NRI INSTITUTE OF INFORMATION SCIENCE & TECHNOLOGY BHOPAL

DEPARTMENT OF INFORMATION TECHNOLOGY

LAB MANUAL

DATA STRUCTURE LAB

(IT-303)

INFORMATION TECHNOLOGY (IT)

Lab Assignment#1

Objective: To learn the concepts ofArray and its operations.
Problem statement: Implement Array operations – Insertion, Deletion, and Display.
Underlying concepts: Array.
An array is a collection of homogeneous elements i.e., all the elements are of an array are of the same type.
To define an array, you tell the compiler to set aside storage for a given number of dataitems of a specified type. You also tell the compiler the name of the array. Here's an example of an array definition that creates storage for four integers:

```
int age[4];
```

The int specifies the type of data to be stored, age is the name of the array, and 4 is thesize of the array; that is, the maximum number of variables of type int that it will hold. Brackets [] surround the size.
Each variable stored in an array is called an element. The elements are numbered. These numbers are called index numbers or indexes or subscripts. The index of the firstarray element is 0, the index of the second is 1, and so on. If the size of the array is n, the last element has the index n-1. For example, in the age array, which has a size of 4,the elements are numbered 0, 1, 2, and 3. This numbering can be the source of some confusion. Keep in mind that the last element in an array has an index one less than thesize of the array.
You refer to individual array elements using their index numbers and the array name. The first element is called age [0], the second is age [1], and so on. You can make statements such as age[2] = 23;
Insertion: To insert a data item in an array, we specify the location of the data item to be inserted.
Suppose we have an array whose maximum size is 10 and the current elements are:

12	23	14	25	14	45	35	23		
0	1	2	3	4	5	6	7	8	9

And it is required to insert '16' at index '3'. Since there is no space available at index three we need to move all the elements from index '3' to '7' upward '4' to '8' respectively.

12	23	14		25	14	45	35	23	
0	1	2	3	4	5	6	7	8	9

Now the element can be inserted at index '3'.

Deletion: To delete a data item from an array, we specify the location of the data item to be deleted.
Suppose we have an array whose maximum size is 10 and the current elements are:

12	23	14	25	14	45	35	23		
0	1	2	3	4	5	6	7	8	9

And it is required to delete element at index '3'. Since after deletion there will be a space available at index '3' we need to move all the elements from index '4' to '7' downward '4' to '8' respectively.

12	23	25	14	45	35	23			
0	1	2	3	4	5	6	7	8	9

Display:

Arrays are useful primarily because the individual variables stored in an array can beaccessed using an index number. This makes it easy to cycle through the array, accessing one variable after another. We can use a for loop to do this.

Code:

```
#include<iostream.h>
#include<conio.h> const
int max=10; void main()
{
	int choice, arr[max],i,cur=0, x, pos;do
	{
		cout<<"1 - INSERTION\n2 - DELETION\n3 - DISPLAY\n4 - EXIT\n";
		cin>>choice;
		switch(choice)
		{
			case 1:
				if(cur==max) // Check whether array is full.cout<<"Array is
					full\n";
				else
				{
					cout<<"Enter the element: ";cin>>x;
					cout<<"Enter the index ("<<0<<"-"<<cur<<") : ";
					cin>>pos;
					// move elements upward.
					for(i=cur-1;i>=pos;i--)
						arr[i+1]=arr[i]; arr[pos]=x;//insert
					the elemtn.cur++;

				}
				break;
			case 2:
				if(cur==0) // Check whether array is empty.cout<<"Array is
					empty\n";
				else
				{
					cout<<"Enter the index: ";cin>>pos;
					x=arr[pos];
					// move elements upward.
					for(i=pos;i<cur;i++)
						arr[i-1]=arr[i];cur--;

				}
				break;
			case 3:
```

```
					for(i=0;i<cur;i++)
						cout<<arr[i]<<" ";
					cout<<endl;
					break;
				default:
					break;
			}
		}while(choice!=4);
}
```

Analysis:

Conclusion
:

Lab Assignment#2

Objective: To learn the concepts oflinear search.

Problem statement: Implement linear search.

Underlying concepts: Linear search.

Linear search is an algorithm for locating the position of an element in an array. It inspects the elements of the list starting from first element of the array: if equal to the sought value, then the position has been found; otherwise, the next element is inspected for further searching and so on. It finds the sought value if it exists in the list or if not determines "not present", in linear time. The process continues till the end of the array is reached.

Code:

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

int linsearch(int*, int, int);void

main()
{
        int a[10], x, loc;
        cout<<"Enter 10 elemets of the array:\n";for(int i=0;
        i<10;i++)
                cin>>a[i];
        cout<<"Enter the element to be searched : ";cin>>x;
        loc=linsearch(a, 10, x);
        if(loc==-1)
                cout<<"Element not in the list\n";
        else
                cout<<"Element found at "<<loc<<endl;
}

int linsearch(int*a, int n, int x)
{
        int i=0;


        while( i<n &&(a[i]!=x))i++;
        if(i<n)
                return i;
        else
                return -1;
}
```

Analysis:
Conclusion
:

Lab Assignment#3

Objective: To learn the concepts ofbinary search.
Problem statement: Implement binary search.
Underlying concepts: Binary search.
Binary search is an algorithm for locating the position of an element in a sorted list. It inspects the middle element of the sorted list: if equal to the sought value, then the position has been found; otherwise, the upper halfor lower half is chosen for further searching based on whether the sought value is greater than or less than the middle element. The method reduces thenumber of elements needed to be checked by a factor of two each time, and finds the sought value if it exists in the list or if not determines "not present", in logarithmic time.
Code:

```
#include<iostream.h>
#include<conio.h>
int binsearch(int*, int, int, int);void main()
{
	int a[10], x, loc, low, high; cout<<"Enter 10
	elemets of the array:\n";for(int i=0; i<10;i++)
		cin>>a[i];
	cout<<"Enter the element to be searched : ";cin>>x;
	loc=binsearch(a, 0, 9, x);
	if(loc==-1)
		cout<<"Element not in the list\n";
	else
		cout<<"Element found at "<<loc<<endl;
}

int binsearch(int*a, int low, int high, int x)
{
	int mid; mid=(low+high)/2;

	while(a[mid]!=x)
	{
		if(a[mid]<x)
			low=mid+1;
		else
			high=mid-1;
		mid=(low+high)/2;
	}

	if(a[mid]==x)
		return mid;
	else
		return -1;
}
```

Analysis:
Conclusion
:

Lab Assignment#4

Objective: To learn the concepts stack and its operations.
Problem statement: Implement Stack.
Underlying concepts: Stack.

A stack is a data structure in which all the insertions and deletions are made at one end. This end is called the TOP of the stack. The insertion operation is called PUSH and the deletion operation is called POP. Stack has LIFO property i.e., Last In First Out. The element inserted last will be the first one to be removed.
Stack can be implemented as an array, where initially the top is set -1 to indicate empty stack.
When we push an element into the stack we check whether the stackis full. If it is not full, we increment the value of top and storethe element at the top position.
Top++;
Stack [top] =x;
When we pop an element from the stack we check whether the stack isempty. If it is not empty, we store the element at the top positionin a variable and decrement the value of top.
X= Stack [top];
Top--;

Code:

```
#include<iostream.h>
#include<conio.h> const
int max=10; void main()
{
        int choice, stack[max],i,top=-1, x;do
        {
                cout<<"1 - PUSH\n2 - POP\n3 - DISPLAY\n4 - EXIT\n";
                cin>>choice;
                switch(choice)
                {
                        case 1:
                                if(top==max-1) // Check whether array is full.cout<<"Stack is
                                        full\n";
                                else
                                {
                                        cout<<"Enter the element: ";cin>>x;
                                        stack[++top]=x;//insert the element.

                                }
                                break;
                        case 2:
                                if(top==-1) // Check whether array is empty.
```

```
                        cout<<"Stack is empty\n";
                    else
                        x=stack[top--];
                    break;
                case 3:
                    for(i=0;i<=top;i++)
                        cout<<stack[i]<<" ";
                    cout<<endl;
                    break;
                default:
                    break;
            }
    }while(choice!=4);
}
```

Analysis:

Conclusion

:

Lab Assignment#5

Objective: To learn the concepts queue and its operations.

Problem statement: Implement Queue.

Underlying concepts: Queue.

A queue is a data structure in which all the insertions are made at one end (called rear or tail) and all deletions are made at the other end(called front or head). The insertion operation is called Enqueue and the deletion operation is called Dequeue. Queue has FIFO property i.e., First In First Out. The element inserted firstwill be the first one to be removed.
Queue can be implemented as an array, where initially the front andrear both are set 0 to indicate empty queue.
When we insert an element into the queue we check whether the queueis full. If it is not full, we increment the value of rear and store the element at the top position.
rear = (rear+1) % max; Queue
[rear] =x;
When we pop an element from the queue we check whether the queue is empty. If it is not empty, we store the element at the top positionin a variable and decrement the value of top.
X= Queue [front];
Front=(front+1) % max;
We maintain a counter size to indicate the number of elements in the queue. Initially size=0. When we insert an element we increment size and when we delete an element we decrement size.

Code:

```
#include<iostream.h>
#include<conio.h> const
int max=10; void main()
{
	int choice, queue[max], i, front=0,rear=0, size=0,x;do
	{
		cout<<"1 - INSERT\n2 - DELETE\n3 - DISPLAY\n4 - EXIT\n";
		cin>>choice;
		switch(choice)
		{
			case 1:
				if(size==max) // Check whether queue is full.
					cout<<"Queue is full\n";
				else
				{
					cout<<"Enter the element: ";cin>>x;
```

```
                rear = (rear+1) % max;
                queue[rear]=x;//insert the element.size++;
                if (size==1)//front and rear are the same.front=rear;
            }
            break;
        case 2:
            if(size == 0) // Check whether queue is empty.
                cout<<"Queue is empty\n";
            else
            {
            x= queue [front];
            front=(front+1) % max;
            size--;
            }
            break;
        case 3:
            for(i=front;i<=front+size-1;i=(i+1)%max)cout<<queue[i]<<" ";
            cout<<endl;
            break;
        default:
            break;
      }
   }while(choice!=4);
}
```

Analysis:

Conclusion

:

Lab Assignment#6

Objective: To learn the concepts linked List and its operations.

Problem statement: Linked List Implementation

Underlying concepts: Linked List.

A linked list is a data structure in which elements are represented as nodes. Each node has two parts:

1. Info: Information part that stores the information of the element.
2. Next: Link part that stores the address of the next element in the list. It is a pointer field.

The linked list also maintains a pointer 'first' that stores the address of the first element in the list. If the list is empty we set first to NULL. The next part of the last node is also kept NULL. To perform insertion and deletion we only need to change few pointers. These changes are shown below:

Illustration for insertion:

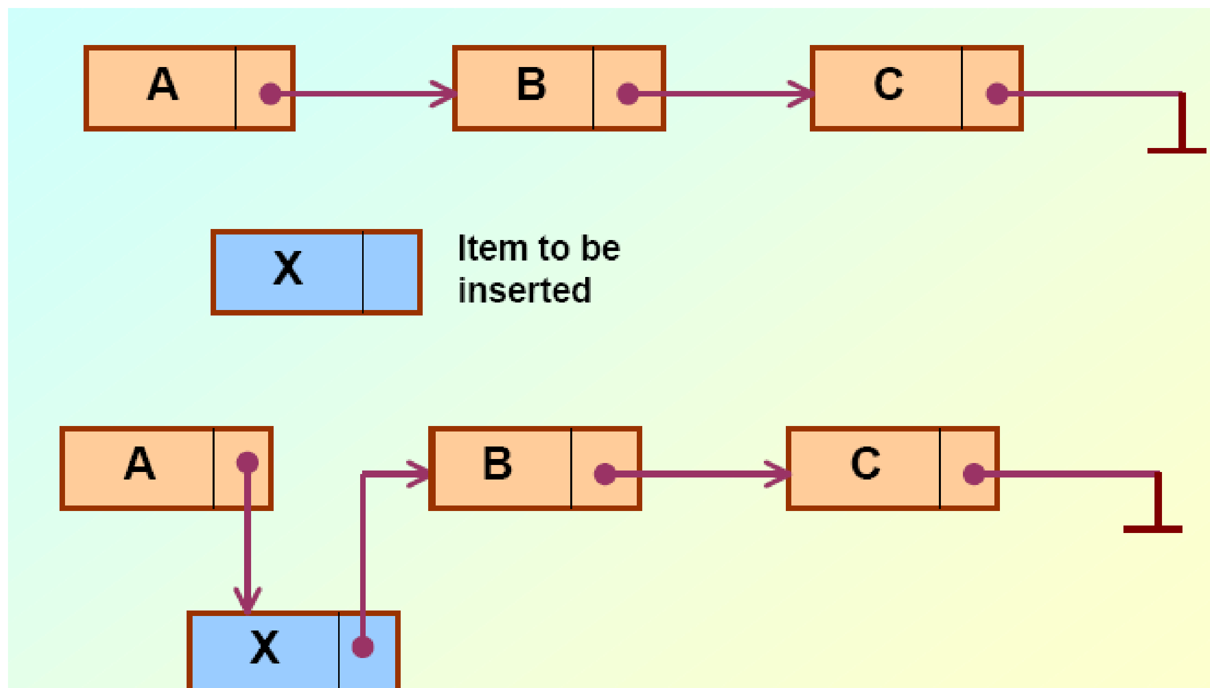
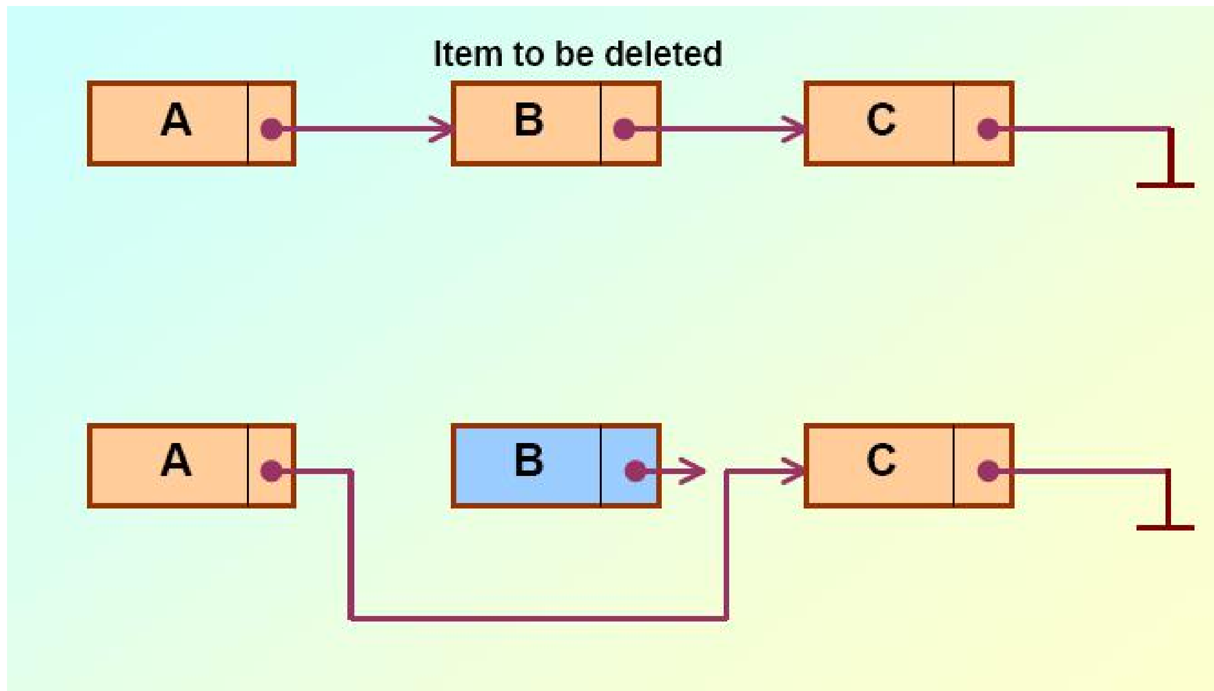


Illustration for deletion:



Code:

```
#include<iostream.h>
#include<conio.h> const
int max=10; void main()
{
	int choice, queue[max], i, front=0,rear=0, size=0,x;do
	{
		cout<<"1 - INSERT\n2 - DELETE\n3 - DISPLAY\n4 - EXIT\n";
		cin>>choice;
		switch(choice)
		{
			case 1:
				if(size==max) // Check whether queue is full.
					cout<<"Queue is full\n";
				else
				{
					cout<<"Enter the element: ";cin>>x;
					rear = (rear+1) % max;
					queue[rear]=x;//insert the element.size++;
					if (size==1)//front and rear are the same.front=rear;


				}
				break;
			case 2:
				if(size == 0) // Check whether queue is empty.
					cout<<"Queue is empty\n";
				else
				{
				x= queue [front];
				front=(front+1) % max;
				size--;
				}
				break;
			case 3:
				for(i=front;i<=front+size-1;i=(i+1)%max)cout<<queue[i]<<" ";
				cout<<endl;
				break;
			default:
				break;
		}
	}while(choice!=4);
}
```

Analysis:

Conclusion

:

Lab Assignment#7

Objective: To learn the concepts of Bubble Sort.

Problem statement: Implement Bubble Sort method.

Underlying concepts: Bubble Sort method.

Bubble sort sorts by employing a very simple strategy.

First it compares first and second element, and if first element is larger than the second then it interchanges first and second element.

Then it compares second and third element, if second element is larger than the third then it interchanges second and third element.

Then it compares third and fourth element, if third element is larger than the fourth then it interchanges third and third element.
And so on.
This completes the first pass. After first pass the largest element of the array is at its proper position.

We repeat the above process starting excluding the last element. This is pass number 2. After pass 2 the second largest element ofthe array is at its proper position.

We repeat the above process for n-1 number of passes.

Code:

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

void BubbleSort (int arr[], int n)
{
        int i, j;
        for (i = 0; i < n; ++i)
        {
                for (j = 0; j < n-i-1; ++j)
                {
                        // Comparing consecutive data and switching values if value at j >
j+1.
                        if (arr[j] > arr[j+1])
                        {
                                arr[j] = arr[j]+arr[j+1];
                                arr[j+1] = arr[j]-arr[j + 1];
                                arr[j] = arr[j]-arr[j + 1];
                        }
                }
                // Value at n-i-1 will be maximum of all the values below this index.
        }
}

int main()
{
```

```
        int n, i;
        cout<<"\nEnter the number of data element to be sorted: ";
        cin>>n;

        int arr[n];
        for(i = 0; i < n; i++)
        {
                cout<<"Enter element "<<i+1<<": ";
                cin>>arr[i];
        }

        BubbleSort(arr, n);

        // Display the sorted data.
        cout<<"\nSorted Data ";
        for (i = 0; i < n; i++)
    cout<<"->"<<arr[i];

        return 0;
}
```

Lab Assignment#8

Objective: To learn the concepts of Kruskal's algorithm

Problem statement: Implement Kruskal's algorithm for minimum spanning trees

Underlying concepts: Minimum spanning tree. Kruskal' algorithm.

Given a connected, undirected graph, a spanning tree of that graph is a sub graph which is a tree and connects all the vertices together. A single graph can have many different spanning trees. We can also assign a *weight* to each edge, which is a number representing how unfavorable it is, and use this to assign a weight to a spanning tree by computing the sum of the weights of the edges in that spanning tree. A **minimum spanning tree** (MST) or **minimum weight spanning tree** is then a spanning tree with weight less than or equal to the weight of every other spanning tree. More generally, any undirected graph (not necessarily connected) has a **minimum spanning forest**, which is a union of minimum spanning trees for its connected components.

Kruskal's algorithm is an algorithm in graph theory that finds a minimum spanning tree for a connected weighted graph. This means it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized. If the graph is not connected, then it finds a *minimum spanning forest* (a minimum spanning tree for each connected component). Kruskal's algorithm is an example of a greedy algorithm.

It works as follows:

- create a forest *F* (a set of trees), where each vertex in the graph is a separate tree
- create a set *S* containing all the edges in the graph
- while *S* is nonempty and F is not yet spanning
  - remove an edge with minimum weight from *S*
  - if that edge connects two different trees, then add it to the forest, combining two trees into a single tree
  - otherwise discard that edge.

At the termination of the algorithm, the forest has only one component and forms a minimum spanning tree of the graph.

1 **function** Kruskal(G)
2 Define an elementary cluster $C(v) \leftarrow \{v\}$.
3 Initialize a priority queue Q to contain all edges in G, using the weights as keys.
4 Define a forest $T \leftarrow \emptyset$ //T will ultimately contain the edges of the MST
5 // n is total number of vertices
6 **while** T has fewer than n-1 edges **do**
7 // edge u,v is the minimum weighted route from/to v
8 $(u,v) \leftarrow Q$.removeMin()
9 // prevent cycles in T. add u,v only if T does not already contain a path between u and v.
10 // the vertices has been added to the tree.
11 Let $C(v)$ be the cluster containing v, and let $C(u)$ be the cluster containing u.
13 **if** $C(v) \neq C(u)$ **then**
14 Add edge (v,u) to T.
15 Merge $C(v)$ and $C(u)$ into one cluster, that is, union $C(v)$ and $C(u)$.
16 **return** tree T

Code:

```
#include<iostream.h>
#define INFINITY 999
typedef struct Graph
{
 int v1;
 int v2;
 int cost;
}GR;
GR G[20];
int tot_edges,tot_nodes;
void create();
void spanning_tree();
int Minimum(int);

void main()
{
 cout<<"\n\t Graph Creation by adjacency matrix ";
 create();
 spanning_tree();
}
void create()
{

 int k;
 cout<<"\n Enter Total number of nodes: ";
 cin>>tot_nodes;
 cout<<"\n Enter Total number of edges: ";
 cin>>tot_edges;
 for(k=0;k<tot_edges;k++)
 {
          cout<<" Enter Edge in (V1 V2)form ";
          cin>>G[k].v1>>G[k].v2;
          cout<<2
          "\n Enter Corresponding Cost ";
          cin>>G[k].cost;
 }
}
void spanning_tree()
{
 int count,k,v1,v2,i,j,tree[10][10],pos,parent[10];
 int sum;
 int Find(int v2,int parent[]);
 void Union(int i,int j,int parent[]);
 count=0;
 k=0;
 sum=0;
 for(i=0;i<tot_nodes;i++)
                    parent[i]=i;
 while(count!=tot_nodes-1)
 {
                  pos=Minimum(tot_edges);//finding the minimum cost edge
                  if(pos==-1)//Perhaps no node in the graph
                            break;
```

```
                v1=G[pos].v1;
                v2=G[pos].v2;
                i=Find(v1,parent);
                j=Find(v2,parent);
                if(i!=j)
                {
                        tree[k][0]=v1;//storing the minimum edge in array tree[]
                        tree[k][1]=v2;
                        k++;
                        count++;
                        sum+=G[pos].cost;//accumulating the total cost of MST
                        Union(i,j,parent);
                }
                G[pos].cost=INFINITY;
        }
                if(count==tot_nodes-1)
                {
                 cout<<"\n Spanning tree is...";
                 cin>>"\n-------------------------\n";
                 for(i=0;i<tot_nodes-1;i++)
                 {
                        cout<<"["<<tree[i][0];
                        cout<<" - ";
                        cout<<"%d"<<tree[i][1];
                        cout<<"]";
                 }
                 cout<<"\n-------------------------";
                 cout<<"\nCost of Spanning Tree is = %d"<<sum;
                }
                else
                 {
                        cout<<"There is no Spanning Tree";
                 }
}
int Minimum(int n)
{
 int i,small,pos;
 small=INFINITY;
 pos=-1;
 for(i=0;i<n;i++)
 {
                if(G[i].cost<small)
                {
                        small=G[i].cost;
                        pos=i;
                }
 }
 return pos;
}
int Find(int v2,int parent[])
{
  while(parent[v2]!=v2)
{
                v2=parent[v2];
        }
```

```
        return v2;
}
void Union(int i,int j,int parent[])
{
 if(i<j)
            parent[j]=i;
 else
            parent[i]=j;
}
```

Lab Assignment#9

Objective: To learn the concepts of Prim's algorithm

Problem statement: Implement Prim's algorithm for minimum spanning trees

Underlying concepts: Minimum spanning tree. Prim's algorithm.

The minimum spanning tree of a planar graph. Each edge is labeled with its weight, which here is roughly proportional to its length.

Given a connected, undirected graph, a spanning tree of that graph is a sub graph which is a tree and connects all the vertices together. A single graph can have many different spanning trees. We can also assign a *weight* to each edge, which is a number representing how unfavorable it is, and use this to assign a weight to a spanning tree by computing the sum of the weights of the edges in that spanning tree. A **minimum spanning tree** (MST) or **minimum weight spanning tree** is then a spanning tree with weight less than or equal to the weight of every other spanning tree. More generally, any undirected graph (not necessarily connected) has a **minimum spanning forest**, which is a union of minimum spanning trees for its connected components.

Prim's algorithm is an algorithm that finds a minimum spanning tree for a connected weighted undirected graph. This means it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized. Prim's algorithm is an example of a greedy algorithm

The algorithm continuously increases the size of a tree starting with a single vertex until it spans all the vertices.

- Input: A connected weighted graph with vertices V and edges E.
- Initialize: $V_{new} = \{x\}$, where x is an arbitrary node (starting point) from V, $E_{new} = \{\}$
- Repeat until $V_{new} = V$:
    - Choose edge (u,v) with minimal weight such that u is in V_{new} and v is not (if there are multiple edges with the same weight, choose arbitrarily but consistently)
    - Add v to V_{new}, add (u, v) to E_{new}
- Output: V_{new} and E_{new} describe a minimal spanning tree

Code:

```
# include<iostream.h>
# include<conio.h>
# define SIZE 20
# define INFINITY 32767


void Prim(int G[][SIZE], int nodes)
{
        int select[SIZE], i, j, k;
        int min_dist, v1, v2,total=0;

        for (i=0 ; i<nodes ; i++)   // Initialize the selected vertices list
                select[i] = 0;
```

```
        cout<<"\n\n The Minimal Spanning Tree Is :\n";
        select[0] = 1;
        for (k=1 ; k<nodes ; k++)
        {
                min_dist = INFINITY;
                for (i=0 ; i<nodes ; i++) // Select an edge such that one vertex is
                {                   // selected and other is not and the edge
        for (j=0 ; j<nodes ; j++) // has the least weight.
        {
            if (G[i][j] && ((select[i] && !select[j]) || (!select[i] && select[j])))
                {
                    if (G[i][j] < min_dist)//obtained edge with minimum wt
                    {
                        min_dist = G[i][j];
                        v1 = i;
                        v2 = j;   //picking up those vertices
                    }
                }
            }
        }
        cout<<"\n Edge "<v1<<v2<<" and weight = "<<min_dist;
        select[v1] = select[v2] = 1;
        total =total+min_dist;
        }
        cout<<"\n\n\t Total Path Length Is = "<<total;
}

void main()
{
        int G[SIZE][SIZE], nodes;
        int v1, v2, length, i, j, n;

        clrscr();
        cout<<"\n\t Prim'S Algorithm\n"

        cout<<"\n Enter Number of Nodes in The Graph  ";
        cin>>nodes;
        cout<<"\n Enter Number of Edges in The Graph  ";
        cin>>n;

        for (i=0 ; i<nodes ; i++)       // Initialize the graph
            for (j=0 ; j<nodes ; j++)
                G[i][j] = 0;
        //entering weighted graph
        cout<<"\n Enter edges and weights \n";
        for (i=0 ; i<n; i++)
        {
                cout<<"\n Enter Edge by V1 and V2 :";
                cin>>v1>>v2;
                cout<<"\n Enter corresponding weight :";
                cin>>length;
                G[v1][v2] = G[v2][v1] = length;
        }
        getch();
```

```
cout<<"\n\t";
clrscr();
Prim(G,nodes);
getch();
}
```

Lab Assignment#10

Objective: To learn the concept of Dijkstra's algorithm

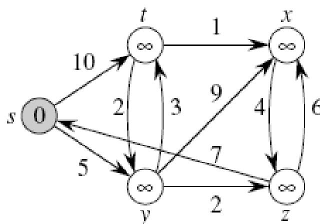
Problem statement: Implement single sources shortest path algorithm.

Underlying concepts: Dijkstra's algorithm.

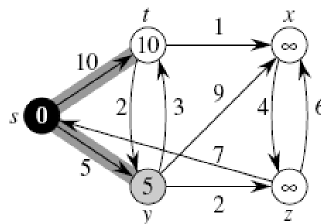
Dijkstra's algorithm

Dijkstra's algorithm solves the single-source shortest-paths problem on a weighted, directed graph $G = (V, E)$ for the case in which all edge weights are nonnegative. Therefore, we assume that $w(u, v) \geq 0$ for each edge $(u, v) \in E$.
Dijkstra's algorithm maintains a set S of vertices whose final shortest-path weights from the source s have already been determined. The algorithm repeatedly selects the vertex $u \in V - S$ with the minimum shortest-path estimate, adds u to S, and relaxes all edges leaving u. In the following implementation, we use a min-priority queue Q of vertices, keyed by their d values.
DIJKSTRA(G,w, s)
1 INITIALIZE-SINGLE-SOURCE(G, s)
2 $S \leftarrow \emptyset$ 3 $Q \leftarrow V[G]$
4 **while** Q != $\emptyset$ **do** $u \leftarrow$ EXTRACT-MIN(Q)
6 $S \leftarrow S \cup \{u\}$ 7 **for** each vertex $v \in Adj[u]$
8 **do** RELAX(u, v,w)

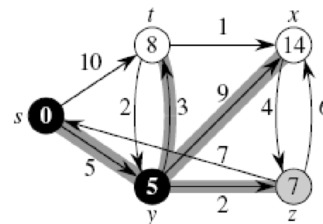
Dijkstra's algorithm relaxes edges as shown



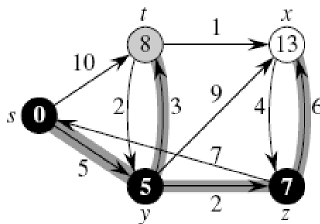
(a)



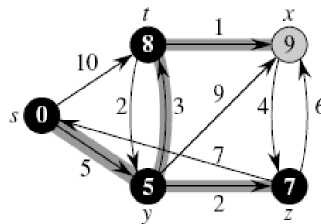
(b)



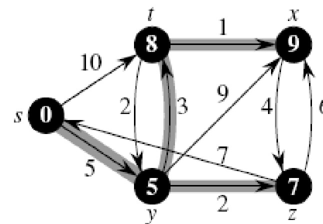
(c)



(d)



(e)



(f)

Code:

```
#include<iostream.h>
#include<conio.h>
#define infinity 999
int path[10];
void main()
{
 int tot_nodes,i,j,cost[10][10],dist[10],s[10];
 void create(int tot_nodes,int cost[][10]);
 void Dijkstra(int tot_nodes,int cost[][10],int i,int dist[10]);
 void display(int i,int j,int dist[10]);
 clrscr();
 cout<<"\n\t\t Creation of graph ";
 cout<<"\n Enter total number of nodes ";
 cin>>tot_nodes;
 create(tot_nodes,cost);
 for(i=0;i<tot_nodes;i++)
 {
			cout<<"\n\t\t\t Press any key to continue...";
			cout<<"\n\t\t When Source =”<<i<<endl;
			for(j=0;j<tot_nodes;j++)
			{
			 Dijkstra(tot_nodes,cost,i,dist);
			 if(dist[j]==infinity)
				cout<<"\n There is no path to "<<j<<endl;
			 else
			 {

				display(i,j,dist);
			 }
			}
		}
}
void create(int tot_nodes,int cost[][10])
{
 int i,j,val,tot_edges,count=0;
 for(i=0;i<tot_nodes;i++)
 {
			for(j=0;j<tot_nodes;j++)
			{
			 if(i==j)
				cost[i][j]=0;//diagonal elements are 0
			 else
				cost[i][j]=infinity;
			}
 }
 cout<<"\n Total number of edges ";
 cin>>&tot_edges;
 while(count<tot_edges)
			{
			 cout<<"\n Enter Vi and Vj";
			 cin>>i>>j;
```

```
                cout<<"\n Enter the cost along this edge ";
                cin>>val;
                cost[j][i]=val;
                cost[i][j]=val;
                count++;
               }
}
```