Graph Algorithms – Introduction and Topological Sort

CSE 373

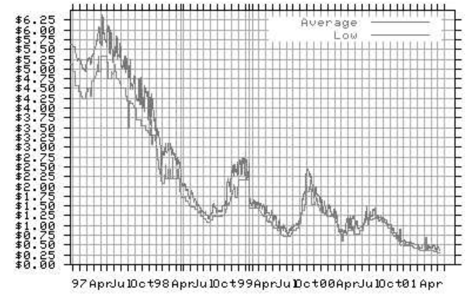
Data Structures

Unit 12

Reading: Sections 9.1 and 9.2

What are graphs?

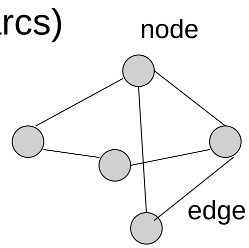
- Yes, this is a graph….



- But we are interested in a different kind of “graph”

Graphs

- Graphs are composed of
  - › Nodes (vertices)
  - › Edges (arcs)

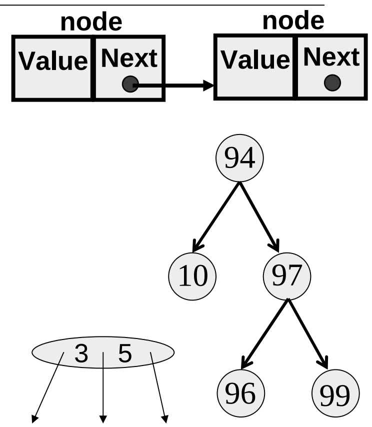


Varieties

- Nodes
  - › Labeled or unlabeled
- Edges
  - › Directed or undirected
  - › Labeled or unlabeled

Motivation for Graphs

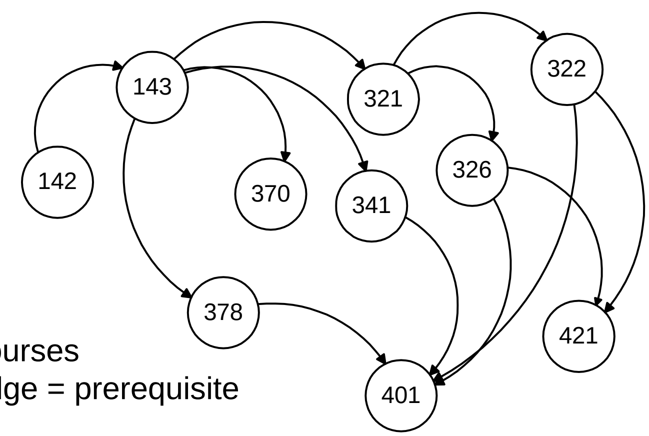
- Consider the data structures we have looked at so far…
- Linked list: nodes with 1 incoming edge + 1 outgoing edge
- Binary trees/heaps: nodes with 1 incoming edge + 2 outgoing edges
- B-trees: nodes with 1 incoming edge + multiple outgoing edges



Motivation for Graphs

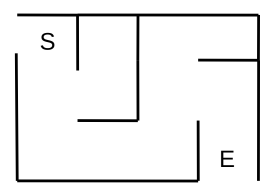
- How can you generalize these data structures?
- Consider data structures for representing the following problems…

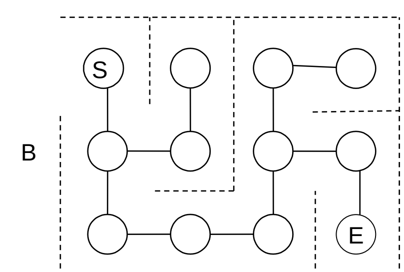
CSE Course Prerequisites at UW



Nodes = courses
Directed edge = prerequisite

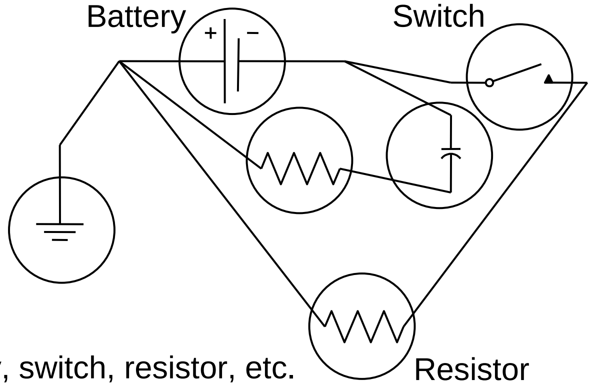
Representing a Maze





Nodes = junctions
Edge = door or passage

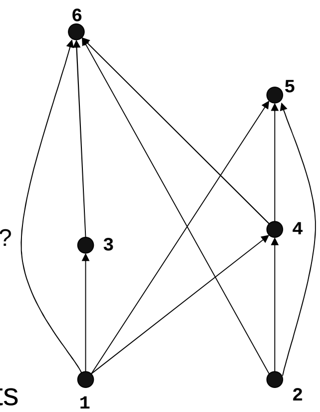
Representing Electrical Circuits



Nodes = battery, switch, resistor, etc.
Edges = connections

Precedence

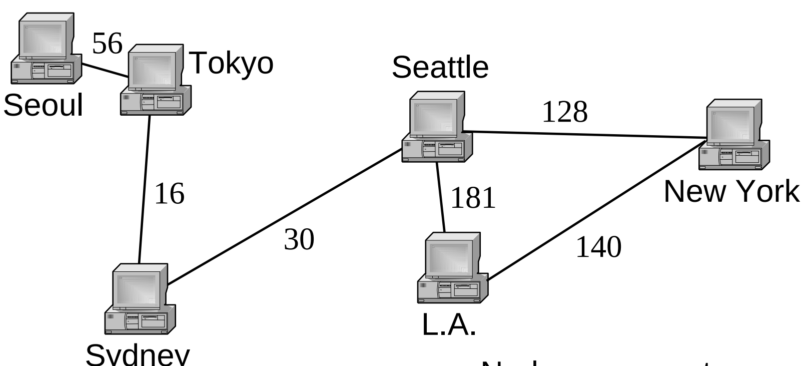
S_1 `a=0;`
S_2 `b=1;`
S_3 `c=a+1`
S_4 `d=b+a;`
S_5 `e=d+1;`
S_6 `e=c+d;`



Which statements must execute before S_6?
S_1, S_2, S_3, S_4

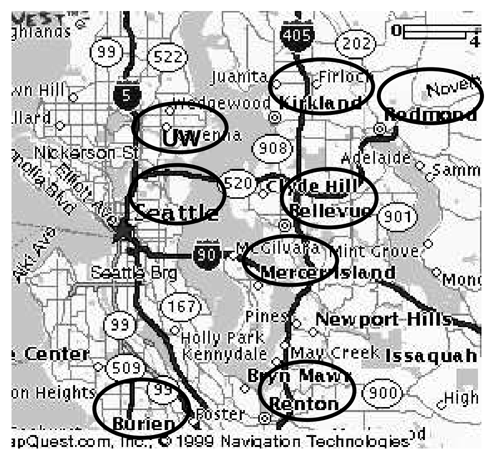
Nodes = statements
Edges = precedence requirements

Information Transmission in a Computer Network



Nodes = computers
Edges = transmission rates

Traffic Flow on Highways



Nodes = cities
Edges = # vehicles on connecting highway

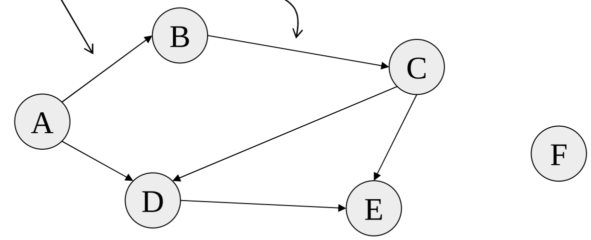
Graph Definition

- A graph is simply a collection of nodes plus edges
  - › Linked lists, trees, and heaps are all special cases of graphs
- The nodes are known as vertices (node = “vertex”)
- Formal Definition: A graph *G* is a pair (*V*, *E*) where
  - › *V* is a set of vertices or nodes
  - › *E* is a set of edges that connect vertices

Graph Example

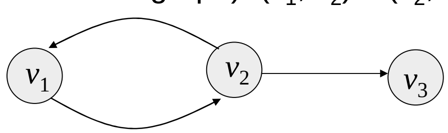
- Here is a directed graph *G* = (*V*, *E*)
  - › Each <u>edge</u> is a pair (v_1, v_2), where v_1, v_2 are vertices in *V*
  - › *V* = {A, B, C, D, E, F}

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

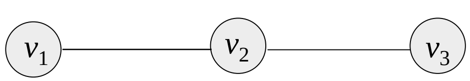


Directed vs Undirected Graphs

- If the order of edge pairs (v_1, v_2) matters, the graph is directed (also called a digraph): (v_1, v_2) ≠ (v_2, v_1)



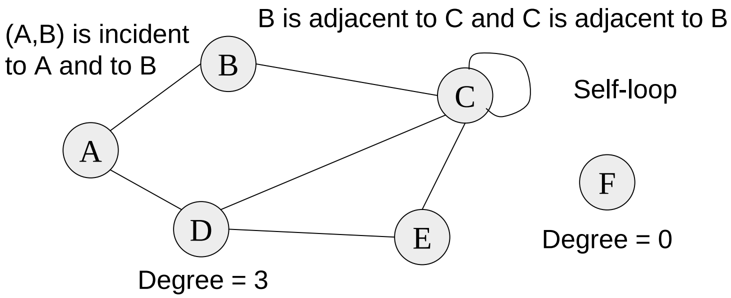
- If the order of edge pairs (v_1, v_2) does not matter, the graph is called an undirected graph: in this case, (v_1, v_2) = (v_2, v_1)



Undirected Terminology

- Two vertices u and v are adjacent in an undirected graph G if {u,v} is an edge in G
  - › edge e = {u,v} is incident with vertex u and vertex v
- The degree of a vertex in an undirected graph is the number of edges incident with it
  - › a self-loop counts twice (both ends count)
  - › denoted with deg(v)

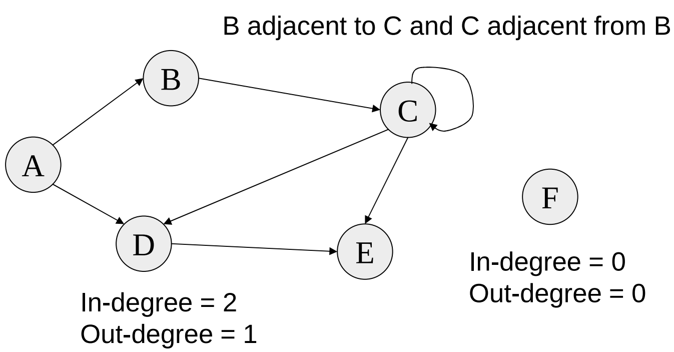
Undirected Terminology



Directed Terminology

- Vertex u is adjacent to vertex v in a directed graph G if (u,v) is an edge in G
  - › vertex u is the initial vertex of (u,v)
- Vertex v is adjacent from vertex u
  - › vertex v is the terminal (or end) vertex of (u,v)
- Degree
  - › in-degree is the number of edges with the vertex as the terminal vertex
  - › out-degree is the number of edges with the vertex as the initial vertex

Directed Terminology



Handshaking Theorem

- Let G=(V,E) be an undirected graph with |E|=m edges. Then

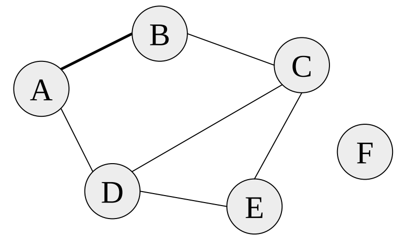
$$2m = \sum_{v \in V} \deg(v)$$

- Proof: Every edge contributes +1 to the degree of each of the two vertices it is incident with
  - › number of edges is exactly half the sum of deg(v)
  - › the sum of the deg(v) values must be even

Graph Representations

- Space and time are analyzed in terms of:
  - Number of vertices, n = |*V*| and
  - Number of edges, m = |*E*|
- There are at least two ways of representing graphs:
  - The *adjacency matrix* representation
  - The *adjacency list* representation

Adjacency Matrix

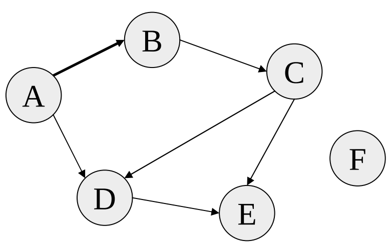


$$M(v, w) = \begin{cases} 1 \text{ if } (v, w) \text{ is in E} \\ 0 \text{ otherwise} \end{cases}$$

$$\begin{array}{c} \\ A \\ B \\ C \\ D \\ E \\ F \end{array} \begin{array}{c} \begin{array}{cccccc} A & B & C & D & E & F \end{array} \\ \begin{pmatrix} 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \end{array}$$

Space = $|V|^2$

Adjacency Matrix for a Digraph



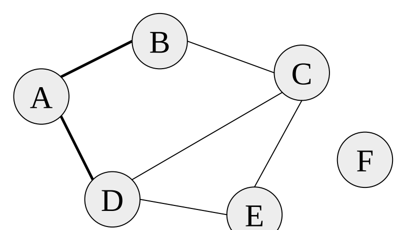
$$M(v, w) = \begin{cases} 1 \text{ if } (v, w) \text{ is in E} \\ 0 \text{ otherwise} \end{cases}$$

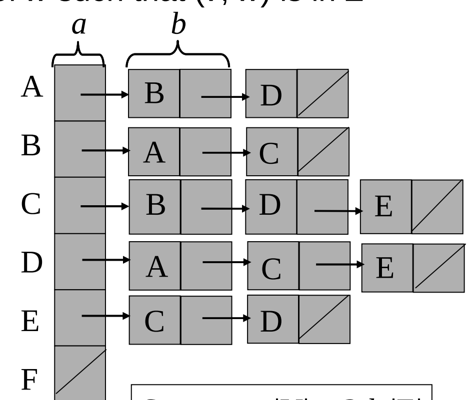
$$\begin{array}{c} \\ A \\ B \\ C \\ D \\ E \\ F \end{array} \begin{array}{c} \begin{array}{cccccc} A & B & C & D & E & F \end{array} \\ \begin{pmatrix} 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \end{array}$$

Space = $|V|^2$

Adjacency List

For each *v* in *V*, *L*(*v*) = list of *w* such that (*v*, *w*) is in *E*

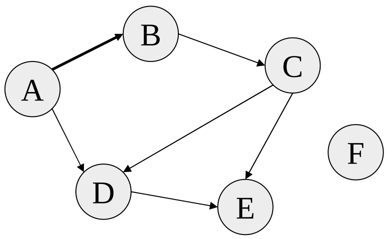


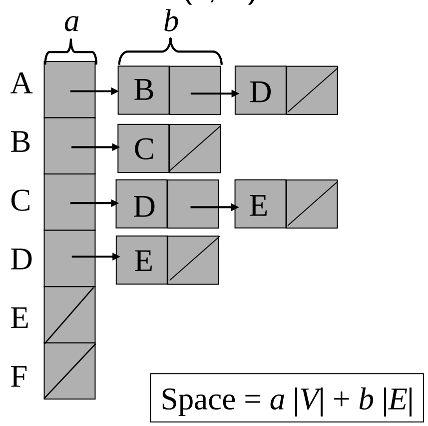


Space = *a* |V| + 2 *b* |E|

Adjacency List for a Digraph

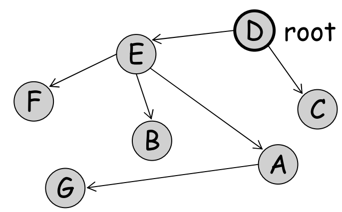
For each *v* in *V*, *L*(*v*) = list of *w* such that (*v*, *w*) is in *E*

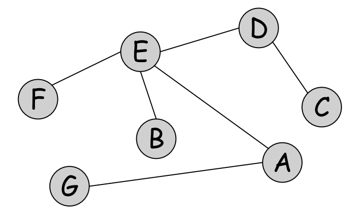




Trees

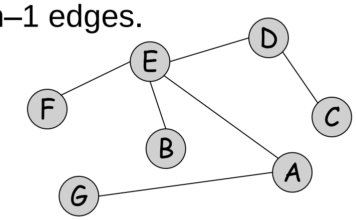
- An undirected graph is a tree if it is connected and contains no cycles.
- A directed graph is a directed tree if it has a root and its underlying undirected graph is a tree.
- $r \in V$ is a root if every vertex $v \in V$ is reachable from r; i.e., there is a directed path which starts in r and ends in v.





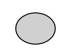
Alternative Definitions of Undirected Trees

- G is cycles-free, but if any new edge is added to G, a cycle is formed.
- for every pair of vertices u,v, there is a unique, simple path from u to v.
- G is connected, but if any edge is deleted from G, the connectivity of G is interrupted.
- G is connected and has n–1 edges.



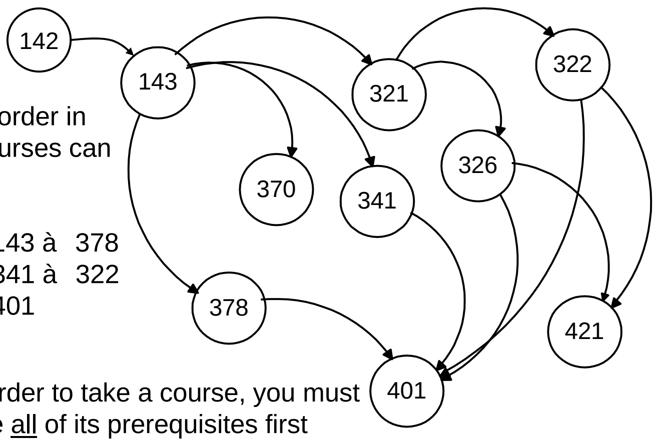
G is a tree ⇒ G is cycle-free and has *n* – 1 edges.

⇒ We show, by induction on n, that if G is a tree (cycle-free and connected), then its number of edges is n–1.

Base: n=1 

Step: Assume that it is true for all n < m, and let G be a tree with m vertices. Delete from G any edge e. By definition (3), G is not connected any more, and is broken into two connected components each of which is cycle-free and therefore is a tree. By the inductive hypothesis, each component has one edge less than the number of vertices. Thus, both have m–2 edges. Add back e, to get m–1.

Topological Sort



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

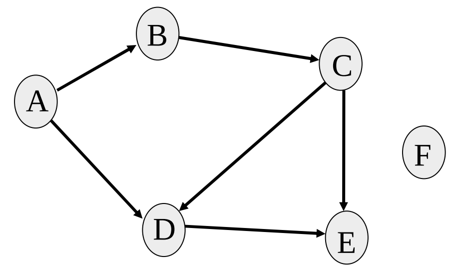
Example: 142 à 143 à 378 à 370 à 321 à 341 à 322 à 326 à 421 à 401

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

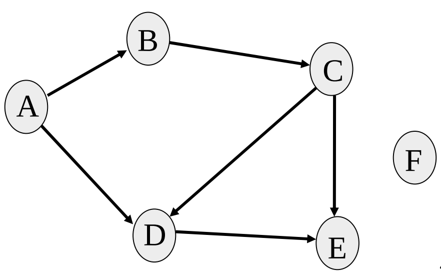
Topological Sort

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

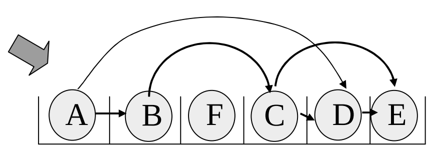
for any edge (*v, w*) in *E*, *v* precedes *w* in the ordering



Topo sort - good example

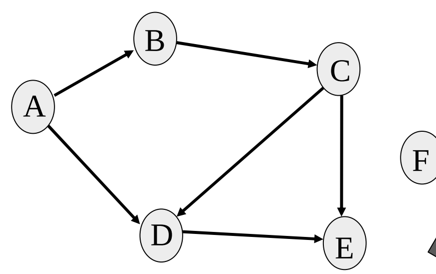


Any linear ordering in which all the arrows go to the right is a valid solution

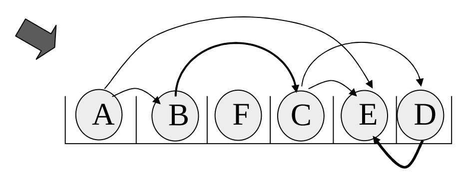


Note that F can go anywhere in this list because it is not connected.
Also the solution is not unique.

Topo sort - bad example



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



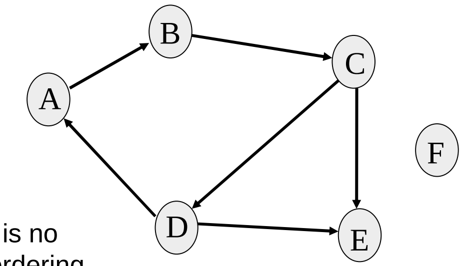
NO!

Paths and Cycles

- Given a digraph G = (V,E), a path is a sequence of vertices $v_1, v_2, \ldots, v_k$ such that:
  - › (v_i, v_{i+1}) in E for all $1 \leq i < k$
  - › path length = number of edges in the path
  - › path cost = sum of costs of participating edges
- A path is a cycle if :
  - › $k > 1$ and $v_1 = v_k$
- G is acyclic if it has no cycles.

Only acyclic graphs can be topologically sorted

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

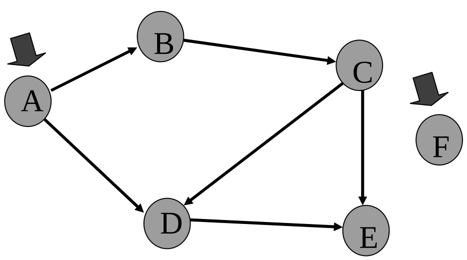


There is no valid ordering of A,B,C,D

Topo sort algorithm - 1

Step 1: Identify vertices that have no incoming edges

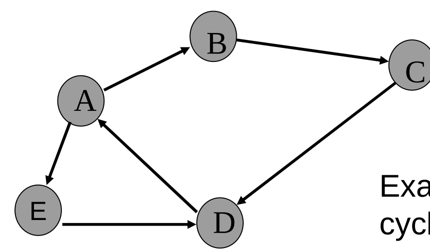
- The "in-degree" of these vertices is zero



Topo sort algorithm - 1a

Step 1: Identify vertices that have no incoming edges

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

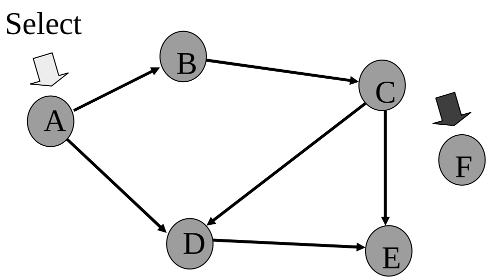


Example of an 'only-cycles' graph

Topo sort algorithm - 1b

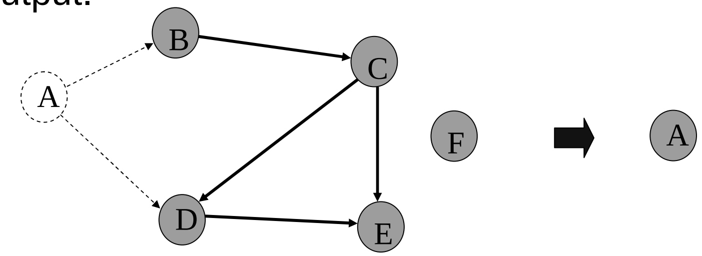
Step 1: Identify vertices that have no incoming edges

- Select one such vertex



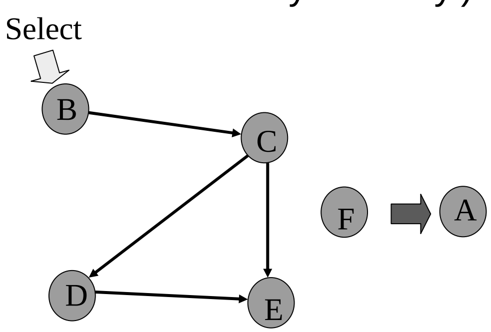
Topo sort algorithm - 2

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



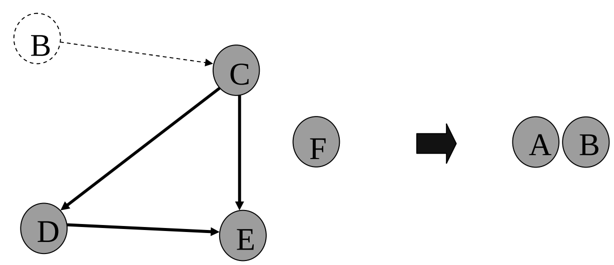
Continue until done

Repeat Step 1 and Step 2 until graph is empty (or until HALT due to cycles-only').



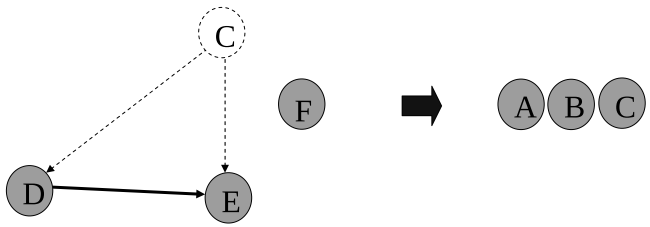
Example (cont') - B

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



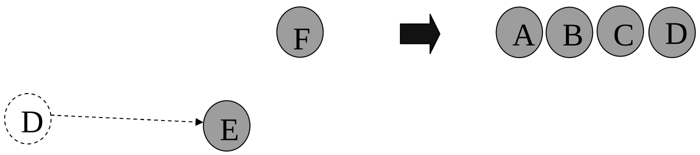
C

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



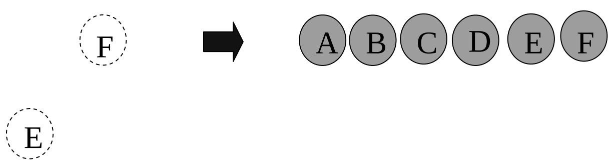
D

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



E, F

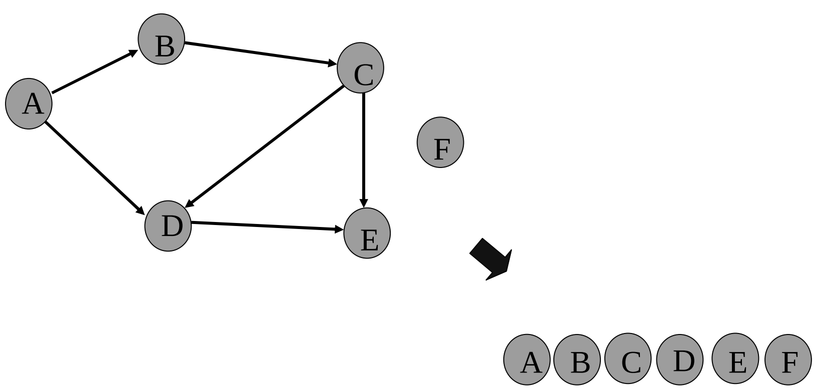
Select E. Copy to sorted list. Delete E and its edges.
Select F. Copy to sorted list. Delete F and its edges.



Yes, we could select F earlier (in any step).

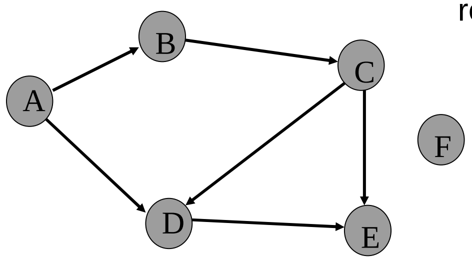
The topological sort is not necessarily unique.

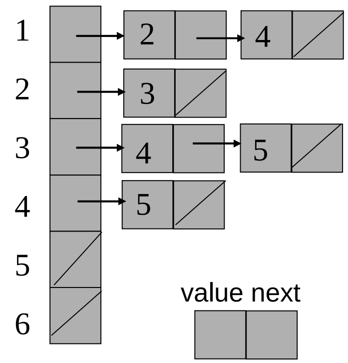
Done



Implementation

Assume adjacency list representation



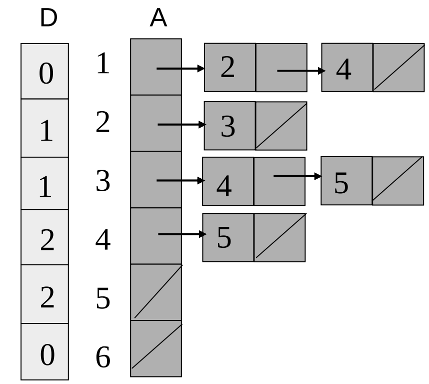


Translation array

1	2	3	4	5	6
A	B	C	D	E	F

Calculate In-degrees

In-Degree array; or add a field to array A



Calculate In-degrees

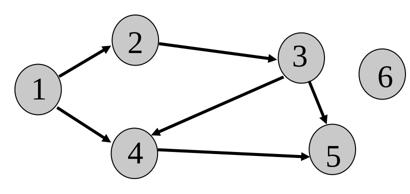
```
for i = 1 to n do D[i] := 0; endfor
for i = 1 to n do
  x := A[i];
  while x ≠ null do
    D[x.value] := D[x.value] + 1;
    x := x.next;
  endwhile
endfor
```

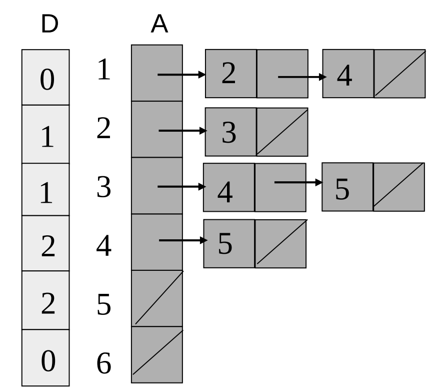
Time Complexity?   O(n+m).

Maintaining Degree 0 Vertices

<u>Key idea</u>: Initialize and maintain a *queue (or stack)* of vertices with In-Degree 0

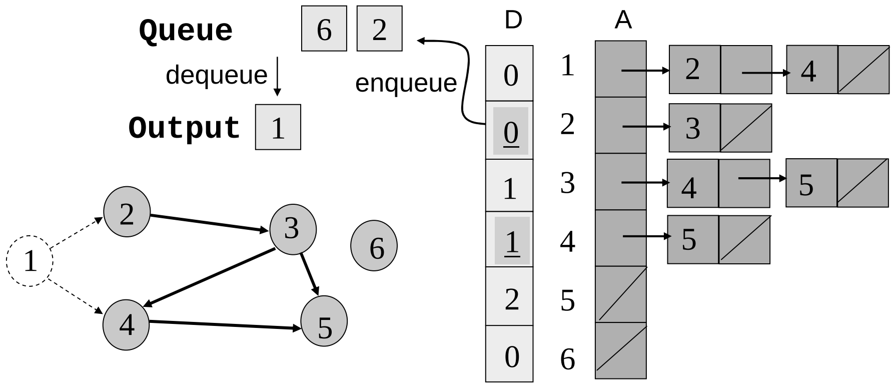
Queue | 1 | 6 |





Topo Sort using a Queue (breadth-first)

After each vertex is output, when updating In-Degree array, *enqueue any vertex whose In-Degree becomes zero*



Topological Sort Algorithm

1. Store each vertex's In-Degree in an array 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.

Some Detail

```
Main Loop
while notEmpty(Q) do
  x := Dequeue(Q)
  Output(x)
  y := A[x];
  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
```

Time complexity? O(out_degree(x)) .

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|) (total out_degree of all vertices)
- For input graph G=(V,E) run time = O(|V| + |E|)
  - › Linear time!

Topo Sort using a Stack (depth-first)

After each vertex is output, when updating In-Degree array, *push any vertex whose In-Degree becomes zero*