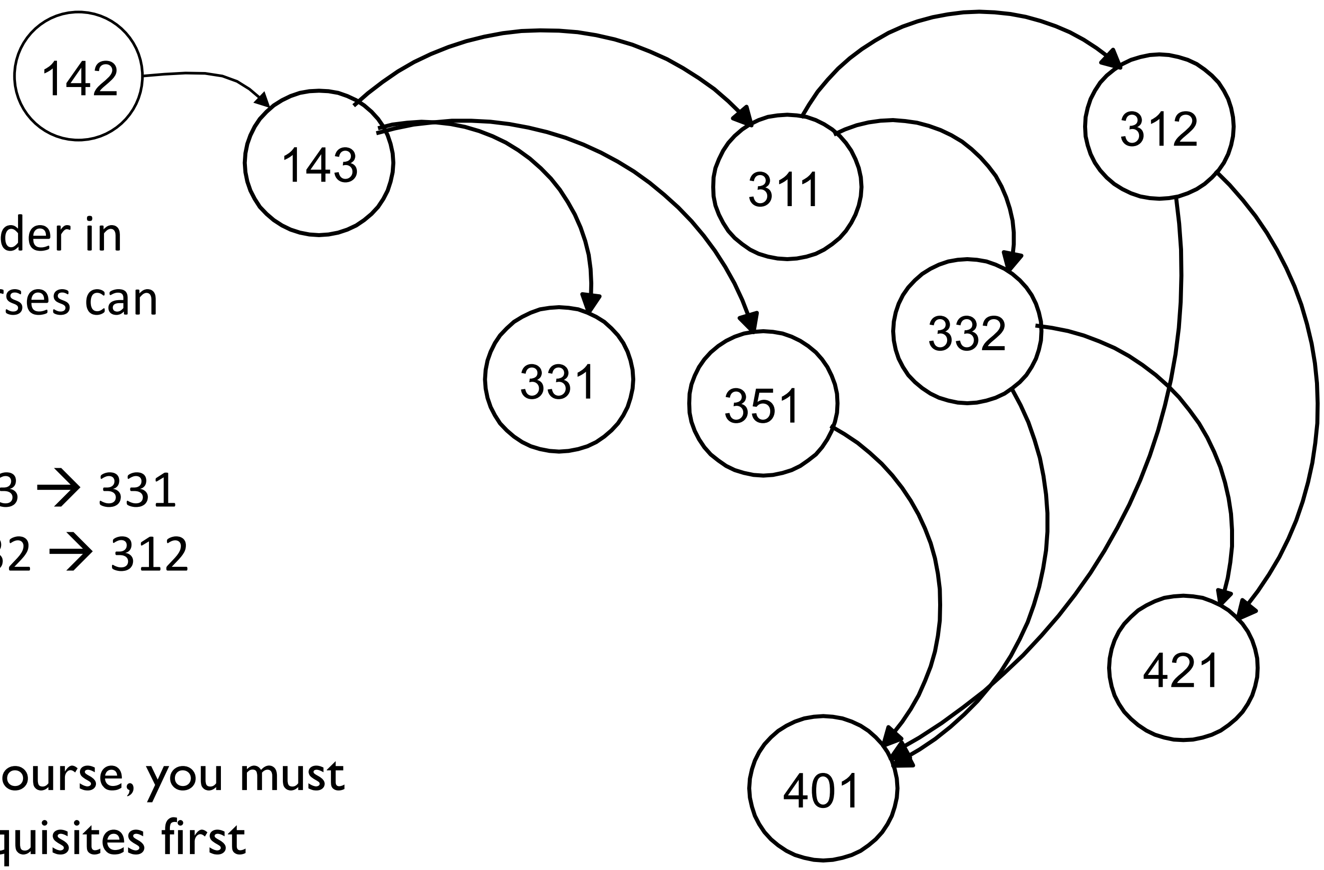
CSE 373
Data Structures and Algorithms

Lecture 23: Graphs V

Topological Sort



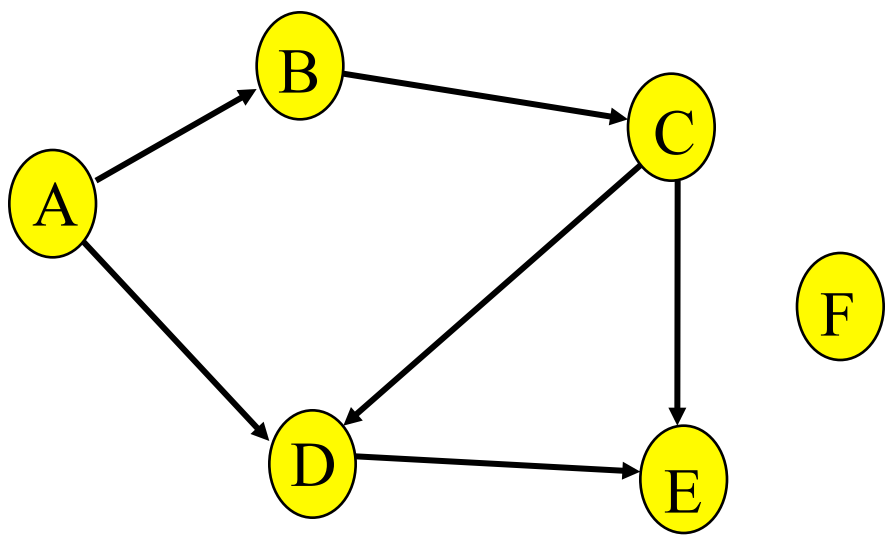
Problem: Find an order in which all these courses can be taken.

Example: 142 → 143 → 331 → 351 → 311 → 332 → 312 → 421 → 401

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

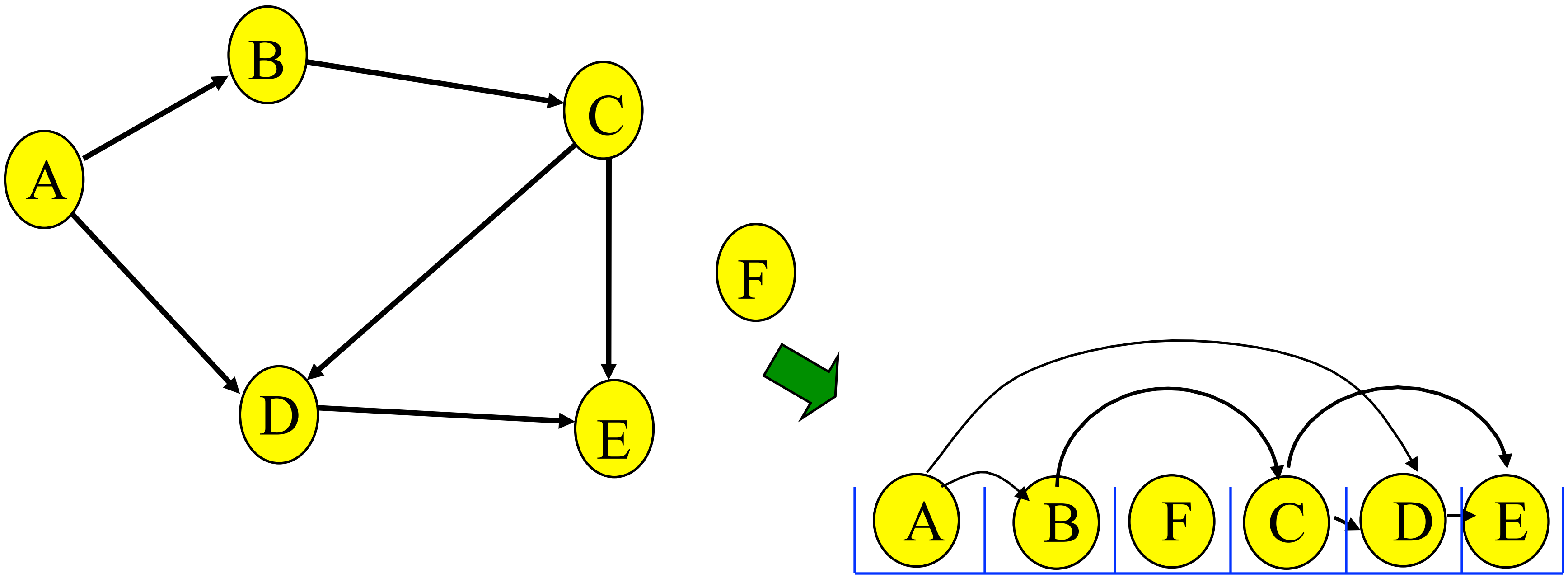
Topological Sort

- In $G = (V, E)$, find total ordering of vertices such that for any edge (v, w), v precedes w in the ordering



Topological Sort: Good Example

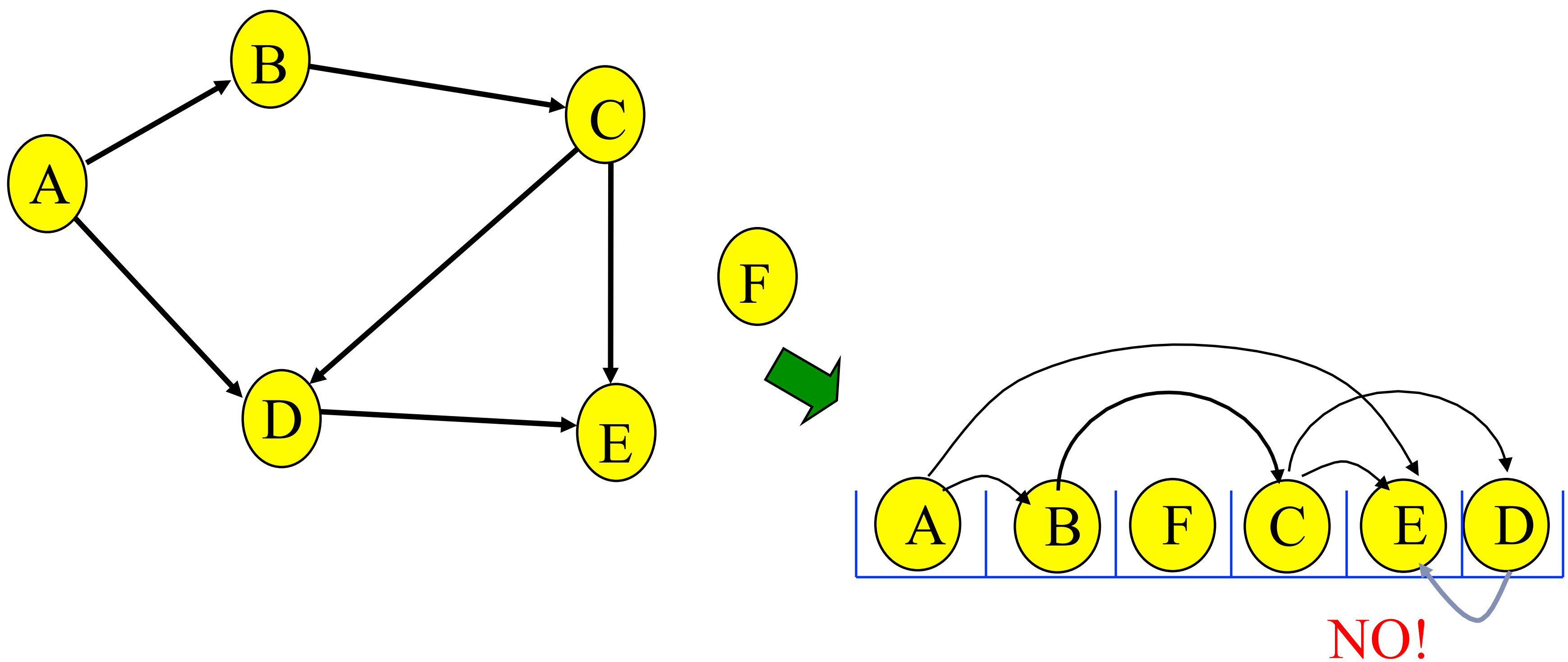
- Any total ordering in which all the arrows go to the right is a valid solution



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

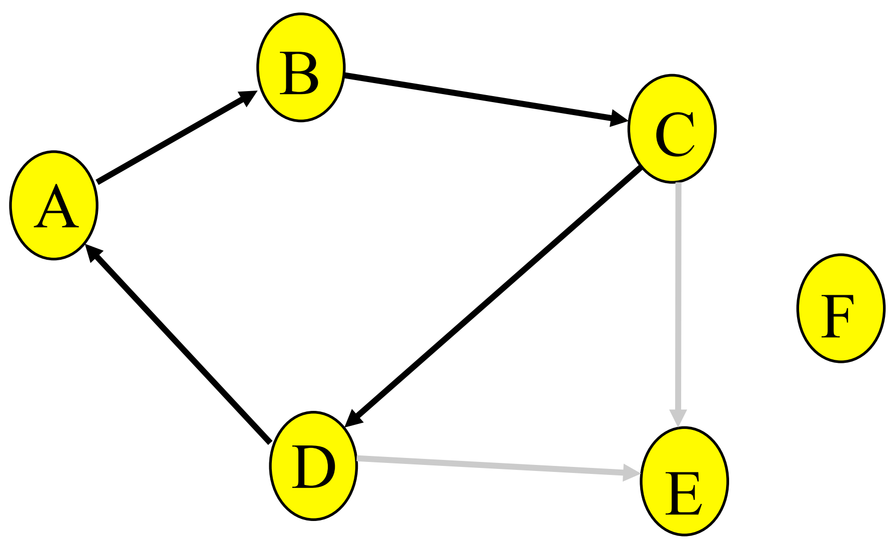
Topological Sort: Bad Example

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



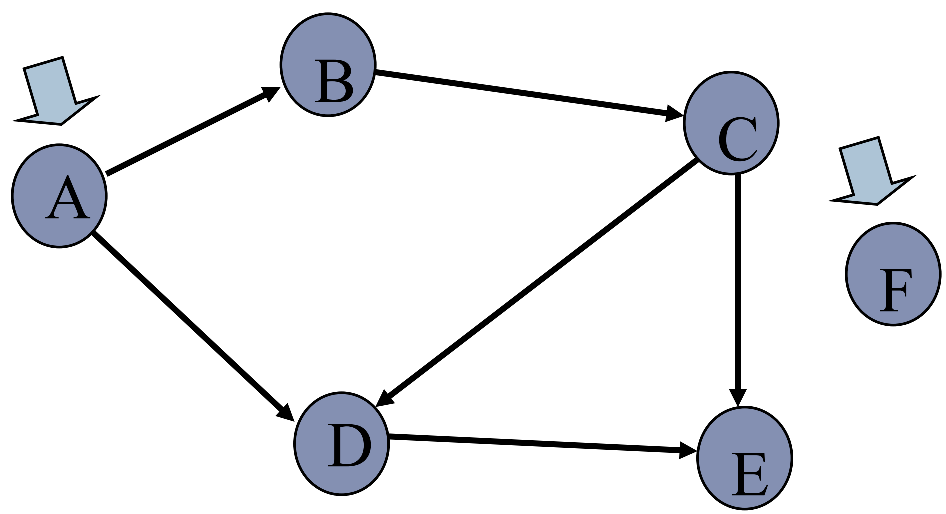
Only acyclic graphs can be topo sorted

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



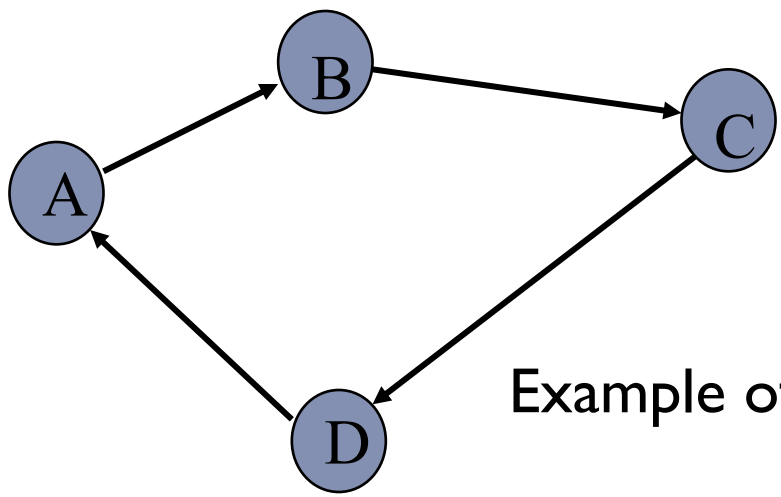
Topological Sort Algorithm: Step 1

- Step 1: Identify vertices that have no incoming edges
  - The "in-degree" of these vertices is zero



Topological Sort Algorithm: Step 1a

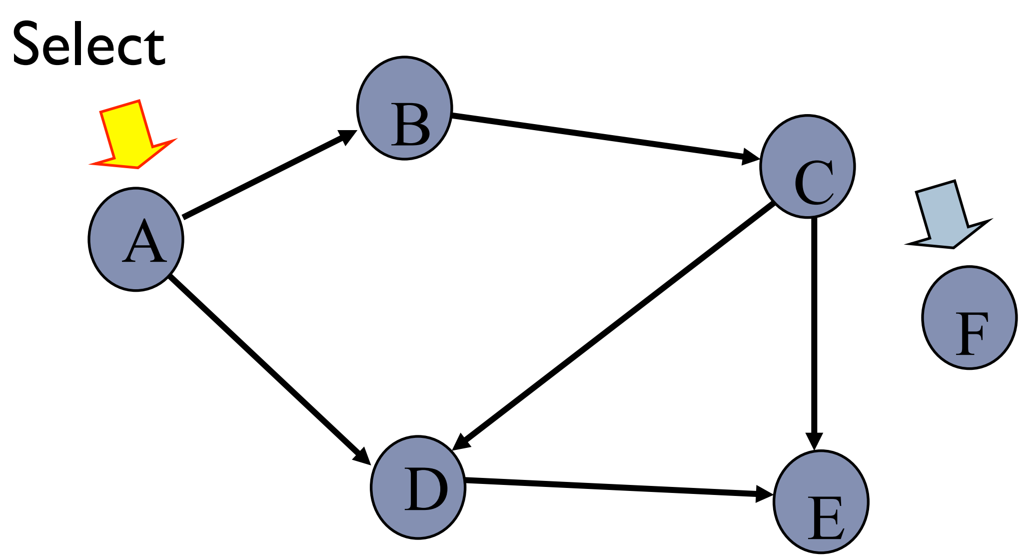
- <u>Step 1</u>: Identify vertices that have no incoming edges
  - If no such vertices, graph has cycle(s)
  - Topological sort not possible – Halt.



Example of a cyclic graph

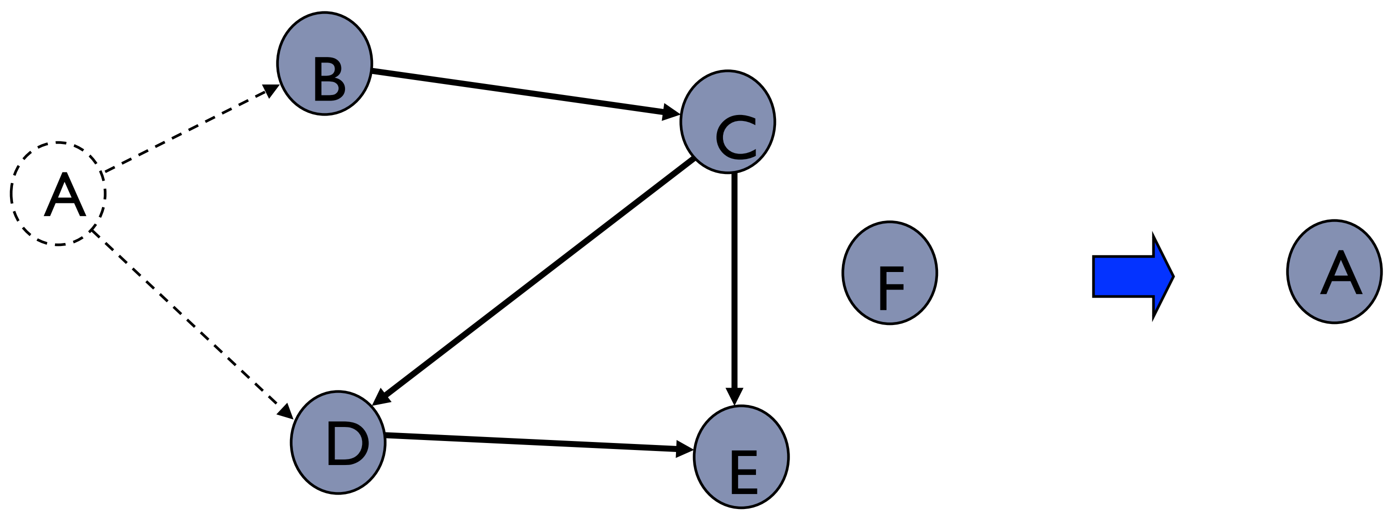
Topological Sort Algorithm: Step 1b

- Step 1: Identify vertices that have no incoming edges
  - Select one such vertex



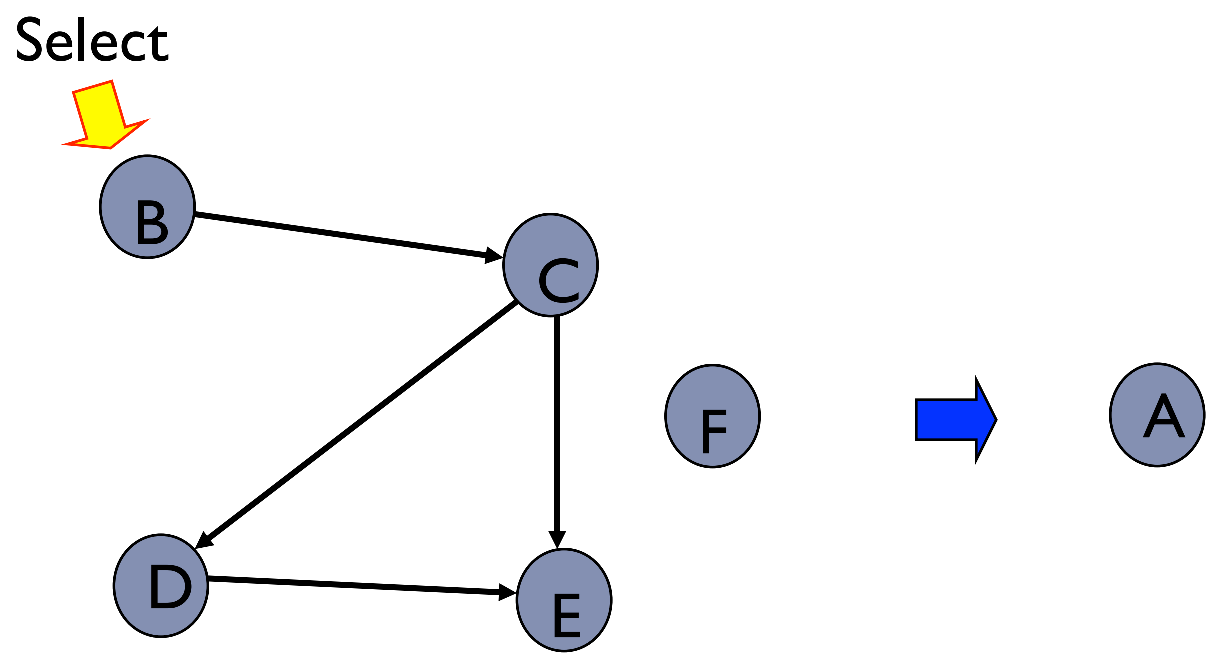
Topological Sort Algorithm: Step 2

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



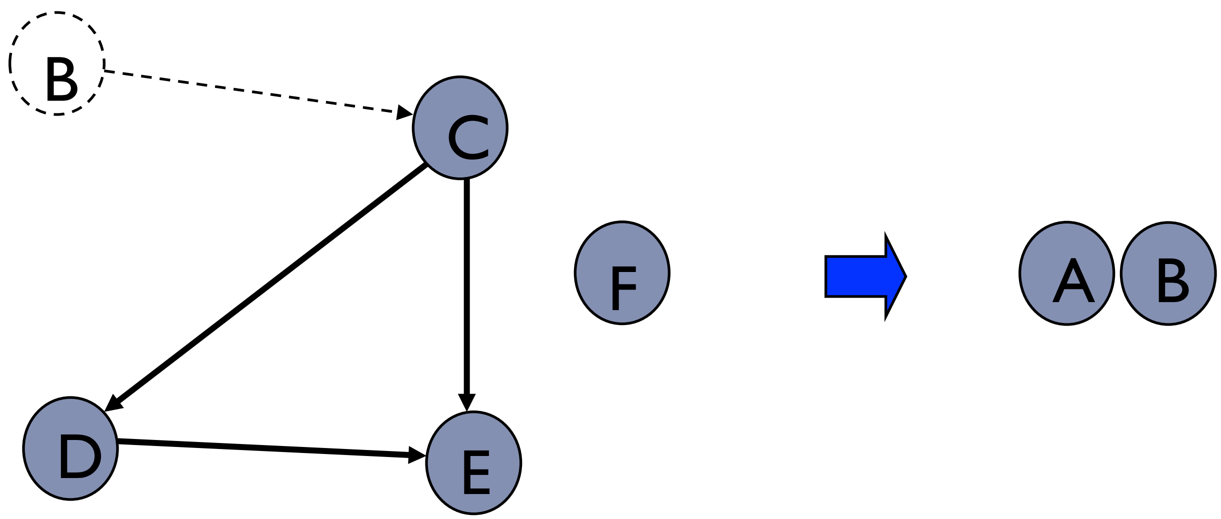
Topological Sort Algorithm: Repeat

- Repeat <u>Step 1</u> and <u>Step 2</u> until graph is empty



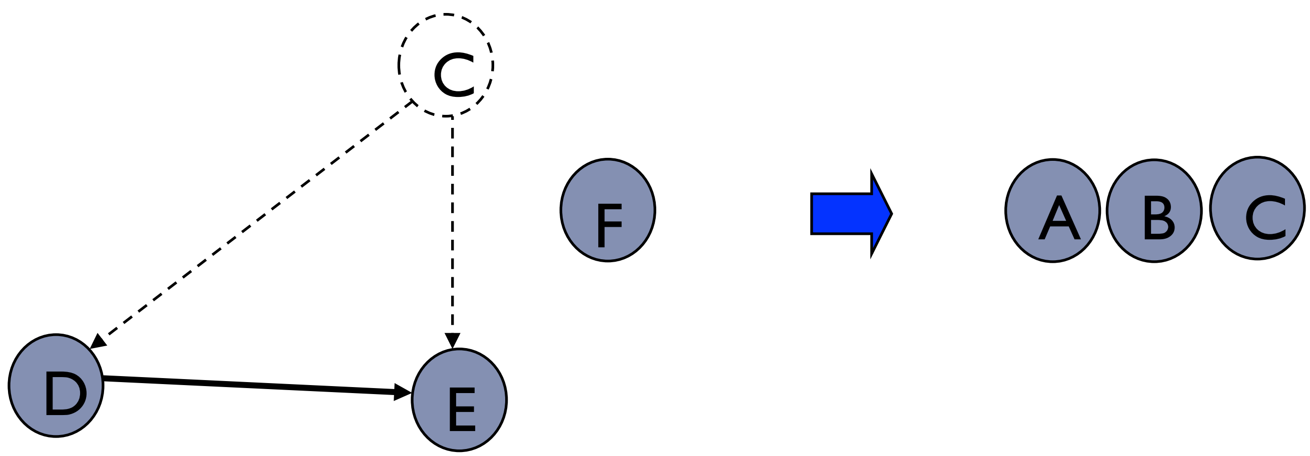
B

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



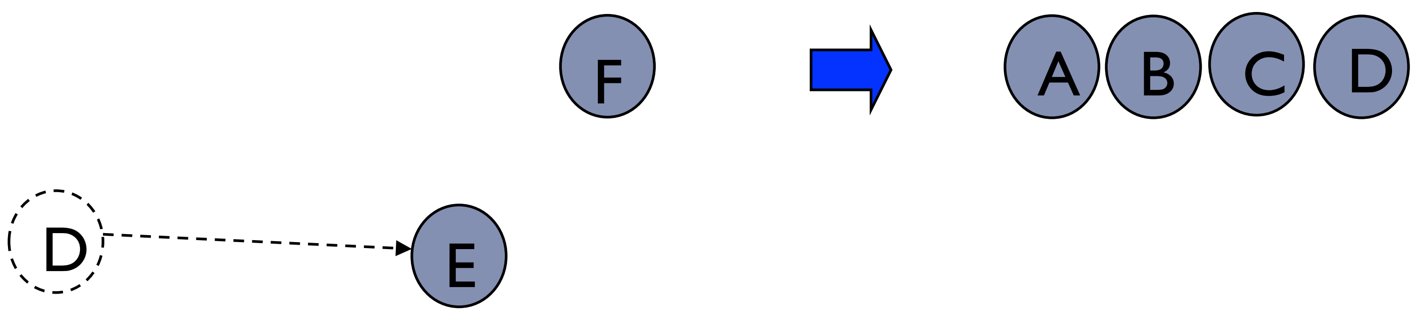
C

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



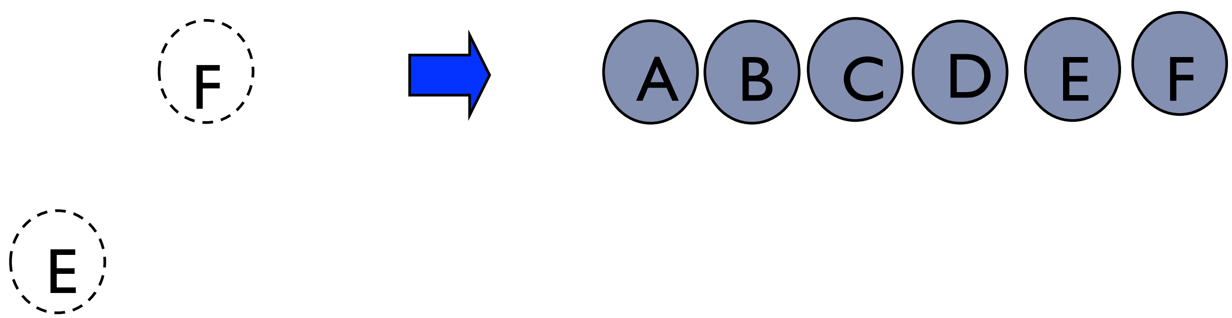
D

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

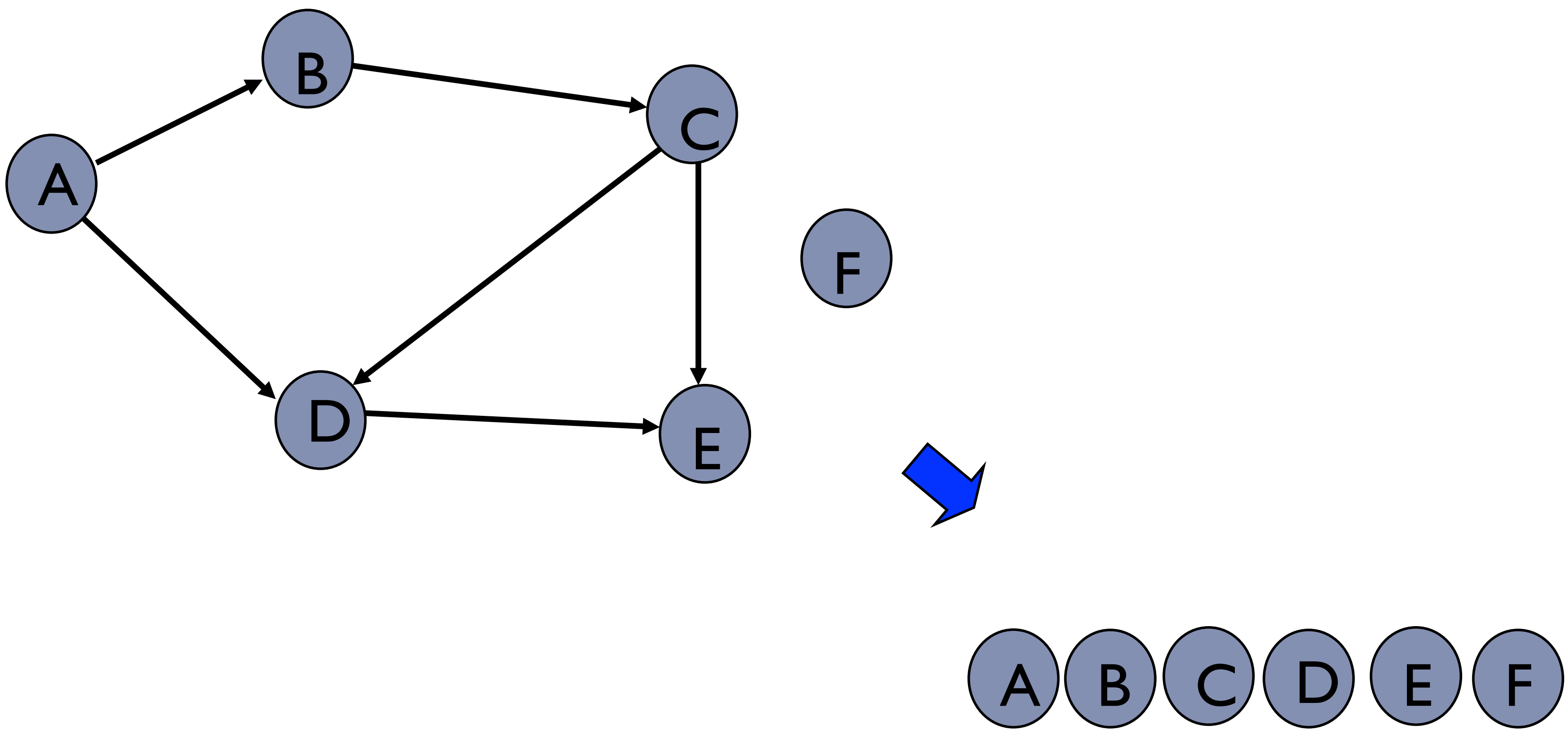


E, F

- 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

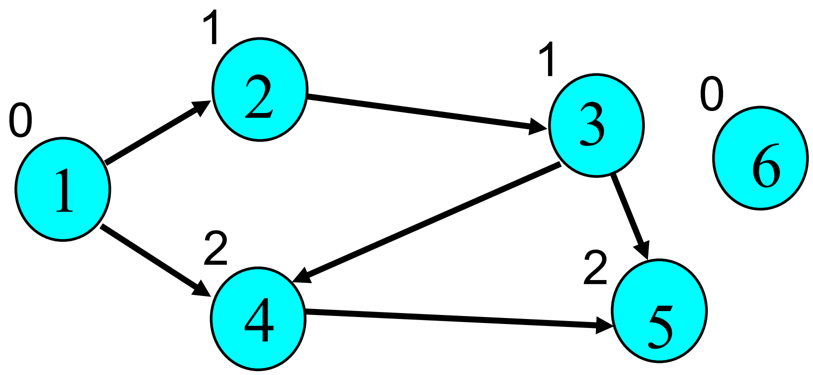
1. Store each vertex's In-Degree in an hash table D
2. Initialize queue with all "in-degree=0" vertices
3. While there are vertices remaining in the queue:
   a) Dequeue and output a vertex
   b) Reduce In-Degree of all vertices adjacent to it by 1
   c) Enqueue any of these vertices whose In-Degree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

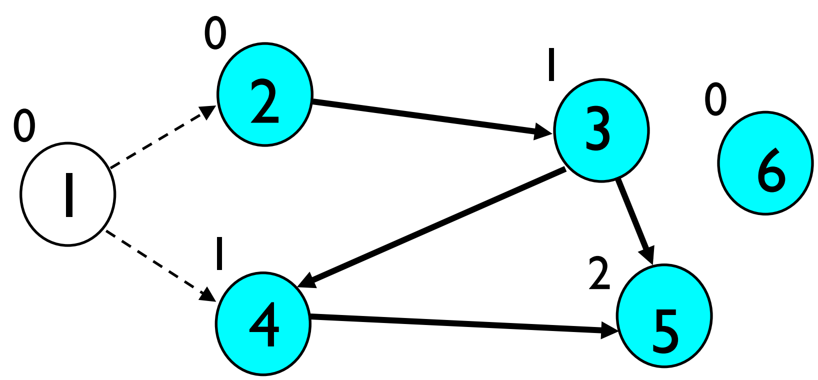
Queue (before):
Queue (after): 1, 6



Answer:

Topological Sort with Queue

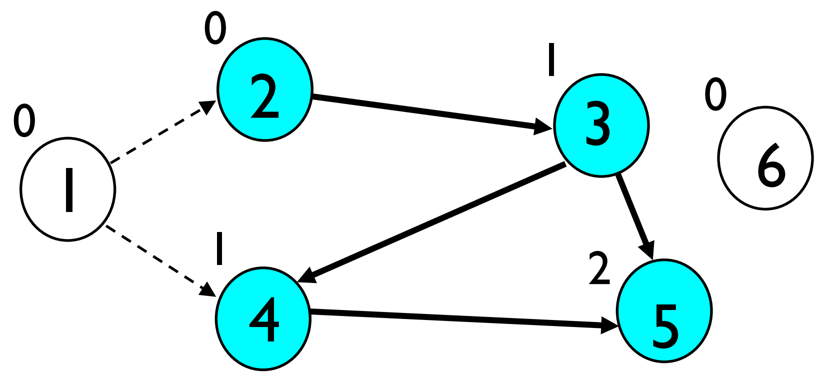
Queue (before): 1, 6
Queue (after): 6, 2



Answer: 1

Topological Sort with Queue

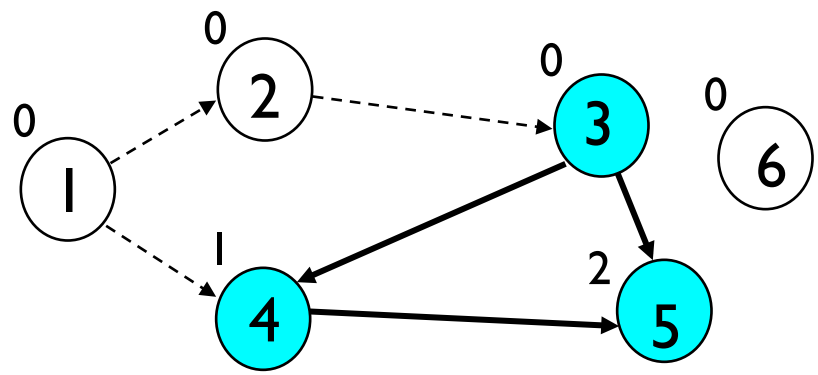
Queue (before): 6, 2
Queue (after): 2



Answer: 1, 6

Topological Sort with Queue

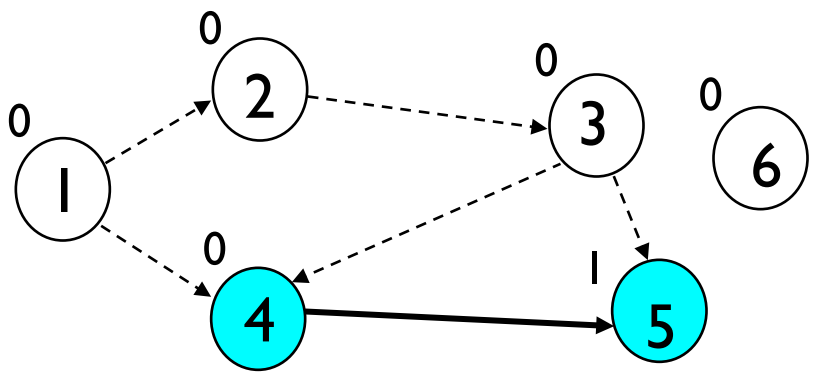
Queue (before): 2
Queue (after): 3



Answer: 1, 6, 2

Topological Sort with Queue

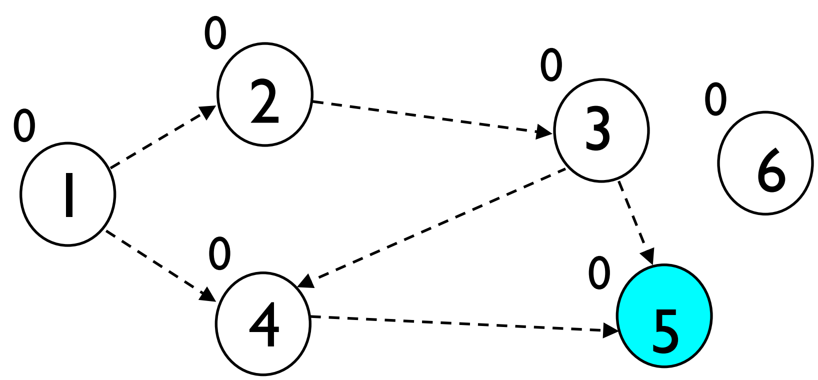
Queue (before): 3
Queue (after): 4



Answer: 1, 6, 2, 3

Topological Sort with Queue

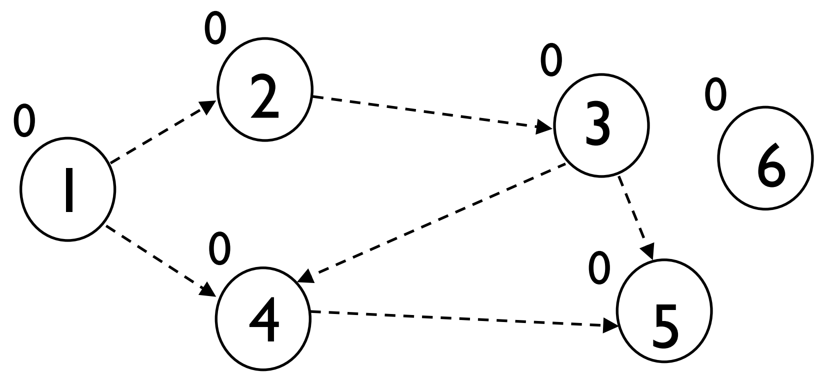
Queue (before): 4
Queue (after): 5



Answer: 1, 6, 2, 3, 4

Topological Sort with Queue

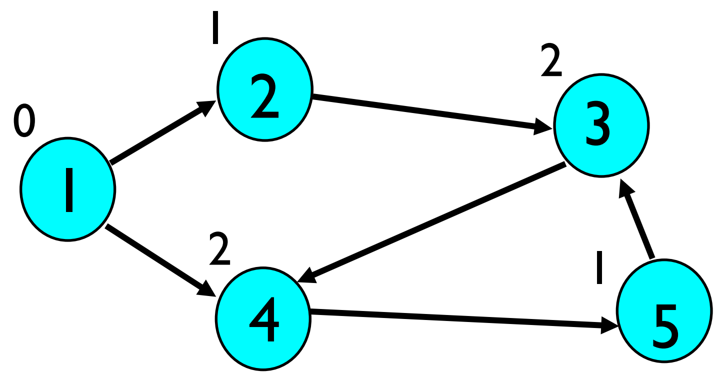
Queue (before): 5
Queue (after):



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

Topological Sort Fails (cycle)

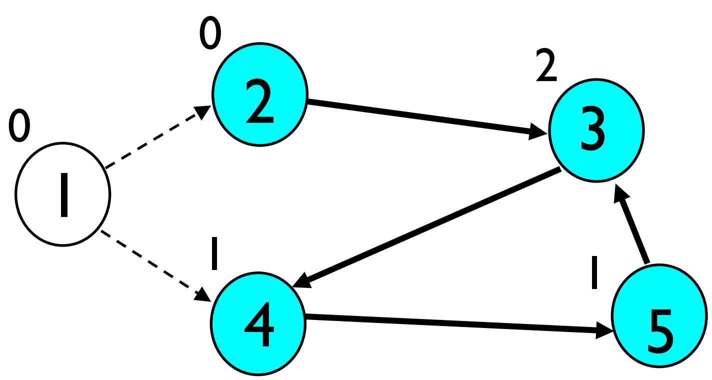
Queue (before):
Queue (after): 1



Answer:

Topological Sort Fails (cycle)

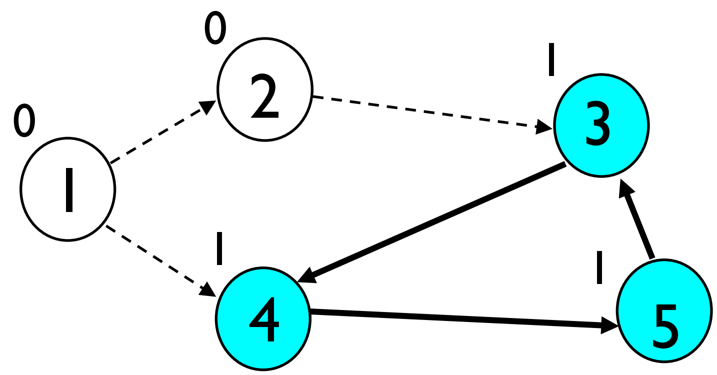
Queue (before): 1
Queue (after): 2



Answer: 1

Topological Sort Fails (cycle)

Queue (before): 2
Queue (after):



Answer: 1, 2

Topological Sort Runtime?

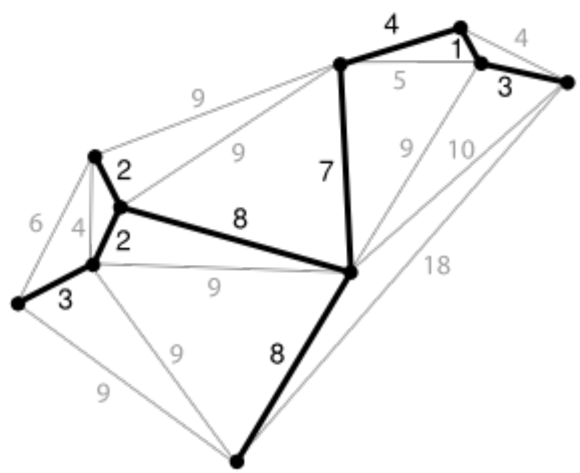
```
Initialize D   // Mapping of vertex to its in-degree
Queue Q := [Vertices with in-degree 0]
while notEmpty(Q) do
  x := Dequeue(Q)
  Output(x)
  y := A[x]; // y gets a linked list of adjacent vertices
  while y ≠ null do
    D[y.value] := D[y.value] – 1;
    if D[y.value] = 0 then Enqueue(Q,y.value);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

Topological Sort Analysis

- Initialize In-Degree map: $O(|V| + |E|)$
- 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:
  - $O(|E|)$
- Runtime = $O(|V| + |E|)$ *Linear*!

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



Minimum Spanning Tree: Applications

- Consider a cable TV company laying cable to a new neighborhood
  - Can only bury the cable only along certain paths
  - Some of paths may be more expensive (i.e. longer, harder to install)
  - 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

Spanning Tree Problem

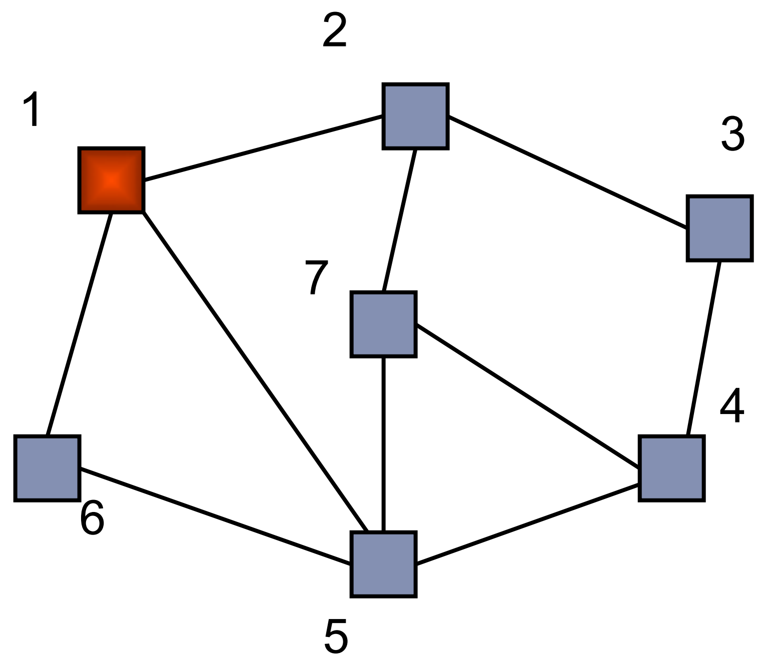
- Input: An undirected graph $G = (V, E)$. G is connected.
- Output: T subset of E such that
  - (V, T) is a connected graph
  - (V, T) has no cycles

Spanning Tree Psuedocode

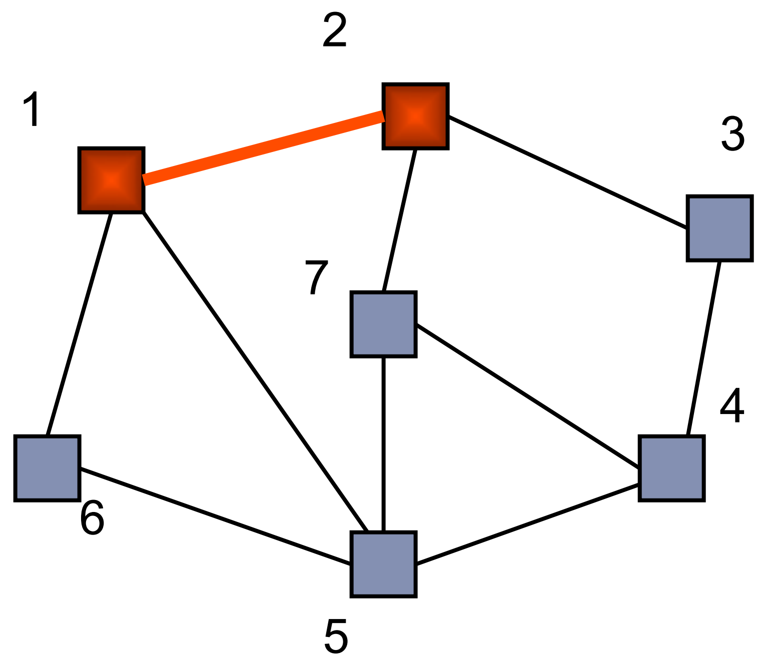
```
spanningTree():
    pick random vertex v.
    T := {}
    spanningTree(v,T)
    return T.

spanningTree(v,T):
    mark v as visited.
    for each neighbor v_i of v where there is an edge from v:
        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

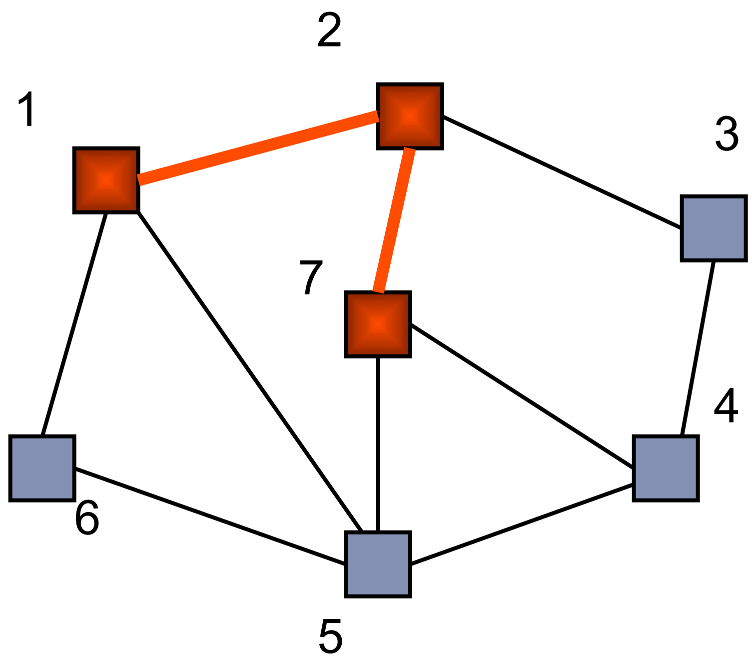


ST(1)
ST(2)

{1,2}

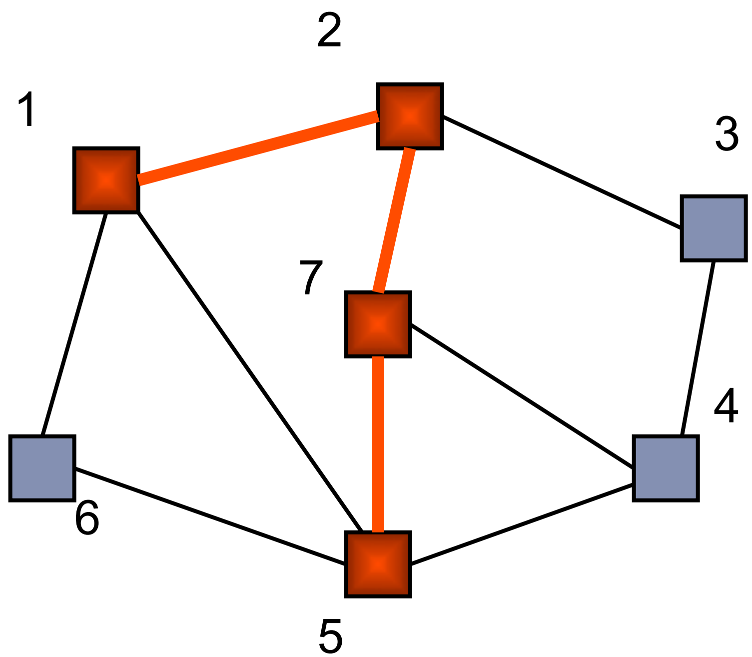
Example Step 3

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



{1,2} {2,7}

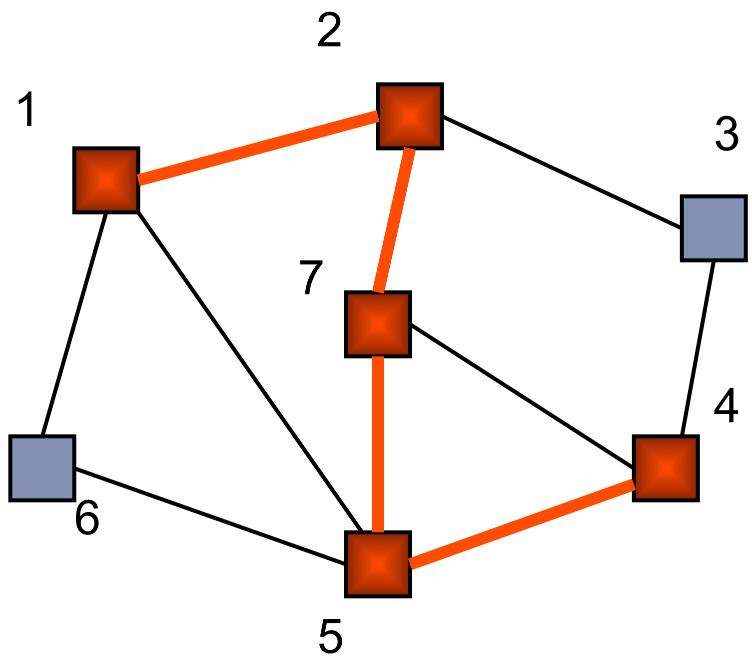
Example Step 4



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

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

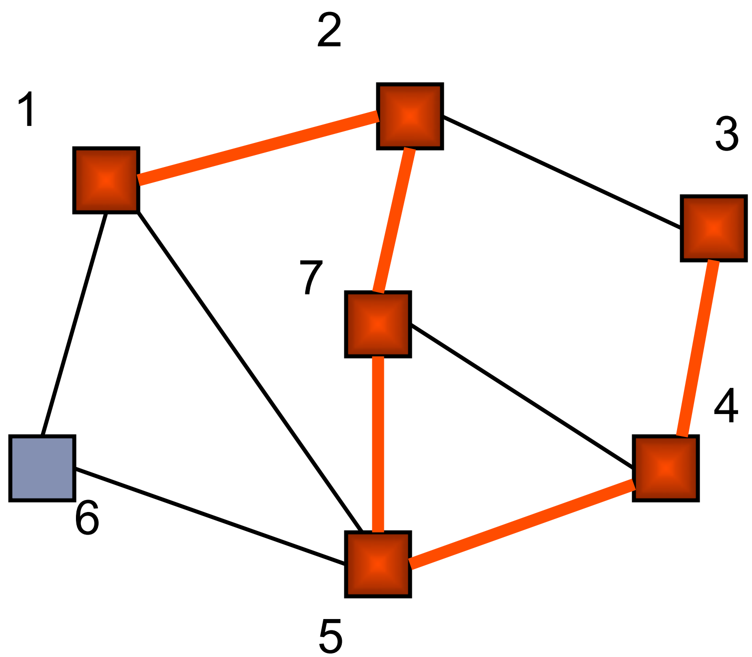
Example Step 5



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

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

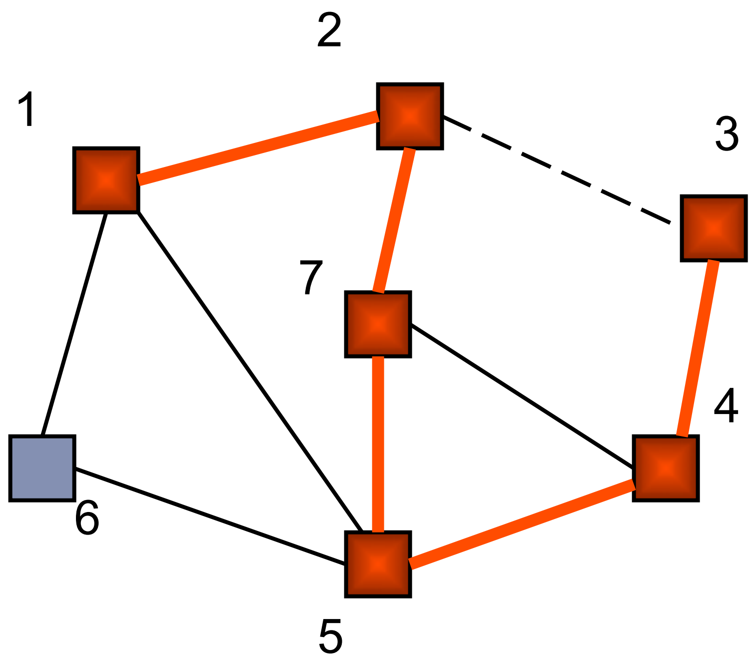
Example Step 6



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

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

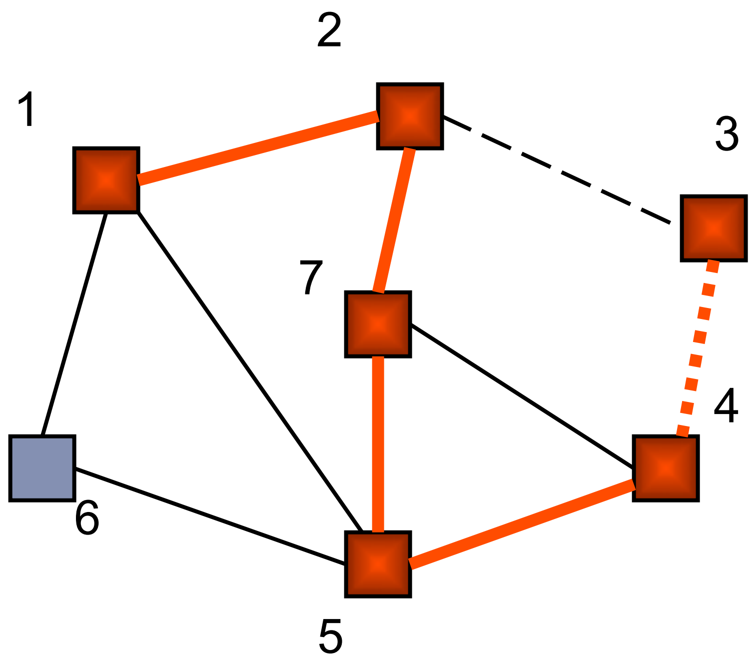
Example Step 7



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

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

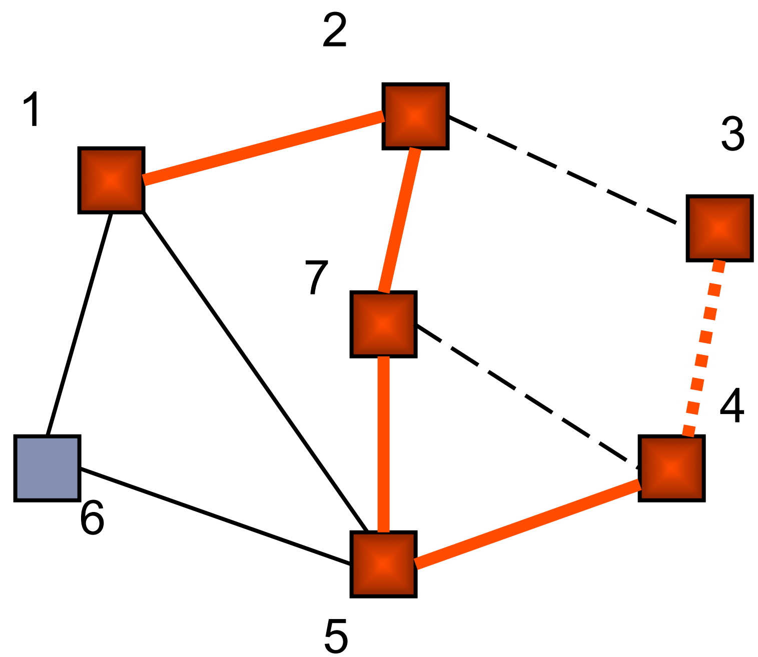
Example Step 8



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

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

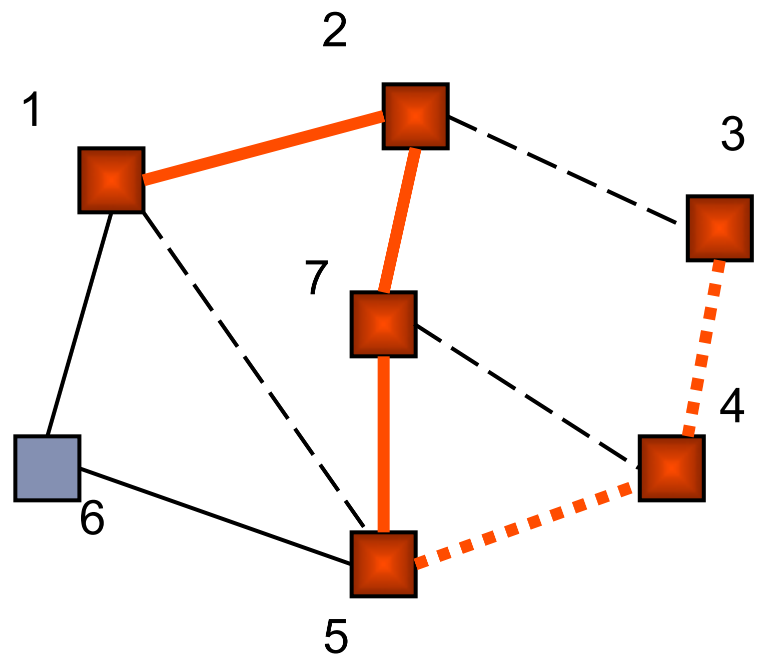
Example Step 9



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

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

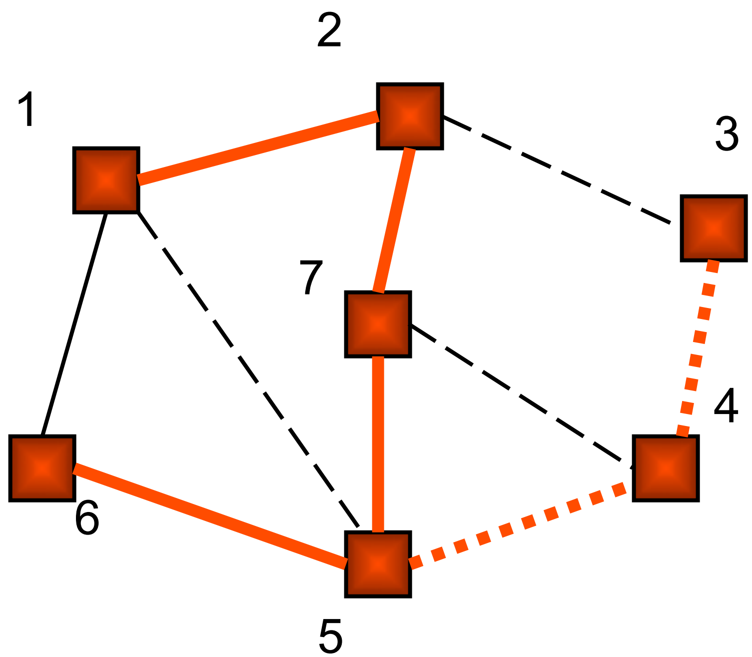
Example Step 10



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

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

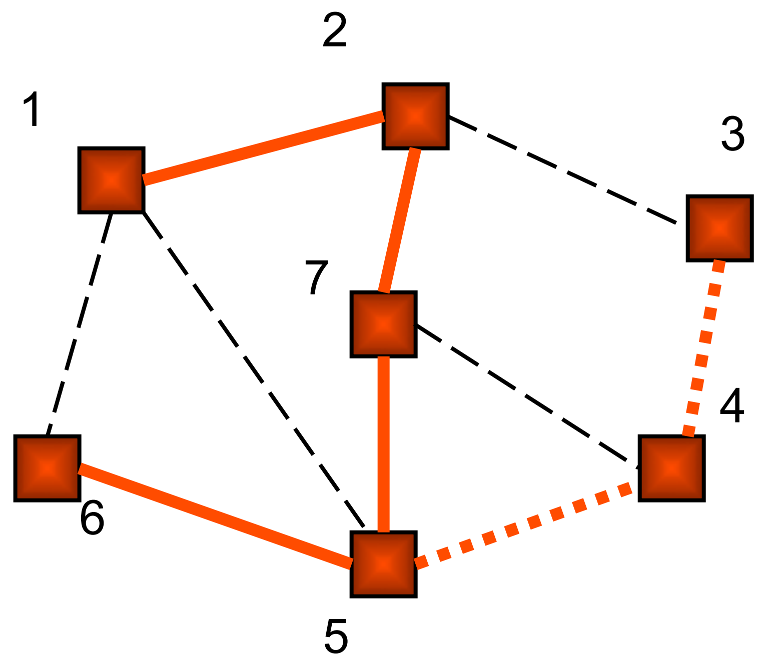
Example Step 11



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

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

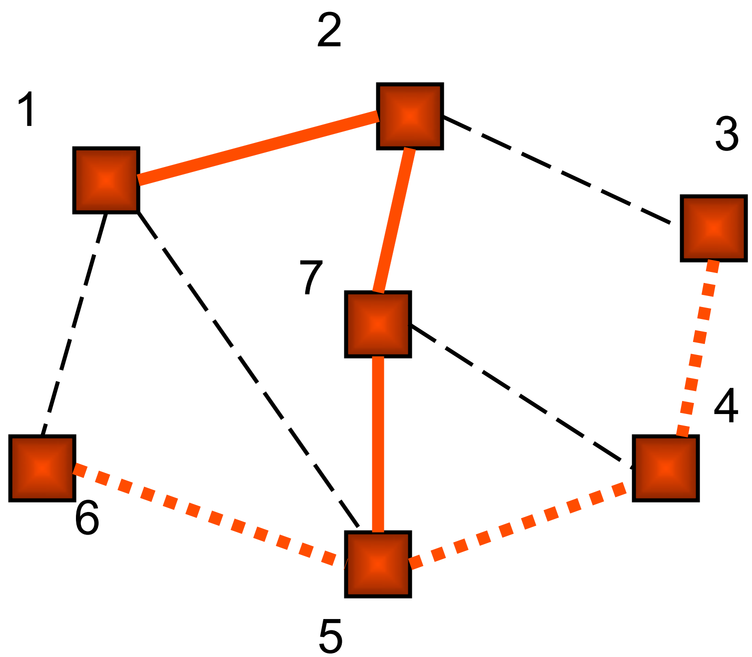
Example Step 12



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

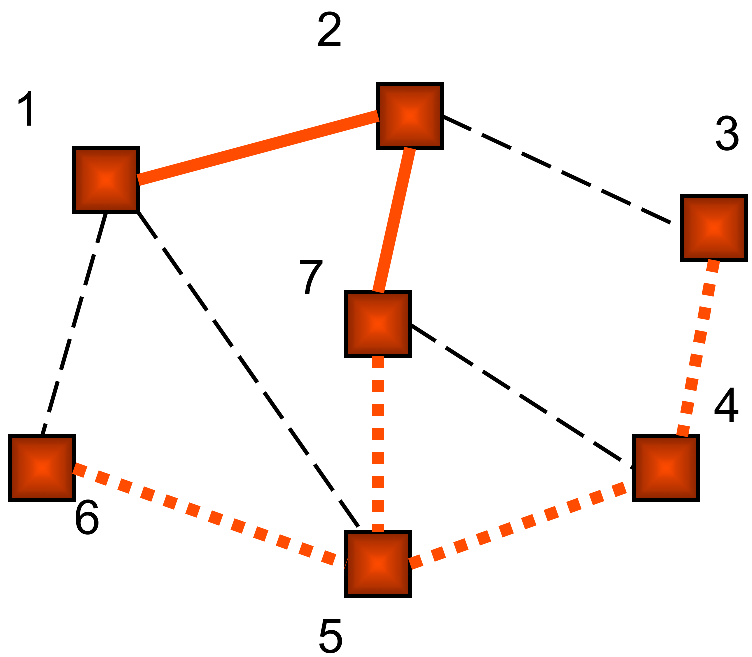


ST(1)
ST(2)
ST(7)
ST(5)

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

Example Step 14

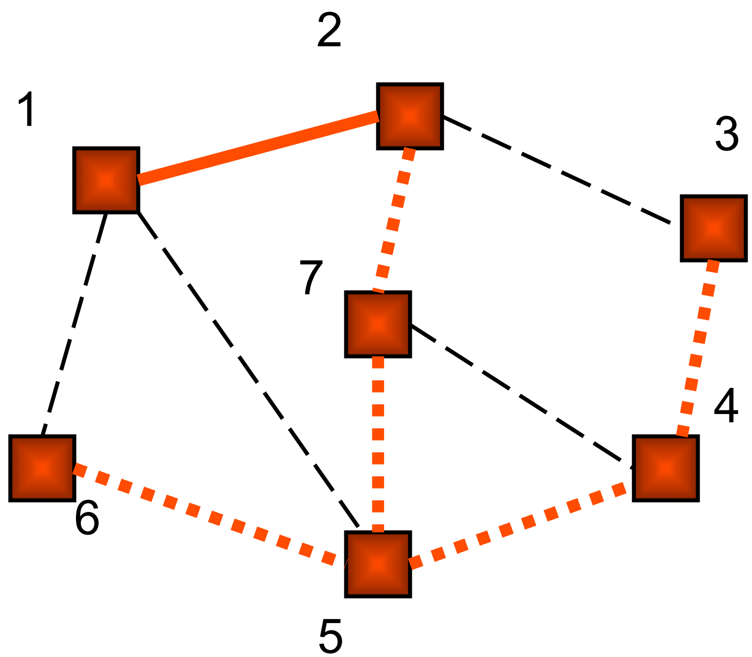
ST(1)
ST(2)
ST(7)



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

Example Step 15

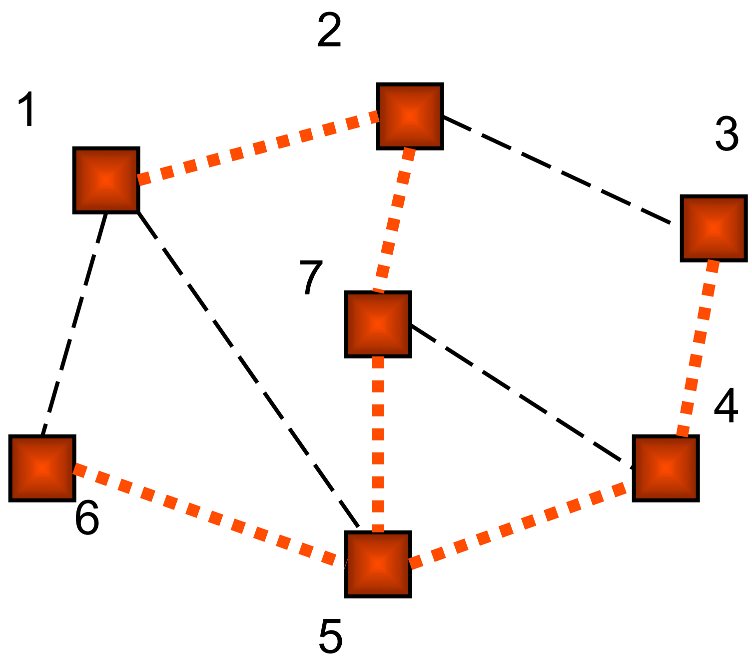
ST(1)
ST(2)



{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

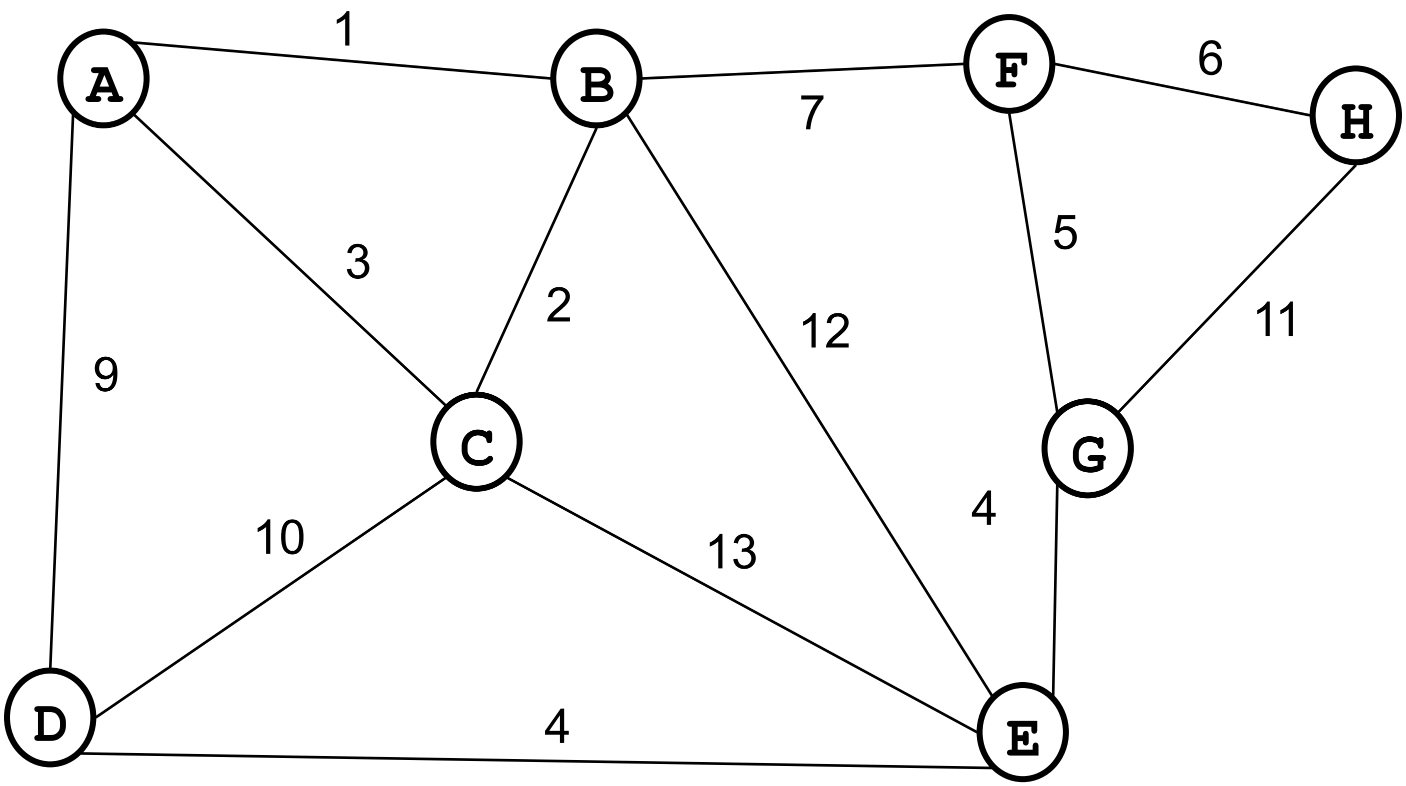
- Input: Undirected Graph $G = (V, E)$ and a cost function C from E to non-negative real numbers. C(e) is the cost of edge e.
- Output: A spanning tree T with minimum total cost. That is: T that minimizes

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

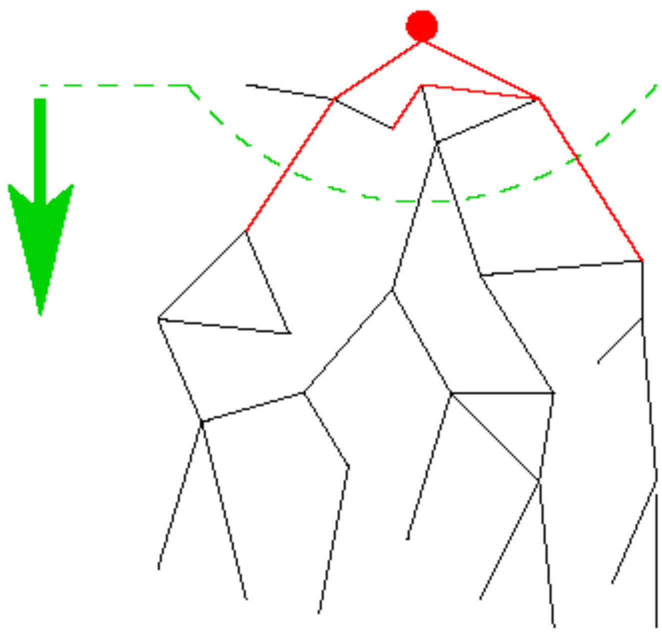
Observations About Spanning Trees

- 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

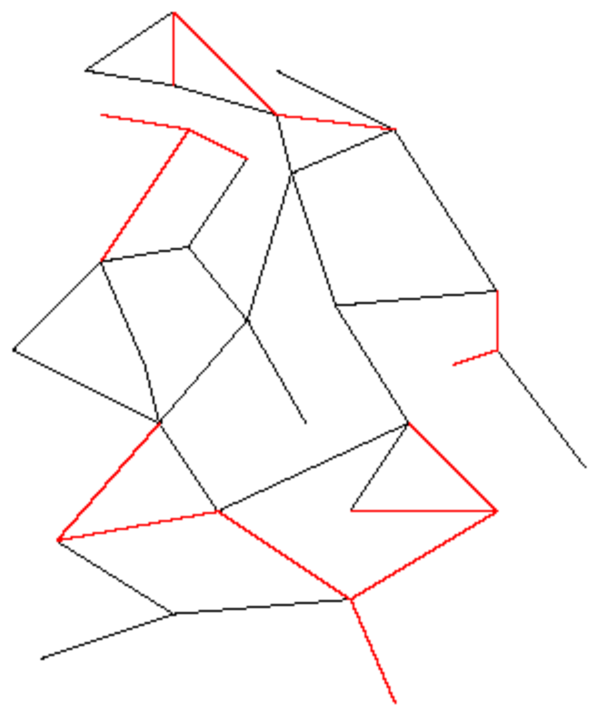


Two Different Approaches



Prim's Algorithm

Looks familiar!

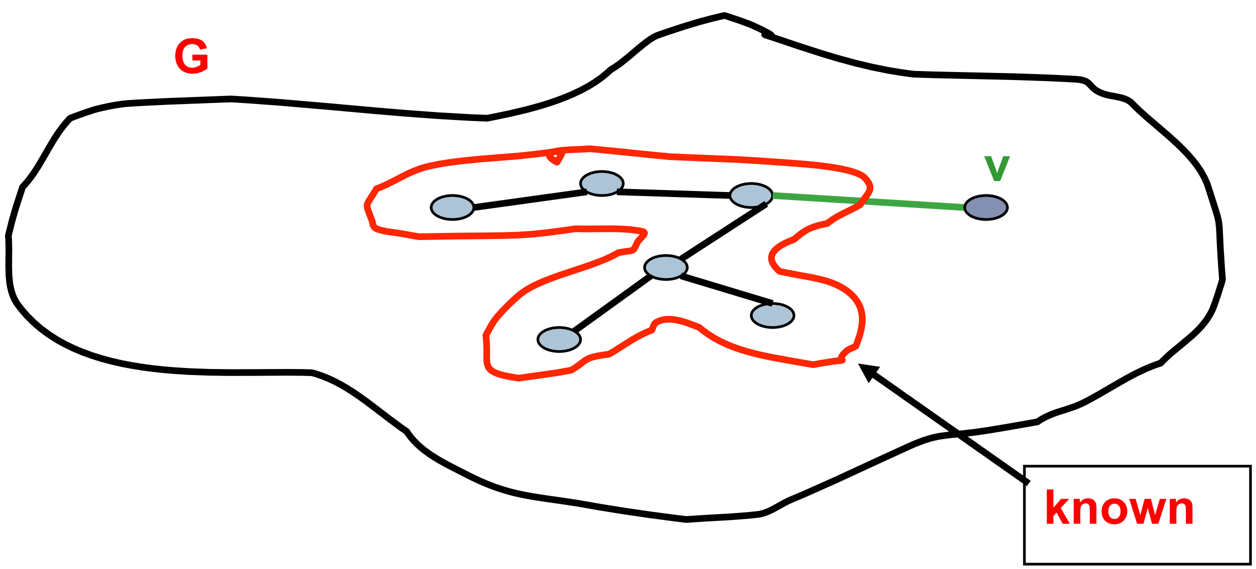


Kruskals' Algorithm

Completely different!

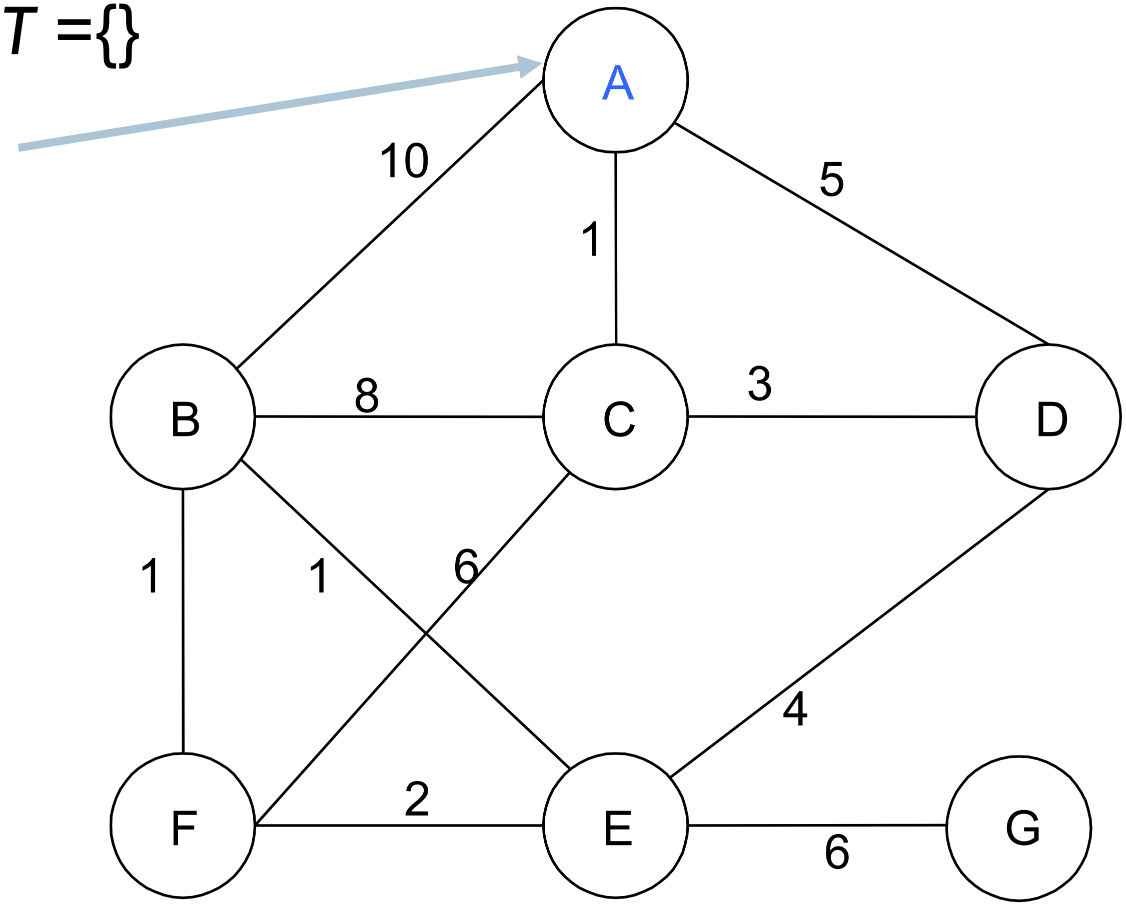
Prim's Algorithm

- **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* ={}

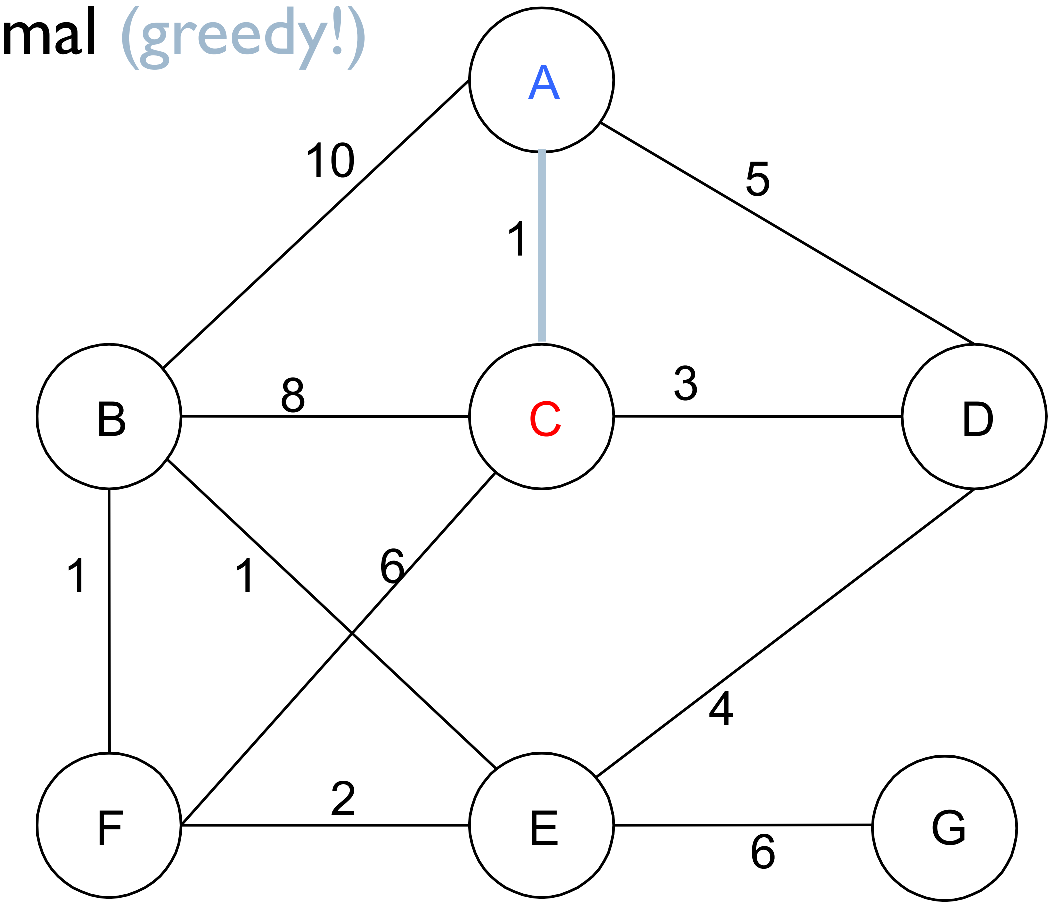


Prim's Algorithm

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

$V = \{A,C\}$

$T = \{ (A,C) \}$

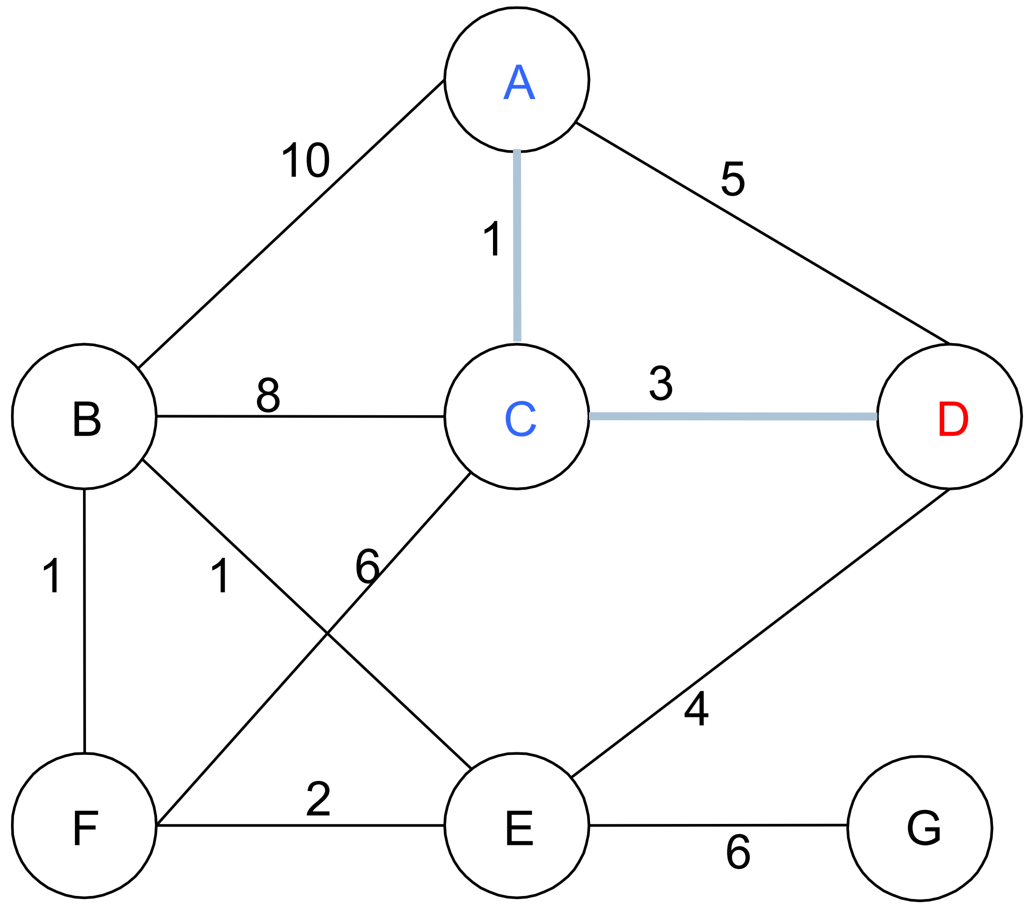


Prim's Algorithm

- Repeat until all vertices have been chosen

  V = {A,C,D}

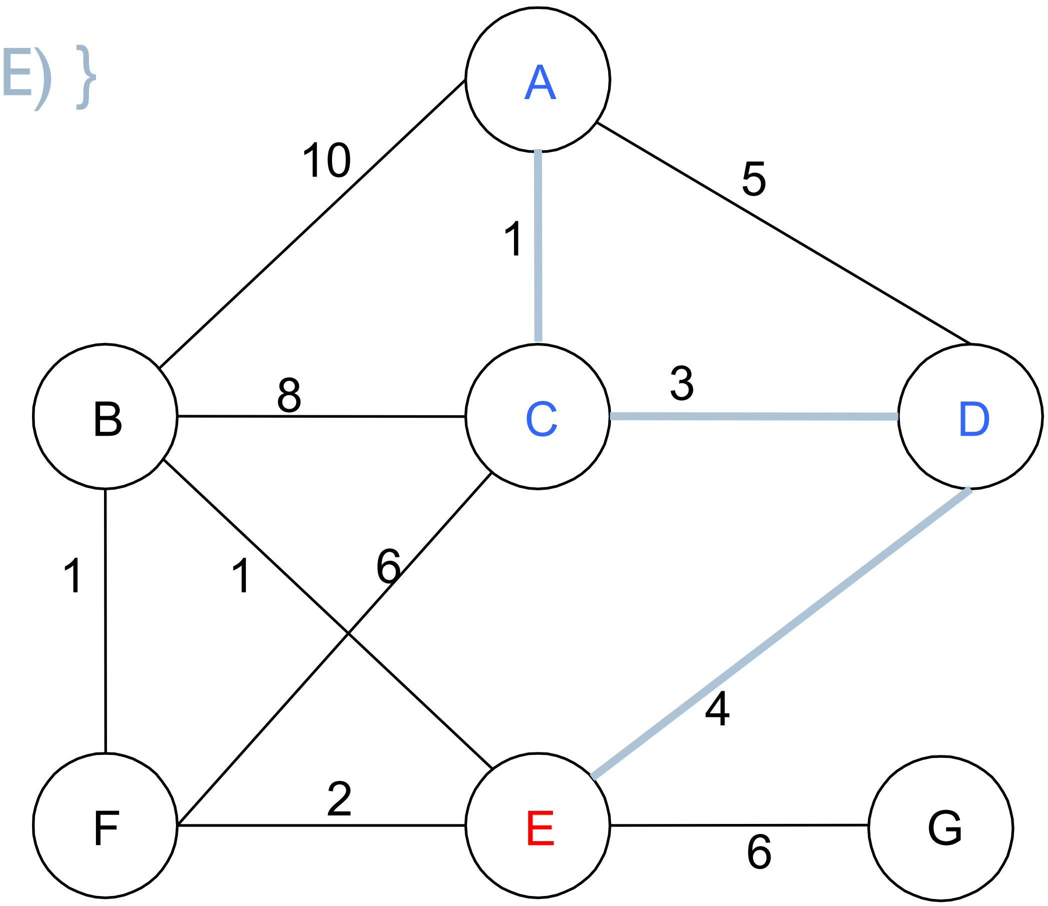
  T= { (A,C), (C,D)}



Prim's Algorithm

V = {A,C,D,E}

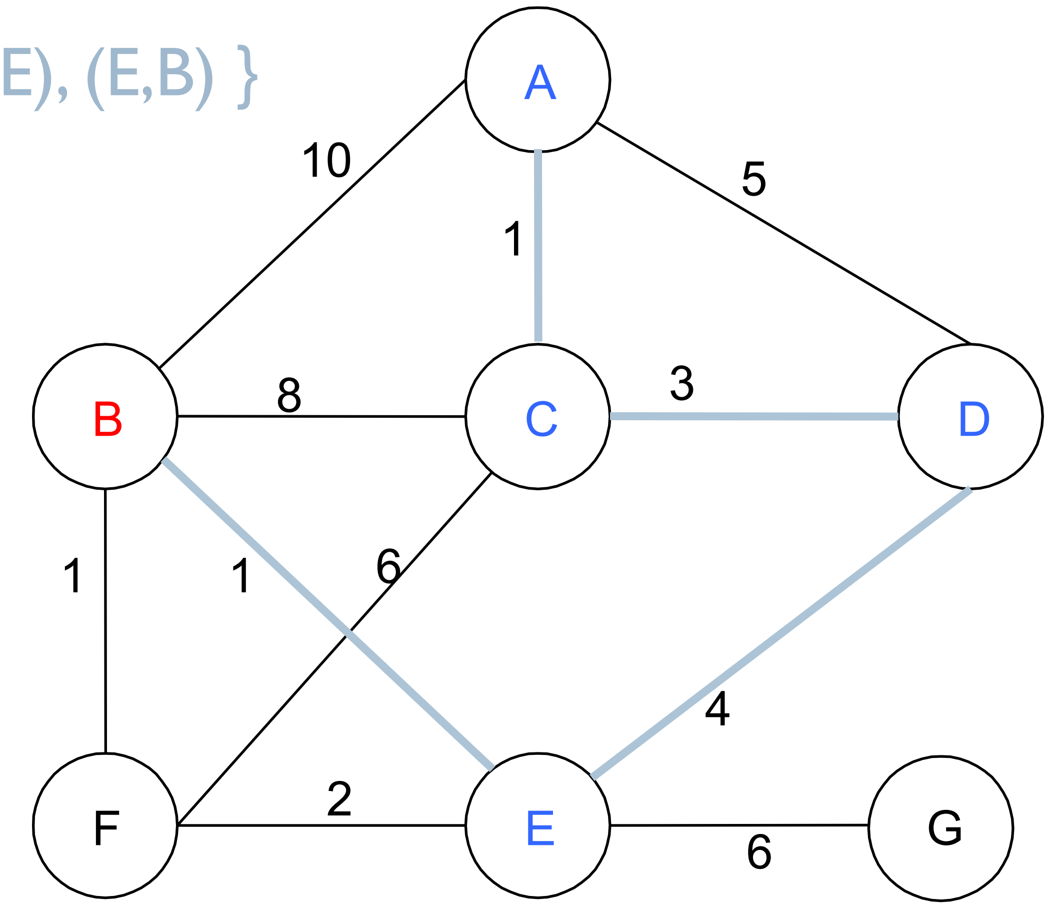
T = { (A,C), (C,D), (D,E) }



Prim's Algorithm

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

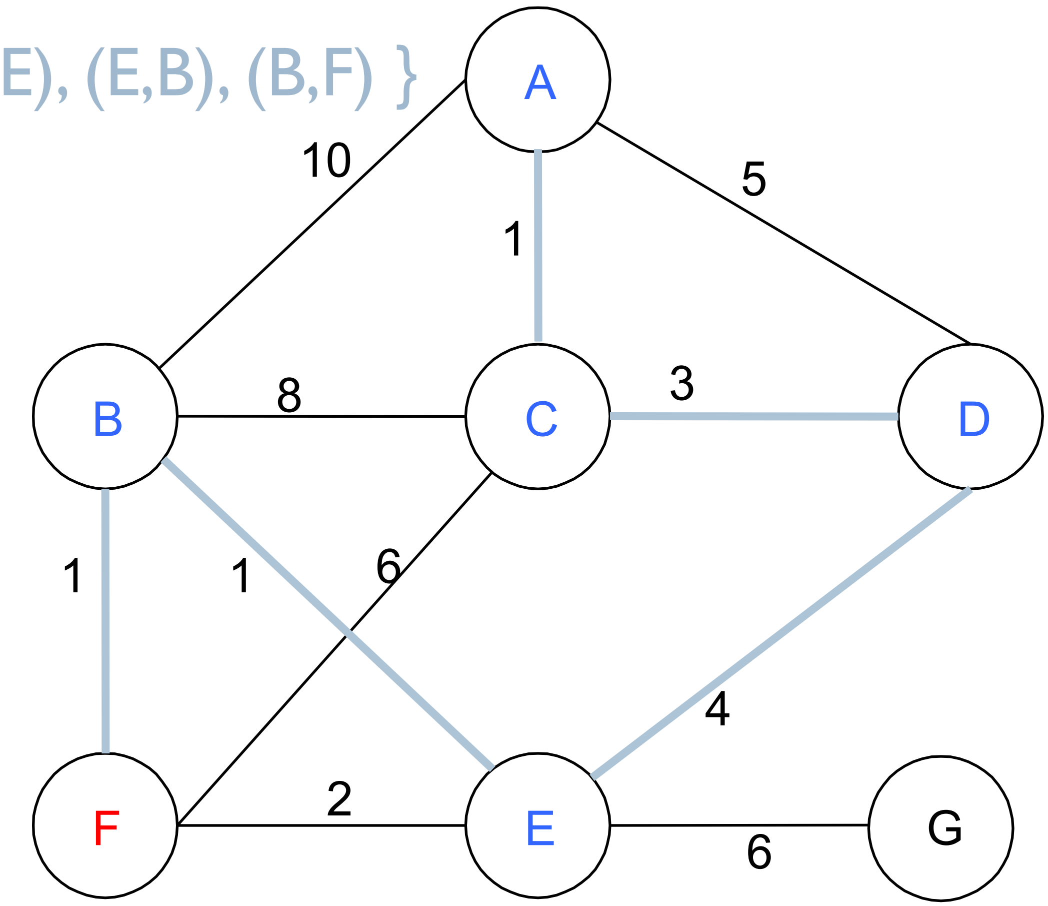
T = { (A,C), (C,D), (D,E), (E,B) }



Prim's Algorithm

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

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

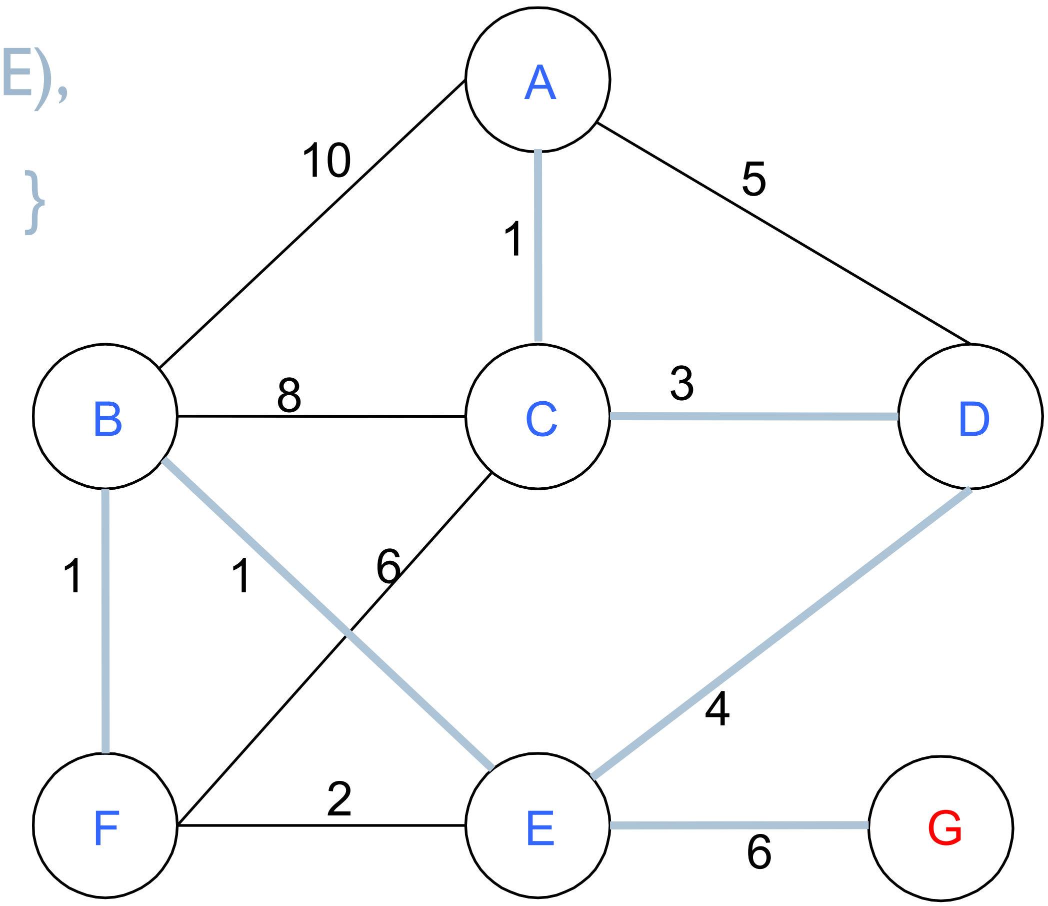


Prim's Algorithm

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

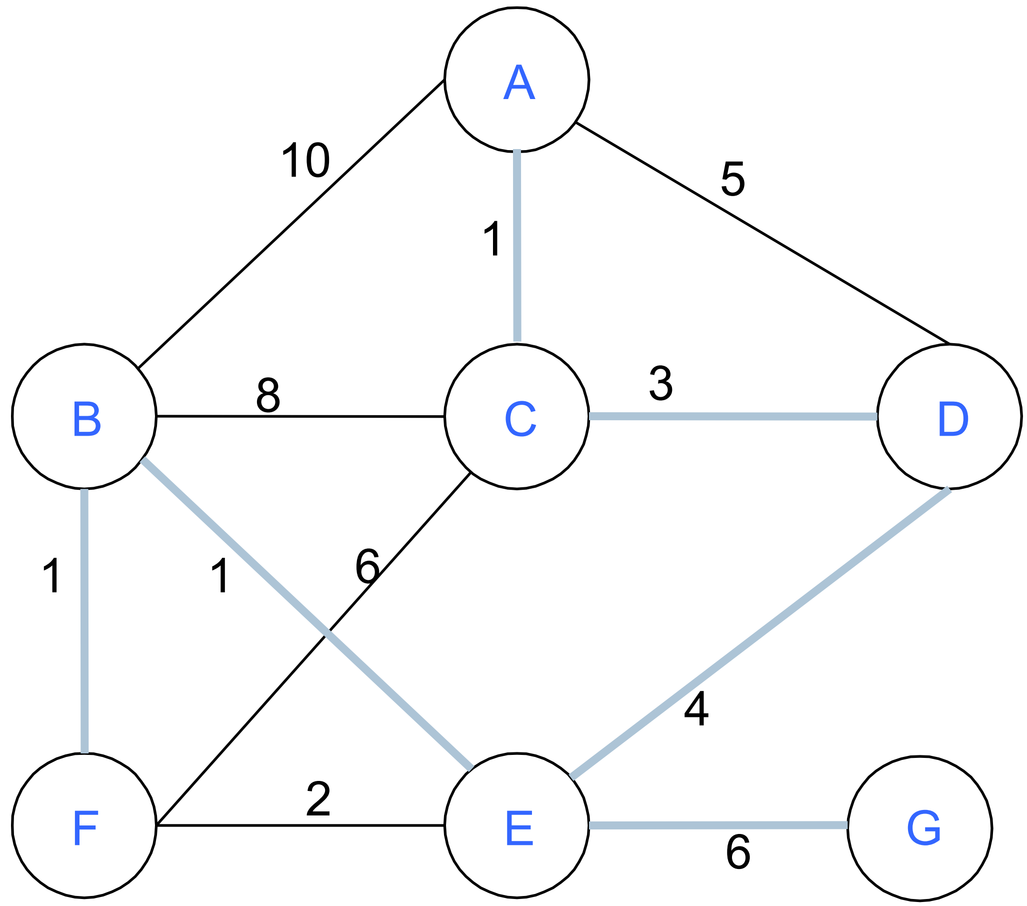
$T =$ { (A,C), (C,D), (D,E),

(E,B), (B,F), (E,G) }



Prim's Algorithm

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



Prim’s Algorithm Analysis

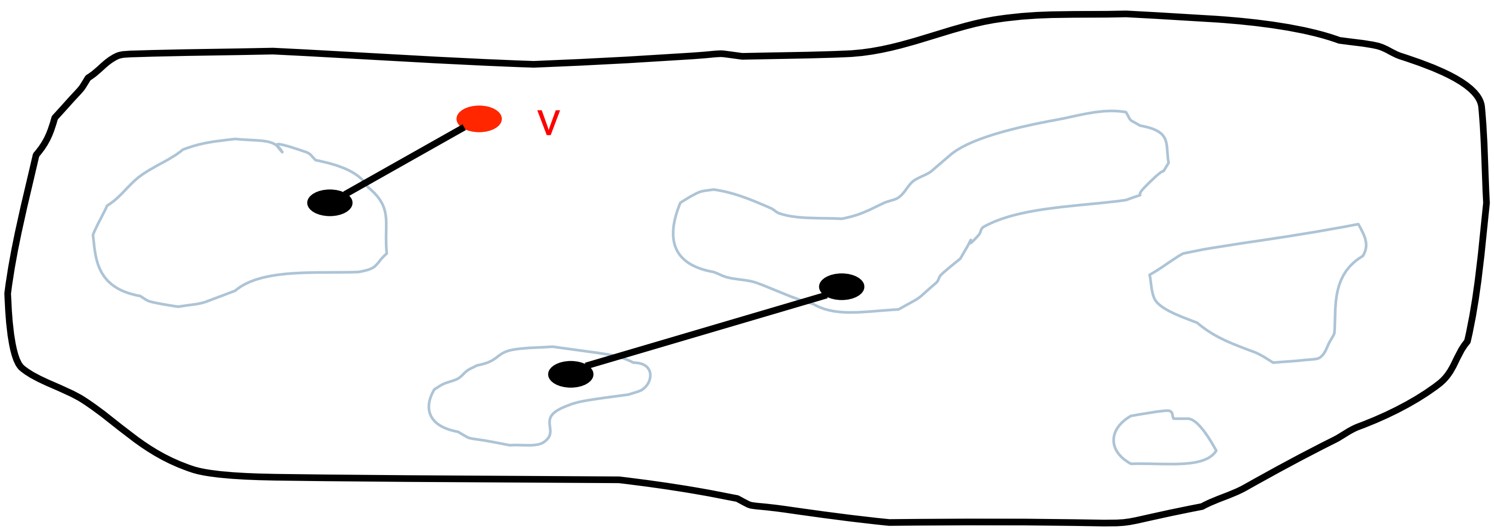
- How is it different from Djikstra's algorithm?

- If the step that removes unknown vertex with minimum distance is done with binary heap, the running time is: $O(|E|\log |V|)$

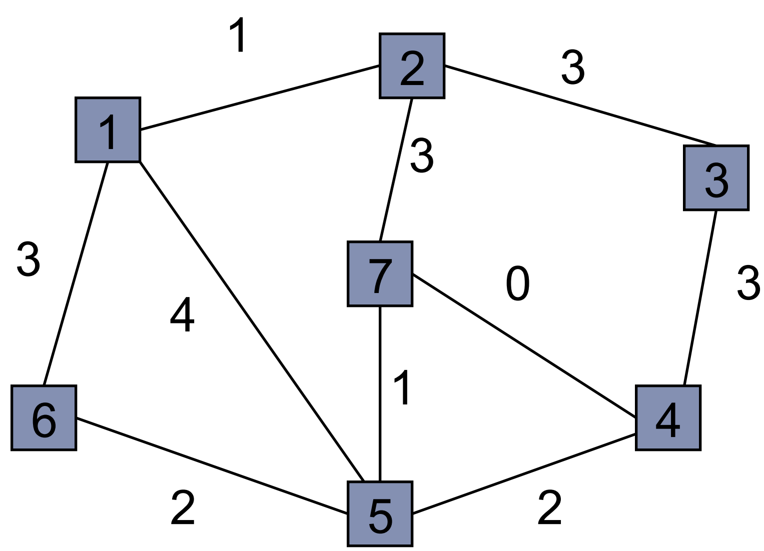
Kruskal's MST Algorithm

- **Idea**: Grow a forest out of edges that do not create a cycle. Pick an edge with the smallest weight.

G=(V,E)

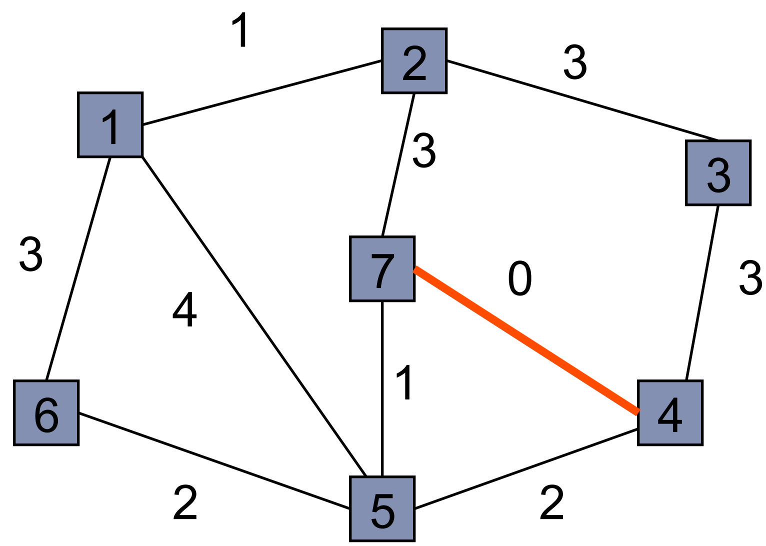


Example of Kruskal 1



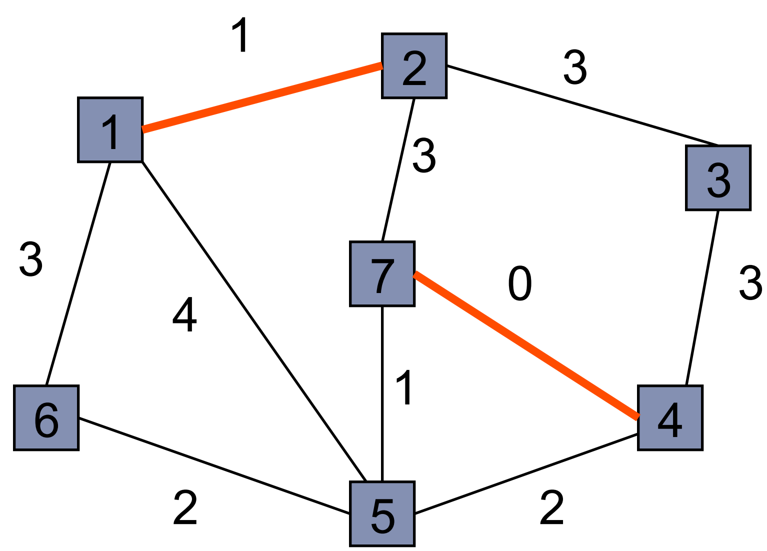
{7,4}	{2,1}	{7,5}	{5,6}	{5,4}	{1,6}	{2,7}	{2,3}	{3,4}	{1,5}
0	1	1	2	2	3	3	3	3	4

Example of Kruskal 2



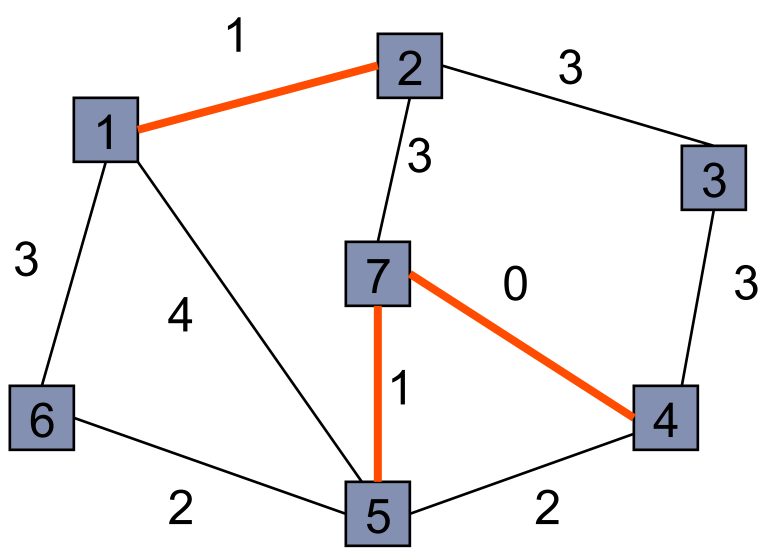
~~{7,4}~~	{2,1}	{7,5}	{5,6}	{5,4}	{1,6}	{2,7}	{2,3}	{3,4}	{1,5}
~~0~~	1	1	2	2	3	3	3	3	4

Example of Kruskal 2



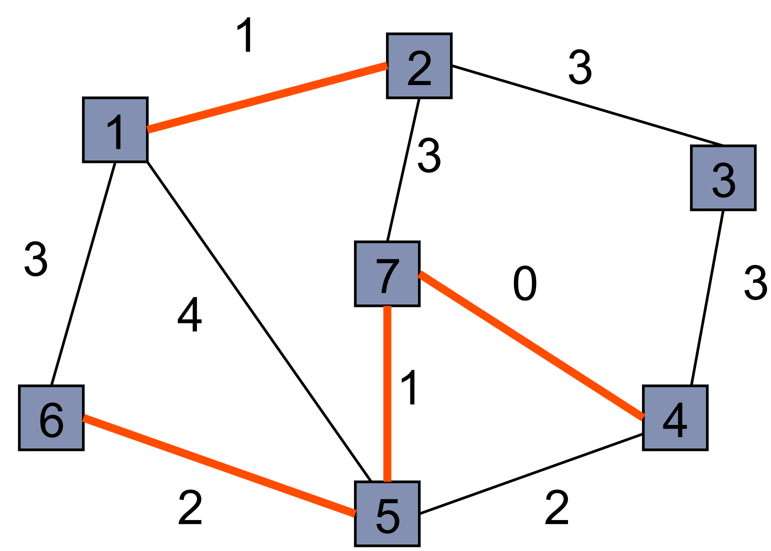
~~{7,4}~~	~~{2,1}~~	{7,5}	{5,6}	{5,4}	{1,6}	{2,7}	{2,3}	{3,4}	{1,5}
~~0~~	~~1~~	1	2	2	3	3	3	3	4

Example of Kruskal 3



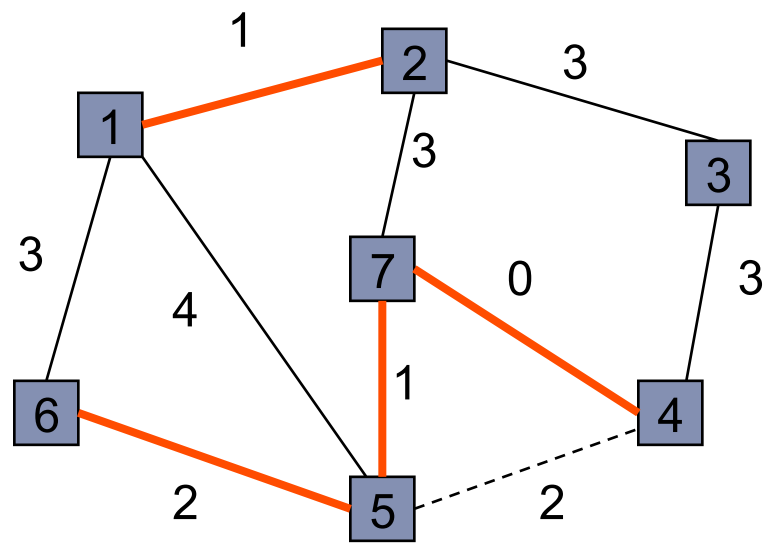
~~{7,4}~~	~~{2,1}~~	~~{7,5}~~	{5,6}	{5,4}	{1,6}	{2,7}	{2,3}	{3,4}	{1,5}
~~0~~	~~1~~	~~1~~	2	2	3	3	3	3	4

Example of Kruskal 4



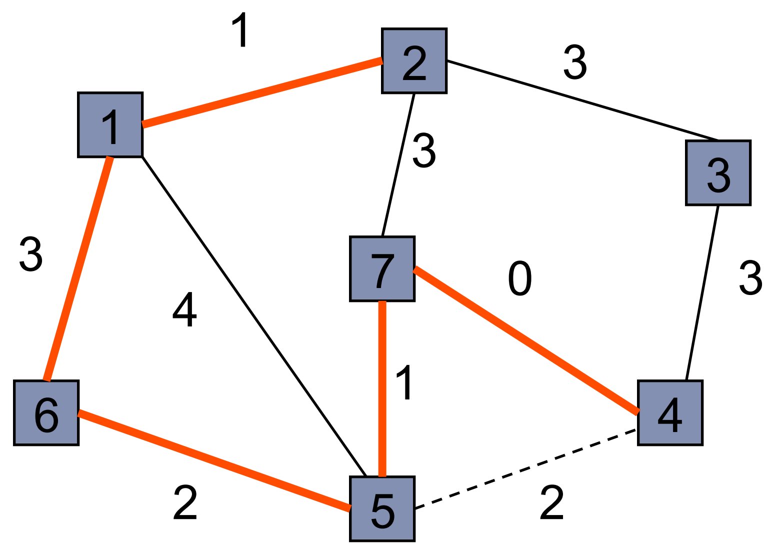
~~{7,4}~~	~~{2,1}~~	~~{7,5}~~	~~{5,6}~~	{5,4}	{1,6}	{2,7}	{2,3}	{3,4}	{1,5}
~~0~~	~~1~~	~~1~~	~~2~~	2	3	3	3	3	4

Example of Kruskal 5



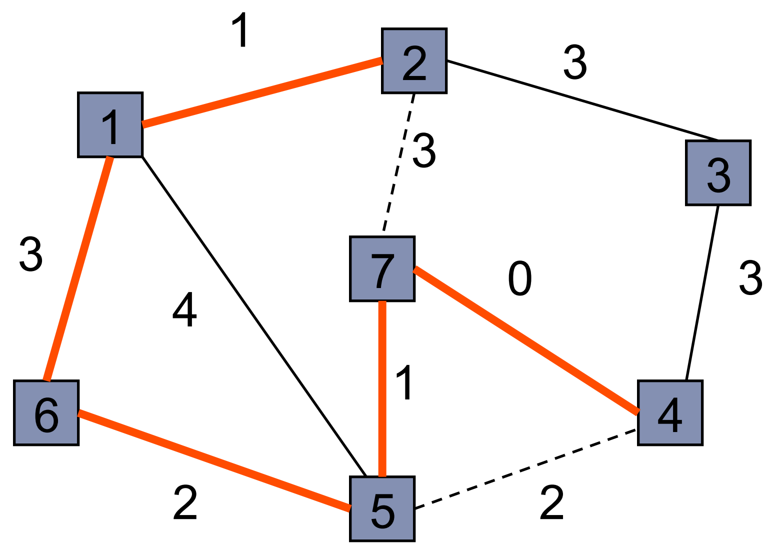
~~{7,4}~~	~~{2,1}~~	~~{7,5}~~	~~{5,6}~~	~~{5,4}~~	{1,6}	{2,7}	{2,3}	{3,4}	{1,5}
~~0~~	~~1~~	~~1~~	~~2~~	~~2~~	3	3	3	3	4

Example of Kruskal 6



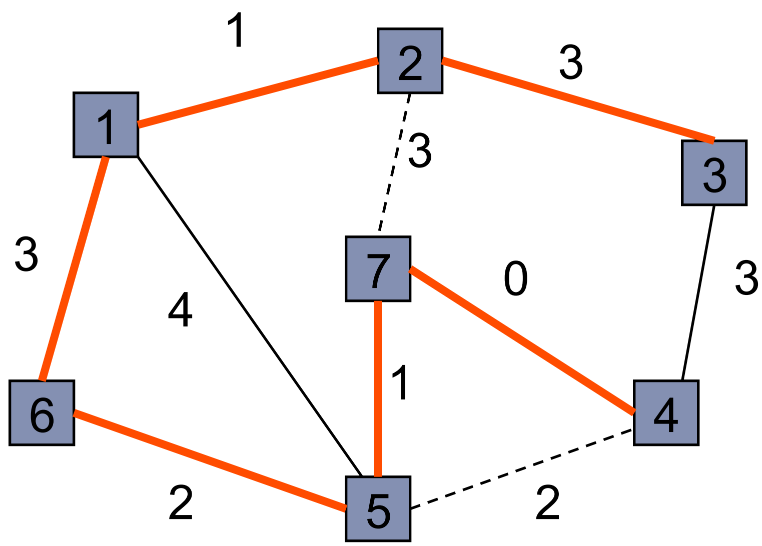
~~{7,4}~~	~~{2,1}~~	~~{7,5}~~	~~{5,6}~~	~~{5,4}~~	~~{1,6}~~	{2,7}	{2,3}	{3,4}	{1,5}
~~0~~	~~1~~	~~1~~	~~2~~	~~2~~	~~3~~	3	3	3	4

Example of Kruskal 7



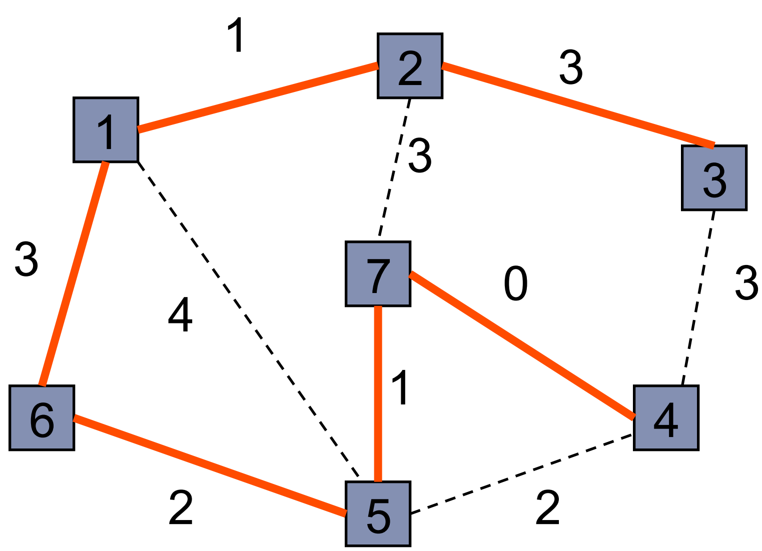
~~{7,4}~~	~~{2,1}~~	~~{7,5}~~	~~{5,6}~~	~~{5,4}~~	~~{1,6}~~	~~{2,7}~~	{2,3}	{3,4}	{1,5}
~~0~~	~~1~~	~~1~~	~~2~~	~~2~~	~~3~~	~~3~~	3	3	4

Example of Kruskal 7



~~{7,4}~~	~~{2,1}~~	~~{7,5}~~	~~{5,6}~~	~~{5,4}~~	~~{1,6}~~	~~{2,7}~~	~~{2,3}~~	{3,4}	{1,5}
~~0~~	~~1~~	~~1~~	~~2~~	~~2~~	~~3~~	~~3~~	~~3~~	3	4

Example of Kruskal 8,9



~~{7,4}~~	~~{2,1}~~	~~{7,5}~~	~~{5,6}~~	~~{5,4}~~	~~{1,6}~~	~~{2,7}~~	~~{2,3}~~	~~{3,4}~~	~~{1,5}~~
~~0~~	~~1~~	~~1~~	~~2~~	~~2~~	~~3~~	~~3~~	~~3~~	~~3~~	~~4~~

Kruskal's Algorithm Implementation

```
Kruskals():
    sort edges in increasing order of length (e_1, e_2, e_3, ..., e_m).

    T := {}.

    for i = 1 to m
        if e_i does not add a cycle:
            add e_i to T.

    return T.
```

- How can we determine that adding e_i to *T* won't add a cycle?