EXPERIMENT NO:1

Date:

Aim: Familiarization of GCC options.

Description: The GCC stands for GNU Compiler Collection. It is a compiler system for the various programming languages. It is mainly used to compile the C and C++ programs. It takes the name of the source program as a necessary argument; rest arguments are optional such as debugging, warning, object file, and linking libraries.

By default, the gcc command creates the object file as 'a.out**.'**

Some GCC options are:

- -o: If we want to change the default output file name, use the **'-o'** option.
  To specify the object file name, execute the command as below:
  gcc hello.c -o hello
- -Wall: To enable all warnings in the output, use the '-Wall' option with the gcc command.
  gcc -wall hello.c
- -E: We can produce only the preprocess output by using the '-E' option.
  The below command contains preprocessed output is generated.
  gcc -E hello.c > hello.i
- -S: To produce the assembly code, execute the command with the '-S' option.
  gcc -S hello.c > hello.s
- -C: We can produce only the compiled code by using the '-C' option.
  gcc -C hello.c
- -save-temp: We can produce all the intermediate files of the compilation process by using the '-save-temp' option.
  gcc -save-temps hello.c

Result: Familiarised with GCC options.

EXPERIMENT NO:2

Date:

Aim: Familiarization of GDB options.

Description: gdb is the acronym for GNU Debugger. This tool helps to debug the programs written in C, C++, Ada, Fortran, etc. The console can be opened using the gdb command on terminal. To prepare your program for debugging with gdb, you must compile it with the -g flag. So, if your program is in a source file called memsim.c and you want to put the executable in the file memsim, then you would compile with the following command:

$ gdb sample1

gcc -g -o sample1 sample1.c

Some GDB options are:

- run [args]: run will start the program running under gdb.
- break: A "breakpoint" is a spot in your program where you would like to temporarily stop execution in order to check the values of variables, or to try to find out where the program is crashing, etc. To set a breakpoint you use the break command.
  break filename:linenumber
- delete: Delete will delete all breakpoints that you have set.
  delete number
  will delete breakpoint numbered number. You can find out what number each breakpoint is by doing info breakpoints.
- clear: clear function will delete the breakpoint set at that function. Similarly for linenumber, filename:function, and filename:linenumber.
- continue: continue will set the program running again, after you have stopped it at a breakpoint.
- step: step will go ahead and execute the current source line, and then stop execution again before the next source line.
- next: next will continue until the next source line in the current function.This is similar to step, except that if the line about to be executed is a function call, then that function call will be completely executed before execution stops again, whereas with step execution will stop at the first line of the function that is called.

Result: Familiarised with GDB options.

EXPERIMENT NO:3

Date:

Aim: Familiarization of Gprof options.

Description: The gprof command produces an execution profile of C, FORTRAN, or COBOL programs. The effect of called routines is incorporated into the profile of each caller. The gprof command is useful in identifying how a program consumes processor resource. To find out which functions (routines) in the program are using the processor, you can profile the program with the gprof command. The GPROF environment variable can be used to set different options for profiling. The syntax of this environment variable is defined as follows:

GPROF = profile:<profile-type>,scale:<scaling-factor>,file:<file-type>,filename:<filename>

- -pg: The -pg option also links in versions of library routines that are compiled for profiling, and reads the symbol table in the named object file (a.out by default).
- -b: Suppresses the printing of a description of each field in the profile.
- -c Filename: Creates a file that contains the information that is needed for remote processing of profiling information. Do not use the -c flag in combination with other flags.
- -E Name: Suppresses the printing of the graph profile entry for routine Name and its descendants, similar to the -e flag, but excludes the time that is spent by routine Name and its descendants from the total and percentage time computations.
- -s: Produces the gmon.sum profile file, which represents the sum of the profile information in all the specified profile files. This summary profile file might be given to subsequent executions of the gprof command (by using the -s flag) to accumulate profile data across several runs of an a.out file.
- -g Filename: Writes call graph information to the specified output filename. It also suppresses the profile information unless the -p flag is used.
- -z: Displays routines that have zero usage (as indicated by call counts and accumulated time).

Result: Familiarised with GDB options.

EXPERIMENT NO:4

Date:

Aim: Write a program to merge two sorted arrays.

Program:

```
#include<stdio.h>
void main()
{
int a[50],b[50],c[100],k=0,i,j,m,n;
printf("Enter the size of first array\t");
scanf("%d",&m);
printf("\nEnter the array elements");
for(i=0;i<m;i++)
{
printf("\nEnter element %d-",i+1);
scanf("%d",&a[i]);
}
printf("Enter the size of second array\t");
scanf("%d",&n);
printf("\nEnter the array elements\n");
for(i=0;i<n;i++)
{
printf("\nEnter element %d-",i+1);
scanf("%d",&b[i]);
}
```

```
i=0;
j=0;
while(i<m && j<n)
{
if(a[i]<b[j])
{c[k]=a[i];
i++;
}
else
{
c[k]=b[j];
j++;
}
k++;
}
if(i>=m)
{
while(j<n)
{
c[k]=b[j];
j++;
k++;
}
}
```

```
if(j>=n)
{
while(i<m)
{c[k]=a[i];
i++;
k++;
}}
printf("\nArray after merge sort \n");
for(i=0;i<m+n;i++)
{printf("%d\t",c[i]);
}
}
```

Result:

The program is executed successfully and output is verified.

EXPERIMENT NO:5

Date:

Aim: Write a program to implement circular queue.

Program:

```
#include <stdio.h>
#include<stdlib.h>
# define max 6
int queue[max];
int front=-1;
int rear=-1;
void enqueue(int element)
{
   if(front==-1 && rear==-1)
   {
      front=0;
      rear=0;
      queue[rear]=element;
   }
   else if((rear+1)%max==front)
   {
      printf("Queue is overflow..");
   }
   else
   {
```

```
        rear=(rear+1)%max;
        queue[rear]=element;
    }
}
void dequeue()
{
    if((front==-1) && (rear==-1))
    {
        printf("\nQueue is underflow..");
    }
 else if(front==rear)
{
   printf("\nThe dequeued element is %d", queue[front]);
   front=-1;
   rear=-1;
}
else
{
    printf("\nThe dequeued element is %d", queue[front]);
   front=(front+1)%max;
}
}
void display()
{
```

```
    int i=front;
    if(front==-1 && rear==-1)
    {
       printf("\n Queue is empty..");
    }
    else
    {
       printf("\nElements in a Queue are :");
       while(i<=rear)
       {
          printf("%d=>", queue[i]);
          i=(i+1)%max;
       }
    }
}
void main()
{  int ch,x;
printf(“\nCircular Queue\n”);
    printf("\nPress 1: Insert an element");
    printf("\nPress 2: Delete an element");
    printf("\nPress 3: Display the element");
    while(ch!=4)   // while loop
   {printf("\nEnter your choice");
    scanf("%d", &ch);
```

```
   switch(ch)
   {case 1: printf("Enter the element which is to be inserted");
      scanf("%d", &x);
      enqueue(x);
      break;
      case 2:
      dequeue();
      break;
      case 3:
      display();
      break;
      case 4: exit(0);
      break;
      default: printf("\nWrong Entry\n");
      }
}
}
```

Result:

The program is executed successfully and output is verified.

EXPERIMENT NO:6

Date:

Aim: Write a program to perform singly linked list operations.

Program:

```
#include<stdio.h>
#include<malloc.h>
#include<stdlib.h>
void insertbeg();
void insertend();
void insertpos();
void display();
int ch,key;
struct node
{
int data;
struct node *next;
};
struct node *head=NULL;
void insertbeg()
{
struct node *ptr;
ptr=(struct node*)malloc(sizeof(struct node));
if(ptr==NULL)
{printf("\nNo space");
```

```
}
else
{
        printf("\nEnter the item to be inserted");
        scanf("%d",&ptr->data);
        ptr->next=NULL;

        if(head==NULL)
        {head=ptr;
        }
        else
        {
        ptr->next=head;
        head=ptr;
        }
        printf("\n%d Inserted into the list",ptr->data);
}
}
void insertend()
{struct node *temp,*ptr;
ptr=(struct node*)malloc(sizeof(struct node));
ptr->next=NULL;
temp=head;
if(ptr==NULL)
```

```
{printf("\nNo space");
}
else
{       while(temp->next!=NULL)
        {temp=temp->next;
        }
        printf("\nEnter the item to be inserted");
        scanf("%d",&ptr->data);
        temp->next=ptr;
        printf("\n%d Inserted into the list",ptr->data);

}
}
void insertpos()
{
int key;
struct node *ptr,*temp;
ptr=(struct node*)malloc(sizeof(struct node));
ptr->next=NULL;
printf("\nEnter the value after which the new node to be inserted");
scanf("%d",&key);
temp=head;
while(temp->data!=key)
{temp=temp->next;
```

```
if(temp==NULL)
{printf("\n%dThe value does not exist",key);
break;
}
}

if(temp->data==key)
{
if(ptr==NULL)
 {printf("\nNo space");}
else
{
printf("\nEnter the item to be inserted");
scanf("%d",&ptr->data);
ptr->next=temp->next;
temp->next=ptr;
printf("\n%d Inserted after%d",ptr->data,key);
}
}
}
void deletefront()
{
struct node*temp;
```

```
temp=head;
head=temp->next;
printf("%d Deleted element is ",temp->data);
free(temp);
}
void deleteend()
{struct node*temp,*p;
temp=head;
while(temp->next!=0)
{
p=temp;
temp=temp->next;
}
printf("%d Deleted element is ",temp->data);
free(temp);
temp->next=0;
}
void deletepos()
{
struct node *temp,*p;
int val;
printf("Enter the value tobe deleted ");
scanf("%d",&val);
temp=head;
```

```
while(temp->data!=val)
{
p=temp;
temp=temp->next;
if(temp==0)
{
printf("value does not exist ");
break;
}

}
if(temp->data==val)
{

printf("The deleted node is %d",temp->data);
if(temp==head)
{
p=head;
head=temp->next;
 free(p);
  }
  else
  {
```

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


free(temp);}


}
}




void display()
{struct node *p;
if(head==NULL)
{printf("\nList is empty ");


}
else
{printf("\nElements in list are:");
p=head;
while(p!=0)
{printf("\t %d",p->data);
p=p->next;
}
}
}
```

```
void search()
{struct node*temp;
int val,pos=1;
temp=head;
printf("Enter the element to be searched ");
scanf("%d",&key);
while(temp->data!=key)
{temp=temp->next;
pos++;
if(temp==0)
{printf("\n %dThe value does not exist ",key);
break;
}
}
if(temp->data==key)
{printf("\n%dThe value found at position %d ", key, pos);
}
}
void main()
{int ch;
printf("\nSingly Linked List Operations\n");
printf("\n1.Insert Front\n2.Insert  end\n3.Insert after a node\n4.Delete from front\n5.Delete from end\n6.Deletion  a node\n7.Display\n8.Search\n");

do
```

```
{
printf("\nEnter the choice");
scanf("%d",&ch);
switch(ch)
{
case 1:insertbeg();
                break;
case 2:insertend();
                break;
case 3:insertpos();
                break;
case 4: deletefront();
        break;
case 5: deleteend();
        break;
case 6: deletepos();
        break;
case 7: display();
        break;
case 8: search();
        break;
default:printf("\nWrong entry");
}
}
```

```
while(ch!=9);

}
```

Result:

The program is executed successfully and output is verified.

EXPERIMENT NO:7

Date:

Aim: Write a program to perform stack operation using linked list operations.

Program:

```
#include<stdio.h>
#include<stdlib.h>
#include<malloc.h>
struct node
{int data;
struct npde*next;
};
struct node*top=0;

void main()
{int ch, val;
printf("-----Stack Using Linked List----\n");
do{
printf("\nMenu\n");
printf("\n1.Push\n2.Pop\n3.Display\n4.Exit\n");
printf
("Enter your choice\t");
scanf("%d",&ch);
switch(ch)
{case 1: printf("Enter the to be inserted ");
```

```
            scanf("%d",&val);
            push(val);
            break;
case 2: pop();
            break;
case 3 :display();
            break;
case 4 :exit(0);
            break;
default: printf("\n Wrong entry enter a valid choice\n");

}
}while(ch!=4);

}
void push(int val)
{
struct node*newnode;
newnode=(struct node*)malloc(sizeof(struct node));
newnode->data=val;
newnode->next=0;

if(top==0)
{
```

```
top=newnode;
}
else
{
newnode->next=top;
top=newnode;
}
printf("\nInsertion successful %d ",newnode->data);
}
void pop()
{
if(top==0)
{printf("\nStack is empty\n");
}
else
{
struct node*temp=top;
printf("\nDeleted element %d ",temp->data);
top=temp->next;
free(temp);
}
}
void display()
{
```

```
if(top==0)
{printf("\nStack is empty");

}
else
{struct node*temp=top;
printf("\nElements in stack are\n");
while(temp->next!=0)
{printf("%d----> ",temp->data);
temp=temp->next;

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

}
}
```

Result:

The program is executed successfully and output is verified.

EXPERIMENT NO:8

Date:

Aim: Write a program to perform doubly linked list operations.

Program:

```
#include<stdio.h>
#include<stdlib.h>
#include<malloc.h>
struct node
{
   struct node *prev;
   struct node *next;
   int data;
};
struct node *head=0;

void main ()
{
int ch;
printf("\nDoubly Linked List\n");
   printf("\n1.Insert in begining\n2.Insert at last\n3.Insert at any random location\n4.Delete from Beginning\n5.Delete from last\n6.Delete the node after the given data\n7.Search\n8.Show\n9.Exit\n");
   printf("\n*********Main Menu*********\n");
   while(ch != 9)
   {
```

```
printf("\nEnter your choice?\n");
scanf("\n%d",&ch);
switch(ch)
{
   case 1:
   insertbeg();
   break;
   case 2:
   insertend();
   break;
   case 3:
   insertpos();
   break;
   case 4:
   delbeg();
   break;
   case 5:
   delend();
   break;
   case 6:
   delpos();
   break;
```

```
            case 7:
            search();
            break;
            case 8:
            display();
            break;
            case 9:
            exit(0);
            break;
            default:
            printf("Please enter valid choice..");
        }
    }
}
void insertbeg()
{
   struct node *ptr;
   int item;
   ptr = (struct node *)malloc(sizeof(struct node));
   if(ptr == NULL)
   {
      printf("\nOVERFLOW");
   }
   else
```

```
  {
   printf("\nEnter Item value");
   scanf("%d",&ptr->data);

  if(head==NULL)
  {
     ptr->next = NULL;
     ptr->prev=NULL;

     head=ptr;
  }
  else
  {

     ptr->prev=NULL;
     ptr->next = head;
     head->prev=ptr;
     head=ptr;
  }
  printf("\nNode inserted\n");
}

}
void insertend()
```

```
{
  struct node *ptr,*temp;
  int data;
  ptr = (struct node *) malloc(sizeof(struct node));
  if(ptr == NULL)
  {
     printf("\nOVERFLOW");
  }
  else
  {
     printf("\nEnter value");
     scanf("%d",&ptr->data);

     if(head == NULL)
     {
        ptr->next = NULL;
        ptr->prev = NULL;
        head = ptr;
     }
     else
     {
       temp = head;
       while(temp->next!=NULL)
       {
```

```
            temp = temp->next;
          }
          temp->next = ptr;
          ptr ->prev=temp;
          ptr->next = NULL;
          }

        }
      printf("\nnode inserted\n");
     }
void insertpos()
{
   struct node *ptr,*temp;
   int loc,i;
   ptr = (struct node *)malloc(sizeof(struct node));
   if(ptr == NULL)
   {
      printf("\n OVERFLOW");
   }
   else
   {
      temp=head;
      printf("Enter the location");
      scanf("%d",&loc);
```

```
        for(i=0;i<loc;i++)
        {
           temp = temp->next;
           if(temp == NULL)
           {
              printf("\n There are less than %d elements", loc);
              return;
           }
        }
        printf("Enter value");
        scanf("%d",&ptr->data);

        ptr->next = temp->next;
        ptr -> prev = temp;
        temp->next = ptr;
        temp->next->prev=ptr;
        printf("\nnode inserted\n");
     }
  }
  void delbeg()
  {
     struct node *ptr;
     if(head == NULL)
     {
```

```
        printf("\n UNDERFLOW");
    }
    else if(head->next == NULL)
    {
        head = NULL;
        free(head);
        printf("\nnode deleted\n");
    }
    else
    {
        ptr = head;
        head = head -> next;
        head -> prev = NULL;
        free(ptr);
        printf("\nnode deleted\n");
    }

}
void delend()
{
    struct node *ptr;
    if(head == NULL)
    {
        printf("\n UNDERFLOW");
```

```
    }
    else if(head->next == NULL)
    {
        head = NULL;
        free(head);
        printf("\nnode deleted\n");
    }
    else
    {
        ptr = head;
        if(ptr->next != NULL)
        {
            ptr = ptr -> next;
        }
        ptr -> prev -> next = NULL;
        free(ptr);
        printf("\nnode deleted\n");
    }
}
void delpos()
{
    struct node *ptr, *temp;
    int val;
    printf("\n Enter the data after which the node is to be deleted : ");
```

```
    scanf("%d", &val);
    ptr = head;
    while(ptr -> data != val)
    ptr = ptr -> next;
    if(ptr -> next == NULL)
    {
        printf("\nCan't delete\n");
    }
    else if(ptr -> next -> next == NULL)
    {
        ptr ->next = NULL;
    }
    else
    {
        temp = ptr -> next;
        ptr -> next = temp -> next;
        temp -> next -> prev = ptr;
        free(temp);
        printf("\nnode deleted\n");
    }
}
void display()
{
    struct node *ptr;
```

```
    printf("\n printing values...\n");
    ptr = head;
    while(ptr != NULL)
    {
        printf("%d\n",ptr->data);
        ptr=ptr->next;
    }
}
void search()
{
    struct node *ptr;
    int item,i=0,flag;
    ptr = head;
    if(ptr == NULL)
    {
        printf("\nEmpty List\n");
    }
    else
    {
        printf("\nEnter item which you want to search?\n");
        scanf("%d",&item);
        while (ptr!=NULL)
        {
            if(ptr->data == item)
```

```
        {
            printf("\nitem found at location %d ",i+1);
            flag=0;
            break;
        }
        else
        {
            flag=1;
        }
        i++;
        ptr = ptr -> next;
      }
      if(flag==1)
      {
        printf("\nItem not found\n");
      }
   }
}
```

Result:

The program is executed successfully and output is verified.

EXPERIMENT NO:9

Date:

Aim: Write a program to implement graph traversal for BFS.

Program:

```
#include<stdio.h>
int V[50],E[50][50],queue[50],n,visited[50],rear=-1,front=-1;
void enqueue(int x)
{
if(front==-1)
front=0;
rear++;
queue[rear]=x;
visited[x]=1;
}
int dequeue()
{
int item=queue[front];
if(front==rear)
{
front=-1;
rear=-1;
}
else
front++;
```

```
return item;
}
void BFS()
{
int u,v,i;
enqueue(0);
while(front!=-1)
{
u=dequeue();
printf("Traversed: %d\n",V[u]);
for(i=0;i<n;i++)
{
if(E[u][i]==1)
{
v=i;
if(visited[v]==0)
enqueue(v);
}
}
}
}
void main()
{
int i,j,k=1,u,v,m;
```

```
printf("Enter the number of vertices");
scanf("%d",&n);
printf("Enter the vertices");
for(i=0;i<n;i++)
{
printf("Enter the vertex %d ",i+1);
scanf("%d",&V[i]);
visited[i]=0;
}
for(i=0;i<n;i++)
for(j=0;j<n;j++)
E[i][j]=0;
printf("Enter the number of edges");
scanf("%d",&m);
while(k<=m)
{
printf("Enter the edge %d ",k);
scanf("%d%d",&u,&v);
for(i=0;i<n;i++)
{
if(u==V[i])
{u=i;
break;
}}
```

```
for(i=0;i<n;i++)
{
if(v==V[i])
{v=i;
break;
}}
E[u][v]=1;
k++;
}
BFS();
}
```

Result:

The program is executed successfully and output is verified.

EXPERIMENT NO:10

Date:

Aim: Write a program to implement graph traversal for DFS.

Program:

```
#include<stdio.h>
int V[50],E[50][50],stack[50],n,visited[50],top=-1;
void push(int x)
{
top++;
stack[top]=x;
visited[x]=1;
}
int pop()
{
int item=stack[top];
top--;
return item;
}
void DFS()
{
int u,v,i;
push(0);
while(top!=-1)
{
```

```
u=pop();
printf("Traversed: %d\n",V[u]);
for(i=0;i<n;i++)
{
if(E[u][i]==1)
{
v=i;
if(visited[v]==0)
push(v);
}
}
}}
void main()
{
int i,j,k=1,u,v,m;
printf("Enter the number of vertices");
scanf("%d",&n);
printf("Enter the vertices");
for(i=0;i<n;i++)
{
printf("\nEnter the vertex %d ",i+1);
scanf("%d",&V[i]);
visited[i]=0;
}
```

```
for(i=0;i<n;i++)
for(j=0;j<n;j++)
E[i][j]=0;
printf("Enter the number of edges");
scanf("%d",&m);
while(k<=m)
{
printf("Enter the edge %d ",k);
scanf("%d%d",&u,&v);
for(i=0;i<n;i++)
{
if(u==V[i])
{
u=i;
break;
}
}
for(i=0;i<n;i++)
{
if(v==V[i])
{v=i;
break;
}
}
```

```
E[u][v]=1;

E[v][u]=1;

k++;

}

DFS();

}
```

Result:

The program is executed successfully and output is verified.

EXPERIMENT NO:11

Date:

Aim: Write a program to implement topological sorting algorithm.

Program:

```
#include <stdio.h>
void main(){
int i,j,k,n,a[10][10],indeg[10],flag[10],count=0;

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

printf("Enter the adjacency matrix:\n");
for(i=0;i<n;i++){
printf("Enter row %d\n",i+1);
for(j=0;j<n;j++)
scanf("%d",&a[i][j]);
}

for(i=0;i<n;i++){
      indeg[i]=0;
      flag[i]=0;
   }

   for(i=0;i<n;i++)
      for(j=0;j<n;j++)
         indeg[i]=indeg[i]+a[j][i];

   printf("\nThe topological order is:");
   while(count<n){
      for(k=0;k<n;k++){
         if((indeg[k]==0) && (flag[k]==0)){
            printf("%d ",(k+1));
```

```
            flag [k]=1;
        }

        for(i=0;i<n;i++){
            if(a[i][k]==1)
                indeg[k]--;
        }
    }

    count++;
  }

}
```

Result:

The program is executed successfully and output is verified.

EXPERIMENT NO:12

Date:

Aim: Write a program to implement Prim's algorithm.

Program:

```
#include<stdio.h>
#include<stdlib.h>
 #define infinity 9999
#define MAX 20
int G[MAX][MAX],spanning[MAX][MAX],n;
int prims();
int main()
{
int i,j,total_cost;
printf("Prim's Algorithm");
printf("\nEnter no. 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]);
total_cost=prims();
printf("\nspanning tree matrix:\n");
for(i=0;i<n;i++)
{
```

```
printf("\n");
for(j=0;j<n;j++)
printf("%d\t",spanning[i][j]);
}
printf("\n\nTotal cost of spanning tree=%d",total_cost);
return 0;
}
int prims()
{
int cost[MAX][MAX];
int u,v,min_distance,distance[MAX],from[MAX];
int visited[MAX],no_of_edges,i,min_cost,j;
//create cost[][] matrix,spanning[][]
for(i=0;i<n;i++)
for(j=0;j<n;j++)
{
if(G[i][j]==0)
cost[i][j]=infinity;
else
cost[i][j]=G[i][j];
spanning[i][j]=0;
}
//initialise visited[],distance[] and from[]
distance[0]=0;
```

```
visited[0]=1;
for(i=1;i<n;i++)
{
distance[i]=cost[0][i];
from[i]=0;
visited[i]=0;
}
min_cost=0; //cost of spanning tree
no_of_edges=n-1; //no. of edges to be added
while(no_of_edges>0)
{
//find the vertex at minimum distance from the tree
min_distance=infinity;
for(i=1;i<n;i++)
if(visited[i]==0&&distance[i]<min_distance)
{
v=i;
min_distance=distance[i];
}
u=from[v];
//insert the edge in spanning tree
spanning[u][v]=distance[v];
spanning[v][u]=distance[v];
no_of_edges--;
```

```
visited[v]=1;
//updated the distance[] array
for(i=1;i<n;i++)
if(visited[i]==0&&cost[i][v]<distance[i])
{
distance[i]=cost[i][v];
from[i]=v;
}
min_cost=min_cost+cost[u][v];
}
return(min_cost);
}
```

Result:

The program is executed successfully and output is verified.

EXPERIMENT NO:13

Date:

Aim: Write a program to implement Kruskal's algorithm.

Program:

```
#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("Kruskal's Algorithm");
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);
}
```

Result:

The program is executed successfully and output is verified.

EXPERIMENT NO:14

Date:

Aim: Write a program to implement single source shortest path algorithm.

Program:

```
#include<stdio.h>
#include<conio.h>
#define INFINITY 9999
#define MAX 10
void dijkstra(int G[MAX][MAX],int n,int startnode);
 int main()
{
int G[MAX][MAX],i,j,n,u;
printf("Enter no. 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]);
printf("\nEnter the starting node:");
scanf("%d",&u);
dijkstra(G,n,u);
return 0;
}
void dijkstra(int G[MAX][MAX],int n,int startnode)
```

```
{
int cost[MAX][MAX],distance[MAX],pred[MAX];
int visited[MAX],count,mindistance,nextnode,i,j;
//pred[] stores the predecessor of each node
//count gives the number of nodes seen so far
//create the cost matrix
for(i=0;i<n;i++)
for(j=0;j<n;j++)
if(G[i][j]==0)
cost[i][j]=INFINITY;
else
cost[i][j]=G[i][j];
//initialize pred[],distance[] and visited[]
for(i=0;i<n;i++)
{
distance[i]=cost[startnode][i];
pred[i]=startnode;
visited[i]=0;
}
distance[startnode]=0;
visited[startnode]=1;
count=1;
while(count<n-1)
{
```

```
mindistance=INFINITY;
//nextnode gives the node at minimum distance
for(i=0;i<n;i++)
if(distance[i]<mindistance&&!visited[i])
{
mindistance=distance[i];
nextnode=i;
}
//check if a better path exists through nextnode
visited[nextnode]=1;
for(i=0;i<n;i++)
if(!visited[i])
if(mindistance+cost[nextnode][i]<distance[i])
{
distance[i]=mindistance+cost[nextnode][i];
pred[i]=nextnode;
}
count++;
}

//print the path and distance of each node
for(i=0;i<n;i++)
if(i!=startnode)
{
```

```
printf("\nDistance of node%d=%d",i,distance[i]);

printf("\nPath=%d",i);

j=i;

do

{

j=pred[j];

printf("<-%d",j);

}while(j!=startnode);

}

}
```

Result:

The program is executed successfully and output is verified.