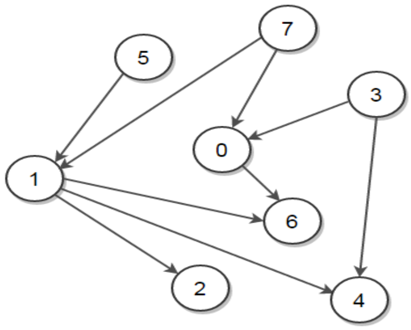
IAT 2 – July 2024 Scheme & Solution

Sub:	**Analysis & Design of Algorithms**				Sub Code:	BCS401	Branch:	ISE
Date:	10/07/2024	Duration:	90 min's	Max Marks: 50	Sem / Sec:	IV/A, B, C		OBE

	Answer any FIVE FULL QUESTIONS	MARKS	CO	RBT
1	**Define topological sorting. Illustrate the topological sorting using DFS method for thefollowing graph.**	[10]	CO2	L2



Defination: - [2 Marks]
Algorithm: - [4 Marks]
Solution:-[4 Marks]

Answer:-
Topological sort is a fundamental algorithm used in directed acyclic graphs(DAGs).

Topological sorting is a way to arrange a collection of tasks or events in such a sequence that each task comes before the tasks that depend on it.

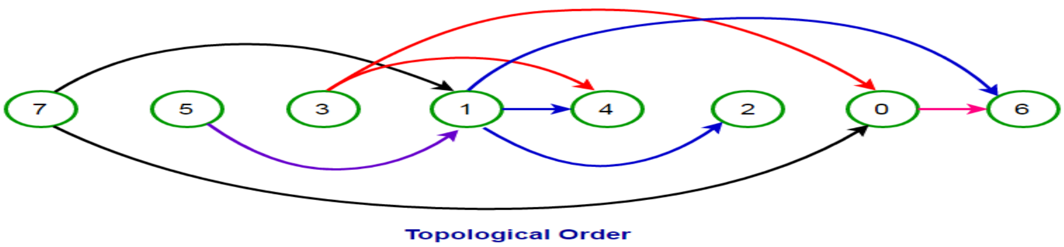
In simple words, it helps you determine the order in which you should perform a set of related tasks, ensuring that you don't start a task until all its prerequisites or dependencies are completed.

Algorithm:-

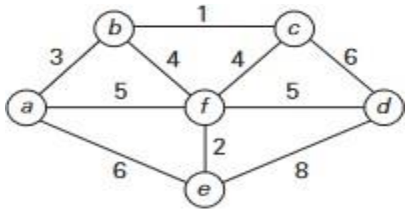
```
L -> An empty list that will contain the sorted elements
S -> A set of all vertices with no incoming edges (i.e., having indegree 0)

while S is non-empty do
    remove a vertex n from S
    add n to tail of L
    for each vertex m with an edge e from n to m do
        remove edge e from the graph
        if m has no other incoming edges, then insert m into S
            insert m into S

if graph has edges then
    return report "graph has at least one cycle"
else
    return L "a topologically sorted order"
```

It’s like making sure you can’t bake a cake until you have all the necessary ingredients and utensils ready, one step at a time. Topological sorting is widely used in project scheduling, building software with dependencies, and more.

It helps to avoid cycles and ensures a valid sequence.
For every directed edge u —> v, u comes before v in the ordering. For example, the pictorial representation of the topological order **[7, 5, 3, 1, 4, 2, 0, 6]** is:



2	**Write Kruskal algorithm and construct minimum cost spanning tree for the following graph.**	[10]	CO4	L3



Algorithm: - [5 Marks]
Solution step by step:-[5 Marks]
Answer: -

3.5.2.1 Algorithm

```
Algorithm spanning_tree()
//Problem Description : This algorithm finds the minimum
//spanning tree using Kruskal's algorithm
//Input : The adjacency matrix graph G containing cost
//Output : prints the spanning tree with the total cost of
//spanning tree
 count←0
 k←0
 sum←0
       for i←0 to tot_nodes do
              parent[i]←i
while(count!=tot_nodes–1)do
 {
   pos←Minimum(tot_edges);//finding the minimum cost edge
       if(pos=–1)then//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)then
        {
         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
```

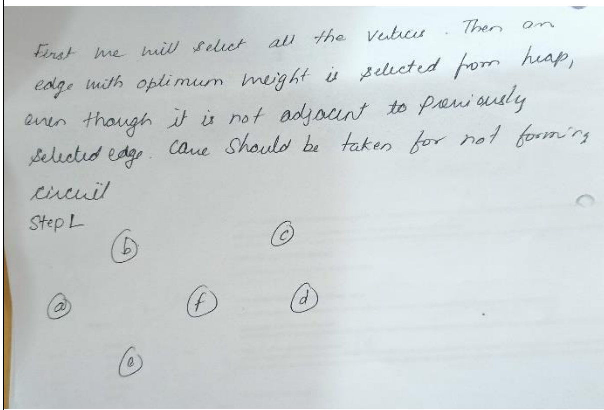
tree [ ] [ ] is an array in which the spanning tree edges are stored.

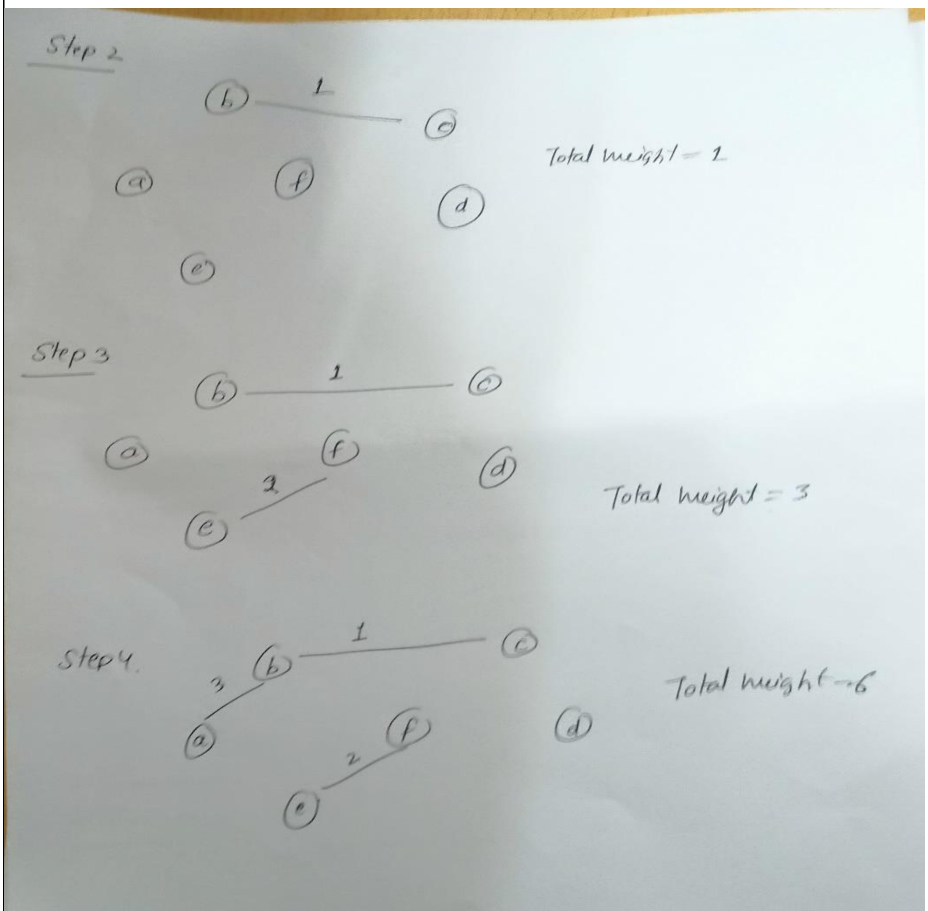
Computing total cost of all the minimum

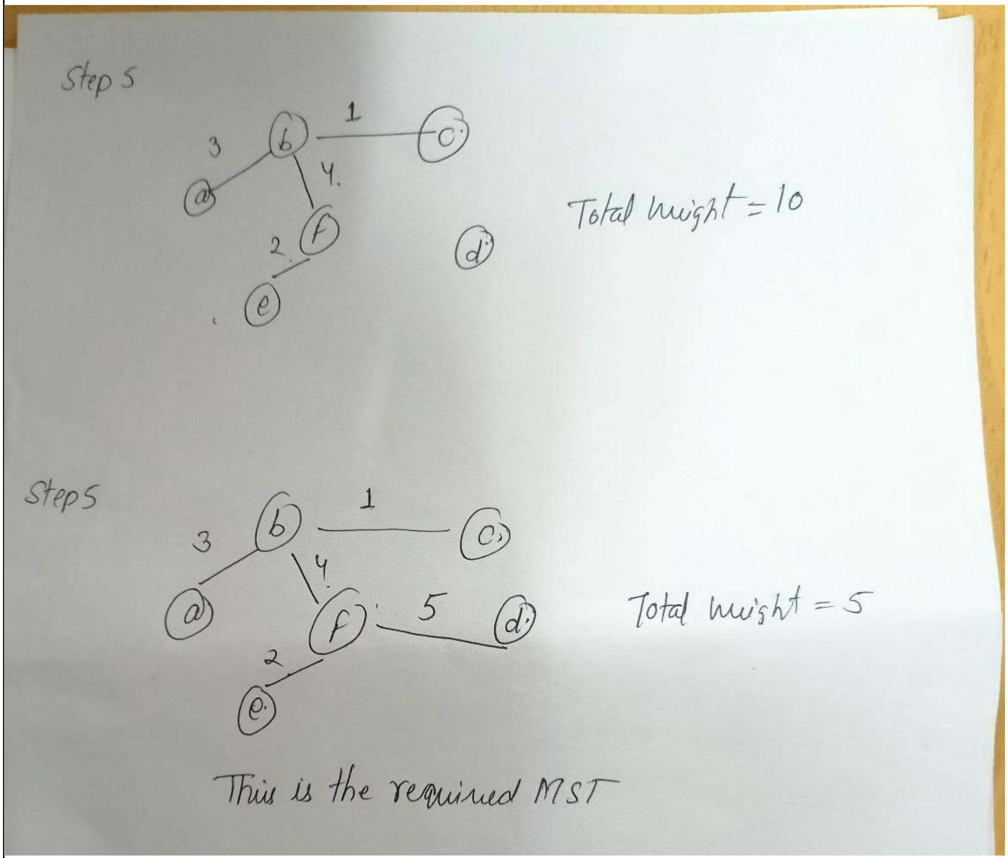
```
        count++;
        sum+←G[pos].cost
//accumulating the total cost of MST
        Union(i,j,parent);
        }
        G[pos].cost INFINITY
       }
       if(count=tot_nodes–1)then
       {
            for i←0 to tot_nodes–1
         {
            write(tree[i][0],tree[i][1])
         }
         write("Cost of Spanning Tree is ",sum)
       }
```

Computing total cost of all the minimum distances.

For each node of i, the minimum distance edges are collected in array **tree [ ] [ ]** . The spanning tree is printed here.


First we will select all the vertices. Then an edge with optimum weight is selected from heap, even though it is not adjacent to previously selected edge. Care should be taken for not forming circuit



	 Total weight is 15 (1+2+3+4+5)			
3	**Find the optimal solution of the knapsack instance is using dynamic programming n=7, M=15, (P1,P2,…..,P7)=(10,5,15,7,6,18,3) and (w1,w2,…..,w7)=(2,3,5,7,1,4,1)** **Algorithm: - [5 Marks]** **Solution step by step:-[5 Marks]** **Answer: -** **Algorithm**	[10]	CO4	L3

```
for (i←0 to n) do
{
  for (j←0 to W) do
  {
    table[i,0]=0   // table initialization
    table[0,j]=0
  }
}
for (i←0 to n) do
{
 for (j←0 to W) do
 {
  if(j<w[i]) then
    table[i,j]←  table[i-1,j]
  else if(j>=w[i]) then
    table[i,j]← max (table[i-1,j],(v[i]+table[i-1,j-w[i]]) )
  }
}
return table[n,W]
```

To solve this problem, we use some strategy to determine the fraction of weight which should be included so as to maximize the profit and fill the Knapsack..

(X1, X2, X3, X4, X5, X6, X7)   ∑WiXi   ∑PiXi

(1) (1/2,1/3,1/4,1/5,1/6,1/7,1/8)   5.51   15.76

Now taking maximum profit 18 with weight 4 as -

X6 = 1, ∑ WiXi < m.

(2) (1/2,1/3,1/4,1/5,1/6,1,1/8)   8.51   31.19

(3) (1/2,1/3,1,1/5,1/6,1,1/8)   12.69   42.44

(4)) (1,1/3,1,1/5,1/6,1,1/8)   13.69   47.44

(5) (1,1/3,1,1/5,1,1,1/8)   14.52   52.44

(6) (1,1/3,1,1/5,1,1,1)   15   54.67

(7) (1,2/3,1,0,1,1,1)   15   55.33

at each step, we try to get the maximum profit. The maximum profit we set by step (7) taking

X1 = 1, X2 = 2/3, X3 = 1, X4 = 0, X5 = 1, X6 = 1, and X7=1

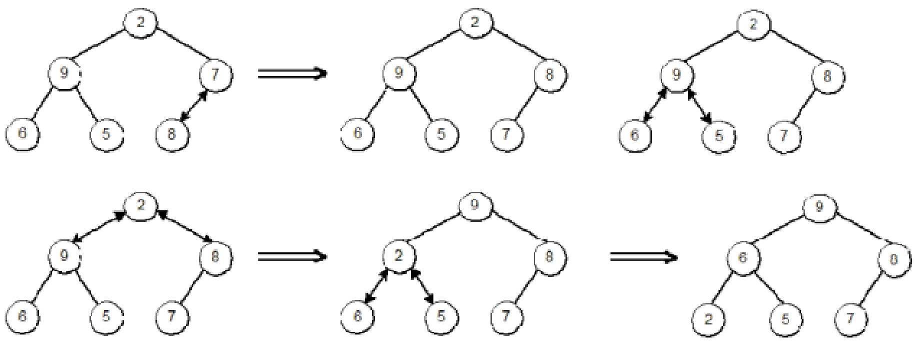
These fraction of weight provided maximum profit.

4	**Write heap sort algorithm.Sort the given list of numbers using heap sort: 2, 9, 7, 6, 5, 8.** **Algorithm: - [5 Marks]** **Solution step by step:-[5 Marks]** **Answer: -**	**[10]**	**CO3**	**L2**

```
Algorithm HeapBottomUp(H[1..n])
//Constructs a heap from the elements of a given array
// by the bottom-up algorithm
//Input: An array H[1..n] of orderable items
//Output: A heap H[1..n]
for i ← ⌊n/2⌋ downto 1 do
    k ← i;   v ← H[k]
    heap ← false
    while not heap and 2 * k ≤ n do
          j ← 2 * k
          if j < n   //there are two children
             if H[j] < H[j + 1]    j ← j + 1
          if v ≥ H[j]
                heap ← true
          else H[k] ← H[j];    k ← j
  H[k] ← v
```

Example of Heap Construction

Construct a heap for the list 2, 9, 7, 6, 5, 8



Heapsort

Stage 1: Construct a heap for a given list of *n* keys

Stage 2: Repeat operation of root removal *n*-1 times:

- **Exchange keys in the root and in the last (rightmost) leaf**
- **Decrease heap size by 1**
- **If necessary, swap new root with larger child until the heap condition holds**

Example of Sorting by Heapsort

Sort the list 2, 9, 7, 6, 5, 8 by heapsort

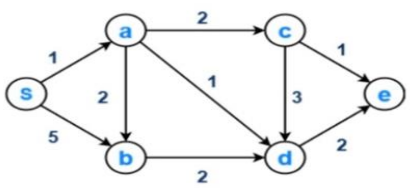
Stage 1 (heap construction)

```
2 9 7 6 5 8
2 9 8 6 5 7
2 9 8 6 5 7
9 2 8 6 5 7
9 6 8 2 5 7
```

Stage 2 (root/max removal)

```
9 6 8 2 5 7
7 6 8 2 5|9
8 6 7 2 5|9
5 6 7 2|8 9
7 6 5 2|8 9
2 6 5|7 8 9
6 2 5|7 8 9
5 2|6 7 8 9
5 2|6 7 8 9
2|5 6 7 8 9
```

Q. No.	Question	Marks	CO	RBT
5	**Apply Dijkstra's algorithm to find single source shortest path for the given graph by considering S as the source vertex.**	**[10]**	**CO4**	**L2**



Algorithm: - [5 Marks]
Solution step by step:-[5 Marks]

Answer: -

Dijkstra Algorithm-

dist[S] ← 0 // The distance to source vertex is set to 0
Π[S] ← **NIL** // The predecessor of source vertex is set as NIL
for all v ∈ V - {S} // For all other vertices
do dist[v] ← ∞ // All other distances are set to ∞
Π[v] ← **NIL** // The predecessor of all other vertices is set as NIL
S ← ∅ // The set of vertices that have been visited 'S' is initially empty
Q ← V // The queue 'Q' initially contains all the vertices
while Q ≠ ∅ // While loop executes till the queue is not empty
do u ← mindistance (Q, dist) // A vertex from Q with the least distance is selected
S ← S ∪ {u} // Vertex 'u' is added to 'S' list of vertices that have been visited
for all v ∈ neighbors[u] // For all the neighboring vertices of vertex 'u'
do if dist[v] > dist[u] + w(u,v) // if any new shortest path is discovered
then dist[v] ← dist[u] + w(u,v) // The new value of the shortest path is selected
return dist

Step-01:

The following two sets are created-

- Unvisited set : {S , a , b , c , d , e}
- Visited set : { }

Step-02:

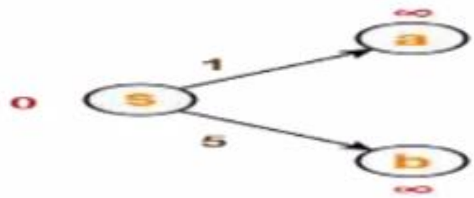
The two variables Π and d are created for each vertex and initialized as-

- Π[S] = Π[a] = Π[b] = Π[c] = Π[d] = Π[e] = NIL
- d[S] = 0
- d[a] = d[b] = d[c] = d[d] = d[e] = ∞

Step-03:

- Vertex 'S' is chosen.
- This is because shortest path estimate for vertex 'S' is least.
- The outgoing edges of vertex 'S' are relaxed.

Before Edge Relaxation-



Now,

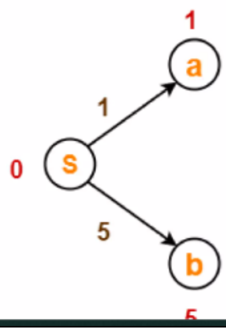
- d[S] + 1 = 0 + 1 = 1 < ∞

∴ d[a] = 1 and Π[a] = S

- d[S] + 5 = 0 + 5 = 5 < ∞

∴ d[b] = 5 and Π[b] = S

After edge relaxation, our shortest path tree is-



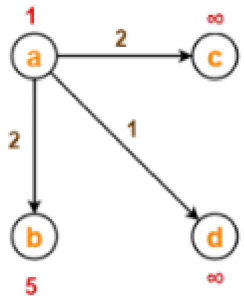
Now, the sets are updated as-

- Unvisited set : {a , b , c , d , e}
- Visited set : {S}

Step-04:

- Vertex 'a' is chosen.
- This is because shortest path estimate for vertex 'a' is least.
- The outgoing edges of vertex 'a' are relaxed.

Before Edge Relaxation-



Now,

- $d[a] + 2 = 1 + 2 = 3 < \infty$

$\therefore$ $d[c] = 3$ and $\Pi[c] = a$

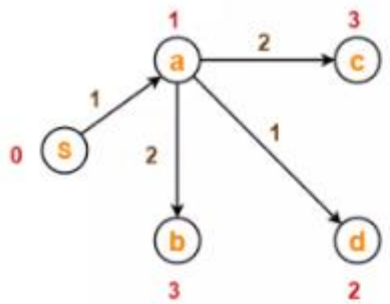
- $d[a] + 1 = 1 + 1 = 2 < \infty$

$\therefore$ $d[d] = 2$ and $\Pi[d] = a$

- $d[b] + 2 = 1 + 2 = 3 < 5$

$\therefore$ $d[b] = 3$ and $\Pi[b] = a$

After edge relaxation, our shortest path tree is-



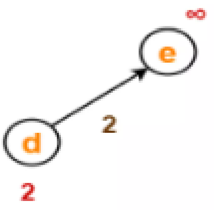
Now, the sets are updated as-

- Unvisited set : {b , c , d , e}
- Visited set : {S , a}

Step-05:

- Vertex 'd' is chosen.
- This is because shortest path estimate for vertex 'd' is least.
- The outgoing edges of vertex 'd' are relaxed.

Before Edge Relaxation-

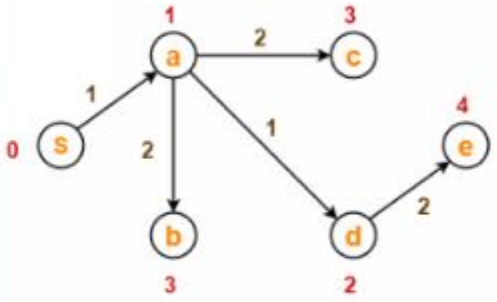


Now,

- d[d] + 2 = 2 + 2 = 4 < ∞

∴ d[e] = 4 and Π[e] = d

After edge relaxation, our shortest path tree is-



Now, the sets are updated as-

- Unvisited set : {b , c , e}
- Visited set : {S , a , d}

Step-06:

- Vertex 'b' is chosen.
- This is because shortest path estimate for vertex 'b' is least.
- Vertex 'c' may also be chosen since for both the vertices, shortest path estimate is least.
- The outgoing edges of vertex 'b' are relaxed.

Before Edge Relaxation-

Now,

d[b] + 2 = 3 + 2 = 5 > 2

∴ No change

After edge relaxation, our shortest path tree remains the same as in Step-05.

Now, the sets are updated as-

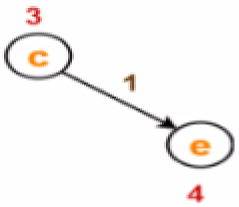
- Unvisited set : {c , e}
- Visited set : {S , a , d , b}

<table>
<tr>
<td></td>
<td>

Step-07:

- Vertex ‘c’ is chosen.
- This is because shortest path estimate for vertex ‘c’ is least.
- The outgoing edges of vertex ‘c’ are relaxed.

Before Edge Relaxation-



Now,

- d[c] + 1 = 3 + 1 = 4 = 4

∴ No change

After edge relaxation, our shortest path tree remains the same as in Step-05.

Now, the sets are updated as-

- Unvisited set : {e}
- Visited set : {S , a , d , b , c}

Step-08:

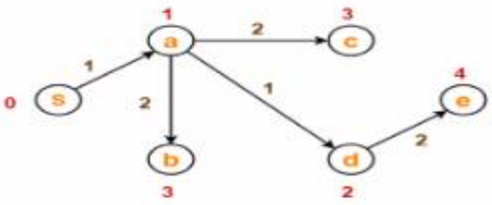
- Vertex ‘e’ is chosen.
- This is because shortest path estimate for vertex ‘e’ is least.
- The outgoing edges of vertex ‘e’ are relaxed.
- There are no outgoing edges for vertex ‘e’.
- So, our shortest path tree remains the same as in Step-05.

Now, the sets are updated as-

- Unvisited set : { }
- Visited set : {S , a , d , b , c , e}

Now,

- All vertices of the graph are processed.
- Our final shortest path tree is as shown below.
- It represents the shortest path from source vertex ‘S’ to all other remaining vertices.



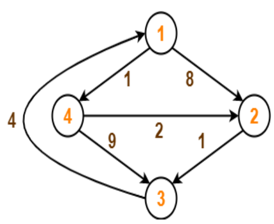
Shortest Path Tree

The order in which all the vertices are processed is :

S , a , d , b , c , e.

</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>

Apply Floyd's algorithm to find all pair shortest path for the graph given below.



</td>
<td>[10]</td>
<td>CO4</td>
<td>L3</td>
</tr>
</table>

Algorithm: - [5 Marks]
Solution step by step:-[5 Marks]
Answer: -

Floyds Algorithm

```
Create a |V| x |V| matrix                // It represents the
distance between every pair of vertices as given

For each cell (i,j) in M do-

    if i = = j

        M[ i ][ j ] = 0                  // For all diagonal
elements, value = 0

    if (i , j) is an edge in E

        M[ i ][ j ] = weight(i,j)        // If there exists a direct
edge between the vertices, value = weight of edge

    else

        M[ i ][ j ] = infinity           // If there is no direct
edge between the vertices, value = ∞

for k from 1 to |V|

    for i from 1 to |V|

        for j from 1 to |V|

            if M[ i ][ j ] > M[ i ][ k ] + M[ k ][ j ]
            M[ i ][ j ] = M[ i ][ k ] + M[ k ][ j ]
```

Step-01:

- Remove all the self loops and parallel edges (keeping the lowest weight edge) from the graph.
- In the given graph, there are neither self edges nor parallel edges.

Step-02:

Write the initial distance matrix

- It represents the distance between every pair of vertices in the form of given weights.
- For diagonal elements (representing self-loops), distance value = 0.
- For vertices having a direct edge between them, distance value = weight of that edge.
- For vertices having no direct edge between them, distance value = ∞.

Initial distance matrix for the given graph is-

$$D_0 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 8 & \infty & 1 \\ 2 & \infty & 0 & 1 & \infty \\ 3 & 4 & \infty & 0 & \infty \\ 4 & \infty & 2 & 9 & 0 \end{array}$$

Step-03:

Using Floyd Warshall Algorithm, write the following 4 matrices-

$$D_1 = \begin{array}{c|cccc} & & & & \\ \hline 1 & 0 & 8 & \infty & 1 \\ 2 & \infty & 0 & 1 & \infty \\ 3 & 4 & 12 & 0 & 5 \\ 4 & \infty & 2 & 9 & 0 \end{array}$$

$$D_2 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 8 & 9 & 1 \\ 2 & \infty & 0 & 1 & \infty \\ 3 & 4 & 12 & 0 & 5 \\ 4 & \infty & 2 & 3 & 0 \end{array}$$

$$D_3 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 8 & 9 & 1 \\ 2 & 5 & 0 & 1 & 6 \\ 3 & 4 & 12 & 0 & 5 \\ 4 & 7 & 2 & 3 & 0 \end{array}$$

$$D_4 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 3 & 4 & 1 \\ 2 & 5 & 0 & 1 & 6 \\ 3 & 4 & 7 & 0 & 5 \\ 4 & 7 & 2 & 3 & 0 \end{array}$$

The last matrix D_4 represents the shortest path distance between every pair of vertices.

7	Obtain the Huffman tree and the code for the following data and apply algorithm	[10]	CO4	L2

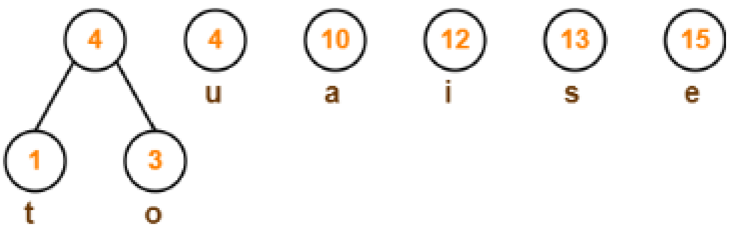
Character	a	e	i	o	u	s	t
Count	10	15	12	3	4	13	1

Algorithm: - [5 Marks]
Solution step by step:-[5 Marks]

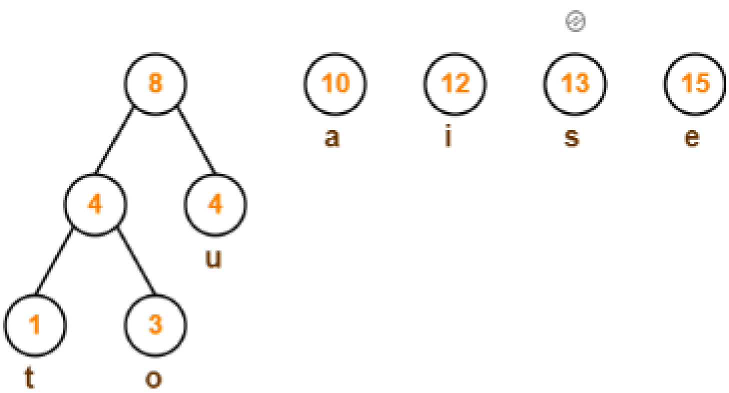
Answer: -

Step-01:

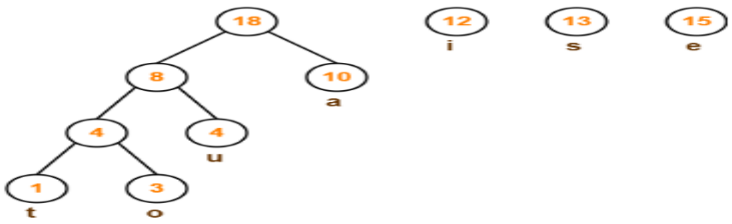
Step-02:



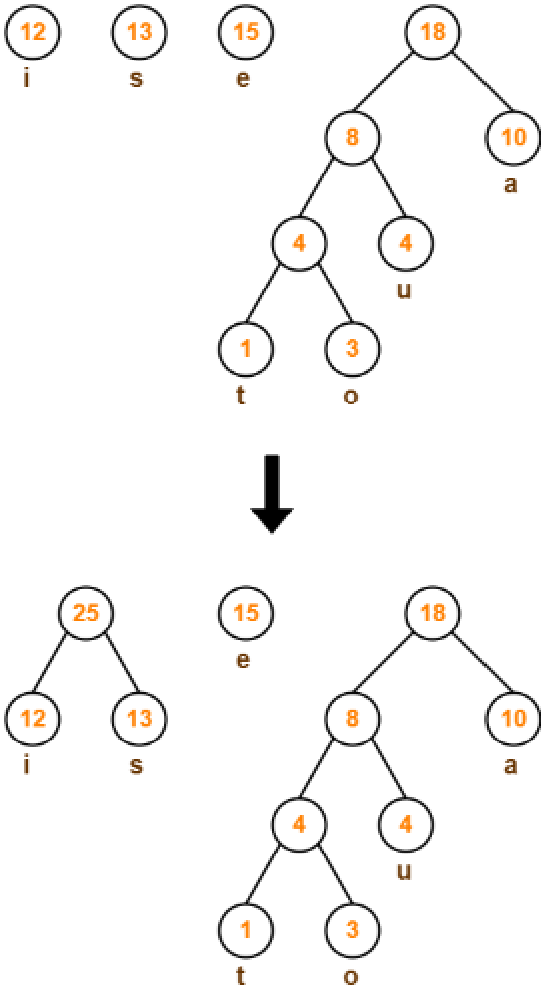
Step-03:



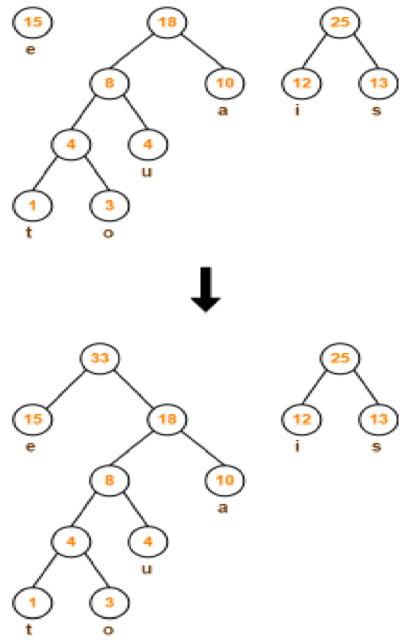
Step-04:



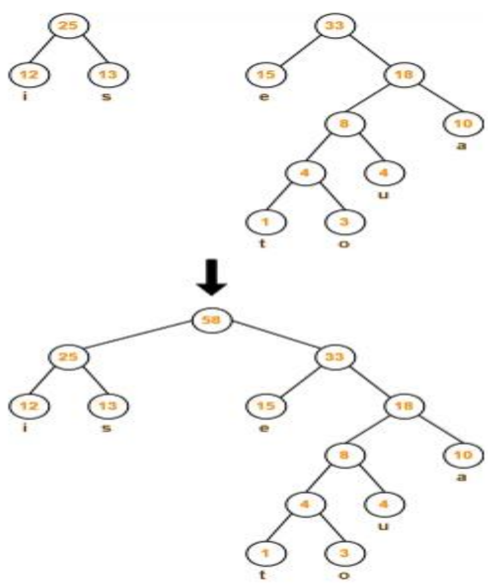
Step-05:



Step-06:

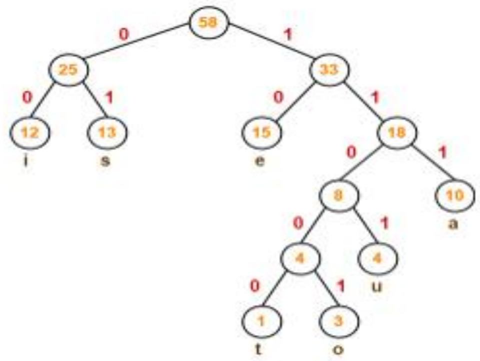


Step-07:



Huffman Tree

After assigning weight to all the edges, the modified Huffman Tree is-



Huffman Tree

Now, let us answer each part of the given problem one by one-

Following this rule, the Huffman Code for each character is-

- a = 111
- e = 10
- i = 00
- o = 11001
- u = 1101
- s = 01
- t = 11000