:-

1. Input
2. Output
3. Finiteness
4. Definiteness
5. Effectiveness

1. **Input: -** An algorithm has zero (0) or more inputs.

2. **Output: -** An algorithm must produce one (1) or more outputs.

3. **Finiteness:-**An algorithm must contain a finite number of steps.

4. **Definiteness: -** Each step of an algorithm must be clear and unambiguous.

5. **Effectiveness: -** An algorithm should be effective i.e. operations can be performed with the given inputs in a finite period of time by a person using paper and pencil.

Performance Analysis (or) Analysis of an algorithm:-

Analysis of algorithm is the task of determining how much computing time (Time complexity) and storage (Space complexity) required by an algorithm.

We can analyze performance of an algorithm using Time complexity and Space complexity.

Time Complexity: - The amount of time that an algorithm requires for its execution is known as time complexity.
Time complexity is mainly classified into 3 types.

1. Best case time complexity
2. Worst case time complexity
3. Average case time complexity

1. Best case time complexity: - If an algorithm takes minimum amount of time for its execution then it is called as Best case time complexity.

Ex: - While searching an element using linear search, if key element found at first position then it is best case.

2. Worst case time complexity: - If an algorithm takes maximum amount of time for its execution then it is called as Worst case time complexity.

Ex: - While searching an element using linear search, if key element found at last position then it is worst case.

3. Average case time complexity: - If an algorithm takes average amount of time for its execution then it is called as Average case time complexity.

Ex: - While searching an element using linear search, if key element found at middle position then it is average case.

There are 2 types of computing times.

1. Compilation time
2. Execution time **or** run time

- The time complexity is generally calculated using execution time or run time.
- It is difficult to calculate time complexity using clock time because in a multiuser system, the execution time depends on many factors such as system load, number of other programs that are running.
- So, time complexity is generally calculated using Frequency count (step count) and asymptotic notations.

Frequency count (or) Step count:-

It denotes the number of times a statement to be executed.
Generally count value will be given depending upon corresponding statements.

- For comments, declarations the frequency count is zero (0).
- For Assignments, return statements the frequency count is 1.
- Ignore lower order exponents when higher order exponents are present.
- Ignore constant multipliers.

Ex: - A sample program to calculate time complexity of sum of cubes of 'n' natural numbers.

```
int   sum(int  n)              ----------0
{                              ----------0
        int  i,s;              ----------0
        s=0;                   ----------1
        for(i=1;i<=n;i++)      ----------(n+1)
                s=s+i*i*i;     ----------n
        return s;              ----------1
}
                                    _____
                                    2n+3
                                    _____
```

So, time complexity=O(n)

General rules or norms for calculating time complexity:-

Rule 1:-The running time of for loop is the running time of statements in for loop.

```
Ex:- for(i=0;i<n;i++)   -----------(n+1)
         s=s+i;    ------------ n
                                 _____
                                 2n+1
                                 _____
```

So, time complexity=O(n)

Rule 2:-The total running time of statements inside a group of nested loops is the product of the sizes of all loops.

```
Ex:- for(i=0;i<n;i++)              ---------------- (n+1)
       for(j=0;j<n;j++)            ---------------- n(n+1)
           c[i][j]=a[i][j]+b[i][j]; -------------   (n*n)
                                                  _____________
                                                  2n^2+2n+1
                                                  _____________
```

So, time complexity=$O(n^2)$

Rule 3:-The running time is the maximum one.

```
Ex:- for(i=0;i<n;i++)              -----------  (n+1)
       a[i]=0;                     -----------     n
     for(i=0;i<n;i++)              ------------ (n+1)
       for(j=0;j<n;j++)            ------------ n(n+1)
           a[i]=a[i]+a[j]+i+j;     -------------  n^2
                                              _____________
                                              2n^2+4n+2
                                              _____________
```

So, time complexity=$O(n^2)$

Rule 4:- if-else

```
if(cond)
    s1
else
    s2
```

Here running time is maximum of running times of s1 and s2.

```
Ex:- int sum(int  n)              --------------  0
     {                            ---------------  0
        int  s,i;                 --------------  0
        s=0;                      --------------  1
        if(n<=0)                  --------------  1
            return  n;            -------------    1
        else                      -------------   0
        {                         -------------   0
            for(i=0;i<n;i++)  --------      n+1
```

```
                    s=s+i;      ----------  n
                return  s;      -----------    1
            }
        }
                                ____________
                                    2n+5
                                ____________________
```

So, time complexity=O(n)

Asymptotic notations:-

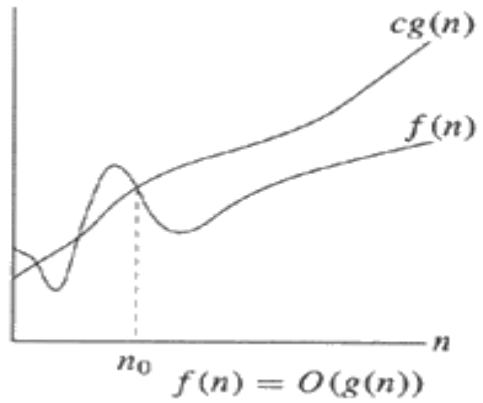
Asymptotic notations are used to calculate time complexity of algorithm.
Using asymptotic notations we can calculate best case, worst case and average case time complexity.
There are 5 types of time complexities.
1. Big Oh notation (O)
2. Big Omega notation (Ω)
3. Theta notation (θ)
4. Little Oh notation (o)
5. Little omega notation (ω)

1. Big Oh notation (O):-

- It is a method of representing upper bound of algorithm's running time.
- Using Big Oh notation we can calculate maximum amount of time taken by an algorithm for its execution.
- So, Big Oh notation is useful to calculate worst case time complexity.

Definition: - Let f(n) , g(n) be 2 non negative functions. Then f(n)=O(g(n)), if there are 2 positive constants c, n_0 such that f(n)<=c*g(n) ∀ n>=n_0.

Graphical representation of Big Oh notation(O):-



Ex:- f(n)=3n+2
g(n)=n;
To show f(n)=O(g(n))
f(n)<=c*g(n) , c>0, n_0>=1

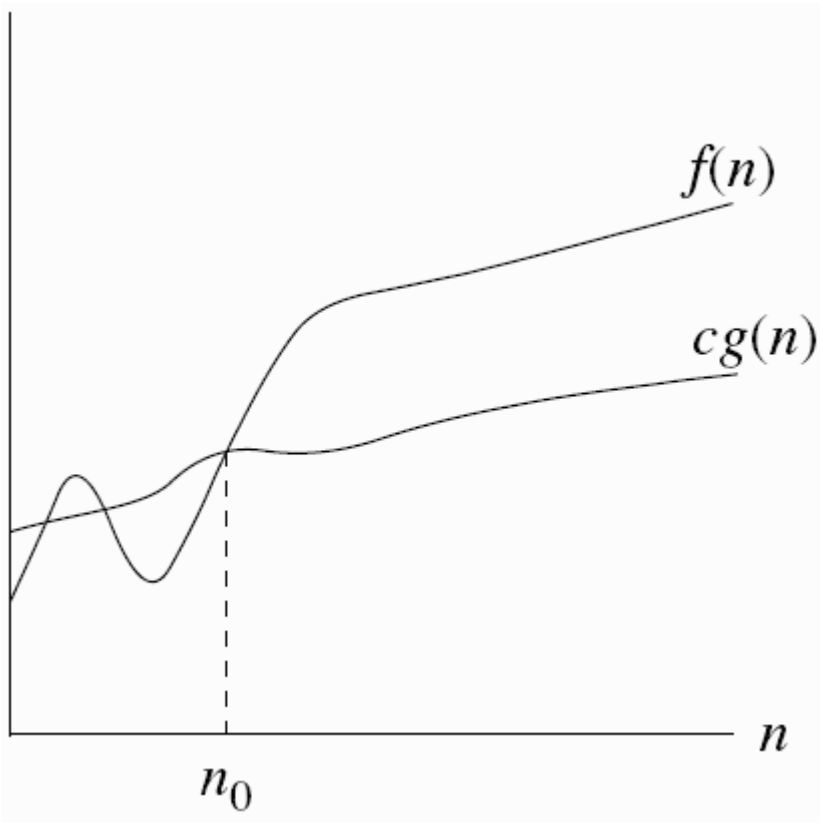
3n+2<=c*n
Let C=4, n_0=1 then 5<=4 wrong
n_0=2 then 8<=8 correct
n_0=3 then 11<=12 correct
n_0=4 then 14<=16 correct
So, n_0>=2
Hence, 3n+2<=4*n ∀ n>=2

2. Big Omega notation (Ω):-

- It is a method of representing lower bound of algorithm's running time.
- Using Big Omega notation we can calculate minimum amount of time taken by an algorithm for its execution.
- So, Big Omega notation is useful to calculate best case time complexity.

Definition: - Let f(n) , g(n) be 2 non negative functions. Then f(n)= Ω (g(n)), if there are 2 positive constants c, n_0 such that f(n)>=c*g(n) ∀ n>=n_0.

Graphical representation of Big Omega notation(Ω)



f(n)= Ω (g(n))

Ex:- f(n)=3n+2
g(n)=n;
To show f(n)= Ω(g(n))
f(n)>=c*g(n) , c>0, n_0>=1
3n+2>=c*n

Let C=1, n_0=1 then 5>=1 correct
n_0=2 then 8>=2 correct
n_0=3 then 11>=3 correct

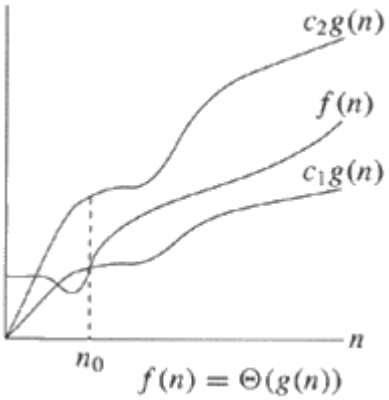
So, n_0>=1
Hence, 3n+2>=1*n ∀ n>=1

3. Theta notation (θ):-

- It is a method of representing an algorithm's running time between upper bound and lower bound.
- Using Theta notation we can calculate average amount of time taken by an algorithm for its execution.
- So, Theta notation is useful to calculate average case time complexity.

Definition: - Let f(n) , g(n) be 2 non negative functions. Then f(n)= θ (g(n)), if there are 3 positive constants c_1, c_2,n_0 such that c_1*g(n)<=f(n)<=c_2*g(n) ∀ n>=n_0.

Graphical representation of Theta notation(O):-



Ex:- f(n)=3n+2
g(n)=n;
To show f(n)= θ(g(n))
c_1*g(n)<=f(n)<=c_2*g(n), c_1>0,
c_2>0
n_0>=1
c_1*n<=3n+2<= c_2*n
c_2=4 , c_1=1,
Let n_0=1 then 1<=5<=4 wrong
Let n_0=2 then 2<=8<=8 correct
Let n_0=3 then 3<=11<=12 correct
Let n_0=4 then 4<=14<=16 correct
So, n_0>=2
Hence, 1*n<=3n+2<=*4n n>=2

4. Little oh notation(o):-

Let f(n), g(n) be 2 non negative functions then f(n)=o(g(n)) such that

5. Little Omega notation(ω):-

Let f(n), g(n) be 2 non negative functions then f(n)=o(g(n)) such that

Various types of computing times or typical growth rates:-

O(1) → constant computing time
O(n) → linear computing time
$O(n^2)$ →Quadratic computing time
$O(n^3)$ →Cubic computing time
$O(2^n)$ →Exponential computing time
O(log n) → Logarithmic computing time
O(n log n) →Logarithmic computing time

N	n^2	n^3	2^n	logn	nlogn
1	1	1	2	0	0
2	4	8	4	1	2
4	16	64	16	2	8
8	64	512	256	3	24

The relation among various computing times is,
$O(\log n)<O(n)<O(n\log n)<O(n^2)<O(2^n)<O(n^3)$

Space complexity:- The amount of space required by an algorithm for its execution.
Space complexity S(P) can be calculated as $S(P)=C+S_P$
Where, S(P) is space complexity of a program or problem
C means constant or fixed part
S_P means variable part
Constant part denotes memory (space) required by variables.
Variable part depends on problem instance.

Definitions of a Data Structure:-

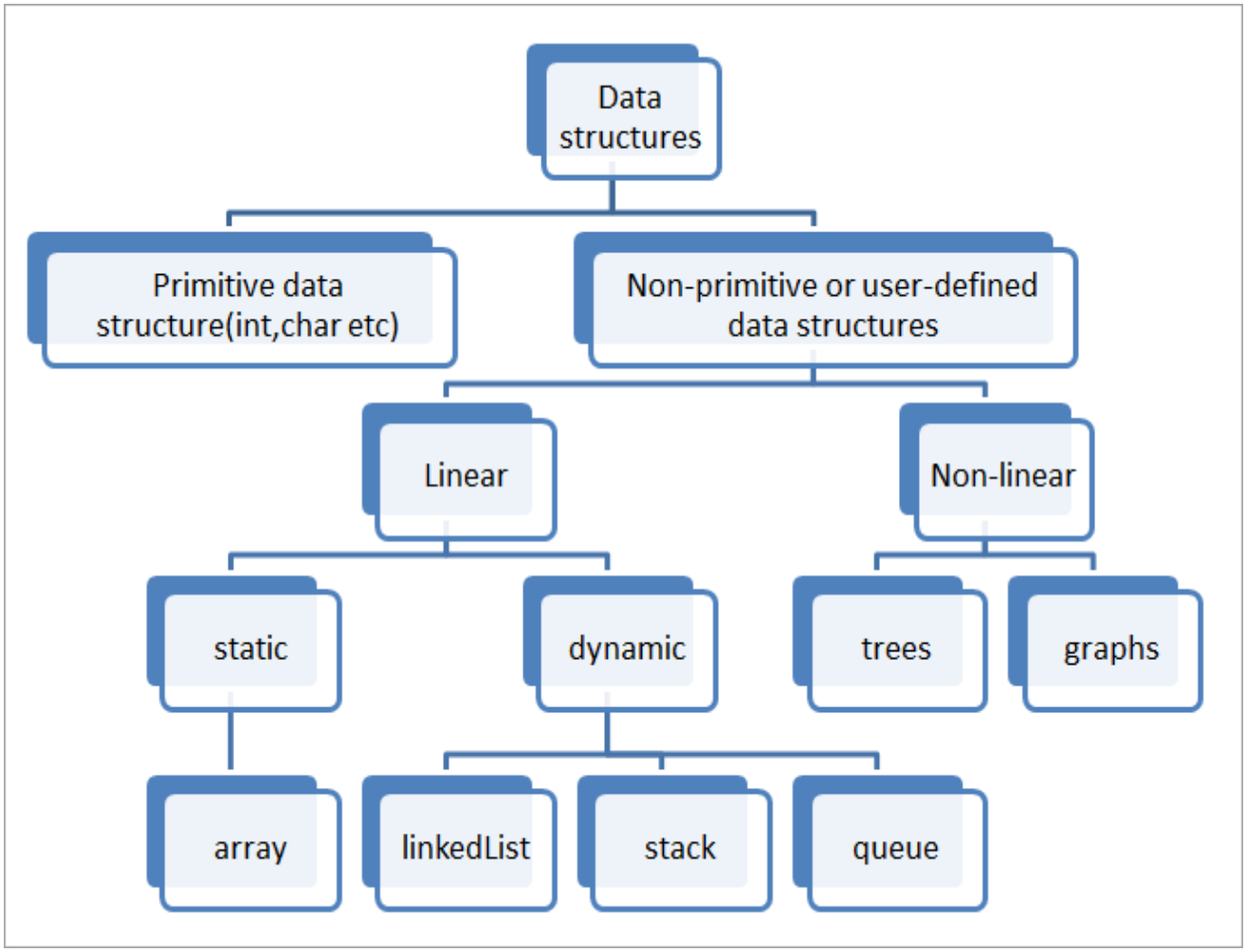
Data Structure is a way of organizing data in a computer so that we can perform operations on these data in an effective way.

Data structure is a data organization, management & storage format that enables efficient access & modification.

A Data structure is a particular way of storing & organizing data in a computer memory so that it can be used efficiently.

To develop a program we should select an appropriate data structure.

Classification of Data Structures:



Types of Data Structures:- Data Structures are mainly classified into 2 types.

1. Linear Data Structures
2. Non Linear Data Structures

1.Linear Data Structures:- In linear data structures all the elements are organized in linear (sequential) fashion.

Eg:- Arrays, stacks, queues and linked list.

2.Non Linear Data Structures:- In Non linear data structures all the elements are organized in non linear (random) fashion.

Eg:- Trees, Graphs.

Abstract Data Types:-

An abstract data type (ADT) is a set of operations. An ADT specifies what is to be done but how it is done is not specified i.e. all the implementation details are hidden.

The basic idea is that the implementation of these operations is written once in the program and any other part of the program that needs to perform an operation on the ADT can do so by calling the appropriate function.

For Example, stack ADT have operations like push, pop but stack ADT doesn't provide any implementation details regarding operations.

List ADT:- List is basically a collection of elements. For example list is of the following form

$A_1, A_2, A_3, - - - - -, A_n$ where A_1 is the first element of the list

A_n is the last element of the list.

The size of the list is 'n'. If the list size is zero (0) then the corresponding list is called as empty list.

Implementation of list:- List is implemented in 2 ways.

1. Implementation of list using arrays
2. Implementation of list using Linked list (or) Implementation of list using pointers

1.Implementation of list using arrays:- An array can be defined as a collection of similar(homogeneous) data type elements and all the elements will be stored in contiguous (Adjacent) memory locations.

Array Operations:- We can perform mainly the following operations on arrays.

1. Create

2. Display (or) Traversing (or) Printing

3. Insertion

4. Deletion

5. Updation

6. Searching

7. Sorting

8. Merging

C program to implement various operations on list using Arrays:-

```
#include<stdio.h>
#include<conio.h>
int n, a[20];
void create();
int insert(int,int);
void display();
void find(int);
int  del(int);
void update(int,int);
void count();
int main()
{
        int choi,ele,pos,val;
        clrscr();
        printf("enter no ele in list \n");
        scanf("%d",&n);
        while(1)
        {
                clrscr();
                printf("\t\t List Adt using arrays \n");
                printf("1.create \n");
                printf("2.insert \n");
                printf("3.display \n");
                printf("4.find \n");
                printf("5.delete \n");
                printf("6.update\n");
                printf("7.count\n");
                printf("8.exit \n");
```

```
            printf("enter choice \n");
            scanf("%d",&choi);
            switch(choi)
            {
                case 1:create();
                    break;
                case 2:printf("enter at what position u want to enter\n");
                    scanf("%d",&pos);
                    printf("enter at value u want to insert\n");
                    scanf("%d",&val);
                    n=insert(pos,val);
                    break;
                case 3:display();
                    break;
                case 4:printf("enter ele to be searched\n");
                    scanf("%d",&val);
                    find(val);
                    break;

                case 5:printf("enter at what position u want to enter\n");
                    scanf("%d",&pos);
                    n=del(pos);
                    break;
                case 6:printf("enter pos to upadte\n");
                    scanf("%d",&pos);
                    printf("enter value to upadate\n");
                    scanf("%d",&val);
                    update(pos,val);
                    break;
                case 7:count();
                    break;

                case 8:exit(0);
                    break;
        }
    }

}
void create()
{
        int i;
        for(i=0;i<n;i++)
```

```
	{
		printf("enter ele of list\n");
		scanf("%d",&a[i]);
	}
}
int  insert(int pos,int val)
{
	int i;
	for(i=n-1;i>=pos;i--)
		a[i+1]=a[i];
	a[pos]=val;
	return n+1;
}
void display()
{
	int i;
	for(i=0;i<n;i++)
	printf("\nele of list are =%d",a[i]);
	getch();
}
int del(int pos)
{
	int i;
	for(i=pos;i<n-1;i++)
	{
		a[i]=a[i+1];
	}
	return n-1;
}
void find(int val)
{
	int i,flag=0;
	for(i=0;i<n;i++)
	{
		if(val==a[i])
		{
			flag=1;
			break;
		}
	}
	if(flag==1)
	{
```

```
            printf("ele found in list at=%d",i);
            getch();
        }
        else
            printf("ele not found");

}
void update(int pos,int val)
{
        a[pos]=val;
}
void count()
{
        printf("no of ele in the list =%d",n);
        getch();
}
```

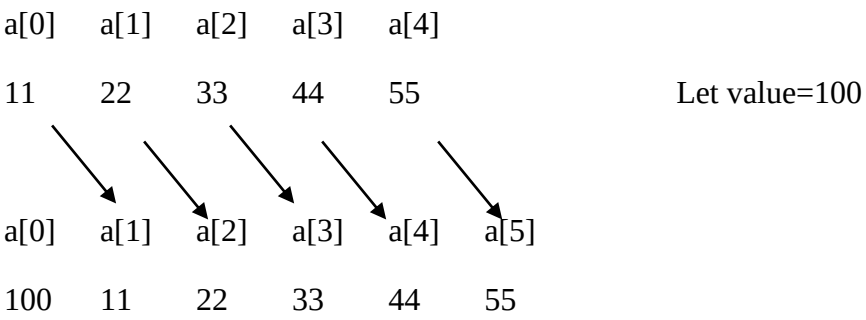
Advantages of Arrays:-

1. Array conations a collection of similar data type elements.
2. Array allows random access of elements i.e. an array element can be randomly accessed using index value. Arrays are simple and easy to implement.
3. Arrays are suitable when the number of elements are already known.
4. Arrays can be used to implement data structures like stacks, queues and so on.

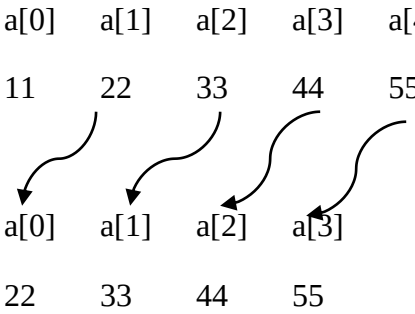
Drawbacks of Arrays:-

1. We must know in advance regarding how many elements are to be stored in array.
2. Arrays uses static memory allocation i.e. memory will be allocated at compilation time. So it's not possible to change size of the array at run time.
3. Since array size is fixed, if we allocate more memory than required then memory space will be wasted.
4. If we allocate less memory than required it will creates a problem.
5. The main drawback of arrays is insertion and deletion operations are expensive.

6. For example to insert an element into the array at beginning position, we need to shift all array elements one position to the right in order to store a new element at beginning position.



7. For example to delete an element from the array at beginning position , we need to shift all array elements one position to its left.



Because of the above limitations simple arrays are not used to implement lists.

---

2. Implementation of list using Linked list (or) Implementation of list using pointers:-

Linked list is a collection of nodes which are not necessary to be in adjacent memory locations.

Each node contains 2 fields.

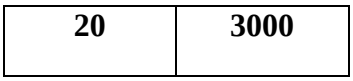
1. Data Field

2. Next field (or) pointer field (or) address field

Data field:- Data field contains values like 9, 6.8, ‘a’ , “ramu” , 9849984900

Next field:- It contains address of its next node. The last node next field contains NULL which indicates end of the linked list.

Diagrammatic representation of linked list:-



1000 2000 3000

Types of linked list:-

1. Single linked list (or) singly linked list

2. Circular linked list (or) circular single linked list

3. Double linked list

4. Circular double linked list

1. Single linked list (or) singly linked list:-

A single linked list is a collection of nodes which are not necessary to be in adjacent memory locations.

Each node contains 2 fields.

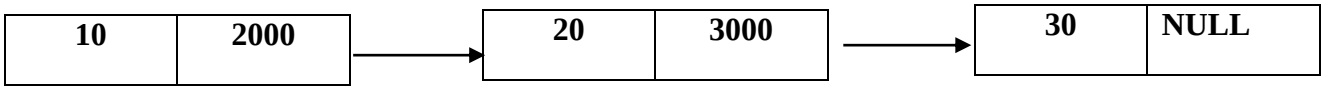
1. Data Field

2. Next field (or) pointer field (or) address field

Data field:- Data field contains values like 9, 6.8, 'a' , "ramu" , 9849984900

Next field:- It contains address of its next node. The last node next field contains NULL which indicates end of the linked list.

Diagrammatic representation of linked list:-



1000 2000 3000

It is called as single linked list because each node contains a single link which points to its next node.

Single Linked list operations:-

We can perform mainly the following operations on single linked list.

1. Creation

2. Display

3. Inserting an element into the list at begin position

4. Inserting an element into the list at end position

5. Inserting an element into the list at specified position

6. Deleting an element from the list at begin position

7. Deleting an element from the list at end position

8. Deleting an element from the list at specified position

9. Counting the number of elements or nodes

10. Sorting a list

11. Reversing a list

12. Merging of 2 lists

C program to implement various operations on list using Linked list:- (OR)

C program to implement various operations on list using Pointers:-

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
        int info;
        struct node *next;
};
struct node *start;
void insend(int ele);
void insbeg(int ele);
void insmiddle(int ele);
void delend();
void delbeg();
void delmiddle();
void traverse();
void main()
{
        int ele,choice;
        start=NULL;
        while(1)
        {
                clrscr();
                printf("\t\t Single List Operations \n");
                printf("1.insbeg\n");
                printf("2.insend\n");
                printf("3.insmiddle\n");
                printf("4.delbeg\n");
                printf("5.delend\n");
                printf("6.traverse\n");
                printf("7.delmiddle\n");
```

```
		printf("8.exit\n");
		printf("enter choice\n");
		scanf("%d",&choice);
		switch(choice)
		{
			case 1: printf("\n enetr ele ");
				scanf("%d",&ele);
				insbeg(ele);
				break;
			case 2:printf("\n enetr ele ");
				scanf("%d",&ele);
				insend(ele);
				break;
			case 3:printf("\n enetr ele ");
				scanf("%d",&ele);
				insmiddle(ele);
				break;
			case 4:delbeg();
				break;
			case 5:delend();
				break;
			case 6:traverse();
				break;
			case 7:delmiddle();
				break;
			case 8:exit(0);
		}
	}
}
void insbeg(int ele)
{
	struct node *temp;
	temp=(struct node*)malloc(sizeof(struct node));
	temp->info=ele;
	if(start==NULL)
		{
		temp->next=NULL;
		start=temp;
	}
	else if(start->next==NULL)
	{
		temp->next=start;
		start=temp;
	}
	else
	{
		temp->next=start;
		start=temp;
	}
```

```
}
void insmiddle(int ele)
{
	struct node *temp,*p;int pos,count=0;
	if(start==NULL)
	{
		temp=(struct node*)malloc(sizeof(struct node));
		temp->info=ele;
		temp->next=NULL;
		start=temp;
	}
	else
	{
		temp=(struct node*)malloc(sizeof(struct node));
		temp->info=ele;
		printf("\n enter position");
		scanf("%d",&pos);
		p=start;
		while(p!=NULL)
		{
			if(pos==count)
			break;
			count=count+1;
			p=p->next;
		}
		temp->next=p->next;
		p->next=temp;
	}

}
void insend(int ele)
{
	struct node *temp,*p;
	temp=(struct node*)malloc(sizeof(struct node));
	temp->info=ele;
	temp->next=NULL;
	if(start==NULL)
	{
		start=temp;
	}
	else if(start->next==NULL)
	{
		start->next=temp;
	}
	else
	{
		p=start;
		while(p->next!=NULL)
		{
```

```
                        p=p->next;
                }
                p->next=temp;
        }
}
void delbeg()
{
        struct node *temp;
        if(start==NULL)
                printf("\n list is empty");
        else if(start->next==NULL)
        {
                temp=start;
                start=NULL;
                free(temp);
        }
        else
        {
                temp=start;
                start=start->next;
                free(temp);
        }
}
void delend()
{
        struct node *temp,*p;
        if(start==NULL)
              printf("\n list is empty");
        else if(start->next==NULL)
        {
                temp=start;
                start=NULL;
                free(temp);
        }
        else
        {
                p=start;
                while(p->next->next!=NULL)
                {
                        p=p->next;
                }
                temp=p->next;
                p->next=NULL;
                free(temp);
        }
}
void traverse()
{
        struct node *p;
```

```
        if(start==NULL)
           printf("\n list is empty");
        else
        {
                p=start;
                printf("Elements of Single Linked List are\n\n");
                while(p!=NULL)
                {
                        printf("%d->",p->info);
                        p=p->next;
                }
        }
        getch();
}
void delmiddle()
{
}
```

Advantages of linked list:- Or **Advantages of Single linked list**

1. Linked list is a dynamic data structure i.e. memory will be allocated at run time. So, no wastage of memory.
2. It is not necessary to know in advance regarding the number of elements to be stored in the list.
3. Insertion and deletion operations are easier, as shifting of values is not necessary.
4. Different types of data can be stored in data field of a node.
5. The memory allocation need not be in contiguous locations.
6. Linear data structures such as stacks, queues are easily implemented using linked list.

Disadvantages or limitations of single linked list:-

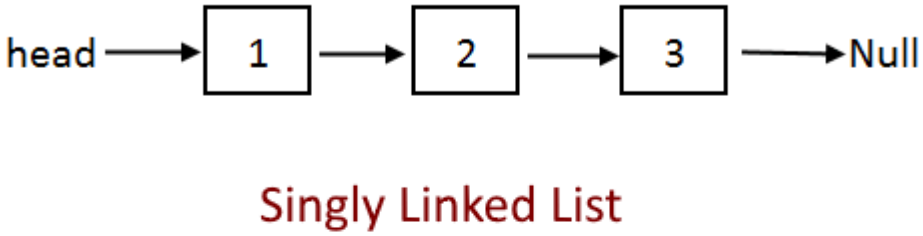
1. Random access of elements in not possible in single linked list i.e. we can't access a particular node directly.
2. Binary search algorithm can't be implemented on a single linked list.
3. There is no way to go back from one node to its previous node i.e. only forward traversal is possible.
4. Extra storage space for pointer is required.
5. Reversing single linked list is difficult.

Circular linked list:-

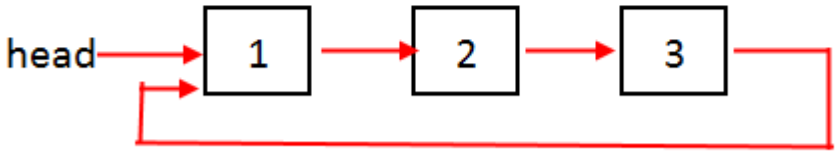
Circular linked list is a linear data structure which contains a collection of nodes where each node contains 2 fields.

1. **Data field:-** It contains a value which may be an integer, float, char and string.
2. **Next field:-** It contains address of its next node. The last node next field contains address of first node.

Diagrammatic representation of circular linked list:-



Singly Linked List



Circular Linked List

Circular Linked list operations:- or **Circular Linked list ADT**

We can perform mainly the following operations on circular linked list.

1. Creation
2. Display
3. Inserting an element into the list at begin position
4. Inserting an element into the list at end position
5. Inserting an element into the list at specified position
6. Deleting an element from the list at begin position
7. Deleting an element from the list at end position
8. Deleting an element from the list at specified position
9. Counting the number of elements or nodes

C program to implement circular linked list:-

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

```
#include<stdlib.h>
struct node
{
        int info;
        struct node *next;
};
struct node *start;
void insend(int ele);
void insbeg(int ele);
void insmiddle(int ele);
void delend();
void delbeg();
void delmiddle();
void traverse();
void main()
{
        int ele,choice;
        start=NULL;
        while(1)
        {
                clrscr();
                printf("\t\t Circular Linked List operations\n");
                printf("1.insbeg\n");
                printf("2.insend\n");
                printf("3.insmiddle\n");
                printf("4.delbeg\n");
                printf("5.delend\n");
                printf("6.traverse\n");
                printf("7.delmiddle\n");
                printf("8.exit\n");
                printf("enter choice\n");
                scanf("%d",&choice);
                switch(choice)
                {
                        case 1:printf("\n enter ele ");
                                scanf("%d",&ele);
                                insbeg(ele);
                                break;
                        case 2:printf("\n enetr ele ");
                                scanf("%d",&ele);
                                insend(ele);
                                break;
                        case 3:printf("\n enetr ele ");
                                scanf("%d",&ele);
                                insmiddle(ele);
                                break;
                        case 4:delbeg();
                                break;
                        case 5:delend();
```

```
                        break;
                case 6:traverse();
                        break;
                case 7:delmiddle();
                        break;
                case 8:exit(0);
                }
        }
}
void insbeg(int ele)
{
        struct node *temp,*p;
        temp=(struct node*)malloc(sizeof(struct node));
        temp->info=ele;
        if(start==NULL)
        {
                start=temp;
                start->next=start;
        }
        else if(start->next==start)
        {
                temp->next=start;
                start->next=temp;
                start=temp;
        }
        else
        {
                p=start;
                while(p->next!=start)
                {
                        p=p->next;
                }
                p->next=temp;
                temp->next=start;
                start=temp;
        }
}
void insmiddle(int ele)
{
        struct node *temp,*p;int pos,count=0;
        if(start==NULL)
        {
                temp=(struct node*)malloc(sizeof(struct node));
                temp->info=ele;
                start=temp;
                temp->next=start;
        }
        else
        {
```

```
                temp=(struct node*)malloc(sizeof(struct node));
                temp->info=ele;
                printf("\n enter position");
                scanf("%d",&pos);
                p=start;
                do
                {
                        count++;
                        if(pos==count)
                        break;
                        p=p->next;

                }while(p!=start);
                temp->next=p->next;
                p->next=temp;
        }
}
void insend(int ele)
{
        struct node *temp,*p;
        temp=(struct node*)malloc(sizeof(struct node));
        temp->info=ele;
        if(start==NULL)
        {
                start=temp;
                temp->next=start;
        }
        else if(start->next==start)
        {
                start->next=temp;
                temp->next=start;
        }
        else
        {
                p=start;
                while(p->next!=start)
                {
                        p=p->next;
                }
                p->next=temp;
                temp->next=start;
        }
}
void delbeg()
{
        struct node *temp,*p;
        if(start==NULL)
        printf("\n list is empty");
        else if(start->next==start)
```

```
	{
		temp=start;
		start=NULL;
		free(temp);
	}
	else
	{
		p=start;
		temp=start;
		while(p->next!=start)
		{
			p=p->next;
		}
		start=start->next;
		p->next=start;
		free(temp);
	}
}
void delend()
{
	struct node *temp,*p;
	if(start==NULL)
	printf("\n list is empty");
	else if(start->next==start)
	{
		temp=start;
		start=NULL;
		free(temp);
	}
	else
	{
		p=start;
		while(p->next->next!=start)
		{
			p=p->next;
		}
		temp=p->next;
		p->next=start;
		free(temp);
	}
}
void traverse()
{
	struct node *p;
	if(start==NULL)
	printf("\n list is empty");
	else
	{
		p=start;
```

```
                do
                {
                        printf("\n elements are %d",p->info);
                        p=p->next;
                }while(p!=start);
        }
        getch();
}
void delmiddle()
{
        struct node *temp,*p;
        int pos,count=0;
        if(start==NULL)
        {
                printf("\n list is empty");
        }
        else if(start->next==start)
        {
                temp=start;
                start=NULL;
                free(temp);
        }
        else if(start->next->next==start)
        {
                temp=start->next;
                free(temp);
                start->next=start;
        }
        else
        {
                printf("\n enter pos");
                scanf("%d",&pos);
                p=start;
                while(p->next!=start)
                {
                        count=count+1;
                        if(count==pos)
                                break;
                        temp=p;
                        p=p->next;
                }
                temp->next=p->next;
                free(p);
        }
}
```

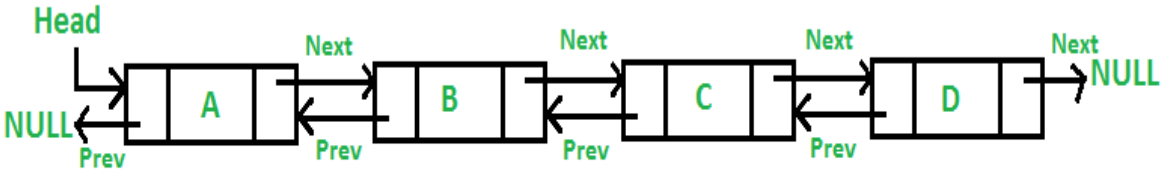
Advantages of circular linked list:- It saves time when we have to go to the first node from the last node. But in double linked list we have to go through in between nodes.

Limitations of circular linked list:- It is not easy to reverse circular linked list.

Double linked list:-

Double linked list is a linear data structure which contains a collection of nodes, where each node contains 3 fields.

1. **prev field**:- It is a pointer field which contains the address of its previous node. The first node prev field contains NULL value.
2. **data field** :- It contains a value which may be an integer, float, char and string.
3. **next field**:-It is a pointer field which contains the address of its next node. The last node next field contains NULL value.



In a single linked list only forward traversal is possible i.e. we can traverse from left to right only where as in a double linked list both forward traversal(Left to right) as well as backward traversal(Right to left) is possible.

C program to implement double linked list:-

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
        struct node *prev;
        int info;
        struct node *next;
};
struct node *start;
void insend(int ele);
void insbeg(int ele);
void insmiddle(int ele);
void delend();
void delbeg();
void delmiddle();
void traverse();
void disbackwards();
void main()
{
```

```
	int ele,choice;
	start=NULL;
	while(1)
	{
		clrscr();
		printf("\t\t Double List operations\n");
		printf("1.insbeg\n");
		printf("2.insend\n");
		printf("3.insmiddle\n");
		printf("4.delbeg\n");
		printf("5.delend\n");
		printf("6.traverse\n");
		printf("7.delmiddle\n");
		printf("8.displaybackwards\n");
		printf("9.exit\n");
		printf("enter choice\n");
		scanf("%d",&choice);
		switch(choice)
		{
			case 1:printf("\n enetr ele ");
				scanf("%d",&ele);
				insbeg(ele);
				break;
			case 2:printf("\n enetr ele ");
				scanf("%d",&ele);
				insend(ele);
				break;
			case 3:printf("\n enetr ele ");
				scanf("%d",&ele);
				insmiddle(ele);
				break;
			case 4:delbeg();
				break;
			case 5:delend();
				break;
			case 6:traverse();
				break;
			case 7:delmiddle();
				break;
			case 8:disbackwards();
				break;

			case 9:exit(0);
		}
	}
}
void insbeg(int ele)
{
	struct node *temp;
```

```
	temp=(struct node*)malloc(sizeof(struct node));
	temp->info=ele;
	temp->prev=NULL;
	if(start==NULL)
	{
		temp->next=NULL;
		start=temp;
	}
	else if(start->next==NULL)
	{
		temp->next=start;
		start->prev=temp;
		start=temp;
	}
	else
	{
		temp->next=start;
		start->prev=temp;
		start=temp;
	}
}
void insmiddle(int ele)
{
	struct node *temp,*p;int pos,count=0;
	if(start==NULL)
	{
		temp=(struct node*)malloc(sizeof(struct node));
		temp->info=ele;
		temp->prev=NULL;
		temp->next=NULL;
		start=temp;
	}
	else
	{
		temp=(struct node*)malloc(sizeof(struct node));
		temp->info=ele;
		printf("\n enter position");
		scanf("%d",&pos);
		p=start;
		while(p!=NULL)
		{
			count++;
			if(pos==count)
			break;
			p=p->next;

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

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

}
void insend(int ele)
{
        struct node *temp,*p;
        temp=(struct node*)malloc(sizeof(struct node));
        temp->info=ele;
        temp->next=NULL;
        if(start==NULL)
        {
                start=temp;
                temp->prev=NULL;
        }
        else if(start->next==NULL)
        {
                temp->prev=start;
                start->next=temp;
        }
        else
        {
                p=start;
                while(p->next!=NULL)
                {
                        p=p->next;
                }
                temp->prev=p;
                p->next=temp;
        }
}
void delbeg()
{
        struct node *temp;
        if(start==NULL)
        printf("\n list is empty");
        else if(start->next==NULL)
        {
                temp=start;
                start=NULL;
                free(temp);
        }
        else
        {
                temp=start;
                start=start->next;
                start->prev=NULL;;
                free(temp);
```

```
        }
}
void delend()
{
        struct node *temp,*p;
        if(start==NULL)
        printf("\n list is empty");
        else if(start->next==NULL)
        {
                temp=start;
                start=NULL;
                free(temp);
        }
        else
        {
                p=start;
                while(p->next->next!=NULL)
                {
                        p=p->next;
                }
                temp=p->next;
                p->next=NULL;
                free(temp);
        }
}
void traverse()
{
        struct node *p;
        if(start==NULL)
        printf("\n list is empty");
        else
        {
                p=start;
                while(p!=NULL)
                {
                        printf("  %d->",p->info);
                        p=p->next;
                }
        }
        getch();
}
void disbackwards()
{
        struct node *p;
        if(start==NULL)
        printf("\n list is empty");
        else
        {
                p=start;
```

```
                while(p->next!=NULL)
                {
                        p=p->next;
                }
                while(p !=NULL)
                {
                        printf(" %d-->",p->info);
                        p=p->prev;
                }
        }
        getch();
}
void delmiddle()
{
        struct node *temp,*p;
        int pos,count=0;
        if(start==NULL)
        {
                printf("\n list is empty");
        }
        else if(start->next==NULL)
        {
                temp=start;
                start=NULL;
                free(temp);
        }
        else if(start->next->next==NULL)
        {
                temp=start->next;
                free(temp);
                start->next=NULL;
        }
        else
        {
                printf("\n enter pos");
                scanf("%d",&pos);
                p=start;
                while(p!=NULL)
                {
                        count=count+1;
                        if(count==pos)
                                break;
                        p=p->next;
                }
                temp=p->prev;
                temp->next=p->next;
                free(p);
        }
}
```

Advantages of double linked list:-

1. We can traverse in both directions i.e. from left to right as well as from right to left.
2. It is easy to reverse a double linked list.

Limitations of double linked list:-

1. It requires more memory because one extra field (prev) is required.
2. Insertion and deletion operations takes more time because more operations are required.

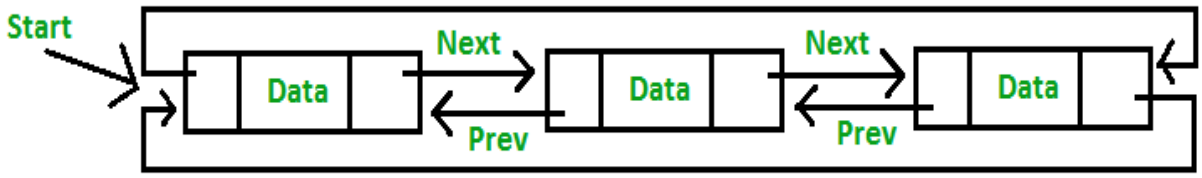
Circular double linked list:-

It is a linear data structure which is a combination of circular single linked list and double linked list.

Circular double linked list is a collection of nodes, where each node contains 3 fields.

1. **prev field**:- It is a pointer field which contains the address of its previous node. The first node prev field contains address of last node.
2. **data field** :- It contains a value which may be an integer, float, char and string.
3. **next field**:-It is a pointer field which contains the address of its next node. The last node next field contains address of first node.

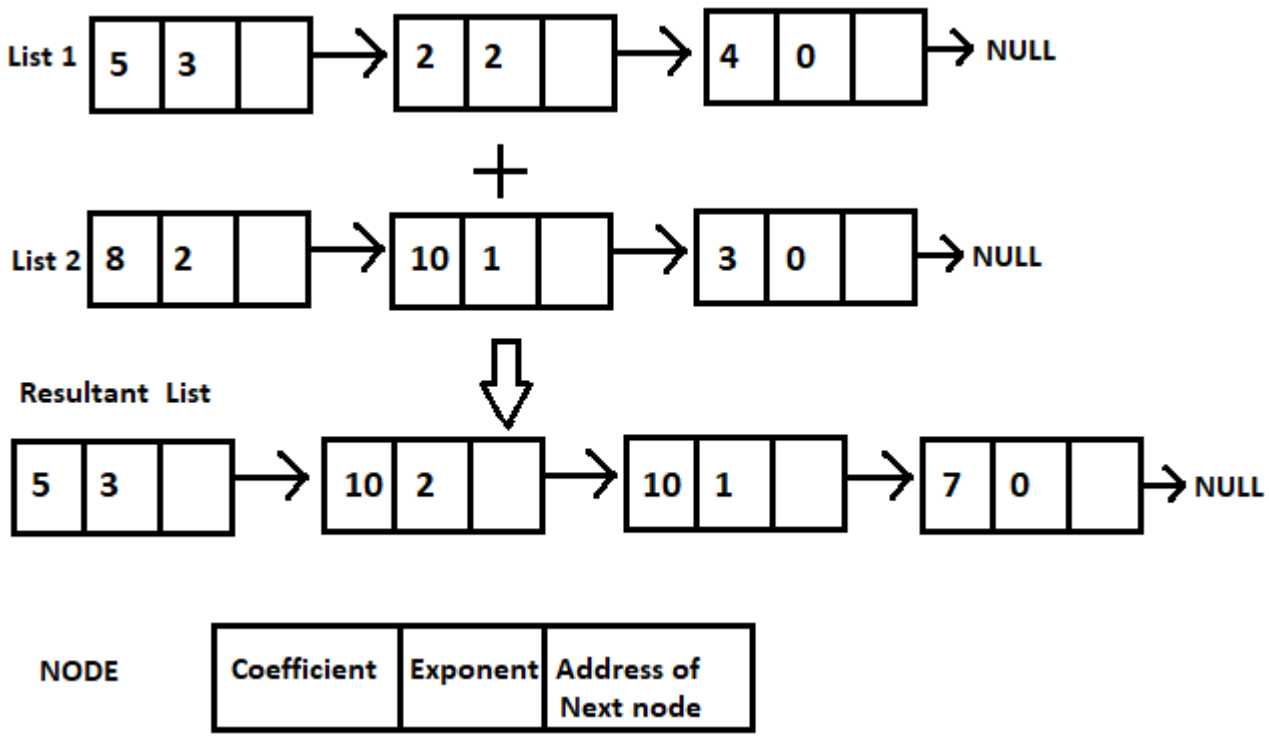
Diagrammatic representation of circular double linked list:-



Linked list applications:-

1.Polynomials can be represented and various operations can be performed on polynomials using linked lists.
2.Sparse matrix can be represented using linked list.
3.Various details like student details, customer details , product details and so on can be implemented using linked list.

Polynomial ADT:- Polynomial is a collection of terms where each term contains coefficient and exponent.



Ex:- $8x^6+5x^4+2x+9$
Where 8,5,2 and 9 are coefficients while 6,4,1 and 0 are exponents.
We can perform various operations on polynomial such as addition, subtraction, multiplication and division.

C program to implement polynomial ADT:-

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
struct node
{
        int coeff, expo;
        struct node *next;
};
struct node *s1=NULL,*s2=NULL,*s3=NULL;
void createp1();
void createp2();
void traversep1();
void traversep2();
void polyadd();
void createp3(int,int);
void traversep3();
int main()
{
        clrscr();
        int choice,ele;
        while(1)
        {
```

```
                clrscr();
                printf("\t\t       Polynomial Adt Operations \n");
                printf("1.create poly1 \n");
                printf("2.create poly2 \n");
                printf("3.traverselist1\n");
                printf("4.traverselist2\n");
                printf("5.polynomial add\n");
                printf("6.treaverse list \n");
                printf("7.exit\n");
                printf("enter choice\n");
                scanf("%d",&choice);
                switch(choice)
                {
                        case 1:createp1();
                                break;
                        case 2:createp2();
                                break;
                        case 3:traversep1();
                                getch();
                                break;
                        case 4:traversep2();
                                getch();
                                break;
                        case 5:polyadd();
                                break;
                        case 6:traversep3();
                                getch();
                                break;
                        case 7:exit(0);
                }
        }
}
void createp1()
{
        struct node *temp,*p;
        char ch;
        do
        {
                temp=(struct node*)malloc(sizeof(struct node));
                printf("enter coeff & expo values of a first polynomial term");
                scanf("%d%d",&temp->coeff,&temp->expo);
                temp->next=NULL;
                if(s1==NULL)
                        s1=temp;
                else if(s1->next==NULL)
                        s1->next=temp;
                else
                {
                        p=s1;
```

```
                        while(p->next!=NULL)
                        p=p->next;
                        p->next=temp;
                }
                printf("do you want another term(y/n)\n");
                fflush(stdin);
                scanf("%c",&ch);
        }while(ch=='y');
}
void createp2()
{
        struct node *temp,*p;
        char ch;
        do
        {
                temp=(struct node*)malloc(sizeof(struct node));
                printf("enter coeff & expo values of a second polynomial term\n");
                scanf("%d%d",&temp->coeff,&temp->expo);
                temp->next=NULL;
                if(s2==NULL)
                        s2=temp;
                else if(s2->next==NULL)
                        s2->next=temp;
                else
                {
                        p=s2;
                        while(p->next!=NULL)
                        p=p->next;
                        p->next=temp;
                }
                printf("do you want another term(y/n)\n");
                fflush(stdin);
                scanf("%c",&ch);
        }while(ch=='y');
}
void traversep1()
{
        struct node *p;
        if(s1==NULL)
        printf("s1 list is empty\n");
        else
        {
                p=s1;
                while(p!=NULL)
                {
                        if(p->coeff>0)
                                printf("+%dx^%d",p->coeff,p->expo);
                        else
                                printf("%dx^%d",p->coeff,p->expo);
```

```
                        p=p->next;
                }
        }
}
void traversep2()
{
        struct node *p;
        if(s2==NULL)
                printf("s1 list is empty\n");
        else
        {
                p=s2;
                while(p!=NULL)
                {
                        if(p->coeff>0)
                                printf("+%dx^%d",p->coeff,p->expo);
                        else
                                printf("%dx^%d",p->coeff,p->expo);
                        p=p->next;
                }
}
}
void polyadd()
{
        struct node *p1,*p2;
        int coef_sum;
        p1=s1;
        p2=s2;
        while(p1!=NULL & p2!=NULL)
        {
                if(p1->expo==p2->expo)
                {
                        coef_sum=p1->coeff+p2->coeff;
                        createp3(coef_sum,p1->expo);
                        p1=p1->next;
                        p2=p2->next;
                }
                else if(p1->expo>p2->expo)
                {
                        createp3(p1->coeff,p1->expo);
                        p1=p1->next;
                }
                else
                {
                        createp3(p2->coeff,p2->expo);
                        p2=p2->next;
                }
        }
        if(p1==NULL)
```

```
	{
		while(p2!=NULL)
		{
			createp3(p2->coeff,p2->expo);
			p2=p2->next;
		}
	}
	else if(p2==NULL)
	{
		while(p1!=NULL)
		{
			createp3(p1->coeff,p1->expo);
			p1=p1->next;
		}
	}
}
void traversep3()
{
	struct node *p;
	if(s3==NULL)
		printf("s1 list is empty\n");
	else
	{
		p=s3;
		while(p!=NULL)
		{
			if(p->coeff>0)
				printf("+%dx^%d",p->coeff,p->expo);
			else
				printf("%dx^%d",p->coeff,p->expo);
			p=p->next;
		}
	}
}
void createp3(int c,int e)
{
	struct node *temp,*p;
	temp=(struct node*)malloc(sizeof(struct node));
	temp->coeff=c;
	temp->expo=e;
	temp->next=NULL;
	if(s3==NULL)
		s3=temp;
	else if(s3->next==NULL)
		s3->next=temp;
	else
	{
		p=s3;
		while(p->next!=NULL)
```

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