# VIDYASAGAR UNIVERSITY

**Department of Computer Science & MCA**

**Data Structures with Algorithms Lab (MCA-108)**

**Dr. Soamdeep Singha**
Assistant Professor, Dept. of Computer Science,
Vidyasagar University-721102

## Program List

1. Write a C program to Implement the following searching techniques
a. Linear Search b. Binary Search.

2. Write a C program to implement the following sorting algorithms using user defined functions:
a. Bubble sort (Ascending order)
b. Selection sort (Descending order).

3. Write a C Program implement STACK with the following operations
a. Push an Element on to Stack
b. Pop an Element from Stack

4. Implement a Program in C for converting an Infix Expression to Postfix Expression.

5. Implement a Program in C for evaluating a Postfix Expression.

6. Write a C program to simulate the working of a singly linked list providing the following operations: a. Display & Insert
b. Delete from the beginning/end
c. Delete a given element

7. Obtain the Topological ordering of vertices in a given graph with the help of a C programming.

8. Check whether a given graph is connected or not using DFS method using C programming.

9. From a given vertex in a weighted connected graph, find shortest paths to other vertices Using Dijkstra's algorithm (C programming)

10. Find Minimum Cost Spanning Tree of a given undirected graph using Kruskal's algorithm ( C programming)

11. Write a program in C to create a minimum spanning tree using Prim's algorithm, etc.

12. Write a C program to implement Breadth First search of any given graph.

13. **Write a C program to implement Depth first search using linked representation of graph.**

14. **Write a C program to solve a given 0/1 knapsack problem.**

15. **Implement Inorder, Preorder and Post Order traversal of a binary tree using both recursive and non-recursive way. ( C programming)**

16. **Develop a C program to implement to create an AVL tree along with insertion and deletion operations.**

17. **Write a menu driven program that implements a Heap (Max or Min) for the following operations. Insert, Delete.**

18. **Develop a program to represent a polynomial. Also write methods for adding and subtracting two polynomials.**

19. **Using a C program, find the minimum number of multiplications needed to multiply a chain of matrices using dynamic programming approach.**

20. **Find the longest common subsequence from two given strings by writing a program using dynamic programming approach.**

21. **Using a program show how the collision resolution (linear probing) works in hashing.**

# Program 1

**1. Write a C program to Implement the following searching techniques**

**a. Linear Search b. Binary Search.**

## a. Linear Search

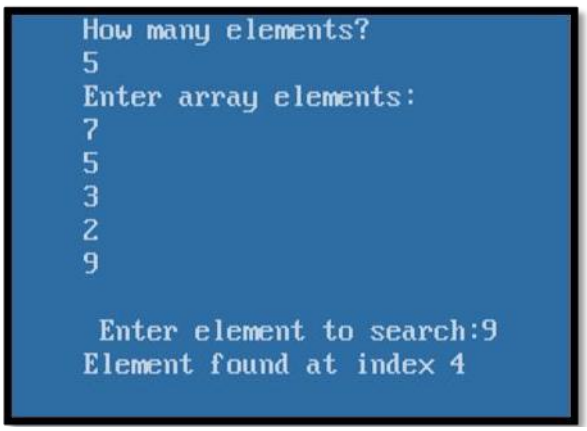
**What is a Linear Search?**

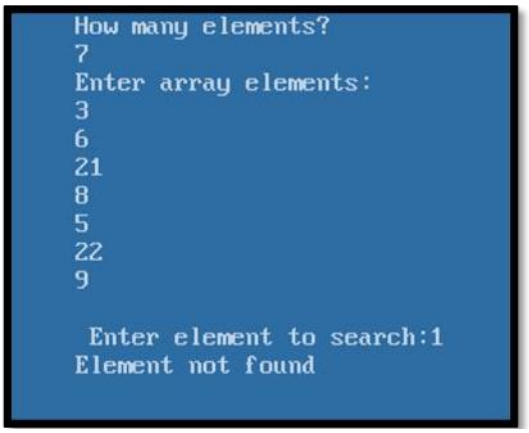
**A linear search, also known as a sequential search, is a method of finding an element within a list. It checks each element of the list sequentially until a match is found or the whole list has been searched.**

```
#include<stdio.h>
 int main ()
{
   int a[20],i,x,n;
   clrscr();
   printf("How many elements?\n");
   scanf("%d",&n);
   printf("Enter %d elements:\n");
   for(i=0;i<n;++i)
   scanf("%d",&a[i]);
   printf("Enter element to search:\n");
   scanf("%d",&x);
   for(i=0;i<n;++i)
         if(a[i]==x)
            break;

   if(i<n)
         printf("Element found at index %d",i);
else
         printf("Element not found");
getch();
return 0;
}
```

**OUTPUT**




**What is a Binary Search?**

- A Binary Search is a sorting algorithm, that is used to search an element
- in a sorted array.
- A binary search technique works only on a sorted array, so an array must be sorted to apply binary search on the array.
- It is a searching technique that is better than the linear search technique as the number of iterations decreases in the binary search.

**1.b.Binary Search**

```
#include<stdio.h>
#include<conio.h>
int main()
{
        int i, low, high, mid, n, key, array[100];
        clrscr();
        printf("Enter number of elements \n");
        scanf("%d",&n);
        printf("Enter %d integers in ascending order\n", n);
        for(i = 0; i < n; i++)
        {
        scanf("%d",&array[i]);
        }
        printf("Enter value to find\n");
```

```
        scanf("%d", &key);
        low = 0;       high =n - 1;
        mid = (low+high)/2;
        while (low <= high)
        {
                if(array[mid] < key)
                        low = mid + 1;
                else if (array[mid] == key)
                {
                        printf("%d found at location %d\n", key, mid);
                        break;
                }
                else
                        high = mid - 1;
        mid = (low + high)/2;
        }
        if(low > high)
        printf("Not found! %d isn't present in the list\n", key);
        getch();
        return 0;
}
```

## OUTPUT

```
Enter number of elements
10
Enter 10 integers in ascending order
1
2
3
4
5
6
7
8
9
10
Enter value to find
5
5 found at location 4
```

```
Enter number of elements
5
Enter 5 integers in ascending order
1
2
3
4
5
Enter value to find
7
Not found! 7 isn't present in the list
```

# Program 2

**2. Write a C program to implement the following sorting algorithms using user defined functions:**
**a. Bubble sort (Ascending order)**
**b. Selection sort (Descending order).**

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

void bubble(int a[], int n)
{
        int i, j, temp;
        for(i=0; i<n;i++)
        {
           for(j=0; j<n-i-1;j++)
              {
                 if(a[j]>a[j+1])
                {
                   temp=a[j];
                   a[j]=a[j+1];
                   a[j+1]=temp;
                }
               }
        }
}

void selection(int a[], int n)
{
        int i, j, temp;
         for(i=0;i<n-1;i++)
                {
                        for(j=i+1;j<n;j++)
                        {
                                if(a[i]<a[j])
                                {
                                        temp=a[i];
                                        a[i]=a[j];
                                        a[j]=temp;
                                }
```

```
                }
            }
}

void main()
{
    int a[20],n,i, opt;
    clrscr();
    printf("Enter the size of the array:");
    scanf("%d",&n);
     for(;;)
    {
    printf("\n ***** Sorting *****\n");
    printf("\n 1. Selection Sort (Descending Order) \n 2. Bubble Sort
                (Ascending Order) \n 3. Exit ");
    printf("\n Enter your
    option:");  scanf("%d",&opt);
    switch(opt)
            {
                case 1: printf("\nEnter array
elements:\n");                          for(i=0;i<n;i++)
                                        scanf("%d",&a[i]);
                                        selection(a, n);
                                printf("\nElements in Descending order:\n");
                                for(i=0;i<n;i++)
                                printf("%d \t", a[i]);
                                 break;
                case 2: printf("\nEnter array elements:\n");
                                 for(i=0;i<n;i++)
                                 scanf("%d",&a[i]);
                                 bubble(a, n);
                                 printf("\nElements in ascending order:\n");
                                 for(i=0;i<n;i++)
                                 printf("%d \t", a[i]);
                                 break;
                case 3:
                                 default: exit(0);
                }
            }
}
```

**OUTPUT**

```
Enter the size of the array : 3

***** Sorting *****
1. Selection Sort (Descending Order)
2. Bubble Sort  (Ascending Order)
3. Exit
Enter your option: 1
Enter array elements:  55 98 74 1 7
Elements in Descending order : 98 74 55 7 1

***** Sorting *****
1. Selection Sort (Descending Order)
2. Bubble Sort  (Ascending Order)
3. Exit
Enter your option:2
Enter array elements:  55 77 99 5 4
Elements in ascending order : 4 5 55 77 99


***** Sorting *****
1. Selection Sort (Descending Order)
2. Bubble Sort  (Ascending Order)
3. Exit

Enter your option:3
```

# Program 3

**3. Write a C Program implement STACK with the following operations**
**a. Push an Element on to Stack**
**b. Pop an Element from Stack**

**Algorithm:**
Data Structure: An Array with TOP as the pointer.

- Start
- Initialize the array of 10 elements and name it as stack
- Initialize other variables like top in the beginning of the program
- Provide the choice to the users for the different operations on stack like Push(insert),Pop(delete),Display and Exit
- If the choice= push then call the function push()
- If the choice= pop then call the function pop()
- If the choice= display then call the function display()
- If the choice= exit then exit from the program
- End

**Function push()**

- Check for the overflow condition of the stack
- if (TOP>= SIZE) then Print "Stack is full"
- If not overflow, increment the value of top
- Get the element to be inserted onto the stack from the user
- Assign it as the topmost value, stack[top]

**Function pop ()**

- Check for the underflow(empty) condition of the stack
- if TOP<0 Print "Stack is Empty"
- If not empty, Output the element to be deleted from the stack
- Decrement the value of top.

**Function display()**

- Display all the elements of the stack

```
#include<stdio.h>
int stack[10],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=10]:");
  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 STACKmax[100]:5**
**STACK OPERATIONS USING ARRAY**
**1.PUSH**
**2.POP**
**3.DISPLAY**
**4.EXIT**

**Enter the choice:1**
**Enter a value to be pushed:6**

**Enter the choice :1**
**Enter a value to be pushed: 2**

**Enter the choice :1**
**Enter a value to be pushed:4**

**Enter the choice :1**
**Enter a value to be pushed:1**

**Enter the choice :1**
**Enter a value to be pushed:5**

**Enter the choice :1**

**Stack is overflow.**

**Enter the choice : 3**

**The elements in the stack**
**5**
**1**
**4**
**2**
**6**
**Press next choice**

**Enter the choice : 2**
**The popped element is 5**

**Enter the choice: 3**

**The elements in the stack**
**1**
**4**
**2**
**6**

**Enter the choice : 2**
**The popped element is 1**

**Enter the choice : 2**
**The popped element is 4**

**Enter the choice : 2**
**The popped element is 2**

**Enter the choice : 2**
**The popped element is 6**

**Enter the choice : 2**
**Stack is underflow**

**Enter the choice : 3**
**Stack is Empty**

**Enter the choice: 6**
**Please Enter a valid choice (1/2/3/4)**

**Enter the choice: 4**

# Program 4

**4. Implement a Program in C for converting an Infix Expression to Postfix Expression**

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

char infix[50],postfix[50],stack[25];
int top=-1;
void convert_fun();
int precedence(char);
void push(char);
char pop();

void main()
{
	clrscr();
	printf("Enter the infix expression\n");
	scanf("%s",infix);
	convert_fun();
	printf("\n\t THE INFIX EXPRESSION :%s",infix);
	printf("\n\t THE POSTFIX EXPRESSION:%s",postfix);
	getch();
}
void convert_fun()
{
	int i=0,j=0;
	char c,tmp;
	push('#');
```

```
        while(i<strlen(infix))
        {
                c=infix[i++];
                switch(c)
                {
                        case '(' :push(c);
                                 break;
                        case ')' :tmp=pop();
                                        while(tmp!='(')
                                 {
                                        postfix[j++]=tmp;
                                        tmp=pop();
                                 }
                                 break;
                        case'+':
                        case'-':
                        case'*':
                        case'/':
                        case'^':
while(precedence(stack[top])>=precedence(c))
                                {
                                        tmp=pop();
                                        postfix[j++]=tmp;
                                }
                                push(c);
                                break;
                        default: postfix[j++]=c;
                                break;
                        }
                }
                while(top>0)
```

```
            {
                tmp=pop();
                postfix[j++]=tmp;
            }
        }
void push(char c)
{
stack[++top]=c;
}

char pop()
{
 return(stack[top--]);
}
int precedence(char c)
{
switch(c)
{
 case'^':return (3);
 case'*':
 case'/': return(2);
 case '+':
 case'-':return(1);
 case'(':
 case')':return(0);
 case'#':return(-1);
 }
}
```

**OUTPUT**

**1.Enter the infix expression a+b**
**The infix expression is a+b**
**The postfix expression is ab+**

**2.Enter the infix expression A + B * C + D**
**The infix expression is A + B * C + D**
**The postfix expression is A B C * + D +**

# Program 5

## 5. Implement a Program in C for evaluating an Postfix Expression

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
#include<math.h>
#include<ctype.h>
int top,s[100];
int cal(char sym,int op1,int op2)
{
switch(sym)
{
case'+' : return(op1+op2);
case'-' : return(op1-op2);
case'*' : return(op1*op2);
case'/' : return(op1/op2);
case'^' : return(pow(op1,op2));
}
return 0;
}
void main()
{
 int i,op1,op2;
char pf[100],sym;
clrscr();
top=-1;
printf("enter the postfix expression:");
```

```
gets(pf);
for(i=0;i<strlen(pf);i++)
    {
        sym=pf[i];
        if(isdigit (sym))
        {
            s[++top]=sym-'0';
        }
        else
        {
            op2=s[top--];
            op1=s[top--];
            s[++top]=cal(sym,op1,op2);
        }
    }

printf("\n value of %s is %d",pf,s[top--]);
getch();
}
```

## OUTPUT

**1.Enter the postfix expression: 3 4 * 2 5 * +**
**Value of 3 4 * 2 5 * + is 22**

**2. Enter the postfix expression: 2^3*5**
**Value of 2^3*5 is 22**

# Program 6

**6.Write a C program to simulate the working of a singly linked list providing the following operations:**

**a. Display & Insert**

**b. Delete from the beginning/end**

**c. Delete a given element**

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

void display();
void insert();
void delete_begin();
void delete_end();
void delete_pos();

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

struct node *start=NULL;

int main()
{
    int choice;
    clrscr();
    while(1){
```

```
            printf("\n MENU\n");
            printf(" 1.Insert\n");
            printf(" 2.Display\n");
            printf(" 3.Delete from beginning\n");
            printf(" 4.Delete from the end\n");
            printf(" 5.Delete from specified position\n");
            printf(" 6.Exit\n");
            printf(" --------------------------------- \n");
            printf("Enter your choice:\t");
            scanf("%d",&choice);
            switch(choice)
            {

                case 1:      insert();
                              break;
                case 2:    display();
                           break;
                case 3:    delete_begin();
                           break;
                case 4:    delete_end();
                           break;
                case 5:    delete_pos();
                           break;
                case 6:    exit(0);
                           break;
                default:    printf("\n Wrong Choice:\n");
                           break;
            }
        }
    }
```

```
void insert()
{
    struct node *temp;
    temp=(struct node *)malloc(sizeof(struct node));
    if(temp==NULL)
    {
        printf("\nOut of Memory Space:\n");
        return;
    }
    printf("\nEnter the data value for the node:\t" );
    scanf("%d",&temp->info);
    temp->next =NULL;
    if(start==NULL)
    {
        start=temp;
    }
    else
    {
        temp->next=start;
        start=temp;
    }
}


void display()
{
    struct node *ptr;
    if(start==NULL)
    {
        printf("\nList is empty:\n");
        return;
    }
```

```
        else
        {
                ptr=start;
                printf("\nThe List elements are:\n");
                while(ptr!=NULL)
                {
                        printf("%d\t",ptr->info );
                        ptr=ptr->next ;
                }
        }
}
void delete_begin()
{
        struct node *ptr;
        if(ptr==NULL)
        {
                printf("\nList is Empty:\n");
                return;
        }
        else
        {
                ptr=start;
                start=start->next ;
                printf("\nThe deleted element is :%d\t",ptr->info);
                free(ptr);
        }
}

void delete_end()
{
        struct node *temp,*ptr;
        if(start==NULL)
```

```
        {
                printf("\nList is Empty:");
                exit(0);
        }
        else if(start->next ==NULL)
        {
                ptr=start;
                start=NULL;
                printf("\nThe deleted element is:%d\t",ptr->info);
                free(ptr);
        }
        else
        {
                ptr=start;
                while(ptr->next!=NULL)
                {
                        temp=ptr;
                        ptr=ptr->next;
                }
                temp->next=NULL;
                printf("\nThe deleted element is:%d\t",ptr->info);
                free(ptr);
        }
}
void delete_pos()
{
        int i,pos;
        struct node *temp,*ptr;
        if(start==NULL)
        {
                printf("\nThe List is Empty:\n");
```

```
            exit(0);
        }
        else
        {
            printf("\nEnter the position of the node to be
                                                    deleted:\t");
            scanf("%d",&pos);
            if(pos==0)
            {
                ptr=start;
                start=start->next ;
                printf("\nThe deleted element is:%d\t",ptr->info
);
                free(ptr);
            }
            else
            {
                ptr=start;
                for(i=0;i<pos;i++) { temp=ptr; ptr=ptr->next ;
                    if(ptr==NULL)
                    {
                        printf("\nPosition not Found:\n");
                        return;
                    }
                    }
                temp->next =ptr->next ;
                printf("\nThe deleted element is:%d\t",ptr->info );
                free(ptr);
            }
        }
}
```

## OUTPUT

```
 MENU
        1.Insert
        2.Display
        3.Delete from beginning
        4.Delete from the end
        5.Delete from specified position
        6.Exit

Enter your choice: 1
Enter the data value for the node: 2

 MENU
        1.Insert
        2.Display
        3.Delete from beginning
        4.Delete from the end
        5.Delete from specified position
        6.Exit
Enter your choice: 1
Enter the data value for the node: 3

 MENU
        1.Insert
        2.Display
        3.Delete from beginning
        4.Delete from the end
        5.Delete from specified position
```

```
            6.Exit
Enter your choice: 1
Enter the data value for the node: 2

 MENU
            1.Insert
            2.Display
            3.Delete from beginning
            4.Delete from the end
            5.Delete from specified position
            6.Exit
Enter your choice: 1
Enter the data value for the node: 5

 MENU
            1.Insert
            2.Display
            3.Delete from beginning
            4.Delete from the end
            5.Delete from specified position
            6.Exit
Enter your choice: 1
Enter the data value for the node: 6

 MENU
            1.Insert
            2.Display
            3.Delete from beginning
            4.Delete from the end
            5.Delete from specified position
            6.Exit
```

**Enter your choice: 2**

**The list elements are:**
**6 5 2 3 2**

**MENU**
**1.Insert**
**2.Display**
**3.Delete from beginning**
**4.Delete from the end**
**5.Delete from specified position**
**6.Exit**
**Enter your choice: 3**
**The deleted element is**
**: 6**

**MENU**
**1.Insert**
**2.Display**
**3.Delete from beginning**
**4.Delete from the end**
**5.Delete from specified position**
**6.Exit**
**Enter your choice: 4**
**The deleted element is**
**: 2**

**MENU**
**1.Insert**
**2.Display**
**3.Delete from beginning**

4.Delete from the end
5.Delete from specified position
6.Exit
Enter your choice: 5
Enter the position of the node to be deleted : 1
The deleted element is : 5

Enter your choice: 5
Enter the position of the node to be deleted : 5
Position not found

MENU
1.Insert
2.Display
3.Delete from beginning
4.Delete from the end
5.Delete from specified position
6.Exit
Enter your choice: 6

# Program 7

**7. Obtain the Topological ordering of vertices in a given graph with the help of a c programming.**

```
#include<stdio.h>
#include<conio.h>
void main()
{
	int n, count =0, am[10][10], indeg[10], flag[10], i, j, k;
	clrscr();
	printf("Enter number of vertices:");
	scanf("%d",&n);
	for(i=0;i<n;i++)
	{
		indeg[i]=0;
		flag[i]=0;
	}
	printf("\nEnter adjacency matrix:\n");
	for(i=0;i<n;i++)
	for(j=0;j<n;j++)
		scanf("%d",&am[i][j]);
		printf("\nMatrix is :\n");
	for(i=0;i<n;i++)
	{
		for(j=0;j<n;j++)
			printf("%d\t",am[i][j]);
		printf("\n");
	}
for(i=0;i<n;i++)
```

```
for(j=0;j<n;j++)
indeg[i] += am[j][i];
printf("\nThe topological ordering is:\n");
while(count<n)
{
for(k=0;k<n;k++)
if((indeg[k]==0) && (flag[k]==0))
{
        printf("%d\n",k);
        flag[k]=1;
        count++;
        for(i=0;i<n;i++)
        if(am[k][i]==1)
        indeg[i]--;
}
}
getch();
}
```

**OUTPUT:**

**Enter number of vertices: 6**
**Enter adjacency matrix:**

**0 1 1 0 0 0**
**0 0 0 1 0 0**
**0 0 0 0 1 0**
**0 0 1 0 0 1**
**0 0 0 0 0 0**
**0 0 0 0 1 0**

**The topological ordering is:**
**0 1 3 5 2 4**

# Program 8

**8. Check whether a given graph is connected or not using DFS method using C programming.**

```
#include<stdio.h>
#include<conio.h>
void dfs(int n, int a[10][10], int u, int visited[])
{
	int v;
	visited[u]=1;
	for(v=0;v<n;v++)
	if((a[u][v]==1)&& (visited[v]==0))
	dfs(n,a,v,visited);
}
void main()
{
	int n, i, j, a[10][10], visited[10],flag, connected;
	clrscr();
	printf("Enter number of vertices:");
	scanf("%d",&n);
	printf("\nEnter adjacency matrix:\n");
	for(i=0;i<n;i++)
	for(j=0;j<n;j++)
	scanf("%d",&a[i][j]);
	connected=0;
	for(i=0;i<n;i++)
	{
		for(j=0;j<n;j++)
		visited[j]=0;
		dfs(n,a, i, visited);
	flag=0;
```

```
        for(j=0;j<n;j++)
        if(visited[ j]==0)
        flag=1;
        if(flag==0)
        connected=1;
}
if(connected==1)
        printf("\nGraph is connected");
else
        printf("\nGraph is not connected");
getch();
}
```

**OUTPUT**

**1. Enter number of vertices: 6**
**Enter adjacency matrix:**
**0 1 0 1 0 0**
**1 0 1 1 0 0**
**0 1 0 1 0 0**
**1 1 1 0 0 0**
**0 0 0 0 0 1**
**0 0 0 0 1 0**
**Graph is not connected**

**2: Enter number of vertices: 3**
**Enter adjacency matrix:**
**0 1 1**
**1 0 1**
**1 1 0**
**Graph is connected**

# Program 9

**9. From a given vertex in a weighted connected graph, find shortest paths to other vertices Using Dijkstra's algorithm (C programming)**

```
#include<stdio.h>
#include<conio.h>
void dijkstra(int n, int v, int cost[10][10],int dist[10])
{
	int count, u, i, w, visited[10], min;
	for(i=0;i<n;i++)
	{
		visited[i]=0;
		dist[i]=cost[v][i];
	}
	visited[v]=1;
	dist[v]=1;
	count=2;
	while(count<=n)
	{
		min=999;
		for(w=0;w<n;w++)
		if((dist[w]<min) && (visited[w]!=1))
		{
			min=dist[w];
			u=w;
		}
		visited[u]=1;
		count++;
		for(w=0;w<n;w++)
		if((dist[u]+cost[u][w]<dist[w]) && (visited[w]!=1))
```

```
        dist[w]=dist[u]+cost[u][w];
}
}
void main()
{
        int n, v, cost[10][10], dist[10], i, j;
        clrscr();
        printf("Enter number of vertices:");
        scanf("%d",&n);
        printf("\nEnter cost matrix (for infinity, enter 999):\n");
        for(i=0;i<n;i++)
        for(j=0;j<n;j++)
        scanf("%d",&cost[i][j]);
        printf("\nEnter source vertex:");
        scanf("%d",&v);
        dijkstra(n,v,cost,dist);
        printf("\nShortest path from \n");
        for(i=0;i<n;i++)
        if(i!=v)
        printf("\n%d -> %d = %d", v, i, dist[i]);
        getch();
}
```

## OUTPUT

```
Enter number of vertices:7
Enter cost matrix (for infinity, enter 999):
      0     2     999   3     999   999   999
      2     0     9     999   1     4     999
      999   9     0     999   999   3     999
      3     999   999   0     5     999   7
      999   1     999   5     0     999   4
      999   4     3     999   999   0     6
      999   999   999   7     4     6     0
Enter source vertex: 0
Shortest path from
0 -> 1 = 2
0 -> 2 = 9
0 -> 3 = 3
0 -> 4 = 3
0 -> 5 = 6
0 -> 6 = 7
```

# Program 10

**10. Find Minimum Cost Spanning Tree of a given undirected graph using Kruskal's algorithm ( C programming)**

```
#include<stdio.h>
#include<conio.h>
void main()
{
int n, v, u,cost[10][10], parent[10]={0}, i, j;
int count=1, mincost=0, min, a, b;
clrscr();
printf("Enter number of vertices:");
scanf("%d",&n);
printf("\nEnter cost matrix:\n");
for(i=1;i<=n;i++)
for(j=1;j<=n;j++)
{
scanf("%d",&cost[i][j]);
if(cost[i][j]==0)
cost[i][j]=999;
}
while(count<n)
{
min=999;
for(i=1;i<=n;i++)
for(j=1;j<=n;j++)
if(cost[i][j]<min)
{
```

```
min=cost[i][j];
a=u=i;
b=v=j;
}
while(parent[u])
u=parent[u];
while(parent[v])
v=parent[v];
if(u!=v)
{
count++;
printf("\nEdge(%d, %d) = %d", a, b, min);
mincost+=min;
parent[v]=u;
}
cost[a][b]=cost[b][a]=999;
}
printf("\nMinimum cost = %d", mincost);
getch();
}
```

## OUTPUT

**Enter number of vertices: 7**

**Enter cost matrix (for infinity, enter 999)**

**0 2 999 3 999 999 999**
**2 0 9 999 1 4 999**
**999 9 0 999 999 3 999**
**3 999 999 0 5 999 7**
**999 1 999 5 0 999 4**
**999 4 3 999 999 0 6**
**999 999 999 7 4 6 0**

**Edge(2, 5) = 1**
**Edge(1, 2) = 2**
**Edge(1, 4) = 3**
**Edge(3, 6) = 3**
**Edge(2, 6) = 4**
**Edge(5, 7) = 4**

**Minimum cost = 17**

# Program 11

## 11. Write a program in C to create a minimum spanning tree using Prim's algorithm, etc.

A Minimum Spanning Tree is a subset of edges that:

- Connects all vertices
- Has no cycles
- Has the minimum possible total edge weight

### 🔄 How Prim's Algorithm Works (Step-by-Step)

1. Start from any vertex.
2. Mark it as visited.
3. Find the minimum weight edge connecting a visited vertex to an unvisited vertex.
4. Add that edge to the MST.
5. Mark the new vertex as visited.
6. Repeat until all vertices are included.

```
#include <stdio.h>
#include <limits.h>
#define MAX 10
int main() {
   int n, i, j;
   int cost[MAX][MAX];
   int visited[MAX] = {0};
   int min, u = 0, v = 0;
   int totalCost = 0;

   printf("Enter number of vertices: ");
   scanf("%d", &n);

   printf("Enter cost adjacency matrix (0 if no edge):\n");
   for (i = 0; i < n; i++) {
      for (j = 0; j < n; j++) {
         scanf("%d", &cost[i][j]);
         if (cost[i][j] == 0)
            cost[i][j] = INT_MAX;
      }
   }

   visited[0] = 1;   // Start from vertex 0
   printf("\nEdges in the Minimum Spanning Tree:\n");
   for (int edge = 0; edge < n - 1; edge++) {
      min = INT_MAX;
```

```
        for (i = 0; i < n; i++) {
            if (visited[i]) {
                for (j = 0; j < n; j++) {
                    if (!visited[j] && cost[i][j] < min) {
                        min = cost[i][j];
                        u = i;
                        v = j;
                    }
                }
            }
        }

        visited[v] = 1;
        printf("%d - %d : %d\n", u, v, min);
        totalCost += min;
    }

    printf("\nTotal cost of MST = %d\n", totalCost);

    return 0;
}
```

# OUTPUT:

***Enter number of vertices: 4***
***Enter cost adjacency matrix:***
***0 2 0 6***
***2 0 3 8***
***0 3 0 0***
***6 8 0 0***

***Edges in the Minimum Spanning Tree:***
***0 - 1 : 2***
***1 - 2 : 3***
***0 - 3 : 6***

***Total cost of MST = 11***

# Program 12

## 12. Write a program to implement Breadth First search of any given graph.

Breadth First Search (BFS) is a graph traversal algorithm that explores vertices level by level, starting from a given source vertex.
It uses a queue (FIFO) data structure to maintain the order of traversal.

**Working Principle of BFS**

1. Start from a selected source vertex.
2. Mark the source vertex as visited.
3. Insert the source vertex into a queue.
4. Remove a vertex from the front of the queue and process it.
5. Visit all unvisited adjacent vertices of that vertex:
   - Mark them as visited.
   - Insert them into the queue.
6. Repeat steps 4–5 until the queue becomes empty.
7. BFS traversal is complete when all reachable vertices are visited.

```
#include <stdio.h>
#define MAX 10
int queue[MAX], front = -1, rear = -1;
int visited[MAX];

void enqueue(int v) {
   if (rear == MAX - 1)
      return;
   if (front == -1)
      front = 0;
   queue[++rear] = v;
}
int dequeue() {
   int v = queue[front++];
   if (front > rear)
      front = rear = -1;
   return v;
}
void bfs(int n, int adj[MAX][MAX], int start) {
   int i, v;
   for (i = 0; i < n; i++)
      visited[i] = 0;

   enqueue(start);
   visited[start] = 1;
```

```
    printf("BFS Traversal: ");

    while (front != -1) {
        v = dequeue();
        printf("%d ", v);

        for (i = 0; i < n; i++) {
            if (adj[v][i] == 1 && !visited[i]) {
                enqueue(i);
                visited[i] = 1;
            }
        }
    }
}

int main() {
    int n, i, j, start;
    int adj[MAX][MAX];

    printf("Enter number of vertices: ");
    scanf("%d", &n);

    printf("Enter adjacency matrix:\n");
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            scanf("%d", &adj[i][j]);

    printf("Enter starting vertex: ");
    scanf("%d", &start);
    bfs(n, adj, start);
    return 0;
}
```

## OUTPUT:

***Enter number of vertices: 4***
***Enter adjacency matrix:***
***0 1 1 0***
***1 0 0 1***
***1 0 0 1***
***0 1 1 0***
***Enter starting vertex: 0***

***BFS Traversal: 0 1 2 3***

# Program 13

**13. Write a program to implement Depth first search using linked representation of graph.**

Depth First Search (DFS) is a graph traversal algorithm that explores a graph by going as deep as possible along each branch before backtracking.
It uses recursion or a stack and is efficiently implemented using a linked representation (adjacency list) of the graph.

🔁 **Working Principle of DFS**

1. Start from a given **source vertex**.
2. Mark the vertex as **visited**.
3. Visit the **first unvisited adjacent vertex**.
4. Repeat the process recursively.
5. When no unvisited adjacent vertex is found, **backtrack**.
6. Continue until all reachable vertices are visited.

➡️ DFS follows the principle **"Go deep first, then backtrack."**

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 10

* Structure for adjacency list node *
struct node {
   int vertex;
   struct node *next;
};
struct node *adjList[MAX];
int visited[MAX];


* Create a new node *
struct node* createNode(int v) {
   struct node *newNode = (struct node*)malloc(sizeof(struct node));
   newNode->vertex = v;
   newNode->next = NULL;
   return newNode;
}
void addEdge(int src, int dest) {
   struct node *newNode = createNode(dest);
```

```
    newNode->next = adjList[src];
    adjList[src] = newNode;

    newNode = createNode(src);
    newNode->next = adjList[dest];
    adjList[dest] = newNode;
}

* DFS function *
void dfs(int v) {
    struct node *temp;
    visited[v] = 1;
    printf("%d ", v);

    temp = adjList[v];
    while (temp != NULL) {
        if (!visited[temp->vertex]) {
            dfs(temp->vertex);
        }
        temp = temp->next;
    }
}

int main() {
    int n, e, i, src, dest, start;

    printf("Enter number of vertices: ");
    scanf("%d", &n);

    for (i = 0; i < n; i++) {
        adjList[i] = NULL;
        visited[i] = 0;
    }

    printf("Enter number of edges: ");
    scanf("%d", &e);

    for (i = 0; i < e; i++) {
        printf("Enter edge (source destination): ");
        scanf("%d %d", &src, &dest);
        addEdge(src, dest);
```

```
    }

    printf("Enter starting vertex: ");
    scanf("%d", &start);

    printf("DFS Traversal: ");
    dfs(start);

    return 0;
}
```

## OUTPUT:

**Enter number of vertices: 4**
**Enter number of edges: 3**
**Enter edge (source destination): 0 1**
**Enter edge (source destination): 0 2**
**Enter edge (source destination): 1 3**
**Enter starting vertex: 0**

**DFS Traversal: 0 2 1 3**

# Program 14

## 14. Write a program to solve a given 0/1 knapsack problem.

The 0/1 Knapsack Problem is a classic optimization problem in, typically solved using dynamic programming. It involves selecting a subset of items to maximize their total value within a given weight capacity, with the constraint that each item can either be taken entirely (1) or left out completely (0).

🔁 **Working Principle**

The 0/1 Knapsack problem is solved using **Dynamic Programming**.

- Create a table dp[i][w]
- dp[i][w] represents the **maximum profit** using the first i items with capacity w
- For each item:
  - Either **include** it (if weight allows)
  - Or **exclude** it
- Choose the **maximum** of the two choices

```
#include <stdio.h>
int max(int a, int b) {
   return (a > b) ? a : b;
}
int main() {
   int n, W, i, w;
   int weight[20], profit[20];
   int dp[20][50];

   printf("Enter number of items: ");
   scanf("%d", &n);

   printf("Enter weights of items:\n");
   for (i = 1; i <= n; i++)
      scanf("%d", &weight[i]);

   printf("Enter profits of items:\n");
   for (i = 1; i <= n; i++)
      scanf("%d", &profit[i]);
```

```
    printf("Enter capacity of knapsack: ");
    scanf("%d", &W);

    for (i = 0; i <= n; i++) {
       for (w = 0; w <= W; w++) {
          if (i == 0 || w == 0)
             dp[i][w] = 0;
          else if (weight[i] <= w)
             dp[i][w] = max(profit[i] + dp[i - 1][w - weight[i]],
                        dp[i - 1][w]);
          else
             dp[i][w] = dp[i - 1][w];
       }
    }

    printf("Maximum Profit = %d\n", dp[n][W]);
    return 0;
}
```

## OUTPUT:

**Enter number of items: 3**
**Enter weights of items:**
**10 20 30**
**Enter profits of items:**
**60 100 120**
**Enter capacity of knapsack: 50**

**Maximum Profit = 220**

# Program 15

## 15. Implement Inorder, Preorder and Post Order traversal of a binary tree using both recursive and non-recursive way.

**Binary Tree Traversal**

Traversal means visiting each node of a binary tree exactly once in a specific order. There are three standard depth-first traversals:

1. **Inorder Traversal (L-R)**
   - Left subtree → Root → Right subtree
   - Used in **BST** to get sorted order
2. **Preorder Traversal (R-L)**
   - Root → Left subtree → Right subtree
   - Used to **create/copy tree**
3. **Postorder Traversal (L-R-Root)**
   - Left subtree → Right subtree → Root
   - Used to **delete tree / expression evaluation**

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

#define MAX 50

* ------ Binary Tree Node -----*
struct node {
   int data;
   struct node *left, *right;
};

* ----- Stack for Iterative Traversal ----- *
struct node* stack[MAX];
int top = -1;

void push(struct node* n) {
   stack[++top] = n;
}
struct node* pop() {
   return stack[top--];
}
int isEmpty() {
   return top == -1;
}
```

```
* ----- Create Node ----- *
struct node* createNode(int data) {
   struct node* newNode = (struct node*)malloc(sizeof(struct node));
   newNode->data = data;
   newNode->left = newNode->right = NULL;
   return newNode;
}

* ----- Recursive Traversals ------ *
void inorder(struct node* root) {
   if (root != NULL) {
      inorder(root->left);
      printf("%d ", root->data);
      inorder(root->right);
   }
}

void preorder(struct node* root) {
   if (root != NULL) {
      printf("%d ", root->data);
      preorder(root->left);
      preorder(root->right);
   }
}

void postorder(struct node* root) {
   if (root != NULL) {
      postorder(root->left);
      postorder(root->right);
      printf("%d ", root->data);
   }
}

* ----- Iterative Inorder ----- *
void inorderIterative(struct node* root) {
   struct node* curr = root;
   while (curr != NULL || !isEmpty()) {
      while (curr != NULL) {
         push(curr);
         curr = curr->left;
      }
      curr = pop();
      printf("%d ", curr->data);
      curr = curr->right;
   }
}
```

```
* ----- Iterative Preorder --- *
void preorderIterative(struct node* root) {
   if (root == NULL) return;
   push(root);
   while (!isEmpty()) {
      struct node* curr = pop();
      printf("%d ", curr->data);
      if (curr->right) push(curr->right);
      if (curr->left) push(curr->left);
   }
}

* --Iterative Postorder (Two Stacks) -- *
void postorderIterative(struct node* root) {
   if (root == NULL) return;

   struct node* s1[MAX];
   struct node* s2[MAX];
   int t1 = -1, t2 = -1;

   s1[++t1] = root;

   while (t1 != -1) {
      struct node* curr = s1[t1--];
      s2[++t2] = curr;

      if (curr->left) s1[++t1] = curr->left;
      if (curr->right) s1[++t1] = curr->right;
   }

   while (t2 != -1) {
      printf("%d ", s2[t2--]->data);
   }
}

* ----- Main Function ----- *
int main() {
   struct node* root = createNode(1);
   root->left = createNode(2);
   root->right = createNode(3);
   root->left->left = createNode(4);
   root->left->right = createNode(5);
   printf("Recursive Inorder: ");
   inorder(root);
   printf("\nRecursive Preorder: ");
   preorder(root);
```

```
    printf("\nRecursive Postorder: ");
    postorder(root);

    printf("\n\nIterative Inorder: ");
    inorderIterative(root);

    printf("\nIterative Preorder: ");
    preorderIterative(root);

    printf("\nIterative Postorder: ");
    postorderIterative(root);

    return 0;
}
```

## OUTPUT:

**Recursive Inorder: 4 2 5 1 3**
**Recursive Preorder: 1 2 4 5 3**
**Recursive Postorder: 4 5 2 3 1**

**Iterative Inorder: 4 2 5 1 3**
**Iterative Preorder: 1 2 4 5 3**
**Iterative Postorder: 4 5 2 3 1**

# Program 16

## 16. Develop a C program to implement to create an AVL tree along with insertion and deletion operations.

An AVL Tree (Adelson-Velsky and Landis Tree) is a self-balancing Binary Search Tree (BST) in which the height difference (balance factor) between the left and right subtrees of any node is at most ±1.

**BF = height(left subtree) – height(right subtree)**

```
#include <stdio.h>
#include <stdlib.h>
struct node {
   int data;
   struct node *left, *right;
   int height;
};
int max(int a, int b) {
   return (a > b) ? a : b;
}
int height(struct node *n) {
   return (n == NULL) ? 0 : n->height;
}
struct node* createNode(int data) {
   struct node* node = (struct node*)malloc(sizeof(struct node));
   node->data = data;
   node->left = node->right = NULL;
   node->height = 1;
   return node;
}
struct node* rightRotate(struct node* y) {
   struct node* x = y->left;
   struct node* T2 = x->right;

   x->right = y;
   y->left = T2;
   y->height = max(height(y->left), height(y->right)) + 1;
   x->height = max(height(x->left), height(x->right)) + 1;

   return x;
}
struct node* leftRotate(struct node* x) {
   struct node* y = x->right;
   struct node* T2 = y->left;
```

```
  y->left = x;
  x->right = T2;

  x->height = max(height(x->left), height(x->right)) + 1;
  y->height = max(height(y->left), height(y->right)) + 1;

  return y;
}
int getBalance(struct node* n) {
  return (n == NULL) ? 0 : height(n->left) - height(n->right);
}
struct node* insert(struct node* node, int data) {
  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);
  else
    return node;

  node->height = 1 + max(height(node->left), height(node->right));
  int balance = getBalance(node);
  if (balance > 1 && data < node->left->data)
    return rightRotate(node);

  if (balance < -1 && data > node->right->data)
    return leftRotate(node);

  if (balance > 1 && data > node->left->data) {
    node->left = leftRotate(node->left);
    return rightRotate(node);
  }

  if (balance < -1 && data < node->right->data) {
    node->right = rightRotate(node->right);
    return leftRotate(node);
  }

  return node;
}
struct node* minValueNode(struct node* node) {
  struct node* current = node;
  while (current->left != NULL)
    current = current->left;
  return current;
```

```
}
struct node* deleteNode(struct node* root, int data) {
  if (root == NULL)
    return root;

  if (data < root->data)
    root->left = deleteNode(root->left, data);
  else if (data > root->data)
    root->right = deleteNode(root->right, data);
  else {
    if ((root->left == NULL) || (root->right == NULL)) {
      struct node* temp = root->left ? root->left : root->right;

      if (temp == NULL) {
        temp = root;
        root = NULL;
      } else
        *root = *temp;

      free(temp);
    } else {
      struct node* temp = minValueNode(root->right);
      root->data = temp->data;
      root->right = deleteNode(root->right, temp->data);
    }
  }

  if (root == NULL)
    return root;

  root->height = 1 + max(height(root->left), height(root->right));
  int balance = getBalance(root);

  if (balance > 1 && getBalance(root->left) >= 0)
    return rightRotate(root);

  if (balance > 1 && getBalance(root->left) < 0) {
    root->left = leftRotate(root->left);
    return rightRotate(root);
  }

  if (balance < -1 && getBalance(root->right) <= 0)
    return leftRotate(root);

  if (balance < -1 && getBalance(root->right) > 0) {
    root->right = rightRotate(root->right);
    return leftRotate(root);
```

```
    }

    return root;
}
void inorder(struct node* root) {
    if (root != NULL) {
        inorder(root->left);
        printf("%d ", root->data);
        inorder(root->right);
    }
}
int main() {
    struct node* root = NULL;

    root = insert(root, 30);
    root = insert(root, 20);
    root = insert(root, 40);
    root = insert(root, 10);

    printf("Inorder traversal after insertion:\n");
    inorder(root);

    root = deleteNode(root, 20);

    printf("\nInorder traversal after deletion:\n");
    inorder(root);

    return 0;
}
```

## OUTPUT:

**Inorder traversal after insertion:**
**10 20 30 40**
**Inorder traversal after deletion:**
**10 30 40**

# Program 17

**17. Write a menu driven program that implements a Heap (Max or Min) for the following operations. Insert, Delete.**

A Heap is a complete binary tree that satisfies the heap property.

- Max Heap

In a Max Heap:

- The value of each parent node is greater than or equal to its children
- The maximum element is always at the root

- Heap Operations

1. Insert
    - Insert the element at the end
    - Restore heap property using heapify-up
2. Delete (Max element)
    - Remove the root element
    - Replace root with last element
    - Restore heap property using heapify-down

```
#include <stdio.h>
#define MAX 50
int heap[MAX];
int size = 0;
void swap(int *a, int *b) {
   int temp = *a;
   *a = *b;
   *b = temp;
}
void heapifyUp(int index) {
   while (index > 0 && heap[(index - 1) / 2] < heap[index]) {
      swap(&heap[index], &heap[(index - 1) / 2]);
      index = (index - 1) / 2;
   }
}
void heapifyDown(int index) {
   int largest = index;
   int left = 2 * index + 1;
   int right = 2 * index + 2;
```

```
    if (left < size && heap[left] > heap[largest])
        largest = left;
    if (right < size && heap[right] > heap[largest])
        largest = right;
    if (largest != index) {
        swap(&heap[index], &heap[largest]);
        heapifyDown(largest);
    }
}
void insert(int value) {
    if (size == MAX) {
        printf("Heap Overflow\n");
        return;
    }
    heap[size] = value;
    heapifyUp(size);
    size++;
    printf("Inserted %d into heap\n", value);
}
void deleteMax() {
    if (size == 0) {
        printf("Heap Underflow\n");
        return;
    }
    printf("Deleted element: %d\n", heap[0]);
    heap[0] = heap[size - 1];
    size--;
    heapifyDown(0);
}
void display() {
    if (size == 0) {
        printf("Heap is empty\n");
        return;
    }
    printf("Heap elements: ");
    for (int i = 0; i < size; i++)
        printf("%d ", heap[i]);
    printf("\n");
```

```
}
int main() {
    int choice, value;

    do {
        printf("\n--- MAX HEAP MENU ---\n");
        printf("1. Insert\n");
        printf("2. Delete Max\n");
        printf("3. Display\n");
        printf("4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
        switch (choice) {
            case 1:
                printf("Enter value to insert: ");
                scanf("%d", &value);
                insert(value);
                break;
            case 2:
                deleteMax();
                break;
            case 3:
                display();
                break;
            case 4:
                printf("Exiting program\n");
                break;
            default:
                printf("Invalid choice\n");
        }
    } while (choice != 4);
    return 0;
}
```

## OUTPUT:

```
--- MAX HEAP MENU ---
1. Insert
2. Delete Max
3. Display
4. Exit
Enter your choice: 1
Enter value to insert: 50
Inserted 50 into heap

Enter your choice: 1
Enter value to insert: 30
Inserted 30 into heap

Enter your choice: 1
Enter value to insert: 40
Inserted 40 into heap

Enter your choice: 3
Heap elements: 50 30 40

Enter your choice: 2
Deleted element: 50

Enter your choice: 3
Heap elements: 40 30

Enter your choice: 4
Exiting program
```

# Program 18

**18. Develop a program to represent a polynomial. Also write methods for adding and subtracting two polynomials.**

## Polynomial Representation

A **polynomial** is an expression of the form:

$$P(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$$

```
#include <stdio.h>
#include <stdlib.h>
struct node {
   int coeff;
   int exp;
   struct node *next;
};
struct node* createNode(int coeff, int exp) {
   struct node* newNode = (struct node*)malloc(sizeof(struct node));
   newNode->coeff = coeff;
   newNode->exp = exp;
   newNode->next = NULL;
   return newNode;
}
void insertTerm(struct node **poly, int coeff, int exp) {
   struct node* newNode = createNode(coeff, exp);
   struct node* temp = *poly;

   if (*poly == NULL) {
      *poly = newNode;
      return;
   }

   while (temp->next != NULL)
      temp = temp->next;

   temp->next = newNode;
}
```

```
void display(struct node* poly) {
   if (poly == NULL) {
      printf("0");
      return;
   }
   while (poly != NULL) {
      printf("%dx^%d", poly->coeff, poly->exp);
      if (poly->next != NULL)
         printf(" + ");
      poly = poly->next;
   }
   printf("\n");
}
struct node* addPoly(struct node* p1, struct node* p2) {
   struct node* result = NULL;

   while (p1 != NULL && p2 != NULL) {
      if (p1->exp == p2->exp) {
         insertTerm(&result, p1->coeff + p2->coeff, p1->exp);
         p1 = p1->next;
         p2 = p2->next;
      } else if (p1->exp > p2->exp) {
         insertTerm(&result, p1->coeff, p1->exp);
         p1 = p1->next;
      } else {
         insertTerm(&result, p2->coeff, p2->exp);
         p2 = p2->next;
      }
   }

   while (p1 != NULL) {
      insertTerm(&result, p1->coeff, p1->exp);
      p1 = p1->next;
   }

   while (p2 != NULL) {
      insertTerm(&result, p2->coeff, p2->exp);
      p2 = p2->next;
   }

   return result;
```

```
}
struct node* subPoly(struct node* p1, struct node* p2) {
   struct node* result = NULL;

   while (p1 != NULL && p2 != NULL) {
      if (p1->exp == p2->exp) {
         insertTerm(&result, p1->coeff - p2->coeff, p1->exp);
         p1 = p1->next;
         p2 = p2->next;
      } else if (p1->exp > p2->exp) {
         insertTerm(&result, p1->coeff, p1->exp);
         p1 = p1->next;
      } else {
         insertTerm(&result, -p2->coeff, p2->exp);
         p2 = p2->next;
      }
   }

   while (p1 != NULL) {
      insertTerm(&result, p1->coeff, p1->exp);
      p1 = p1->next;
   }

   while (p2 != NULL) {
      insertTerm(&result, -p2->coeff, p2->exp);
      p2 = p2->next;
   }

   return result;
}
int main() {
   struct node *poly1 = NULL, *poly2 = NULL;
   struct node *sum = NULL, *diff = NULL;

   * Polynomial 1: 5x^3 + 4x^2 + 2x^0 *
   insertTerm(&poly1, 5, 3);
   insertTerm(&poly1, 4, 2);
   insertTerm(&poly1, 2, 0);

   * Polynomial 2: 3x^3 + 1x^1 + 4x^0 *
   insertTerm(&poly2, 3, 3);
```

```
    insertTerm(&poly2, 1, 1);
    insertTerm(&poly2, 4, 0);

    printf("Polynomial 1: ");
    display(poly1);

    printf("Polynomial 2: ");
    display(poly2);

    sum = addPoly(poly1, poly2);
    printf("Addition Result: ");
    display(sum);

    diff = subPoly(poly1, poly2);
    printf("Subtraction Result (P1 - P2): ");
    display(diff);

    return 0;
}
```

## OUTPUT:

**Polynomial 1: 5x^3 + 4x^2 + 2x^0**
**Polynomial 2: 3x^3 + 1x^1 + 4x^0**
**Addition Result: 8x^3 + 4x^2 + 1x^1 + 6x^0**
**Subtraction Result (P1 - P2): 2x^3 + 4x^2 + -1x^1 + -2x^0**

# Program 19

## 19. Using a C program, find the minimum number of multiplications needed to multiply a chain of matrices using dynamic programming approach.

In **Matrix Chain Multiplication**, we are given a sequence of matrices

$$A_1, A_2, A_3, \dots, A_n$$

Matrix multiplication is **associative**, so we can change the order of multiplication, but:

- The **result remains the same**
- The **number of scalar multiplications changes**

The goal is to find the **minimum number of scalar multiplications** needed to multiply the entire chain.

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

int min(int a, int b) {
   return (a < b) ? a : b;
}

int matrixChainOrder(int p[], int n) {
   int m[n][n];

   // Cost is zero when multiplying one matrix
   for (int i = 1; i < n; i++)
      m[i][i] = 0;

   // L is chain length
   for (int L = 2; L < n; L++) {
      for (int i = 1; i < n - L + 1; i++) {
         int j = i + L - 1;
         m[i][j] = INT_MAX;

         for (int k = i; k < j; k++) {
            int cost = m[i][k] + m[k + 1][j]
                  + p[i - 1] * p[k] * p[j];
            m[i][j] = min(m[i][j], cost);
         }
      }
```

```
    }
    return m[1][n - 1];
}

int main() {
    int n;

    printf("Enter number of matrices: ");
    scanf("%d", &n);

    int p[n + 1];
    printf("Enter dimensions array:\n");
    for (int i = 0; i <= n; i++)
        scanf("%d", &p[i]);

    int result = matrixChainOrder(p, n + 1);
    printf("Minimum number of multiplications = %d\n", result);

    return 0;
}
```

## OUTPUT:

**Enter number of matrices: 3**
**Enter dimensions array:**
**10 30 5 60**

**Minimum number of multiplications = 4500**

# Program 20

## 20. Find the longest common subsequence from two given strings by writing a program using dynamic programming approach.

A subsequence is a sequence that appears in the same relative order but not necessarily contiguously.

**Example:**

Subsequences of "ABC" → A, B, C, AB, AC, BC, ABC

- **Working Principle**

1. Create a DP table L[m+1][n+1]
2. L[i][j] stores the **length of LCS** of:
   - first i characters of string 1
   - first j characters of string 2
3. Rules:
   - If characters match → L[i][j] = L[i-1][j-1] + 1
   - Else → L[i][j] = max(L[i-1][j], L[i][j-1])
4. Backtrack the table to **print the LCS**

```
#include <stdio.h>
#include <string.h>
#define MAX 50

int max(int a, int b) {
   return (a > b) ? a : b;
}
void findLCS(char X[], char Y[]) {
   int m = strlen(X);
   int n = strlen(Y);
   int L[MAX][MAX];
   for (int i = 0; i <= m; i++) {
      for (int j = 0; j <= n; j++) {
         if (i == 0 || j == 0)
            L[i][j] = 0;
         else if (X[i - 1] == Y[j - 1])
            L[i][j] = L[i - 1][j - 1] + 1;
         else
            L[i][j] = max(L[i - 1][j], L[i][j - 1]);
```

```
      }
    }
    int index = L[m][n];
    char lcs[MAX];
    lcs[index] = '\0';

    int i = m, j = n;
    while (i > 0 && j > 0) {
      if (X[i - 1] == Y[j - 1]) {
        lcs[index - 1] = X[i - 1];
        i--;
        j--;
        index--;
      } else if (L[i - 1][j] > L[i][j - 1])
        i--;
      else
        j--;
    }

    printf("Longest Common Subsequence: %s\n", lcs);
    printf("Length of LCS: %d\n", L[m][n]);
}

int main() {
    char X[MAX], Y[MAX];

    printf("Enter first string: ");
    scanf("%s", X);

    printf("Enter second string: ");
    scanf("%s", Y);

    findLCS(X, Y);

    return 0;
}
```

## OUTPUT:

**Enter first string : ABCDGH**
**Enter second string : AEDFHR**

**Longest Common Subsequence: ADH**
**Length of LCS: 3**

# Program 21

## 21. Using a program show how the collision resolution (linear probing) works in hashing.

Hashing is a technique used to map a key value to a position (index) in a hash table using a hash function.

**Example hash function:**

$$h(key) = keymodtable\_size$$

**Collision:**

A collision occurs when two different keys produce the same hash index.

Example:

For table size = 10

h(12) = 12 % 10 = 2

h(22) = 22 % 10 = 2 ← collision

```
#include <stdio.h>
#define SIZE 10
int hashTable[SIZE];
void init() {
   for (int i = 0; i < SIZE; i++)
      hashTable[i] = -1;
}
int hashFunction(int key) {
   return key % SIZE;
}
void insert(int key) {
   int index = hashFunction(key);
   if (hashTable[index] == -1) {
      hashTable[index] = key;
   } else {
      printf("Collision occurred at index %d for key %d\n", index, key);
      int i = (index + 1) % SIZE;
      while (i != index) {
         if (hashTable[i] == -1) {
            hashTable[i] = key;
            return;
```

```
            }

            i = (i + 1) % SIZE;
        }
        printf("Hash table is full\n");
    }
}
void display() {
    printf("\nHash Table:\n");
    for (int i = 0; i < SIZE; i++) {
        if (hashTable[i] != -1)
            printf("Index %d : %d\n", i, hashTable[i]);
        else
            printf("Index %d : EMPTY\n", i);
    }
}

int main() {
    int keys[] = {23, 43, 13, 27, 56};
    int n = sizeof(keys) / sizeof(keys[0]);

    init();

    for (int i = 0; i < n; i++) {
        insert(keys[i]);
    }

    display();
    return 0;
}
```

**OUTPUT:**

```
Collision occurred at index 3 for key 43
Collision occurred at index 3 for key 13

Hash Table:
Index 0 : EMPTY
Index 1 : EMPTY
Index 2 : EMPTY
Index 3 : 23
Index 4 : 43
Index 5 : 13
Index 6 : EMPTY
Index 7 : 27
Index 8 : 56
Index 9 : EMPTY
```