Topological Sort

Outlines

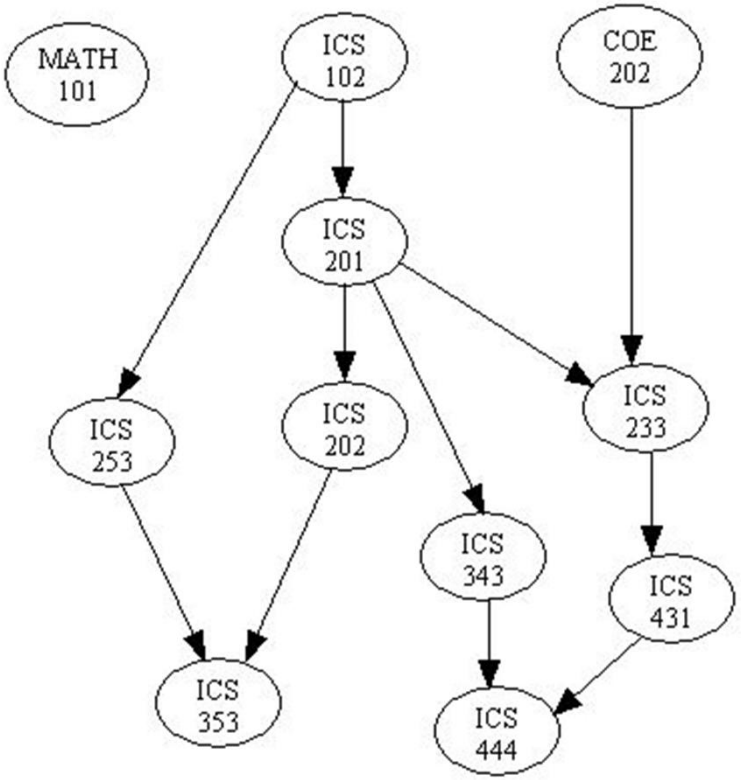
- ✓ Introduction
- ✓ Definition of Topological Sort
- ✓ Topological Sort is Not Unique
- ✓ Overview of DFS
- ✓ Topological Sort Algorithms
  - Å DFS Based Algorithm
  - Å Source Removal Based Algorithm
- ✓ Example
- ✓ Proof of Correctness

Introduction

There are many problems involving a set of tasks in which some of the tasks must be done before others.

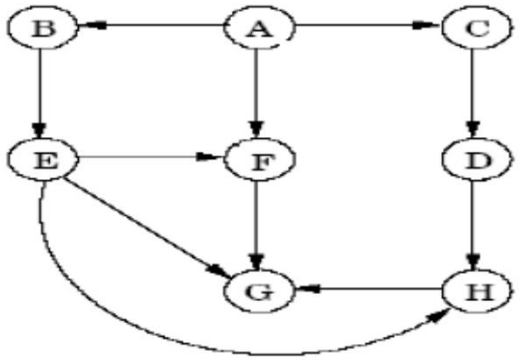
For example, consider the problem of taking a course only after taking its prerequisites.

Is there any systematic way of linearly arranging the courses in the order that they should be taken?



What is a DAG?

A directed acyclic graph **(DAG)** is a directed graph without cycles.



DAGs arise in modeling many problems that involve prerequisite constraints (construction projects, course prerequisites, document version control, compilers, etc.)

Some properties of DAGs:

- Every DAG must have at least one vertex with in-degree zero and at least one vertex with out-degree zero.

  A vertex with in-degree zero is called a source vertex, a vertex with out-degree zero is called a sink vertex.

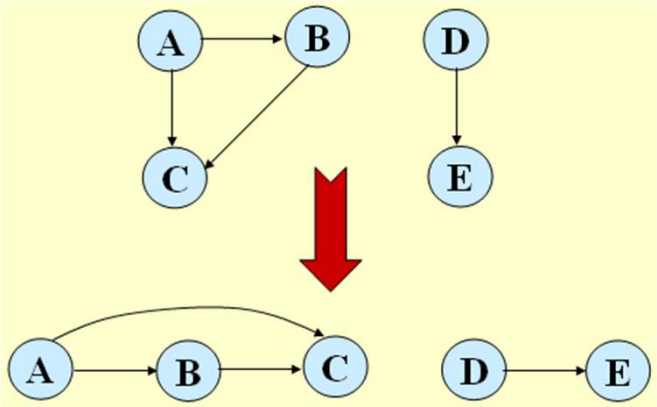
- G is a DAG iff each vertex in G is in its own strongly connected component.

- Every edge (v, w) in a DAG G has finishingTime[w] < finishingTime[v] in a DFS traversal of G.

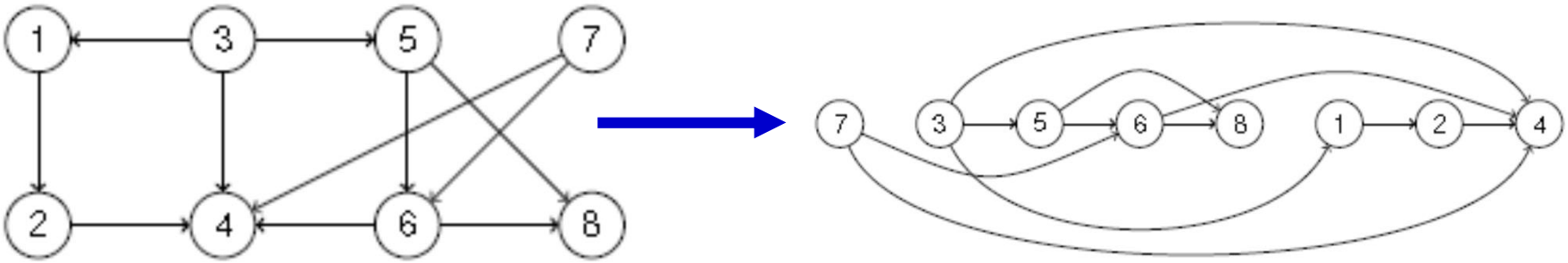
Definition of Topological Sort

Given a directed graph G = (V, E), a topological sort of G is an ordering of V such that for any edge (u, v), u comes before v in the ordering.

Example 1:

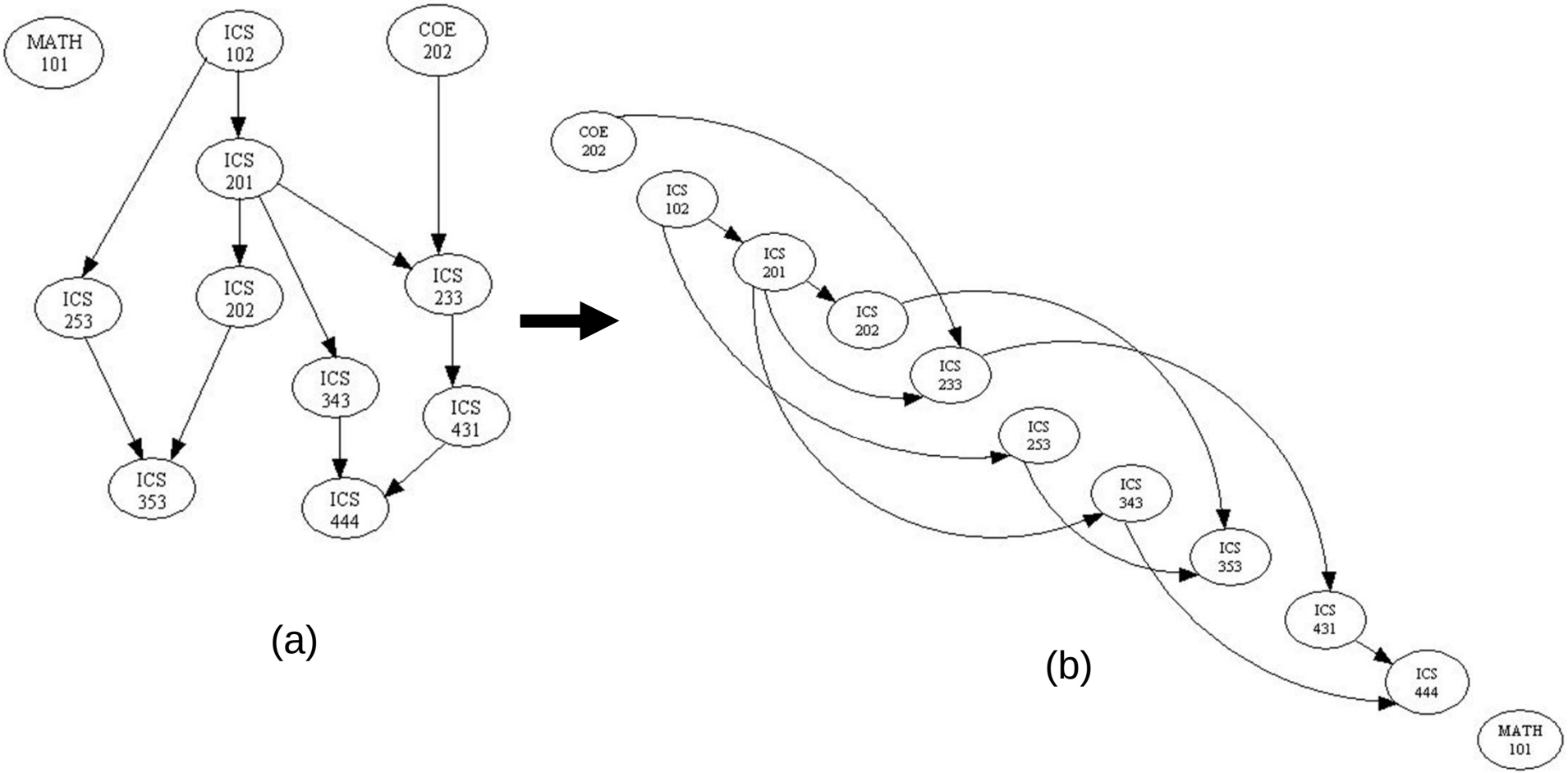


Example 2:



Definition of Topological Sort

Example 3: The graph in (a) can be topologically sorted as in (b).



Topological sort more formally

- Suppose that in a **directed** graph **G = (V, E)** vertices **V** represent tasks, and each edge (**u**, **v**)∈**E** means that task **u** must be done before task **v.**

- What is an ordering of vertices 1, ..., |**V**| such that for every edge (**u**, **v**), **u** appears before **v** in the ordering?

- Such an ordering is called a **topological sort of G.**

Note: There can be multiple topological sorts of G.

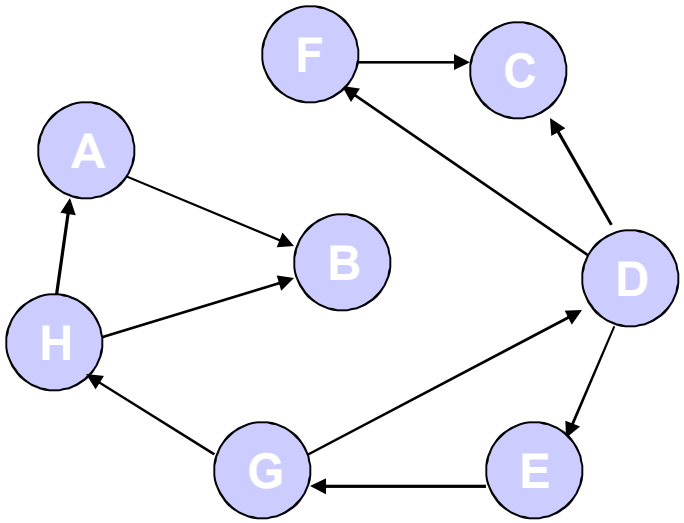
Topological sort more formally

- Is it possible to execute all the tasks in **G** in an order that respects all the precedence requirements given by the graph edges?

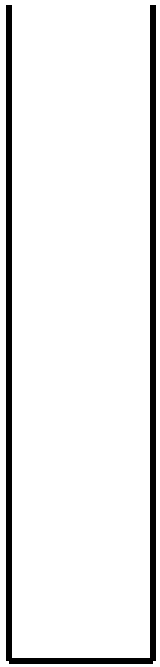
- The answer is "**yes**" *if and only if* the directed graph **G** has **no cycle**! (otherwise we have a **deadlock**)

- Such a **G** is called a Directed Acyclic Graph, or just a **DAG.**

DFS

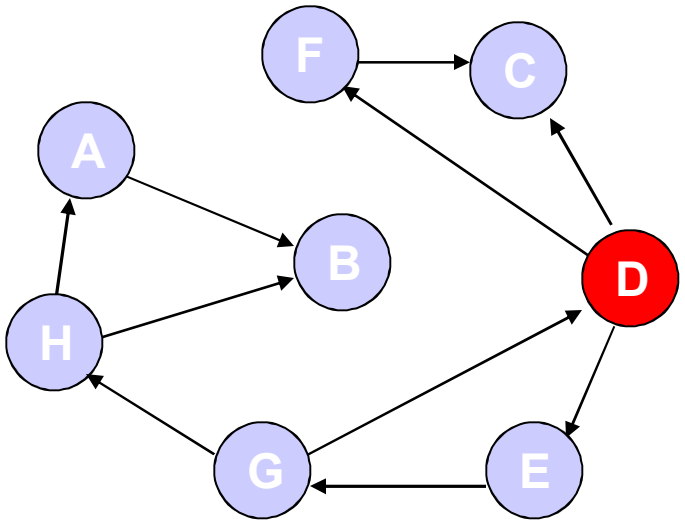


A	
B	
C	
D	
E	
F	
G	
H	

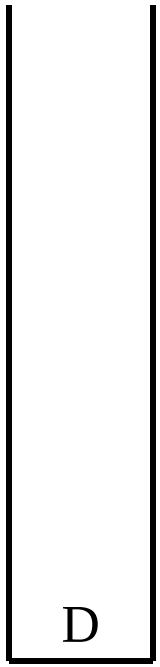


Task: Conduct a depth-first search of the graph starting with node D

DFS



A	
B	
C	
D	√
E	
F	
G	
H	

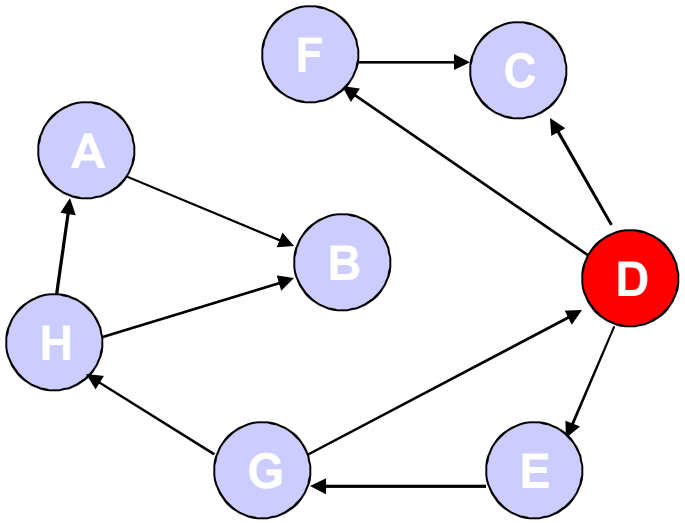


The order nodes are visited:

D

Visit D

DFS



A	
B	
C	
D	√
E	
F	
G	
H	

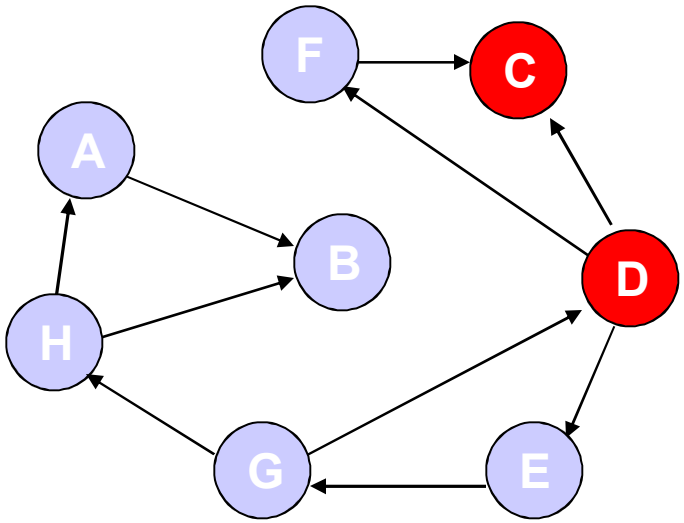
D

The order nodes are visited:

D

Consider nodes adjacent to D, decide to visit C first (Rule: visit adjacent nodes in alphabetical order)

DFS



A	
B	
C	√
D	√
E	
F	
G	
H	

C

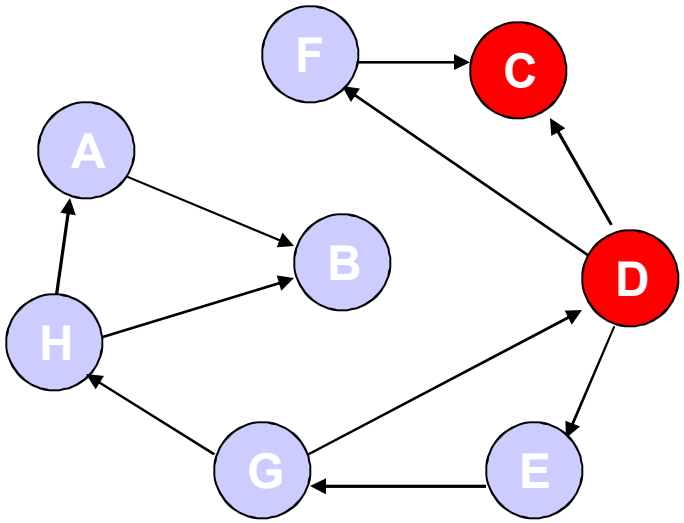
D

Visit C

The order nodes are visited:

D, C

DFS



A	
B	
C	√
D	√
E	
F	
G	
H	

C

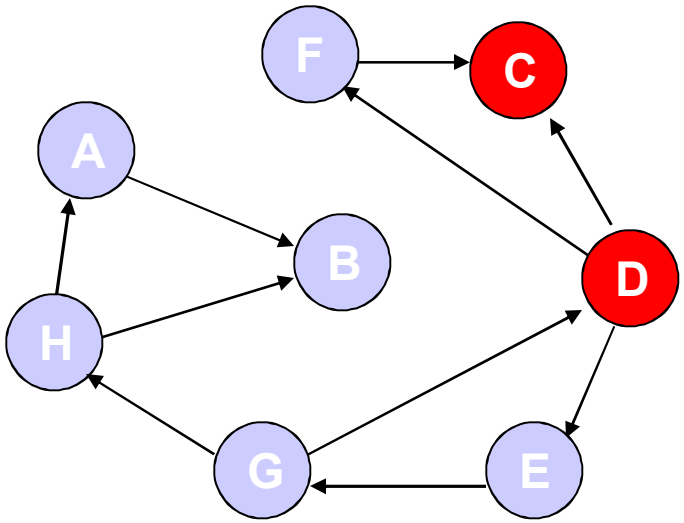
D

The order nodes are visited:

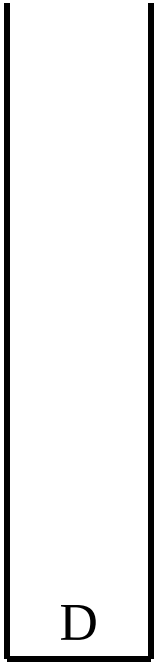
D, C

No nodes adjacent to C; cannot continue ➔ *backtrack*, i.e., pop stack and restore previous state

DFS



A	
B	
C	√
D	√
E	
F	
G	
H	

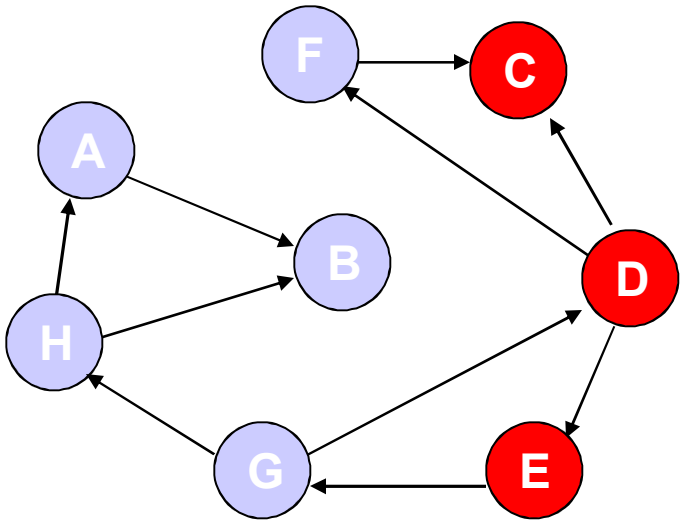


The order nodes are visited:

D, C

Back to D – C has been visited, decide to visit E next

DFS



A	
B	
C	√
D	√
E	√
F	
G	
H	

E

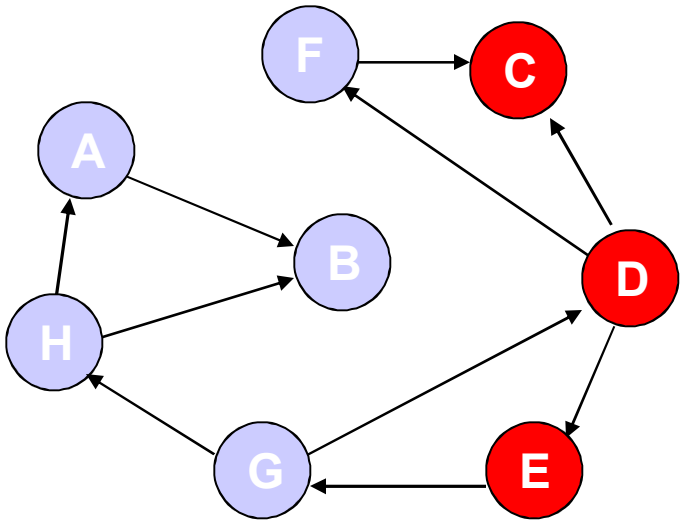
D

The order nodes are visited:

D, C, E

Back to D – C has been visited, decide to visit E next

DFS



A	
B	
C	√
D	√
E	√
F	
G	
H	

E

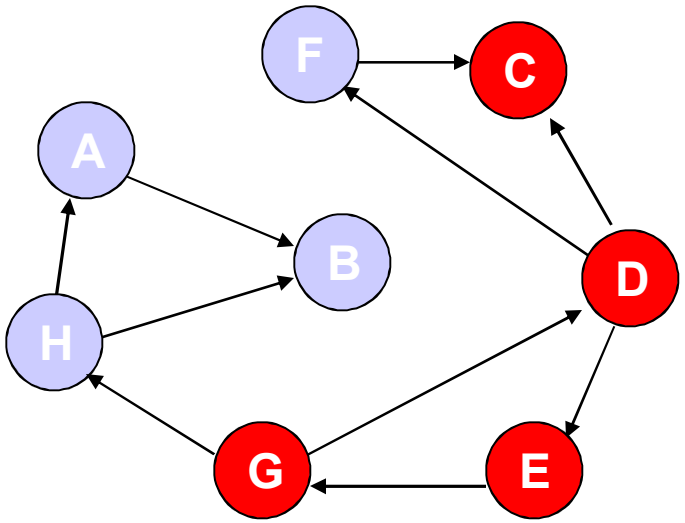
D

The order nodes are visited:

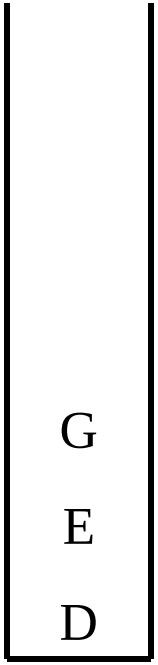
D, C, E

Only G is adjacent to E

DFS



A	
B	
C	√
D	√
E	√
F	
G	√
H	

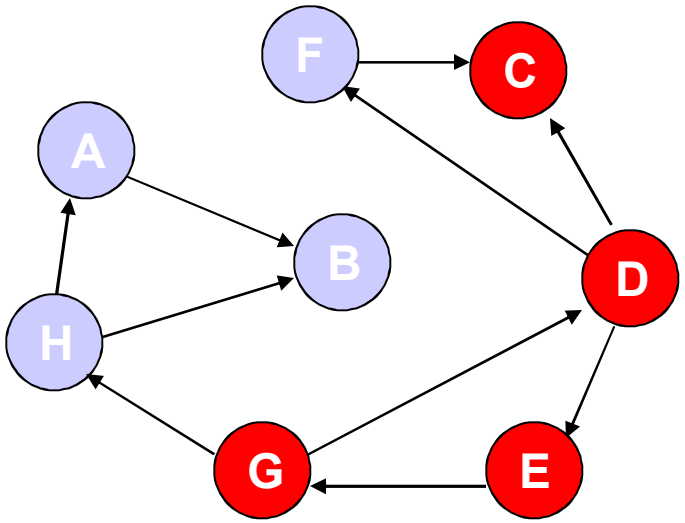


Visit G

The order nodes are visited:

D, C, E, G

DFS



A	
B	
C	√
D	√
E	√
F	
G	√
H	

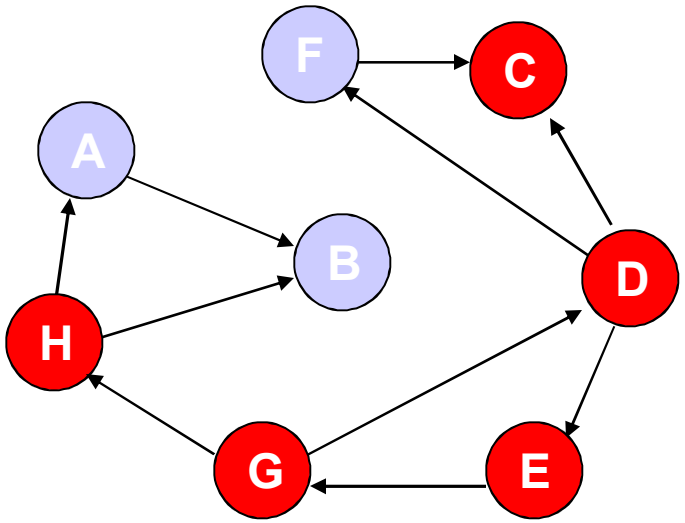
G
E
D

The order nodes are visited:

D, C, E, G

Nodes D and H are adjacent to G. D has already been visited. Decide to visit H.

DFS



A	
B	
C	√
D	√
E	√
F	
G	√
H	√

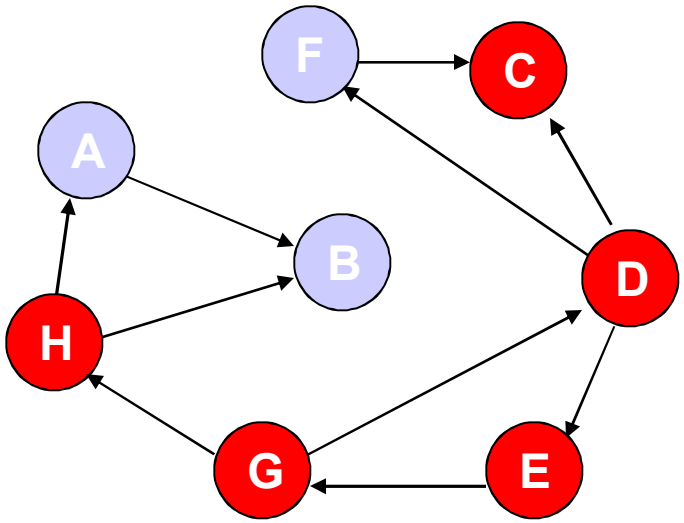
H
G
E
D

Visit H

The order nodes are visited:

D, C, E, G, H

DFS



A	
B	
C	√
D	√
E	√
F	
G	√
H	√

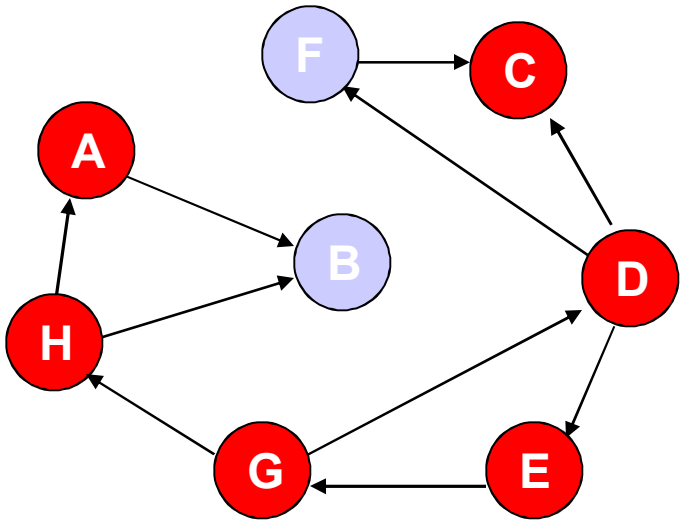
H
G
E
D

The order nodes are visited:

D, C, E, G, H

Nodes A and B are adjacent to F.
Decide to visit A next.

DFS



A	√
B	
C	√
D	√
E	√
F	
G	√
H	√

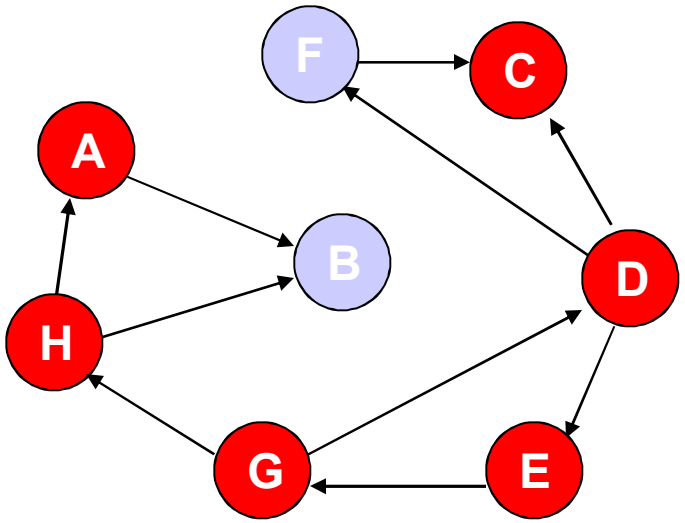
A
H
G
E
D

Visit A

The order nodes are visited:

D, C, E, G, H, A

DFS



A	√
B	
C	√
D	√
E	√
F	
G	√
H	√

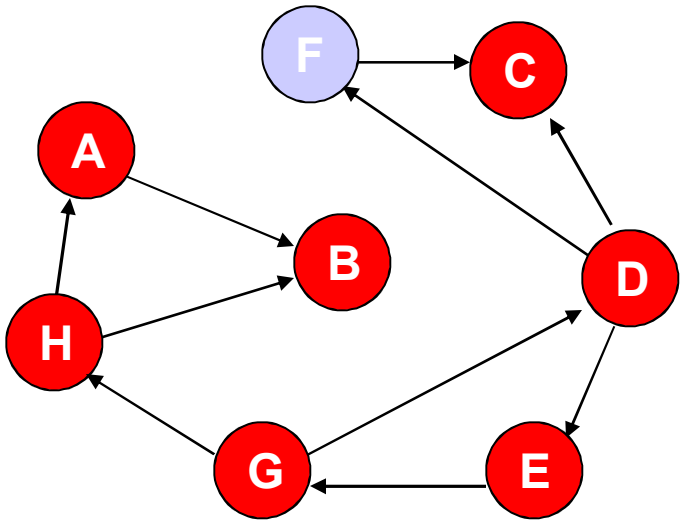
A
H
G
E
D

The order nodes are visited:

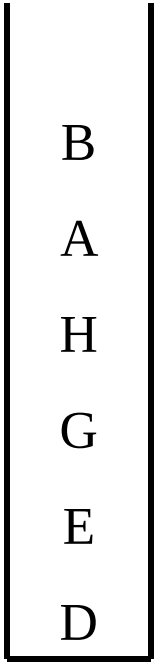
D, C, E, G, H, A

Only Node B is adjacent to A.
Decide to visit B next.

DFS



A	√
B	√
C	√
D	√
E	√
F	
G	√
H	√

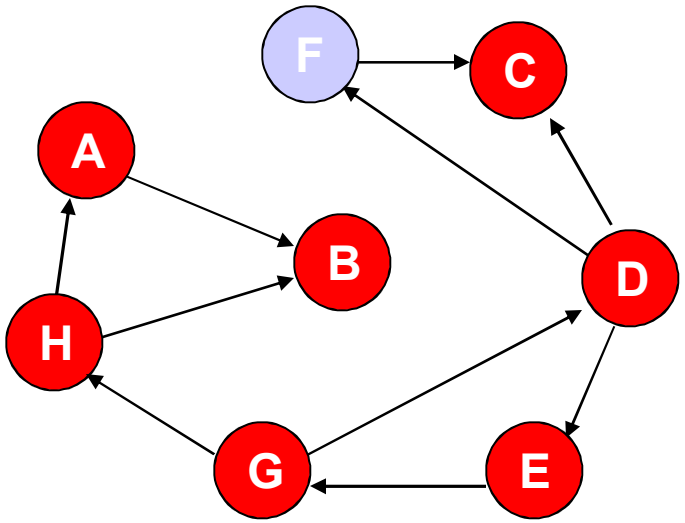


Visit B

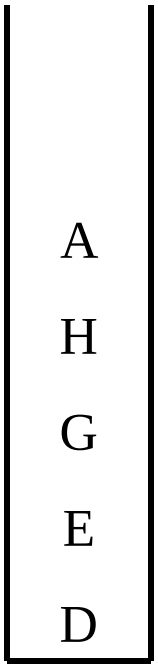
The order nodes are visited:

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

DFS



A	√
B	√
C	√
D	√
E	√
F	
G	√
H	√

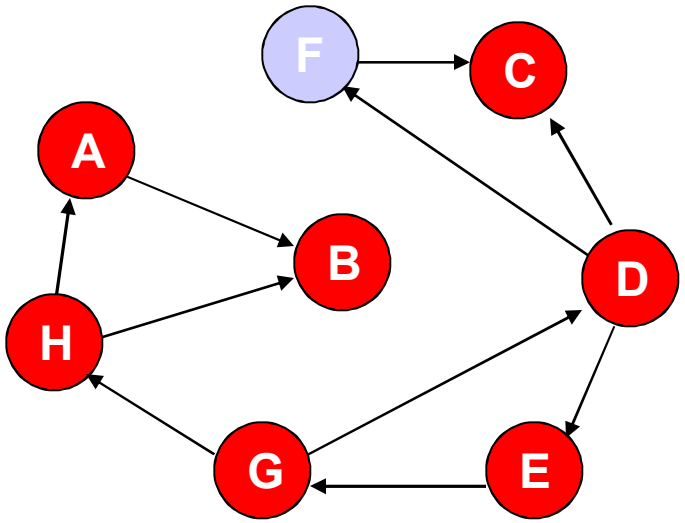


The order nodes are visited:

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

No unvisited nodes adjacent to B. Backtrack (pop the stack).

DFS



A	√
B	√
C	√
D	√
E	√
F	
G	√
H	√

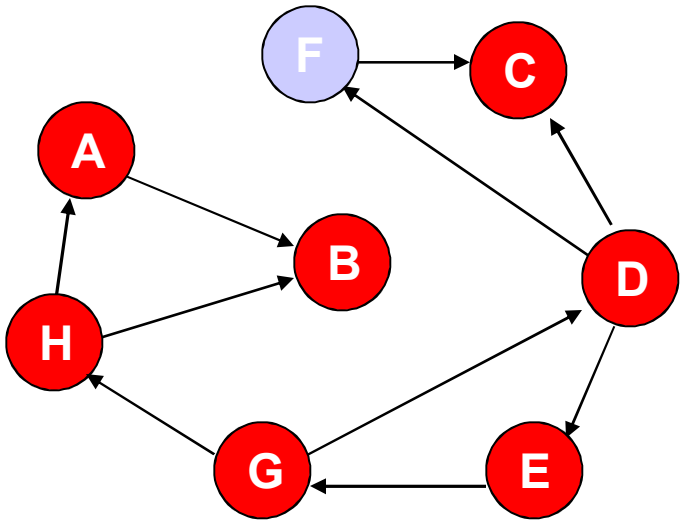
H
G
E
D

The order nodes are visited:

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

No unvisited nodes adjacent to A. Backtrack (pop the stack).

DFS



A	√
B	√
C	√
D	√
E	√
F	
G	√
H	√

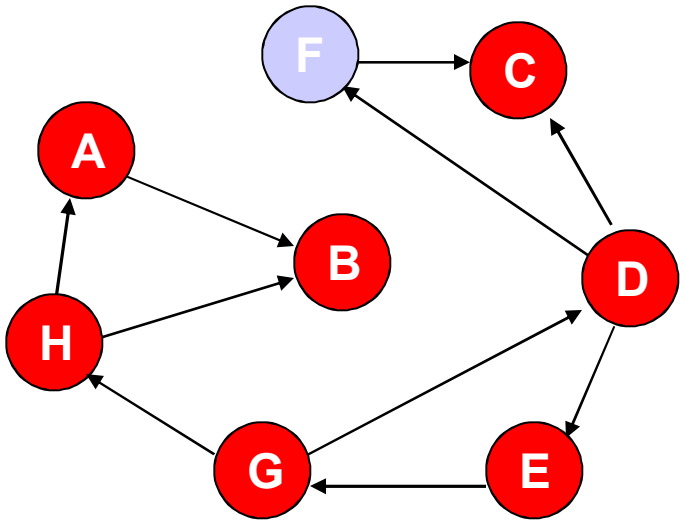
G
E
D

The order nodes are visited:

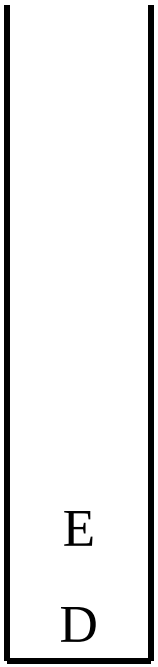
D, C, E, G, H, A, B

No unvisited nodes adjacent to H. Backtrack (pop the stack).

DFS



A	√
B	√
C	√
D	√
E	√
F	
G	√
H	√

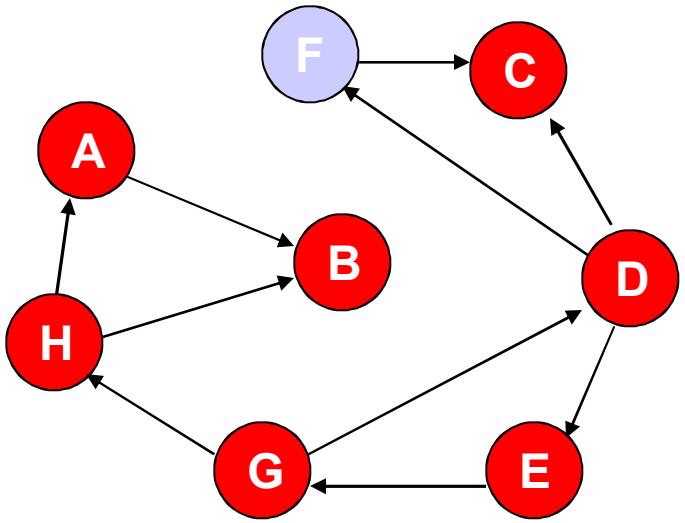


The order nodes are visited:

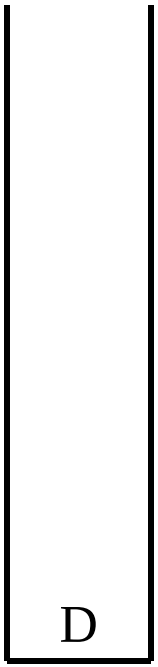
D, C, E, G, H, A, B

No unvisited nodes adjacent to G. Backtrack (pop the stack).

DFS



A	√
B	√
C	√
D	√
E	√
F	
G	√
H	√

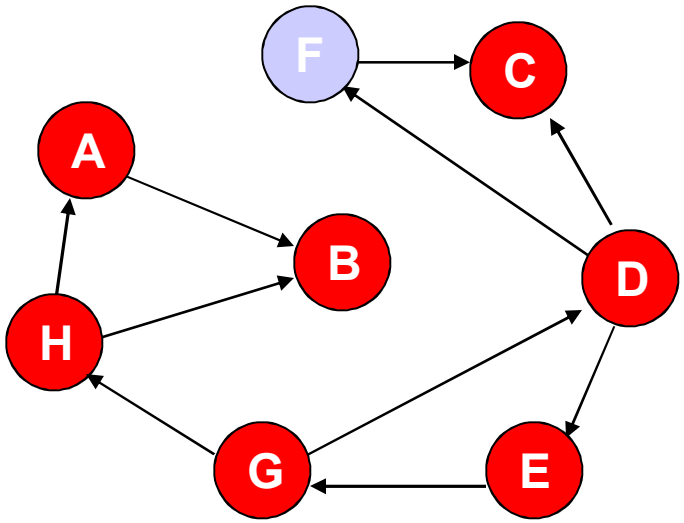


The order nodes are visited:

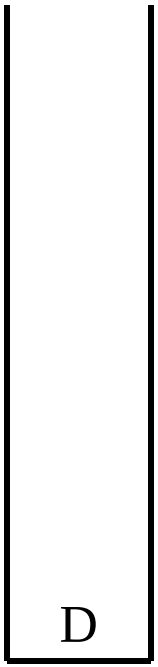
D, C, E, G, H, A, B

No unvisited nodes adjacent to E. Backtrack (pop the stack).

DFS



A	√
B	√
C	√
D	√
E	√
F	
G	√
H	√

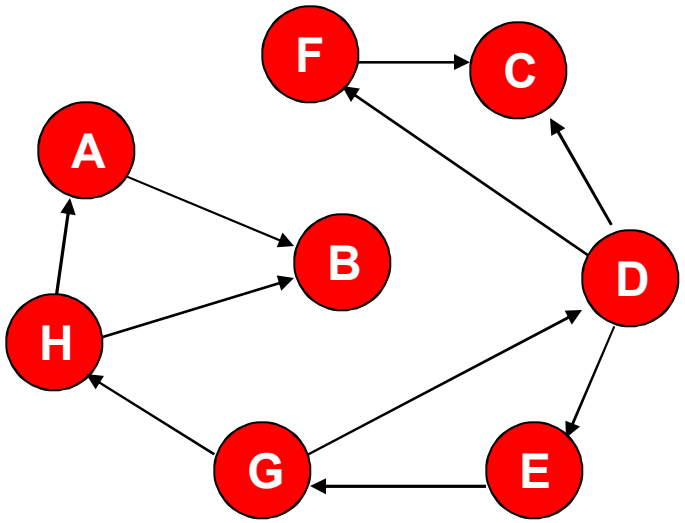


The order nodes are visited:

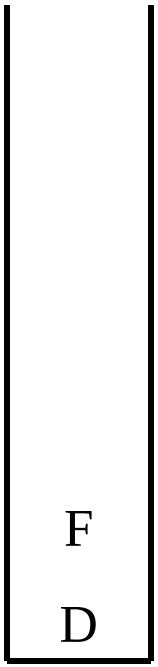
D, C, E, G, H, A, B

F is unvisited and is adjacent to D. Decide to visit F next.

DFS



A	√
B	√
C	√
D	√
E	√
F	√
G	√
H	√

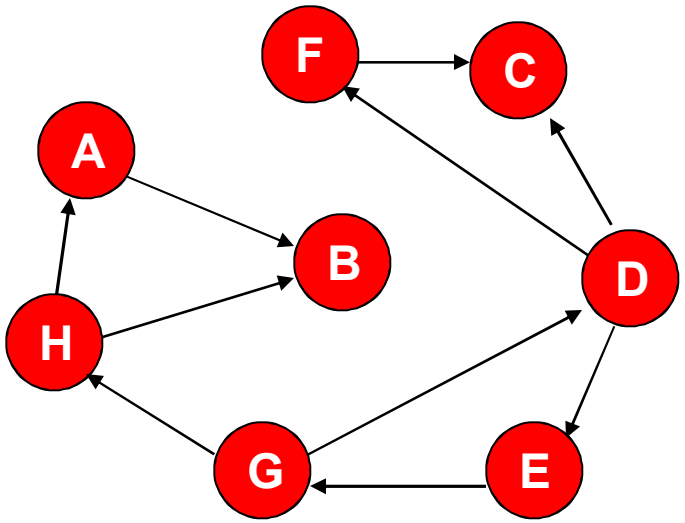


Visit F

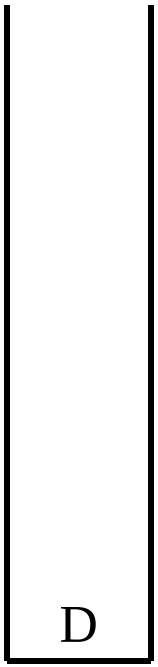
The order nodes are visited:

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

DFS



A	√
B	√
C	√
D	√
E	√
F	√
G	√
H	√

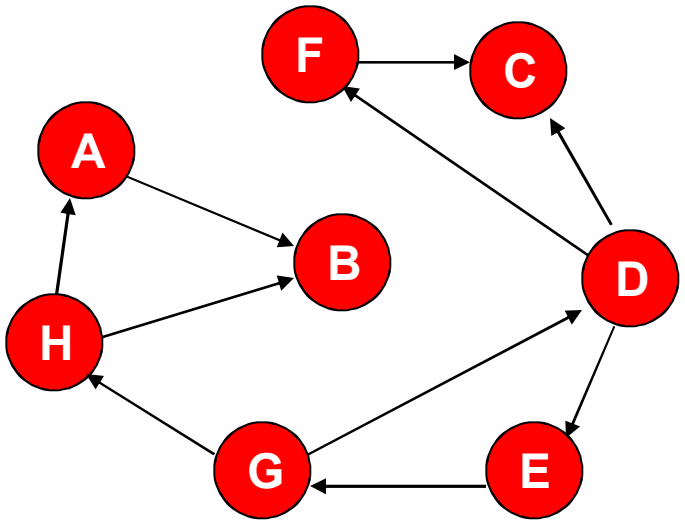


The order nodes are visited:

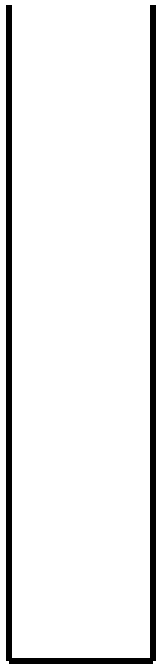
D, C, E, G, H, A, B, F

No unvisited nodes adjacent to F. Backtrack.

DFS



A	√
B	√
C	√
D	√
E	√
F	√
G	√
H	√

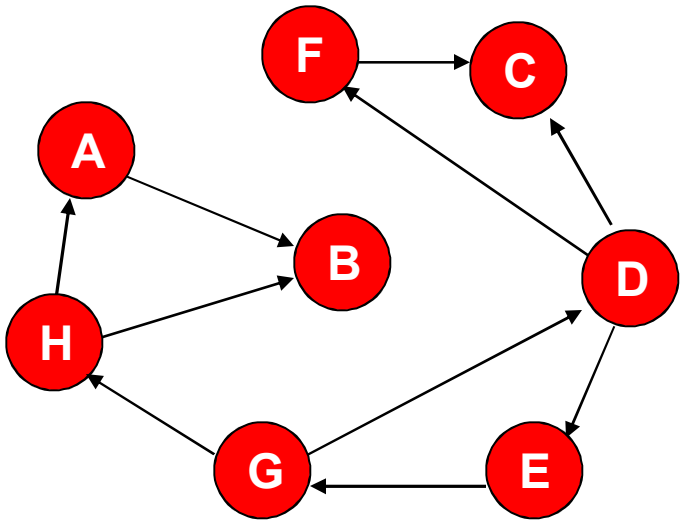


The order nodes are visited:

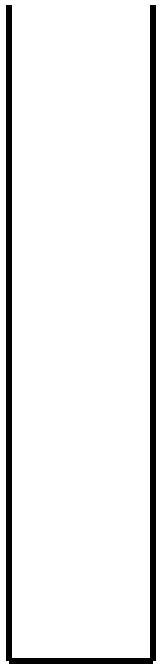
D, C, E, G, H, A, B, F

No unvisited nodes adjacent to D. Backtrack.

DFS



A	√
B	√
C	√
D	√
E	√
F	√
G	√
H	√



The order nodes are visited:

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

Stack is empty. Depth-first traversal is done.

Algorithm for Topological Sort

- TOPOLOGICAL-SORT(**G**):
  1) call DFS(G) to compute **finishing** times **f**[**v**] for each vertex **v.**
  2) as each vertex is finished, insert it onto the **front** of a linked list.
  3) return the linked list of vertices.

- Note that the result is just a list of vertices in order of **decreasing** finish times **f**[].

Edge classification by DFS

Edge (**u**,**v**) of **G** is classified as a:

(1) Tree edge iff **u** discovers **v** during the DFS: **P**[**v**] = **u.**

If (**u**, **v**) is NOT a tree edge then it is a:

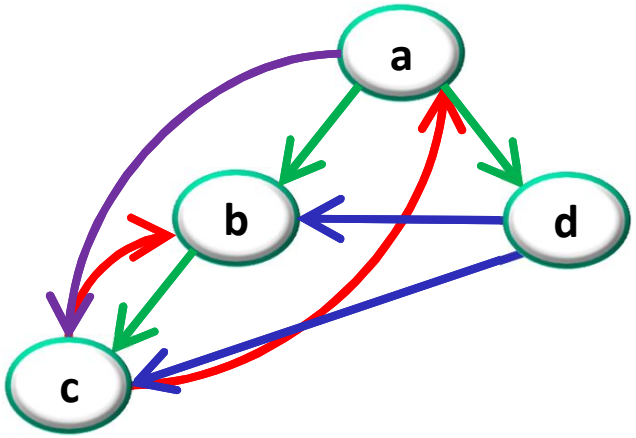
(2) **Forward** edge : Connecting **u** to a descendant **v** in the DFS tree.

(3) **Back** edge : Connecting **u** to an ancestor **v** in the DFS tree.

(4) **Cross** edge : all other edges.

Edge classification by DFS

Tree edges
Forward edges
Back edges
Cross edges

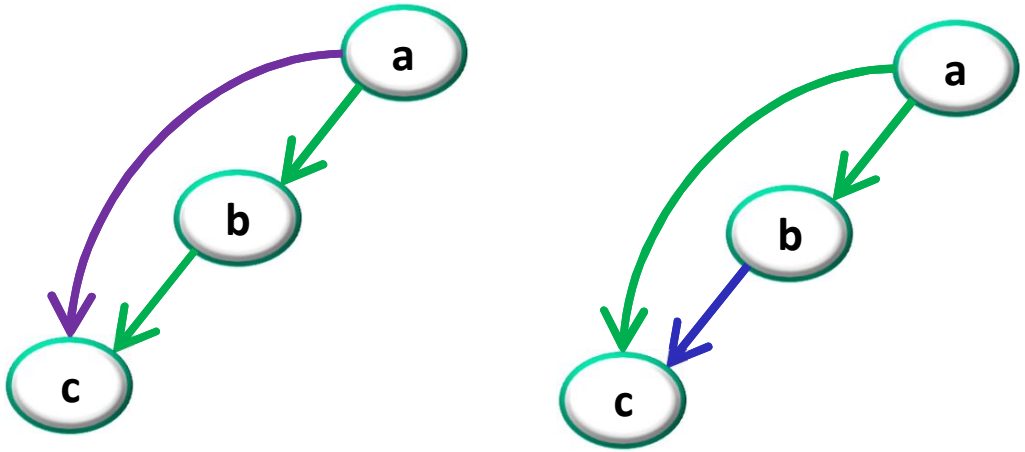


The edge classification depends on the particular DFS tree!

Edge classification by DFS

Tree edges
Forward edges
Back edges
Cross edges

Both are valid



The edge classification depends on the particular DFS tree!

DAGs and back edges

Can there be a **back** edge in a DFS on a DAG?

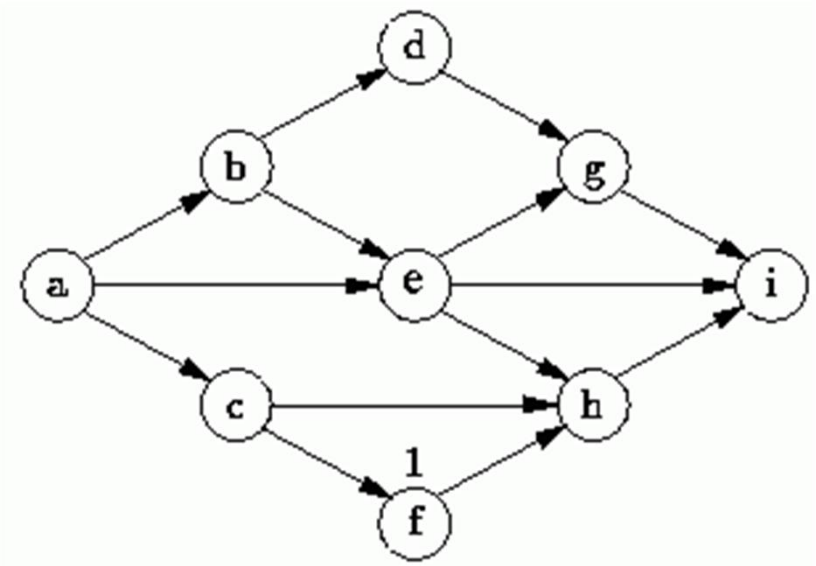
NO! Back edges close a cycle!

A graph **G** is a DAG <=> there is no back edge classified by DFS(**G**).

Topological Sort is not unique

Topological sort is not unique.

The following are all topological sort of the graph below:



s1 = {a, b, c, d, e, f, g, h, i}

s2 = {a, c, b, f, e, d, h, g, i}

s3 = {a, b, d, c, e, g, f, h, i}

s4 = {a, c, f, b, e, h, d, g, i}

etc.

Back to Topological Sort

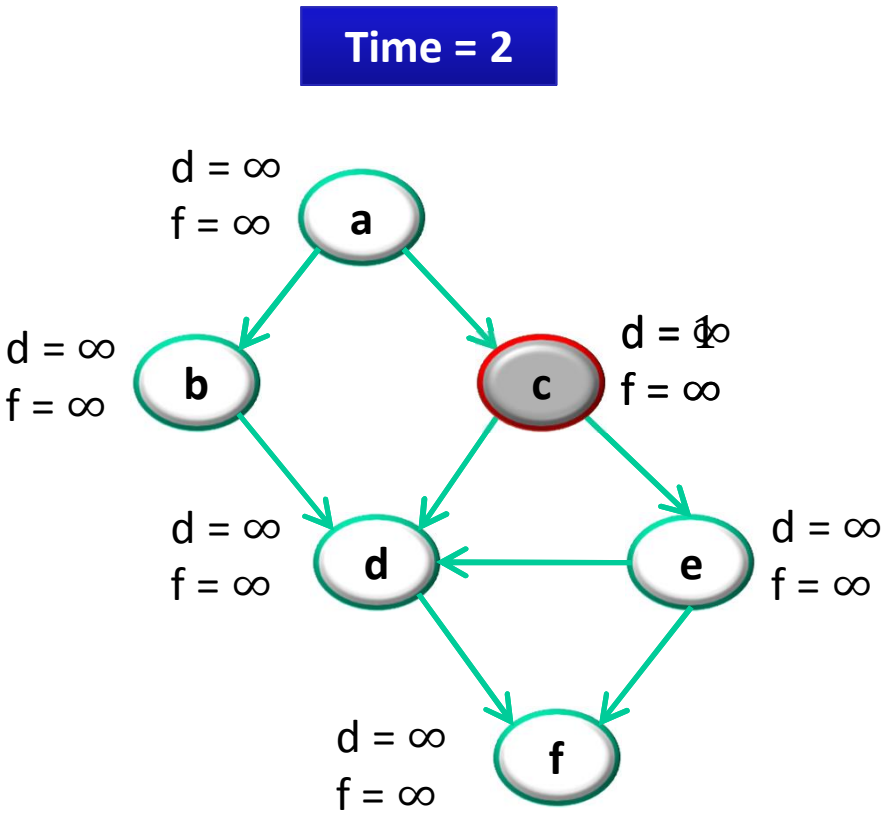
TOPOLOGICAL-SORT(**G**):

1) Call DFS(G) to compute **finishing** times **f**[**v**] for each vertex **v.**

2) As each vertex is finished, insert it onto the **front** of a linked list.

3) Return the linked list of vertices.

Topological Sort

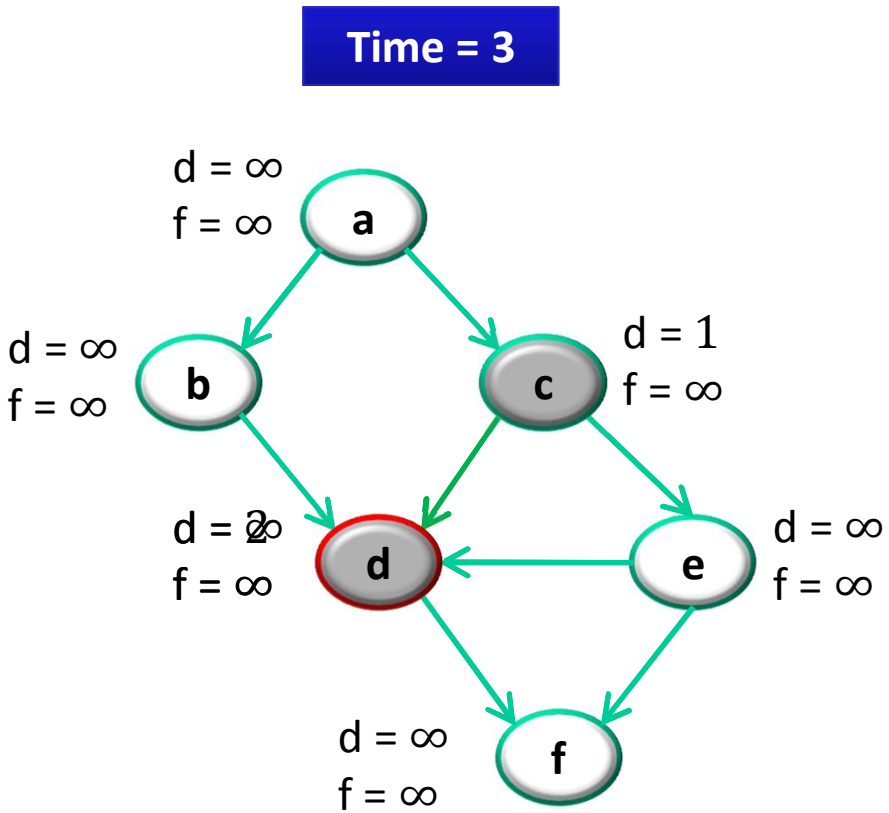


1) Call DFS(**G**) to compute the finishing times **f**[**v**]

Let's say we start the DFS from the vertex **c**

Next we discover the vertex **d**

Topological Sort



1) Call DFS(**G**) to compute the finishing times **f**[**v**]

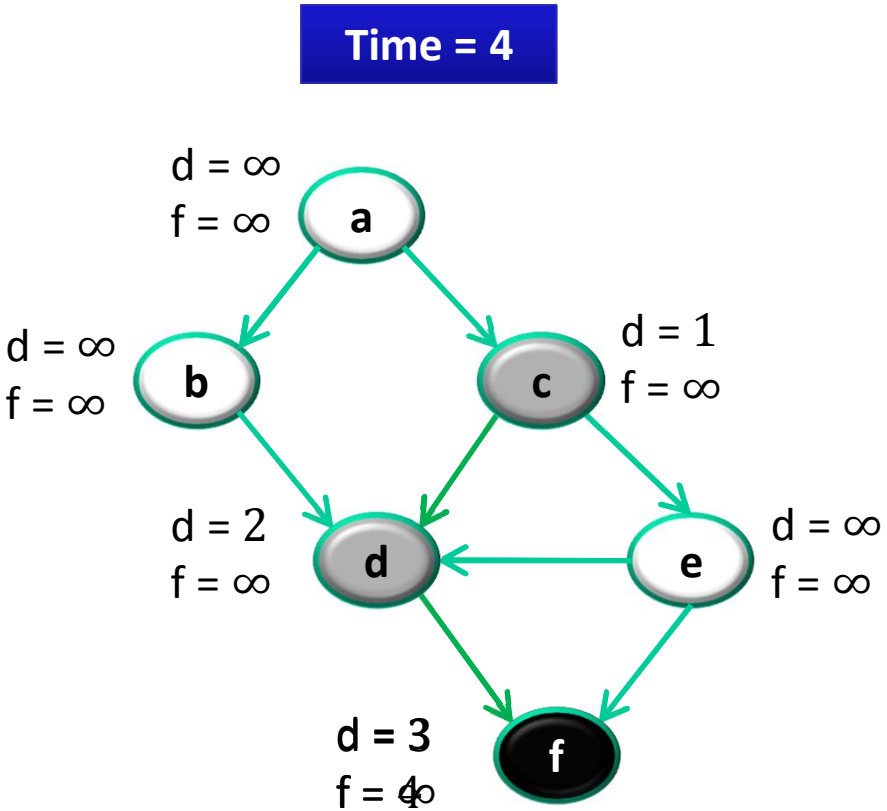
Let's say we start the DFS from the vertex **c**

Next we discover the vertex **d**

Topological Sort

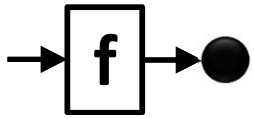
1) Call DFS(**G**) to compute the finishing times **f**[**v**]

2) as each vertex is finished, insert it onto the **front** of a linked list



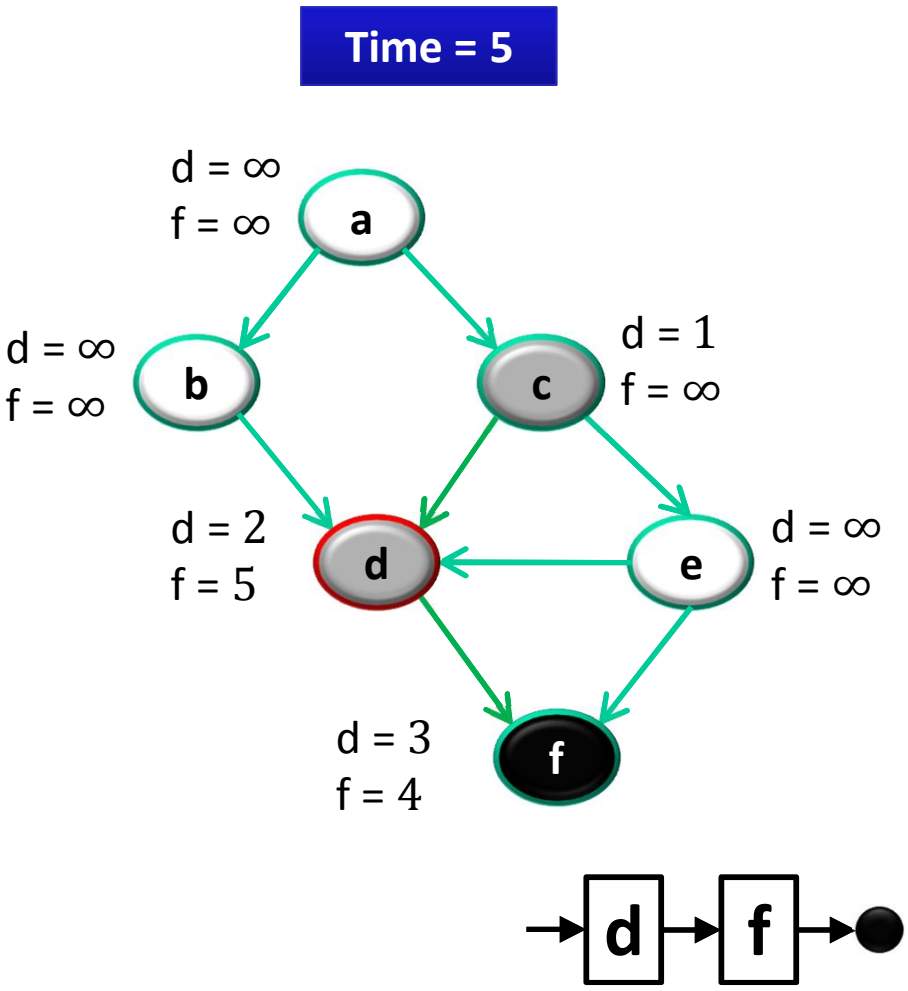
Next we discover the vertex **f**

f is done, move back to **d**



Topological Sort

1) Call DFS(**G**) to compute the finishing times **f**[**v**]



Let's say we start the DFS from the vertex **c**

Next we discover the vertex **d**

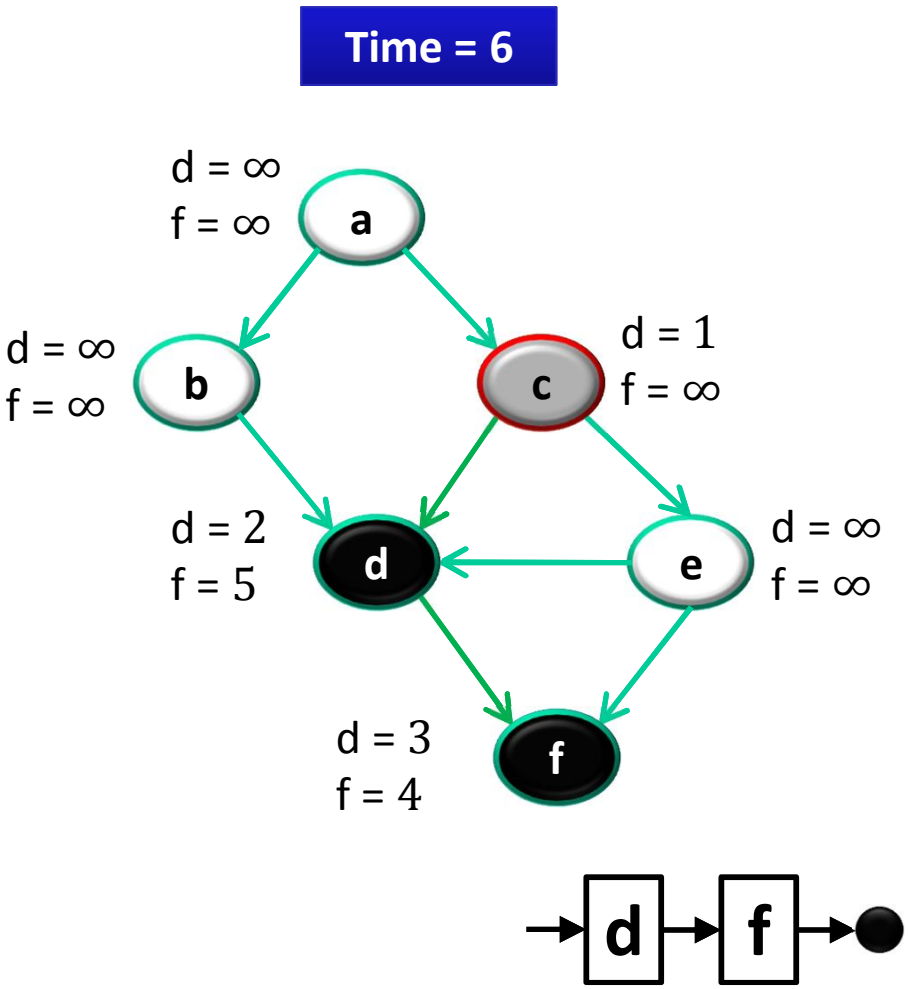
Next we discover the vertex **f**

f is done, move back to **d**

d is done, move back to **c**

Topological Sort

1) Call DFS(**G**) to compute the finishing times **f**[**v**]



Let's say we start the DFS from the vertex **c**

Next we discover the vertex **d**

Next we discover the vertex **f**

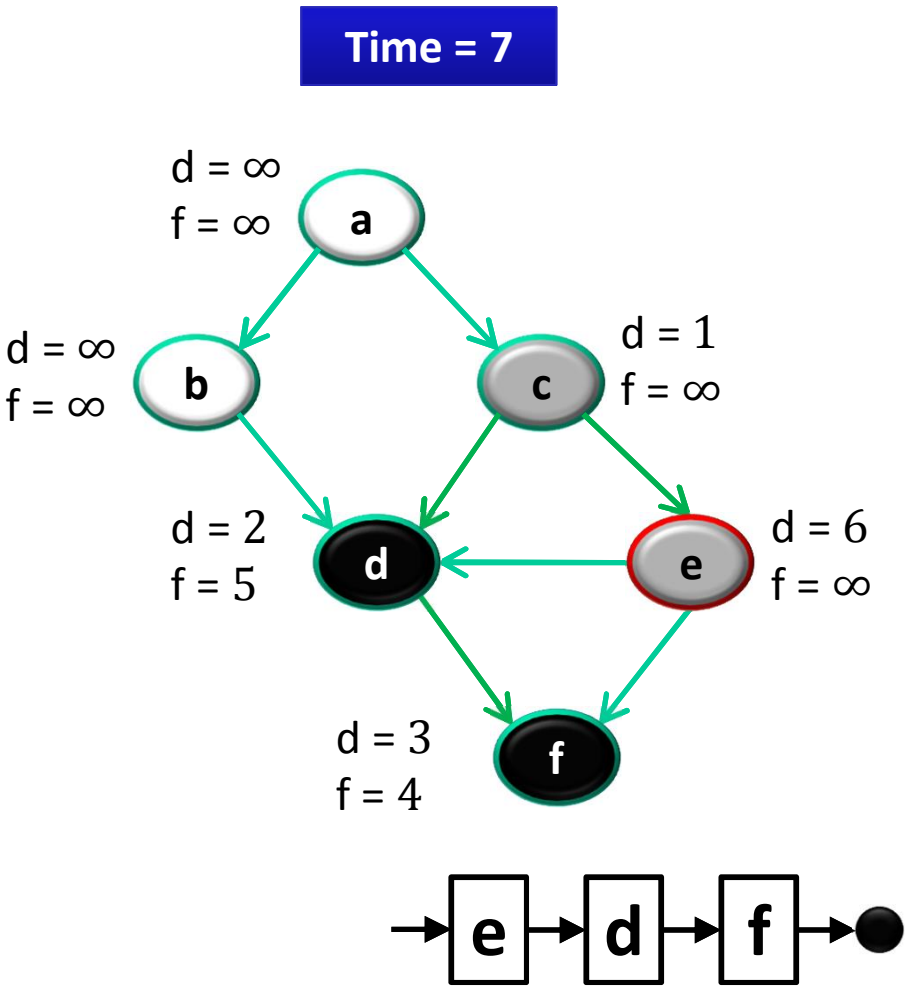
f is done, move back to **d**

d is done, move back to **c**

Next we discover the vertex **e**

Topological Sort

1) Call DFS(**G**) to compute the finishing times **f**[**v**]



Let's say we start the DFS from the vertex **c**

Next we discover the vertex **d**

Both edges from **e** are **cross edges**

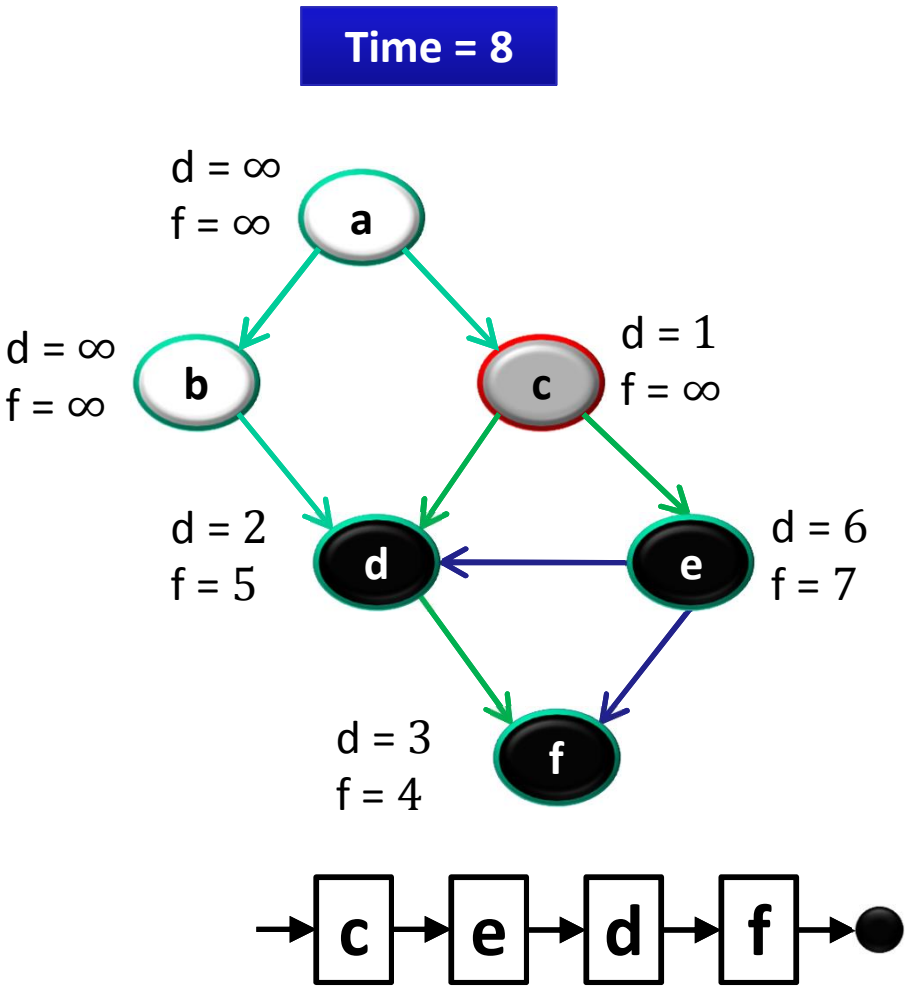
d is done, move back to **c**

Next we discover the vertex **e**

e is done, move back to **c**

Topological Sort

1) Call DFS(**G**) to compute the finishing times **f**[**v**]



Let's say we start the DFS from the vertex **c**

Just a note: If there was (**c**,**f**) edge in the graph, it would be classified as a **forward edge** (in this particular DFS run)

d is done, move back to **c**

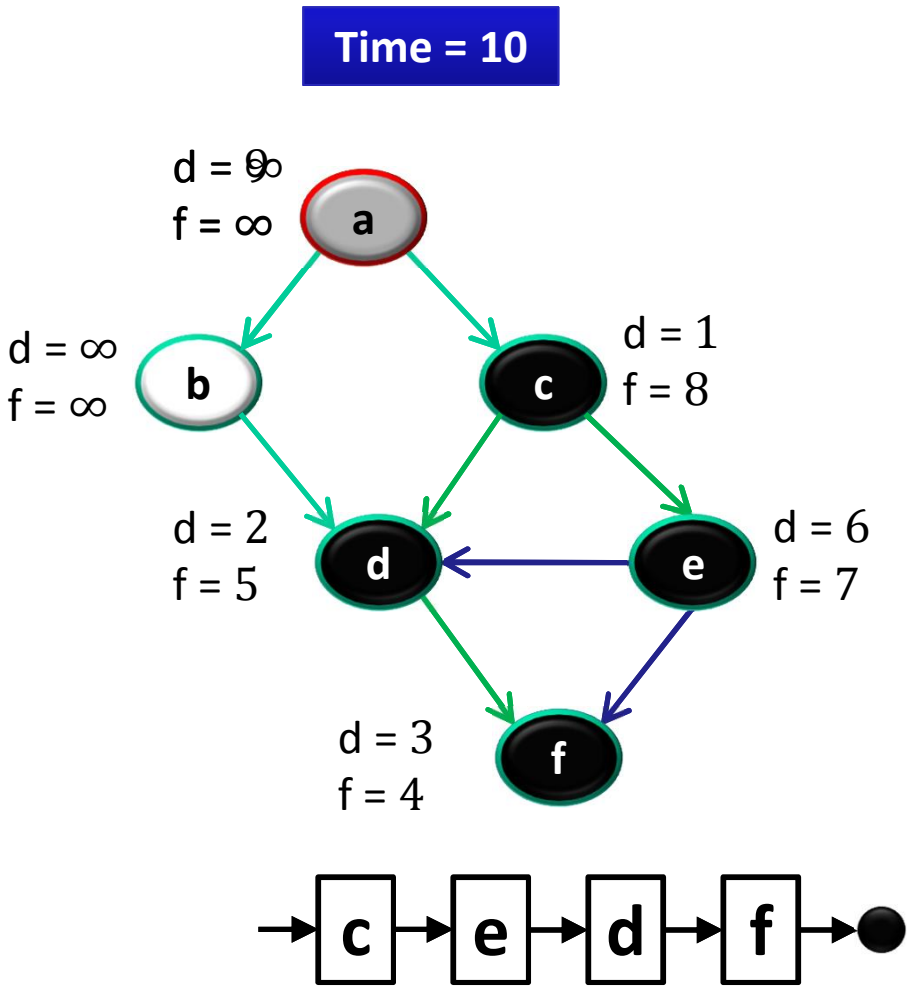
Next we discover the vertex **e**

e is done, move back to **c**

c is done as well

Topological Sort

1) Call DFS(**G**) to compute the finishing times **f**[**v**]



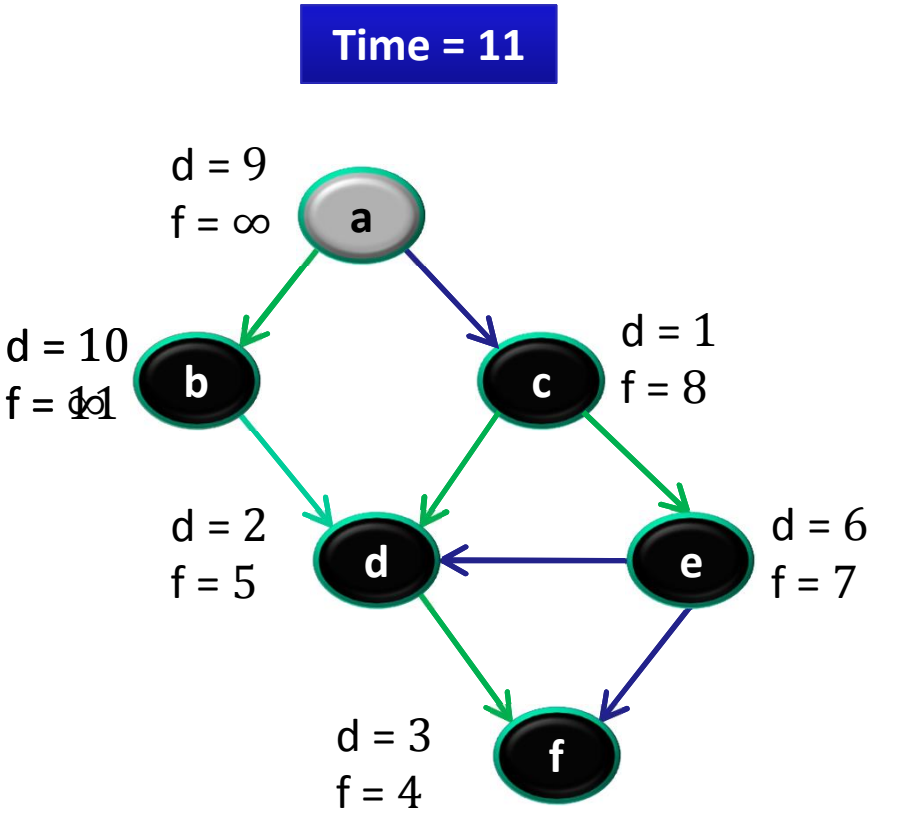
Let's now call DFS visit from the vertex **a**

Next we discover the vertex **c**, but **c** was already processed => (**a**,**c**) is a cross edge

Next we discover the vertex **b**

Topological Sort

1) Call DFS(**G**) to compute the finishing times **f**[**v**]

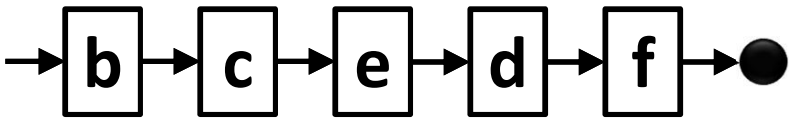


Let's now call DFS visit from the vertex **a**

Next we discover the vertex **c**, but **c** was already processed => (**a**,**c**) is a cross edge

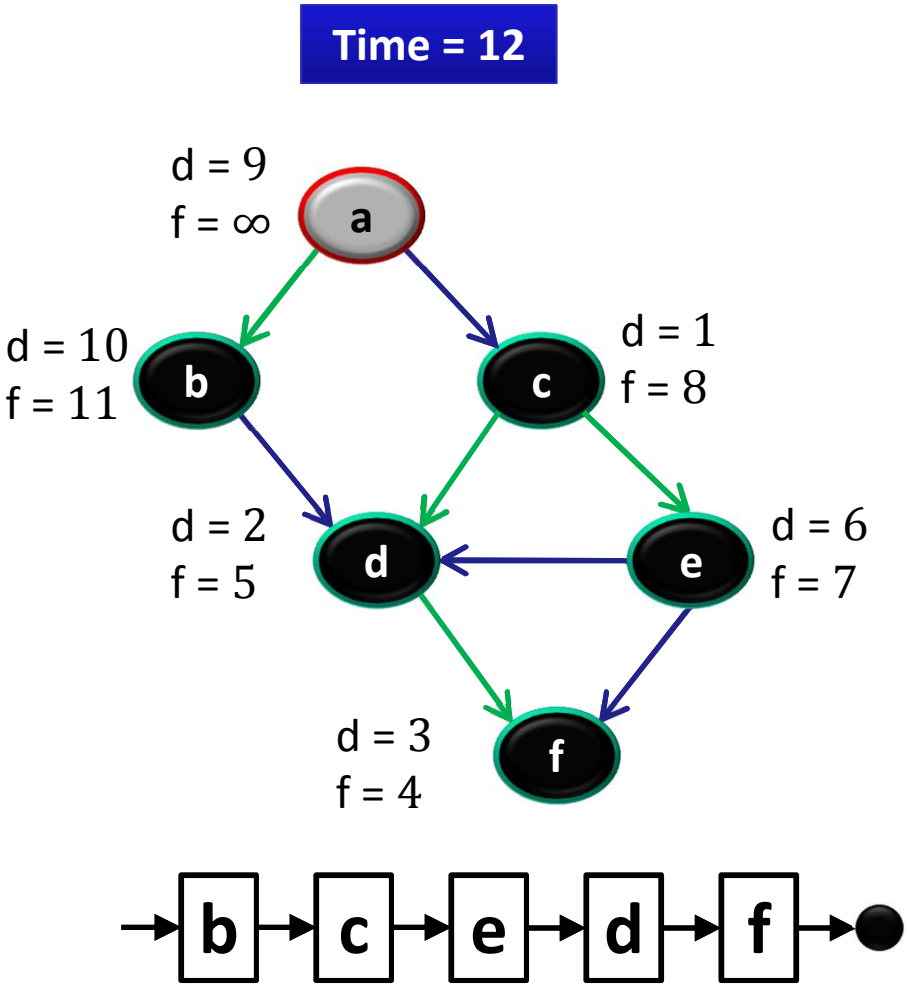
Next we discover the vertex **b**

b is done as (**b**,**d**) is a cross edge => now move back to **c**



Topological Sort

1) Call DFS(**G**) to compute the finishing times **f**[**v**]



Let's now call DFS visit from the vertex **a**

Next we discover the vertex **c**, but **c** was already processed => (**a**,**c**) is a cross edge

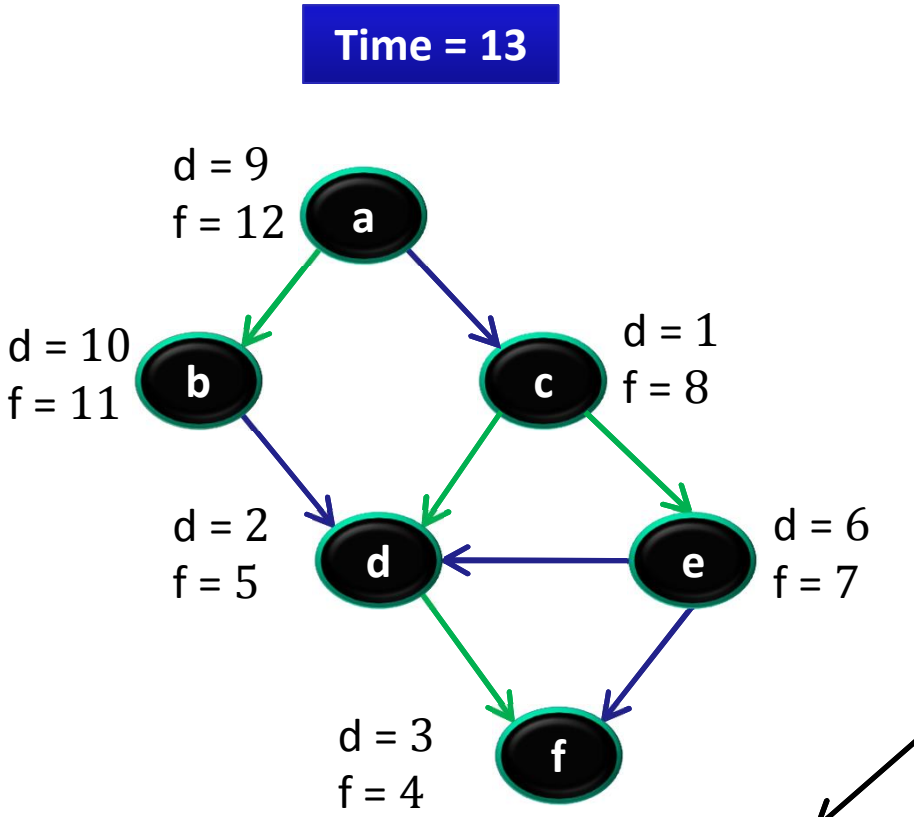
Next we discover the vertex **b**

b is done as (**b**,**d**) is a cross edge => now move back to **c**

a is done as well

Topological Sort

1) Call DFS(**G**) to compute the finishing times **f**[**v**]



WE HAVE THE RESULT!

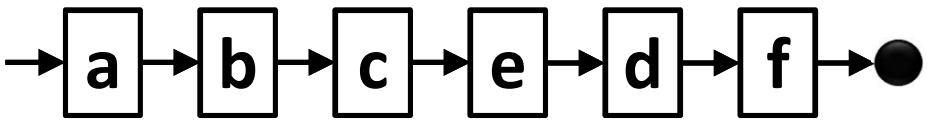
3) return the linked list of vertices

=> (**a**,**c**) is a cross edge

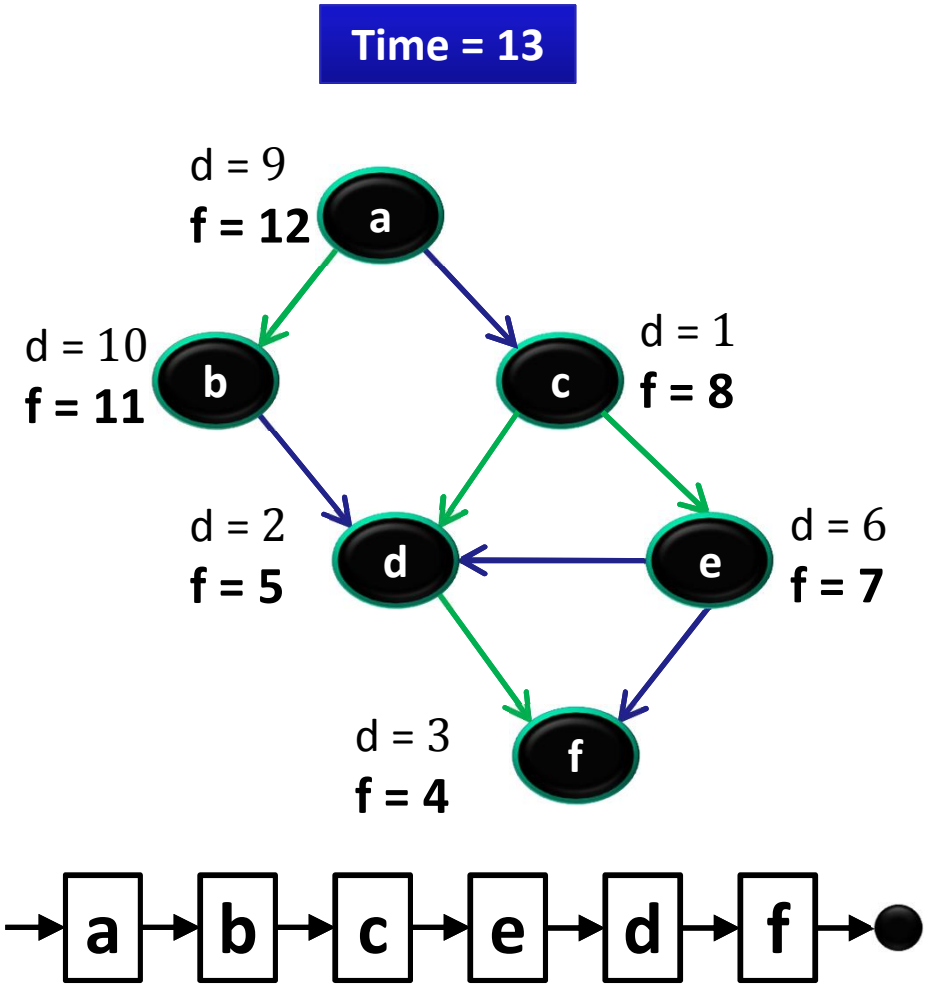
Next we discover the vertex **b**

b is done as (**b**,**d**) is a cross edge => now move back to **a**

a is done as well



Topological Sort



The linked list is sorted in **decreasing** order of finishing times **f**[]

Try yourself with different vertex order for DFS visit

Note: If you redraw the graph so that all vertices are in a line ordered by a valid topological sort, then all edges point „**from left to right**“

Time complexity of TS(G)

Running time of topological sort:

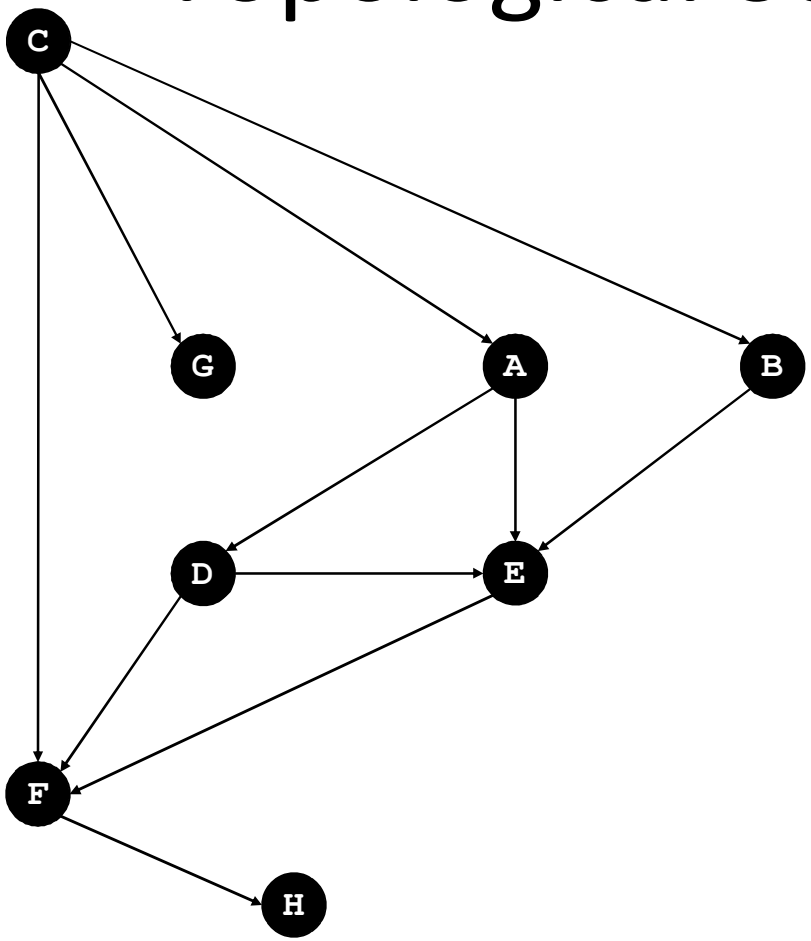
$$\Theta(n + m)$$

where $n = |V|$ and $m = |E|$

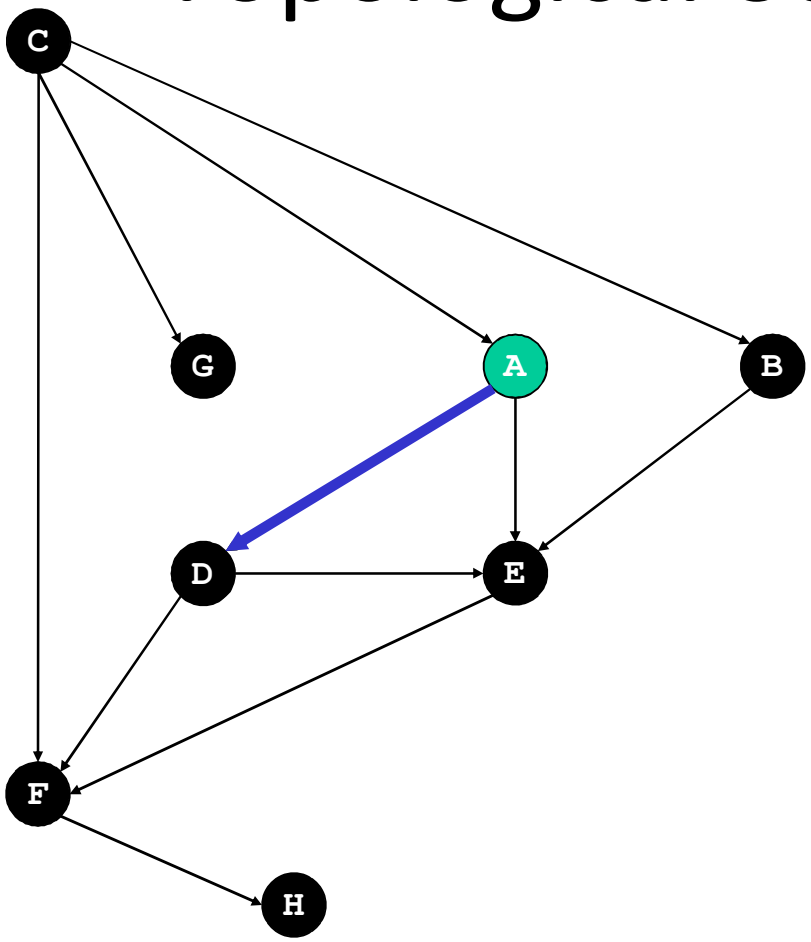
Why?

Depth first search takes $\Theta(n + m)$ time in the worst case, and inserting into the front of a linked list takes $\Theta(1)$ time.

Topological Sort: DFS

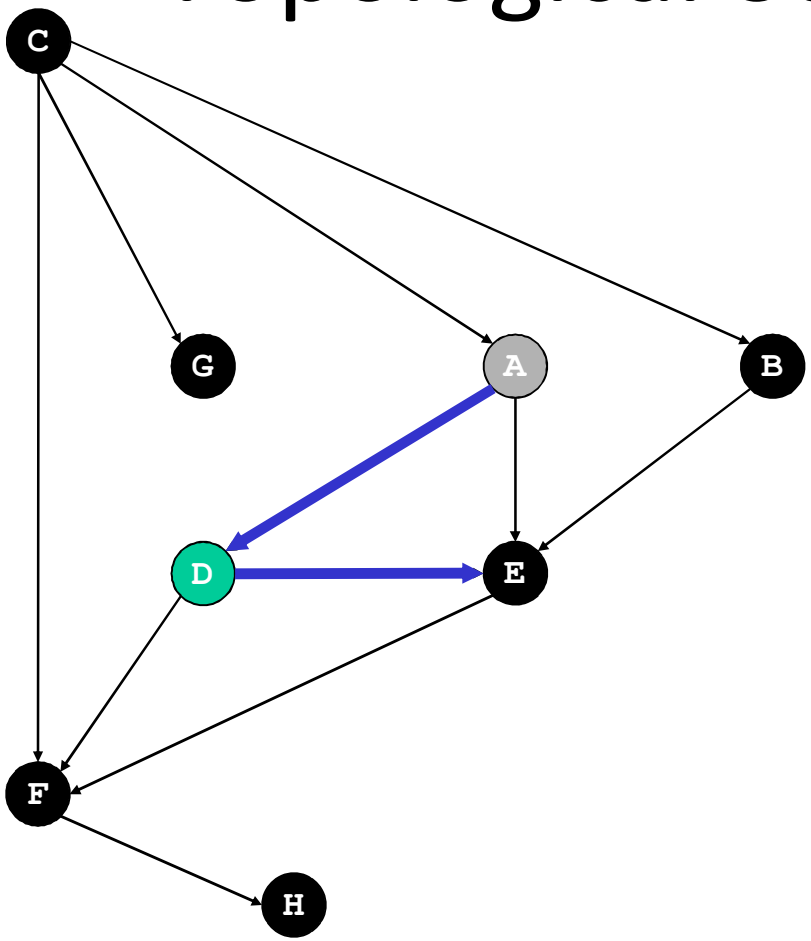


Topological Sort: DFS



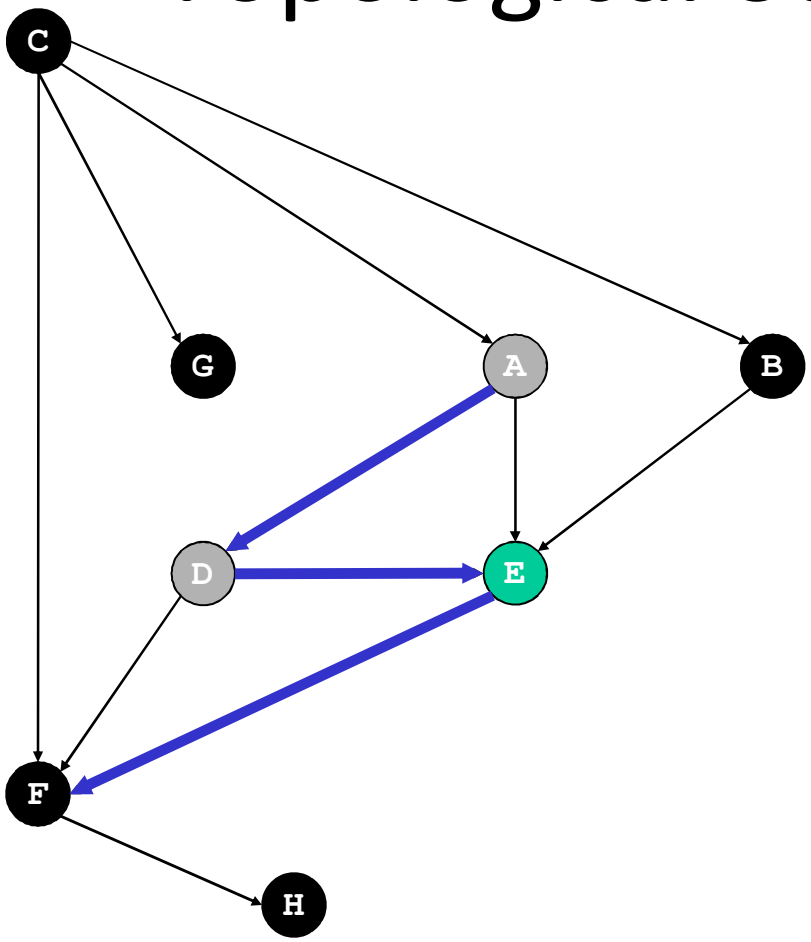

```
dfs(A)
```

Topological Sort: DFS



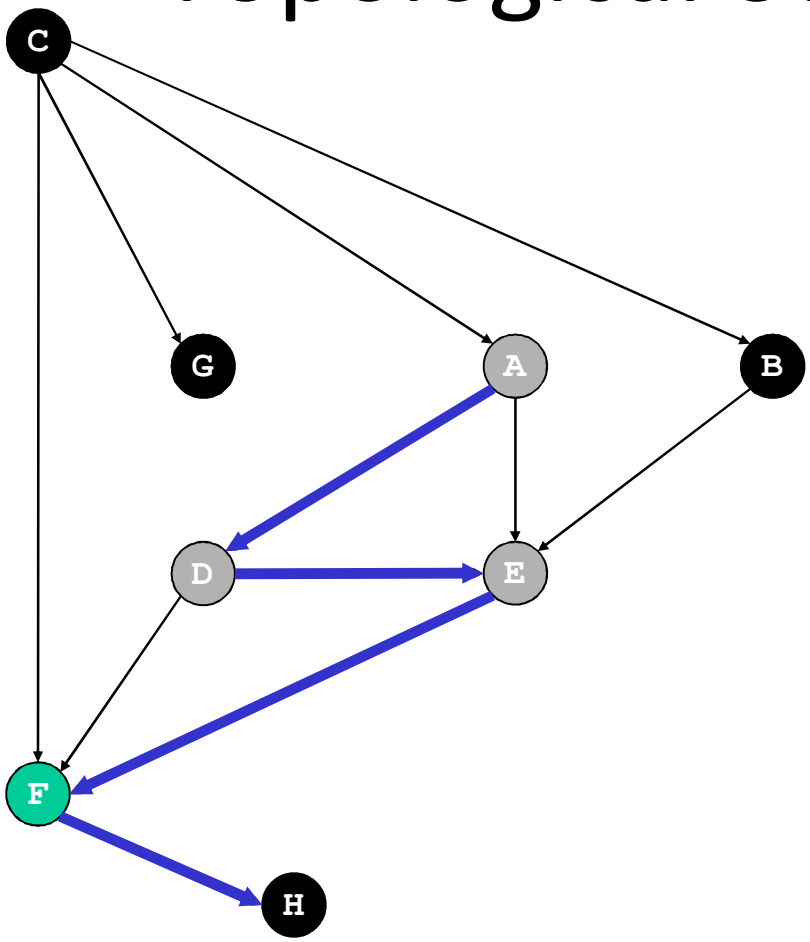

```
dfs(A)
  dfs(D)
```

Topological Sort: DFS



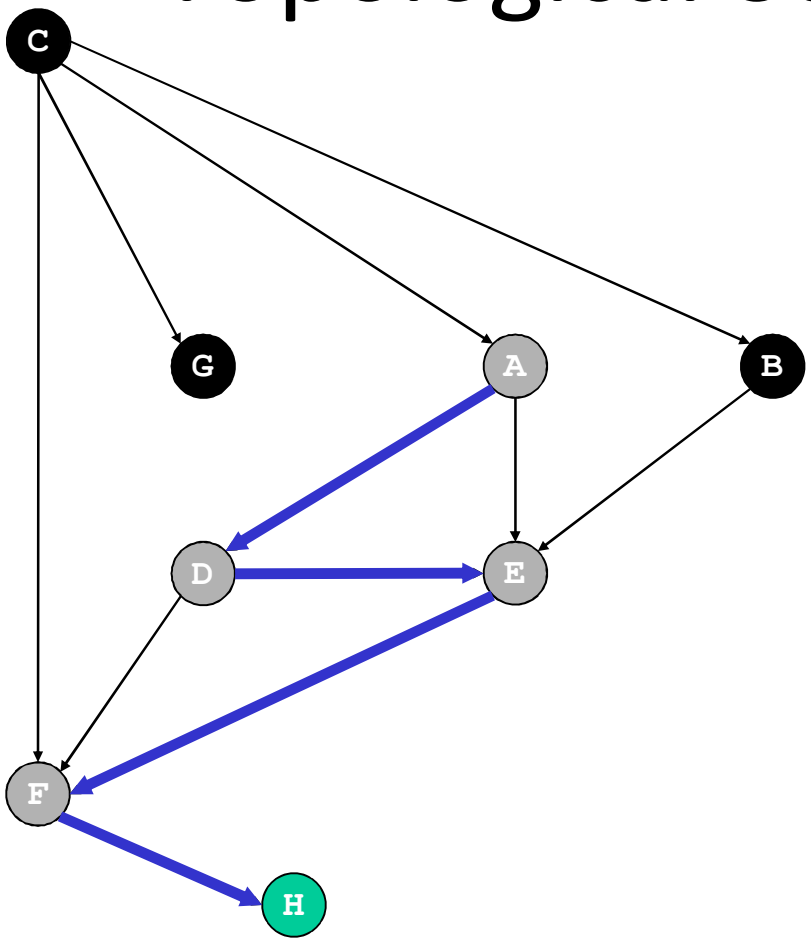

```
dfs(A)
  dfs(D)
    dfs(E)
```

Topological Sort: DFS



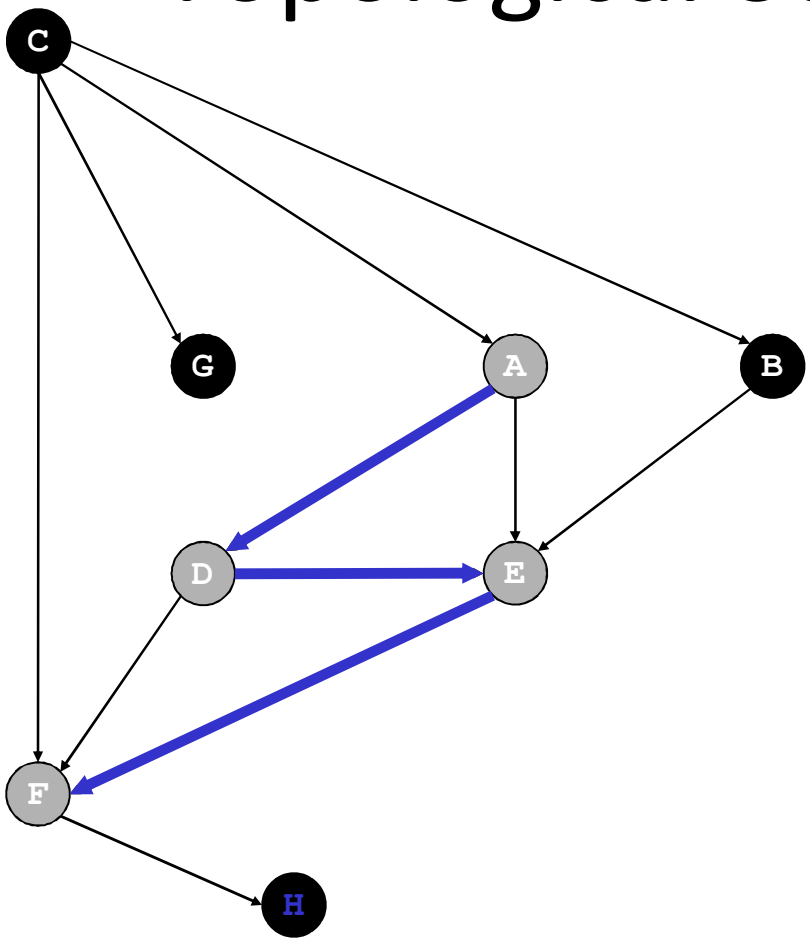

```
dfs(A)
  dfs(D)
    dfs(E)
      dfs(F)
```

Topological Sort: DFS




```
dfs(A)
  dfs(D)
    dfs(E)
      dfs(F)
        dfs(H)
```

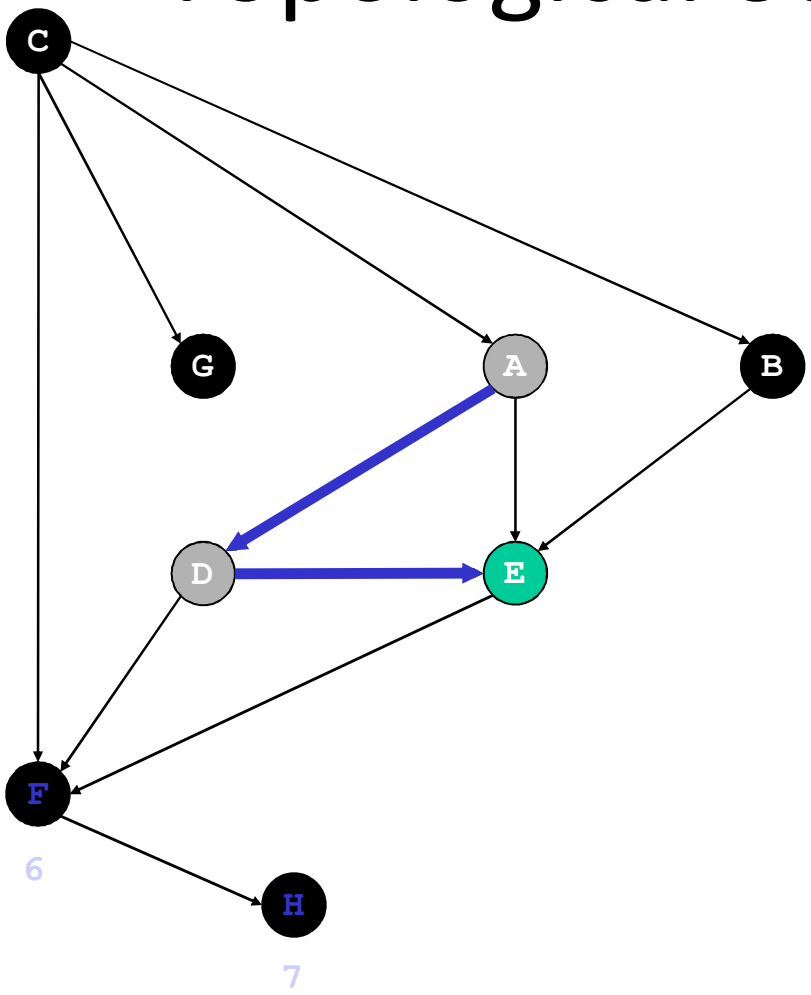
Topological Sort: DFS




```
dfs(A)
  dfs(D)
    dfs(E)
      dfs(F)
```

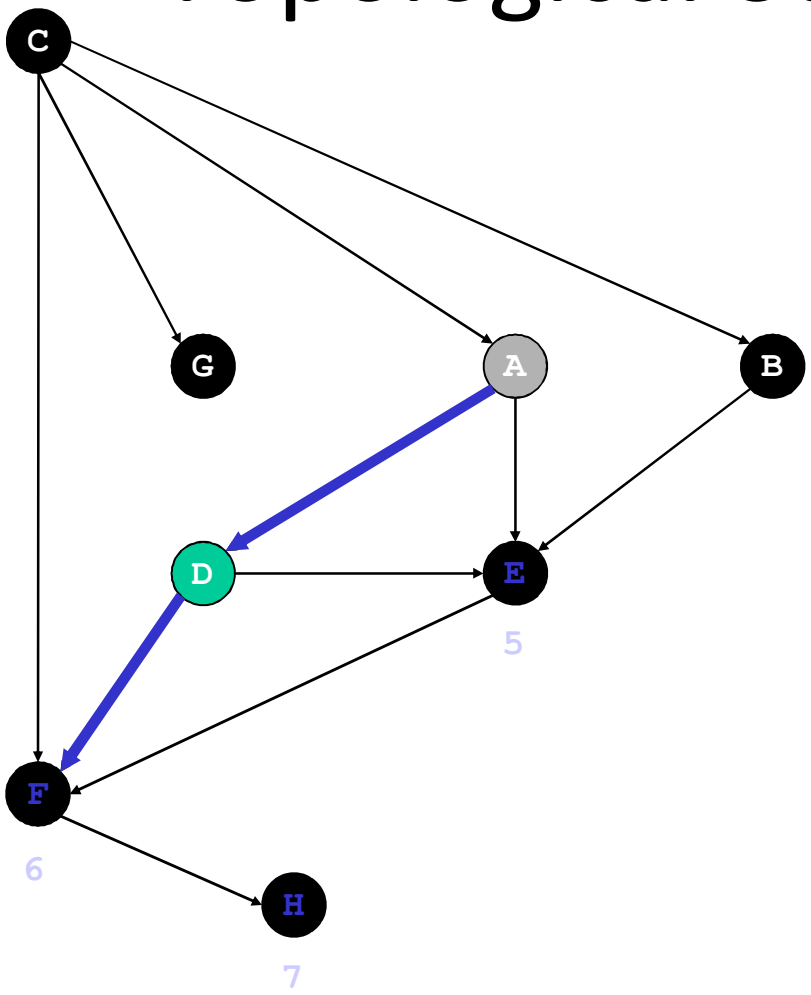
7

Topological Sort: DFS



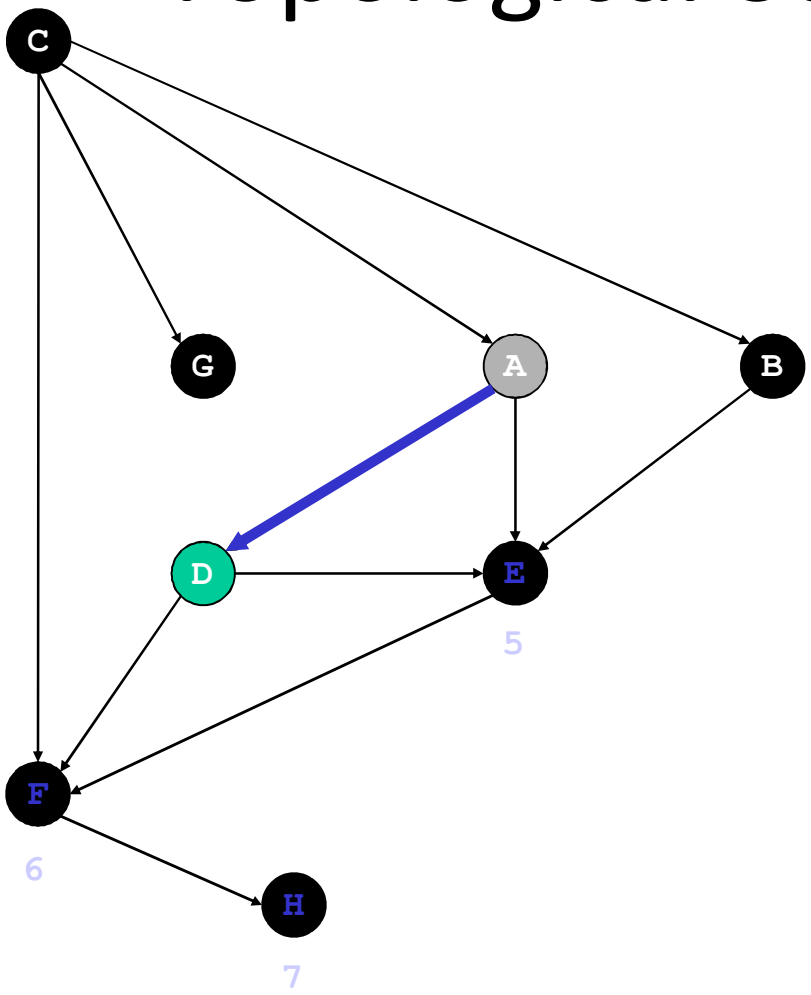

```
dfs(A)
  dfs(D)
    dfs(E)
```

Topological Sort: DFS



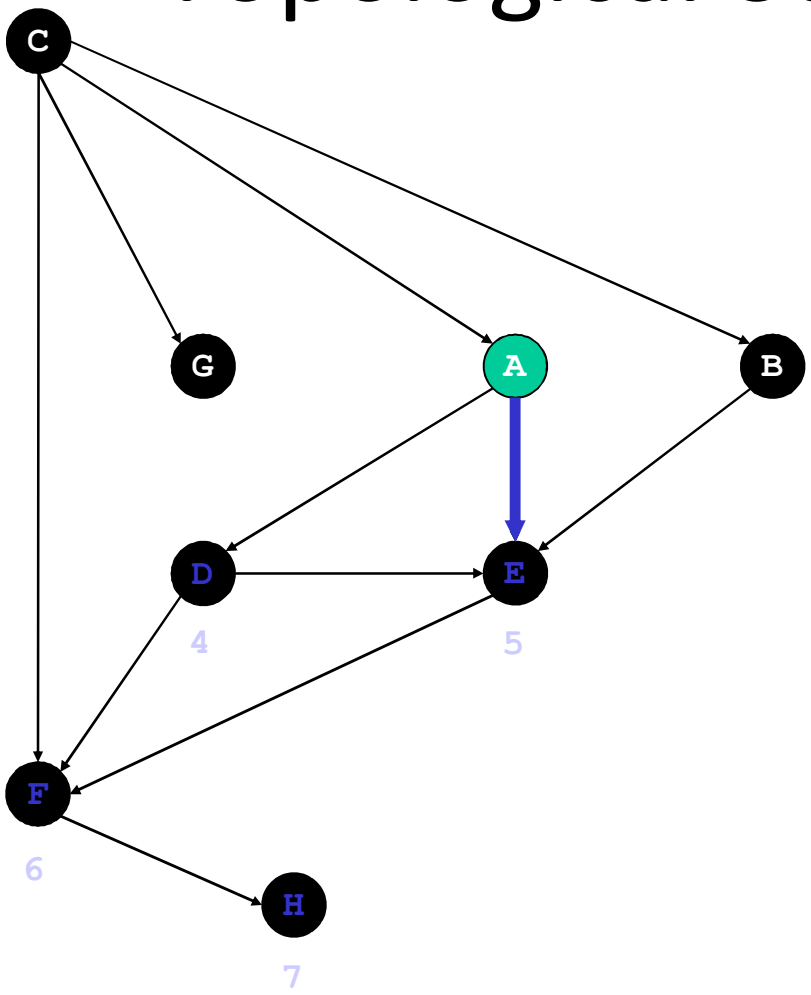

```
dfs(A)
  dfs(D)
```

Topological Sort: DFS



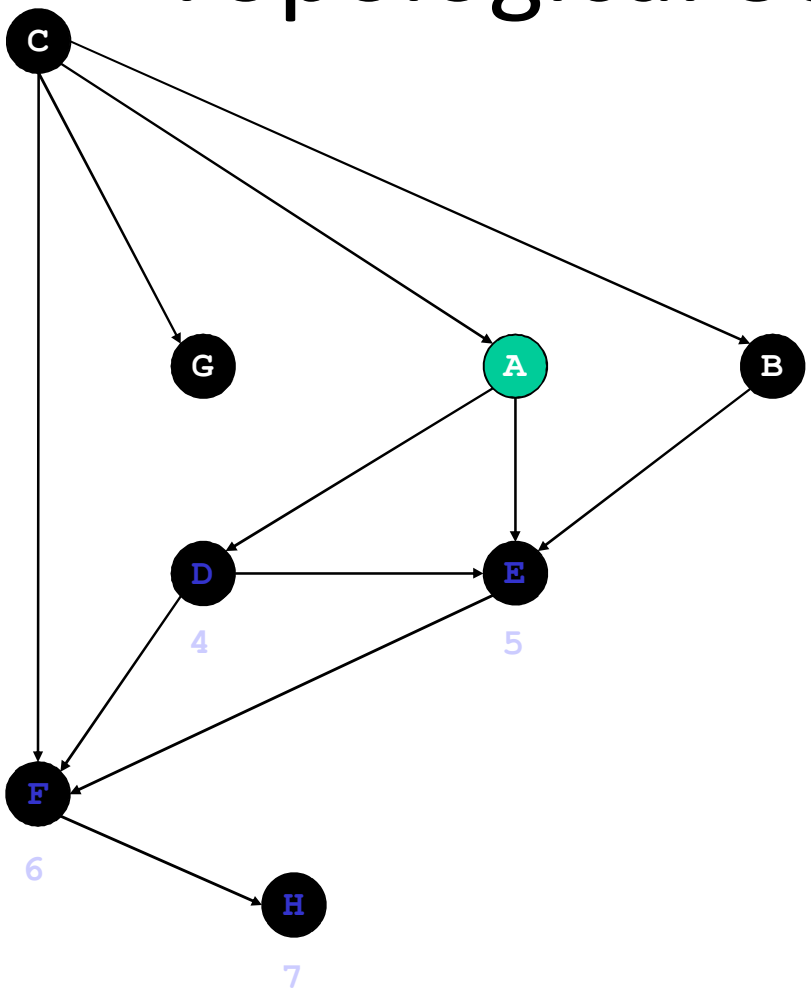

```
dfs(A)
  dfs(D)
```

Topological Sort: DFS



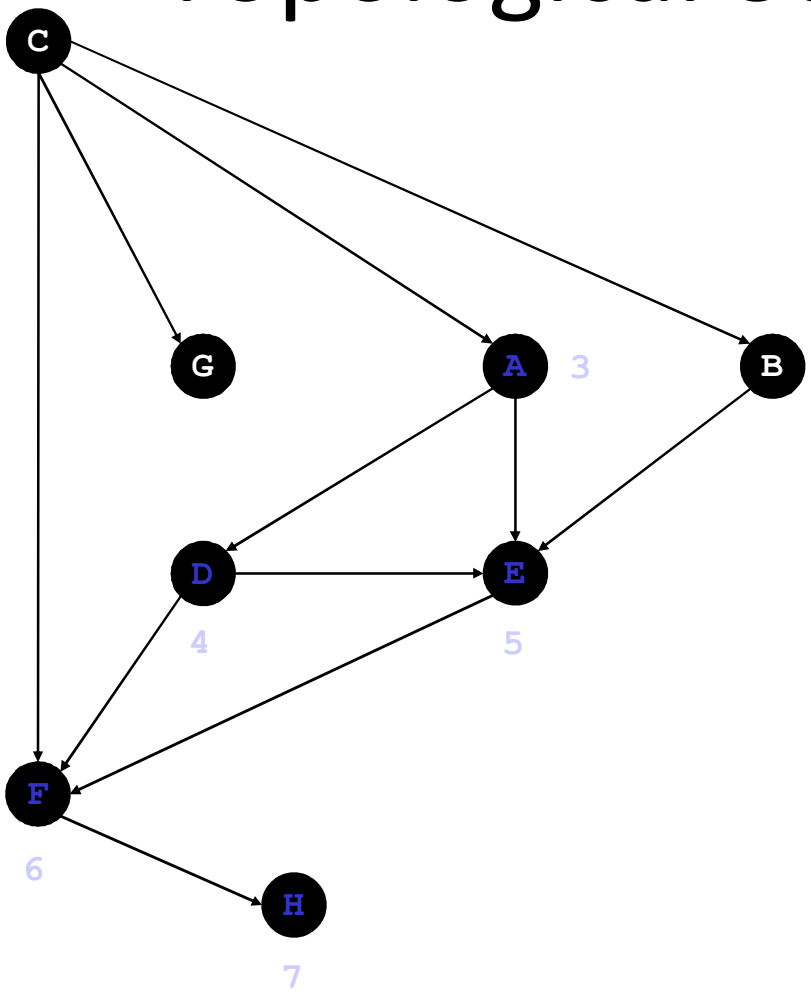

```
dfs(A)
```

Topological Sort: DFS

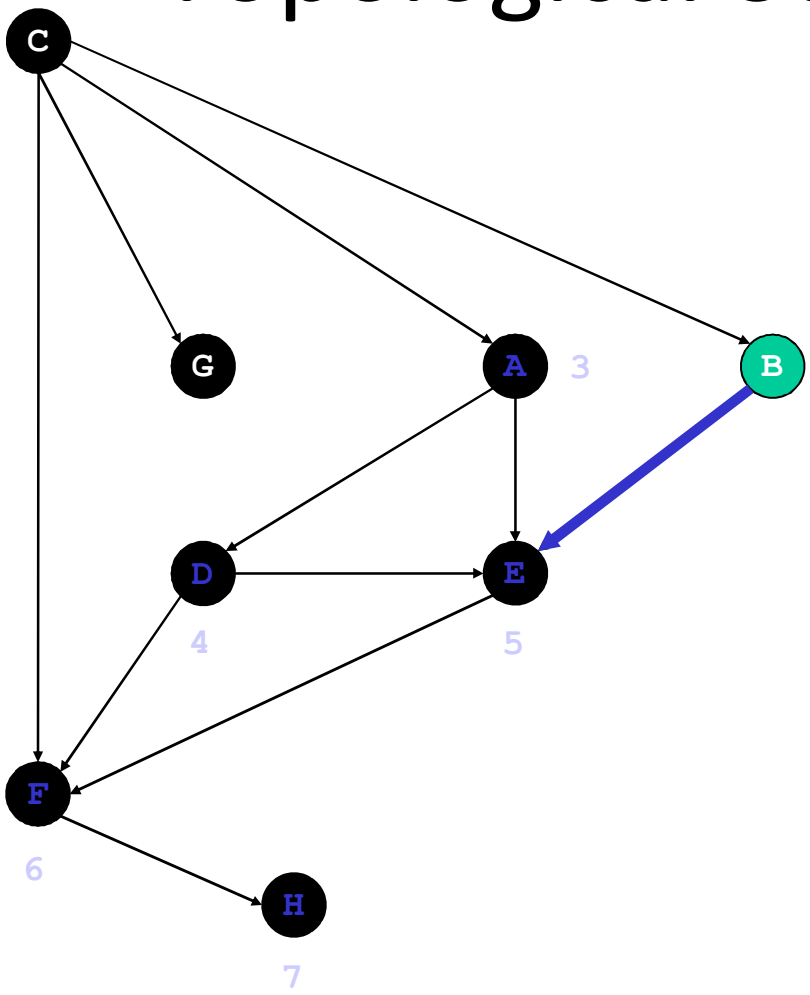



```
dfs(A)
```

Topological Sort: DFS

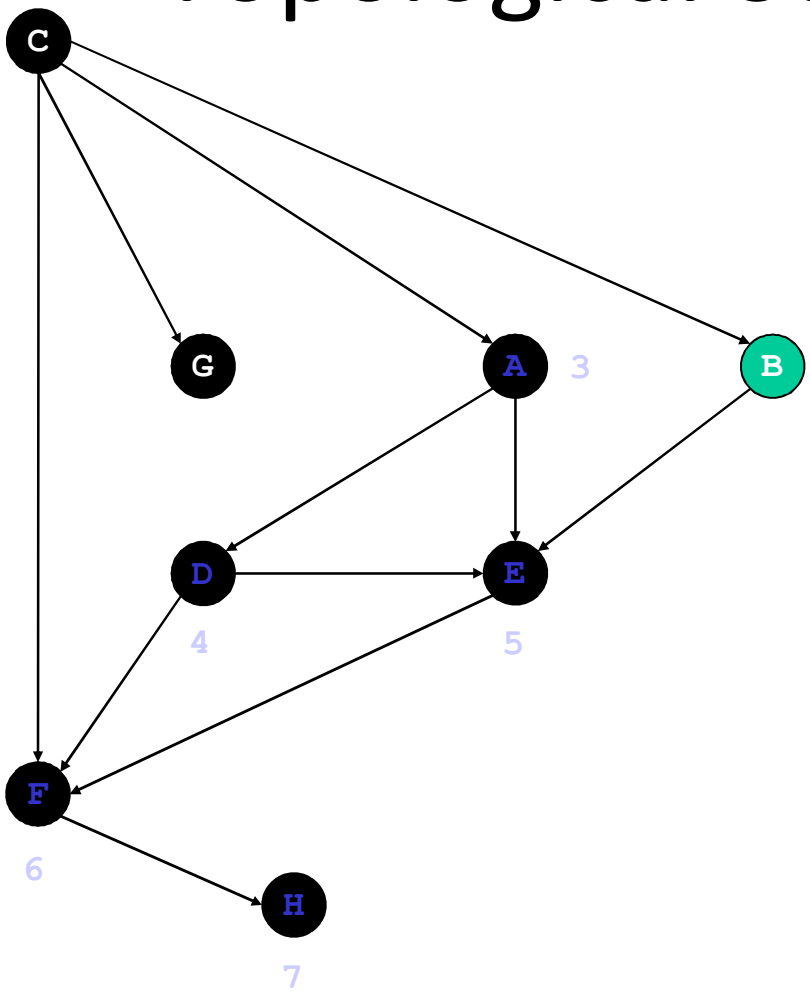


Topological Sort: DFS



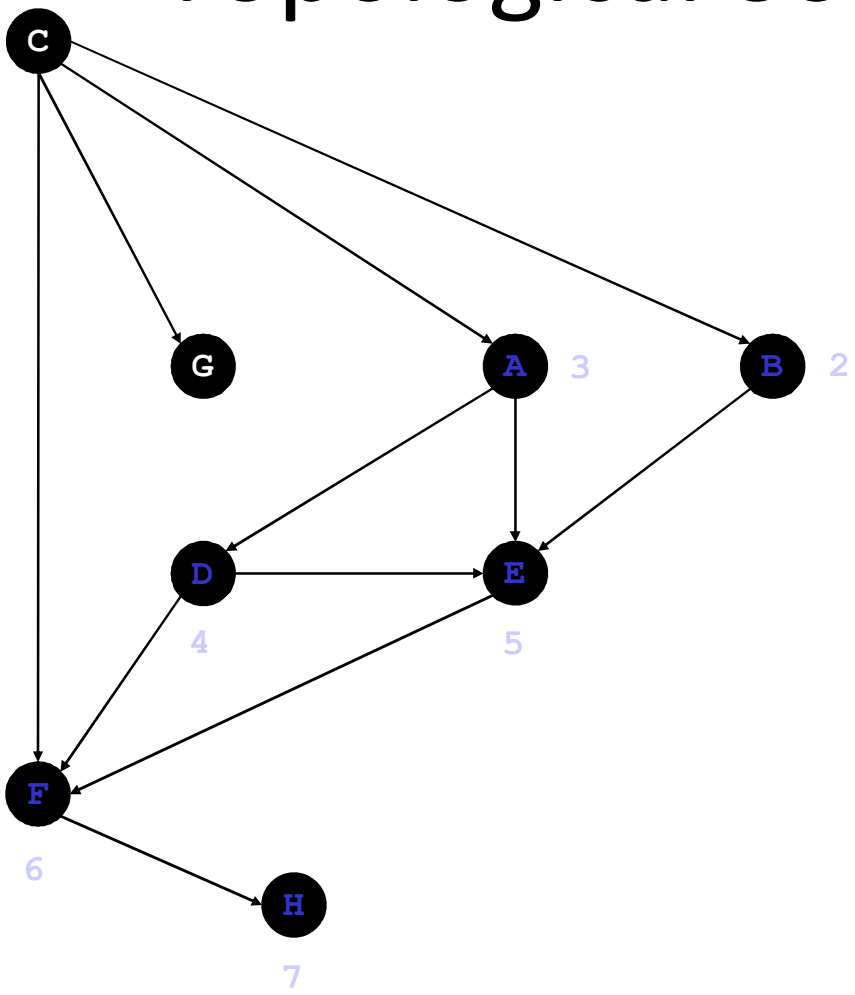

```
dfs(B)
```

Topological Sort: DFS

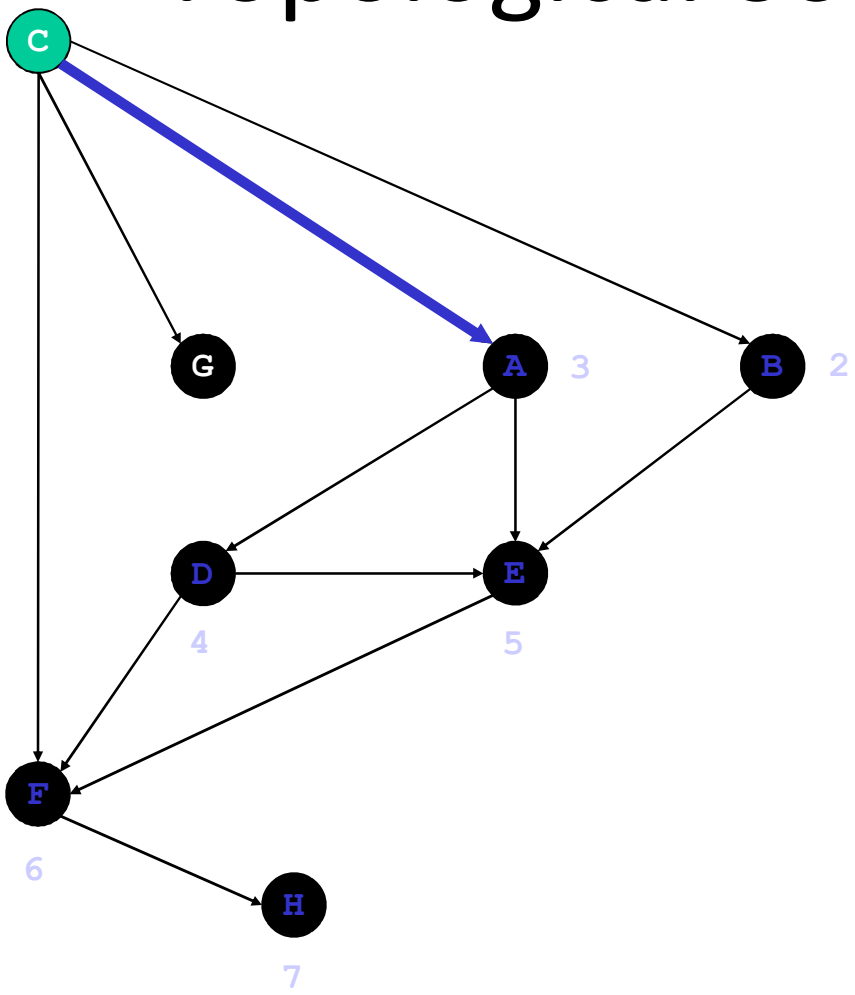



```
dfs(B)
```

Topological Sort: DFS

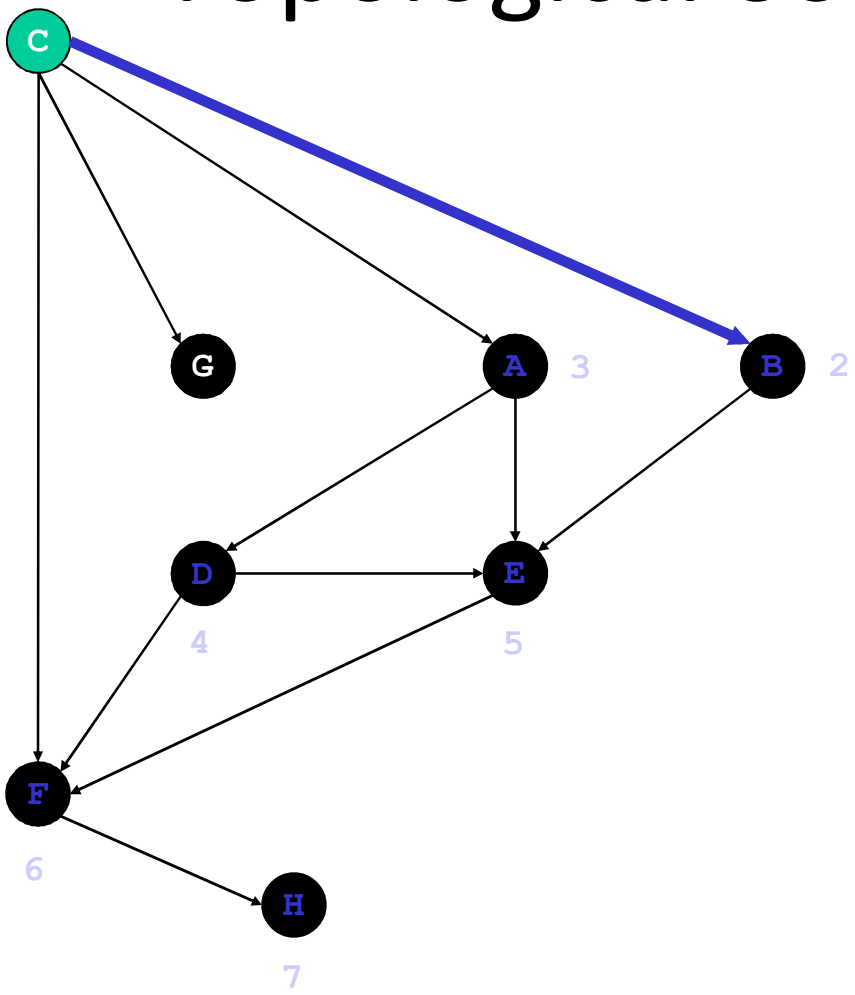


Topological Sort: DFS



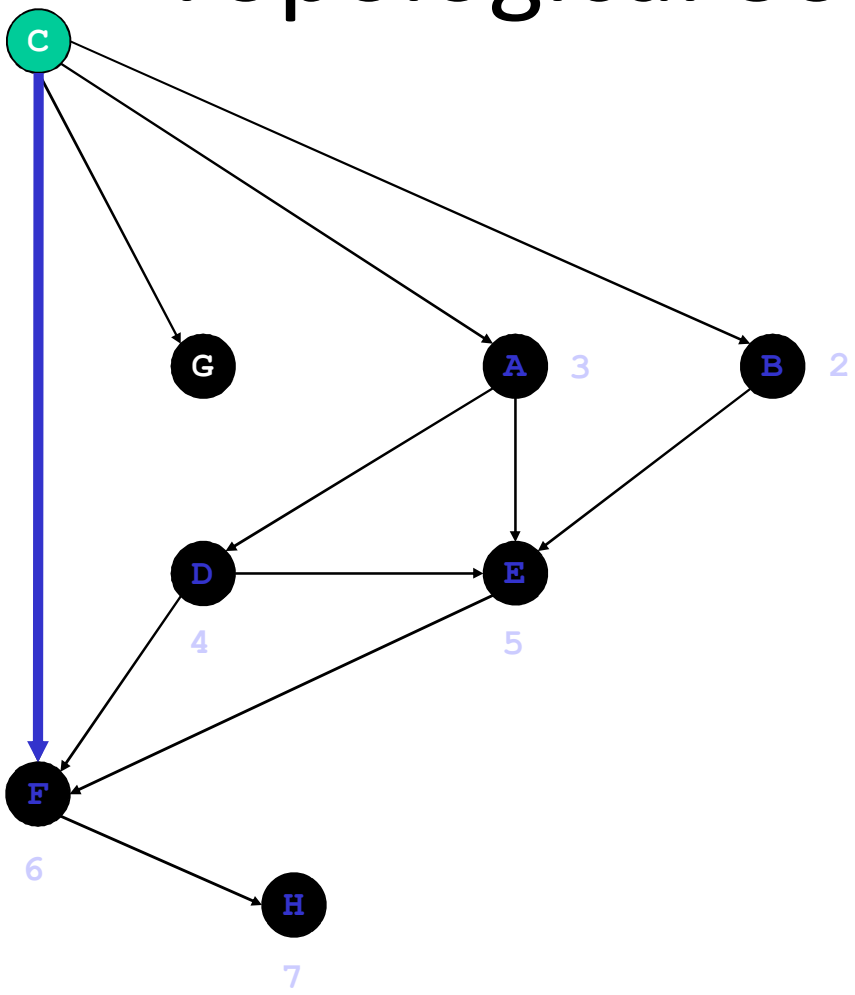

```
dfs(C)
```

Topological Sort: DFS



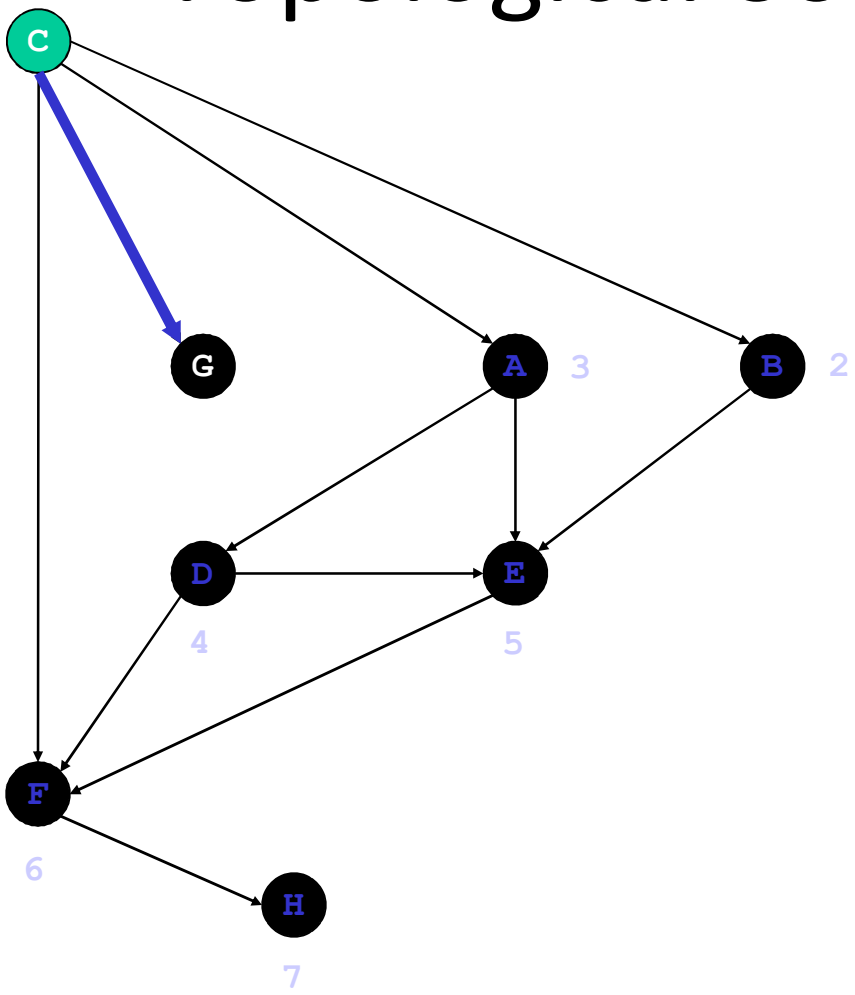

```
dfs(C)
```

Topological Sort: DFS



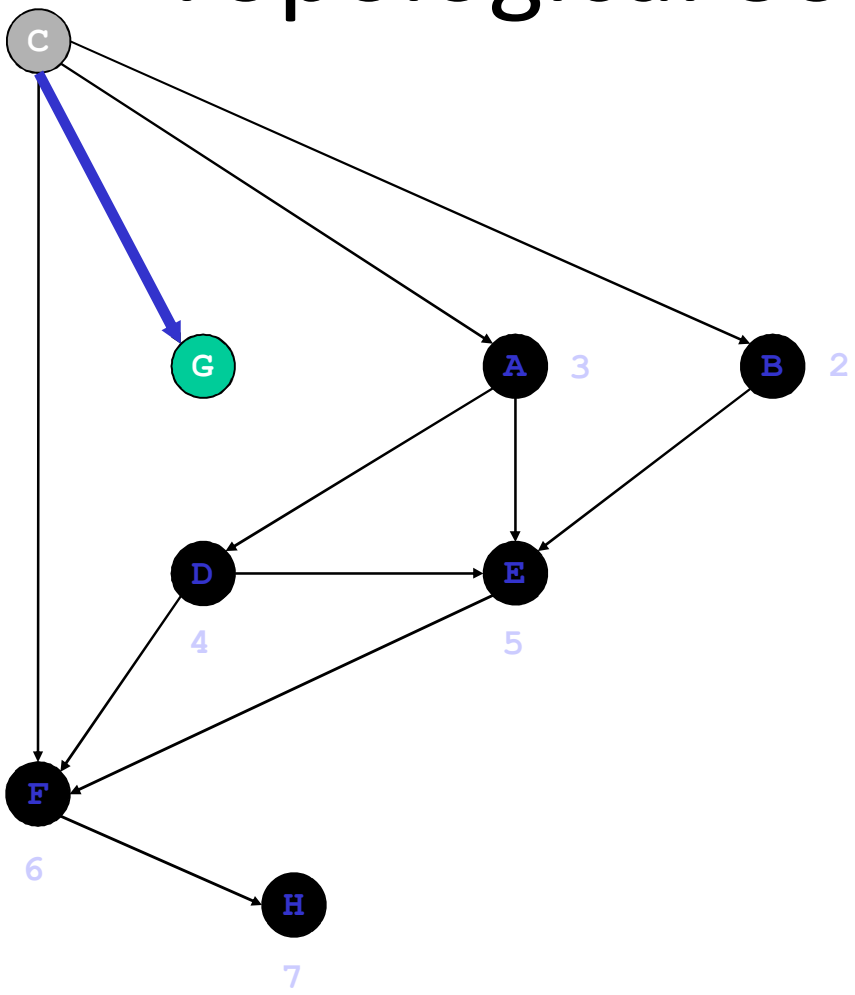

```
dfs(C)
```

Topological Sort: DFS



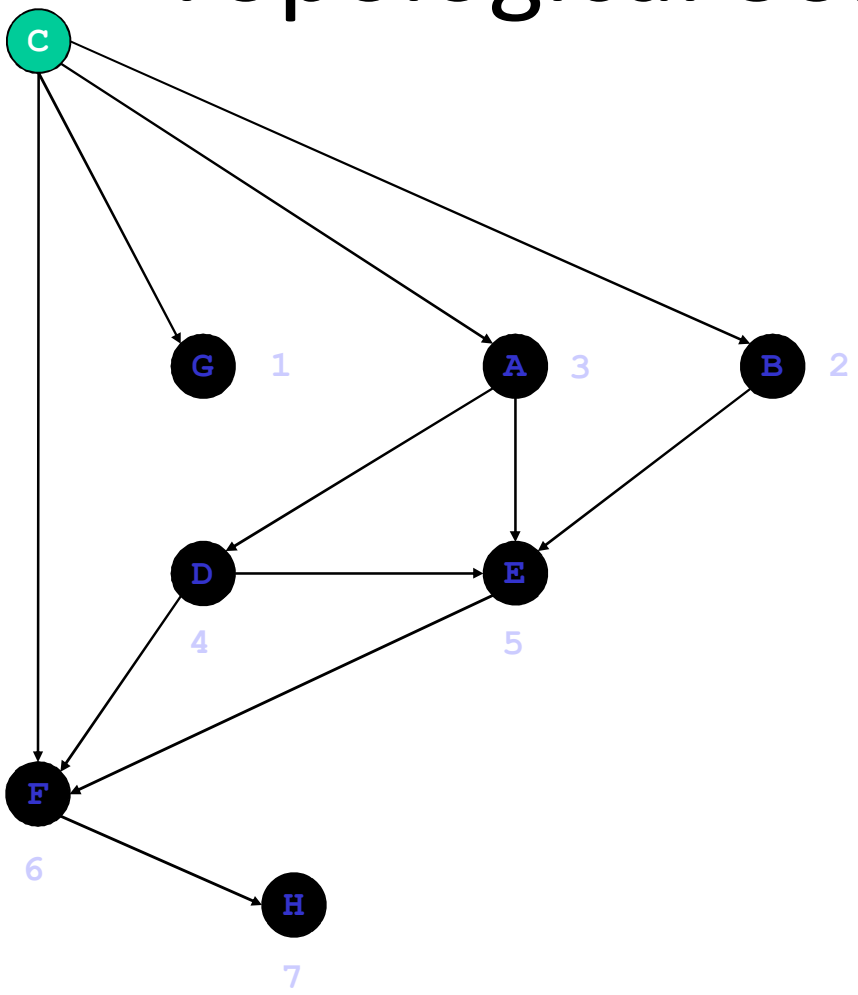

```
dfs(C)
```

Topological Sort: DFS



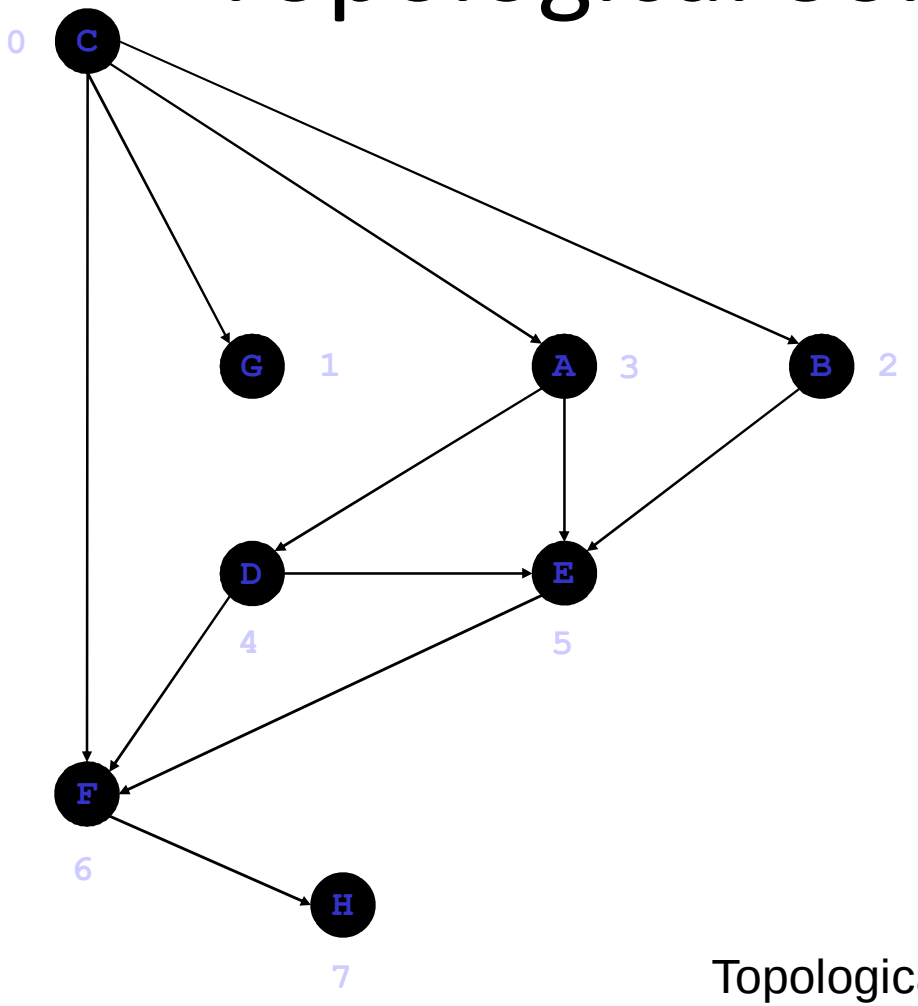

```
dfs(C)
   dfs(G)
```

Topological Sort: DFS




```
dfs(C)
```

Topological Sort: DFS



Topological Order: C G B A D E F H

Topological Sort Algorithms: Source Removal Algorithm

The Source Removal Topological sort algorithm is:

Pick a source u [vertex with in-degree zero], output it.

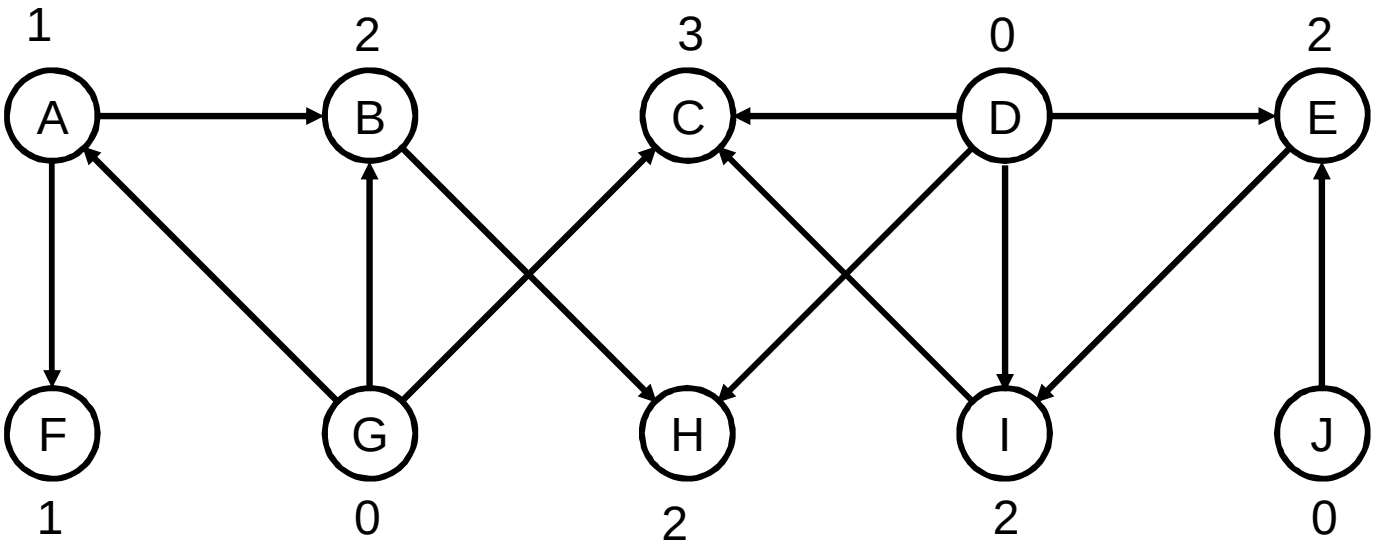
Remove u and all edges out of u.

Repeat until graph is empty.

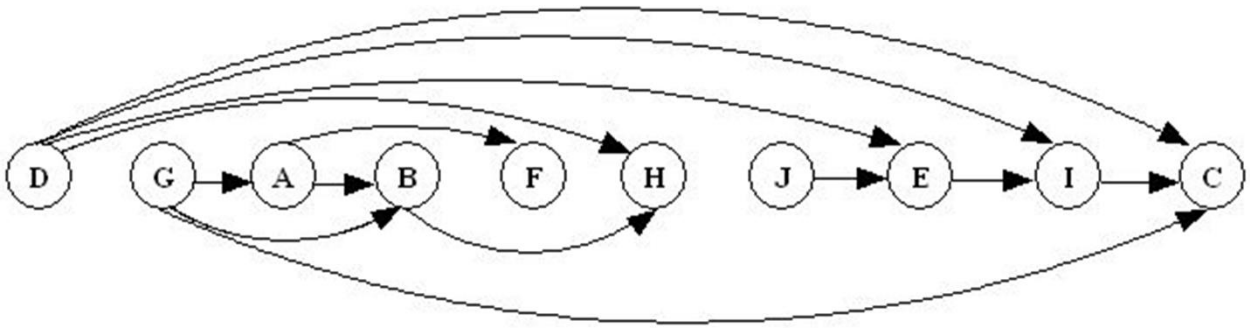
```
int topologicalOrderTraversal( ){
    int numVisitedVertices = 0;
    while(there are more vertices to be visited){
        if(there is no vertex with in-degree 0)
            break;
        else{
         select a vertex v that has in-degree 0;
         visit v;
         numVisitedVertices++;
         delete v and all its emanating edges;
        }
    }

   return numVisitedVertices;
}
```

Topological Sort: Source Removal Example

The number beside each vertex is the in-degree of the vertex at the start of the algorithm.



D	G	A	B	F	H	J	E	I	C



Proof of correctness

- **Theorem**: TOPOLOGICAL-SORT(**G**) produces a topological sort of a DAG **G.**
- The TOPOLOGICAL-SORT(**G**) algorithm does a DFS on the DAG **G**, and it lists the nodes of **G** in order of decreasing finish times **f**[].
- We must show that this list satisfies the topological sort property, namely, that for every edge (**u**,**v**) of **G**, **u** appears before **v** in the list.
- **Claim**: For every edge (**u**, **v**) of **G**: **f**[**v**] < **f**[**u**] in DFS.

Proof of correctness

"For every edge (**u**,**v**) of **G**, **f**[**v**] < **f**[**u**] in this DFS"

The DFS classifies (**u**,**v**) as a **tree edge**, a **forward edge** or a **cross-edge** (it cannot be a back-edge since **G** has no cycles):

i. If (**u**,**v**) is a **tree** or a **forward edge** $\Rightarrow$ **v** is a descendant of **u** $\Rightarrow$ **f**[**v**] < **f**[**u**].

ii. If (**u**,**v**) is a **cross-edge.**

Proof of correctness

"For every edge (**u**,**v**) of **G**: **f**[**v**] < **f**[**u**] in this DFS"

ii. If (**u**,**v**) is a **cross-edge**:

Å as (**u**,**v**) is a cross-edge, by definition, neither **u** is a descendant of **v** nor **v** is a descendant of **u:**

d[**u**] < f[**u**] < d[**v**] < f[**v**]

or

d[**v**] < f[**v**] < d[**u**] < f[**u**]

since (**u**,**v**) is an edge, **v** is surely discovered **before** **u**'s exploration completes

f[v] < f[u]

Proof of correctness

TOPOLOGICAL-SORT(G) lists the nodes of G from highest to lowest finishing times

By the **Claim**, for every edge (**u**,**v**) of **G**:

$$\mathbf{f}[\mathbf{v}] < \mathbf{f}[\mathbf{u}]$$

$\Rightarrow$ **u** will be before **v** in the algorithm's list