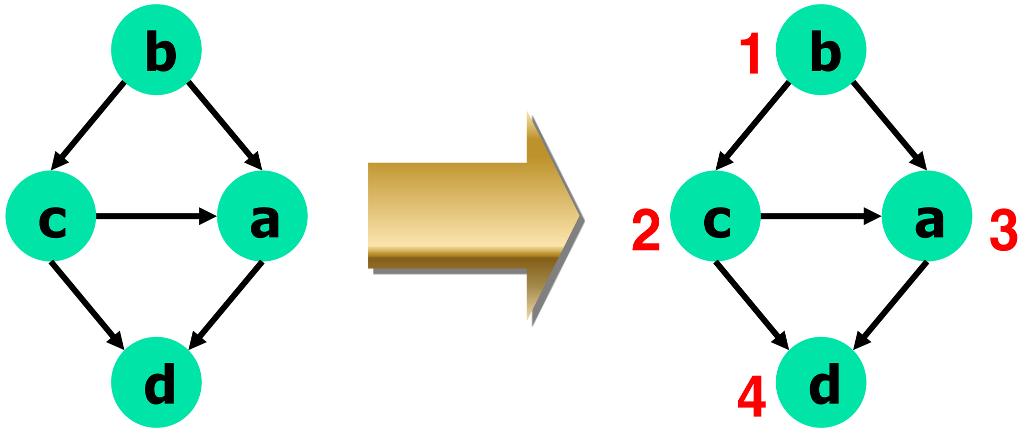
Graph Algorithm

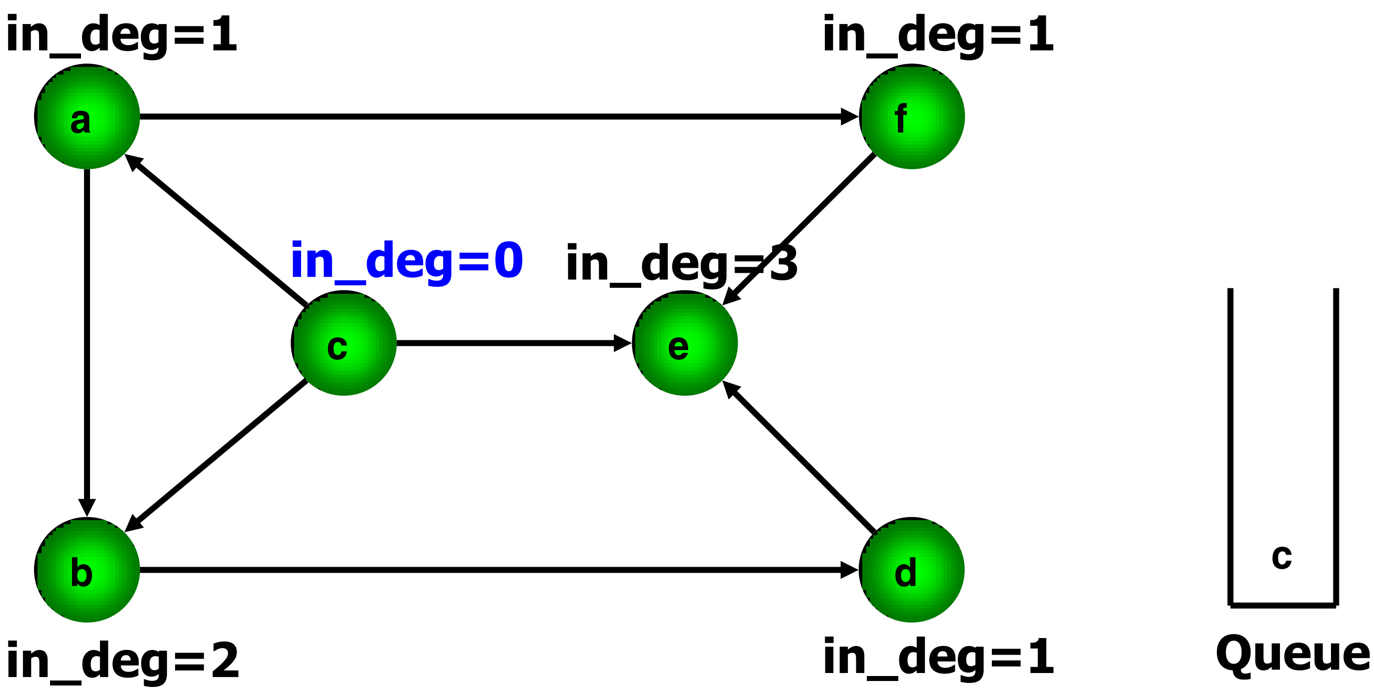
Topological Sort

- Definition
  - A *topological sort* of a DAG *G* is a linear ordering of all its vertices such that if *G* contains a link (u,v), then node u appears before node v in the ordering



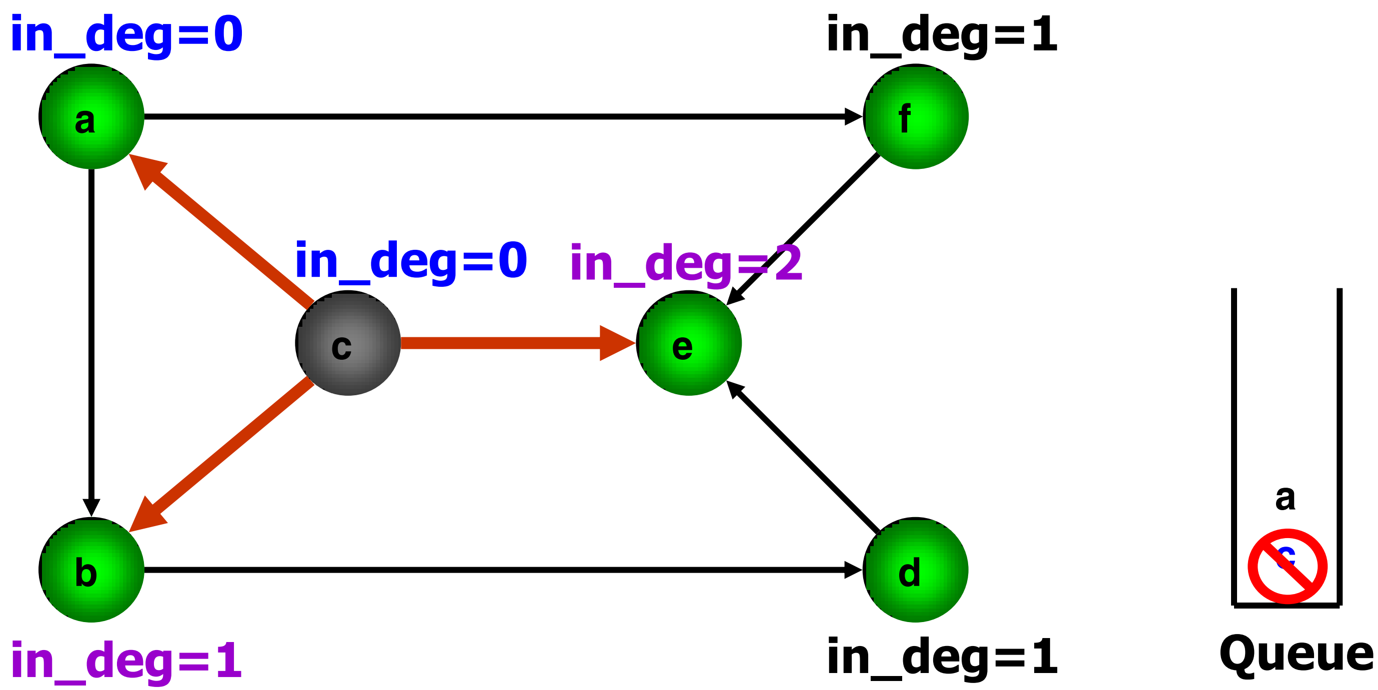
Algorithm Example

- find source nodes (indegree = 0)
  - if there is no such node, the graph is NOT DAG



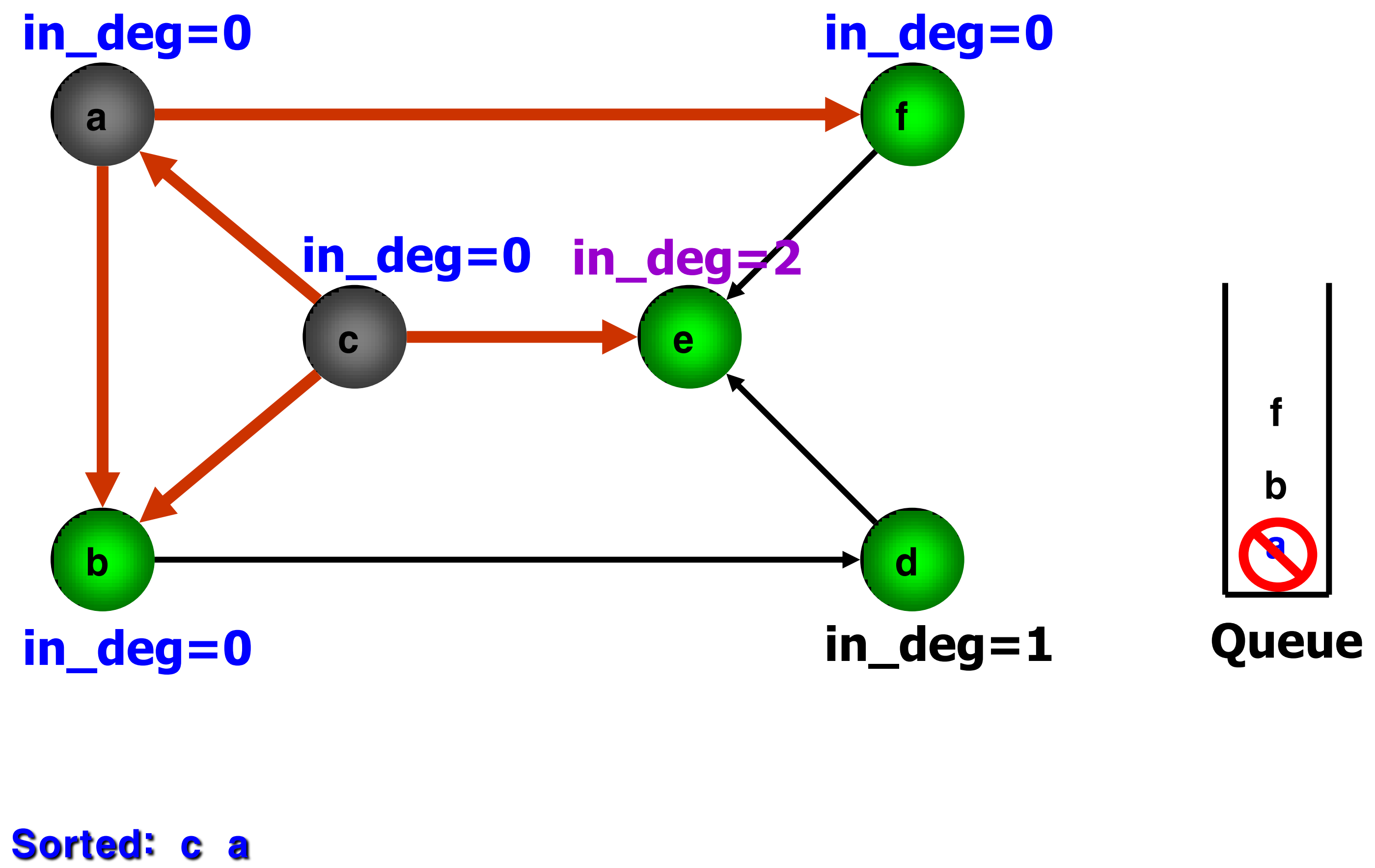
Sorted: -

- span c; decrement in_deg of a, b, e
  - store a in Queue since in_deg becomes 0

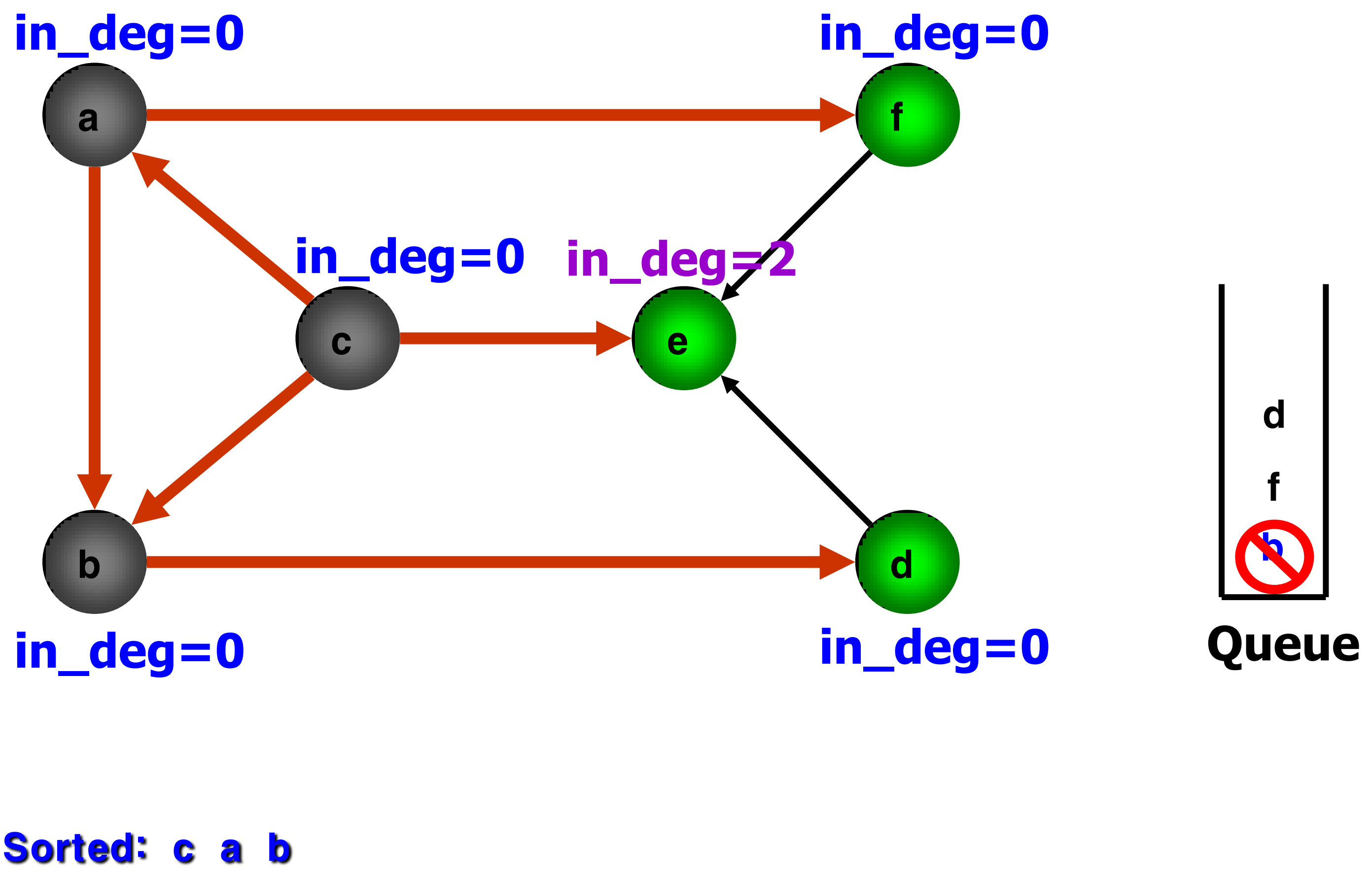


Sorted: c

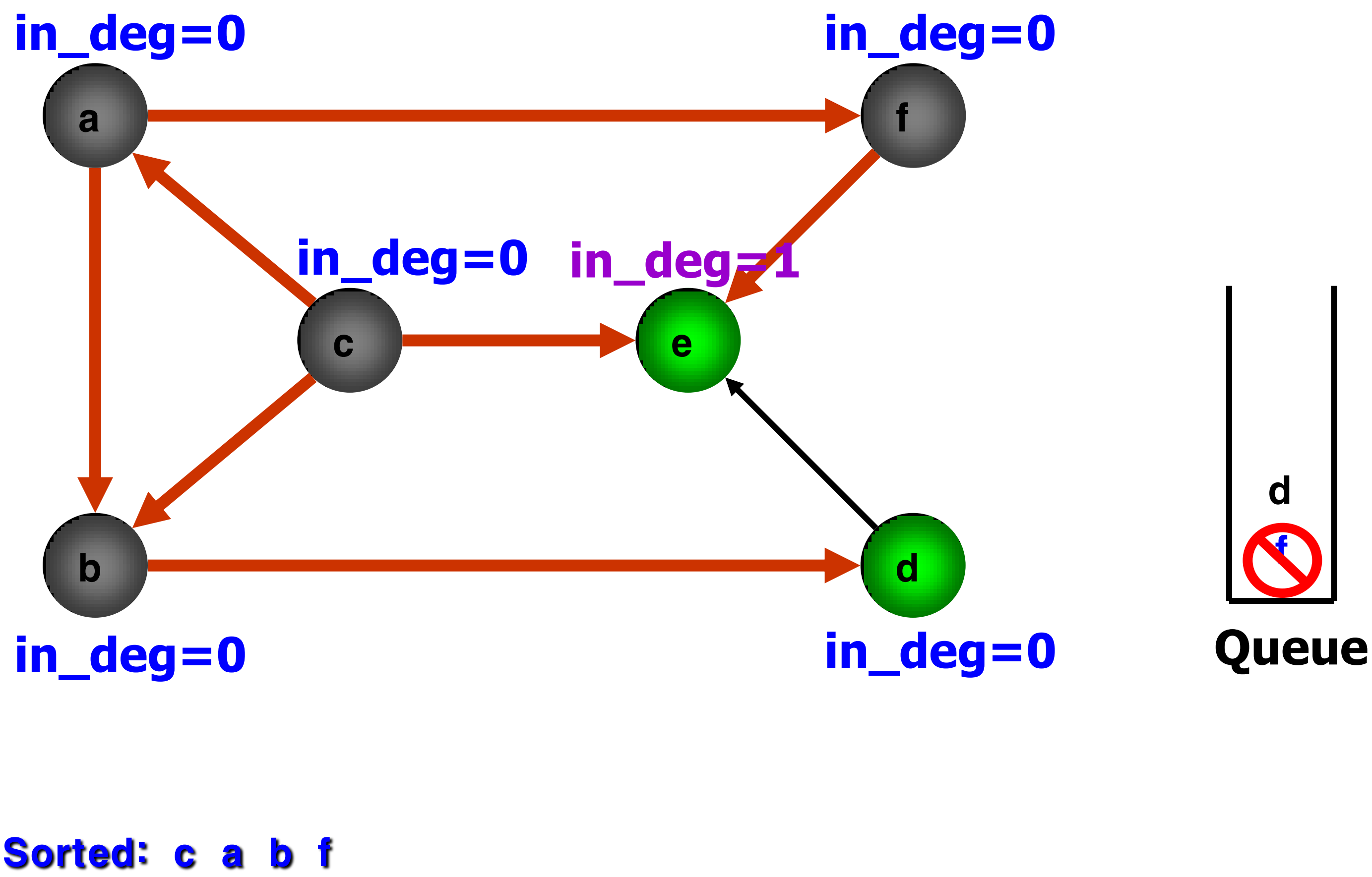
- span a; decrement in_deg of b, f
  - store b, f in Queue since ...



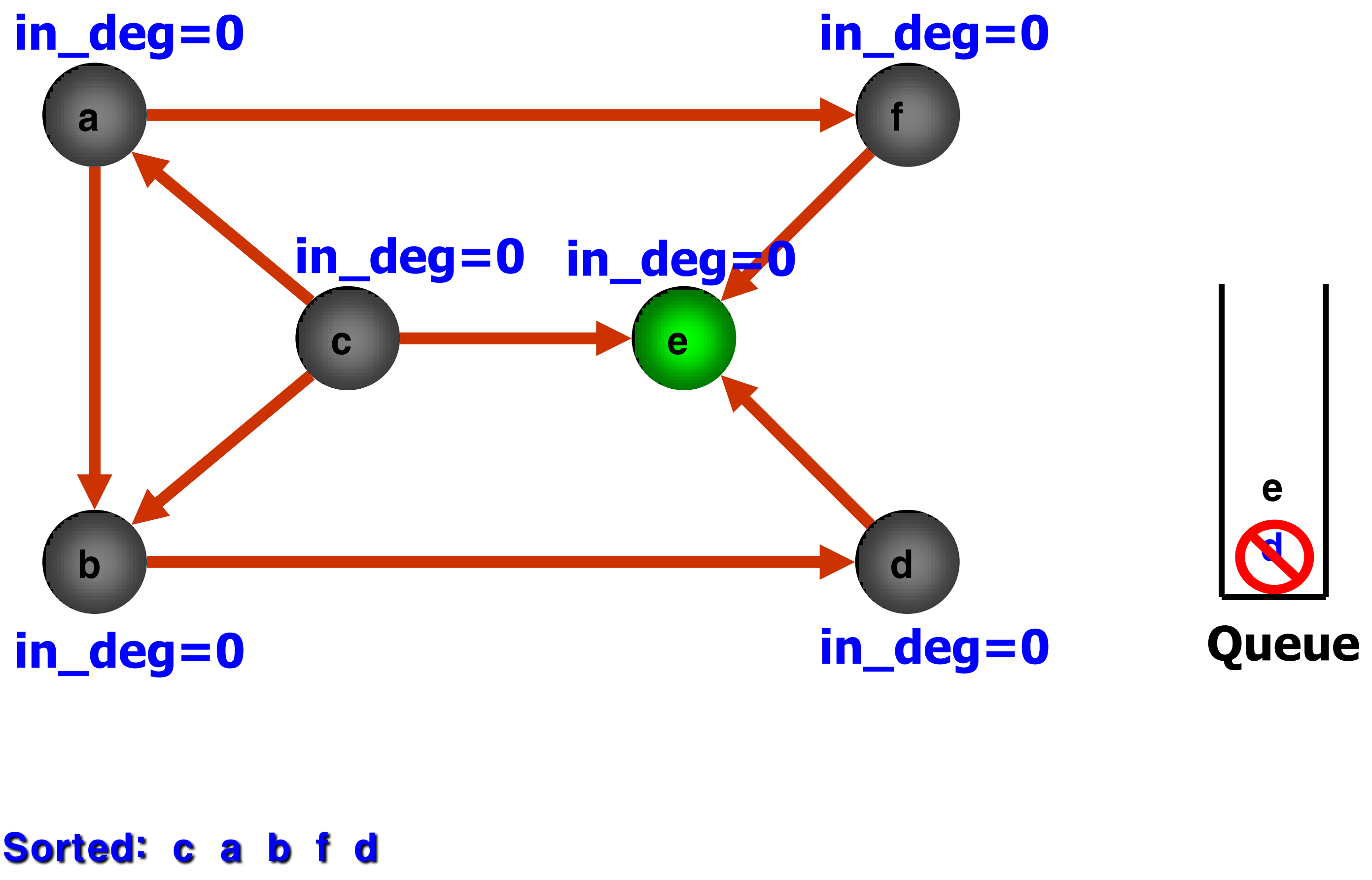
- span b; store d in Queue



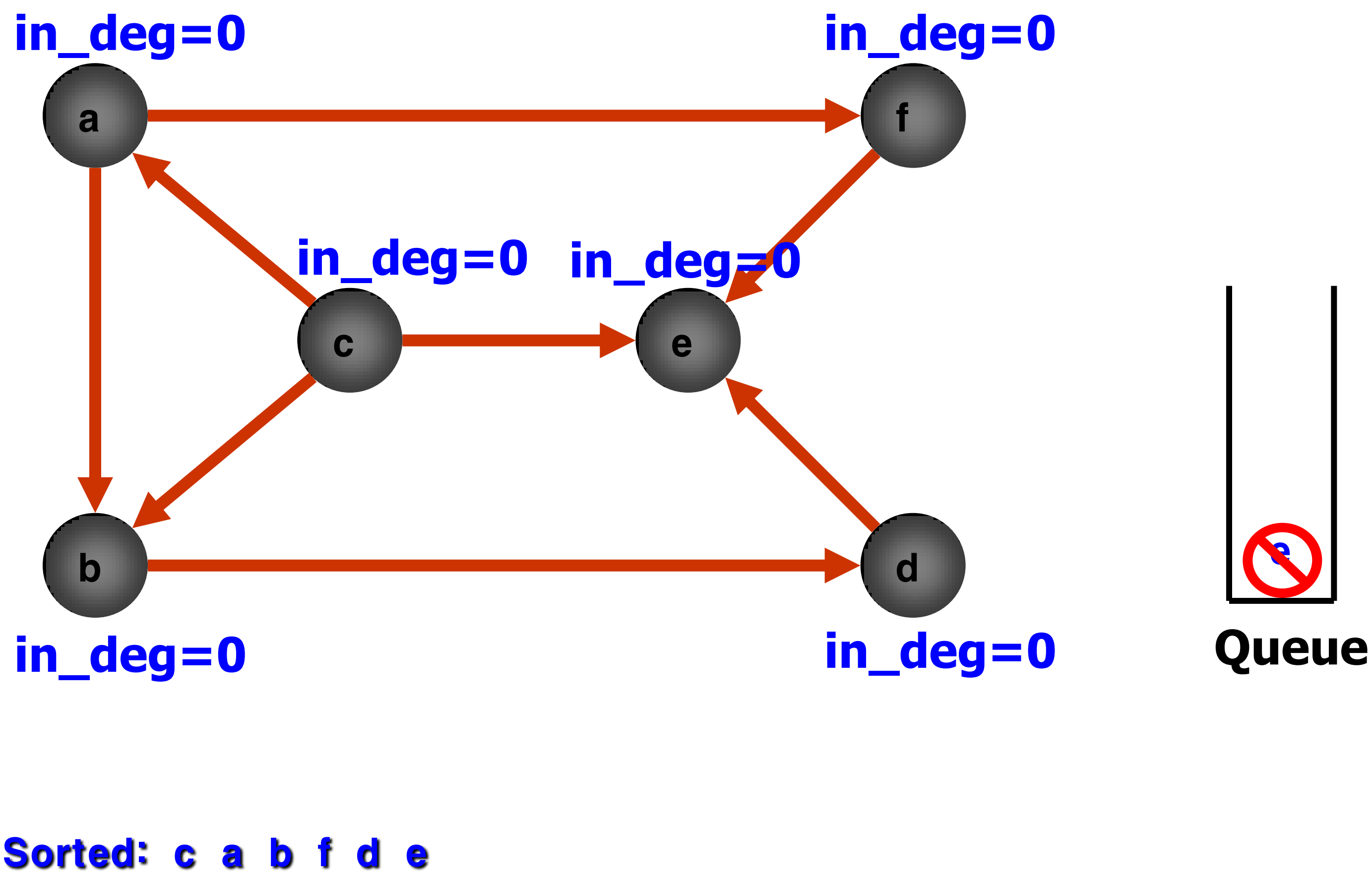
- span f; decrement in_deg of e
  - no node with in_deg = 0 is found



- span d; store e in Queue.



- span e; Queue is empty



Example Algorithm Summary

- Based on indegree of each vertex
  - if it is 0, this node is the first one in the sorted list
  - span this node
    - move this node from Queue to the sorted list
    - find nodes edged from this node
    - decrement indegrees of them
- It is so similar to BFS
  - can you do it like DFS?

```
topsort(graph *g, int sorted[])
{
        int indegree[MAXV];
        queue zeroin;
        int x, y;
        int i, j;

        compute_indegrees(g,indegree);
        init_queue(&zeroin);
        for (i=1; i<=g->nvertices; i++)
                if (indegree[i] == 0) enqueue(&zeroin,i);

        j=0;
        while (empty(&zeroin) == FALSE) {
                j = j+1;
                x = dequeue(&zeroin);
                sorted[j] = x;
                for (i=0; i<g->degree[x]; i++) {
                        y = g->edges[x][i];
                        indegree[y] --;
                        if (indegree[y] == 0) enqueue(&zeroin,y);
                }
        }

        if (j != g->nvertices)
                printf("Not a DAG -- only %d vertices found\n",j);
}
```

```
compute_indegrees(graph *g, int in[])
{
        int i,j;                        /* counters */

        for (i=1; i<=g->nvertices; i++) in[i] = 0;

        for (i=1; i<=g->nvertices; i++)
                for (j=0; j<g->degree[i]; j++)
                        in[ g->edges[i][j] ] ++;
}
```

입력차수 ‘0’인 노드에서 시작!

큐가 비워질 때 까지 루프내의 동작을 수행!

노드 y와 연결된 노드의 입력차수를 하나씩 감소!

입력차수가 ‘0’인 노드가 생성되면 큐에 저장!

Degrees Summary

- number of edges connected to a vertex
- for undirected graphs
  - sum of all degrees = 2 X edges
  - the number of nodes with odd numbered degrees is even?
- for directed graph
  - sum of in-degree = sum of out-degree

Conectivity

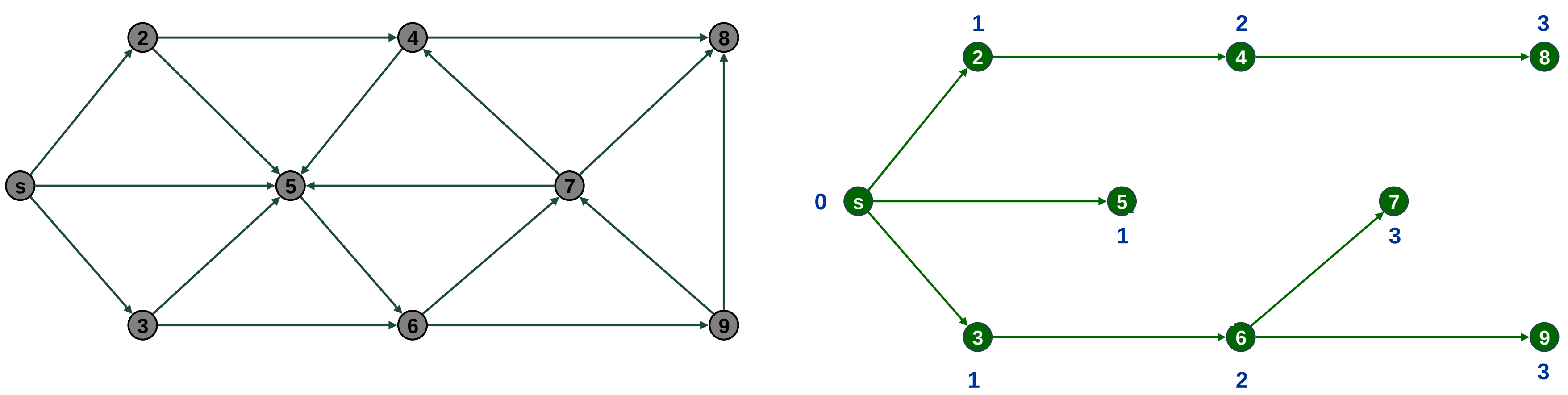
- connected
  - there exists a path between every pair of vertices
- articulation vertex
  - deleting this vertex makes the graph disconnected
  - if a graph does not have any such vertex is biconnected
  - deleting a bridge edge makes the graph disconnected

Cycles

- A tree does not have a cycle
- Eulerian cycle
  - a tour that visits every edge exactly once
- Hamiltonian cycle (path)
  - a tour that visits every vertex exactly once

Spanning Tree

- Given a graph G = (V, E) and tree T = (V, E′)
  - $E' \subset E$
  - for all (u, v) in E′ u, v $\in$ V
  - for all connected graph, there exists a spanning tree

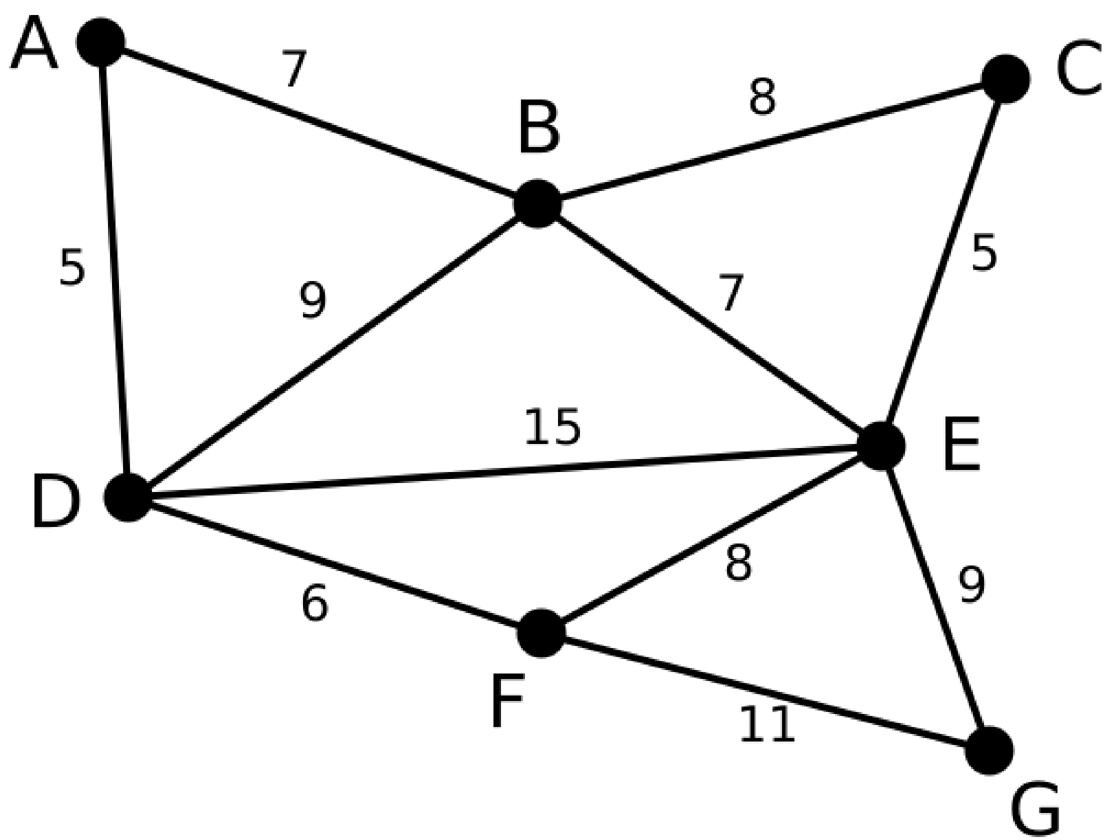


- A spanning tree can be constructed using DFS or BFS

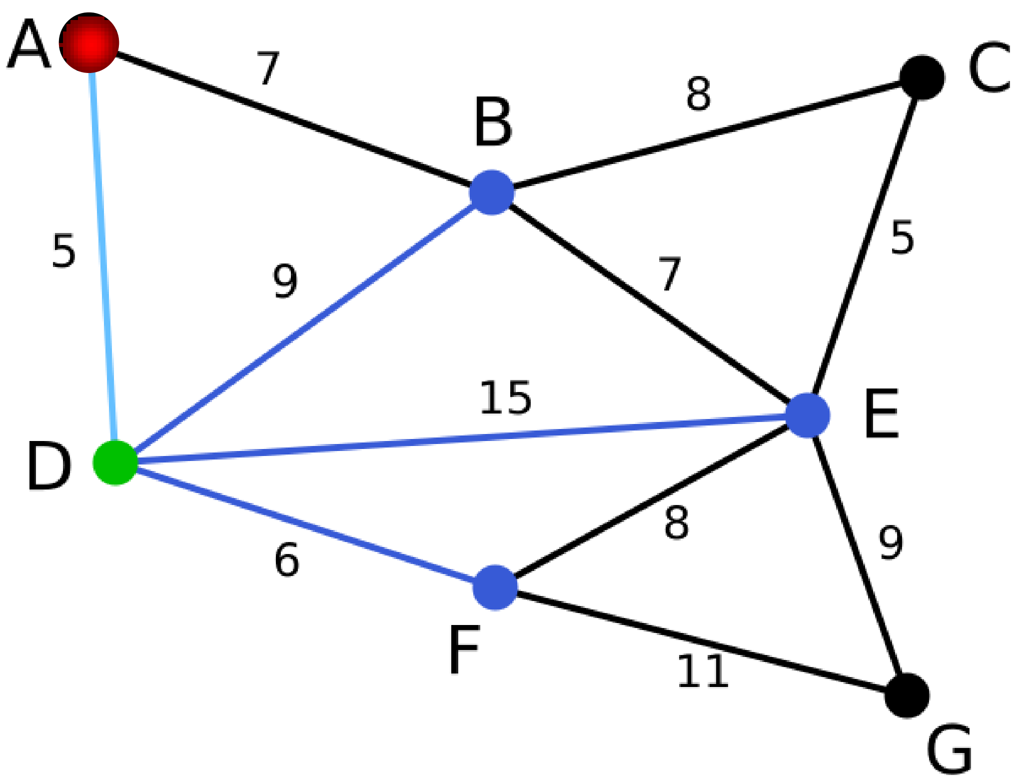
Minimal Spanning Tree

- sum of edge weights is minimal
- if there is no weight
  - number of edges is minimal
- why is it so important?
  - search space is minimal for most problems

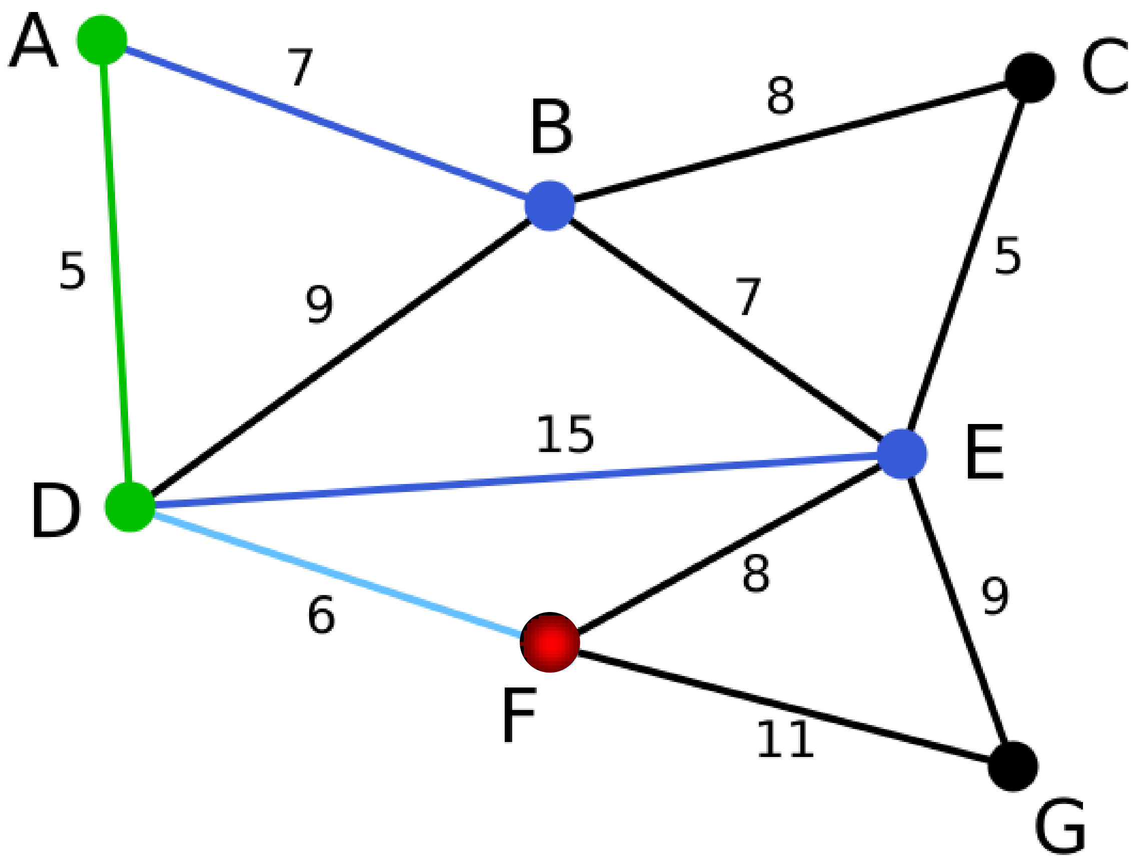
Example of MST: Prim's Algorithm



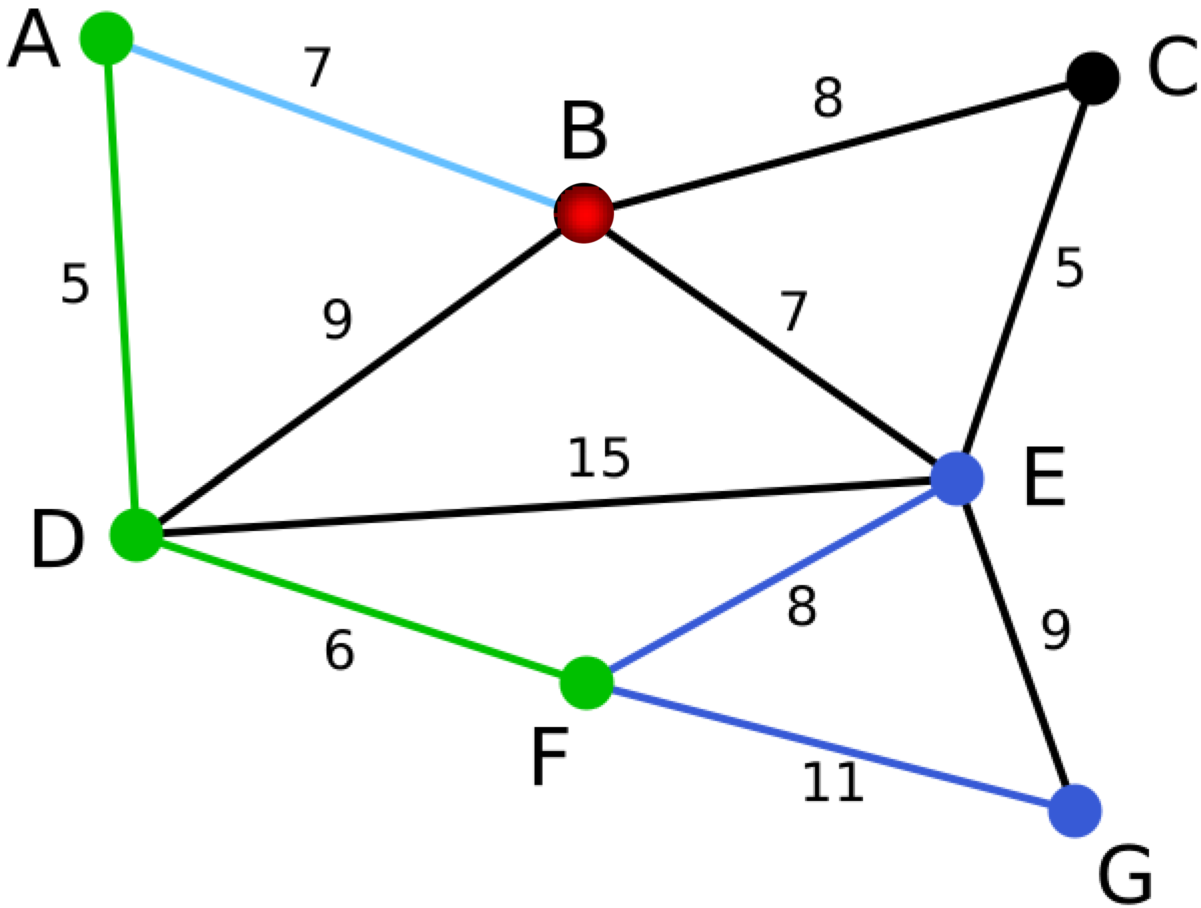
1. Vertex D has been chosen as a starting point
   ① Vertices A, B, E, F are connected to D through a single edge.
   ② A is the nearest to D and thus chosen as the 2nd vertex along with the edge AD



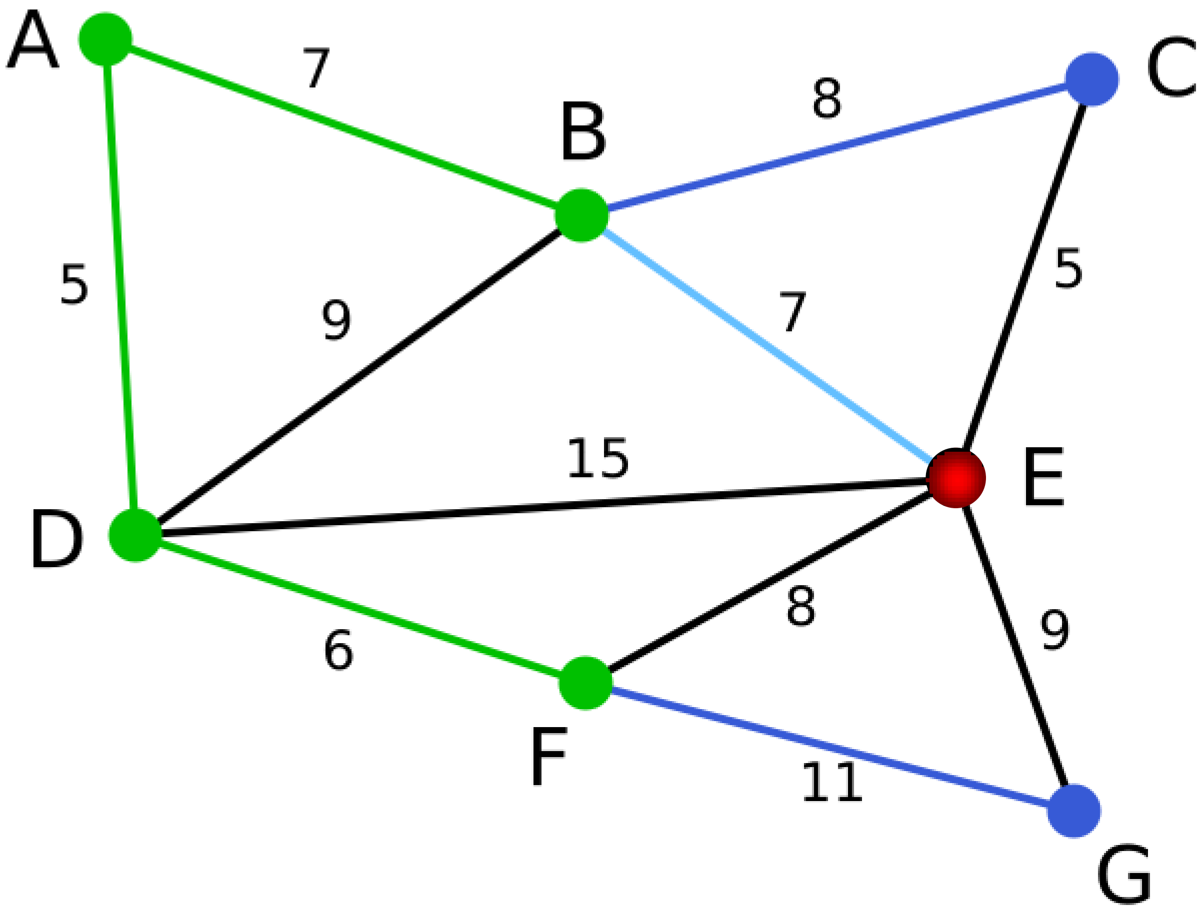
2. The next vertex chosen is the vertex nearest to either D or A. So the vertex F is chosen along with the edge DF



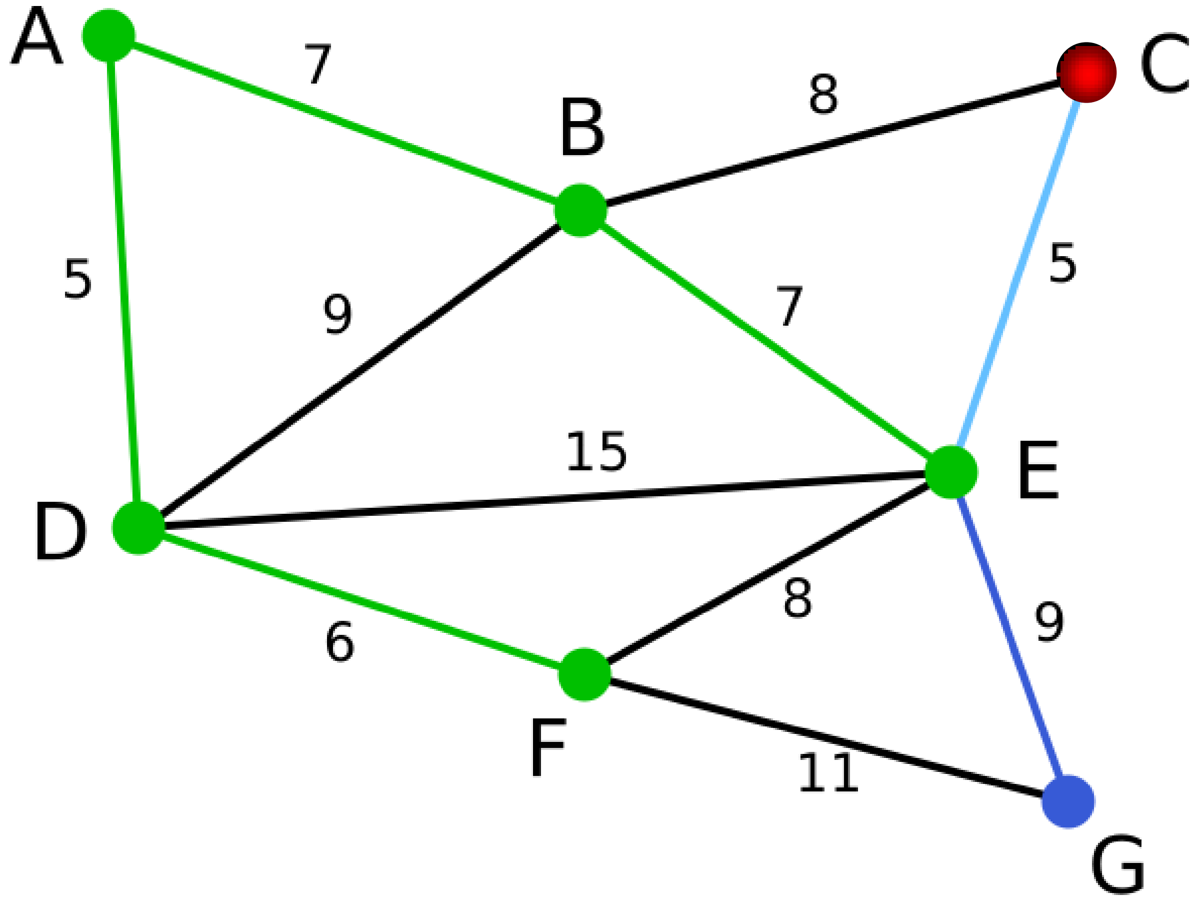
3. same as 2, Vertex B is chosen.



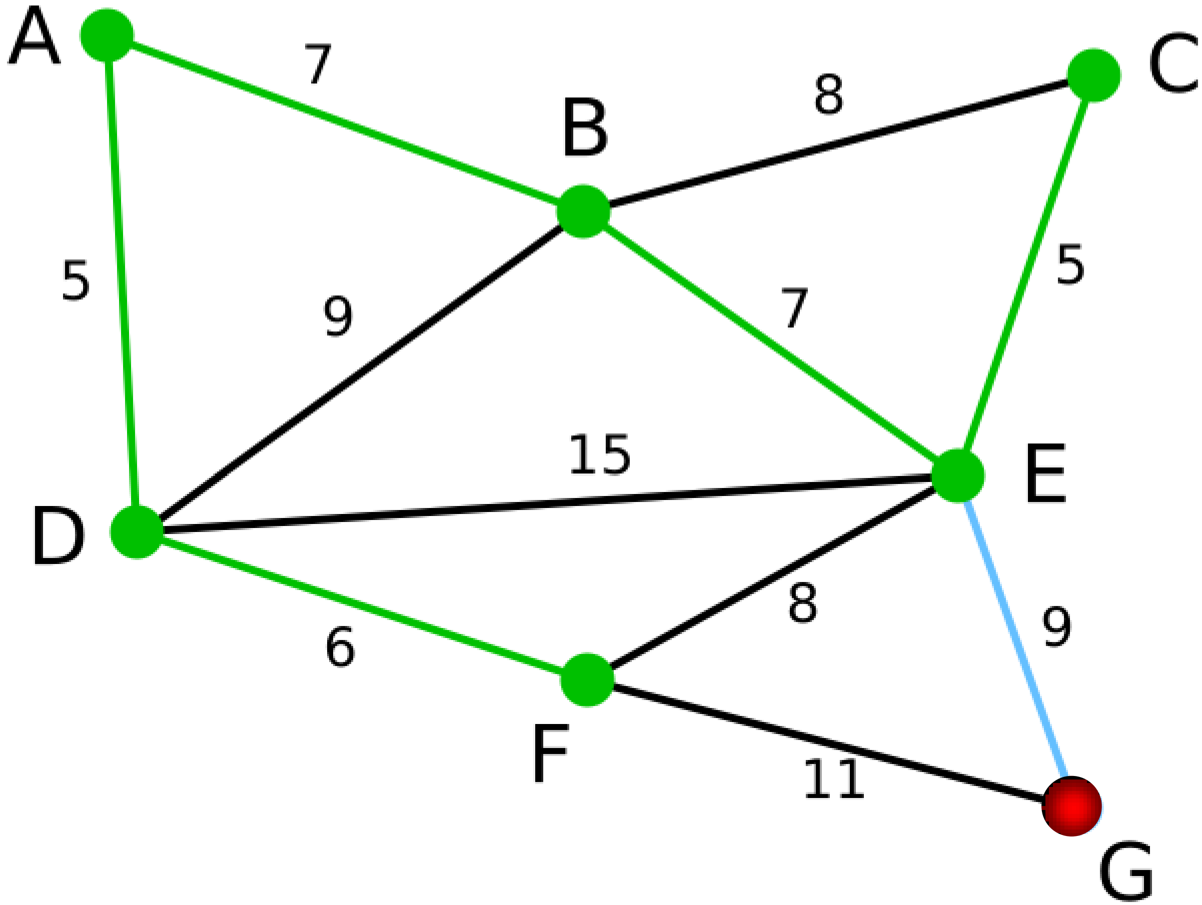
4. among C, E, G, E is chosen.



5. among C, G, C is chosen.

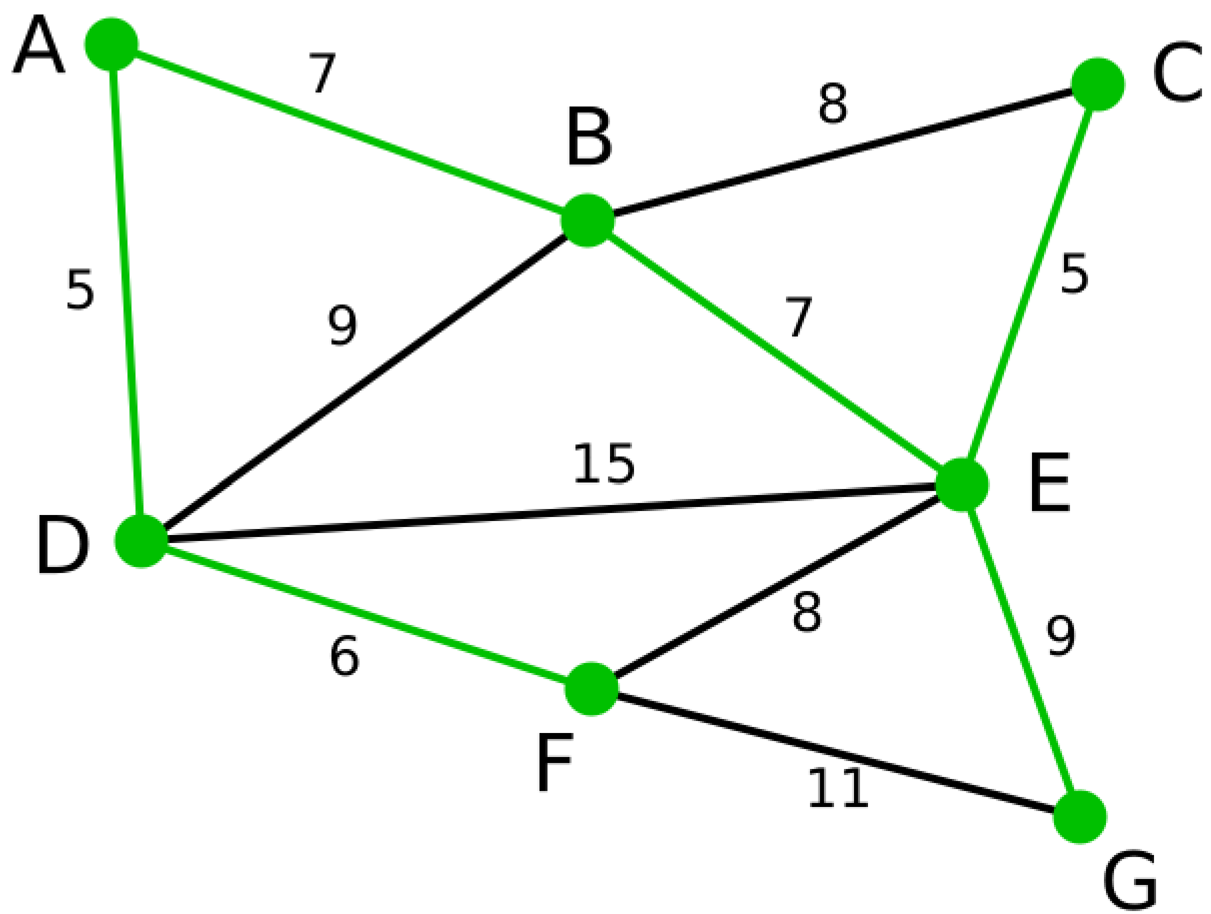


6. G is the only remaining vertex. E is chosen.



7. The finally obtained minimum spanning tree

⇒ the total weight is 39



dist: array of distances from the source to each vertex

edges: array indicating, for a given vertex, which vertex in the tree it is closest to

i: loop index

F: list of finished vertices

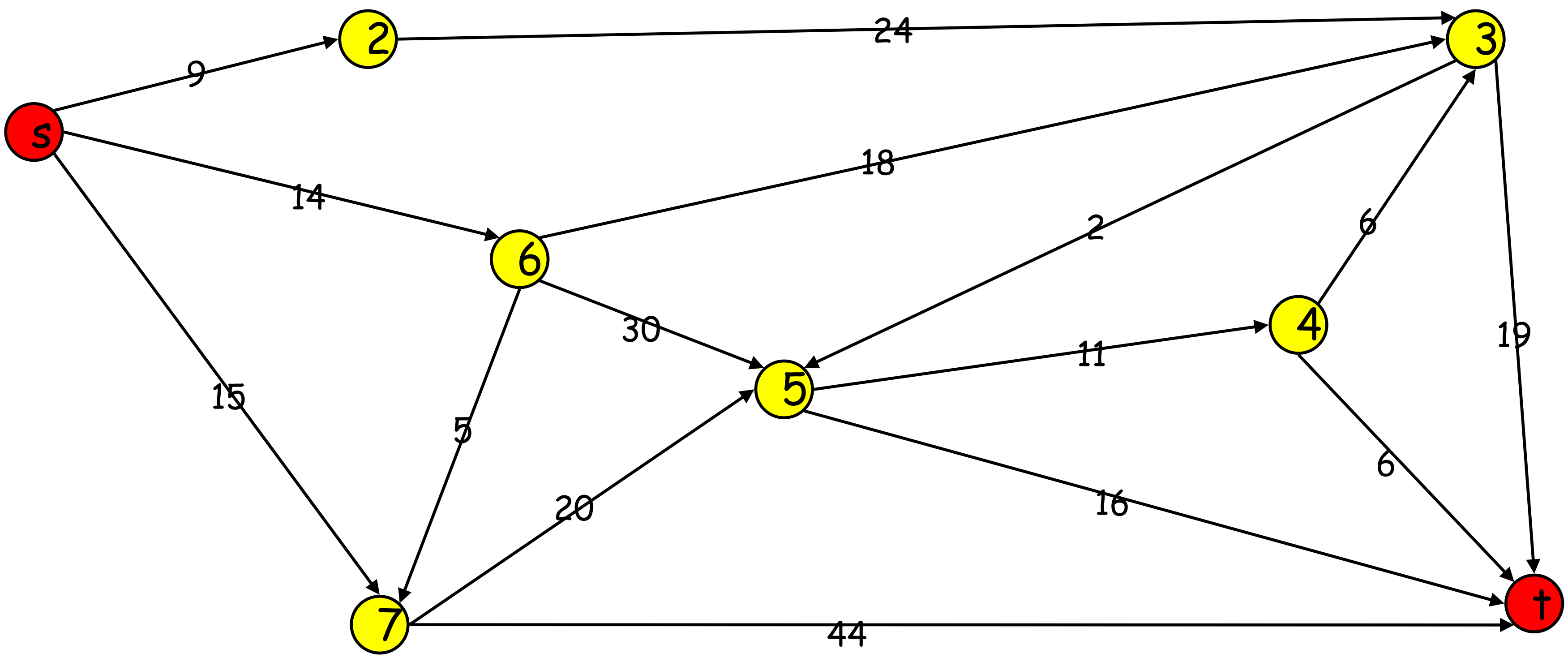
U: list or heap of unfinished vertices

```
/* initialization */
for i=1 to |V|
        dist[i] = INFINITY
        edges[i] = NULL
end for
pick a vertex s to be the seed for the MST
dist[s] = 0
while(F is missing a vertex)
        pick the vertex v in U with the shortest edge and add v to F
        for each edge of v, (v1, v2)
                if (length(v1, v2) < dist[v2])
                        dist[v2] = length(v1, v2)
                        edges[v2] = v1
                        possibly update U, depending on implementation
                end if
        end for
end while
```

/* this loop looks through every neighbor of v and checks to see if that neighbor could reach the MST **more cheaply** through v than by linking a previous vertex

Dijkstra's Algorithm

- Goal: Find the shortest path from s to t



All Pairs Shortest Paths

- Use Dijkstra's method for all the vertices
  - complexity?

- Floyd's method
  - Given the adjacency matrix with vertices numbered (1..n)

$$W[i,j]^k = \min(W[i,j]^{k-1}, W[i,k]^{k-1} + W[k,j]^{k-1})$$

Network Flow (Today's Problem)

- Think edges as pipes
  - what's the maximum flow from node 1 to node 5?