CSE 373: Data Structures and Algorithms

Lecture 23: Graphs V

Dijkstra pseudocode

```
Dijkstra(v1, v2):
   for each vertex v:                    // Initialization
       v's distance := infinity.
       v's previous := none.
   v1's distance := 0.
   List := {all vertices}.

   while List is not empty:
       v := remove List vertex with minimum distance.
       mark v as known.
       for each unknown neighbor n of v:
           dist := v's distance + edge (v, n)'s weight.

           if dist is smaller than n's distance:
               n's distance := dist.
               n's previous := v.

   reconstruct path from v2 back to v1,
   following previous pointers.
```

Time Complexity: Using List

The simplest implementation of the Dijkstra's algorithm stores vertices in an ordinary linked list or array

– Good for dense graphs (many edges)

- |V| vertices and |E| edges
- Initialization O(|V|)
- While loop O(|V|)
  - Find and remove min distance vertices O(|V|)
  - Potentially |E| updates
    - Update costs O(1)
- Reconstruct path O(|E|)

Total time $O(|V^2| + |E|) = O(|V^2|)$

Time Complexity: Priority Queue

For sparse graphs, (i.e. graphs with much less than $|V^2|$ edges) Dijkstra's implemented more efficiently by *priority queue*

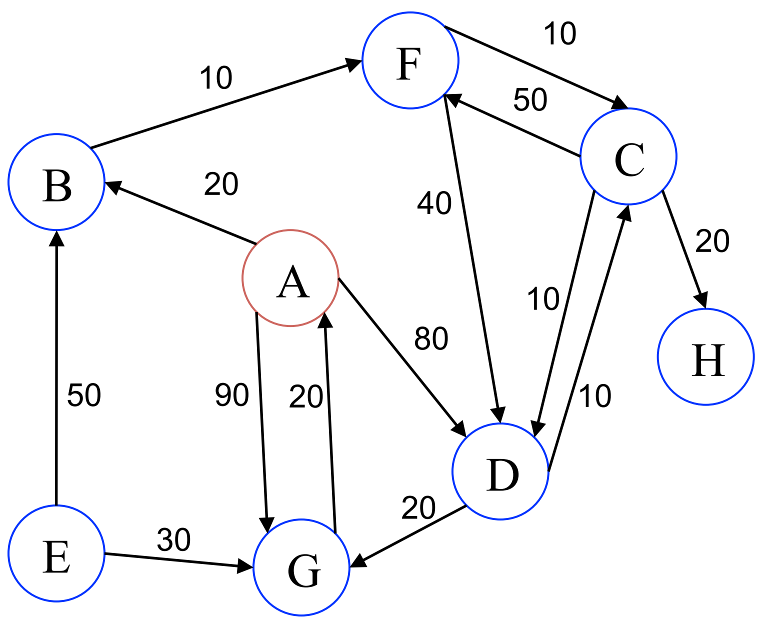
- Initialization $O(|V|)$ using $O(|V|)$ buildHeap
- While loop $O(|V|)$
  - Find and remove min distance vertices $O(\log |V|)$ using $O(\log |V|)$ deleteMin
  - Potentially $|E|$ updates
    - Update costs $O(\log |V|)$ using decreaseKey
- Reconstruct path $O(|E|)$

Total time $O(|V|\log|V| + |E|\log|V|) = O(|E|\log|V|)$

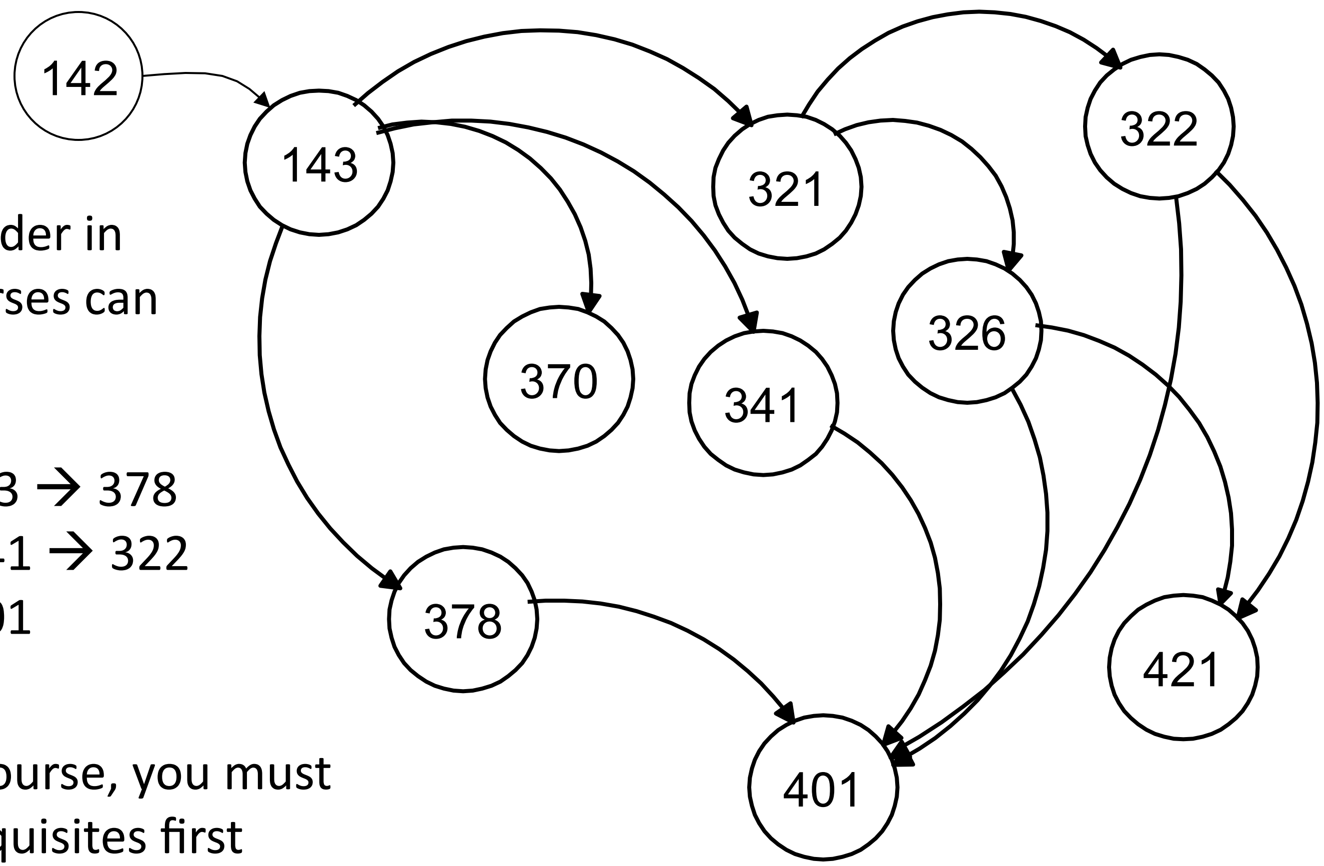
- $|V| = O(|E|)$ assuming a connected graph

Dijkstra's Exercise

- Use Dijkstra's algorithm to determine the lowest cost path from vertex A to all of the other vertices in the graph. Keep track of previous vertices so that you can reconstruct the path later.



Topological Sort



Problem: Find an order in which all these courses can be taken.

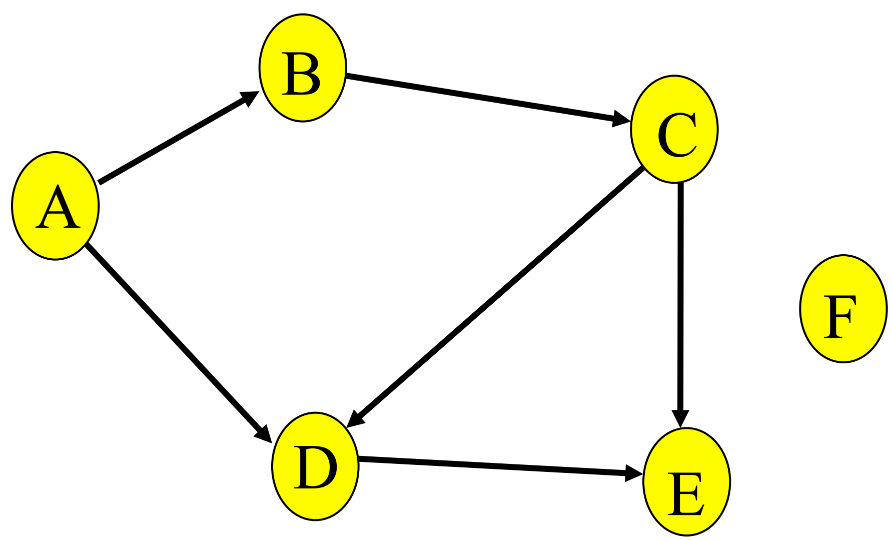
Example: 142 → 143 → 378 → 370 → 321 → 341 → 322 → 326 → 421 → 401

In order to take a course, you must take all of its prerequisites first

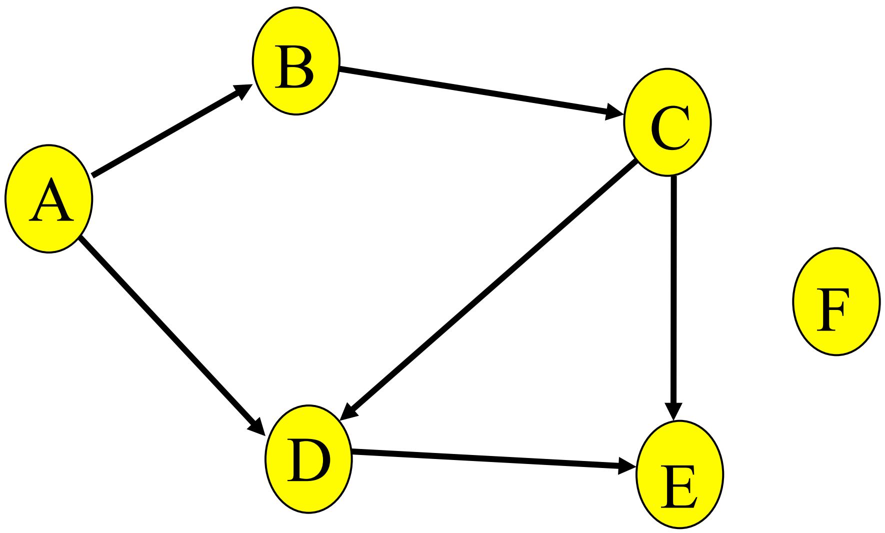
Topological Sort

Given a digraph $G = (V, E)$, find a total ordering of its vertices such that:

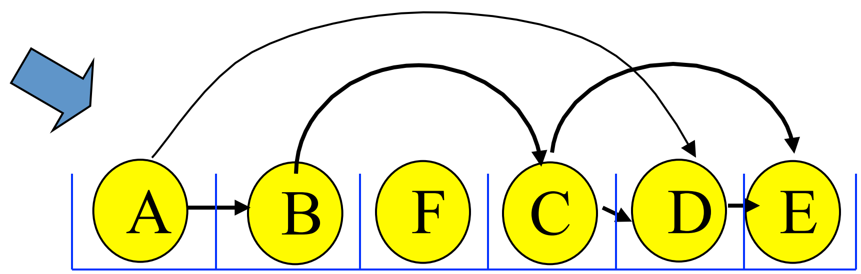
for any edge (v, w) in E, v precedes w in the ordering



Topo sort - good example

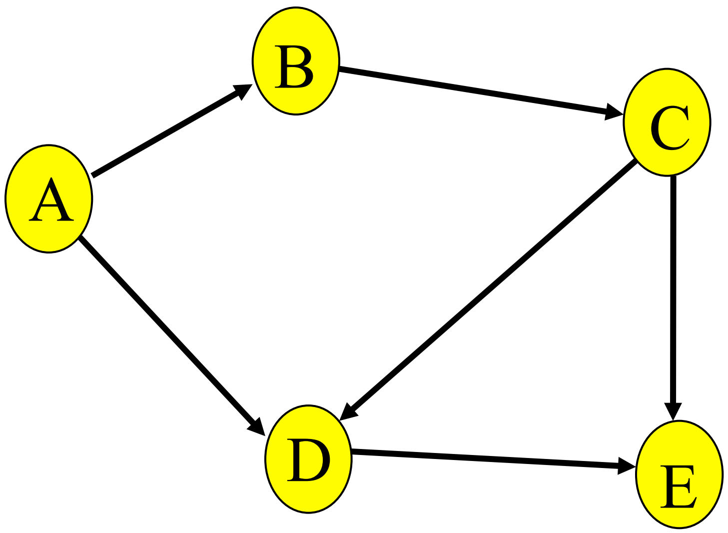


Any total ordering in which all the arrows go to the right is a valid solution

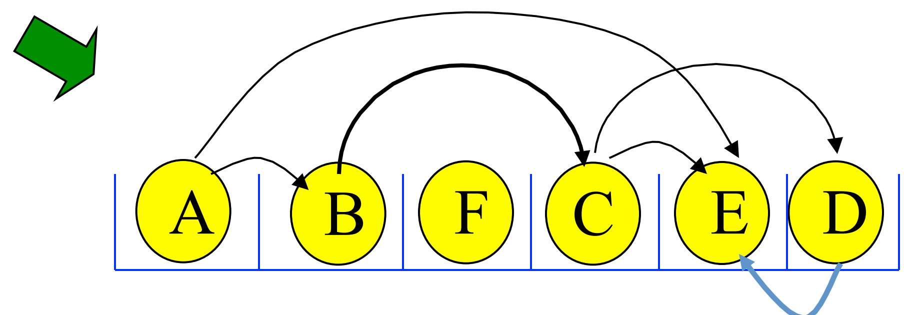


Note that F can go anywhere in this list because it is not connected.
Also the solution is not unique.

Topo sort - bad example



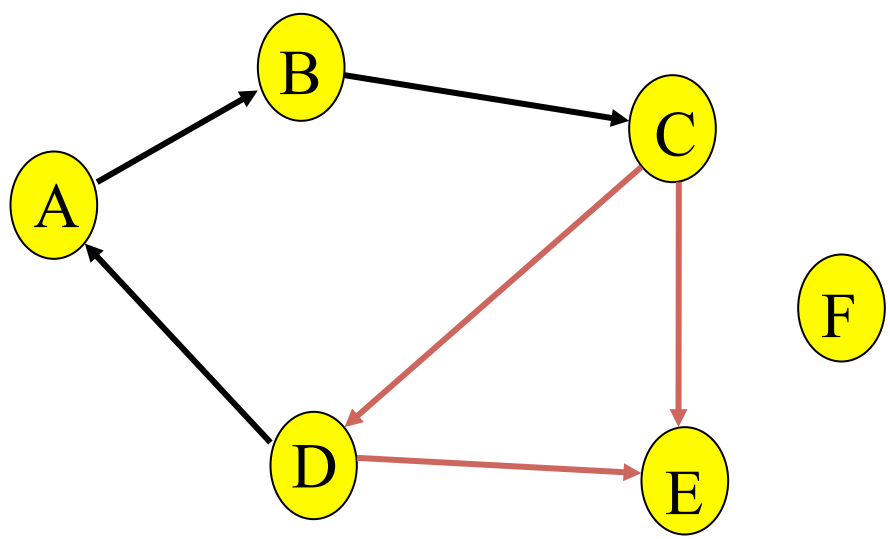
Any ordering in which
an arrow goes to the left
is not a valid solution



NO!

Only acyclic graphs can be topologically sorted

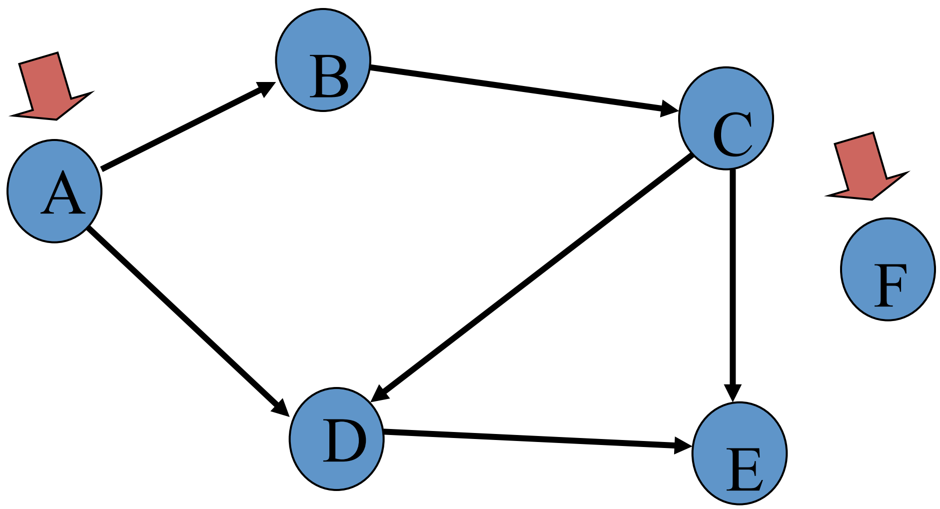
- A directed graph with a cycle cannot be topologically sorted.



Topological sort algorithm: 1

Step 1: Identify vertices that have no incoming edges

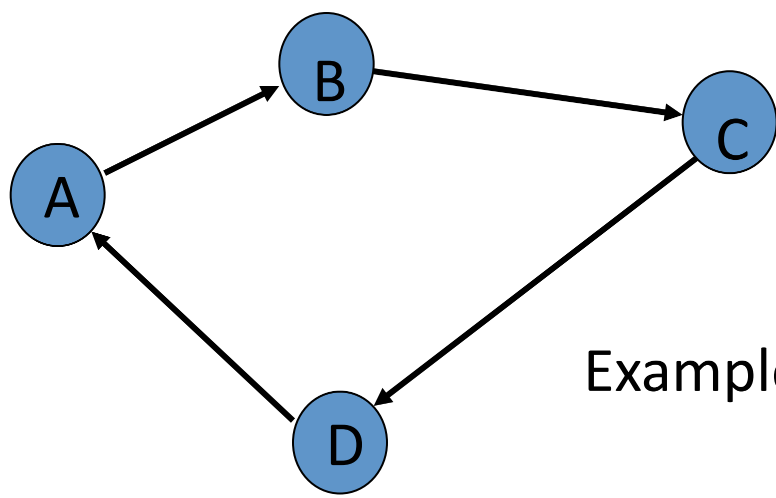
- The “in-degree” of these vertices is zero



Topo sort algorithm: 1a

Step 1: Identify vertices that have no incoming edges

- If *no such vertices*, graph has cycle(s)
- Topological sort not possible – Halt.

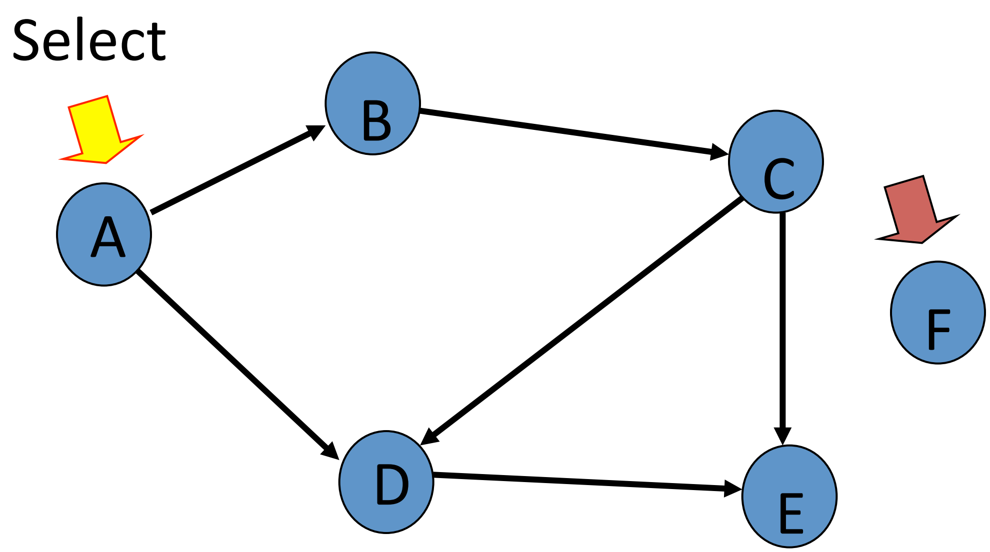


Example of a cyclic graph

Topo sort algorithm:1b

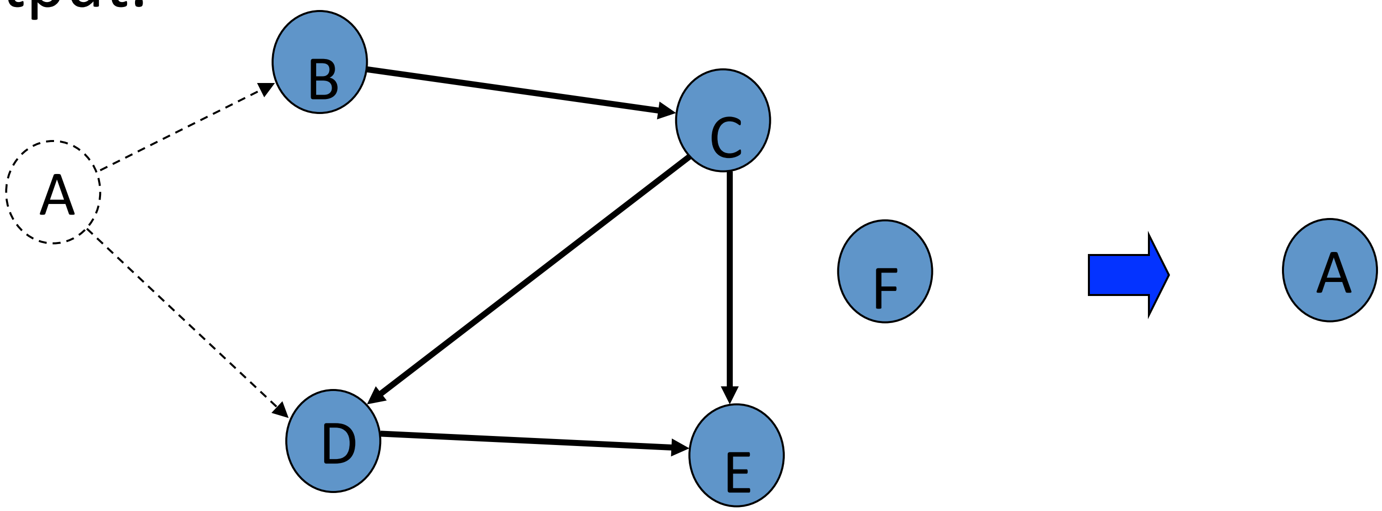
<u>Step 1</u>: Identify vertices that have no incoming edges

- Select one such vertex



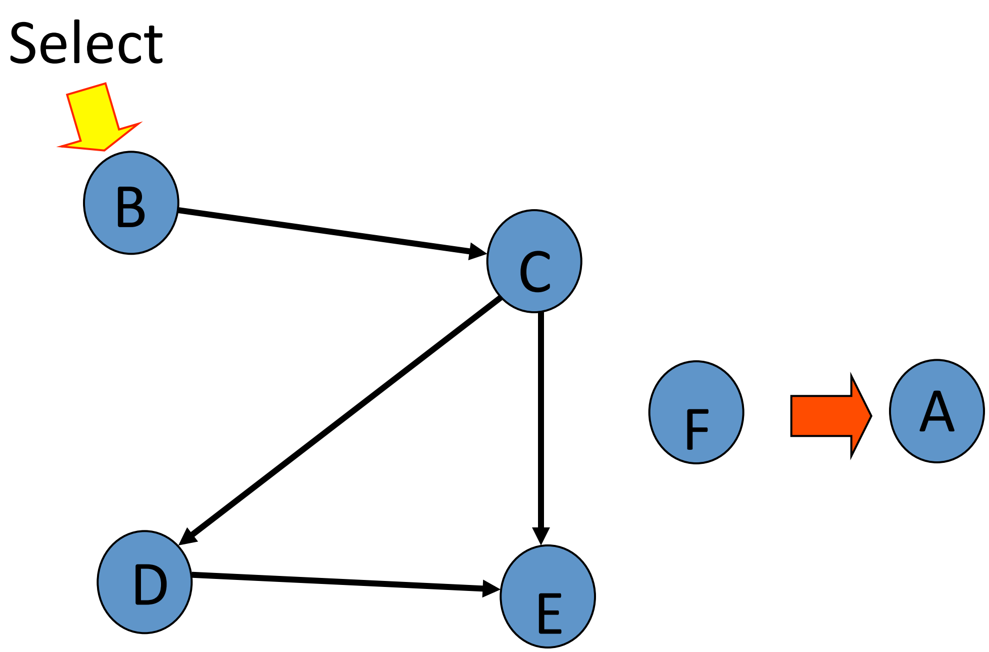
Topo sort algorithm: 2

Step 2: Delete this vertex of in-degree 0 and all its outgoing edges from the graph. Place it in the output.



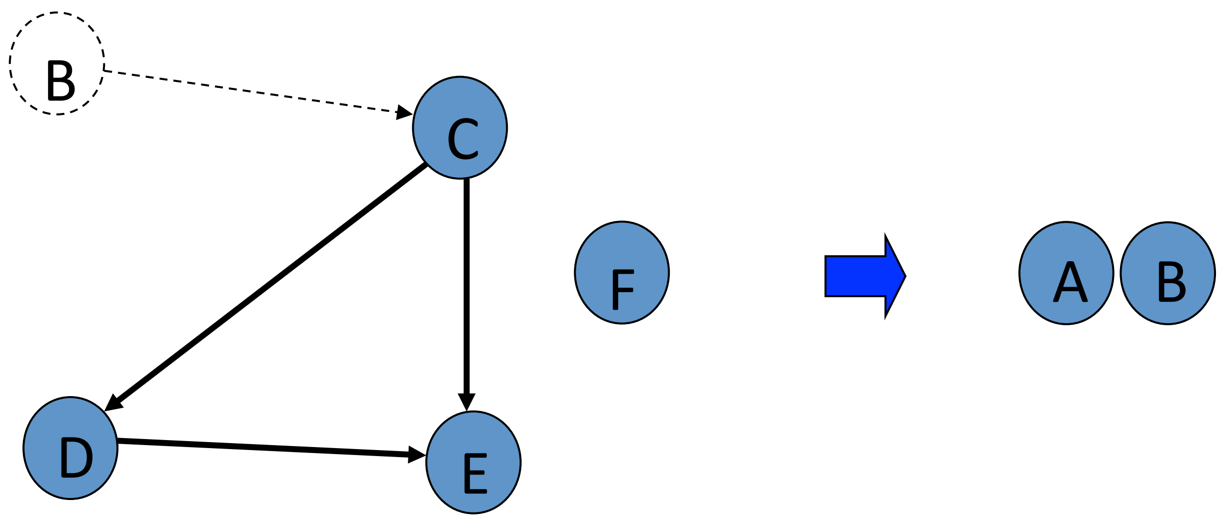
Continue until done

Repeat Step 1 and Step 2 until graph is empty



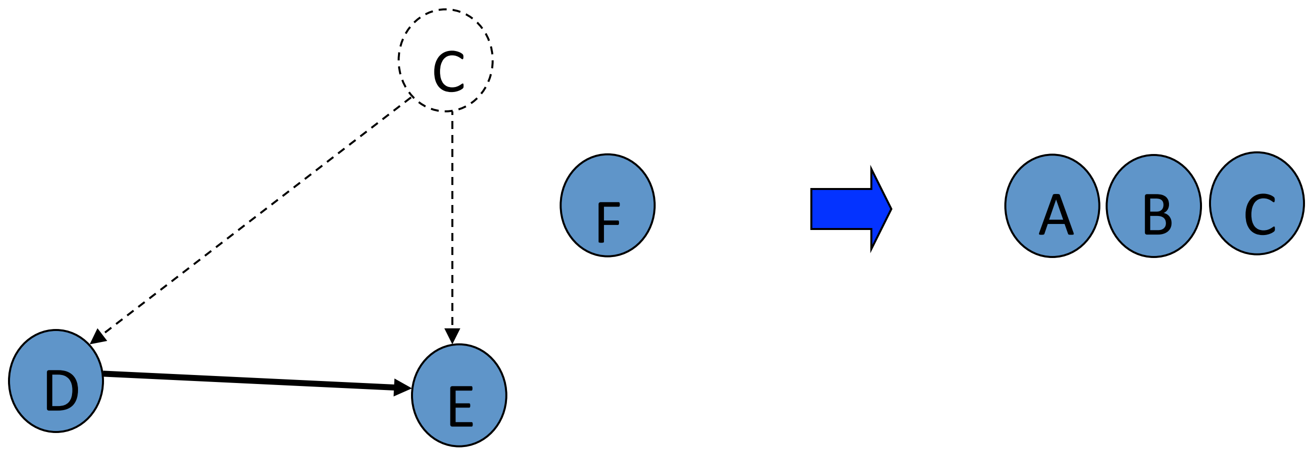
B

Select B. Copy to sorted list. Delete B and its edges.



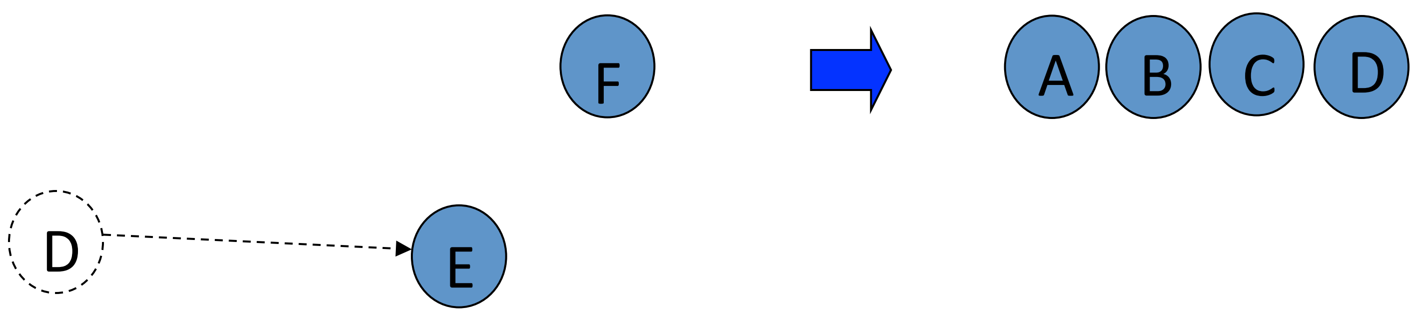
C

Select C. Copy to sorted list. Delete C and its edges.



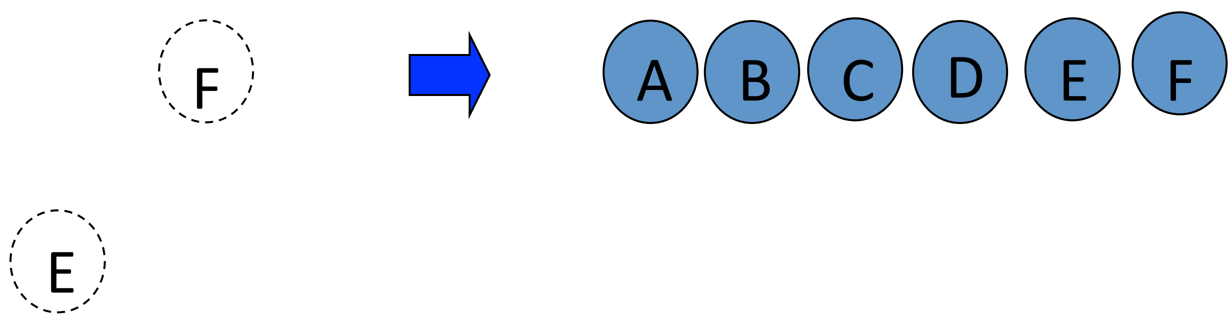
D

Select D. Copy to sorted list. Delete D and its edges.

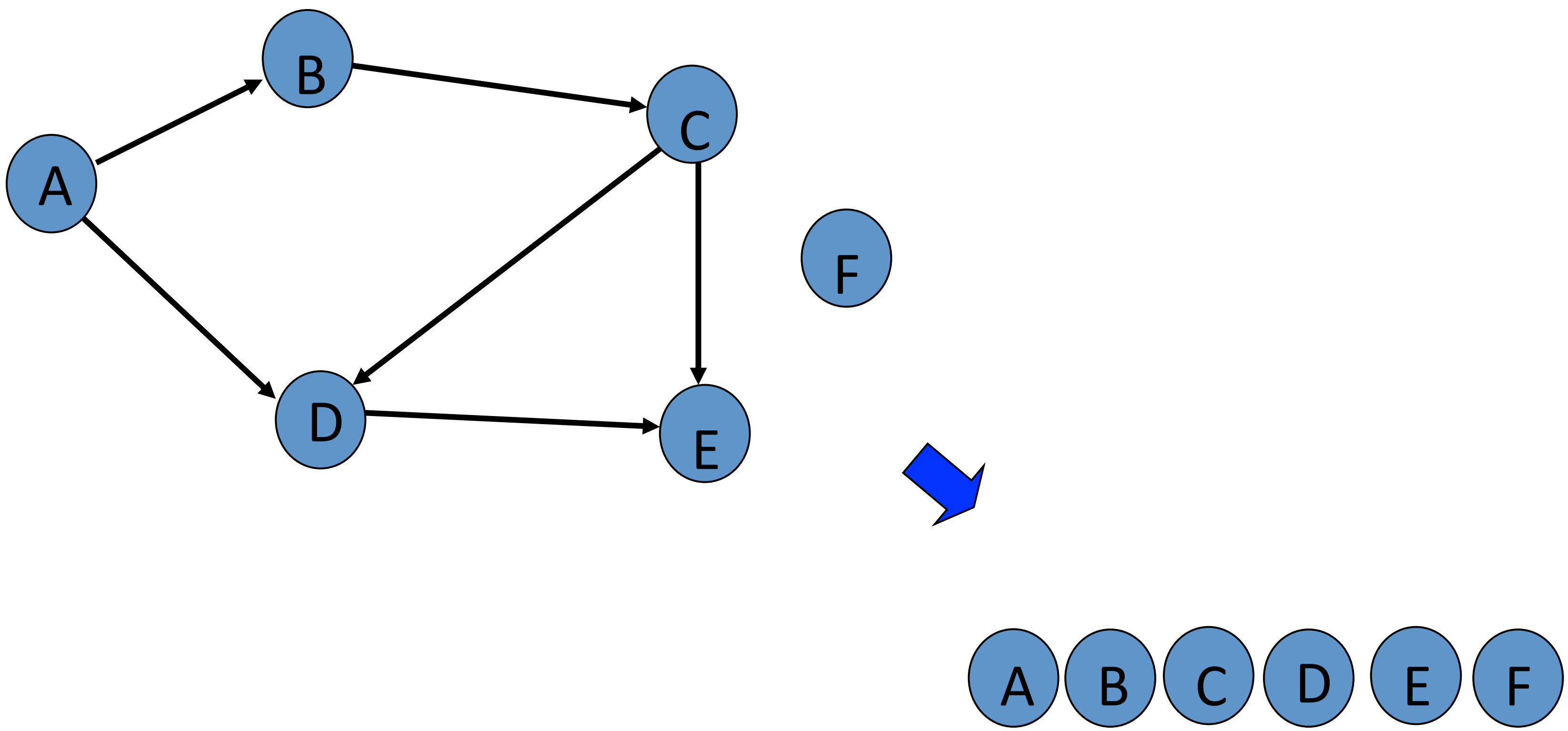


E, F

Select E. Copy to sorted list. Delete E and its edges.
Select F. Copy to sorted list. Delete F and its edges.



Done



Topological Sort Algorithm

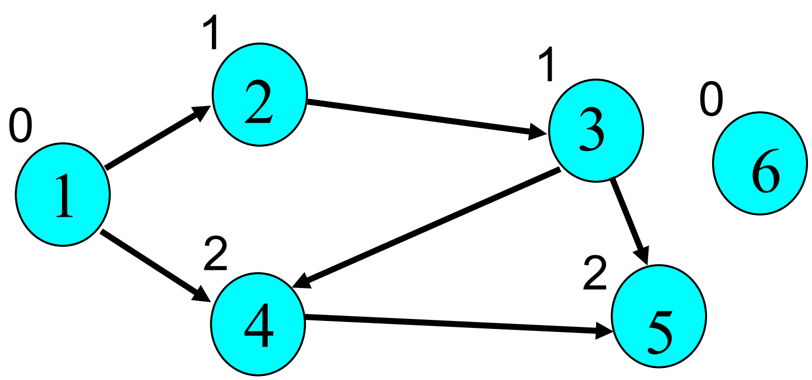
1. Store each vertex's In-Degree in an hash table D
2. Initialize queue with all "in-degree=0" vertices
3. While there are vertices remaining in the queue:
   (a) Dequeue and output a vertex
   (b) Reduce In-Degree of all vertices adjacent to it by 1
   (c) Enqueue any of these vertices whose In-Degree became zero
4. If all vertices are output then success, otherwise there is a cycle.

Pseudocode

```
Initialize D   // Mapping of vertex to its in-degree
Queue Q := [Vertices with in-degree 0]
while notEmpty(Q) do
  x := Dequeue(Q)
  Output(x)
  y := A[x]; // y gets a linked list of vertices
  while y ≠ null do
    D[y.value] := D[y.value] – 1;
    if D[y.value] = 0 then Enqueue(Q,y.value);
    y := y.next;
  endwhile
endwhile
```

Topo Sort w/ queue

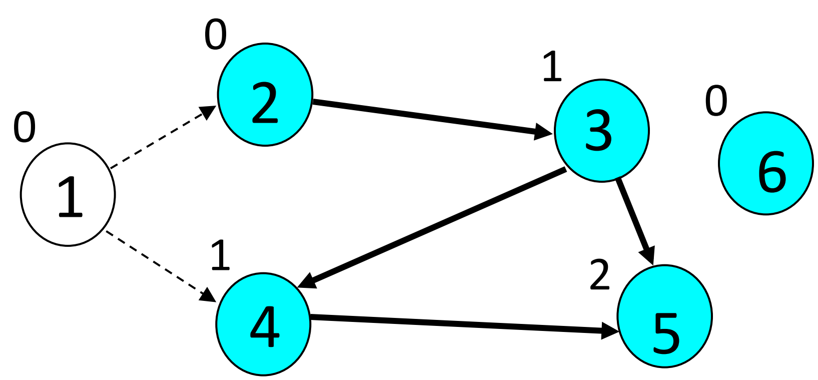
Queue (before):
Queue (after): 1, 6



Answer:

Topo Sort w/ queue

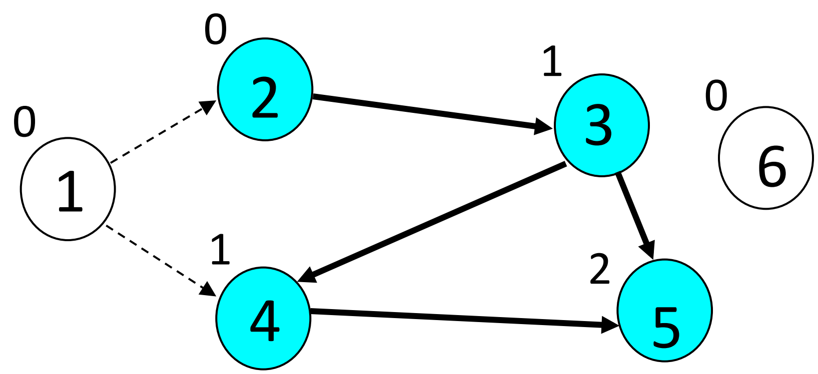
Queue (before): 1, 6
Queue (after): 6, 2



Answer: 1

Topo Sort w/ queue

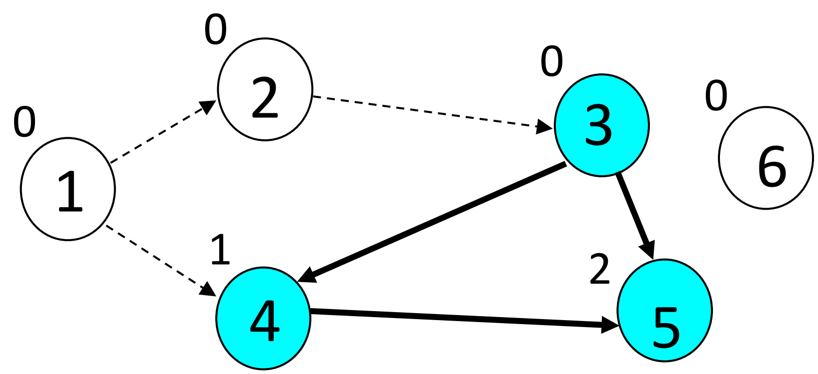
Queue (before): 6, 2
Queue (after): 2



Answer: 1, 6

Topo Sort w/ queue

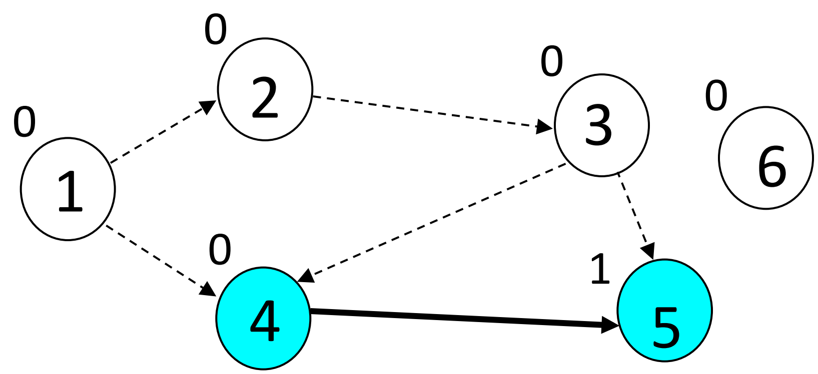
Queue (before): 2
Queue (after): 3



Answer: 1, 6, 2

Topo Sort w/ queue

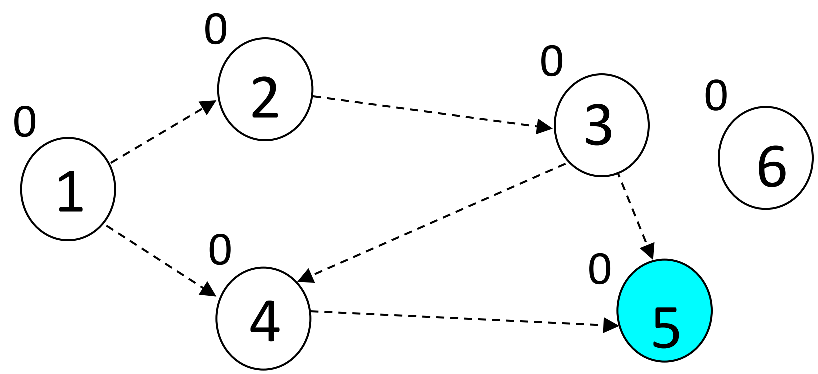
Queue (before): 3
Queue (after): 4



Answer: 1, 6, 2, 3

Topo Sort w/ queue

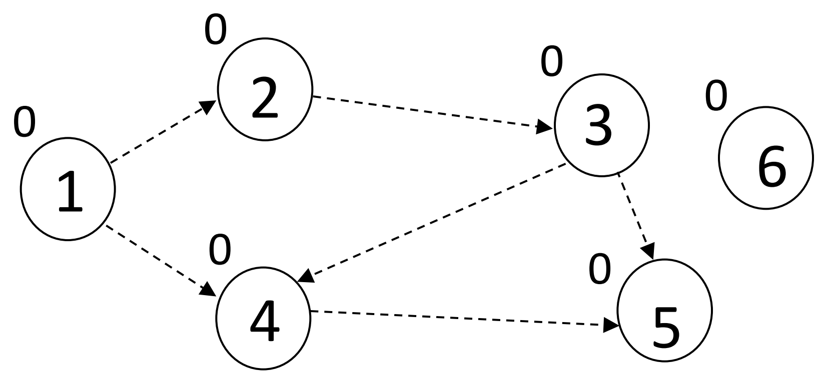
Queue (before): 4
Queue (after): 5



Answer: 1, 6, 2, 3, 4

Topo Sort w/ queue

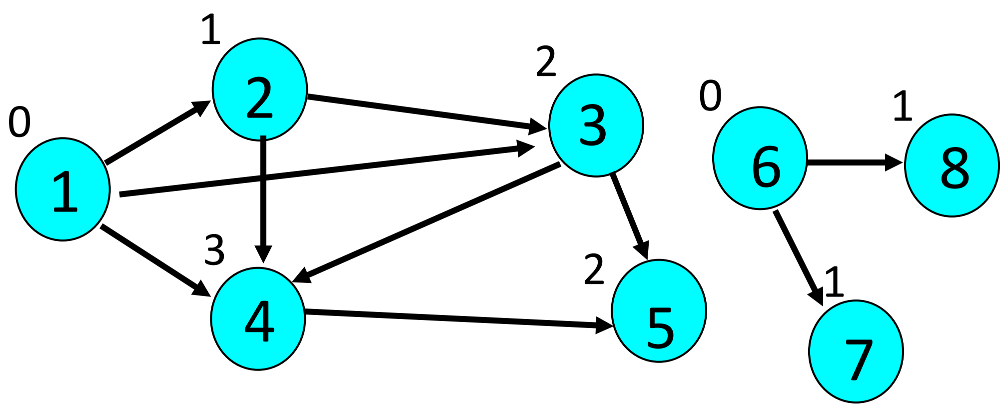
Queue (before): 5
Queue (after):



Answer: 1, 6, 2, 3, 4, 5

Topo Sort w/ stack

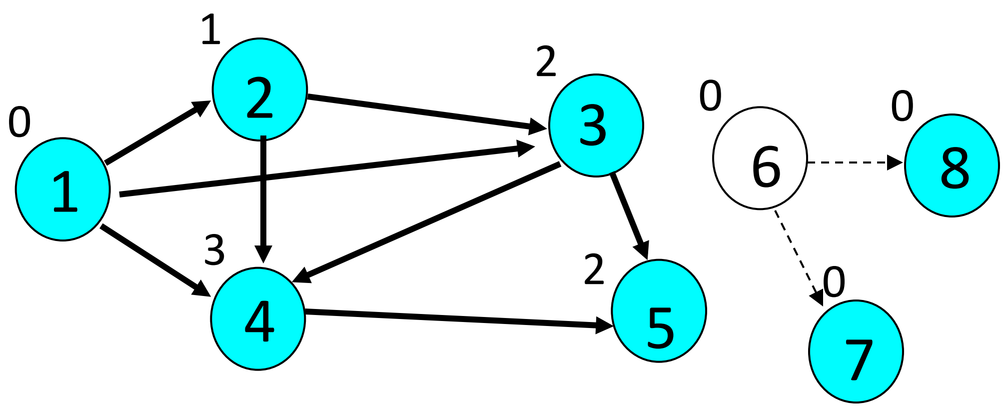
Stack (before):
Stack (after): 1, 6



Answer:

Topo Sort w/ stack

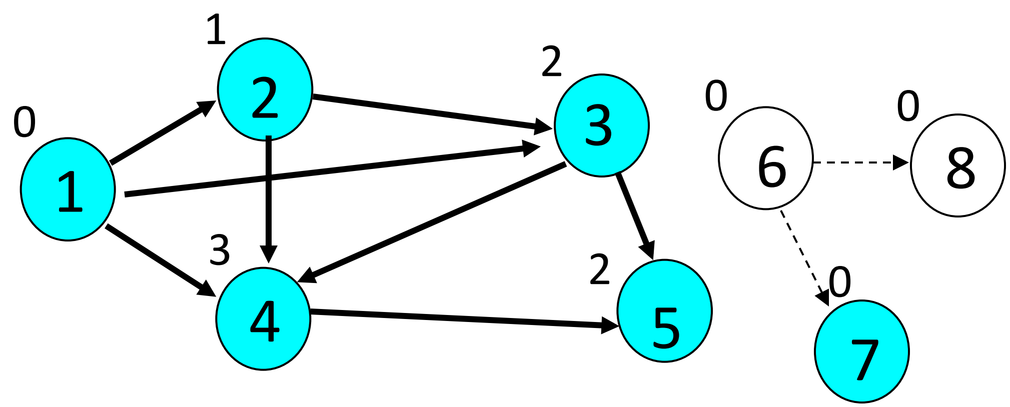
Stack (before): 1, 6
Stack (after): 1, 7, 8



Answer: 6

Topo Sort w/ stack

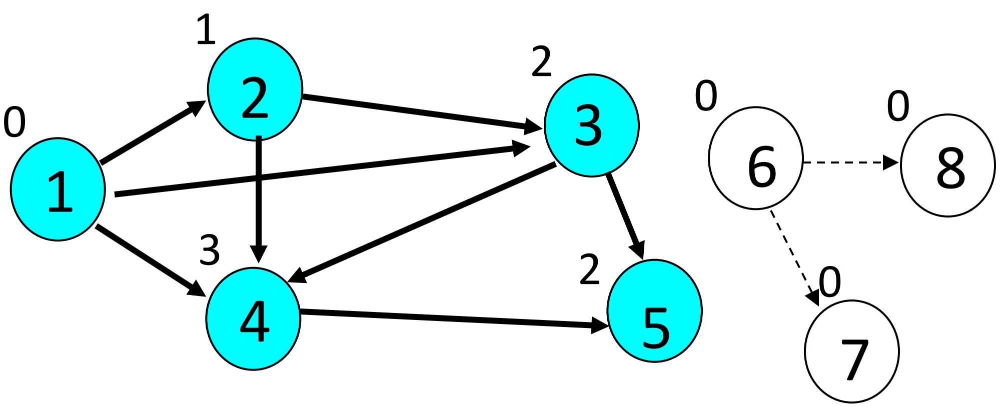
Stack (before): 1, 7, 8
Stack (after): 1, 7



Answer: 6, 8

Topo Sort w/ stack

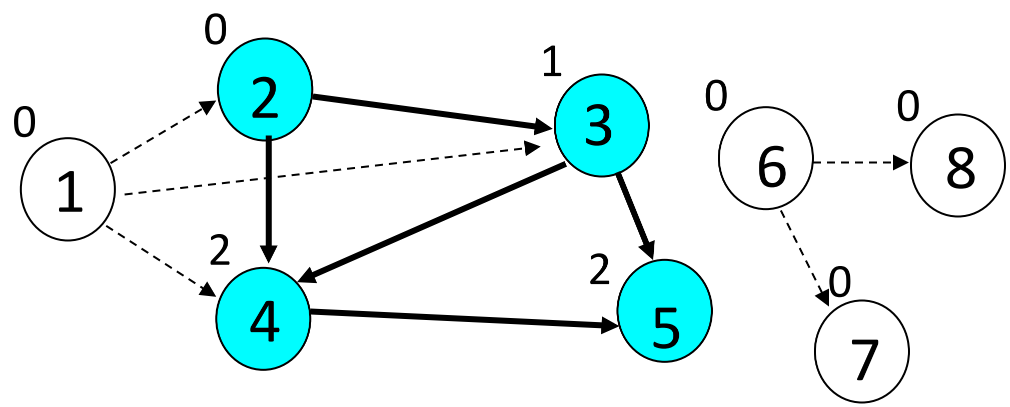
Stack (before): 1, 7
Stack (after): 1



Answer: 6, 8, 7

Topo Sort w/ stack

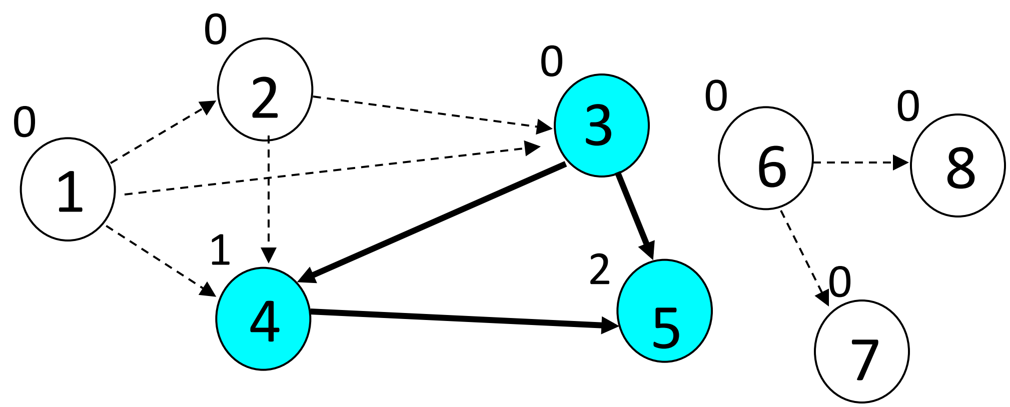
Stack (before): 1
Stack (after): 2



Answer: 6, 8, 7, 1

Topo Sort w/ stack

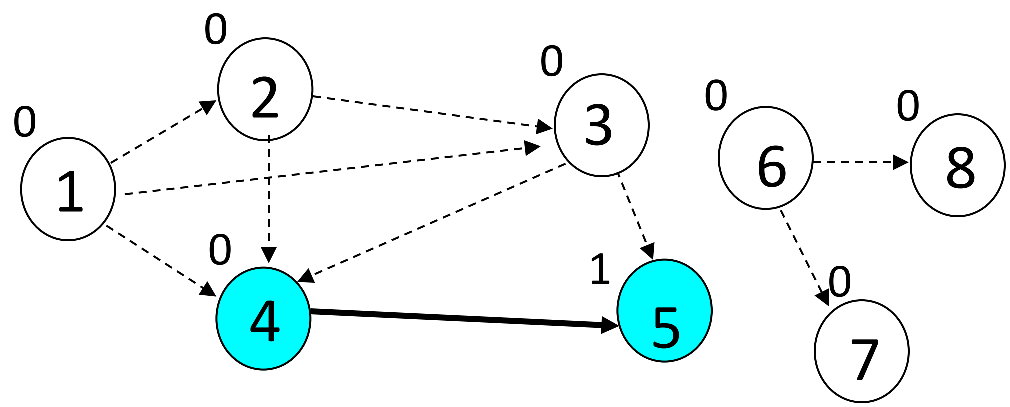
Stack (before): 2
Stack (after): 3



Answer: 6, 8, 7, 1, 2

Topo Sort w/ stack

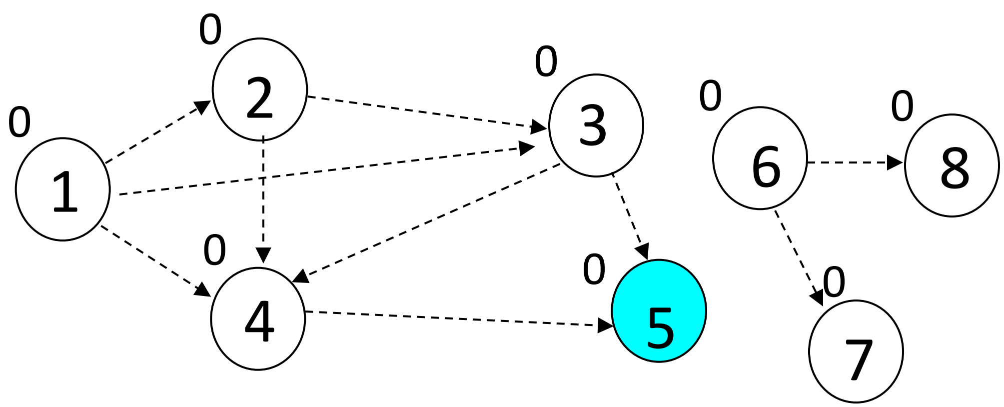
Stack (before): 3
Stack (after): 4



Answer: 6, 8, 7, 1, 2, 3

Topo Sort w/ stack

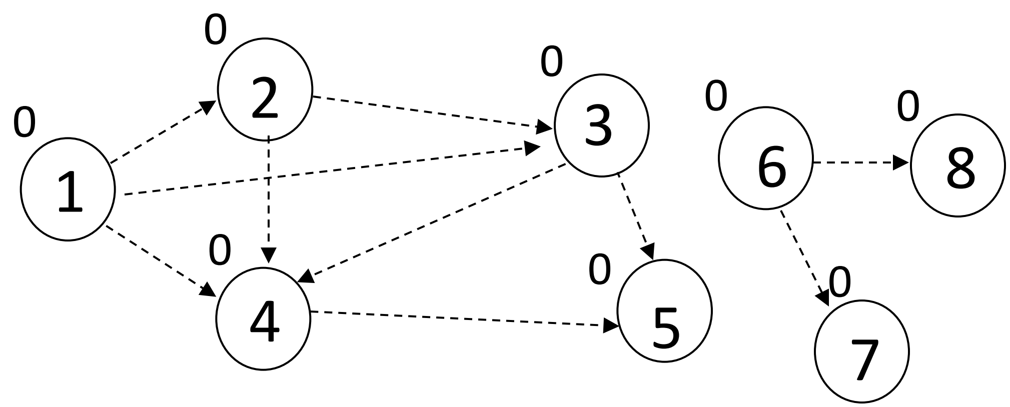
Stack (before): 4
Stack (after): 5



Answer: 6, 8, 7, 1, 2, 3, 4

Topo Sort w/ stack

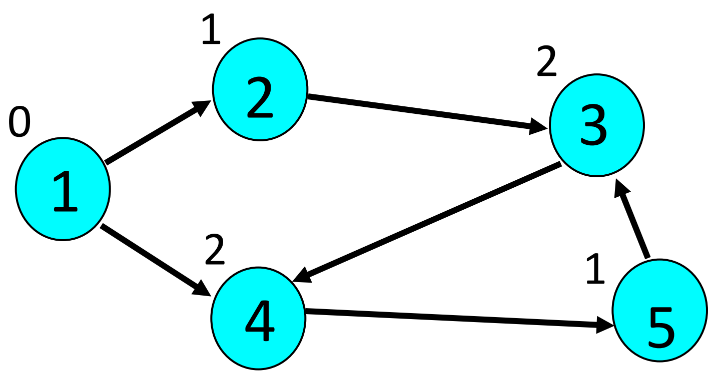
Stack (before): 5
Stack (after):



Answer: 6, 8, 7, 1, 2, 3, 4, 5

TopoSort Fails (cycle)

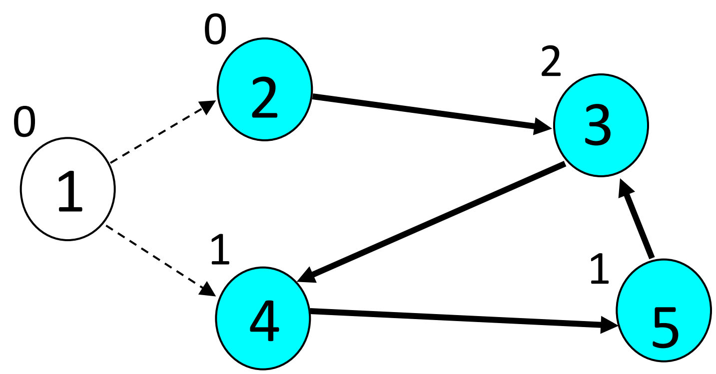
Queue (before):
Queue (after): 1



Answer:

TopoSort Fails (cycle)

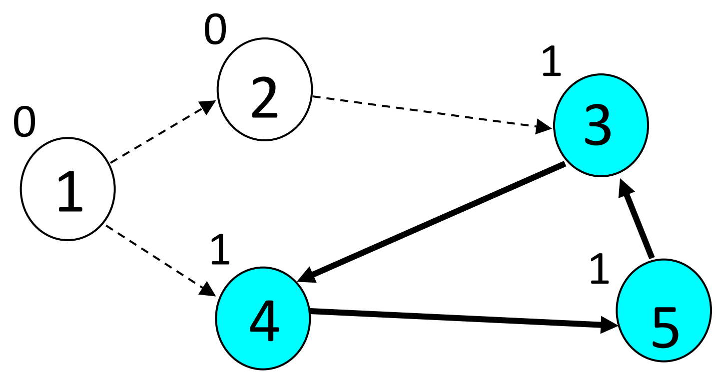
Queue (before): 1
Queue (after): 2



Answer: 1

TopoSort Fails (cycle)

Queue (before): 2
Queue (after):



Answer: 1, 2

What is the run-time???

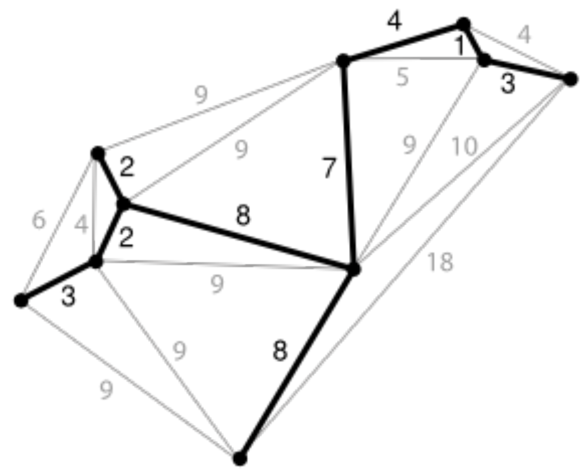
```
Initialize D   // Mapping of vertex to its in-degree
Queue Q := [Vertices with in-degree 0]
while notEmpty(Q) do
  x := Dequeue(Q)
  Output(x)
  y := A[x]; // y gets a linked list of vertices
  while y ≠ null do
    D[y.value] := D[y.value] – 1;
    if D[y.value] = 0 then Enqueue(Q,y.value);
    y := y.next;
  endwhile
endwhile
```

Topological Sort Analysis

- Initialize In-Degree array: O(|V| + |E|)
- Initialize Queue with In-Degree 0 vertices: O(|V|)
- Dequeue and output vertex:
  - |V| vertices, each takes only O(1) to dequeue and output: O(|V|)
- Reduce In-Degree of all vertices adjacent to a vertex and Enqueue any In-Degree 0 vertices:
  - O(|E|)
- For input graph G=(V,E) run time = O(|V| + |E|)
  - Linear time!

Minimum spanning tree

- **tree**: a connected, directed acyclic graph
- **spanning tree**: a subgraph of a graph, which meets the constraints to be a tree (connected, acyclic) and connects every vertex of the original graph
- **minimum spanning tree**: a spanning tree with weight less than or equal to any other spanning tree for the given graph



Min. span. tree applications

- Consider a cable TV company laying cable to a new neighborhood...
  - Can only bury the cable only along certain paths, then a graph could represent which points are connected by those paths.
  - Some of paths may be more expensive (i.e. longer, harder to install), so these paths could be represented by edges with larger weights.
  - A spanning tree for that graph would be a subset of those paths that has no cycles but still connects to every house.

- Similar situations: installing electrical wiring in a house, installing computer networks between cities, building roads between neighborhoods, etc.

Spanning Tree Problem

- Input: An undirected graph $G = (V, E)$. G is connected.
- Output: T subset of E such that
  - (V, T) is a connected graph
  - (V, T) has no cycles

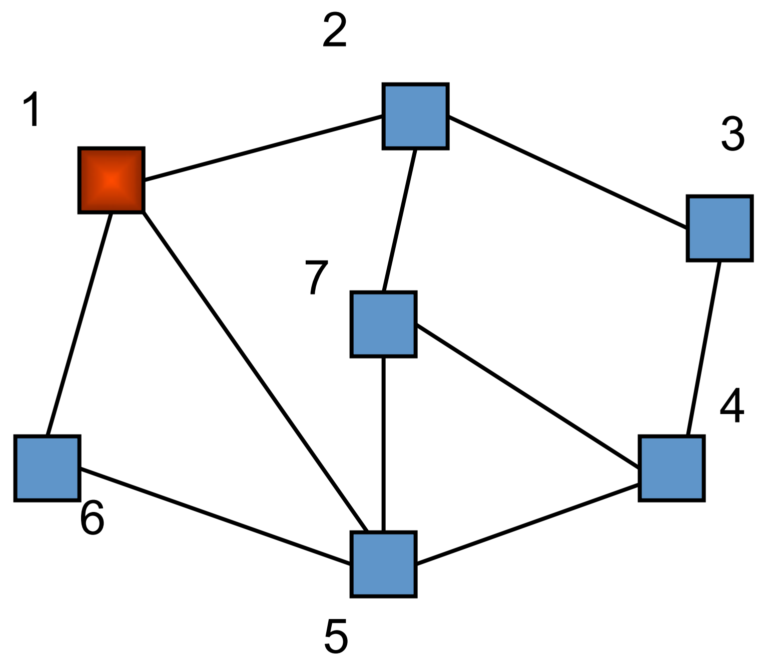
Spanning Tree Psuedocode

```
spanningTree():
    pick random vertex v.
    T := {}
    spanningTree(v, T)
    return T.

spanningTree(v, T):
    mark v as visited.
    for each neighbor v_i of v where there is an edge from v to v_i:
        if v_i is not visited
            add edge {v, v_i} to T.
            spanningTree(v_i, T)
    return T.
```

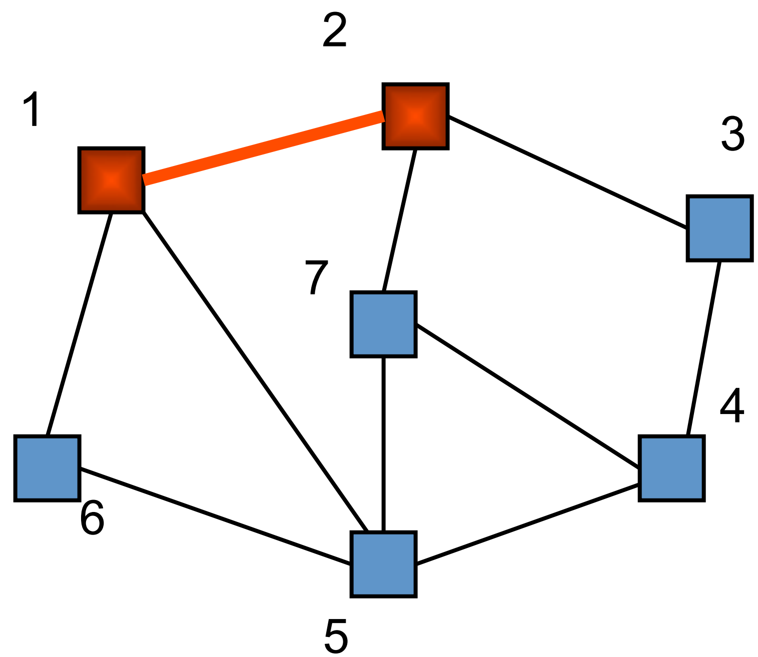
Example of Depth First Search

ST(1)



Example Step 2

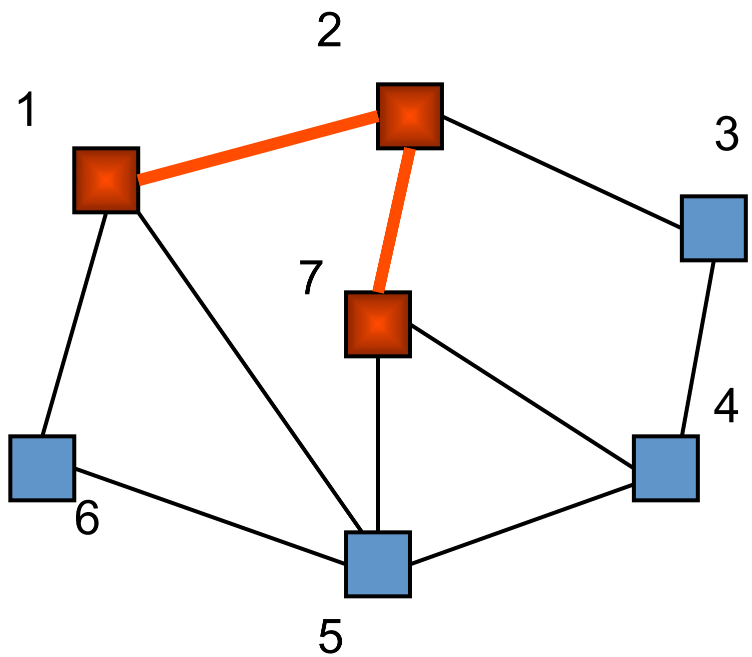
ST(1)
ST(2)



{1,2}

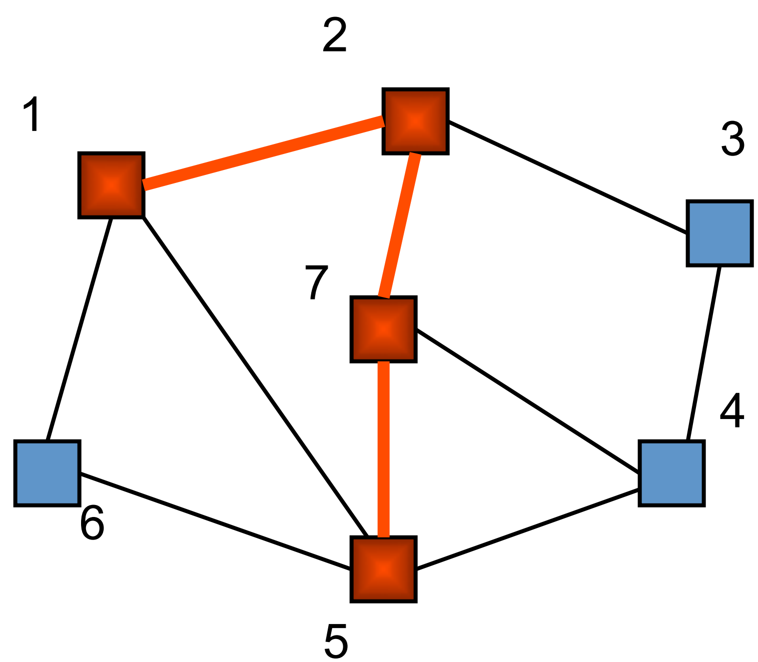
Example Step 3

ST(1)
ST(2)
ST(7)



{1,2} {2,7}

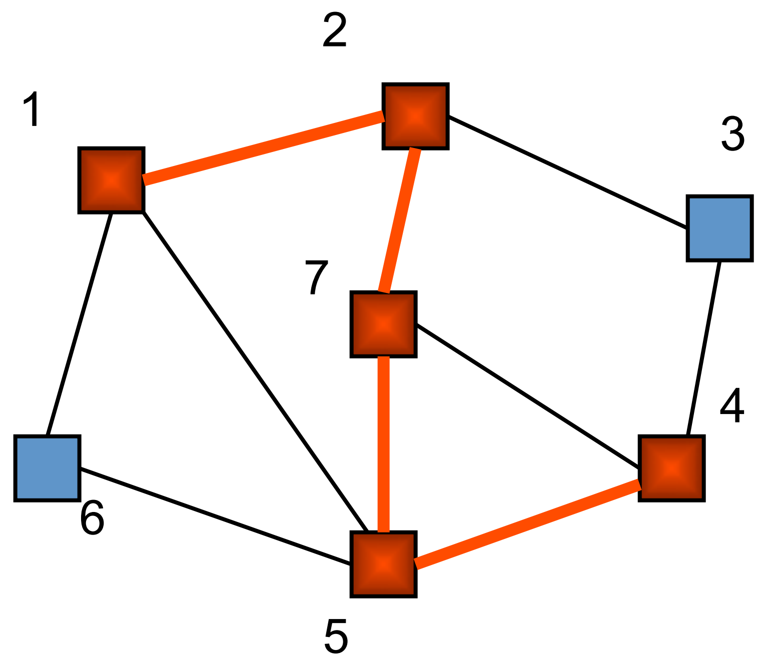
Example Step 4



ST(1)
ST(2)
ST(7)
ST(5)

{1,2} {2,7} {7,5}

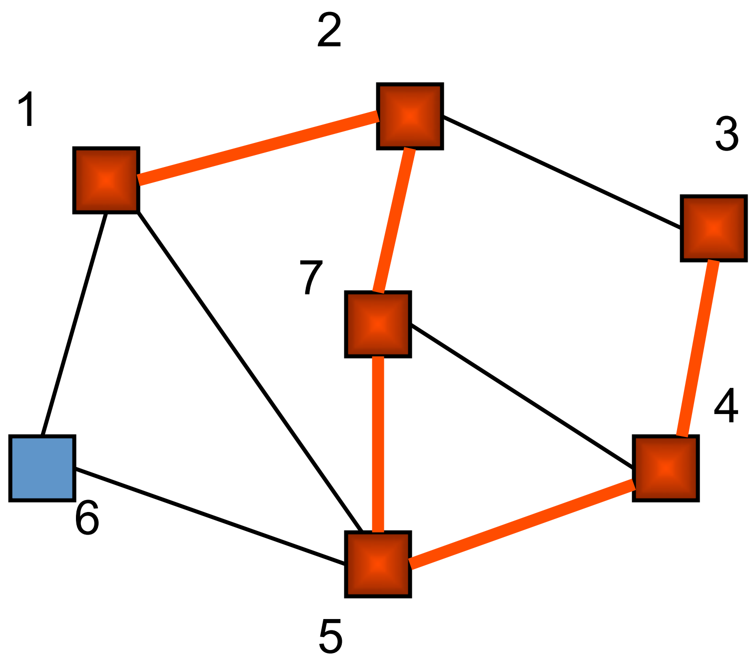
Example Step 5



ST(1)
ST(2)
ST(7)
ST(5)
ST(4)

{1,2} {2,7} {7,5} {5,4}

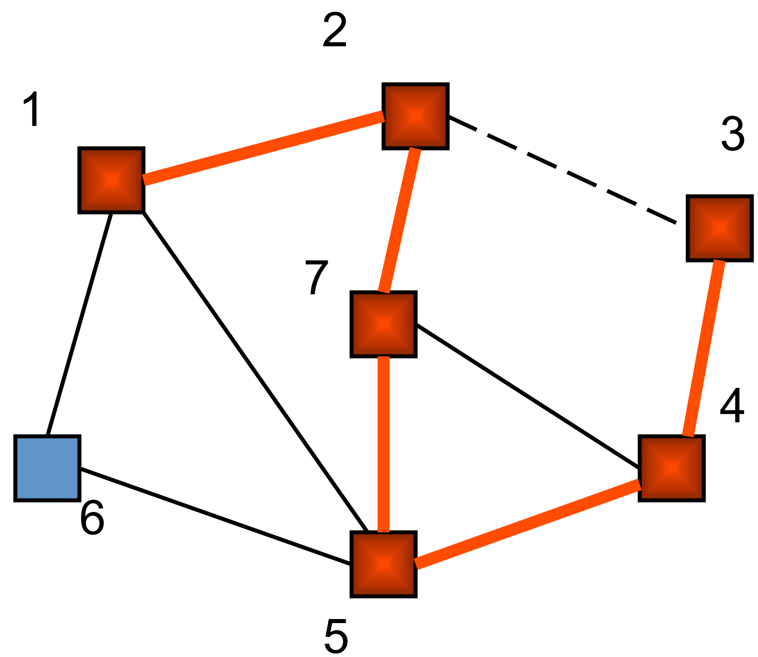
Example Step 6



ST(1)
ST(2)
ST(7)
ST(5)
ST(4)
ST(3)

{1,2} {2,7} {7,5} {5,4} {4,3}

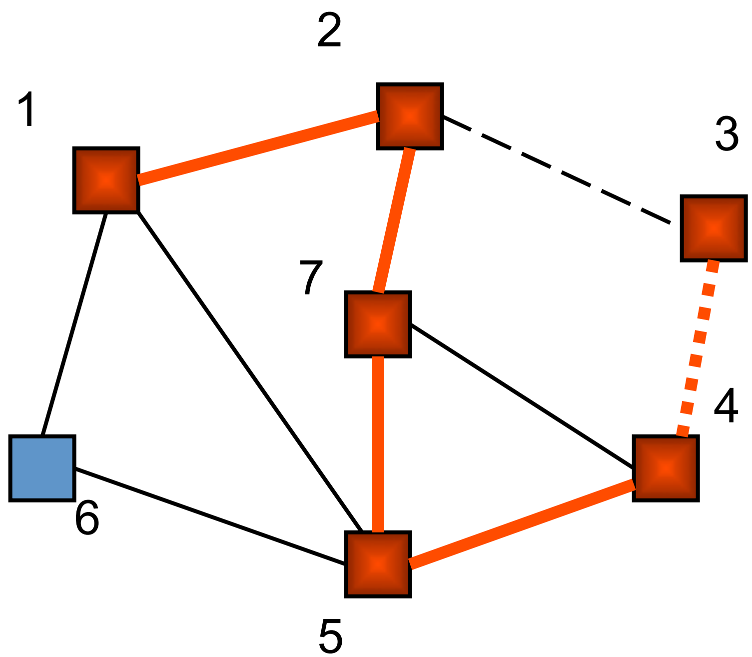
Example Step 7



ST(1)
ST(2)
ST(7)
ST(5)
ST(4)
ST(3)

{1,2} {2,7} {7,5} {5,4} {4,3}

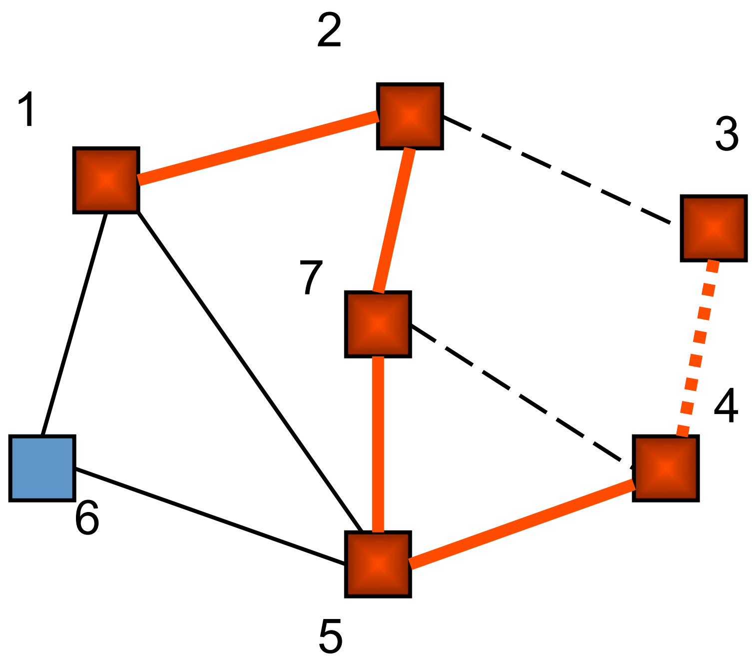
Example Step 8



ST(1)
ST(2)
ST(7)
ST(5)
ST(4)

{1,2} {2,7} {7,5} {5,4} {4,3}

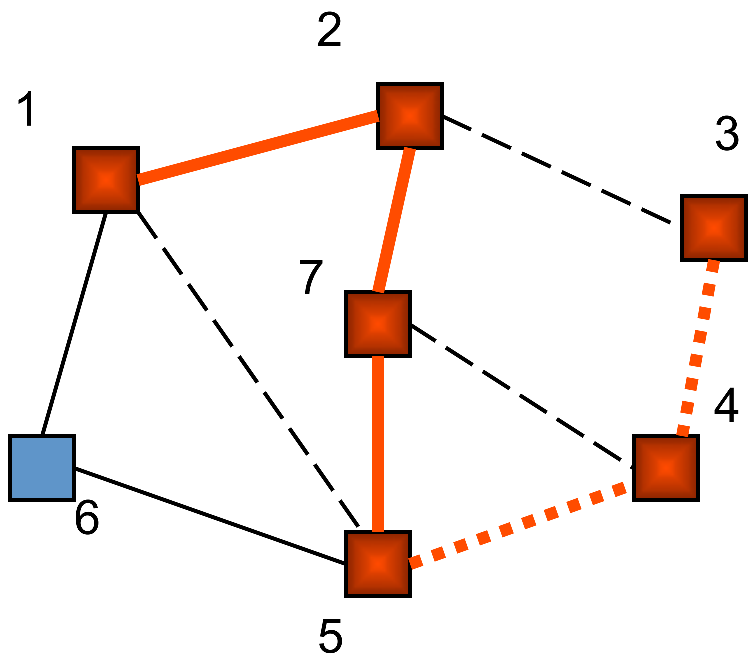
Example Step 9



ST(1)
ST(2)
ST(7)
ST(5)
ST(4)

{1,2} {2,7} {7,5} {5,4} {4,3}

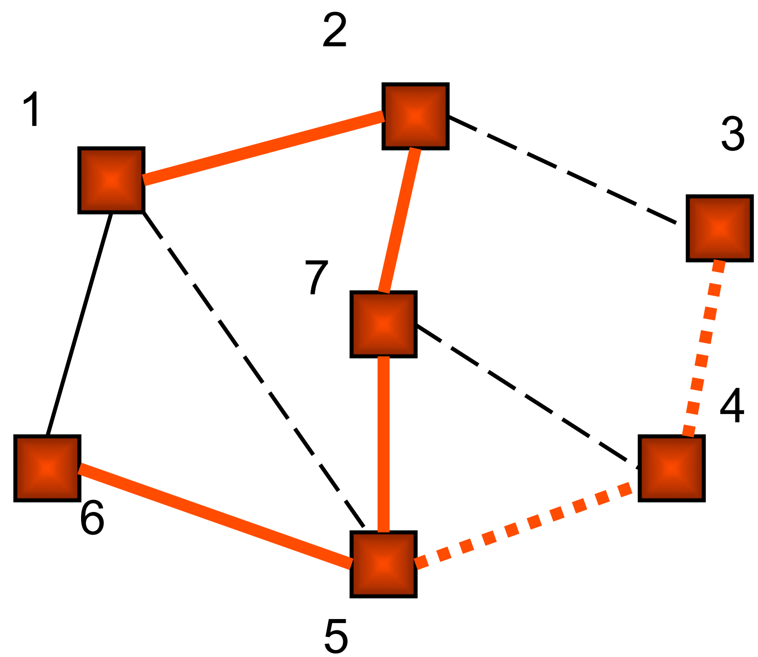
Example Step 10



ST(1)
ST(2)
ST(7)
ST(5)

{1,2} {2,7} {7,5} {5,4} {4,3}

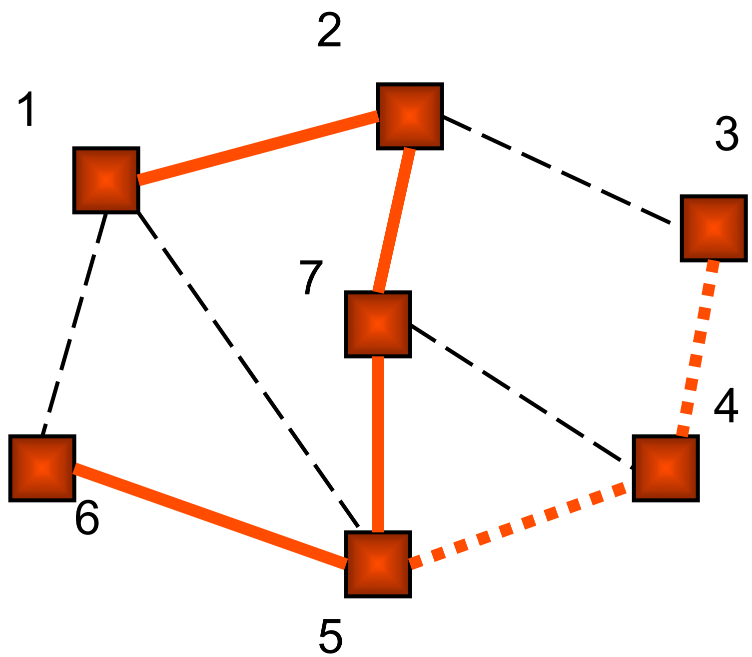
Example Step 11



ST(1)
ST(2)
ST(7)
ST(5)
ST(6)

{1,2} {2,7} {7,5} {5,4} {4,3} {5,6}

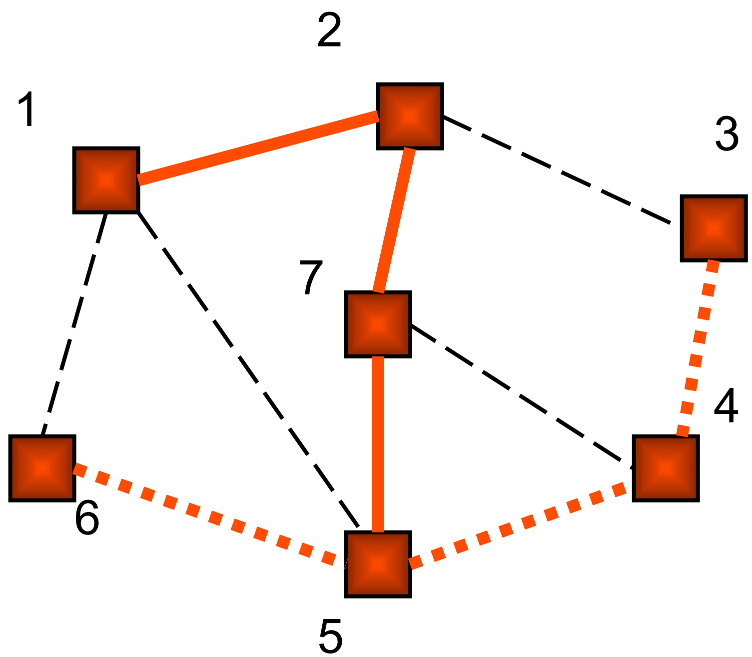
Example Step 12



ST(1)
ST(2)
ST(7)
ST(5)
ST(6)

{1,2} {2,7} {7,5} {5,4} {4,3} {5,6}

Example Step 13

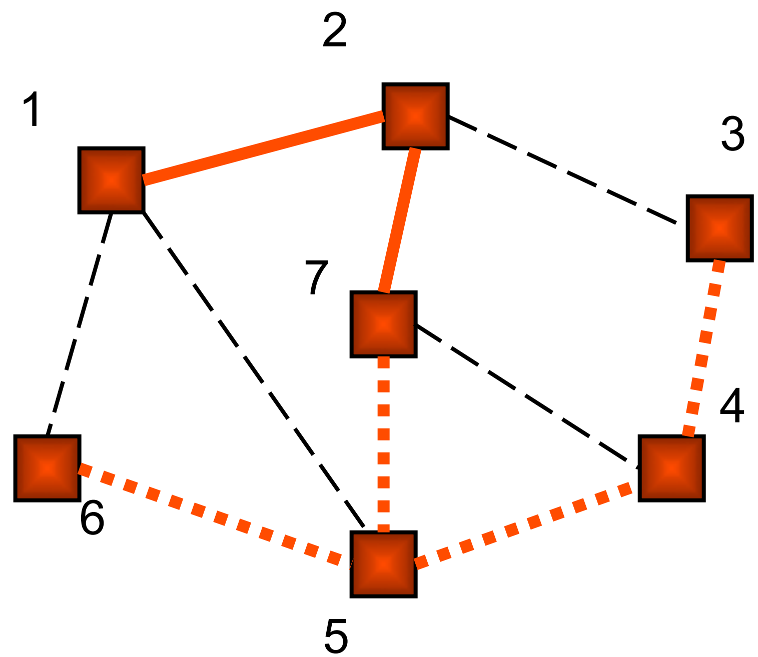


ST(1)
ST(2)
ST(7)
ST(5)

{1,2} {2,7} {7,5} {5,4} {4,3} {5,6}

Example Step 14

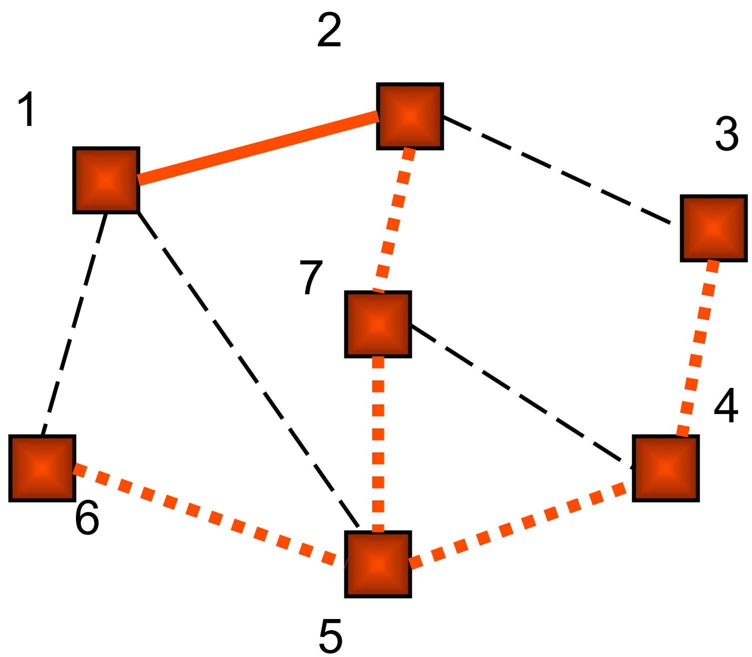
ST(1)
ST(2)
ST(7)



{1,2} {2,7} {7,5} {5,4} {4,3} {5,6}

Example Step 15

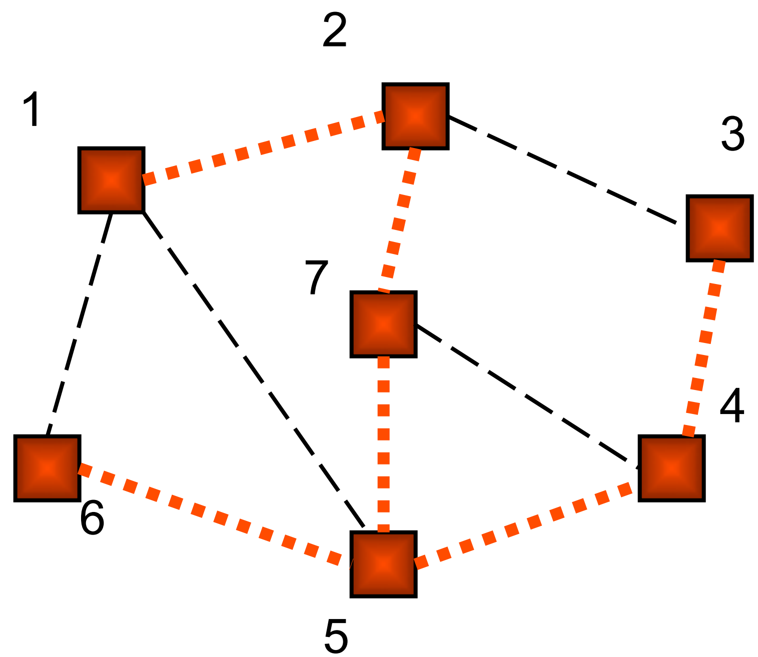
ST(1)
ST(2)



{1,2} {2,7} {7,5} {5,4} {4,3} {5,6}

Example Step 16

ST(1)



{1,2} {2,7} {7,5} {5,4} {4,3} {5,6}

Minimum Spanning Tree Problem

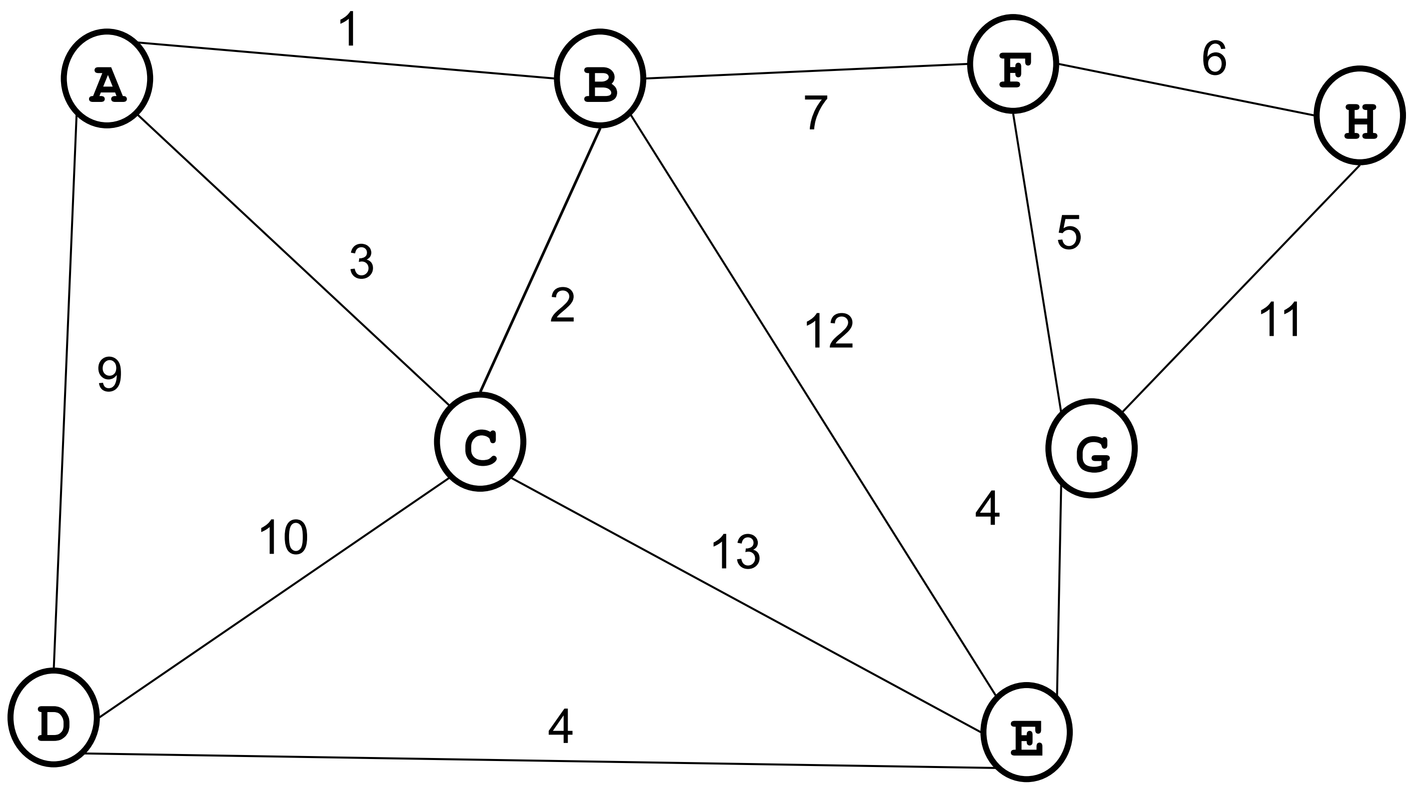
- Input: Undirected Graph $G = (V, E)$ and a cost function C from E to non-negative real numbers. $C(e)$ is the cost of edge e.
- Output: A spanning tree T with minimum total cost. That is: T that minimizes

$$C(T) = \sum_{e \in T} C(e)$$

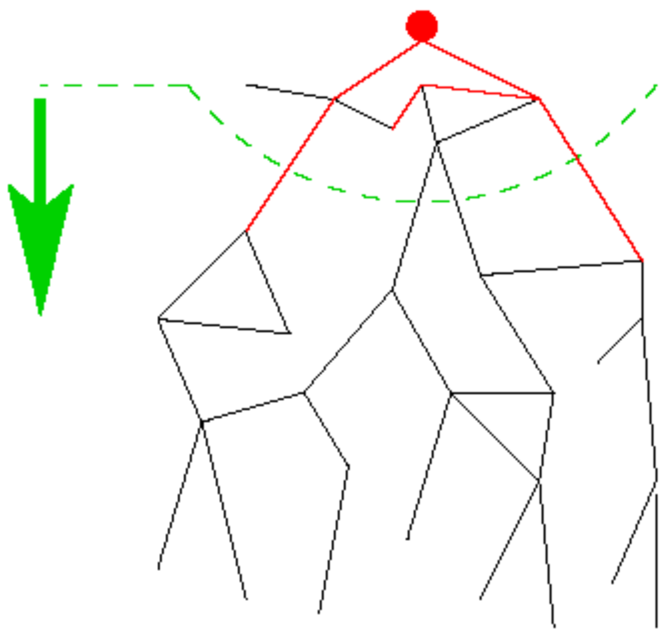
Observations about Spanning Trees

- For any spanning tree T, inserting an edge e_{new} not in T creates a cycle
- But
  - Removing any edge e_{old} from the cycle gives back a spanning tree
  - If e_{new} has a lower cost than e_{old} we have progressed!

Find the MST

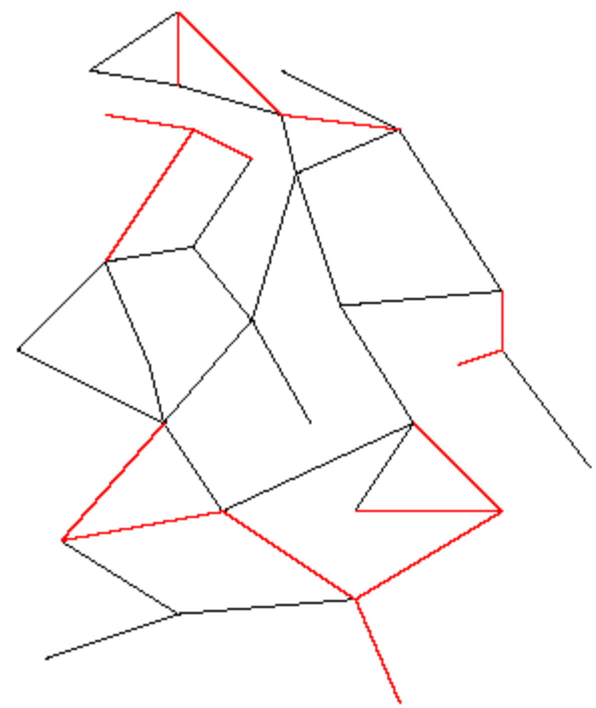


Two Different Approaches



Prim's Algorithm

Looks familiar!

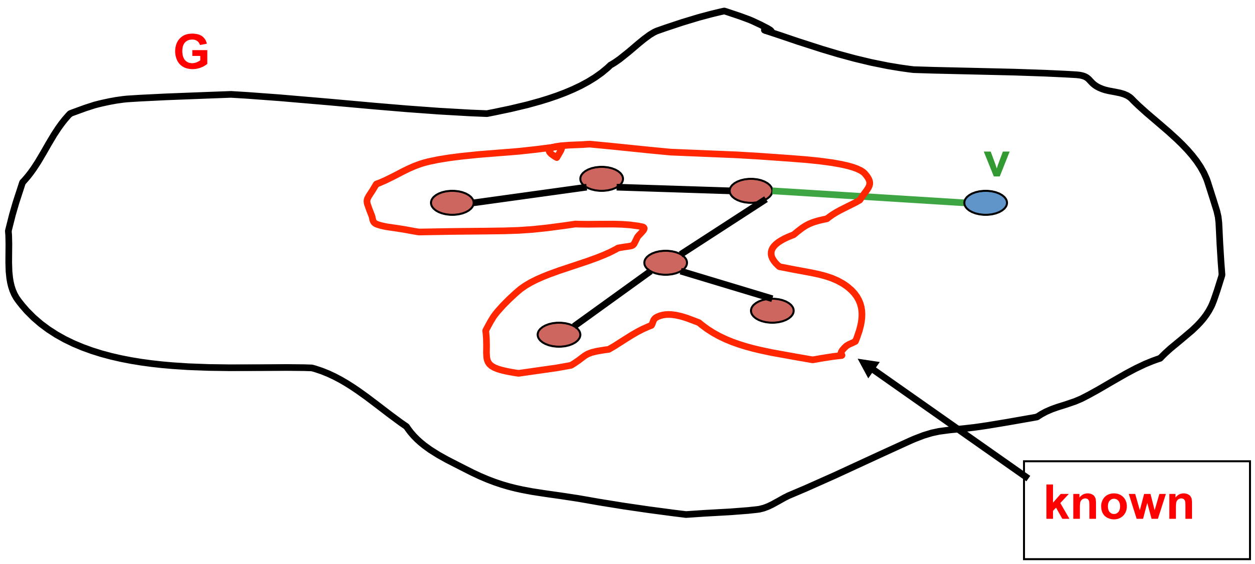


Kruskals's Algorithm

Completely different!

Prim's algorithm

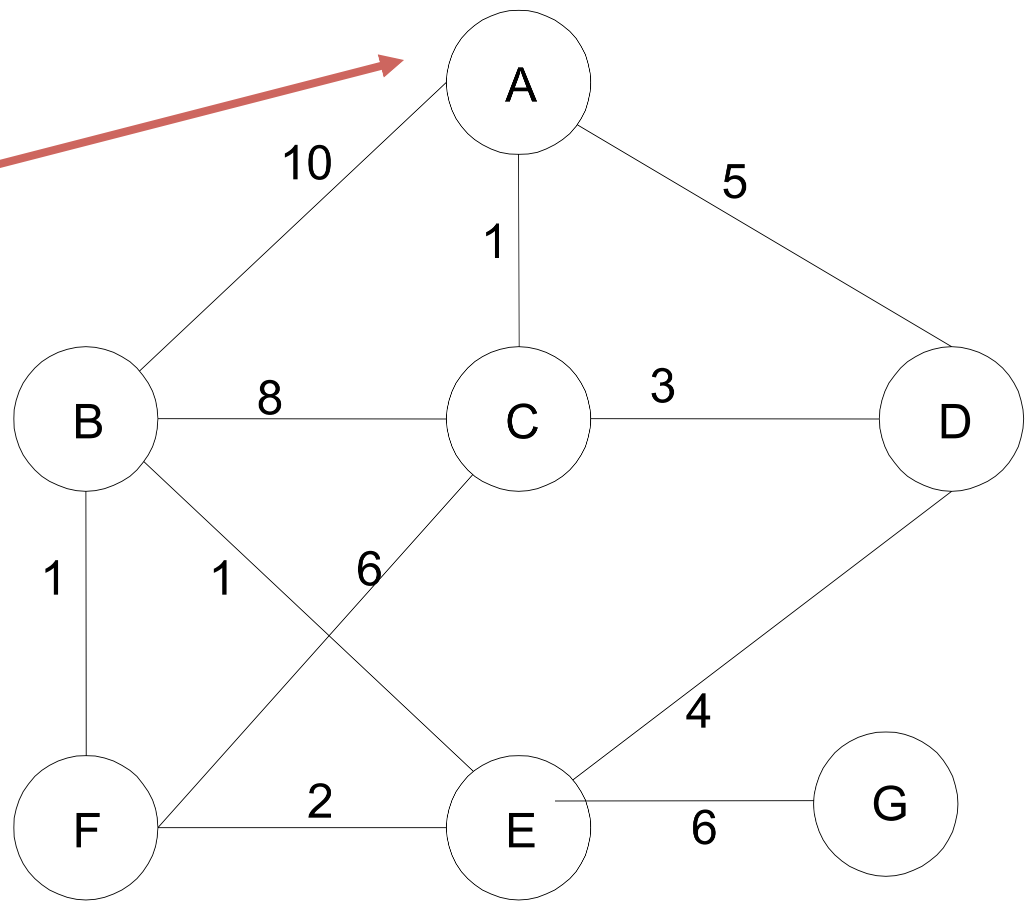
Idea: Grow a tree by adding an edge from the "known" vertices to the "unknown" vertices. Pick the edge with the smallest weight.



Prim's algorithm

Starting from empty *T*, choose a vertex at random and initialize

V = {A}, *T* ={}

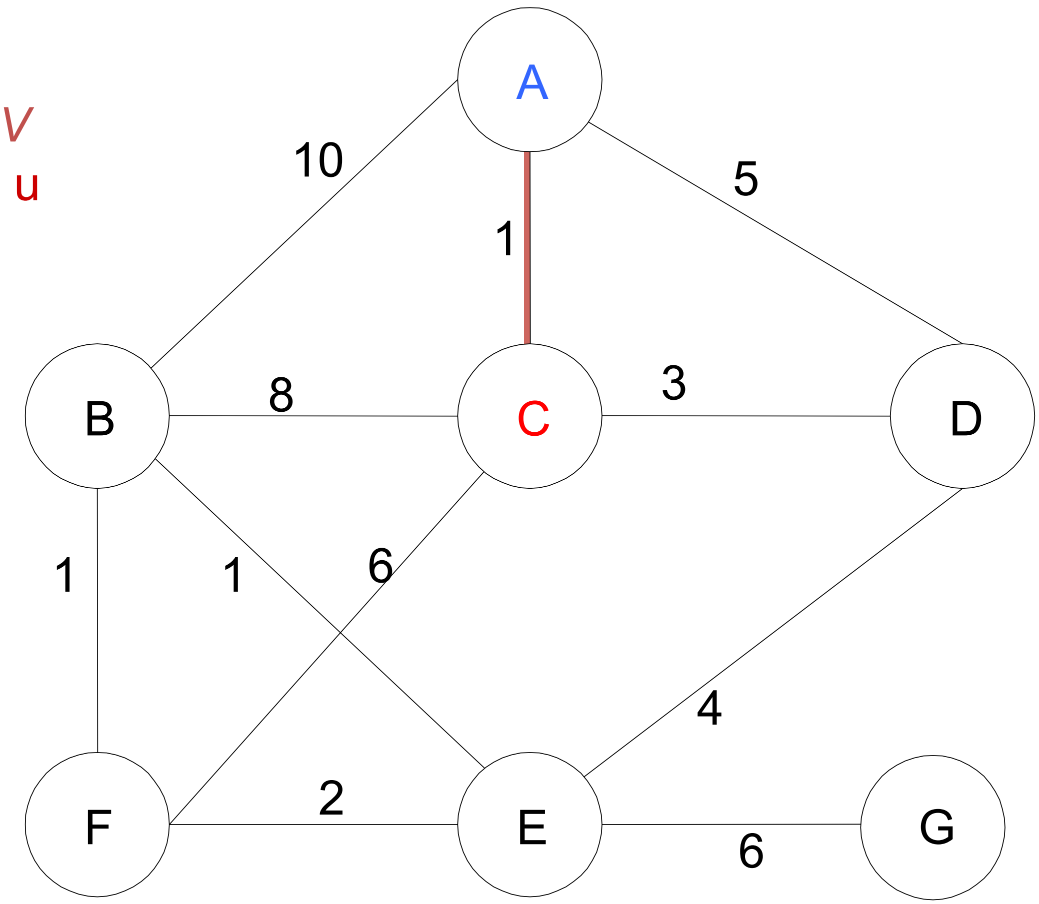


Prim's algorithm

Choose the vertex u not in *V* such that edge weight from u to a vertex in *V* is minimal (greedy!)

V = {A,C}

T = { (A,C) }

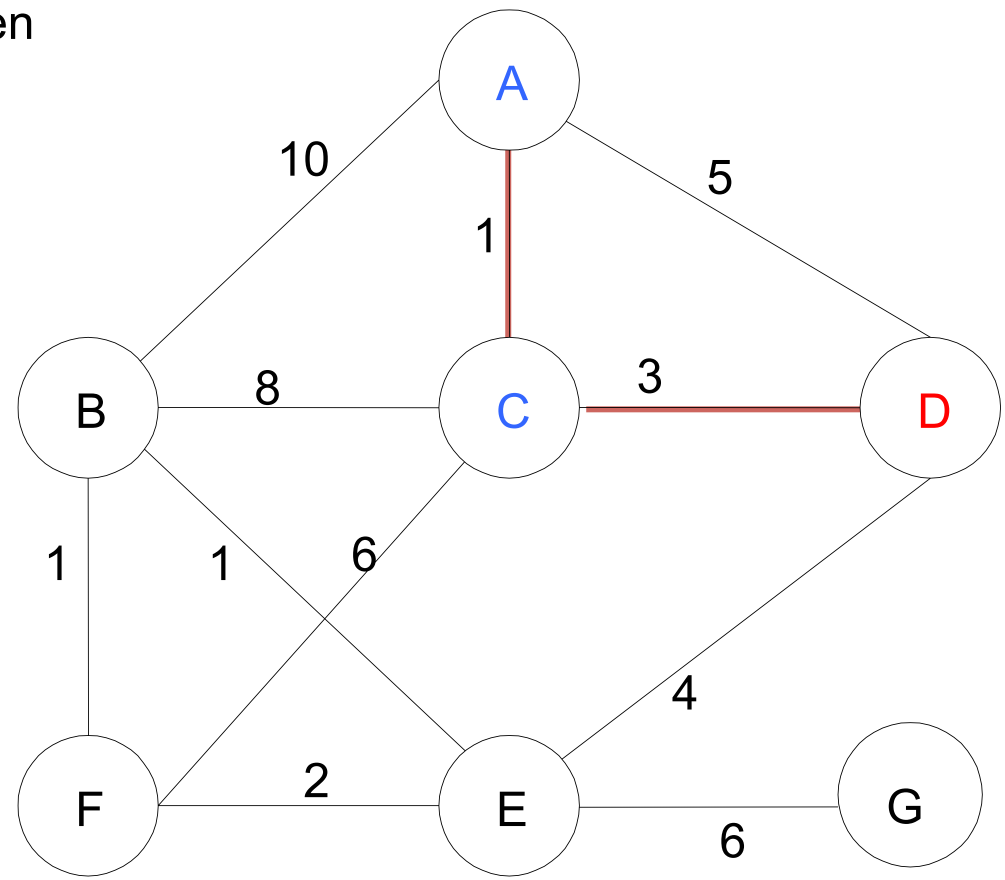


Prim's algorithm

Repeat until all vertices have been chosen

V = {A,C,D}

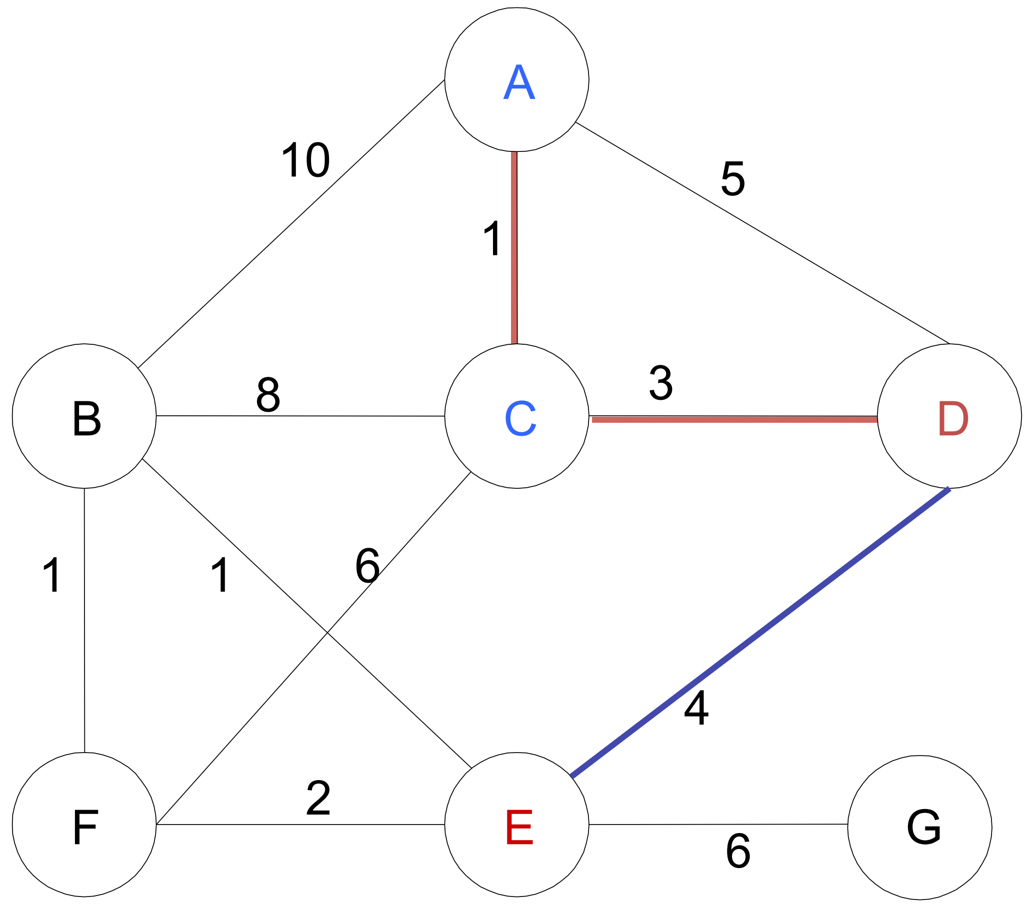
T= { (A,C), (C,D)}



Prim's algorithm

V = {A,C,D,E}

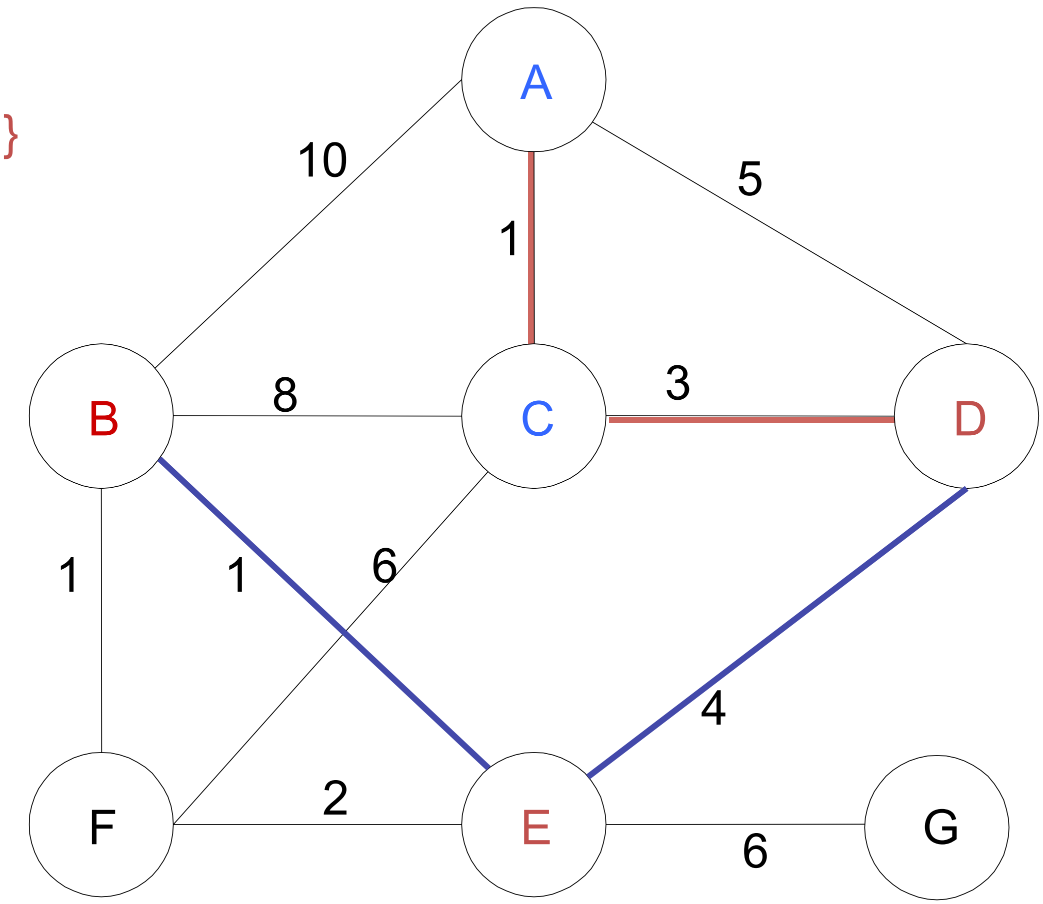
T = { (A,C), (C,D), (D,E) }



Prim's algorithm

V = {A,C,D,E,B}

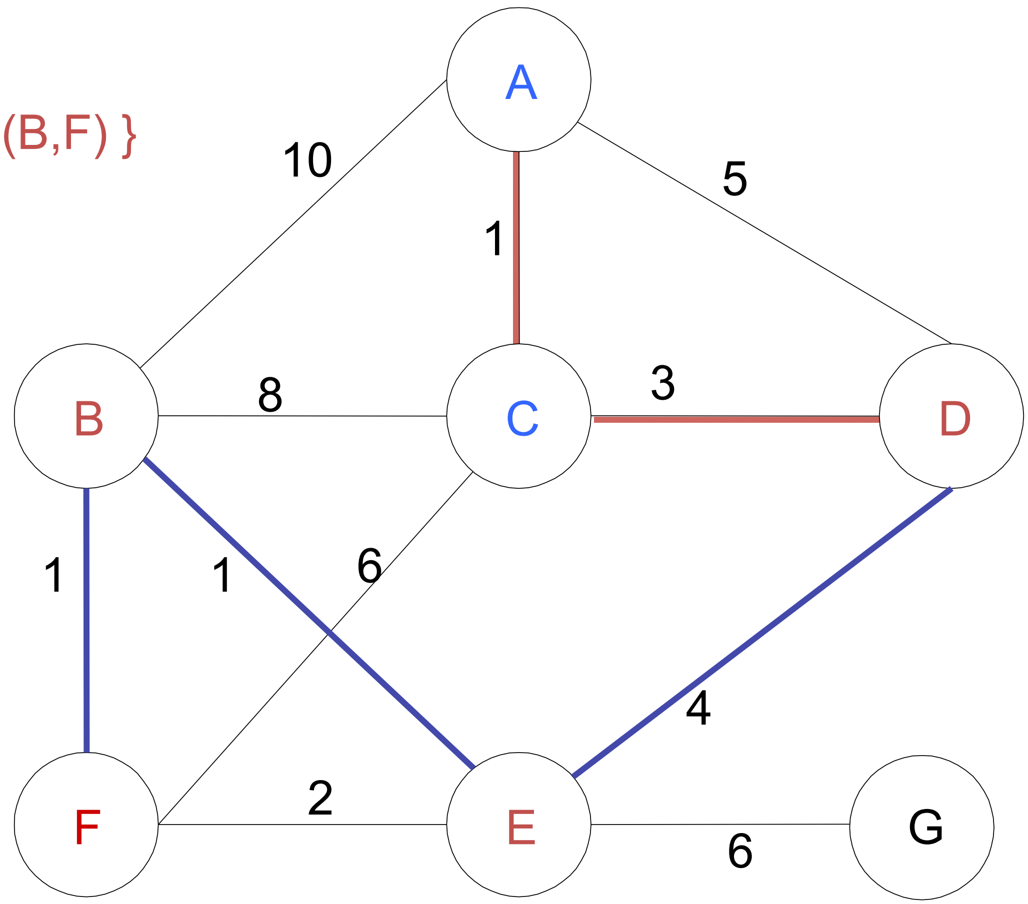
T = { (A,C), (C,D), (D,E), (E,B) }



Prim's algorithm

V = {A,C,D,E,B,F}

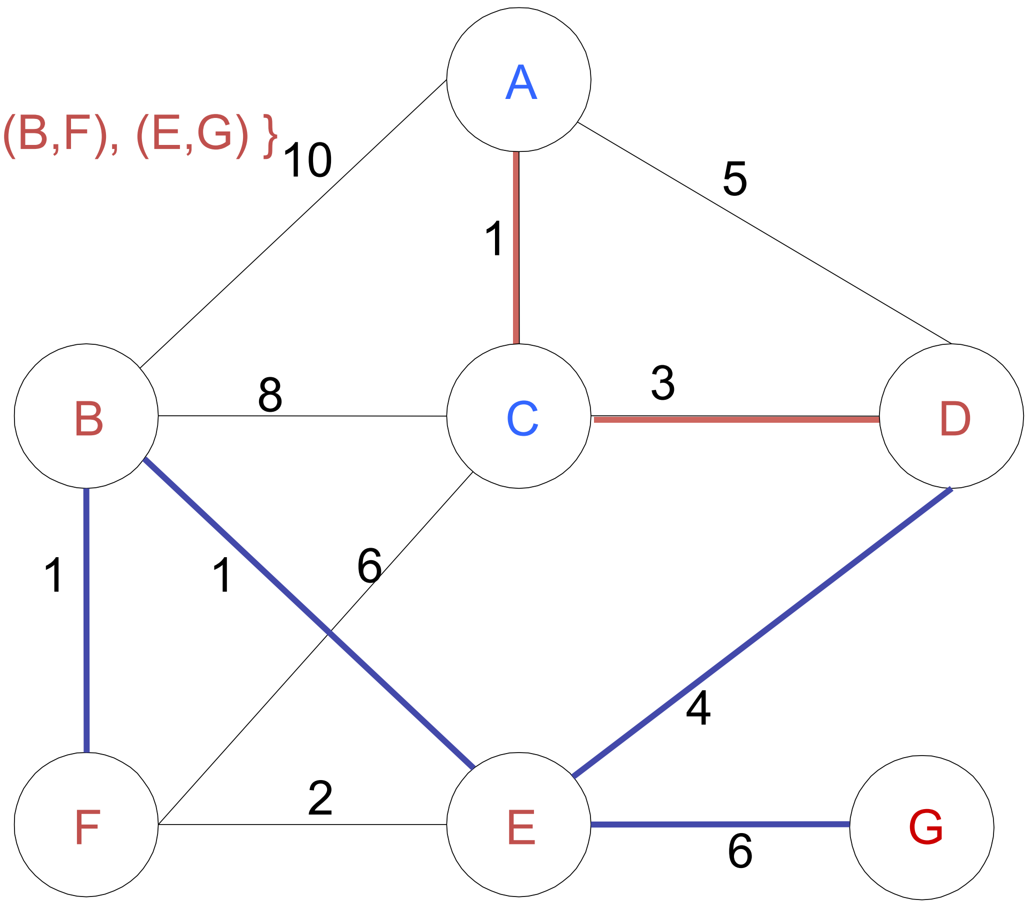
T = { (A,C), (C,D), (D,E), (E,B), (B,F) }



Prim’s algorithm

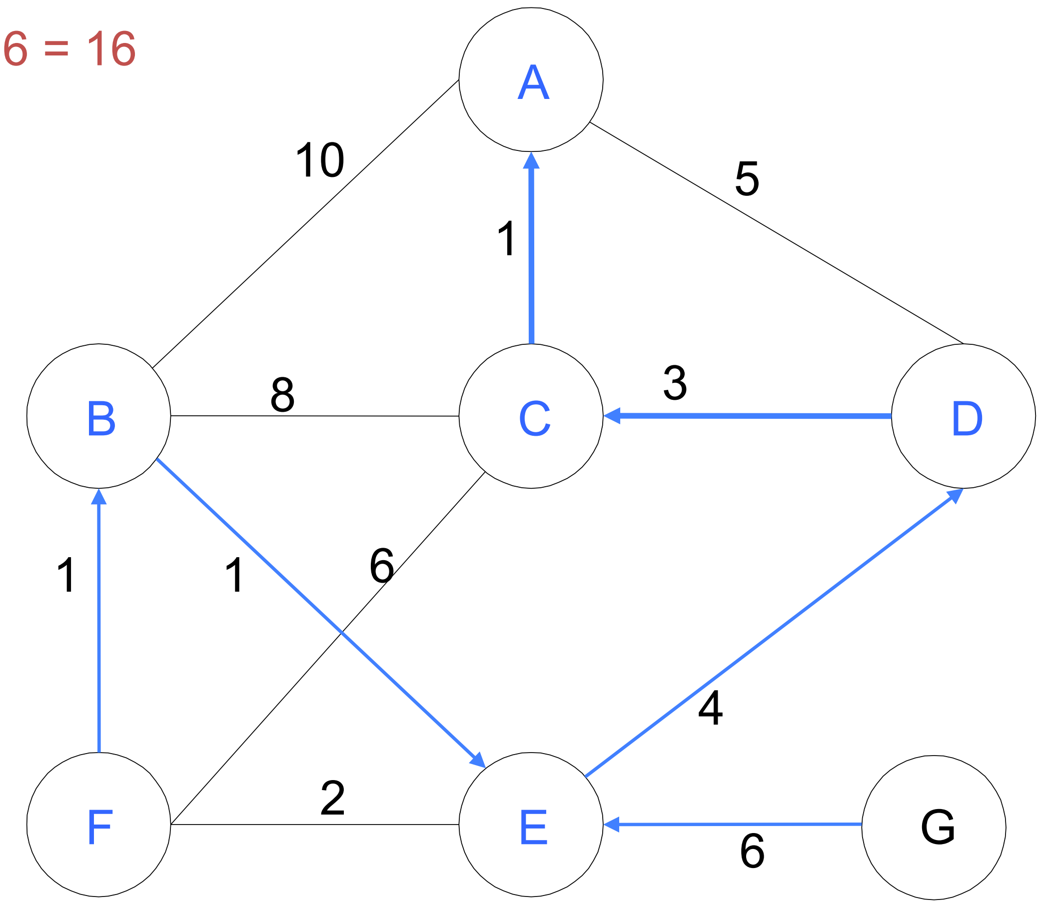
V = {A,C,D,E,B,F,G}

T = { (A,C), (C,D), (D,E), (E,B), (B,F), (E,G) }



Prim's algorithm

Final Cost: 1 + 3 + 4 + 1 + 1 + 6 = 16



Prim's Algorithm Implementation

```
Prim():
  for each vertex v:                    // Initialization
      v's distance := infinity.
      v's previous := none.
      mark v as unknown.
  choose random node v1.
  v1's distance := 0.
  List := {all vertices}.
  T := {}.

  while List is not empty:
      v := remove List vertex with minimum distance.
      add edge {v, v's previous} to T.
      mark v as known.
      for each unknown neighbor n of v:
         if distance(v, n) is smaller than n's distance:
            n's distance := distance(v, n).
            n's previous := v.

  return T.
```

Prim’s algorithm Analysis

- How is it different from Djikstra's algorithm?

- If the step that removes unknown vertex with minimum distance is done with binary heap the running time is:

  $O(|E| \log |V|)$