***ASSIGNMENT-14***

1. ***WAP to construct a Graph using linked representation and implement the following:  
   a. Create an edge between two nodes.  
   b. Remove an edge between two nodes.  
   c. Degree of a particular node.  
   d. Create a new node.  
   e. Remove an existing node.***

#include<stdio.h>

#include<stdlib.h>

struct graphnode{

struct graphnode \*point,\*next;

int info;

};

typedef struct graphnode Graph;

Graph \*create\_edge(Graph \*start,int x,int y);

Graph \*delete\_edge(Graph \*start,int x,int y);

void find\_degree(Graph \*start,int x);

Graph \*create\_node(Graph \*start,int ch);

Graph \*delete\_node(Graph \*start,int x);

void display(Graph \*start);

int main()

{

int n,i,ch,x,y;

printf("Enter number of nodes : ");

scanf("%d",&n);

Graph \*start = NULL,\*p=start;

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

{

Graph \*q = (Graph \*)malloc(sizeof(Graph));

q->info = i;

q->next = q->point = NULL;

if(p==NULL)

start = p = q;

else

{

p->next = q;

p = p->next;

}

}

printf("1.Directed Graph\n");

printf("2.Undirected Graph\n");

scanf("%d",&ch);

if(ch==1)

printf("Enter edges as V1 V2(v1-->v2)\n");

else

printf("Enter edges as V1 V2\n");

printf("Enter -1 to stop\n");

while(1)

{

scanf("%d%d",&x,&y);

if(x==-1)

break;

start = create\_edge(start,x,y);

if(ch==2)

start = create\_edge(start,y,x);

}

printf("1. Create an edge between two nodes.\n");

printf("2. Remove an edge between two nodes.\n");

printf("3. Degree of a particular node.\n");

printf("4. Create a new node.\n");

printf("5. Remove an existing node.\n");

printf("6. Display adjancency list.\n");

printf("7. Exit\n");

while(1)

{

scanf("%d",&i);

switch(i)

{

case 1:

printf("Enter vertices having edge :");

scanf("%d%d",&x,&y);

start = create\_edge(start,x,y);

break;

case 2:

printf("Enter vertices having edge :");

scanf("%d%d",&x,&y);

start = delete\_edge(start,x,y);

if(ch==2)

start = delete\_edge(start,y,x);

break;

case 3:

printf("Enter Node number :");

scanf("%d",&x);

find\_degree(start,x);

break;

case 4:

start = create\_node(start,ch);

break;

case 5:

printf("Enter node which is to be deleted :");

scanf("%d",&x);

start = delete\_node(start,x);

break;

case 6:

display(start);

break;

case 7:

exit(1);

}

}

return 0;

}

Graph \*create\_edge(Graph \*start,int x,int y)

{

Graph \*p = start,\*q=start;

while(q->info!=y)

q = q->next;

while(p->info!=x)

p = p->next;

Graph \*r = (Graph \*)malloc(sizeof(Graph));

r->info = y;

r->next = NULL;

r->point = q;

if(p->point==NULL)

p->point = r;

else

{

q = p->point;

while(q->next!=NULL)

q = q->next;

q->next = r;

}

return start;

}

Graph \*delete\_edge(Graph \*start,int x,int y)

{

Graph \*p = start,\*q;

while(p->info!=x)

p = p->next;

q = p->point;

if(q!=NULL)

{

if(q->next==NULL && q->info==y)

p->point = NULL;

else if(q->next!=NULL)

{

while(q!=NULL && q->info!=y)

{

p = q;

q = q->next;

}

if(q!=NULL)

p->next = q->next;

free(q);

}

}

return start;

}

void find\_degree(Graph \*start,int x)

{

Graph \*p = start,\*q;

int in=0,out=0;

while(p!=NULL)

{

q = p->point;

while(q!=NULL)

{

if(p->info==x)

out++;

if(q->info == x)

in++;

q = q->next;

}

p = p->next;

}

printf("In-Degree : %d\nOut-Degree : %d\n",in,out);

}

Graph \*create\_node(Graph \*start,int ch)

{

Graph \*p = start;

while(p->next!=NULL)

p = p->next;

Graph \*temp = (Graph \*)malloc(sizeof(Graph));

temp->info = p->info + 1;

temp->point = temp->next = NULL;

p->next = temp;

printf("New node %d is created\n",temp->info);

printf("Enter edges connected to node %d (-1 to stop)\n",temp->info);

int x,y;

while(1)

{

scanf("%d%d",&x,&y);

if(x==-1)

break;

start = create\_edge(start,x,y);

if(ch==2)

start = create\_edge(start,y,x);

}

return start;

}

Graph \*delete\_node(Graph \*start,int x)

{

Graph \*p=start,\*q,\*r;

if(p->info==x)

start = start->next;

else

{

while(p!=NULL && p->info!=x)

{

q=p;

p = p->next;

}

if(p==NULL)

{

printf("Entered Node is not found\n");

return start;

}

q->next = p->next;

}

q=p->next;

p = start;

while(p!=NULL)

{

start = delete\_edge(start,p->info,x);

p = p->next;

}

while(q!=NULL)

{

q->info = q->info-1;

q = q->next;

}

free(p);

return start;

}

void display(Graph \*start)

{

Graph \*p = start,\*q;

while(p!=NULL)

{

printf("%d ",p->info);

q = p->point;

while(q!=NULL)

{

printf("--> %d",q->info);

q = q->next;

}

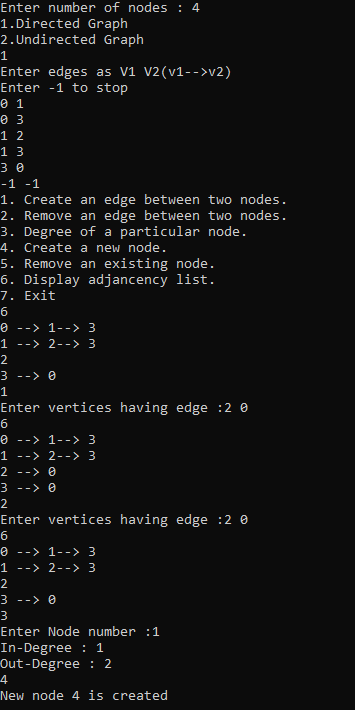
printf("\n");

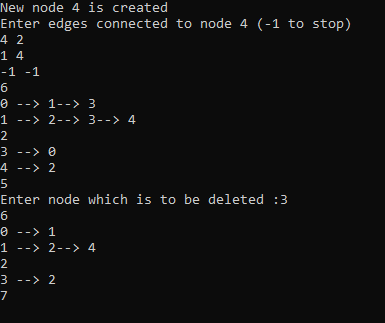
p = p->next;

}

}

***OUTPUT :***





1. ***WAP to construct a Minimum Spanning Tree using   
   a. Krushkal’s Algorithm.***

#include<stdio.h>

#define SIZE 100

struct edge{

int u,v,w; // u = source , v = destination , w = weight

};

typedef struct edge Edge;

struct edgelist

{

Edge data[SIZE];

int n;

};

typedef struct edgelist EDGELIST;

EDGELIST Edgelist,spanlist;

int Graph[SIZE][SIZE],n;

void Krushkal();

void sort();

int find(int belong[],int i);

void union\_func(int belong[],int u,int v);

void print();

int main()

{

int i,j;

printf("Enter number of nodes : ");

scanf("%d",&n);

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

for(j=0;j<n;j++)

scanf("%d",&Graph[i][j]);

Krushkal();

print();

return 0;

}

void Krushkal()

{

int belong[SIZE],i,j;

Edgelist.n=0;

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

belong[i]=i;

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

for(j=0;j<n;j++)

if(Graph[i][j]!=0)

{

Edgelist.data[Edgelist.n].u = i;

Edgelist.data[Edgelist.n].v = j;

Edgelist.data[Edgelist.n].w = Graph[i][j];

(Edgelist.n)++;

}

sort();

spanlist.n=0;

int cnt=0,x,y;

for(i=0;i<Edgelist.n && cnt!=n-1;i++)

{

x = find(belong,Edgelist.data[i].u);

y = find(belong,Edgelist.data[i].v);

if(x!=y)

{

spanlist.data[spanlist.n] = Edgelist.data[i];

(spanlist.n)++;

union\_func(belong,x,y);

cnt++;

}

}

}

void union\_func(int belong[],int u,int v)

{

int i;

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

if(belong[i]==v)

belong[i] = u;

}

int find(int belong[],int i)

{

return belong[i];

}

void sort()

{

int i,j,k,temp;

for(i=0;i<Edgelist.n;i++)

{

k = i;

for(j=i+1;j<Edgelist.n;j++)

if(Edgelist.data[j].w < Edgelist.data[k].w)

k = j;

if(k!=i)

{

Edge temp = Edgelist.data[k];

Edgelist.data[k] = Edgelist.data[i];

Edgelist.data[i] = temp;

}

}

}

void print()

{

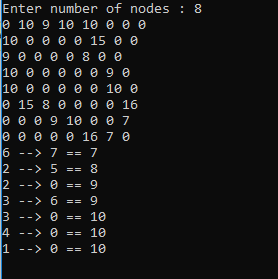
int i,j;

for(i=0;i<spanlist.n;i++)

printf("%d --> %d == %d\n",spanlist.data[i].u,spanlist.data[i].v,spanlist.data[i].w);

}

***OUTPUT :***



***b. Prim’s Algorithm.***

#include<stdio.h>

#include<limits.h>

#define SIZE 100

void prim();

void display(int parent[],int key[]);

int graph[SIZE][SIZE],n;

int main()

{

int i,j;

printf("Enter number of nodes : ");

scanf("%d",&n);

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

for(j=0;j<n;j++)

scanf("%d",&graph[i][j]);

prim();

return 0;

}

void prim()

{

int parent[SIZE],key[SIZE],visited[SIZE];

int i;

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

{

key[i] = INT\_MAX;

visited[i] = 0;

}

key[0] = 0;

parent[0]=-1;

int cnt=0;

while(cnt!=n-1)

{

int j,min = INT\_MAX,k;

for(j=0;j<n;j++)

if(visited[j]==0 && key[j]<min)

{

k = j;

min = key[j];

}

visited[k]=1;

for(j=0;j<n;j++)

if(graph[k][j] && !visited[j] && (graph[k][j] < key[j]))

{

parent[j]=k;

key[j] = graph[k][j];

}

cnt++;

}

display(parent,key);

}

void display(int parent[],int key[])

{

int i;

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

printf("%d --> %d == %d\n",parent[i],i,key[i]);

}

***OUTPUT :***

