SABANCI UNIVERSITY

CS300 Data Structures

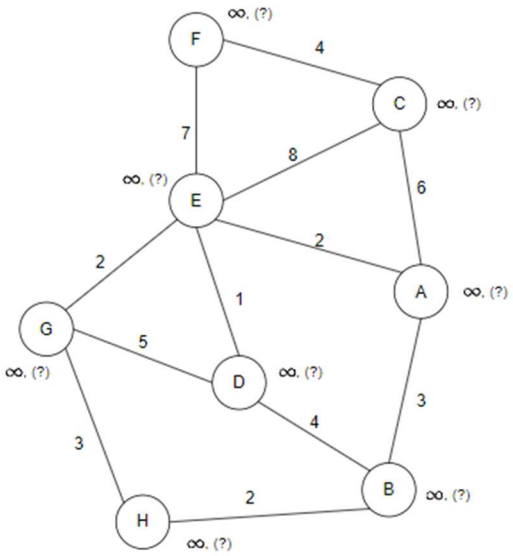
Homework – 4

Pelinsu Sarac - 28820

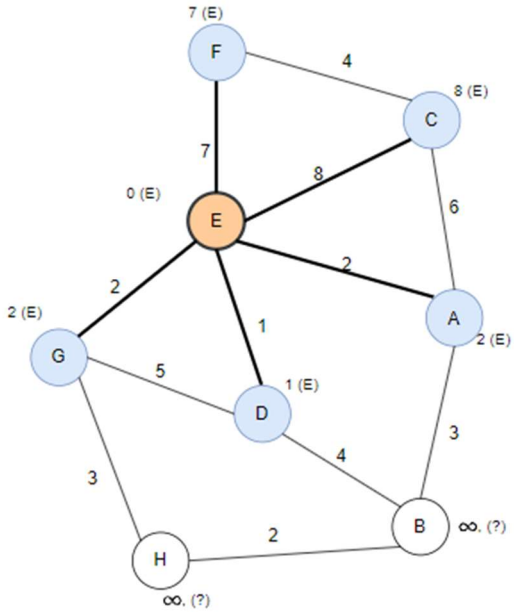
Fall 2022/23

Question 1:

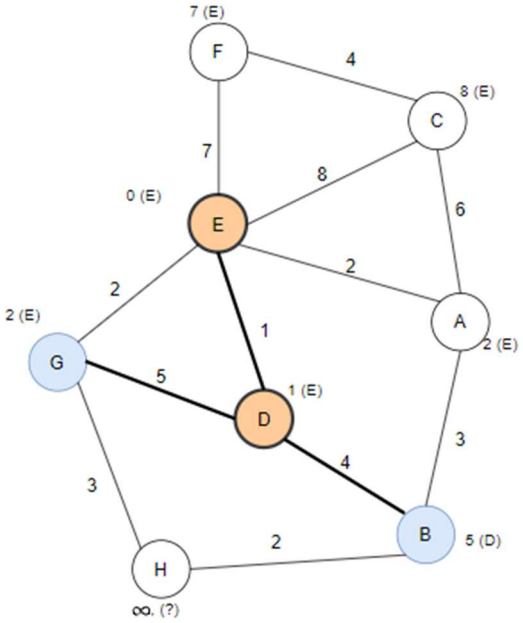
Before starting, all nodes are initialized to have infinite distance. Legend for the graphs below: bold circle and pink color → visited, blue circle → vertex to be updated.



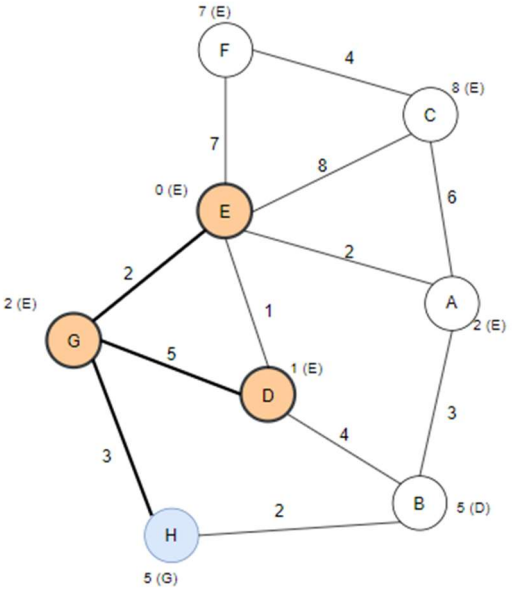
Then, starting with writing 0 and E to vertex E as it has been given as the starting vertex and distance from itself is 0. By doing this, minimum distance vertex becomes E and we visit the vertex E, thus we can also update the adjacent vertices. Here, all adjacent nodes are updated in terms of distance from E with edge weights (w + 0 = w) as all edges from E has lower weight than inifinty. Also, path information of vertices are updated to E as current path comes to them from E.



On the second step, we must choose the vertex with minimum distance value among unvisited vertices, so we continue with visiting vertex D. Vertex D has 2 unvisited adjacent vertices. Vertex G's distance is smaller than the path coming from D to it (2 < 5 + 1) so we don't update it. Vertex B has infinite distance, so we are updating distance with 4 + 1 = 5 (distance from E to D + weight of edge from D to B) and path as D.

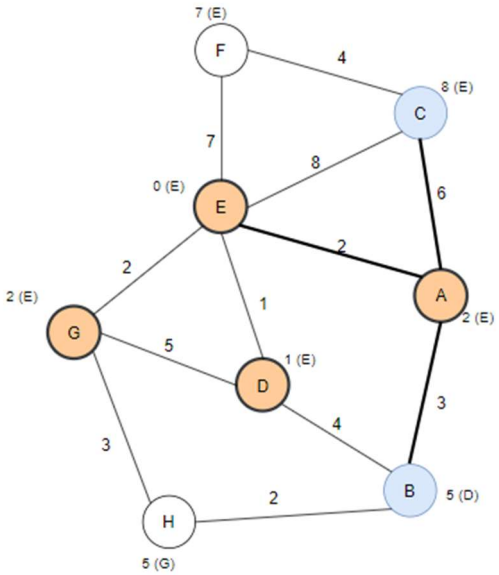


As third step, we must again choose minimum distance vertex but there are two with same distance (G and A), so we arbitrarily choose vertex G among them. Vertex G has only one unvisited adjacent vertex, H, and because it has infinite distance we update distance as 2 + 3 = 5 and path as G.

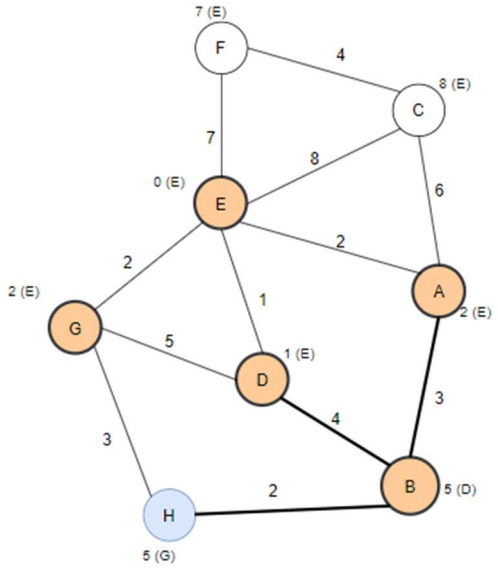


For fourth step, we continue with choosing minimum distance vertex and it's vertex A this time. Vertex A has two unvisited adjacent vertices. However, none of them needs an update as:

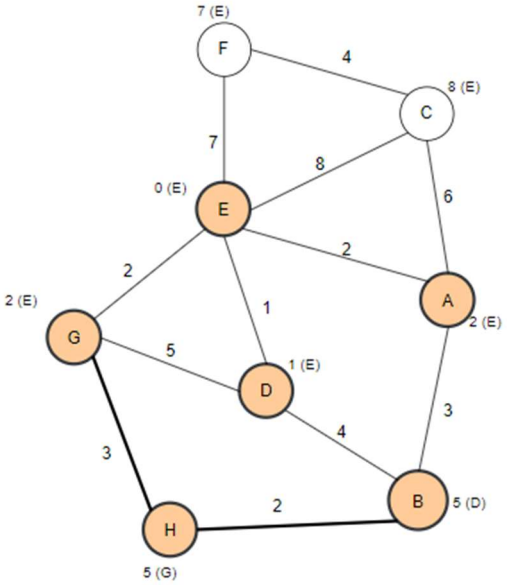
- 6 + 2 = 8 for C
- 2 + 3 = 5 for B, meaning that paths from A are not any shorter than current distance.



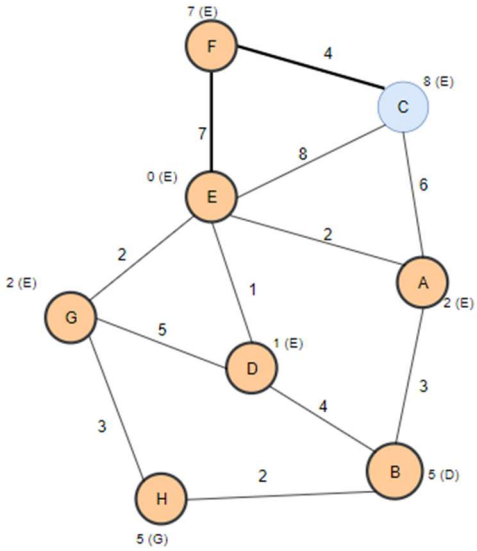
Continuing with next minimum distance vertex among unvisited ones, we have two with 5, H and B. We can arbitrarily choose B. B only has one unvisited adjacent vertex but path distance it can provide is not smaller than current distance of H so we don't update anything.



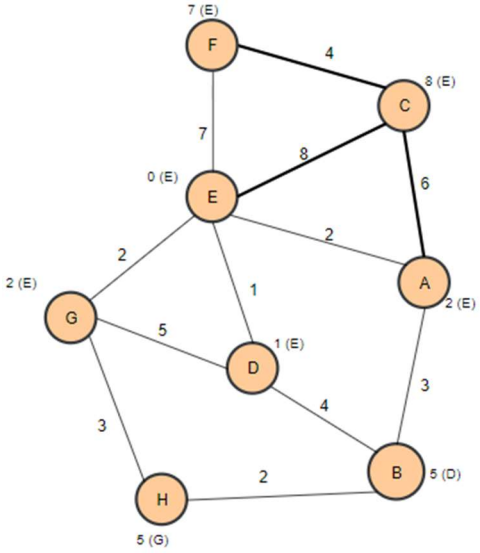
Then we can continue with vertex H as it has the minimum distance. Vertex H has no unvisited adjacent vertex so we don't do any update.



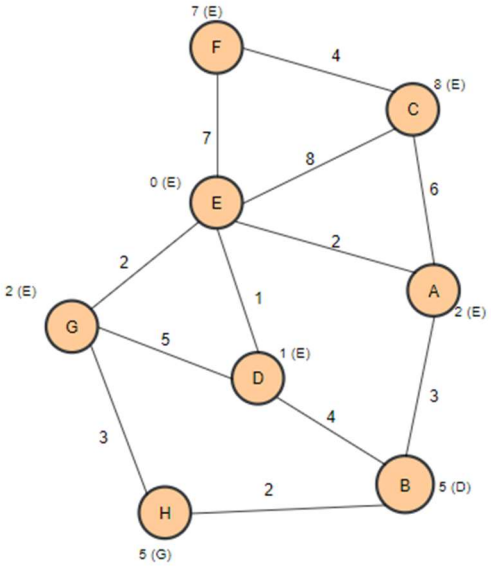
Then we continue with vertex F as it has the minimum distance among remainings. It only has one unvisited adjacent vertex but path F provides is not shorter than what C has currently (7 + 4 > 8)



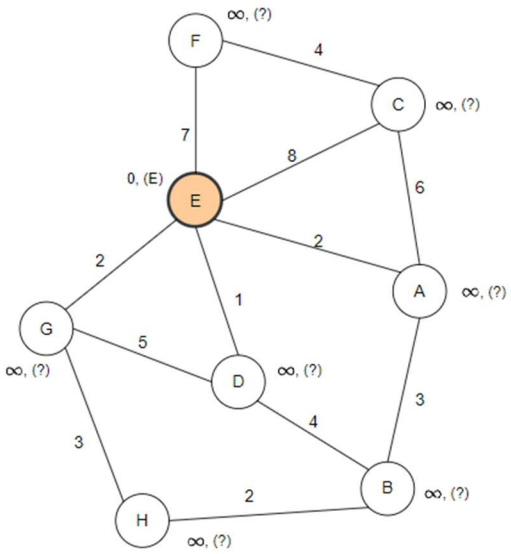
Finally, we visit the vertex C as it's last one unvisited. It has no unvisited adjacent vertex so we don't do any update.



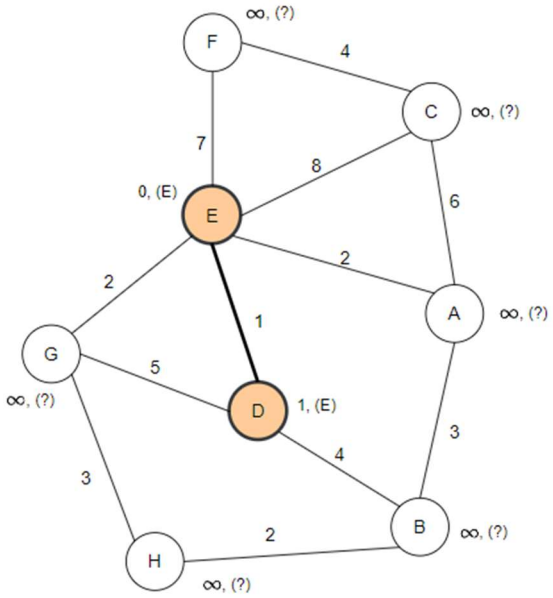
By doing that, all vertices are visited and minimum distances from vertex E has been recorded for each.



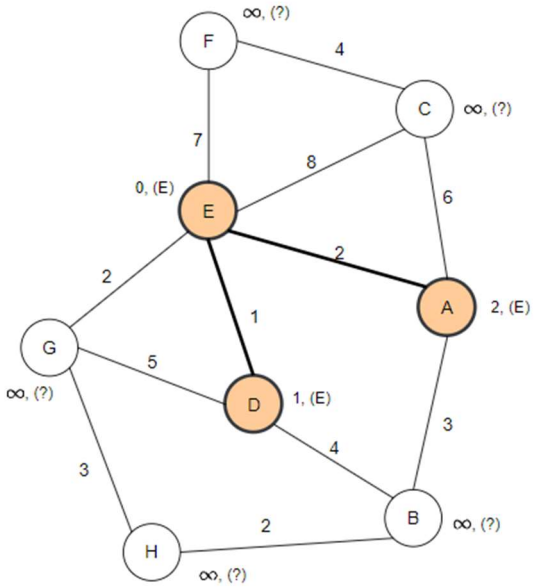
Question 2: Starting with initializing the tree, for now, tree only consists of vertex E as it's the starting vertex. In graphs below, infinity and ? are used to show that such node has not yet connected to the tree. Vertices and edges in the tree are shown with bold lines and pinkish color.



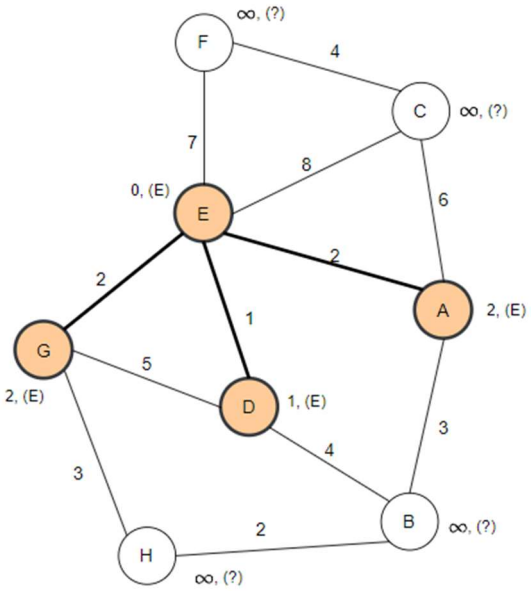
Then, we can continue with “closest vertex to the tree” which is vertex D in this case, due to (E, D) having weight 1. We connect vertex D to the tree and we can update vertex D’s information as it connects to tree via vertex E and edge connecting it has weight 1.



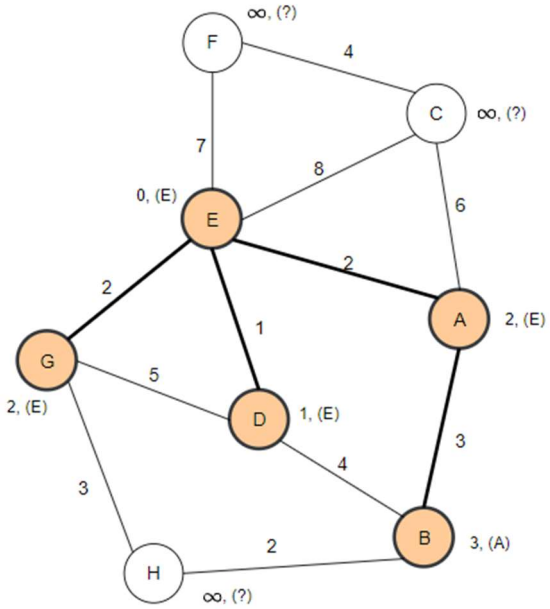
After that, there are two “closest vertices to tree”, namely A and G. We choose vertex A as an arbitrary choice among them. We connect the vertex to tree and update its information as it connects to tree via vertex E and edge connecting it has weight 2. Edge accepted to the tree becomes (E, A).



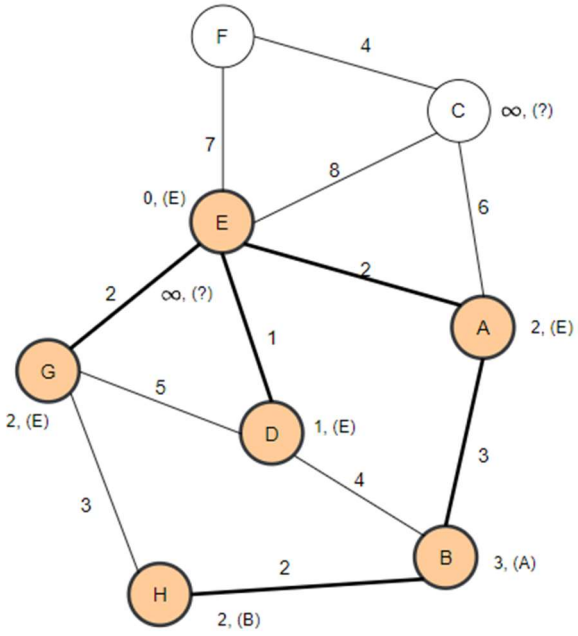
Next, closest vertex to tree is vertex G. We connect the vertex to tree and update its information as it connects to tree via vertex E and edge connecting it has weight 2. Edge accepted to tree becomes (E, G).



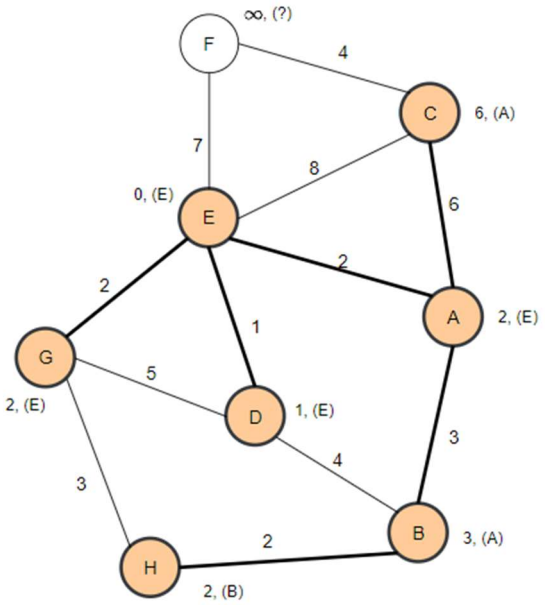
On next step, there are two “closest vertices to tree”, namely B and H. We choose vertex B as an arbitrary choice among them. We connect the vertex to tree and update its information as it connects to tree via vertex A and edge connecting it has weight 3. Edge accepted to tree becomes (A, B). (Note: if one started the algorithm by finding the shortest edge, he/she would see that H-B is the shortest, but it does not connect any vertices to the tree, so we continue with the next shortest)



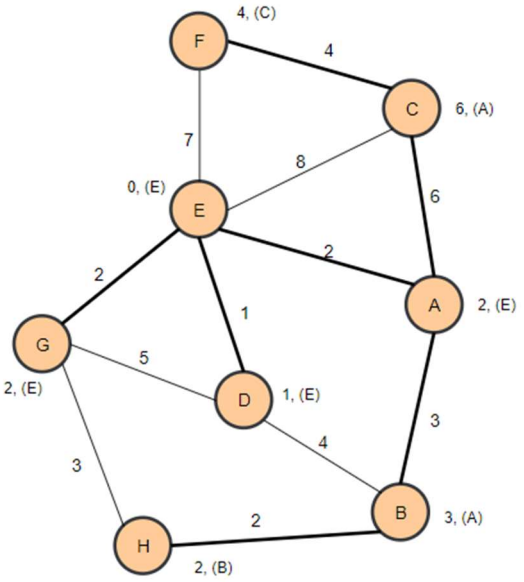
After that, closest vertex to tree is vertex H. We connect the vertex to tree and update its information as it connects to tree via vertex B and edge connecting it has weight 2. Edge accepted to tree becomes (B, H).

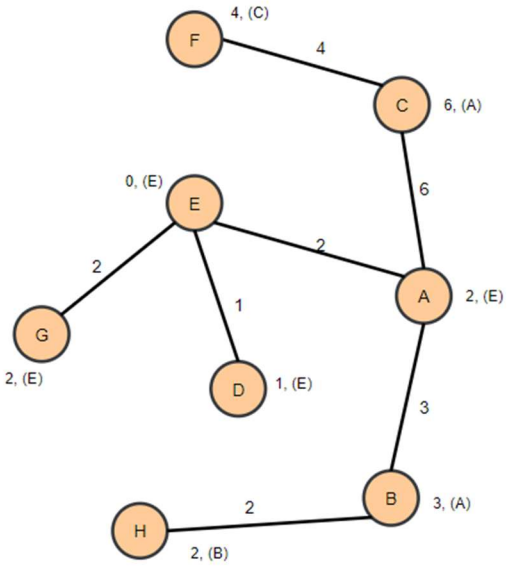


Next, closest vertex to tree is vertex C. We connect the vertex to tree and update its information as it connects to tree via vertex A and edge connecting it has weight 6. Edge accepted to tree becomes (A, C).



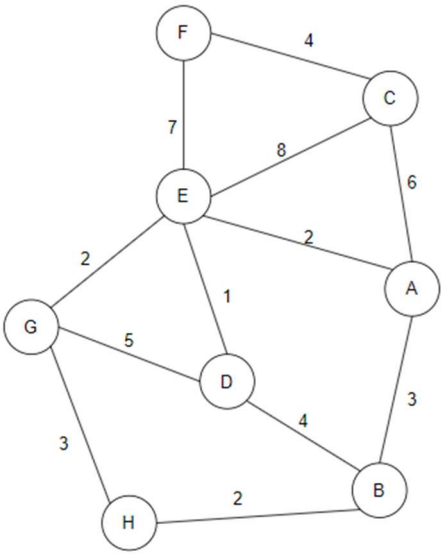
Finally, there is only one vertex left, namely F. Shortest edge that connects it to the tree is (F, C) so we connect vertex F with this edge and update its information as below (left).



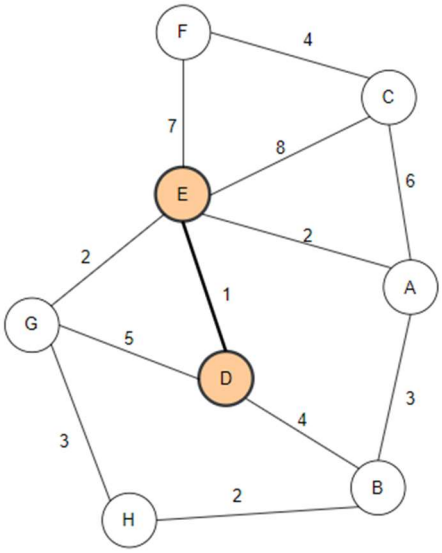


Final version of the minimum spanning tree is on the right.

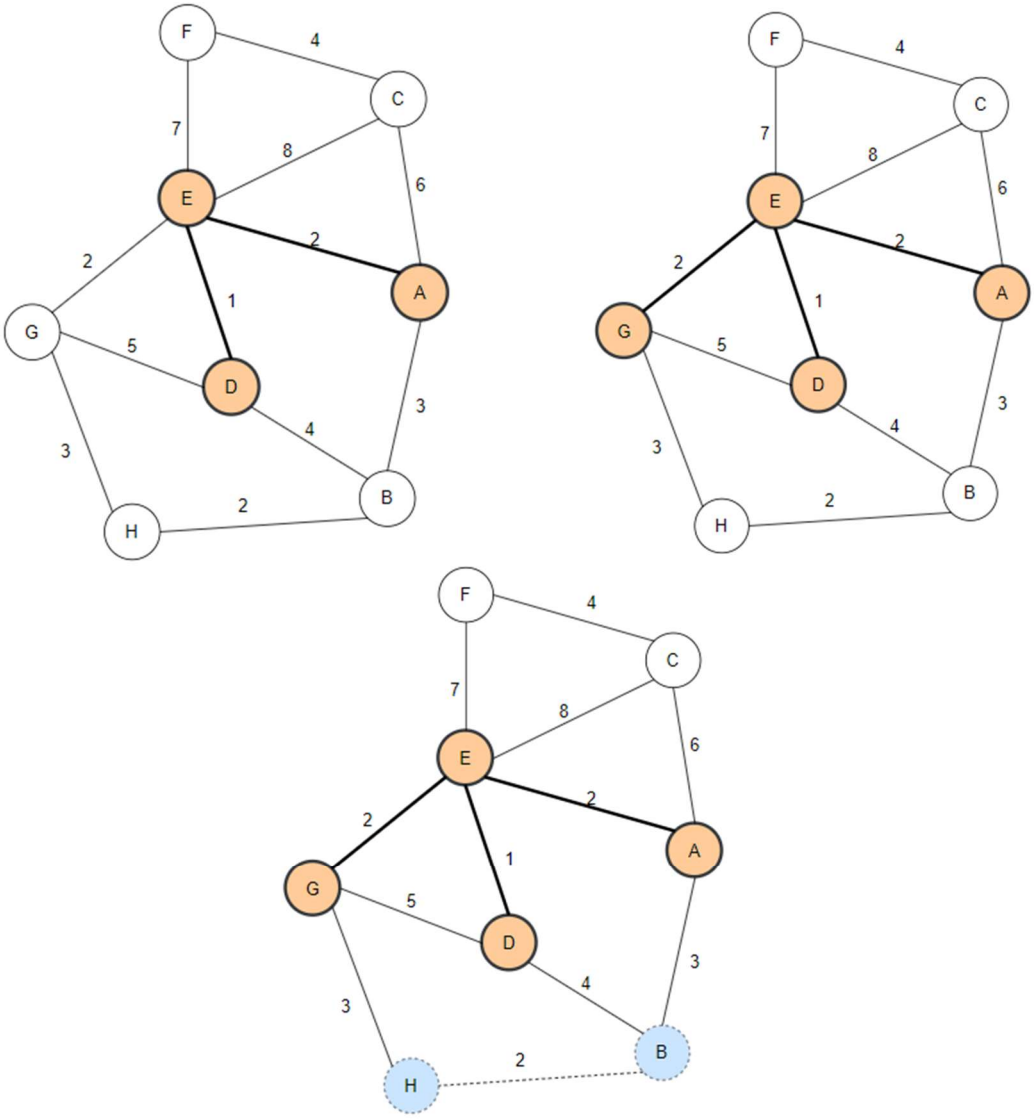
Question 3: Before starting, we must initialize sets that consist of single vertices. For the sake of simplicity, I didn't create a seperate set view, each of the white vertices are a set by themselves. When two single vertex sets united, they change color to show they are one set now.



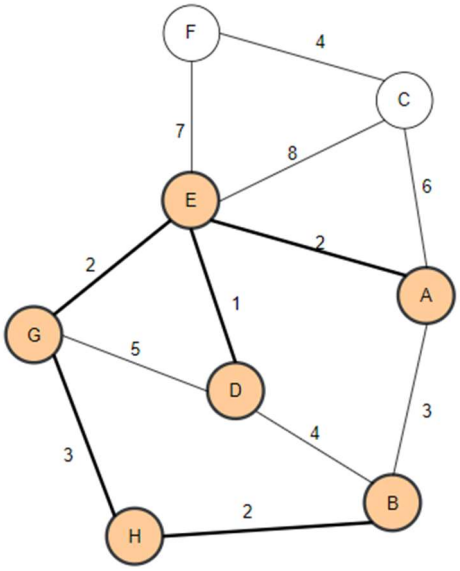
At each step, we must find the shortest edge that is connecting two seperate sets. First such edge is (E, D) so we accept it to the tree. (Note that because E and D are unioned, they changed color to signal that they form a larger set now)



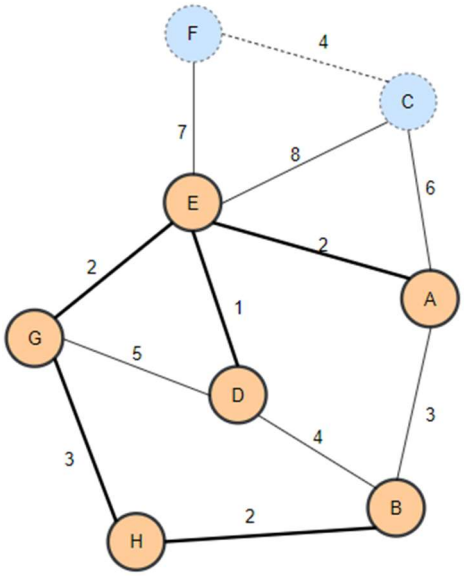
Next, we have three “shortest” edge in remaining sets. Each of these edges are connecting two seperate sets together (and order or connecting does not create a problem in terms of seperate sets) so we can accept all three one by one. In other words, (E, A), (E, G) and (H, B) are accepted. Vertices A and B are connected to a previously unioned set so their will be pinkish. However, vertices H and B are forming a new set other than the previous union, so their color is blue and their lines are dashed for now.



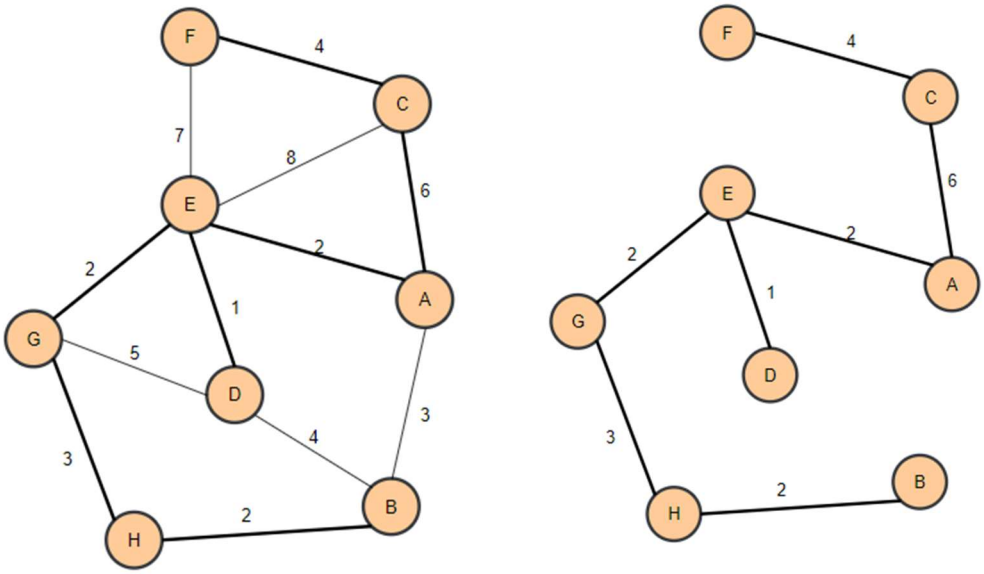
After that, shortest edges among remaining ones are (G, H) and (A, B), as both of them connects two seperate sets we can choose one arbitrarily, let's say (G, H). We accept it to tree and union pinkish and blue set (bold and dashes for monochrome print).



If we continue with shortest edges, we will see that (A, B) and (D, B) are the shortest edges in order. However, they aren't connecting two seperate sets, vertices they connect are already in the same set. Thus, we pass them. Next, we have (F, C), we accept it and a new set is formed out of union.



Then, we see that (G, D) is the shortest edge but due to it connecting vertices from same set, we cannot accept them. We continue with (C, A) as it has the next smallest weight. It connects two sets, blue and pinkish (dashed and bold for monochrome print). (Figure on the left)



We have accepted enough edges to connect all vertices, we also have only one set left so we can finish the algorithm. Final version of the tree is on the right.

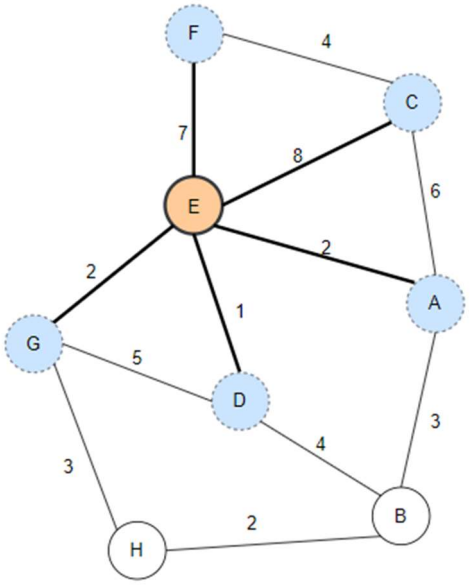
Question 4: For this question, rather than level by level method I've implemented the queue method. Pink-bold vertices are visited ones, blue-dashed ones are ones that are added to the queue and rest of them are neither, also bold edges are to show adjacent vertices. Pre-order traversal is assumed.

To start the traversal, we add vertex E to the queue,

Front [E] Rear

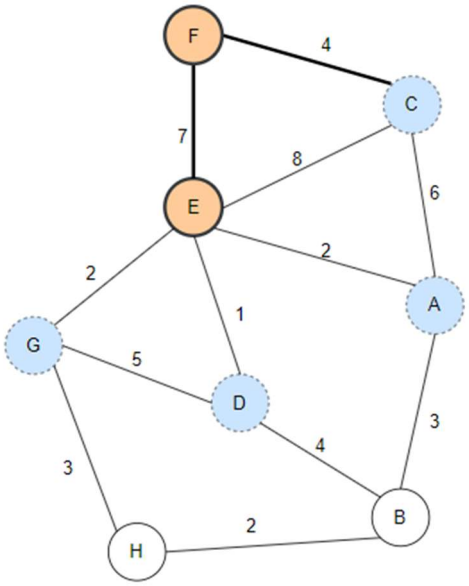
then we visit it (by removing it from queue) and add its adjacent vertices to the queue.

Front [F, C, A, D, G] Rear



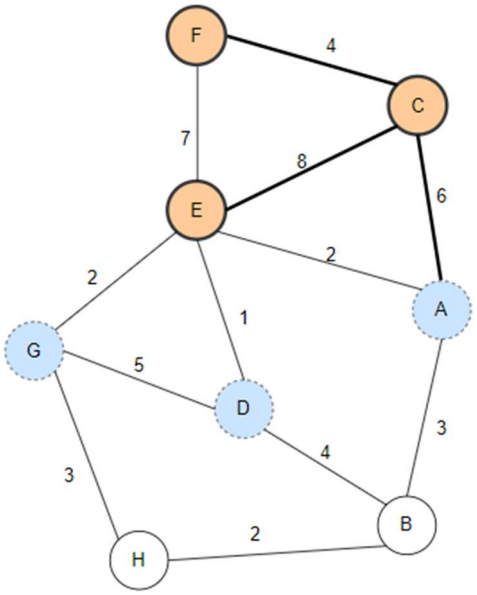
Then we continue with what's on front of queue, in this case it's vertex F. We visit the vertex F and remove it from queue, but it doesn't have any adjacent that is neither unvisited or not on queue so we continue.

Front [C, A, D, G] Rear



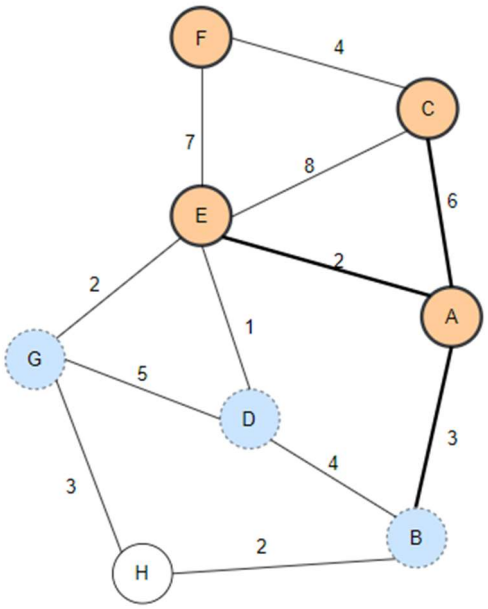
Next, we have vertex C in front. We visit it and remove it from queue. Vertex C doesn't have any adjacent that is neither unvisited nor not on queue, so we continue.

Front [A, D, G] Rear



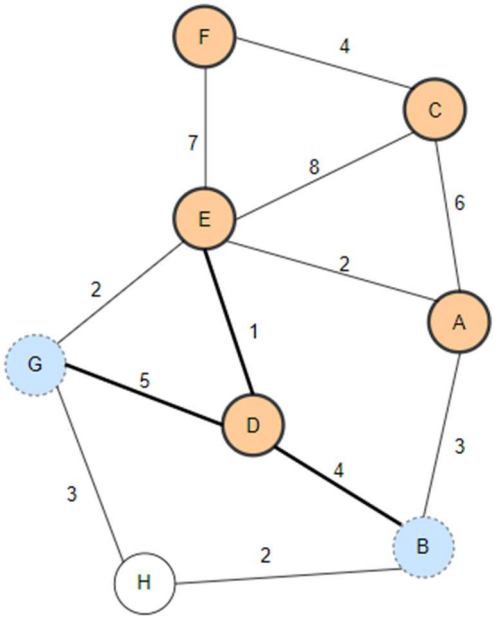
After that, we have vertex A in front. We visit it and remove it from queue. Vertex A has one adjacent vertex that is not visited and not on queue, namely vertex B, so we add it to the queue.

Front [D, G, B] Rear



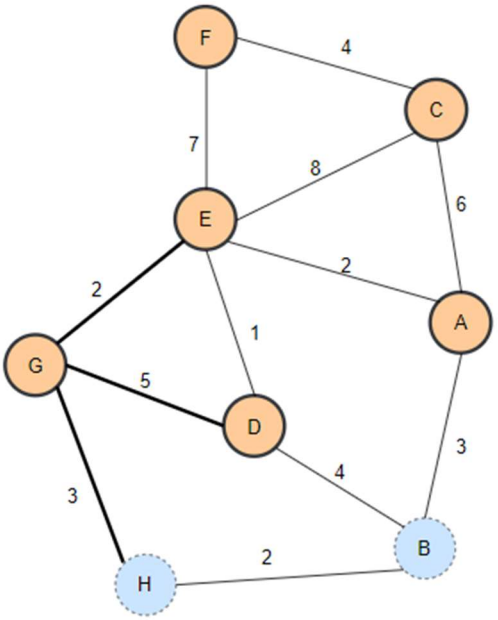
Next on line, we have vertex D. We visit it and remove it from queue. Vertex D doesn't have any adjacent that is neither unvisited nor not on queue, so we continue.

Front [G, B] Rear



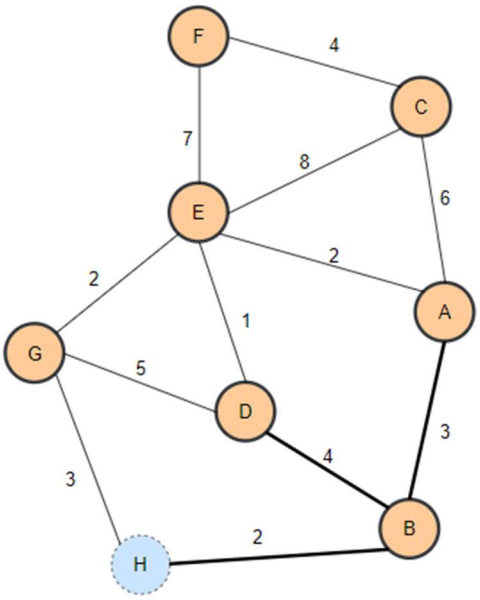
After that, we have vertex G in front. We visit it and remove it from queue. Vertex G has one adjacent vertex that is not visited and not on queue, namely vertex H, so we add it to the queue.

Front [B, H] Rear



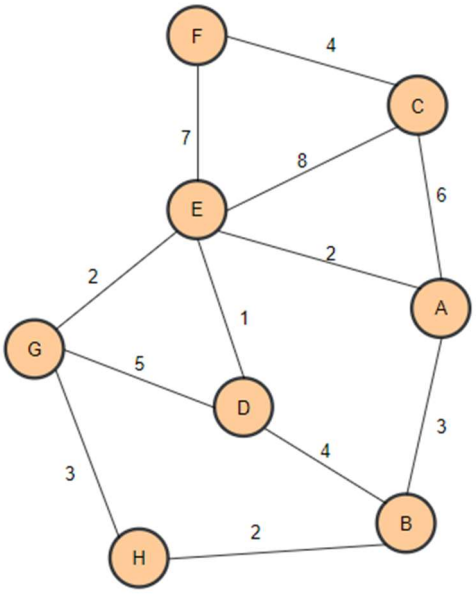
Next vertex in front is vertex B. We visit it and remove it from queue. Vertex B doesn't have any adjacent that is neither unvisited nor not on queue, so we continue.

Front [H] Rear



We have only one vertex left on the queue, it's also the last vertex that is not visited yet. We visit vertex H and remove it from the queue. It has no adjacent vertex that is not yet visited or not on queue. By visiting vertex H, we finish the traversal.

In short, order of traversal is as follows: E - F - C - A - D - G - B – H



Question 5: In each step of topological ordering, we find a vertex with in-degree 0 (i.e. no edge leads to this vertex), write it to the result and remove it from graph. If there are multiple such vertices, one of them is chosen arbitrarily. This process continues until there is no vertex left.

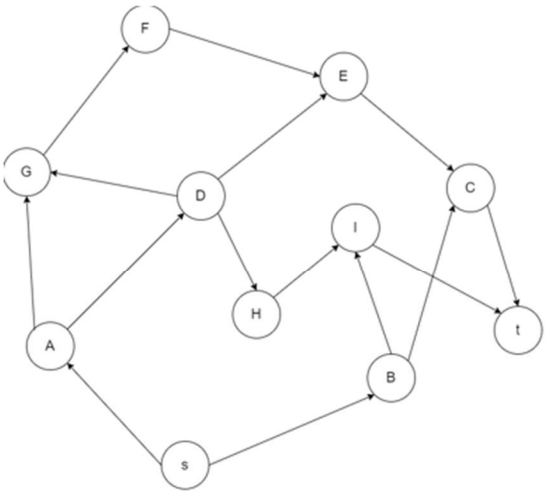
Result of one possible topological ordering is as:

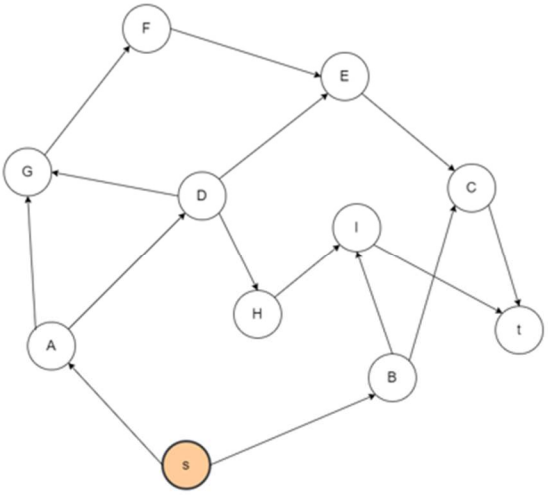
s – A – B – D – G – H – F – I – E – C – t

Of course, it's not unique due to the arbitrary choices. Rest of the file contains the steps topological ordering. Colored vertices at each step represent vertices with in-degree 0. Bold circles indicate which one is selected and added to the ordering at that stage.

Note: Order of the graphs are from left to right.

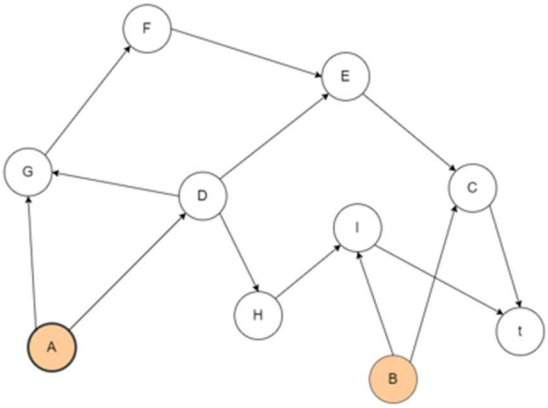
At initial version of the graph, there is only one vertex with in-degree = 0. Thus, we start with it, namely s. We remove it from graph and add to order. Order is now: s

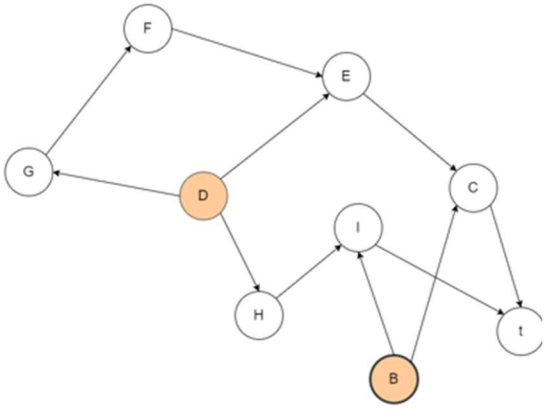




After that, we have two vertices with in-degree = 0, namely A and B. We choose A as an arbitrary choice, remove it and add it to the order. Order is now: s- A

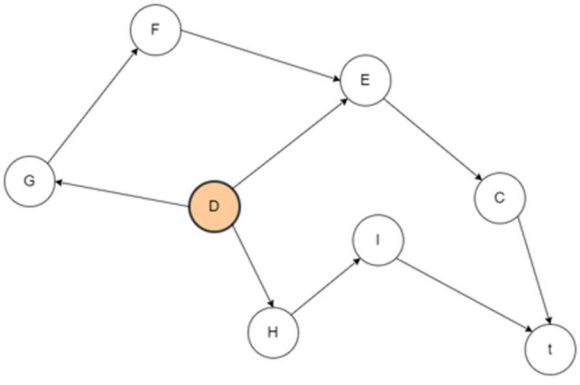
Then, we have two vertices with in-degree = 0, namely D and B. We choose B as an arbitrary choice, remove it and add it to the order. Order is now: s - A – B

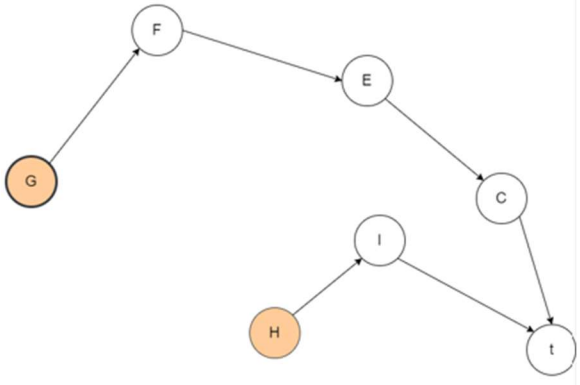




Next, we have only one vertex that has in-degree = 0, namely D. We choose it, remove it from graph and add it to the order. Order is now: s – A – B – D

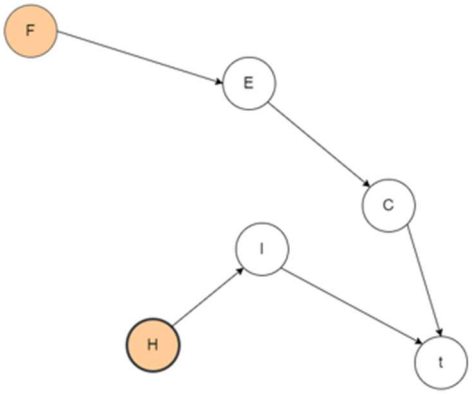
Then, we have two vertices with in-degree = 0, namely G and H. We choose G as an arbitrary choice, remove it and add it to the order. Order is now: s – A – B – D – G

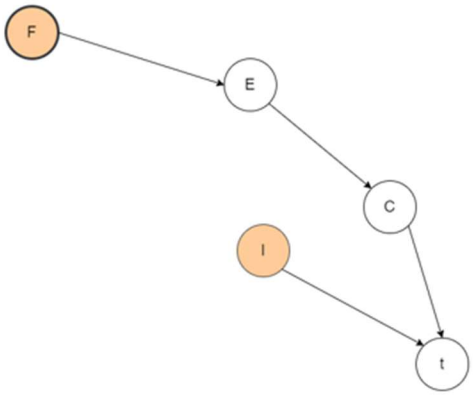




After that, we have two vertices with in-degree = 0, namely F and H. We choose H as an arbitrary choice, remove it and add it to the order. Order is now: s – A – B – D – G – H

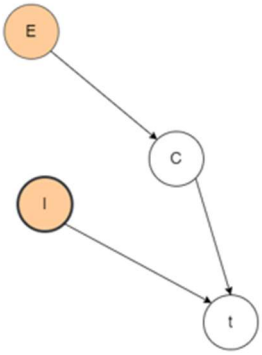
Then, we have two vertices with in-degree = 0, namely F and I. We choose F as an arbitrary choice, remove it and add it to the order. Order is now: s – A – B – D – G – H – F

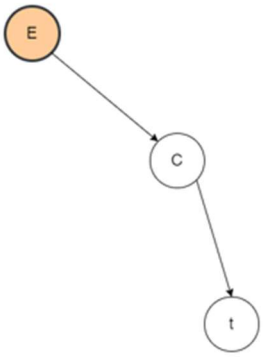




Next, we have two vertices with in-degree = 0, namely I and E. We choose I as an arbitrary choice, remove it and add it to the order. Order is now: s – A – B – D – G – H – F – I

Then, we have only one vertex that has in-degree = 0, namely E. We choose it, remove it from graph and add it to the order. Order is now: s – A – B – D – G – H – F – I – E





After that, we have only one vertex that has in-degree = 0, namely C. We choose it, remove it from graph and add it to the order. Order is now: s – A – B – D – G – H – F – I – E – C

Finally, only vertex t is left and it has in-degree = 0. We add it to the order and finish. Order is: s – A – B – D – G – H – F – I – E – C – t