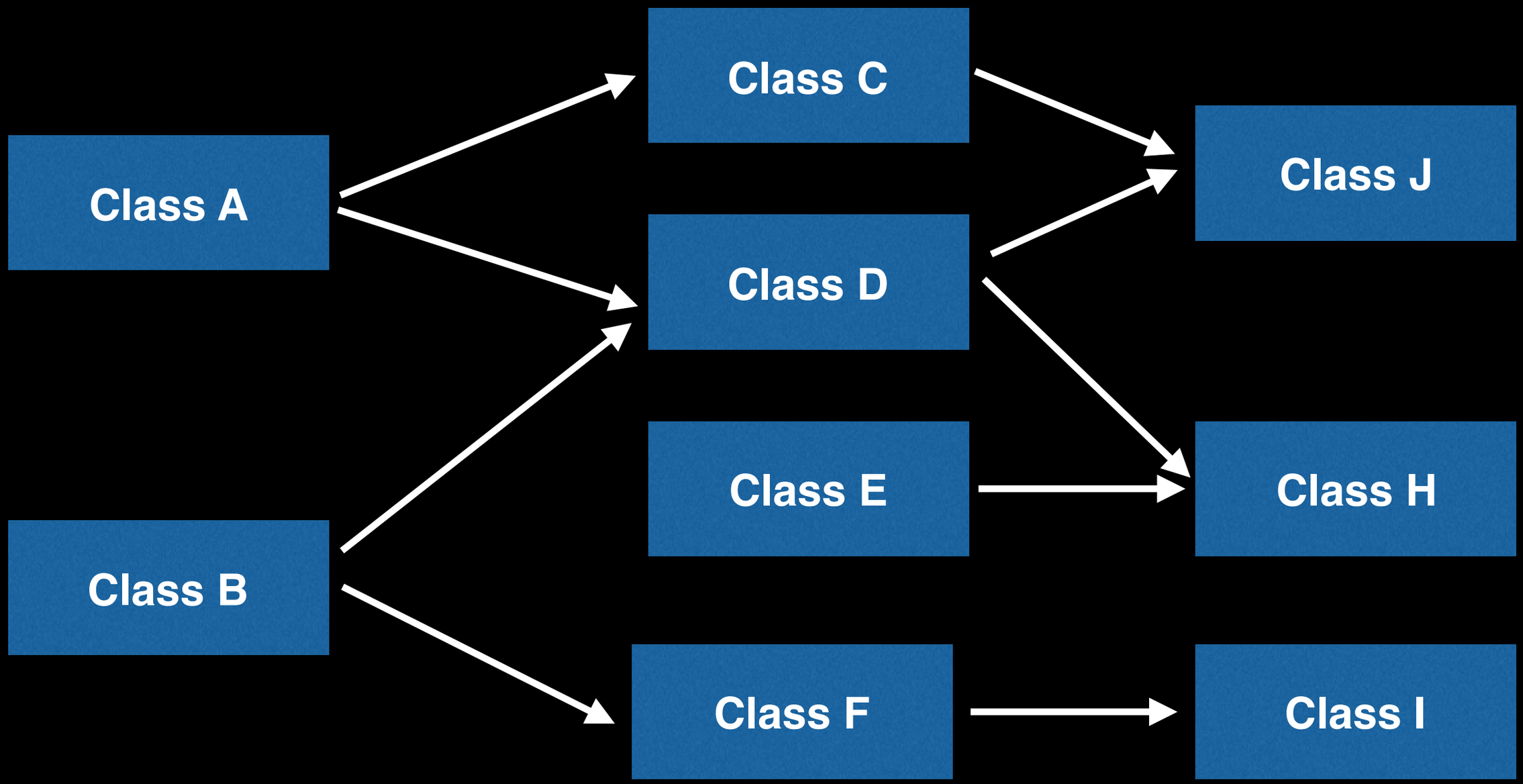
Topological Sort

William Fiset

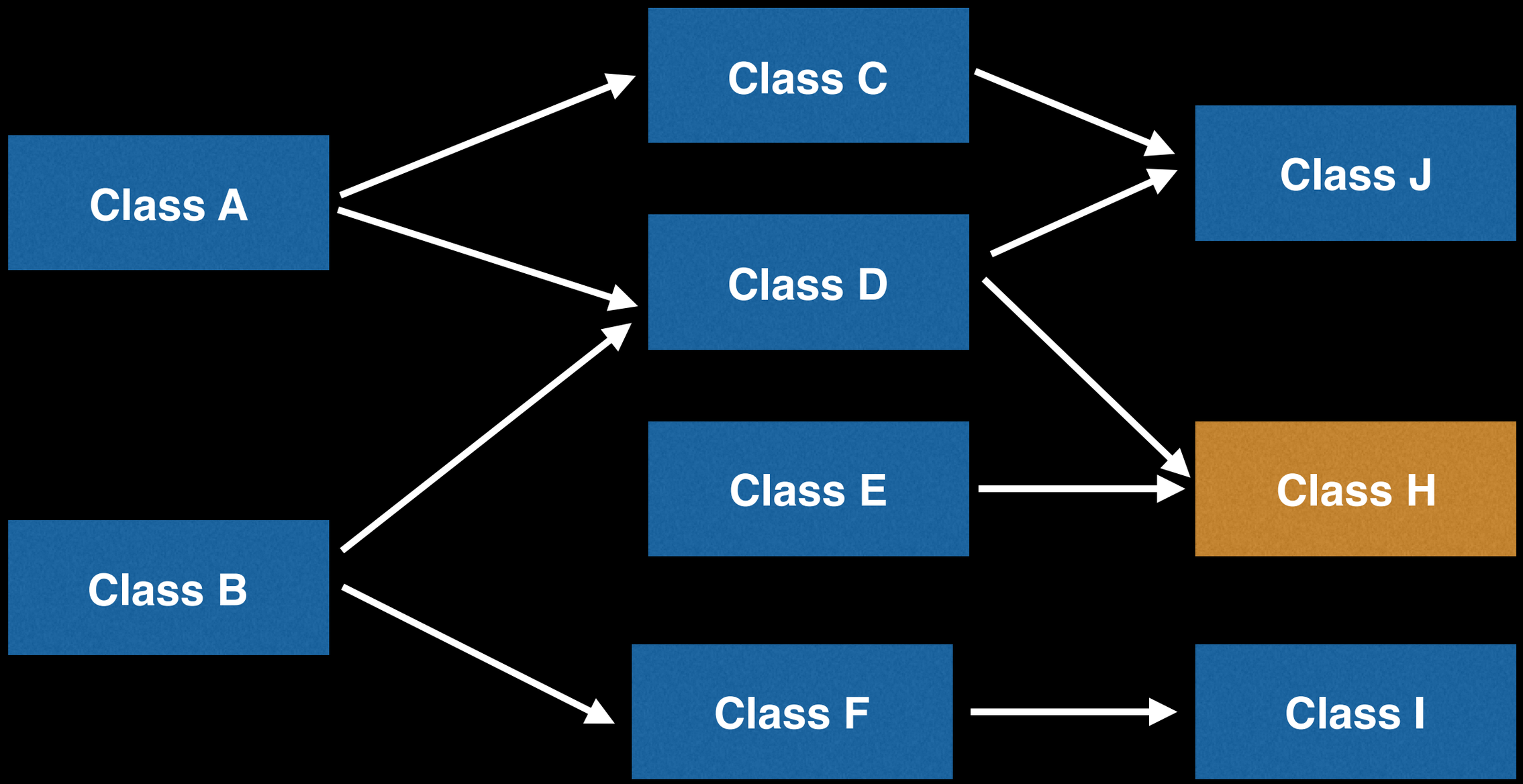
Many real world situations can be modelled as a graph with directed edges where some events must occur before others.

- School class prerequisites
- Program dependencies
- Event scheduling
- Assembly instructions
- Etc…

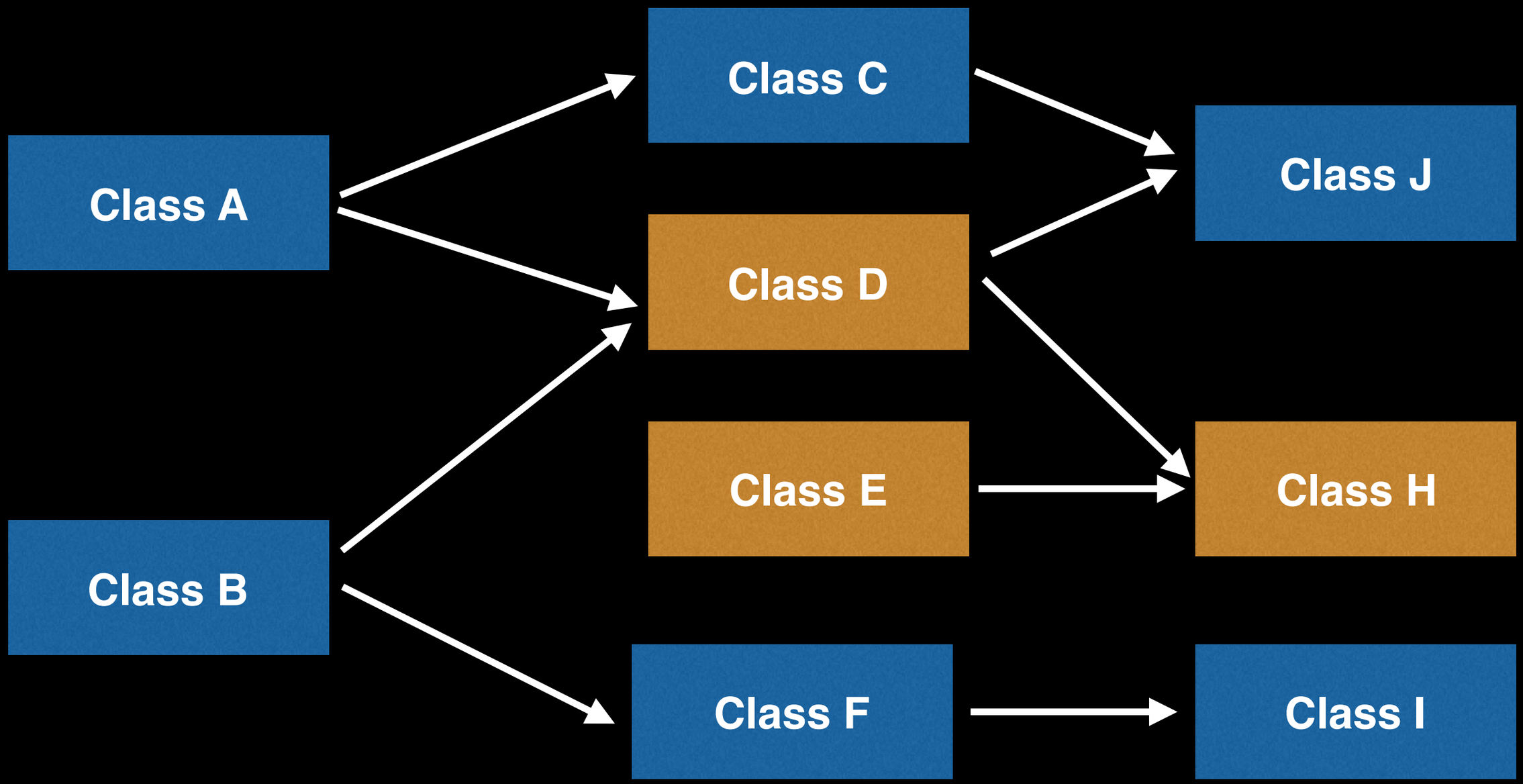
Suppose you're a student at university X and you want to take Class H, then you must take classes A, B, D and E as prerequisites.
In this sense there is an **ordering** on the nodes of the graph.



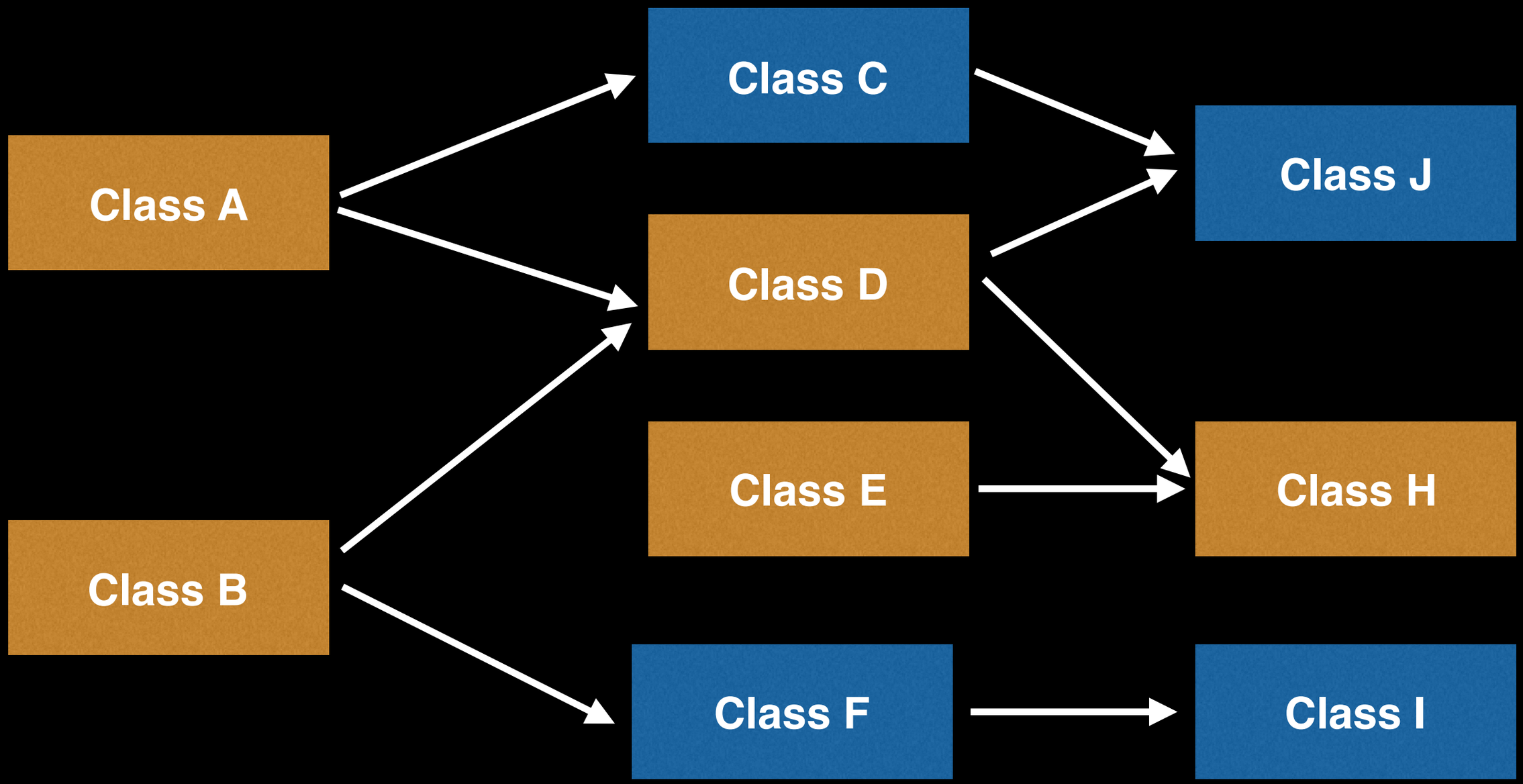
Suppose you're a student at university X and you want to take Class H, then you must take classes A, B, D and E as prerequisites.
In this sense there is an **ordering** on the nodes of the graph.



Suppose you're a student at university X and you want to take Class H, then you must take classes A, B, D and E as prerequisites.
In this sense there is an **ordering** on the nodes of the graph.

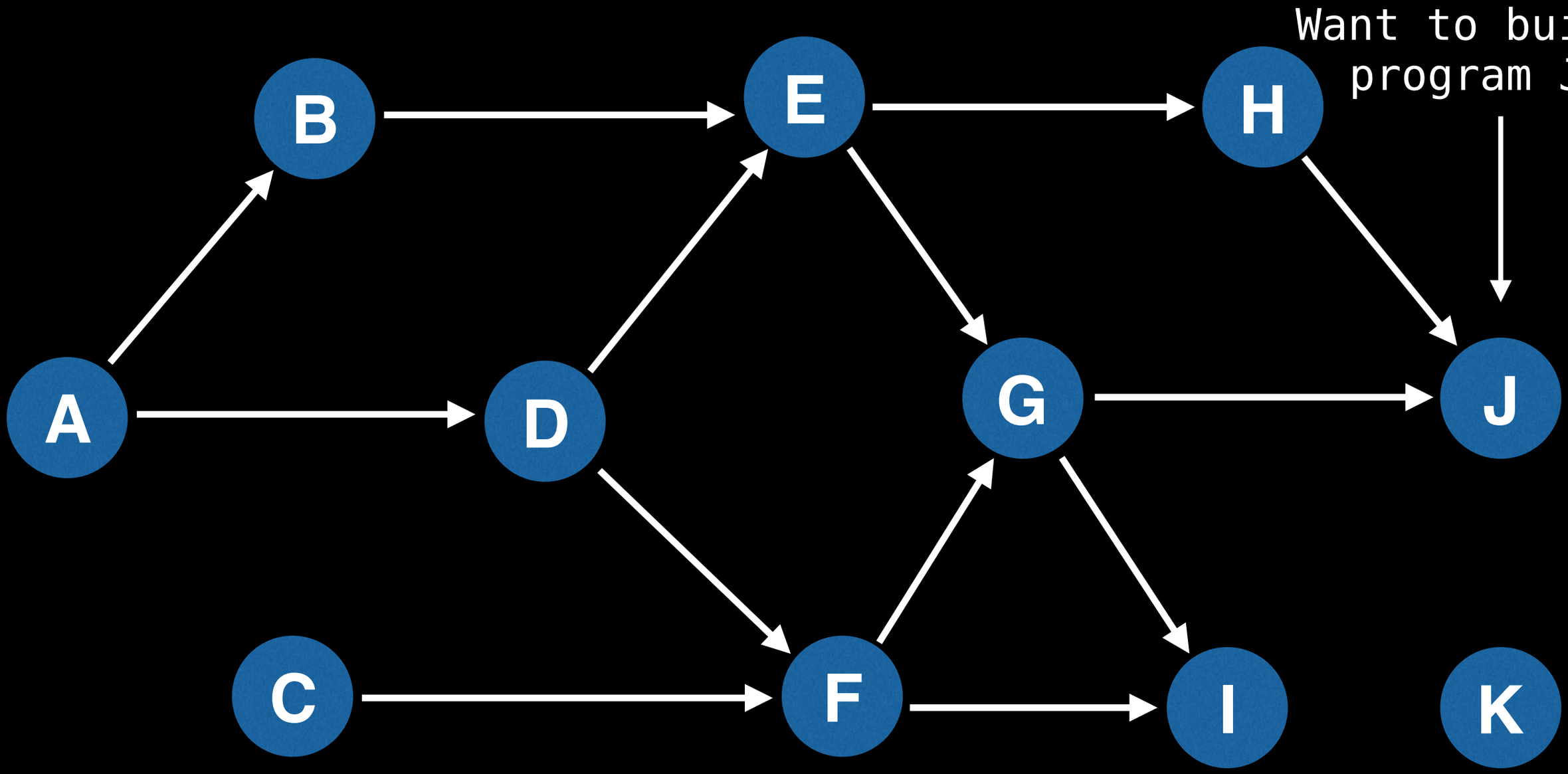


Suppose you're a student at university X and you want to take Class H, then you must take classes A, B, D and E as prerequisites.
In this sense there is an **ordering** on the nodes of the graph.

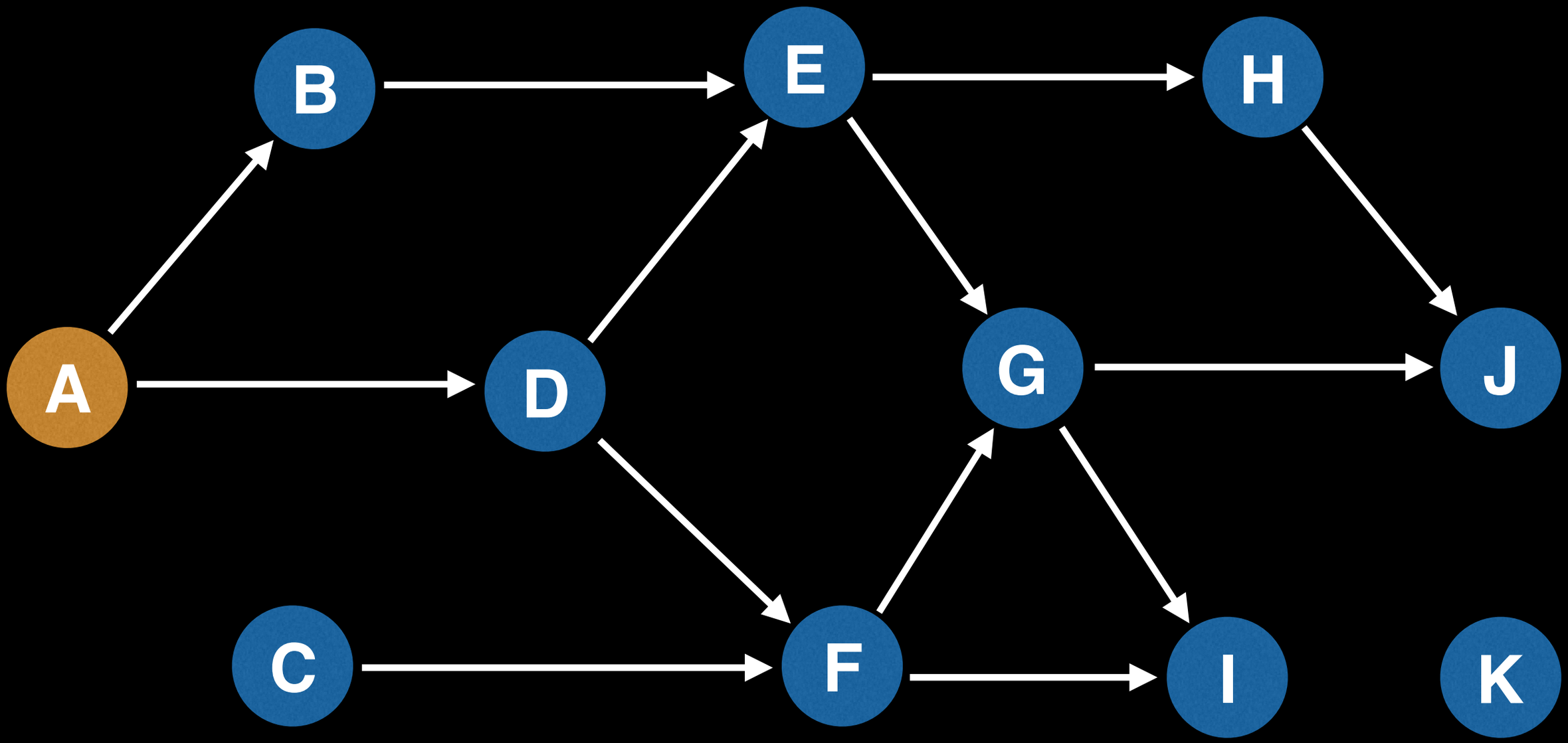


Another canonical example where an ordering on the nodes of the graph matters is for program build dependencies. A program cannot be built unless its dependencies are first built.

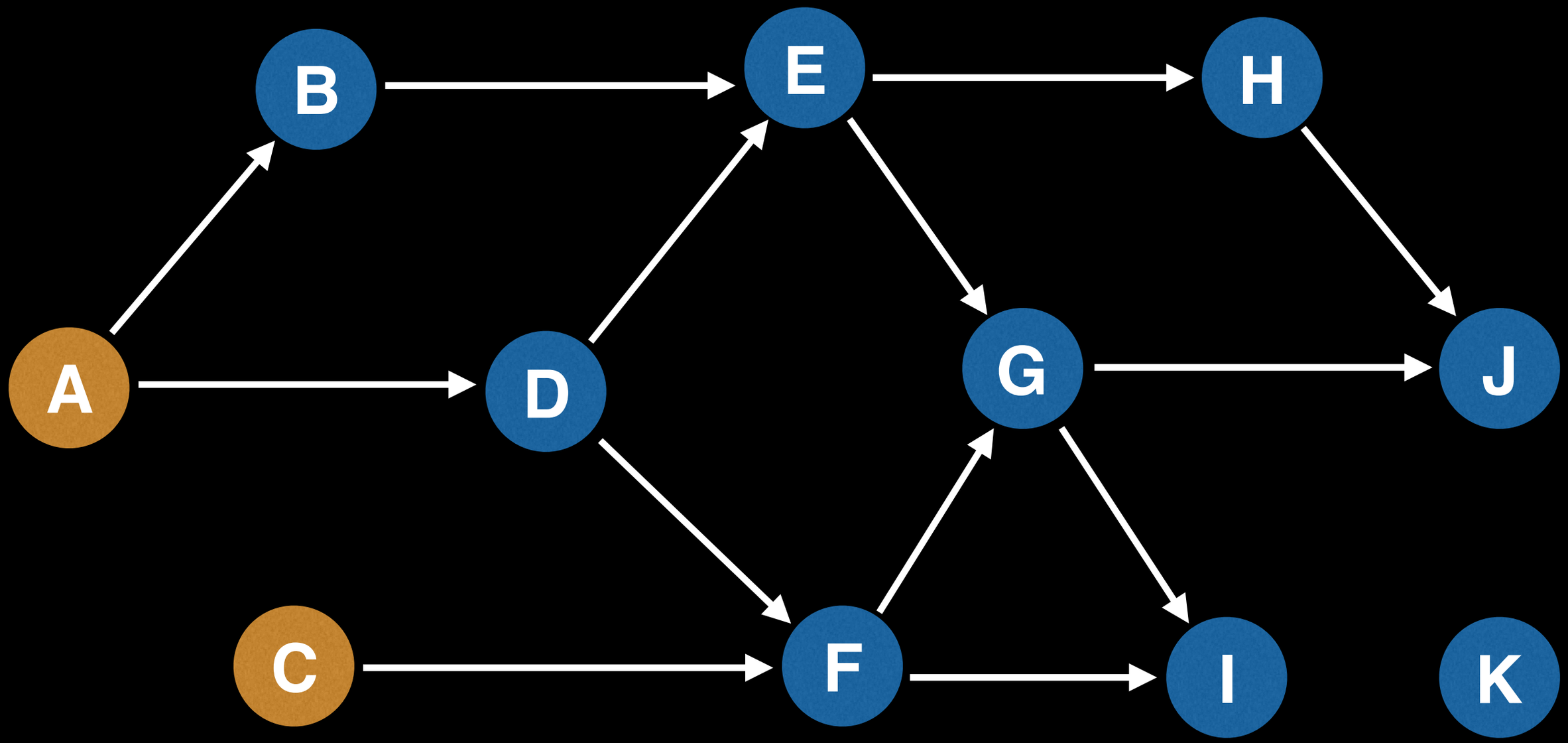
Another canonical example where an ordering on the nodes of the graph matters is for program build dependencies. A program cannot be built unless its dependencies are first built.



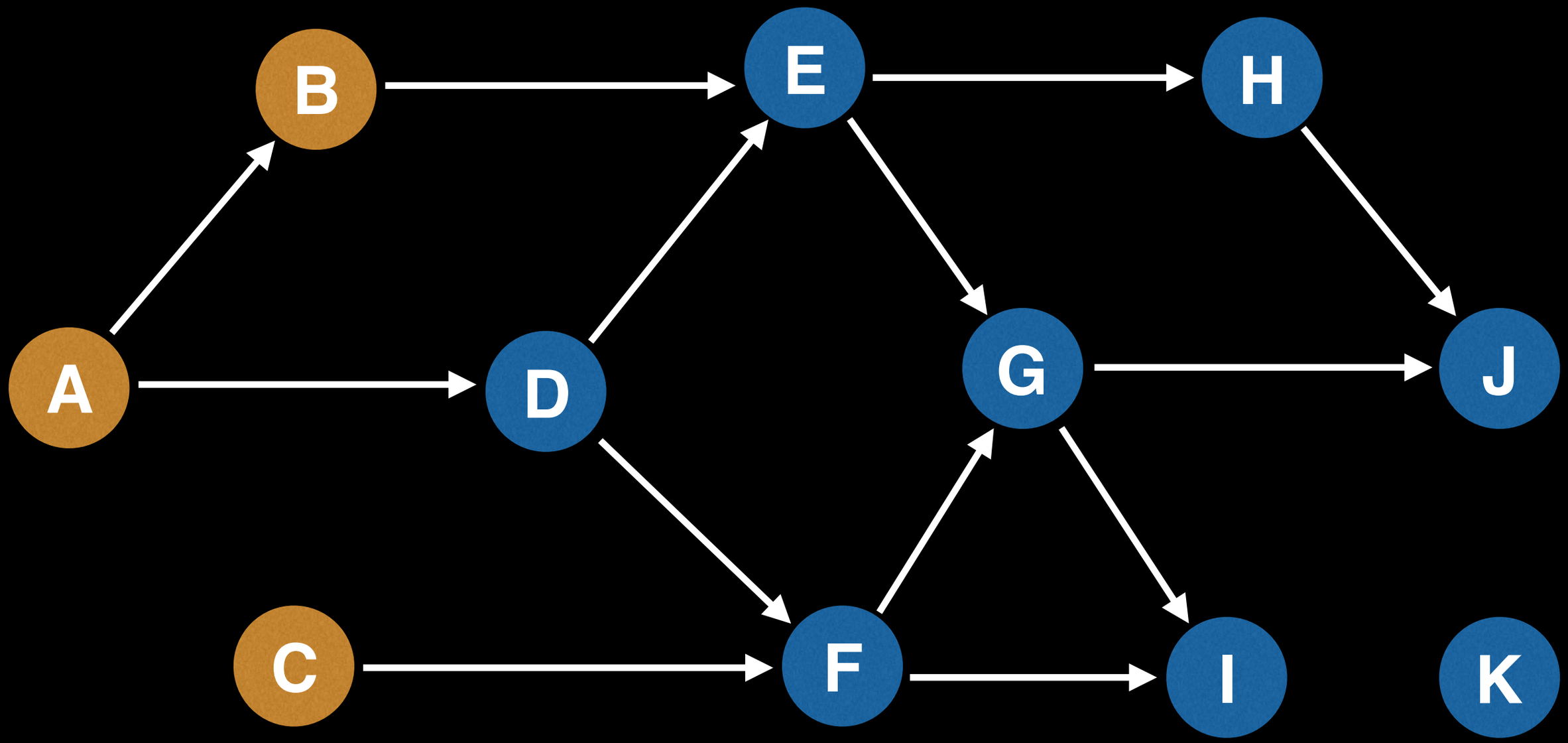
Another canonical example where an ordering on the nodes of the graph matters is for program build dependencies. A program cannot be built unless its dependencies are first built.



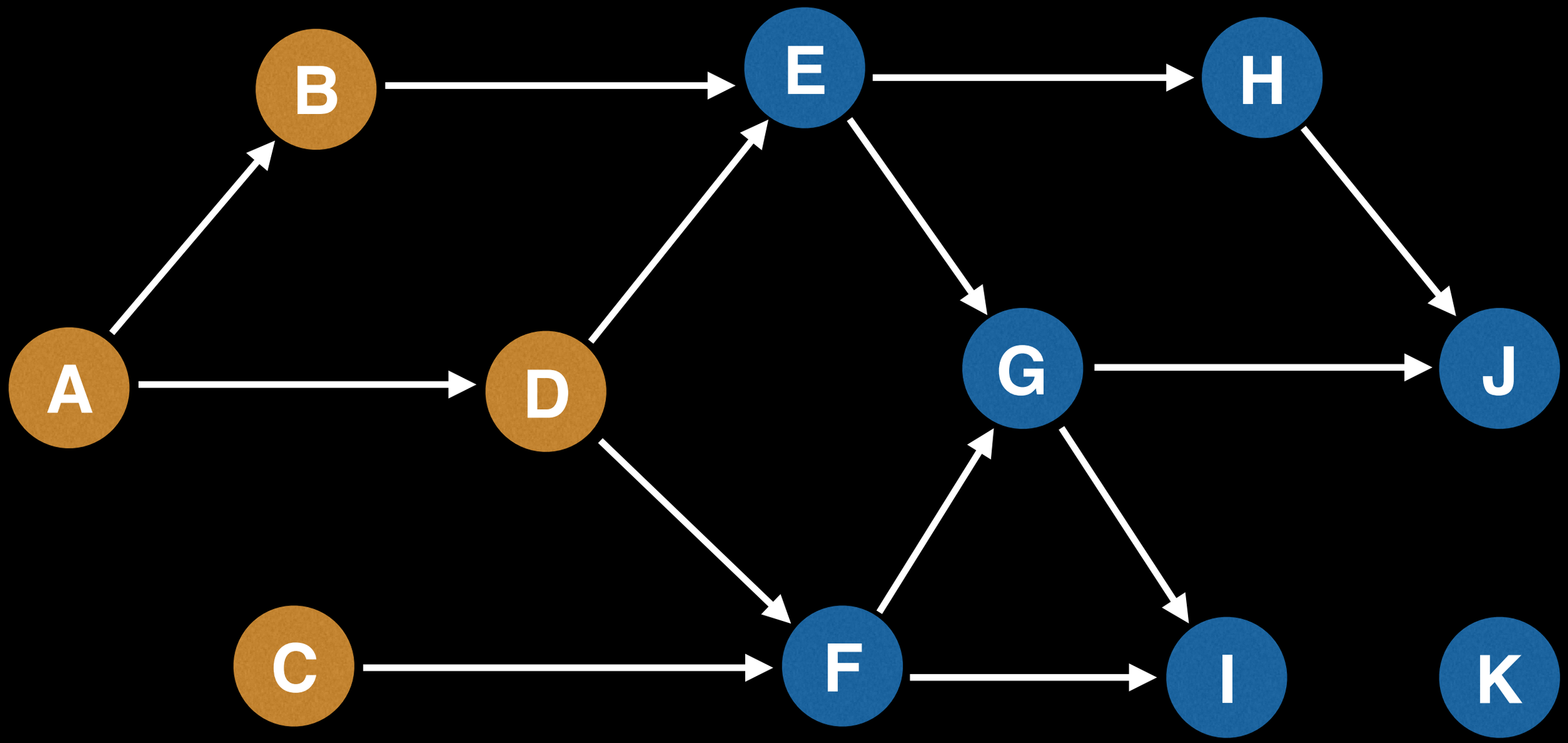
Another canonical example where an ordering on the nodes of the graph matters is for program build dependencies. A program cannot be built unless its dependencies are first built.



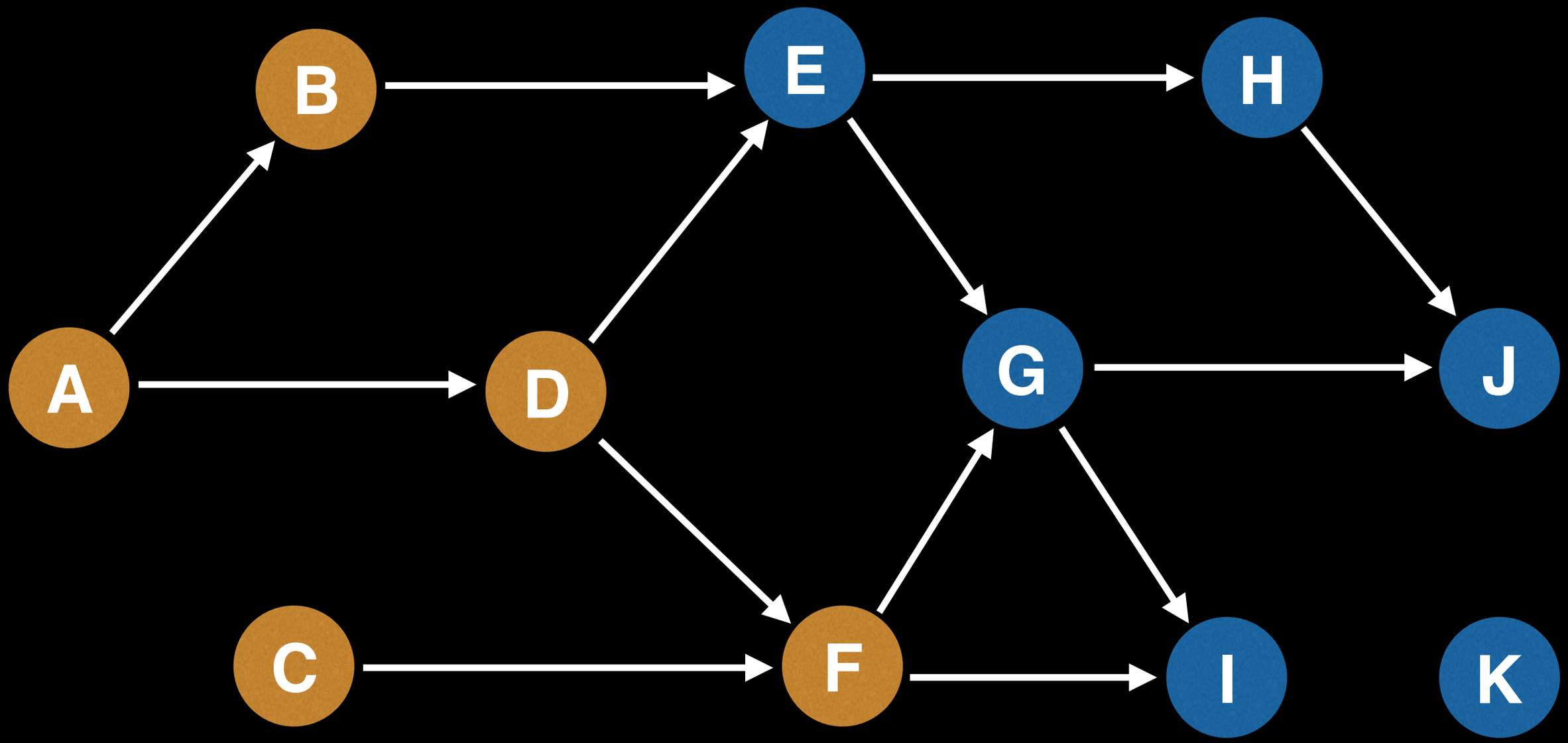
Another canonical example where an ordering on the nodes of the graph matters is for program build dependencies. A program cannot be built unless its dependencies are first built.



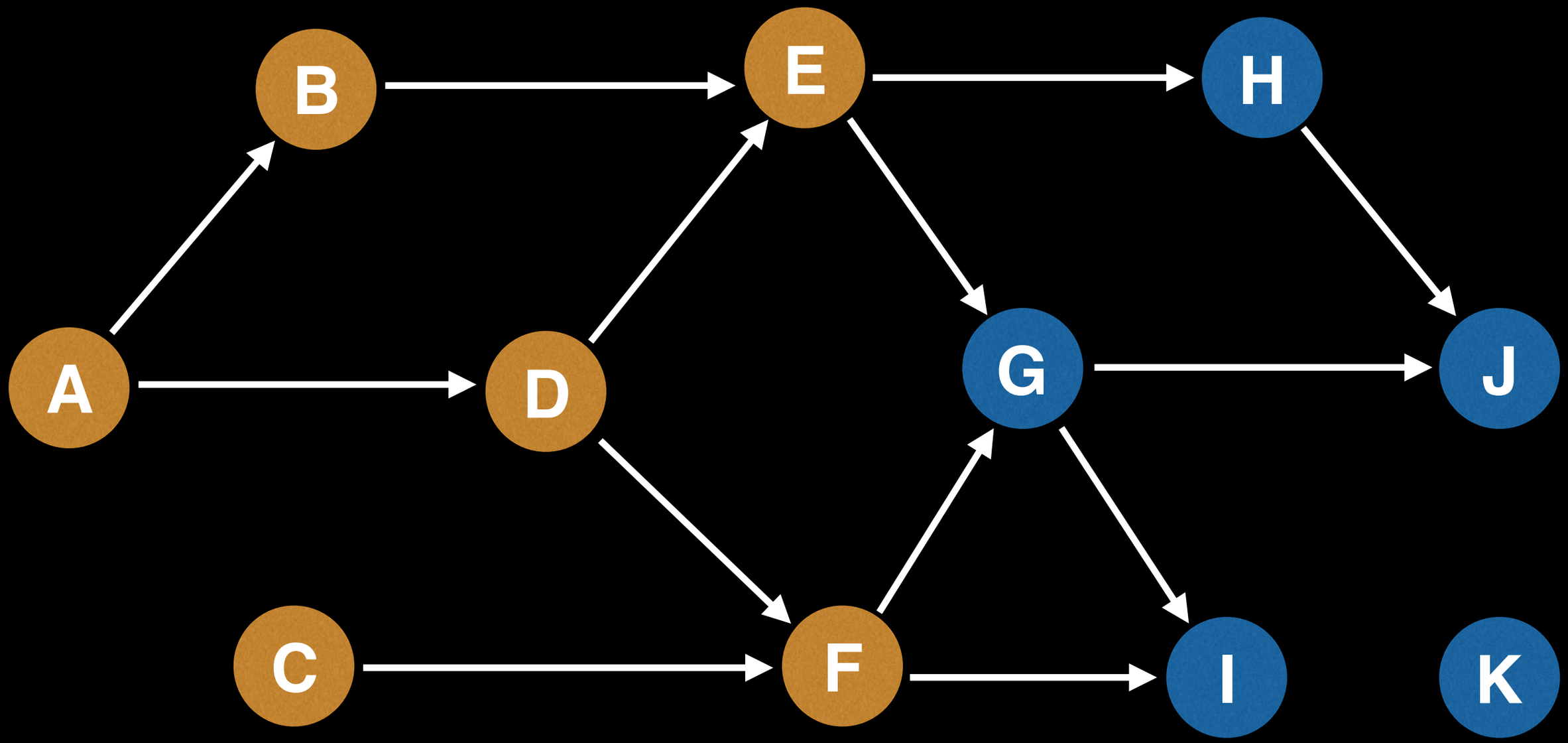
Another canonical example where an ordering on the nodes of the graph matters is for program build dependencies. A program cannot be built unless its dependencies are first built.



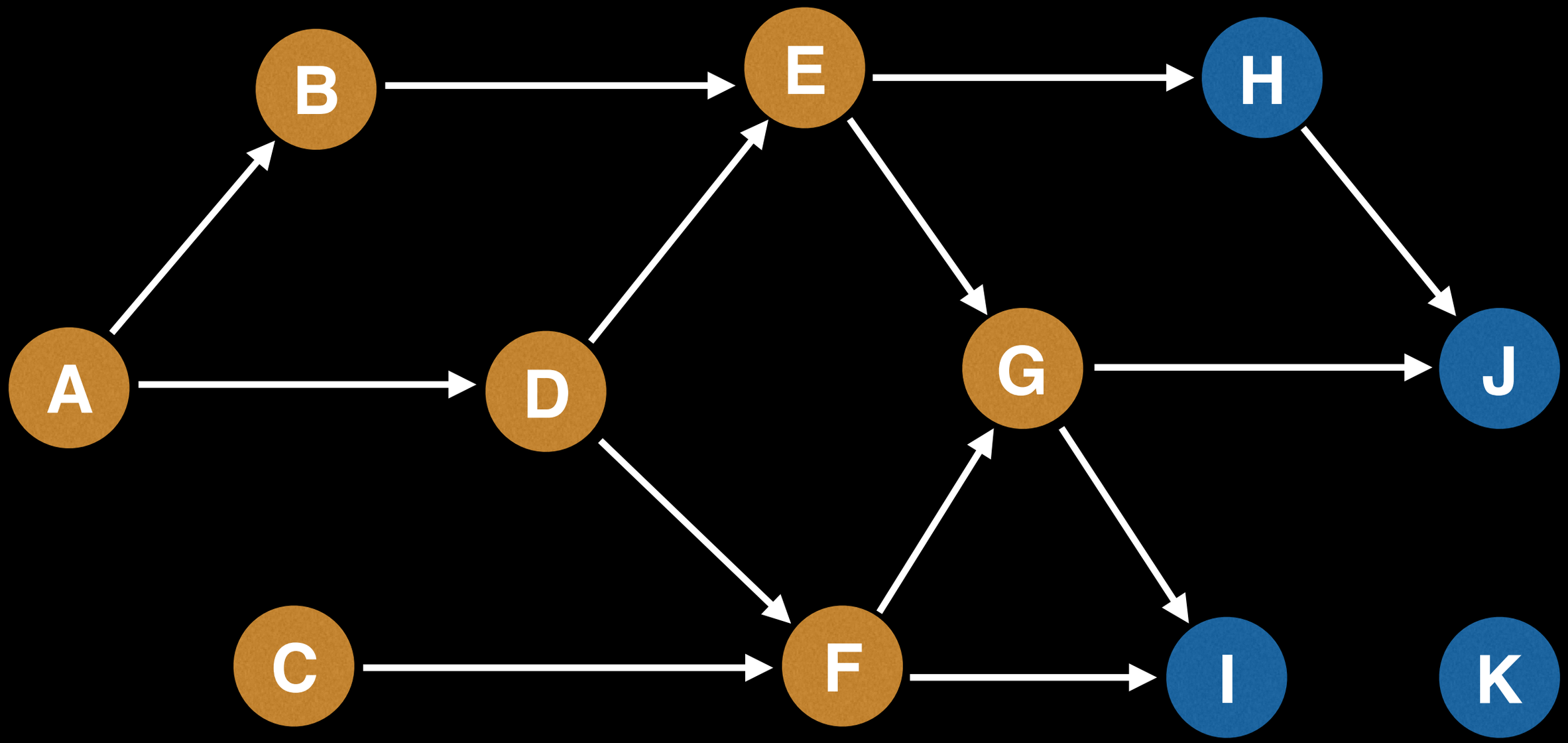
Another canonical example where an ordering on the nodes of the graph matters is for program build dependencies. A program cannot be built unless its dependencies are first built.



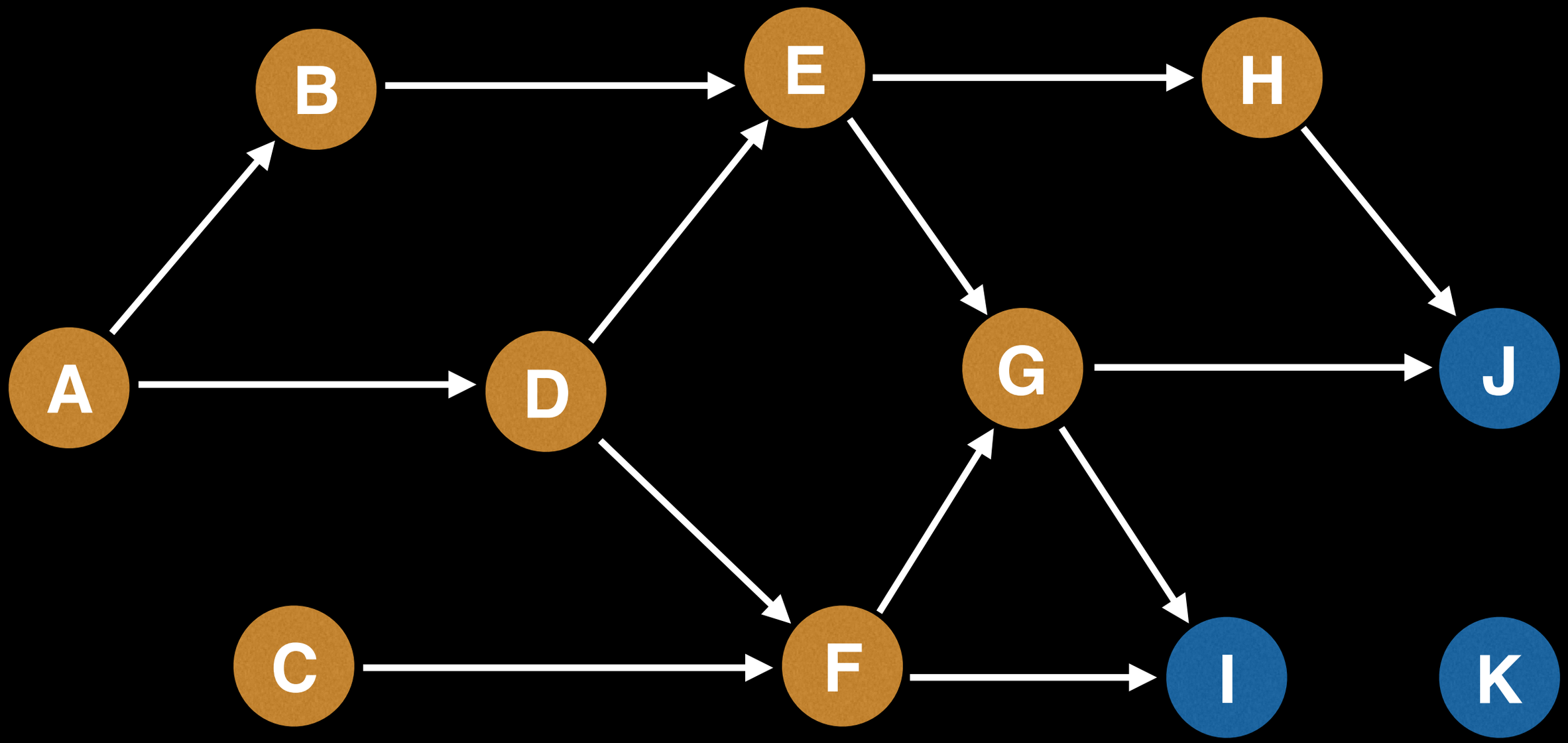
Another canonical example where an ordering on the nodes of the graph matters is for program build dependencies. A program cannot be built unless its dependencies are first built.



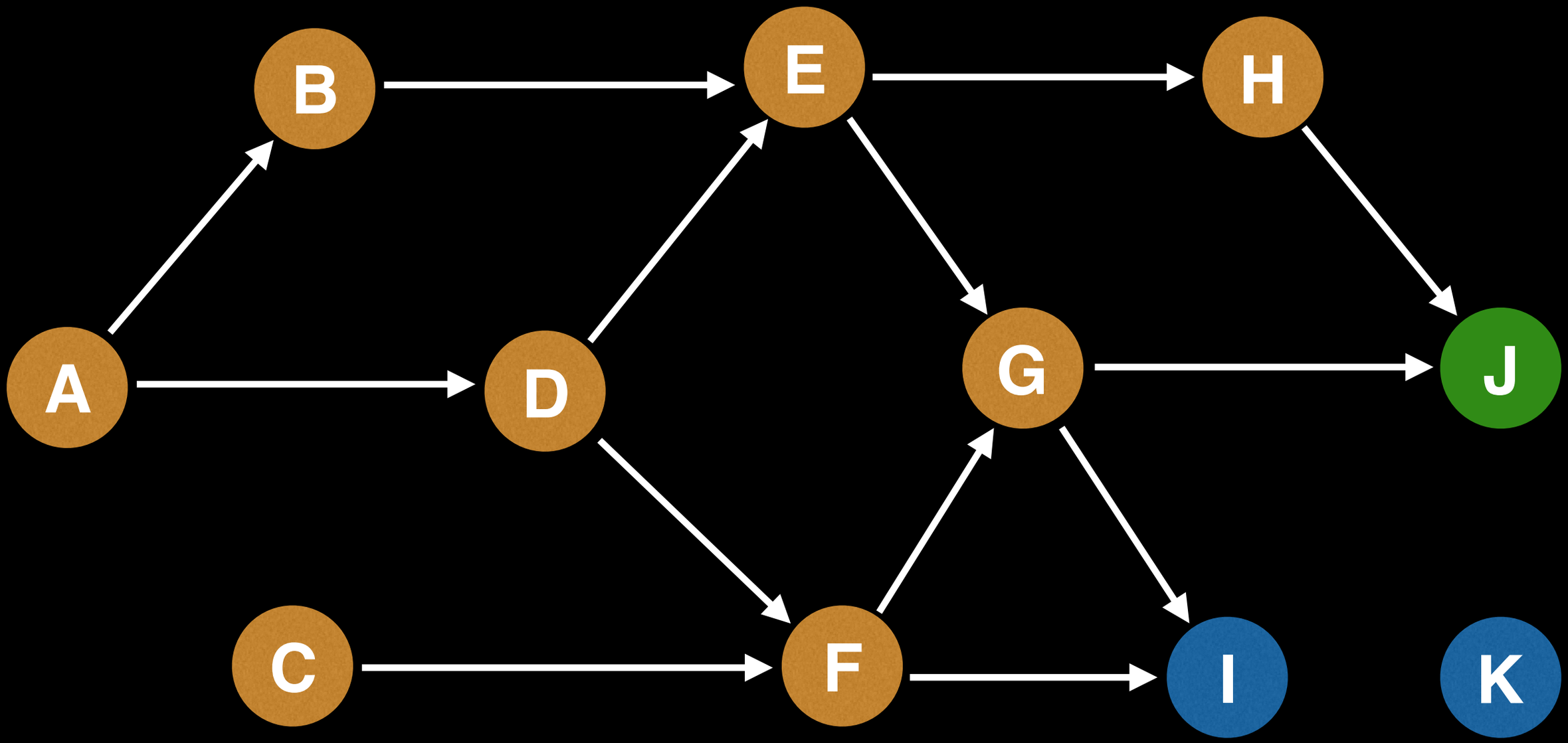
Another canonical example where an ordering on the nodes of the graph matters is for program build dependencies. A program cannot be built unless its dependencies are first built.



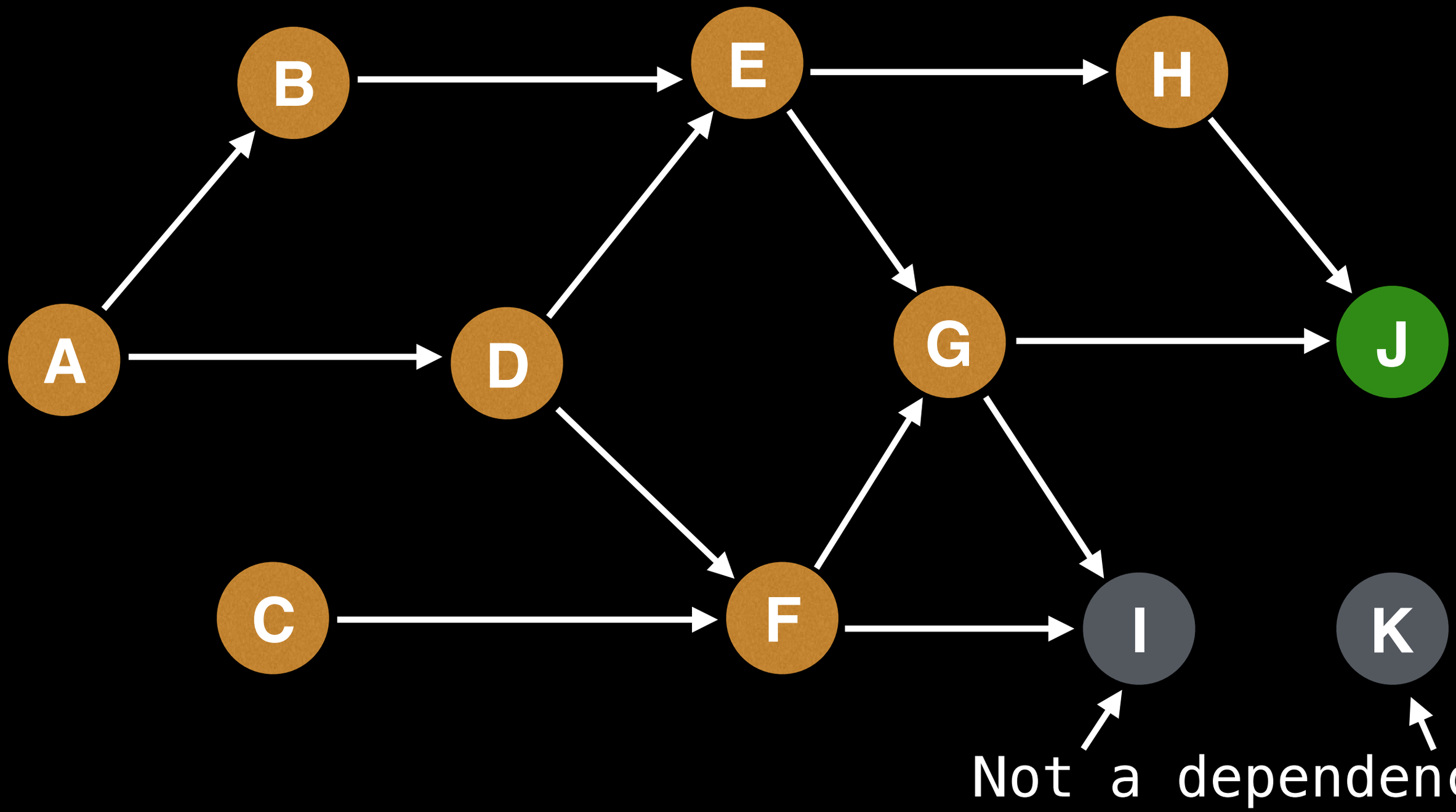
Another canonical example where an ordering on the nodes of the graph matters is for program build dependencies. A program cannot be built unless its dependencies are first built.

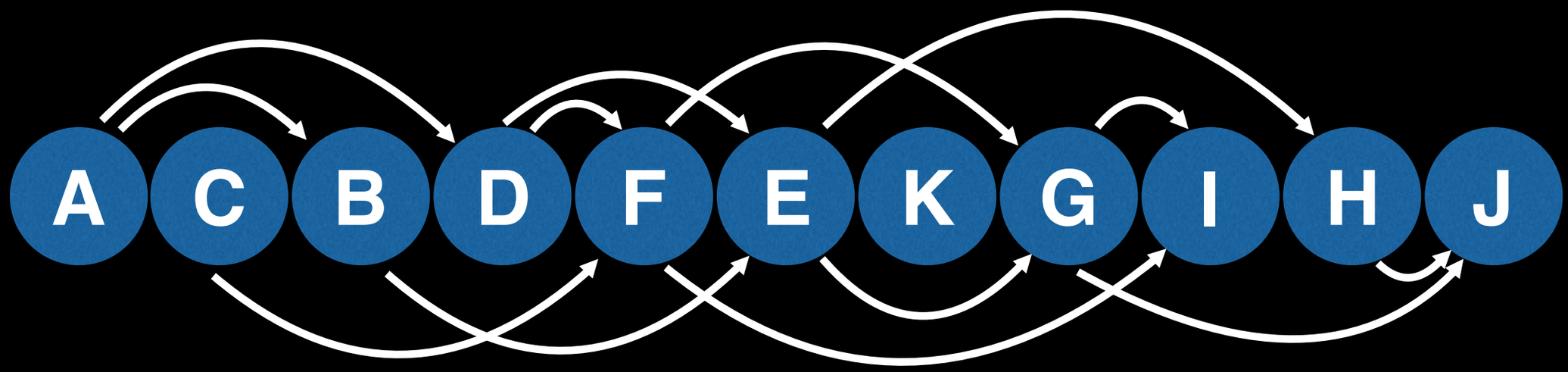


Another canonical example where an ordering on the nodes of the graph matters is for program build dependencies. A program cannot be built unless its dependencies are first built.



Another canonical example where an ordering on the nodes of the graph matters is for program build dependencies. A program cannot be built unless its dependencies are first built.





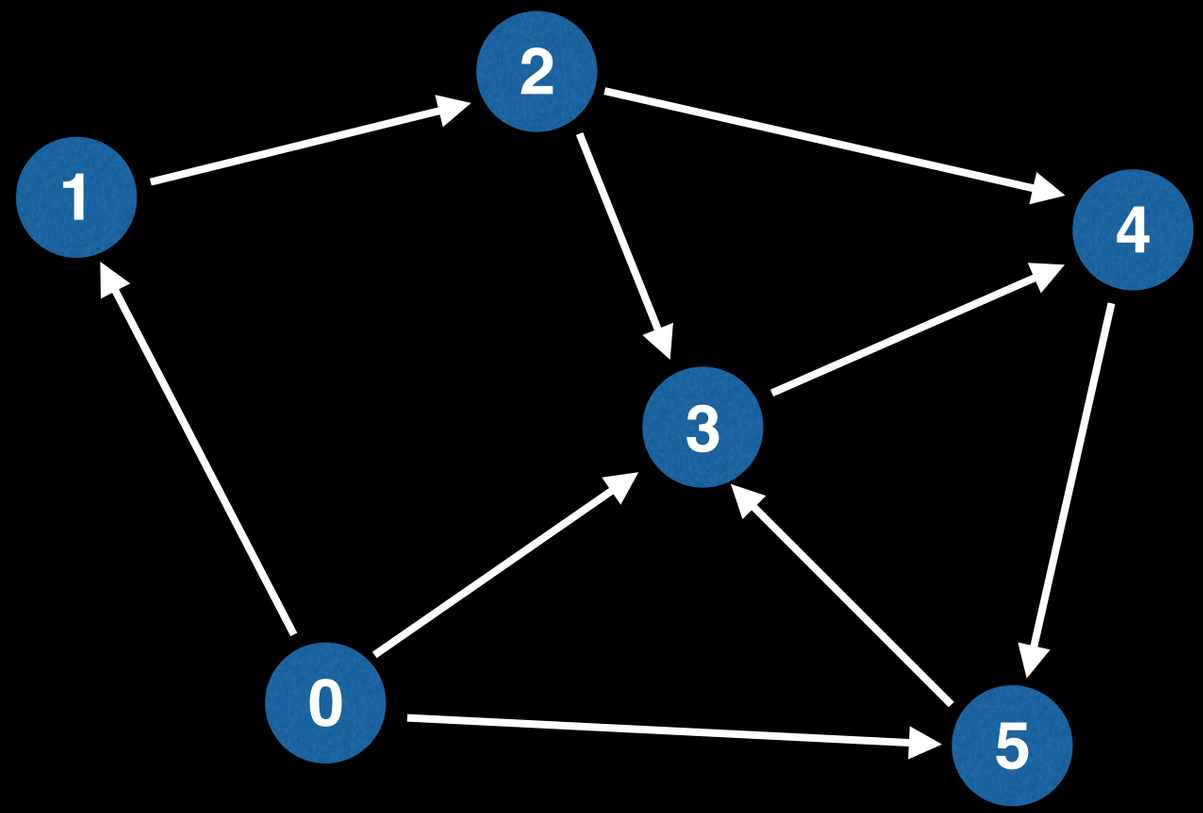
A **topological ordering** is an ordering of the nodes in a directed graph where for each directed edge from node A to node B, node A appears before node B in the ordering.

The **topological sort** algorithm can find a topological ordering in **O(V+E)** time!

NOTE: Topological orderings are NOT unique.

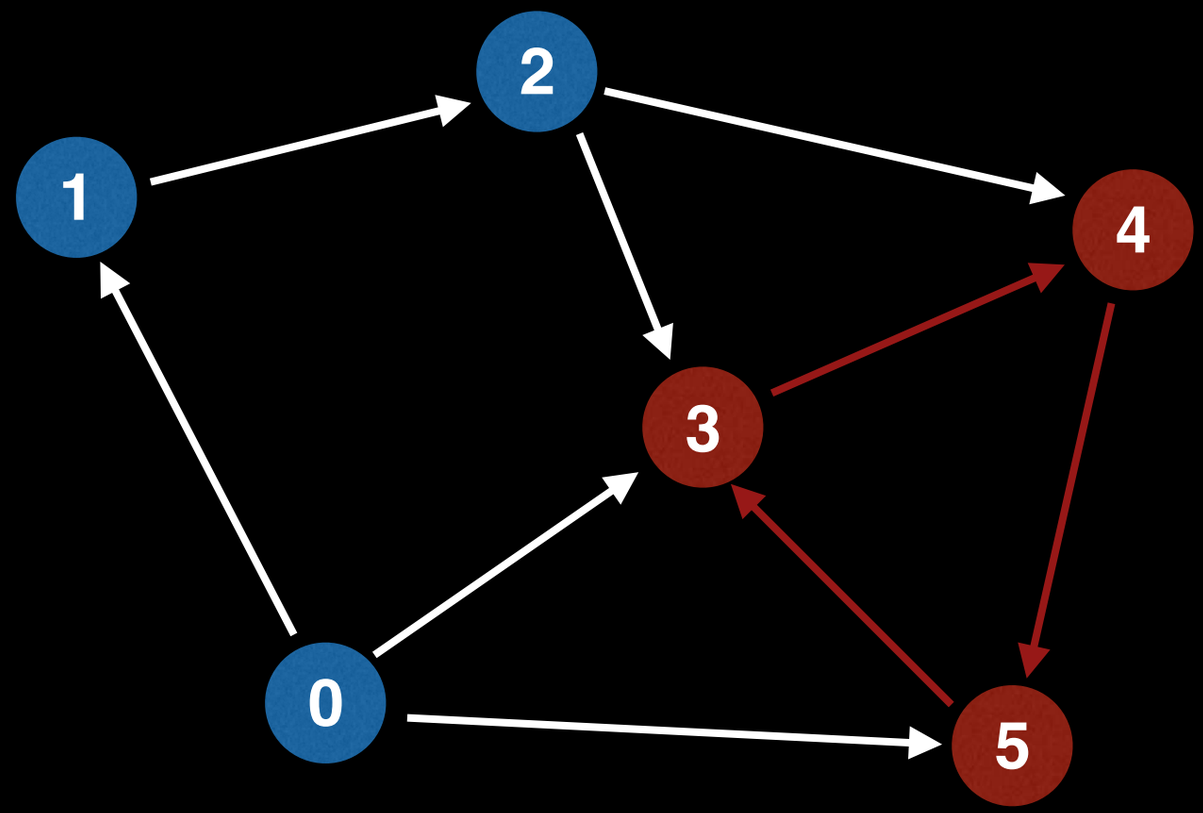
Directed Acyclic Graphs(DAG)

Not every graph can have a topological ordering. A graph which contains a **cycle** cannot have a valid ordering:



Directed Acyclic Graphs(DAG)

Not every graph can have a topological ordering. A graph which contains a **cycle** cannot have a valid ordering:



Directed Acyclic Graphs(DAG)

The only type of graph which has a valid topological ordering is a **Directed Acyclic Graph (DAG)**. These are graphs with directed edges and no cycles.

Directed Acyclic Graphs(DAG)

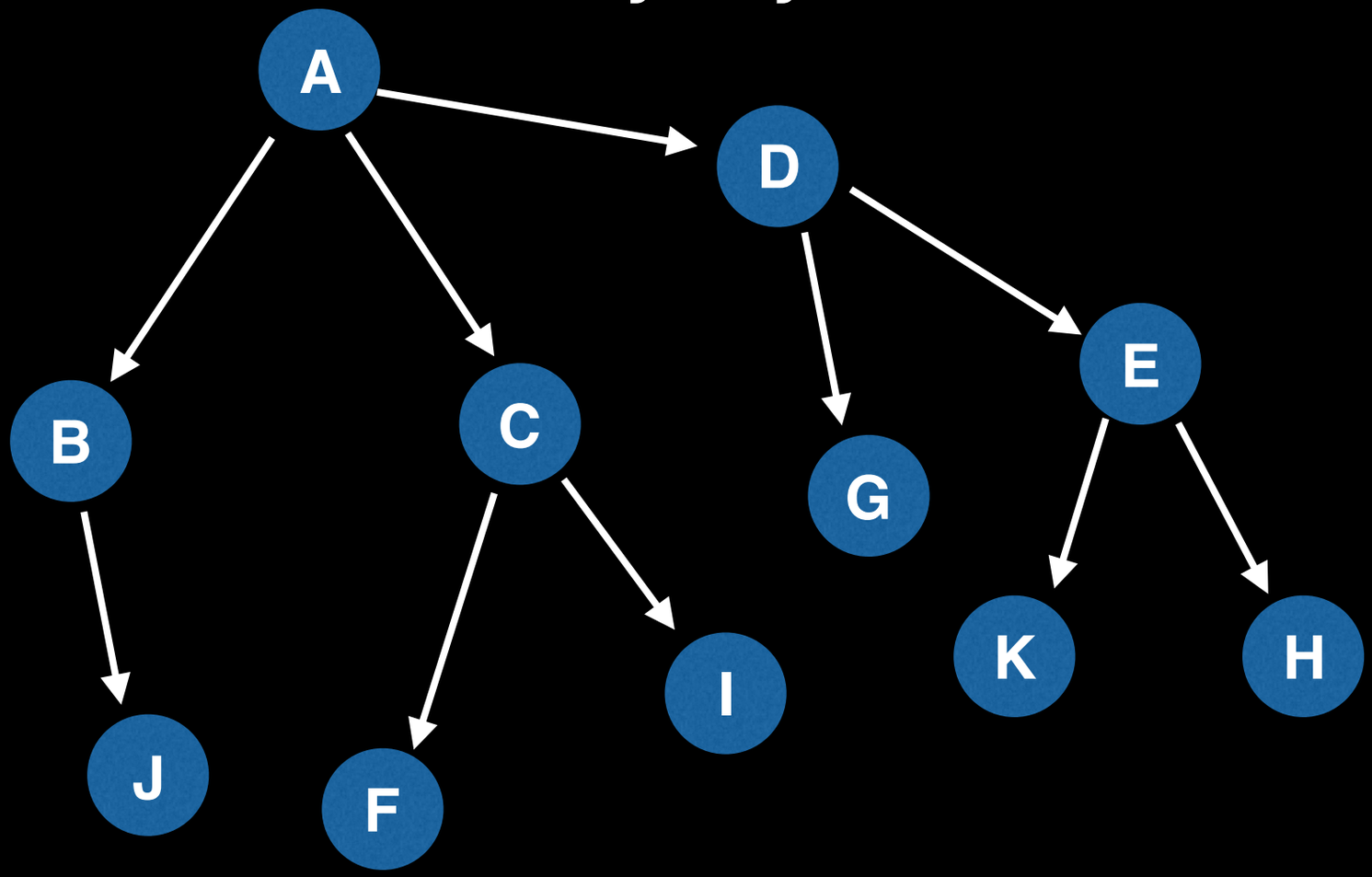
The only type of graph which has a valid topological ordering is a **Directed Acyclic Graph (DAG)**. These are graphs with directed edges and no cycles.

Q: How do I verify that my graph does not contain a directed cycle?

A: One method is to use Tarjan’s strongly connected component algorithm which can be used to find these cycles.

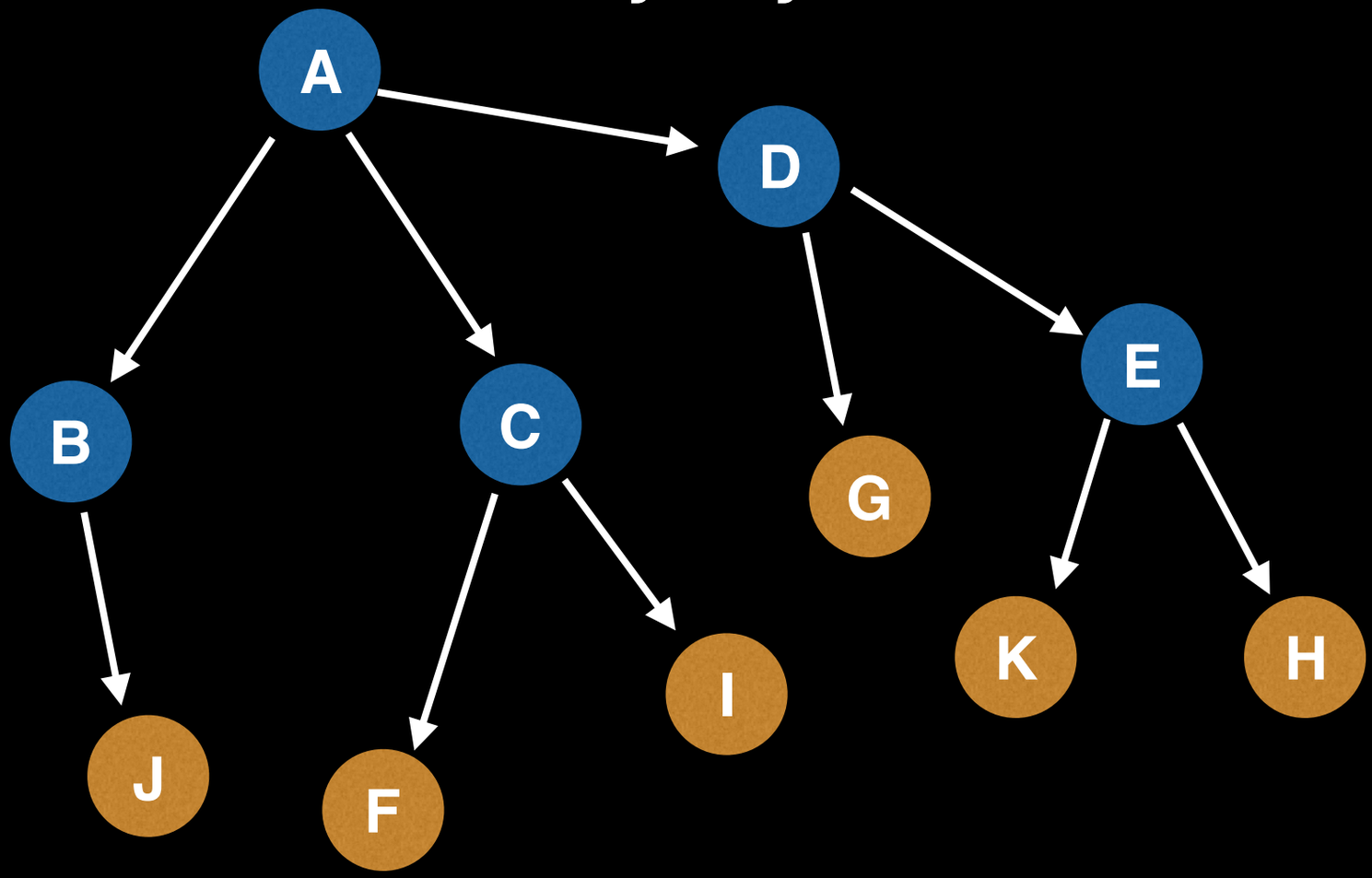
Directed Acyclic Graphs(DAG)

By definition, all rooted trees have a topological ordering since they do not contain any cycles.



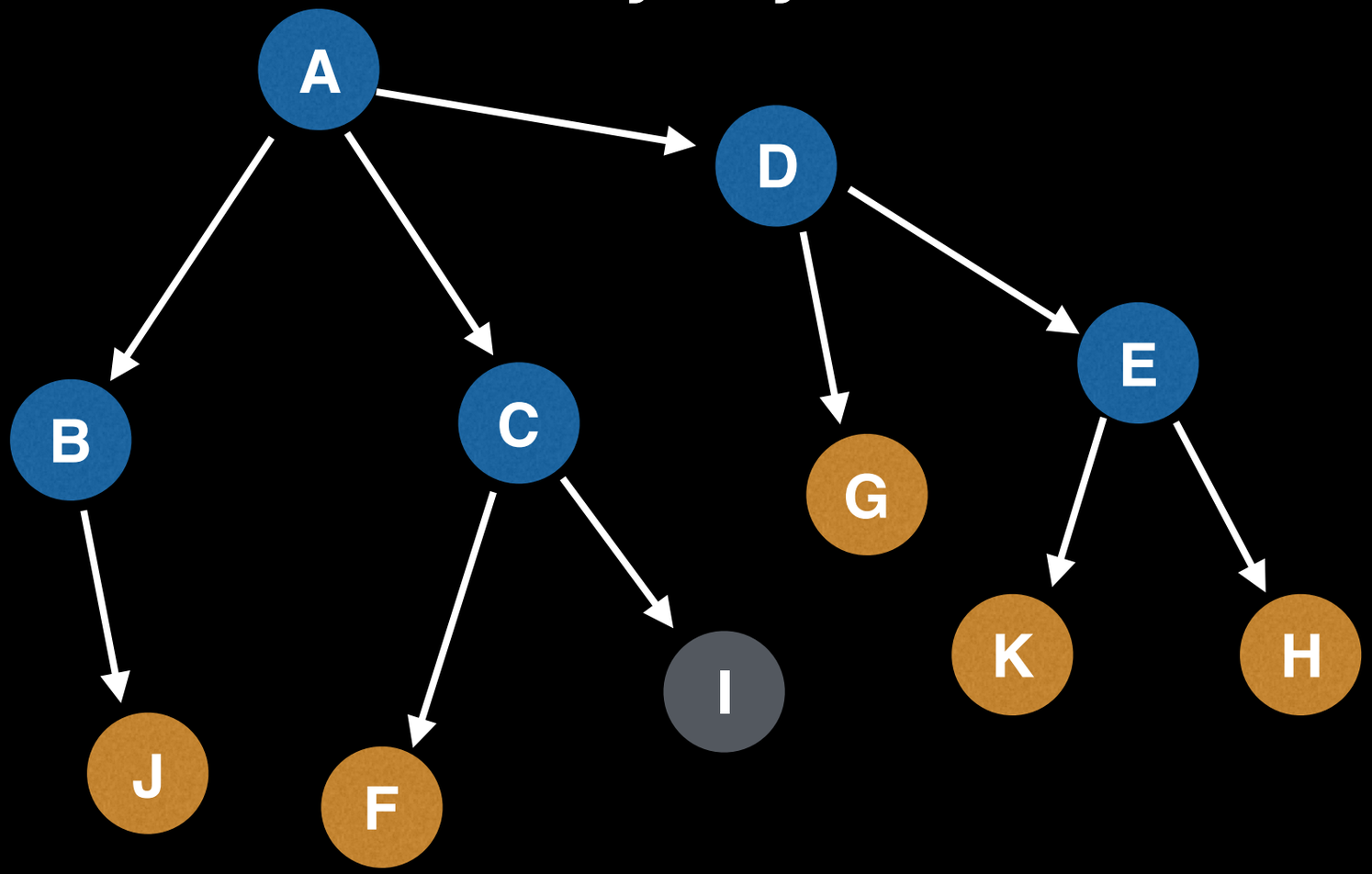
Directed Acyclic Graphs(DAG)

By definition, all rooted trees have a topological ordering since they do not contain any cycles.



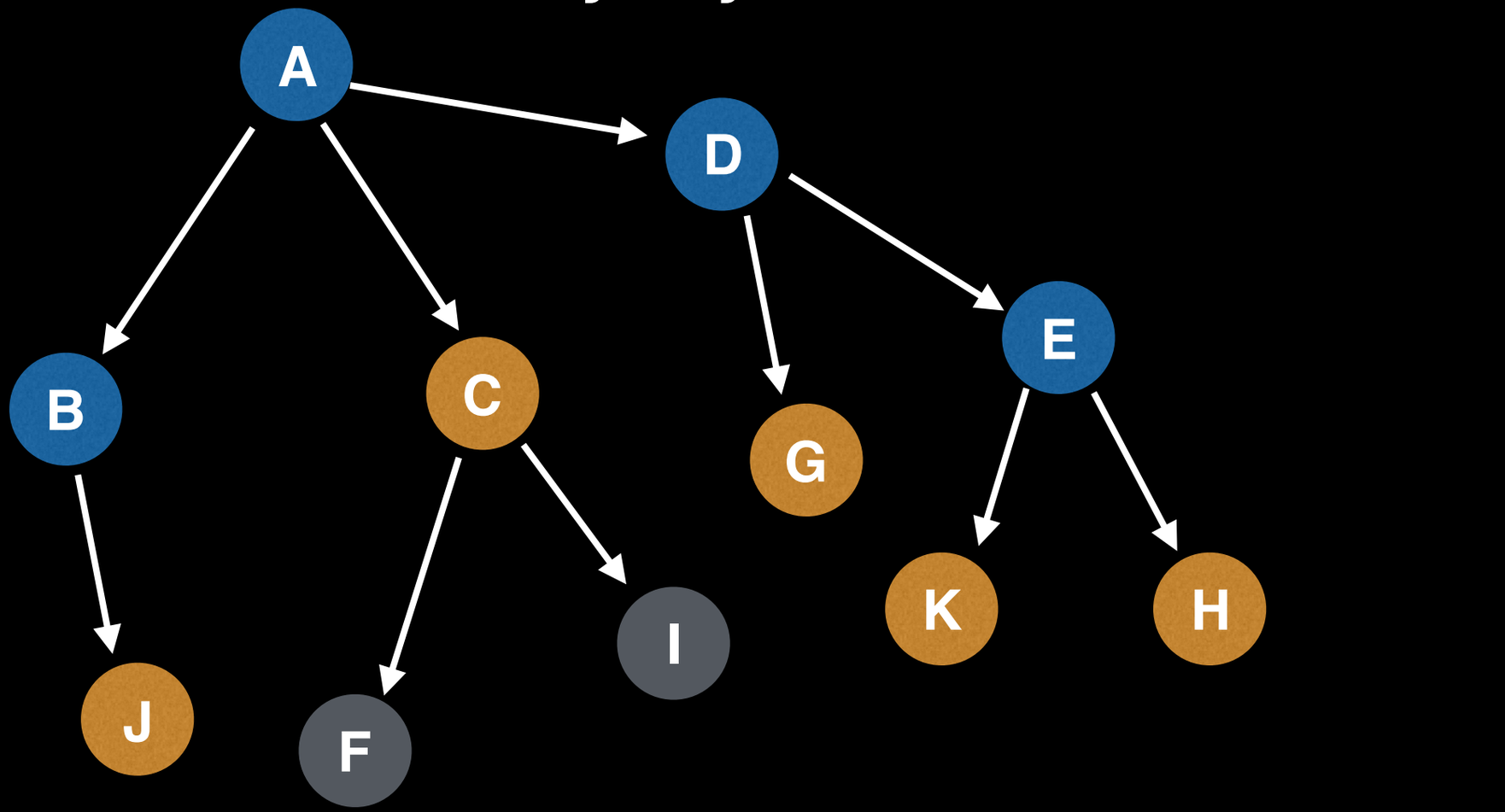
Directed Acyclic Graphs(DAG)

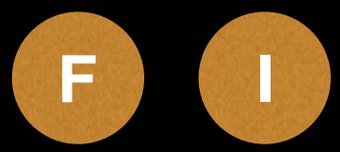
By definition, all rooted trees have a topological ordering since they do not contain any cycles.



Directed Acyclic Graphs(DAG)

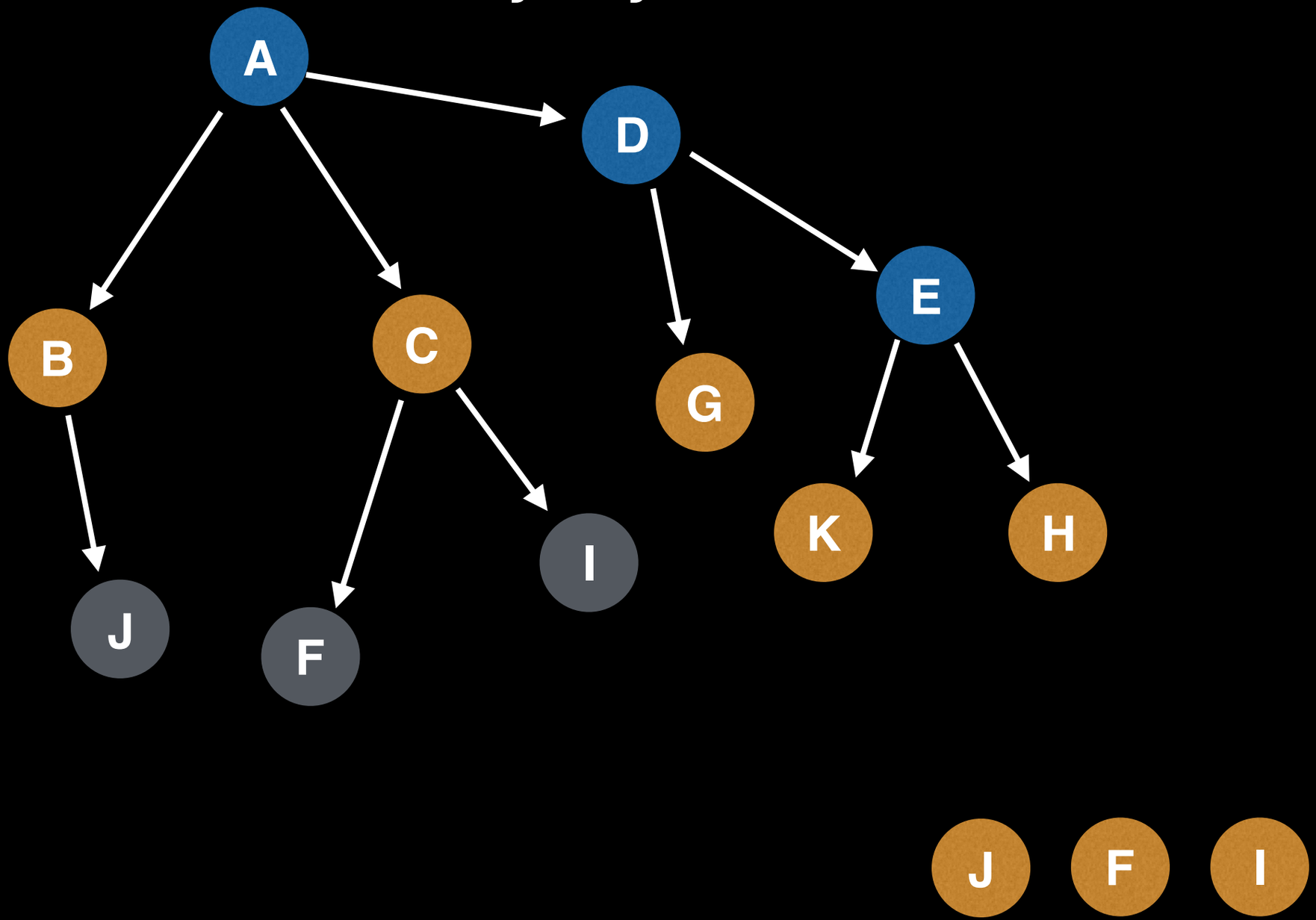
By definition, all rooted trees have a topological ordering since they do not contain any cycles.





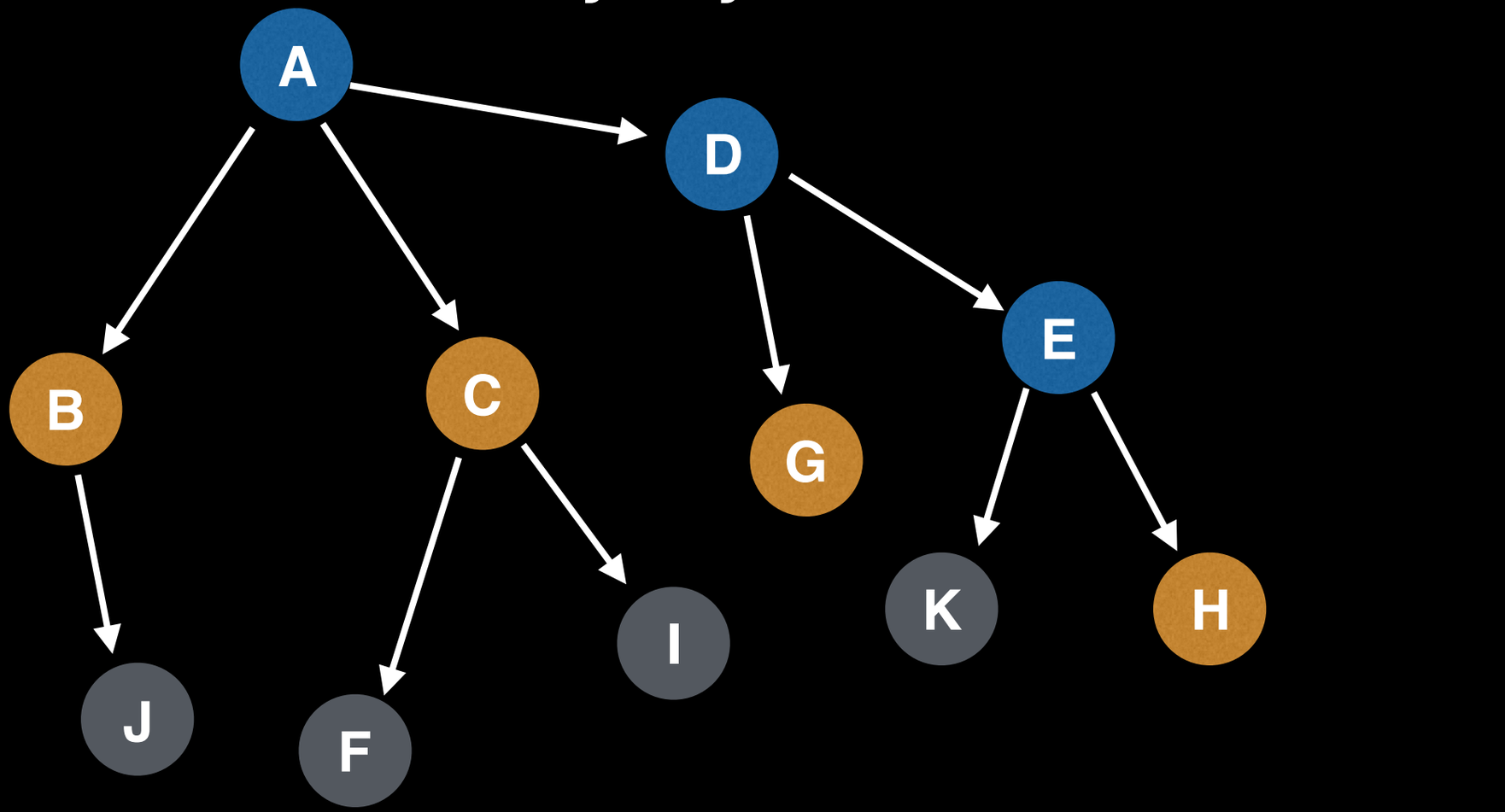
Directed Acyclic Graphs(DAG)

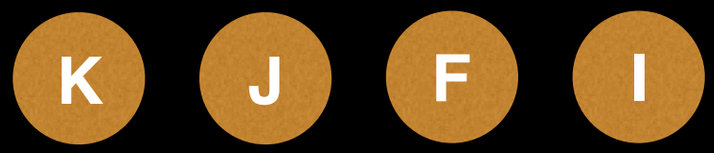
By definition, all rooted trees have a topological ordering since they do not contain any cycles.



Directed Acyclic Graphs(DAG)

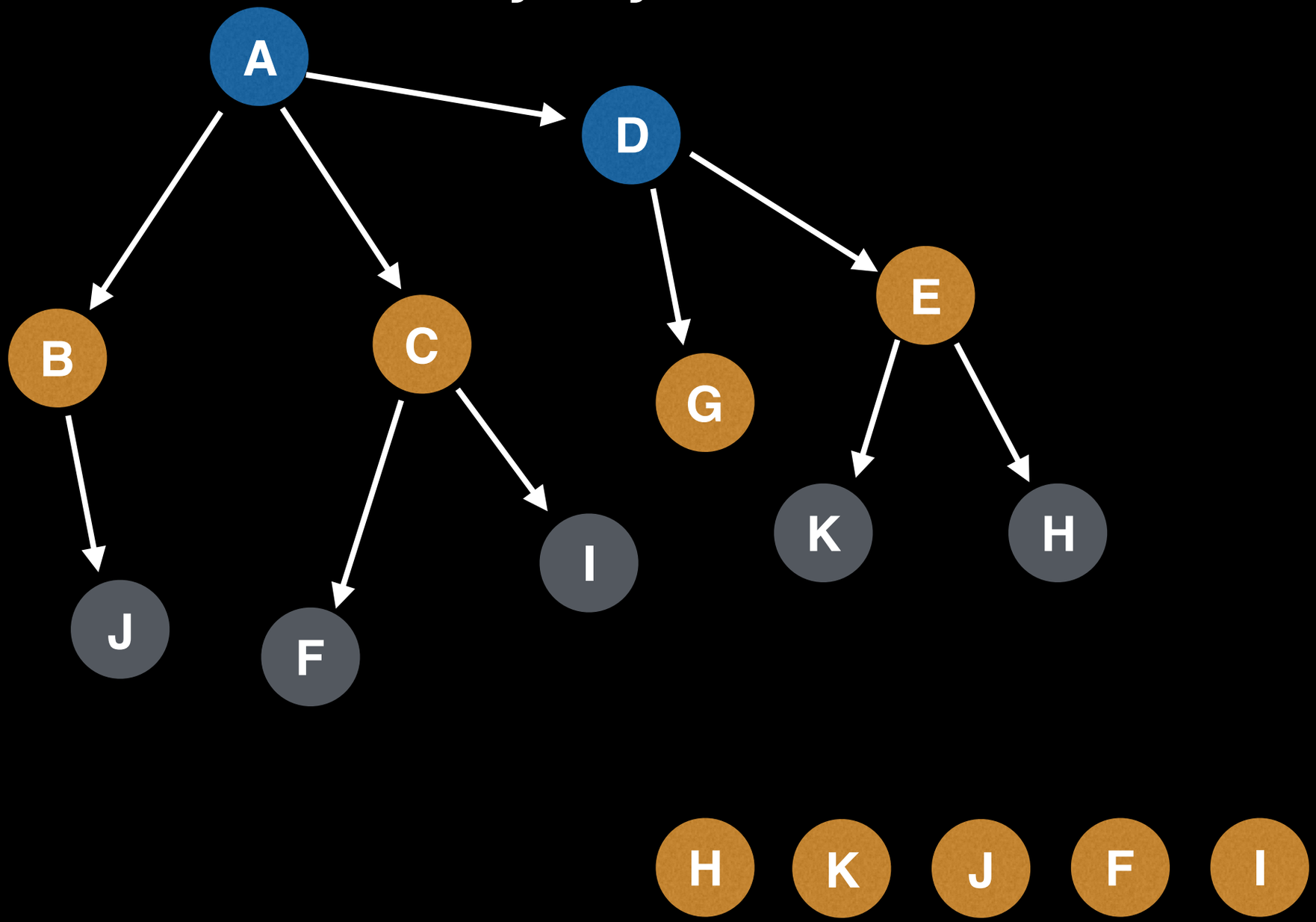
By definition, all rooted trees have a topological ordering since they do not contain any cycles.





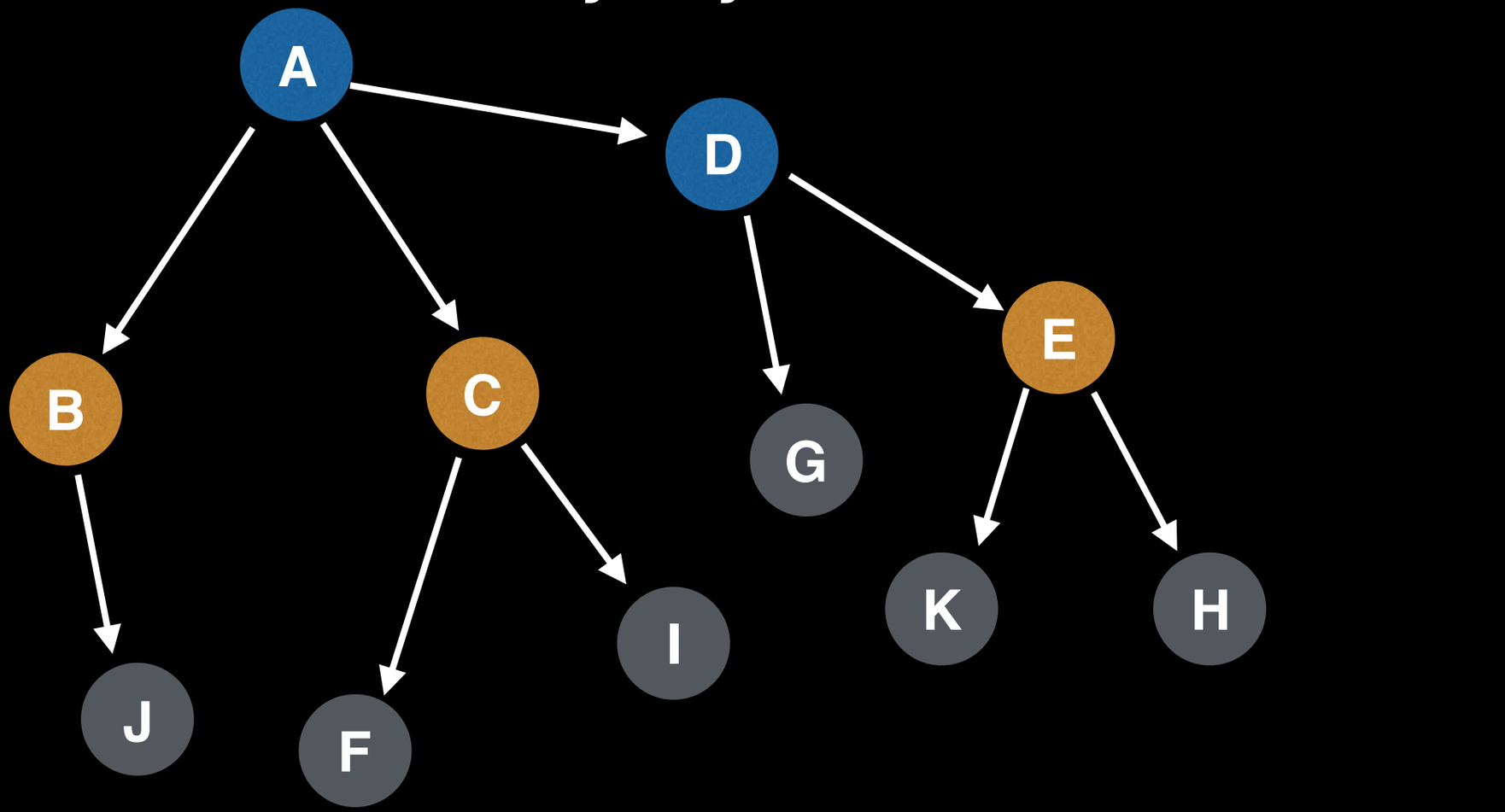
Directed Acyclic Graphs(DAG)

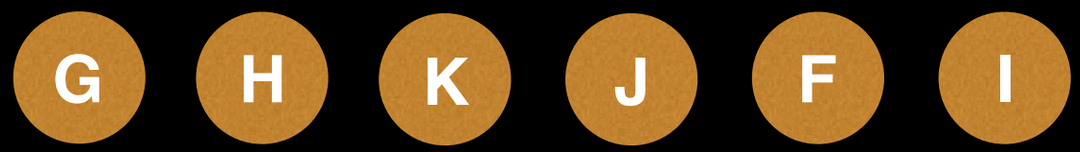
By definition, all rooted trees have a topological ordering since they do not contain any cycles.



Directed Acyclic Graphs(DAG)

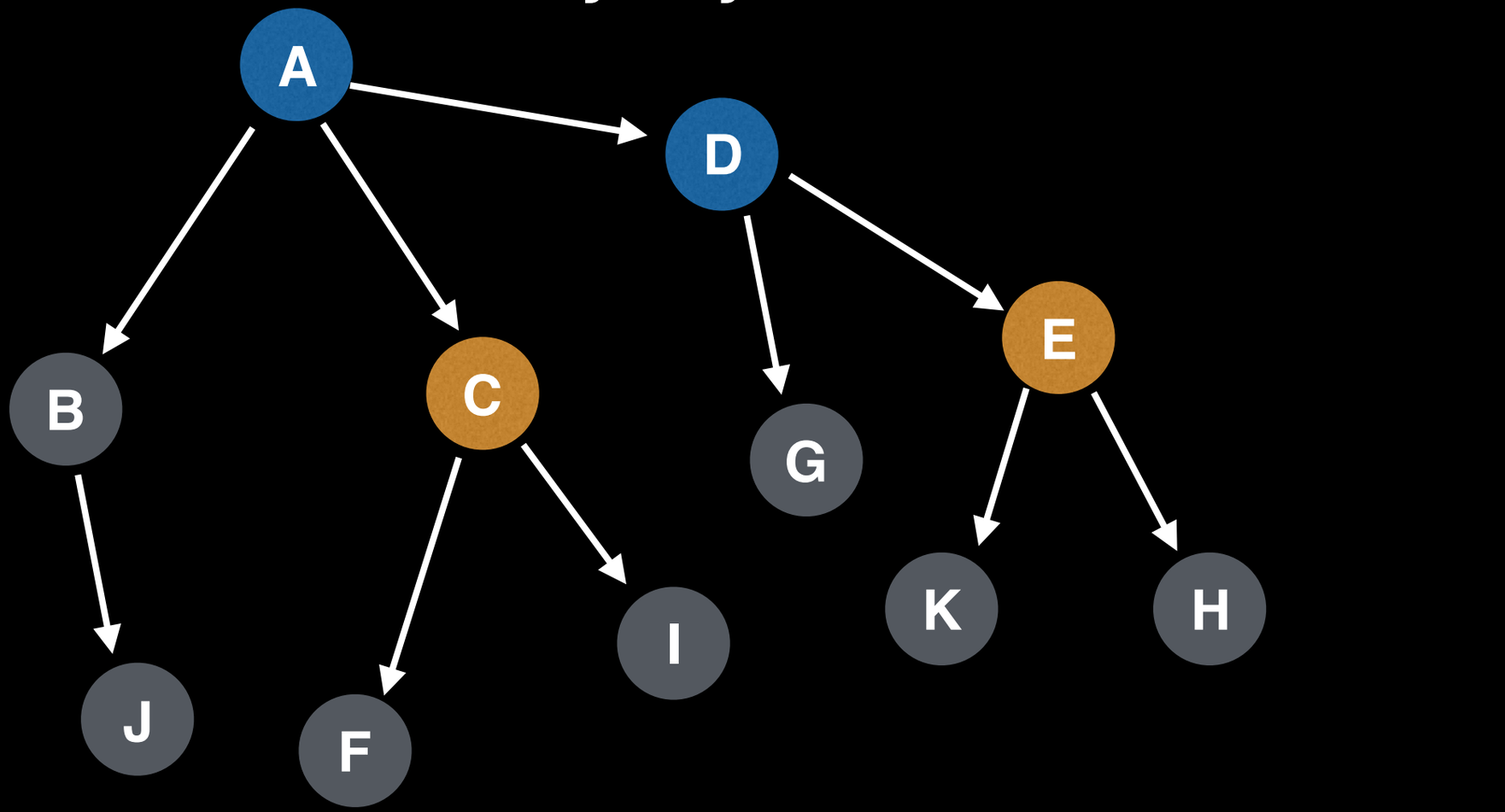
By definition, all rooted trees have a topological ordering since they do not contain any cycles.





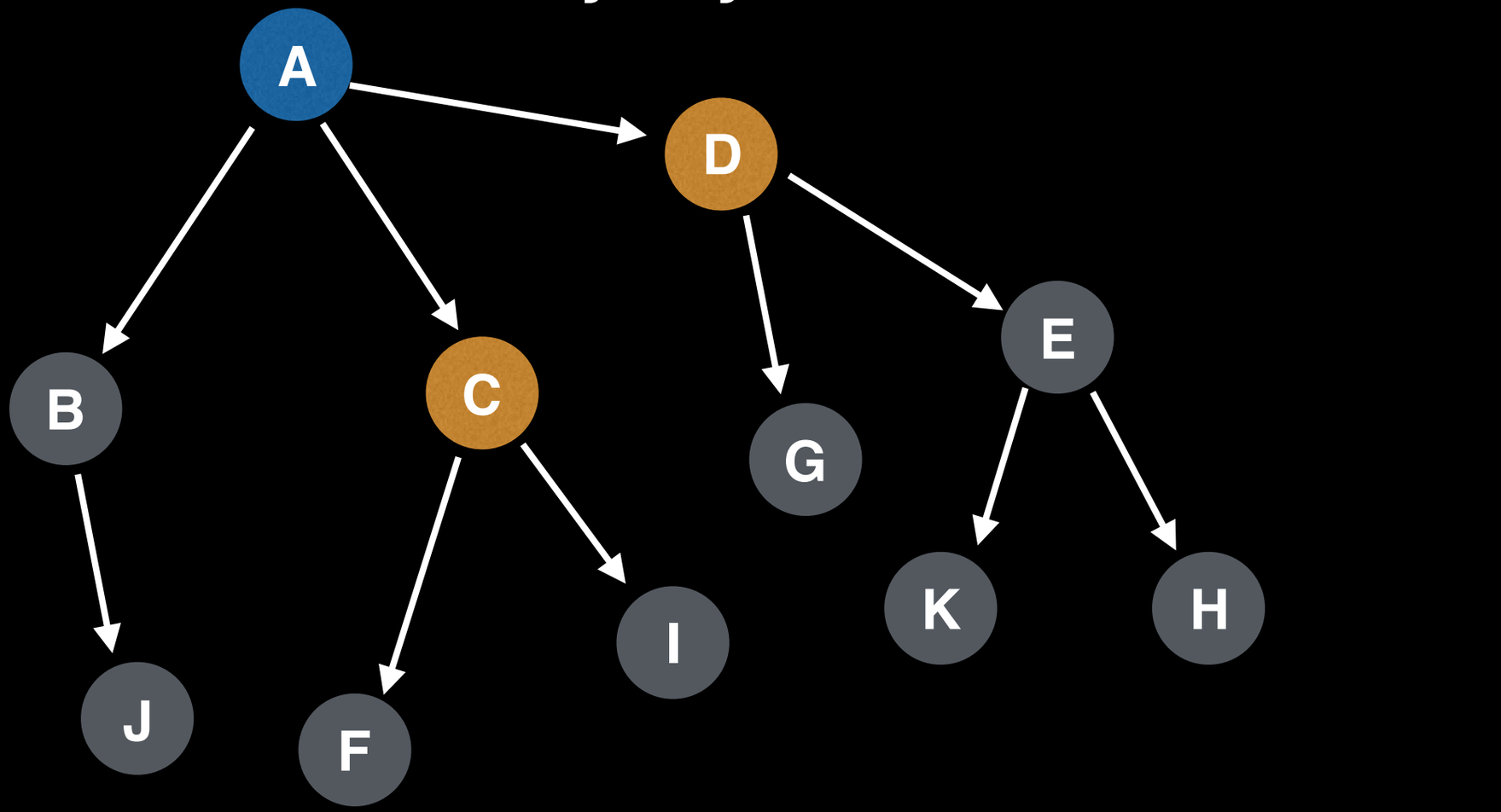
Directed Acyclic Graphs(DAG)

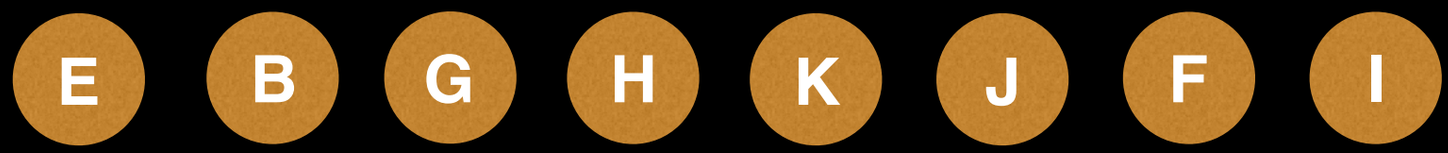
By definition, all rooted trees have a topological ordering since they do not contain any cycles.



Directed Acyclic Graphs(DAG)

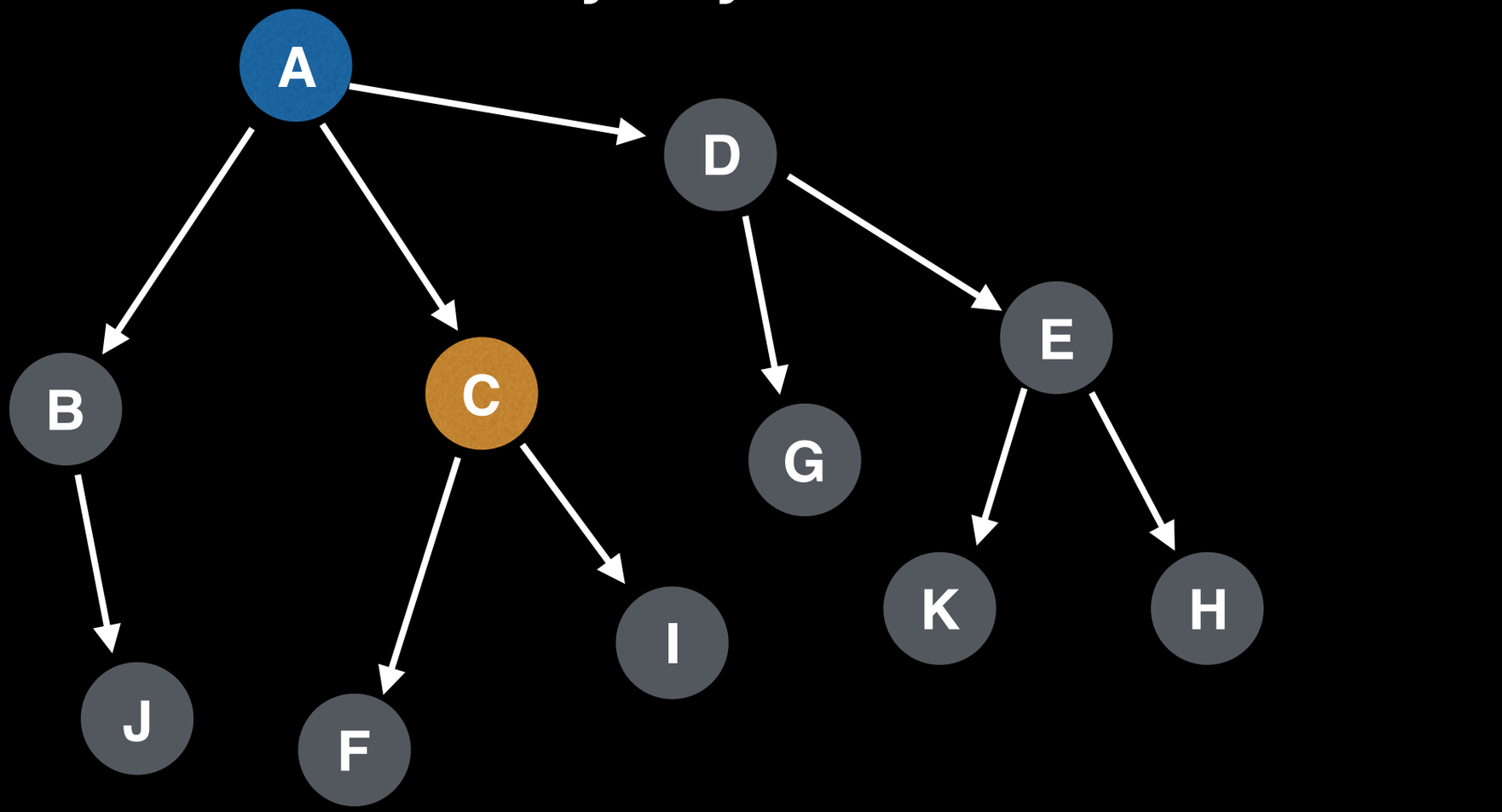
By definition, all rooted trees have a topological ordering since they do not contain any cycles.





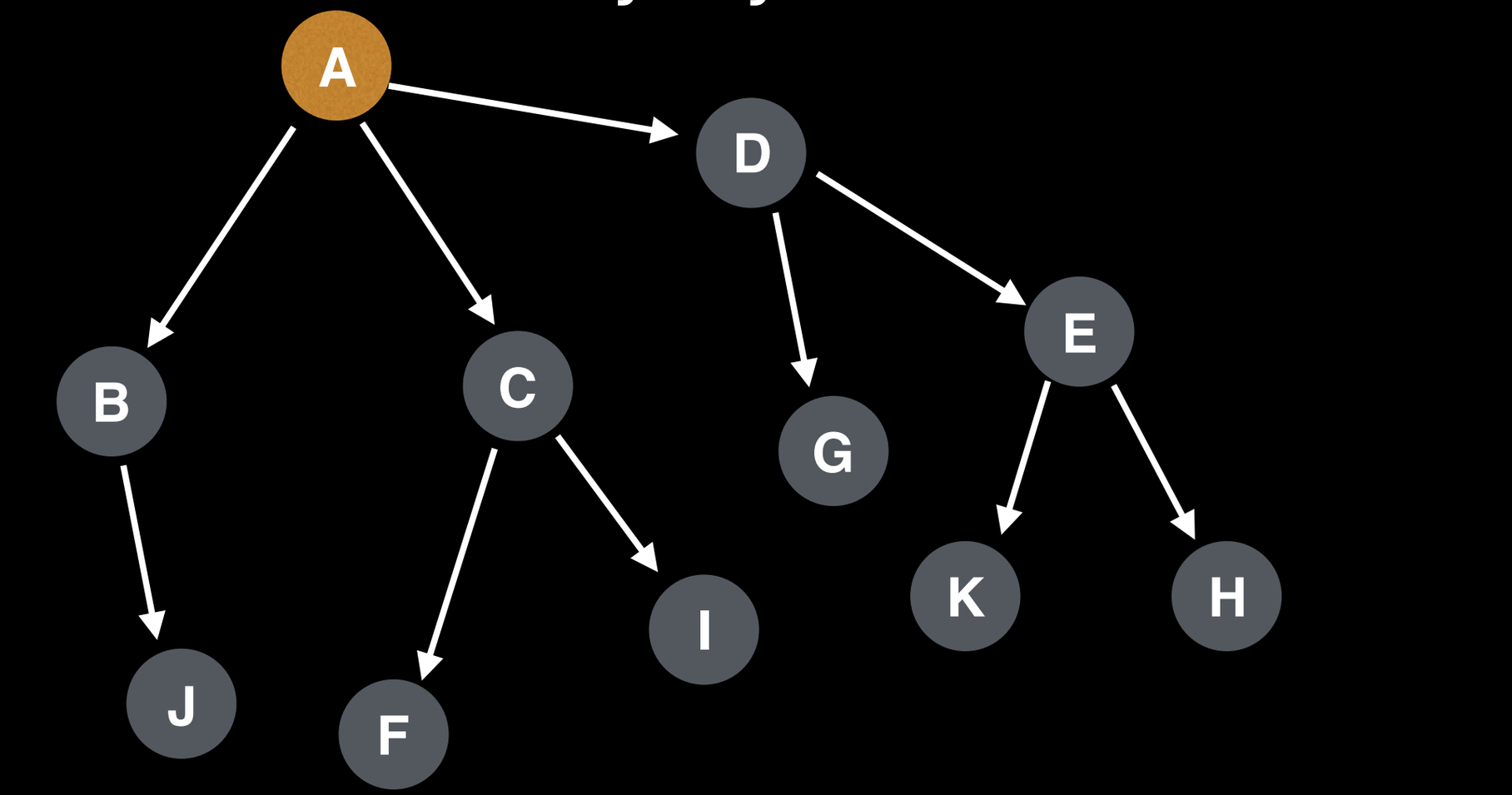
Directed Acyclic Graphs(DAG)

By definition, all rooted trees have a topological ordering since they do not contain any cycles.



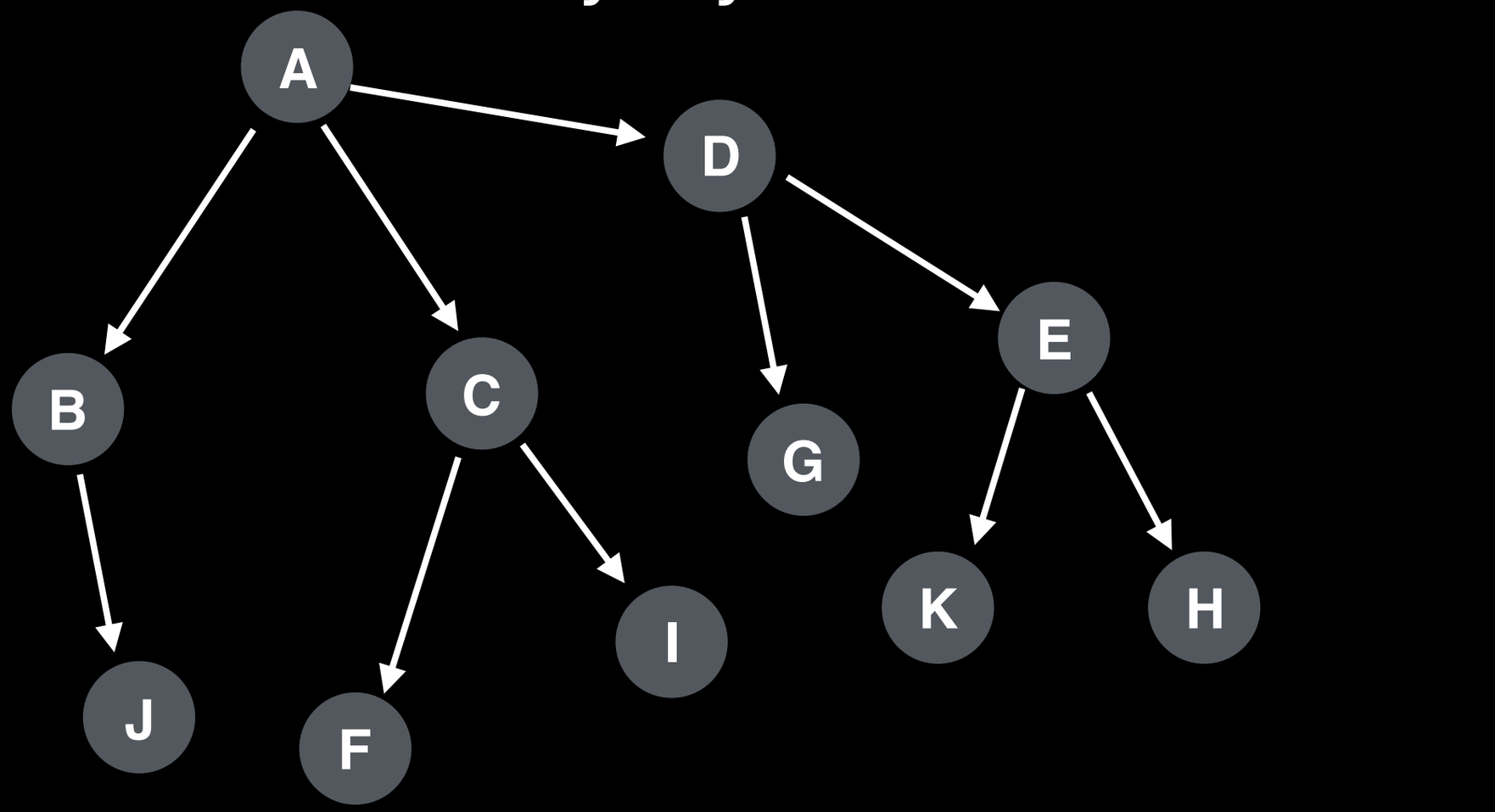
Directed Acyclic Graphs(DAG)

By definition, all rooted trees have a topological ordering since they do not contain any cycles.



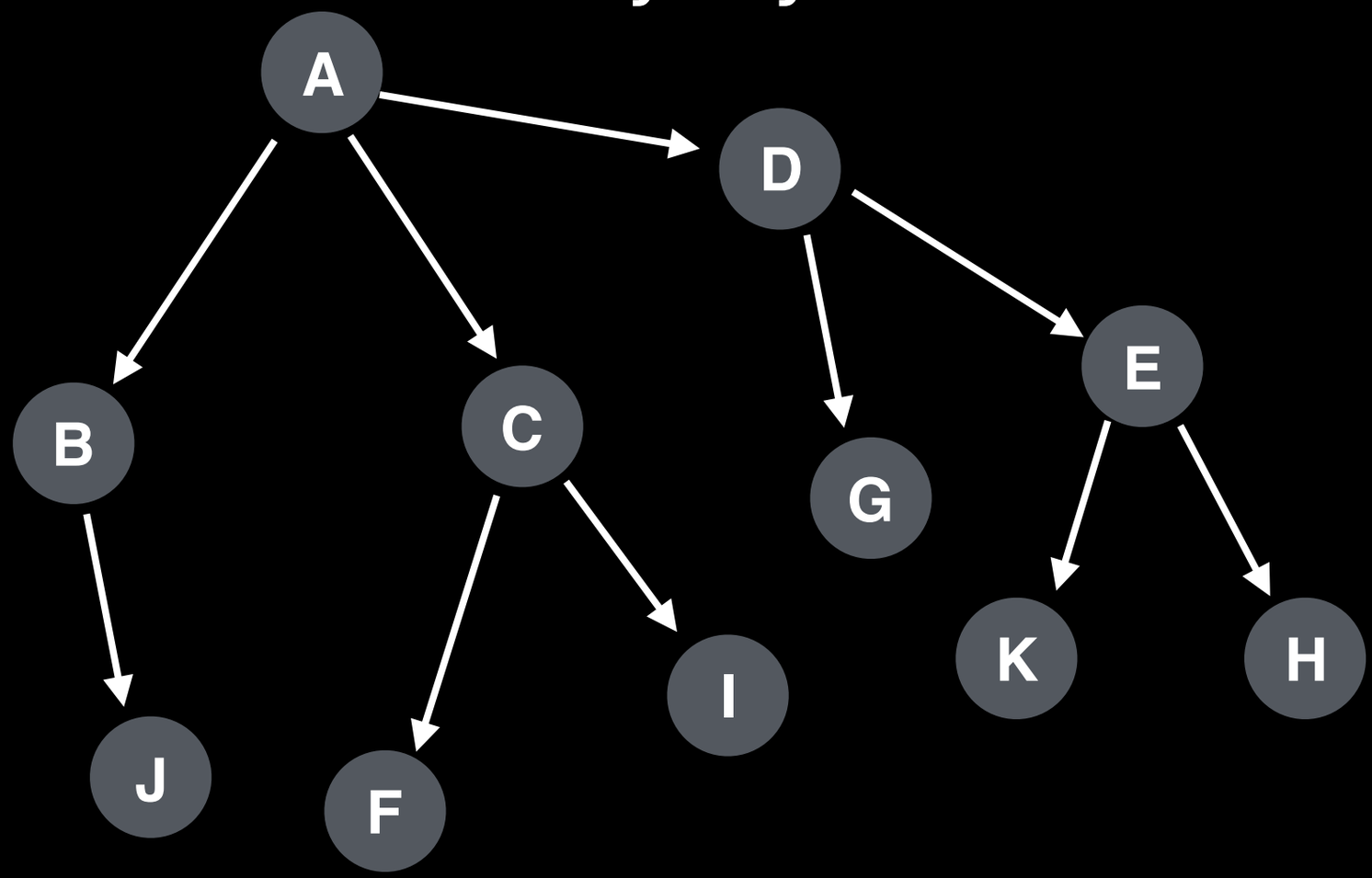
Directed Acyclic Graphs(DAG)

By definition, all rooted trees have a topological ordering since they do not contain any cycles.



Directed Acyclic Graphs(DAG)

By definition, all rooted trees have a topological ordering since they do not contain any cycles.



Topological ordering from left to right:

A C D E B G H K J F I

Topological Sort Algorithm

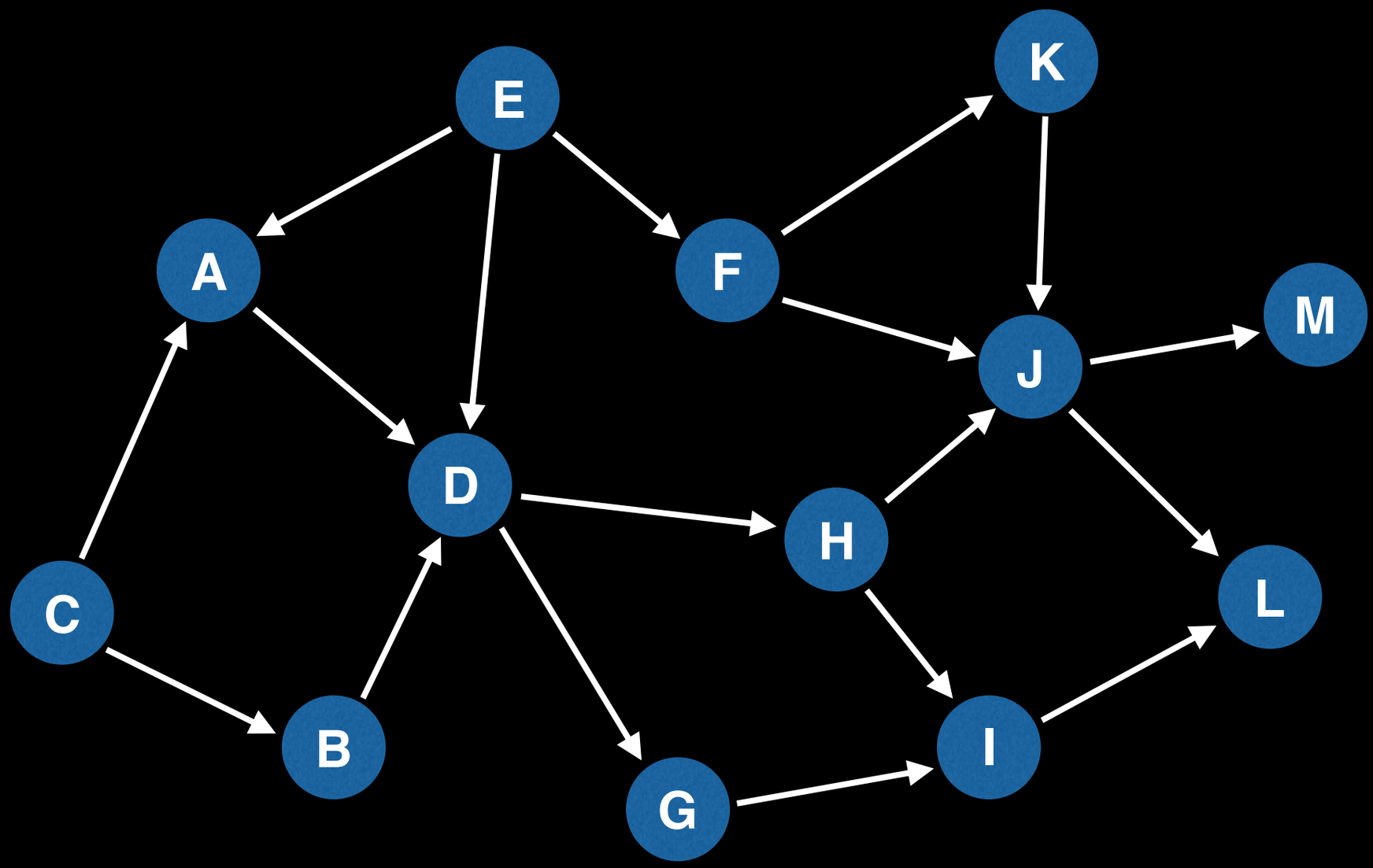
Pick an unvisited node

Beginning with the selected node, do a Depth First Search (DFS) exploring only unvisited nodes.

On the recursive callback of the DFS, add the current node to the topological ordering in reverse order.

Topological Sort Algorithm

DFS recursion call stack:



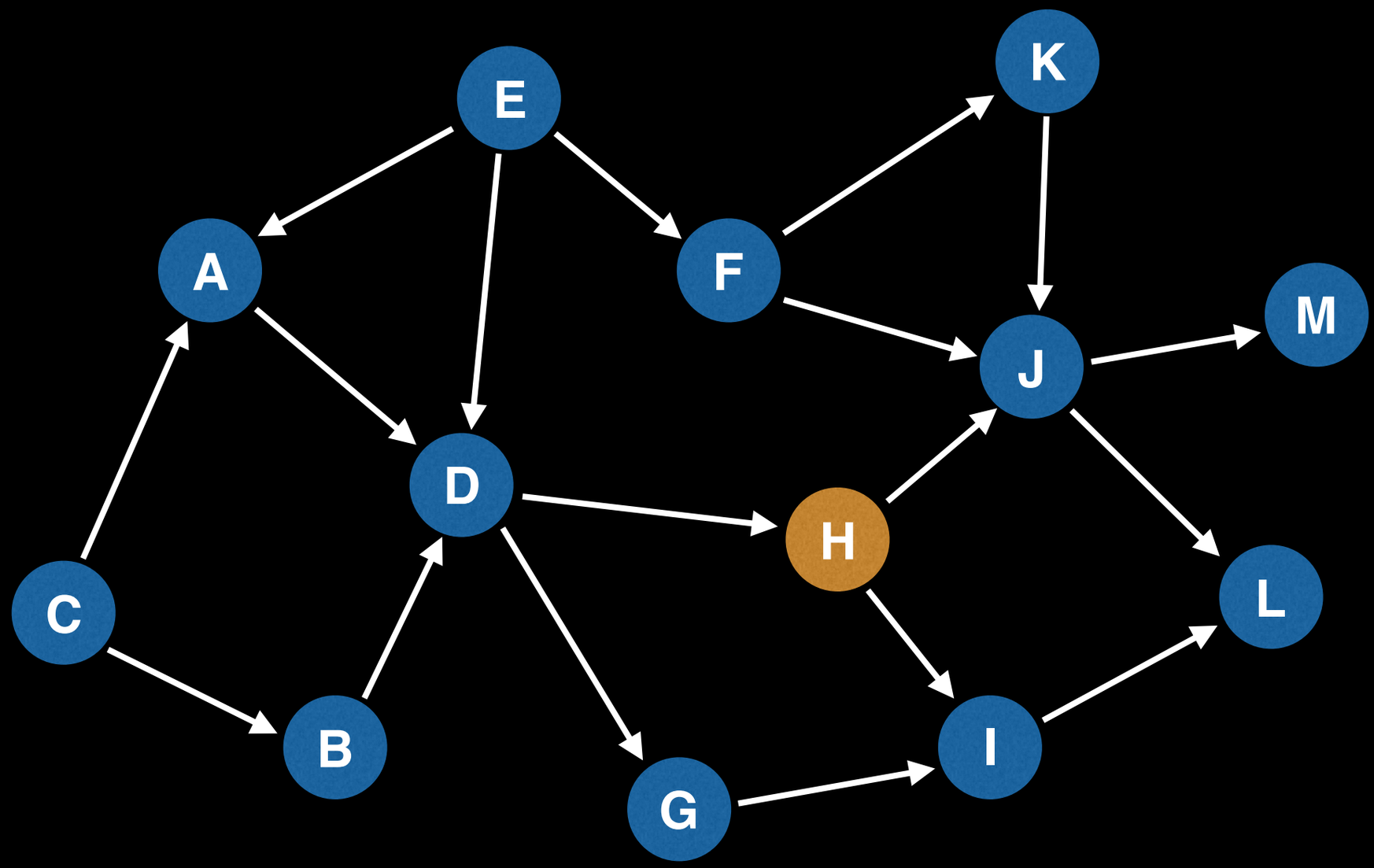
Topological ordering:

_ _ _ _ _ _ _ _ _ _ _ _ _

Topological Sort Algorithm

DFS recursion call stack:

Node H



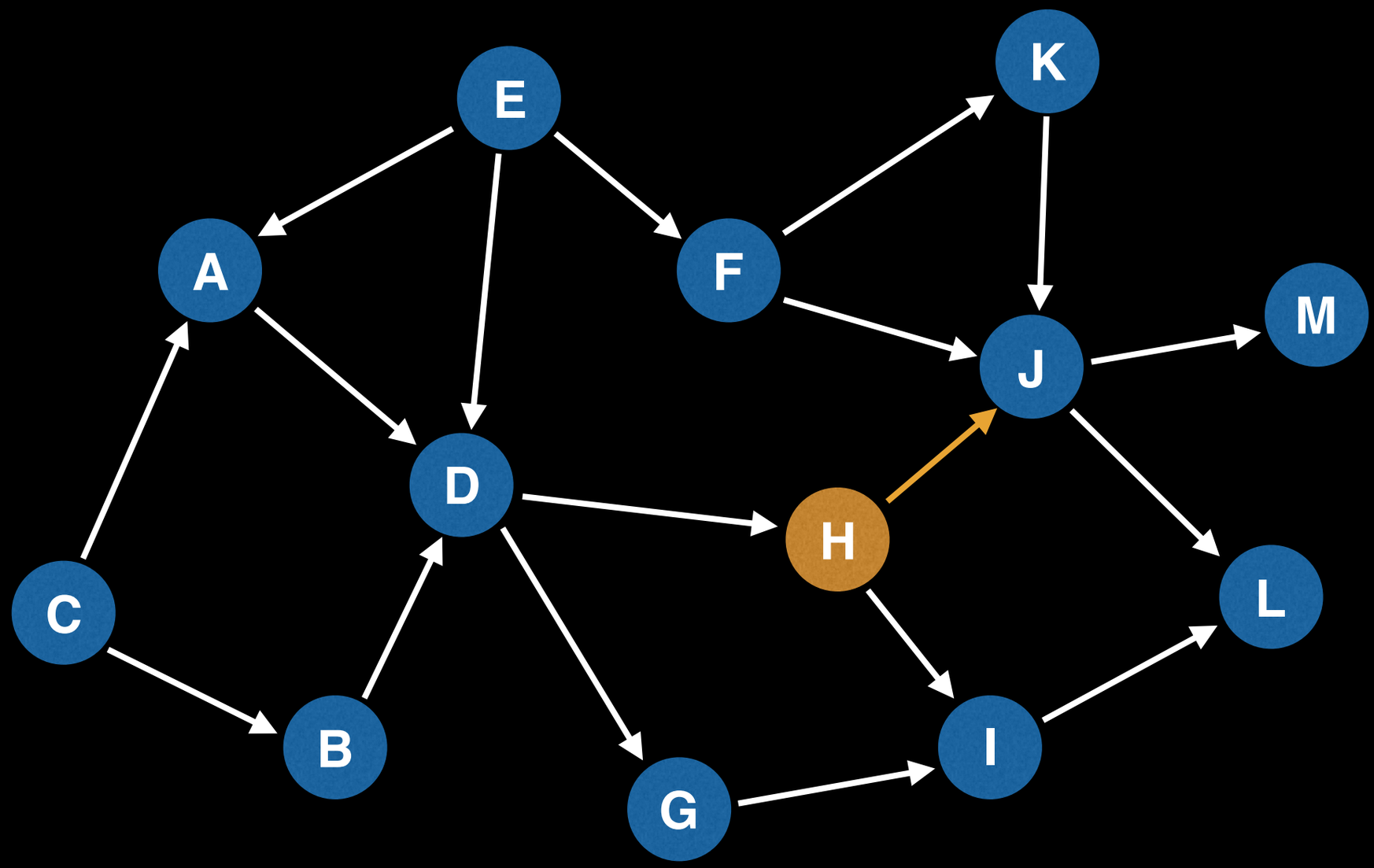
Topological ordering:

_ _ _ _ _ _ _ _ _ _ _ _ _

Topological Sort Algorithm

DFS recursion call stack:

Node H



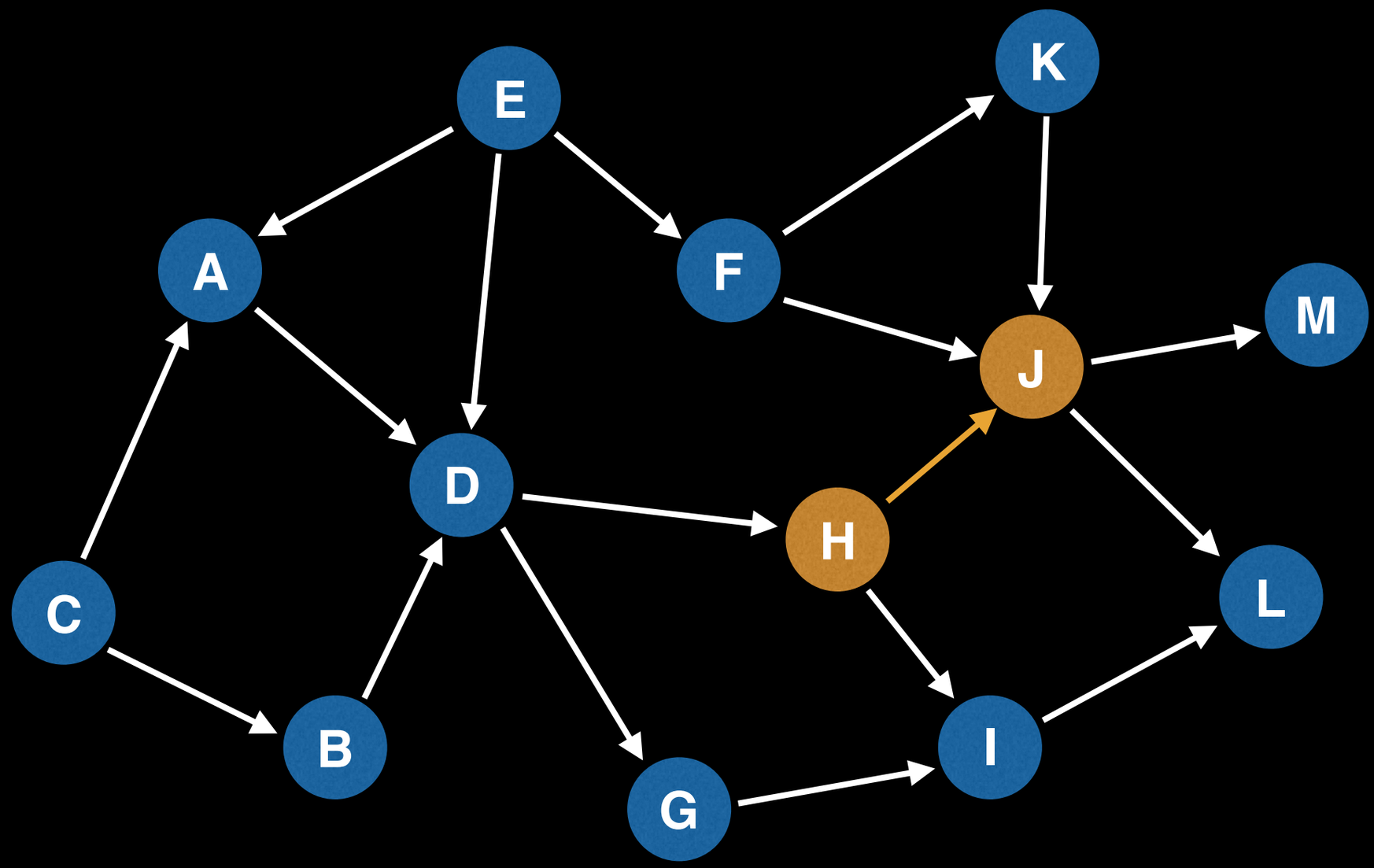
Topological ordering:

_ _ _ _ _ _ _ _ _ _ _ _ _

Topological Sort Algorithm

DFS recursion call stack:

Node H
Node J



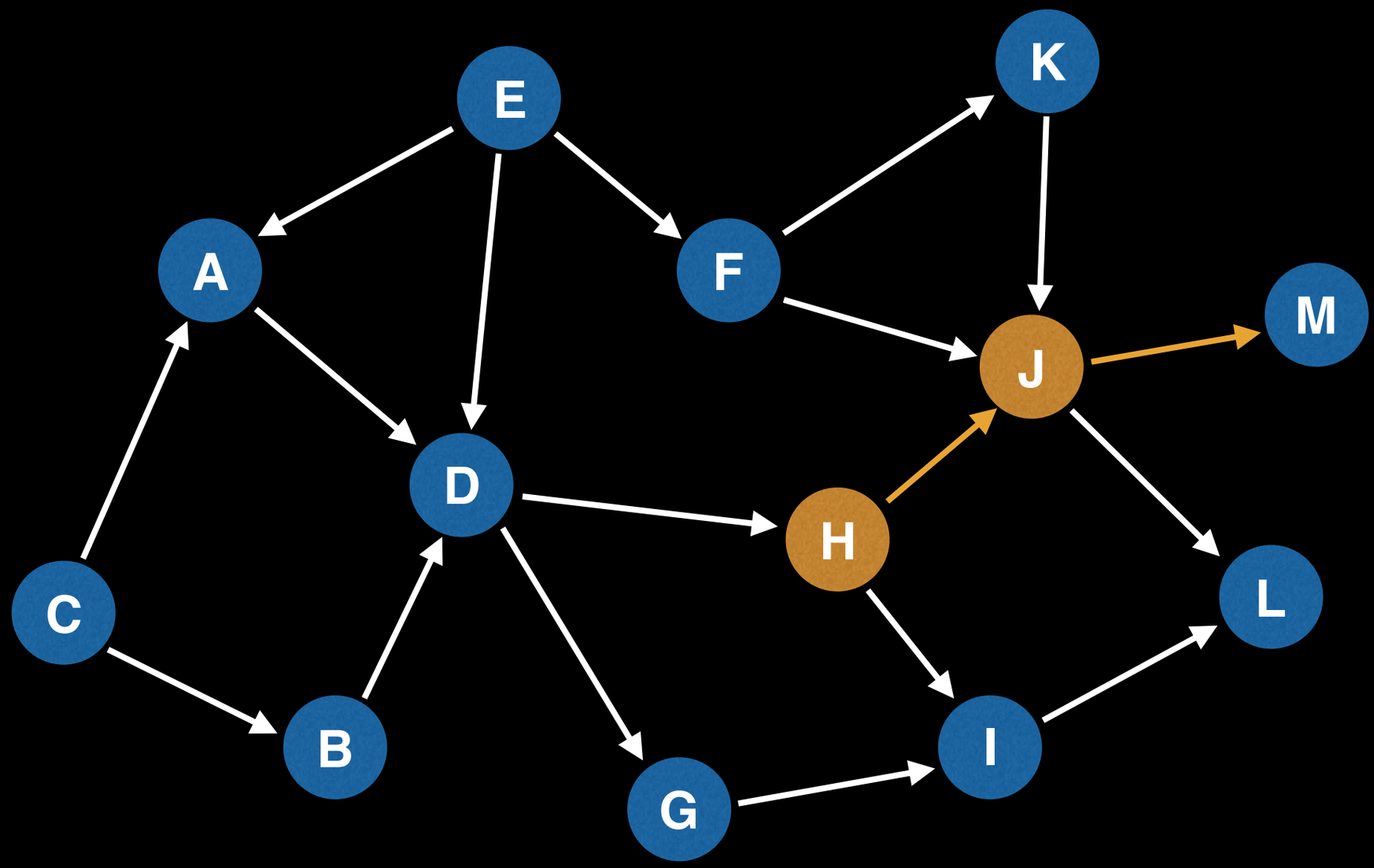
Topological ordering:

_ _ _ _ _ _ _ _ _ _ _ _ _

Topological Sort Algorithm

DFS recursion call stack:

Node H
Node J



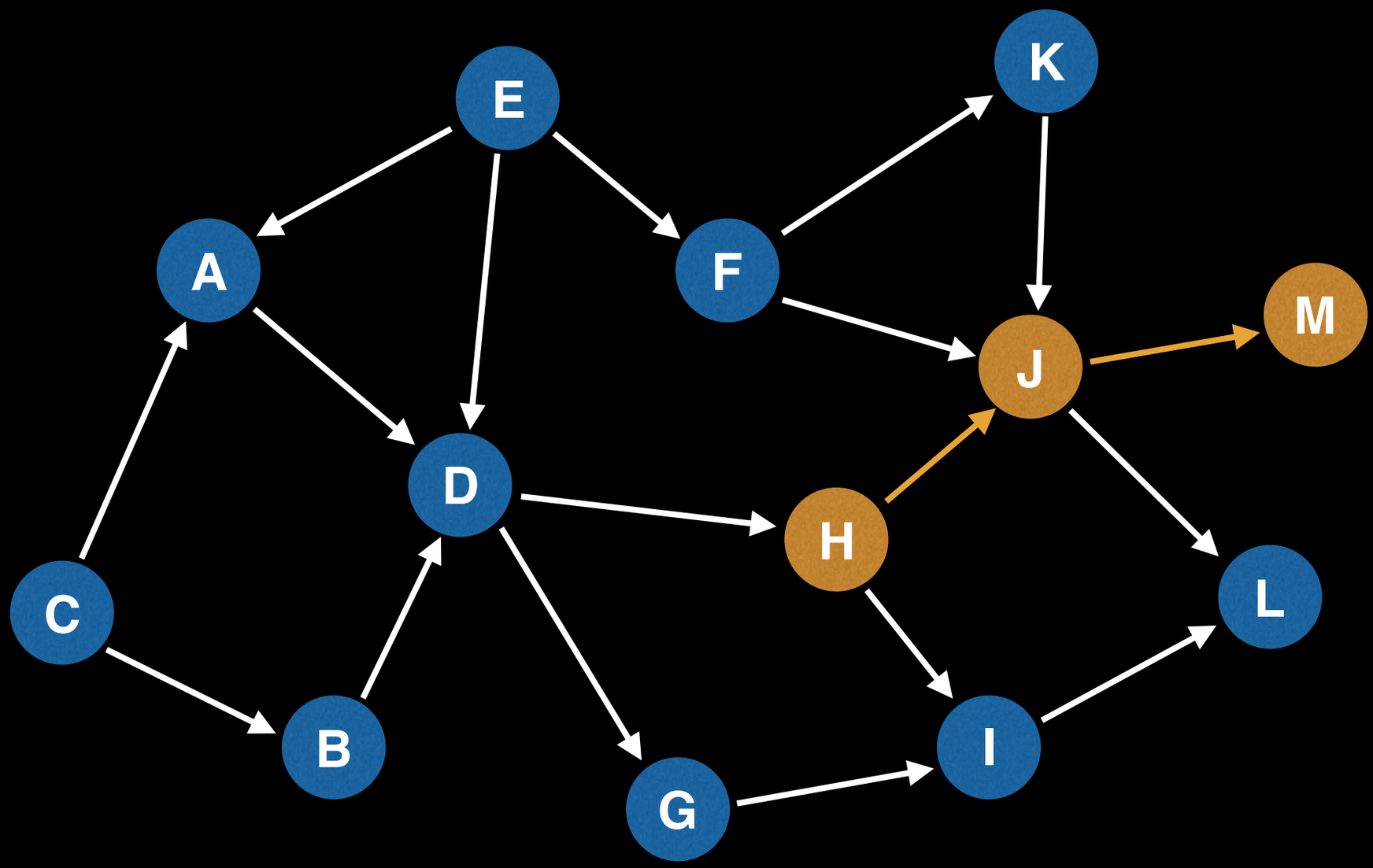
Topological ordering:

_ _ _ _ _ _ _ _ _ _ _ _ _

Topological Sort Algorithm

DFS recursion call stack:

Node H
Node J
Node M



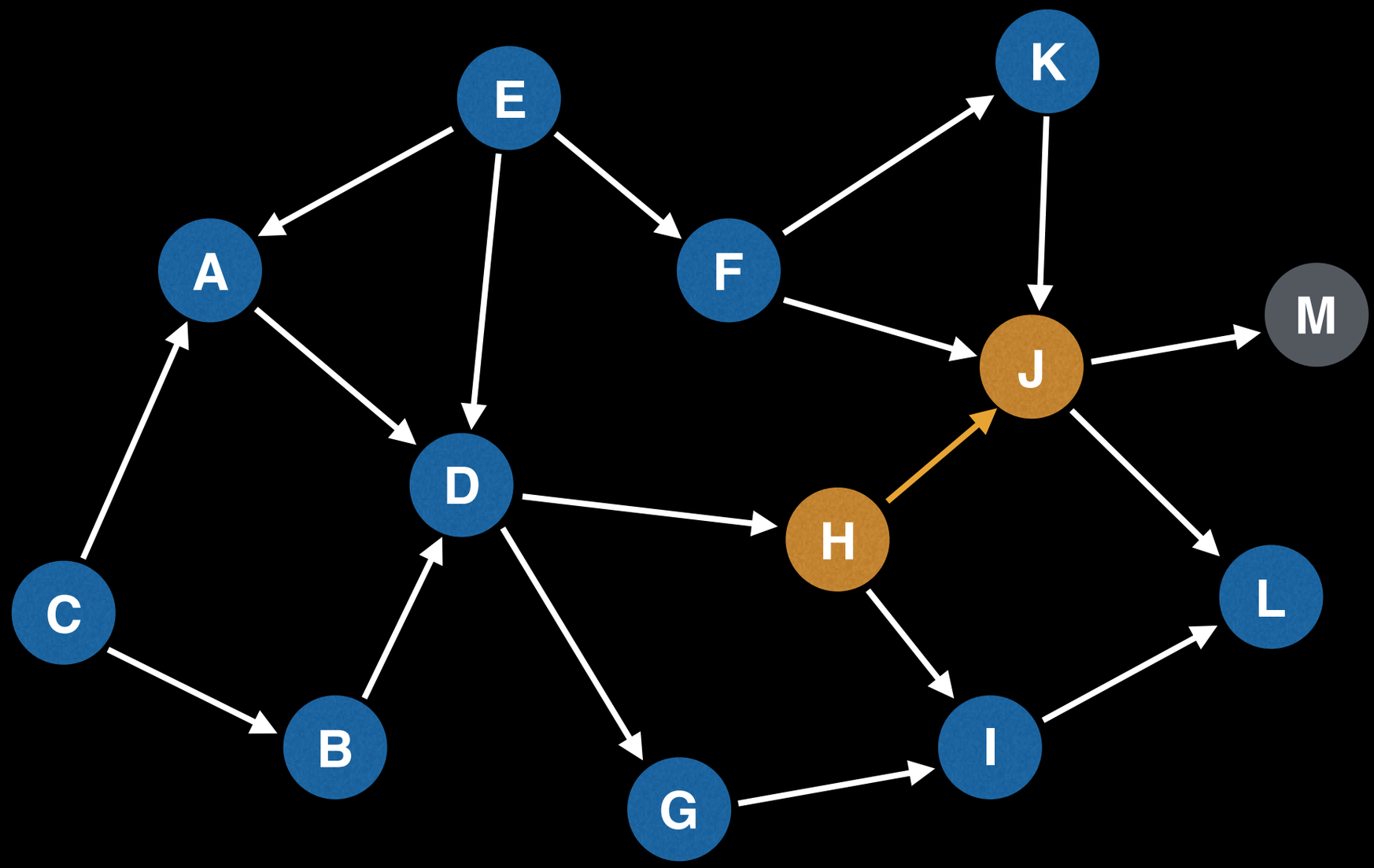
Topological ordering:

_ _ _ _ _ _ _ _ _ _ _ _ _

Topological Sort Algorithm

DFS recursion
call stack:

Node H
Node J



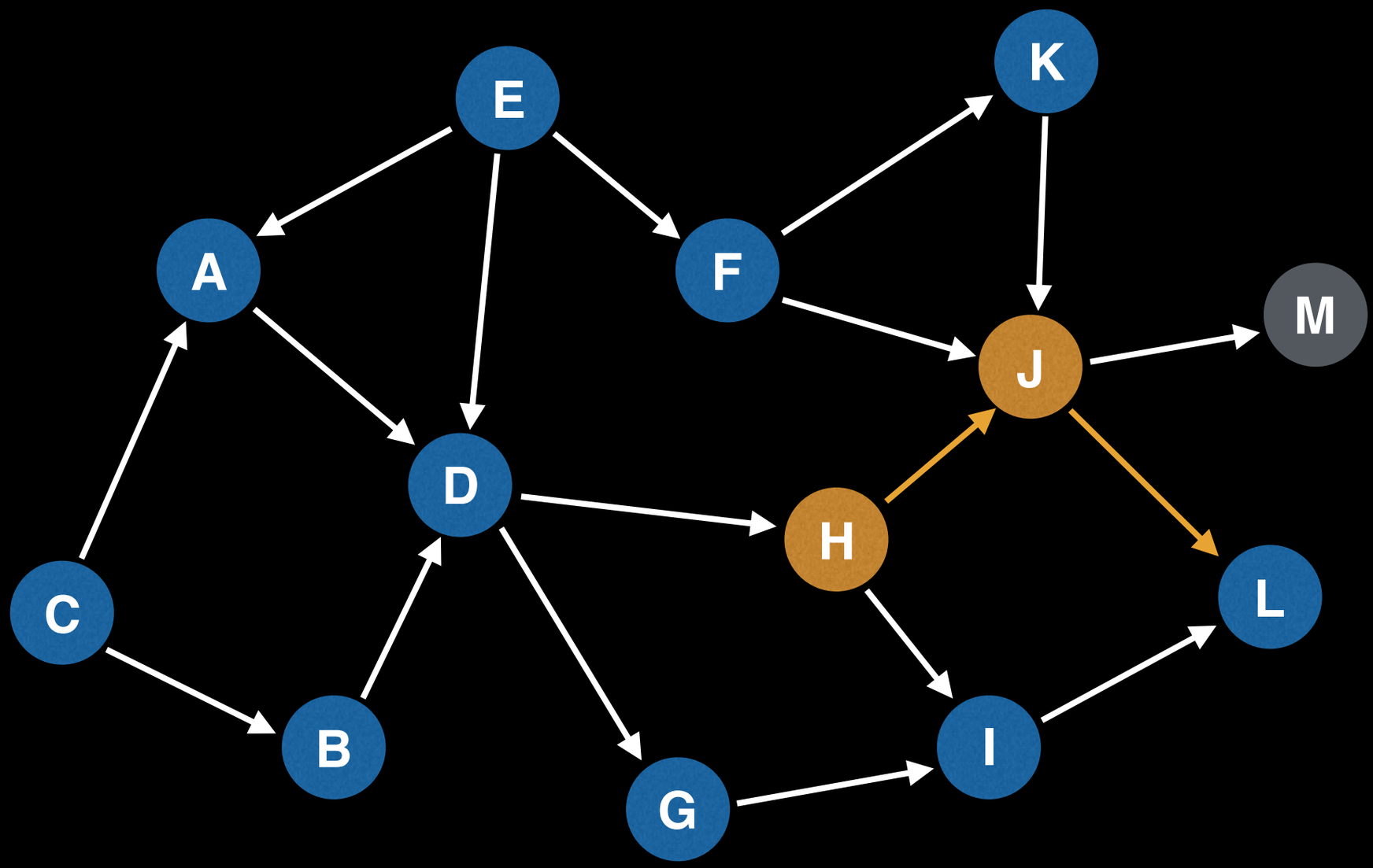
Topological ordering:

M

Topological Sort Algorithm

DFS recursion
call stack:

Node H
Node J



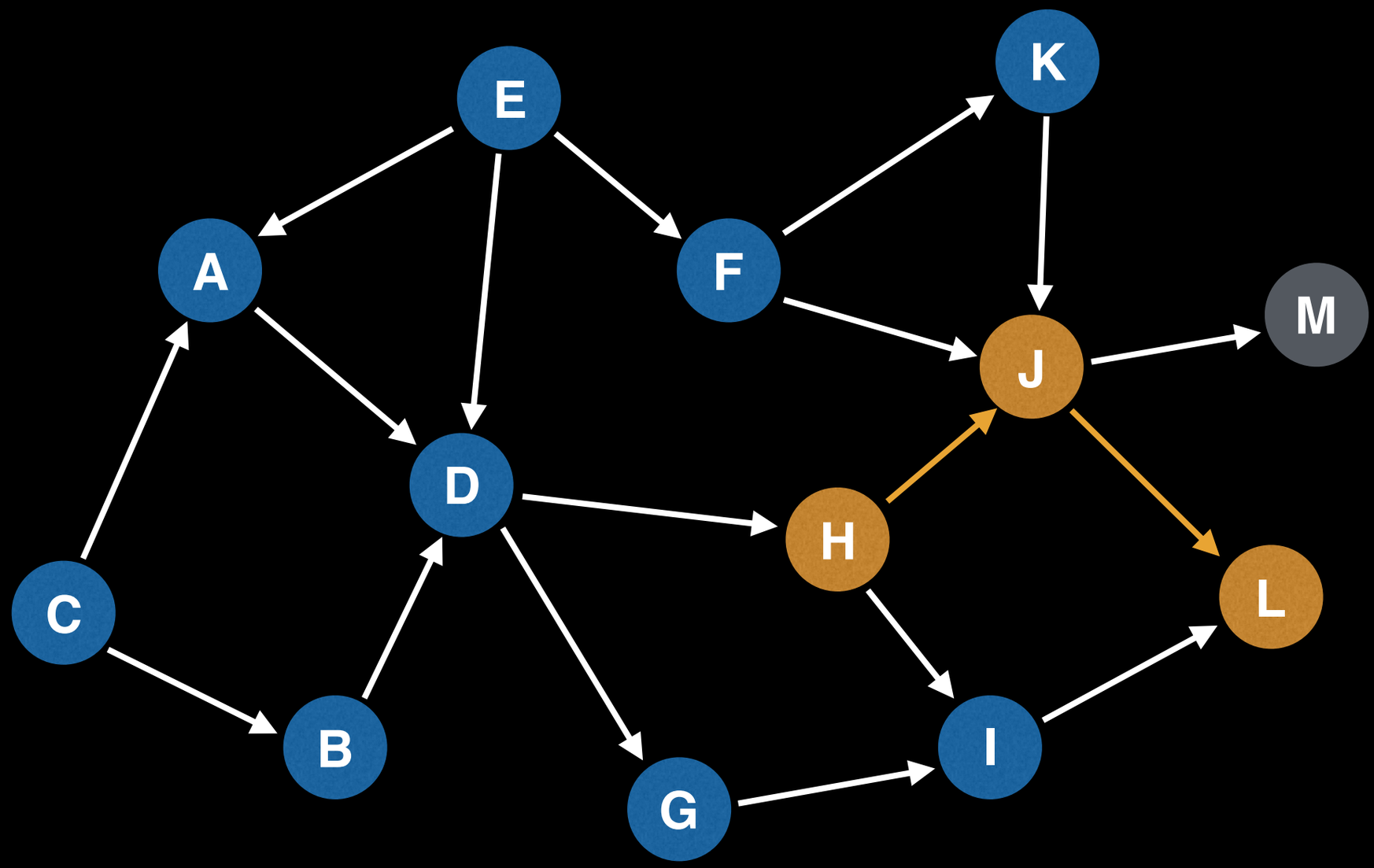
Topological ordering:

M

Topological Sort Algorithm

DFS recursion call stack:

Node H
Node J
Node L



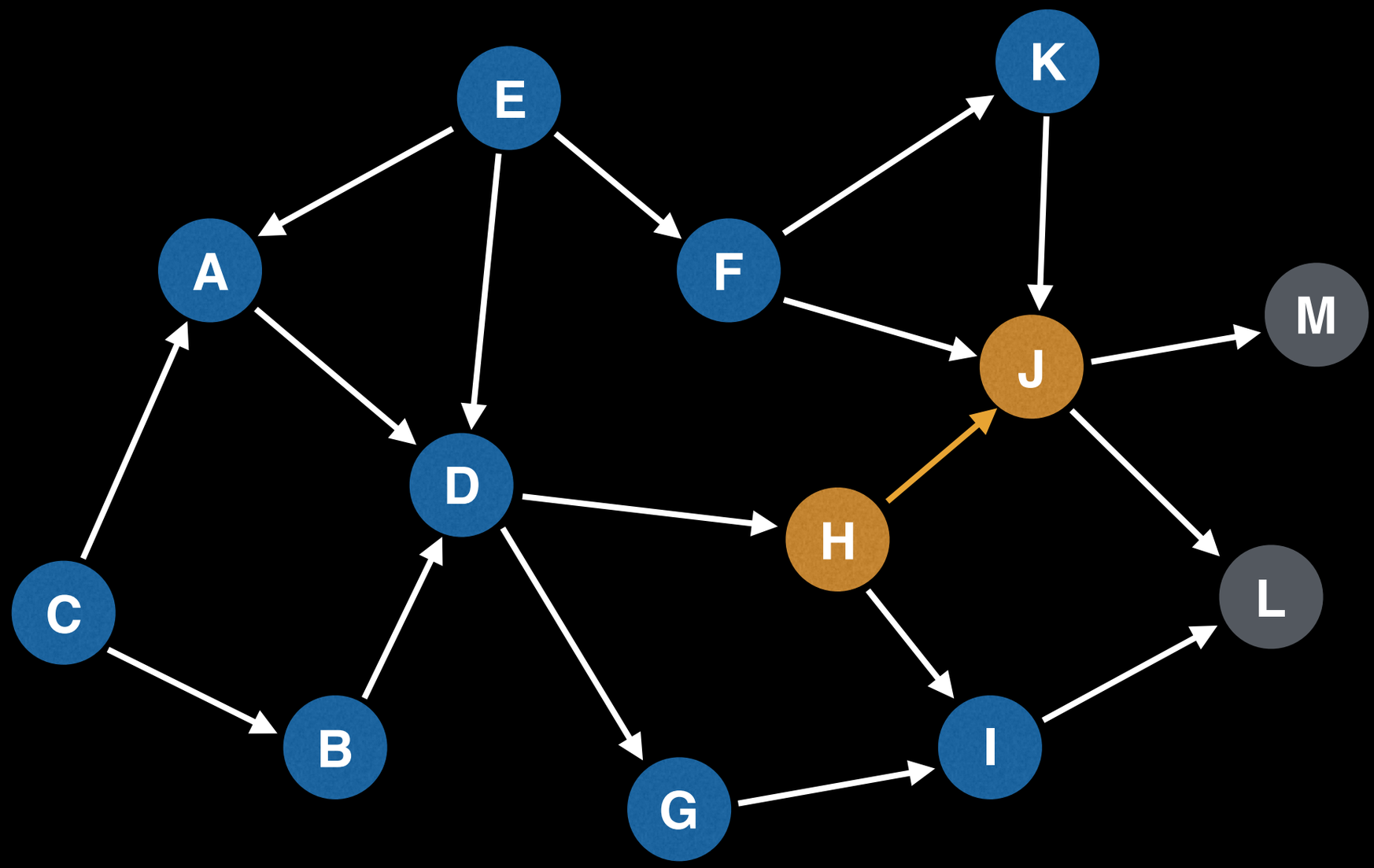
Topological ordering:

_ _ _ _ _ _ _ _ _ _ _ _ M

Topological Sort Algorithm

DFS recursion call stack:

Node H
Node J



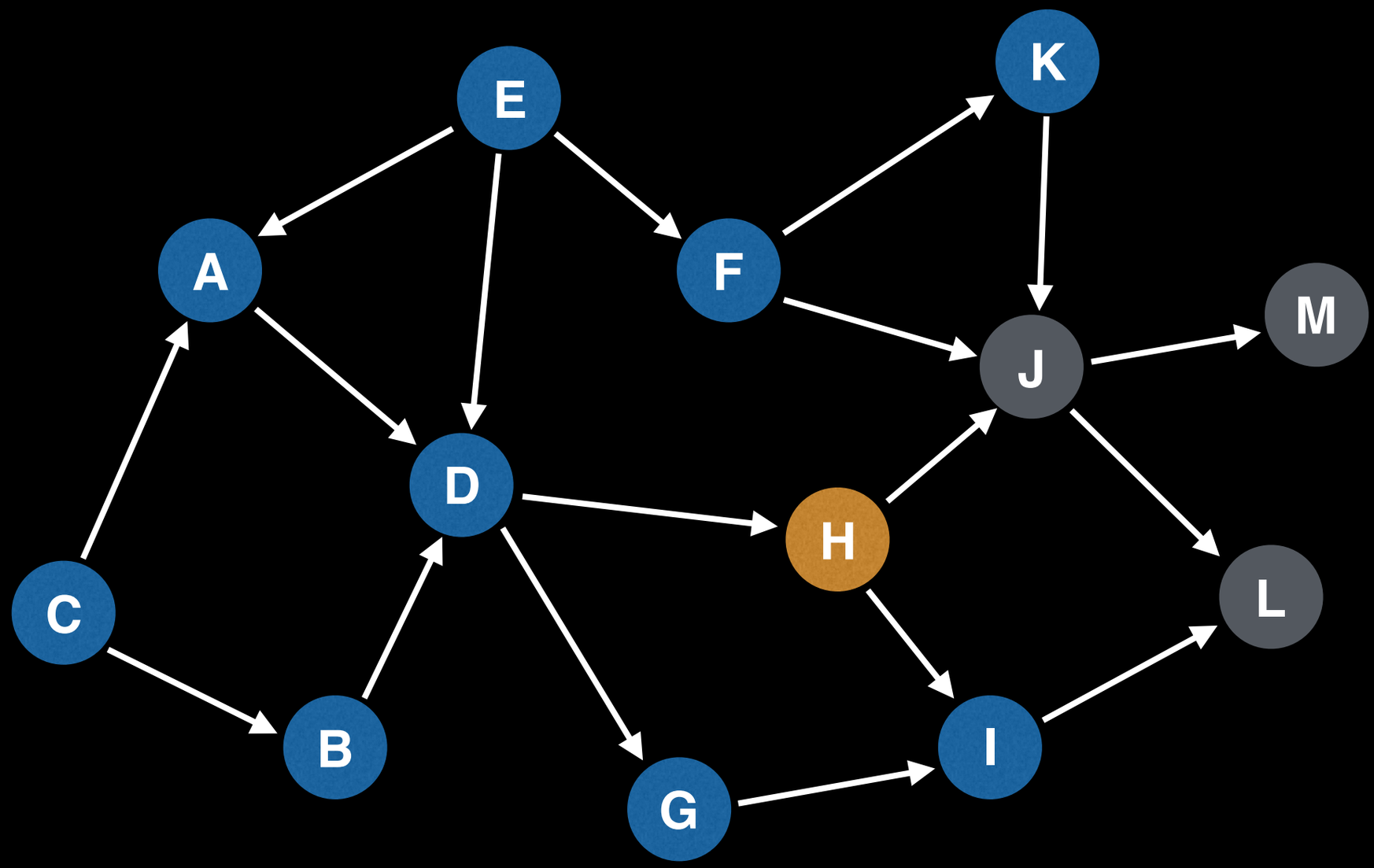
Topological ordering:

_ _ _ _ _ _ _ _ _ _ _ L M

Topological Sort Algorithm

DFS recursion call stack:

Node H



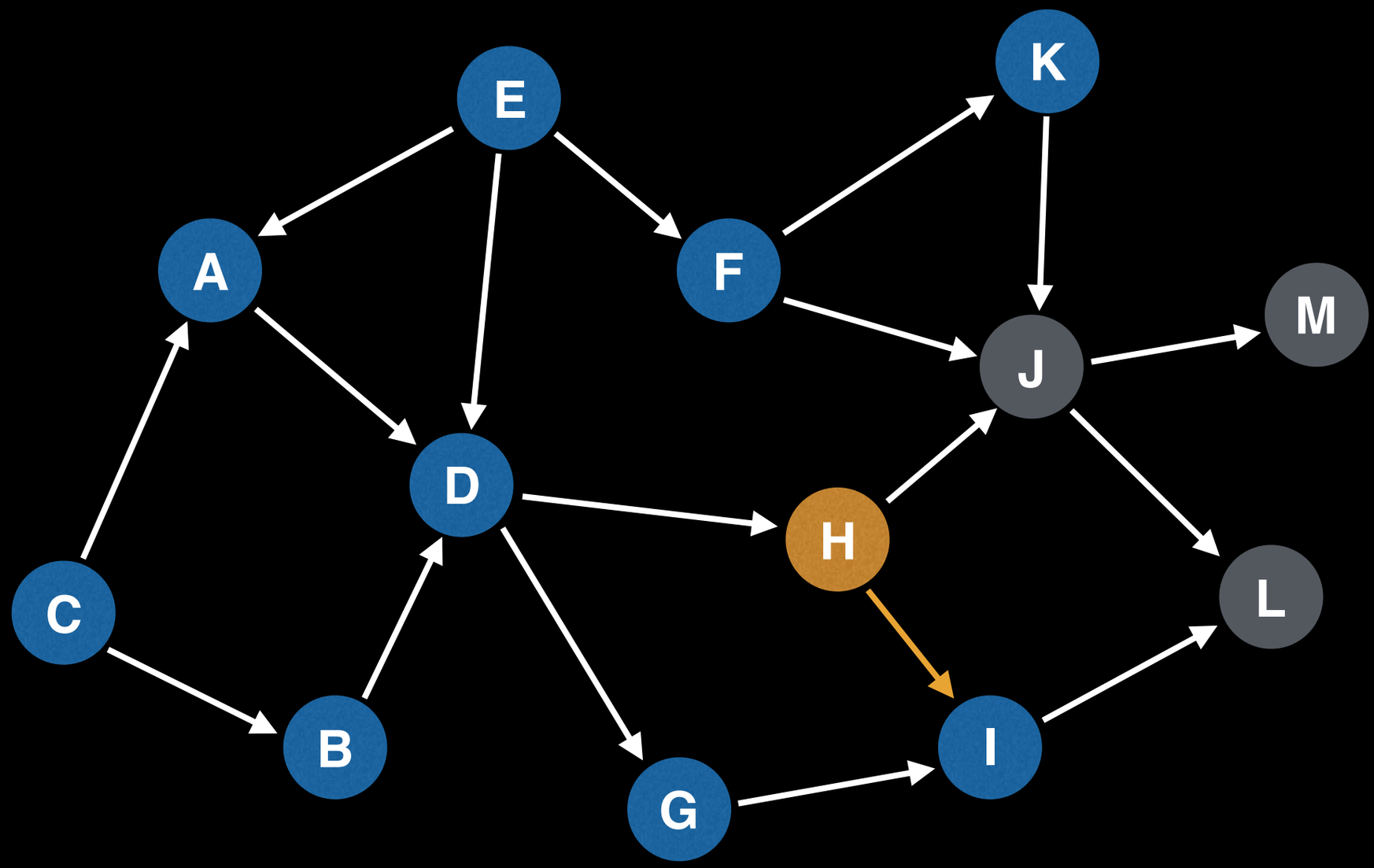
Topological ordering:

_ _ _ _ _ _ _ _ _ _ J L M

Topological Sort Algorithm

DFS recursion call stack:

Node H



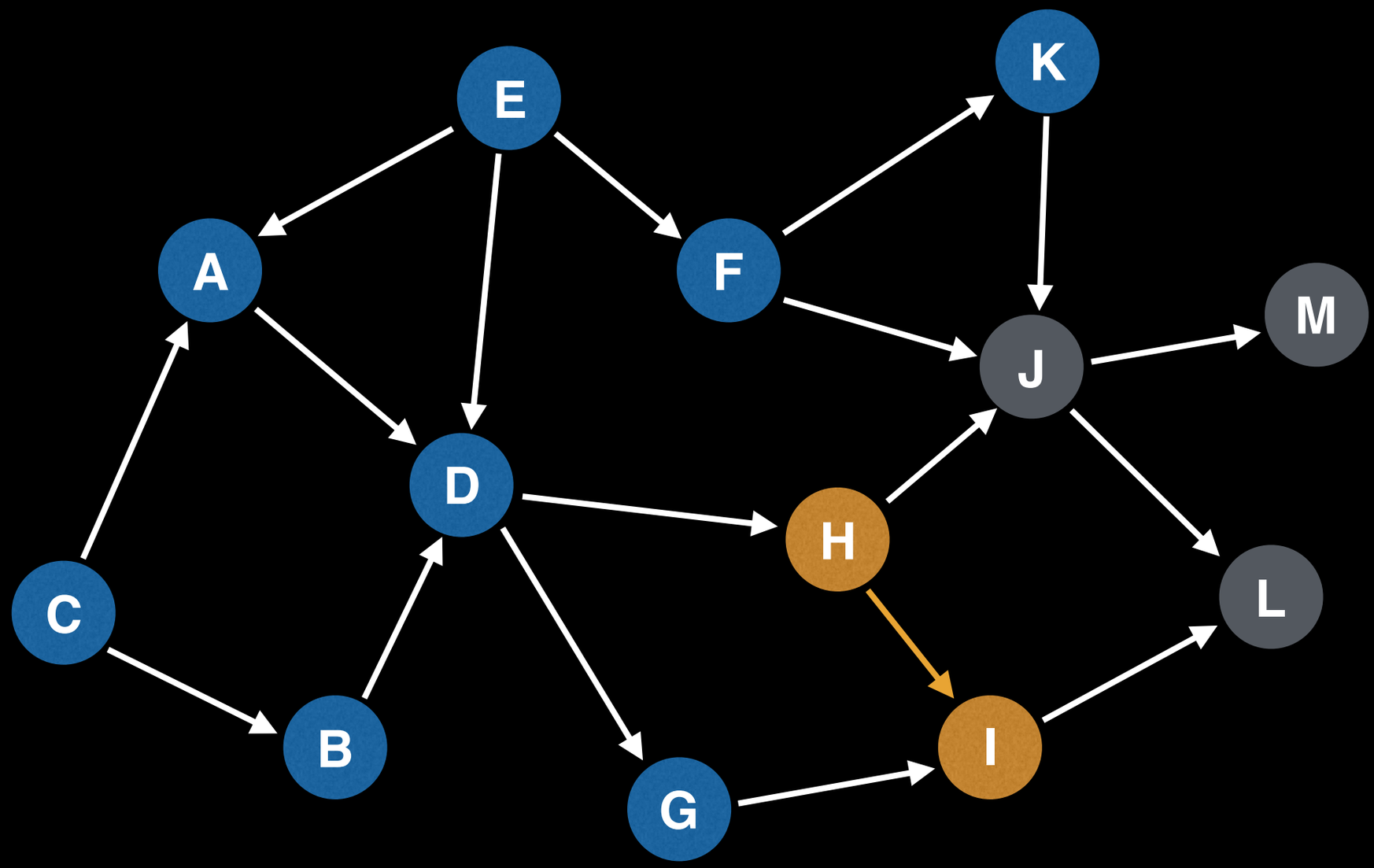
Topological ordering:

_ _ _ _ _ _ _ _ _ _ J L M

Topological Sort Algorithm

DFS recursion call stack:

Node H
Node I



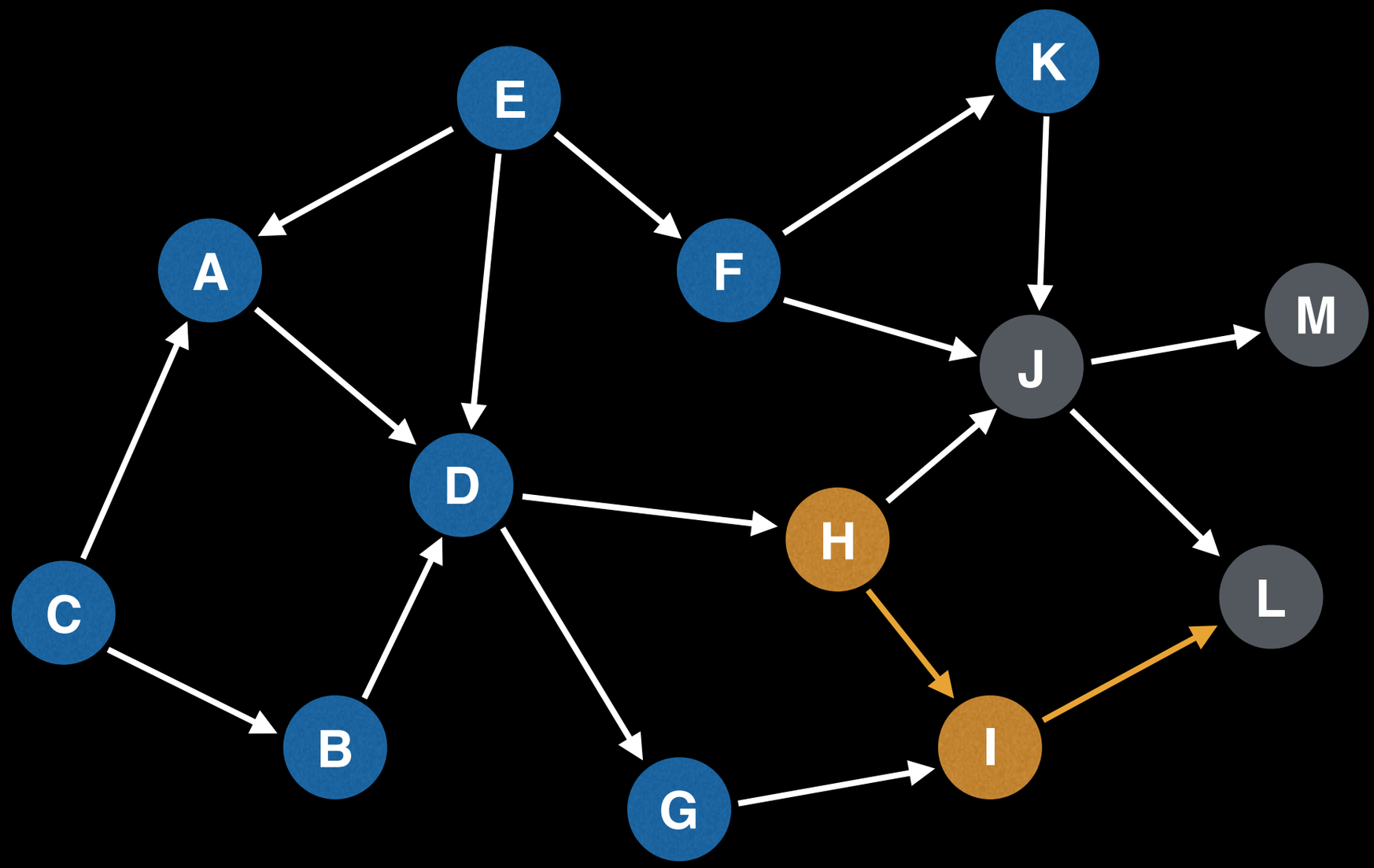
Topological ordering:

_ _ _ _ _ _ _ _ _ _ J L M

Topological Sort Algorithm

DFS recursion call stack:

Node H
Node I



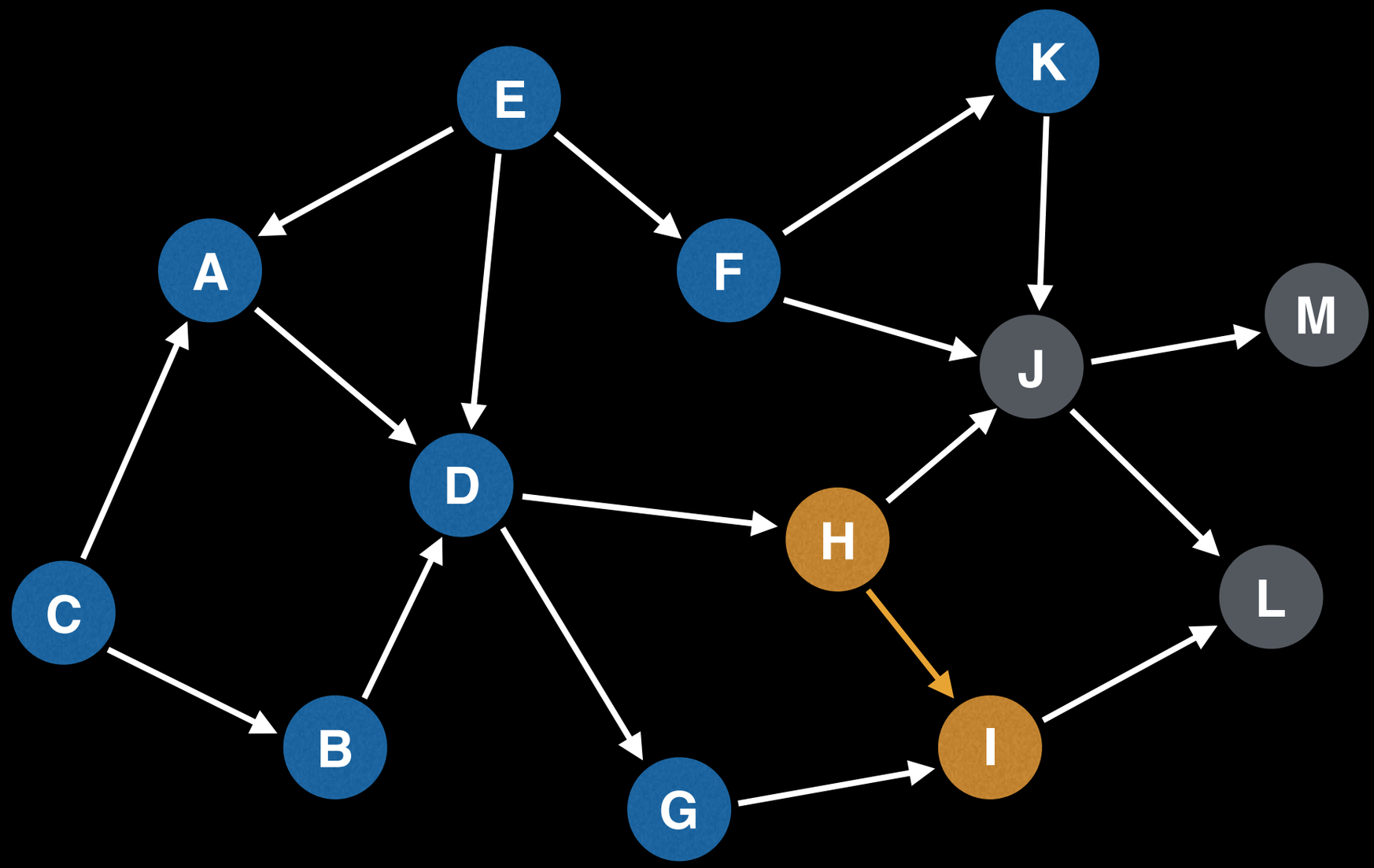
Topological ordering:

_ _ _ _ _ _ _ _ _ _ J L M

Topological Sort Algorithm

DFS recursion call stack:

Node H
Node I



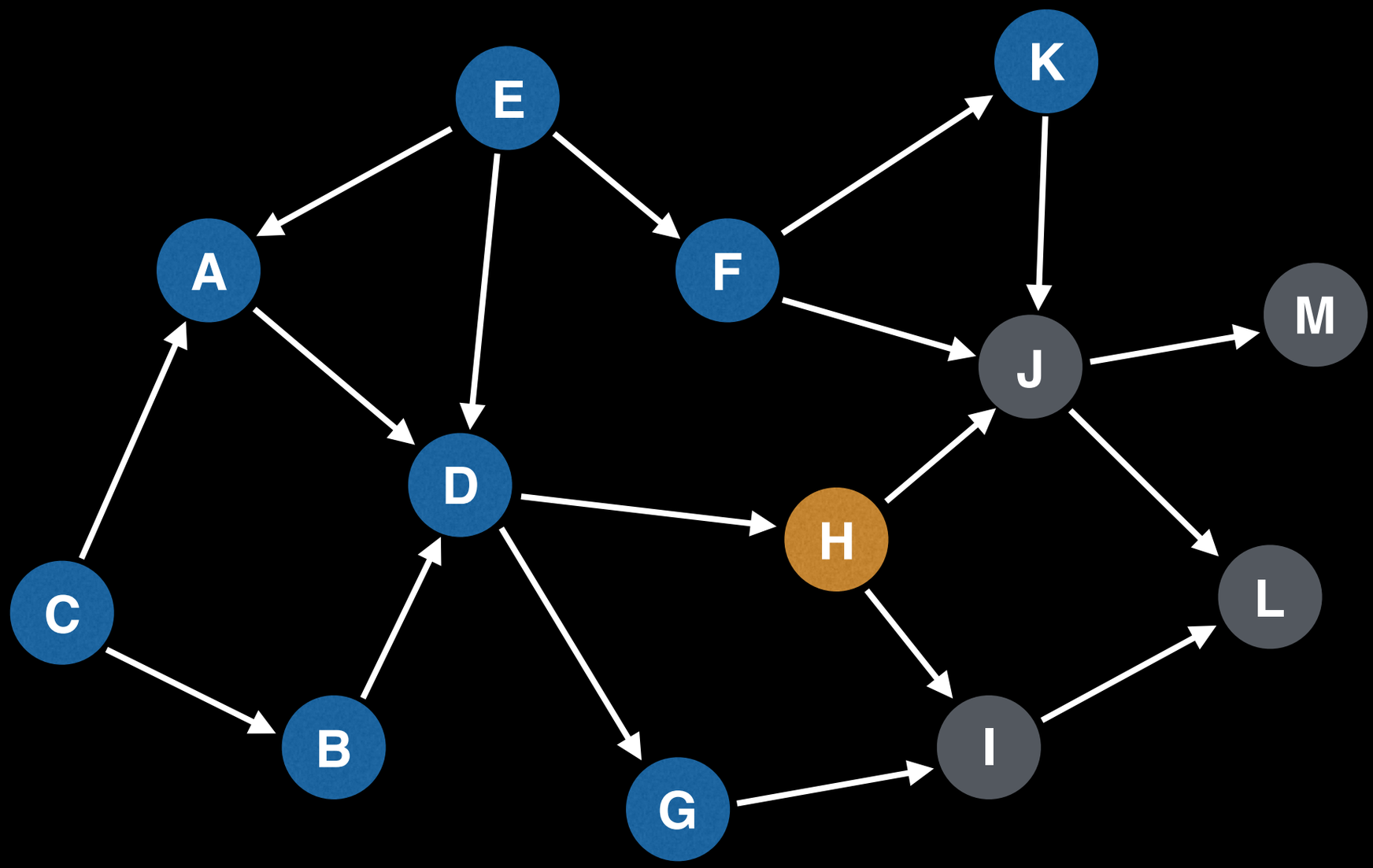
Topological ordering:

_ _ _ _ _ _ _ _ _ _ J L M

Topological Sort Algorithm

DFS recursion call stack:

Node H

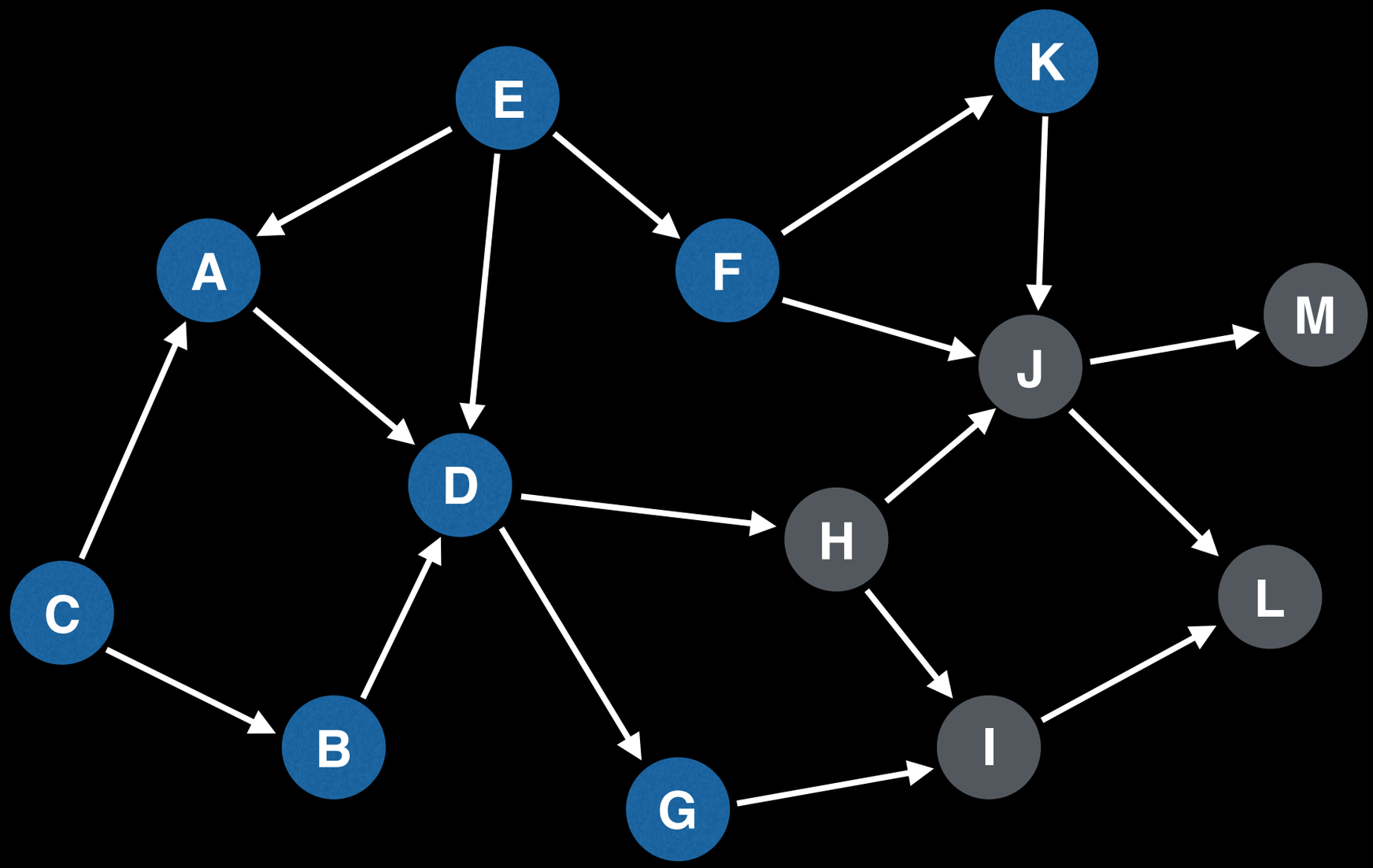


Topological ordering:

_ _ _ _ _ _ _ _ _ _ I J L M

Topological Sort Algorithm

DFS recursion call stack:



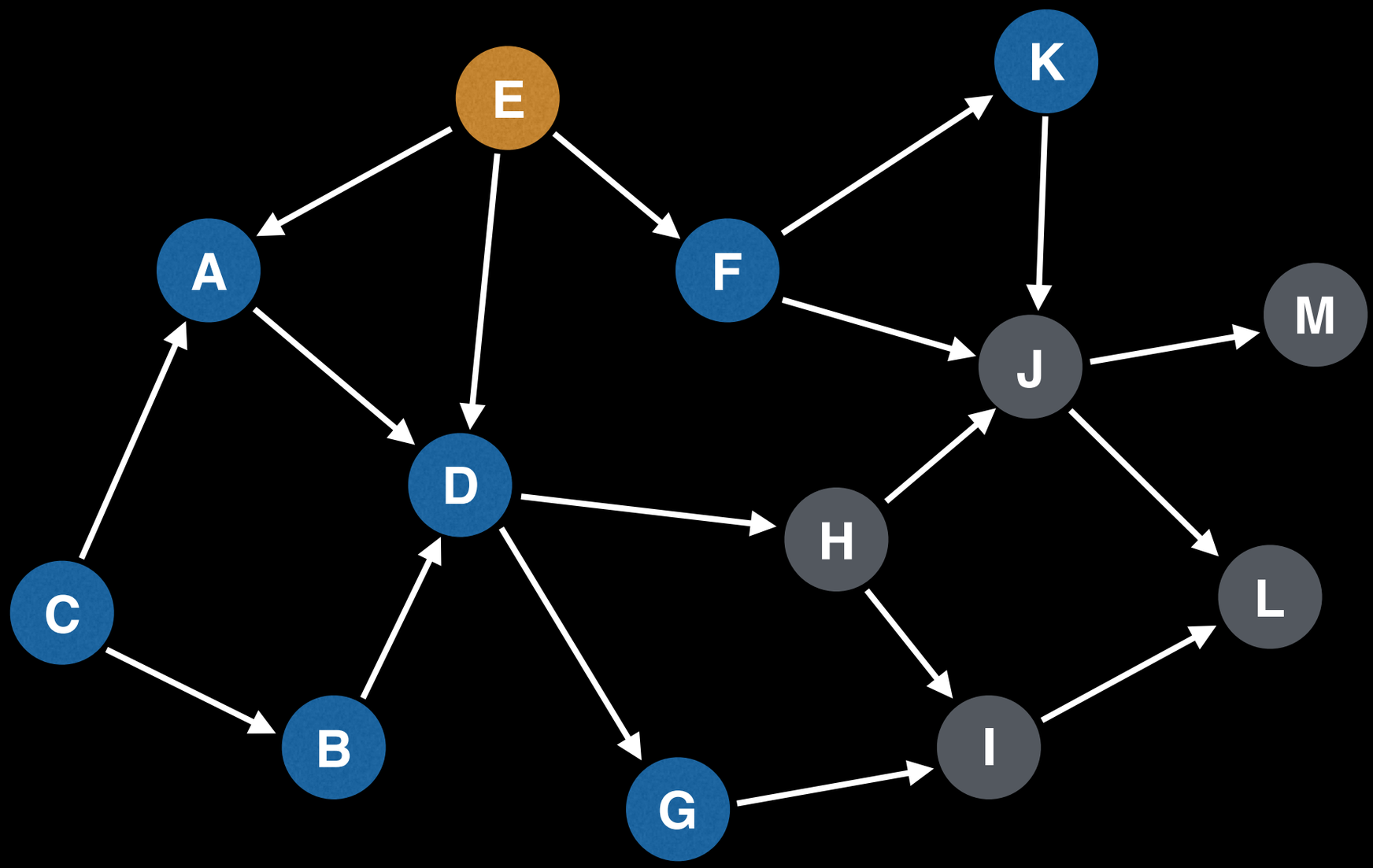
Topological ordering:

_ _ _ _ _ _ _ _ _ _ H I J L M

Topological Sort Algorithm

DFS recursion call stack:

Node E



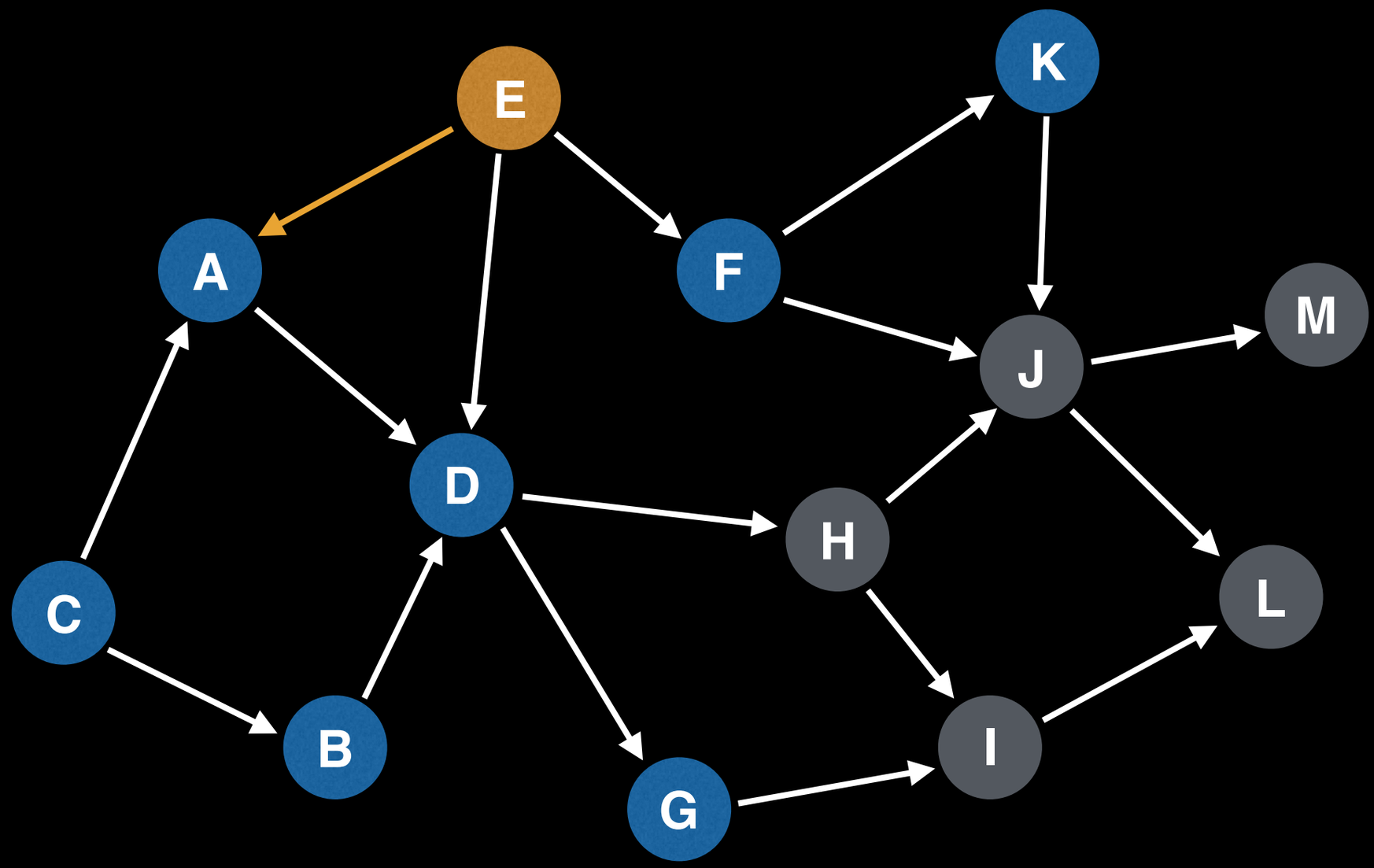
Topological ordering:

_ _ _ _ _ _ _ _ _ H I J L M

Topological Sort Algorithm

DFS recursion call stack:

Node E



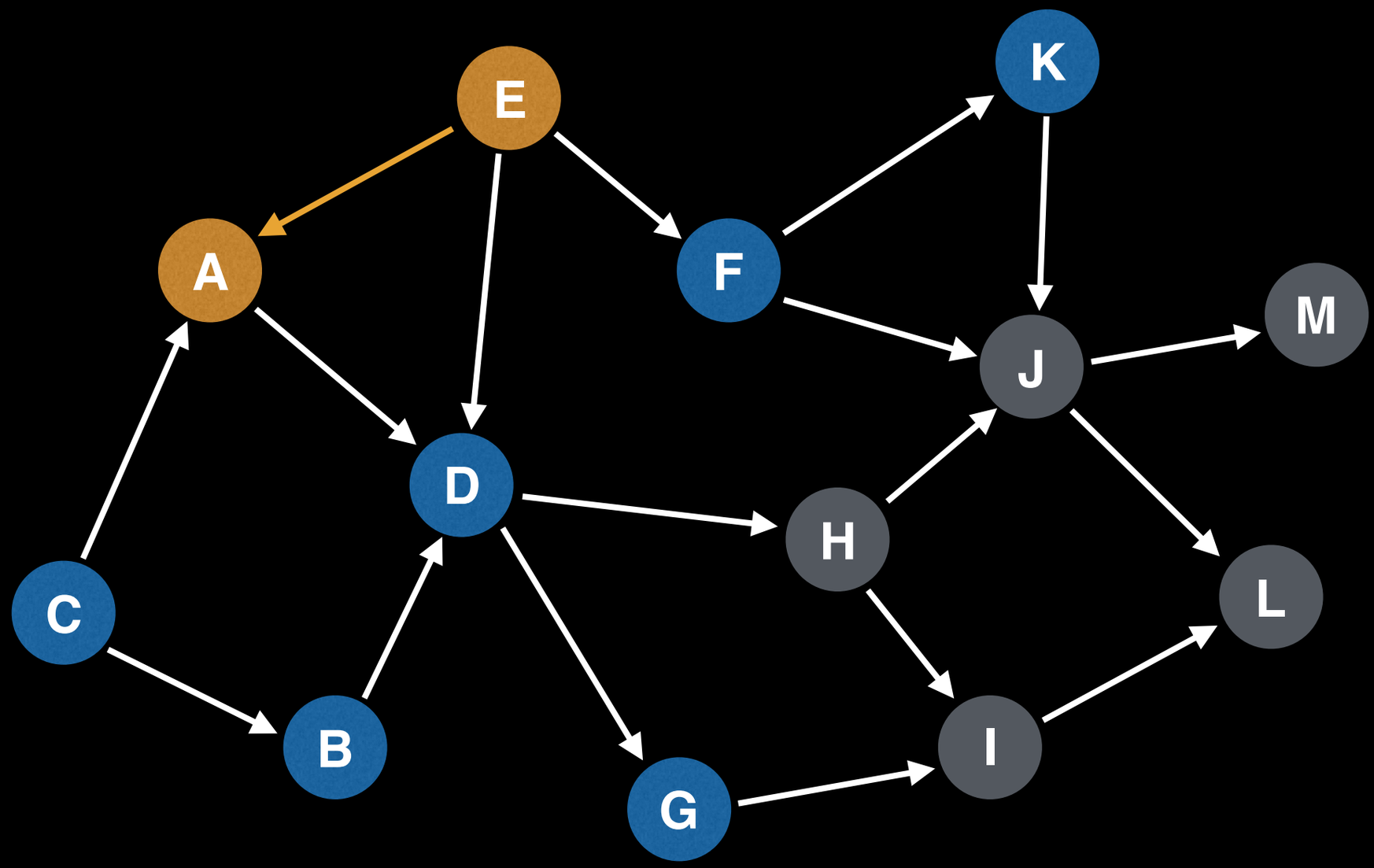
Topological ordering:

_ _ _ _ _ _ _ _ _ _ H I J L M

Topological Sort Algorithm

DFS recursion call stack:

Node E
Node A



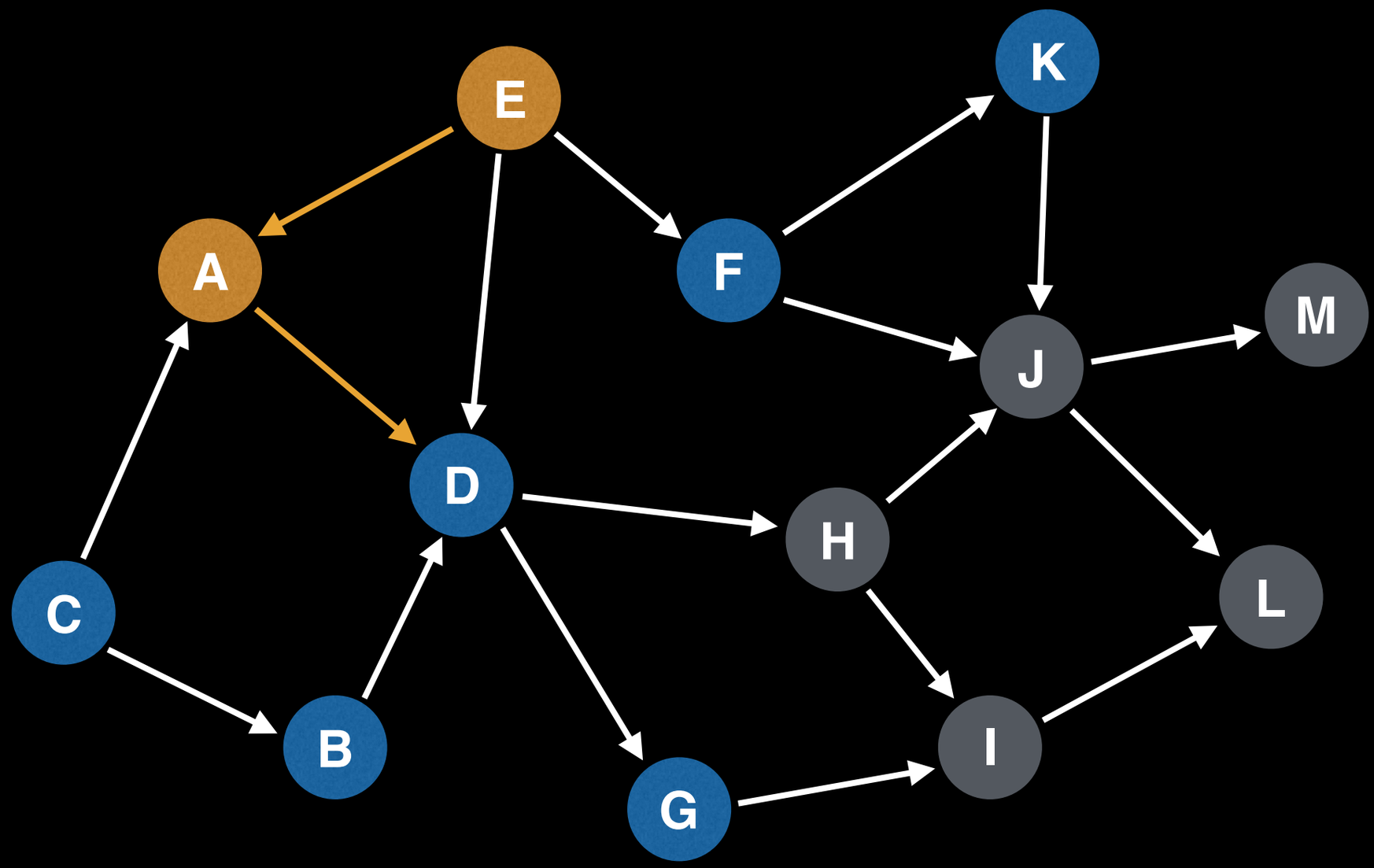
Topological ordering:

_ _ _ _ _ _ _ _ _ _ H I J L M

Topological Sort Algorithm

DFS recursion call stack:

Node E
Node A



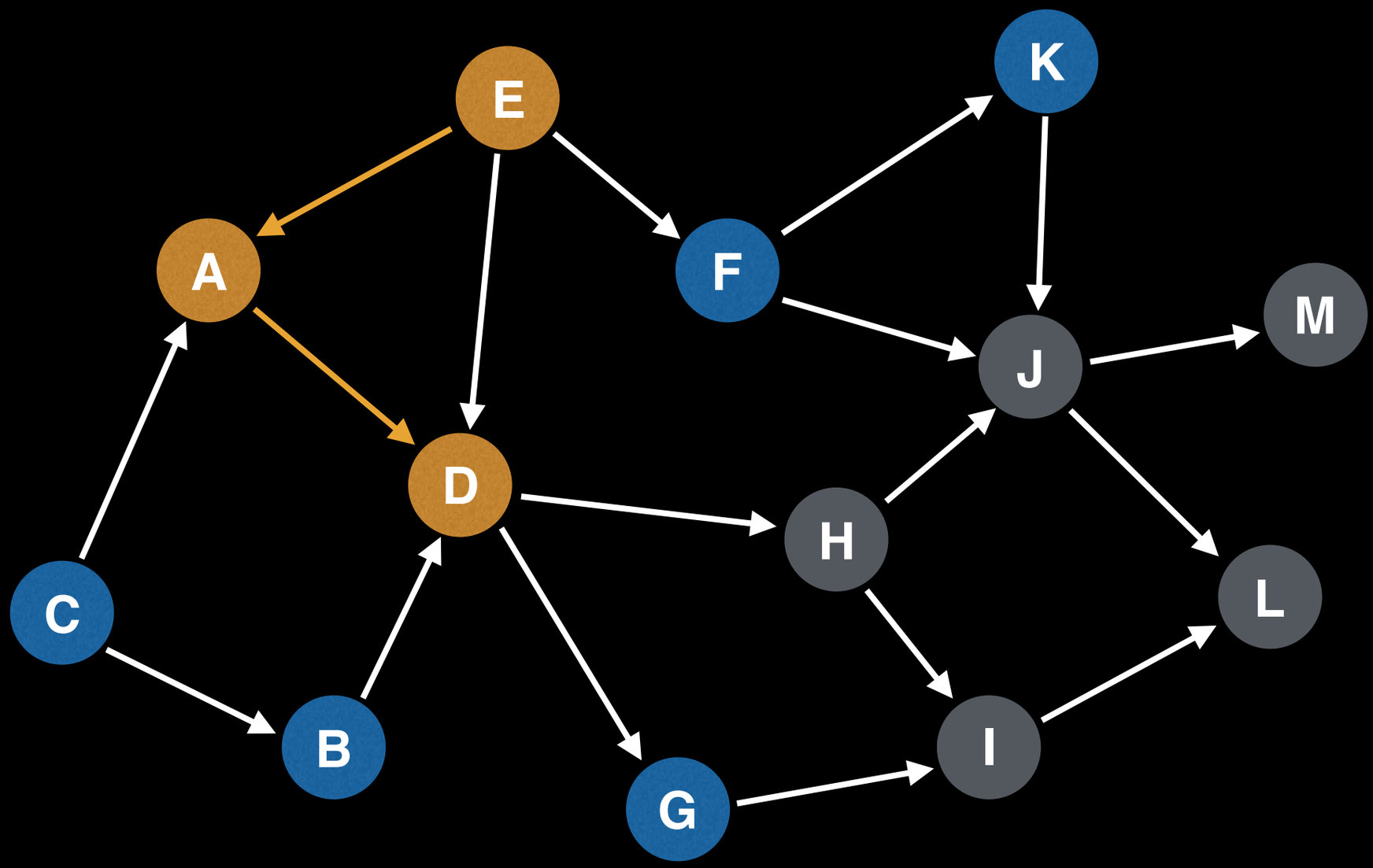
Topological ordering:

_ _ _ _ _ _ _ _ _ _ H I J L M

Topological Sort Algorithm

DFS recursion call stack:

Node E
Node A
Node D



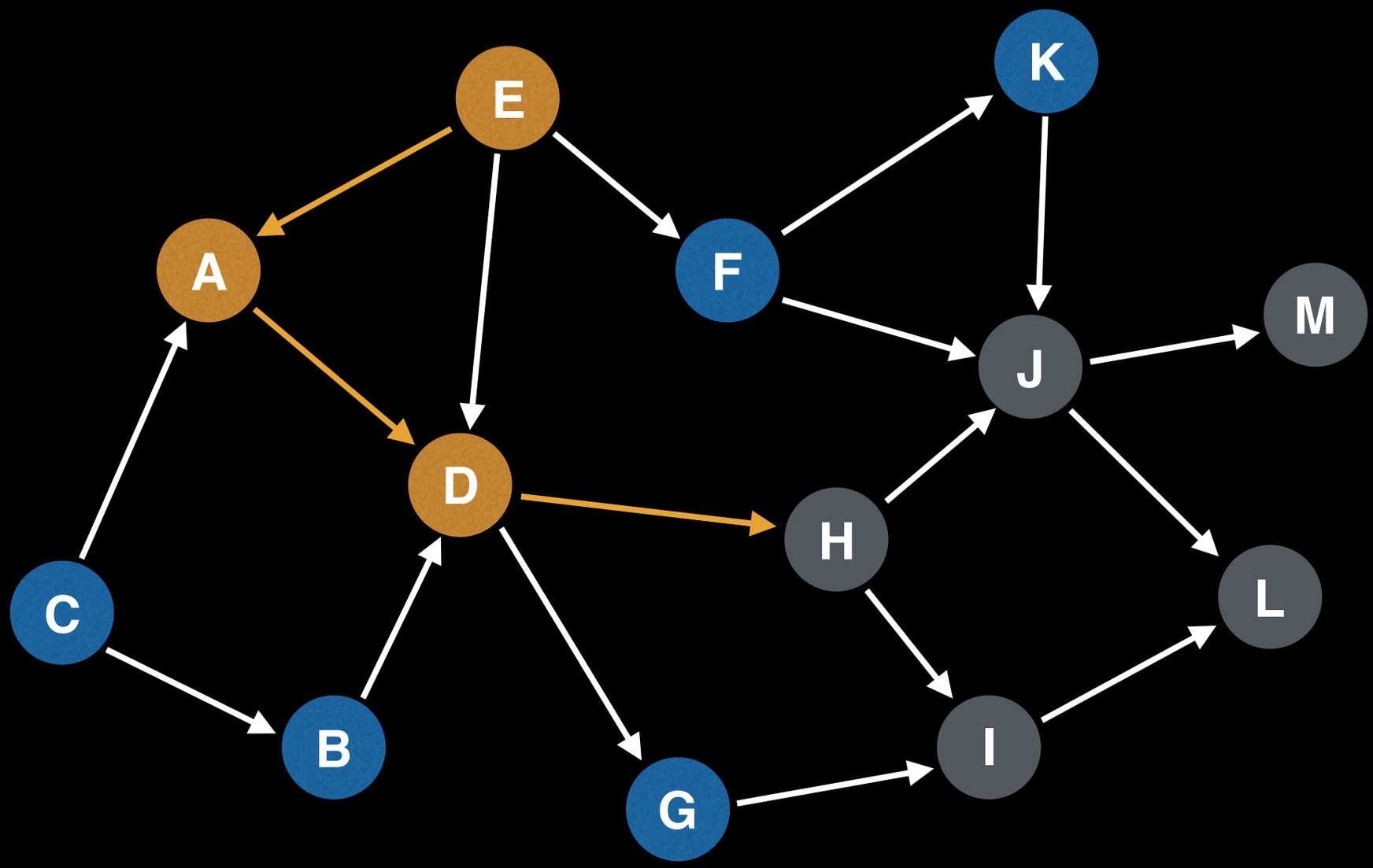
Topological ordering:

_ _ _ _ _ _ _ _ _ _ H I J L M

Topological Sort Algorithm

DFS recursion call stack:

Node E
Node A
Node D



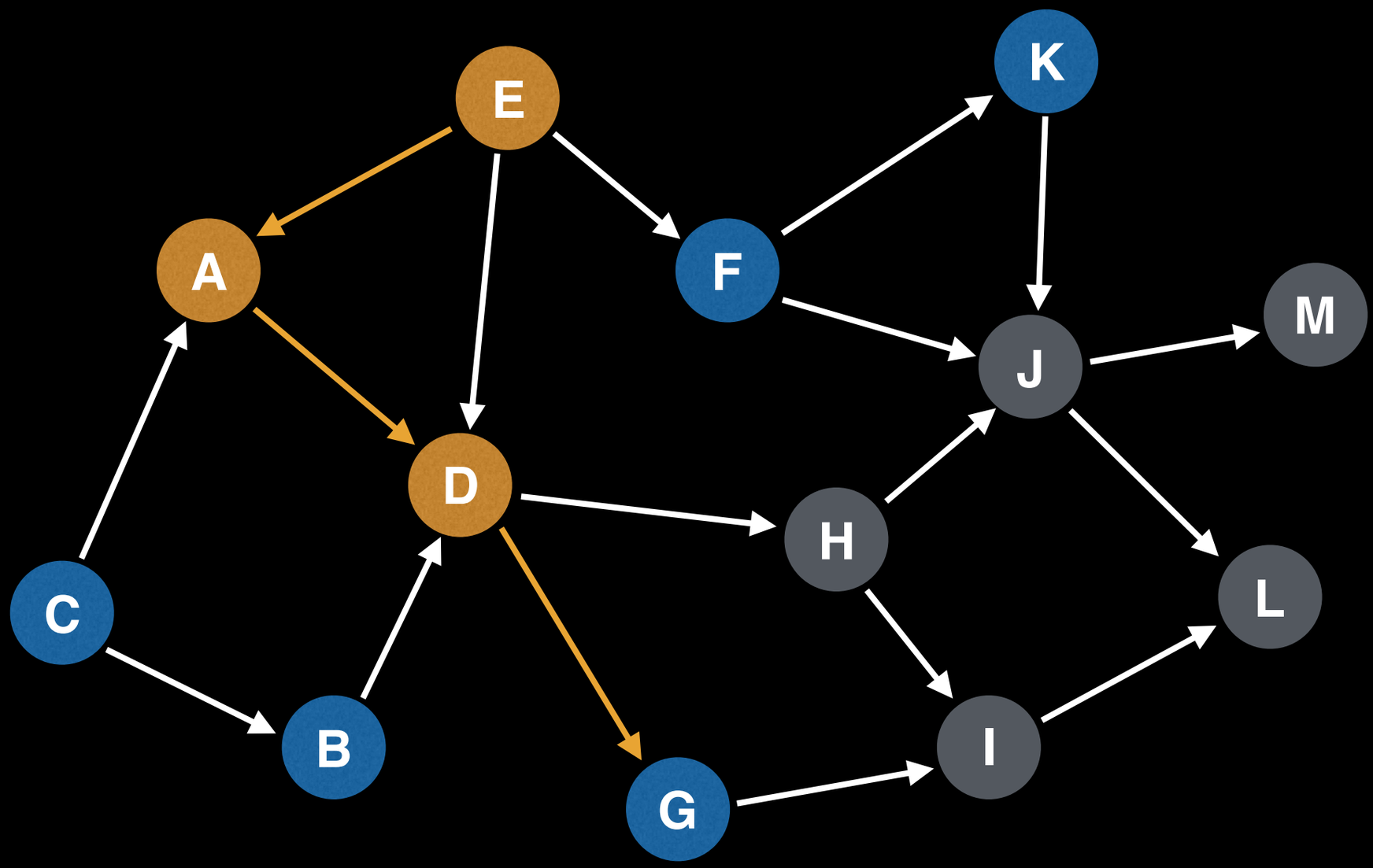
Topological ordering:

_ _ _ _ _ _ _ _ _ H I J L M

Topological Sort Algorithm

DFS recursion call stack:

Node E
Node A
Node D



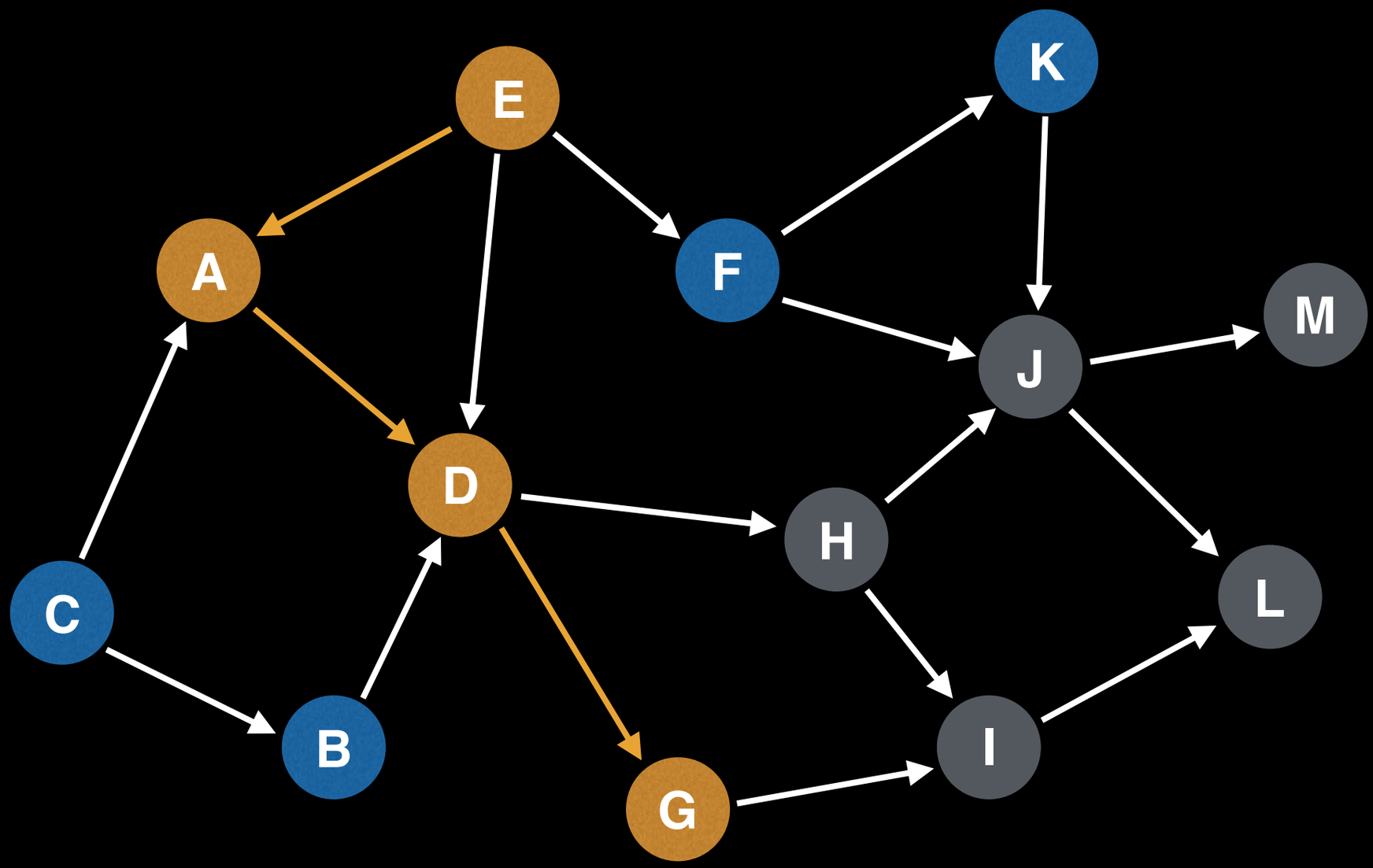
Topological ordering:

_ _ _ _ _ _ _ _ _ H I J L M

Topological Sort Algorithm

DFS recursion call stack:

Node E
Node A
Node D
Node G



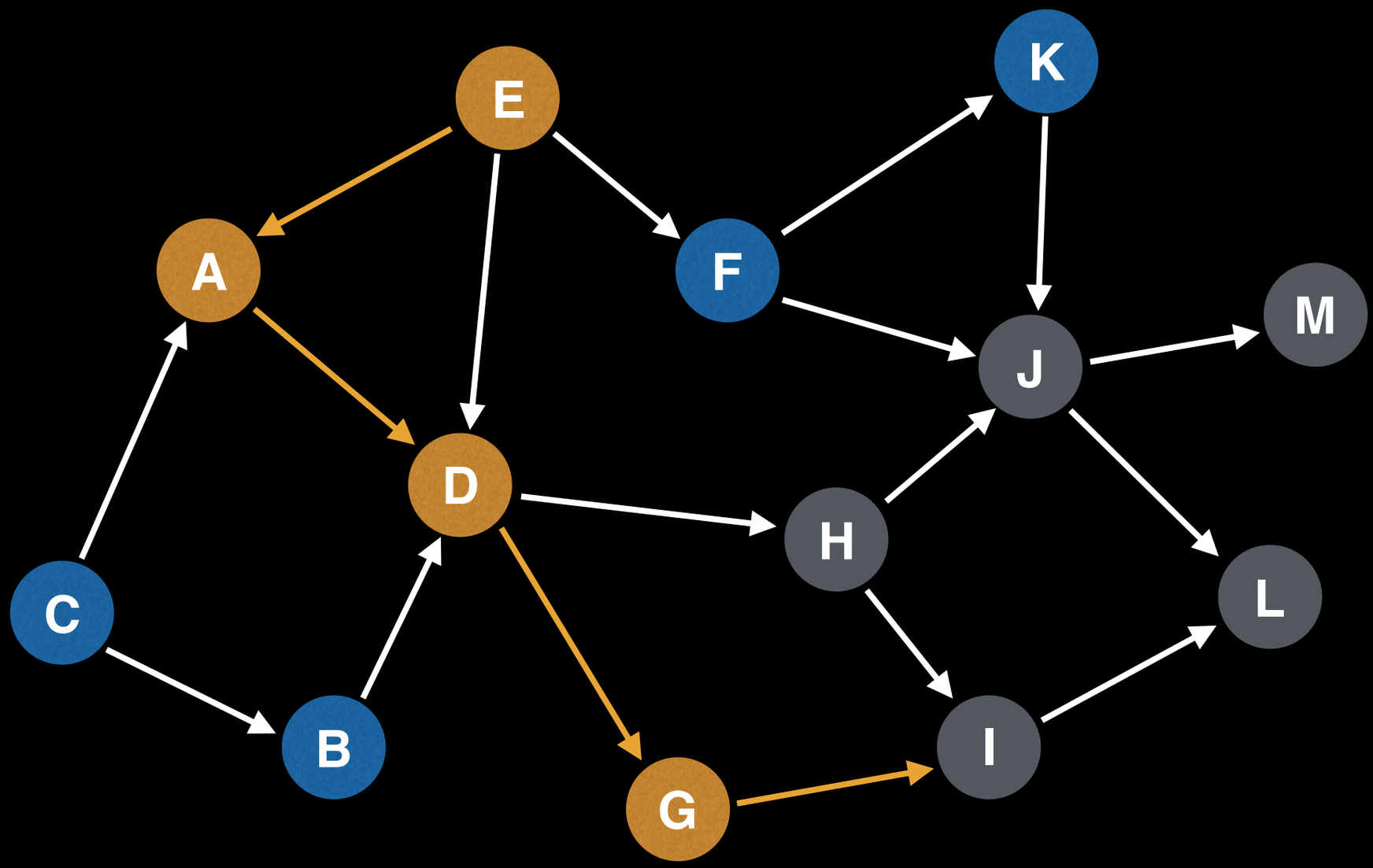
Topological ordering:

_ _ _ _ _ _ _ _ _ H I J L M

Topological Sort Algorithm

DFS recursion call stack:

Node E
Node A
Node D
Node G



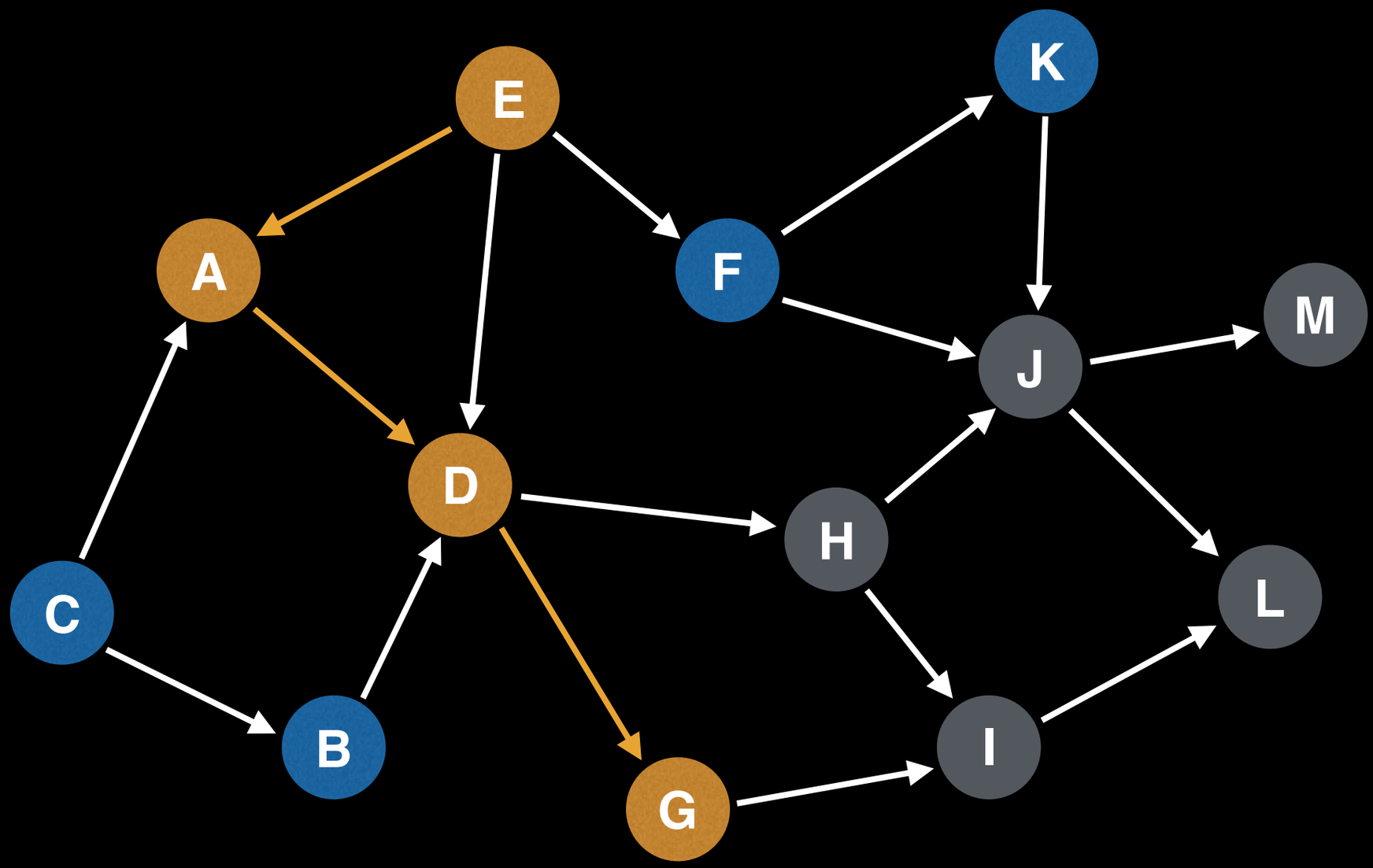
Topological ordering:

_ _ _ _ _ _ _ _ _ H I J L M

Topological Sort Algorithm

DFS recursion call stack:

Node E
Node A
Node D
Node G



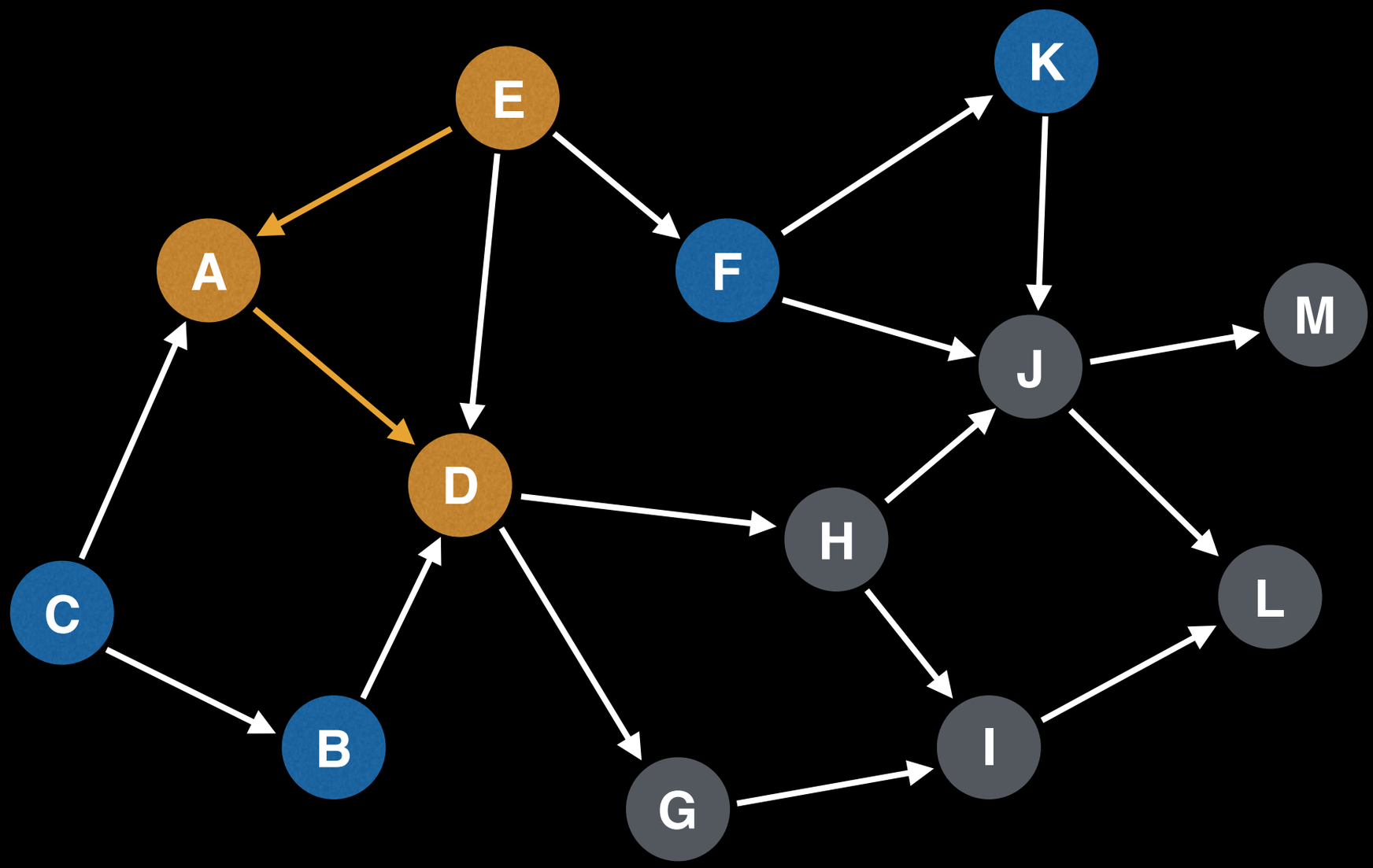
Topological ordering:

_ _ _ _ _ _ _ _ _ H I J L M

Topological Sort Algorithm

DFS recursion call stack:

Node E
Node A
Node D



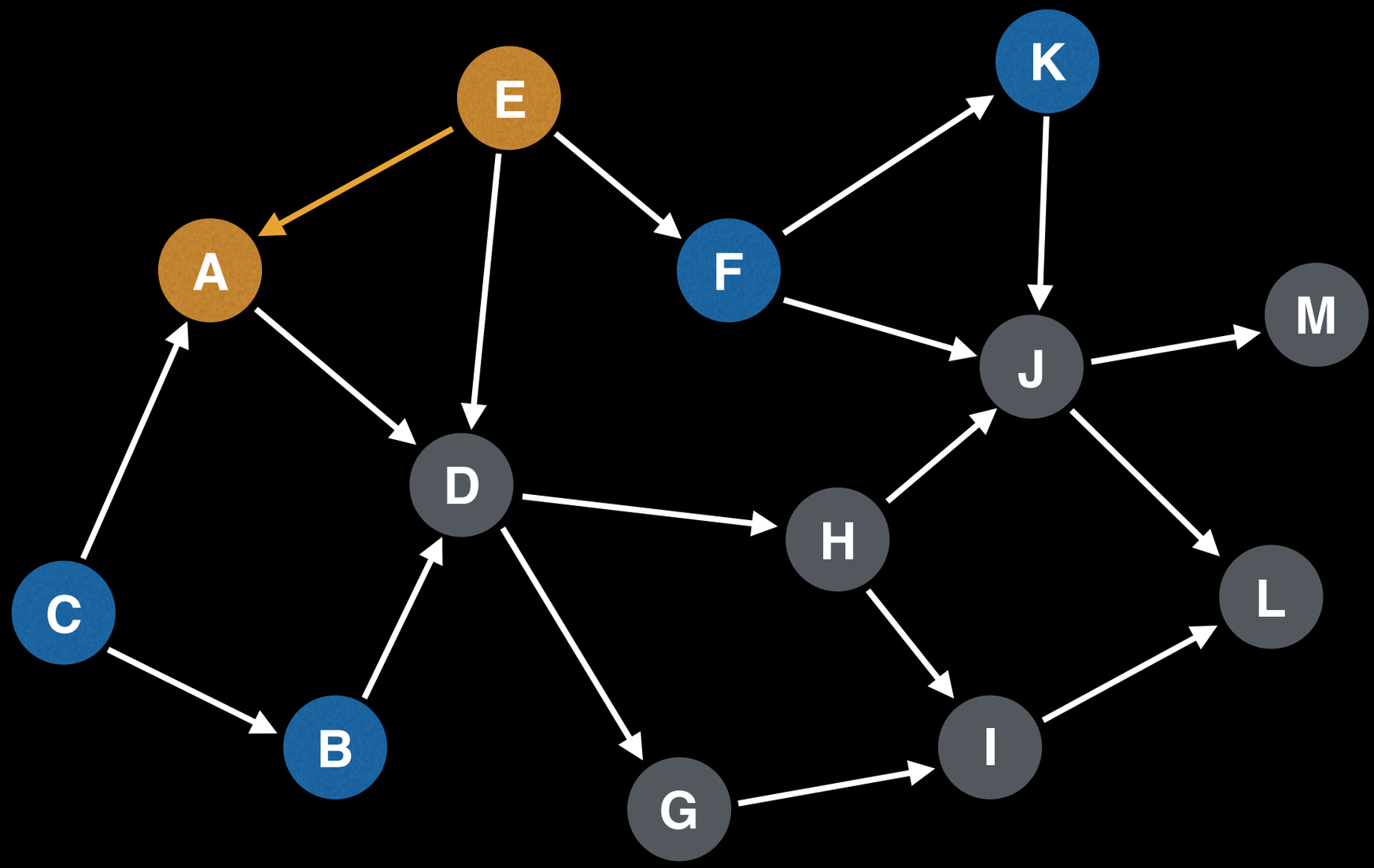
Topological ordering:

_ _ _ _ _ _ _ _ _ G H I J L M

Topological Sort Algorithm

DFS recursion call stack:

Node E
Node A



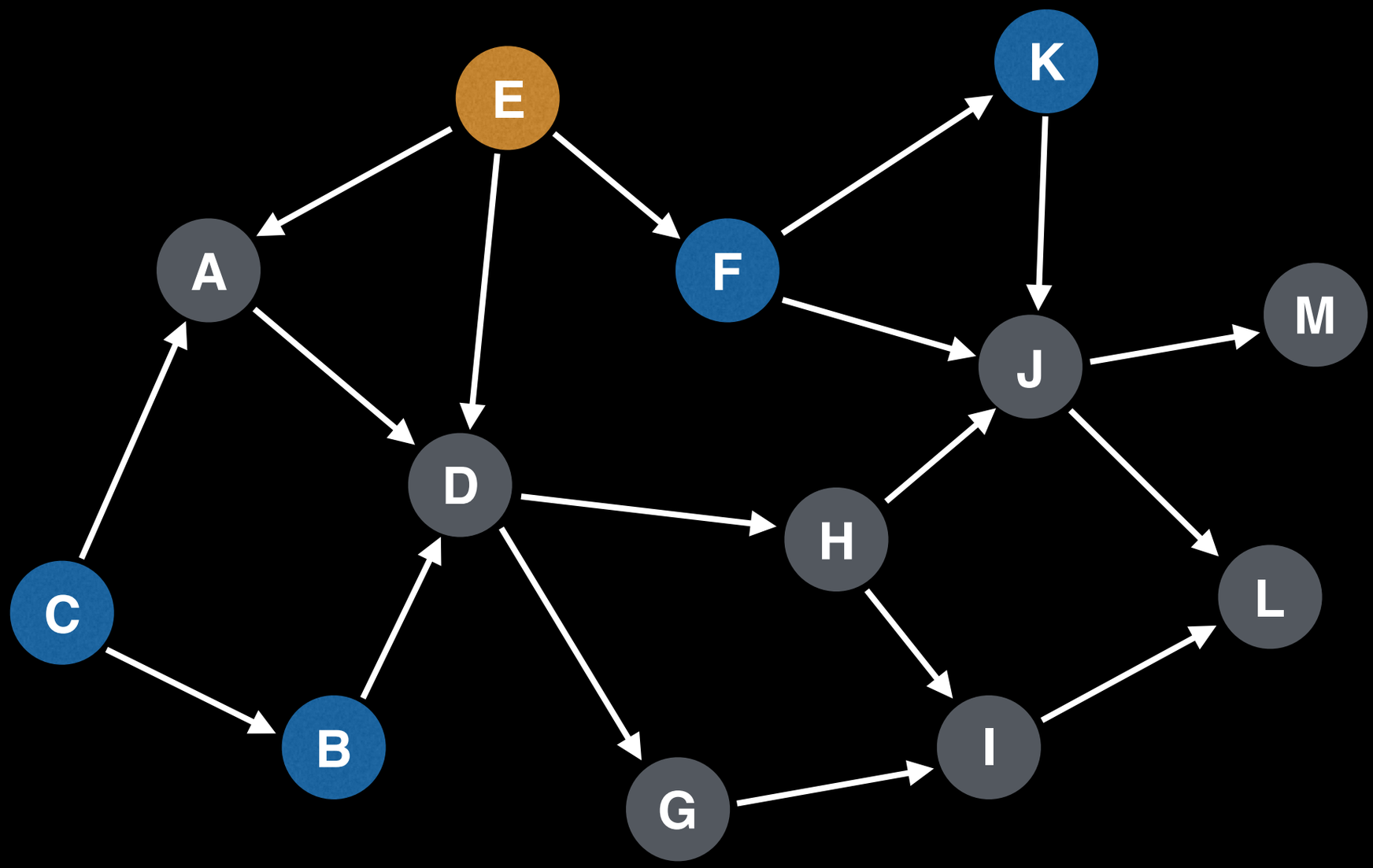
Topological ordering:

_ _ _ _ _ _ _ D G H I J L M

Topological Sort Algorithm

DFS recursion call stack:

Node E



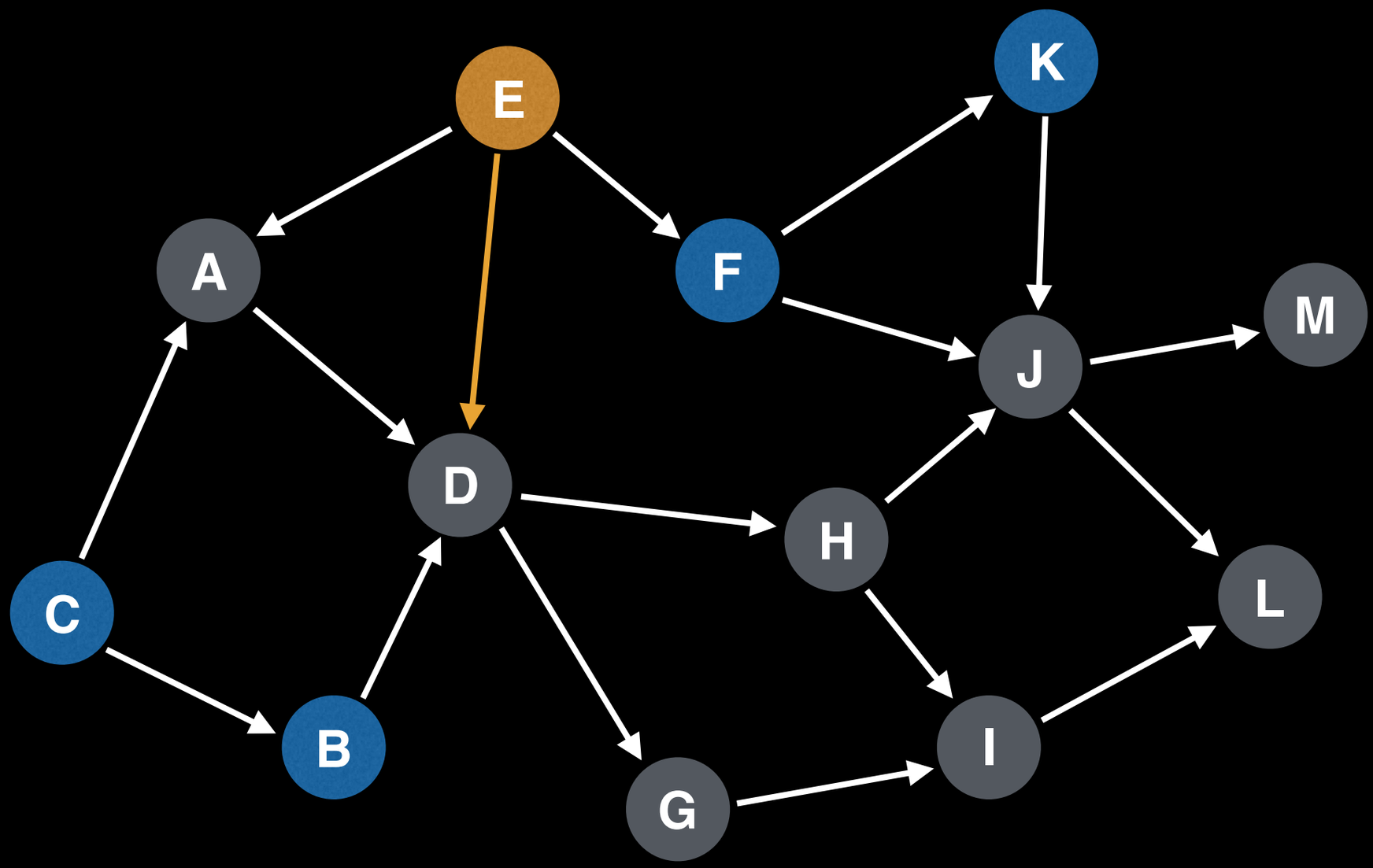
Topological ordering:

_ _ _ _ _ _ A D G H I J L M

Topological Sort Algorithm

DFS recursion call stack:

Node E



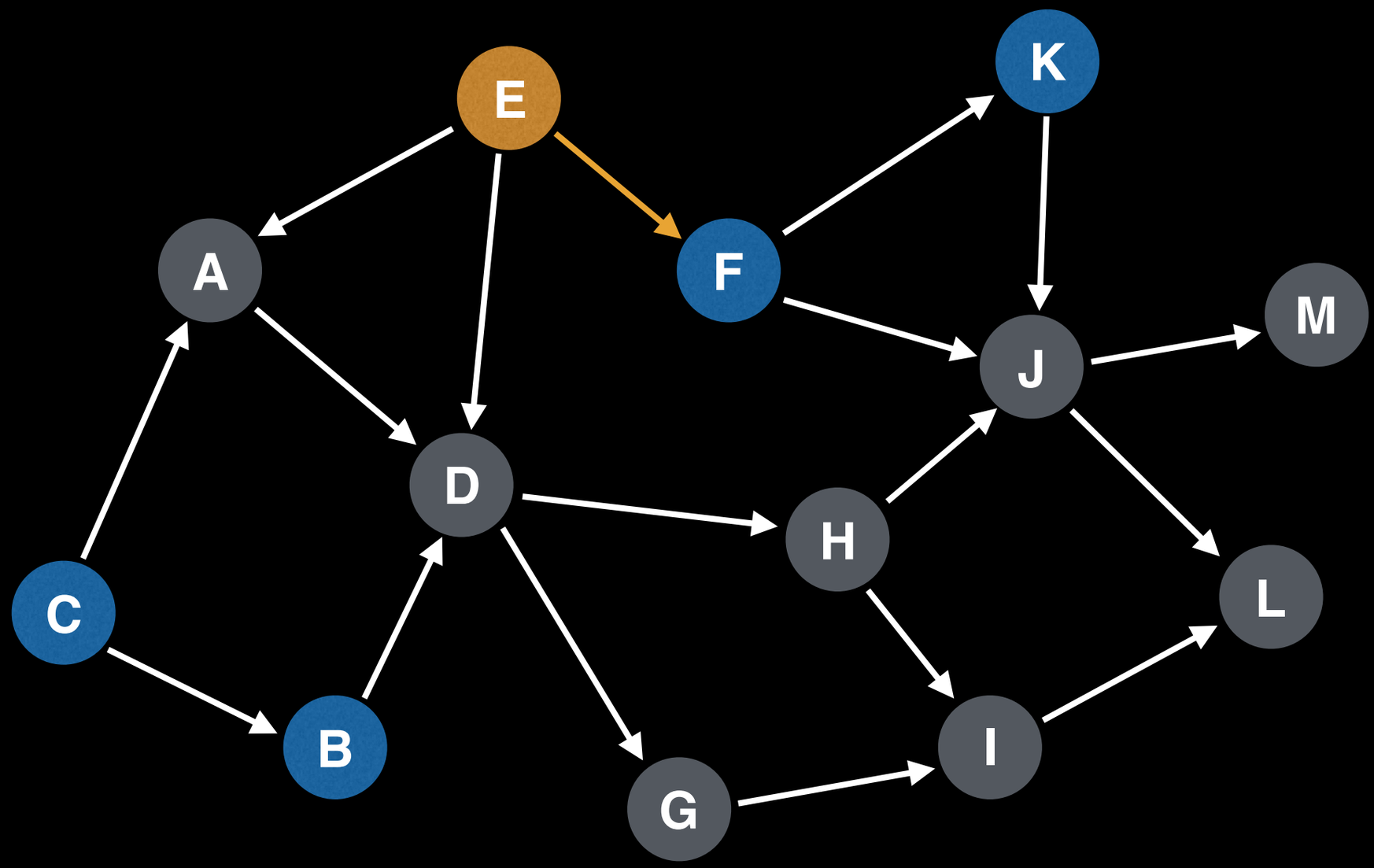
Topological ordering:

_ _ _ _ _ _ A D G H I J L M

Topological Sort Algorithm

DFS recursion call stack:

Node E



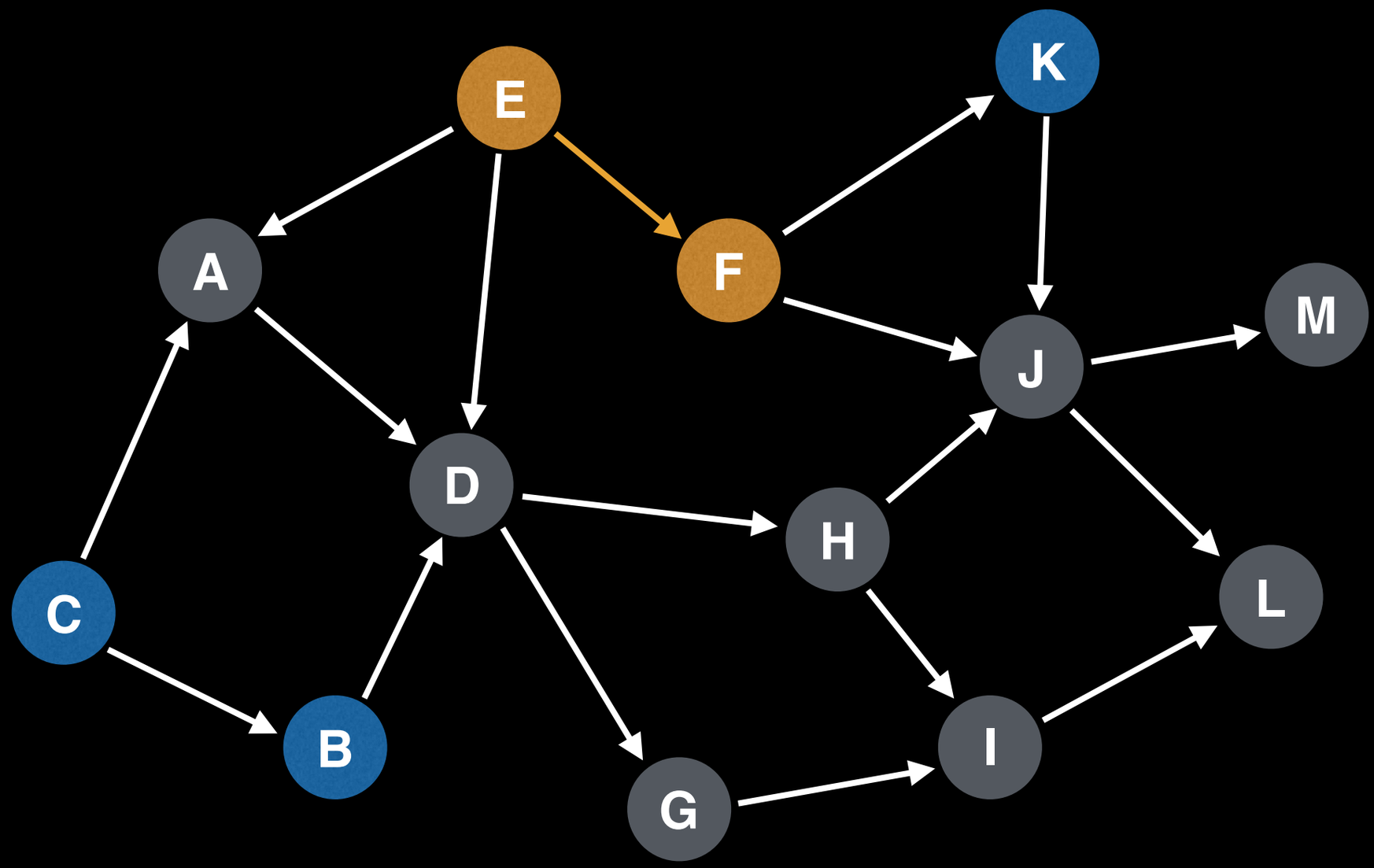
Topological ordering:

_ _ _ _ _ _ A D G H I J L M

Topological Sort Algorithm

DFS recursion call stack:

Node E
Node F



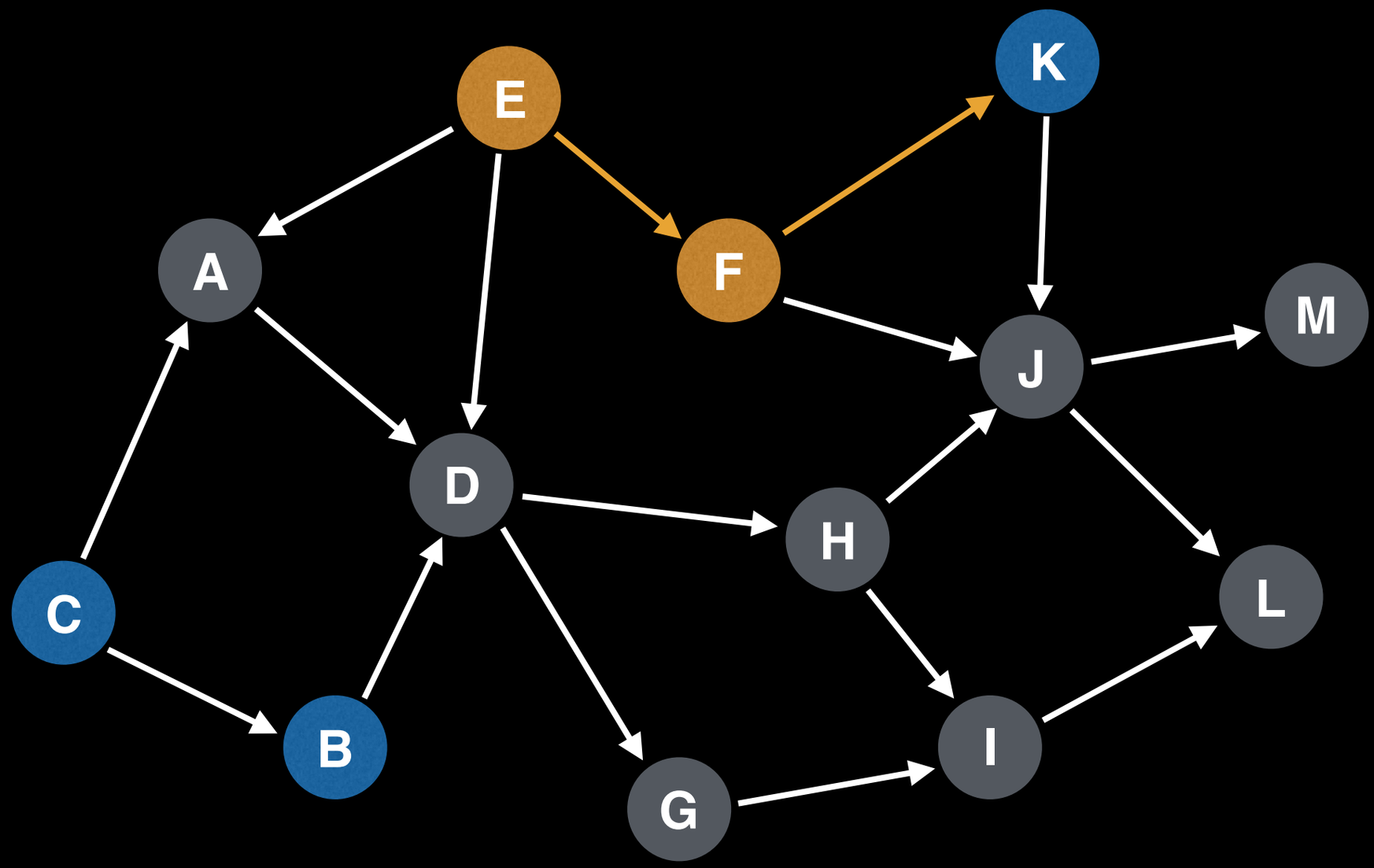
Topological ordering:

_ _ _ _ _ _ A D G H I J L M

Topological Sort Algorithm

DFS recursion call stack:

Node E
Node F



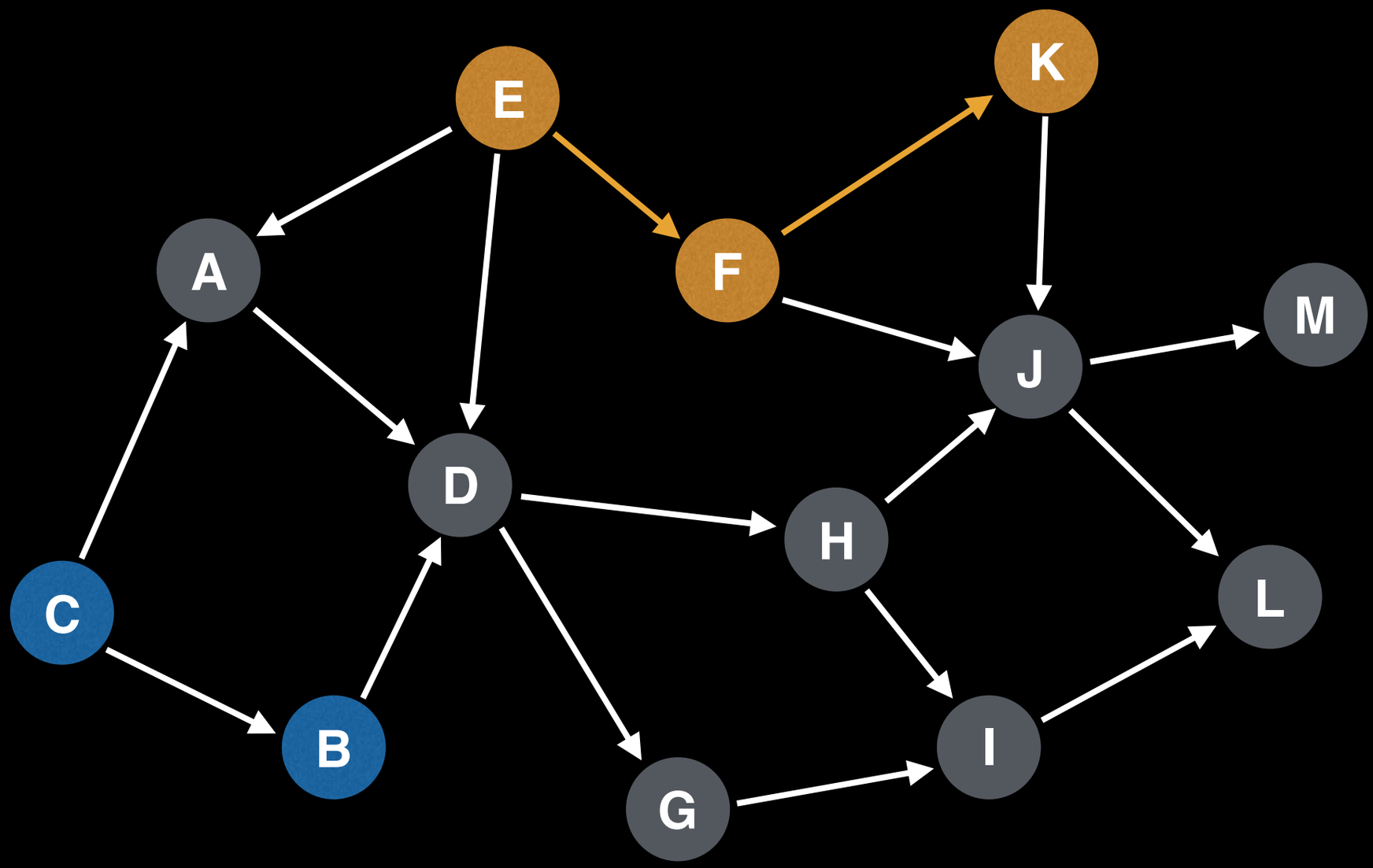
Topological ordering:

_ _ _ _ _ _ A D G H I J L M

Topological Sort Algorithm

DFS recursion call stack:

Node E
Node F
Node K



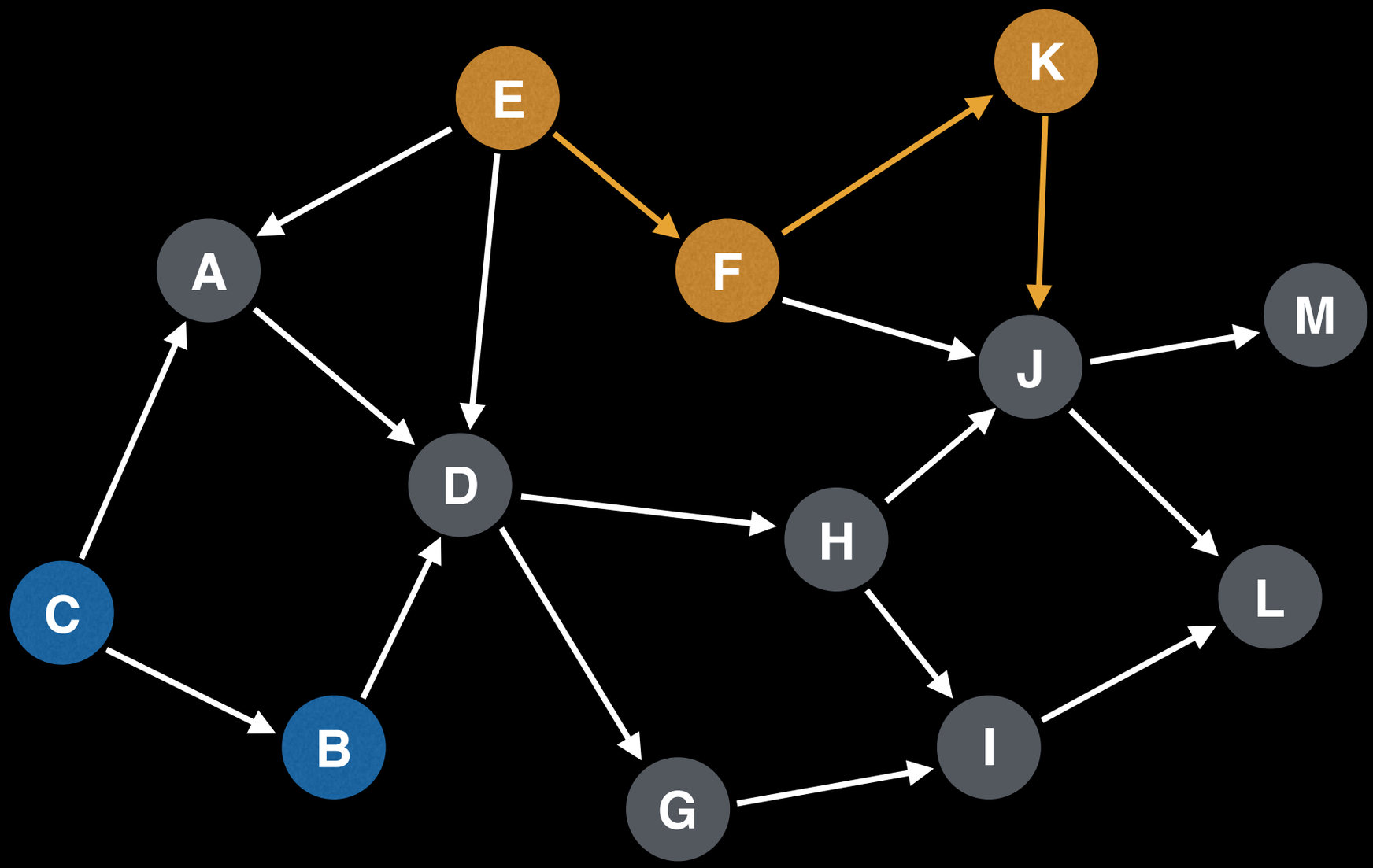
Topological ordering:

_ _ _ _ _ _ A D G H I J L M

Topological Sort Algorithm

DFS recursion call stack:

Node E
Node F
Node K



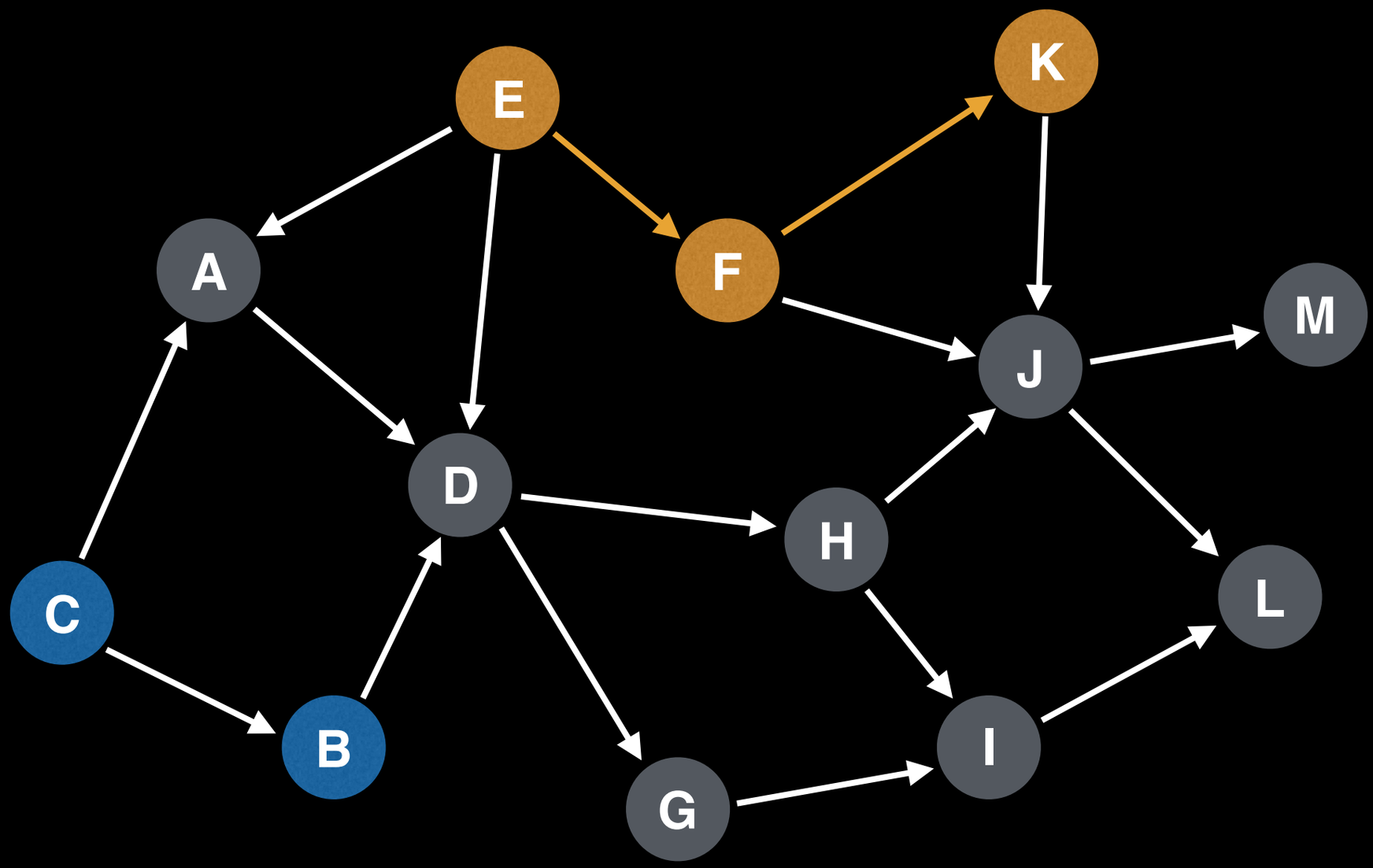
Topological ordering:

_ _ _ _ _ _ A D G H I J L M

Topological Sort Algorithm

DFS recursion call stack:

Node E
Node F
Node K



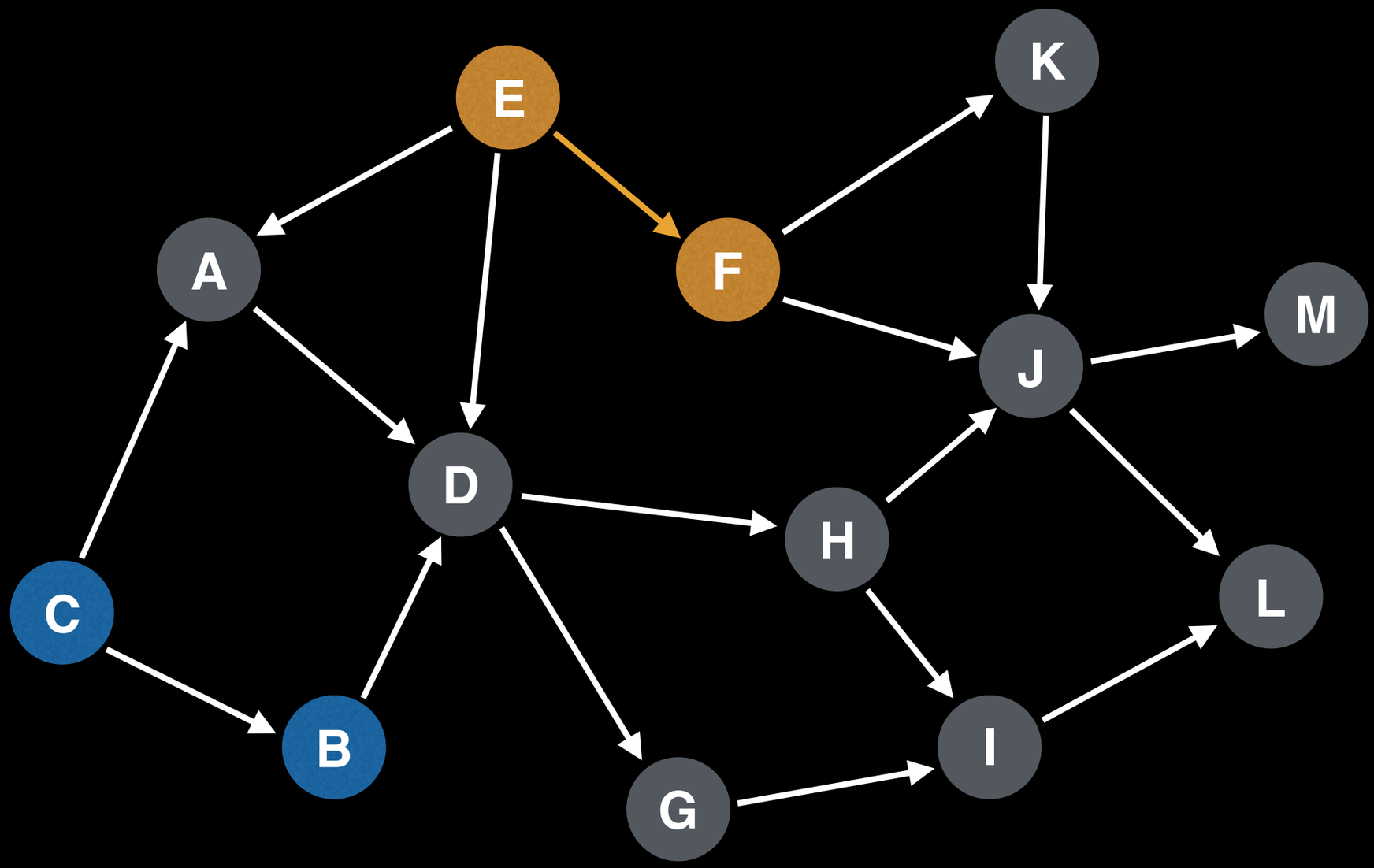
Topological ordering:

_ _ _ _ _ _ A D G H I J L M

Topological Sort Algorithm

DFS recursion call stack:

Node E
Node F



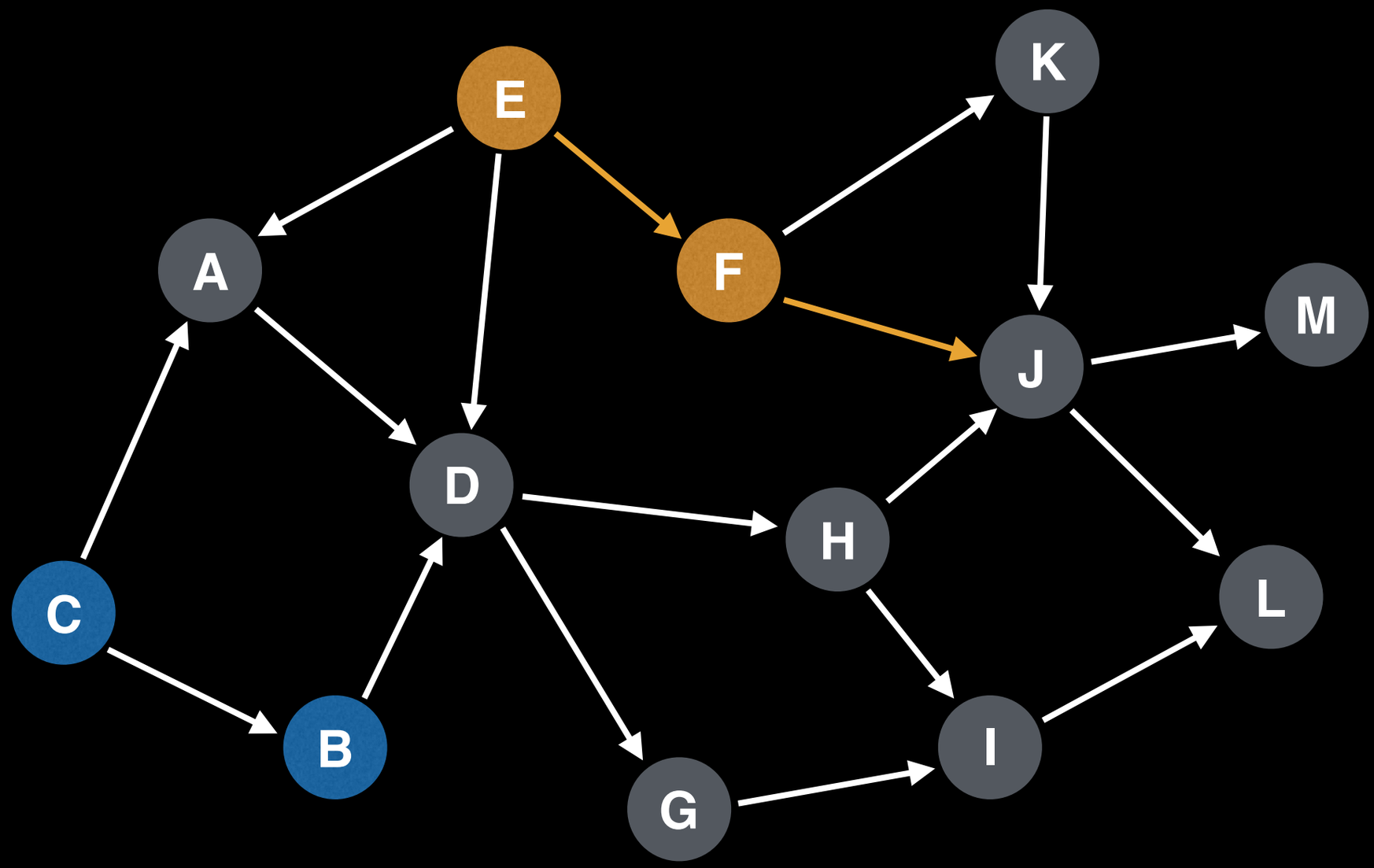
Topological ordering:

_ _ _ _ _ K A D G H I J L M

Topological Sort Algorithm

DFS recursion call stack:

Node E
Node F



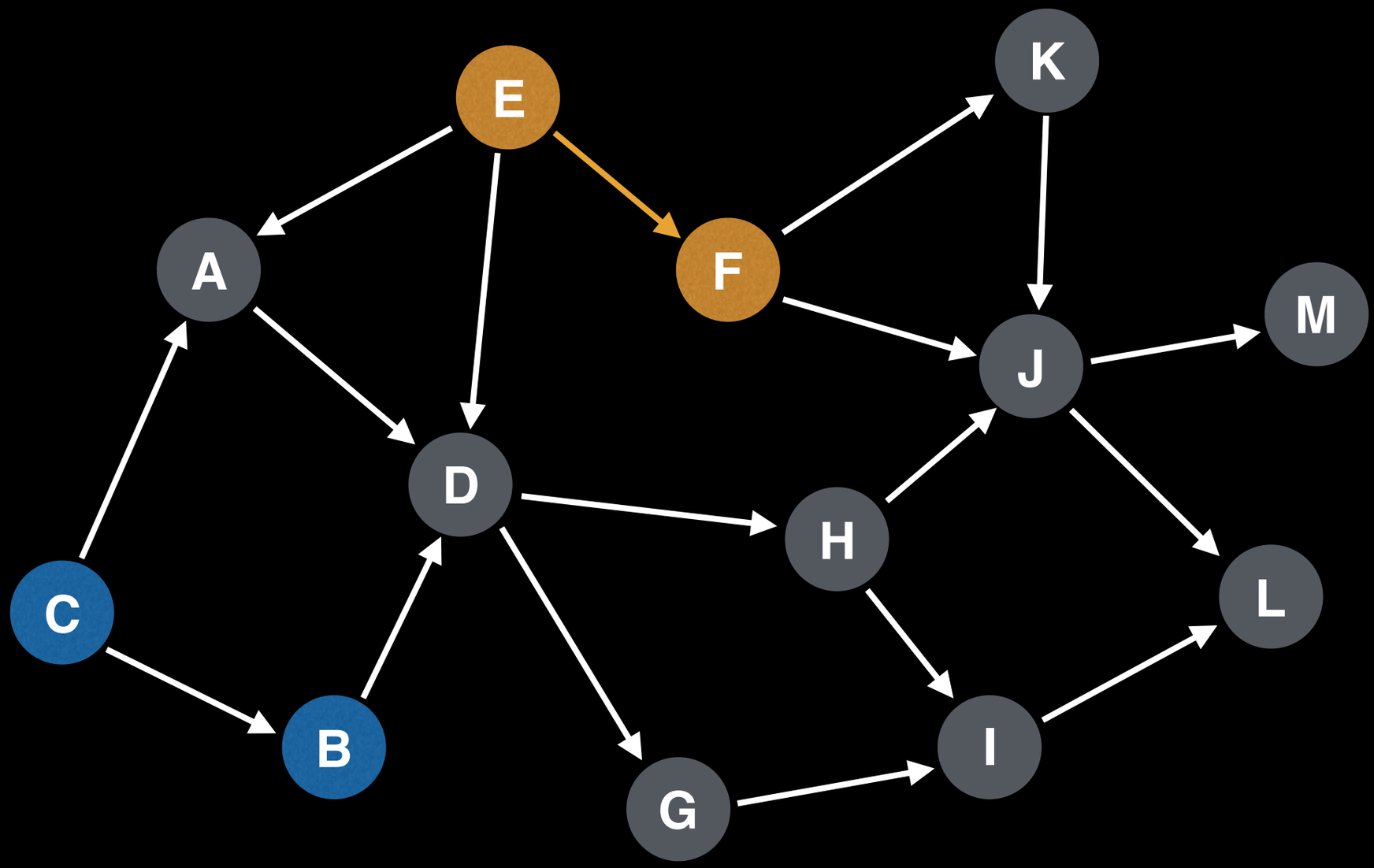
Topological ordering:

_ _ _ _ _ K A D G H I J L M

Topological Sort Algorithm

DFS recursion call stack:

Node E
Node F



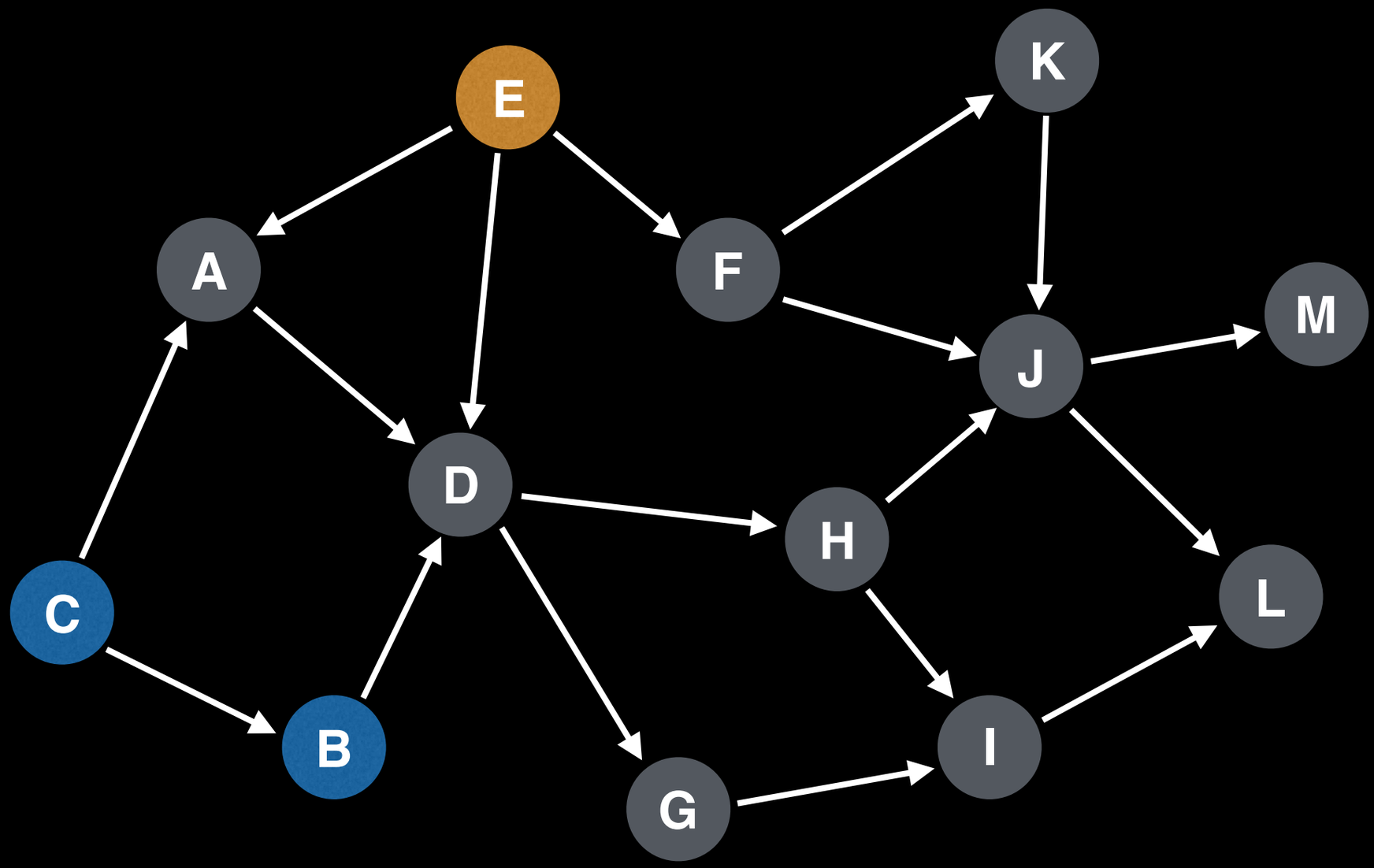
Topological ordering:

_ _ _ _ _ K A D G H I J L M

Topological Sort Algorithm

DFS recursion call stack:

Node E

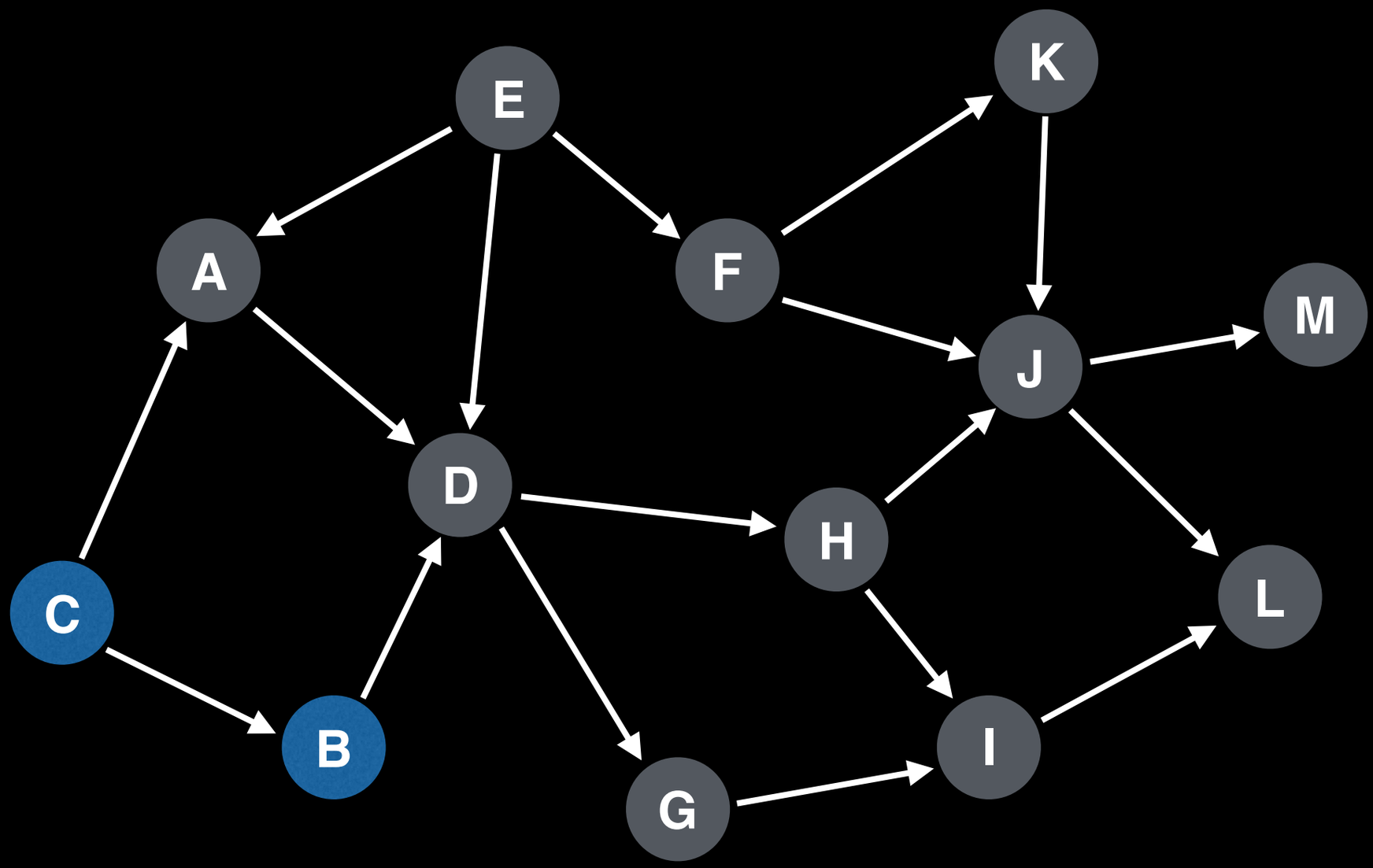


Topological ordering:

_ _ _ F K A D G H I J L M

Topological Sort Algorithm

DFS recursion call stack:



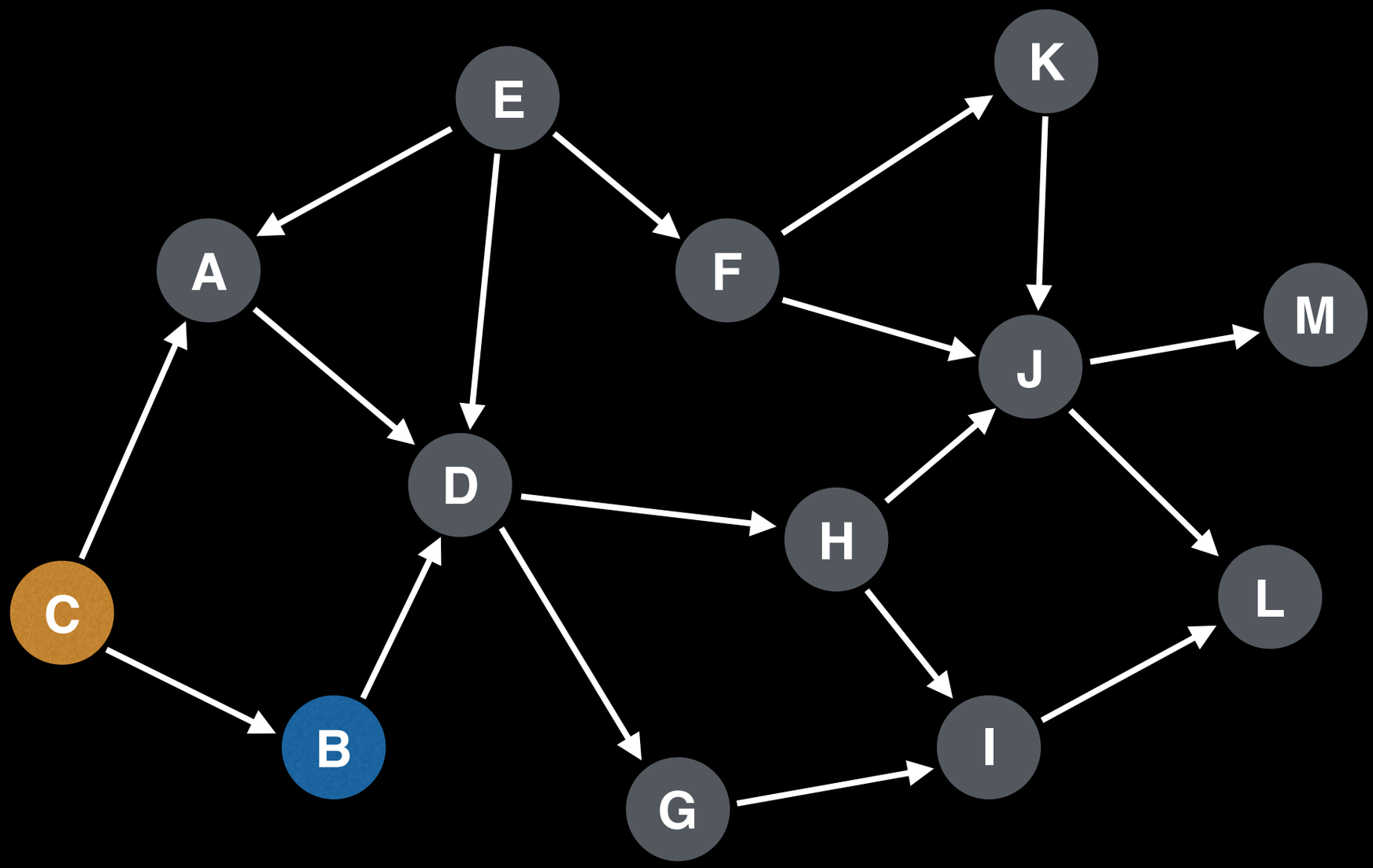
Topological ordering:

_ _ E F K A D G H I J L M

Topological Sort Algorithm

DFS recursion call stack:

Node C



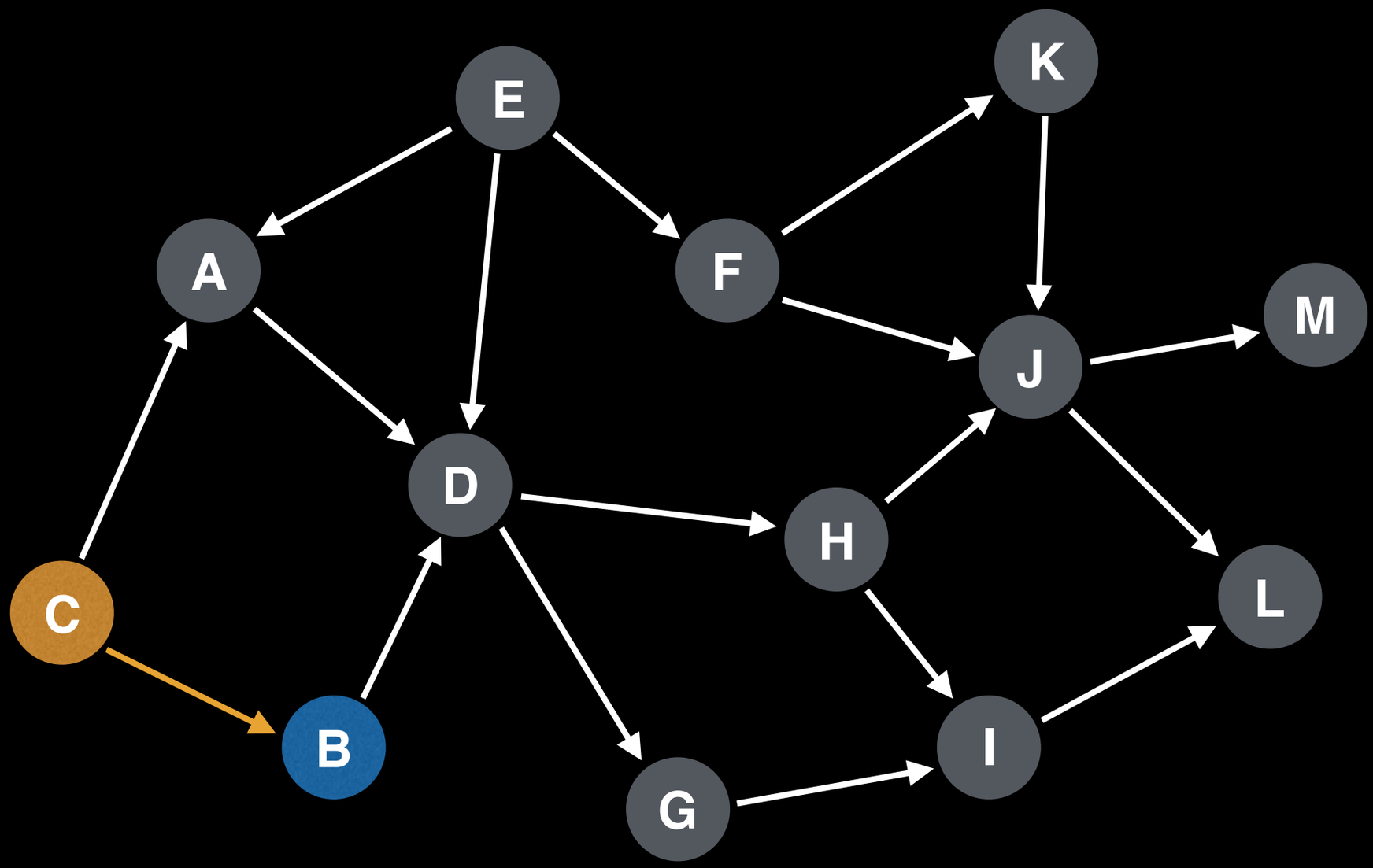
Topological ordering:

_ _ E F K A D G H I J L M

Topological Sort Algorithm

DFS recursion call stack:

Node C



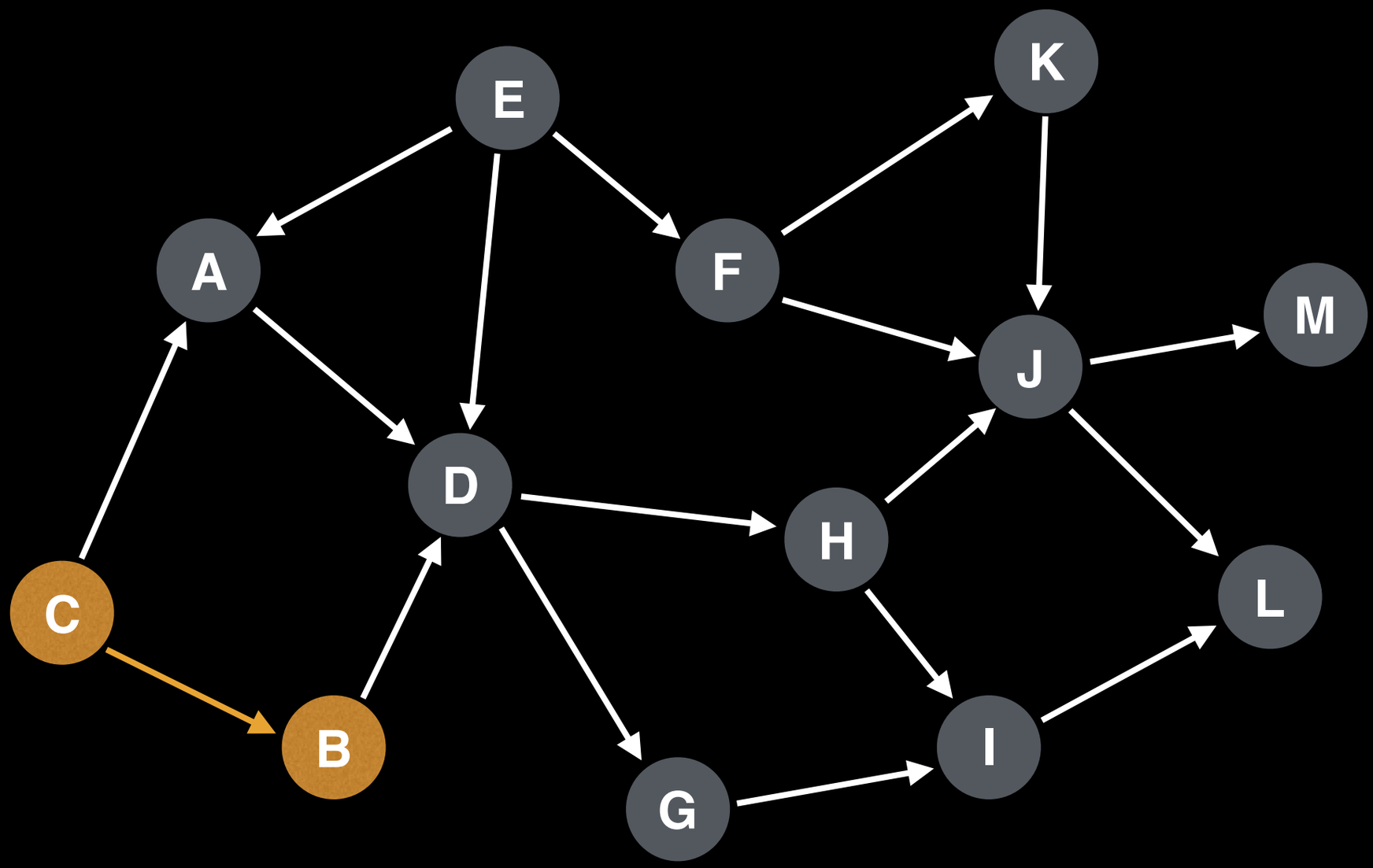
Topological ordering:

_ _ E F K A D G H I J L M

Topological Sort Algorithm

DFS recursion call stack:

Node C
Node B



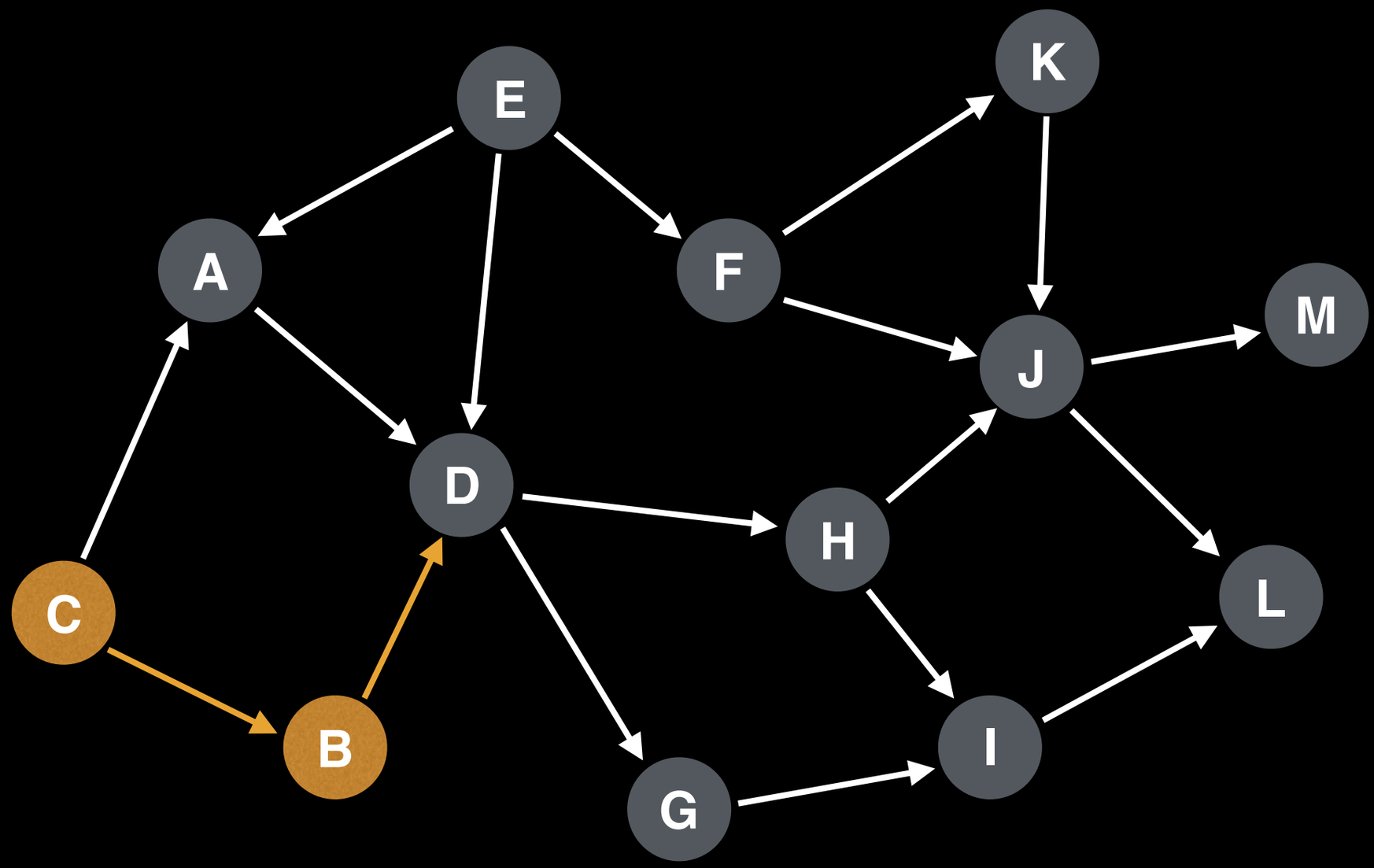
Topological ordering:

_ _ E F K A D G H I J L M

Topological Sort Algorithm

DFS recursion call stack:

Node C
Node B



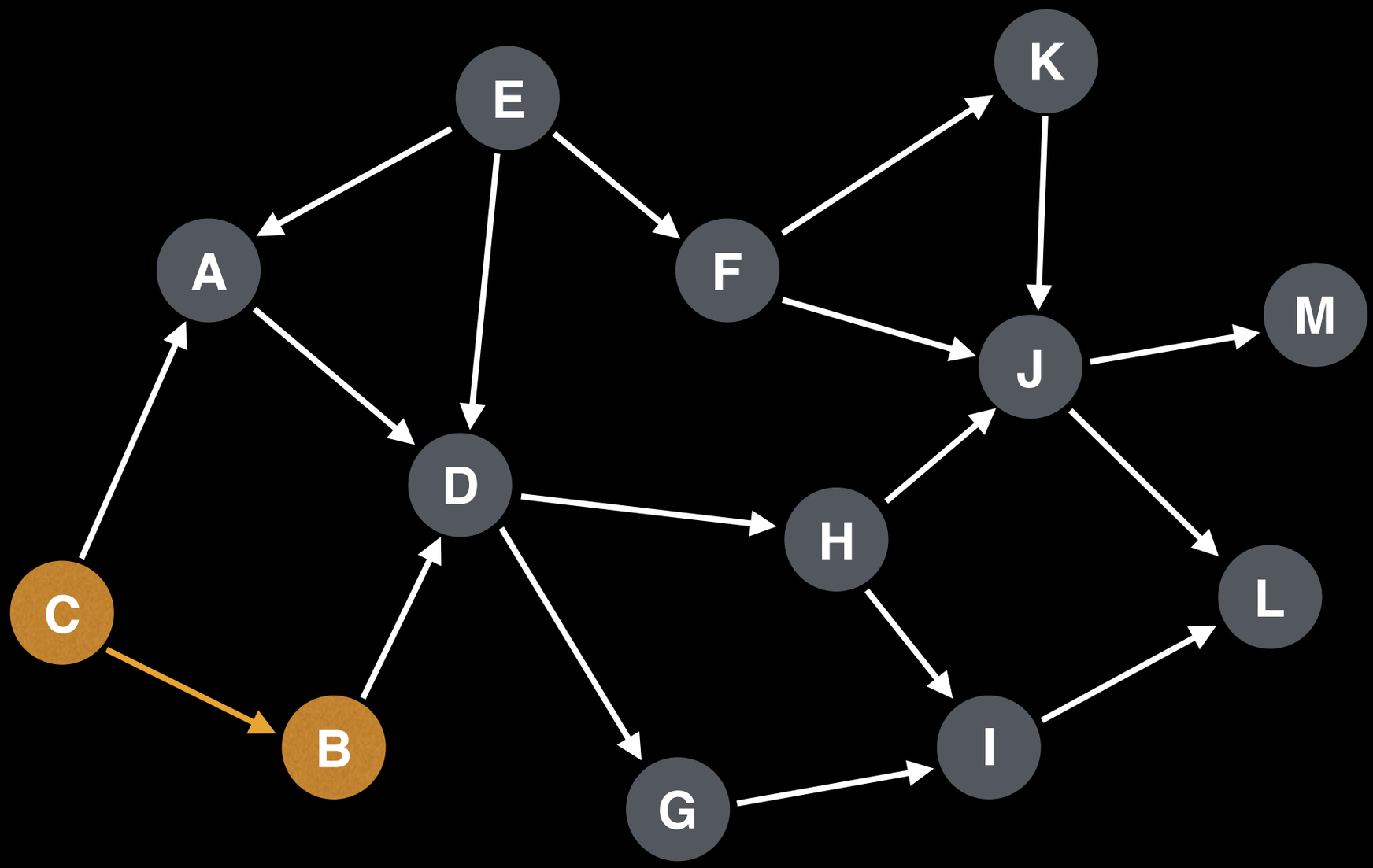
Topological ordering:

_ _ E F K A D G H I J L M

Topological Sort Algorithm

DFS recursion call stack:

Node C
Node B



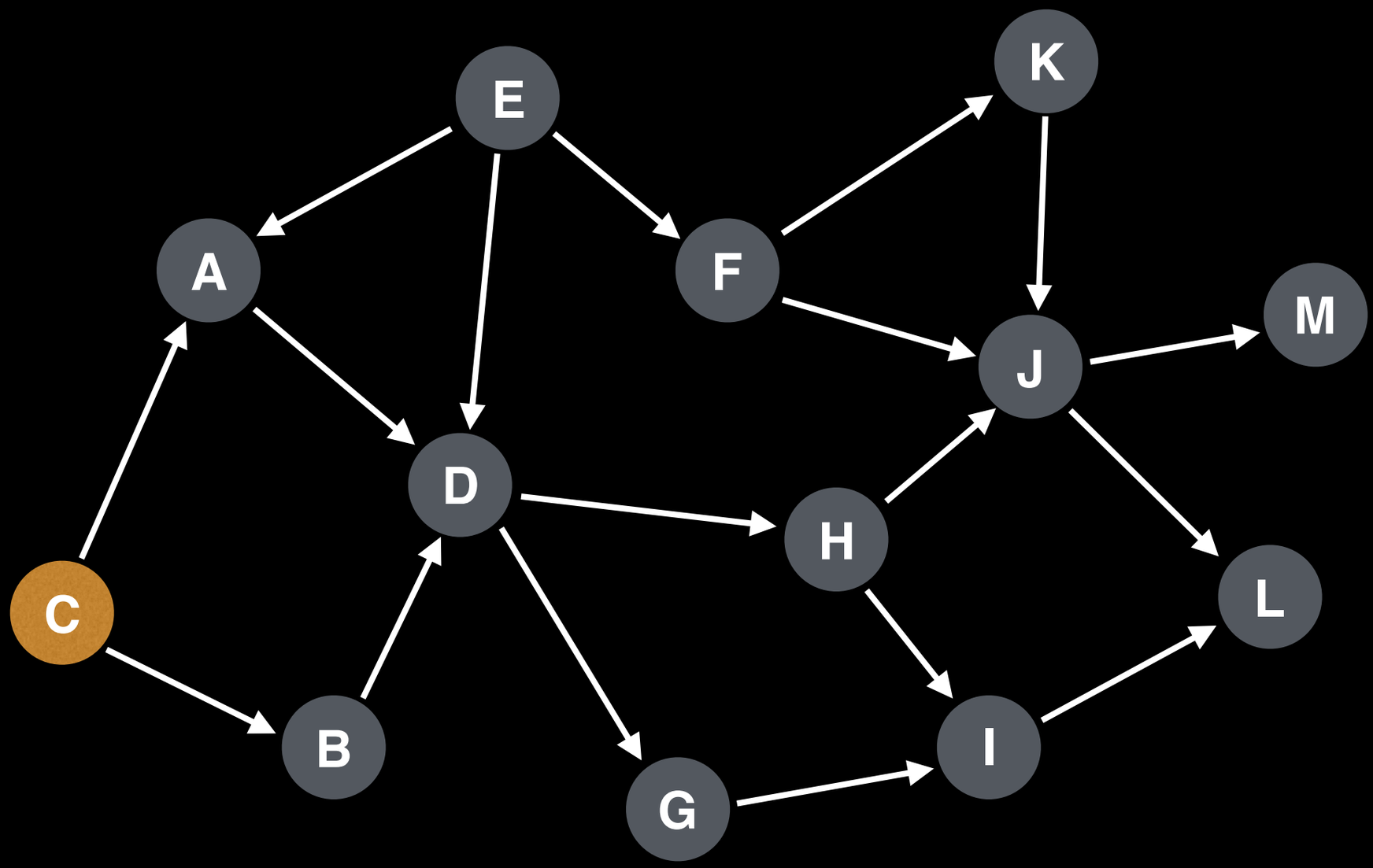
Topological ordering:

_ _ E F K A D G H I J L M

Topological Sort Algorithm

DFS recursion call stack:

Node C



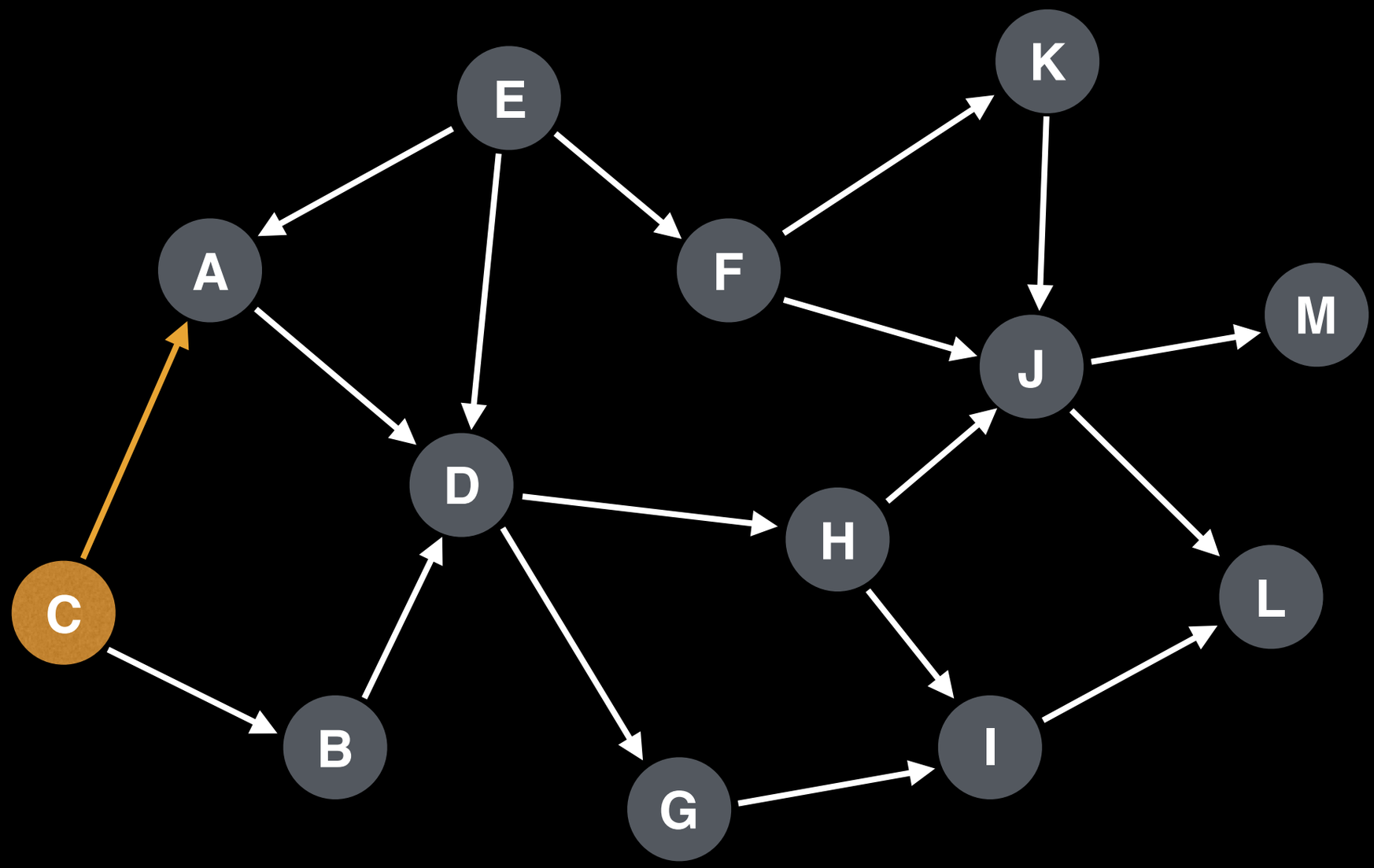
Topological ordering:

_ B E F K A D G H I J L M

Topological Sort Algorithm

DFS recursion call stack:

Node C



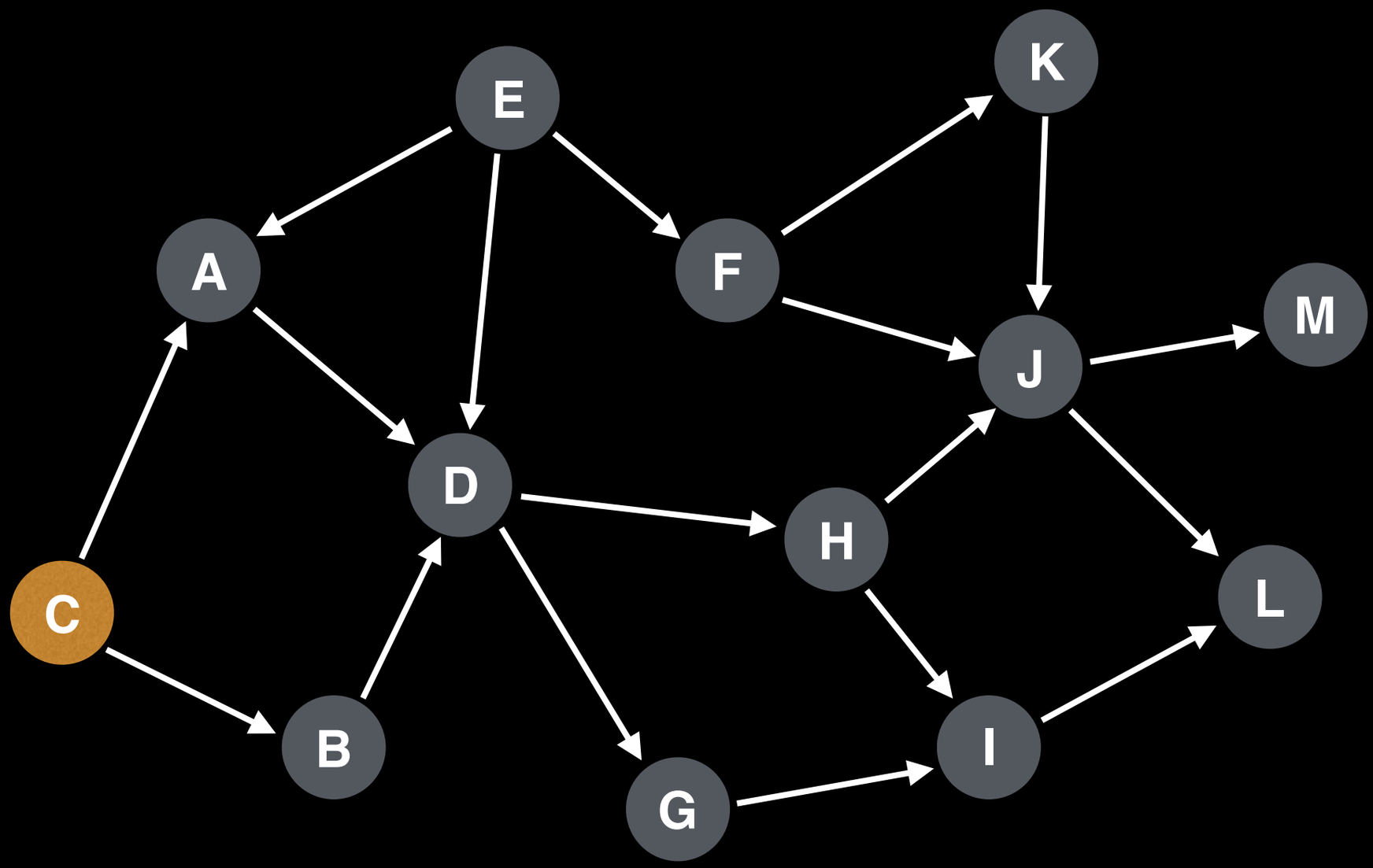
Topological ordering:

_ B E F K A D G H I J L M

Topological Sort Algorithm

DFS recursion call stack:

Node C

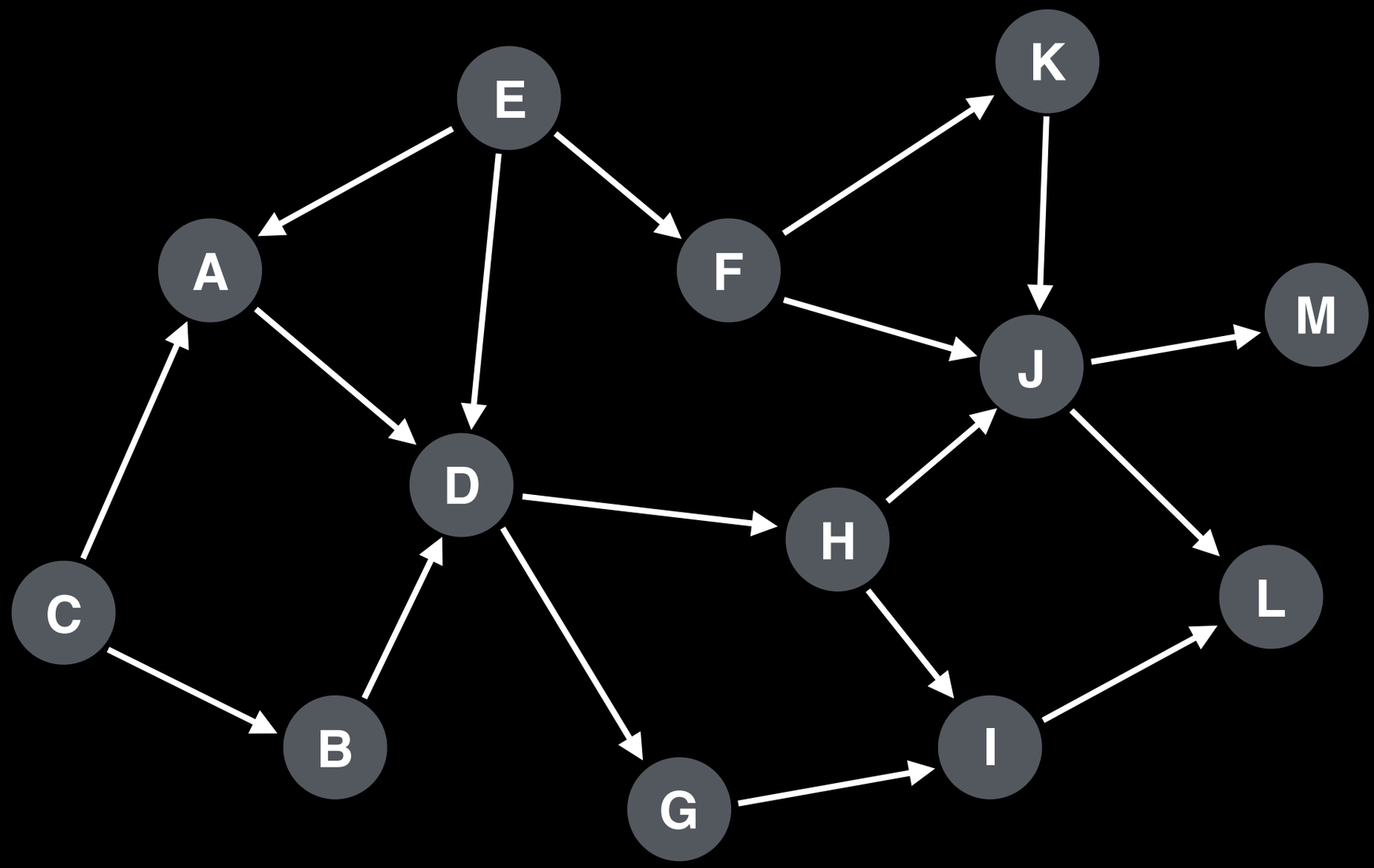


Topological ordering:

_ B E F K A D G H I J L M

Topological Sort Algorithm

DFS recursion call stack:



Topological ordering:

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

Topsort pseudocode

```
# Assumption: graph is stored as adjacency list
function topsort(graph):

  N = graph.numberOfNodes()
  V = [false,…,false] # Length N
  ordering = [0,…,0]  # Length N
  i = N - 1 # Index for ordering array

  for(at = 0; at < N; at++):
    if V[at] == false:
      visitedNodes = []
      dfs(at, V, visitedNodes, graph)
      for nodeId in visitedNodes:
        ordering[i] = nodeId
        i = i - 1
  return ordering
```

Topsort pseudocode

```
# Execute Depth First Search (DFS)
function dfs(at, V, visitedNodes, graph):

  V[at] = true

  edges = graph.getEdgesOutFromNode(at)
  for edge in edges:
    if V[edge.to] == false:
      dfs(edge.to, V, visitedNodes, graph)

  visitedNodes.add(at)
```

Topsort pseudocode

```
# Assumption: graph is stored as adjacency list
function topsort(graph):

  N = graph.numberOfNodes()
  V = [false,…,false] # Length N
  ordering = [0,…,0]  # Length N
  i = N - 1 # Index for ordering array

  for(at = 0; at < N; at++):
    if V[at] == false:
      visitedNodes = []
      dfs(at, V, visitedNodes, graph)
      for nodeId in visitedNodes:
        ordering[i] = nodeId
        i = i - 1
  return ordering
```

Topsort pseudocode

```
# Assumption: graph is stored as adjacency list
function topsort(graph):

  N = graph.numberOfNodes()
  V = [false,…,false] # Length N
  ordering = [0,…,0]  # Length N
  i = N - 1 # Index for ordering array

  for(at = 0; at < N; at++):
    if V[at] == false:
      visitedNodes = []
      dfs(at, V, visitedNodes, graph)
      for nodeId in visitedNodes:
        ordering[i] = nodeId
        i = i - 1
  return ordering
```

Topsort Optimization

```
# Assumption: graph is stored as adjacency list
function topsort(graph):

  N = graph.numberOfNodes()
  V = [false,…,false] # Length N
  ordering = [0,…,0]  # Length N
  i = N - 1 # Index for ordering array

  for(at = 0; at < N; at++):
    if V[at] == false:
      i = dfs(i, at, V, ordering, graph)

  return ordering
```

Topsort Optimization

```
# Execute Depth First Search (DFS)
function dfs(i, at, V, ordering, graph):

  V[at] = true

  edges = graph.getEdgesOutFromNode(at)
  for edge in edges:
    if V[edge.to] == false:
      i = dfs(i, edge.to, V, ordering, graph)

  ordering[i] = at
  return i - 1
```