# Experiment – 1

A) **Write a program in C to display the n terms of even natural number and their sum.**

## Aim:

A program in C to display the n terms of even natural number and their sum

## Description:

Any number that can be exactly divided by 2 is called as an even number. Even numbers always end up with the last digit as 0, 2, 4, 6 or 8. Some examples of even numbers are 2, 4, 6, 8, 10, 12, 14, 16. These are even numbers as these numbers can easily be divided by 2.

## Algorithm:

**Step – 1:** start

**Step – 2:** initialize n

**Step – 3:** check the remainder, when n is divided by 2

**Step – 4:** if the remainder is 0, then n is even number

**Step -5:** if the remainder is 1, then n is odd number

**Step – 6:** stop

## Program:

```
/*A program in C to display the n terms of even natural number and their sum*/
#include <stdio.h>
#include<conio.h>
void main()
{
int n;
printf("Enter a number=");
scanf("%d",&n);
if(n%2==0)
printf("Number is even");
else
printf("Number is odd");
}
```

**Output:**

```
Enter a number=8
Number is even
```

**B) Write a program in C to display the n terms of harmonic series and their sum. 1 + 1/2 + 1/3 + 1/4 + 1/5 ... 1/n terms**

## Aim:

A program in C to display the n terms of harmonic series and their sum. 1 + 1/2 + 1/3 + 1/4 + 1/5 ... 1/n terms

## Description:

Harmonic series is a sequence of terms formed by taking the reciprocals of an arithmetic progression.

## Algorithm:

**Step – 1:** start

**Step – 2:** initialize i, n

**Step – 3:** check i is less than or equal to n and increment i value

**Step – 4:** do sum=sum + 1.0/i, where sum is the harmonic sum

**Step – 5:** stop

## Program:

```
/*a program in C to display the n terms of harmonic series and their sum. 1 + 1/2 + 1/3 + 1/4
+ 1/5 ... 1/n terms*/

#include<stdio.h>

#include<conio.h>

int main()

{

int n;

printf("Enter the value of n:");

scanf("%d",&n);

float sum=0;

int i;

for(i=1;i<=n;i++)

sum=sum + 1.0/i;

printf("%f is the harmonic sum \n",sum);

}
```

**Output:**

```
Enter the value of n: 5

2.283334 is the harmonic sum
```

C) **Write a C program to check whether a given number is an Armstrong number or not**

## Aim:

A C program to check whether a given number is an Armstrong number or not.

## Description:

Armstrong number is a number that is equal to the sum of cubes of its digits. For example 0, 1, 153, 370, 371 and 407 are the Armstrong numbers.

## Algorithm:

```
Step 1: Start
Step 2: Declare Variable sum, temp, num
Step 3: Read num from User
Step 4: Initialize Variable sum=0 and temp=num
Step 5: Repeat Until num>=0
        5.1 sum=sum + cube of last digit i.e [(num%10)*(num%10)*(num%10)]
        5.2 num=num/10
Step 6: IF sum==temp
            Print "Armstrong Number"
            ELSE
            Print "Not Armstrong Number"
Step 7: Stop
```

**Program:**

```
#include<stdio.h>

#include<conio.h>

void main()

{

int n,r,num=0,res;

printf("Enter the number=");

scanf("%d",&n);

res=n;

while(n!=0)

{

r=n%10;

num+=(r*r*r);

n=n/10;

}

if(res==num)

printf("This is Armstrong  number ");

else

printf("This is not a Armstrong number");

}
```

**Output:**

```
Enter a number = 153

This is Armstrong number
```

**D) Write a C program to calculate the factorial of a given number**

## Aim:

A c program to find factorial of a number

## Description:

The factorial of a number is the function that multiplies the number by every natural number below it. Symbolically, factorial can be represented as "!". So, *n* factorial is the product of the first *n* natural numbers and is represented as n!

## Algorithm:

```
Step 1: Start
Step 2: Declare Variable n, fact, i
Step 3: Read number from User
Step 4: Initialize Variable fact=1 and i=1
Step 5: Repeat Until i<=number
    5.1 fact=fact*i
    5.2 i=i+1
Step 6: Print fact
Step 7: Stop
```

## Program:

```
#include <stdio.h>

#include<conio.h>

void main()

{

int i, n, fact=1,num;

printf("Enter the value of num");

scanf("%d",&num);

if(num<0)

printf("Error");

else

printf("Enter a number to calculate its factorial\n");

scanf("%d", &n);

for (i = 1; i <= n; i++)

fact = fact * i;

printf("Factorial of %d = %d\n", n, fact);

}
```

**Output:**

```
Enter the value of num = 4

Enter a number to calculate its factorial = 4

Factorial of 4 = 24
```

## Experiment 2:

**A) Write a program in C for multiplication of two square Matrices**

### Aim:

A c program for multiplication of square matrices

### Description:

In linear algebra, matrices play an important role in dealing with different concepts. A matrix is a rectangular *array* or *table* of numbers, symbols, or expressions, arranged in *rows* and *columns* in mathematics. We can perform various operations on matrices such as addition, subtraction, multiplication and so on.

### Algorithm:

```
Step 1: Start
Step 2: Declare matrix A[m][n]
        and matrix B[p][q]
        and matrix C[m][q]
Step 3: Read m, n, p, q.
Step 4: Now check if the matrix can be multiplied or not, if n is not equal to q matrix can't be
        multiplied and an error message is generated.
Step 5: Read A[][] and B[][]
Step 6: Declare variable i=0, k=0 , j=0 and sum=0
Step 7: Repeat Step until i < m
    7.1: Repeat Step until j < q
   7.1.1: Repeat Step until k < p
           Set sum= sum + A[i][k] * B[k][j]
           Set multiply[i][j] = sum;
           Set sum = 0 and k=k+1
    7.1.2: Set j=j+1
      7.2: Set i=i+1
Step 8: C is the required matrix.
Step 9: Stop
```

## Program:

```
//A c program to find multiplication of two square matrices
#include<stdio.h>
void main()
{
int a[10][10],b[10][10],mul[10][10],m=3,n=3,i,j,k;
//matrixA
printf("Enter elements in matix A:\n");
for(i=0;i<m;i++)
{
for(j=0;j<n;j++)
{
scanf("%d",&a[i][j]);
}
}
//matrixB
printf("Enter elements in matix B:\n");
for(i=0;i<m;i++)
{
for(j=0;j<n;j++)
{
scanf("%d",&b[i][j]);

}
}
//multiplication of both matrices
for(i=0; i<m; i++)
{
for(j=0; j<n; j++)
{

mul[i][j]=0;
for(k=0; k<m; k++)
{
mul[i][j] += a[i][k] * b[k][j];
}

}
}
//product of two matrices
printf("\n Multiplication result of the two given Matrix is:=\n");
for(i=0;i<m;i++)
{
for(j=0;j<n;j++)
{
printf("%d\t",mul[i][j]);
}
printf("\n");
}
}
```

Output:

```
Enter elements in matrix A:
1 2 3
```

```
4 5 6

7 8 9

Enter elements in matrix B:

9 8 7

6 5 4

3 2 1

Multiplication result of the two given matrix is =

30 24 18

84 69 54

138 114 90
```

**B) Write a program in C to find transpose of a given matrix.**

## Aim:

A c program to find transpose of a matrix

## Description:

The new matrix obtained by interchanging the rows and columns of the original matrix is called as the transpose of the matrix. If A = $[a_{ij}]$ be an m × n matrix, then the matrix obtained by interchanging the rows and columns of A would be the transpose of A. of It is denoted by A′or ($A^T$). In other words, if A = $[a_{ij}]_{mxn}$ ,thenA′ = $[a_{ji}]_{nxm}$

## Algorithm:

```
Step 1: Start

Step 2: Declare matrix a[m][n] of order mxn

Step 3: Read matrix a[m][n] from User

Step 4: Declare matrix b[m][n] of order mxn

Step 5: // Transposing the Matrix

5.1: Declare variables i, j
5.2: Set i=0, j=0
5.3: Repeat until i < n
5.3.1: Repeat until j < m
5.3.1.1: b[i][j] = a[j][i]
5.3.1.2: j=j+1 // Increment j by 1
5.3.2: i=i+1 // Increment i by 1
5.4: Print matrix b
// The matrix b is the transpose of a and can be printed now
Step 6: Stop
```

**Program:**

```
//A c program for transpose of a matrix
#include<stdio.h>
void main()
{
int a[10][10],t[10][10],r=3,c=3,i,j;
printf("Enter elements in matix A:\n");
for(i=0;i<r;i++)
{
for(j=0;j<c;j++)
{
scanf("%d",&a[i][j]);
}
}
for(i=0;i<r;i++)
{
for(j=0;j<c;j++)
{
t[j][i]=a[i][j];
}
}
printf("Transpose of a matrix is:\n");
for (i=0; i<r; i++)
{
for (j=0; j < c; j++)
{
printf("%d\t", t[i][j]);
}
printf("\n");
}
}
```

**Output:**

```
Enter elements in a matrix A:
1 2 3
4 5 6
7 8 9
Transpose of a matrix is:
1 4 7
2 5 8
3 6 9
```

## Experiment 3:

**A) Write a program in C to check whether a number is a prime number or not using the function.**

### Aim:

A c program to check prime number without using function

### Description:

Prime number in C: Prime number is a number that is greater than 1 and divided by 1 or itself. In other words, prime numbers can't be divided by other numbers than itself or 1. For example 2, 3, 5, 7, 11, 13, 17, 19, 23.... are the prime numbers.

### Algorithm:

```
Step 1: Start
Step 2: Initialize variables num,flag=1, j=2
Step 3: Read num from user
Step 4: If num<=1            // Any number less than 1 is not a prime number
             Display "num is not a prime number"
              Goto step 7
Step 5: Repeat the steps until j<[(n/2)+1]
      5.1 If remainder of number divide j equals to 0,
            Set flag=0
            Goto step 6
       5.2 j=j+1
Step 6: If flag==0,
             Display num+" is not prime number"
            Else
            Display num+" n is prime number"
Step 7: Stop
```

**Program:**

```
#include<stdio.h>
int checkPrime(int number)
{
int count = 0;
for(int i=2; i<=number/2; i++)
{
if(number%i == 0)
{
count=1;
break;
}
}
if(number == 1) count = 1;
return count;
}
int main()
{
int number ;
printf("Enter number: ");
scanf("%d",&number);
if(checkPrime(number) == 0)
printf("%d is a prime number.", number);
else
printf("%d is not a prime number.", number);
return 0;

}
```

**Output:**

```
Enter number: 56

56 is not a prime number
```

**B) Write recursive program which computes the nth Fibonacci number, for appropriate values of n**

## Aim:

A c program to print Fibonacci series using recursion

## Description:

The Fibonacci sequence**,** also known as Fibonacci numbers, is defined as the sequence of numbers in which each number in the sequence is equal to the sum of two numbers before it

## Algorithm:

```
Step 1: Start
Step 2: Declare variable a, b, c, n, i
Step 3: Initialize variable a=0, b=1 and  i=2
Step 4: Read n from user
Step 5: Print a and b
Step 6: Repeat until i<=n :
Step 6.1: c=a+b
Step 6.2: print c
Step 6.3: a=b, b=c
Step 6.4: i=i+1
Step 7: Stop
```

## Program:

```
#include<stdio.h>
void printFibonacci(int m){
static int m1=0,m2=1,m3;
if(m>0){
m3 = m1 + m2;
m1 = m2;
m2 = m3;
printf("%d ",m3);
printFibonacci(m-1);
}
}
int main(){
int m;
printf("Please enter your preferred number of elements here: ");
scanf("%d",&m);
printf("The Fibonacci Series will be: ");
printf("%d %d ",0,1);
printFibonacci(m-2); //We have used m-2 because we have 2 numbers already printed here
return 0;
}
```

**Output:**

```
Please enter your preferred number of elements here: 15
The Fibonacci Series will be:
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

**C) Write a program in C to add numbers using call by reference.**

## Aim:

A c program to add numbers using call by reference

## Description:

The call by reference method of passing arguments to a function copies the reference of an argument into the formal parameter. Inside the function, the reference is used to access the actual argument used in the call. This means that changes made to the parameter affect the passed argument.

## Algorithm:

```
// function definition to swap the values.
Step – 1: void swap(int &x, int &y)
Step – 2:   int temp;
Step – 3:   temp = x; /* save the value at address x */
Step – 4:   x = y;    /* put y into x */
Step – 5:   y = temp; /* put x into y */
Step – 6: return
```

## Program

```
#include <stdio.h>

#include <stdlib.h>

int main(){

int num1, num2, result;

/*Accept the numbers from the user*/

printf("\nEnter the two number: ");

scanf("%d %d", &num1, &num2);

/* Pass the value of num1 and num2 as parameter to function add.

The value returned is stored in the variable result*/

result = add(&num1, &num2);

printf("\nAddition of %d and %d is %d", num1, num2, result);

}

/*Defining the function add()*/

int add(int *no1, int *no2)

{

int res;

res = *no1 + *no2;

return res;

}
```

**Output:**

```
Enter the two number: 10 20

Addition of 10 and 20 is 30
```

# Experiment 4:

**A) Write a program in C to append multiple lines at the end of a text file**

## Aim:

A c program to append multiple lines at the end of text file

## Description:

**Source file content**

I love programming.
Programming with files is fun.

**String to append**

Learning C programming at Codeforwin is simple and easy.

**Output file content after append**

I love programming.
Programming with files is fun.
Learning C programming at Codeforwin is simple and easy.

## Algorithm:

```
Step – 1: Input file path from user to append data, store it in some variable say
          filePath.
Step – 2: Declare a FILE type pointer variable say, fPtr.
Step – 3: Open file in a (append file) mode and store reference to fPtr using fPtr =
          fopen(filePath, "a");
Step – 4: Input data to append to file from user, store it to some variable say
          dataToAppend.
Step – 5: Write data to append into file using fputs(dataToAppend, fPtr);
Step – 6: Finally close file to save all changes. Use fclose(fPtr);
```

**Program:**

```
#include <stdio.h>

int main ()

{

FILE * fptr;

int i,n;

char str[100];

char fname[20];

char str1;

printf("\n\n Append multiple lines at the end of a text file :\n");

printf("------------------------------------------------------\n");

printf(" Input the file name to be opened : ");

scanf("%s",fname);

fptr = fopen(fname, "a");

printf(" Input the number of lines to be written : ");

scanf("%d", &n);

printf(" The lines are : \n");

for(i = 0; i < n+1;i++)

{

fgets(str, sizeof str, stdin);

fputs(str, fptr);

}

fclose (fptr);

//----- Read the file after appended -------

fptr = fopen (fname, "r");

printf("\n The content of the file %s is  :\n",fname);

str1 = fgetc(fptr);
while (str1 != EOF)
{
printf ("%c", str1);
str1 = fgetc(fptr);
}
printf("\n\n");
fclose (fptr);
//------- End of reading -----------------
return 0;
}
```

**Output:**

```
Append multiple lines at the end of a text file :
------------------------------------------------------
Input the file name to be opened : test.txt
Input the number of lines to be written : 3
The lines are :
test line 5
test line 6
test line 7

The content of the file test.txt is  :

test line 1
test line 2
test line 3
test line 4
```

test line 5

test line 6

test line 7

**B) Write a program in C to copy a file in another name**

## Aim:

A c program to copy a file in another name

## Description:

To copy the content of one file to another in C programming, you have to first open both the file, that is, the source file and the target file. And then start reading the source file's content character by character and place or write the content of source file to target file on every read of character.

## Algorithm:

```
Step – 1: Input file path of source and destination file.
Step – 2: Open source file in r (read) and destination file in w (write) mode.
Step – 3: Read character from source file and write it to destination file using fputc().
Step – 4: Repeat step 3 till source file has reached end.
Step – 5: Close both source and destination file.
```

**Program:**

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

void main()
{
FILE *fptr1, *fptr2;
char ch, fname1[20], fname2[20];

printf("\n\n Copy a file in another name :\n");
printf("----------------------------------\n");

printf(" Input the source file name : ");
scanf("%s",fname1);

fptr1=fopen(fname1, "r");
if(fptr1==NULL)
{
printf(" File does not found or error in opening.!!");
exit(1);
}
printf(" Input the new file name : ");
scanf("%s",fname2);
fptr2=fopen(fname2, "w");
if(fptr2==NULL)
{
printf(" File does not found or error in opening.!!");
fclose(fptr1);
exit(2);
}
while(1)
{
ch=fgetc(fptr1);
if(ch==EOF)
{
break;
}
else
{
fputc(ch, fptr2);
}
}
printf(" The file %s  copied successfully in the file %s. \n\n",fname1,fname2);
fclose(fptr1);
fclose(fptr2);
getchar();
}
```

**Output:**

```
Copy a file in another name :
----------------------------------
Input the source file name : test.txt
Input the new file name : test1.txt
The file test.txt  copied successfully in the file test1.txt.
```

## Experiment 5:

**Write recursive program for the following**

**A) Write recursive and non-recursive C program for calculation of Factorial of an integer.**

### Aim:

A c program to find factorial of a number using recursion and non-recursion

### Description:

The factorial of a number is the function that multiplies the number by every natural number below it. Symbolically, factorial can be represented as "!". So, *n* factorial is the product of the first *n* natural numbers and is represented as n!

### Algorithm:

```
Step – 1: start
Step – 2: read number n
Step – 3: call factorial (n)
Step – 4: print factorial f
Step – 5: stop
```

```
Factorial (n)
Step – 1: if n==1 then return 1
Step – 2: else
          F=n*factorial(n-1)
Step – 3: return f
```

## Program:

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

void main( )
{
clrscr( )
int factorial(int);
int n,f;
printf("Enter the number: ");
scanf("%d",&n);
f=factorial(n);
printf("Factorial of the number is %d",f);
getch();
}
int factorial(int n)
{
int f;
if(n==1)
return 1;
else
f=n*factorial(n-1);
return f;
}
```

**Output:**

```
Enter the number : 5
Factorial of the number is 120
```

## Aim:

A c program to find factorial of a number using non recursion

## Description:

The factorial of a number is the function that multiplies the number by every natural number below it. Symbolically, factorial can be represented as "!". So, *n* factorial is the product of the first *n* natural numbers and is represented as n!

## Algorithm:

```
Step – 1: int factorial ( int input ) {
Step – 2: int x, fact = 1;
Step – 3: for ( x = input; x > 1; x--)
Step – 4:    fact *= x;
Step – 5: return fact; }
```

**Program:**

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

int main()
{
int n, i;
unsigned long long factorial = 1;

printf("Enter a number to find factorial: ");
scanf("%d",&n);

// show error if the user enters a negative integer
if (n < 0)
printf("Error! Please enter any positive integer number");

else
{
for(i=1; i<=n; ++i)
{
factorial *= i;              // factorial = factorial*i;
}
printf("Factorial of Number %d = %llu", n, factorial);
}

getch();
}
```

**Output:**

```
Enter a number to find factorial: 5
Factorial of number 5 = 120
```

**B) Write recursive and non recursive C program for calculation of GCD (n, m)**

## Aim:

A c program to find GCD using recursion

## Description:

The GCD is a mathematical term for the Greatest Common Divisor of two or more numbers. It is the Greatest common divisor that completely divides two or more numbers without leaving any remainder. Therefore, it is also known as the Highest Common Factor (HCF) of two numbers. For example, the GCD of two numbers, 20 and 28, is 4 because both 20 and 28 are completely divisible by 1, 2, 4 (the remainder is 0), and the largest positive number among the factors 1, 2, and 4 is 4. Similarly, the GCD of two numbers, 24 and 60, is 12.

## Algorithm:

```
Step – 1: if i>j
Step – 2: then return the function with parameters i,j
Step – 3: if i==0
Step – 4: then return j
Step – 5: else return the function with parameters i,j%i.
```

**Program:**

```
#include<stdio.h>
#include<math.h>
unsigned int GCD(unsigned i, unsigned j);
int main(){
int a,b;
printf("Enter the two integers: \n");
scanf("%d%d",&a,&b);
printf("GCD of %d and %d is %d\n",a,b,GCD(a,b));
return 0;
}
/* Recursive Function*/
unsigned int GCD(unsigned i, unsigned j){
if(j>i)
return GCD(j,i);
if(j==0)
return i;
else
return GCD(j,i%j);
}
```

**Output:**

```
Enter the two integers: 4 8
GCD of 4 and 8 is 4
```

## Aim:

A c program to find GCD using non-recursion

## Description:

The GCD is a mathematical term for the Greatest Common Divisor of two or more numbers. It is the Greatest common divisor that completely divides two or more numbers without leaving any remainder. Therefore, it is also known as the Highest Common Factor (HCF) of two numbers. For example, the GCD of two numbers, 20 and 28, is 4 because both 20 and 28 are completely divisible by 1, 2, 4 (the remainder is 0), and the largest positive number among the factors 1, 2, and 4 is 4. Similarly, the GCD of two numbers, 24 and 60, is 12.

## Algorithm:

```
Step – 1: Initialize the i=1, j, remainder
Step -2:  Remainder=i-(i/j*j)
Step – 3:  Remainder=0 return j else goto step 4
Step – 4: GCD(G,remainder) return to main program
```

**Program:**

```
#include<stdio.h>
#include<conio.h>
#include<math.h>
int gcdnonR(int i,int j){
int rem;
rem=i-(i/j*j);
if(rem==0)
return j;
else
gcdnonR(j,rem);
}
void main(){
int a,b;
printf("enter the two numbers:");
scanf("%d%d",&a,&b);
printf("GCD of %d",gcdnonR(a,b));
getch();
}
```

**Output:**

```
enter the two numbers:10 30
GCD of 10
```

C) **Write recursive and non-recursive C program for Towers of Hanoi: N disks are to be transferred from peg S to peg D with Peg I as the intermediate peg**

## Aim:

A c program for towers of Hanoi using recursion

## Description:

Tower of Hanoi is a mathematical puzzle where we have three rods and n disks. The objective of the puzzle is to move the entire stack to another rod, obeying the following simple rules:

1. Only one disk can be moved at a time.
2. Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack i.e. a disk can only be moved if it is the uppermost disk on a stack.
3. No disk may be placed on top of a smaller disk.

## Algorithm:

```
Step – 1: START
Step – 2: Procedure Hanoi(disk, source, dest, aux)
Step – 3:    IF disk == 1, THEN
Step – 4: move disk from source to dest
               ELSE
Step – 5:   Hanoi(disk - 1, source, aux, dest)     // Step 1
Step – 6:  move disk from source to dest          // Step 2
Step – 7: Hanoi(disk - 1, aux, dest, source)     // Step 3
              END IF
              END Procedure
Step – 8: STOP
```

**Program:**

```
/* C program for Tower of Hanoi*/
/*Application of Recursive function*/
#include <stdio.h>
void hanoifun(int n, char fr, char tr, char ar)//fr=from rod,tr =to rod, ar=aux rod
{
if (n == 1)
{
printf("\n Move disk 1 from rod %c to rod %c", fr, tr);
return;
}
hanoifun(n-1, fr, ar, tr);
printf("\n Move disk %d from rod %c to rod %c", n, fr, tr);
hanoifun(n-1, ar, tr, fr);
}

int main()
{
int n = 4; // n immplies the number of discs
hanoifun(n, 'A', 'C', 'B');  // A, B and C are the name of rod
return 0;
}
```

**Output:**

```
Move disk 1 from rod A to rod B
Move disk 2 from rod A to rod C
Move disk 1 from rod B to rod C
Move disk 3 from rod A to rod B
Move disk 1 from rod C to rod A
Move disk 2 from rod C to rod B
Move disk 1 from rod A to rod B
Move disk 4 from rod A to rod C
Move disk 1 from rod B to rod C
Move disk 2 from rod B to rod A
Move disk 1 from rod C to rod A
Move disk 3 from rod B to rod C
Move disk 1 from rod A to rod B
Move disk 2 from rod A to rod C
Move disk 1 from rod B to rod C
```

## Aim:

A c program for towers of Hanoi using non-recursion

## Description:

Tower of Hanoi is a mathematical puzzle where we have three rods and n disks. The objective of the puzzle is to move the entire stack to another rod, obeying the following simple rules:

1. Only one disk can be moved at a time.
2. Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack i.e. a disk can only be moved if it is the uppermost disk on a stack.
3. No disk may be placed on top of a smaller disk.

## Algorithm:

```
Repeat below steps till the total number of moves becomes 2^n - 1
If (n is even)
Make_move (S, E)
Make_move (S, D)
Make_move (E, D)
Else
Make_move (S, D)
Make_move (S, E)
Make_move (E, D)
```

## Program:

```
/* Write C programs that use both recursive and non-recursive functions

To solve Towers of Hanoi problem.*/
#include<conio.h>
#include<stdio.h>
/* Non-Recursive Function*/
void hanoiNonRecursion(int num,char sndl,char indl,char dndl)
{
char stkn[50],stksndl[50],stkindl[50],stkdndl[50],stkadd[50],temp;
int top,add;
top=NULL;
one:
if(num==1)
{
printf("\nMove top disk from needle %c to needle %c ",sndl,dndl);
goto four;
}
two:
top=top+1;
stkn[top]=num;
stksndl[top]=sndl;
```

```
stkindl[top]=indl;
stkdndl[top]=dndl;
stkadd[top]=3;
num=num-1;
sndl=sndl;
temp=indl;
indl=dndl;
dndl=temp;
goto one;
three:
printf("\nMove top disk from needle %c to needle %c ",sndl,dndl);
top=top+1;
stkn[top]=num;
stksndl[top]=sndl;
stkindl[top]=indl;
stkdndl[top]=dndl;
stkadd[top]=5;
num=num-1;
temp=sndl;
sndl=indl;
indl=temp;
dndl=dndl;
goto one;
four:
if(top==NULL)
return;
num=stkn[top];
sndl=stksndl[top];
indl=stkindl[top];
dndl=stkdndl[top];
add=stkadd[top];
top=top-1;
if(add==3)
goto three;
else if(add==5)
goto four;
}
/* Recursive Function*/
void  hanoiRecursion( int num,char ndl1, char ndl2, char ndl3)
{
if ( num == 1 ) {
printf( "\nMove top disk from needle %c to needle %c.", ndl1, ndl2 );
return;
}
hanoiRecursion( num - 1,ndl1, ndl3, ndl2 );
printf( "\nMove top disk from needle %c to needle %c.", ndl1, ndl2 );
hanoiRecursion( num - 1,ndl3, ndl2, ndl1 );
}
int main()
{
```

```
int no;
//clrscr();
printf("Enter the no. of disks to be transferred: ");
scanf("%d",&no);
if(no<1)
printf("\nThere's nothing to move.");
else
printf("Non-Recursive");
hanoiNonRecursion(no,'A','B','C');
printf("\nRecursive");
hanoiRecursion(no,'A','B','C');
return 0;
}
```

**Output:**

```
Enter the no. of disks to be transferred: 3

Non-Recursive

Move top disk from needle A to needle C

Move top disk from needle A to needle B

Move top disk from needle C to needle B

Move top disk from needle A to needle C

Move top disk from needle B to needle A

Move top disk from needle B to needle C

Move top disk from needle A to needle C

Recursive

Move top disk from needle A to needle B.

Move top disk from needle A to needle C.

Move top disk from needle B to needle C.

Move top disk from needle A to needle B.

Move top disk from needle C to needle A.

Move top disk from needle C to needle B.

Move top disk from needle A to needle B.
```

## Experiment 6:

A) **Write C program that use both recursive and non-recursive functions to perform Linear search for a Key value in a given list**

### Aim:

A c program for linear search using both recursion and non-recursion

### Description:

A linear search, also known as a sequential search, is a method of finding an element within a list. It checks each element of the list sequentially until a match is found or the whole list has been searched.

### Algorithm:

```
Linear Search ( Array A, Value x)

Step 1: Set i to 1
Step 2: if i > n then go to step 7
Step 3: if A[i] = x then go to step 6
Step 4: Set i to i + 1
Step 5: Go to Step 2
Step 6: Print Element x Found at index i and go to step 8
Step 7: Print element not found
Step 8: Exit
```

## Program:

```
// Write C programs  that use both recursive and non recursive functions
// to perform the following searching operation for a Key value in a given list of
integers :
// i) Linear search

#include <stdio.h>
#include <conio.h>
#define MAX_LEN 10

void l_search_recursive(int l[],int num,int ele);
void l_search_nonrecursive(int l[],int num,int ele);
void read_list(int l[],int num);
void print_list(int l[],int num);

int main()
{
int l[MAX_LEN], num, ele;
int ch;

printf("======================================================");
printf("\n\t\t\tMENU");
printf("\n=====================================================")
;
printf("\n[1] Linary Search using Recursion method");
printf("\n[2] Linary Search using Non-Recursion method");
printf("\n\nEnter your Choice:");
scanf("%d",&ch);

if(ch<=2 & ch>0)
{
printf("Enter the number of elements :");
scanf("%d",&num);
read_list(l,num);
printf("\nElements present in the list are:\n\n");
print_list(l,num);
printf("\n\nElement you want to search:\n\n");
scanf("%d",&ele);

switch(ch)
{
case 1:
```

```
printf("\n**Recursion method**\n");
l_search_recursive(l,num,ele);
getch();
break;

case 2:
printf("\n**Non-Recursion method**\n");
l_search_nonrecursive(l,num,ele);
getch();
break;
}
}
getch();
}

//end main

//Non-Recursive method

void l_search_nonrecursive(int l[],int num,int ele)
{
int j, f=0;
for(j=0; j<num; j++)
if( l[j] == ele)
{
printf("\nThe element %d is present at position %d in list\n",ele,j);
f=1;
break;
}
if(f==0)
printf("\nThe element is %d is not present in the list\n",ele);
}

//Recursive method

void l_search_recursive(int l[],int num,int ele)
{
int f = 0;

if( l[num] == ele)
{
printf("\nThe element %d is present at position %d in list\n",ele,num);
f=1;
}
```

```
else
{
if((num==0) && (f==0))
{
printf("The element %d is not found.",ele);
}
else
{
l_search_nonrecursive(l,num-1,ele);
}
}
getch();
}

void read_list(int l[],int num)
{
int j;
printf("\nEnter the elements:\n");
for(j=0; j<num; j++)
scanf("%d",&l[j]);
}

void print_list(int l[],int num)
{
int j;
for(j=0; j<num; j++)
printf("%d\t",l[j]);
}
```

**Output:**

```
=====================================================
MENU
=====================================================
[1] Linary Search using Recursion method
[2] Linary Search using Non-Recursion method

Enter your Choice:1
Enter the number of elements :12 22 32 42
Elements present in the list are: 12 22 32 42
Enter the element you want to search: 42
Recursive method:
Element is found at 4 position
```

B) **Write C program that use both recursive and non recursive functions to perform Binary search for a Key value in a given list.**

## Aim:

A c program for binary search using both recursion and non-recursion

## Description:

Binary Search is a searching algorithm used in a sorted array by repeatedly dividing the search interval in half. The idea of binary search is to use the information that the array is sorted and reduce the time complexity to O(Log n).

## Algorithm:

```
Procedure binary_search
A ← sorted array
n ← size of array
x ← value to be searched

Set lowerBound = 1
Set upperBound = n

while x not found
if upperBound < lowerBound
EXIT: x does not exists.

set midPoint = lowerBound + ( upperBound - lowerBound ) / 2

if A[midPoint] < x
set lowerBound = midPoint + 1

if A[midPoint] > x
set upperBound = midPoint - 1

if A[midPoint] = x
EXIT: x found at location midPoint
end while

end procedure
```

**Program:**

```
/* Binary search program in C using both recursive and non recursive functions */

#include <stdio.h>
#define MAX_LEN 10

/* Non-Recursive function*/
void b_search_nonrecursive(int l[],int num,int ele)
{
int l1,i,j, flag = 0;
l1 = 0;
i = num-1;
while(l1 <= i)
{
j = (l1+i)/2;
if( l[j] == ele)
{
printf("\nThe element %d is present at position %d in list\n",ele,j);
flag =1;
break;
}
else
if(l[j] < ele)
l1 = j+1;
else
i = j-1;
}
if( flag == 0)
printf("\nThe element %d is not present in the list\n",ele);
}

/* Recursive function*/
int b_search_recursive(int l[],int arrayStart,int arrayEnd,int a)
{
int m,pos;
if (arrayStart<=arrayEnd)
{
m=(arrayStart+arrayEnd)/2;
if (l[m]==a)
return m;
else if (a<l[m])
return b_search_recursive(l,arrayStart,m-1,a);
else
```

```
return b_search_recursive(l,m+1,arrayEnd,a);
}
return -1;
}
void read_list(int l[],int n)
{
int i;
printf("\nEnter the elements:\n");
for(i=0;i<n;i++)
scanf("%d",&l[i]);
}
void print_list(int l[],int n)
{
int i;
for(i=0;i<n;i++)
printf("%d\t",l[i]);
}
/*main function*/
main()
{
int l[MAX_LEN], num, ele,f,l1,a;
int ch,pos;
//clrscr();
printf("======================================================");
printf("\n\t\t\tMENU");
printf("\n=====================================================");
printf("\n[1] Binary Search using Recursion method");
printf("\n[2] Binary Search using Non-Recursion method");
printf("\n\nEnter your Choice:");
scanf("%d",&ch);
if(ch<=2 & ch>0)
{
printf("\nEnter the number of elements : ");
scanf("%d",&num);
read_list(l,num);
printf("\nElements present in the list are:\n\n");
print_list(l,num);
printf("\n\nEnter the element you want to search:\n\n");
scanf("%d",&ele);
switch(ch)
{
case 1:printf("\nRecursive method:\n");
pos=b_search_recursive(l,0,num,ele);
if(pos==-1)
{
printf("Element is not found");
}
else
{
printf("Element is found at %d position",pos);
}
//getch();
break;
case 2:printf("\nNon-Recursive method:\n");
b_search_nonrecursive(l,num,ele);
//getch();
break;
}
}
//getch();
}
```

**Output:**

```
===================================================
MENU
==================================================
[1] Binary Search using Recursion method
[2] Binary Search using Non-Recursion method
Enter your Choice:1
Enter the number of elements : 5
Enter the elements:
12
22
32
42
52
Elements present in the list are:
12      22      32      42      52
Enter the element you want to search:
42
Recursive method:
Element is found at 3 position
```

# Experiment 7:

**A) Write C program that implement stack (its operations) using arrays**

## Aim:

A c program for implementation of stack using arrays

## Description:

Stacks can be maintained in a program by using a linear array of elements and pointer variables TOP and MAX. The TOP pointer contains the location of the top element of the stack and MAX gives the maximum number of elements in the stack.

The stack is empty if TOP=-1 or TOP=NULL. The following figure shows an array representation of the stack.

## Algorithm:

**//push**

```
Step 1: Start

Step 2: Declare  Stack[MAX];    //Maximum size of Stack

Step 3: Check if the stack is full or not by comparing top with (MAX-1)
        If the stack is full, Then print "Stack Overflow" i.e, stack is full and cannot be pushed
        with another element

Step 4: Else, the stack is not full
        Increment top by 1 and Set, a[top] = x
        which pushes the element x into the address pointed by top.
// The element x is stored in a[top]

Step 5: Stop
```

**//pop**

```
Step 1: Start

Step 2: Declare  Stack[MAX]

Step 3: Push the elements into the stack

Step 4: Check if the stack is empty or not by comparing top with base of array i.e 0
If top is less than 0, then stack is empty, print "Stack Underflow"

Step 5: Else, If top is greater than zero the stack is not empty, then store the value pointed by
top in a variable x=a[top] and decrement top by 1. The popped element is x.
```

**//peek**

```
Step 1: Start
Step 2: Declare  Stack[MAX]
Step 3: Push the elements into the stack
Step 4: Print the value stored in the stack pointed by top.
Step 6: Stop
```

**Program:**

```
#include<stdio.h>
int stack[100],choice,n,top,x,i;
void push(void);
void pop(void);
void display(void);
int main()
{
//clrscr();
top=-1;
printf("\n Enter the size of STACK[MAX=100]:");
scanf("%d",&n);
printf("\n\t STACK OPERATIONS USING ARRAY");
printf("\n\t--------------------------------");
printf("\n\t 1.PUSH\n\t 2.POP\n\t 3.DISPLAY\n\t 4.EXIT");
do
{
printf("\n Enter the Choice:");
scanf("%d",&choice);
switch(choice)
{
case 1:
{
push();
break;
}
case 2:
{
pop();
break;
}
case 3:
{
display();
break;
}
case 4:
{
printf("\n\t EXIT POINT ");
break;
}
default:
{
```

```
printf ("\n\t Please Enter a Valid Choice(1/2/3/4)");
}

}
}
while(choice!=4);
return 0;
}
void push()
{
if(top>=n-1)
{
printf("\n\tSTACK is over flow");

}
else
{
printf(" Enter a value to be pushed:");
scanf("%d",&x);
top++;
stack[top]=x;
}
}
void pop()
{
if(top<=-1)
{
printf("\n\t Stack is under flow");
}
else
{
printf("\n\t The popped elements is %d",stack[top]);
top--;
}
}
void display()
{
if(top>=0)
{
printf("\n The elements in STACK \n");
for(i=top; i>=0; i--)
printf("\n%d",stack[i]);
printf("\n Press Next Choice");
}
```

```
else
{
printf("\n The STACK is empty");
}

}
```

**Output:**

```
Enter the size of STACK[MAX=100]:10

STACK OPERATIONS USING ARRAY

--------------------------------

1.PUSH

2.POP

3.DISPLAY

4.EXIT

Enter the Choice:1

Enter a value to be pushed:12

Enter the Choice:1

Enter a value to be pushed:24

Enter the Choice:1

Enter a value to be pushed:98

Enter the Choice:3

The elements in STACK

98

24

12

Press Next Choice

Enter the Choice:2

The popped elements is 98

Enter the Choice:3
```

```
The elements in STACK
24
12
Press Next Choice
Enter the Choice:4
EXIT POINT
```

**B) Write C program that implement stack (its operations) using Linked list.**

## Aim:

A c program to implementation of stack using linked list

## Description:

Instead of using array, we can also use linked list to implement stack. Linked list allocates the memory dynamically. However, time complexity in both the scenario is same for all the operations i.e. push, pop and peek.

In linked list implementation of stack, the nodes are maintained non-contiguously in the memory. Each node contains a pointer to its immediate successor node in the stack. Stack is said to be overflown if the space left in the memory heap is not enough to create a node.

## Algorithm:

**//push**

Step 1: Start

Step 2: Create a node new and declare variable top

Step 3: Set new data part to be Null
// The first node is created, having null value and top pointing to it

Step 4: Read the node to be inserted.

Step 5: Check if the node is Null, then print "Insufficient Memory"

Step 6: If node is not Null, assign the item to data part of new and assign top to link part of new and also point stack head to new.

**//pop**

Step 1: Start

Step 2: Check if the top is Null, then print "Stack Underflow."

Step 3: If top is not Null, assign the top's link part to ptr and assign ptr to stack_head's link part.

Step 4: Stop

**//peek**

```
Step 1: Start
Step 2: Print or store the node pointed by top variable
Step 3: Stop
```

**Program:**

```
/* C Program to Implement a Stack using Linked List*/
#include <stdio.h>
#include <stdlib.h>
struct node
{
int info;
struct node *ptr;
}*top,*top1,*temp;

int topelement();
void push(int data);
void pop();
void empty();
void display();
void destroy();
void stack_count();
void create();
int count = 0;
void main()
{
int no, ch, e;
printf("\n 1 - Push");
printf("\n 2 - Pop");
printf("\n 3 - Top");
printf("\n 4 - Empty");
printf("\n 5 - Exit");
printf("\n 6 - Dipslay");
printf("\n 7 - Stack Count");
printf("\n 8 - Destroy stack");
create();
while (1)
{
printf("\n Enter choice : ");
scanf("%d", &ch);
switch (ch)
{
case 1:
printf("Enter data : ");
scanf("%d", &no);
push(no);
break;
case 2:
pop();
break;
case 3:
if (top == NULL)
printf("No elements in stack");
else
{
```

```
e = topelement();
printf("\n Top element : %d", e);
}
break;
case 4:
empty();
break;
case 5:
exit(0);
case 6:
display();
break;
case 7:
stack_count();
break;
case 8:
destroy();
break;
default :
printf(" Wrong choice, Please enter correct choice  ");
break;
}
}
}
/* Create empty stack */
void create()
{
top = NULL;
}
/* Count stack elements */
void stack_count()
{
printf("\n No. of elements in stack : %d", count);
}
/* Push data into stack */
void push(int data)
{
if (top == NULL)
{
top =(struct node *)malloc(1*sizeof(struct node));
top->ptr = NULL;
top->info = data;
}
else
{
temp =(struct node *)malloc(1*sizeof(struct node));
temp->ptr = top;
temp->info = data;
top = temp;
}
```

```
count++;
}
/* Display stack elements */
void display()
{
top1 = top;
if (top1 == NULL)
{
printf("Stack is empty");
return;
}
while (top1 != NULL)
{
printf("%d ", top1->info);
top1 = top1->ptr;
}
}
/* Pop Operation on stack */
void pop()
{
top1 = top;
if (top1 == NULL)
{
printf("\n Error : Trying to pop from empty stack");
return;
}
else
top1 = top1->ptr;
printf("\n Popped value : %d", top->info);
free(top);
top = top1;
count--;
}
/* Return top element */
int topelement()
{
return(top->info);
}
/* Check if stack is empty or not */
void empty()
{
if (top == NULL)
printf("\n Stack is empty");
else
printf("\n Stack is not empty with %d elements", count);
}
/* Destroy entire stack */
void destroy()
{
top1 = top;
```

```
while (top1 != NULL)
{
top1 = top->ptr;
free(top);
top = top1;
top1 = top1->ptr;
}
free(top1);
top = NULL;

printf("\n All stack elements destroyed");
count = 0;
}
```

**Output:**

```
$ cc pgm2.c
$ a.out

1 - Push
2 - Pop
3 - Top
4 - Empty
5 - Exit
6 - Dipslay
7 - Stack Count
8 - Destroy stack
Enter choice : 1
Enter data : 56

Enter choice : 1
Enter data : 80

Enter choice : 2

Popped value : 80
Enter choice : 3

Top element : 56
Enter choice : 1
Enter data : 78

Enter choice : 1
Enter data : 90
Enter choice : 6
90 78 56
Enter choice : 7

No. of elements in stack : 3
Enter choice : 8

All stack elements destroyed
Enter choice : 4

Stack is empty
Enter choice : 5
```

## Experiment 8:

A) **Write a C program that uses Stack operations to convert infix expression into postfix expression**

### Aim:

A c program to convert infix to postfix expression

### Description:

Infix expressions are readable and solvable by humans. We can easily distinguish the order of operators, and also can use the parenthesis to solve that part first during solving mathematical expressions. The computer cannot differentiate the operators and parenthesis easily, that's why postfix conversion is needed.

To convert infix expression to postfix expression, we will use the stack data structure. By scanning the infix expression from left to right, when we will get any operand, simply add them to the postfix form, and for the operator and parenthesis, add them in the stack maintaining the precedence of them.

### Algorithm:

```
1. Push "("onto Stack, and add ")" to the end of X.
2. Scan X from left to right and repeat Step 3 to 6 for each element of X until the Stack
   is empty.
3. If an operand is encountered, add it to Y.
4. If a left parenthesis is encountered, push it onto Stack.
5. If an operator is encountered ,then:
    1. Repeatedly pop from Stack and add to Y each operator (on the top of Stack)
       which has the same precedence as or higher precedence than operator.
    2. Add operator to Stack.
       [End of If]
6. If a right parenthesis is encountered ,then:
    1. Repeatedly pop from Stack and add to Y each operator (on the top of Stack)
       until a left parenthesis is encountered.
    2. Remove the left Parenthesis.
       [End of If]
       [End of If]
7. END.
```

## Program:

```
#include <stdio.h>
#include <ctype.h>
#define SIZE 50
char stack[SIZE];
int top=-1;       /* Global declarations */
push(char elem)
{                        /* Function for PUSH operation */
stack[++top]=elem;
}
char pop()
{                       /* Function for POP operation */
return(stack[top--]);
}
int pr(char symbol)
{                  /* Function for precedence */
if(symbol == '^')/* exponent operator, highest precedence*/
{
return(3);
}
else if(symbol == '*' || symbol == '/')
{
return(2);
}
else if(symbol == '+' || symbol == '-')          /* lowest precedence */
{
return(1);
}
else
{
return(0);
```

```
}
}
/* Main Program */
void main()
{
char infix[50],postfix[50],ch,elem;
int i=0,k=0;
printf("Enter Infix Expression : ");
scanf("%s",infix);
push('#');
while( (ch=infix[i++]) != '\0')
{
if( ch == '(') push(ch);
else
if(isalnum(ch)) postfix[k++]=ch;
else
if( ch == ')')
{
while( stack[top] != '(')
postfix[k++]=pop();
elem=pop(); /* Remove ( */
}
else
{       /* Operator */
while( pr(stack[top]) >= pr(ch) )
postfix[k++]=pop();
push(ch);
}
}
while( stack[top] != '#')     /* Pop from stack till empty */
postfix[k++]=pop();
postfix[k]='\0';          /* Make postfix as valid string */
printf("\nPostfix Expression =  %s\n",postfix);
}
```

**Output:**

```
Enter Infix Expression : A*(B+C)-D

Postfix Expression =  ABC+*D-
```

**B) Write C program that implement Queue (its operations) using arrays**

## Aim:

A c program for implementation of queue using array

## Description:

To implement a queue using array, create an array *arr* of size *n* and take two variables *front* and *rear* both of which will be initialized to *0* which means the queue is currently empty. Element *rear* is the index upto which the elements are stored in the array and *front* is the index of the first element of the array.

## Algorithm:

```
Step 1: IF REAR = MAX - 1
Write OVERFLOW
Go to step
[END OF IF]

Step 2: IF FRONT = -1 and REAR = -1
SET FRONT = REAR = 0
ELSE
SET REAR = REAR + 1
[END OF IF]

Step 3: Set QUEUE[REAR] = NUM

Step 4: EXIT
```

**Program:**

```
#include <stdio.h>

#include<stdlib.h>

#define MAX 6

void insert();

void remov();

void display();

int queue[MAX], rear=-1, front=-1, item;

main()

{

int ch;

do

{

printf("\n\n1. Insert\n2. Delete\n3. Display\n4. Exit\n");

printf("\nEnter your choice:");

scanf("%d", &ch);

switch(ch)

{

case 1:

insert();

break;

case 2:
```

```
remov();

break;

case 3:

display();

break;

case 4:

exit(0);

default:

printf("\n\nInvalid entry. Please try again...\n");

}

} while(ch<=4);

}

void insert()

{

if(rear == MAX-1)

printf("\nQueue is full.");

else

{

printf("\n\nEnter ITEM:");

scanf("%d", &item);

if (rear == -1 && front == -1)

{
```

```
rear = 0;

front = 0;

}

else

rear++;

queue[rear] = item;

printf("\n\nItem inserted: %d", item);

}

}

void remov()

{

if(front == -1)

printf("\n\nQueue is empty.");

else

{

item = queue[front];

if (front == rear)

{

front = -1;

rear = -1;

}

else
```

```
front++;

printf("\n\nItem deleted: %d", item);

}

}

void display()

{

int i;

if(front == -1)

printf("\n\nQueue is empty.");

else


{

printf("\n\n");

for(i=front; i<=rear; i++)

printf( "%d", queue[i]);

}

}
```

**Output:**

Input:

```
1 22 1 33 1 44 2 3 4
```

Output:

```
1. Insert

2. Delete

3. Display

4. Exit

Enter your choice:

Enter ITEM:

Item inserted: 22

1. Insert

2. Delete

3. Display

4. Exit

Enter your choice:

Enter ITEM:

Item inserted: 33

1. Insert

2. Delete

3. Display

4. Exit

Enter your choice:

Enter ITEM:

Item inserted: 44
```

```
1. Insert

2. Delete

3. Display

4. Exit

Enter your choice:

Item deleted: 22

1. Insert

2. Delete

3. Display

4. Exit

Enter your choice:

3344

1. Insert

2. Delete

3. Display

4. Exit

Enter your choice:
```

C) **Write C program that implement Queue (its operations) using linked lists**

## Aim:

A c program for implementation of queue using linked lists

## Description:

The major problem with the queue implemented using an array is, It will work for an only fixed number of data values. That means, the amount of data must be specified at the beginning itself. Queue using an array is not suitable when we don't know the size of data which we are going to use. A queue data structure can be implemented using a linked list data structure. The queue which is implemented using a linked list can work for an unlimited number of values. That means, queue using linked list can work for the variable size of data (No need to fix the size at the beginning of the implementation). The Queue implemented using linked list can organize as many data values as we want.

In linked list implementation of a queue, the last inserted node is always pointed by 'rear' and the first node is always pointed by 'front'.

## Algorithm:

```
Step 1: Allocate the space for the new node PTR
Step 2: SET PTR -> DATA = VAL
Step 3: IF FRONT = NULL
            SET FRONT = REAR = PTR
            SET FRONT -> NEXT = REAR -> NEXT = NULL
            ELSE
            SET REAR -> NEXT = PTR
            SET REAR = PTR
            SET REAR -> NEXT = NULL
            [END OF IF]
Step 4: END
```

**Program:**

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
int data;
struct node *next;
};
struct node *front;
struct node *rear;
void insert();
void dequeue();
void display();
void main ()
{
int choice;
printf("\nQUEUE OPERATIONS USING LINKED LIST");
printf("\n1.insert an element\n2.Delete an element\n3.Display the queue\n4.Exit");
while(choice != 4)
{
printf("\nEnter your choice ?");
scanf("%d",& choice);
switch(choice)
{
case 1:
insert();
break;
case 2:
dequeue();
break;
case 3:
display();
break;
case 4:
exit(0);
break;
default:
printf("\nEnter valid choice\n");
}
}
}
void insert()
{
```

```
struct node *ptr;
int item;

ptr = (struct node *) malloc (sizeof(struct node));
if(ptr == NULL)
{
printf("\nOVERFLOW\n");
return;
}
else
{
printf("\nEnter value : ");
scanf("%d",&item);
ptr -> data = item;
if(front == NULL)
{
front = ptr;
rear = ptr;
front -> next = NULL;
rear -> next = NULL;
}
else
{
rear -> next = ptr;
rear = ptr;
rear->next = NULL;
}
}
}
void dequeue()
{
struct node *ptr;
if(front == NULL)
{
printf("\nUNDERFLOW\n");
return;
}
else
{
ptr = front;
front = front -> next;
free(ptr);
}
}
```

```
void display()
{
struct node *ptr;
ptr = front;
if(front == NULL)
{
printf("\nEmpty queue\n");
}
else
{   printf("\nElements in the queue are : \n");
while(ptr != NULL)
{
printf("\n%d\n",ptr -> data);
ptr = ptr -> next;
}
}
}
```

**Output:**

```
QUEUE OPERATIONS USING LINKED LIST
1.insert an element
2.Delete an element
3.Display the queue
4.Exit
Enter your choice ?1
Enter value : 5
Enter your choice ?1
Enter value : 6
Enter your choice ?1
Enter value : 7
Enter your choice ?1
Enter value : 8
Enter your choice ?3
Elements in the queue are :
5
6
7
8
Enter your choice ?2
Enter your choice ?3
Elements in the queue are :

6
7
```

```
8
Enter your choice ?4
```

## Experiment 9:

**Write a C program that uses functions to create a singly linked list and perform various operations on it**

### Aim:

A c program for simple singly linked list example using functions

### Description:

The singly linked list is a linear data structure in which each element of the list contains a pointer which points to the next element in the list. Each element in the singly linked list is called a node. Each node has two components: data and a pointer next which points to the next node in the list. The first node of the list is called as head, and the last node of the list is called a tail. The last node of the list contains a pointer to the null. Each node in the list can be accessed linearly by traversing through the list from head to tail.

### Algorithm:

```
Step 1: [INITIALIZE] SET = 0
Step 2: [INITIALIZE] SET PTR = START
Step 3: Repeat Steps 4 and 5 while PTR != NULL
Step 4: SET COUNT = COUNT + 1
Step 5: SET PTR = PTR-> NEXT
        [END OF LOOP]
Step 6: Write COUNT
Step 7: EXIT
```

**Program:**

```
/* Simple Singly(Single) Linked List Example Program Using Functions in C*/
/* Data Structure Programs,C Linked List Examples */

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

struct node {
int value;
struct node *next;
};

void insert(int);
void display();

typedef struct node DATA_NODE;

DATA_NODE *head_node, *first_node, *temp_node = 0;

int main() {
int loop = 1;
int data;
first_node = 0;

printf("Singly(Single) Linked List Example - Using Functions\n");

while (loop) {
printf("\nEnter Element for Insert Linked List (-1 to Exit ) : \n");
scanf("%d", &data);

if (data >= 0) {
insert(data);
} else {
loop = 0;
temp_node->next = 0;
}
}

display();
return 0;
}
```

```
void insert(int data) {
temp_node = (DATA_NODE *) malloc(sizeof (DATA_NODE));

temp_node->value = data;

if (first_node == 0) {
first_node = temp_node;
} else {
head_node->next = temp_node;
}
head_node = temp_node;
fflush(stdin);
}

void display() {
int count = 0;
temp_node = first_node;
printf("\nDisplay Linked List : \n");
while (temp_node != 0) {
printf("# %d # ", temp_node->value);
count++;
temp_node = temp_node -> next;
}
printf("\nNo Of Items In Linked List : %d", count);
}
```

**Output:**

```
Singly(Single) Linked List Example - Using Functions

Enter Element for Insert Linked List (-1 to Exit ) :
555

Enter Element for Insert Linked List (-1 to Exit ) :
444

Enter Element for Insert Linked List (-1 to Exit ) :
333

Enter Element for Insert Linked List (-1 to Exit ) :
222

Enter Element for Insert Linked List (-1 to Exit ) :
```

```
111

Enter Element for Insert Linked List (-1 to Exit ) :
-1

Display Linked List :
# 555 # # 444 # # 333 # # 222 # # 111 #
No Of Items In Linked List : 5

------------------
(program exited with code: 0)

Press any key to continue . . .
```

## Experiment 10:

**Write a C program to store a polynomial expression in memory using linked list and perform polynomial addition.**

## Aim:

A c program to add two polynomials using linked list

## Description:

linked list is a data structure that stores each element as an object in a node of the list. every note contains two parts data han and links to the next node.

Polynomial is a mathematical expression that consists of variables and coefficients. for example x^2 - 4x + 7

In the Polynomial linked list, the coefficients and exponents of the polynomial are defined as the data node of the list.

For adding two polynomials that are stored as a linked list. We need to add the coefficients of variables with the same power. In a linked list node contains 3 members, coefficient value link to the next node.

## Algorithm:

Step 1: loop around all values of linked list and follow step 2& 3.

Step 2: if the value of a node’s exponent. is greater copy this node to result node and head towards the next node.

Step 3: if the values of both node’s exponent is same add the coefficients and then copy the added value with node to the result.

Step 4: Print the resultant node.

**Program:**

```
#include<stdio.h>
#include<stdlib.h>
typedef struct link {
int coeff;
int pow;
struct link * next;
} my_poly;

/** The prototypes */
void my_create_poly(my_poly **);
void my_show_poly(my_poly *);
void my_add_poly(my_poly **, my_poly *, my_poly *);

/*The simple menu driven main function*/
int main(void) {
int ch;
do {
my_poly * poly1, * poly2, * poly3;

printf("\nCreate 1st expression\n");
my_create_poly(&poly1);
printf("\nStored the 1st expression");
my_show_poly(poly1);

printf("\nCreate 2nd expression\n");
my_create_poly(&poly2);
printf("\nStored the 2nd expression");
my_show_poly(poly2);

my_add_poly(&poly3, poly1, poly2);
my_show_poly(poly3);

printf("\nAdd two more expressions? (Y = 1/N = 0): ");
scanf("%d", &ch);
} while (ch);
return 0;
}

void my_create_poly(my_poly ** node) {
int flag; //A flag to control the menu
int coeff, pow;
```

```
my_poly * tmp_node; //To hold the temporary last address
tmp_node = (my_poly *) malloc(sizeof(my_poly)); //create the first node
*node = tmp_node; //Store the head address to the reference variable
do {
//Get the user data
printf("\nEnter Coeff:");
scanf("%d", &coeff);
tmp_node->coeff = coeff;
printf("\nEnter Pow:");
scanf("%d", &pow);
tmp_node->pow = pow;
//Done storing user data

//Now increase the Linked on user condition
tmp_node->next = NULL;

//Ask user for continuation
printf("\nContinue adding more terms to the polynomial list?(Y = 1/N = 0): ");
scanf("%d", &flag);
//printf("\nFLAG: %c\n", flag);
//Grow the linked list on condition
if(flag) {
tmp_node->next = (my_poly *) malloc(sizeof(my_poly)); //Grow the list
tmp_node = tmp_node->next;
tmp_node->next = NULL;
}
} while (flag);
}

void my_show_poly(my_poly * node) {
printf("\nThe polynomial expression is:\n");
while(node != NULL) {
printf("%dx^%d", node->coeff, node->pow);
node = node->next;
if(node != NULL)
printf(" + ");
}
}

void my_add_poly(my_poly ** result, my_poly * poly1, my_poly * poly2) {
my_poly * tmp_node; //Temporary storage for the linked list
tmp_node = (my_poly *) malloc(sizeof(my_poly));
tmp_node->next = NULL;
*result = tmp_node; //Copy the head address to the result linked list
```

```
//Loop while both of the linked lists have value
while(poly1 && poly2) {
if (poly1->pow > poly2->pow) {
tmp_node->pow = poly1->pow;
tmp_node->coeff = poly1->coeff;
poly1 = poly1->next;
}
else if (poly1->pow < poly2->pow) {
tmp_node->pow = poly2->pow;
tmp_node->coeff = poly2->coeff;
poly2 = poly2->next;
}
else {
tmp_node->pow = poly1->pow;
tmp_node->coeff = poly1->coeff + poly2->coeff;
poly1 = poly1->next;
poly2 = poly2->next;
}

//Grow the linked list on condition
if(poly1 && poly2) {
tmp_node->next = (my_poly *) malloc(sizeof(my_poly));
tmp_node = tmp_node->next;
tmp_node->next = NULL;
}
}

//Loop while either of the linked lists has value
while(poly1 || poly2) {
//We have to create the list at beginning
//As the last while loop will not create any unnecessary node
tmp_node->next = (my_poly *) malloc(sizeof(my_poly));
tmp_node = tmp_node->next;
tmp_node->next = NULL;

if(poly1) {
tmp_node->pow = poly1->pow;
tmp_node->coeff = poly1->coeff;
poly1 = poly1->next;
}
if(poly2) {
tmp_node->pow = poly2->pow;
tmp_node->coeff = poly2->coeff;
poly2 = poly2->next;
}
}

printf("\nAddition Complete");
}
```

**Output:**

```
Create 1st expression

Enter Coeff:1

Enter Pow:1

Continue adding more terms to the polynomial list?(Y = 1/N = 0): 1

Enter Coeff:5

Enter Pow:2

Continue adding more terms to the polynomial list?(Y = 1/N = 0): 0

Stored the 1st expression
The polynomial expression is:
1x^1 + 5x^2
Create 2nd expression

Enter Coeff:5

Enter Pow:3

Continue adding more terms to the polynomial list?(Y = 1/N = 0): 1

Enter Coeff:16

Enter Pow:4

Continue adding more terms to the polynomial list?(Y = 1/N = 0): 0

Stored the 2nd expression
The polynomial expression is:
5x^3 + 16x^4
```

```
Addition Complete
The polynomial expression is:
5x^3 + 16x^4 + 1x^1 + 5x^2
Add two more expressions? (Y = 1/N = 0): 0
```

## Experiment 11:

**A) Write a recursive C program for traversing a binary tree in preorder, inorder and postorder**

### Aim:

A c program for binary tree traversing using recursion

### Description:

To process data stored in the tree data structure, we need to traverse each node of the tree. The process to visit all nodes of a tree is called tree traversal. It is just like searching the tree and accessing each node only once. During the visit of a tree node, we can perform various operations with the data stored in the node.
We access data elements in sequential order in the case of linear data structures. But in binary tree data structures, there can be many different ways to access data elements. Starting from the root node, there are two directions to go: either we can go left via the left pointer or we can go right via the right pointer.

### Algorithm:

**//inorder**

```
Step 1: Repeat Steps 2 to 4 while TREE != NULL
Step 2:     INORDER(TREE -> LEFT)
Step 3:     Write TREE -> DATA
Step 4:     INORDER(TREE -> RIGHT)
[END OF LOOP]
Step 5: END
```

**//preorder**

```
Step 1: Repeat Steps 2 to 4 while TREE != NULL
Step 2:     Write TREE -> DATA
Step 3:     PREORDER(TREE -> LEFT)
Step 4:     PREORDER(TREE -> RIGHT)
[END OF LOOP]
```

```
Step 5: END
```

**//postorder**

```
Step 1: Repeat Steps 2 to 4 while TREE != NULL
Step 2:     POSTORDER(TREE -> LEFT)
Step 3:     POSTORDER(TREE -> RIGHT)
Step 4:     Write TREE -> DATA
[END OF LOOP]
Step 5: END
```

**Program:**

```
#include<stdio.h>

#include<malloc.h>

#include<stdlib.h>

struct node

{

int data;

struct node *lptr,*rptr;

};

void Insert(struct node *,int);

void Preorder(struct node *);

void Postorder(struct node *);

void Inorder(struct node *);

void Delete(struct node *,int);

struct node *Header;

int main()

{

int ch,x;

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

Header->lptr=Header;

Header->rptr=Header;

do

{

printf("\n1.Insert a node in a Tree");
printf("\n2.Preorder Traversal(Recursively)");

printf("\n3.Postorder Traversal(Recursively)");

printf("\n4.Inorder Traversal(Recursively)");

printf("\n5.Delete a node from Binary Search Tree");

printf("\n6.Exit");

printf("\n\nEnter Choice: ");

scanf("%d",&ch);

switch(ch)

{

case 1:

printf("\n\tEnter Element : ");

scanf("%d",&x);

Insert(Header,x);

printf("\n\tInorder Traversal(Recursively)=\n\t");

printf("\n\t------------------------------------\n\t");

Inorder(Header->lptr);

printf("\n\t------------------------------------\n\t");

break;

case 2:

printf("\n\tPreorder Traversal(Recursively):\n\t");

printf("\n\t------------------------------------\n\t");

Preorder(Header->lptr);

printf("\n\t------------------------------------\n\t");

break;
case 3:

printf("\n\tPostorder Traversal(Recursively):\n\t");

printf("\n\t------------------------------------\n\t");

Postorder(Header->lptr);

printf("\n\t------------------------------------\n\t");

break;

case 4:

printf("\n\tInorder Traversal(Recursively):\n\t");

printf("\n\t------------------------------------\n\t");

Inorder(Header->lptr);

printf("\n\t------------------------------------\n\t");

break;

case 5:

printf("\n\tEnter Element which u want to Delete: ");

scanf("%d",&x);

printf("\n\t------------------------------------\n\t");

Delete(Header,x);

printf("\n\t------------------------------------\n\t");

break;

case 6:  exit(0);

break;

default:

printf("\n\t Plz Try Again.");

}
```

```
}while(6);

}

void Insert(struct node *h,int x)

{

struct node *New,*parent,*cur;

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

if(New==NULL)

{

printf("\n\tNo Tree Node Available.");

return;

}

New->data=x;

New->lptr=NULL;

New->rptr=NULL;

if(h->lptr==h)

{

h->lptr=New;

return;

}

cur=h->lptr;

parent=h;

while(cur!=NULL)

{

if(x<cur->data)
```

```
{
parent=cur;
cur=cur->lptr;
}
else if(x>cur->data)
{
parent=cur;
cur=cur->rptr;
}
else
{
printf("\n\t Element Already Exist.\n");
return;
}
}
if(x<parent->data)
{
parent->lptr=New;
return;
}
if(x>parent->data)
{
parent->rptr=New;
return;
```

```
}
return;
}
void Preorder(struct node *t)
{
if(t!=NULL)
printf("%d  ",t->data);
else
{
printf("\n\t Empty Tree.");
return;
}
if(t->lptr!=NULL)
Preorder(t->lptr);
if(t->rptr!=NULL)
Preorder(t->rptr);
return;
}
void Postorder(struct node *t)
{
if(t==NULL)
{
printf("\n\t Empty Tree.");
return;
```

```
}
Postorder(t->lptr);
Postorder(t->rptr);
printf("%d  ",t->data);
return;
}
void Inorder(struct node *t)
{
if(t==NULL)
{
printf("\n\t Empty Tree.");
return;
}
if(t->lptr!=NULL)
Inorder(t->lptr);
printf("%d  ",t->data);
if(t->rptr!=NULL)
Inorder(t->rptr);
return;
}
void Delete(struct node *h,int x)
{
int found;
char d;
```

```
struct node *cur,*parent,*pred,*suc,*q;

if(h->lptr==h)

{

printf("\n\t Empty Tree.");

return;

}

parent=h;

cur=h->lptr;

d='L';

found=0;

while(!found && cur!=NULL)

{

if(x==cur->data)

found=1;

else if(x<cur->data)

{

parent=cur;

cur=cur->lptr;

d='L';

}

else

{

parent=cur;

cur=cur->rptr;
```

```
d='R';

}

}

if(!found)

{

printf("\n\t Node is not found.");

return;

}

if(cur->lptr==NULL)

q=cur->rptr;

else

{

if(cur->rptr==NULL)

q=cur->lptr;

else

{

suc=cur->rptr;

if(suc->lptr==NULL)

{

suc->lptr=cur->lptr;

q=suc;

}

else

{
```

```
pred=cur->rptr;

suc=pred->lptr;

while(suc->lptr!=NULL)

{

pred=suc;

suc=pred->lptr;

}

pred->lptr=suc->rptr;

suc->lptr=cur->lptr;

suc->rptr=cur->rptr;

q=suc;

}

}

}

if(d=='L')

parent->lptr=q;

else

parent->rptr=q;

printf("\n\t%d is Deleted.",x);

}
```

**Output:**

```
1.Insert a node in a Tree

2.Preorder Traversal(Recursively)

3.Postorder Traversal(Recursively)

4.Inorder Traversal(Recursively)

5.Delete a node from Binary Search Tree

6.Exit

Enter Choice: 1

Enter Element : 7

Inorder Traversal(Recursively)=

------------------------------------

7

------------------------------------


1.Insert a node in a Tree

2.Preorder Traversal(Recursively)

3.Postorder Traversal(Recursively)

4.Inorder Traversal(Recursively)

5.Delete a node from Binary Search Tree

6.Exit

Enter Choice: 1

Enter Element : 5

Inorder Traversal(Recursively)=
------------------------------------

5  7

------------------------------------

1.Insert a node in a Tree

2.Preorder Traversal(Recursively)

3.Postorder Traversal(Recursively)

4.Inorder Traversal(Recursively)

5.Delete a node from Binary Search Tree

6.Exit

Enter Choice: 1

Enter Element : 4

Inorder Traversal(Recursively)=

------------------------------------

4  5  7

------------------------------------


1.Insert a node in a Tree

2.Preorder Traversal(Recursively)

3.Postorder Traversal(Recursively)

4.Inorder Traversal(Recursively)

5.Delete a node from Binary Search Tree

6.Exit

Enter Choice: 1

Enter Element : 6
```

```
Inorder Traversal(Recursively)=

------------------------------------

4  5  6  7

------------------------------------

1.Insert a node in a Tree

2.Preorder Traversal(Recursively)

3.Postorder Traversal(Recursively)

4.Inorder Traversal(Recursively)

5.Delete a node from Binary Search Tree

6.Exit

Enter Choice: 1

Enter Element : 9

Inorder Traversal(Recursively)=

------------------------------------

4  5  6  7  9

------------------------------------

1.Insert a node in a Tree

2.Preorder Traversal(Recursively)

3.Postorder Traversal(Recursively)

4.Inorder Traversal(Recursively)

5.Delete a node from Binary Search Tree

6.Exit

Enter Choice: 1

Enter Element : 8
```

```
Inorder Traversal(Recursively)=


------------------------------------
4  5  6  7  8  9
-------------------------------------
1.Insert a node in a Tree
2.Preorder Traversal(Recursively)
3.Postorder Traversal(Recursively)
4.Inorder Traversal(Recursively)
5.Delete a node from Binary Search Tree
6.Exit
Enter Choice: 1
Enter Element : 11
Inorder Traversal(Recursively)=
-------------------------------------
4  5  6  7  8  9  11
-------------------------------------
1.Insert a node in a Tree
2.Preorder Traversal(Recursively)
3.Postorder Traversal(Recursively)
4.Inorder Traversal(Recursively)
5.Delete a node from Binary Search Tree
6.Exit
Enter Choice: 2
```

```
Preorder Traversal(Recursively):

------------------------------------

7  5  4  6  9  8  11

------------------------------------

1.Insert a node in a Tree

2.Preorder Traversal(Recursively)

3.Postorder Traversal(Recursively)

4.Inorder Traversal(Recursively)

5.Delete a node from Binary Search Tree

6.Exit

Enter Choice: 3

Postorder Traversal(Recursively):

------------------------------------

Empty Tree.

Empty Tree.4

Empty Tree.

Empty Tree.6  5

Empty Tree.

Empty Tree.8

Empty Tree.

Empty Tree.11  9  7

------------------------------------

1.Insert a node in a Tree

2.Preorder Traversal(Recursively)
```

```
3.Postorder Traversal(Recursively)

4.Inorder Traversal(Recursively)

5.Delete a node from Binary Search Tree

6.Exit

Enter Choice: 4

Inorder Traversal(Recursively):

------------------------------------

4  5  6  7  8  9  11

------------------------------------

1.Insert a node in a Tree

2.Preorder Traversal(Recursively)

3.Postorder Traversal(Recursively)

4.Inorder Traversal(Recursively)

5.Delete a node from Binary Search Tree

6.Exit

Enter Choice: 6
```

**B) Write a non recursive C program for traversing a binary tree in preorder, inorder and postorder.**

## Aim:

A c program for binary tree traversing using non recursion

## Description:

To process data stored in the tree data structure, we need to traverse each node of the tree. The process to visit all nodes of a tree is called tree traversal. It is just like searching the tree and accessing each node only once. During the visit of a tree node, we can perform various operations with the data stored in the node.
We access data elements in sequential order in the case of linear data structures. But in binary tree data structures, there can be many different ways to access data elements. Starting from the root node, there are two directions to go: either we can go left via the left pointer or we can go right via the right pointer.

## Algorithm:

**//inorder**

```
Step 1: Repeat Steps 2 to 4 while TREE != NULL
Step 2:     INORDER(TREE -> LEFT)
Step 3:     Write TREE -> DATA
Step 4:     INORDER(TREE -> RIGHT)
[END OF LOOP]
Step 5: END
```

**//preorder**

```
Step 1: Repeat Steps 2 to 4 while TREE != NULL
Step 2:     Write TREE -> DATA
Step 3:     PREORDER(TREE -> LEFT)
Step 4:     PREORDER(TREE -> RIGHT)
[END OF LOOP]
Step 5: END
```

**//postorder**

```
Step 1: Repeat Steps 2 to 4 while TREE != NULL
Step 2:     POSTORDER(TREE -> LEFT)
Step 3:     POSTORDER(TREE -> RIGHT)
Step 4:     Write TREE -> DATA
[END OF LOOP]
Step 5: END
```

**Program:**

```
/*  C Program for Inorder Preorder Postorder traversal of Binary Tree */

#include<stdio.h>

#include<stdlib.h>

#define MAX 50

struct node

{

struct node *lchild;

int info;

struct node *rchild;

};

struct node *insert_nrec(struct node *root, int ikey );

void nrec_pre(struct node *root);

void nrec_in(struct node *root);

void nrec_post(struct node *root);

void display(struct node *ptr,int level);

struct node *queue[MAX];

int front=-1,rear=-1;

void insert_queue(struct node *item);

struct node *del_queue();

int queue_empty();

struct node *stack[MAX];

int top=-1;

void push_stack(struct node *item);
```

```
struct node *pop_stack();

int stack_empty();

int main( )

{

struct node *root=NULL, *ptr;

int choice,k;

while(1)

{

printf("\n");

printf("1.Insert\n");

printf("2.Display\n");

printf("3.Preorder Traversal\n");

printf("4.Inorder Traversal\n");

printf("5.Postorder Traversal\n");

printf("6.Quit\n");

printf("\nEnter your choice : ");

scanf("%d",&choice);

switch(choice)

{

case 1:

printf("\nEnter the key to be inserted : ");

scanf("%d",&k);

root = insert_nrec(root, k);

break;
case 2:

printf("\n");

display(root,0);

printf("\n");

break;

case 3:

nrec_pre(root);

break;

case 4:

nrec_in(root);

break;

case 5:

nrec_post(root);

break;

case 6:

exit(1);

default:

printf("\nWrong choice\n");

}/*End of switch*/

}/*End of while */

return 0;

}/*End of main( )*/

struct node *insert_nrec(struct node *root, int ikey)

{
```

```
struct node *tmp,*par,*ptr;

ptr = root;

par = NULL;

while( ptr!=NULL)

{

par = ptr;

if(ikey < ptr->info)

ptr = ptr->lchild;

else if( ikey > ptr->info )

ptr = ptr->rchild;

else

{

printf("\nDuplicate key");

return root;

}

}

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

tmp->info=ikey;

tmp->lchild=NULL;

tmp->rchild=NULL;

if(par==NULL)

root=tmp;

else if( ikey < par->info )

par->lchild=tmp;
```

```
else
par->rchild=tmp;
return root;
}/*End of insert_nrec( )*/
void nrec_pre(struct node *root)
{
struct node *ptr = root;
if( ptr==NULL )
{
printf("Tree is empty\n");
return;
}
push_stack(ptr);
while( !stack_empty() )
{
ptr = pop_stack();
printf("%d  ",ptr->info);
if(ptr->rchild!=NULL)
push_stack(ptr->rchild);
if(ptr->lchild!=NULL)
push_stack(ptr->lchild);
}
printf("\n");
}/*End of nrec_pre*/
```

```
void nrec_in(struct node *root)
{
struct node *ptr=root;
if( ptr==NULL )
{
printf("Tree is empty\n");
return;
}
while(1)
{
while(ptr->lchild!=NULL )
{
push_stack(ptr);
ptr = ptr->lchild;
}
while( ptr->rchild==NULL )
{
printf("%d  ",ptr->info);
if(stack_empty())
return;
ptr = pop_stack();
}
printf("%d  ",ptr->info);
ptr = ptr->rchild;
```

```
}
printf("\n");
}/*End of nrec_in( )*/
void nrec_post(struct node *root)
{
struct node *ptr = root;
struct node *q;
if( ptr==NULL )
{
printf("Tree is empty\n");
return;
}
q = root;
while(1)
{
while(ptr->lchild!=NULL)
{
push_stack(ptr);
ptr=ptr->lchild;
}
while( ptr->rchild==NULL || ptr->rchild==q )
{
printf("%d  ",ptr->info);
q = ptr;
if( stack_empty() )
return;
ptr = pop_stack();
}
push_stack(ptr);
ptr = ptr->rchild;
}
printf("\n");
}/*End of nrec_post( )*/
/*Functions for implementation of queue*/
void insert_queue(struct node *item)
{
if(rear==MAX-1)
{
printf("Queue Overflow\n");
return;
}
if(front==-1)  /*If queue is initially empty*/
front=0;
rear=rear+1;
queue[rear]=item ;
}/*End of insert()*/
struct node *del_queue()
{
```

```
struct node *item;
if(front==-1 || front==rear+1)
{
printf("Queue Underflow\n");
return 0;
}
item=queue[front];
front=front+1;
return item;
}/*End of del_queue()*/
int queue_empty()
{
if(front==-1 || front==rear+1)
return 1;
else
return 0;
}
/*Functions for implementation of stack*/
void push_stack(struct node *item)
{
if(top==(MAX-1))
{
printf("Stack Overflow\n");
return;
}
top=top+1;
stack[top]=item;
}/*End of push_stack()*/
struct node *pop_stack()
{
struct node *item;
if(top==-1)
{
printf("Stack Underflow....\n");
exit(1);
}
item=stack[top];
top=top-1;
return item;
}/*End of pop_stack()*/
int stack_empty()
{
if(top==-1)
return 1;
else
return 0;
} /*End of stack_empty*/
```

```
void display(struct node *ptr,int level)
{
int i;
if(ptr == NULL )/*Base Case*/
return;
else
{
display(ptr->rchild, level+1);
printf("\n");
for (i=0; i<level; i++)
printf("    ");
printf("%d", ptr->info);
display(ptr->lchild, level+1);
}
}/*End of display()*/
```

**Output:**

```
/*  C Program for Inorder Preorder Postorder traversal of Binary Tree */
1.Insert
2.Display
3.Preorder Traversal
4.Inorder Traversal
5.Postorder Traversal
```

```
6.Quit

Enter your choice : 1

Enter the key to be inserted : 7

1.Insert

2.Display

3.Preorder Traversal

4.Inorder Traversal

5.Postorder Traversal

6.Quit

Enter your choice : 1

Enter the key to be inserted : 5

1.Insert

2.Display

3.Preorder Traversal

4.Inorder Traversal

5.Postorder Traversal

6.Quit

Enter your choice : 1

Enter the key to be inserted : 6

1.Insert

2.Display

3.Preorder Traversal

4.Inorder Traversal

5.Postorder Traversal
```

```
6.Quit

Enter your choice : 1

Enter the key to be inserted : 4

1.Insert

2.Display

3.Preorder Traversal

4.Inorder Traversal

5.Postorder Traversal

6.Quit

Enter your choice : 1

Enter the key to be inserted : 9

1.Insert

2.Display

3.Preorder Traversal

4.Inorder Traversal

5.Postorder Traversal

6.Quit

Enter your choice : 1

Enter the key to be inserted : 8

1.Insert

2.Display

3.Preorder Traversal

4.Inorder Traversal

5.Postorder Traversal
```

```
6.Quit

Enter your choice : 1

Enter the key to be inserted : 11

1.Insert

2.Display

3.Preorder Traversal

4.Inorder Traversal

5.Postorder Traversal

6.Quit

Enter your choice : 2

11

9

8

7

6

5

4

1.Insert

2.Display

3.Preorder Traversal

4.Inorder Traversal

5.Postorder Traversal

6.Quit

Enter your choice : 3
```

```
7  5  4  6  9  8  11

1.Insert

2.Display

3.Preorder Traversal

4.Inorder Traversal

5.Postorder Traversal

6.Quit

Enter your choice : 4

4  5  6  7  8  9  11

1.Insert

2.Display

3.Preorder Traversal

4.Inorder Traversal

5.Postorder Traversal

6.Quit

Enter your choice : 5

4  6  5  8  11  9  7

1.Insert

2.Display

3.Preorder Traversal

4.Inorder Traversal

5.Postorder Traversal

6.Quit
```

```
Enter your choice : 6

Process returned 1
```

## Experiment 12:

**A) Write a C program to implement Prims‟ algorithm.**

Aim:

A c program to implement prims algorithm

### Description:

Prim's Algorithm is a greedy algorithm that is used to find the minimum spanning tree from a graph. Prim's algorithm finds the subset of edges that includes every vertex of the graph such that the sum of the weights of the edges can be minimized.

Prim's algorithm starts with the single node and explores all the adjacent nodes with all the connecting edges at every step. The edges with the minimal weights causing no cycles in the graph got selected.

### Algorithm:

```
Algorithm PRIM_MST(G, n)
// Description : Find MST of graph G using Prim’s algorithm
// Input : Weighted connected graph G and number of vertices n
// Output : Minimum spanning tree MST and weight of it i.e. cost
cost ← 0
// Initialize tree nodes
for i ← 0 to n – 1 do
MST[i] ← 0
end
MST[0] ← 1    // 1 is the initial vertex
for k ← 1 to n do
min_dist ← ∞
for i ← 1 to n – 1 do
for j ← 1 to n – 1 do
if G[i, j] AND ((MST[i] AND ¬MST[j]) OR (¬MST[i] AND MST[j])) then
if G[i, j] < min_dist then
min_dist ← G[i, j]
u ← i
v ← j
end
end
end
end
print (u, v, min_dist)
MST[u] ← MST[v] ← 1
```

```
cost ← cost + min_dist
end
print (“Total cost = ”, cost)
```

**Program:**

```
#include <stdio.h>
#include <limits.h>
#define vertices 5  /*Define the number of vertices in the graph*/
/* create minimum_key() method for finding the vertex that has minimum key-value
and that is not added in MST yet */
int minimum_key(int k[], int mst[])
{
int minimum  = INT_MAX, min,i;

/*iterate over all vertices to find the vertex with minimum key-value*/
for (i = 0; i < vertices; i++)
if (mst[i] == 0 && k[i] < minimum )
minimum = k[i], min = i;
return min;
}
/* create prim() method for constructing and printing the MST.
The g[vertices][vertices] is an adjacency matrix that defines the graph for MST.*/
void prim(int g[vertices][vertices])
{
/* create array of size equal to total number of vertices for storing the MST*/
int parent[vertices];
/* create k[vertices] array for selecting an edge having minimum weight*/
int k[vertices];
int mst[vertices];
int i, count,edge,v; /*Here 'v' is the vertex*/
for (i = 0; i < vertices; i++)
{
k[i] = INT_MAX;
mst[i] = 0;
}
k[0] = 0; /*It select as first vertex*/
parent[0] = -1;   /* set first value of parent[] array to -1 to make it root of MST*/
for (count = 0; count < vertices-1; count++)
{
/*select the vertex having minimum key and that is not added in the MST yet from the
set of vertices*/
edge = minimum_key(k, mst);
mst[edge] = 1;
for (v = 0; v < vertices; v++)
{
if (g[edge][v] && mst[v] == 0 && g[edge][v] <  k[v])
{
```

```
parent[v]  = edge, k[v] = g[edge][v];
}
}
}
/*Print the constructed Minimum spanning tree*/
printf("\n Edge \t  Weight\n");
for (i = 1; i < vertices; i++)
printf(" %d <-> %d    %d \n", parent[i], i, g[i][parent[i]]);

}
int main()
{
int g[vertices][vertices] = {{0, 0, 3, 0, 0},
{0, 0, 10, 4, 0},
{3, 10, 0, 2, 6},
{0, 4, 2, 0, 1},
{0, 0, 6, 1, 0},
};
prim(g);
return 0;
}
```

**Output:**

```
Edge    Weight
3 <-> 1   4
0 <-> 2   3
2 <-> 3   2
3 <-> 4   1
```

**B) Write a C program to implement Kruskal‟s algorithm.**

## Aim:

A c program to implement kruskal’s algorithm

## Description:

Kruskal's Algorithm is used to find the minimum spanning tree for a connected weighted graph. The main target of the algorithm is to find the subset of edges by using which we can traverse every vertex of the graph. It follows the greedy approach that finds an optimum solution at every stage instead of focusing on a global optimum.

## Algorithm:

```
Step 1: Create a forest F in such a way that every vertex of the graph is a separate tree.
Step 2: Create a set E that contains all the edges of the graph.
Step 3: Repeat Steps 4 and 5 while E is NOT EMPTY and F is not spanning
Step 4: Remove an edge from E with minimum weight
Step 5: IF the edge obtained in Step 4 connects two different trees, then add it to the forest F
(for combining two trees into one tree).
ELSE
Discard the edge
Step 6: END
```

Program:

```c
#include<stdio.h>

#define MAX 30

typedef struct edge
{
int u,v,w;
}edge;

typedef struct edgelist
{
edge data[MAX];
int n;
}edgelist;

edgelist elist;

int G[MAX][MAX],n;
```

```
edgelist spanlist;

void kruskal();
int find(int belongs[],int vertexno);
void union1(int belongs[],int c1,int c2);
void sort();
void print();

void main()
{
int i,j,total_cost;
printf("\nEnter number of vertices:");
scanf("%d",&n);
printf("\nEnter the adjacency matrix:\n");
for(i=0;i<n;i++)
for(j=0;j<n;j++)
scanf("%d",&G[i][j]);
kruskal();
print();
}

void kruskal()
{
int belongs[MAX],i,j,cno1,cno2;
elist.n=0;

for(i=1;i<n;i++)
for(j=0;j<i;j++)
{
if(G[i][j]!=0)
{
elist.data[elist.n].u=i;
elist.data[elist.n].v=j;
elist.data[elist.n].w=G[i][j];
elist.n++;
}
}

sort();
for(i=0;i<n;i++)
belongs[i]=i;
spanlist.n=0;
for(i=0;i<elist.n;i++)
{
```

```
cno1=find(belongs,elist.data[i].u);
cno2=find(belongs,elist.data[i].v);
if(cno1!=cno2)
{
spanlist.data[spanlist.n]=elist.data[i];
spanlist.n=spanlist.n+1;
union1(belongs,cno1,cno2);
}
}
}

int find(int belongs[],int vertexno)
{
return(belongs[vertexno]);
}

void union1(int belongs[],int c1,int c2)
{
int i;
for(i=0;i<n;i++)
if(belongs[i]==c2)
belongs[i]=c1;
}

void sort()
{
int i,j;
edge temp;
for(i=1;i<elist.n;i++)
for(j=0;j<elist.n-1;j++)
if(elist.data[j].w>elist.data[j+1].w)
{
temp=elist.data[j];
elist.data[j]=elist.data[j+1];
elist.data[j+1]=temp;
}
}

void print()
{
int i,cost=0;
for(i=0;i<spanlist.n;i++)
{
printf("\n%d\t%d\t%d",spanlist.data[i].u,spanlist.data[i].v,spanlist.data[i].w);
```

```
cost=cost+spanlist.data[i].w;
}

printf("\n\nCost of the spanning tree=%d",cost);
}
```

**Output:**

```
Enter number of vertices:6

Enter the adjacency matrix:
0 3 1 6 0 0
3 0 5 0 3 0
1 5 0 5 6 4
6 0 5 0 0 2
0 3 6
0 0 4 2 6 0
2 0 1

2       0       1
5       0       2
5       3       2
1       0       3
4       1       3

Cost of the spanning tree=11
```

# Experiment 13:

**Implementation of Hash table using double hashing as collision resolution function**

## Aim:

A c program for implementation of hash table

## Description:

The basic idea behind hashing is to distribute key/value pairs across an array of placeholders or "buckets" in the hash table.

A hash table is typically an array of linked lists. When you want to insert a key/value pair, you first need to use the hash function to map the key to an index in the hash table. Given a key, the hash function can suggest an index where the value can be found or stored:

index = f(key, array_size)

## Algorithm:

1. **Create a Hash Table**
2. **Hashtable<Integer, String> ht = new Hashtable<Integer, String>();**
3. **Insert values in hash table using put(key,value)**
4. **ht.put(key, value);**
5. **Get values from hash table using get(key)**
6. **ht.get(key);**

**Program:**

```
 #include <stdio.h>
 #include <string.h>
 #include <stdlib.h>
 #include <stdbool.h>
#define SIZE 20
struct DataItem {
 int data;
 int key;
 };
 struct DataItem* hashArray[SIZE];
 struct DataItem* dummyItem;
 struct DataItem* item;
 int hashCode(int key) {
 return key % SIZE;
 }
 struct DataItem *search(int key) {
 //get the hash
 int hashIndex = hashCode(key);
 //move in array until an empty
 while(hashArray[hashIndex] != NULL) {
 if(hashArray[hashIndex]->key == key)
 return hashArray[hashIndex];
 //go to next cell
 ++hashIndex;
 //wrap around the table
 hashIndex %= SIZE;
```

```
}
return NULL;
}
void insert(int key,int data) {
struct DataItem *item = (struct DataItem*) malloc(sizeof(struct DataItem));
item->data = data;
item->key = key;
//get the hash
int hashIndex = hashCode(key);
//move in array until an empty or deleted cell
while(hashArray[hashIndex] != NULL && hashArray[hashIndex]->key != -1) {
//go to next cell
++hashIndex;
//wrap around the table
hashIndex %= SIZE;
}
hashArray[hashIndex] = item;
}
struct DataItem* delete(struct DataItem* item) {
int key = item->key;
//get the hash
int hashIndex = hashCode(key);
//move in array until an empty
while(hashArray[hashIndex] != NULL) {
if(hashArray[hashIndex]->key == key) {
struct DataItem* temp = hashArray[hashIndex];
//assign a dummy item at deleted position
hashArray[hashIndex] = dummyItem;
return temp;
}
//go to next cell
++hashIndex;
//wrap around the table
hashIndex %= SIZE;
}
return NULL;
}
void display() {
int i = 0;
for(i = 0; i<SIZE; i++) {
if(hashArray[i] != NULL)
printf(" (%d,%d)",hashArray[i]->key,hashArray[i]->data);
else
printf(" ~~ ");
}
printf("\n");
}
int main() {
dummyItem = (struct DataItem*) malloc(sizeof(struct DataItem));
dummyItem->data = -1;
dummyItem->key = -1;
insert(1, 20);
insert(2, 70);
insert(42, 80);
insert(4, 25);
insert(12, 44);
insert(14, 32);
insert(17, 11);
insert(13, 78);
insert(37, 97);
display();
item = search(37);
if(item != NULL) {
printf("Element found: %d\n", item->data);
} else {
printf("Element not found\n");
}
delete(item);
item = search(37);
if(item != NULL) {
printf("Element found: %d\n", item->data);
} else {
printf("Element not found\n");
}
}
```

**Output:**

```
~~  (1,20) (2,70) (42,80) (4,25) ~~  ~~  ~~  ~~  ~~  ~~  ~~  (12,44) (13,78) (14,32) ~~  ~~
(17,11) (37,97) ~~
Element found: 97
Element not found
```

## Experiment 14:

**Implementation of Binary Search trees- Insertion and deletion.**

## Aim:

A c program for implementation of binary search tree

## Description:

A binary tree is a tree in which no node can have more than two children. In a binary search tree ,for every node, X, in the tree, the values of all the keys in its left sub tree are *smaller than* the key value of X, and the values of all the keys in its right sub tree are *larger than* the key value of X. The basic operations on a binary search tree take time proportional to the height of the tree.

In the linked list implementation of binary search trees: Each element is represented by node with two link fields and a data field. Each connecting line (or edge) in a binary tree drawing will be represented by a link field. A leaf node has a leftChild and rightChild link of NULL. Root node will be pointed to by a pointer variable.

## Algorithm:

```
//insertion

If node == NULL

return createNode(data)

if (data < node->data)

node->left  = insert(node->left, data);

else if (data > node->data)

node->right = insert(node->right, data);

return node;


//deletion

struct node* delete(struct node *root, int x)

{

if(root==NULL)
```

```
return NULL;
if (x>root->data)
root->right_child = delete(root->right_child, x);
else if(x<root->data)
root->left_child = delete(root->left_child, x);
else
{
//No Children
if(root->left_child==NULL && root->right_child==NULL)
{
free(root);
return NULL;
}
//One Child
else if(root->left_child==NULL || root->right_child==NULL)
{
struct node *temp;
if(root->left_child==NULL)
temp = root->right_child;
else
temp = root->left_child;
free(root);
return temp;
}
//Two Children
else
{
```

```
struct node *temp = find_minimum(root->right_child);

root->data = temp->data;

root->right_child = delete(root->right_child, temp->data);

}

}

return root;

}
```

**Program:**

```
#include<stdlib.h>

#include<stdio.h>


struct bin_tree {

int data;

struct bin_tree * right, * left;

};

typedef struct bin_tree node;


void insert(node ** tree, int val)

{

node *temp = NULL;

if(!(*tree))

{

temp = (node *)malloc(sizeof(node));

temp->left = temp->right = NULL;

temp->data = val;

*tree = temp;

return;

}


if(val < (*tree)->data)

{

insert(&(*tree)->left, val);

}

else if(val > (*tree)->data)
```

```
{
insert(&(*tree)->right, val);
}}
void print_preorder(node * tree)
{
if (tree)
{
printf("%d\n",tree->data);
print_preorder(tree->left);
print_preorder(tree->right);
}}
void print_inorder(node * tree)
{
if (tree)
{
print_inorder(tree->left);
printf("%d\n",tree->data);
print_inorder(tree->right);
}
}
void print_postorder(node * tree)
{
if (tree)
{
print_postorder(tree->left);
print_postorder(tree->right);
printf("%d\n",tree->data);
```

```
}
}
void deltree(node * tree)
{
if (tree)
{
deltree(tree->left);
deltree(tree->right);
free(tree);
}
}
node* search(node ** tree, int val)
{
if(!(*tree))
{
return NULL;
}
if(val < (*tree)->data)
{
search(&((*tree)->left), val);
}
else if(val > (*tree)->data)
{
search(&((*tree)->right), val);
}
else if(val == (*tree)->data)
{
```

```
return *tree;
}
}
void main()
{
node *root;
node *tmp;
//int i;
root = NULL;
/* Inserting nodes into tree */
insert(&root, 9);
insert(&root, 4);
insert(&root, 15);
insert(&root, 6);
insert(&root, 12);
insert(&root, 17);
insert(&root, 2);
/* Printing nodes of tree */
printf("Pre Order Display\n");
print_preorder(root);
printf("In Order Display\n");
print_inorder(root);
printf("Post Order Display\n");
print_postorder(root);
/* Search node into tree */
tmp = search(&root, 4);
if (tmp)
```

```
{
printf("Searched node=%d\n", tmp->data);
}
else
{
printf("Data Not found in tree.\n");
}
/* Deleting all nodes of tree */
deltree(root);
}
```

**Output:**

```
Pre Order Display
9
4
2
6
15
12
17
In Order Display
2
4
6
9
12
15
17
```

```
Post Order Display

2

6

4

12

17

15

9

Searched node=4
```

## Experiment 15:

**A) Write C program that implement Bubble sort, to sort a given list of integers in ascending order**

### Aim:

A c program to implement bubble sort

### Description:

Bubble sort is a sorting algorithm that compares two adjacent elements and swaps them until they are not in the intended order.

Just like the movement of air bubbles in the water that rise up to the surface, each element of the array move to the end in each iteration. Therefore, it is called a bubble sort.

### Algorithm:

```
begin BubbleSort(list)

for all elements of list
if list[i] > list[i+1]
swap(list[i], list[i+1])
end if
end for

return list

end BubbleSort
```

program:

```
/* Bubble sort code */
#include <stdio.h>

int main()
{
int array[100], n, c, d, swap;

printf("Enter number of elements\n");
scanf("%d", &n);

printf("Enter %d integers\n", n);

for (c = 0; c < n; c++)
scanf("%d", &array[c]);
for (c = 0 ; c < n - 1; c++)
{
for (d = 0 ; d < n - c - 1; d++)
{
if (array[d] > array[d+1]) /* For decreasing order use '<' instead of '>' */
{
swap       = array[d];
array[d]   = array[d+1];
array[d+1] = swap;
}
}
}

printf("Sorted list in ascending order:\n");

for (c = 0; c < n; c++)
printf("%d\n", array[c]);

return 0;
}
```

**Output:**

```
Enter number of elements
6
Enter 6 integers
2 -4 7 8 4 7
Sorted list in ascending order:
-4
2
4
7
7
8
```

**B) Write C program that implement Quick sort, to sort a given list of integers in ascending order**

## Aim:

A c program to implement quick sort

## Description:

Quick sort is a highly efficient sorting algorithm and is based on partitioning of array of data into smaller arrays. A large array is partitioned into two arrays one of which holds values smaller than the specified value, say pivot, based on which the partition is made and another array holds values greater than the pivot value.

Quicksort partitions an array and then calls itself recursively twice to sort the two resulting subarrays. This algorithm is quite efficient for large-sized data sets as its average and worst-case complexity are O($n^2$), respectively.

## Algorithm:

```
procedure quickSort(left, right)

  if right-left <= 0
    return
  else
    pivot = A[right]
    partition = partitionFunc(left, right, pivot)
    quickSort(left,partition-1)
    quickSort(partition+1,right)
  end if

end procedure
```

**Program:**

```
#include<stdio.h>
void quicksort(int number[25],int first,int last){
   int i, j, pivot, temp;
  if(first<last){
      pivot=first;
      i=first;
      j=last;

      while(i<j){
         while(number[i]<=number[pivot]&&i<last)
            i++;
         while(number[j]>number[pivot])
            j--;
         if(i<j){
            temp=number[i];
            number[i]=number[j];
            number[j]=temp;
         }
      }

      temp=number[pivot];
      number[pivot]=number[j];
      number[j]=temp;
      quicksort(number,first,j-1);
      quicksort(number,j+1,last);

   }
}

int main(){
   int i, count, number[25];

   printf("How many elements are u going to enter?: ");
   scanf("%d",&count);

   printf("Enter %d elements: ", count);
   for(i=0;i<count;i++)
      scanf("%d",&number[i]);

   quicksort(number,0,count-1);

   printf("Order of Sorted elements: ");
   for(i=0;i<count;i++)
      printf(" %d",number[i]);

   return 0;
}
```

**Output:**

```
How many elements are u going to enter?: 5
Enter 5 elements: 88 0 -9 78 27
Order of Sorted elements:  -9 0 27 78 88
```

C) **Write C program that implement merge sort, to sort a given list of integers in ascending order**

## Aim:

A c program to implement merge sort

## Description:

Merge sort is similar to the quick sort algorithm as it uses the divide and conquer approach to sort the elements. It is one of the most popular and efficient sorting algorithm. It divides the given list into two equal halves, calls itself for the two halves and then merges the two sorted halves. We have to define the **merge()** function to perform the merging.

The sub-lists are divided again and again into halves until the list cannot be divided further. Then we combine the pair of one element lists into two-element lists, sorting them in the process. The sorted two-element pairs is merged into the four-element lists, and so on until we get the sorted list.

## Algorithm:

```
MERGE_SORT(arr, beg, end)

if beg < end
set mid = (beg + end)/2
MERGE_SORT(arr, beg, mid)
MERGE_SORT(arr, mid + 1, end)
MERGE (arr, beg, mid, end)
end of if

END MERGE_SORT
```

**Program:**

```
#include <stdio.h>

#define max 10

int a[11] = { 10, 14, 19, 26, 27, 31, 33, 35, 42, 44, 0 };
int b[10];

void merging(int low, int mid, int high) {
   int l1, l2, i;

   for(l1 = low, l2 = mid + 1, i = low; l1 <= mid && l2 <= high; i++) {
      if(a[l1] <= a[l2])
         b[i] = a[l1++];
      else
         b[i] = a[l2++];
   }

   while(l1 <= mid)
      b[i++] = a[l1++];

   while(l2 <= high)
      b[i++] = a[l2++];

   for(i = low; i <= high; i++)
      a[i] = b[i];
}

void sort(int low, int high) {
   int mid;

   if(low < high) {
      mid = (low + high) / 2;
      sort(low, mid);
      sort(mid+1, high);
      merging(low, mid, high);
   } else {
      return;
   }
}

int main() {
   int i;
   printf("List before sorting\n");

   for(i = 0; i <= max; i++)
      printf("%d ", a[i]);

   sort(0, max);

   printf("\nList after sorting\n");

   for(i = 0; i <= max; i++)
      printf("%d ", a[i]);
}
```

**Output:**

```
List before sorting
10 14 19 26 27 31 33 35 42 44 0
List after sorting
0 10 14 19 26 27 31 33 35 42 44
```