UNIT 5 PROGRAMS

HASHING WITH LINEAR PROBING:

```
#include<iostream>
using namespace std;
class Hash
{
	int *arr;
	int *inserted;
	int n,i=0;
	public:
		Hash()
		{

			cout<<"Enter the max value of array:";
			cin>>n;
			arr=new int[n];
			inserted=new int[n];
			for(i=0;i<n;i++)
			{
				arr[i]=0;	//initialising to zero to avoid garbage values
				inserted[i]=0;
			}
		}
		void linearprobing(int key)
		{
			int h_val=key%n;	//using hash funstion h(key)=key mod n to attain mapping between key and the bucket value
			for(i=0;i<n;i++)
			{
				if(h_val==i&&inserted[i]==0)
				{
					arr[i]=key;
					inserted[i]=1;
					cout<<"Element is inserted "<<endl;
					break;
				}
				else if(h_val==i&&inserted[i]!=0)
				{
					int i=h_val;
```

```
                while(i<=n)
                {
                        if(arr[i]==0&&inserted[i]==0)

                        {
                                arr[i]=key;
                                inserted[i]=1;
                                cout<<"Element is inserted "<<endl;
                                break;
                        }
                        i++;
                        i=i%n;
                }
        }
        else;
    }
        if(i==n)

    {
       cout<<"Hash table is full - no more elements are allowed "<<endl;
    }
}
void display()
{
    for(int i=0;i<n;i++)
    {
        cout<<arr[i]<<endl;
    }
}
void search()
{
   int found=-1;
   int ele,i;

   cout<<"Enter the elment you want to search : ";

   cin>>ele;

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

   {

      if(arr[i]==ele)
```

```
                {
                    cout<<"Element is found at index: "<<i<<endl;
                    found=1;
                }
            }
            if(found==-1)
            {
                cout<<"Element is not found"<<endl;
            }
        }
};
int main()
{
    Hash h;
    int ch,key;
    while(1)
    {
        cout<<"1.Inserting key value to hash table"<<endl<<"2.Display"<<endl<<"3.Search"<<endl<<"4.Exit"<<endl;
        cout<<"Enter your choice:";
        cin>>ch;
        switch(ch)
        {
```

```
                    case 1: cout<<"Enter the key value:";
                            cin>>key;
                            h.linearprobing(key);
                            break;
                    case 2: h.display();
                            break;
                    case 3: h.search();
                            break;
                    case 4: return 0;
                    default:cout<<endl<<"WRONG CHOICE";
                }

        }
}
```

HASHING WITH SEPARATE CHAINING:

```
#include <iostream>
const int T_S = 200;

using namespace std;

struct HashTableEntry
{
  int v, k;
  HashTableEntry *n;
  HashTableEntry *p;
  HashTableEntry(int k, int v) {
    this->k = k;
    this->v = v;
```

```
    this->n = NULL;
  }
};
class HashMapTable {
  public:
    HashTableEntry **ht, **top;
    HashMapTable() {
      ht = new HashTableEntry*[T_S];
      for (int i = 0; i < T_S; i++)
        ht[i] = NULL;
    }
    int HashFunc(int key) {
      return key % T_S;
    }
    void Insert(int k, int v) {
      int hash_v = HashFunc(k);
      HashTableEntry* p = NULL;
      HashTableEntry* en = ht[hash_v];
      while (en!= NULL) {
        p = en;
        en = en->n;
      }
      if (en == NULL) {
        en = new HashTableEntry(k, v);
        if (p == NULL) {
          ht[hash_v] = en;
        } else {
          p->n = en;
        }
      } else {
        en->v = v;
      }
    }
    void Remove(int k)
    {
      int hash_v = HashFunc(k);
      HashTableEntry* en = ht[hash_v];
      HashTableEntry* p = NULL;
      if (en == NULL || en->k != k) {
        cout<<"No Element found at key "<<k<<endl;
        return;
      }
      while (en->n != NULL) {
        p = en;
```

```
      en = en->n;
    }
    if (p != NULL) {
      p->n = en->n;
    }
    delete en;
    cout<<"Element Deleted"<<endl;
  }
  void SearchKey(int k) {
    int hash_v = HashFunc(k);
    bool flag = false;
    HashTableEntry* en = ht[hash_v];
    if (en != NULL) {
      while (en != NULL) {
        if (en->k == k) {
          flag = true;
        }
        if (flag) {
          cout<<"Element found at key "<<k<<": ";
          cout<<en->v<<endl;
        }
        en = en->n;
      }
    }
    if (!flag)
      cout<<"No Element found at key "<<k<<endl;
  }
  ~HashMapTable() {
    delete [] ht;
  }
};
int main() {
  HashMapTable hash;
  int k, v;
  int c;
  while (1) {
    cout<<"1.Insert element into the table"<<endl;
    cout<<"2.Search element from the key"<<endl;
    cout<<"3.Delete element at a key"<<endl;
    cout<<"4.Exit"<<endl;
    cout<<"Enter your choice: ";
    cin>>c;
    switch(c) {
      case 1:
```

```
        cout<<"Enter element to be inserted: ";
        cin>>v;
        cout<<"Enter key at which element to be inserted: ";
        cin>>k;
        hash.Insert(k, v);
      break;
      case 2:
        cout<<"Enter key of the element to be searched: ";
        cin>>k;
        hash.SearchKey(k);
      break;
      case 3:
        cout<<"Enter key of the element to be deleted: ";
        cin>>k;
        hash.Remove(k);
      break;
      case 4:
        exit(1);
      default:
        cout<<"\nEnter correct option\n";
    }
  }
  return 0;
}
```

PRIMS ALGORITHM:

```
#include <iostream>
#include <conio.h>
#define ROW 7
#define COL 7
#define infi 5000  //infi for infinity
using namespace std;
class prims
{
  int graph[ROW][COL],nodes;
  public:
  prims();
  void createGraph();
  void primsAlgo();
};

prims :: prims(){
```

```
    for(int i=0;i<ROW;i++)
      for(int j=0;j<COL;j++)
     graph[i][j]=0;
}

void prims :: createGraph(){
   int i,j;
   cout<<"Enter Total Nodes : ";
   cin>>nodes;
   cout<<"\nEnter Adjacency Matrix : \n";
   for(i=0;i<nodes;i++)
      for(j=0;j<nodes;j++)
      cin>>graph[i][j];

   //Assign infinity to all graph[i][j] where weight is 0.
   for(i=0;i<nodes;i++){
      for(j=0;j<nodes;j++){
        if(graph[i][j]==0)
       graph[i][j]=infi;
      }
   }
}

void prims :: primsAlgo(){
   int selected[ROW],i,j,ne; //ne for no. of edges
   int min,x,y;

   for(i=0;i<nodes;i++)
     selected[i]=false;

   selected[0]=true;
   ne=0;

   while(ne < nodes-1){
     min=infi;

     for(i=0;i<nodes;i++)
     {
       if(selected[i]==true){
      for(j=0;j<nodes;j++){
         if(selected[j]==false){
            if(min > graph[i][j])
            {
            min=graph[i][j];
```

```
            x=i;
            y=j;
            }
          }
        }
         }
      }
      selected[y]=true;
      cout<<"\n"<<x+1<<" --> "<<y+1;
      ne=ne+1;
    }
}

int main(){
   prims MST;

   cout<<"\nPrims Algorithm to find Minimum Spanning Tree\n";
   MST.createGraph();
   MST.primsAlgo();
   return 0;
}
```

KRUSKAL'S ALOGORITHM:

```
#include<iostream>
#include<string.h>
using namespace std;

class Graph
{
  char vertices[10][10];
  int cost[10][10],no;
public:
  Graph();
  void creat_graph();
  void display();
  int Position(char[]);
  void kruskal_algo();
};

/* Initialzing adj matrix with 999 */
/* 999 denotes infinite distance */
Graph::Graph()
```

```
{
  no=0;
  for(int i=0;i<10;i++)
  for(int j=0;j<10;j++)
  {
    cost[i][j]=999;
   }
}

/* Taking inputs for creating graph */
void Graph::creat_graph()
{
  char ans,Start[10],End[10];
  int wt,i,j;
  cout<<"Enter the number of vertices: ";
  cin>>no;
  cout<<"\nEnter the vertices: ";
  for(i=0;i<no;i++)
        cin>>vertices[i];
  do
  {
    cout<<"\nEnter Start and End vertex of the edge: ";
    cin>>Start>>End;
    cout<<"Enter weight: ";
    cin>>wt;
    i=Position(Start);
    j=Position(End);
    cost[i][j]=cost[j][i]=wt;
    cout<<"\nDo you want to add more edges (Y=YES/N=NO)? : ";  /* Type 'Y' or 'y' for YES and
'N' or 'n' for NO */
    cin>>ans;
  }while(ans=='y' || ans=='Y');
}

/* Displaying Cost matrix */
void Graph::display()
{
  int i,j;
  cout<<"\n\nCost matrix: ";
  for(i=0;i<no;i++)
  {
    cout<<"\n";
    for(j=0;j<no;j++)
    cout<<"\t"<<cost[i][j];
```

```
 }
}

/* Retrieving position of vertices in 'vertices' array */
int Graph::Position(char key[10])
{
 int i;
 for(i=0;i<10;i++)
 if(strcmp(vertices[i],key)==0)
  return i;
return -1;
}

void Graph::kruskal_algo()
{
 int i,j,v[10]={0},x,y,Total_cost=0,min,gr=1,flag=0,temp,d;

 while(flag==0)
 {
  min=999;
   for(i=0;i<no;i++)
  {
    for(j=0;j<no;j++)
      {
     if(cost[i][j]<min)
        {
       min=cost[i][j];
       x=i;
       y=j;
        }
      }
   }

  if(v[x]==0 && v[y]==0)
  {
    v[x]=v[y]=gr;
    gr++;
  }
  else if(v[x]!=0 && v[y]==0)
    v[y]=v[x];
  else if(v[x]==0 && v[y]!=0)
    v[x]=v[y];
  else
  {
```

```
      if(v[x]!=v[y])
      {
        d=v[x];
        for(i=0;i<no;i++)
        {
          if(v[i]==d)
          v[i]=v[y];
        }//end for
      }
    }

    cost[x][y]=cost[y][x]=999;
    Total_cost=Total_cost+min;      /* calculating cost of minimum spanning tree */
    cout<<"\n\t"<<vertices[x]<<"\t\t"<<vertices[y]<<"\t\t"<<min;

       temp=v[0]; flag=1;
       for(i=0;i<no;i++)
       {
         if(temp!=v[i])
         {
         flag=0;
         break;
         }
       }
  }
cout<<"\nTotal cost of the tree= "<<Total_cost;
}

int main()
{
  Graph g;
  g.creat_graph();
  g.display();

  cout<<"\n\n\nMinimum Spanning tree using kruskal algo=>";
  cout<<"\nSource vertex\tDestination vertex\tWeight\n";
  g.kruskal_algo();

return 0;
}
```

BFS AND DFS IN GRAPHS

```
#include<iostream>
#include<vector>
using namespace std;
class queue
{
	int *Q,n,front,rear;
	public:
	queue()
	{
		cout<<"Enter the maximum size of the queue for BFS traversal:";
		cin>>n;
		Q = new int[n];
		front=0;
		rear=-1;
	}
	bool isempty();
	bool isfull();
	void enqueue(int x);
	int dequeue();
	int firstelement();
};
bool queue::isempty()
{
	if(front>rear)
	return true;
	else
	return false;
}
bool queue::isfull()
{
	if(front==0&&rear==n-1)
	return true;
	else
	return false;
}
void queue::enqueue(int x)
{
	if(!isfull())
	Q[++rear]=x;
	else
	cout<<"Queue is full";
```

```
}
int queue::dequeue()
{
	if(!isempty())
	{
		int x=Q[front];
		front++;
		return x;
	}
	else
		cout<<"\n Queue is empty";
}
int queue::firstelement()
{
	return Q[front];
}
template<class T>
class Stack
{
	int top,N;
	T *S;
	public:
	Stack()
	{
		cout<<endl<<"Enter the maximum size of the stack for DFS traversal:";
		cin>>N;
		S=new T[N];
		top=-1;
	}
	~Stack() { delete S;}
	bool isEmpty();
	bool isFull();
	void push(T x);
	T pop();
	T topElement();
};
template<class T>
bool Stack<T>::isEmpty()
{
	if(top==-1) return true;
	return false;
}
template<class T>
bool Stack<T>::isFull()
```

```
{
	if(top==N-1) return true;
	return false;
}
template<class T>
void Stack<T>::push(T x)
{
	if(!isFull())
		S[++top]=x;
	else
		cout<<endl<<"Stack is FULL";
}
template<class T>
T Stack<T>::pop()
{
	if(!isEmpty())
		return S[top--];
	else
		cout<<endl<<"Stack is empty";
}
template<class T>
T Stack<T>::topElement()
{
	if(!isEmpty())
		return S[top];
	else
		cout<<endl<<"Stack is empty";
}
void edge(vector<int>adj[],int u,int v)
{
  adj[u].push_back(v);
}
void bfs(int s,vector<int>adj[],bool visit[])
{
	queue q;
	q.enqueue(s);
	visit[s]=true;
	while(!q.isempty())
	{
		int u=q.firstelement();
	cout<<u<<" ";
	q.dequeue();
	for(int i=0;i<adj[u].size();i++)
		{
```

```
                    if(!visit[adj[u][i]])
                            {
                    q.enqueue(adj[u][i]);
                    visit[adj[u][i]]=true;
                    }
                    }
          }
}
void dfs(int s,vector<int>adj[],bool visit[])
{
          Stack<int> st;
          st.push(s);
          visit[s]=true;
          while(!st.isEmpty())
          {
                    int u=st.topElement();
          cout<<u<<" ";
          st.pop();
          for(int i=0;i<adj[u].size();i++)
                    {
                    if(!visit[adj[u][i]])
                            {
                    st.push(adj[u][i]);
                    visit[adj[u][i]]=true;
         }
             }
          }
}
int main()
{
  vector<int>adj[5];
  bool visit[5];
  //initially all node are unvisited
  for(int i=0;i<5;i++)
  {
    visit[i]=false;
  }
  //input for edges
  edge(adj,0,2);
  edge(adj,0,1);
  edge(adj,1,3);
  edge(adj,2,0);
  edge(adj,2,3);
  edge(adj,2,4);
```

```
 //cout<<"BFS traversal is"<<"  ";
 //call bfs funtion
 bfs(0,adj,visit);
 cout<<endl;
 //again initialise all node unvisited for dfs
 for(int i=0;i<5;i++)
 {
  visit[i]=false;
 }
 //cout<<"DFS traversal is"<<"  ";
 //call dfs function
 dfs(0,adj,visit);
}
```

ADJACENCY LIST

```
#include<iostream>

#include<list>

#include<iterator>

using namespace std;

void displayAdjList(list<int> adj_list[], int v) {

   for(int i = 0; i<v; i++) {

      cout << i << "--->";

      list<int> :: iterator it;

      for(it = adj_list[i].begin(); it != adj_list[i].end();
++it) {

         cout << *it << " ";

      }

      cout << endl;

   }

}

void add_edge(list<int> adj_list[], int u, int v) {    //add v
into the list u, and u into list v

   adj_list[u].push_back(v);
```

```
   adj_list[v].push_back(u);
}
main(int argc, char* argv[]) {
   int v = 6;    //there are 6 vertices in the graph
   //create an array of lists whose size is 6
   list<int> adj_list[v];
   add_edge(adj_list, 0, 4);
   add_edge(adj_list, 0, 3);
   add_edge(adj_list, 1, 2);
   add_edge(adj_list, 1, 4);
   add_edge(adj_list, 1, 5);
   add_edge(adj_list, 2, 3);
   add_edge(adj_list, 2, 5);
   add_edge(adj_list, 5, 3);
   add_edge(adj_list, 5, 4);
   displayAdjList(adj_list, v);
}
```

Output

```
0--->4 3
1--->2 4 5
2--->1 3 5
3--->0 2 5
4--->0 1 5
5--->1 2 3 4
```

ADJACENCY MATRIX

```
#include<iostream>
using namespace std;
int vertArr[20][20]; //the adjacency matrix initially 0
int count = 0;
```

```
void displayMatrix(int v) {
   int i, j;
   for(i = 0; i < v; i++) {
      for(j = 0; j < v; j++) {
         cout << vertArr[i][j] << " ";
      }
      cout << endl;
   }
}
void add_edge(int u, int v) {        //function to add edge into
the matrix
   vertArr[u][v] = 1;
   vertArr[v][u] = 1;
}
main(int argc, char* argv[]) {
   int v = 6;    //there are 6 vertices in the graph
   add_edge(0, 4);
   add_edge(0, 3);
   add_edge(1, 2);
   add_edge(1, 4);
   add_edge(1, 5);
   add_edge(2, 3);
   add_edge(2, 5);
   add_edge(5, 3);
   add_edge(5, 4);
   displayMatrix(v);
}
```

Output

```
0 0 0 1 1 0
0 0 1 0 1 1
```

```
0 1 0 1 0 1
1 0 1 0 0 1
1 1 0 0 0 1
0 1 1 1 1 0
```