CS 300 ASSIGNMENT-5

Question 1

```
Set each vertex’s distance to infinity;
Set S’s distance to 0;
Find the unknown vertex with smallest distance (finds S)
S.known = true;
For each adjacent vertex to S (A and B):
    If the path vertex is not known (both A and B are not known)
        If the distance of the vertex to S + S’s distance < the vertex’s distance (A and B)
            Set A’s distance to $3 + 0 = 3$, set A’s path to S
            Set B’s distance to $2 + 0 = 2$, set B’s path to S
Find the unknown vertex with smallest distance (finds B)
Set B.known = true;
For each adjacent vertex to B (A, S and D):
    If the path vertex is not known (A and D aren’t known)
        If the distance of the vertex to B + B’s distance < the vertex’s distance (D)
            Set D’s distance to $2 + 1 = 3$, set D’s path to B
Find the unknown vertex with smallest distance (finds A or D since they both have distance = 3,
but I will continue with A since it was interpreted before D)
Set A.known = true;
For each adjacent vertex to A (D and E):
    If the path vertex is not known (both D and E are not known)
        If the distance of the vertex to A + A’s distance < the vertex’s distance (E)
            Set E’s distance to $3 + 2 = 5$, set E’s path to A
Find the unknown vertex with smallest distance (finds D)
Set D.known = true;
For each adjacent vertex to D (E, G and F):
    If the path vertex is not known (E, G and F):
        If the distance of the vertex to D + D’s distance < the vertex’s distance (E, G, and F)
                Set E’s distance to 3 + 1 = 4, set E’s path to D
                Set F’s distance to 3 + 2 = 5, set F’s path to D
                Set G’s distance to 3 + 2 = 5, set G’s path to D
Find the unknown vertex with smallest distance (finds E)
Set E.known = true;
For each adjacent vertex to E (G and F):
    If the path vertex is not known (G and F):
          If the distance of the vertex to E + E’s distance < the vertex’s distance (there is no
          such vertex)
Find the unknown vertex with smallest distance (finds F or G since they both have distance = 5,
but I will continue with F)
Set F.known = true;
For each adjacent vertex to F (C):
    If the path vertex is not known (C):
        If the distance of the vertex to F + F’s distance < the vertex’s distance (C)
          Set C’s distance to 5 + 1 = 6, set C’s path to F
Find the unknown vertex with smallest distance (finds G)
Set G.known = true;
For each adjacent vertex to G (F):
    If the path vertex is not known (there is no such vertex)
Find the unknown vertex with smallest distance (finds C)
Set C.known = true;
For each adjacent vertex to C (B):
    If the path vertex is not known (there is no such vertex)
Find the unknown vertex with smallest distance (there is no such vertex left)
    END THE OPERATIONS
```

Question 2

Set S.known = true;

Find the edge with the smallest distance that has one vertex known and the other unknown (finds (S, B))

Add the edge (S, B) to the tree

Set B.known = true;

Find the edge with the smallest distance that has one vertex known and the other unknown (finds (B, D))

Add the edge (B, D) to the tree

Set D.known = true;

Find the edge with the smallest distance that has one vertex known and the other unknown (finds (D, E))

Add the edge (D, E) to the tree

Set E.known = true;

Find the edge with the smallest distance that has one vertex known and the other unknown (finds (A, D))

Add the edge (A, D) to the tree

Set A.known = true;

Find the edge with the smallest distance that has one vertex known and the other unknown (finds (D, G), (E, F) and (D, F), I will choose (D, G) arbitrarily)

Add the edge (D, G) to the tree

Set G.known = true;

Find the edge with the smallest distance that has one vertex known and the other unknown (finds (D, F), (E, F) and (G, F), I will choose (G, F) arbitrarily)

Add the edge (G, F) to the tree

Set F.known = true;

Find the edge with the smallest distance that has one vertex known and the other unknown (finds (C, F))

Add the edge (C, F) to the tree

Set C.known = true;

Find the edge with the smallest distance that has one vertex known and the other unknown (there is no such edge, all the vertices are known)

END THE OPERATIONS

At the end, the edges consisting Prim’s minimum spanning tree:

(S, B)

(B, D)

(D, E)

(A, D)

(D, G)

(G, F)

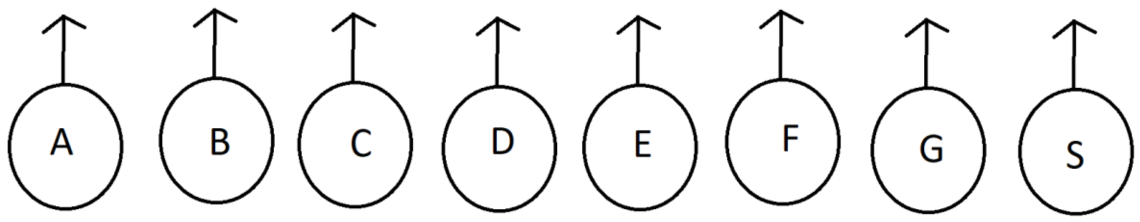
(C, F)

Question 3

Build the heap of edges

Number of edges accepted = 0

The initial vertex forest:



(number of edges accepted < number of vertices – 1) = (0 < 7): True

Find the edge with minimum length from the heap using deleteMin (finds (B, D), (A, D), (D, E) or (C, F), I will choose (B, D) arbitrarily)

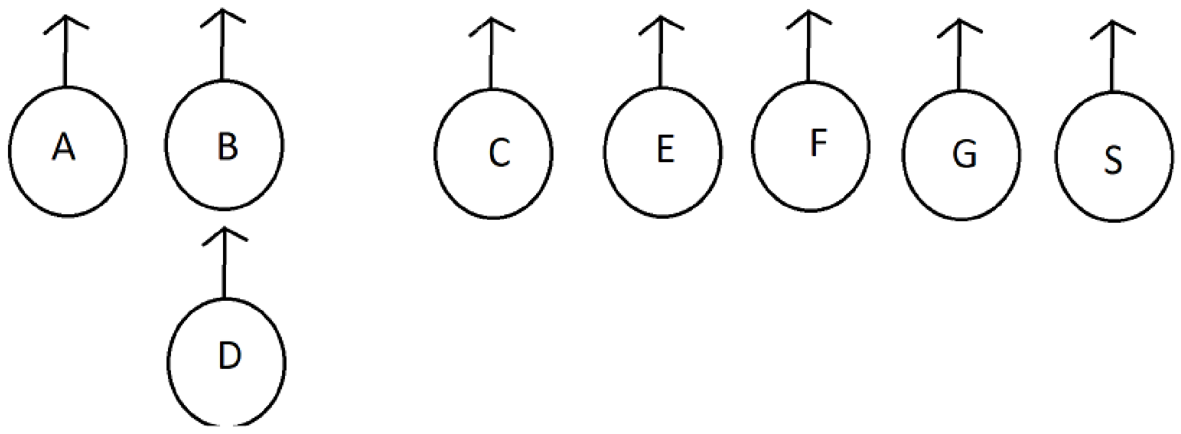
As a result of deleteMin (B, D) is removed from the heap

If the vertices B and D are not already in the same tree: True

Number of edges accepted = 1

Union the trees of B and D

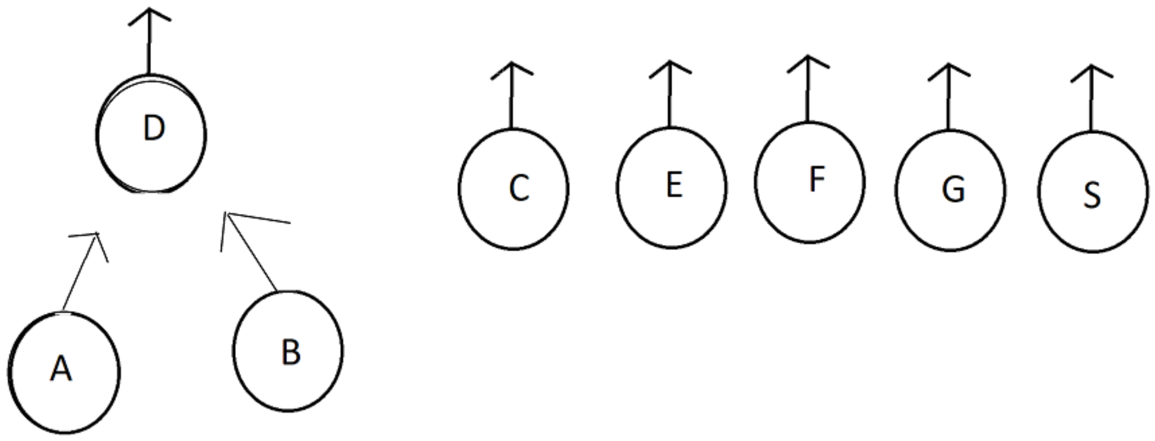
The resulting forest:



(number of edges accepted < number of vertices – 1) = (1 < 7): True

> Find the edge with minimum length from the heap using deleteMin (finds (A, D), (D, E) or (C, F) I will choose (A, D) arbitrarily)
>
> As a result of deleteMin (A, D) is removed from the heap
>
> If the vertices A and D are not already in the same tree: True
>
> > Number of edges accepted = 2
> >
> > Union the trees of A and D

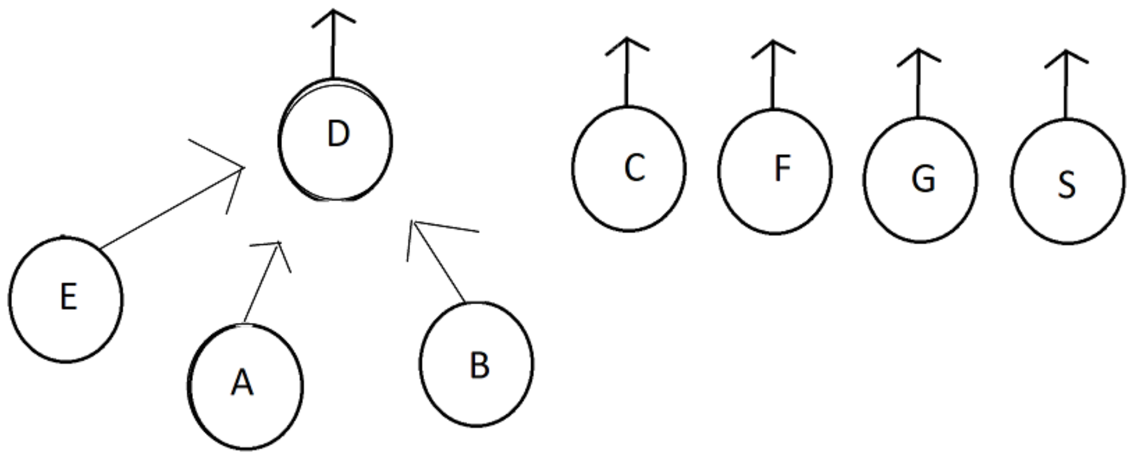
The resulting forest:



(number of edges accepted < number of vertices – 1) = (2 < 7): True

> Find the edge with minimum length from the heap using deleteMin (finds (D, E) or (C, F) I will choose (D, E) arbitrarily)
>
> As a result of deleteMin (D, E) is removed from the heap
>
> If the vertices D and E are not already in the same tree: True
>
> > Number of edges accepted = 3
> >
> > Union the trees of D and E

The resulting forest:



(number of edges accepted < number of vertices – 1) = (3 < 7): True

    Find the edge with minimum length from the heap using deleteMin (finds (C, F))

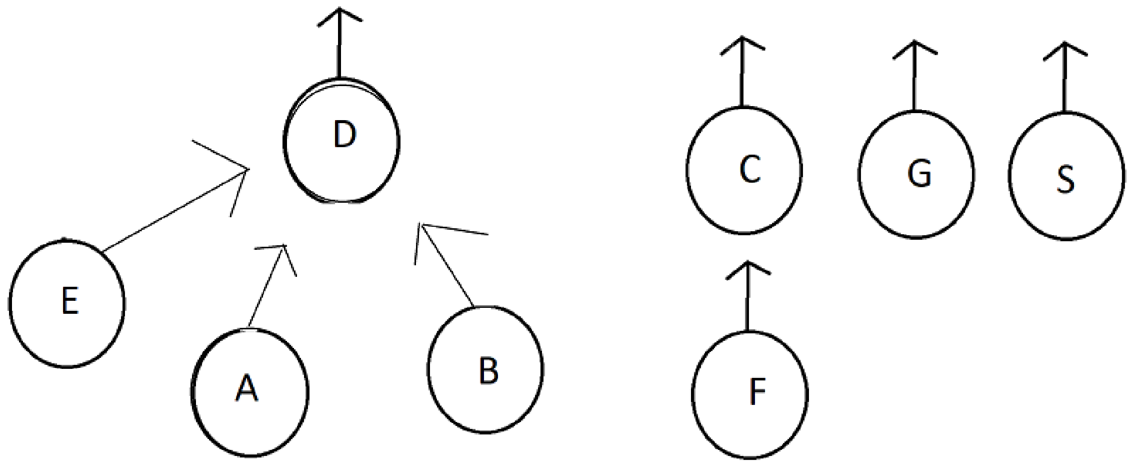
    As a result of deleteMin (C, F) is removed from the heap

    If the vertices C and F are not already in the same tree: True

        Number of edges accepted = 4

        Union the trees of C and F

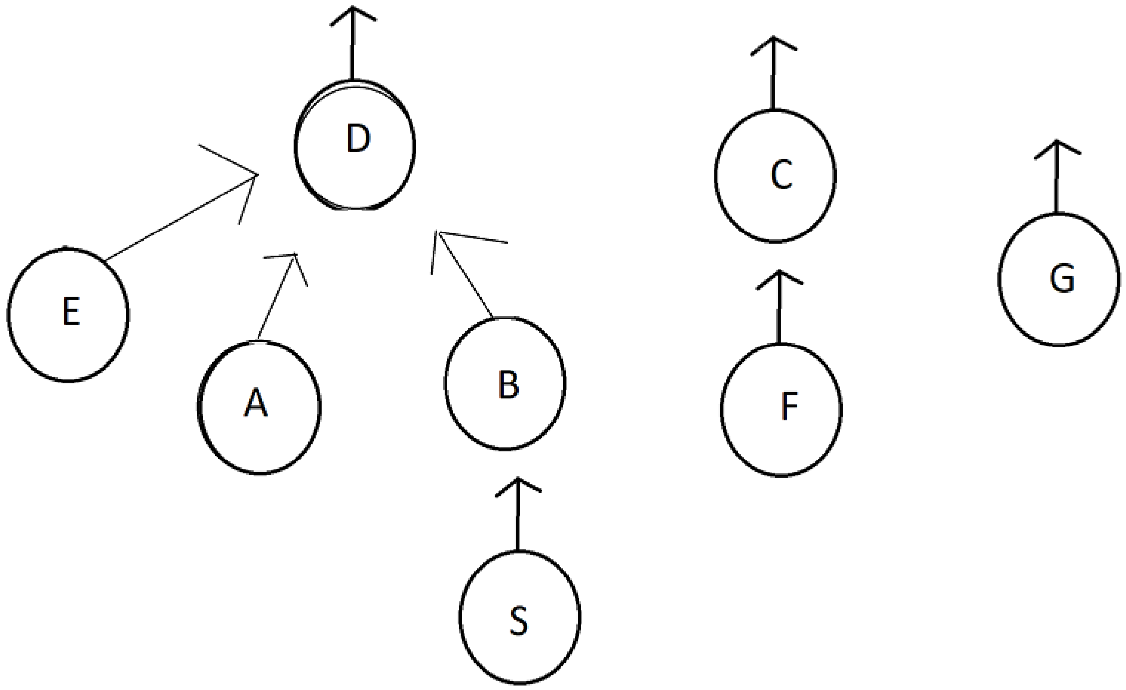
The resulting forest:



(number of edges accepted < number of vertices – 1) = (4 < 7): True

> Find the edge with minimum length from the heap using deleteMin (finds (S, B), (B, A), (A, E), (E, F), (G, F), (D, G) or (D, F), I will choose (S, B) arbitrarily)
>
> As a result of deleteMin (S, B) is removed from the heap
>
> If the vertices S and B are not already in the same tree: True
>
>> Number of edges accepted = 5
>>
>> Union the trees of S and B

The resulting forest:



(number of edges accepted < number of vertices – 1) = (5 < 7): True

> Find the edge with minimum length from the heap using deleteMin (finds (B, A), (A, E), (E, F), (G, F), (D, G) or (D, F), I will choose (B, A) arbitrarily)
>
> As a result of deleteMin (B, A) is removed from the heap
>
> If the vertices B and A are not already in the same tree: False
>
>> DO NOT EXECUTE INNER OPERATIONS

(number of edges accepted < number of vertices – 1) = (5 < 7): True

> Find the edge with minimum length from the heap using deleteMin (finds (A, E), (E, F), (G, F), (D, G) or (D, F), I will choose (A, E) arbitrarily)

As a result of deleteMin (A, E) is removed from the heap

If the vertices A and E are not already in the same tree: False

DO NOT EXECUTE INNER OPERATIONS

(number of edges accepted < number of vertices – 1) = (5 < 7): True

Find the edge with minimum length from the heap using deleteMin (finds (E, F), (G, F), (D, G) or (D, F), I will choose (E, F) arbitrarily)

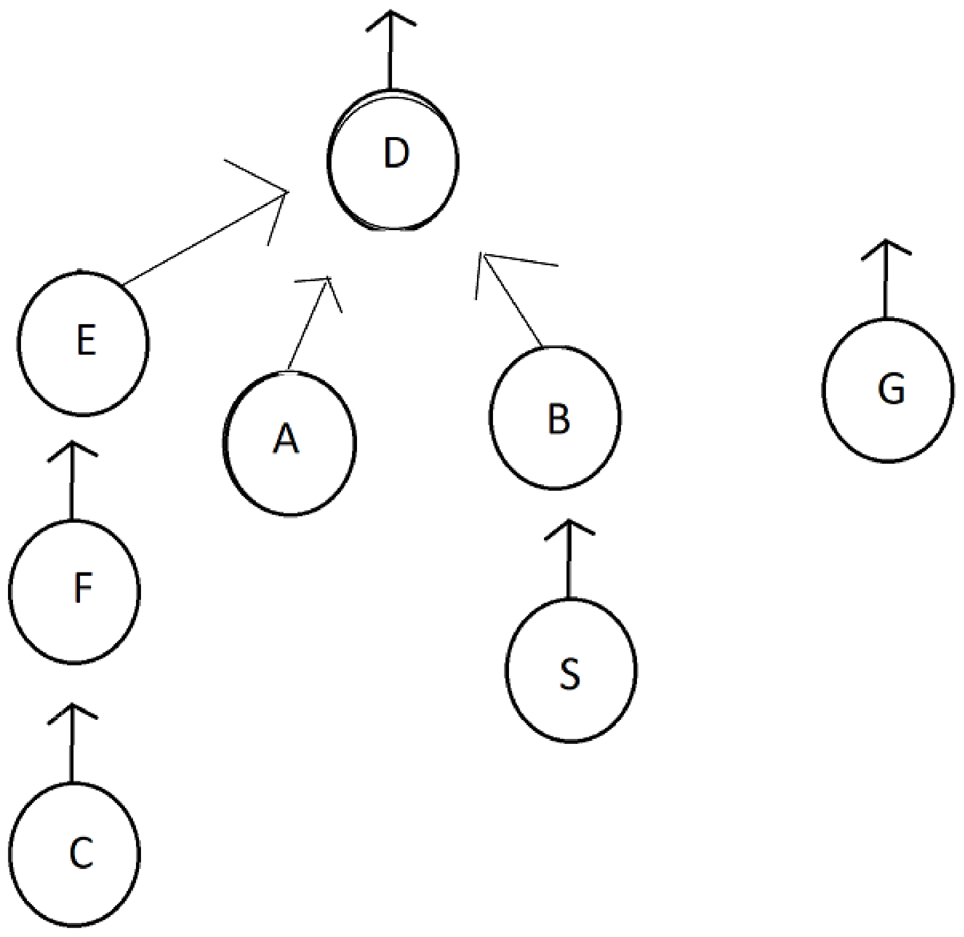
As a result of deleteMin (E, F) is removed from the heap

If the vertices E and F are not already in the same tree: True

Number of edges accepted = 6

Union the trees of E and F

The resulting forest:



(number of edges accepted < number of vertices – 1) = (6 < 7): True

Find the edge with minimum length from the heap using deleteMin (finds (G, F), (D, G) or (D, F), I will choose (G, F) arbitrarily)

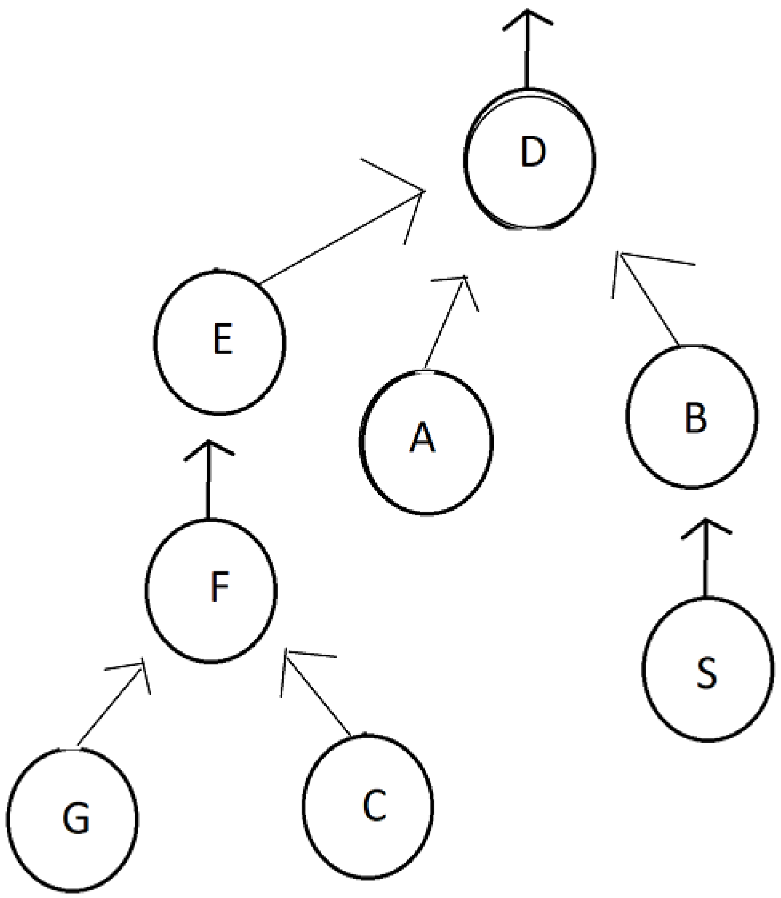
As a result of deleteMin (G, F) is removed from the heap

If the vertices G and F are not already in the same tree: True

Number of edges accepted = 7

Union the trees of G and F

The tree we ended up with at the end:



(number of edges accepted < number of vertices – 1) = (7 < 7): False

END OF THE OPERATIONS

Question 4

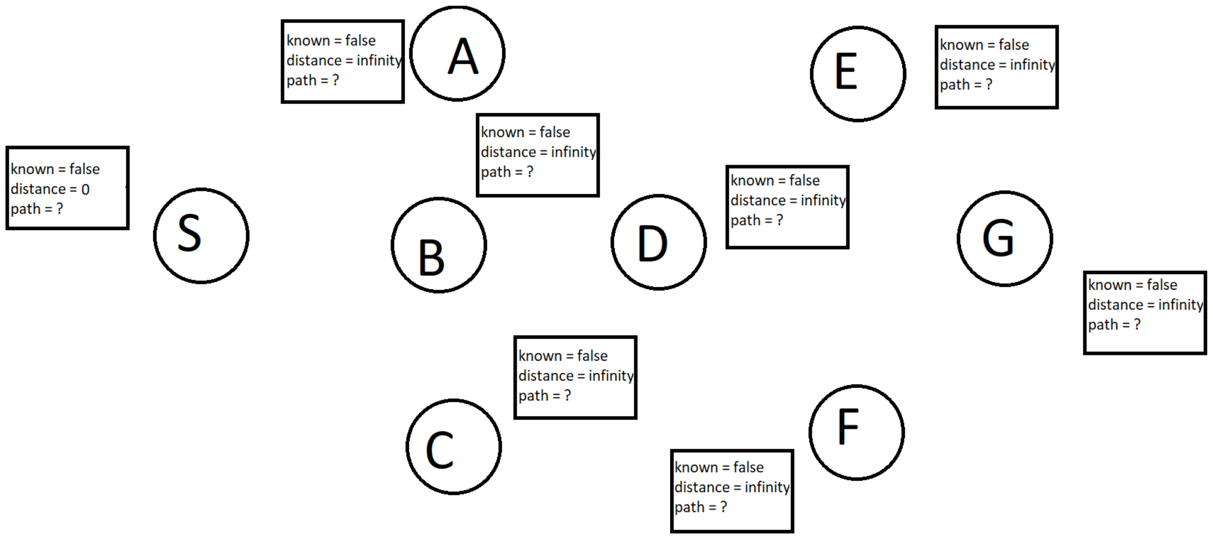
Breadth-First Traversal Using Queue

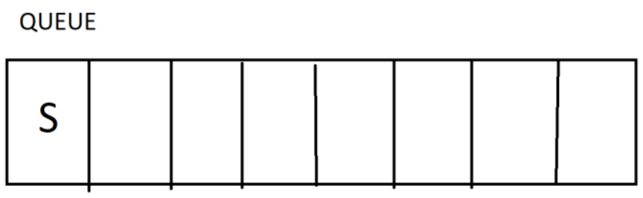
Build a queue with number of vertices length

Set each vertex's distance to infinity

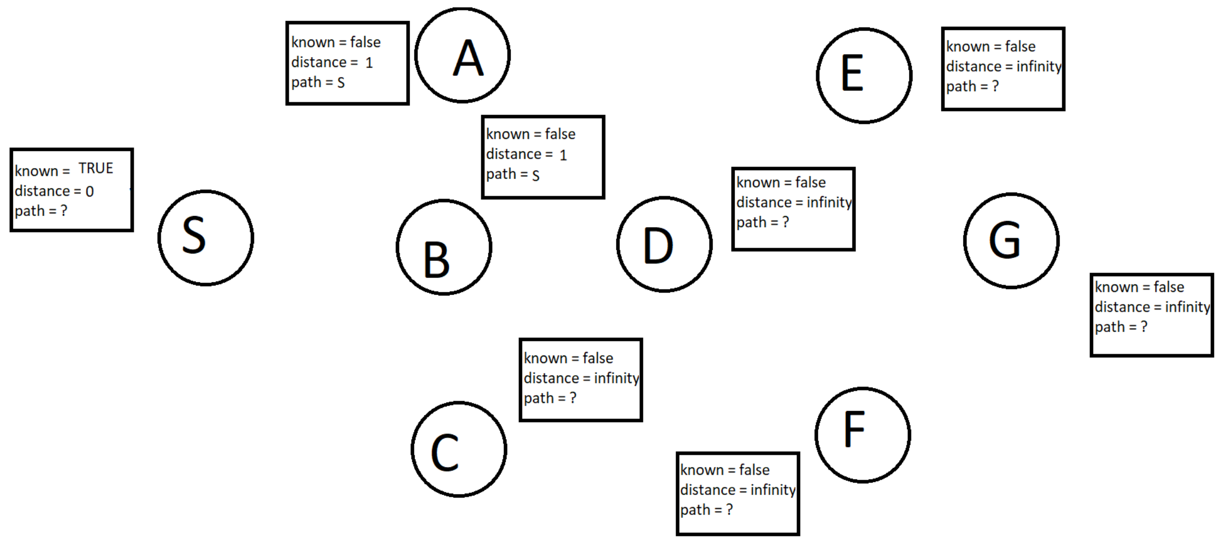
Set S's distance to 0

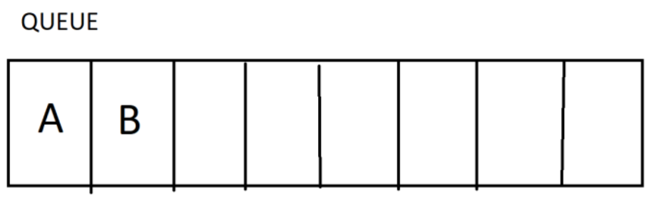
Enqueue S to the queue



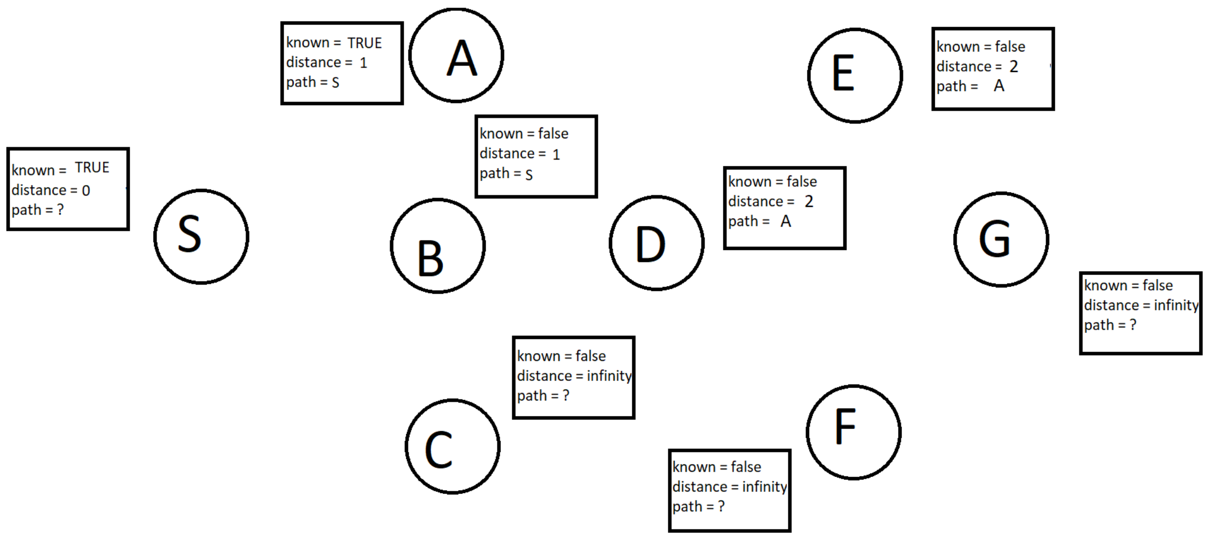


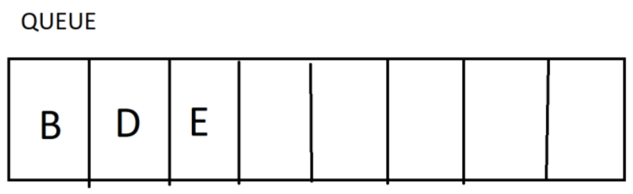
```
! (Queue.isEmpty): True
    Queue.Dequeue (we get S)
    S.known = true
    For each adjacent vertex to S (A and B)
        If the distance of the vertex == infinity (A and B)
            Set A's distance to S's distance + 1 = 2, set A's path to S
            Set B's distance to S's distance + 1 = 2, set B's path to S
            Enqueue A
            Enqueue B
```



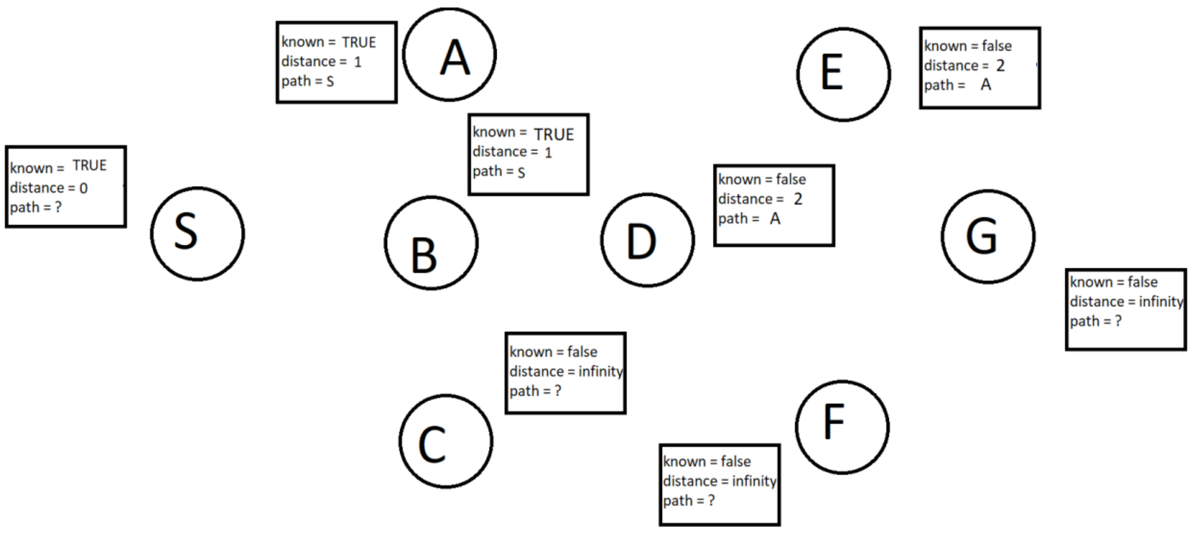


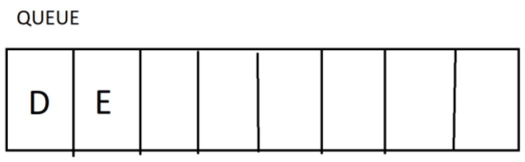
```
! (Queue.isEmpty): True
    Queue.Dequeue (we get A)
    A.known = true
    For each adjacent vertex to A (D and E)
        If the distance of the vertex == infinity (D and E)
            Set D’s distance to A’s distance + 1 = 2, set D’s path to A
            Set E’s distance to A’s distance + 1 = 2, set E’s path to A
            Enqueue D
            Enqueue E
```



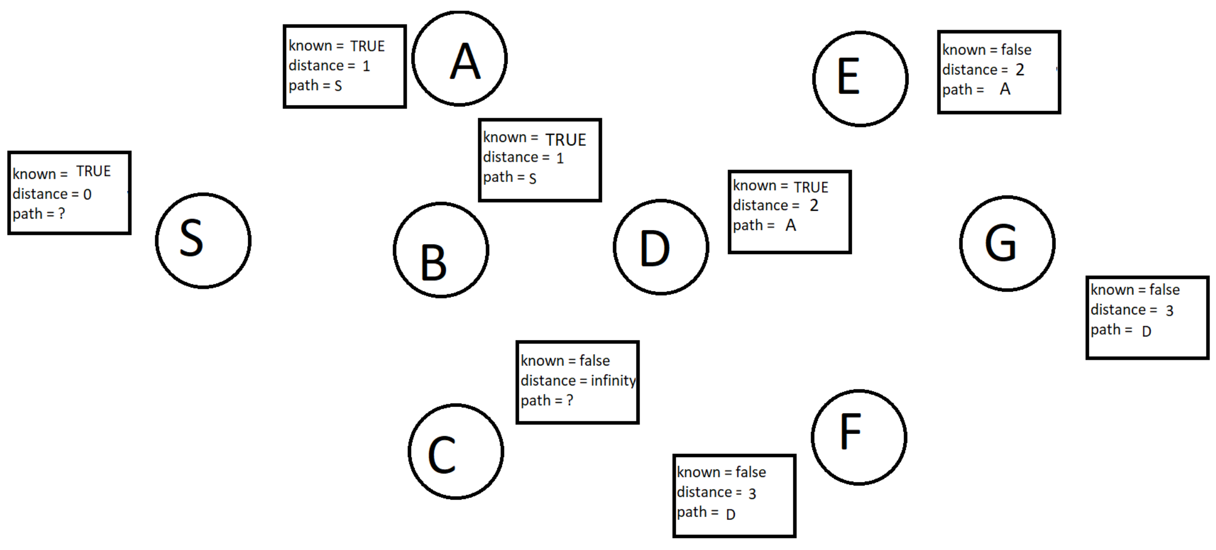


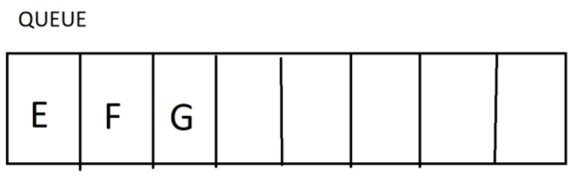
```
! (Queue.isEmpty): True
    Queue.Dequeue (we get B)
    B.known = true
    For each adjacent vertex to B (S, A and D)
        If the distance of the vertex == infinity (there is no such vertex)
            DO NOT EXECUTE INNER OPERATIONS
```



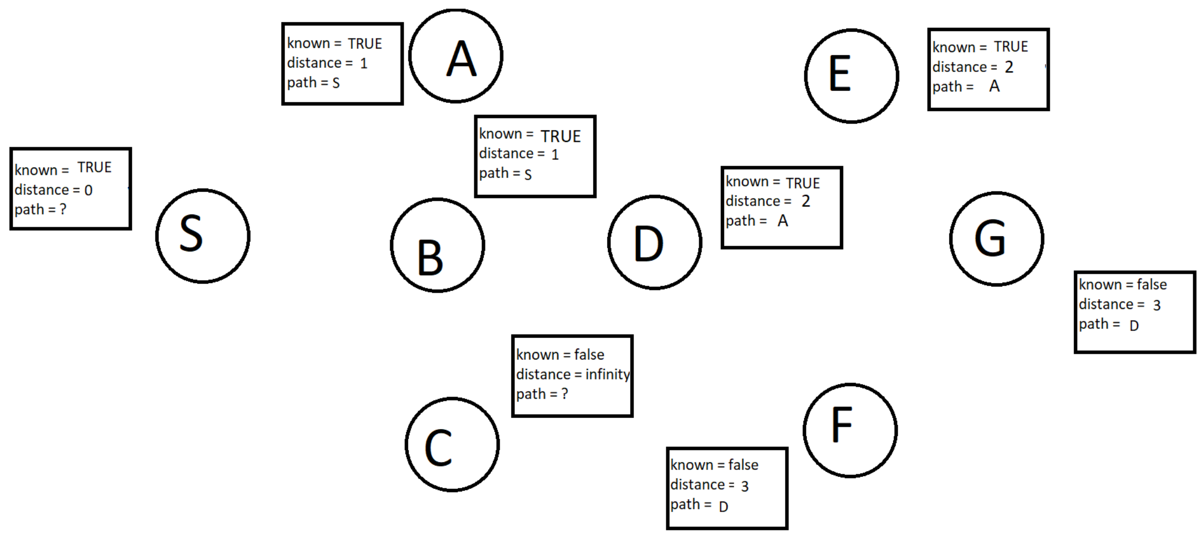


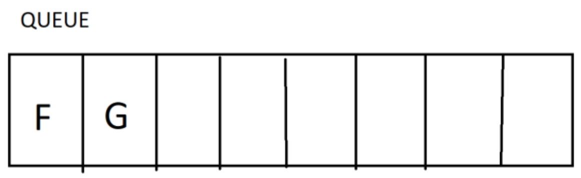
```
! (Queue.isEmpty): True
        Queue.Dequeue (we get D)
        D.known = true
        For each adjacent vertex to D (E, F and G)
                If the distance of the vertex == infinity (F and G)
                        Set F's distance to D's distance + 1 = 3, set F's path to D
                        Set G's distance to D's distance + 1 = 3, set G's path to D
                        Enqueue F
                        Enqueue G
```



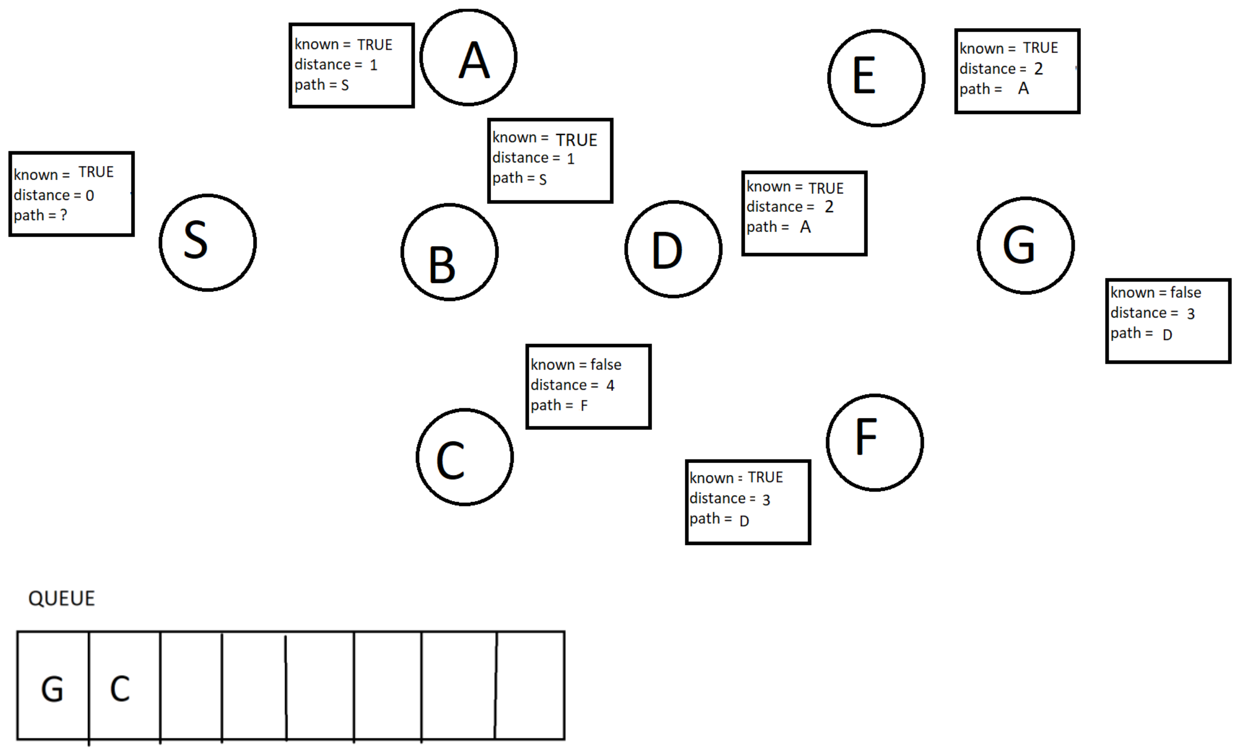


```
! (Queue.isEmpty): True
    Queue.Dequeue (we get E)
    E.known = true
    For each adjacent vertex to E (F and G)
        If the distance of the vertex == infinity (there is no such vertex)
            DO NOT EXECUTE INNER OPERATIONS
```





```
! (Queue.isEmpty): True
    Queue.Dequeue (we get F)
    F.known = true
    For each adjacent vertex to F (C)
        If the distance of the vertex == infinity (C)
            Set C’s distance to F’s distance + 1 = 4, set C’s path to F
            Enqueue C
```



known = TRUE
distance = 1
path = S

known = TRUE
distance = 2
path = A

known = TRUE
distance = 0
path = ?

known = TRUE
distance = 1
path = S

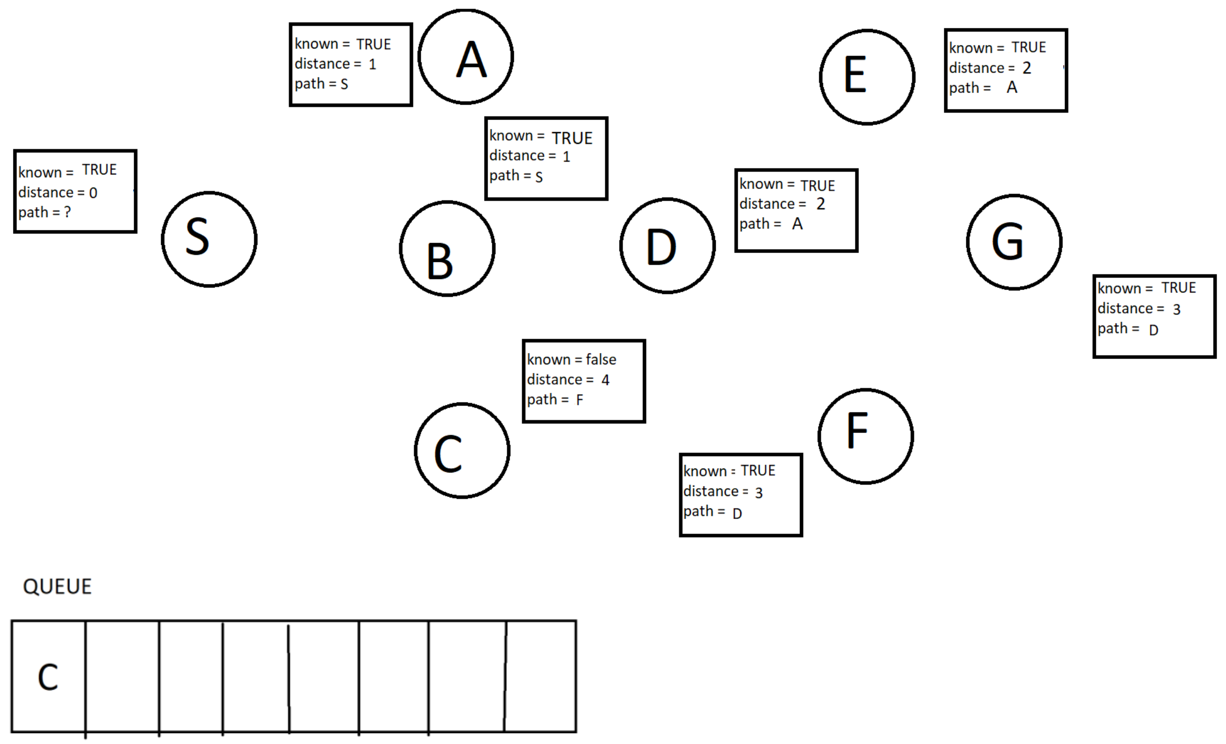
known = TRUE
distance = 2
path = A

known = false
distance = 3
path = D

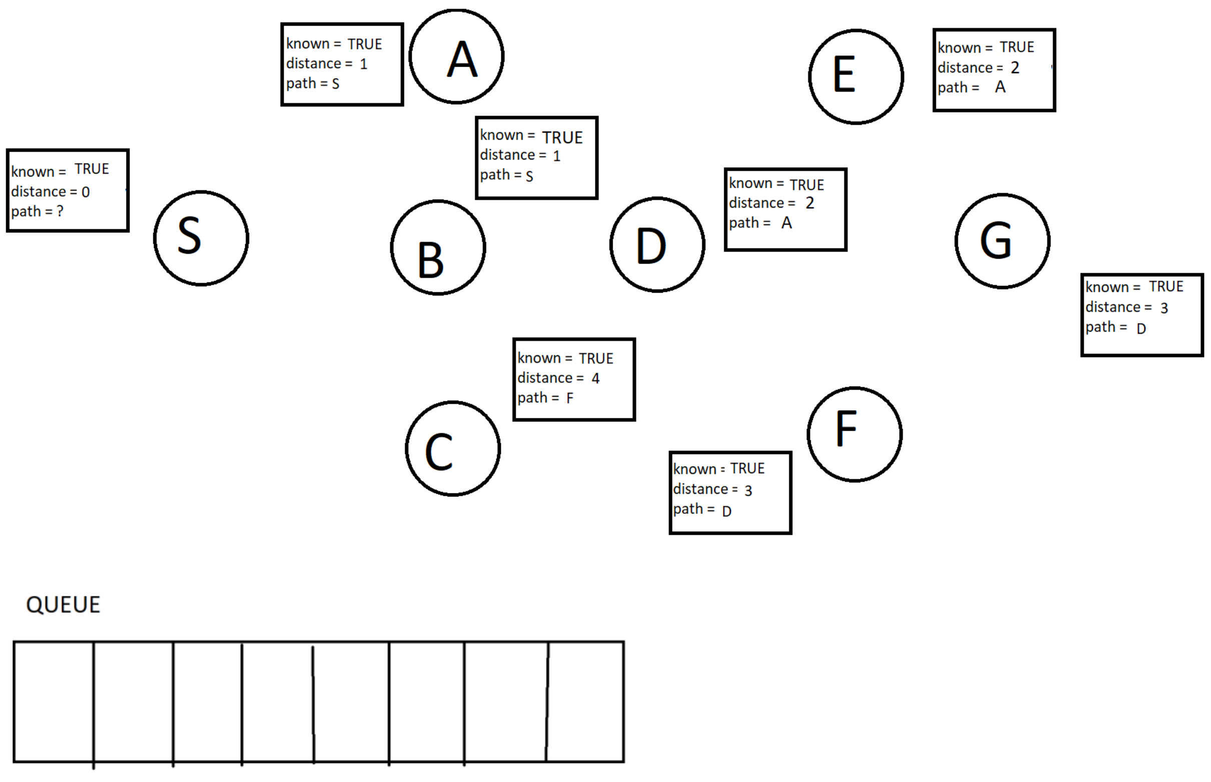
known = false
distance = 4
path = F

known = TRUE
distance = 3
path = D

```
! (Queue.isEmpty): True
        Queue.Dequeue (we get G)
        G.known = true
        For each adjacent vertex to G (F)
                If the distance of the vertex == infinity (there is no such vertex)
                        DO NOT EXECUTE INNER OPERATIONS
```



```
! (Queue.isEmpty): True
    Queue.Dequeue (we get C)
    C.known = true
    For each adjacent vertex to C (B)
        If the distance of the vertex == infinity (there is no such vertex)
            DO NOT EXECUTE INNER OPERATIONS
```



! (Queue.isEmpty): False

END OF THE OPERATIONS

Question 5

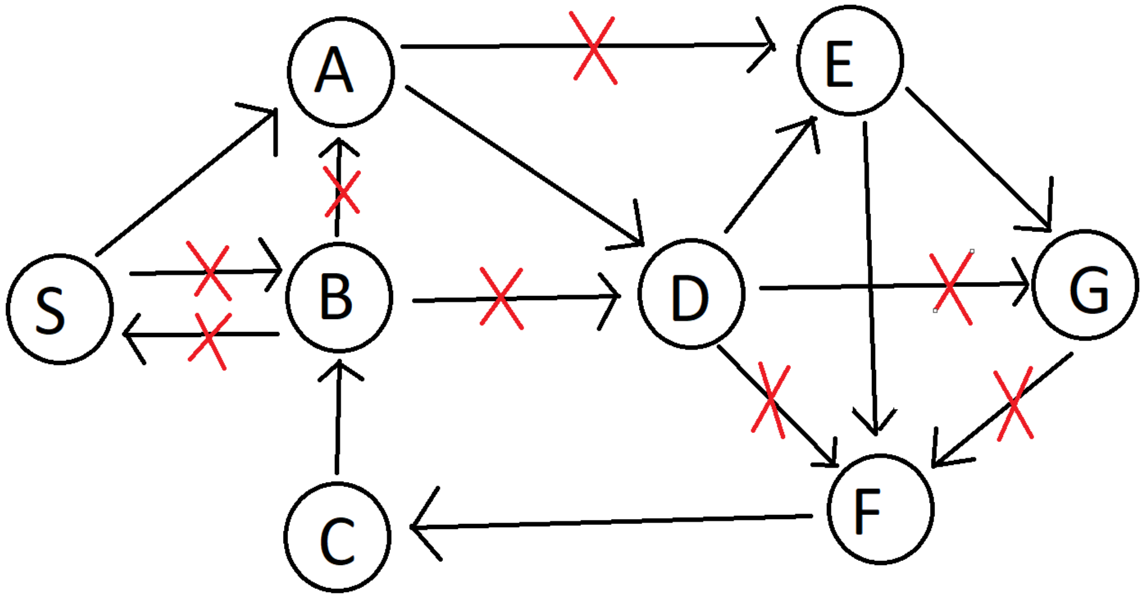
a)

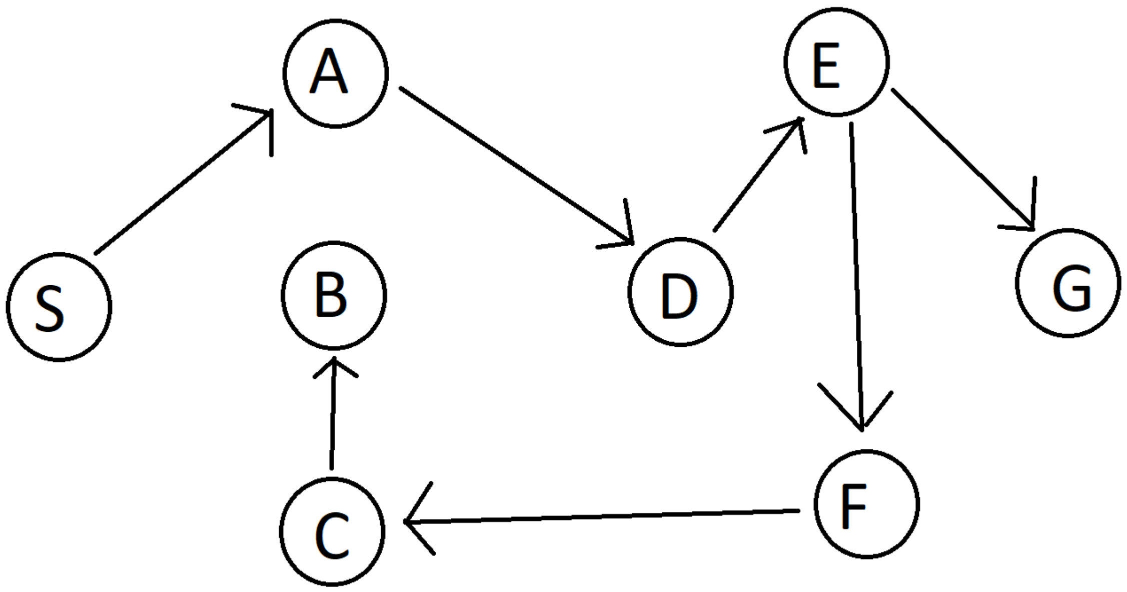
```
Dfc(S)
{
  S.visited = true
  For each adjacent vertex to S (A and B, arbitrarily in the order of A then B)
 {
        If A is not visited yet (True)
        Dfc(A)
        {
          A.visited = true
          For each adjacent vertex to A (D and E, arbitrarily in the order of D then E)
         {
                If D is not visited yet (True)
                Dfc(D)
                {
                  D.visited = true
                  For each adjacent vertex to D (E, F and G, arbitrarily in the order of E then F
                then G)
                  {
                        If E is not visited yet (True)
                        Dfc(E)
                        {
                          E.visited = true
                          For each adjacent vertex to E (F and G, arbitrarily in the order of F then
                        G)
                          {
                                If F is not visited yet (True)
                                Dfc(F)
```

```
{
   F.visited = true
   For each adjacent vertex to F (C)
  {
         If C is not visited yet (True)
         Dfc(C )
         {
            C.visited = true
            For each adjacent vertex to C (B)
            {
              If B is not visited yet (True)
                  Dfc(B)
                  {
                    B.visited = true
                    For each adjacent vertex to B (S, A and D,
                    arbitrarily in the order of S then A then D)
                    {
                           If S is not visited yet (False, S is already
                           visited)
                           If A is not visited yet (False, A is already
                           visited)
                           If D is not visited yet (False, D is already
                           visited)
                    }
                    END OF DFC(B) RETURN BACK TO DFC(C)
                  }
              }
              END OF DFC(C) RETURN BACK TO DFC(F)
            }
   }
```

```
                        END OF DFC(F) RETURN BACK TO DFC(E)
                      }
                      If G is not visited yet (True)
                      DFC(G)
                      {
                        G.visited = true
                        For each adjacent vertex to G(F)
                        {
                              If F is not visited yet (False, F is already visited)
                        }
                        END OF DFC(G) RETURN BACK TO DFC(E)
                      }
                  }
                  END OF DFC(E) RETURN BACK TO DFC(D)
                }
                If F is not visited yet (False, F is already visited)
                If G is not visited yet (False, G is already visited)
            }
            END OF DFC(D) RETURN BACK TO DFC(A)
          }
          If E is not visited yet (False, E is already visited)
      }
      END OF DFC(A) RETURN BACK TO DFC(S)
    }
    If B is not visited yet (False, B is already visited)
  }
  END OF DFC(S), END OF THE OPERATIONS
}
```

THE DEPTH FIRST TREE AT THE END:





b)

We set post-order numbers to nodes just before we end the operations for its Dfc() function and return back to the previous Dfc() function for another node. The order of the Dfc() functions to be ended is:

B-C-F-G-E-D-A-S

So, starting from post-order number = 1, the post-order value for each node is:

B.postorder = 1

C.postorder = 2

F.postorder = 3

G.postorder = 4

E.postorder = 5

D.postorder = 6

A.postorder = 7

S.postorder = 8

c)

We set pre-order numbers to nodes just after we mark them as visited. The order of the nodes to be visited is:

S-A-D-E-F-C-B-G

So, starting from pre-order number = 1, the pre-order value for each node is:

S.preorder = 1

A.preorder = 2

D.preorder = 3

E.preorder = 4

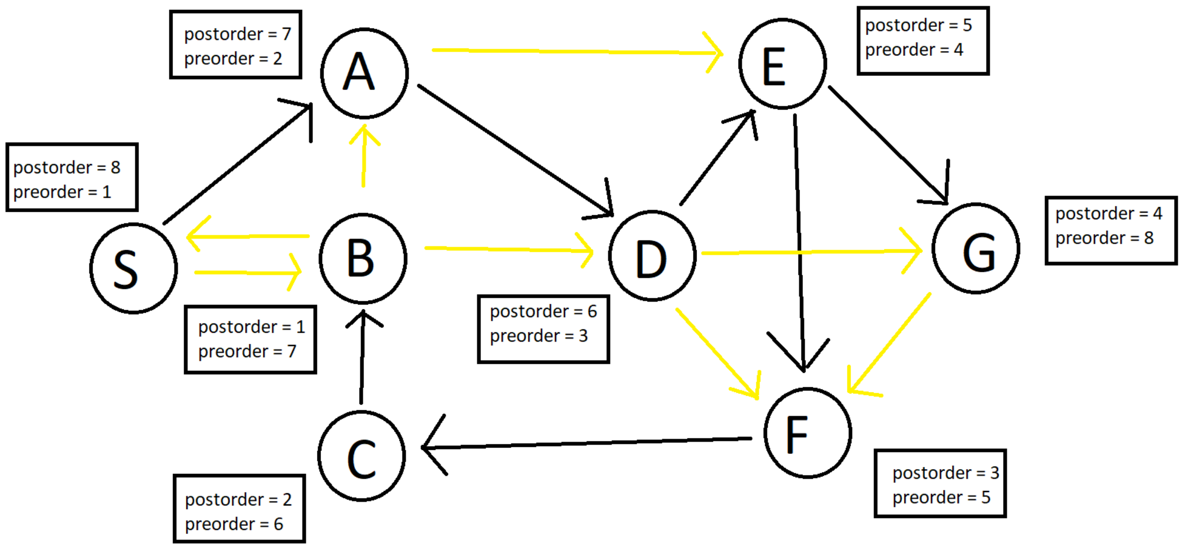
F.preorder = 5

C.preorder = 6

B.preorder = 7

G.preorder = 8

THE DEPTH FIRST TREE WITH POST-ORDER AND PRE-ORDER NUMBERS:



d)

LIST OF THE ARCS:

Tree Arcs:

S->A	A->D	D->E	E->F	E->G	F->C	C->B

Backward Arcs:

B->S	B->D	B->A

Forward Arcs:

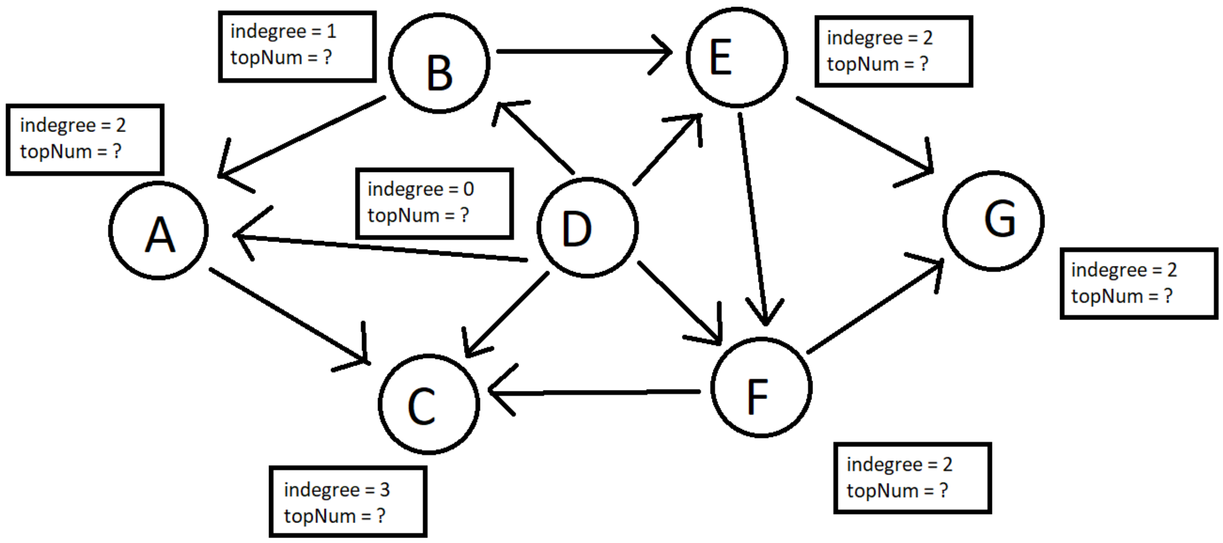
S->A	S->B	A->D	A->E	D->E	E->F	E->G
D->F	F->C	C->B	D->G			

Cross Arcs:

G->F

Question 6

Initial topological sort graph:



Counter = 0

Counter < number of vertices = (0 < 7): True

Find a vertex with an indegree of 0 and topNum is not assigned yet (finds D)

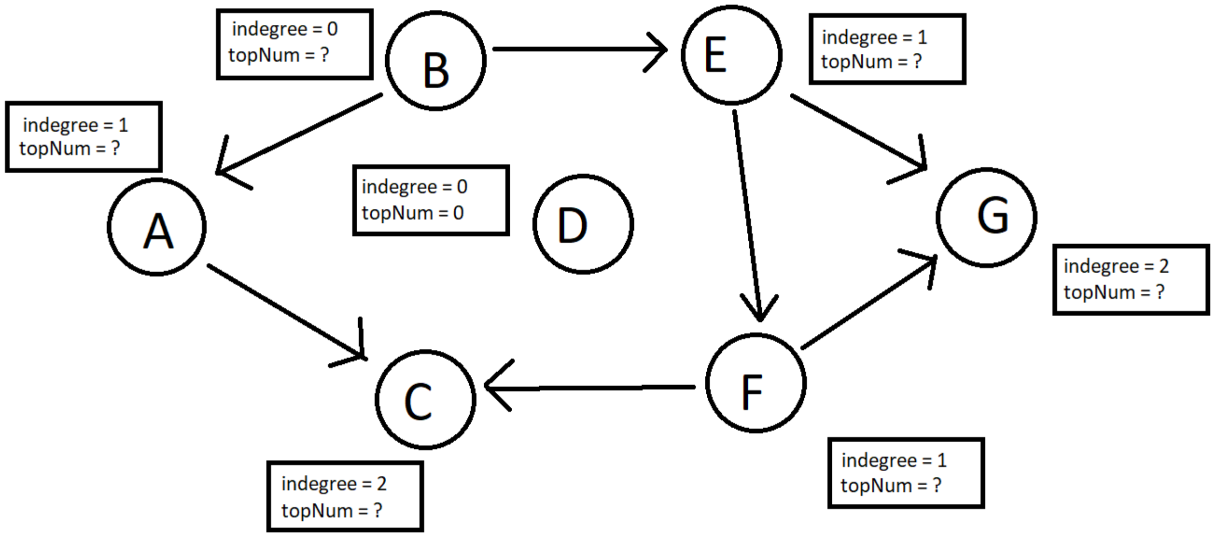
D.topNum = counter (0)

For each adjacent vertex to D (A, B, C, E and F)

Decrement indegrees of A, B, C, E and F by 1

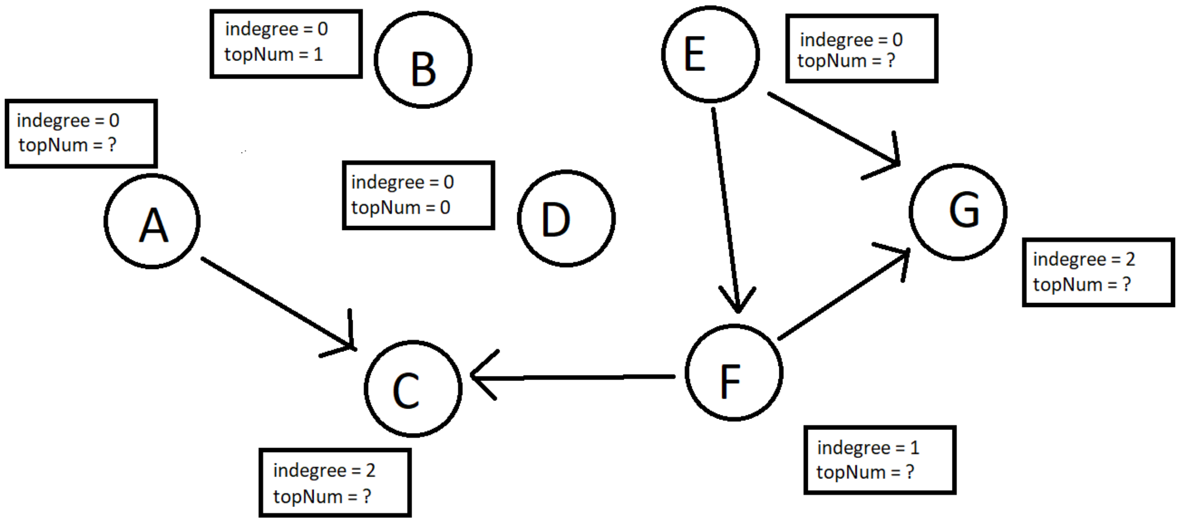
Remove the connections of D from the graph

Topological sort graph after first loop:



```
Counter = 1
Counter < number of vertices = (1 < 7): True
Find a vertex with an indegree of 0 and topNum is not assigned yet (finds B)
B.topNum = counter (1)
For each adjacent vertex to B (A and E)
        Decrement indegrees of A and E by 1
Remove the connections of B from the graph
```

Topological sort graph after second loop:



Counter = 2

Counter < number of vertices = (2 < 7): True

Find a vertex with an indegree of 0 and topNum is not assigned yet (finds A and E, arbitrarily chosen A)

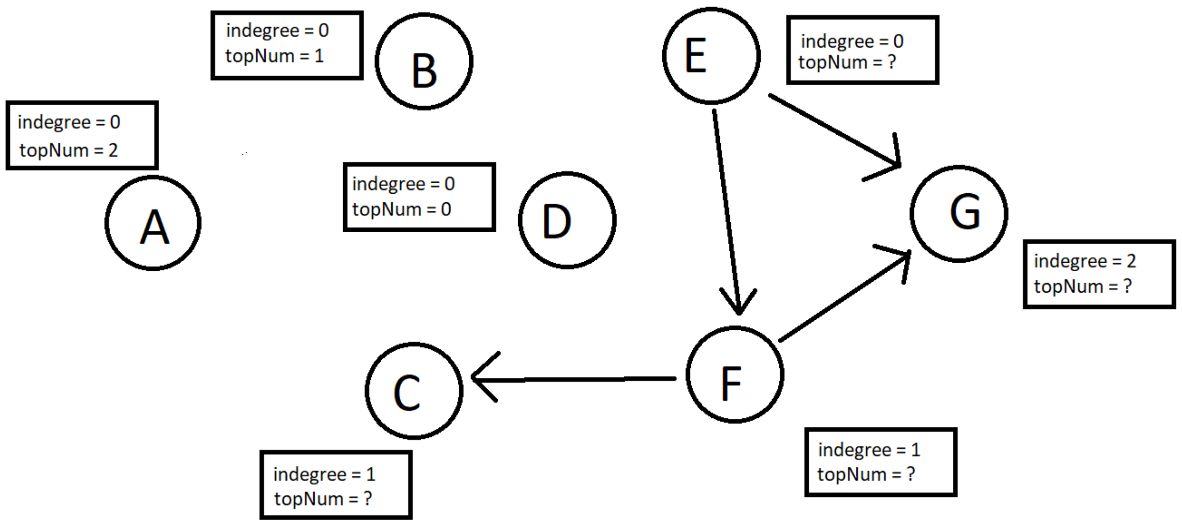
A.topNum = counter (2)

For each adjacent vertex to A (C)

    Decrement indegrees of C by 1

Remove the connections of A from the graph

Topological sort graph after third loop:



Counter = 3

Counter < number of vertices = (3 < 7): True

Find a vertex with an indegree of 0 and topNum is not assigned yet (finds E)

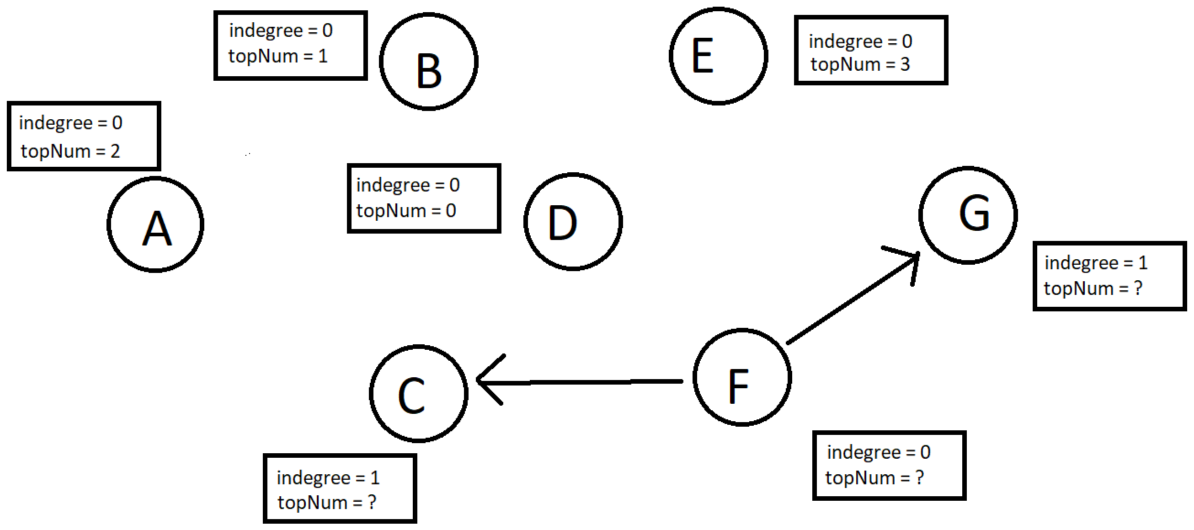
E.topNum = counter (3)

For each adjacent vertex to E (F and G)

    Decrement indegrees of F and G by 1

Remove the connections of E from the graph

Topological sort graph after fourth loop:



Counter = 4

Counter < number of vertices = (4 < 7): True

Find a vertex with an indegree of 0 and topNum is not assigned yet (finds F)

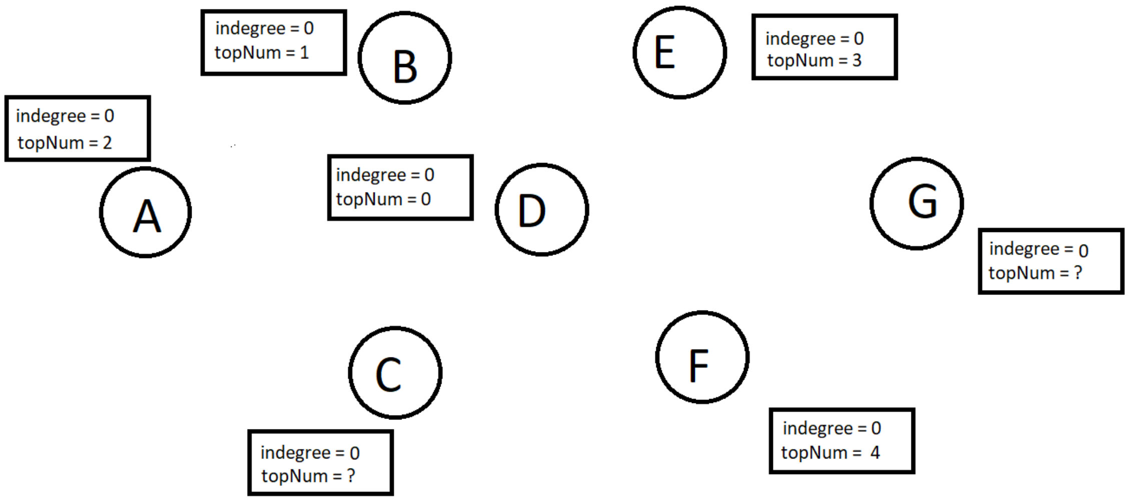
F.topNum = counter (4)

For each adjacent vertex to F (C and G)

    Decrement indegrees of C and G by 1

Remove the connections of F from the graph

Topological sort graph after fifth loop:



Counter = 5

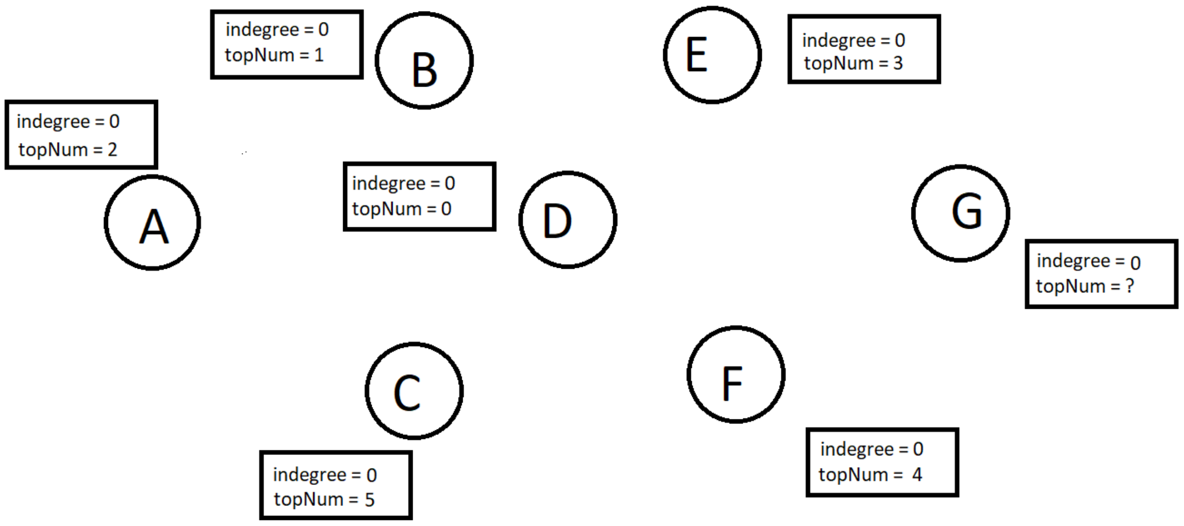
Counter < number of vertices = (5 < 7): True

Find a vertex with an indegree of 0 and topNum is not assigned yet (finds C and G, arbitrarily chosen C)

C.topNum = counter (5)

For each adjacent vertex to C (there is no such vertex)

Topological sort graph after sixth loop:



Counter = 6

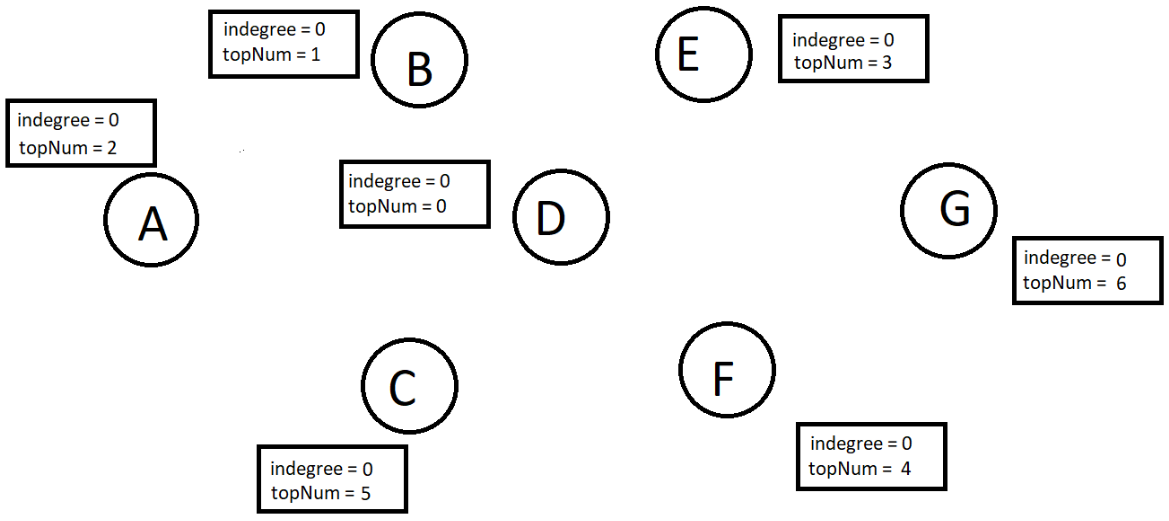
Counter < number of vertices = (6 < 7): True

Find a vertex with an indegree of 0 and topNum is not assigned yet (finds G)

G.topNum = counter (6)

For each adjacent vertex to G (there is no such vertex)

Topological sort graph after seventh loop:



Counter = 7

Counter < number of vertices = (7 < 7): False

END THE OPERATIONS