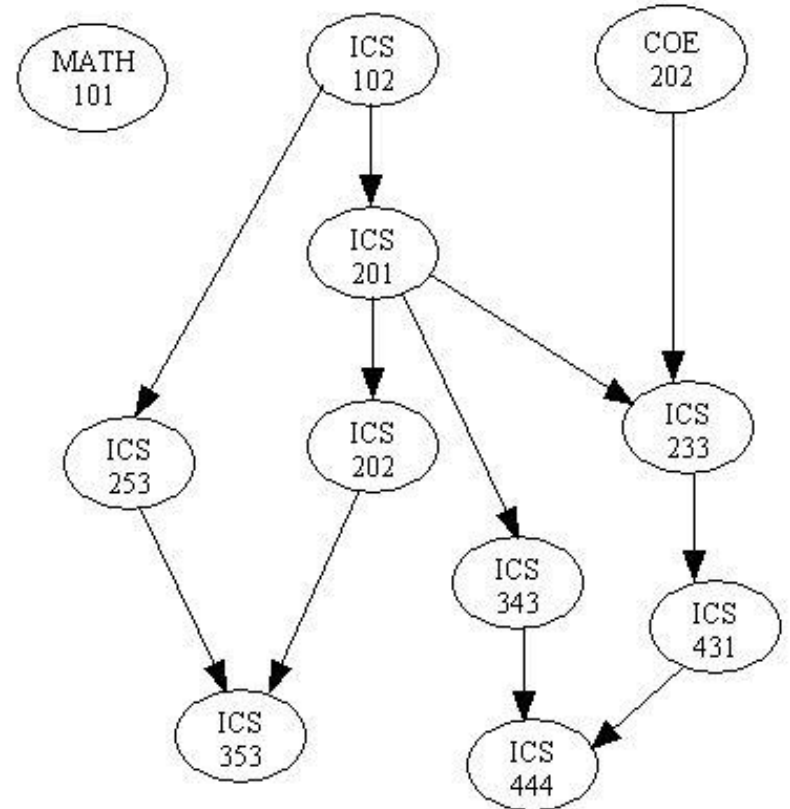


Topological sort

Motivation

Given a set of tasks with dependencies,
is there an order in which we can complete the tasks?

Cycles in dependencies can cause issues...



Definition of topological sorting

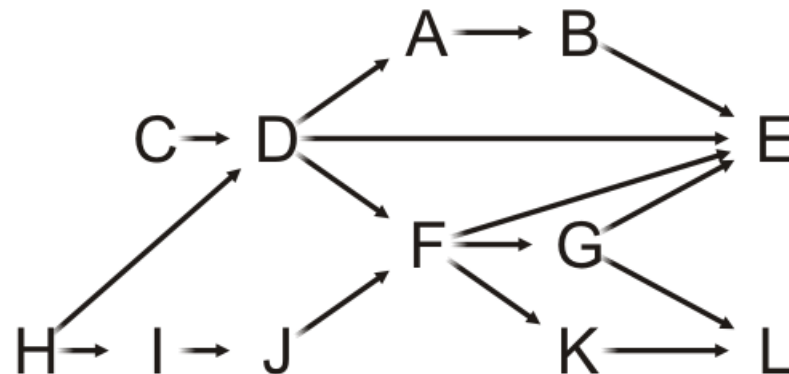
A topological sorting of the vertices in a DAG is an ordering

$$v_1, v_2, v_3, \dots, v_{|V|}$$

such that v_j appears before v_k if there is a path from v_j to v_k

Given this DAG, a topological sort is

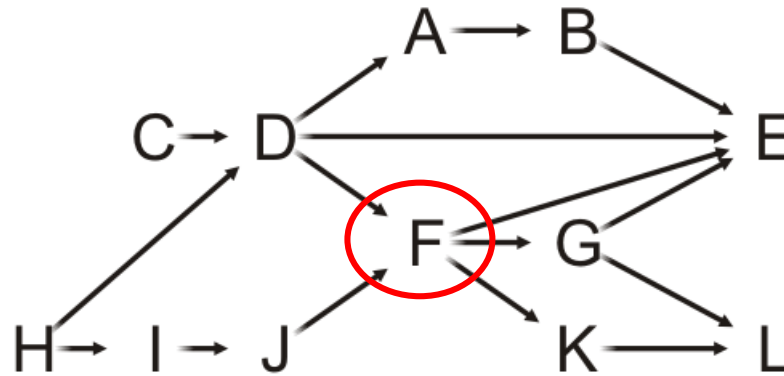
H, C, I, D, J, A, F, B, G, K, E, L



Example

For example, there are paths from H, C, I, D and J to F, so all these must come before F in a topological sort

H, C, I, D, J, A, F, B, G, K, E, L



Clearly, this sorting need not be unique

Applications

Consider someone is getting ready for a dinner out

He must wear the following:

- jacket, shirt, socks, tie, shoes etc.

There are certain constraints:

- the tie really should go on after the shirt,
- socks are put on before shoes

Applications

C++ header and source files have `#include` statements

- A change to an included file requires a recompilation of the current file
- On a large project, it is desirable to recompile only those source files that depended on those files which changed
- For large software projects, full compilations may take hours

Different scheduling programs in operating system

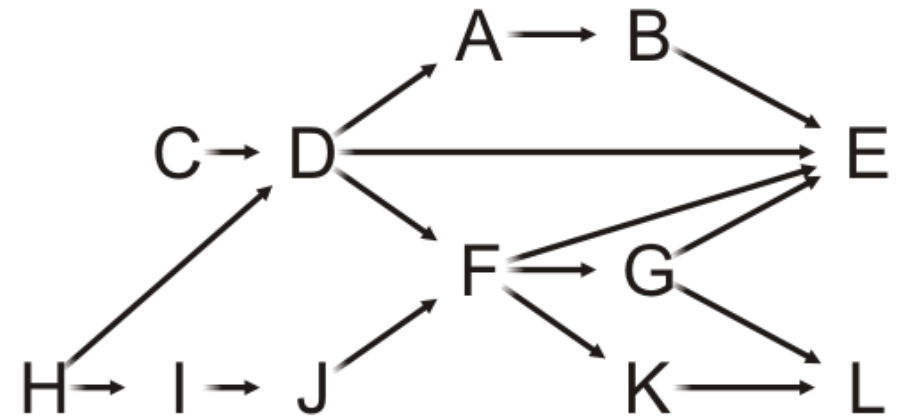
Topological Sort

Idea:

- Given a DAG V , make a copy W and iterate:
 - Find a vertex v in W with in-degree zero
 - Let v be the next vertex in the topological sort
 - Continue iterating with the vertex-induced sub-graph $W \setminus \{v\}$

On this graph, iterate the following $|V| = 12$ times

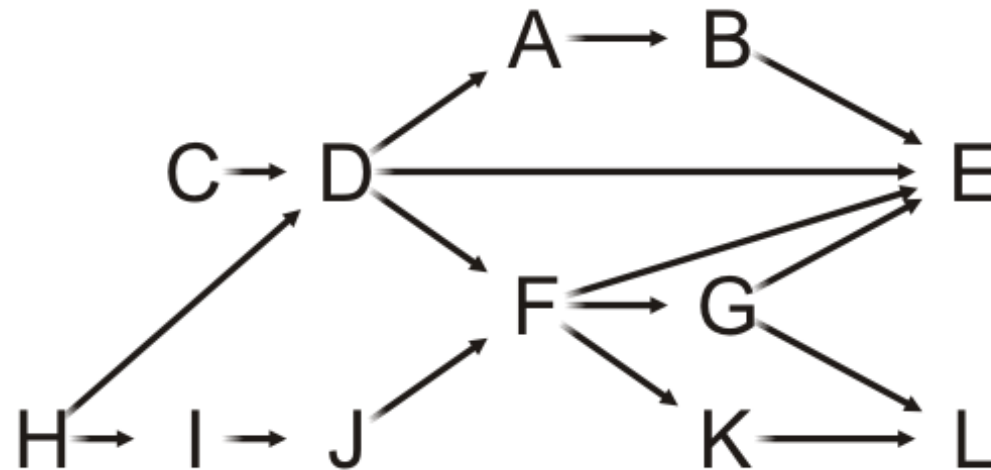
Choose a vertex v that has in-degree zero
Let v be the next vertex in our topological sort
Remove v and all edges connected to it



Example

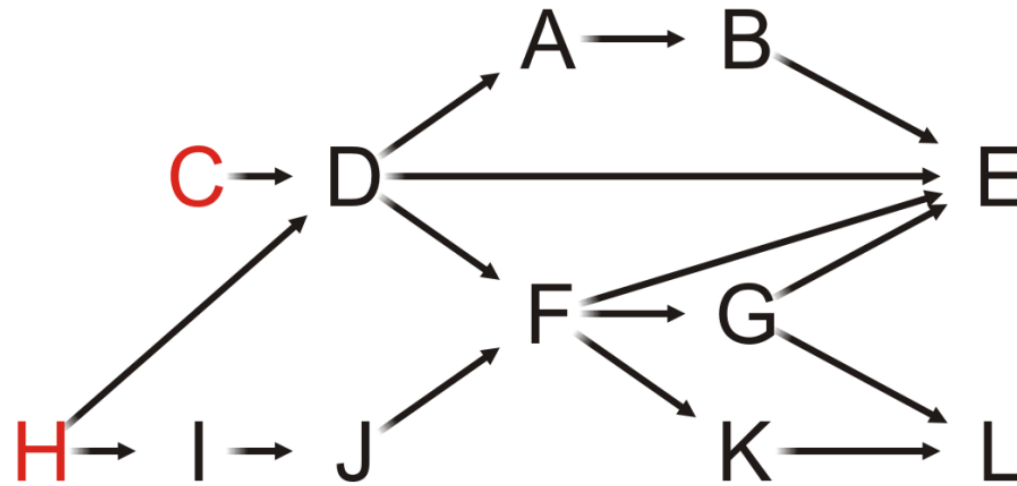
Let's step through this algorithm with this example

- Which task can we start with?



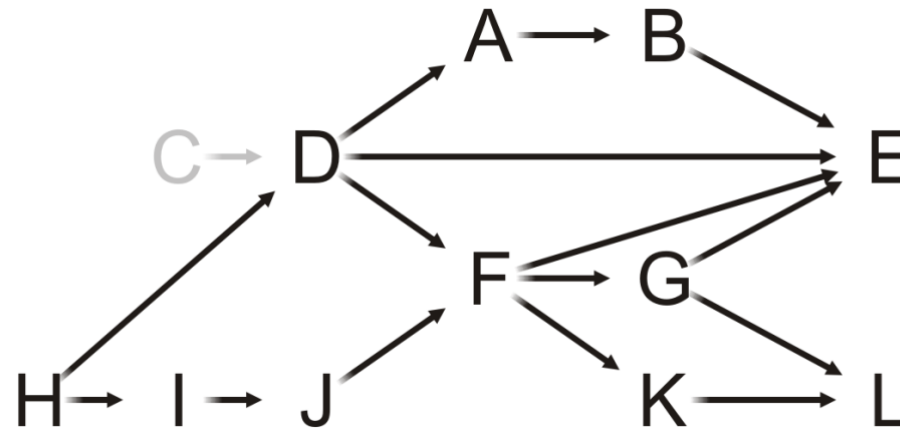
Example

Of Tasks C or H, choose Task C



Example

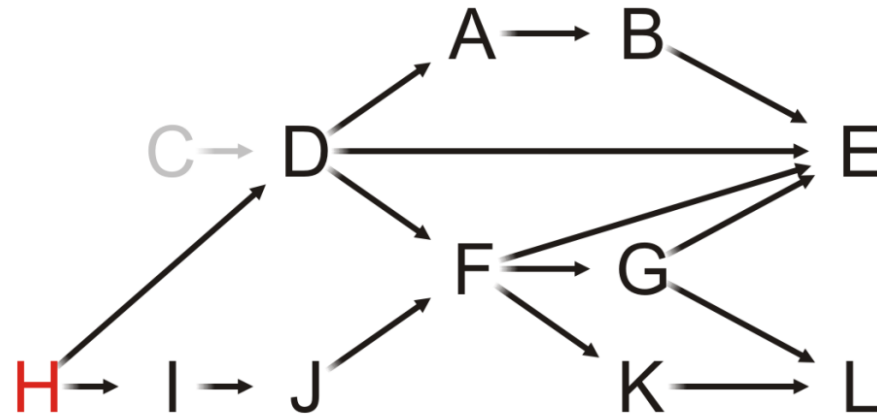
Having completed Task C, which vertices have in-degree zero?



C

Example

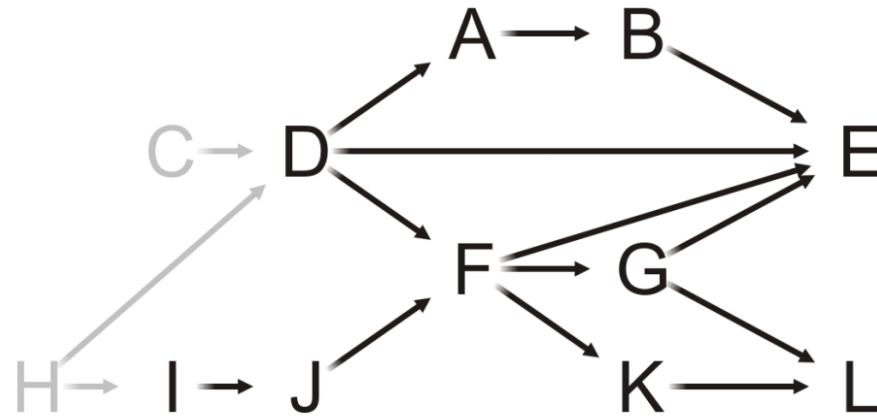
Only Task H can be completed, so we choose it



C

Example

Having removed H, what is next?

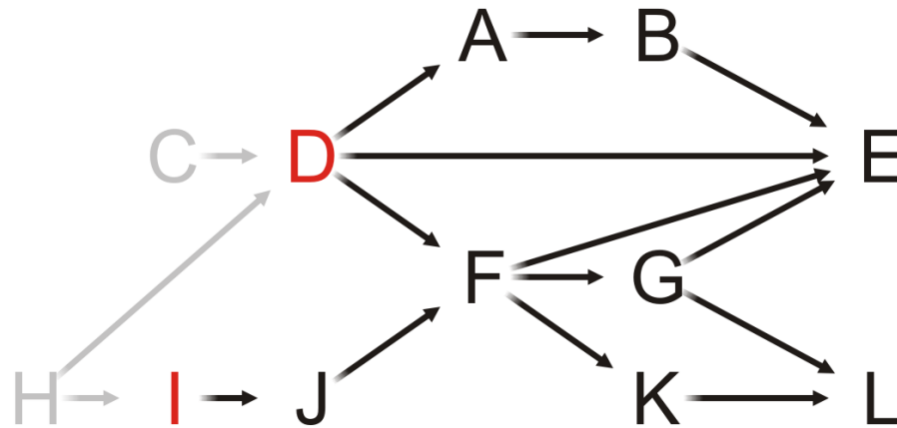


C, H

Example

Both Tasks D and I have in-degree zero

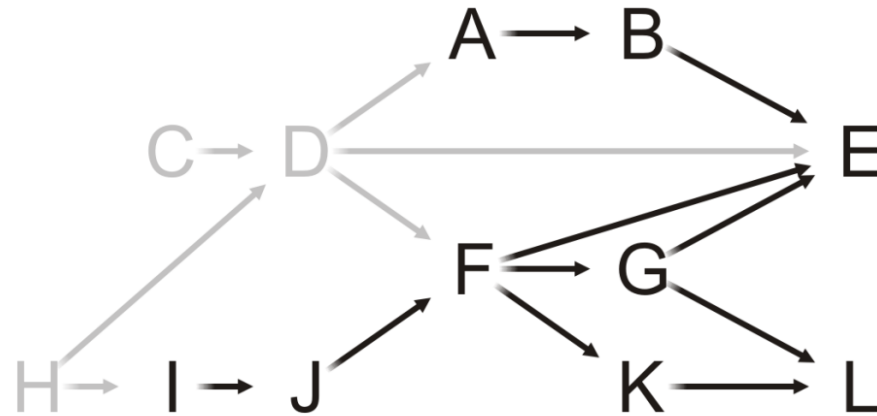
- Let us choose Task D



C, H

Example

We remove Task D, and now?

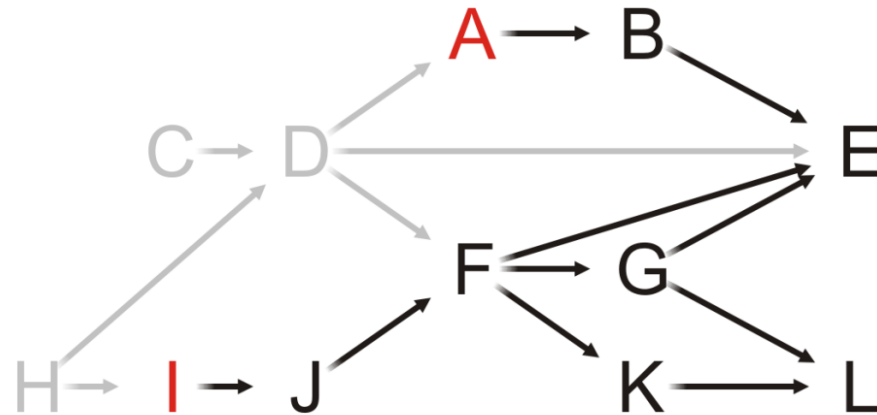


C, H, D

Example

Both Tasks A and I have in-degree zero

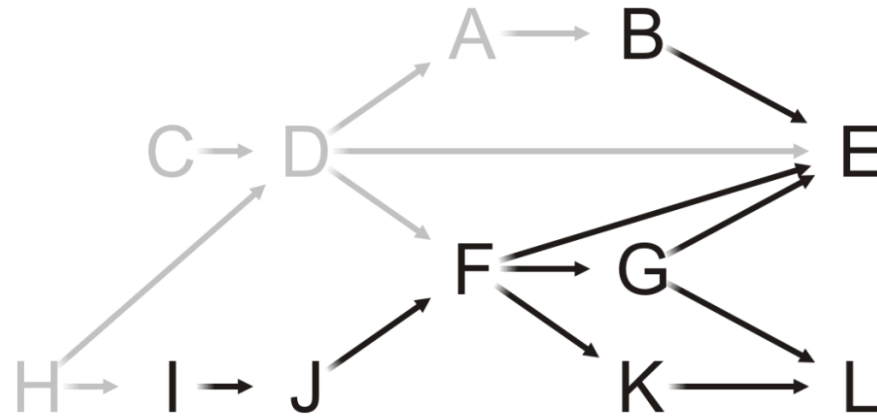
- Let's choose Task A



C, H, D

Example

Having removed A, what now?

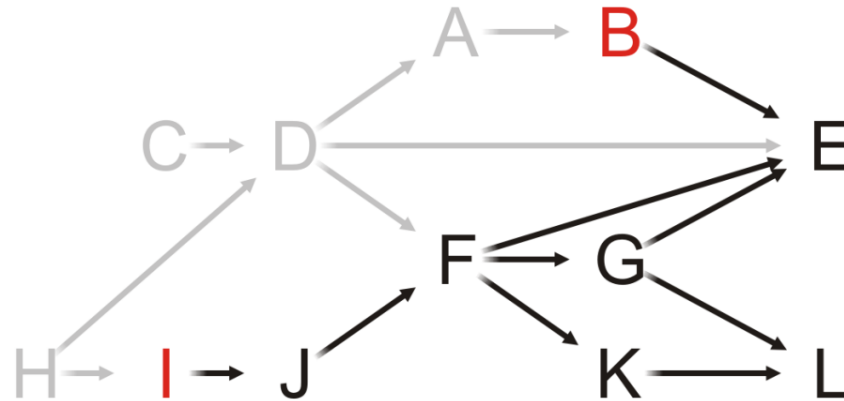


C, H, D, A

Example

Both Tasks B and I have in-degree zero

- Choose Task B

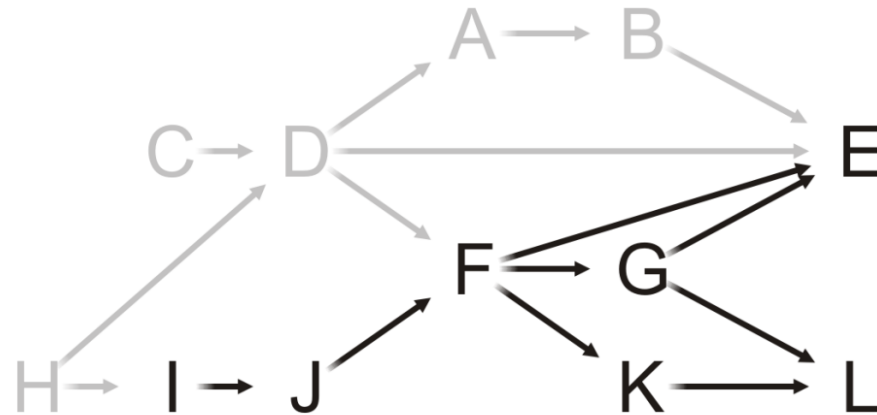


C, H, D, A

Example

Removing Task B, we note that Task E still has an in-degree of two

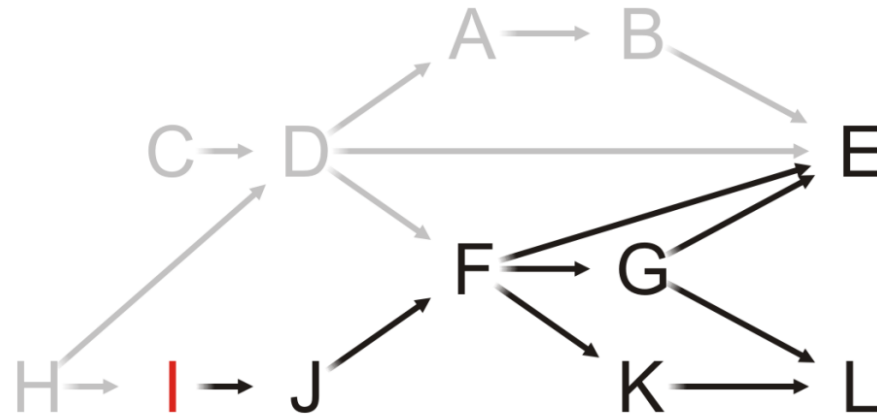
- Next?



C, H, D, A, B

Example

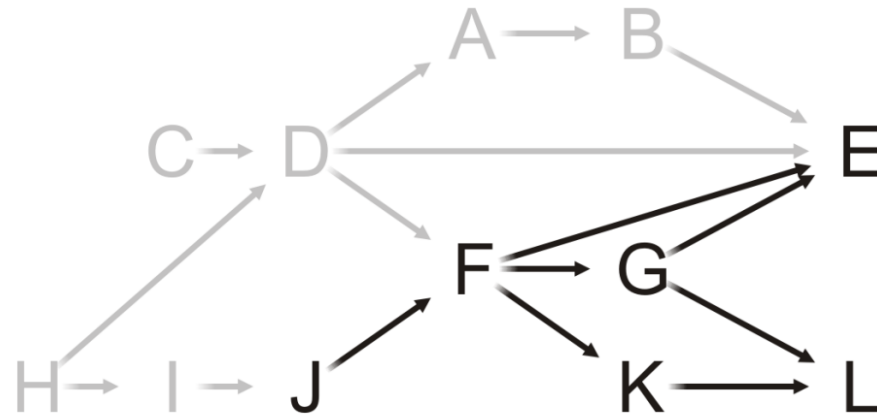
As only Task I has in-degree zero, we choose it



C, H, D, A, B

Example

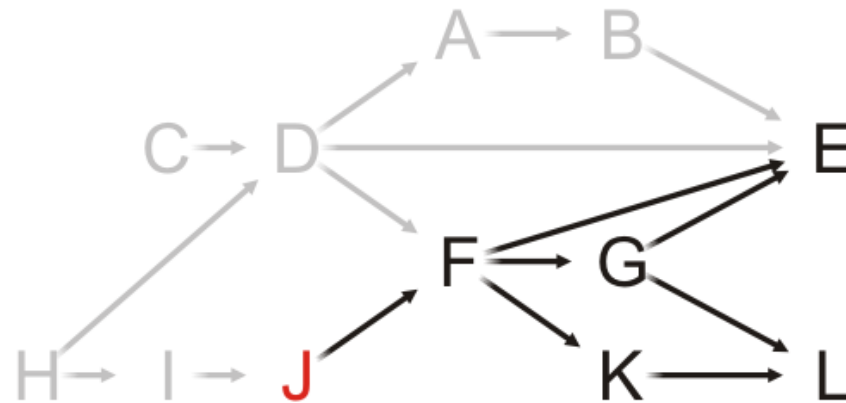
Having completed Task I, what now?



C, H, D, A, B, I

Example

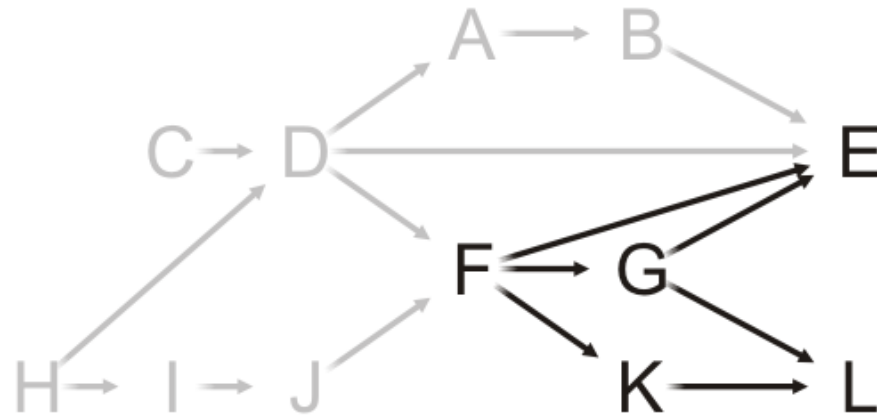
Only Task J has in-degree zero: choose it



C, H, D, A, B, I

Example

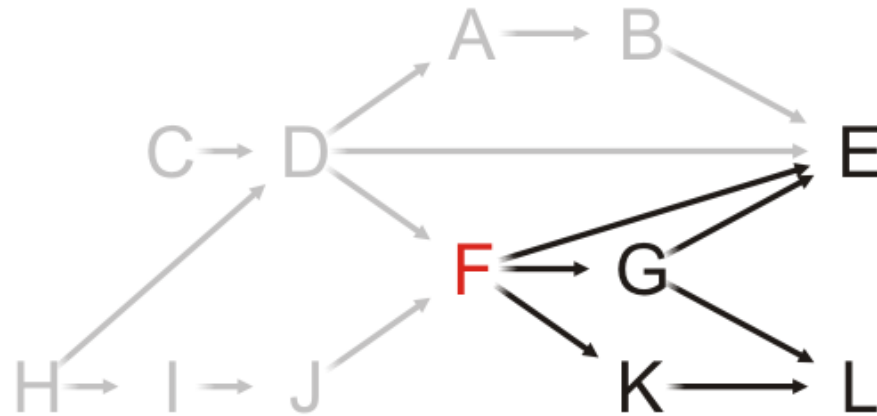
Having completed Task J, what now?



C, H, D, A, B, I, J

Example

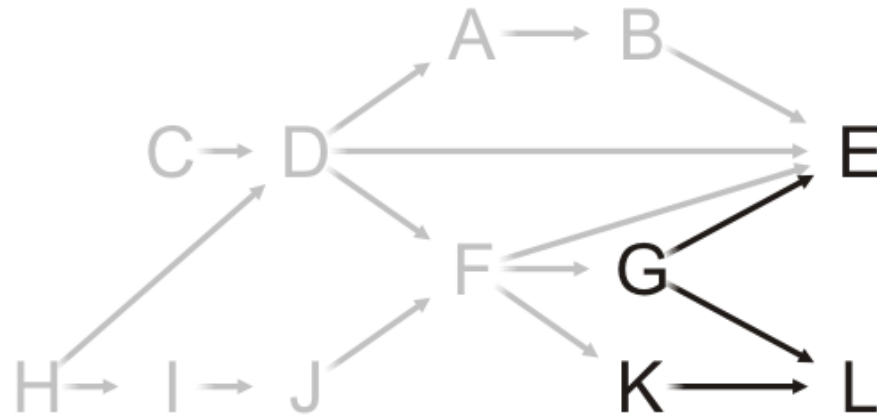
Only Task F can be completed, so choose it



C, H, D, A, B, I, J

Example

What choices do we have now?

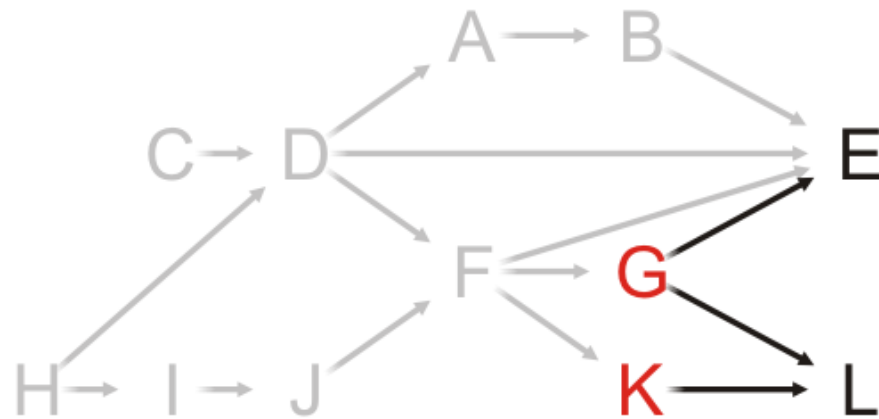


C, H, D, A, B, I, J, F

Example

We can perform Tasks G or K

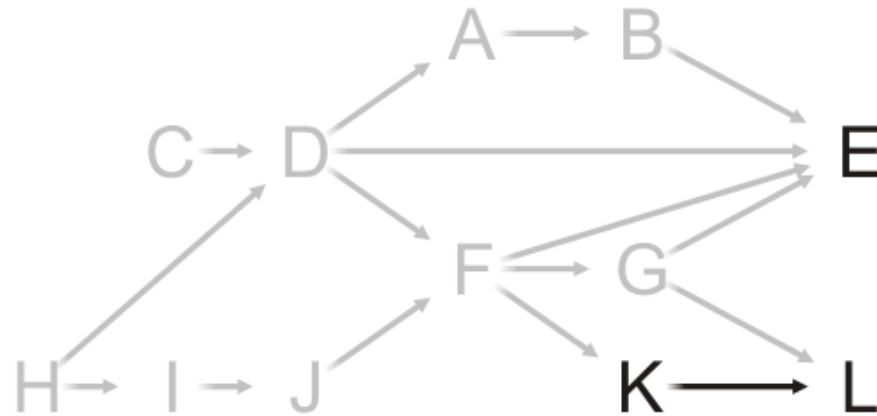
- Choose Task G



C, H, D, A, B, I, J, F

Example

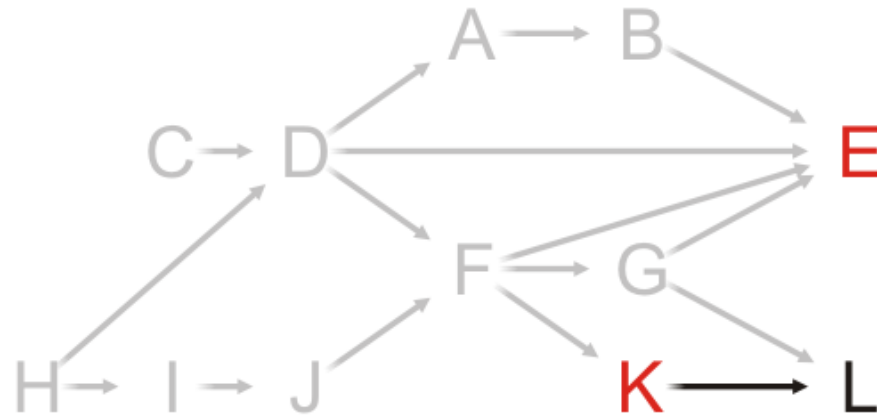
Having removed Task G from the graph, what next?



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

Example

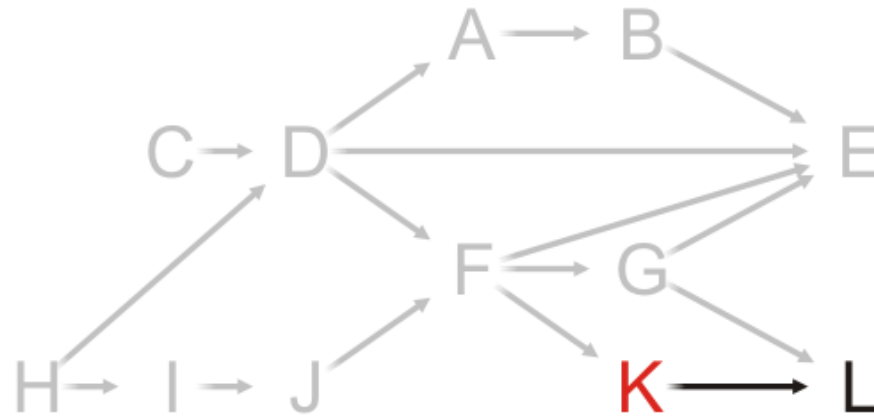
Choosing between Tasks E and K, choose Task E



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

Example

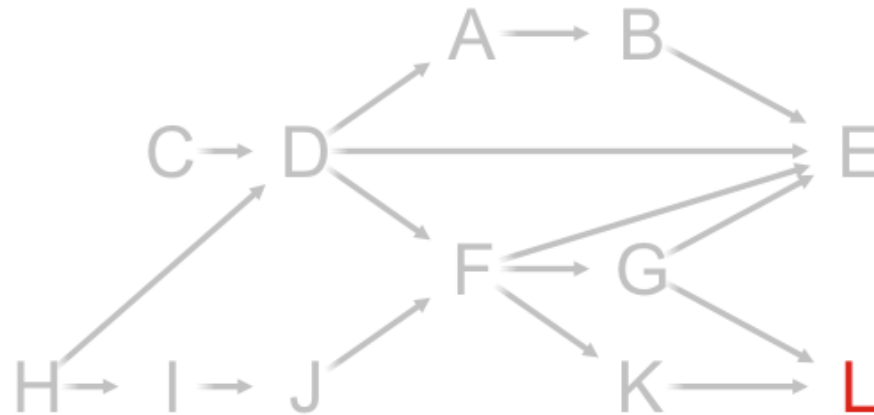
At this point, Task K is the only one that can be run



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

Example

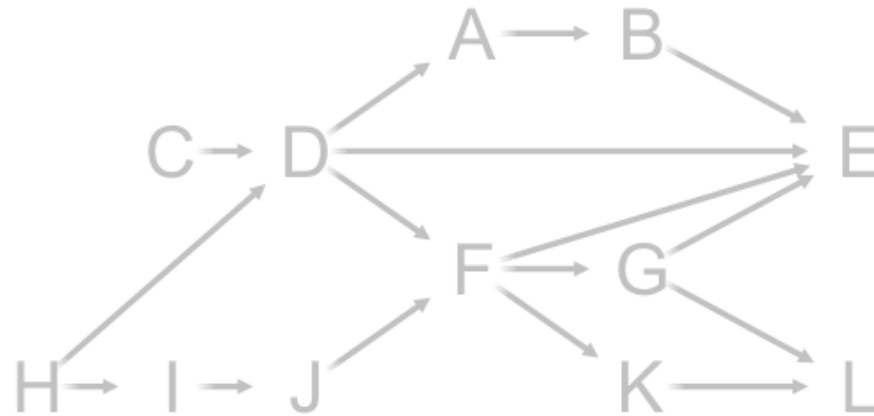
And now that both Tasks G and K are complete,
we can complete Task L



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

Example

There are no more vertices left

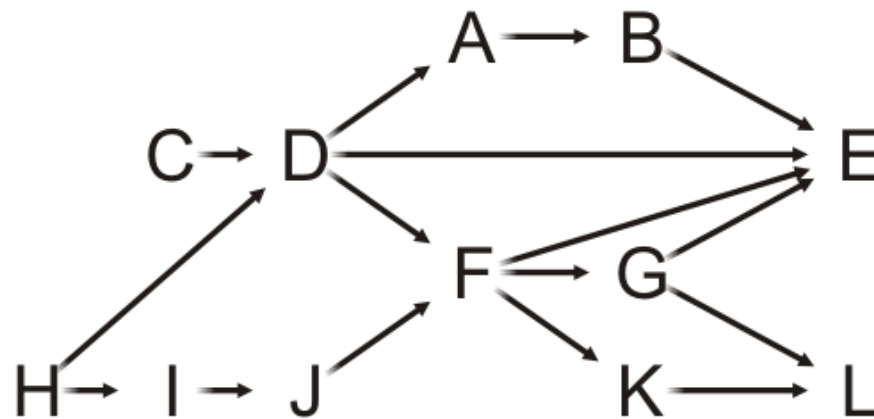


C, H, D, A, B, I, J, F, G, E, K, L

Example

Thus, one possible topological sort would be:

C, H, D, A, B, I, J, F, G, E, K, L

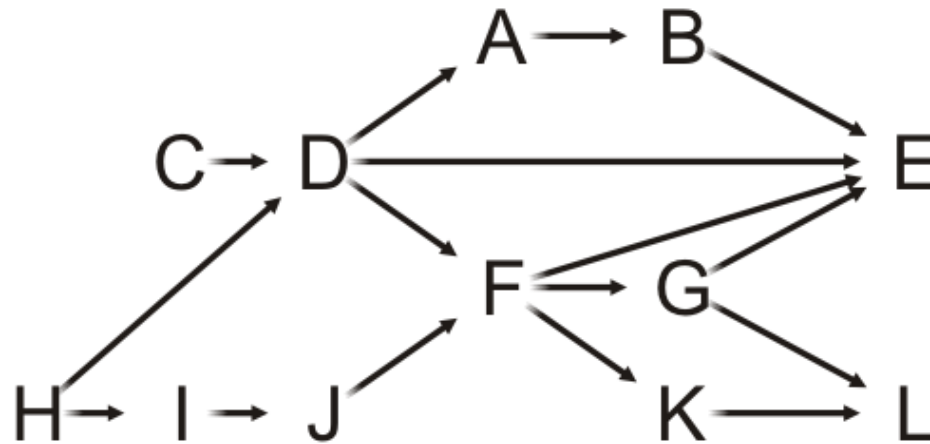


Example

Note that topological sorts need not be unique:

C, H, D, A, B, I, J, F, G, E, K, L

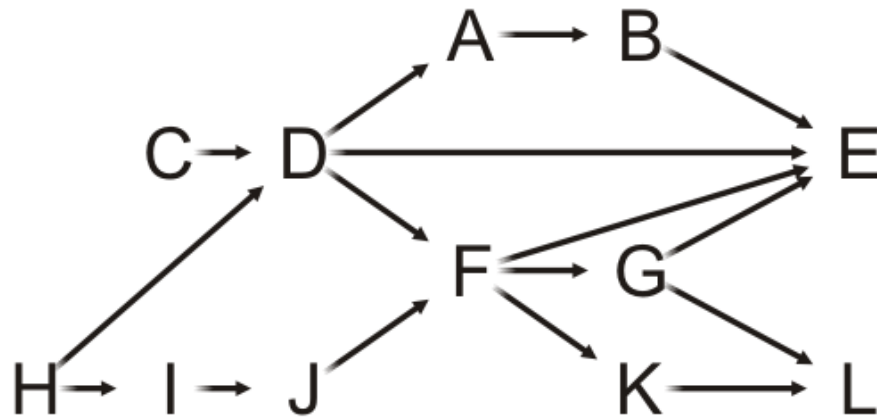
H, I, J, C, D, F, G, K, L, A, B, E



Analysis

What are the tools necessary for a topological sort?

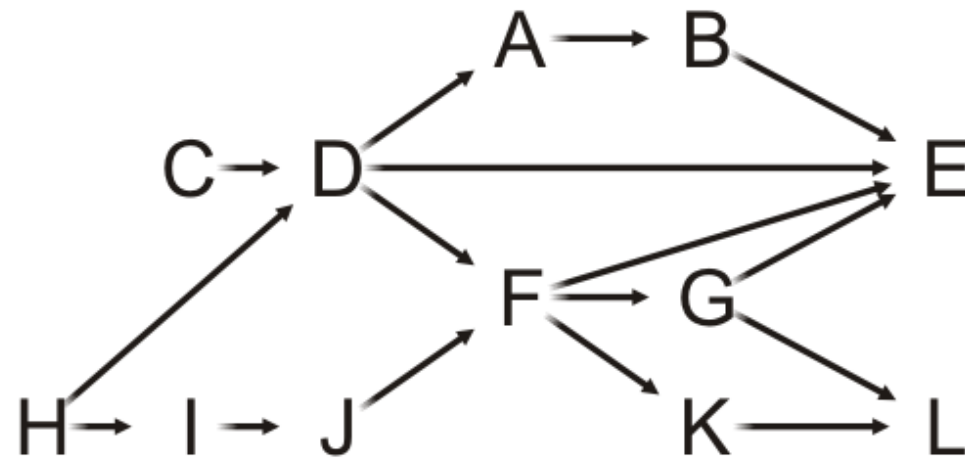
- We must know and be able to update the in-degrees of each of the vertices
- We could do this with a table of the in-degrees of each of the vertices
- This requires $\Theta(|V|)$ memory



A	1
B	1
C	0
D	2
E	4
F	2
G	1
H	0
I	1
J	1
K	1
L	2

Analysis

We must iterate at least $|V|$ times, so the run-time must be $\Omega(|V|)$

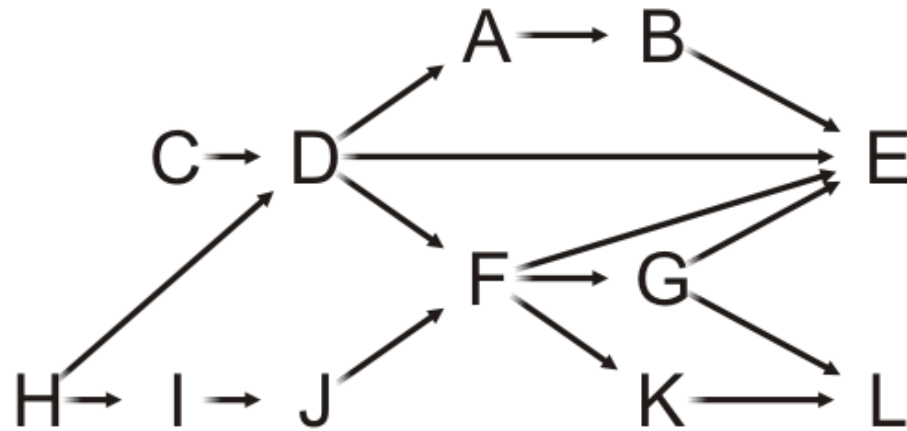


A	1
B	1
C	0
D	2
E	4
F	2
G	1
H	0
I	1
J	1
K	1
L	2

Analysis

We need to find vertices with in-degree zero

- We could loop through the array with each iteration
- The run time would be $O(|V|^2)$

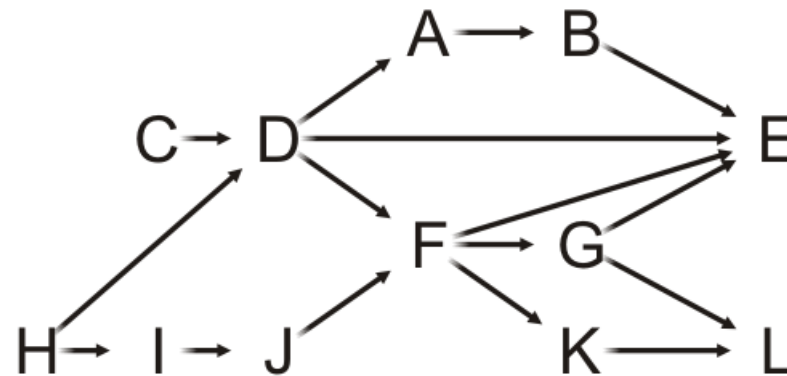


A	1
B	1
C	0
D	2
E	4
F	2
G	1
H	0
I	1
J	1
K	1
L	2

Analysis

What did we do with tree traversals?

- Use a queue (or other container) to temporarily store those vertices with in-degree zero
- Each time the in-degree of a vertex is decremented to zero, push it onto the queue

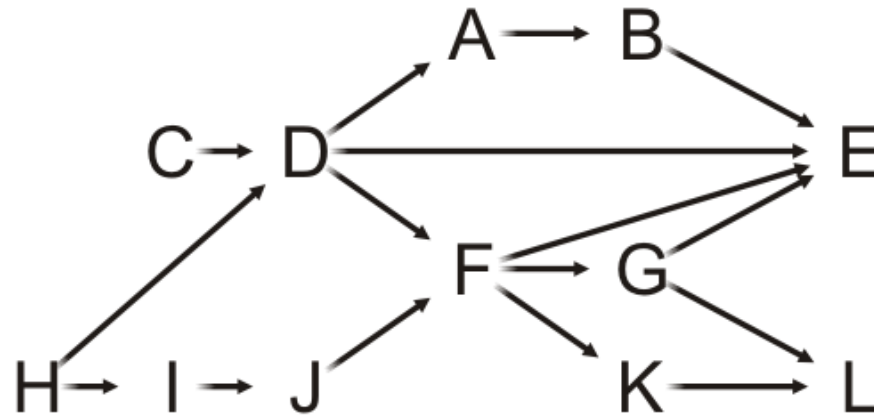


A	1
B	1
C	0
D	2
E	4
F	2
G	1
H	0
I	1
J	1
K	1
L	2

Analysis

What are the run times associated with the queue?

- Initially, we must scan through each of the vertices: $\Theta(|V|)$
- For each vertex, we will have to push onto and pop off the queue once, also $\Theta(|V|)$

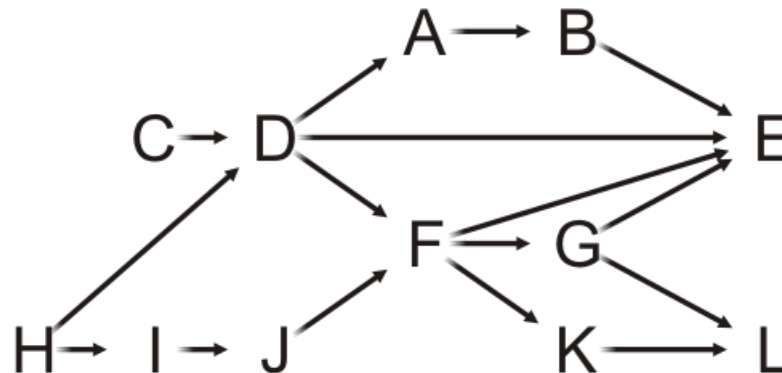


A	1
B	1
C	0
D	2
E	4
F	2
G	1
H	0
I	1
J	1
K	1
L	2

Analysis

Finally, each value in the in-degree table is associated with an edge

- Here, $|E| = 16$
- Each of the in-degrees must be decremented to zero
- The run time of these operations is $\Omega(|E|)$
- If we are using an adjacency matrix: $\Theta(|V|^2)$
- If we are using an adjacency list: $\Theta(|E|)$



A	1
B	1
C	0
D	2
E	4
F	2
G	1
H	0
I	1
J	1
K	1
L	2

+

16

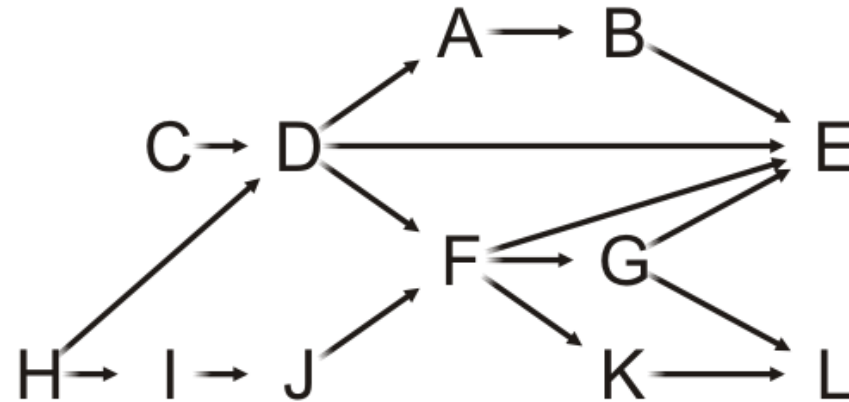
Analysis

Therefore, the run time of a topological sort is:

$\Theta(|V| + |E|)$ if we use an adjacency list

$\Theta(|V|^2)$ if we use an adjacency matrix

and the memory requirements is $\Theta(|V|)$

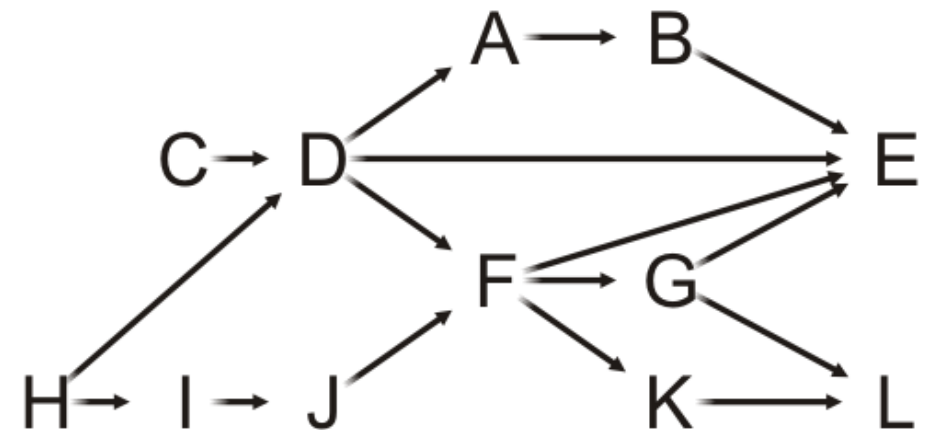


A	1
B	1
C	0
D	2
E	4
F	2
G	1
H	0
I	1
J	1
K	1
L	2

Analysis

What happens if at some step, all remaining vertices have an in-degree greater than zero?

Consequence: we now have an $\Theta(|V| + |E|)$ algorithm for determining if a graph has a cycle



Implementation

Thus, to implement a topological sort:

- Allocate memory for and initialize an array of in-degrees
- Create a queue and initialize it with all vertices that have in-degree zero

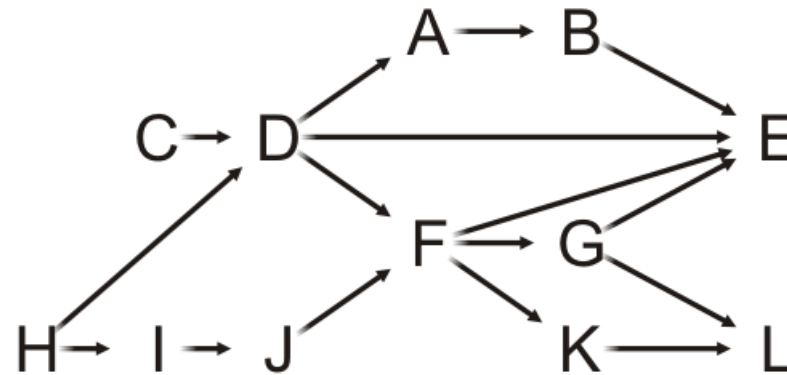
While the queue is not empty:

- Pop a vertex from the queue
- Decrement the in-degree of each neighbor
- Those neighbors whose in-degree was decremented to zero are pushed onto the queue

Example

With the previous example, we initialize:

- The array of in-degrees
- The queue



A	1
B	1
C	0
D	2
E	4
F	2
G	1
H	0
I	1
J	1
K	1
L	2

Queue:

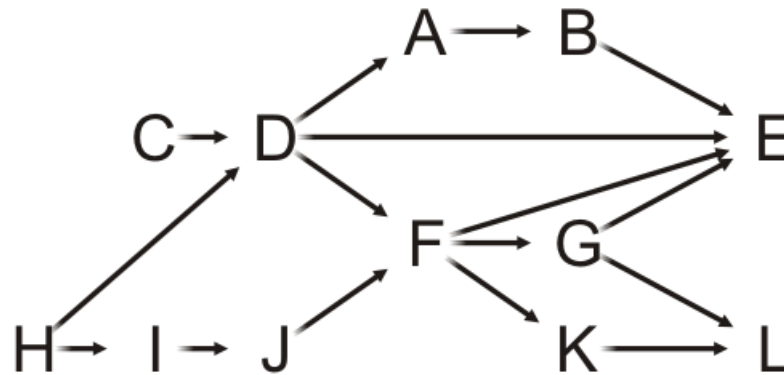
--	--	--	--	--	--	--	--	--	--	--	--



The queue is empty

Example

Stepping through the table, push all source vertices into the queue



A	1
B	1
C	0
D	2
E	4
F	2
G	1
H	0
I	1
J	1
K	1
L	2

Queue:

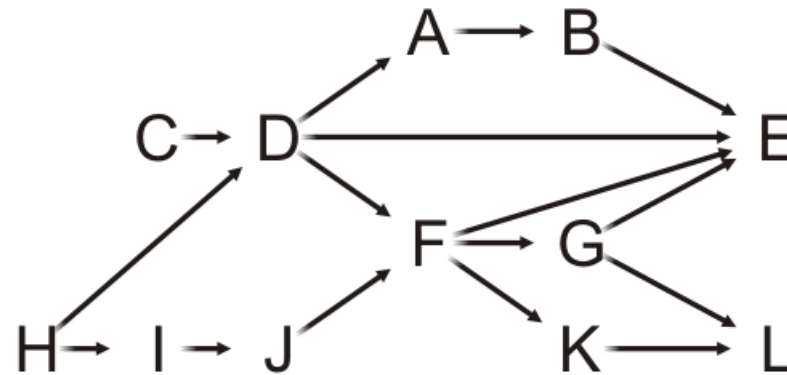
--	--	--	--	--	--	--	--	--	--	--	--



The queue is empty

Example

Stepping through the table, push all source vertices into the queue



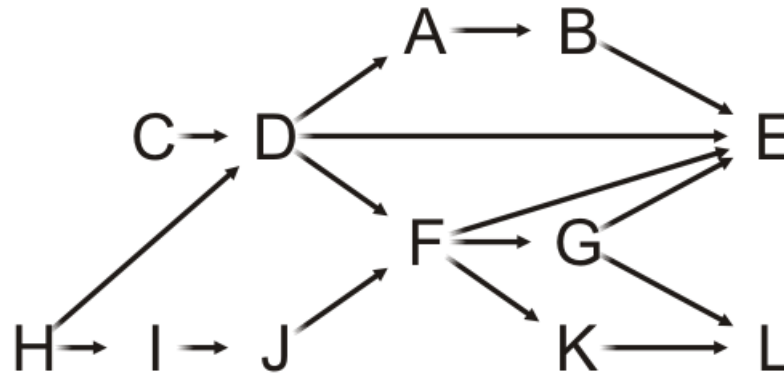
A	1
B	1
C	0
D	2
E	4
F	2
G	1
H	0
I	1
J	1
K	1
L	2



The queue is empty

Example

Pop the front of the queue

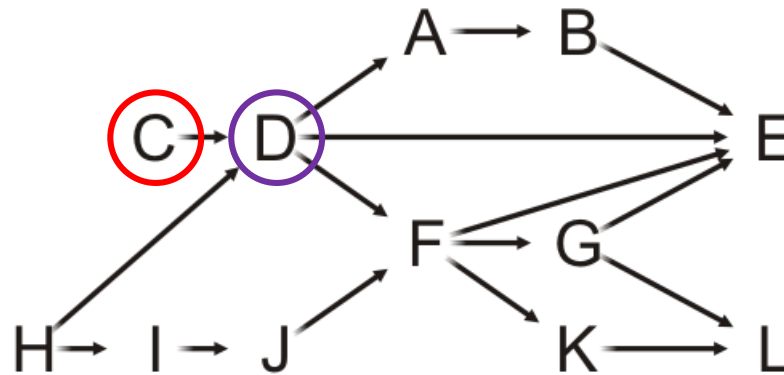


A	1
B	1
C	0
D	2
E	4
F	2
G	1
H	0
I	1
J	1
K	1
L	2

Example

Pop the front of the queue

- C has one neighbor: D



Queue:

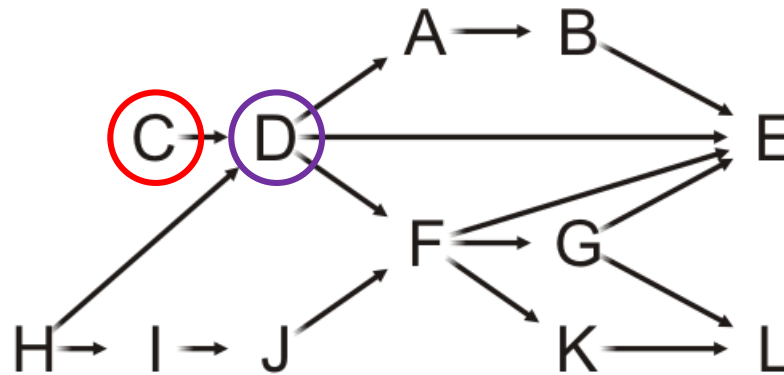


A	1
B	1
C	0
D	2
E	4
F	2
G	1
H	0
I	1
J	1
K	1
L	2

Example

Pop the front of the queue

- C has one neighbor: D
- Decrement its in-degree



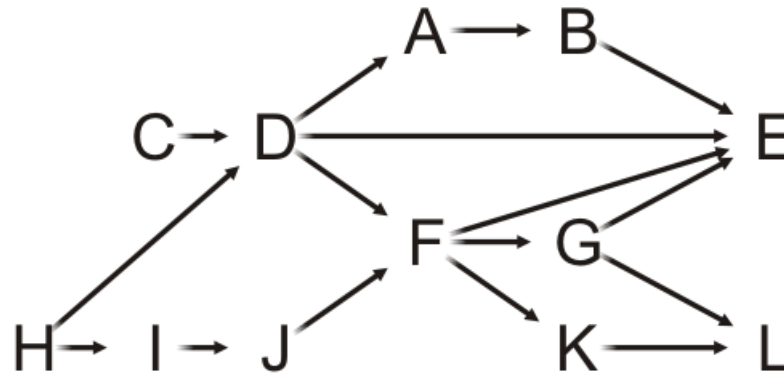
Queue:



A	1
B	1
C	0
D	1
E	4
F	2
G	1
H	0
I	1
J	1
K	1
L	2

Example

Pop the front of the queue

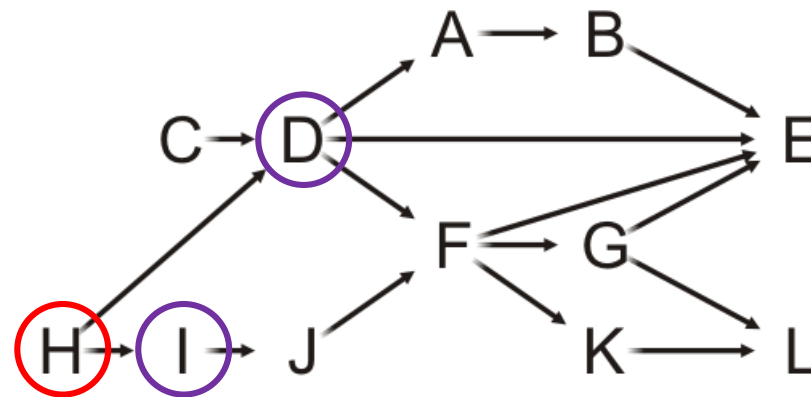


A	1
B	1
C	0
D	1
E	4
F	2
G	1
H	0
I	1
J	1
K	1
L	2

Example

Pop the front of the queue

- H has two neighbors: D and I



Queue:

C	H										
---	---	--	--	--	--	--	--	--	--	--	--

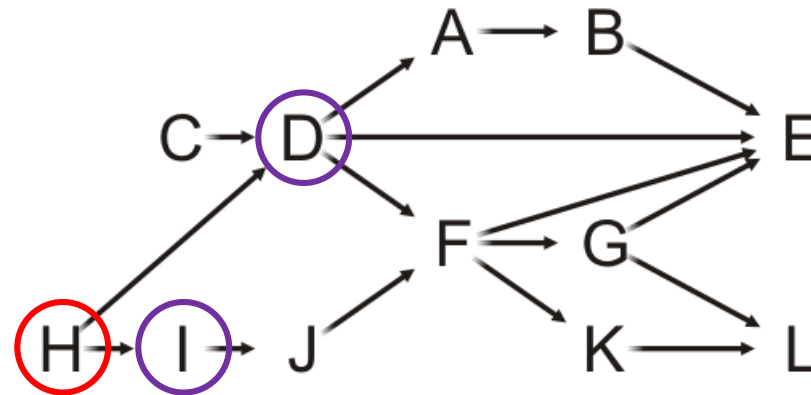


A	1
B	1
C	0
D	1
E	4
F	2
G	1
H	0
I	1
J	1
K	1
L	2

Example

Pop the front of the queue

- H has two neighbors: D and I
- Decrement their in-degrees



Queue:

C	H										
---	---	--	--	--	--	--	--	--	--	--	--

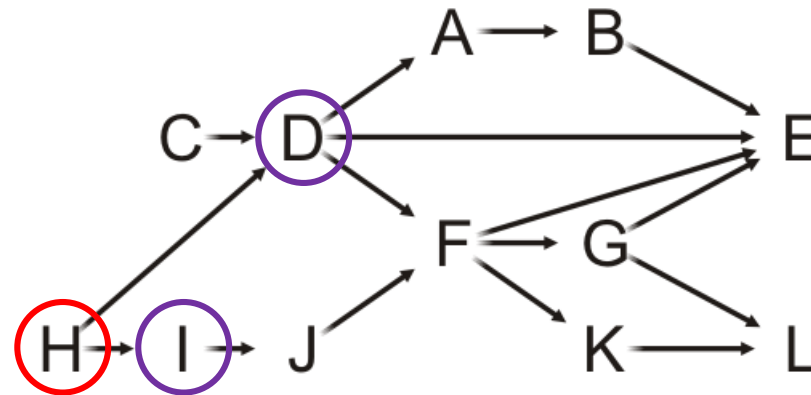


A	1
B	1
C	0
D	0
E	4
F	2
G	1
H	0
I	0
J	1
K	1
L	2

Example

Pop the front of the queue

- H has two neighbors: D and I
- Decrement their in-degrees
 - Both are decremented to zero, so push them onto the queue

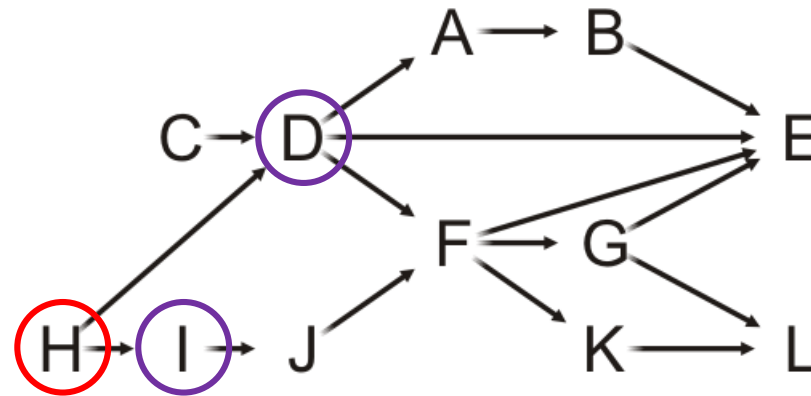


A	1
B	1
C	0
D	0
E	4
F	2
G	1
H	0
I	0
J	1
K	1
L	2

Example

Pop the front of the queue

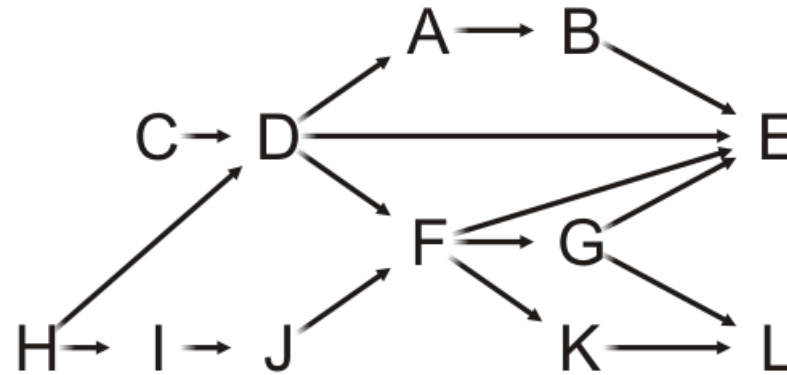
- H has two neighbors: D and I
- Decrement their in-degrees
 - Both are decremented to zero, so push them onto the queue



A	1
B	1
C	0
D	0
E	4
F	2
G	1
H	0
I	0
J	1
K	1
L	2

Example

Pop the front of the queue

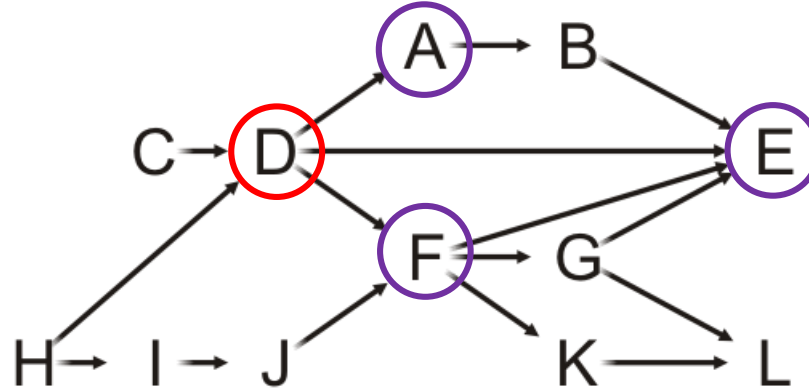


A	1
B	1
C	0
D	0
E	4
F	2
G	1
H	0
I	0
J	1
K	1
L	2

Example

Pop the front of the queue

- D has three neighbors: A, E and F

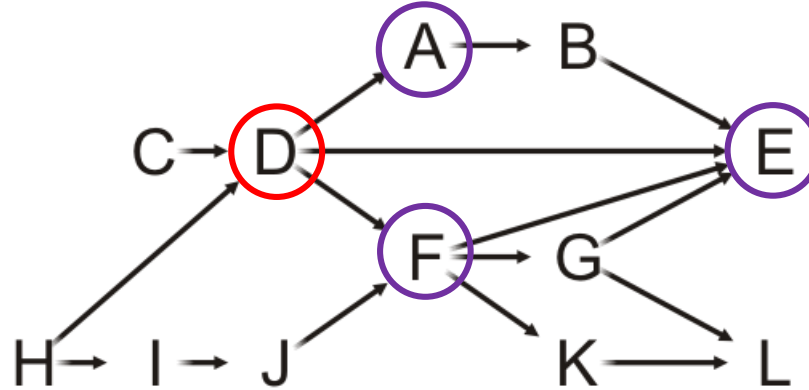


A	1
B	1
C	0
D	0
E	4
F	2
G	1
H	0
I	0
J	1
K	1
L	2

Example

Pop the front of the queue

- D has three neighbors: A, E and F
- Decrement their in-degrees

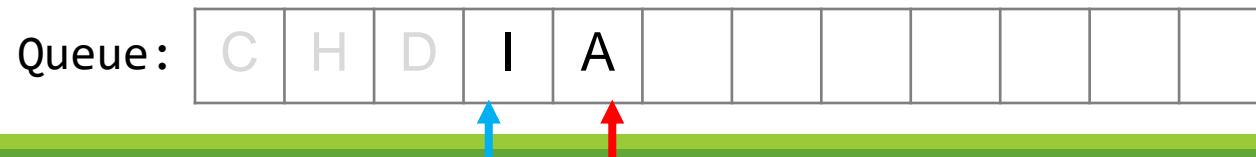
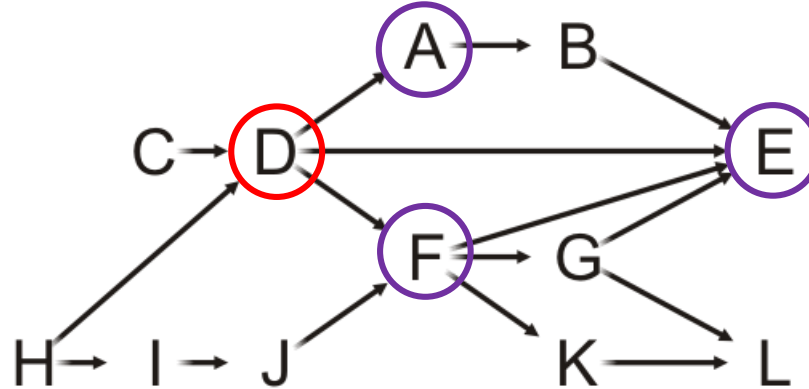


A	0
B	1
C	0
D	0
E	3
F	1
G	1
H	0
I	0
J	1
K	1
L	2

Example

Pop the front of the queue

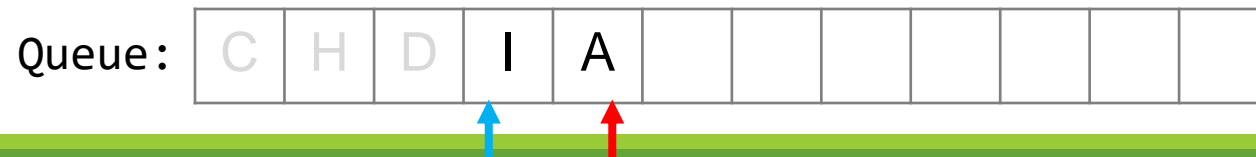
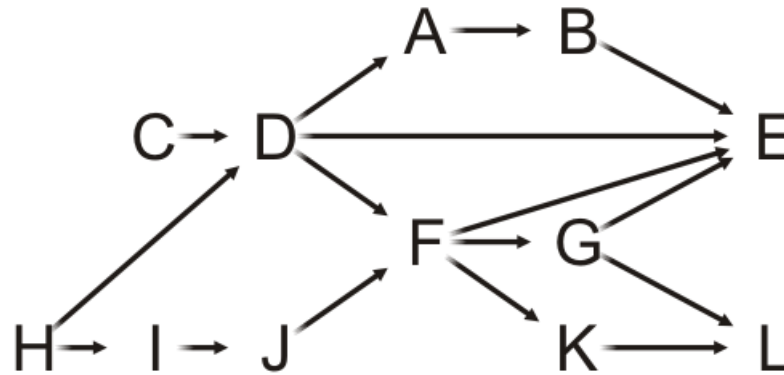
- D has three neighbors: A, E and F
- Decrement their in-degrees
 - A is decremented to zero, so push it onto the queue



A	0
B	1
C	0
D	0
E	3
F	1
G	1
H	0
I	0
J	1
K	1
L	2

Example

Pop the front of the queue

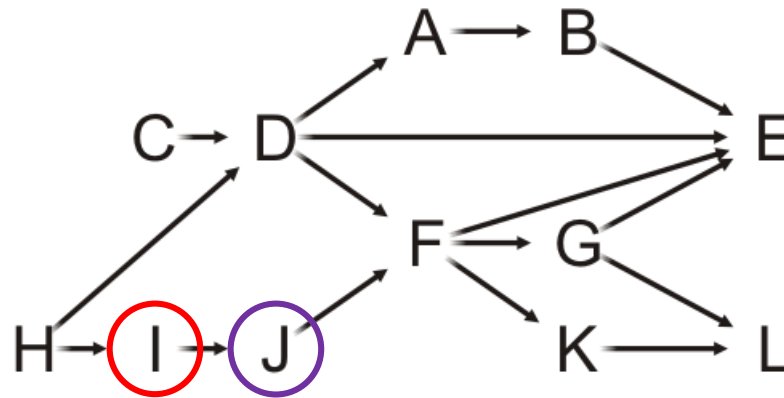


A	0
B	1
C	0
D	0
E	3
F	1
G	1
H	0
I	0
J	1
K	1
L	2

Example

Pop the front of the queue

- I has one neighbor: J



Queue:

C	H	D	I	A							
---	---	---	---	---	--	--	--	--	--	--	--

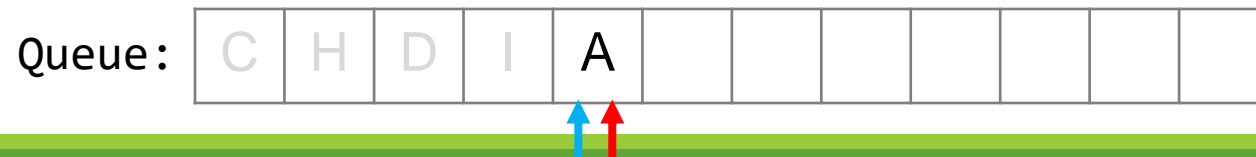
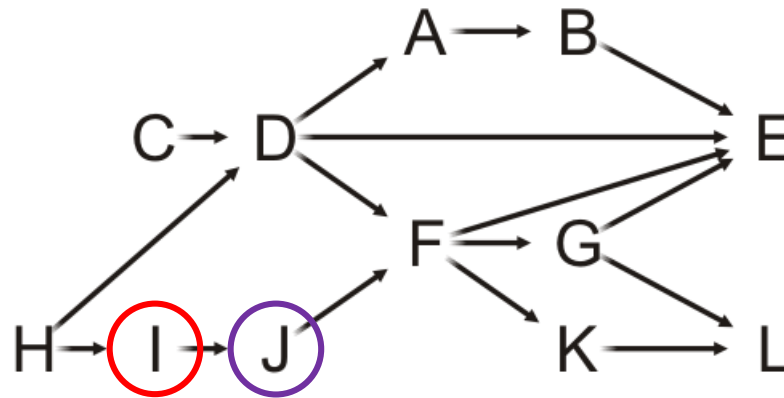


A	0
B	1
C	0
D	0
E	3
F	1
G	1
H	0
I	0
J	1
K	1
L	2

Example

Pop the front of the queue

- I has one neighbor: J
- Decrement its in-degree

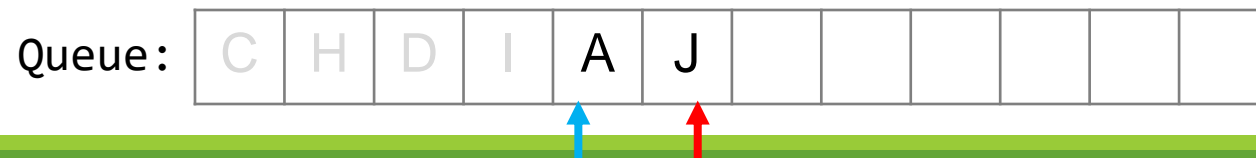
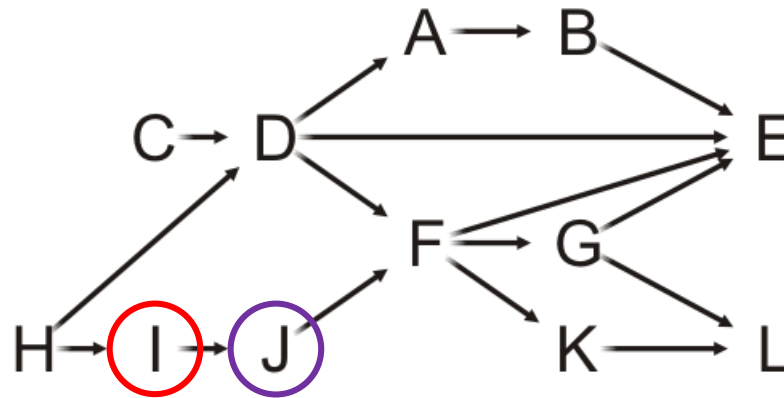


A	0
B	1
C	0
D	0
E	3
F	1
G	1
H	0
I	0
J	0
K	1
L	2

Example

Pop the front of the queue

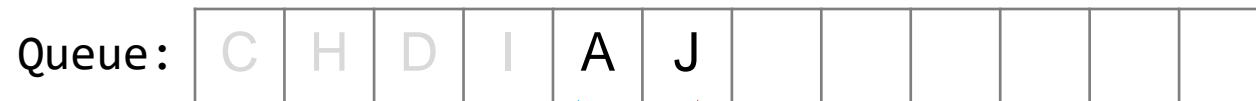
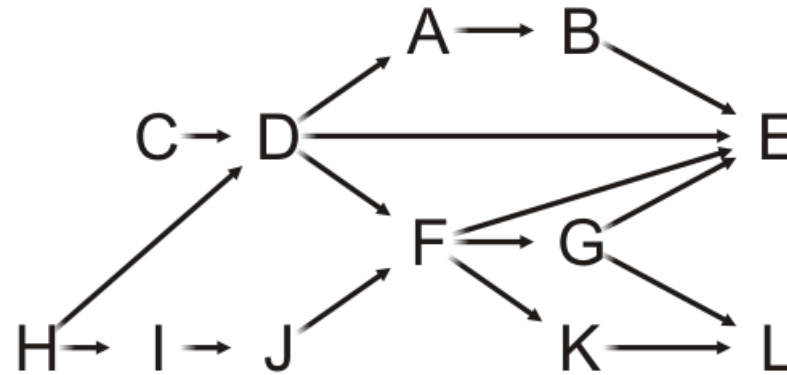
- I has one neighbor: J
- Decrement its in-degree
 - J is decremented to zero, so push it onto the queue



A	0
B	1
C	0
D	0
E	3
F	1
G	1
H	0
I	0
J	0
K	1
L	2

Example

Pop the front of the queue

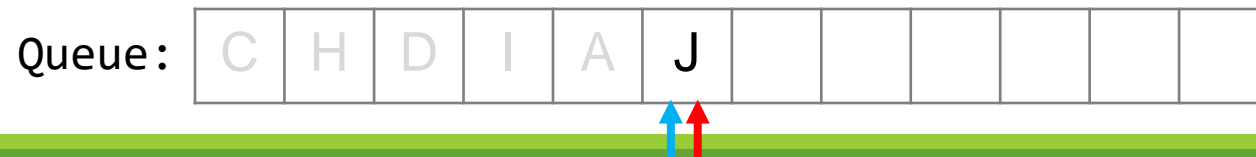
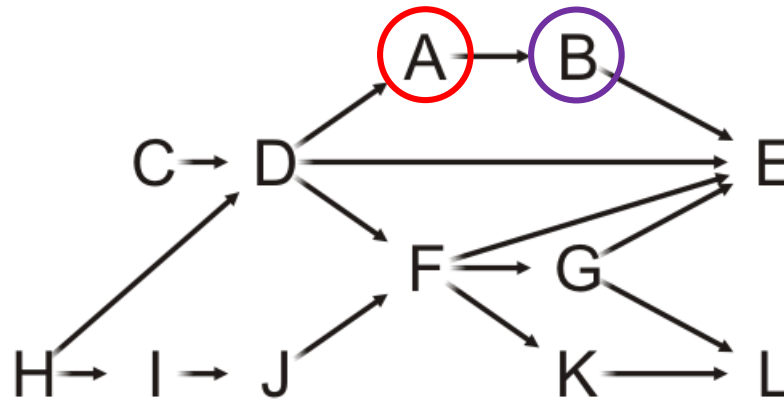


A	0
B	1
C	0
D	0
E	3
F	1
G	1
H	0
I	0
J	0
K	1
L	2

Example

Pop the front of the queue

- A has one neighbor: B

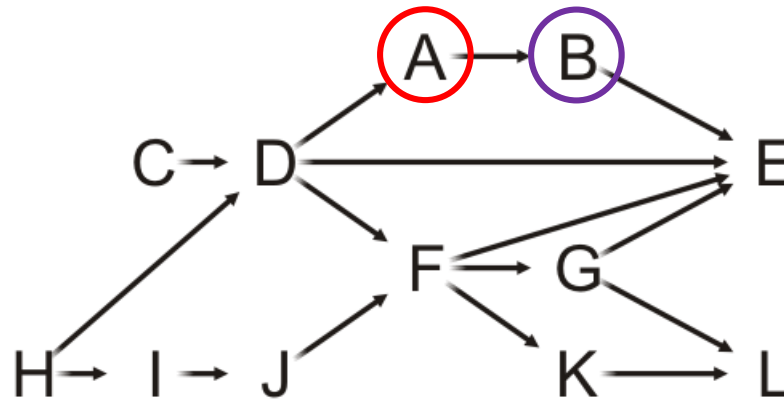


A	0
B	1
C	0
D	0
E	3
F	1
G	1
H	0
I	0
J	0
K	1
L	2

Example

Pop the front of the queue

- A has one neighbor: B
- Decrement its in-degree



Queue:

C	H	D	I	A	J						
---	---	---	---	---	---	--	--	--	--	--	--

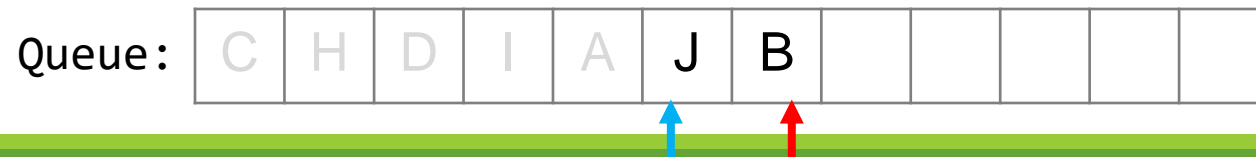
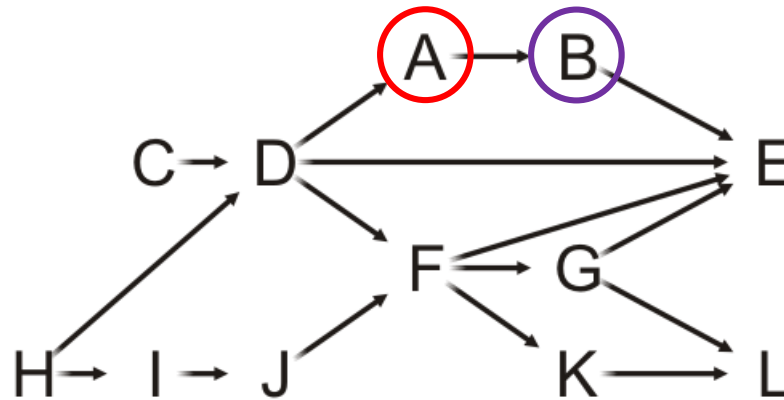


A	0
B	0
C	0
D	0
E	3
F	1
G	1
H	0
I	0
J	0
K	1
L	2

Example

Pop the front of the queue

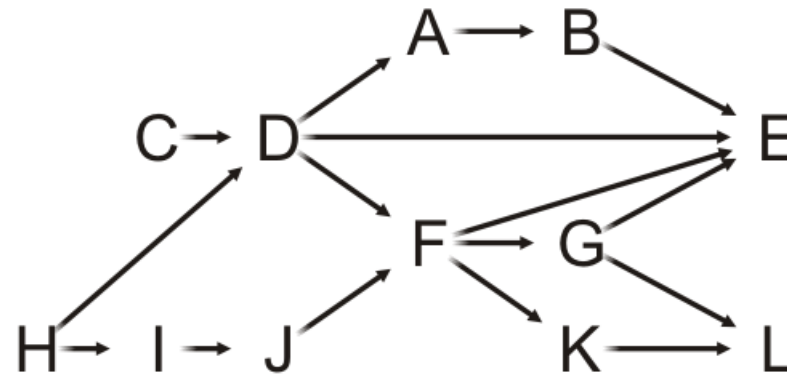
- A has one neighbor: B
- Decrement its in-degree
 - B is decremented to zero, so push it onto the queue



A	0
B	0
C	0
D	0
E	3
F	1
G	1
H	0
I	0
J	0
K	1
L	2

Example

Pop the front of the queue



Queue:

C	H	D	I	A	J	B					
---	---	---	---	---	---	---	--	--	--	--	--

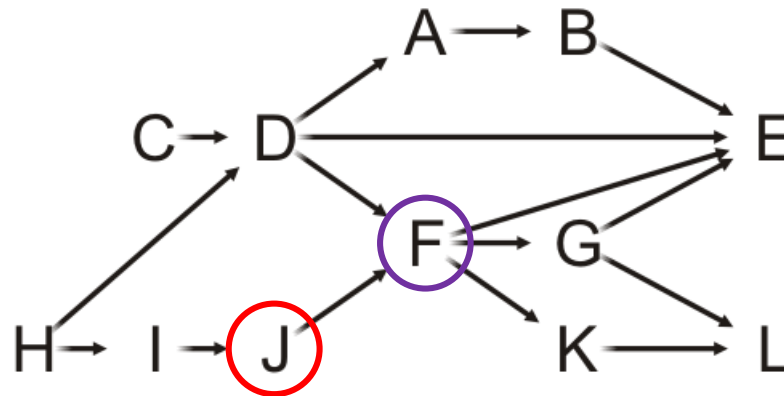


A	0
B	0
C	0
D	0
E	3
F	1
G	1
H	0
I	0
J	0
K	1
L	2

Example

Pop the front of the queue

- J has one neighbor: F

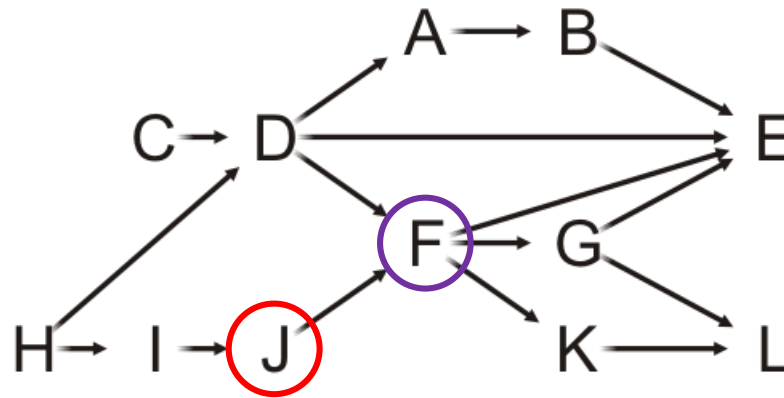


A	0
B	0
C	0
D	0
E	3
F	1
G	1
H	0
I	0
J	0
K	1
L	2

Example

Pop the front of the queue

- J has one neighbor: F
- Decrement its in-degree

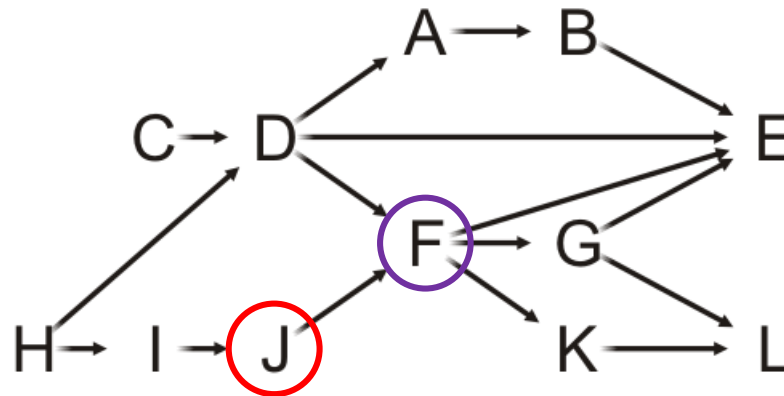


A	0
B	0
C	0
D	0
E	3
F	0
G	1
H	0
I	0
J	0
K	1
L	2

Example

Pop the front of the queue

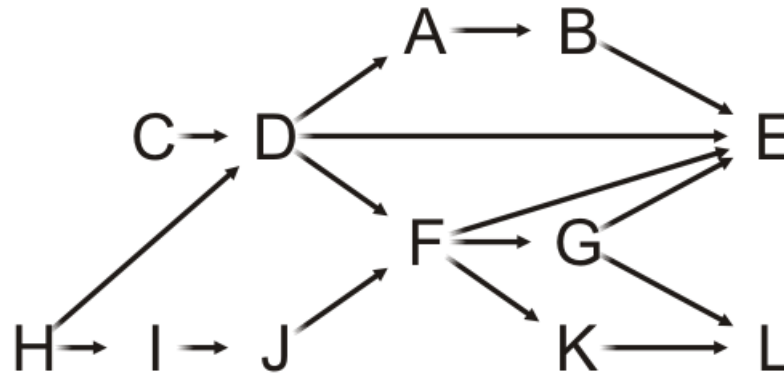
- J has one neighbor: F
- Decrement its in-degree
 - F is decremented to zero, so push it onto the queue



A	0
B	0
C	0
D	0
E	3
F	0
G	1
H	0
I	0
J	0
K	1
L	2

Example

Pop the front of the queue

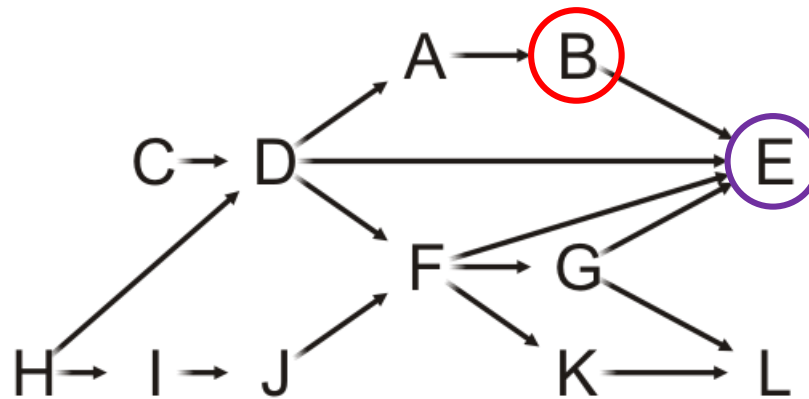


A	0
B	0
C	0
D	0
E	3
F	0
G	1
H	0
I	0
J	0
K	1
L	2

Example

Pop the front of the queue

- B has one neighbor: E



Queue:

C	H	D	I	A	J	B	F				
---	---	---	---	---	---	---	---	--	--	--	--

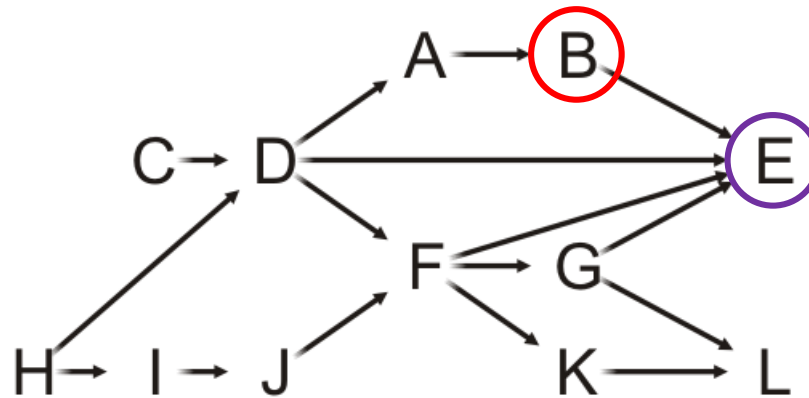


A	0
B	0
C	0
D	0
E	3
F	0
G	1
H	0
I	0
J	0
K	1
L	2

Example

Pop the front of the queue

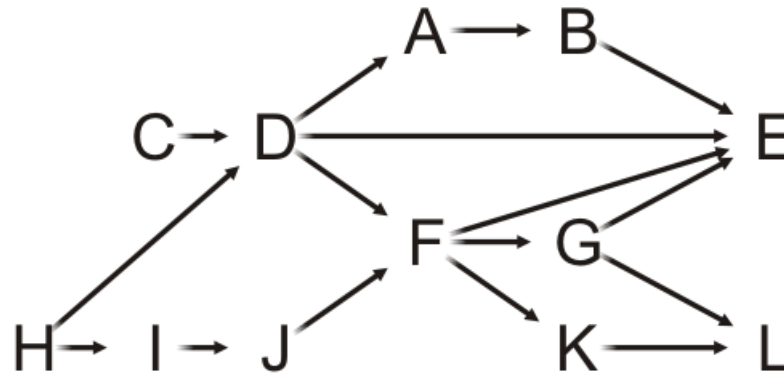
- B has one neighbor: E
- Decrement its in-degree



A	0
B	0
C	0
D	0
E	2
F	0
G	1
H	0
I	0
J	0
K	1
L	2

Example

Pop the front of the queue



Queue:

C	H	D	I	A	J	B	F				
---	---	---	---	---	---	---	---	--	--	--	--

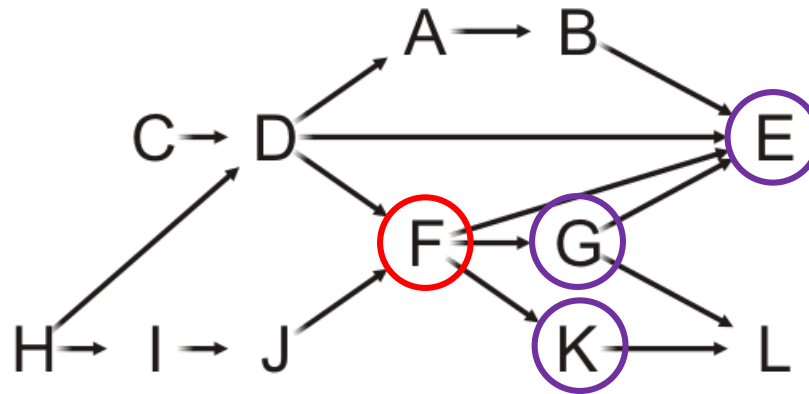


A	0
B	0
C	0
D	0
E	2
F	0
G	1
H	0
I	0
J	0
K	1
L	2

Example

Pop the front of the queue

- F has three neighbors: E, G and K



Queue:

C	H	D	I	A	J	B	F				
---	---	---	---	---	---	---	---	--	--	--	--

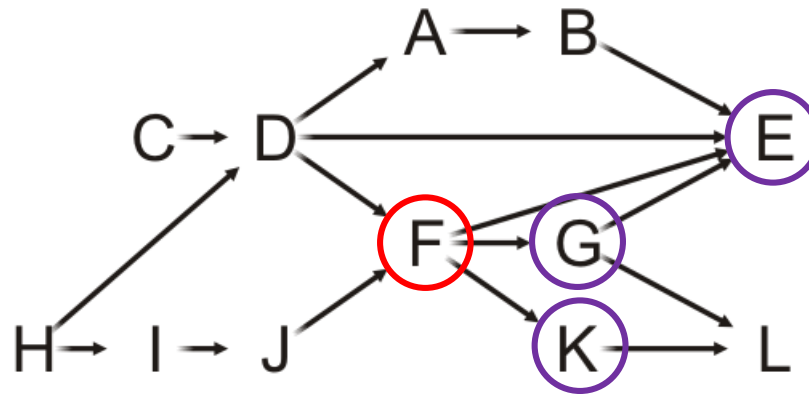


A	0
B	0
C	0
D	0
E	2
F	0
G	1
H	0
I	0
J	0
K	1
L	2

Example

Pop the front of the queue

- F has three neighbors: E, G and K
- Decrement their in-degrees

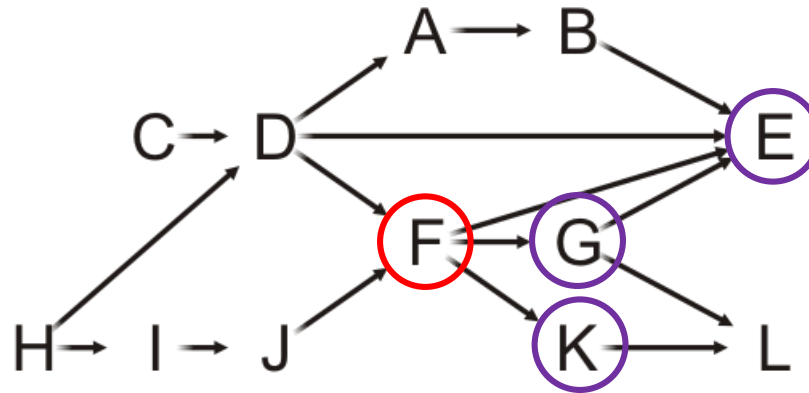


A	0
B	0
C	0
D	0
E	1
F	0
G	0
H	0
I	0
J	0
K	0
L	2

Example

Pop the front of the queue

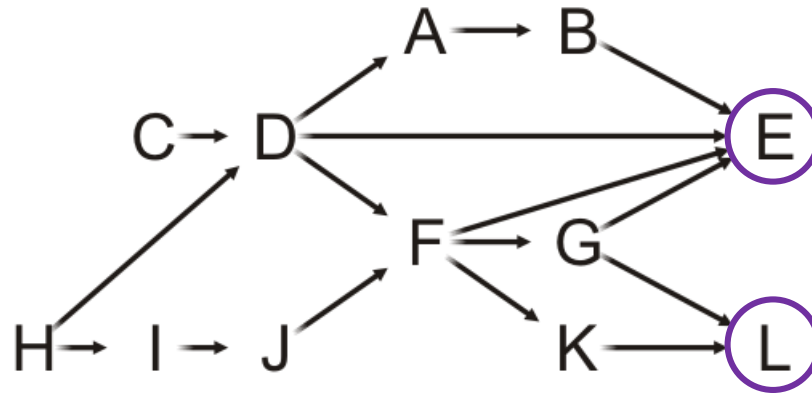
- F has three neighbors: E, G and K
- Decrement their in-degrees
 - G and K are decremented to zero, so push them onto the queue



A	0
B	0
C	0
D	0
E	1
F	0
G	0
H	0
I	0
J	0
K	0
L	2

Example

Pop the front of the queue

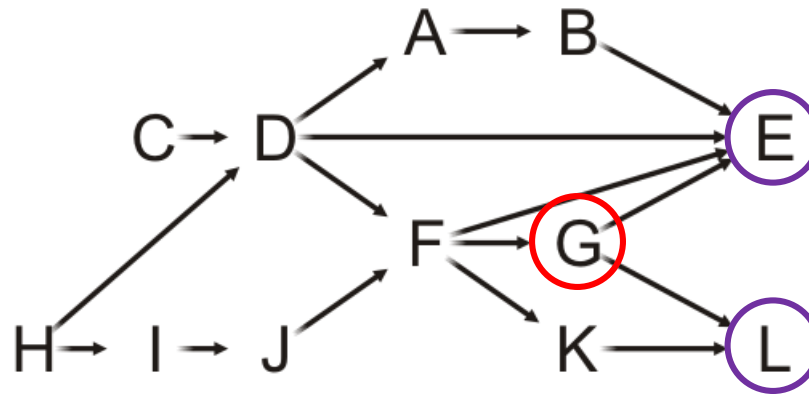


A	0
B	0
C	0
D	0
E	1
F	0
G	0
H	0
I	0
J	0
K	0
L	2

Example

Pop the front of the queue

- G has two neighbors: E and L

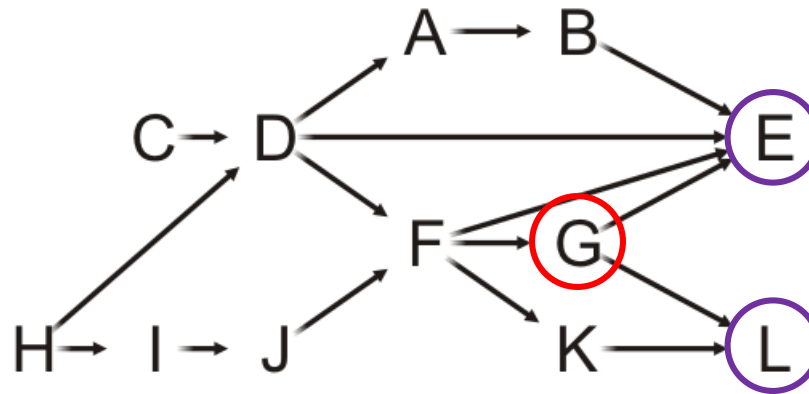


A	0
B	0
C	0
D	0
E	1
F	0
G	0
H	0
I	0
J	0
K	0
L	2

Example

Pop the front of the queue

- G has two neighbors: E and L
- Decrement their in-degrees

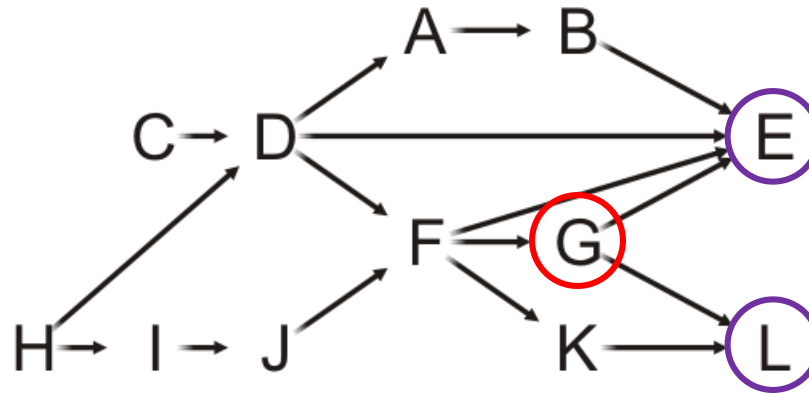


A	0
B	0
C	0
D	0
E	0
F	0
G	0
H	0
I	0
J	0
K	0
L	1

Example

Pop the front of the queue

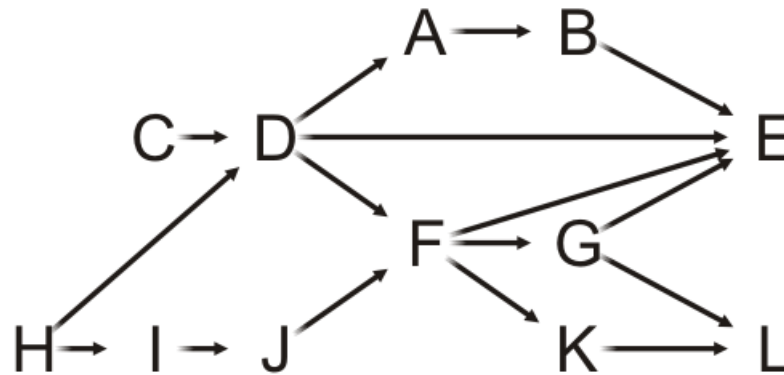
- G has two neighbors: E and L
- Decrement their in-degrees
 - E is decremented to zero, so push it onto the queue



A	0
B	0
C	0
D	0
E	0
F	0
G	0
H	0
I	0
J	0
K	0
L	1

Example

Pop the front of the queue



Queue:

C	H	D	I	A	J	B	F	G	K	E	
---	---	---	---	---	---	---	---	---	---	---	--

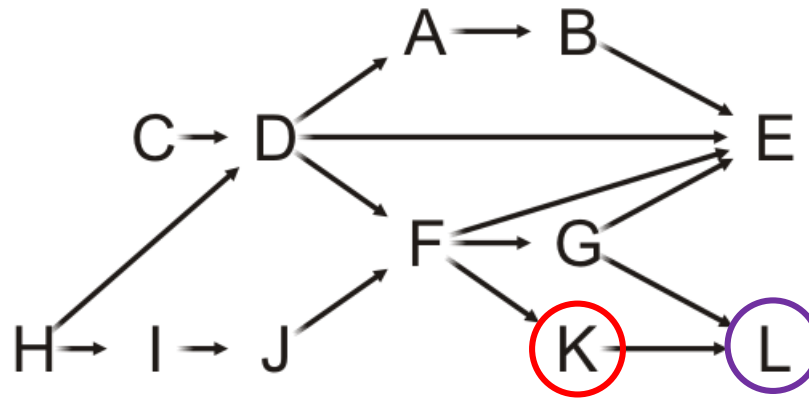


A	0
B	0
C	0
D	0
E	0
F	0
G	0
H	0
I	0
J	0
K	0
L	1

Example

Pop the front of the queue

- K has one neighbors: L



Queue:

C	H	D	I	A	J	B	F	G	K	E	
---	---	---	---	---	---	---	---	---	---	---	--

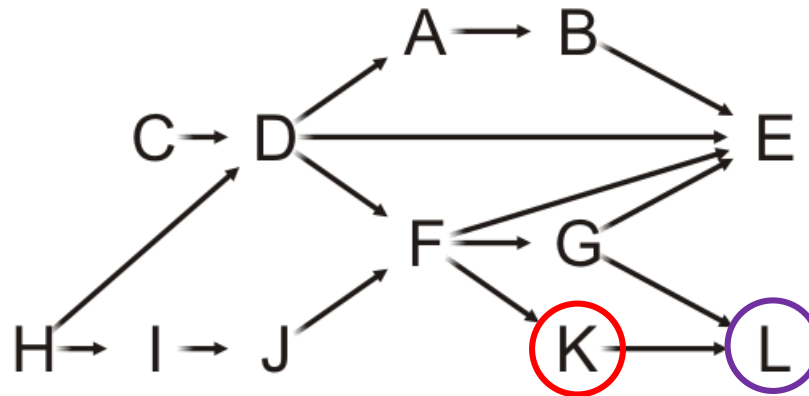


A	0
B	0
C	0
D	0
E	0
F	0
G	0
H	0
I	0
J	0
K	0
L	1

Example

Pop the front of the queue

- K has one neighbors: L
- Decrement its in-degree



Queue:

C	H	D	I	A	J	B	F	G	K	E	
---	---	---	---	---	---	---	---	---	---	---	--

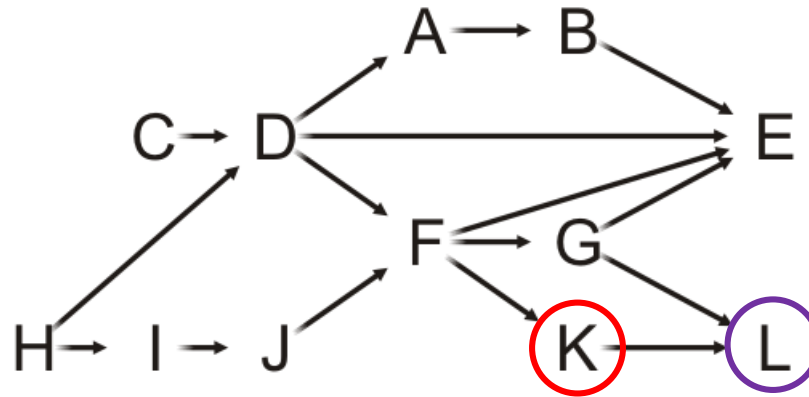


A	0
B	0
C	0
D	0
E	0
F	0
G	0
H	0
I	0
J	0
K	0
L	0

Example

Pop the front of the queue

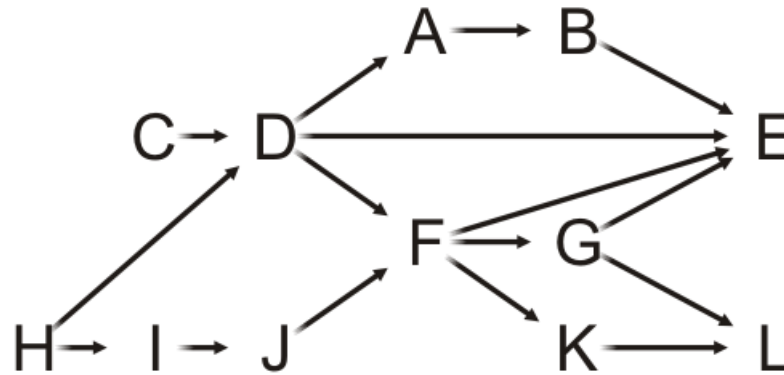
- K has one neighbors: L
- Decrement its in-degree
 - L is decremented to zero, so push it onto the queue



A	0
B	0
C	0
D	0
E	0
F	0
G	0
H	0
I	0
J	0
K	0
L	0

Example

Pop the front of the queue



Queue:

C	H	D	I	A	J	B	F	G	K	E	L
---	---	---	---	---	---	---	---	---	---	---	---

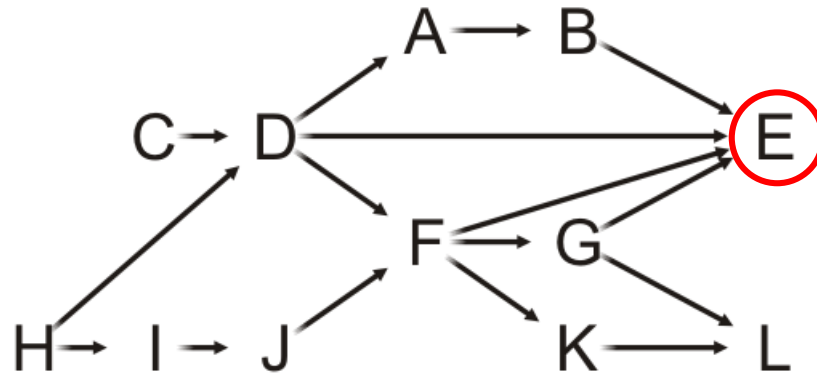


A	0
B	0
C	0
D	0
E	0
F	0
G	0
H	0
I	0
J	0
K	0
L	0

Example

Pop the front of the queue

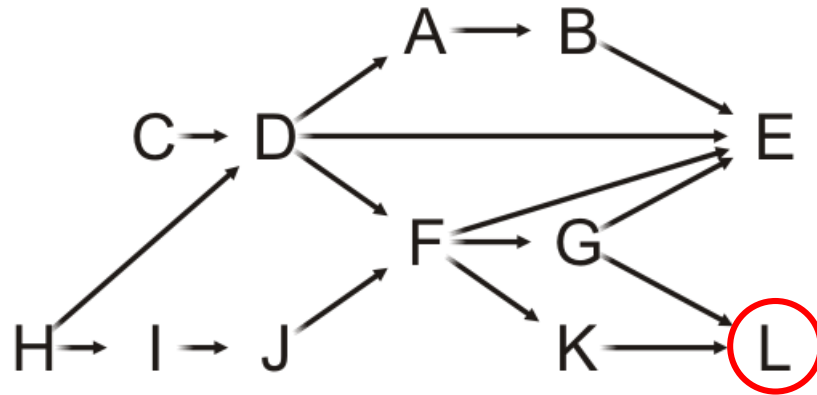
- E has no neighbors—it is a *sink*



A	0
B	0
C	0
D	0
E	0
F	0
G	0
H	0
I	0
J	0
K	0
L	0

Example

Pop the front of the queue

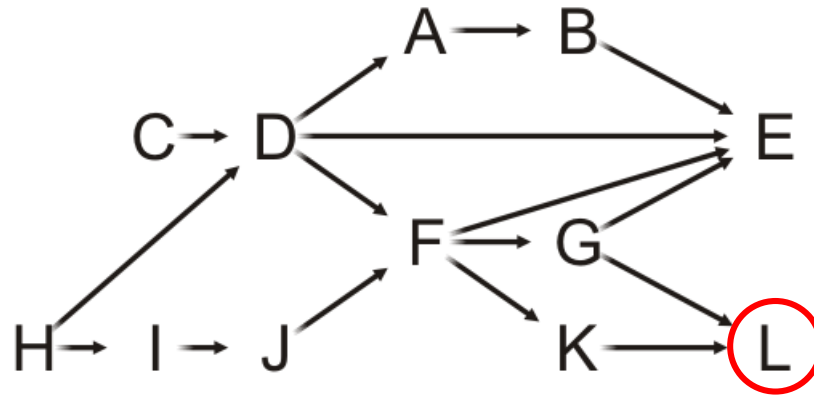


A	0
B	0
C	0
D	0
E	0
F	0
G	0
H	0
I	0
J	0
K	0
L	0

Example

Pop the front of the queue

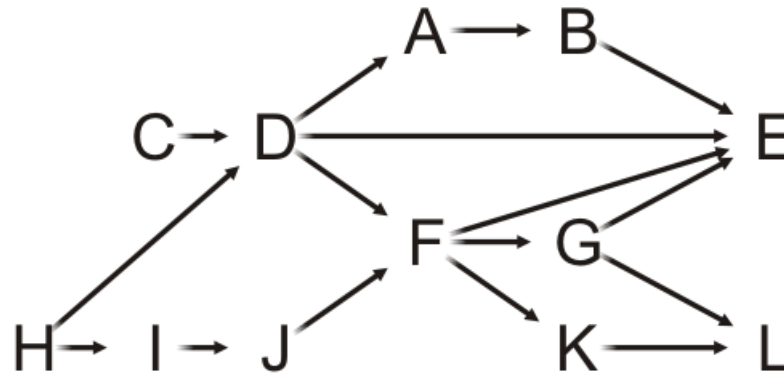
- L has no neighbors—it is also a *sink*



A	0
B	0
C	0
D	0
E	0
F	0
G	0
H	0
I	0
J	0
K	0
L	0

Example

The queue is empty, so we are done



Queue:

C	H	D	I	A	J	B	F	G	K	E	L
---	---	---	---	---	---	---	---	---	---	---	---

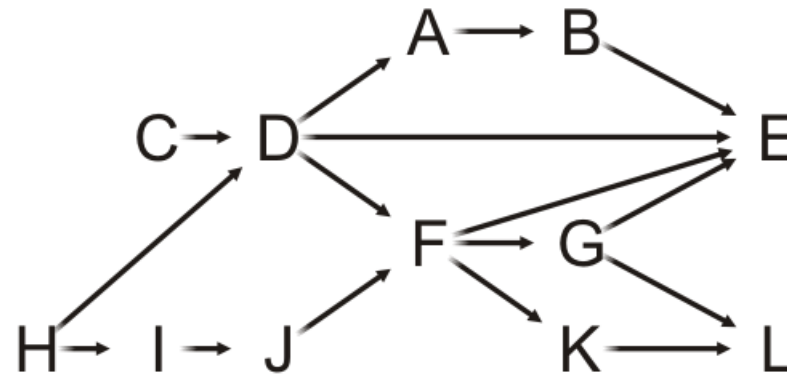


A	0
B	0
C	0
D	0
E	0
F	0
G	0
H	0
I	0
J	0
K	0
L	0

Example

We deallocate the memory for the temporary in-degree array

The array stores the topological sorting

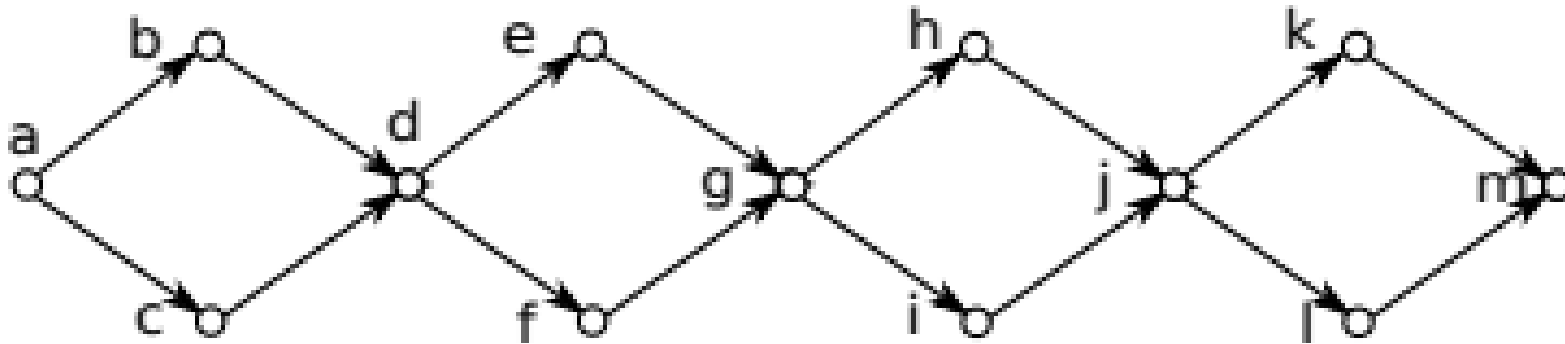


C	H	D	I	A	J	B	F	G	K	E	L
---	---	---	---	---	---	---	---	---	---	---	---

A	0
B	0
C	0
D	0
E	0
F	0
G	0
H	0
I	0
J	0
K	0
L	0

Question

Give a topological sort of the following graph

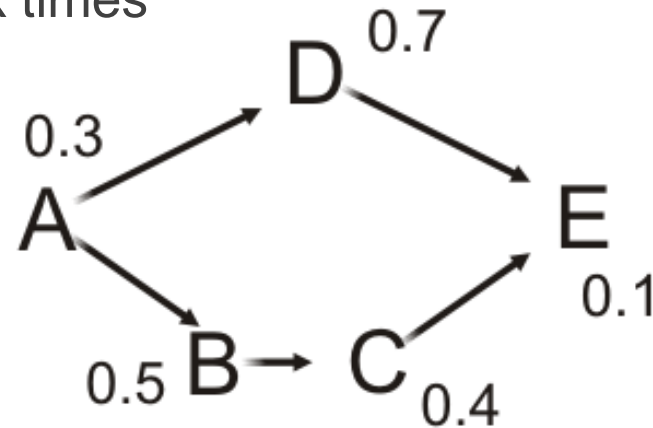


What is the number of different topological sorts of this graph? (**Bonus Mark Question**) (**Negative mark for wrong answer**)

Critical path

Suppose each task has a performance time associated with it

- If the tasks are performed serially, the time required to complete the last task equals to the sum of the individual task times

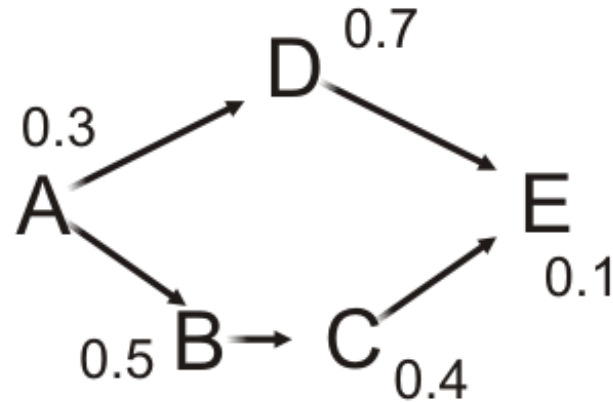


- These tasks require $0.3 + 0.7 + 0.5 + 0.4 + 0.1 = 2.0$ s to execute serially

Critical path

Suppose two tasks are ready to execute

- We could perform these tasks in parallel

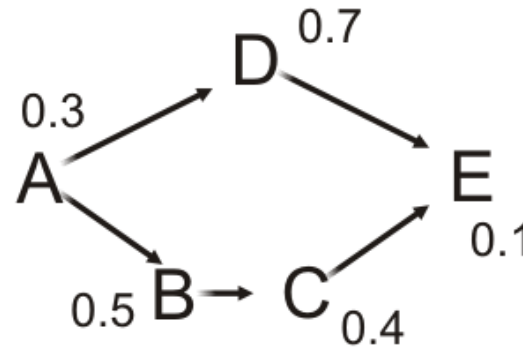


- Computer tasks can be executed in parallel (multi-processing)
- Different tasks can be completed by different teams in a company

Critical path

Suppose Task A completes

- We can now execute Tasks B and D in parallel



- However, Task E cannot execute until Task C completes, and Task C cannot execute until Task B completes
 - The least time in which these five tasks can be completed is
$$0.3 + 0.5 + 0.4 + 0.1 = 1.3 \text{ s}$$
 - This is called the *critical time of all tasks*
 - The path (A, B, C, E) is said to be the *critical path*

Finding the critical path

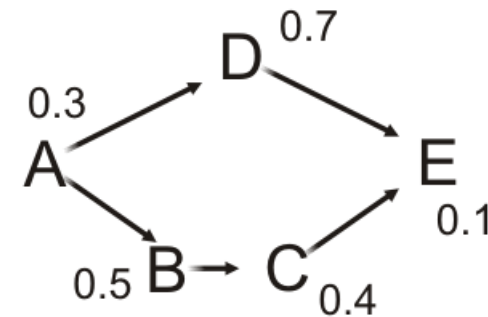
Tasks that have no prerequisites have a critical time equal to the time it takes to complete that task

For tasks that depend on others, the critical time will be:

- The maximum critical time that it takes to complete a prerequisite
- Plus the time it takes to complete this task

In this example, the critical times are:

- Task A completes in 0.3 s
- Task B must wait for A and completes after 0.8 s
- Task D must wait for A and completes after 1.0 s
- Task C must wait for B and completes after 1.2 s
- Task E must wait for both C and D, and completes after $\max(1.0, 1.2) + 0.1 = 1.3$ s



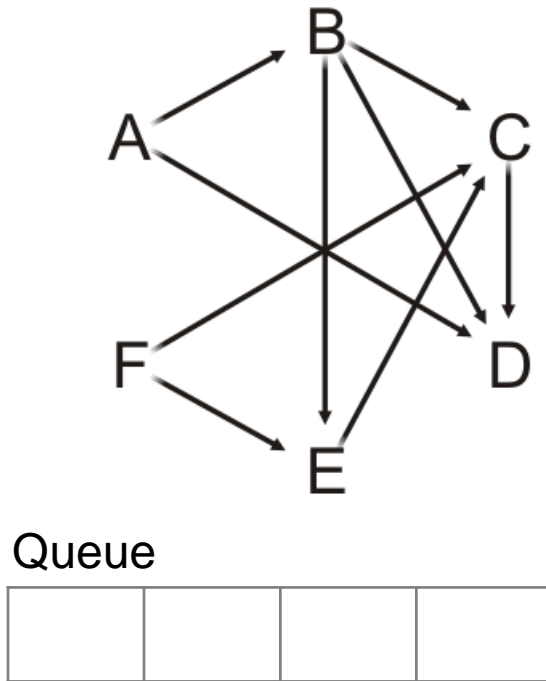
Finding the critical path

Thus, we require more information:

- We must know the execution time of each task
- We will have to record the critical time for each task
 - Initialize these to zero
- We will need to know the previous task with the longest critical time to determine the critical path
 - Set these to null

Finding the critical path

Suppose we have the following times for the tasks



Task	In-degree	Task Time	Critical Time	Previous Task
A	0	5.2	0.0	∅
B	1	6.1	0.0	∅
C	3	4.7	0.0	∅
D	3	8.1	0.0	∅
E	2	9.5	0.0	∅
F	0	17.1	0.0	∅

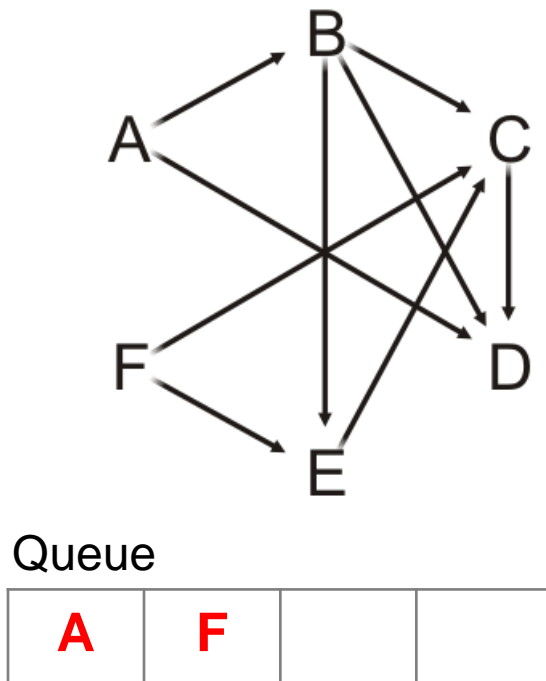
Finding the critical path

Each time we pop a vertex v , in addition to what we already do:

- For v , add the task time onto the critical time for that vertex:
 - That is the critical time for v
- For each adjacent vertex w :
 - If the critical time for v is greater than the currently stored critical time for w
 - Update the critical time with the critical time for v
 - Set the previous pointer to the vertex v

Finding the critical path

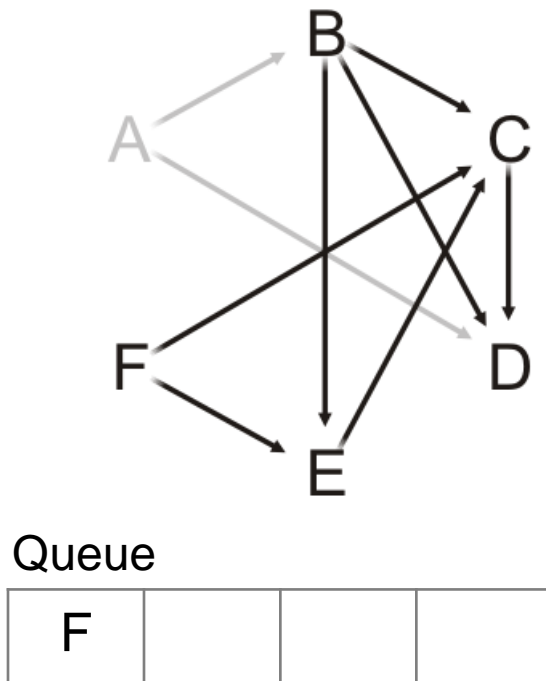
So we initialize the queue with those vertices with in-degree zero



Task	In-degree	Task Time	Critical Time	Previous Task
A	0	5.2	0.0	∅
B	1	6.1	0.0	∅
C	3	4.7	0.0	∅
D	3	8.1	0.0	∅
E	2	9.5	0.0	∅
F	0	17.1	0.0	∅

Finding the critical path

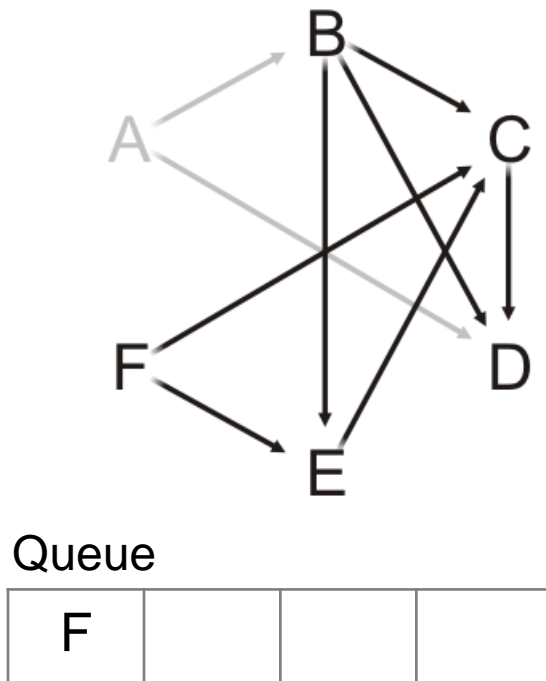
Pop Task A and update its critical time $0.0 + 5.2 = 5.2$



Task	In-degree	Task Time	Critical Time	Previous Task
A	0	5.2	0.0	∅
B	1	6.1	0.0	∅
C	3	4.7	0.0	∅
D	3	8.1	0.0	∅
E	2	9.5	0.0	∅
F	0	17.1	0.0	∅

Finding the critical path

Pop Task A and update its critical time $0.0 + 5.2 = 5.2$

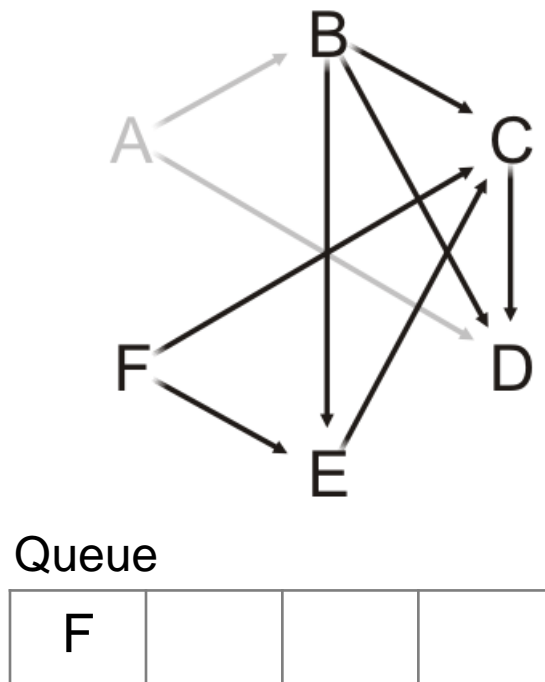


Task	In-degree	Task Time	Critical Time	Previous Task
A	0	5.2	5.2	∅
B	1	6.1	0.0	∅
C	3	4.7	0.0	∅
D	3	8.1	0.0	∅
E	2	9.5	0.0	∅
F	0	17.1	0.0	∅

Finding the critical path

For each neighbor of Task A:

- Decrement the in-degree, push if necessary, and check if we must update the critical time

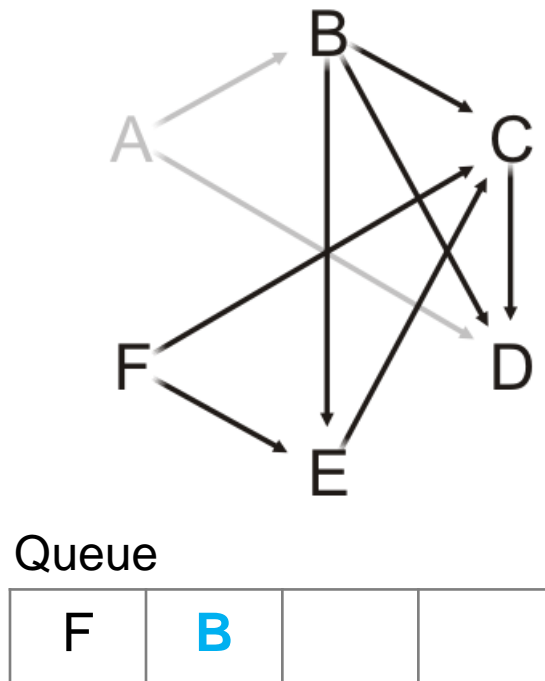


Task	In-degree	Task Time	Critical Time	Previous Task
A	0	5.2	5.2	∅
B	1	6.1	0.0	∅
C	3	4.7	0.0	∅
D	3	8.1	0.0	∅
E	2	9.5	0.0	∅
F	0	17.1	0.0	∅

Finding the critical path

For each neighbor of Task A:

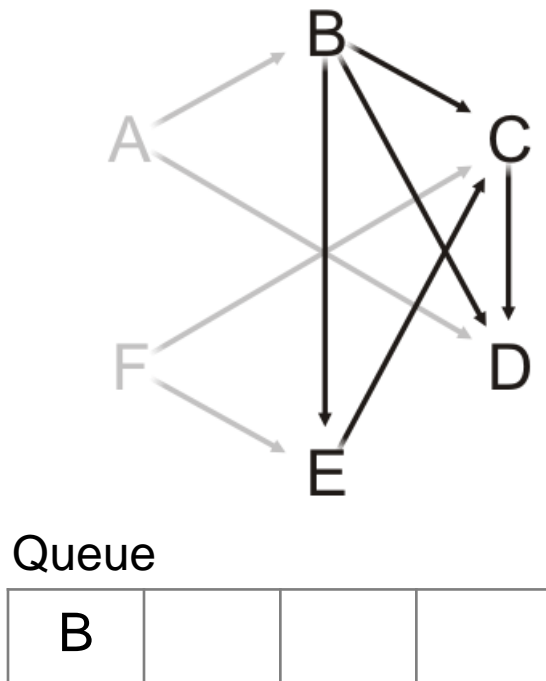
- Decrement the in-degree, push if necessary, and check if we must update the critical time



Task	In-degree	Task Time	Critical Time	Previous Task
A	0	5.2	5.2	∅
B	0	6.1	5.2	A
C	3	4.7	0.0	∅
D	2	8.1	5.2	A
E	2	9.5	0.0	∅
F	0	17.1	0.0	∅

Finding the critical path

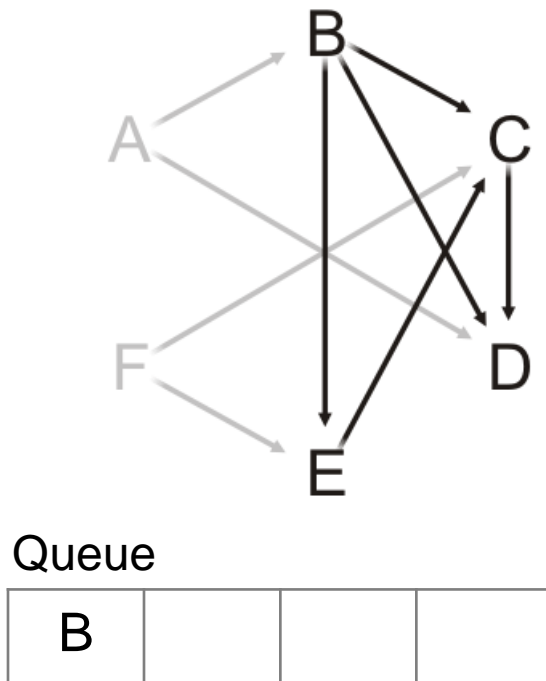
Pop Task F and update its critical time $0.0 + 17.1 = 17.1$



Task	In-degree	Task Time	Critical Time	Previous Task
A	0	5.2	5.2	∅
B	0	6.1	5.2	A
C	3	4.7	0.0	∅
D	2	8.1	5.2	A
E	2	9.5	0.0	∅
F	0	17.1	0.0	∅

Finding the critical path

Pop Task F and update its critical time $0.0 + 17.1 = 17.1$

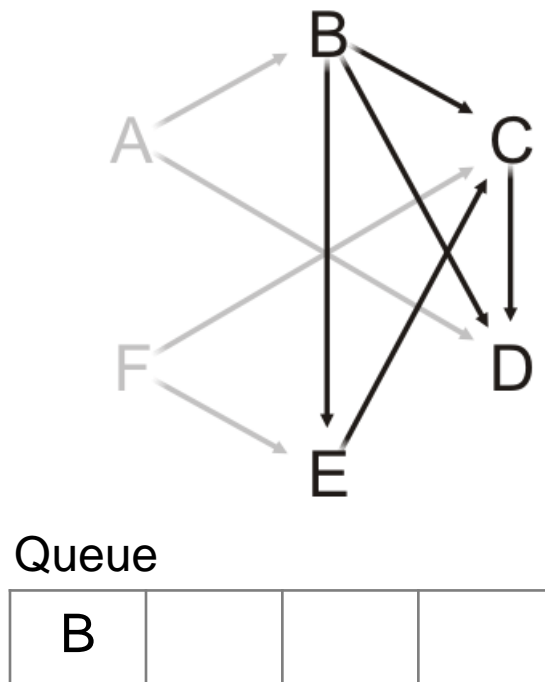


Task	In-degree	Task Time	Critical Time	Previous Task
A	0	5.2	5.2	∅
B	0	6.1	5.2	A
C	3	4.7	0.0	∅
D	2	8.1	5.2	A
E	2	9.5	0.0	∅
F	0	17.1	17.1	∅

Finding the critical path

For each neighbor of Task F:

- Decrement the in-degree, push if necessary, and check if we must update the critical time

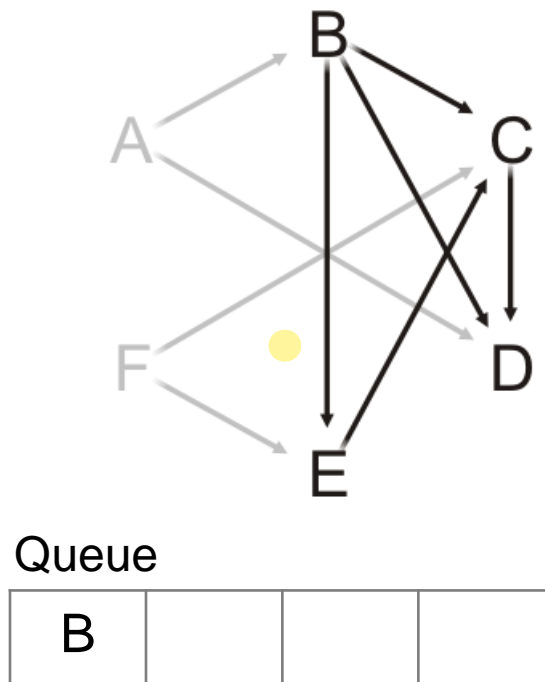


Task	In-degree	Task Time	Critical Time	Previous Task
A	0	5.2	5.2	∅
B	0	6.1	5.2	A
C	3	4.7	0.0	∅
D	2	8.1	5.2	A
E	2	9.5	0.0	∅
F	0	17.1	17.1	∅

Finding the critical path

For each neighbor of Task F:

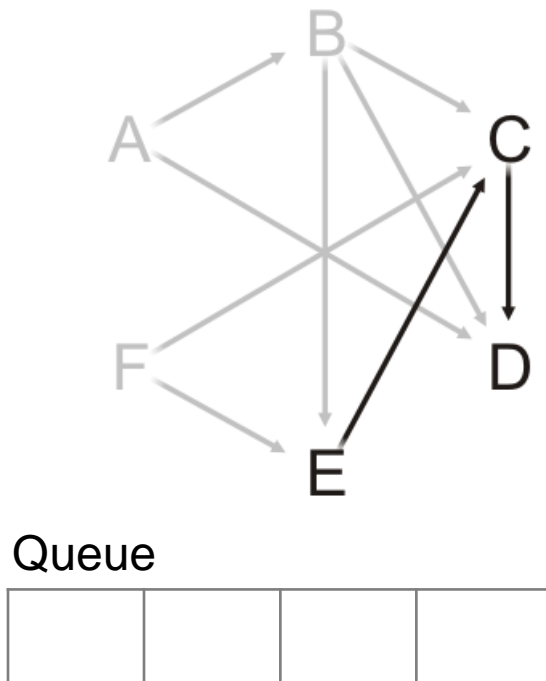
- Decrement the in-degree, push if necessary, and check if we must update the critical time



Task	In-degree	Task Time	Critical Time	Previous Task
A	0	5.2	5.2	∅
B	0	6.1	5.2	A
C	2	4.7	17.1	F
D	2	8.1	5.2	A
E	1	9.5	17.1	F
F	0	17.1	17.1	∅

Finding the critical path

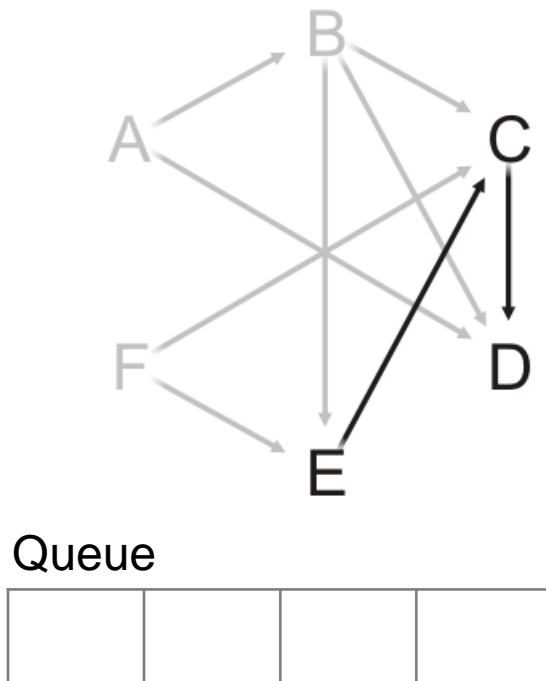
Pop Task B and update its critical time $5.2 + 6.1 = 11.3$



Task	In-degree	Task Time	Critical Time	Previous Task
A	0	5.2	5.2	∅
B	0	6.1	5.2	A
C	2	4.7	17.1	F
D	2	8.1	5.2	A
E	1	9.5	17.1	F
F	0	17.1	17.1	∅

Finding the critical path

Pop Task B and update its critical time $5.2 + 6.1 = 11.3$

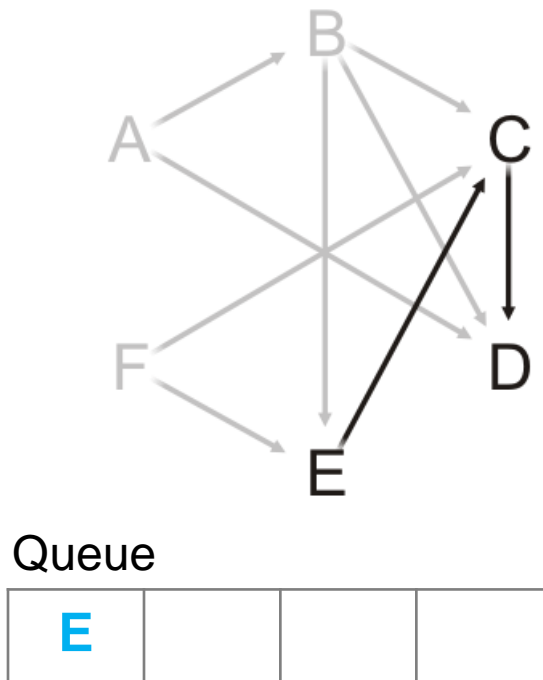


Task	In-degree	Task Time	Critical Time	Previous Task
A	0	5.2	5.2	∅
B	0	6.1	11.3	A
C	2	4.7	17.1	F
D	2	8.1	5.2	A
E	1	9.5	17.1	F
F	0	17.1	17.1	∅

Finding the critical path

For each neighbor of Task B:

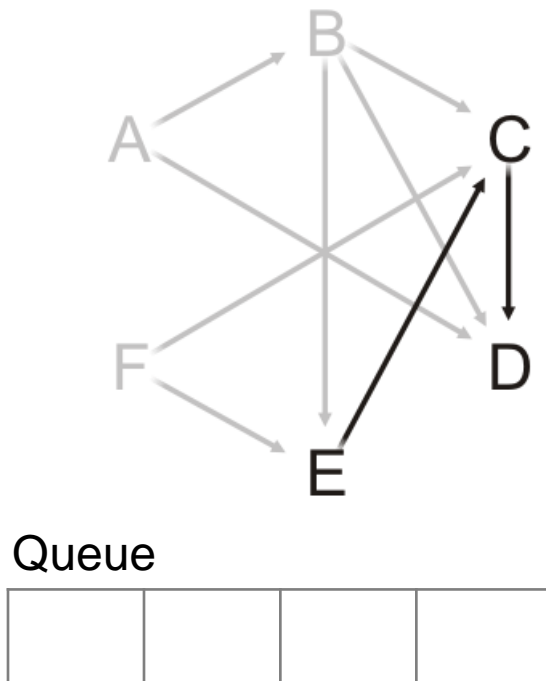
- Decrement the in-degree, push if necessary, and check if we must update the critical time
- Both C and E are waiting on F



Task	In-degree	Task Time	Critical Time	Previous Task
A	0	5.2	5.2	∅
B	0	6.1	11.3	A
C	1	4.7	17.1	F
D	1	8.1	11.3	B
E	0	9.5	17.1	F
F	0	17.1	17.1	∅

Finding the critical path

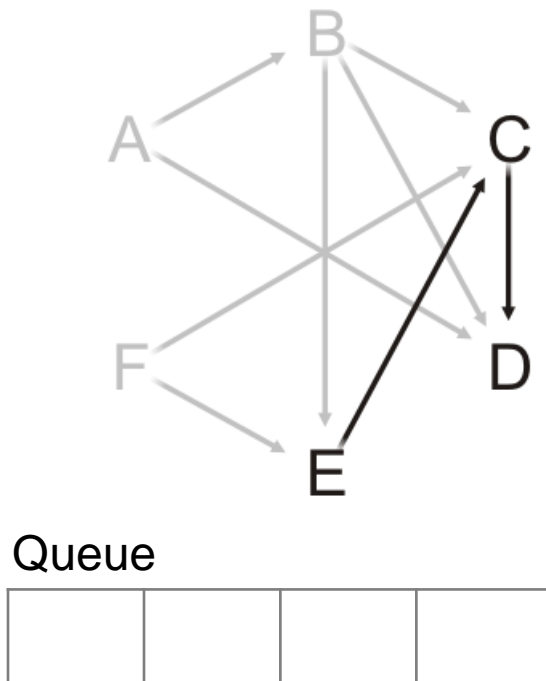
Pop Task E and update its critical time $17.1 + 9.5 = 26.6$



Task	In-degree	Task Time	Critical Time	Previous Task
A	0	5.2	5.2	∅
B	0	6.1	11.3	A
C	1	4.7	17.1	F
D	1	8.1	11.3	B
E	0	9.5	17.1	F
F	0	17.1	17.1	∅

Finding the critical path

Pop Task E and update its critical time $17.1 + 9.5 = 26.6$

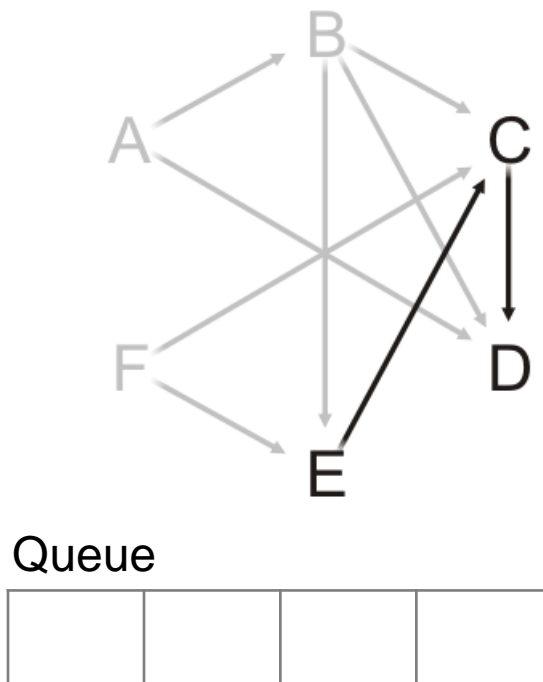


Task	In-degree	Task Time	Critical Time	Previous Task
A	0	5.2	5.2	∅
B	0	6.1	11.3	A
C	1	4.7	17.1	F
D	1	8.1	11.3	B
E	0	9.5	26.6	F
F	0	17.1	17.1	∅

Finding the critical path

For each neighbor of Task E:

- Decrement the in-degree, push if necessary, and check if we must update the critical time

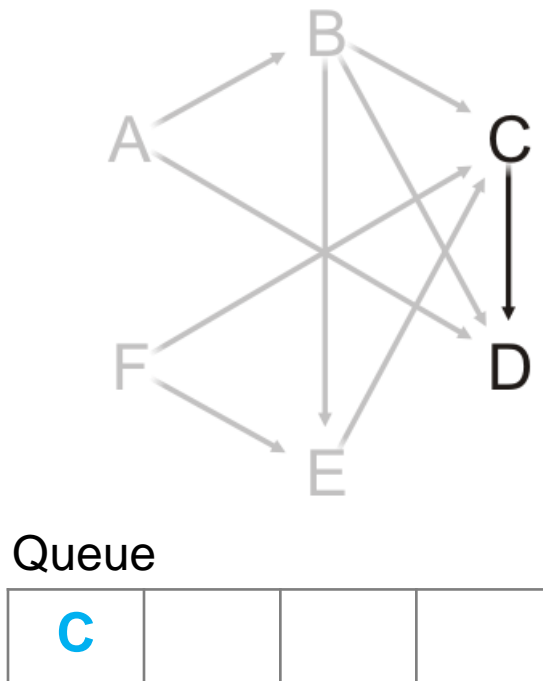


Task	In-degree	Task Time	Critical Time	Previous Task
A	0	5.2	5.2	∅
B	0	6.1	11.3	A
C	1	4.7	17.1	F
D	1	8.1	11.3	B
E	0	9.5	26.6	F
F	0	17.1	17.1	∅

Finding the critical path

For each neighbor of Task E:

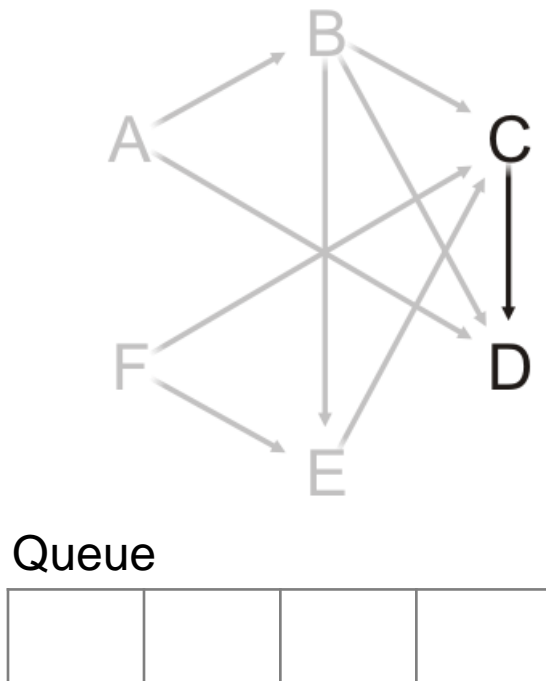
- Decrement the in-degree, push if necessary, and check if we must update the critical time



Task	In-degree	Task Time	Critical Time	Previous Task
A	0	5.2	5.2	∅
B	0	6.1	11.3	A
C	0	4.7	26.6	E
D	1	8.1	11.3	B
E	0	9.5	26.6	F
F	0	17.1	17.1	∅

Finding the critical path

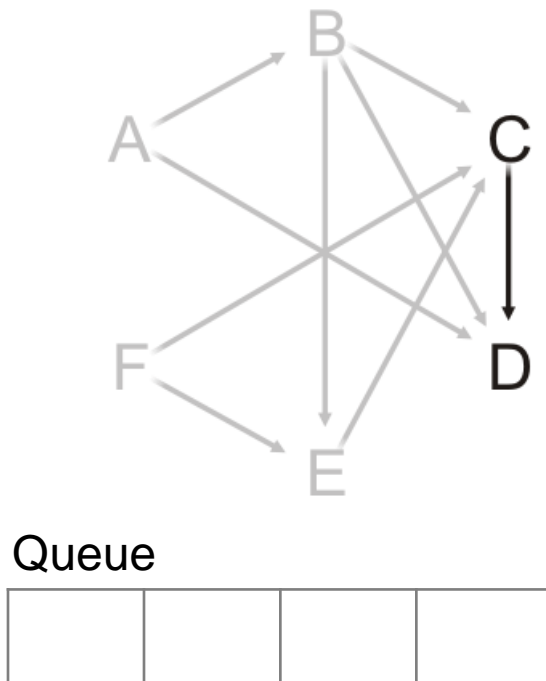
Pop Task C and update its critical time $26.6 + 4.7 = 31.3$



Task	In-degree	Task Time	Critical Time	Previous Task
A	0	5.2	5.2	∅
B	0	6.1	11.3	A
C	0	4.7	26.6	E
D	1	8.1	11.3	B
E	0	9.5	26.6	F
F	0	17.1	17.1	∅

Finding the critical path

Pop Task C and update its critical time $26.6 + 4.7 = 31.3$

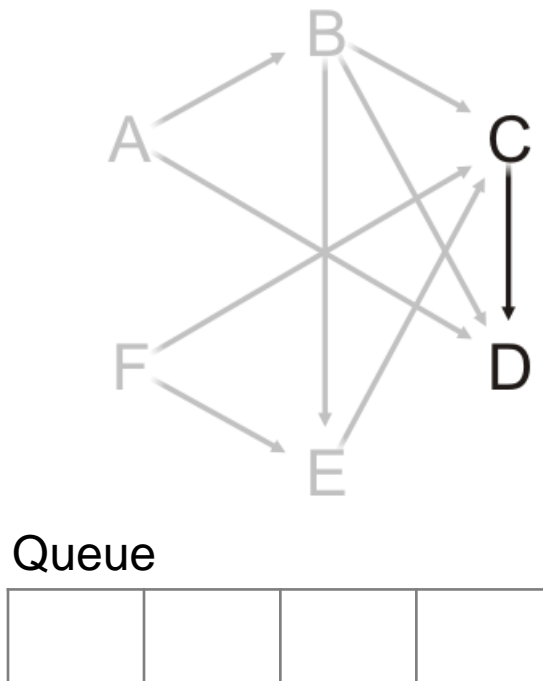


Task	In-degree	Task Time	Critical Time	Previous Task
A	0	5.2	5.2	∅
B	0	6.1	11.3	A
C	0	4.7	31.3	E
D	1	8.1	11.3	B
E	0	9.5	26.6	F
F	0	17.1	17.1	∅

Finding the critical path

For each neighbor of Task C:

- Decrement the in-degree, push if necessary, and check if we must update the critical time

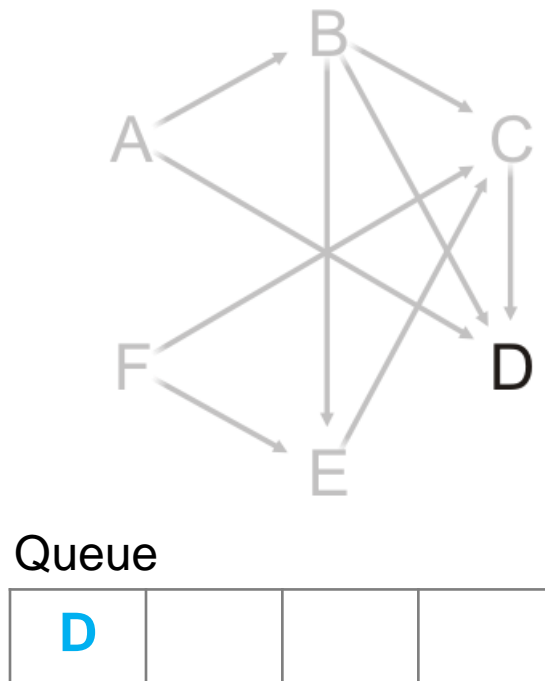


Task	In-degree	Task Time	Critical Time	Previous Task
A	0	5.2	5.2	∅
B	0	6.1	11.3	A
C	0	4.7	31.3	E
D	1	8.1	11.3	B
E	0	9.5	26.6	F
F	0	17.1	17.1	∅

Finding the critical path

For each neighbor of Task C:

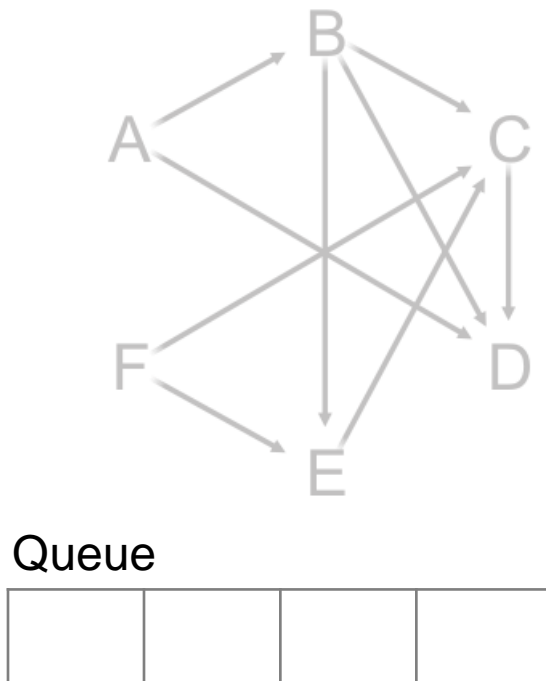
- Decrement the in-degree, push if necessary, and check if we must update the critical time



Task	In-degree	Task Time	Critical Time	Previous Task
A	0	5.2	5.2	∅
B	0	6.1	11.3	A
C	0	4.7	31.3	E
D	0	8.1	31.3	C
E	0	9.5	26.6	F
F	0	17.1	17.1	∅

Finding the critical path

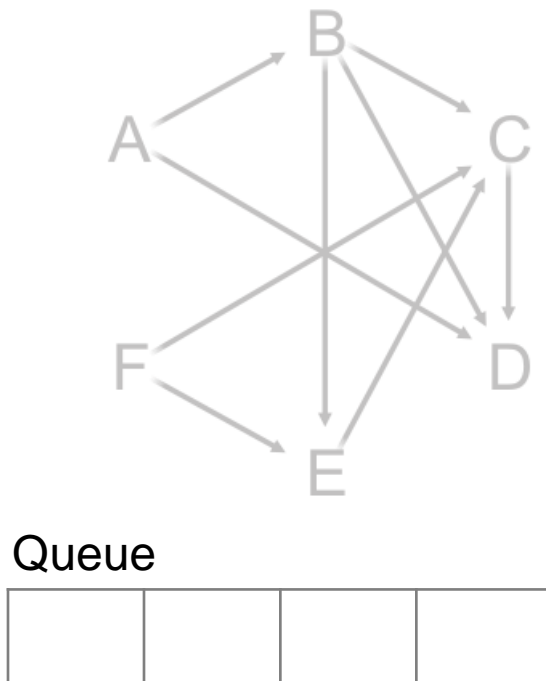
Pop Task D and update its critical time $31.3 + 8.1 = 39.4$



Task	In-degree	Task Time	Critical Time	Previous Task
A	0	5.2	5.2	∅
B	0	6.1	11.3	A
C	0	4.7	31.3	E
D	0	8.1	31.3	C
E	0	9.5	26.6	F
F	0	17.1	17.1	∅

Finding the critical path

Pop Task D and update its critical time $31.3 + 8.1 = 39.4$

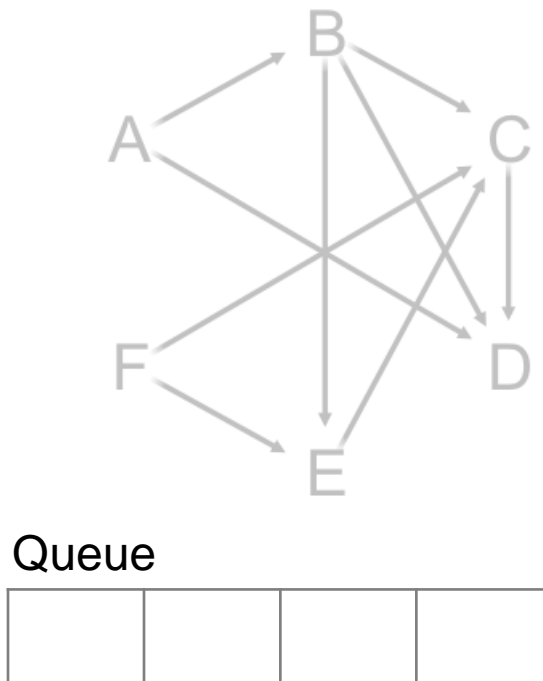


Task	In-degree	Task Time	Critical Time	Previous Task
A	0	5.2	5.2	∅
B	0	6.1	11.3	A
C	0	4.7	31.3	E
D	0	8.1	39.4	C
E	0	9.5	26.6	F
F	0	17.1	17.1	∅

Finding the critical path

Task D has no neighbors and the queue is empty

- We are done

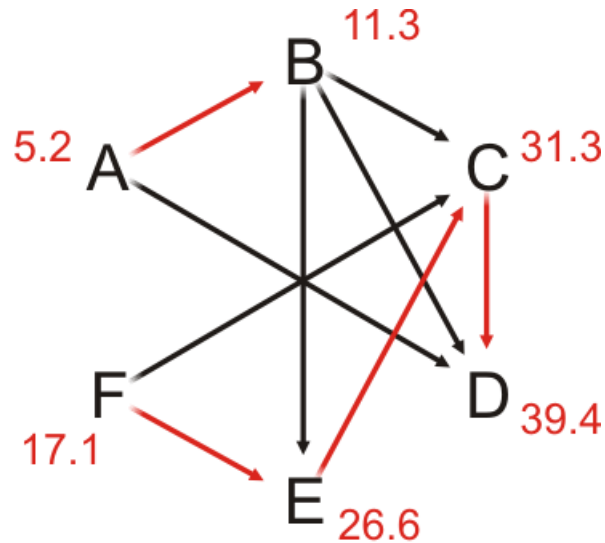


Task	In-degree	Task Time	Critical Time	Previous Task
A	0	5.2	5.2	∅
B	0	6.1	11.3	A
C	0	4.7	31.3	E
D	0	8.1	39.4	C
E	0	9.5	26.6	F
F	0	17.1	17.1	∅

Finding the critical path

Task D has no neighbors and the queue is empty

- We are done



Task	In-degree	Task Time	Critical Time	Previous Task
A	0	5.2	5.2	∅
B	0	6.1	11.3	A
C	0	4.7	31.3	E
D	0	8.1	39.4	C
E	0	9.5	26.6	F
F	0	17.1	17.1	∅