Graphs

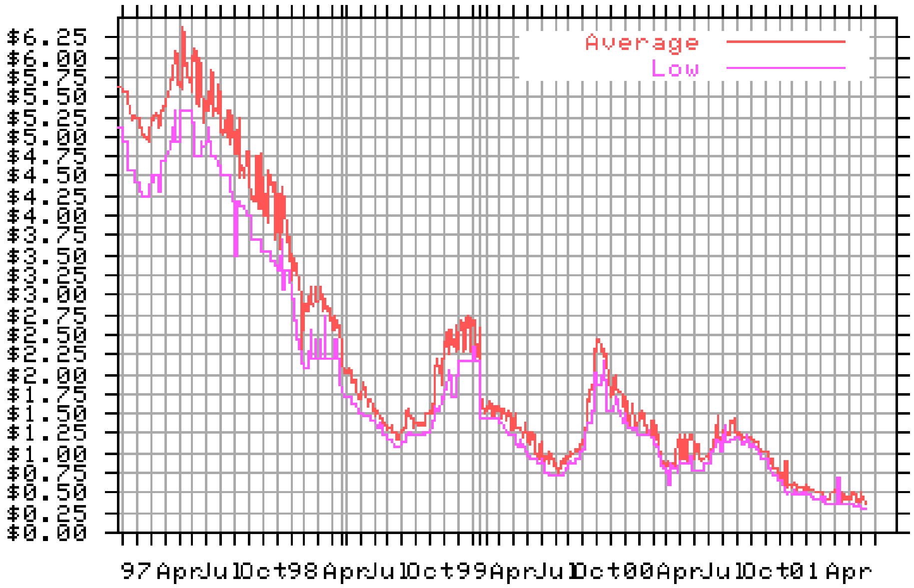
COL 106

Slide Courtesy : http://courses.cs.washington.edu/courses/cse373/

Douglas W. Harder, U Waterloo

What are graphs?

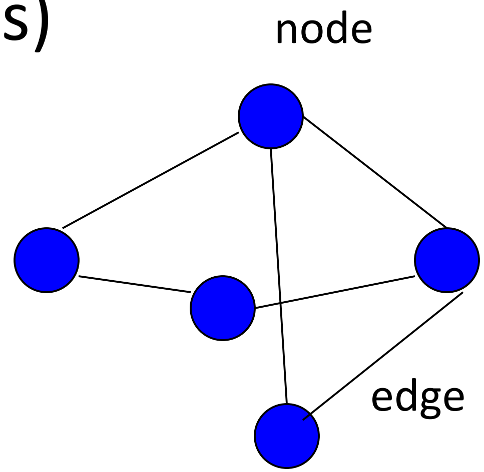
- Yes, this is a graph….



- But we are interested in a different kind of "graph"

Graphs

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

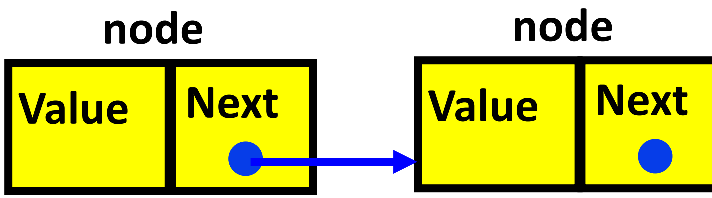


Varieties

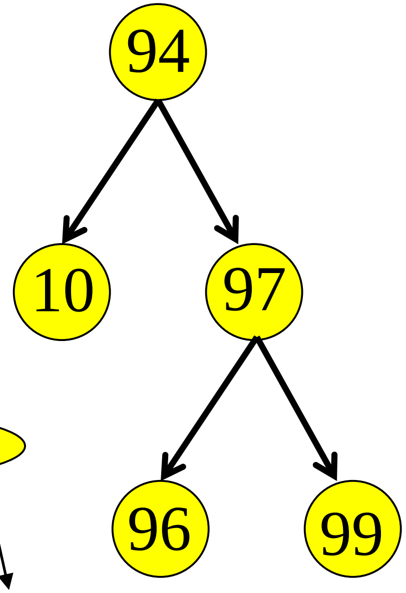
- Nodes
  - Labeled or unlabeled
- Edges
  - Directed or undirected
  - Labeled or unlabeled

Motivation for Graphs

- Consider the data structures we have looked at so far…

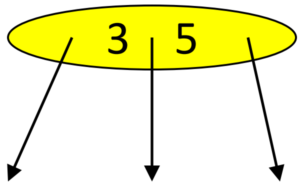


- Linked list: nodes with 1 incoming edge + 1 outgoing edge



- Binary trees/heaps: nodes with 1 incoming edge + 2 outgoing edges

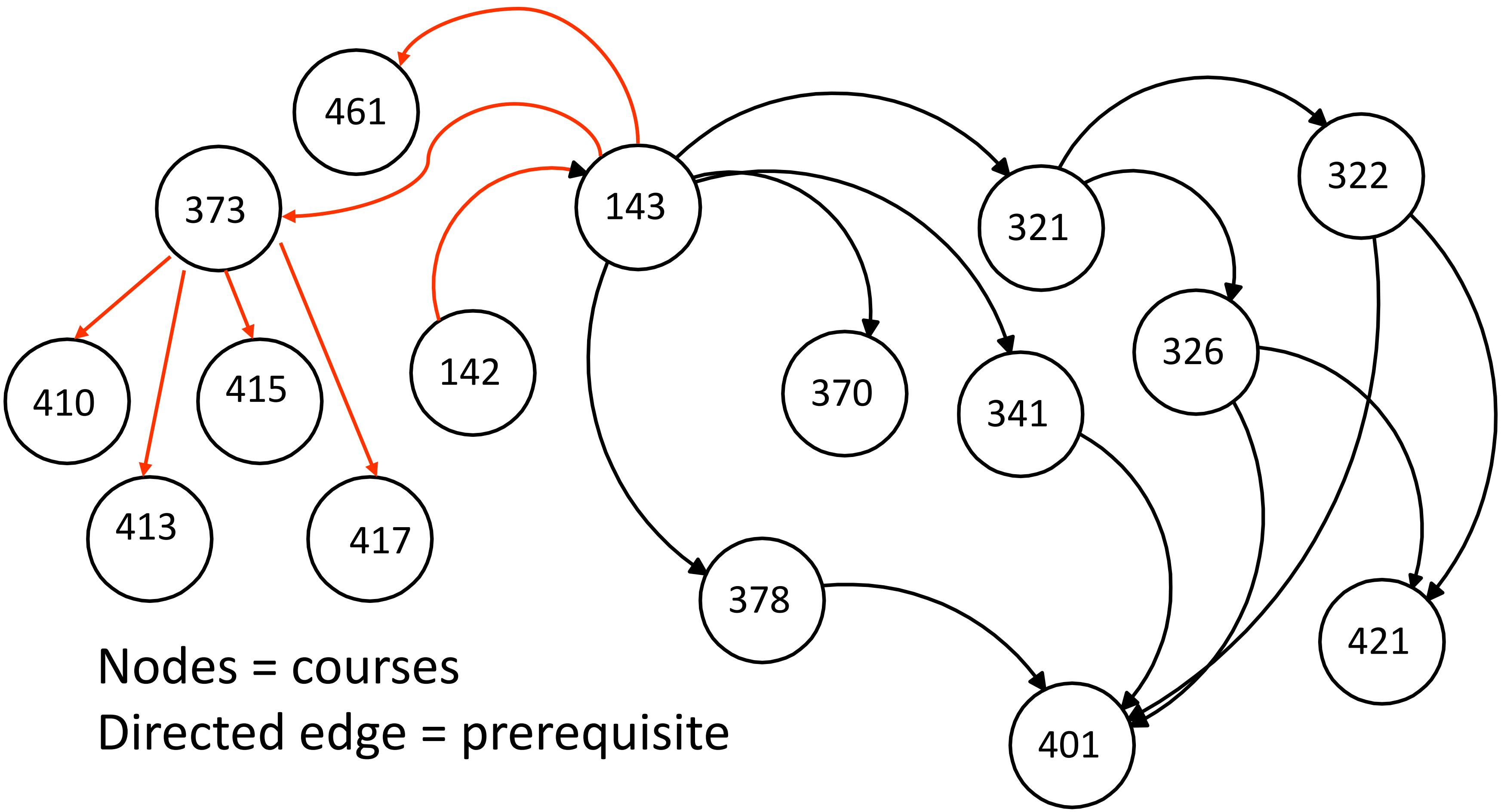
- B-trees: nodes with 1 incoming edge + multiple outgoing edges



Motivation for Graphs

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

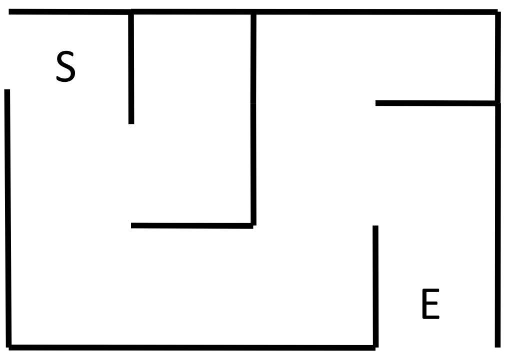
CSE Course Prerequisites

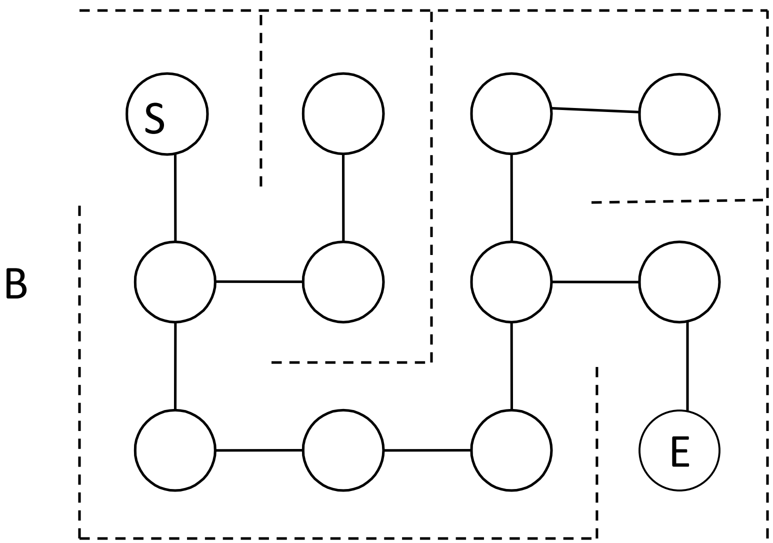


Nodes = courses

Directed edge = prerequisite

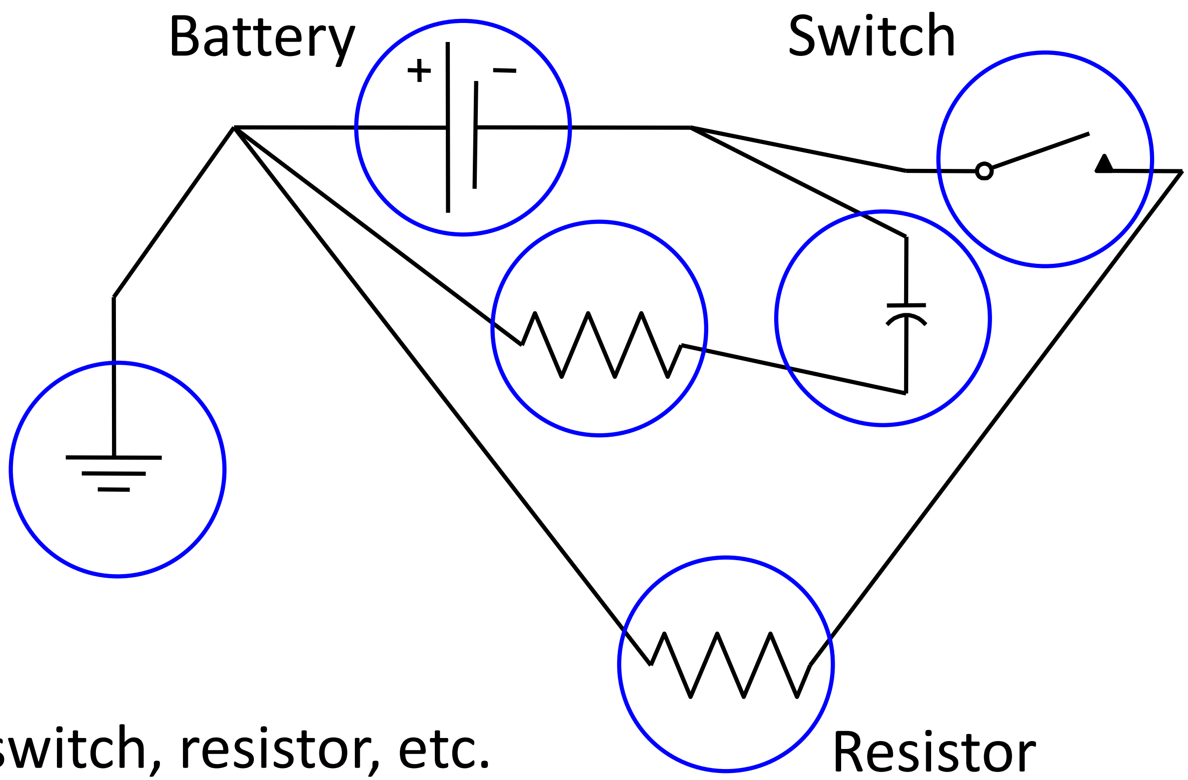
Representing a Maze





Nodes = rooms
Edge = door or passage

Representing Electrical Circuits



Nodes = battery, switch, resistor, etc.

Edges = connections

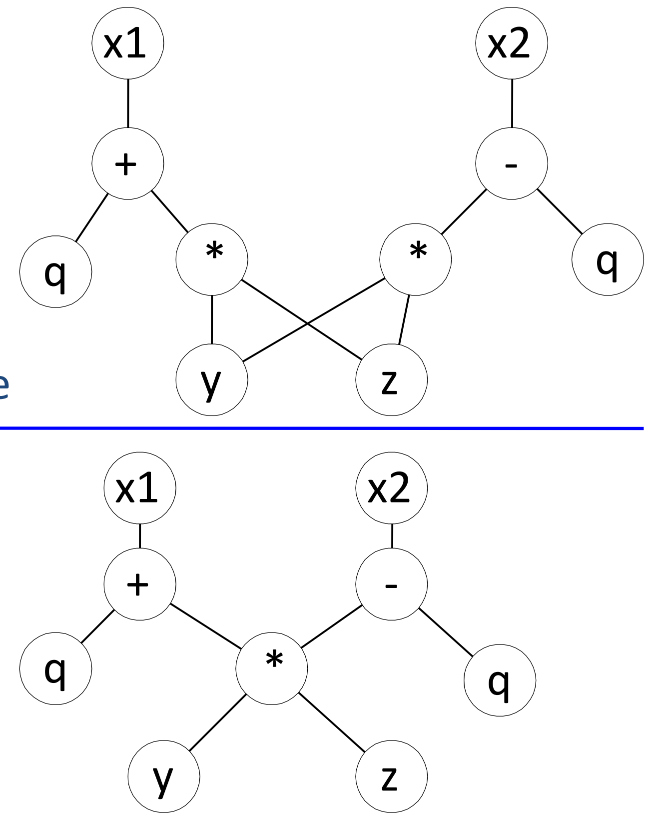
Program statements

```
x1=q+y*z
x2=y*z-q
```

Naive:

y*z calculated twice

common subexpression eliminated:



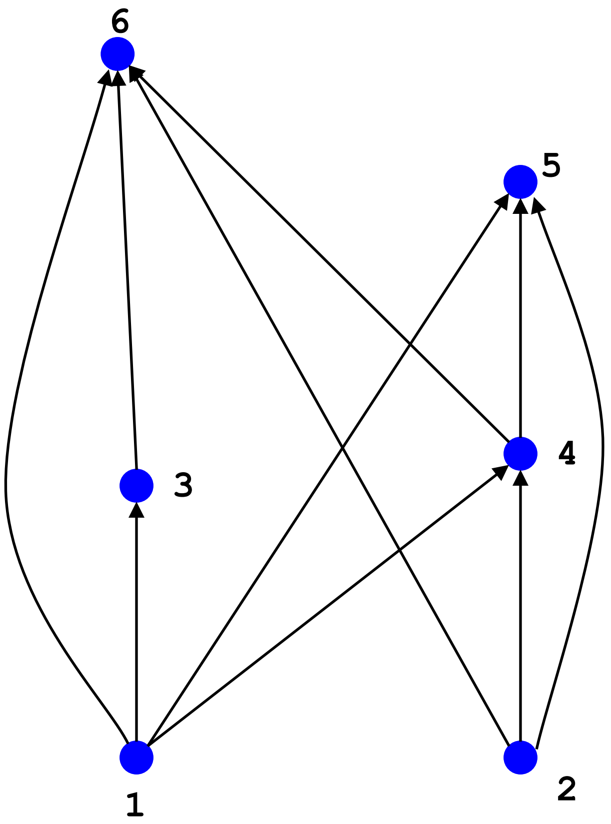
Nodes = symbols/operators
Edges = relationships

Precedence

```
S1  a=0;
S2  b=1;
S3  c=a+1
S4  d=b+a;
S5  e=d+1;
S6  e=c+d;
```

Which statements must execute before S_6?

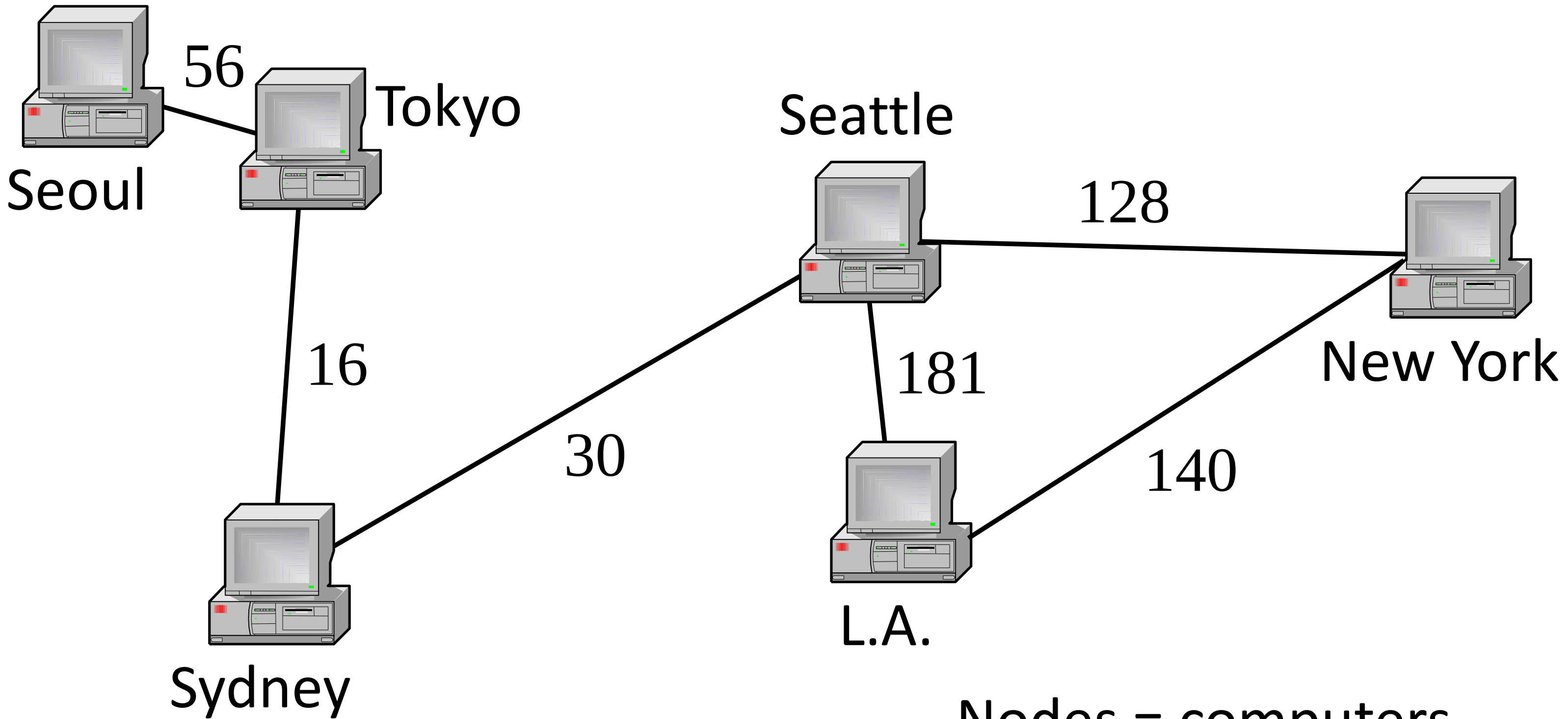
S_1, S_2, S_3, S_4



Nodes = statements

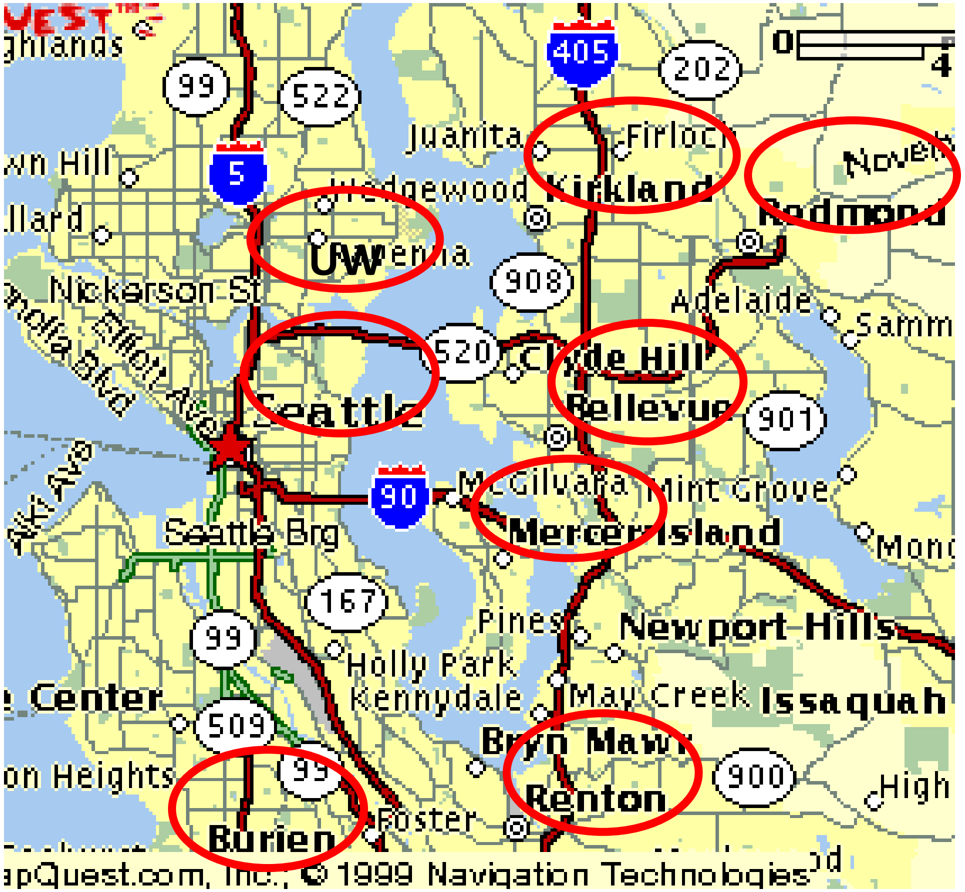
Edges = precedence requirements

Information Transmission in a Computer Network



Nodes = computers
Edges = transmission rates

Traffic Flow on Highways



Nodes = cities

Edges = # vehicles on connecting highway

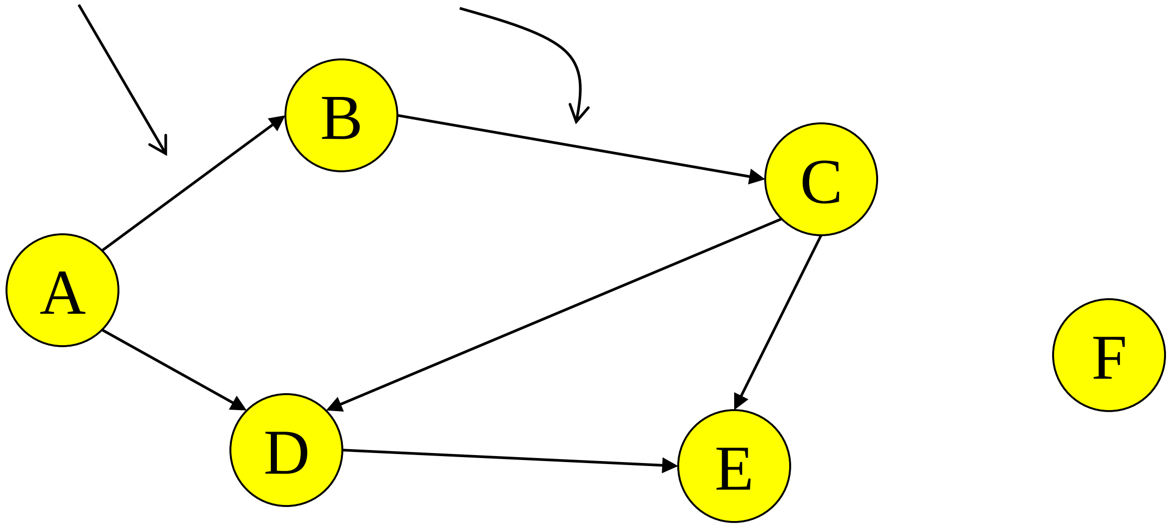
Graph Definition

- 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

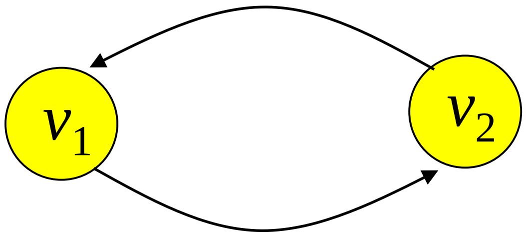
- Here is a directed graph $G = (V, E)$
  - Each edge is a pair (v_1, v_2), where v_1, v_2 are vertices in V
  - $V = \{A, B, C, D, E, F\}$

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

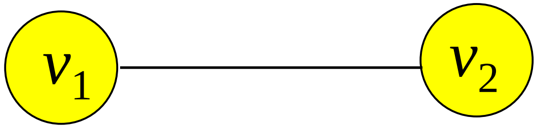


Directed vs Undirected Graphs

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



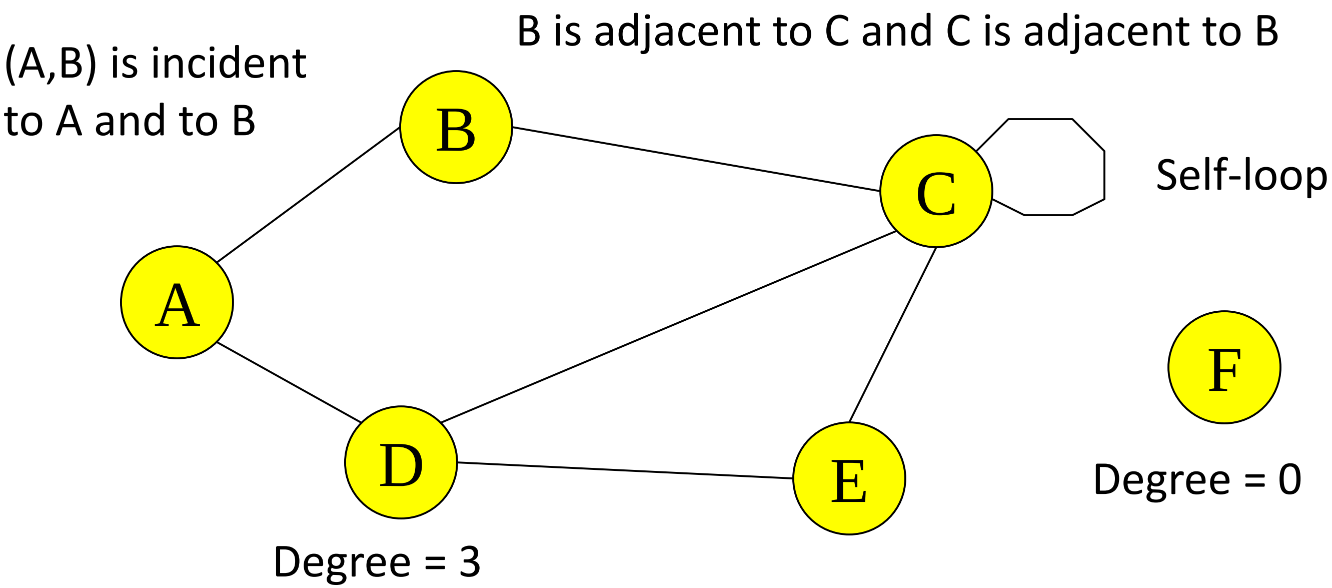
- If the order of edge pairs (v_1, v_2) does not matter, the graph is called an undirected graph: in this case, $(v_1, v_2) = (v_2, v_1)$



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
- A graph is connected if given any two vertices u and v, there is a path from u to 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)

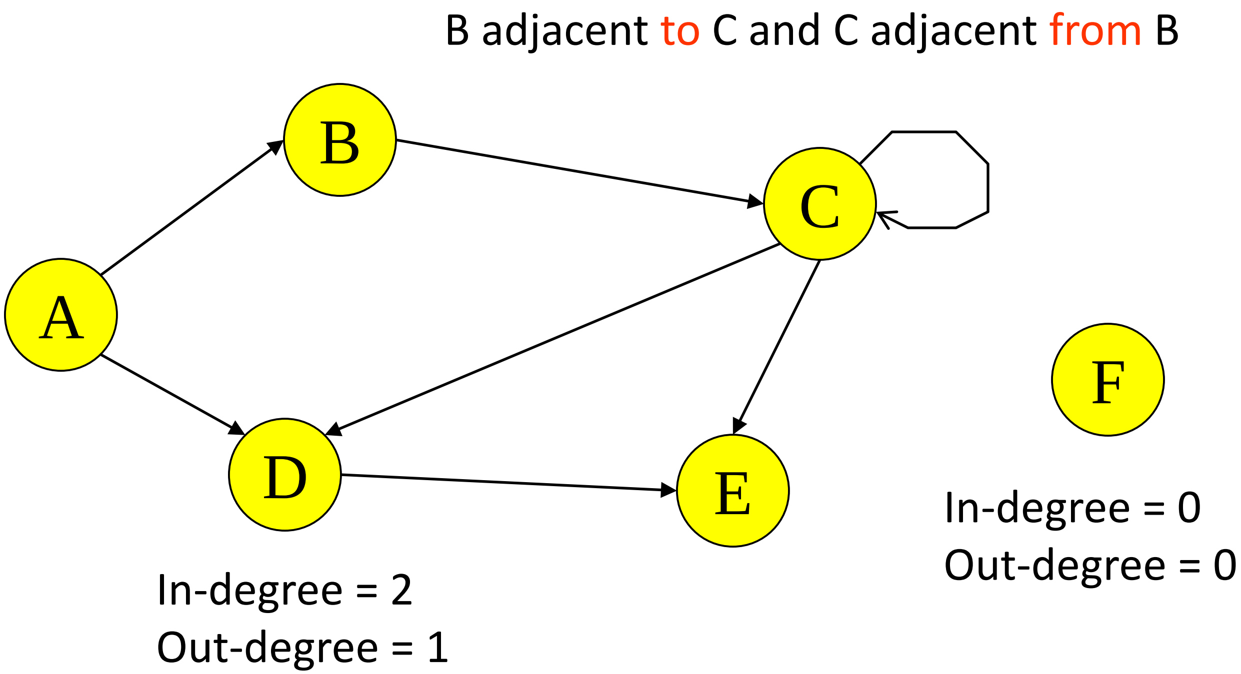
Undirected Terminology



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|=e edges. Then

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

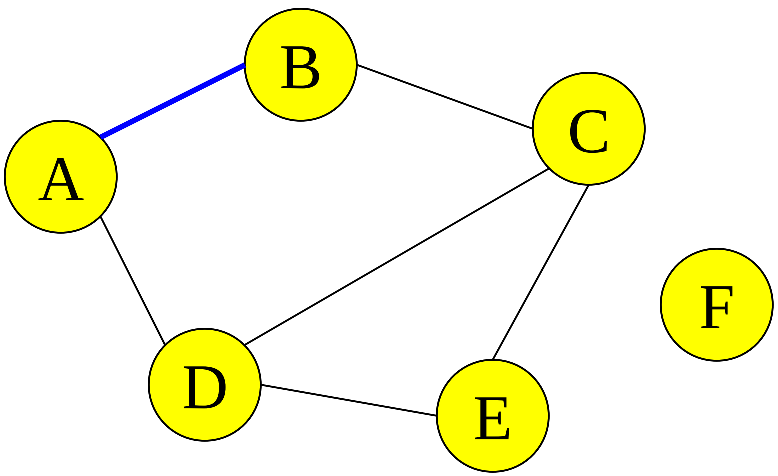
Add up the degrees of all vertices.

- 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 = $|V|$ and
  - Number of edges = $|E|$
- There are at least two ways of representing graphs:
  - The *adjacency matrix* representation
  - The *adjacency list* representation

Adjacency Matrix

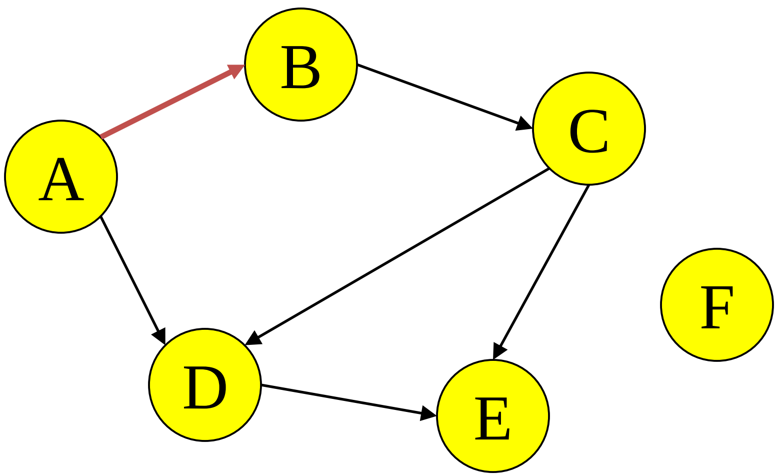


	A	B	C	D	E	F
A	0	1	0	1	0	0
B	1	0	1	0	0	0
C	0	1	0	1	1	0
D	1	0	1	0	1	0
E	0	0	1	1	0	0
F	0	0	0	0	0	0

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

Space = $|V|^2$

Adjacency Matrix for a Digraph



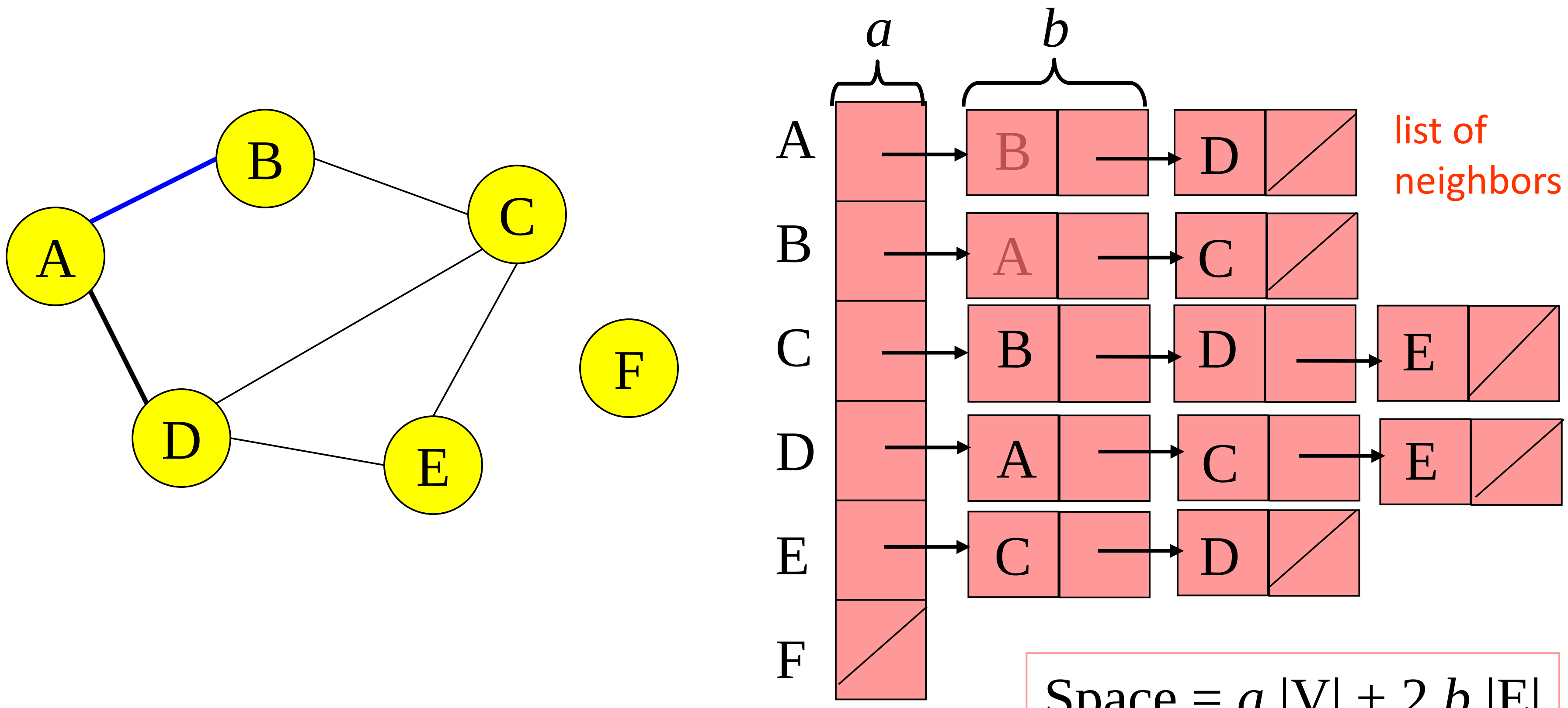
$$M(v, w) = \begin{cases} 1 \text{ if } (v, w) \text{ is in E} \\ 0 \text{ otherwise} \end{cases}$$

	A	B	C	D	E	F
A	0	1	0	1	0	0
B	0	0	1	0	0	0
C	0	0	0	1	1	0
D	0	0	0	0	1	0
E	0	0	0	0	0	0
F	0	0	0	0	0	0

Space = $|V|^2$

Adjacency List

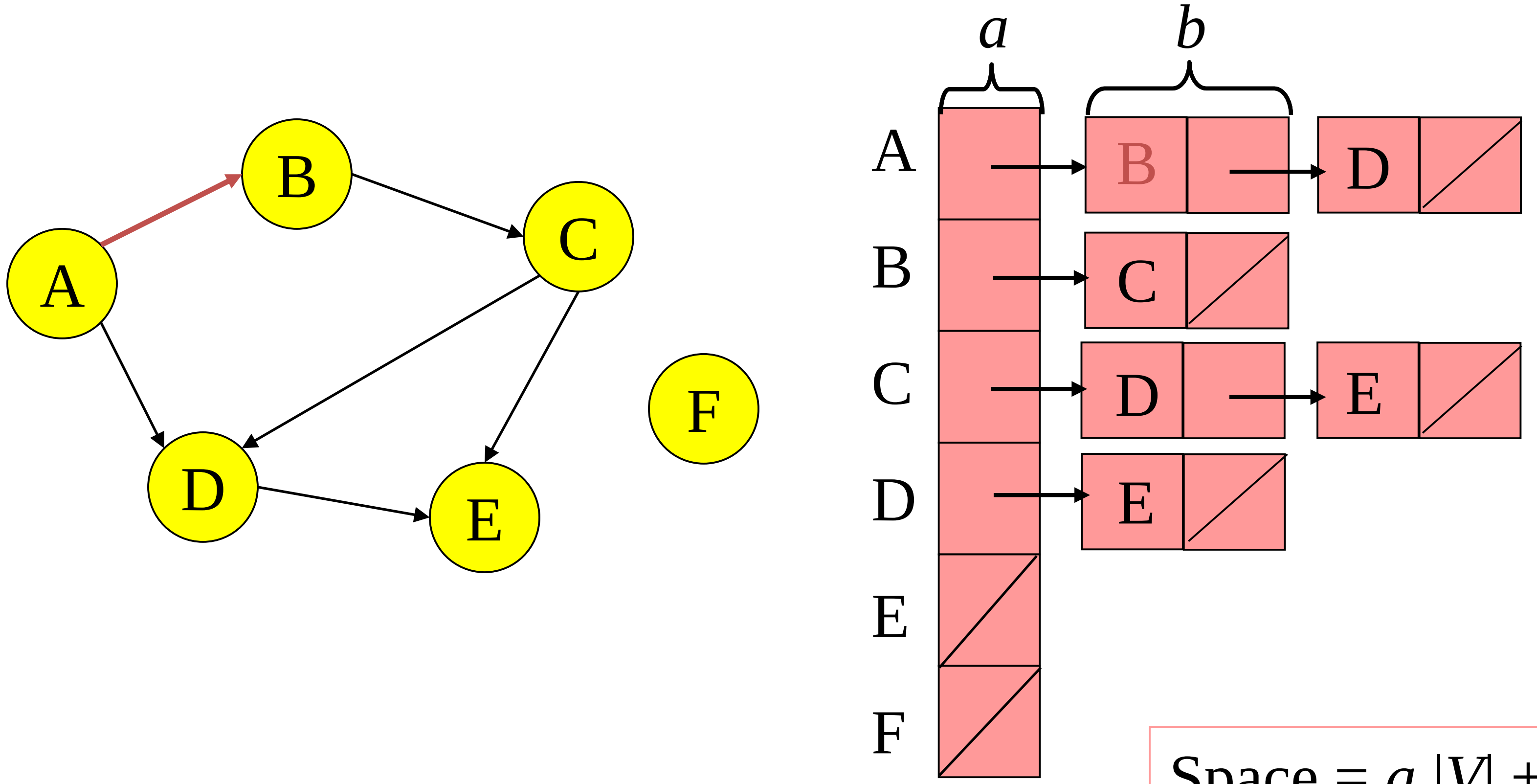
For each *v* in *V*, *L*(*v*) = list of *w* such that (*v*, *w*) is in *E*



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



Space = $a\,|V| + b\,|E|$

Searching in graphs

- Find Properties of Graphs
  - Spanning trees
  - Connected components
  - Bipartite structure
  - Biconnected components
- Applications
  - Finding the web graph – used by Google and others
  - Garbage collection – used in Java run time system

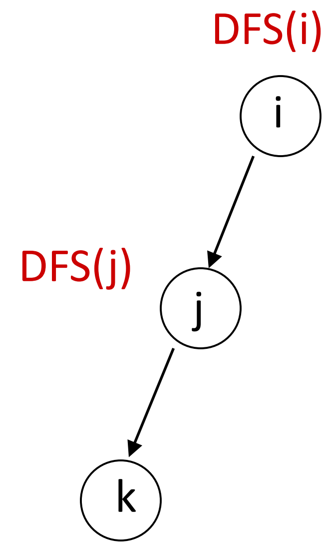
Graph Searching Methodology Depth-First Search (DFS)

- Depth-First Search (DFS)
  - Searches down one path as deep as possible
  - When no nodes available, it backtracks
  - When backtracking, it explores side-paths that were not taken
  - Uses a stack (instead of a queue in BFS)
  - Allows an easy recursive implementation

Depth First Search Algorithm

- Recursive marking algorithm
- Initially every vertex is unmarked

```
DFS(i: vertex)
  mark i;
  for each j adjacent to i do
    if j is unmarked then DFS(j)
end{DFS}
```

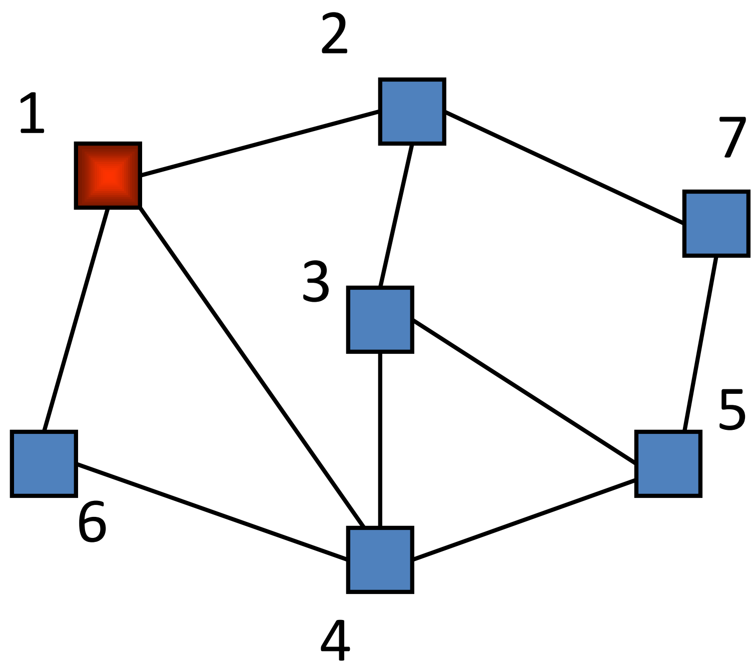


Marks all vertices reachable from i

DFS Application: Spanning Tree

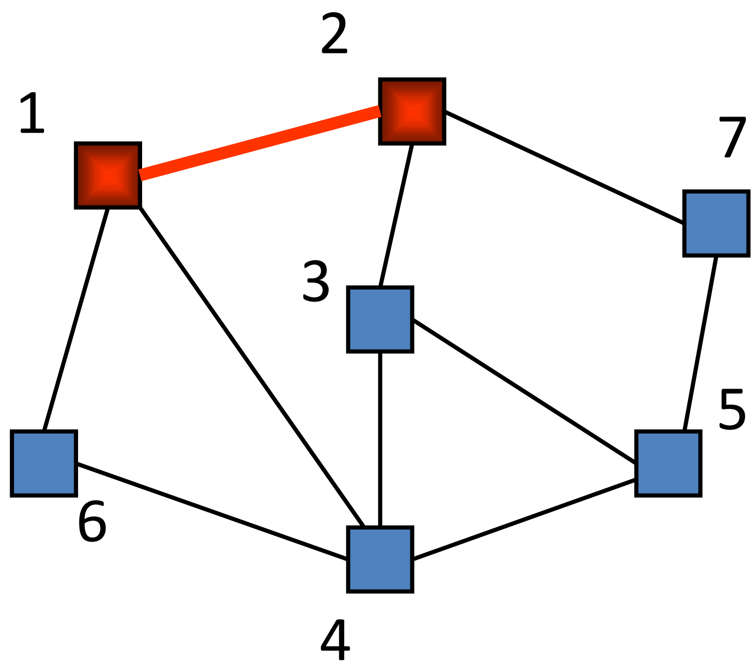
- Given a (undirected) connected graph G(V,E) a spanning tree of G is a graph G'(V',E')
  - V' = V, the tree touches all vertices (spans) the graph
  - E' is a subset of E such that G' is connected and there is no cycle in G'

Example of DFS: Graph connectivity and spanning tree



DFS(1)

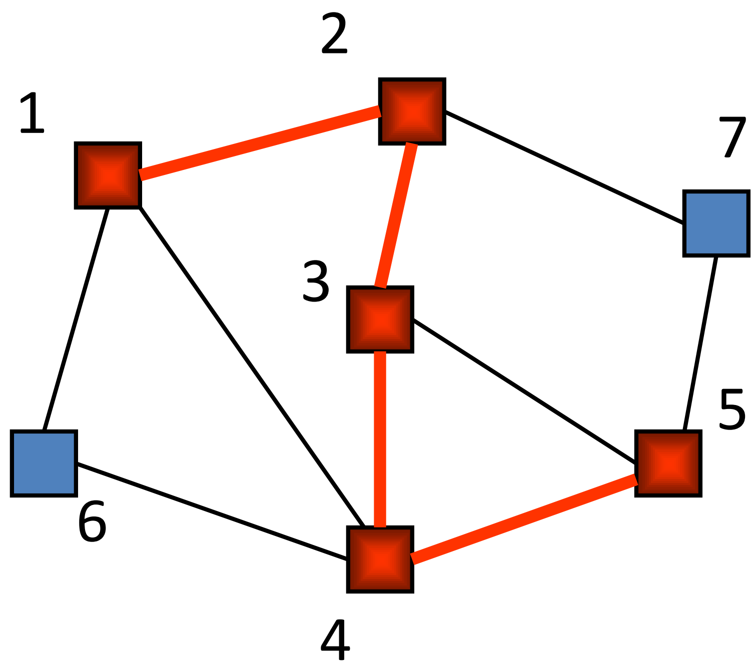
Example Step 2



DFS(1)
 DFS(2)

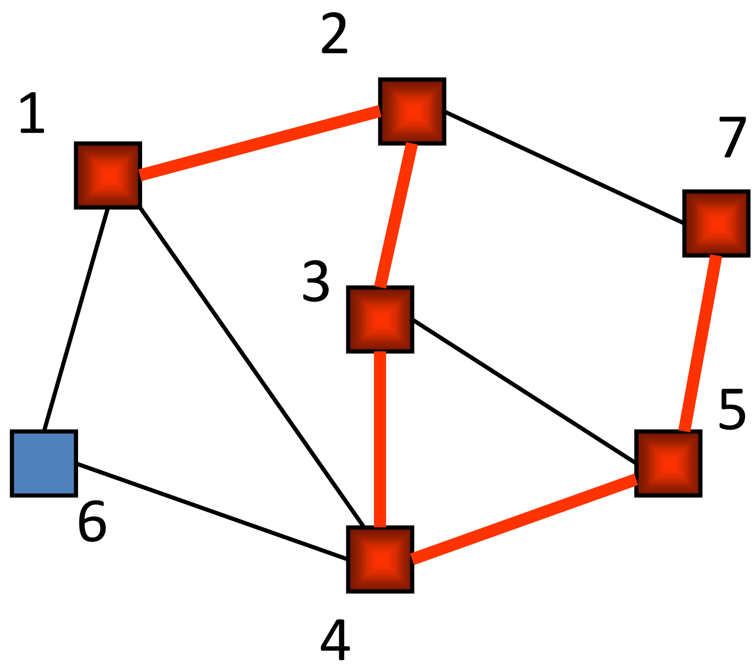
Red links will define the spanning tree if the graph is connected

Example Step 5



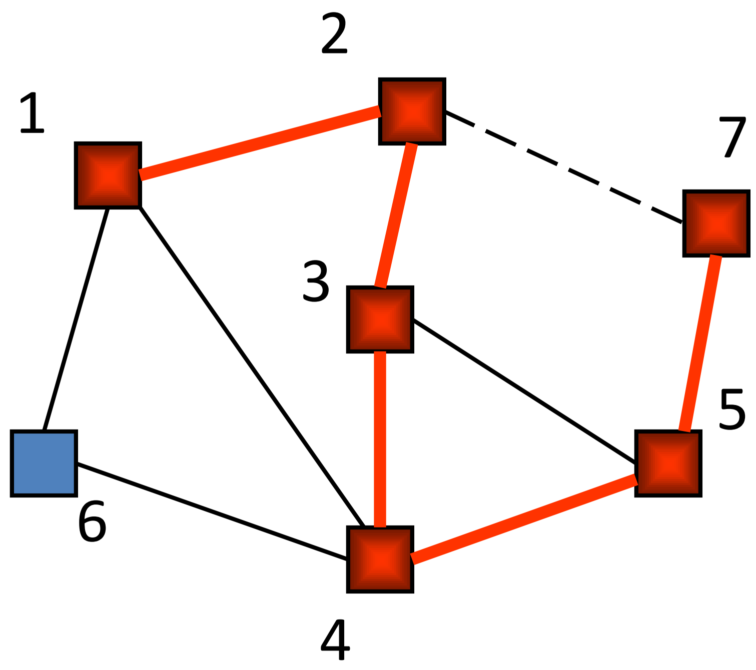
DFS(1)
DFS(2)
DFS(3)
DFS(4)
DFS(5)

Example Steps 6 and 7



DFS(1)
DFS(2)
DFS(3)
DFS(4)
DFS(5)
~~DFS(3)~~
DFS(7)

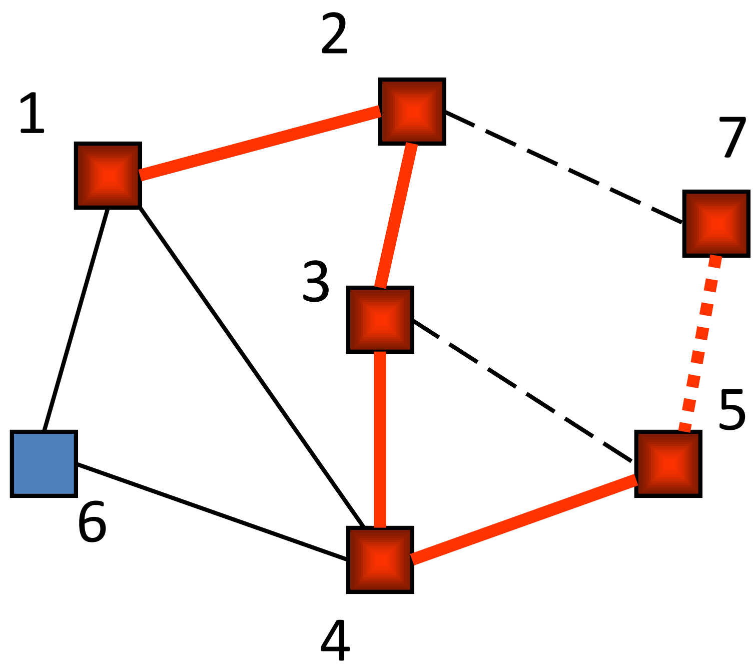
Example Steps 8 and 9



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

Now back up.

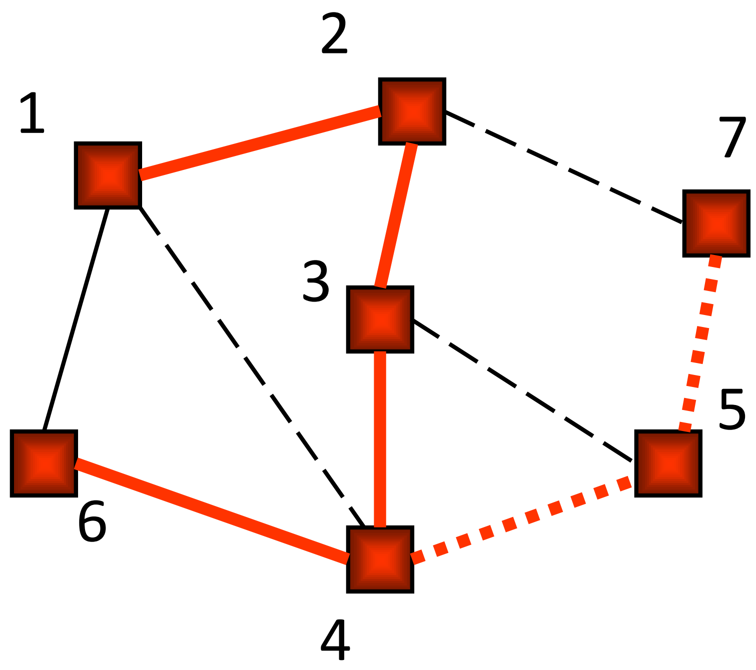
Example Step 10 (backtrack)



DFS(1)
 DFS(2)
  DFS(3)
   DFS(4)
    DFS(5)

Back to 5,
but it has no
more neighbors.

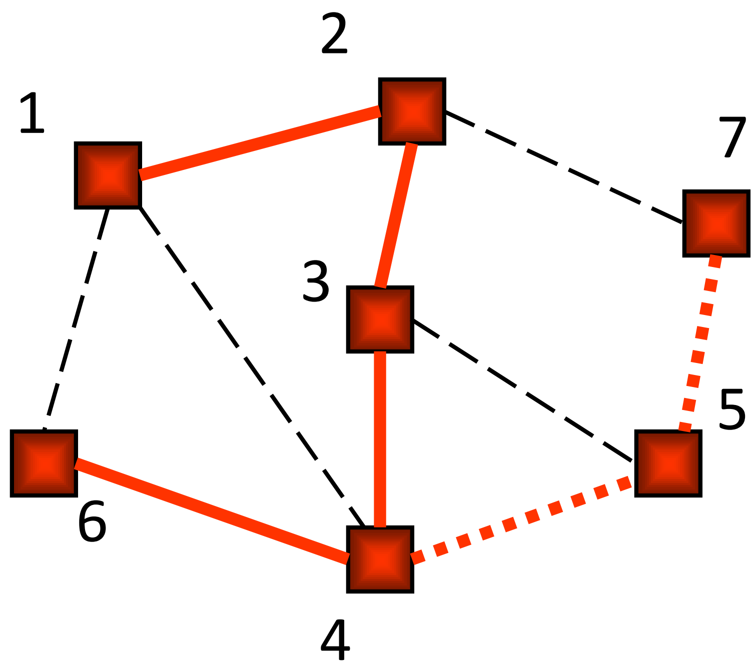
Example Step 12



DFS(1)
DFS(2)
DFS(3)
DFS(4)
DFS(6)

Back up to 4.
From 4 we can
get to 6.

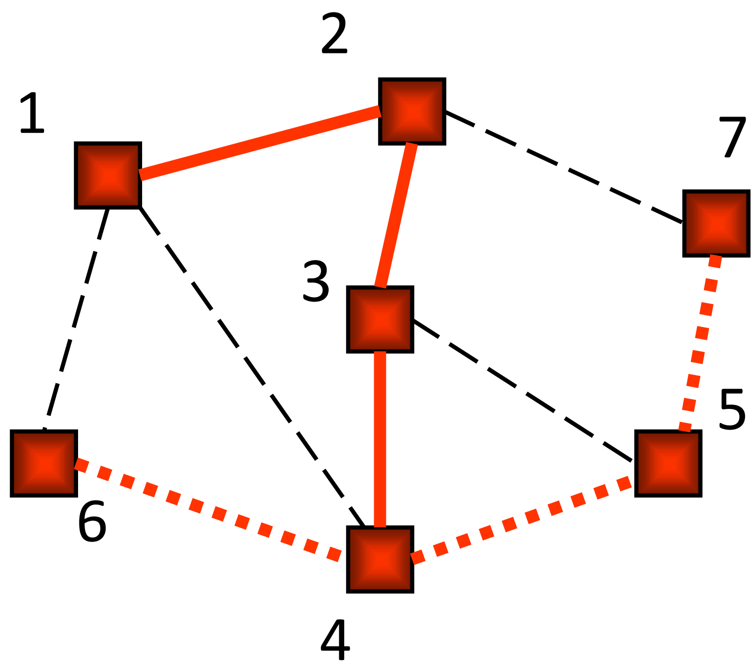
Example Step 13



DFS(1)
DFS(2)
DFS(3)
DFS(4)
DFS(6)

From 6 there is nowhere new to go. Back up.

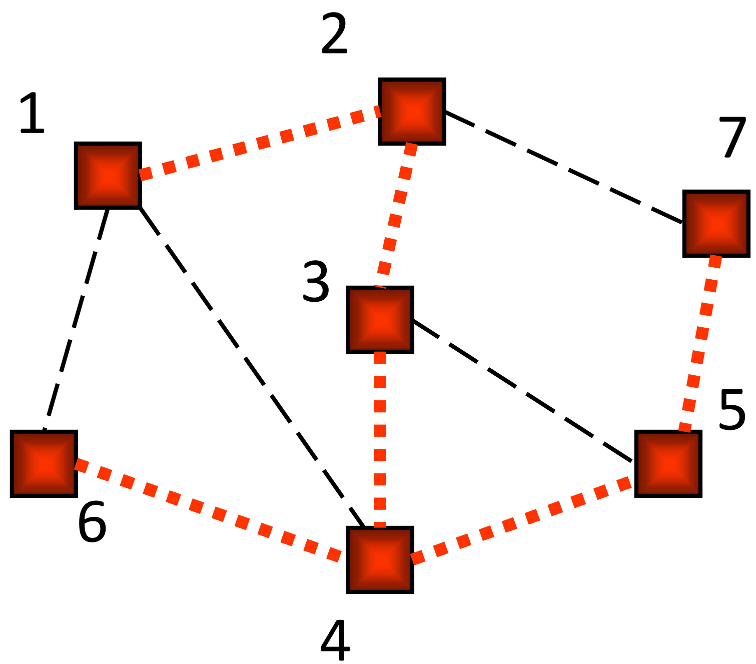
Example Step 14



DFS(1)
 DFS(2)
  DFS(3)
   DFS(4)

Back to 4.
Keep backing up.

Example Step 17



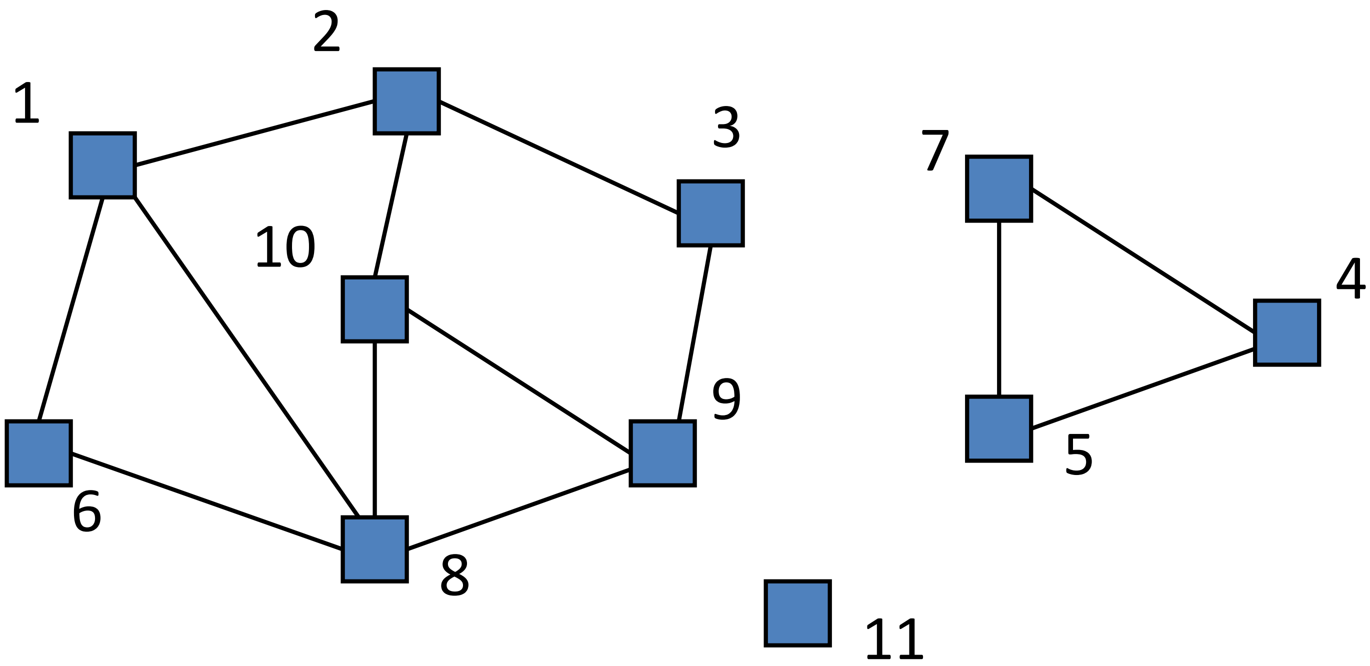
DFS(1)

All the way back to 1.

Done.

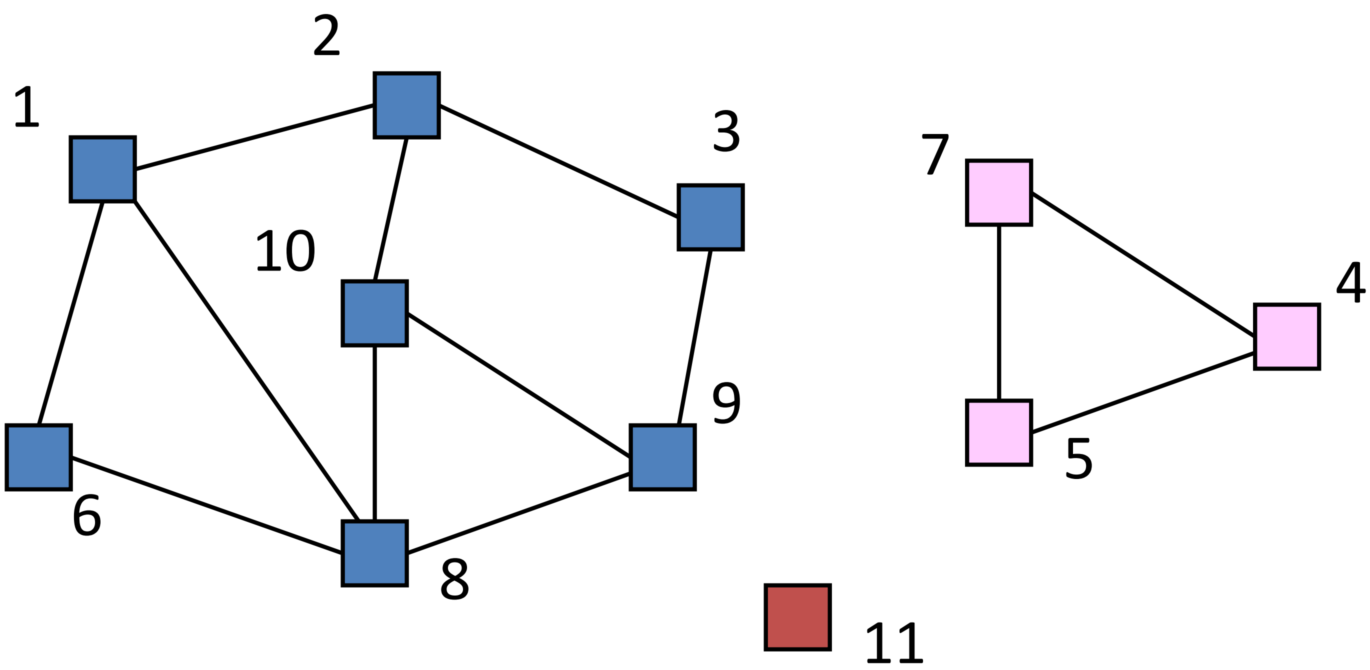
All nodes are marked so graph is connected; red links define a spanning tree

Finding Connected Components using DFS



3 connected components

Connected Components



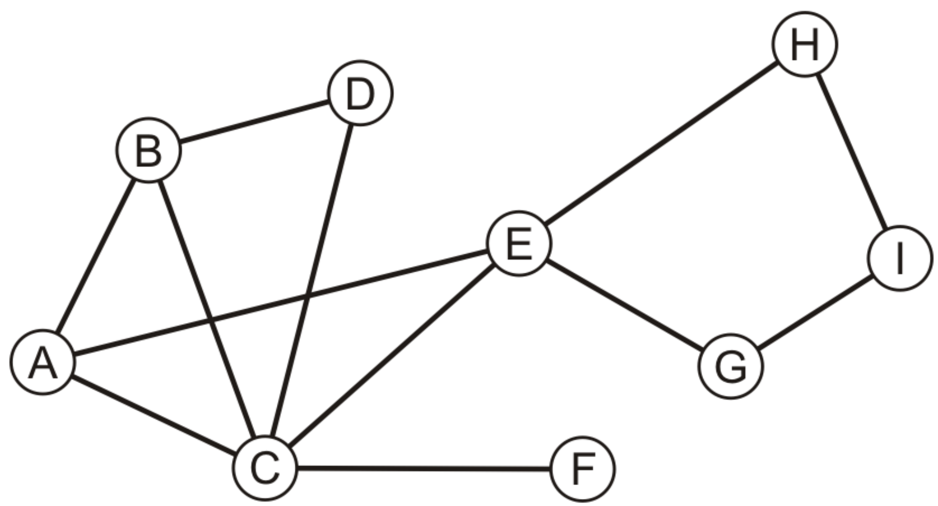
3 connected components are labeled

Performance DFS

- n vertices and m edges
- Storage complexity $O(n + m)$
- Time complexity $O(n + m)$
- Linear Time!

Another Example

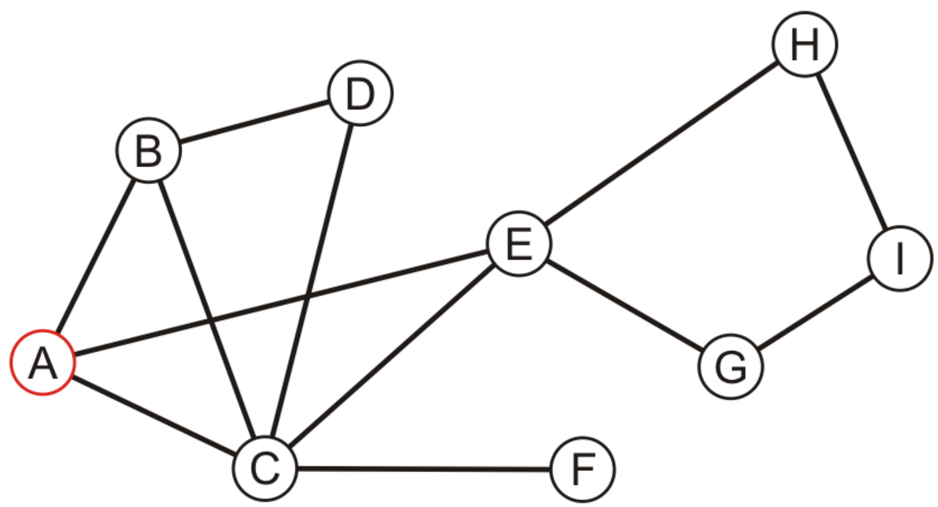
Perform a recursive depth-first traversal on this graph



Another Example

– Visit the first node

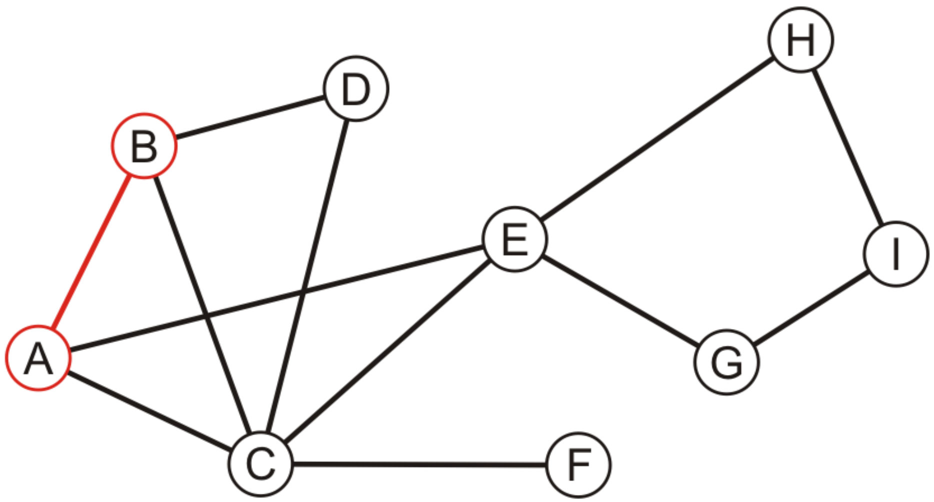
A



Another Example

– A has an unvisited neighbor

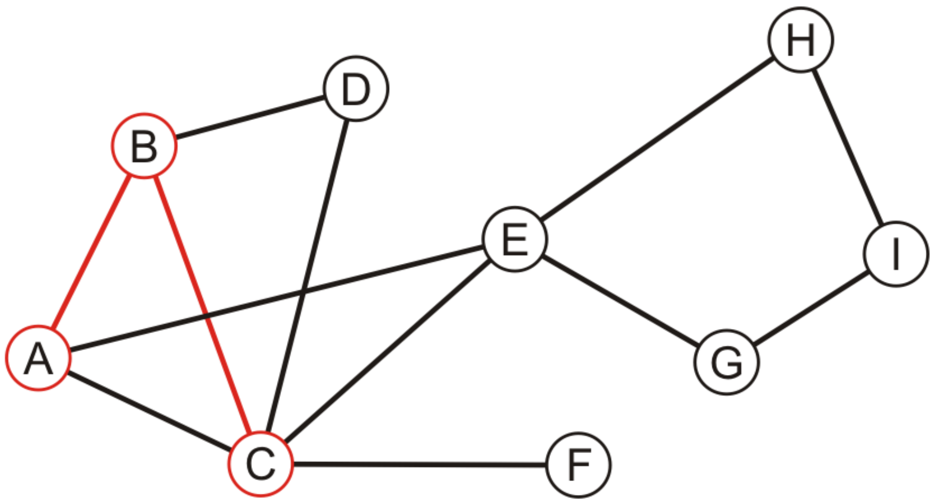
A, B



Another Example

– B has an unvisited neighbor

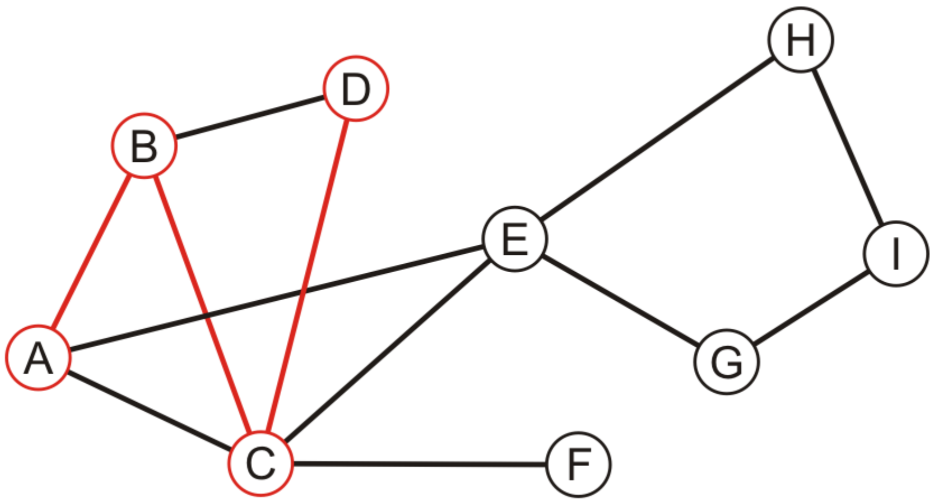
A, B, C



Another Example

– C has an unvisited neighbor

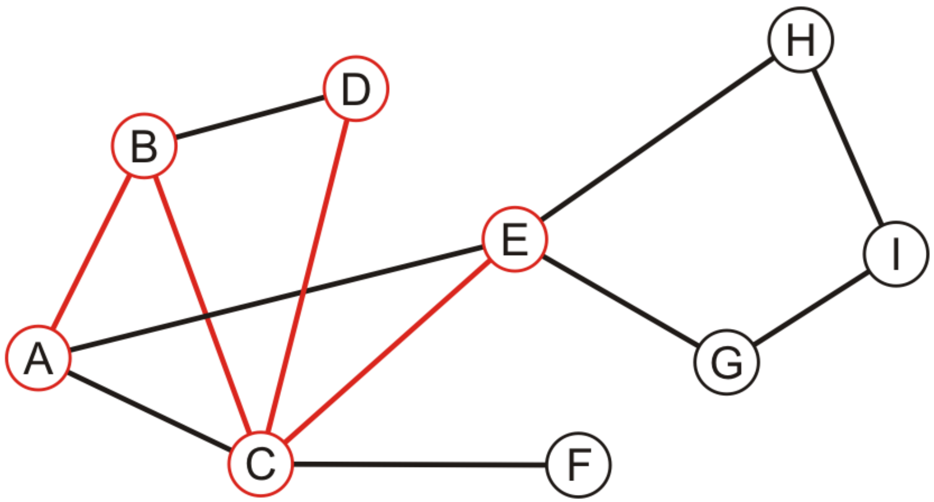
A, B, C, D



Another Example

– D has no unvisited neighbors, so we return to C

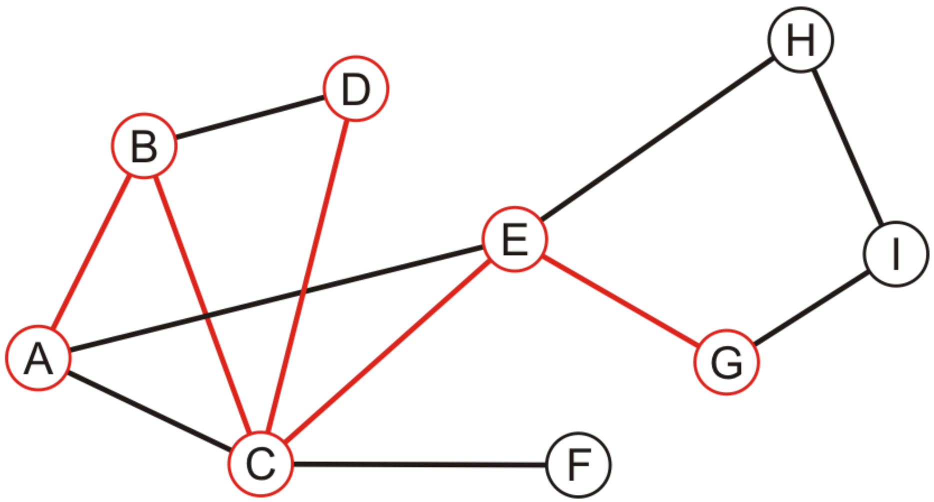
A, B, C, D, E



Another Example

– E has an unvisited neighbor

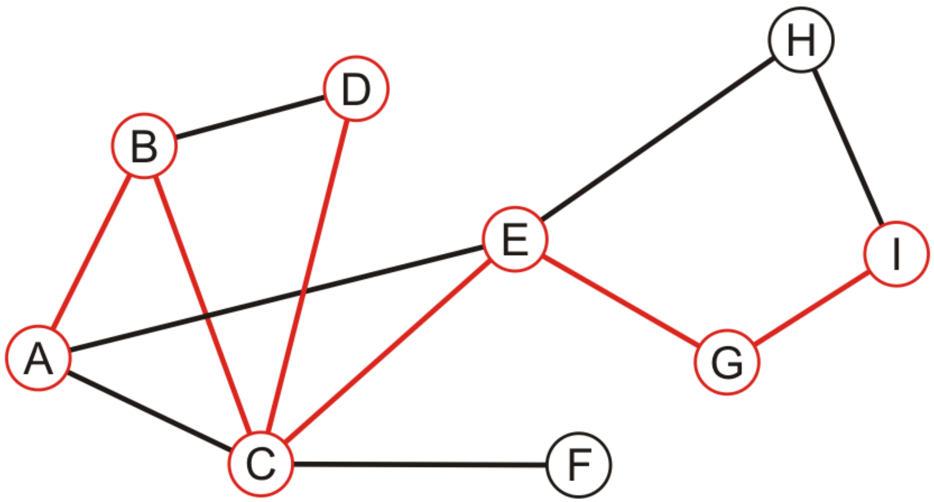
A, B, C, D, E, G



Another Example

– F has an unvisited neighbor

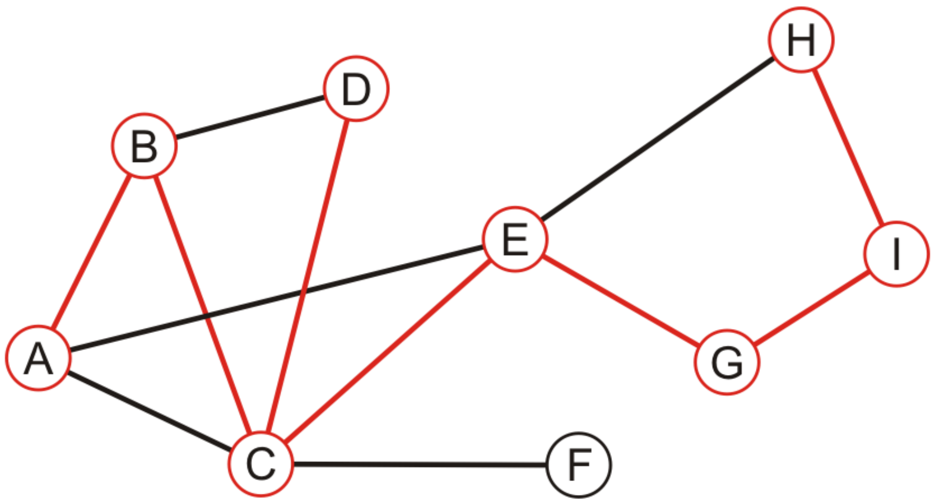
A, B, C, D, E, G, I



Another Example

– H has an unvisited neighbor

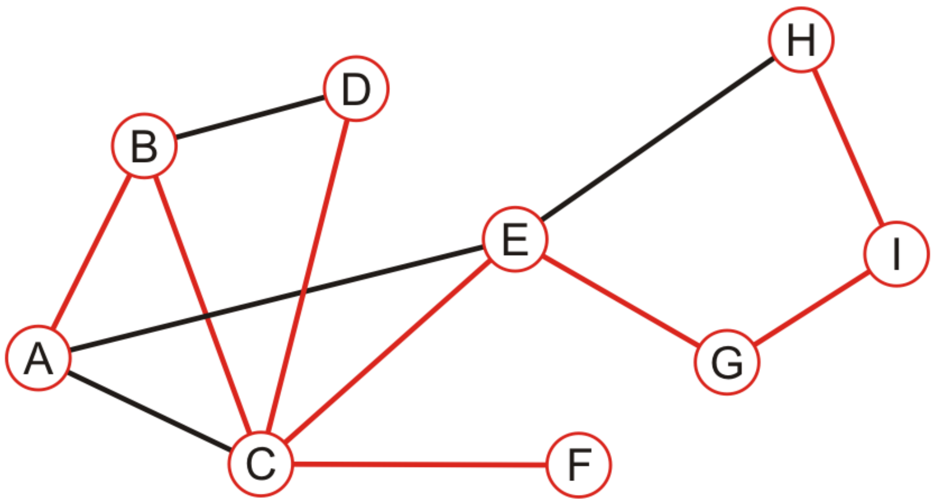
A, B, C, D, E, G, I, H



Another Example

– We recurse back to C which has an unvisited neighbour

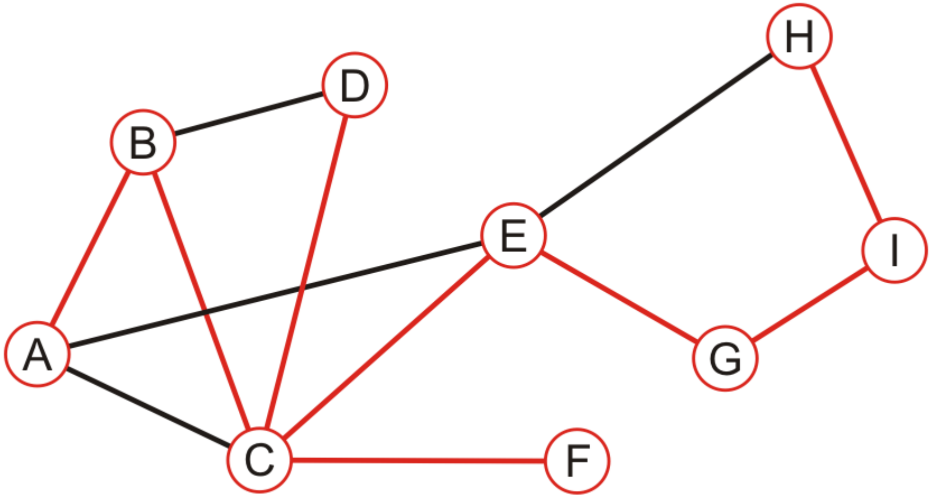
A, B, C, D, E, G, I, H, F



Another Example

– We recurse finding that no nodes have unvisited neighbours

A, B, C, D, E, G, I, H, F

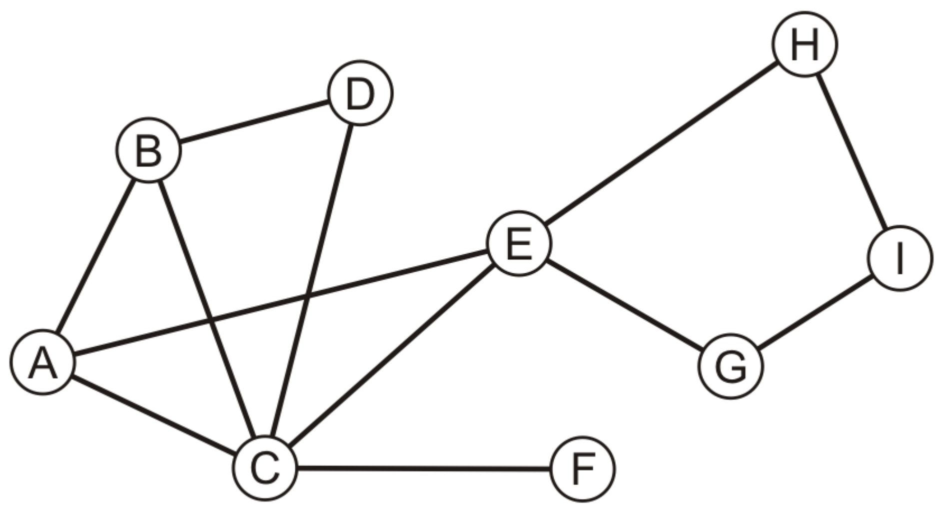


Graph Searching Methodology Breadth-First Search (BFS)

- Breadth-First Search (BFS)
  - Use a queue to explore neighbors of source vertex, then neighbors of neighbors etc.
  - All nodes at a given distance (in number of edges) are explored before we go further

Example

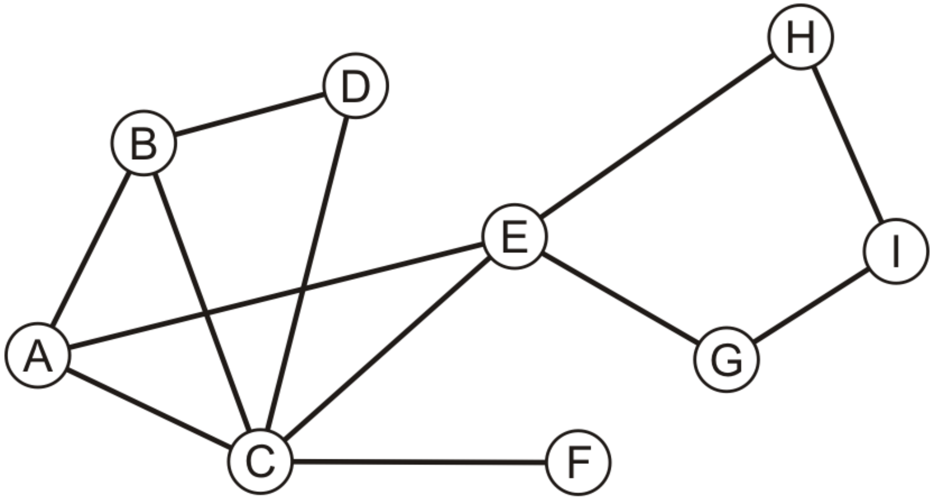
Consider the graph from previous example



Example

Performing a breadth-first traversal

– Push the first vertex onto the queue



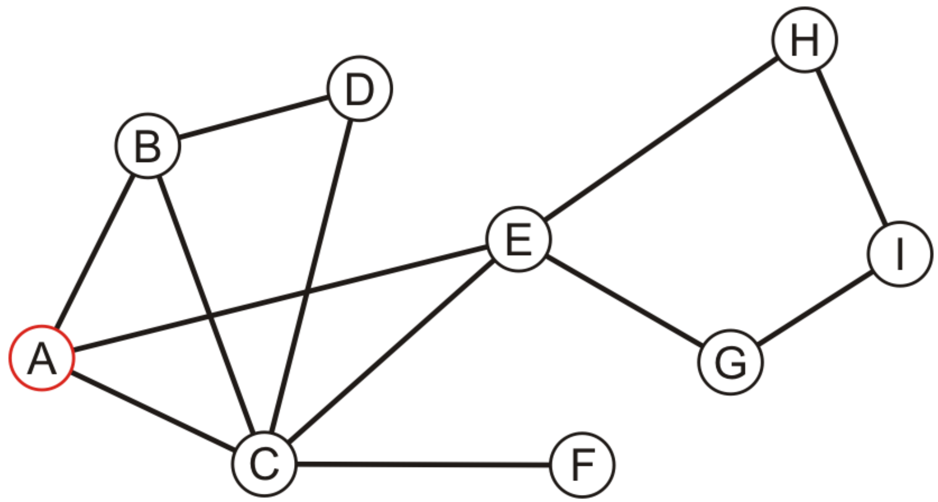
A					

Example

Performing a breadth-first traversal

– Pop A and push B, C and E

A



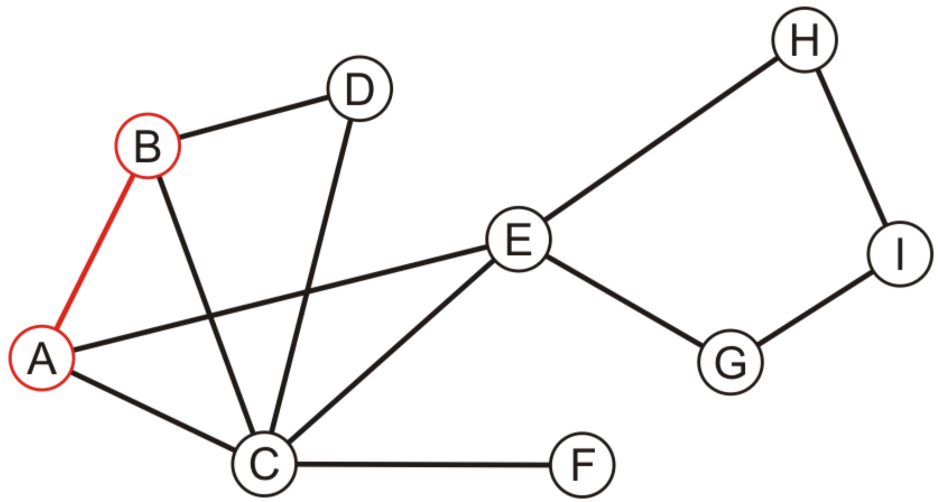
B	C	E			

Example

Performing a breadth-first traversal:

– Pop B and push D

A, B



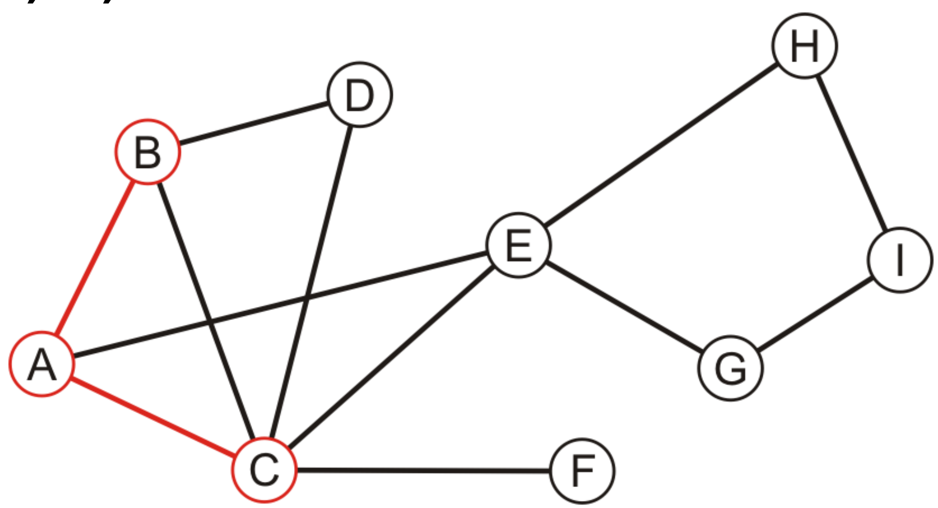
C	E	D			

Example

Performing a breadth-first traversal:

– Pop C and push F

A, B, C



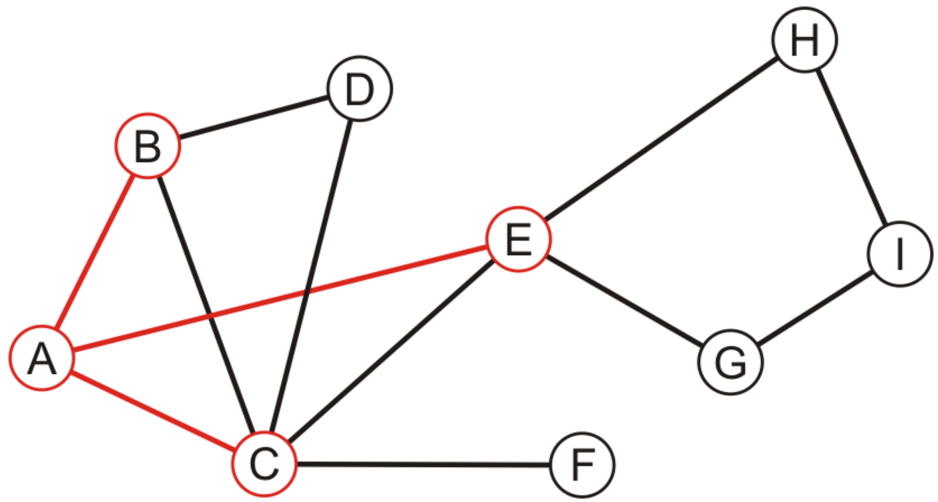
E	D	F			

Example

Performing a breadth-first traversal:

– Pop E and push G and H

A, B, C, E



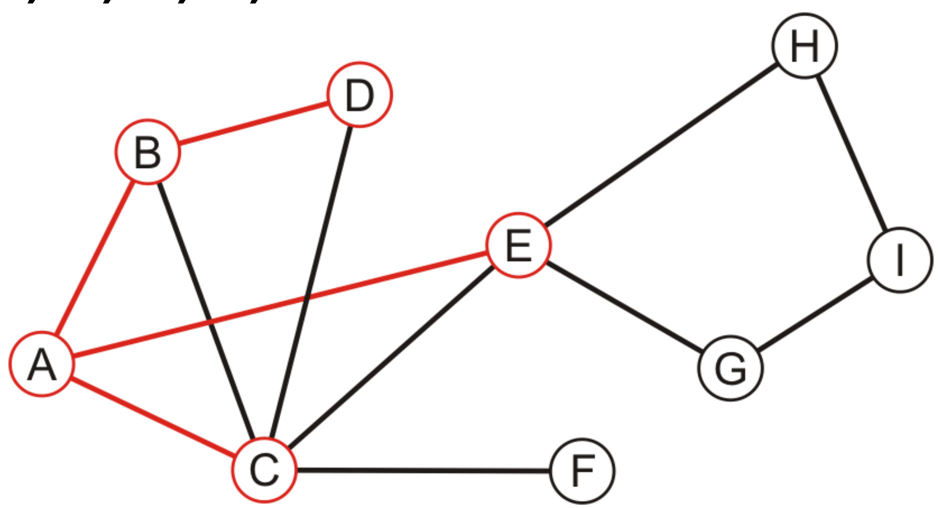
D	F	G	H		

Example

Performing a breadth-first traversal:

– Pop D

A, B, C, E, D



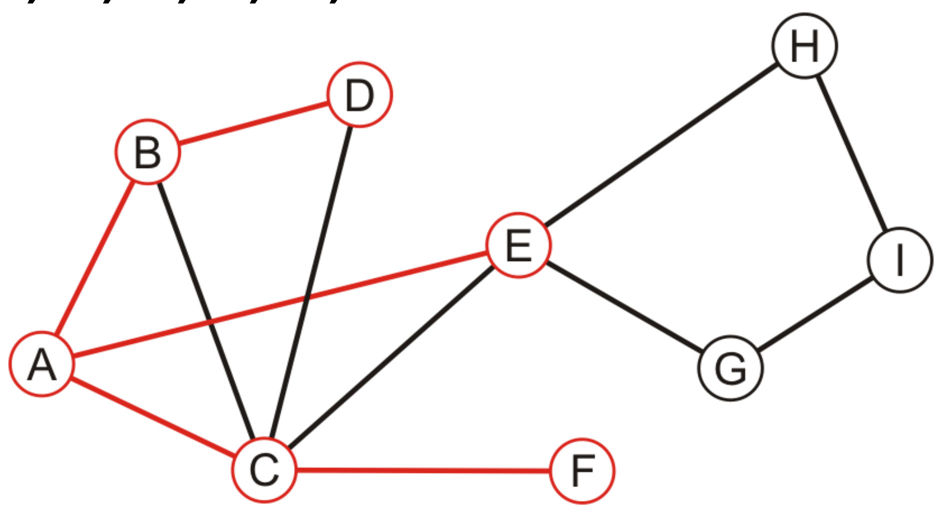
F	G	H			

Example

Performing a breadth-first traversal:

– Pop F

A, B, C, E, D, F



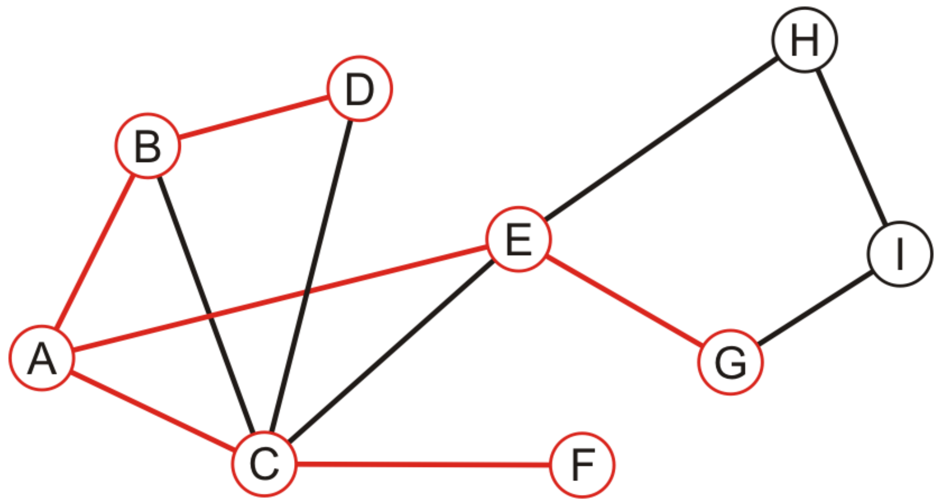
G	H				

Example

Performing a breadth-first traversal:

– Pop G and push I

A, B, C, E, D, F, G



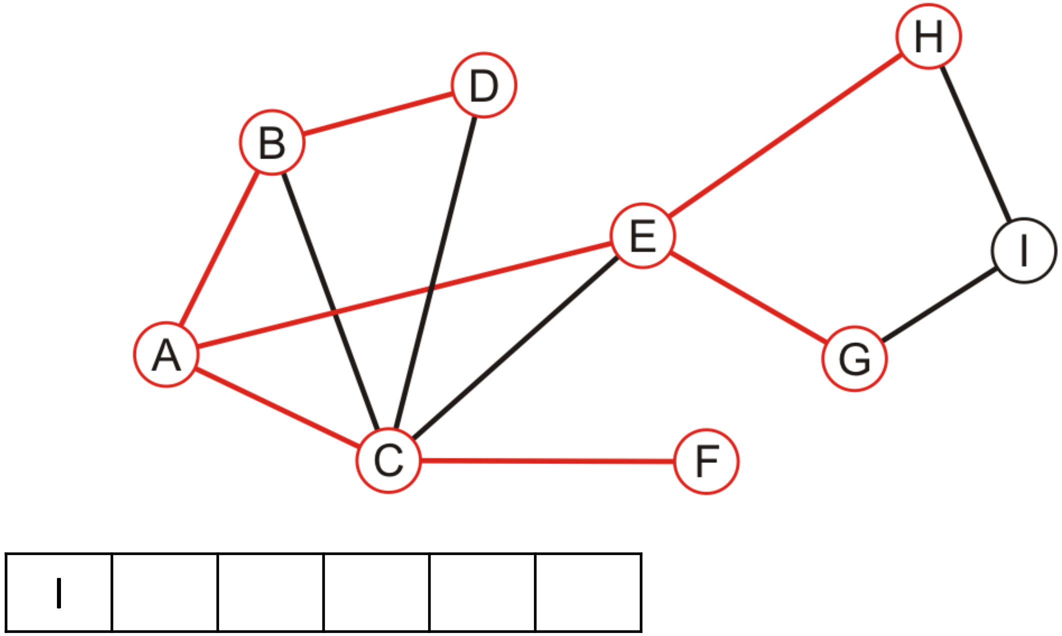
H	I				

Example

Performing a breadth-first traversal:

– Pop H

A, B, C, E, D, F, G, H

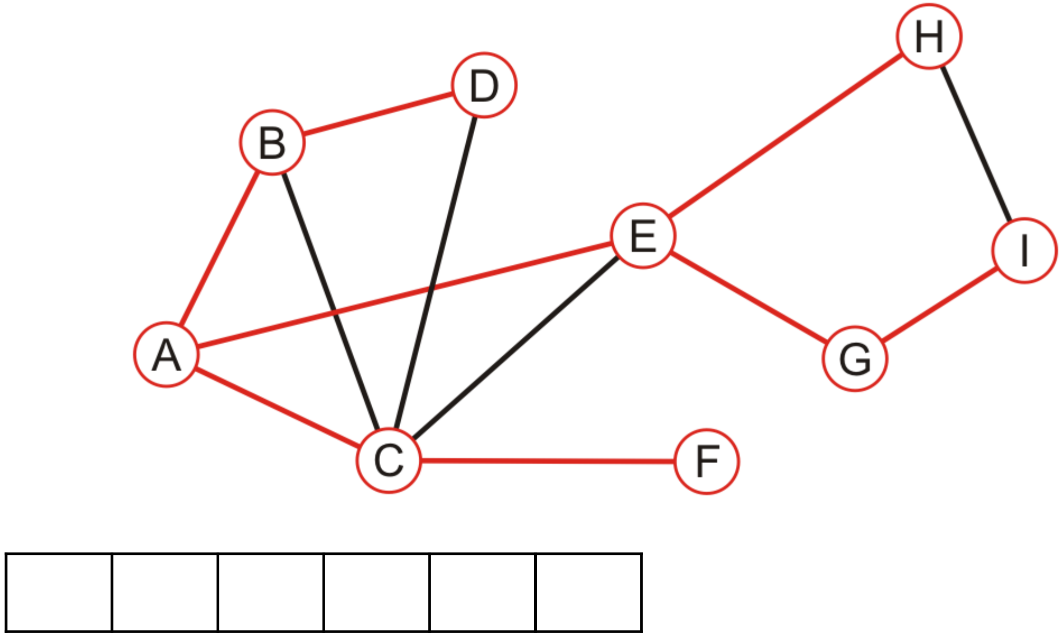


Example

Performing a breadth-first traversal:

– Pop I

A, B, C, E, D, F, G, H, I

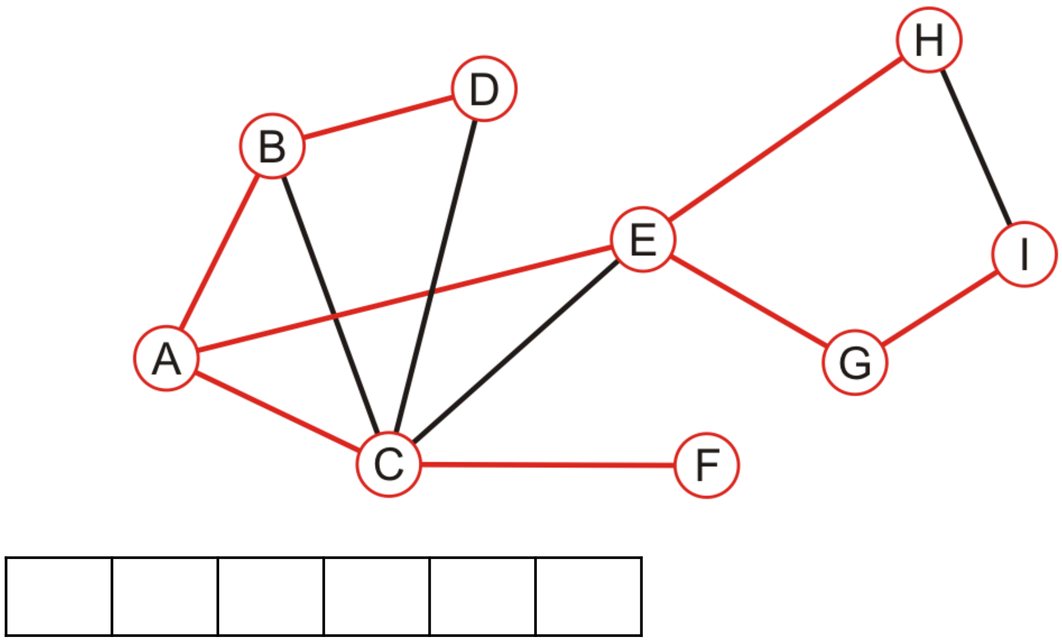


Example

Performing a breadth-first traversal:

– The queue is empty: we are finished

A, B, C, E, D, F, G, H, I



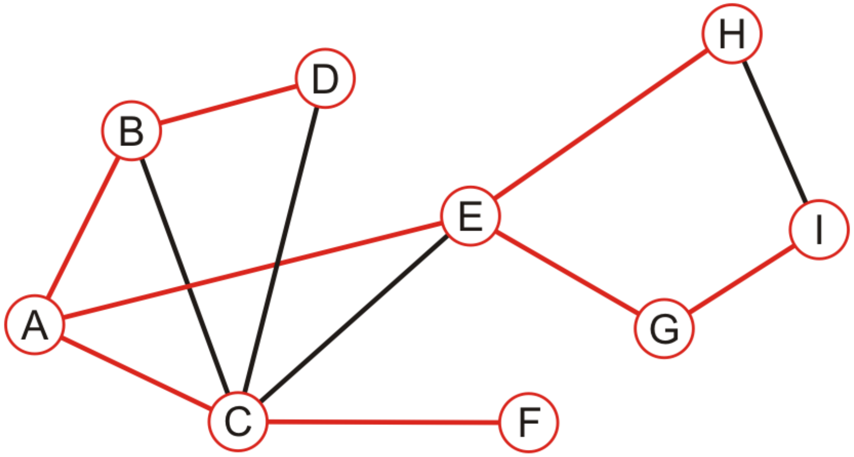
Breadth-First Search

```
BFS
Initialize Q to be empty;
Enqueue(Q,1) and mark 1;
while Q is not empty do
   i := Dequeue(Q);
   for each j adjacent to i do
      if j is not marked then
         Enqueue(Q,j) and mark j;
end{BFS}
```

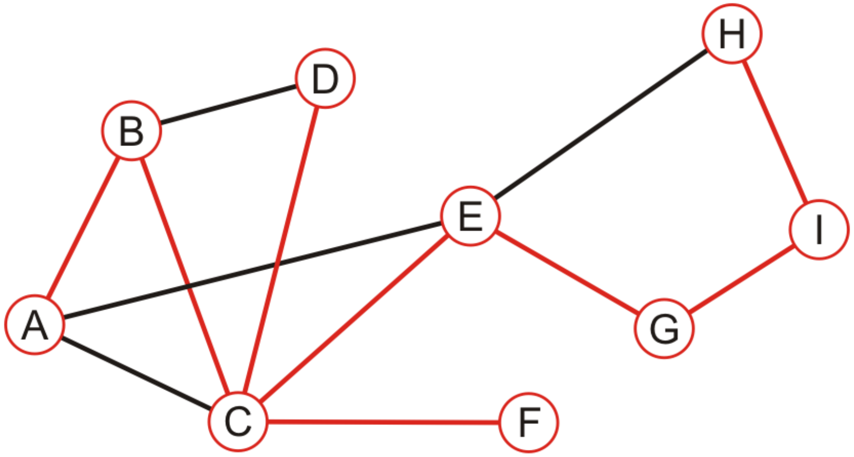
Comparison

The order in which vertices can differ greatly

A, B, C, E, D, F, G, H, I



A, B, C, D, E, G, I, H, F



Depth-First vs Breadth-First

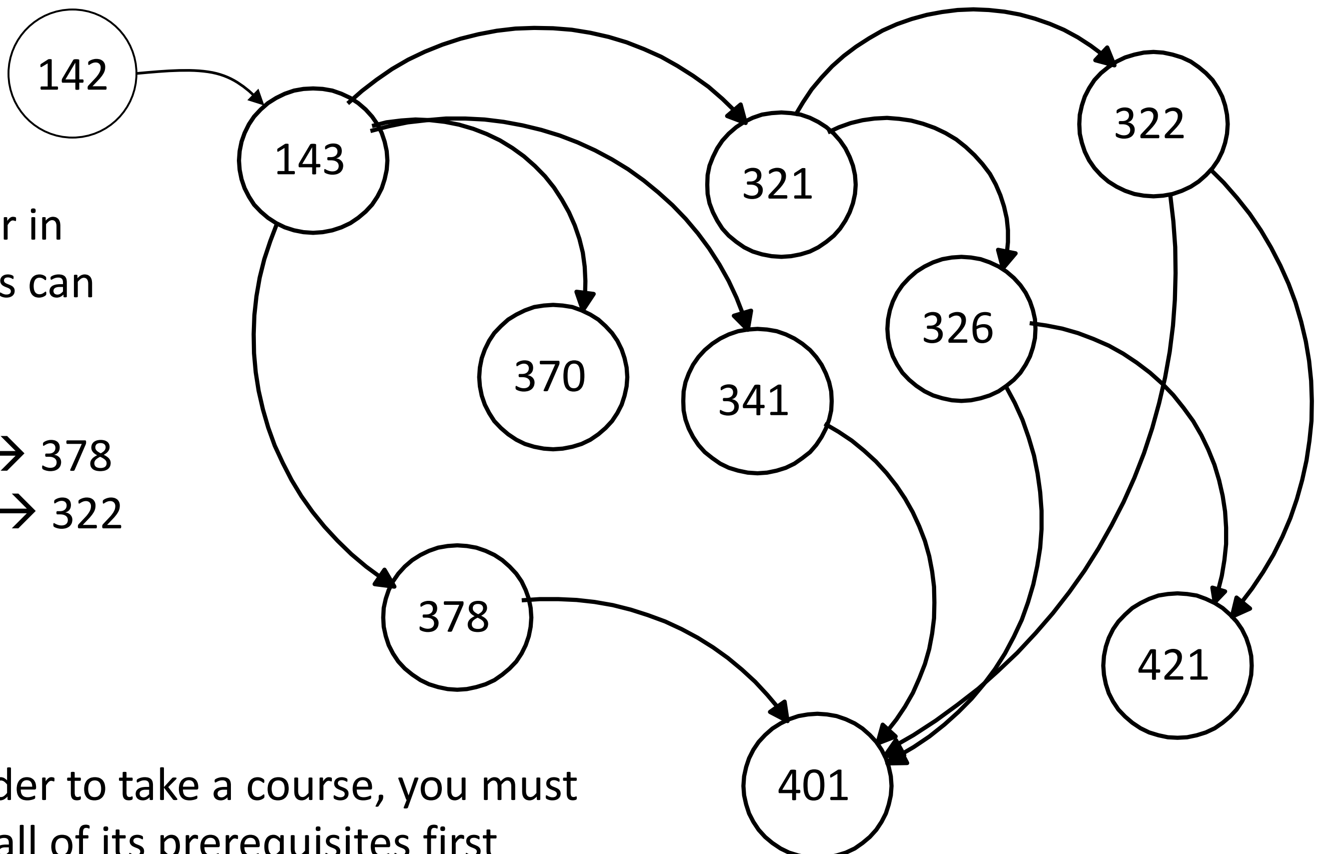
- Depth-First
  - Stack or recursion
  - Many applications
- Breadth-First
  - Queue (recursion no help)
  - Can be used to find shortest paths from the start vertex

Topological Sort

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

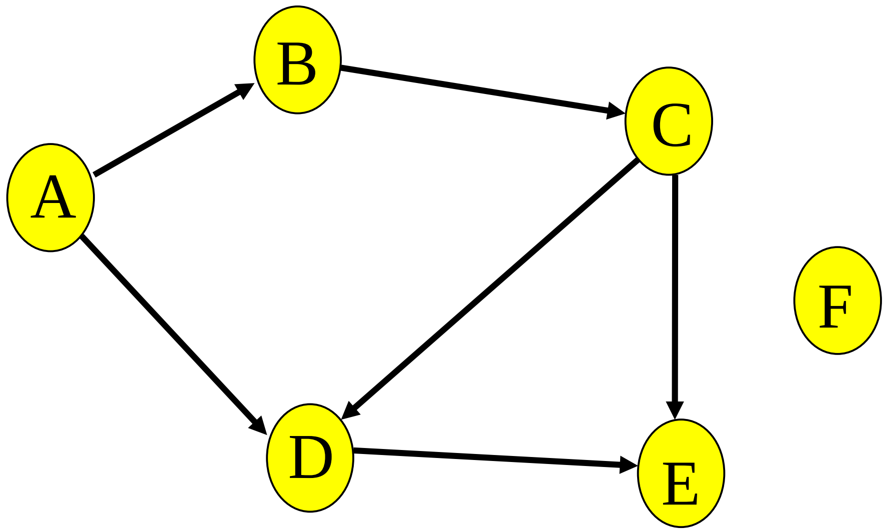


In order to take a course, you must take all of its prerequisites first

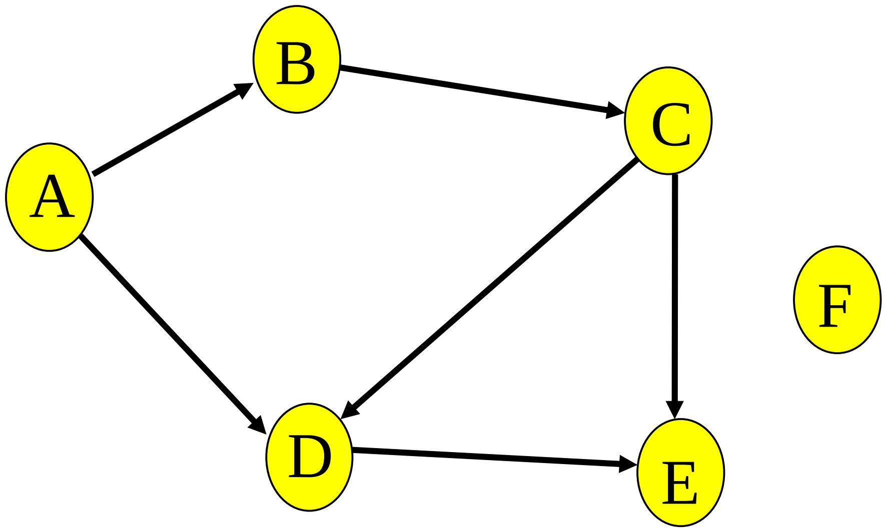
Topological Sort

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

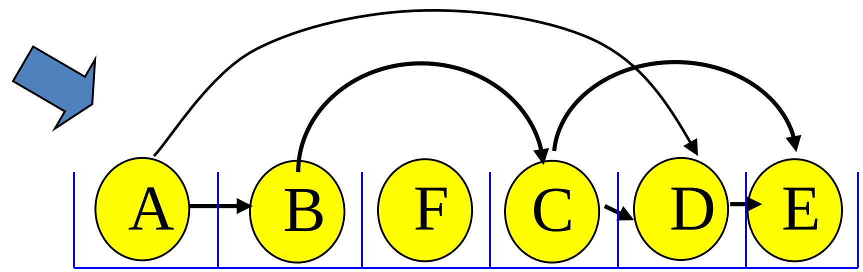
for any edge (v, w) in E, v precedes w in the ordering



Topo sort - good example

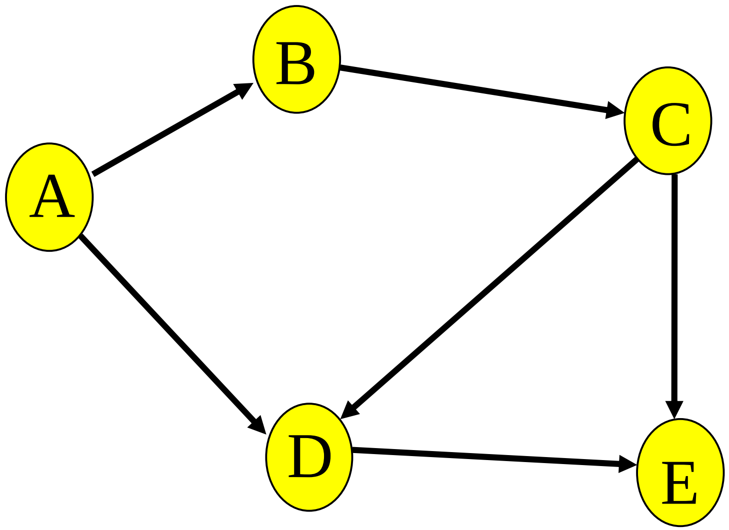


Any linear ordering in which all the arrows go to the right is a valid solution

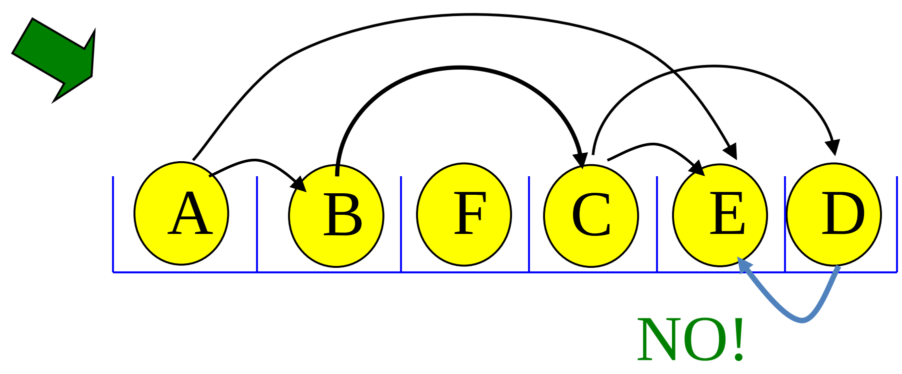


Note that F can go anywhere in this list because it is not connected.
Also the solution is not unique.

Topo sort - bad example



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

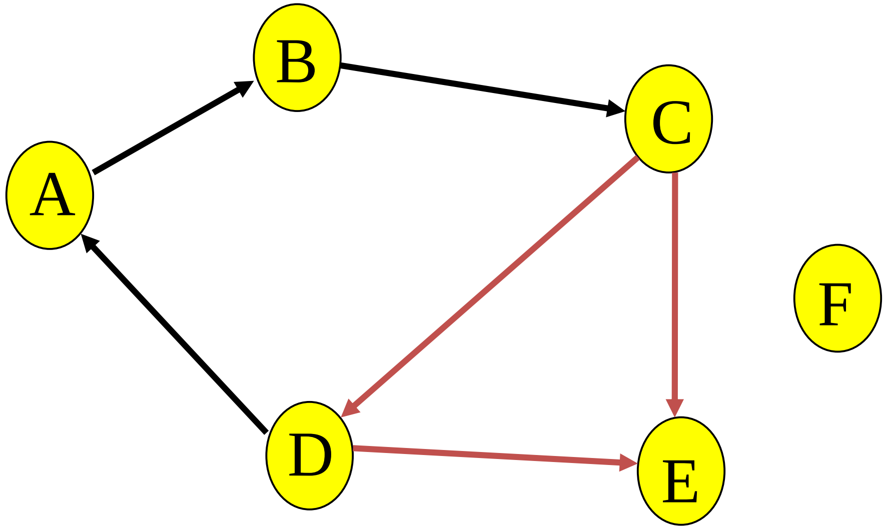


Paths and Cycles

- 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 $1 \leq i < k$
  - path length = number of edges in the path
  - path cost = sum of costs of each edge
- A path is a cycle if :
  - $k > 1$; $v_1 = v_k$
- G is acyclic if it has no cycles.

Only acyclic graphs can be topo. sorted

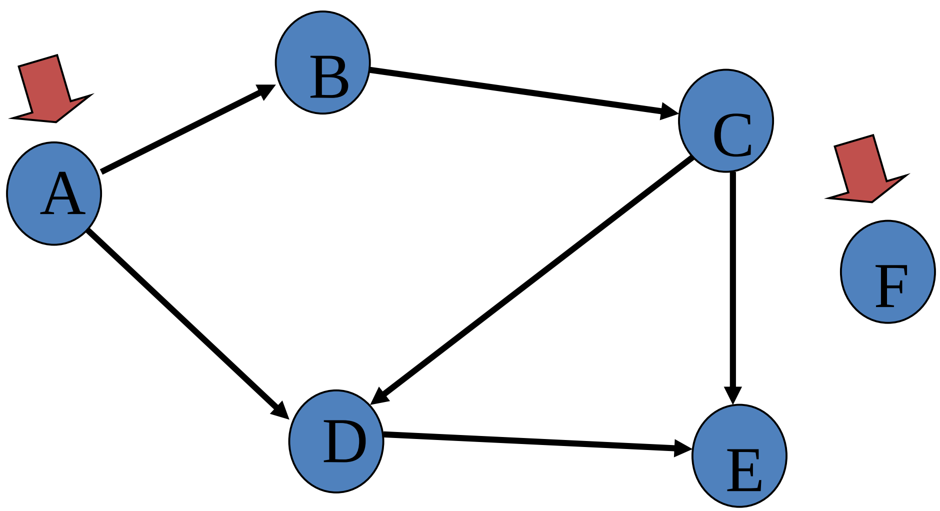
- A directed graph with a cycle cannot be topologically sorted.



Topo sort algorithm - 1

Step 1: Identify vertices that have no incoming edges

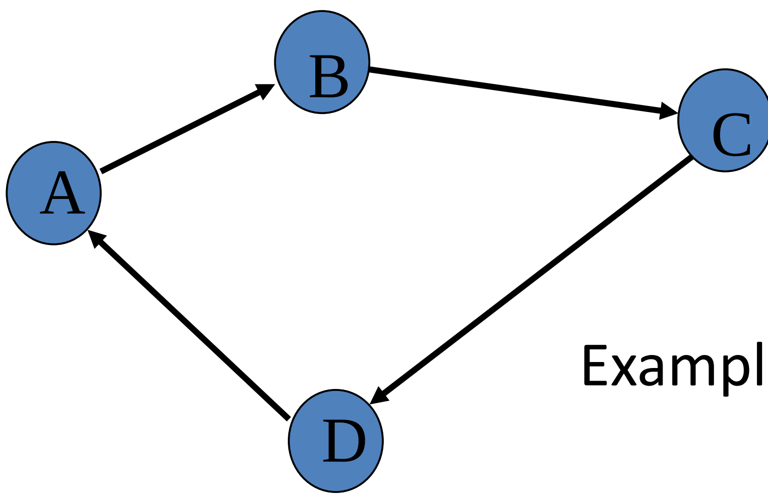
- The "in-degree" of these vertices is zero



Topo sort algorithm - 1a

Step 1: Identify vertices that have no incoming edges

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

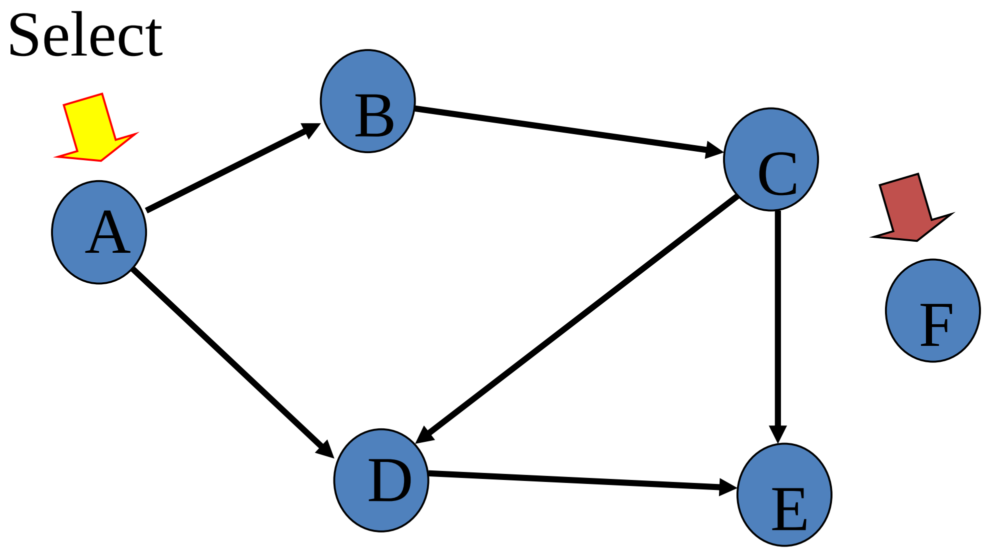


Example of a cyclic graph

Topo sort algorithm - 1b

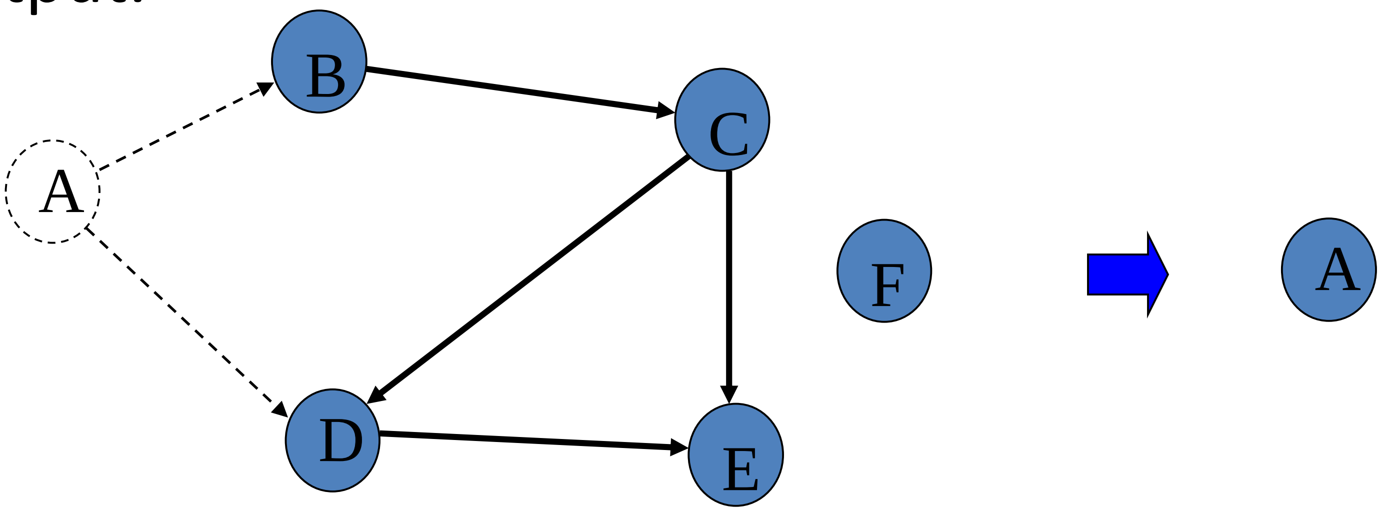
Step 1: Identify vertices that have no incoming edges

- Select one such vertex



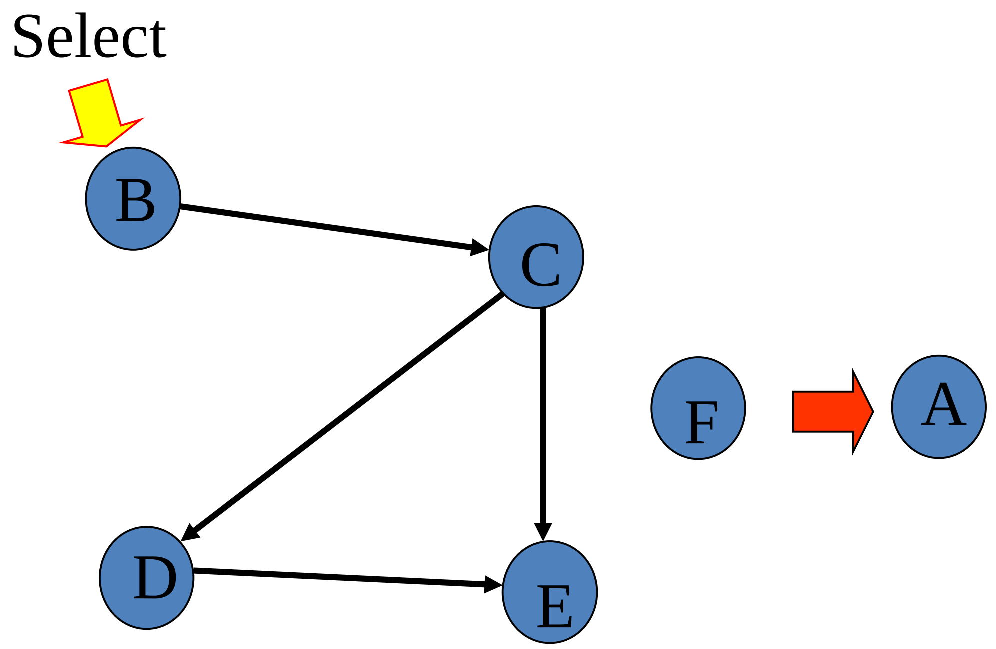
Topo sort algorithm - 2

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



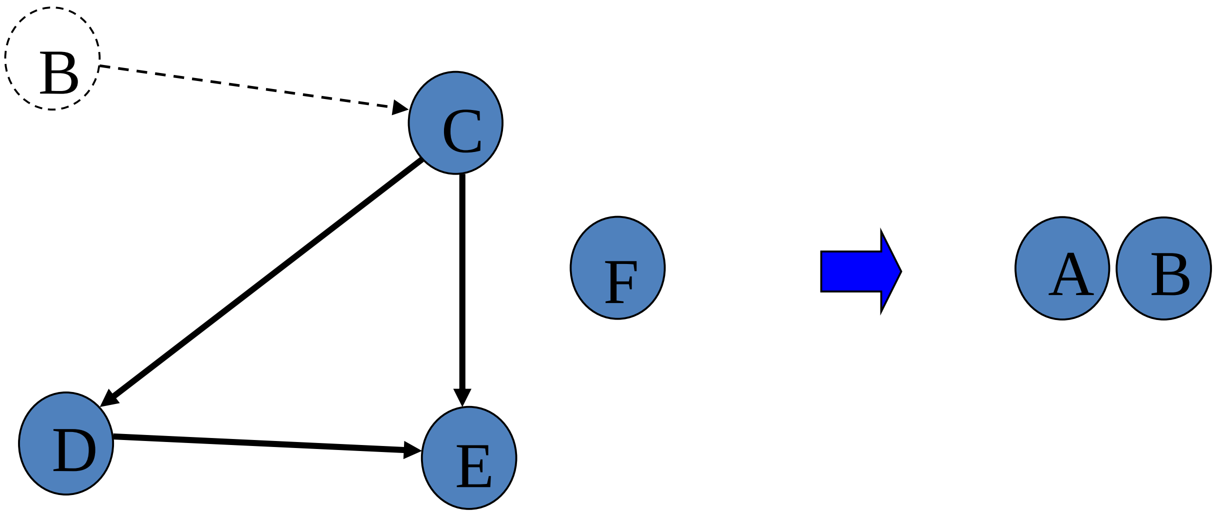
Continue until done

Repeat Step 1 and Step 2 until graph is empty



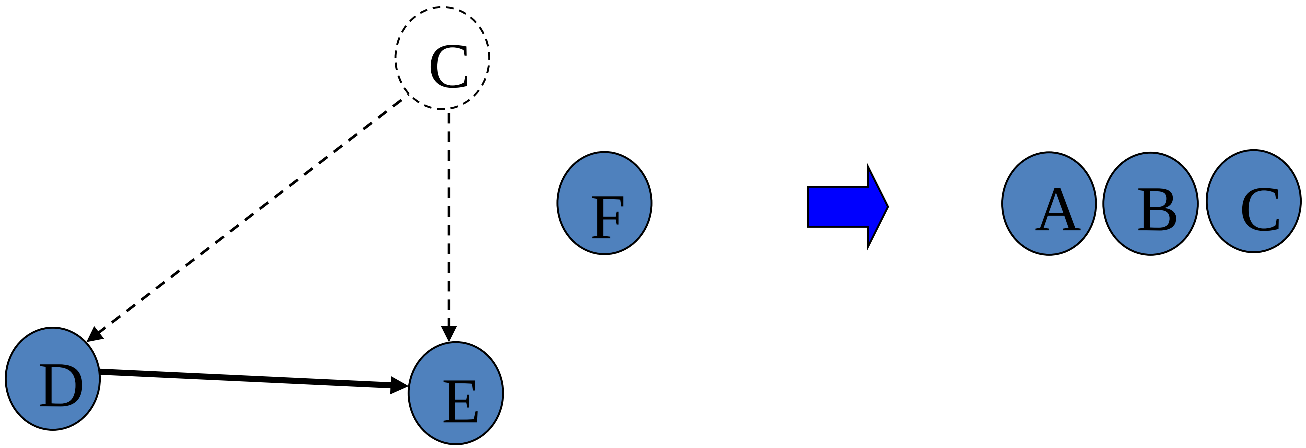
B

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



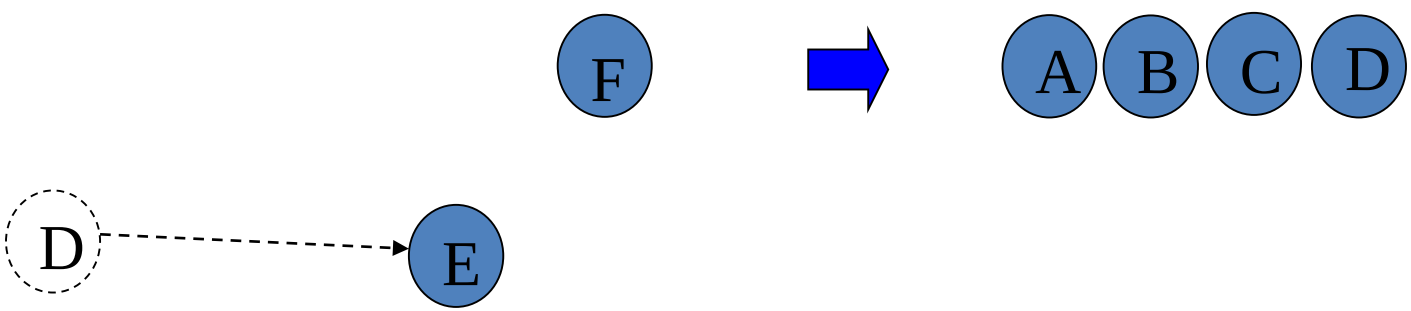
C

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



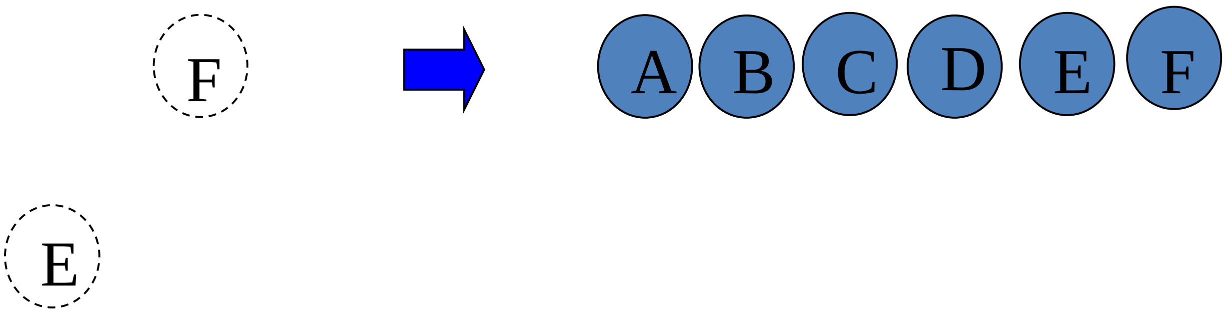
D

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

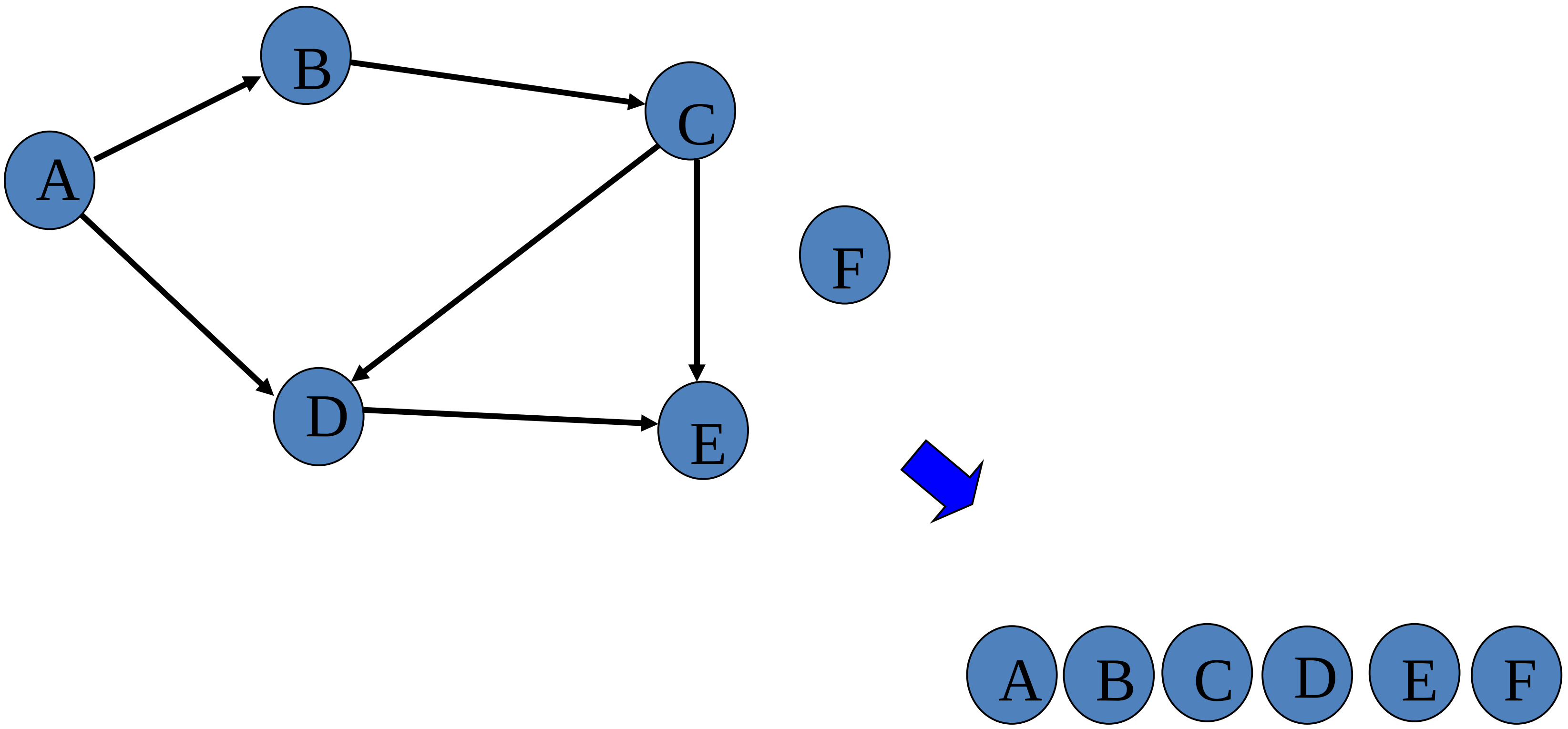


E, F

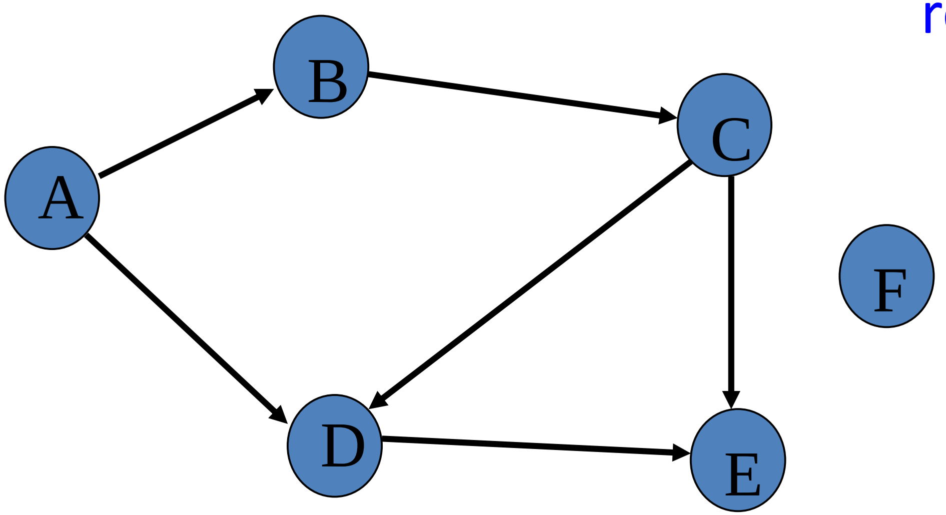
Select E. Copy to sorted list. Delete E and its edges.
Select F. Copy to sorted list. Delete F and its edges.



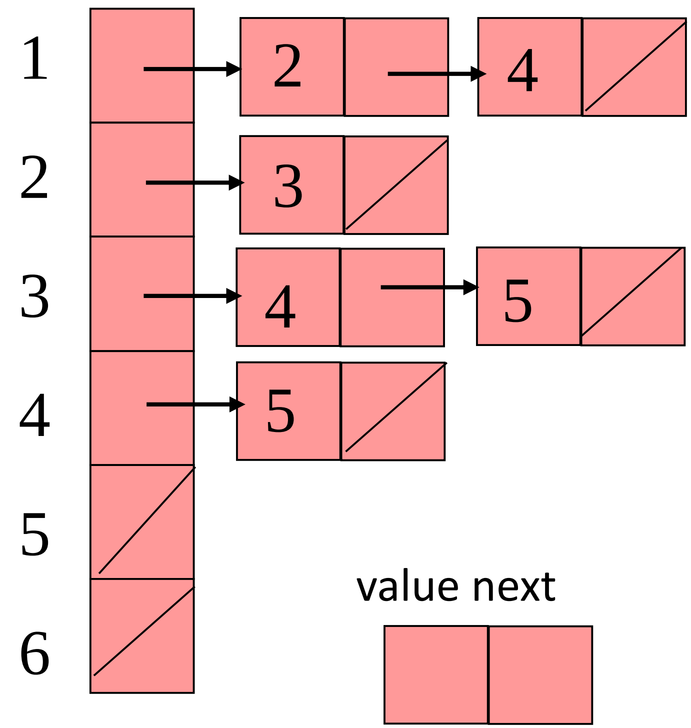
Done



Implementation



Assume adjacency list representation

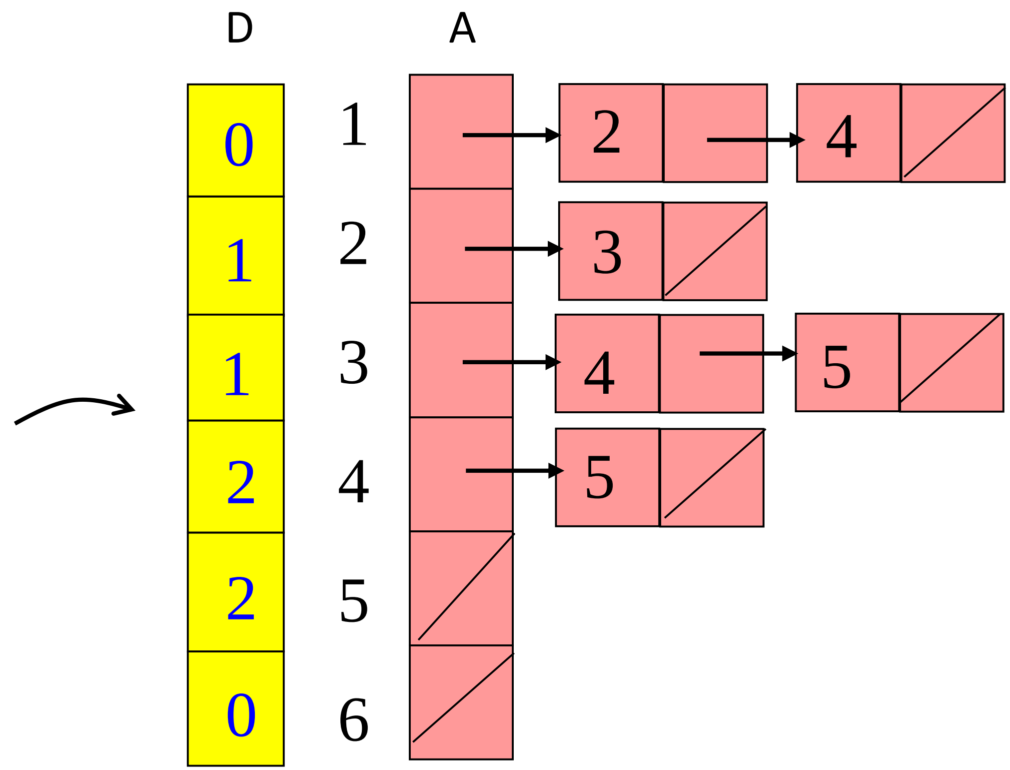


Translation array

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



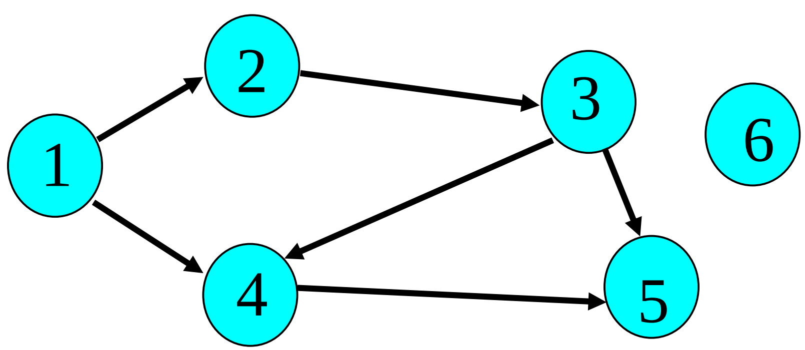
Calculate In-degrees

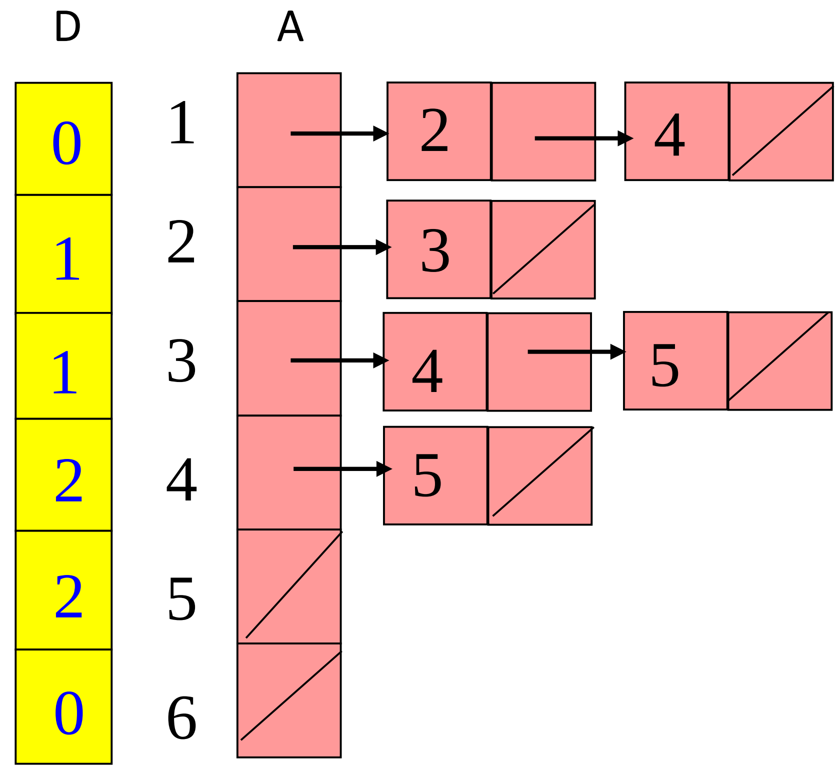
```
for i = 1 to n do D[i] := 0; endfor
for i = 1 to n do
  x := A[i];
  while x ≠ null do
    D[x.value] := D[x.value] + 1;
    x := x.next;
  endwhile
endfor
```

Maintaining Degree 0 Vertices

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

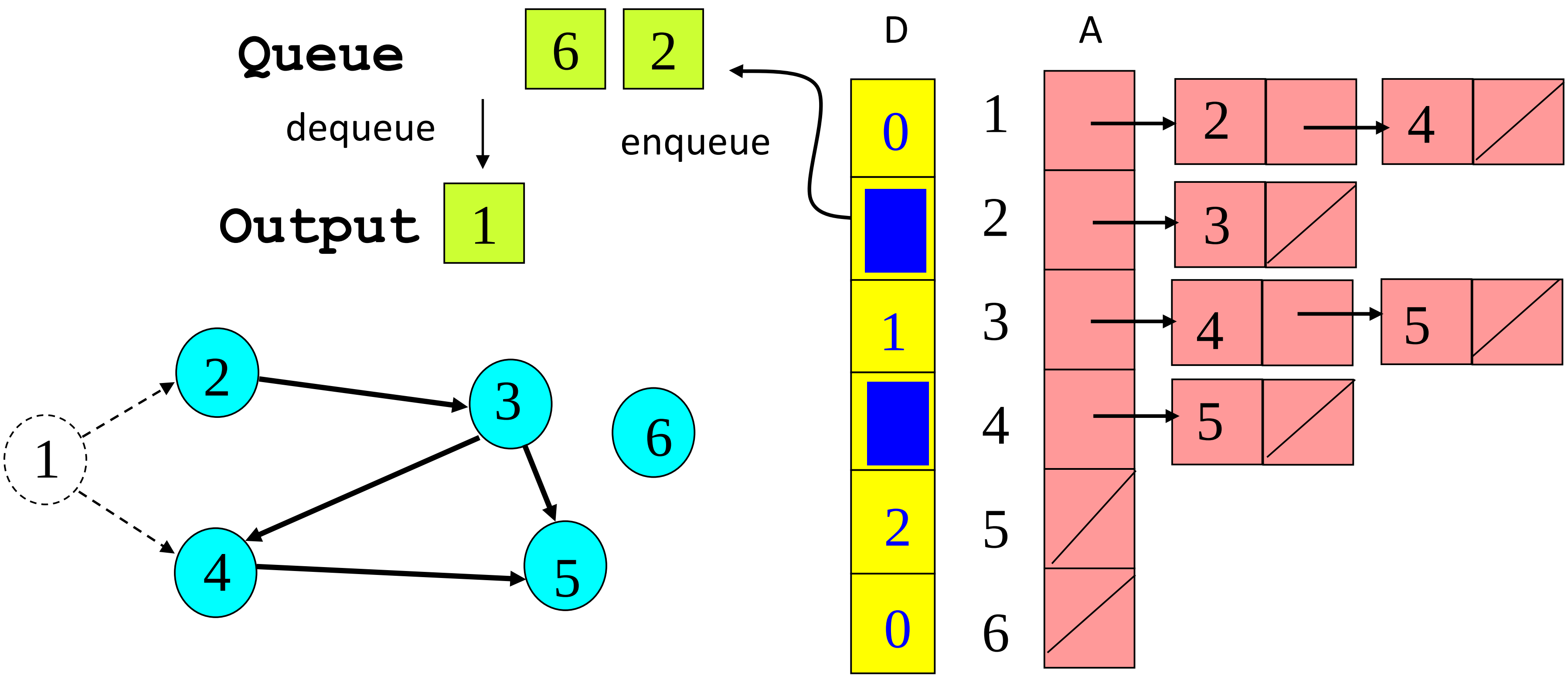
Queue 1 6





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.