UNIT - IV

GREEDY TECHNIQUE

Solution should be:-

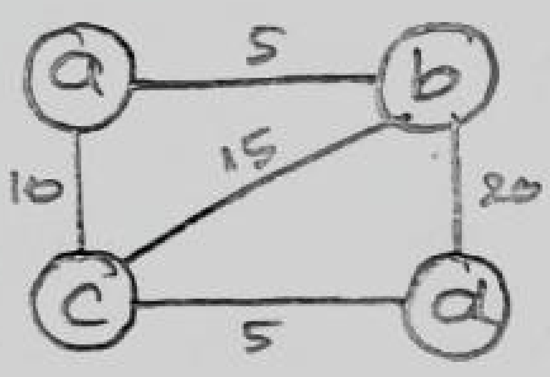
→ feasible

→ locally optional

→ irrevocable.

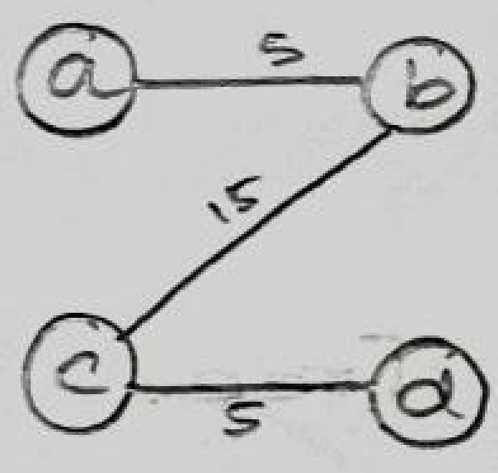
PRIM'S ALGORITHM:-

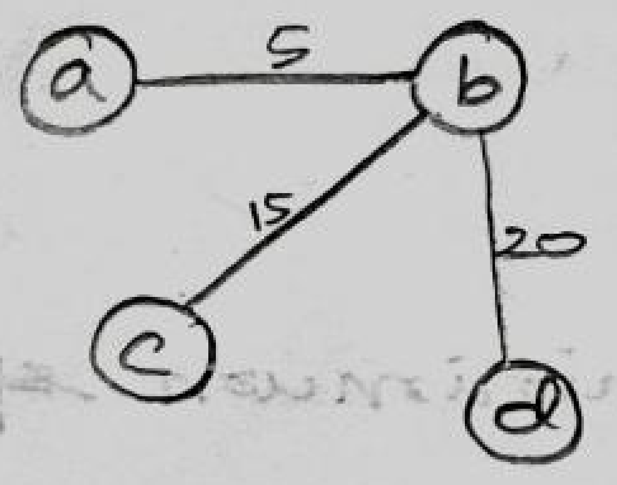
↳ To generate minimum spanning tree.



Spanning tree of a graph:-

Connected acyclic subgraph which is having all the vertices of the graph.



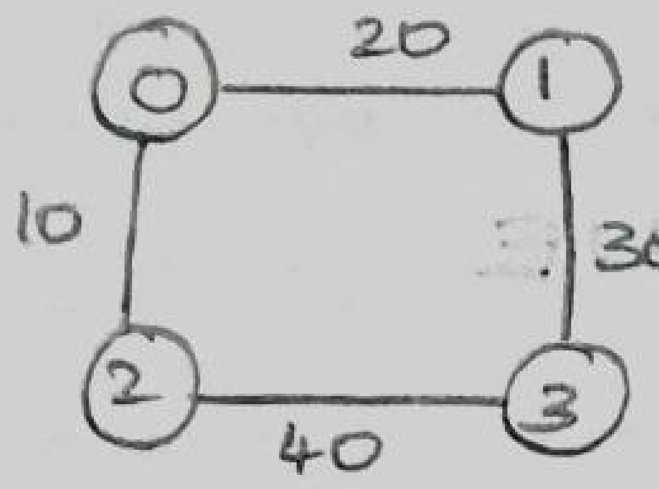


Minimum spanning tree:-

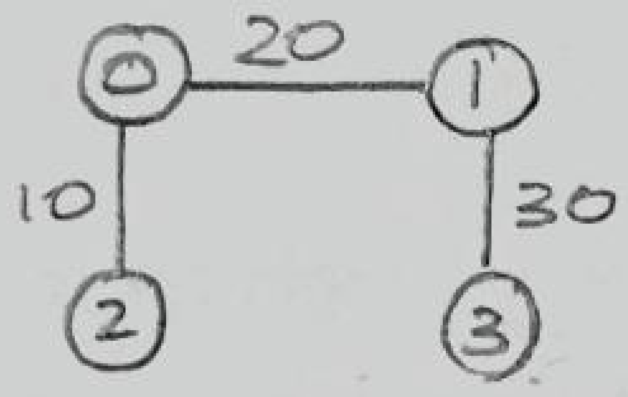
Spanning tree with minimum weight.

A spanning tree of a connected graph is its connected acyclic subgraph (tree) that contains all the vertices of the graph.

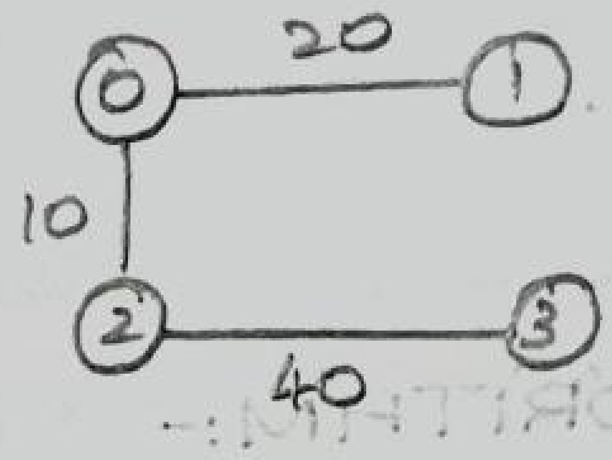
A minimum spanning tree of a weighted connected graph is its spanning tree of smallest weight, where weight of a tree is defined as the sum of weights of all its edges.



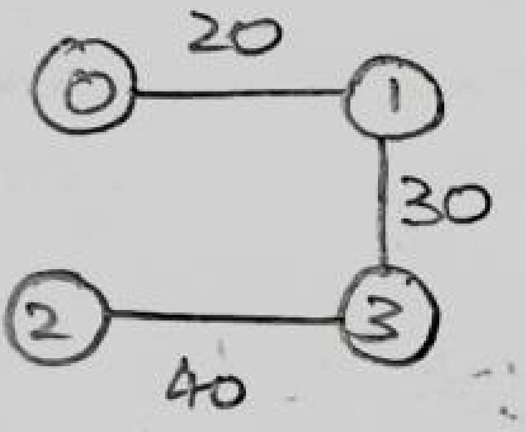
spanning trees:-



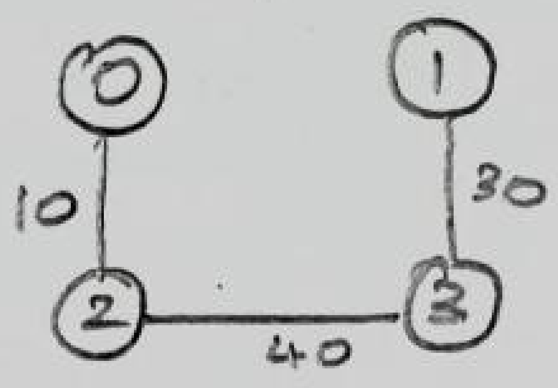
T_1



T_2



T_3



T_4

$w(T_1) = 60$

$w(T_2) = 70$

$w(T_4) = 80$

$w(T_3) = 90$

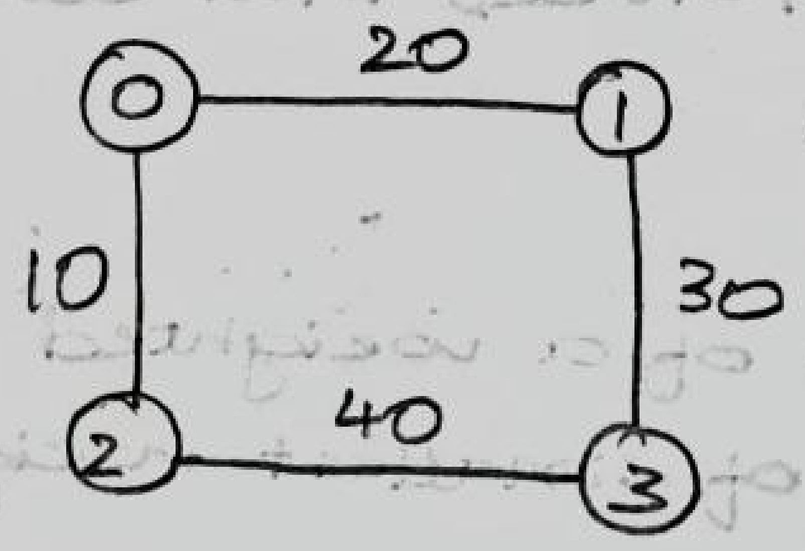
∴ T_1 is minimum spanning Tree.

NOTE:-

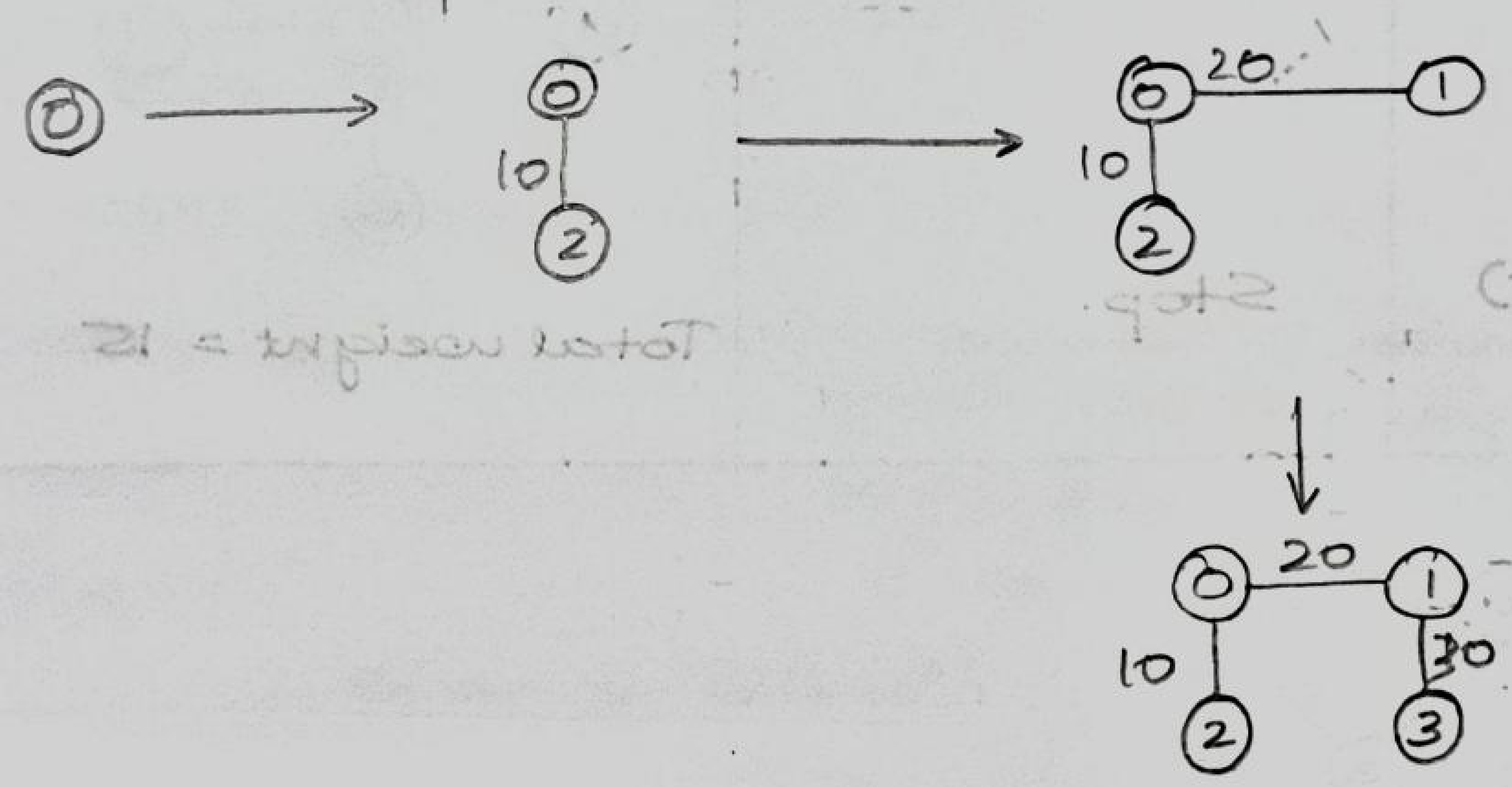
Property of spanning tree:-

$|E_T| = |V_T| - 1$

1)

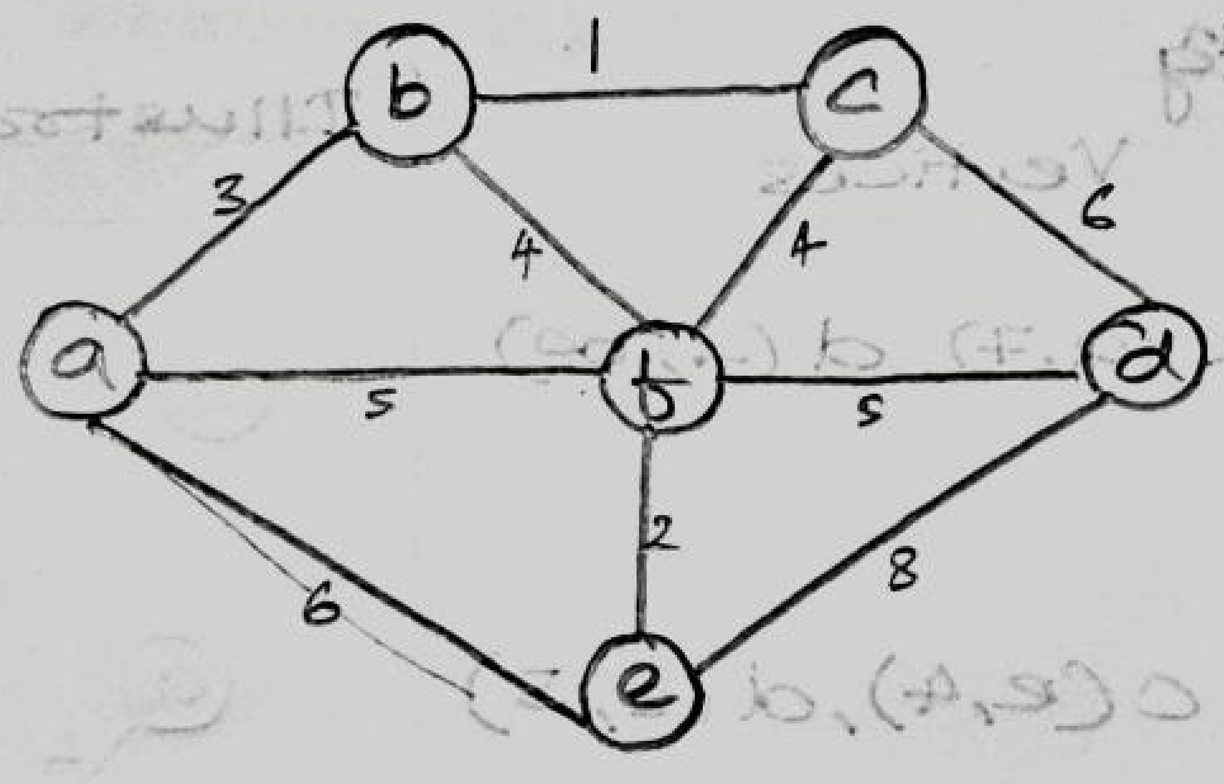


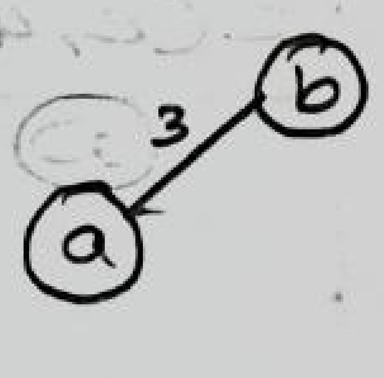
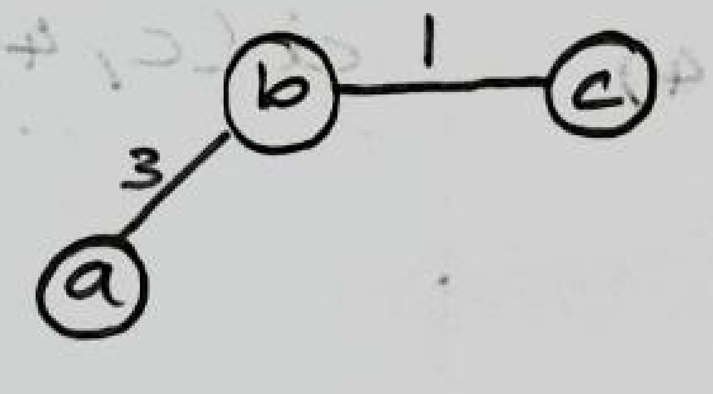
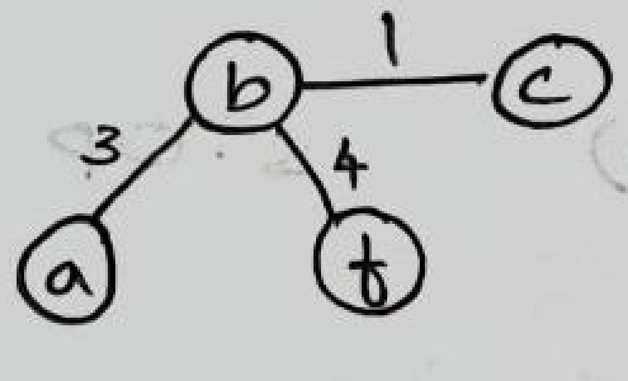
Tree Vertices	Remaining Vertices.
0(–,–)	1(0,20), $\boxed{2(0,10)}$, 3(–,∞)
2(0,10)	$\boxed{1(0,20)}$, 3(2,40)
1(0,20)	$\boxed{3(1,30)}$
3(1,30)	stop

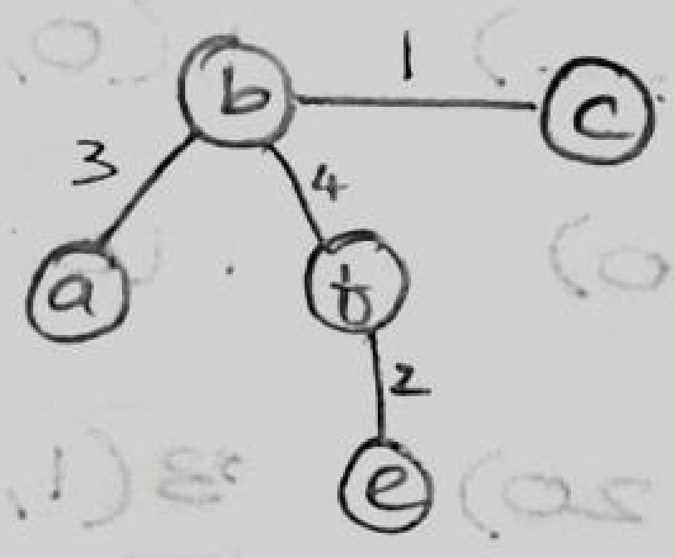
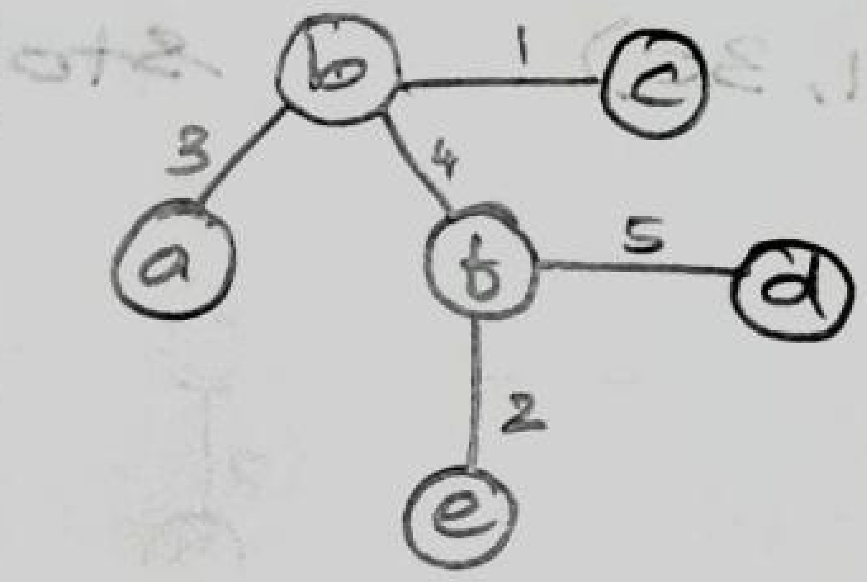


Total weight = 60.

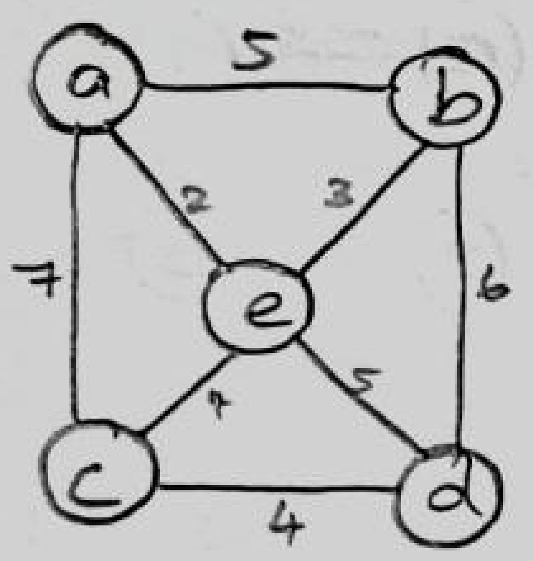
2.

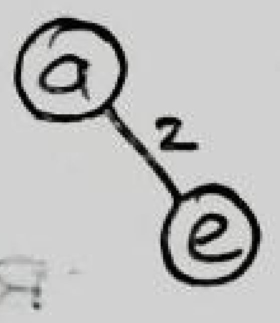
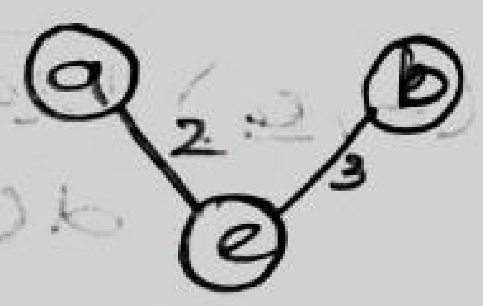
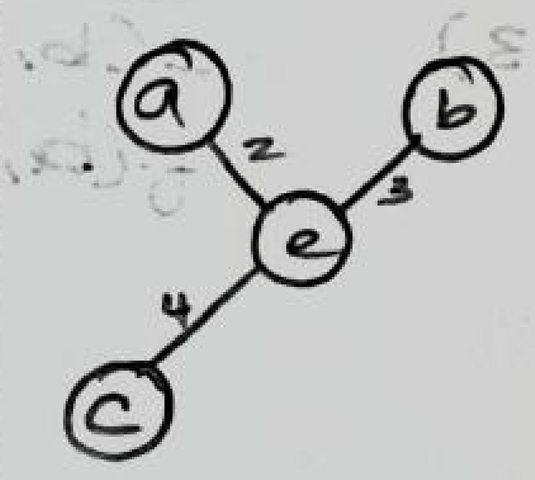
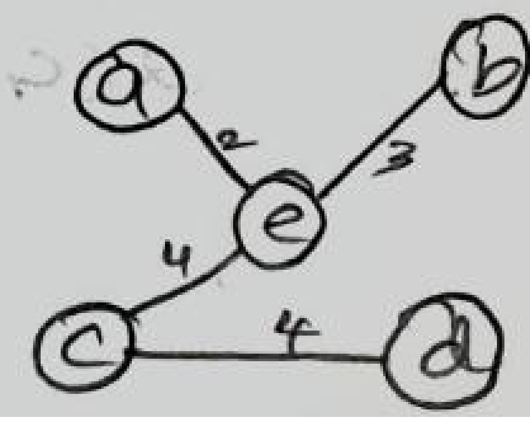


Tree Vertices	Remaining Vertices	Illustration.
a(–,∞)	$\boxed{b(a,3)}$ c(–,∞), d(–,∞), e(a,6), f(a,5)	
b(a,3)	$\boxed{c(b,1)}$, d(–,∞), e(a,6) f(b,4)	
c(b,1)	d(c,6), e(a,6), $\boxed{f(b,4)}$	

Tree Vertices	Remaining Vertices	Illustration.
f(b,4)	d(f,5), $\boxed{e(f,2)}$	
e(f,2)	$\boxed{d(f,5)}$	
d(f,5)	Stop.	Total weight = 15

3.



Tree Vertices	Remaining Vertices	Illustration.
a(–,–)	b(a,5), c(a,7), d(–,∞) $\boxed{e(a,2)}$	
e(a,2)	$\boxed{b(ae,3)}$, c(e,4), d(e,5)	
b(e,3)	$\boxed{c(e,4)}$, d(e,5)	
c(e,4)	$\boxed{d(c,4)}$	
d(c,4)	STOP	

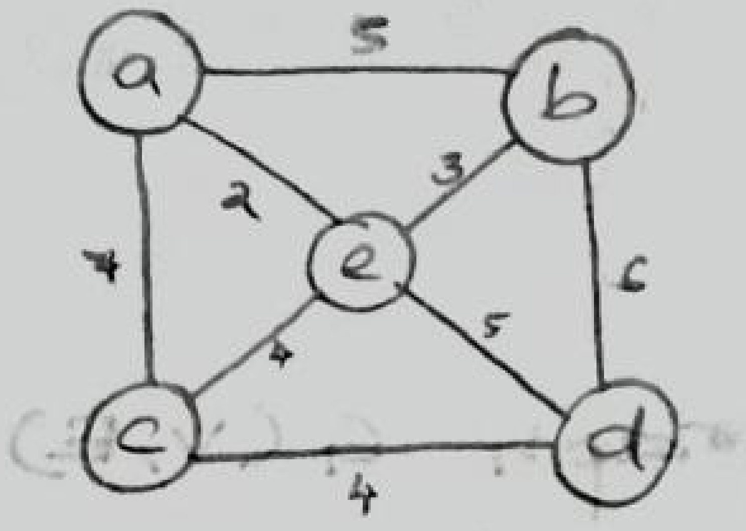
Total weight = 13.

Algorithm:-

```
Prims (G)
//Input: Weighted connected graph G (V,E)
//Output: $E_T$

    $V_T \leftarrow \{v_0\}$
    $E_T \leftarrow \phi$
    for $i \leftarrow$ ~~0~~1 to $|V|-1$ do
        find a minimum weight edge $e^* = (v^*, u^*)$
                from all the edges $(v, u)$
                such that $v^*$ is in $V_T$ and
                $u^*$ is in $V - V_T$
        $V_T \leftarrow V_T \cup \{u^*\}$
        $E_T \leftarrow E_T \cup \{e^*\}$
    return $E_T$
```

Kruskal's Algorithm:- (To find MST)

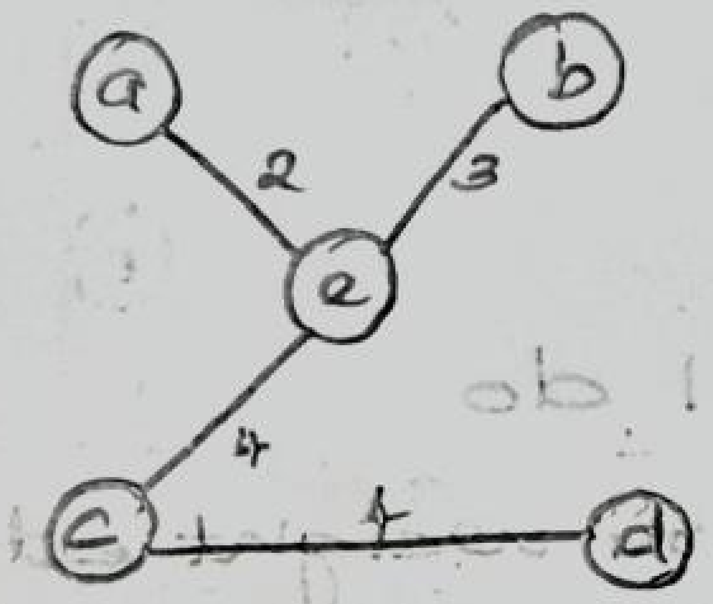


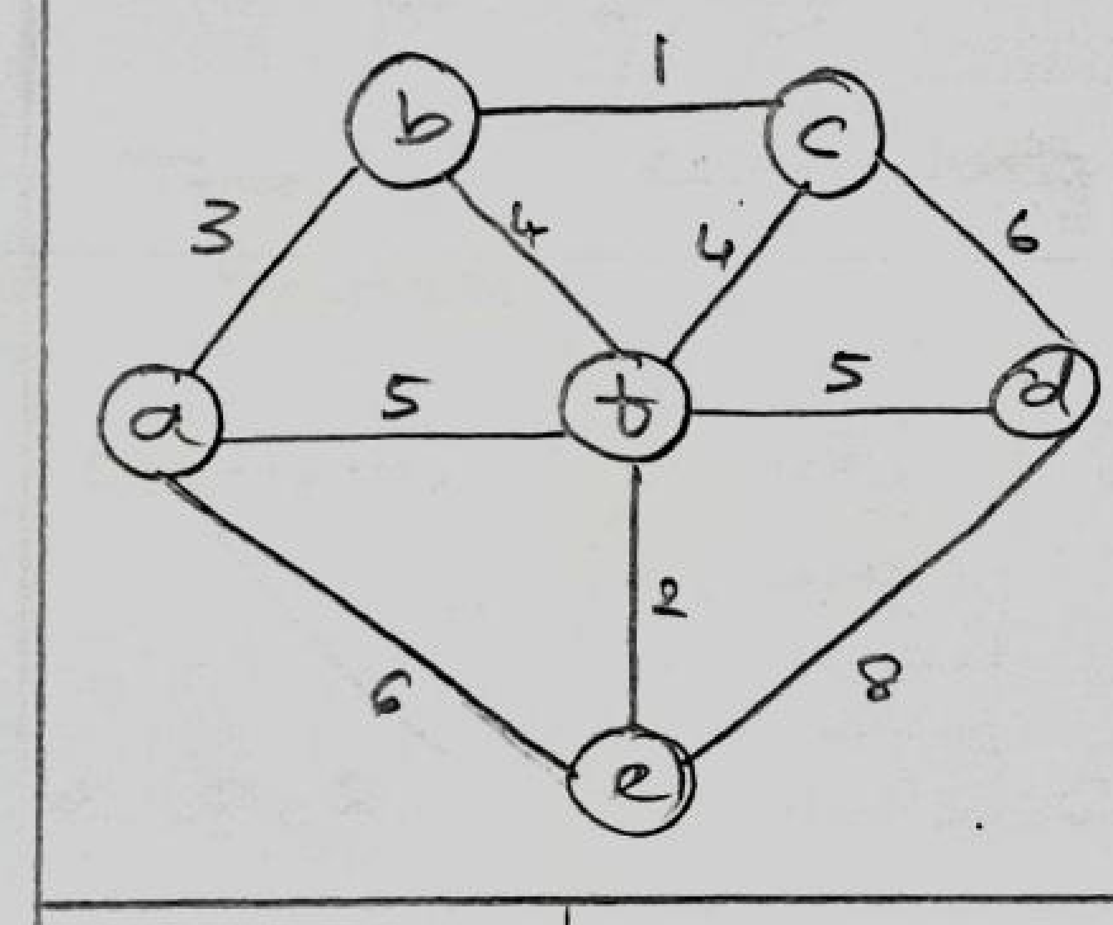
ae ✓	eb ✓	ec ✓	cd ✓	ad	ed	bd	ac
2	3	4	4	5	5	6	7

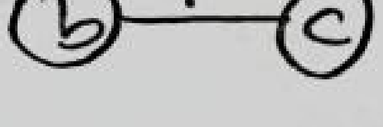
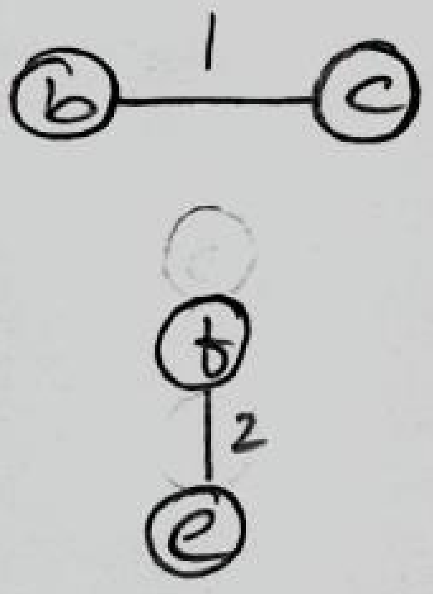
→ sort according to weight

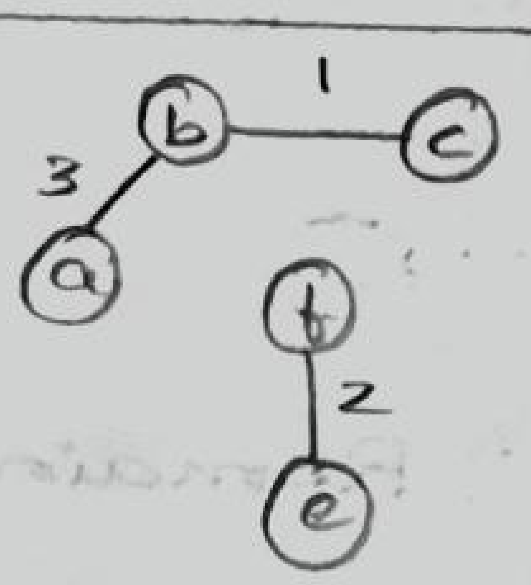
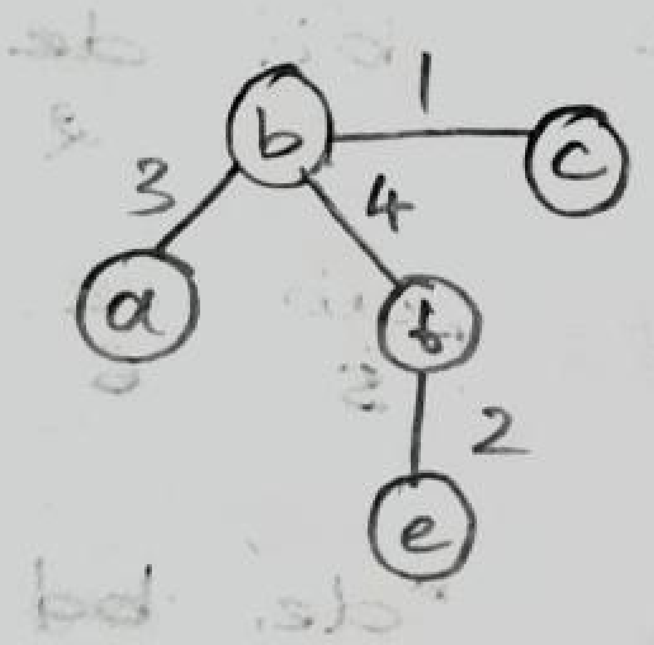
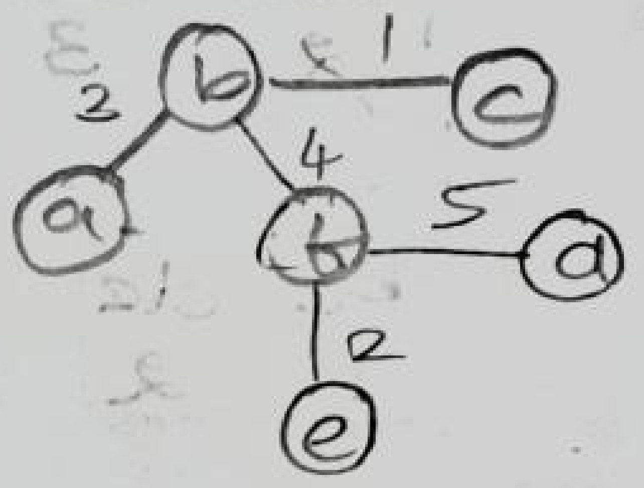
→ skip edges that make the graph cyclic.

→ add edges until all vertices are added.





Tree edges	Edge Set (sorted)	Illustration.
bc 1	bc ✓ ~~f~~ef ab bf cf 1 2 3 4 4 af df cd ae de 5 5 6 6 8	
ef 2	bc ✓ ef ✓ ab bf cf 1 2 3 4 4 af df cd ae de 5 5 6 6 8	

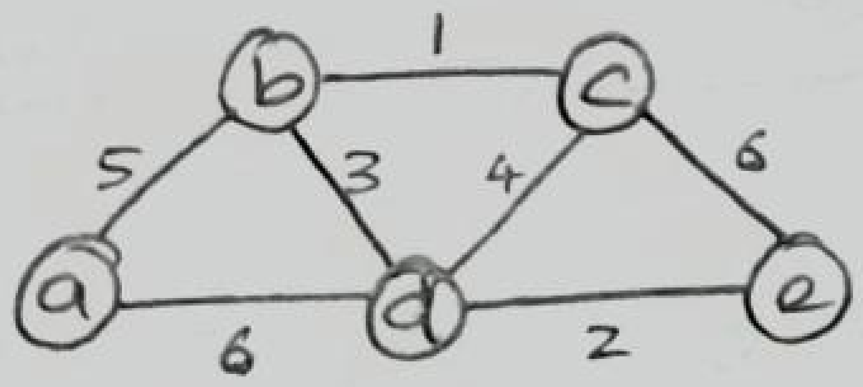
ab 3	bc ef ab bf cf 1 2 3 4 4 af df cd ae de 5 5 6 6 8	
bf 4	bc ef ab bf cf 1 2 3 4 4 af df cd ae de 5 5 6 6 8	
df 5	bc ef ab bf cf 1 2 3 4 4 af df cd ae de 5 5 6 6 8	

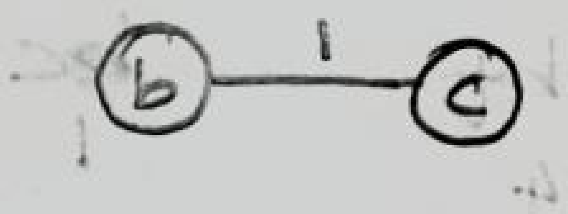
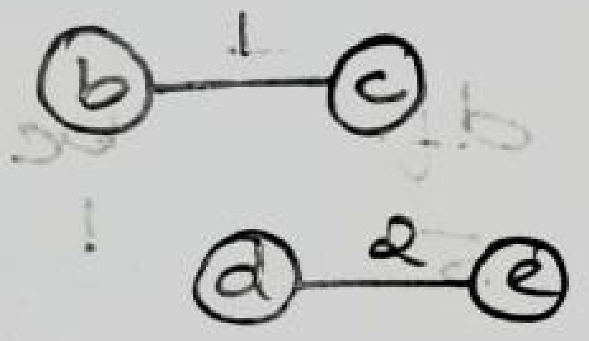
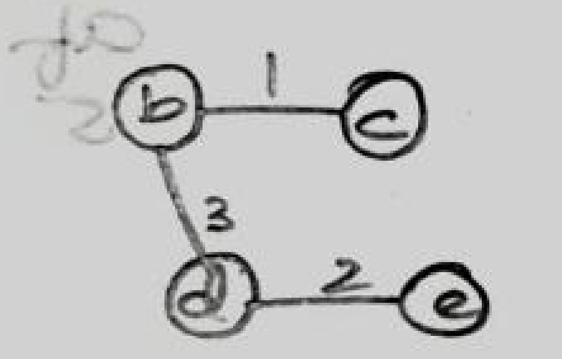
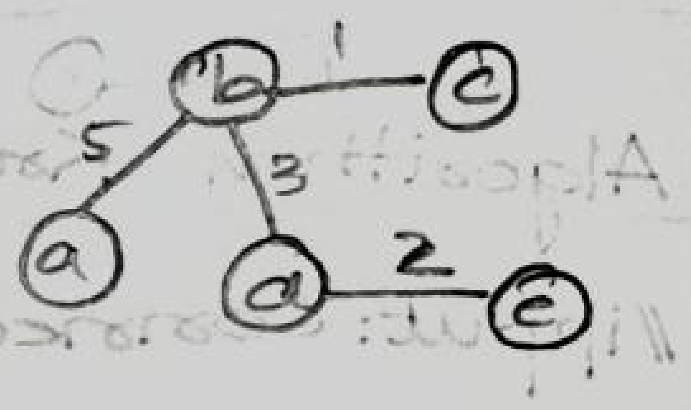
Algorithm Kruskals (G)

```
//input: Connected weighted graph G(V, E)
//output: $E_T$

• sort the edges in E in non-decreasing order.
        of the edge weight.
    $w(e_{11}) \le w(e_{12}) \le \dots \le w(e_{i|E|})$

    ecounter ← 0
    $E_T$ ← ϕ
    k ← 0 //no. of edges processed
    while ecounter < |V|-1 do
        k ← k+1
        if $E_k \cup \{e_{ik}\}$ is acyclic.
            $E_T \leftarrow E_T \cup \{e_{ik}\}$
            ecounter ← ecounter +1
    return $E_T$
```



Tree vertices	Remaining Vertices	Illustration.
bc 1	bc 1 ✓ de 2 bd 3 cd 4 ab 5 ce 6	
de 2	bc 1 ✓ de 2 ✓ bd 3 cd 4 ab 5 ce 6	
bd 3	bc 1 ✓ de 2 ✓ bd 3 ✓ cd 4 ab 5 ce 6	
ab 5	bc 1 ✓ de 2 ✓ bd 3 ✓ cd 4 ✗ ab 5 ✓ ce 6	

HUFFMAN TREES:- → variable length encoding.

fixed length encoding → ASCII

Step1: Initialize 'n' one-node trees. and label them with the characters of the alphabet. Record the frequency of each character in its tree's root to indicate its weight (The weight of a tree will be equal to the sum of frequencies in the leaves)

Step2: Repeat the following operation until single tree is obtained.

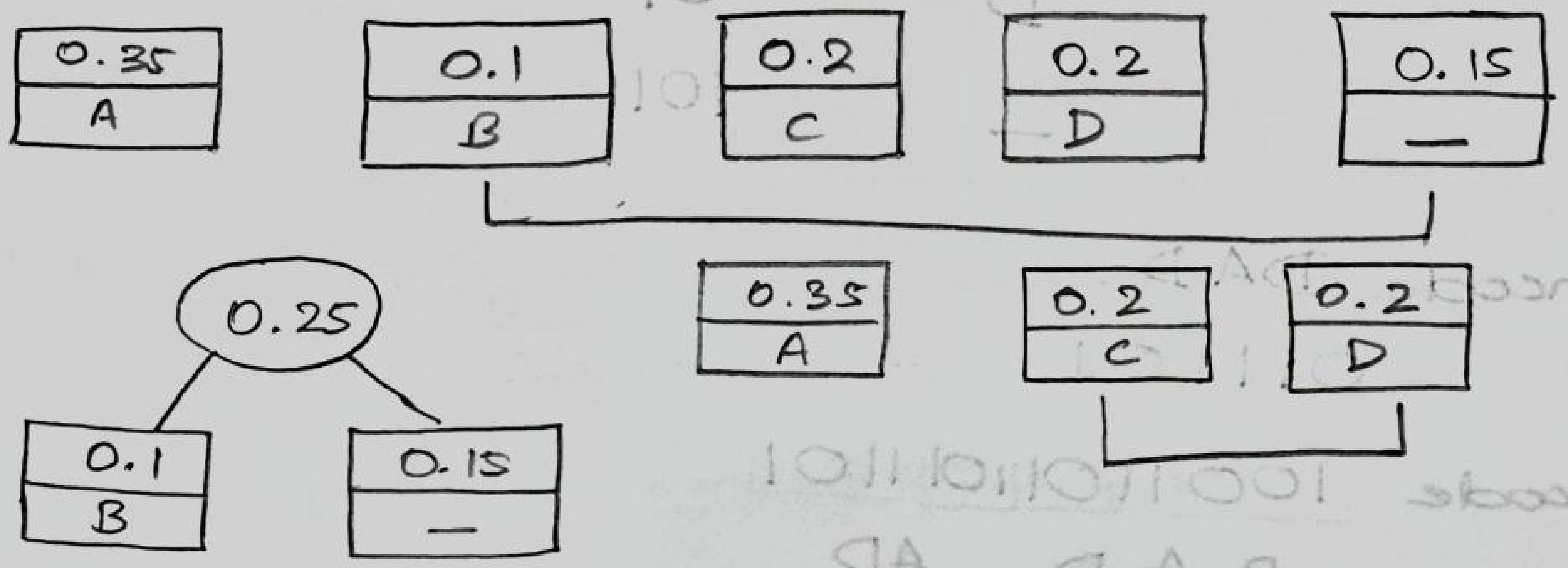
↳ Find 2 trees with the smallest weights (ties can be broken arbitrarily)

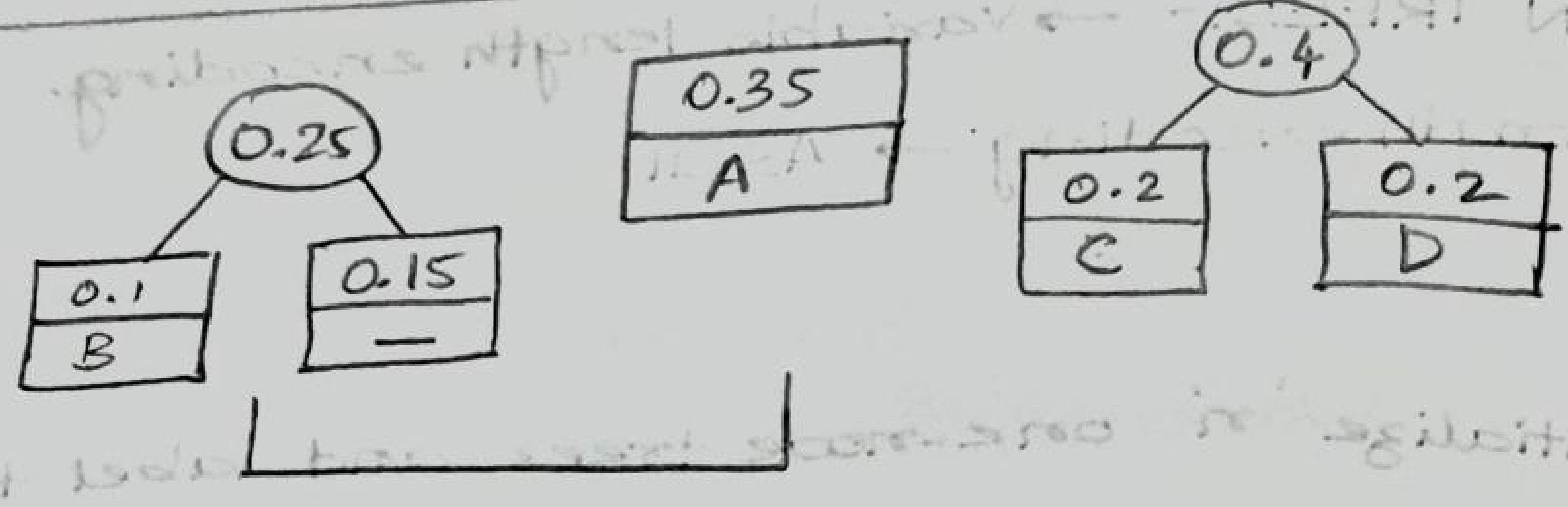
↳ Makes them the left and right subtree of a new tree and record the sum of their weights in the root of the new tree as its weight.

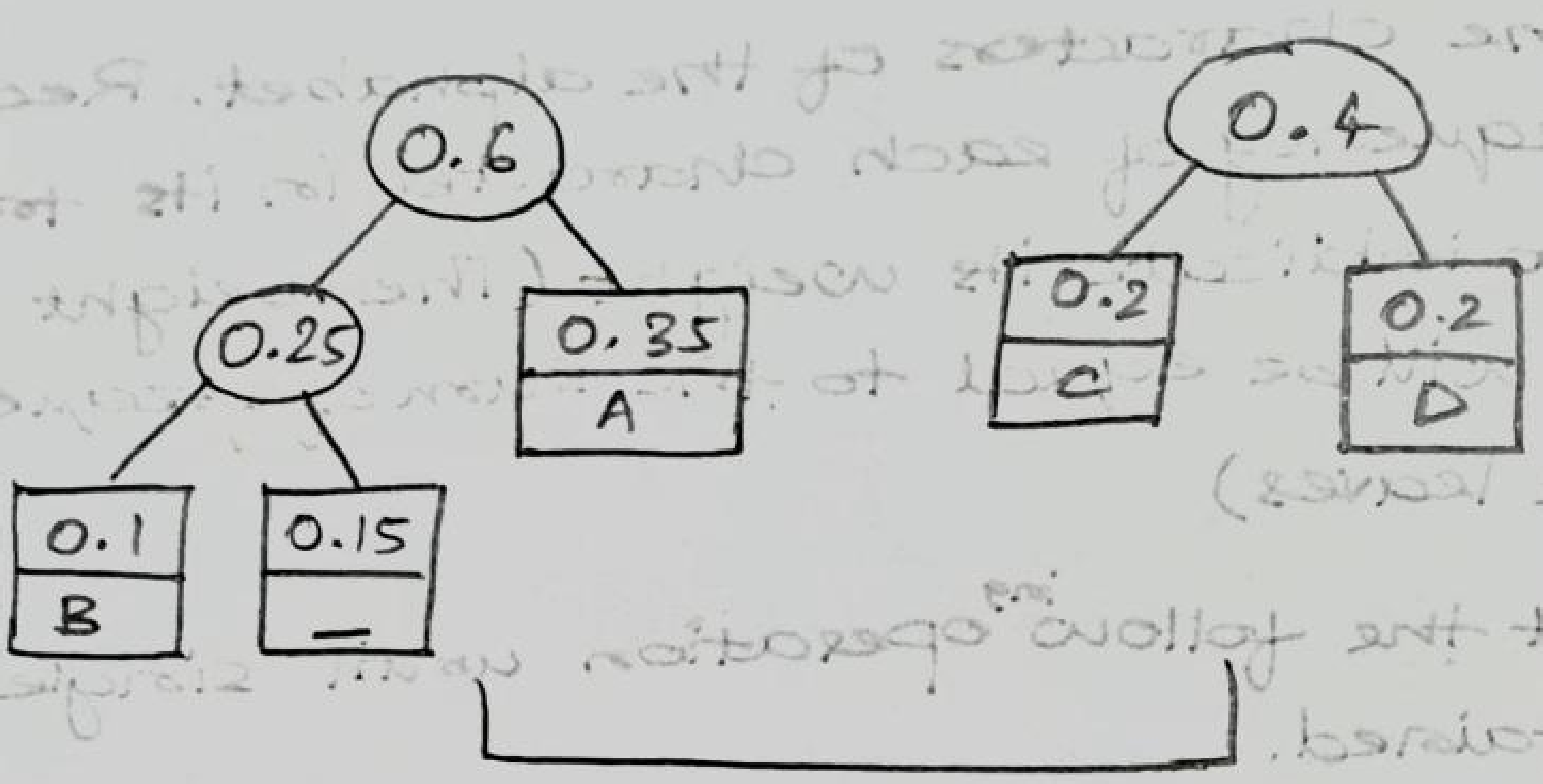
The tree constructed by the above algorithm is Huffman tree.

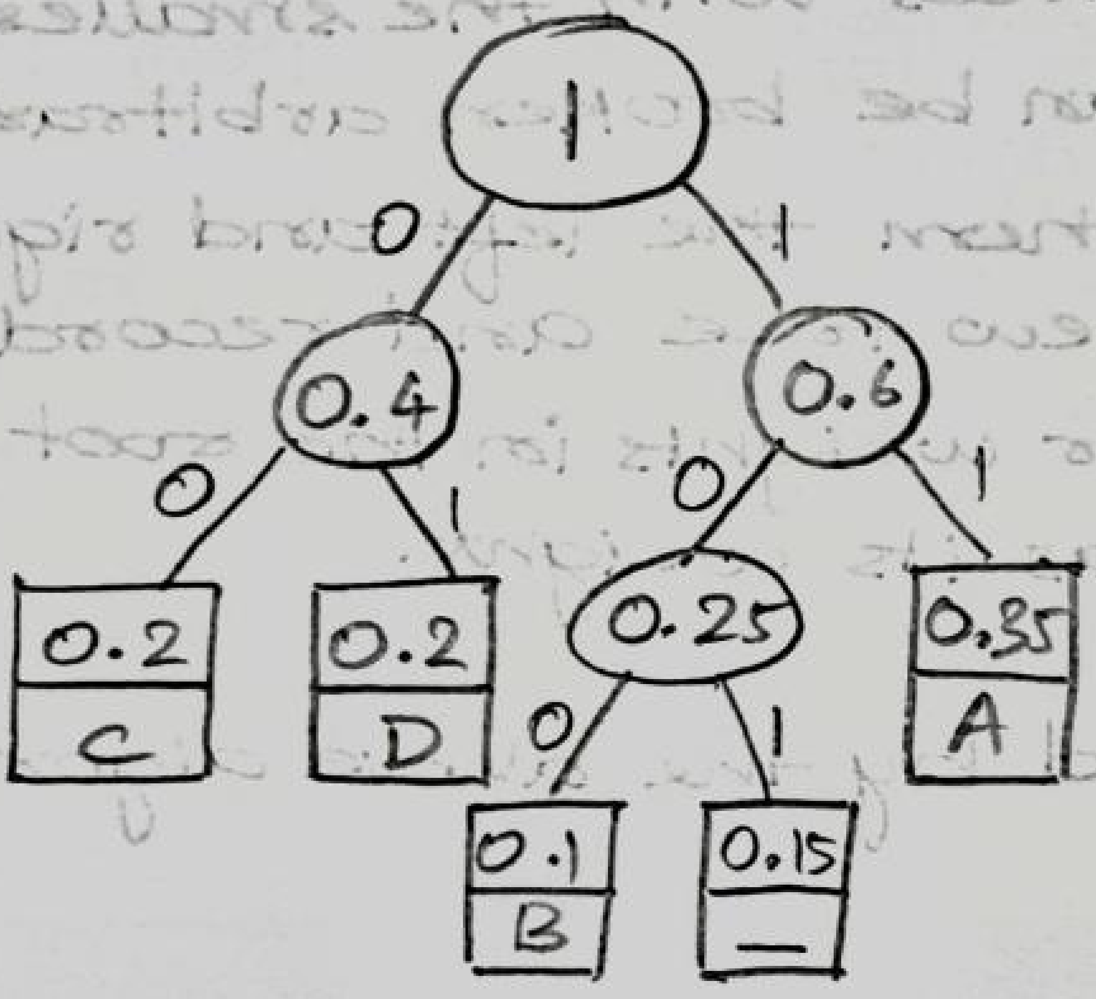
1. Consider the following 5 character alphabet {A, B, C, D, _} with the following occurance probabilit

character	A	B	C	D	_
probability.	0.35	0.1	0.2	0.2	0.15









Character	Code.
A	11
B	100
C	00
D	01
—	101

Encode DAD :-

011101

Decode 1001101101101

B A D — AD

Average number of bits required:-

$$2\times0.35+3.0.1+2\times0.2+2\times0.2+3\times0.15$$
$$= 2.25$$

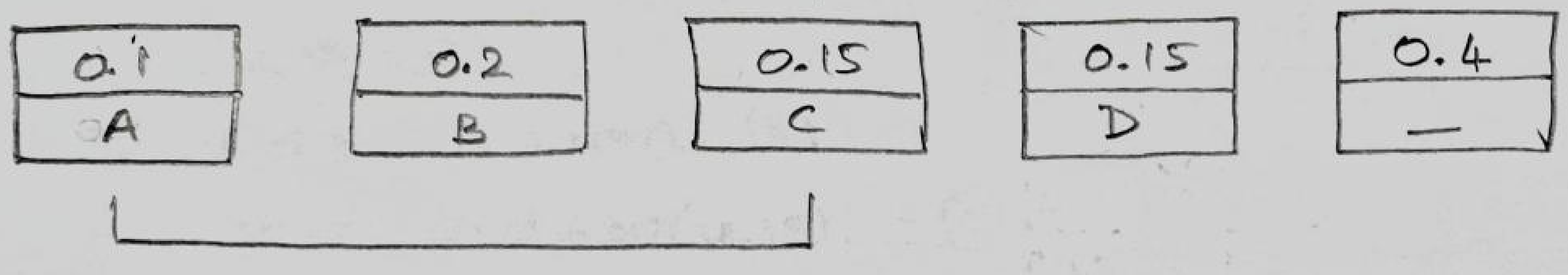
Average number of bits required for fixed length encoding:- <u>3bits</u>

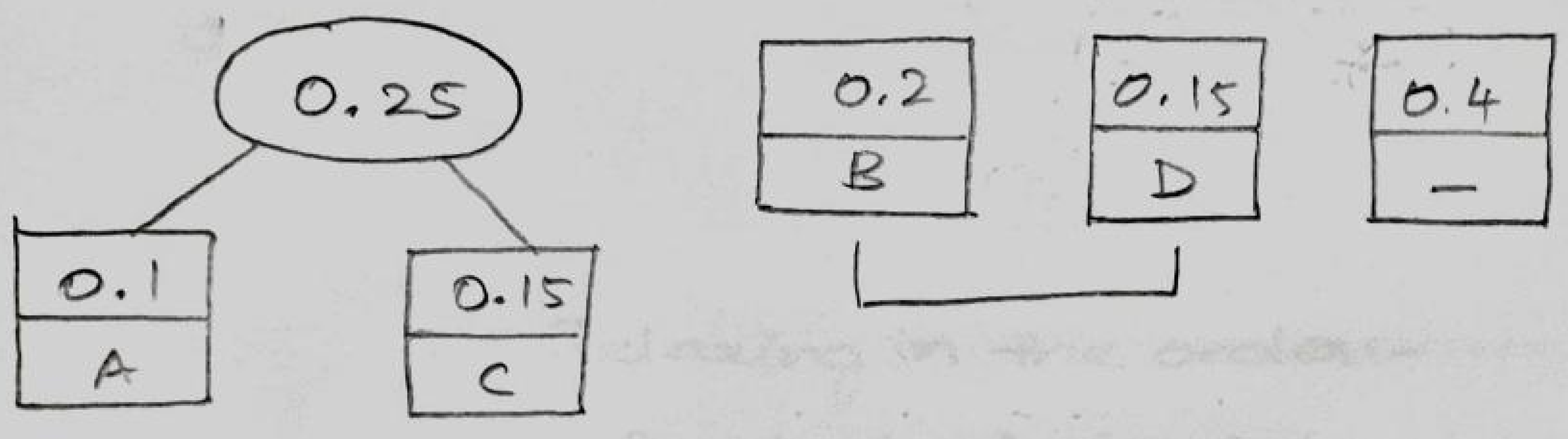
Compression:-

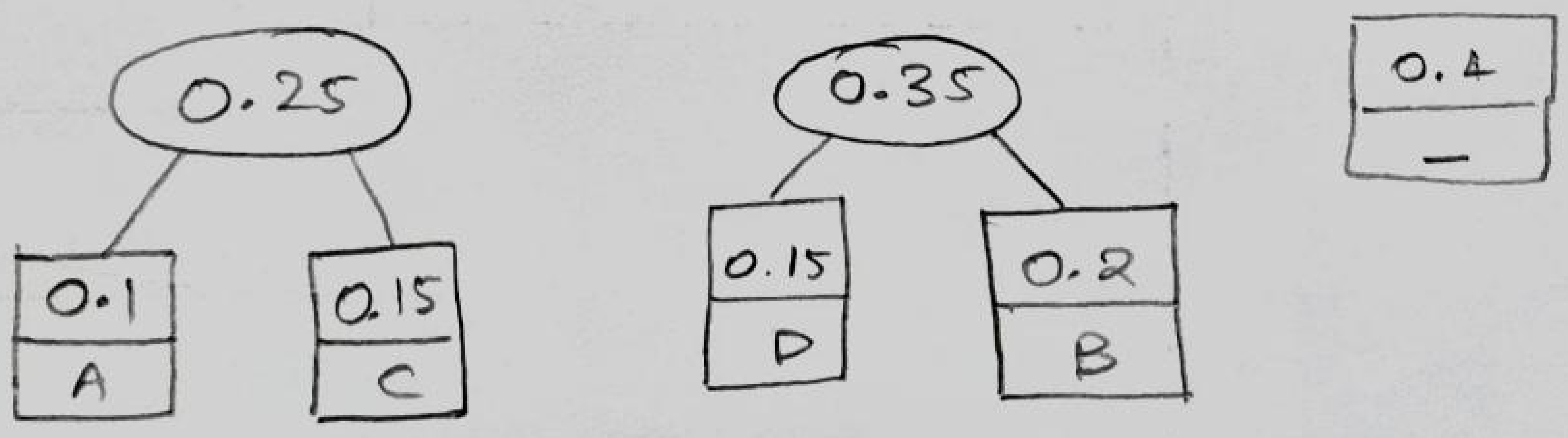
$$\frac{3-2.25}{3}\times 100 = 25\%$$

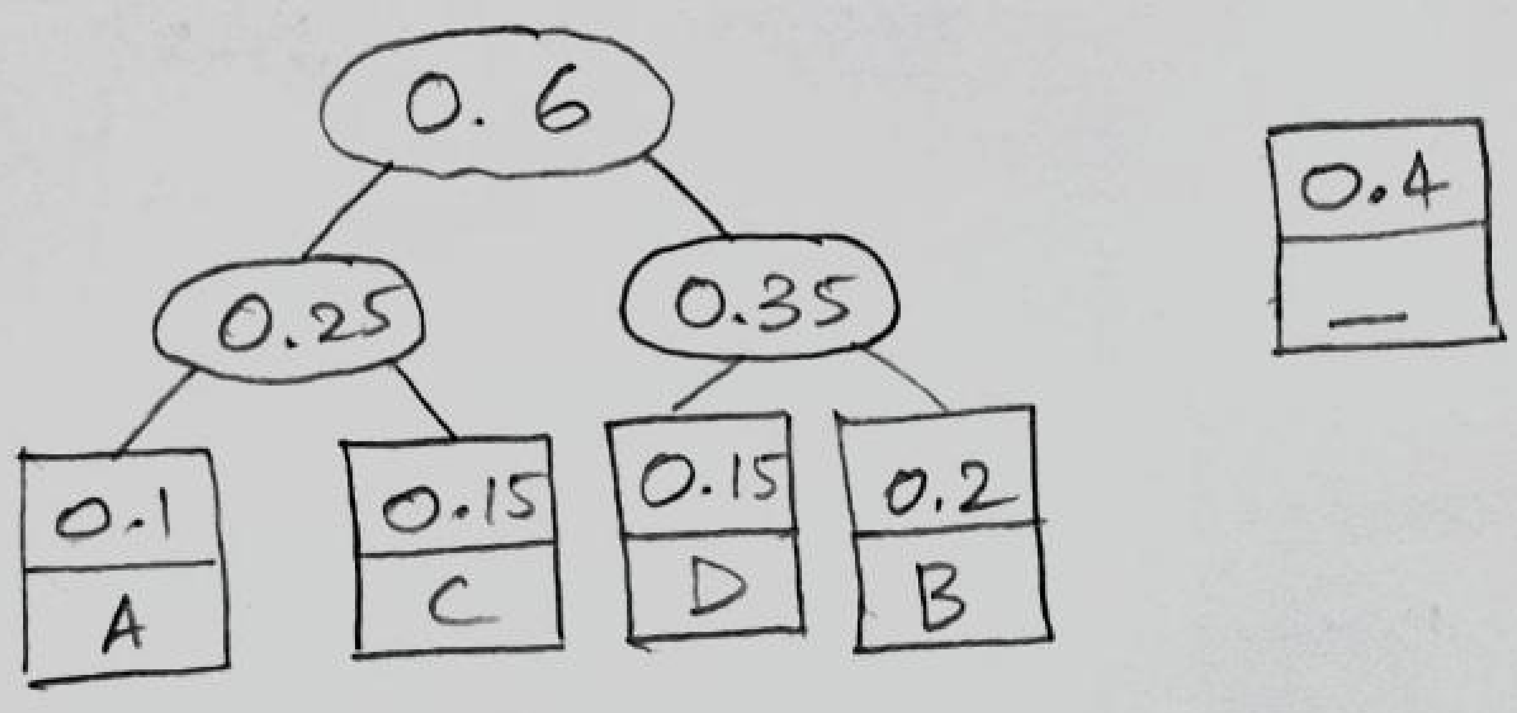
2. Consider the probability:-

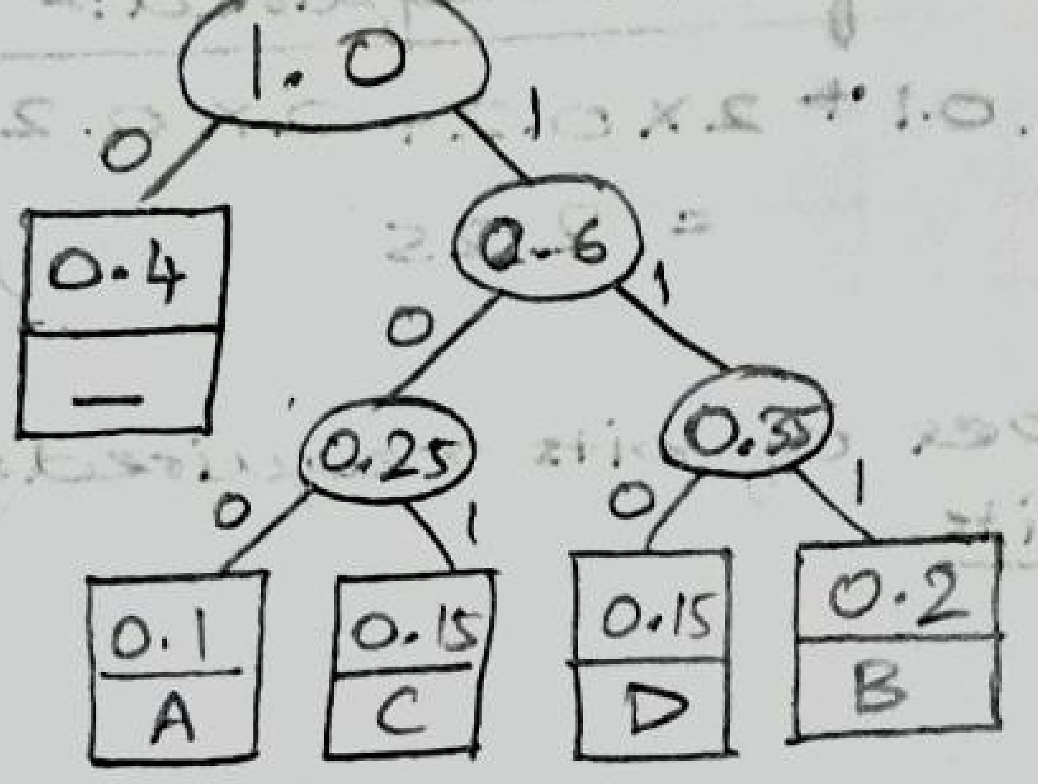
Characters	A	B	C	D	_
probability.	0.1	0.2	0.15	0.15	0.4





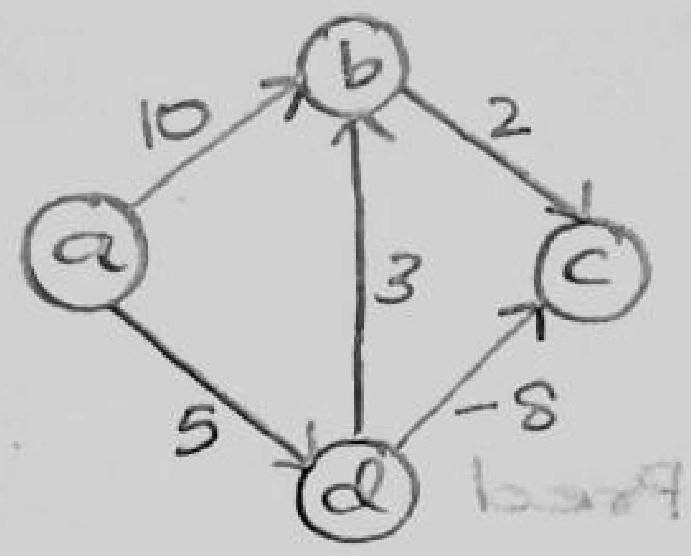






characters	code
A	100
B	111
C	101
D	110
—	0

SINGLE - SOURCE SHORTEST PATH PROBLEM:-

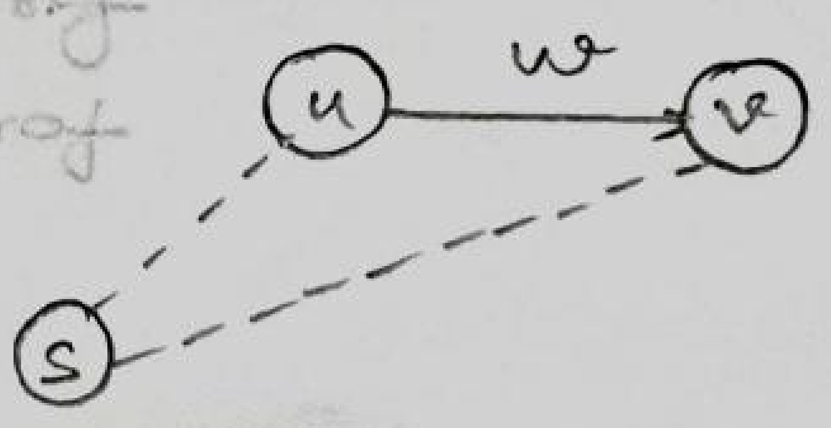


Algorithms common to all single source.

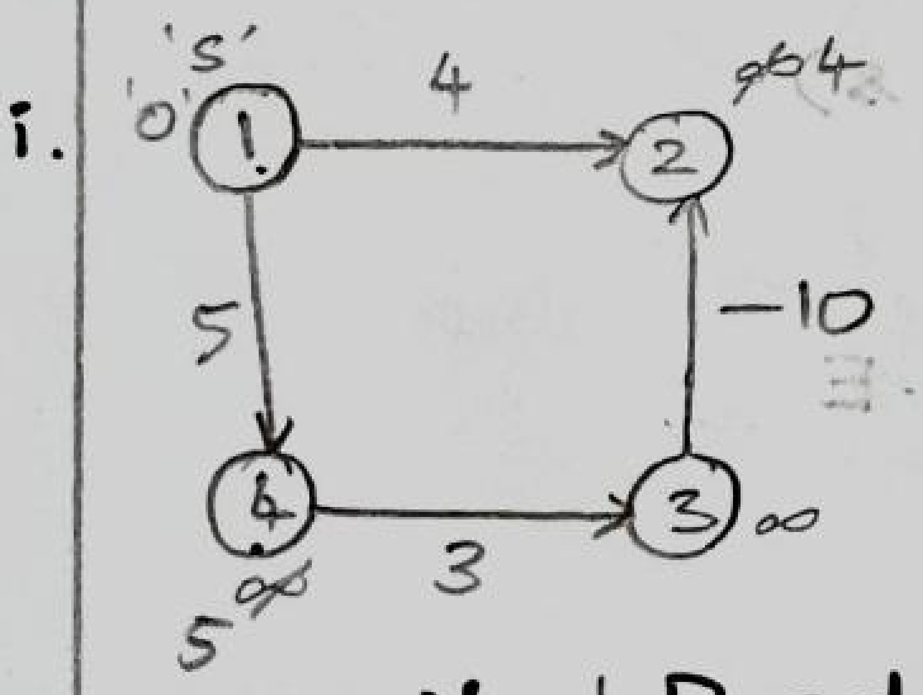
1. BELLMAN - FORD ALGORITHM:-

```
Initialize_single_source (G, s)
    for each vertex v in G.V
        v.d = ∞
        v.π = NIL
    s.d = 0

Relax (u, v, w)
    if v.d > u.d + w(u, v)
        v.d = u.d + w(u, v)
        v.π = u
```

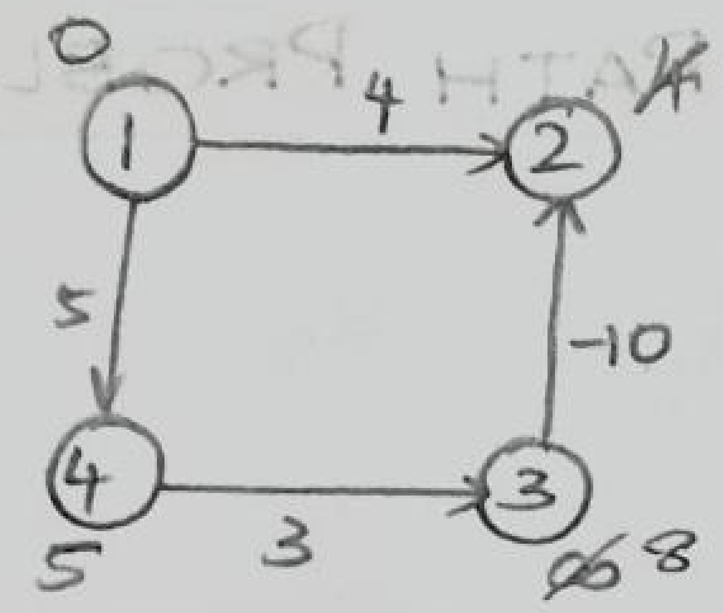


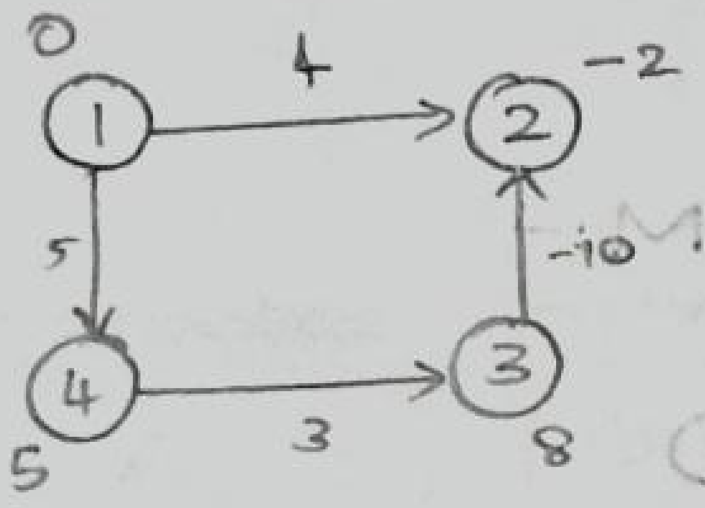
i.



Relaxing in the order:-
(4,3), (1,2), (3,2), (1,4)

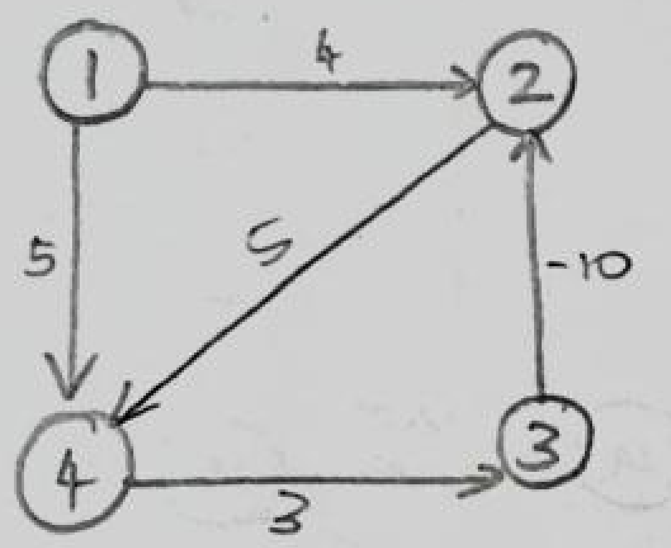
v	Pred
1	NIL
2	~~NIL~~ → ~~1~~ → 3
3	~~NIL~~ → 4
4	~~NIL~~ → 1





v	Pred
1	NIL
2	3
3	4
4	1

ii)



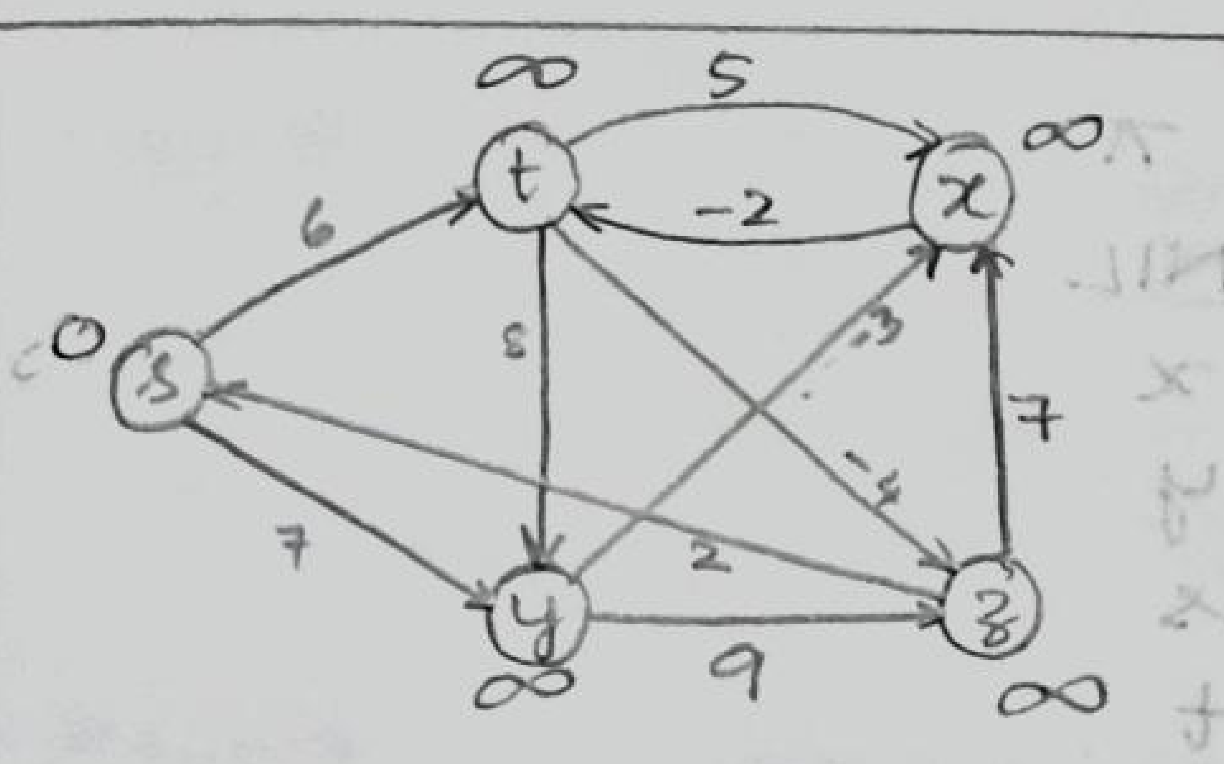
This graph has a negative weight edge cycle → hence we can never find shortest path using Bellman-ford Algorithm.

Algorithm:-

```
Bellman_ford (G,w,s)
    Initialize-single-source (G, s)
    for i ← 1 to |G.V| - 1 do
        for each edge (u, v) ∈ G.E
            Relax (u, v, w)

    for ~~for~~ each edge(u,v) ∈ G.E.
        if ∈v.d > u.d + w (u, v)
            return False.
    return True
```

iii)



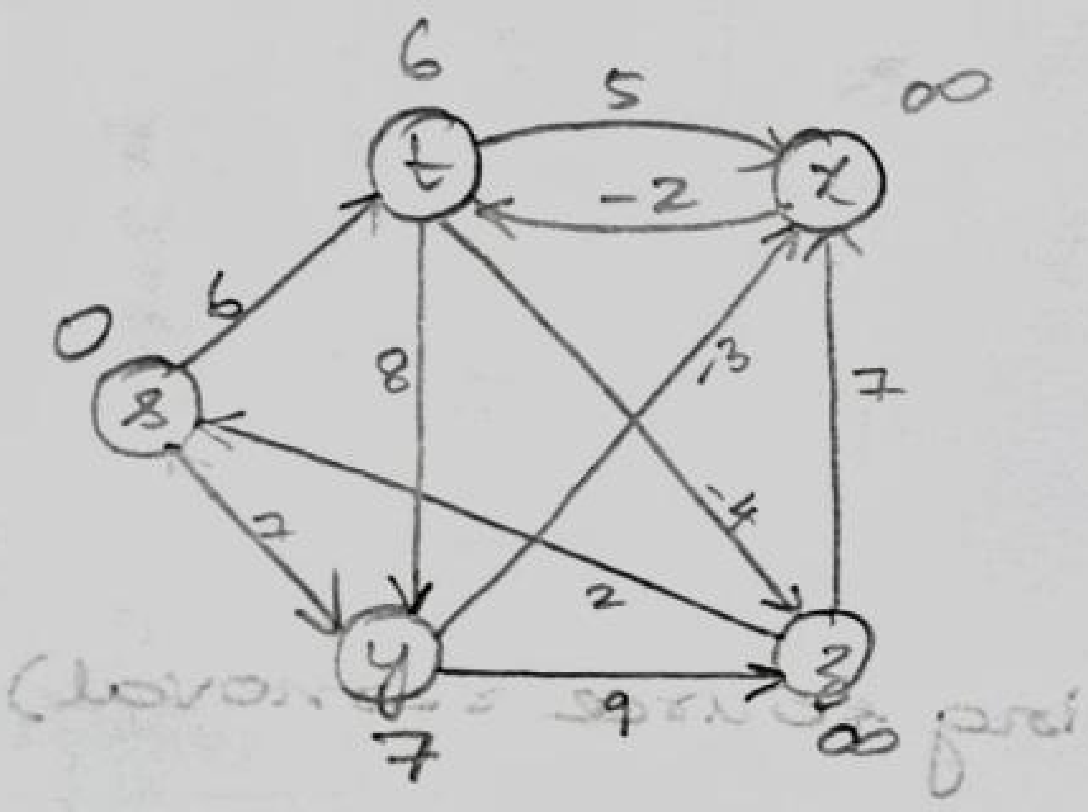
Vertex	π
s	NIL
t	~~NIL~~ s
x	~~NIL~~ t
y	~~NIL~~ s
z	~~NIL~~ t

Order of relaxation:-

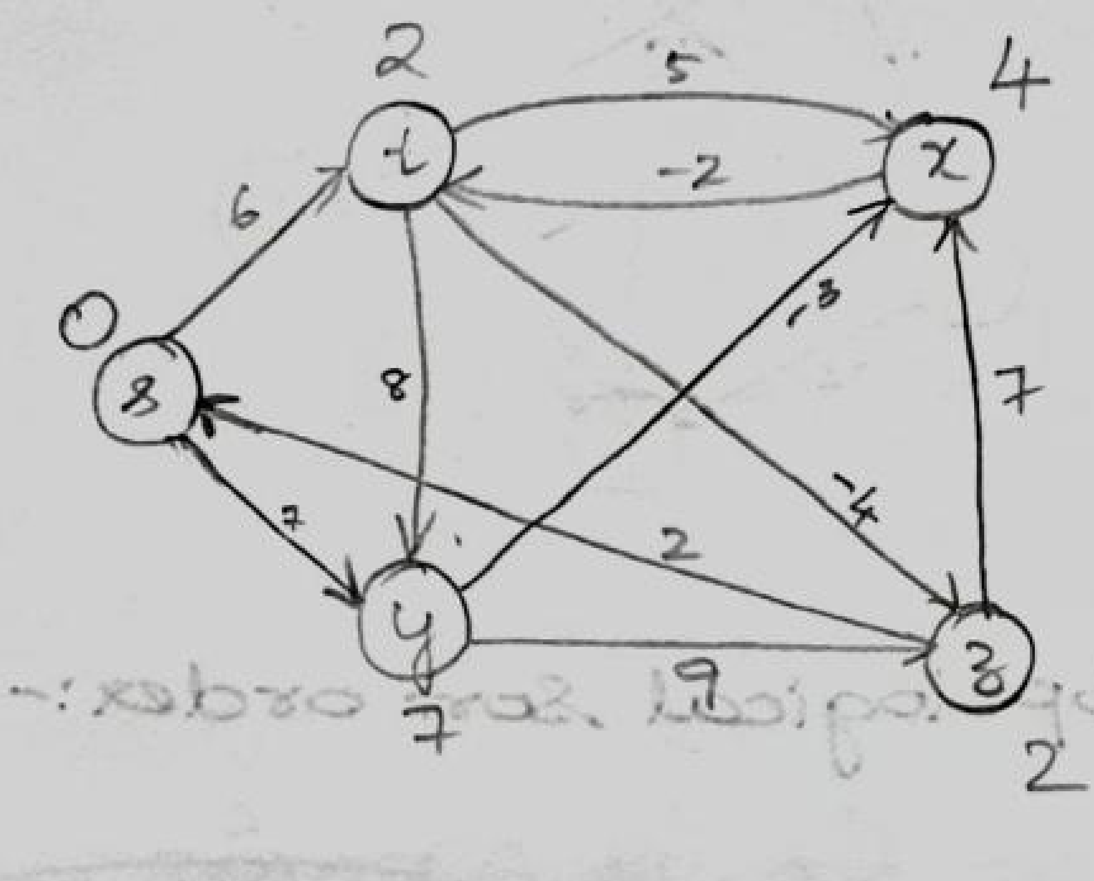
(t, x), (t, y), (t, z), ~~(t, z~~, (x, t), (y, x), (y, z), (z, x)

(z, s), (s, t) (s, y).

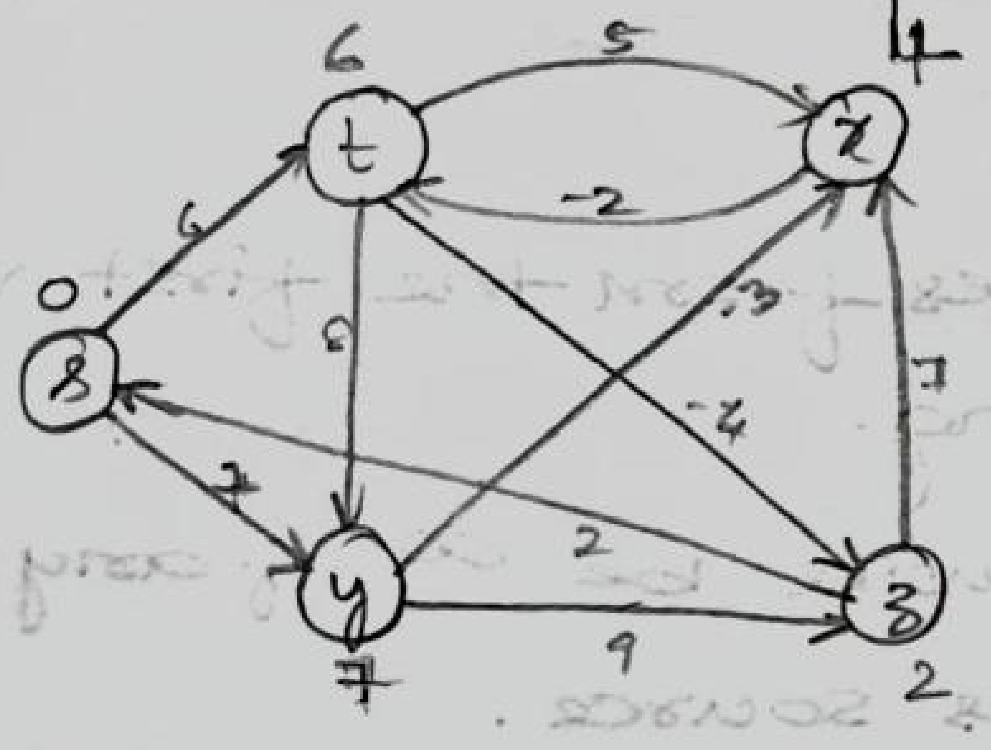
Step 1:-



Step 3:-



Step 2:-

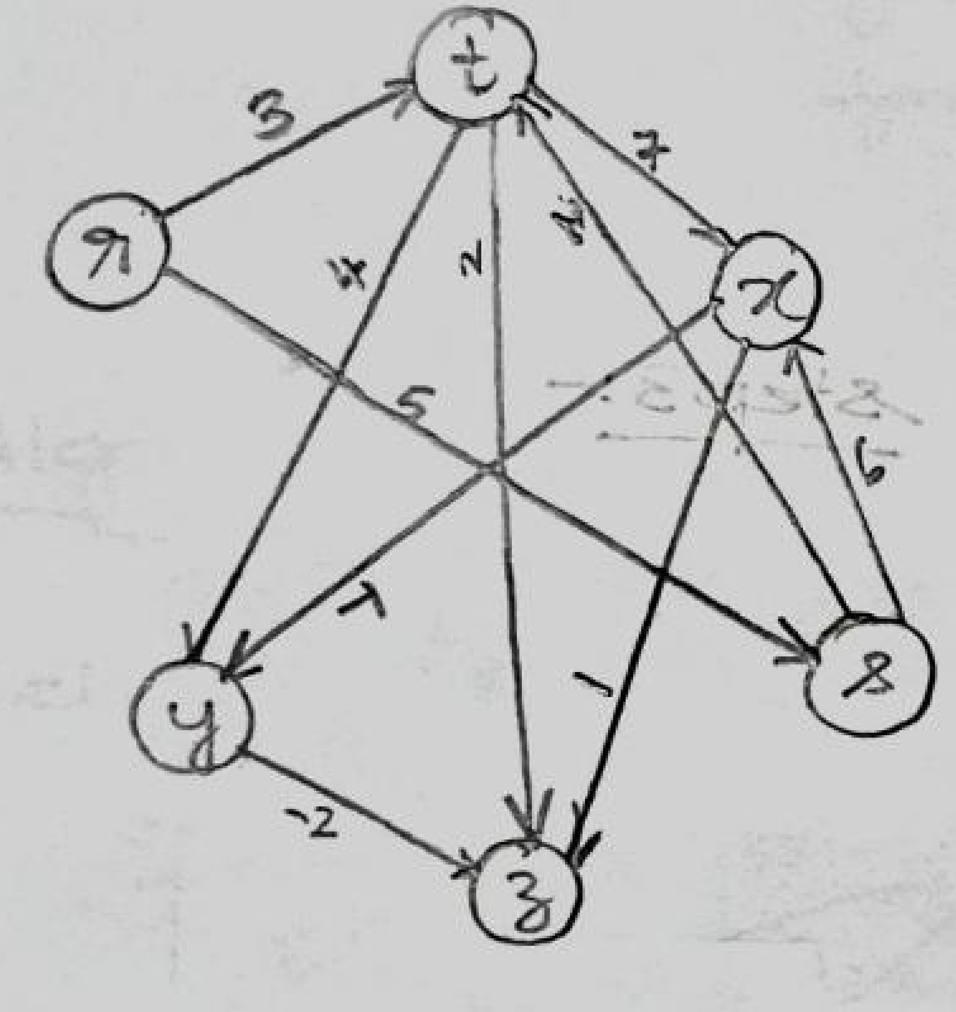


Vertex	d	π
s	0	NIL
t	2	x
x	4	y
y	7	s
z	-2	t

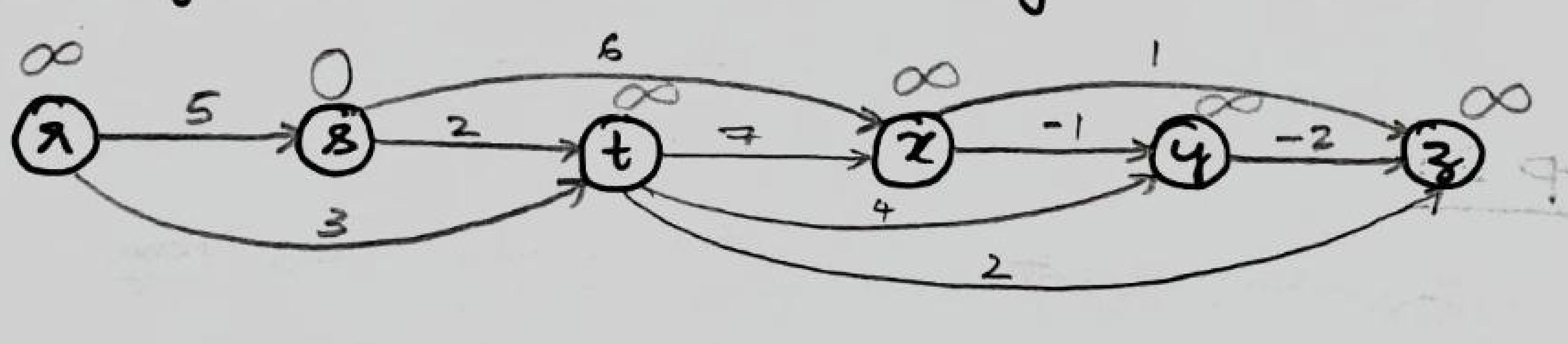
To check if there is a negative-weight cycle:
since no more relaxations can be done hence there is no negative-weight cycle

SINGLE SOURCE SHORTEST PATH IN DAG:-

↓ Directed Acyclic graph



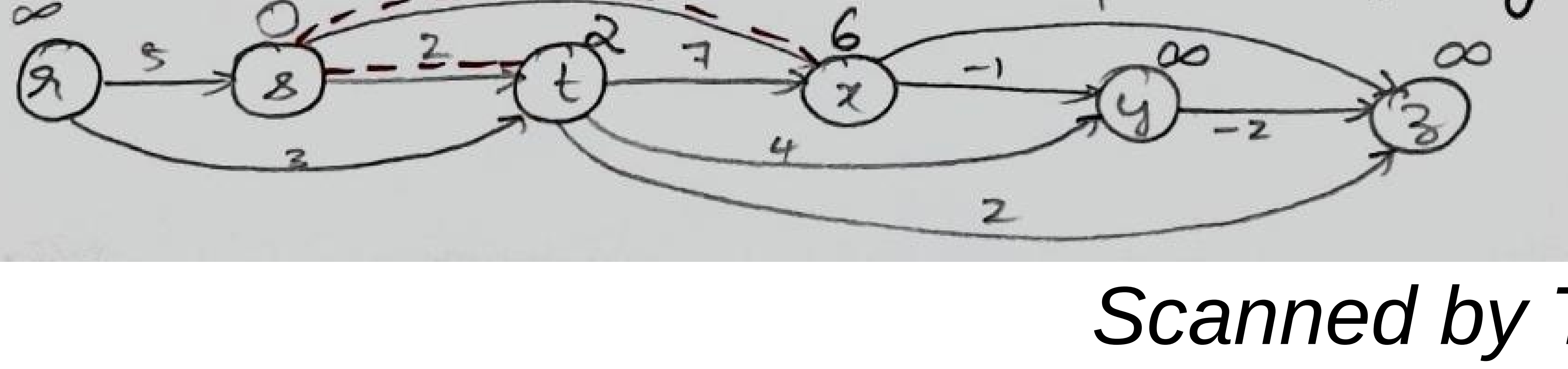
Topological sort order:- (using source removal)



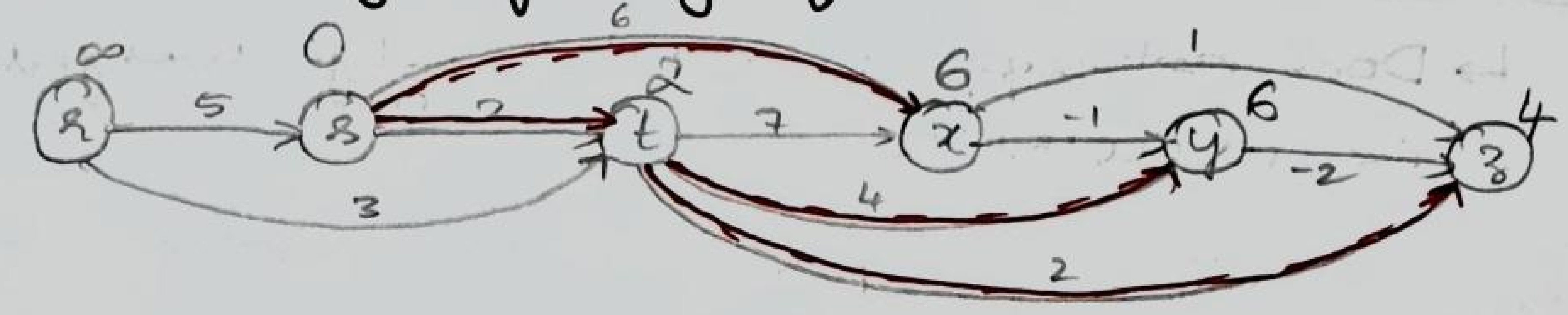
Step 1: Relax outgoing edges from the first vertex in topological sorting.
(No updation)

NOTE:-(here r) First vertex will be ∞ if any other vertex is considered as source.

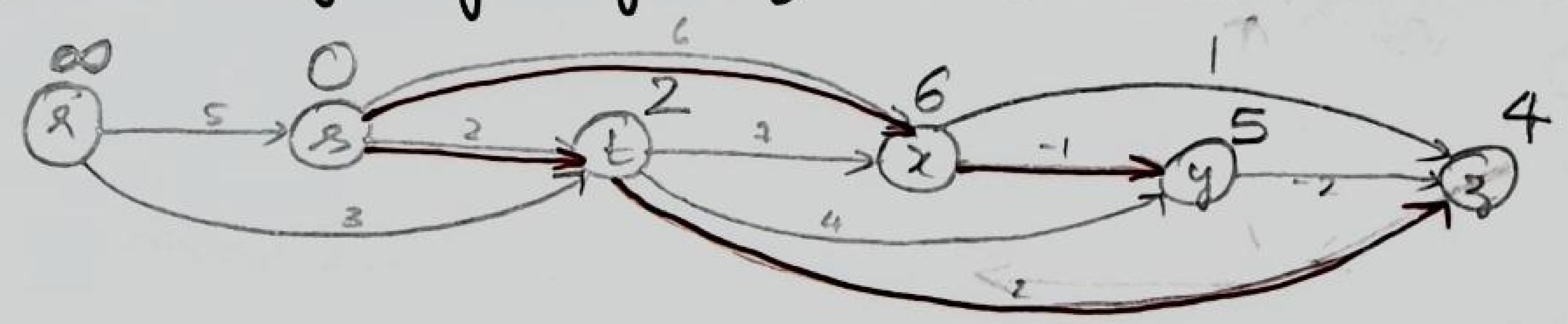
step2: Go to the next vertex. Relax outgoing edges.
(s)



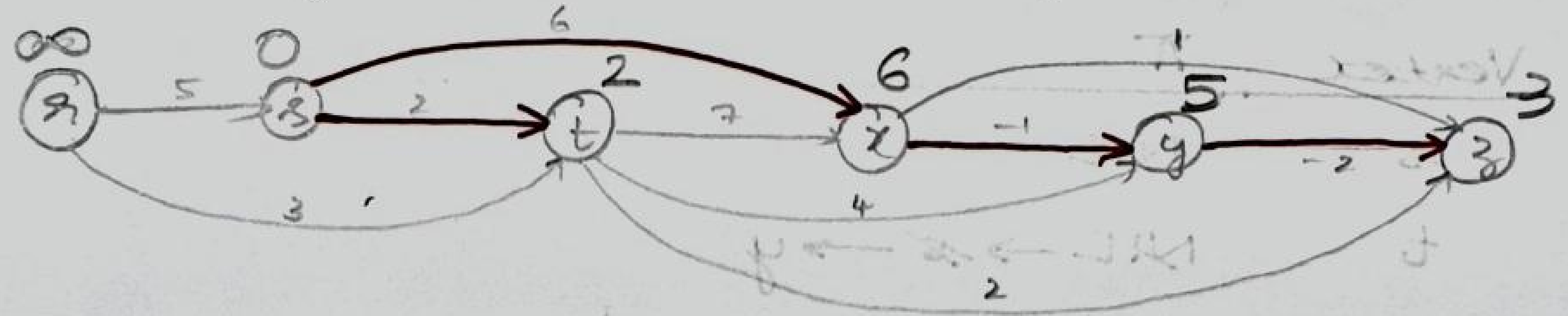
step 3 : outgoing edges from t.



Step 4 : outgoing edges from x.



Step 5: outgoing edges from y.



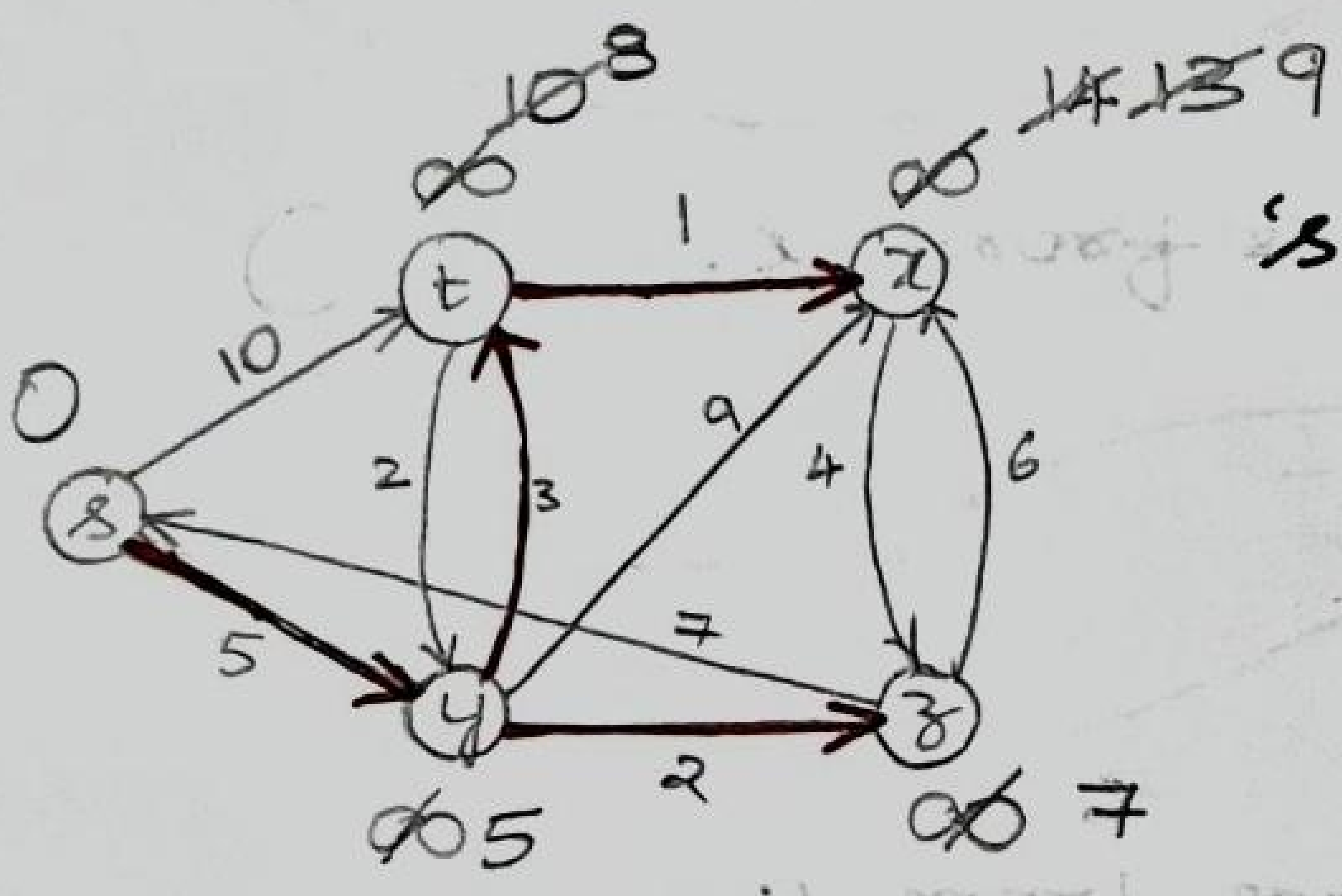
Vertex	d	π
r	∞	NIL
s	0	NIL
t	2	s
x	6	s
y	5	x
z	3	y

Algorithm:-

```
Single_Source_Shortest_path_DAG (G, w, s)
    topologically sort the vertices in G
    Initialize_single_source (G, s)
    for each vertex u, taken in topological sort order do.
        for each vertex v ∈ G.adj [u] do.
            Relax (u, v, w).
```

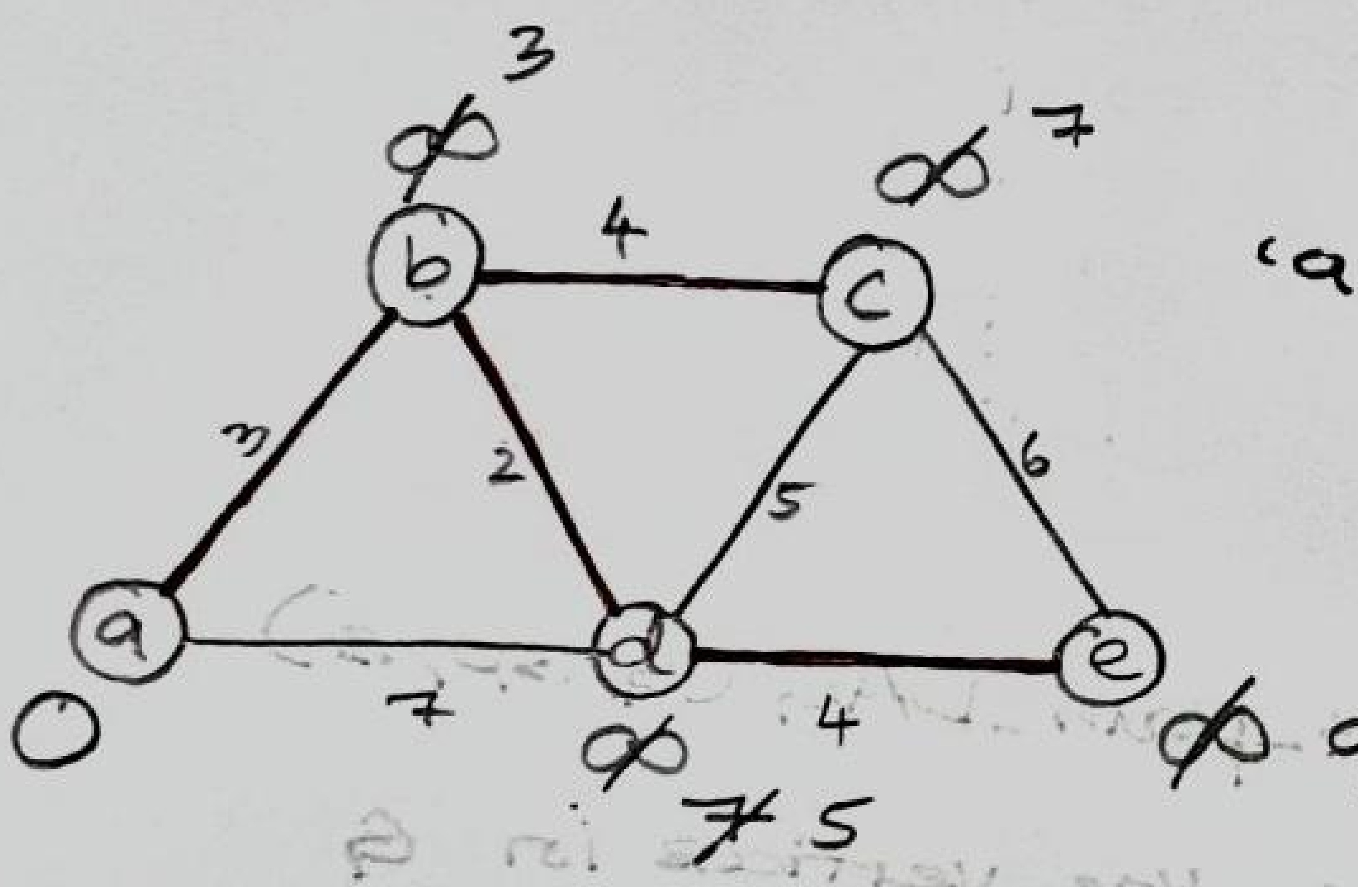
DIJKSTRA'S ALGORITHM:-

↳ Does not support negative edge weights.
i.e. w(u,v) > 0



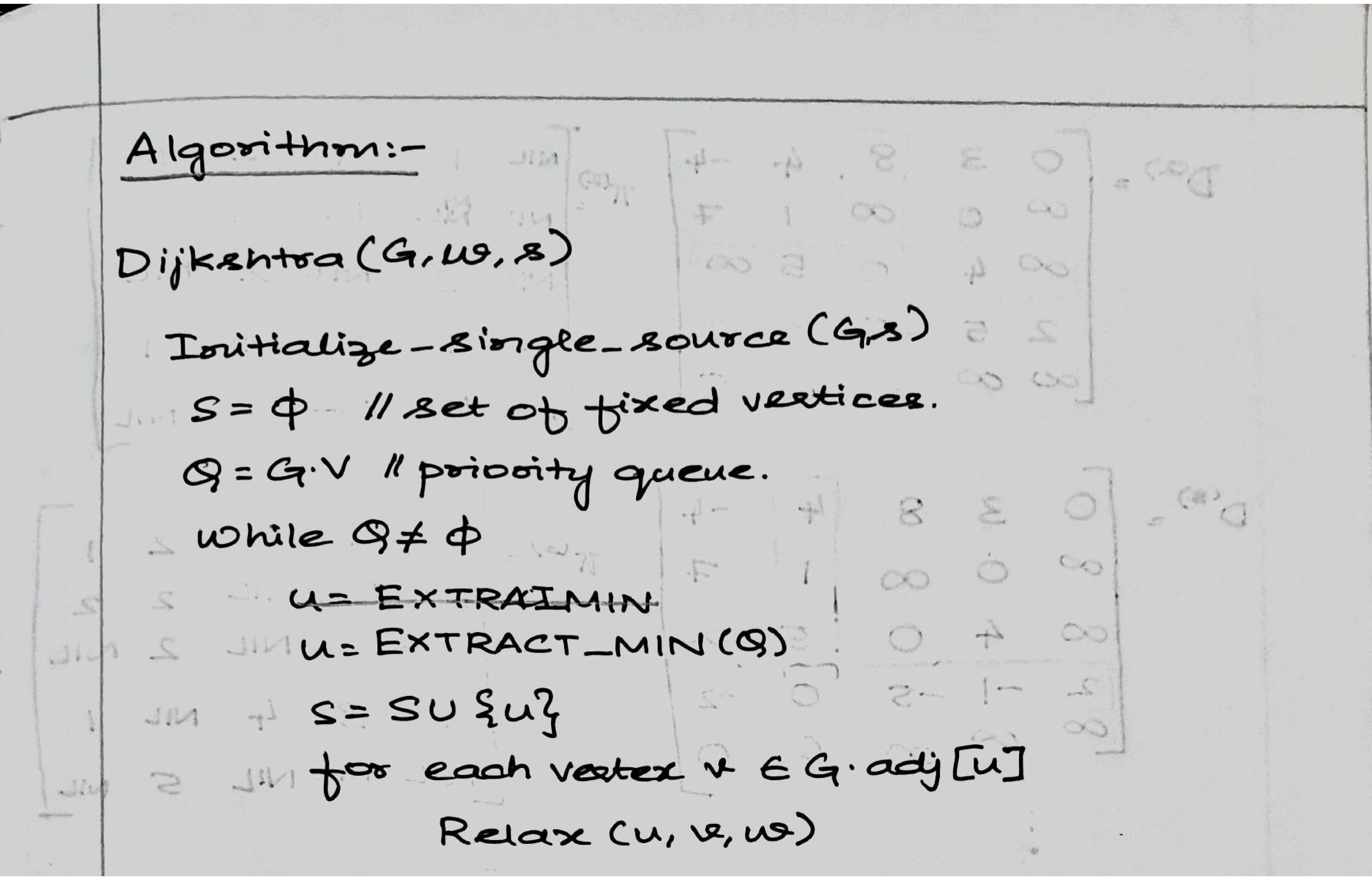
'S' is the source vertex.

Vertex	π
s	NIL
t	~~NIL~~ → ~~s~~ → y
x	~~NIL~~ → ~~y~~ → ~~z~~ → t
y	~~NIL~~ → s
z	~~NIL~~ → y



'a' is the source vertex

Vertex	π
a	NIL
b	~~NIL~~ → a
c	~~NIL~~ → b
d	~~NIL~~ → ~~a~~ → b
e	NIL → d



Algorithm:-

```
Dijkshtra(G, w, s)
  Initialize-single-source(G,s)
   S = φ  // set of fixed vertices.
   Q = G.V  // priority queue.
   While Q ≠ φ
       ~~u= EXTRAIMIN~~
       u= EXTRACT_MIN(Q)
        S = S ∪ {u}
        for each vertex v ∈ G.adj[u]
              Relax(u, v, w)
```