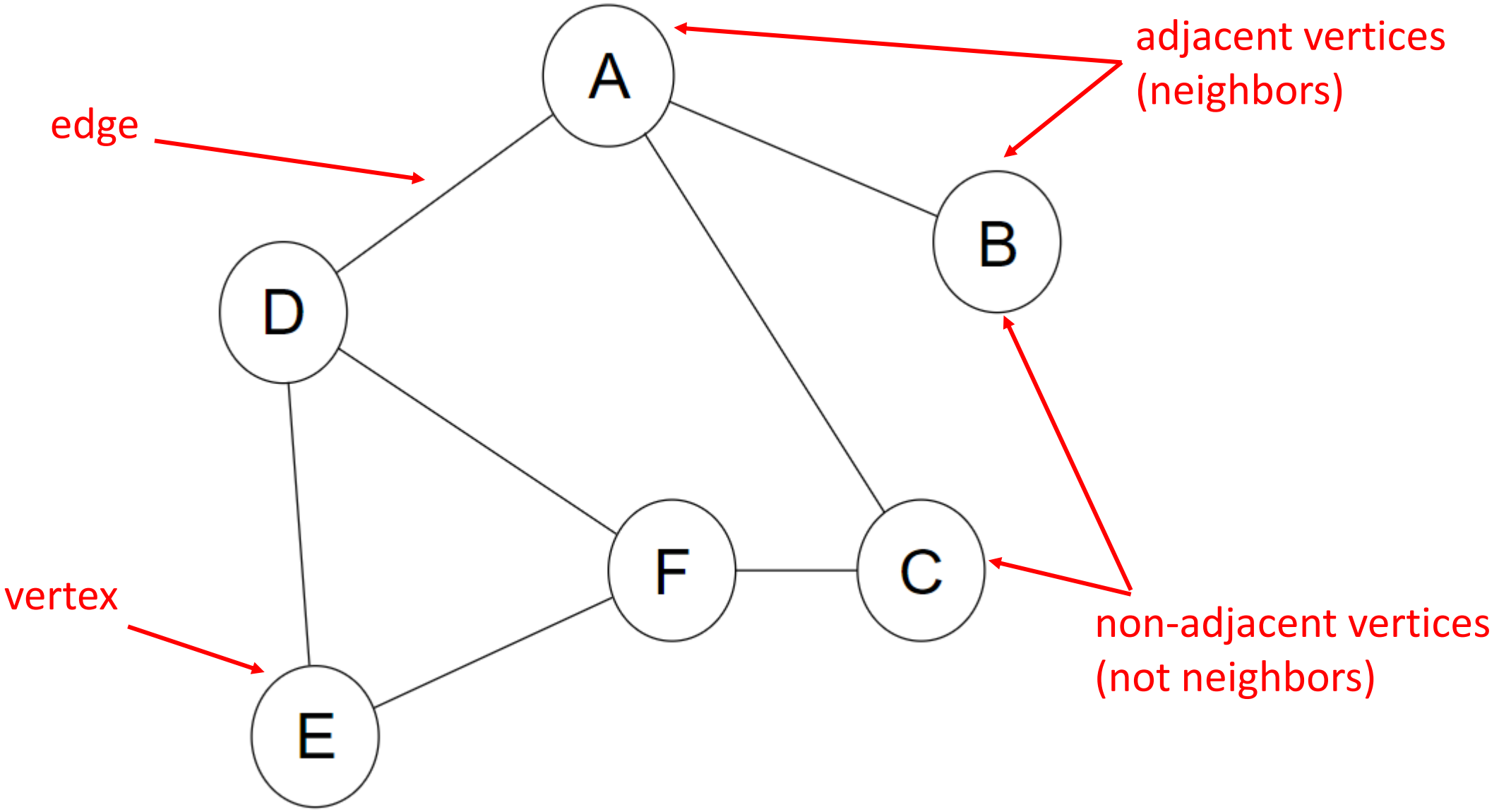
GRAPHS

Terminology

- graph: a collection of vertices and edges
- edge: a connection between two vertices
- adjacent: two vertices are *adjacent* if they are connected by an edge
  - these are also called *neighbors*

- sparse: few edges (e.g., O(n) edges)
  - most graphs are sparse!
- dense: many edges (e.g., $O(n^2)$ edges)

Example: Graph

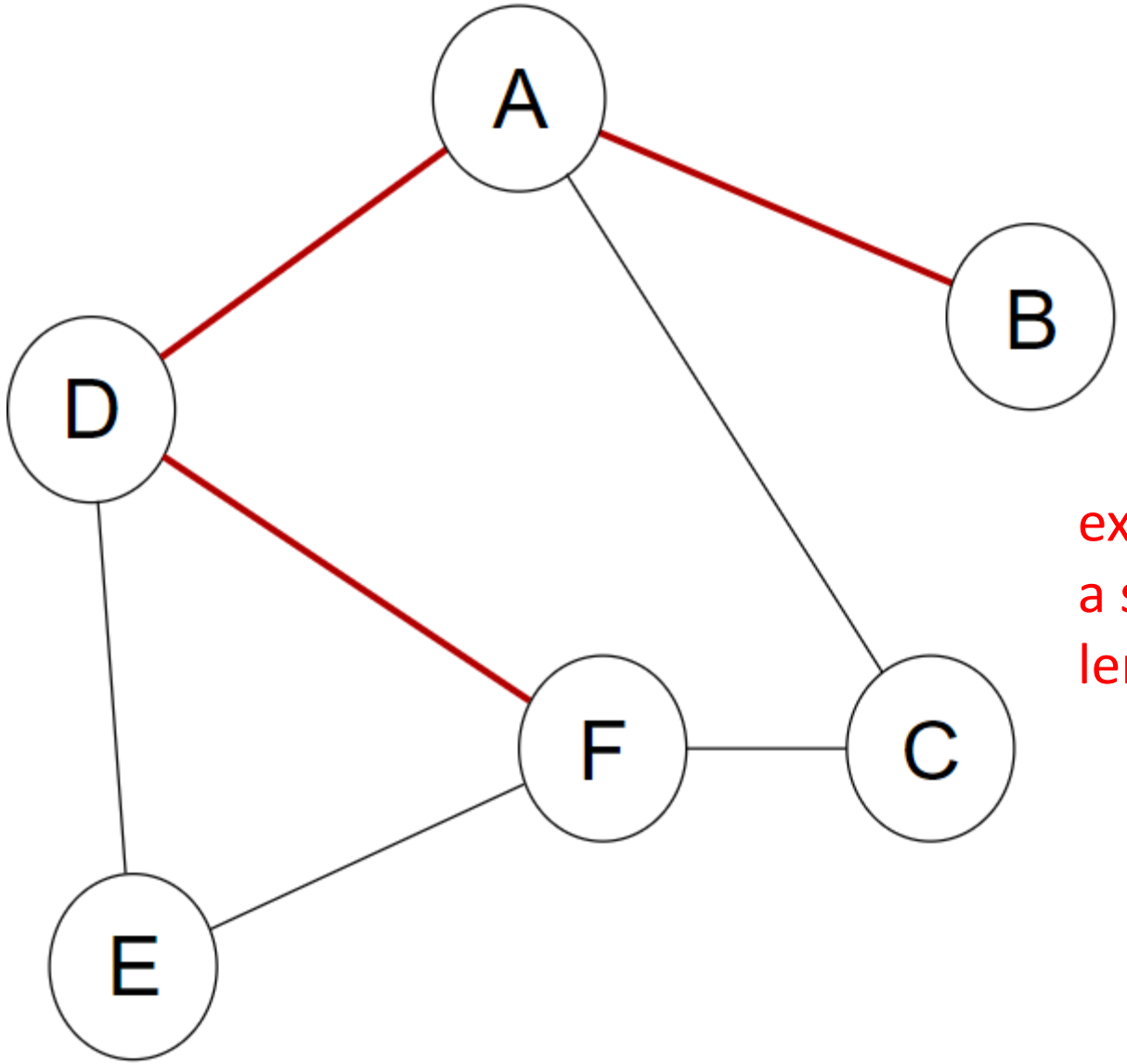


Terminology

- path: a sequence of edges that connect two vertices
- simple path: a path that repeats no vertex
- length of a path: the number of edges

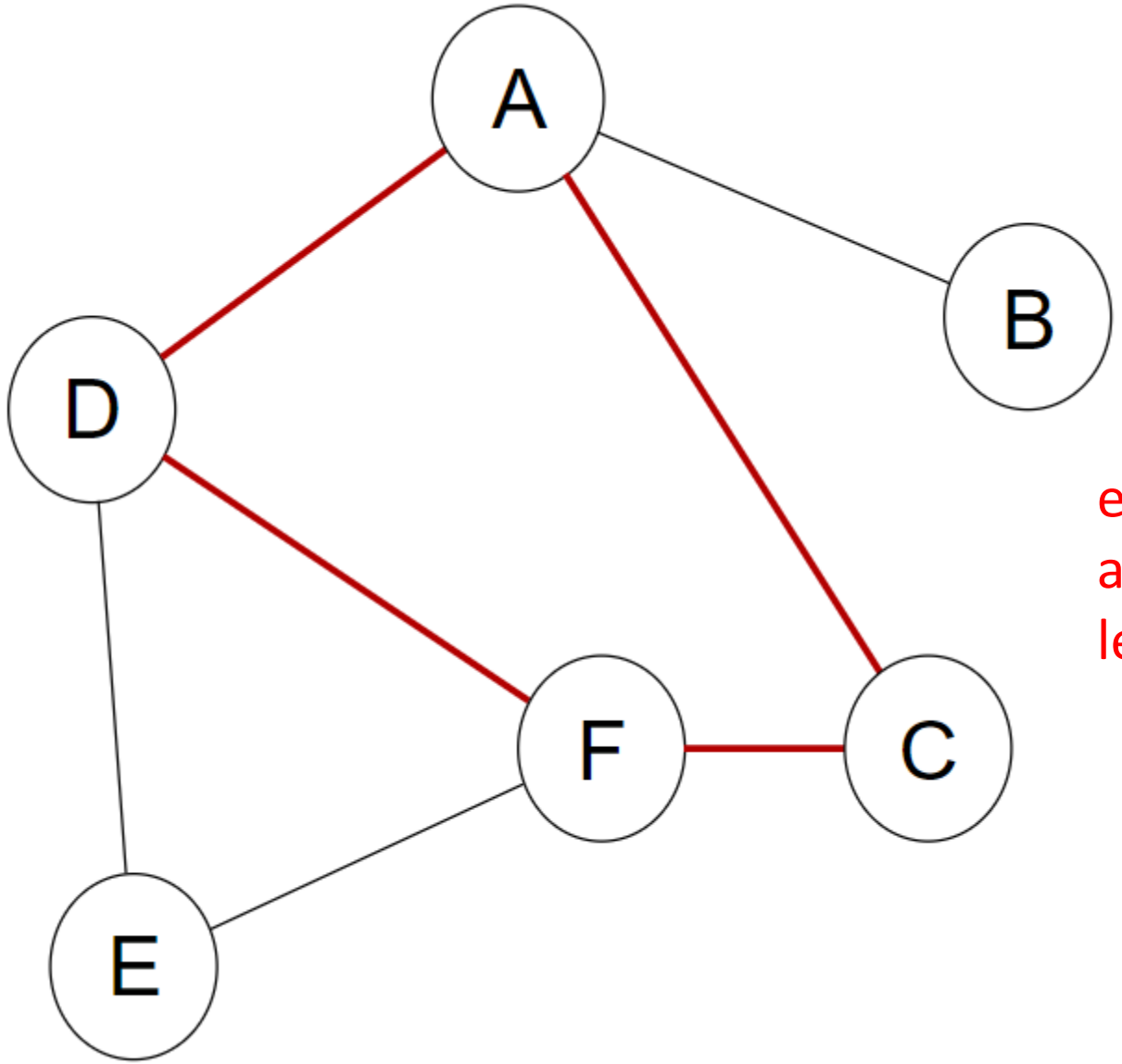
- cycle: a path with the same starting and ending vertex
- simple cycle: a cycle that passes through vertices only once, except for the starting/ending vertex
- acyclic: a graph with no cycles
  - also known as a tree!

Example: Path



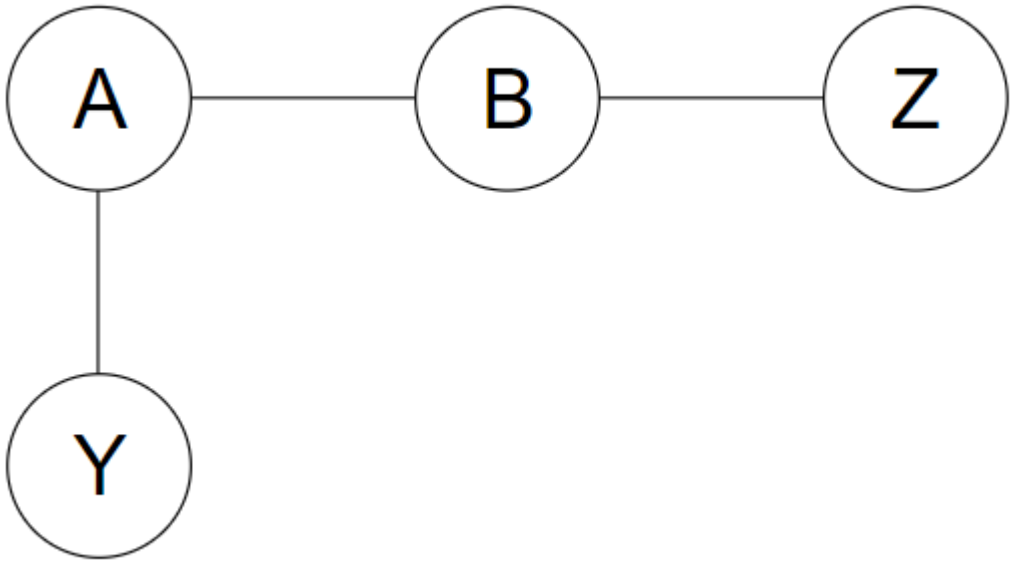
example:
a simple path from B to F
length: 3

Example: Cycle



example:
a cycle
length: 4

Example: Acyclic



acyclic- this graph has no cycles

Edges

- An edge connects two vertices.

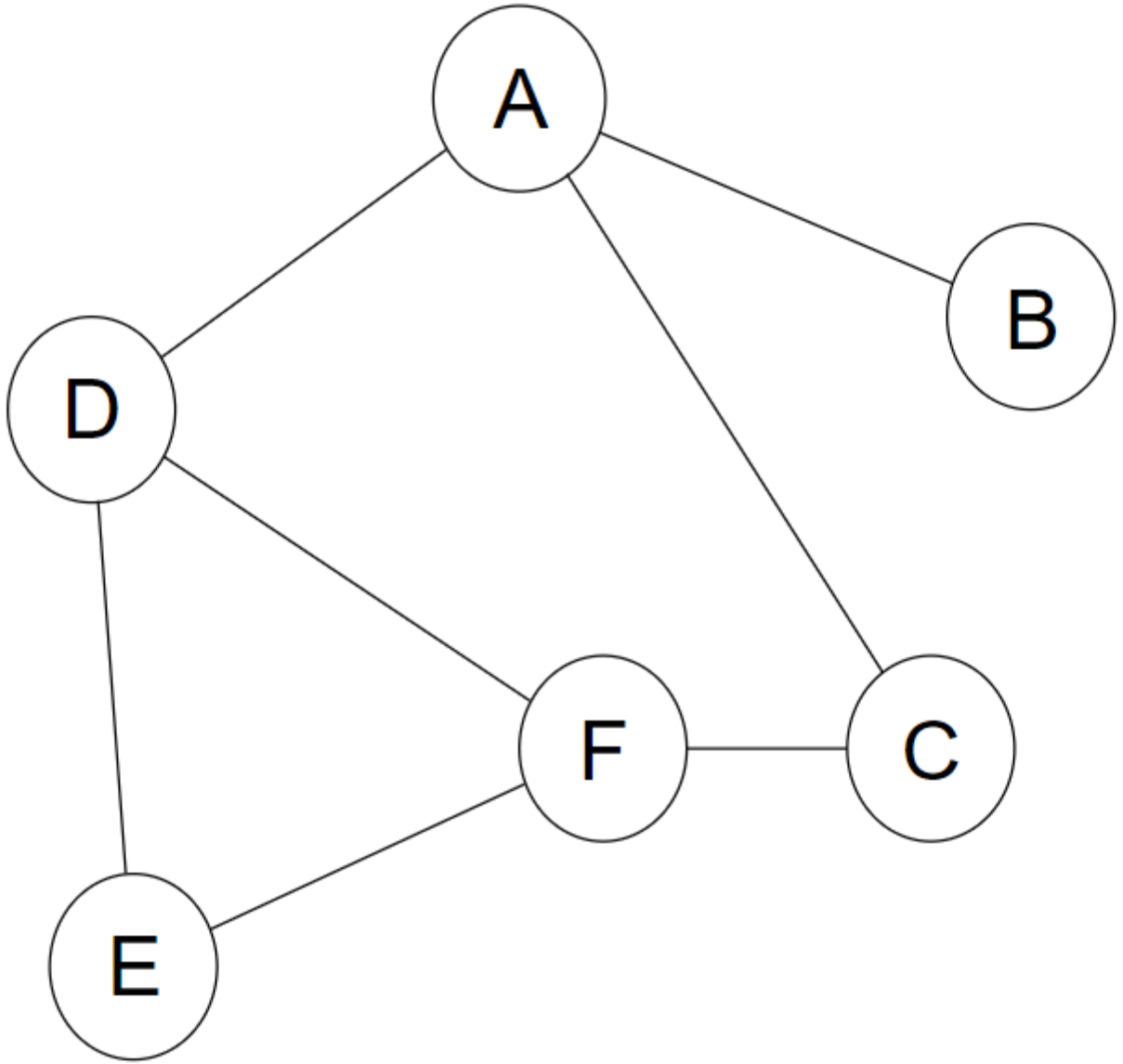
- An edge can be directed or undirected.
  - Directed edges have a direction from one vertex to another.
  - Undirected edges go in both directions.
- A graph with directed edges is called a *digraph*.

- Edges can have weights.
  - A weight is a value assigned to the edge.
  - In a weighted graph, you can compare paths by the sum of edge weights.

Example: Graph

undirected

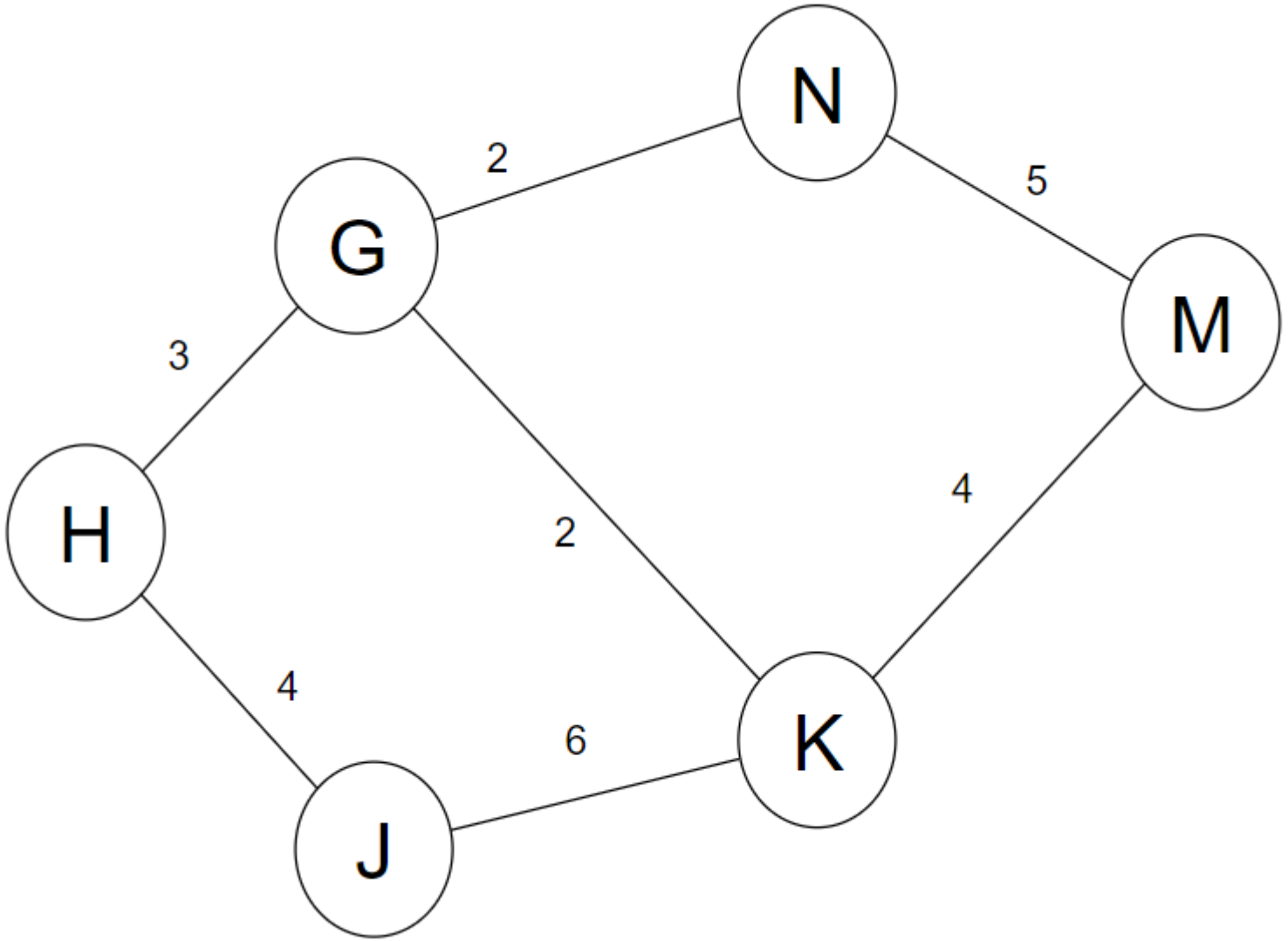
unweighted



Example: Graph

undirected

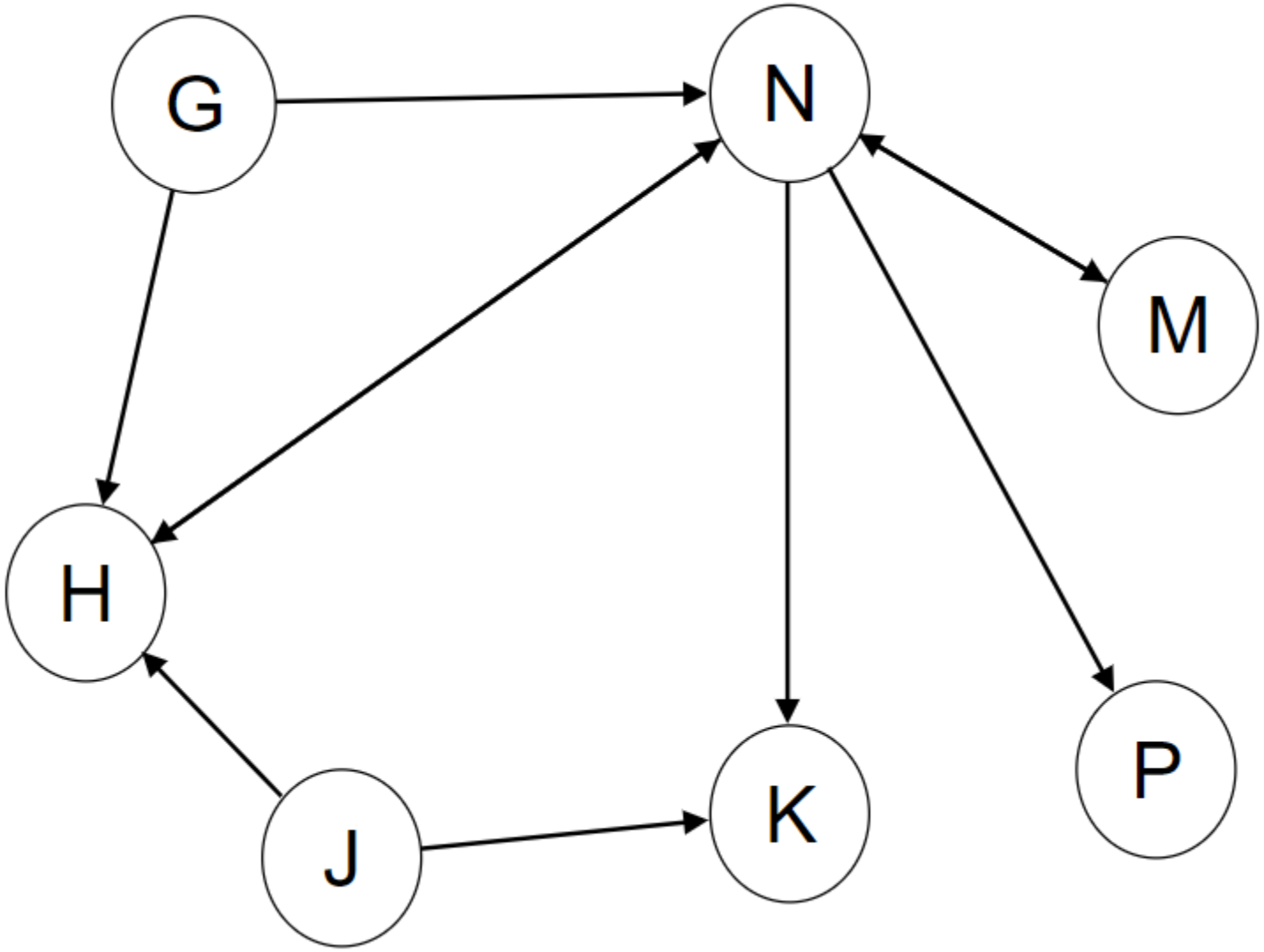
weighted



Example: Digraph

directed

unweighted



Digraph Neighbors

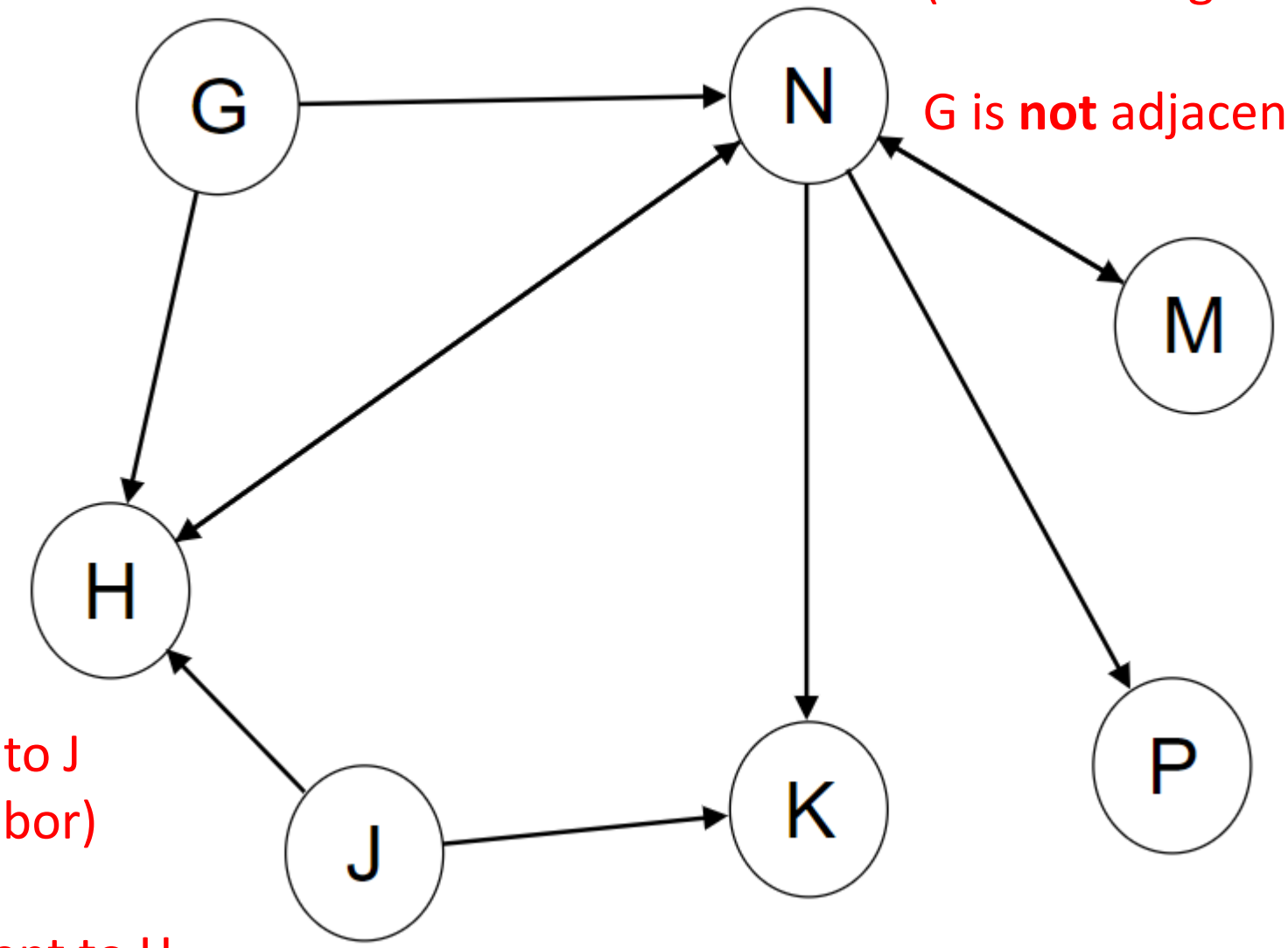
- A digraph is a graph with directed edges.
- Directed edges have a direction from one vertex to another.

- In a digraph, vertex i is adjacent to vertex j (i is j's neighbor) if there is an edge from j to i.

Example: Digraph Neighbors

N is adjacent to G
(N is G's neighbor)

G is **not** adjacent to N



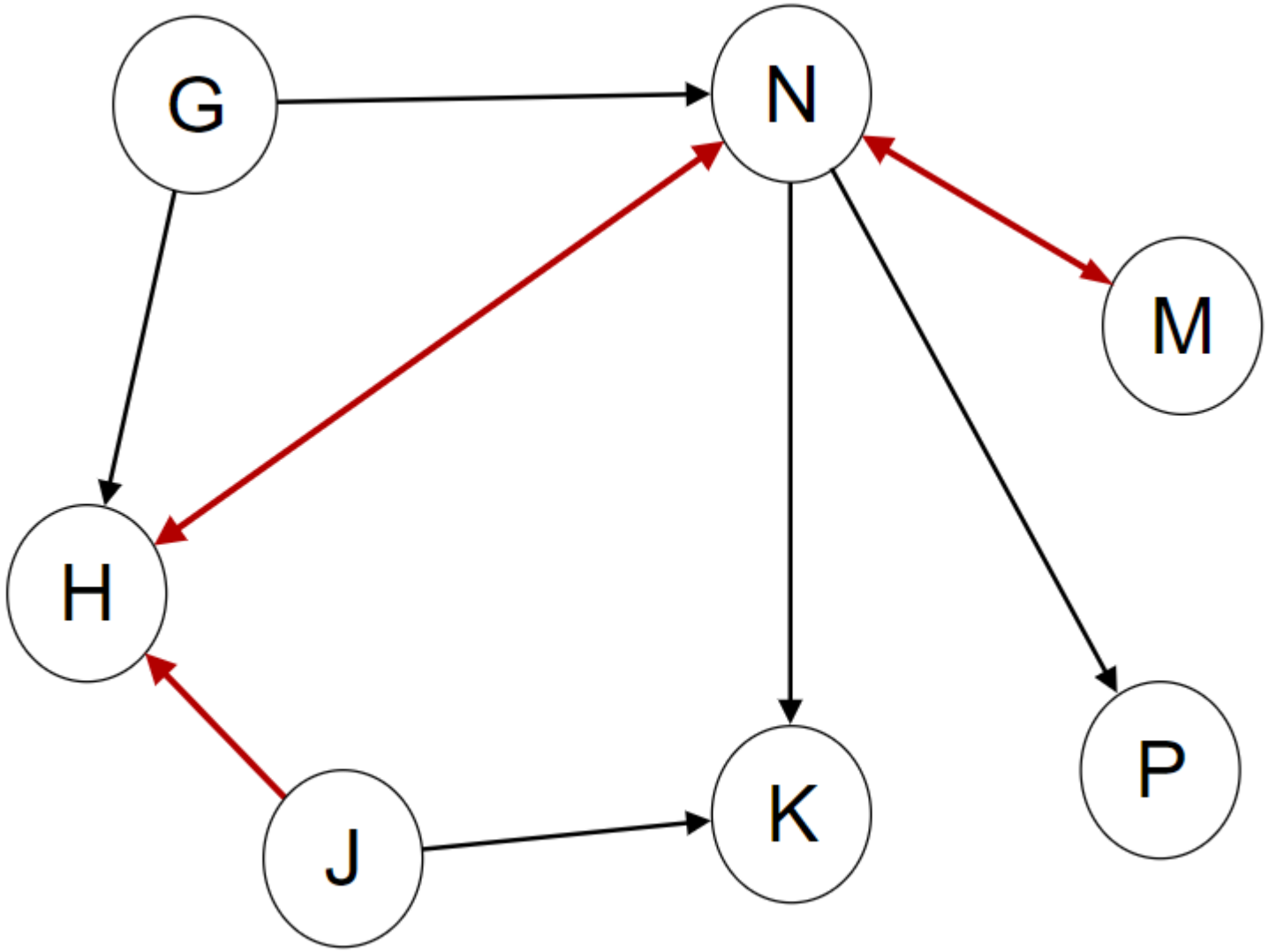
N is adjacent to M
M is adjacent to N

H is adjacent to J
(H is J's neighbor)

J is **not** adjacent to H

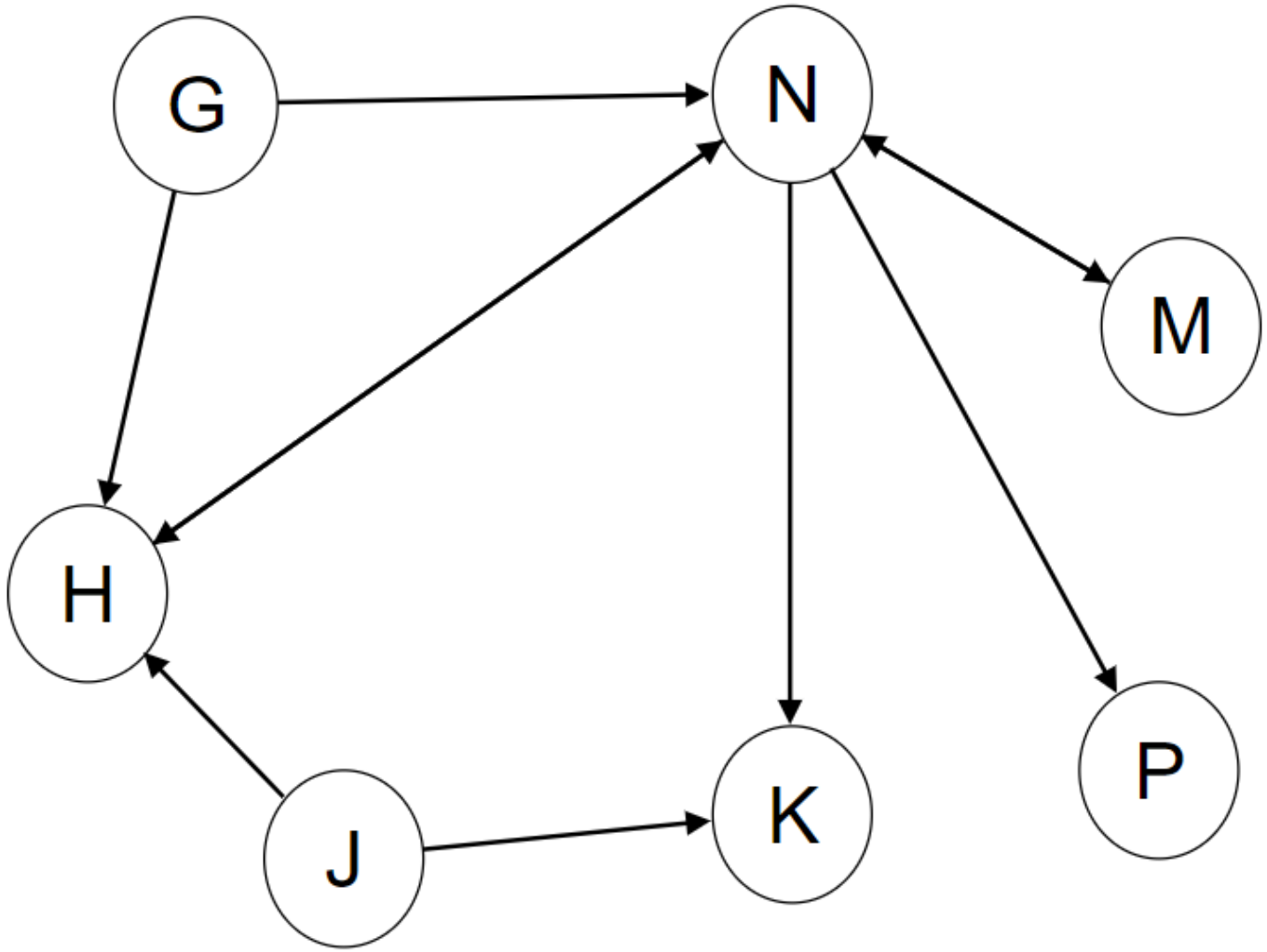
Example: Digraph Paths/Cycles

a simple path
from J to M



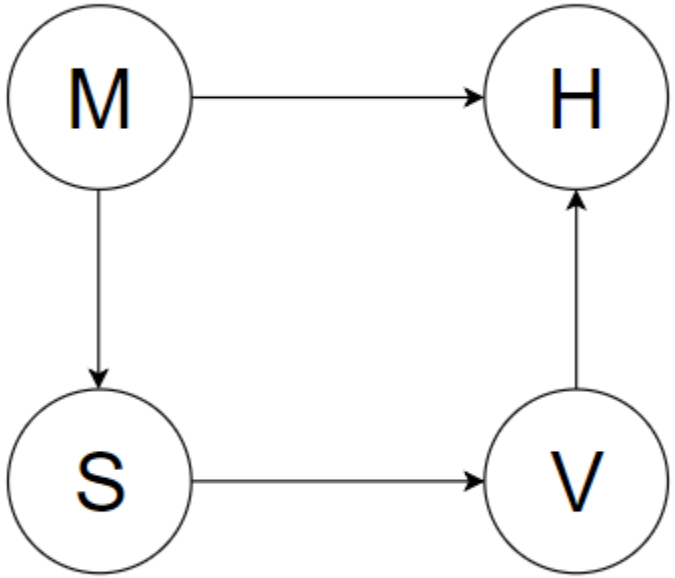
Example: Digraph Paths/Cycles

the graph is *cyclic*- it contains at least one cycle (the bidirectional edge H<->N and N <-> M are both cycles)



Example: Digraph Paths/Cycles

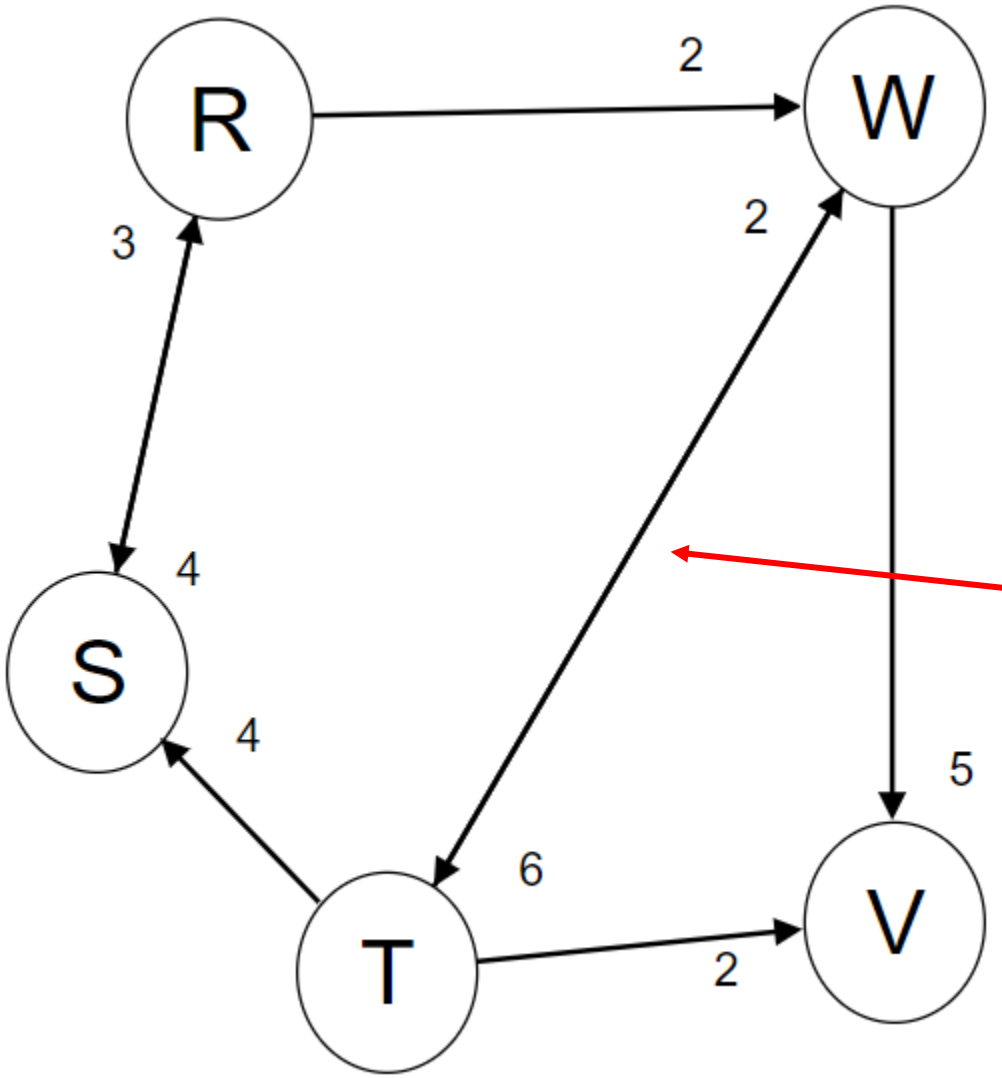
the graph is *acyclic*- it contains no cycles



Example: Digraph

directed

weighted

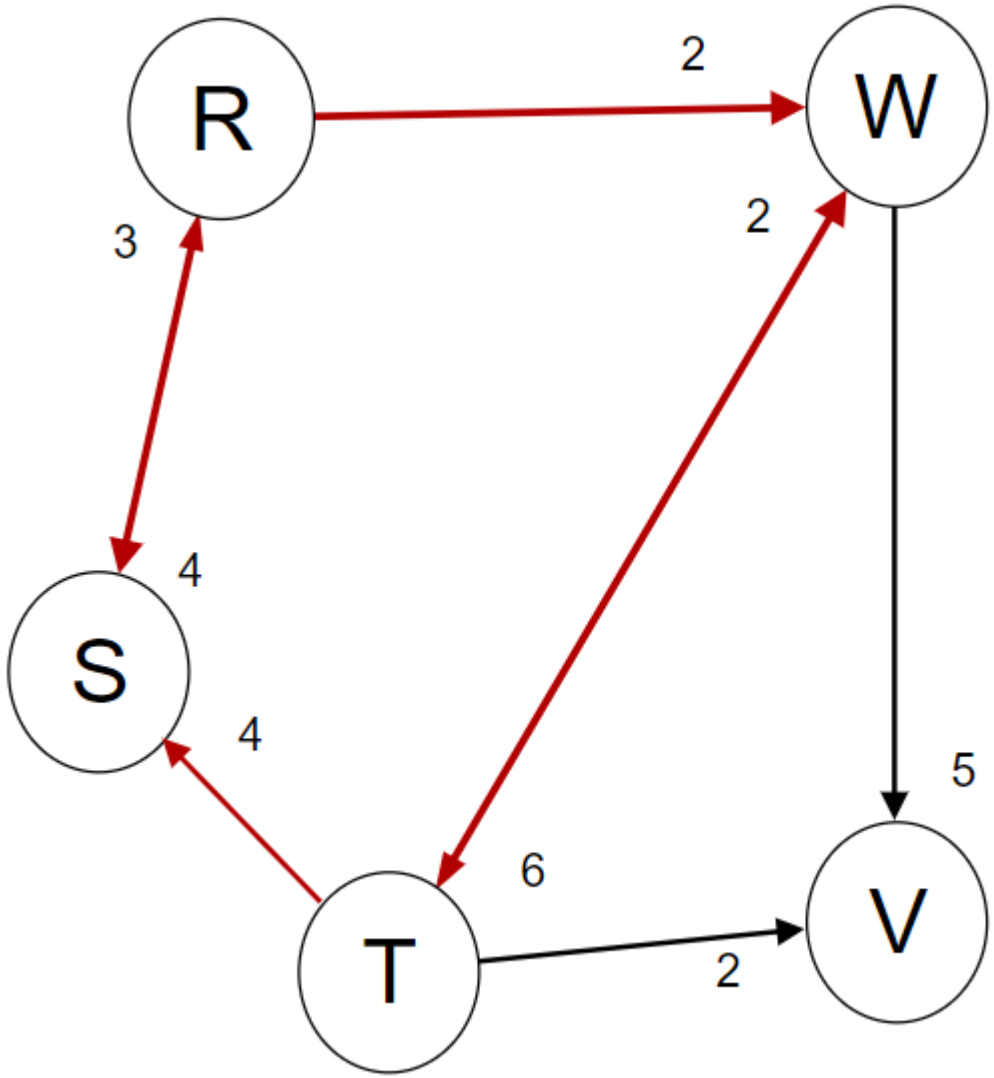


note that this edge has two weights!

Example: Digraph Paths/Cycles

the graph is cyclic- it has at least one cycle

cycles must follow the directionality of edges!

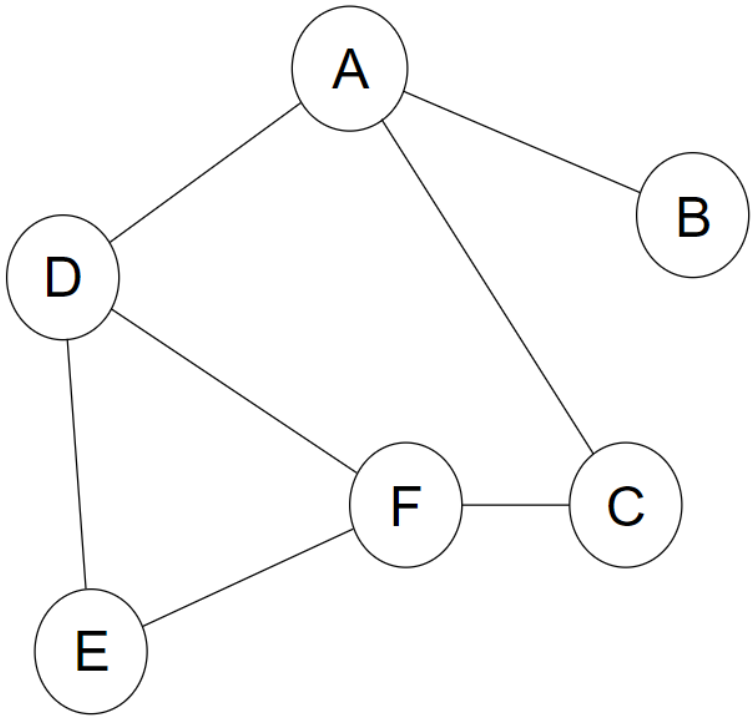


Connected and Complete

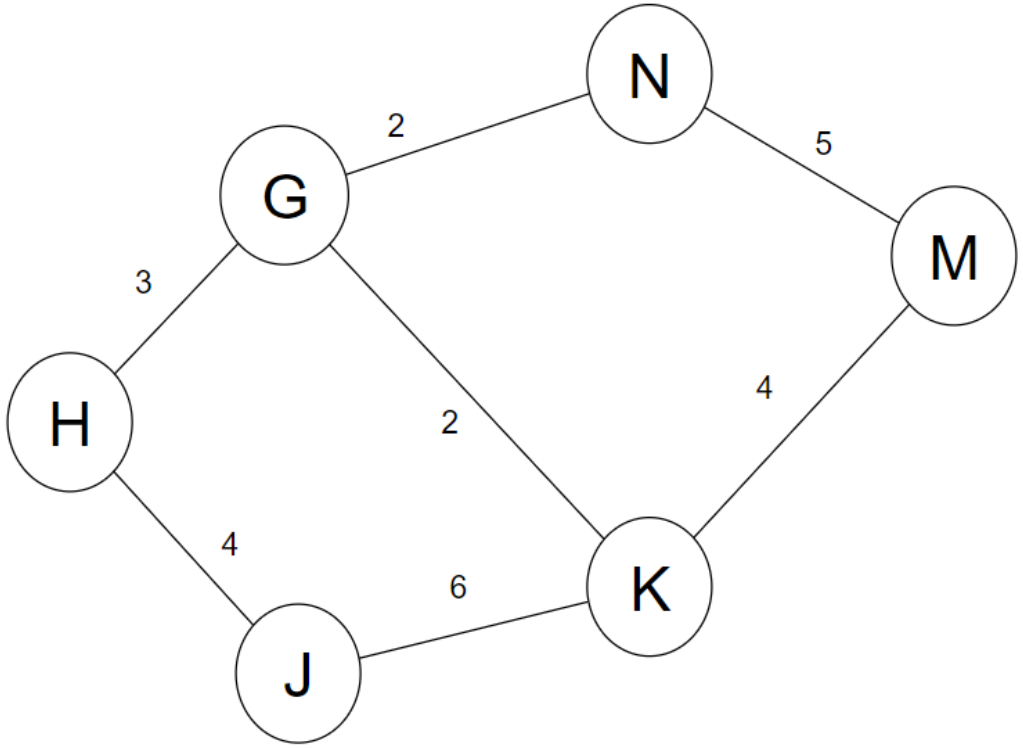
- A **connected** graph has a **path** to and from every vertex.
- A **complete** graph has an **edge** connecting every pair of vertices.
  - For a directed graph, there are v * (v-1) edges in a complete graph
  - For an undirected graph, there are (v *(v-1)) / 2 edges in a complete graph

Example: Connected

connected
not complete

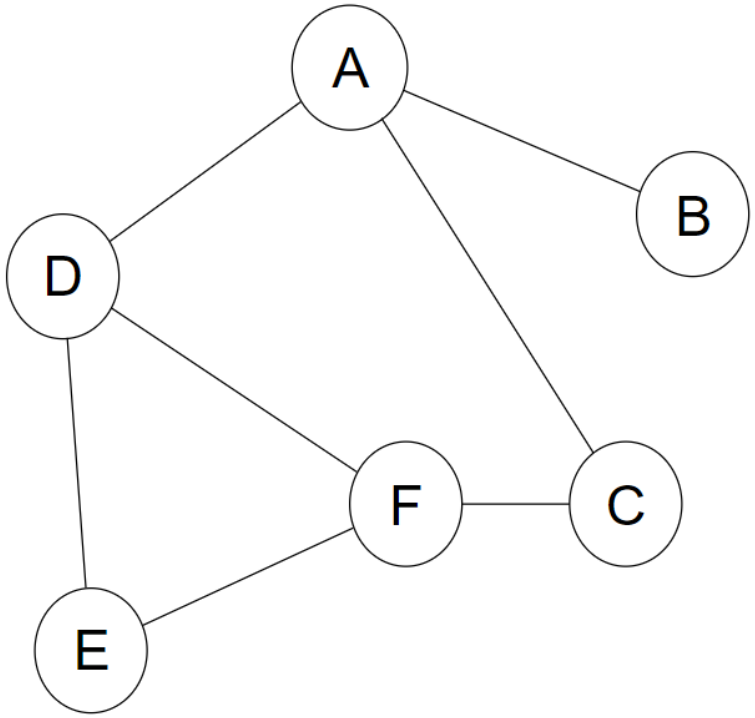


connected
not complete

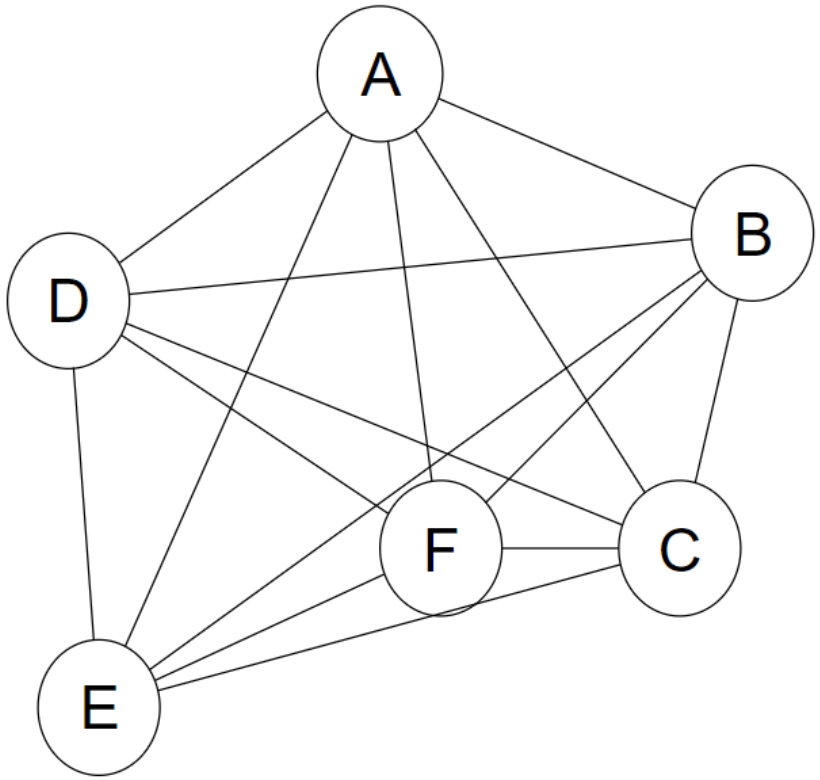


Example: Connected/Complete

connected
not complete

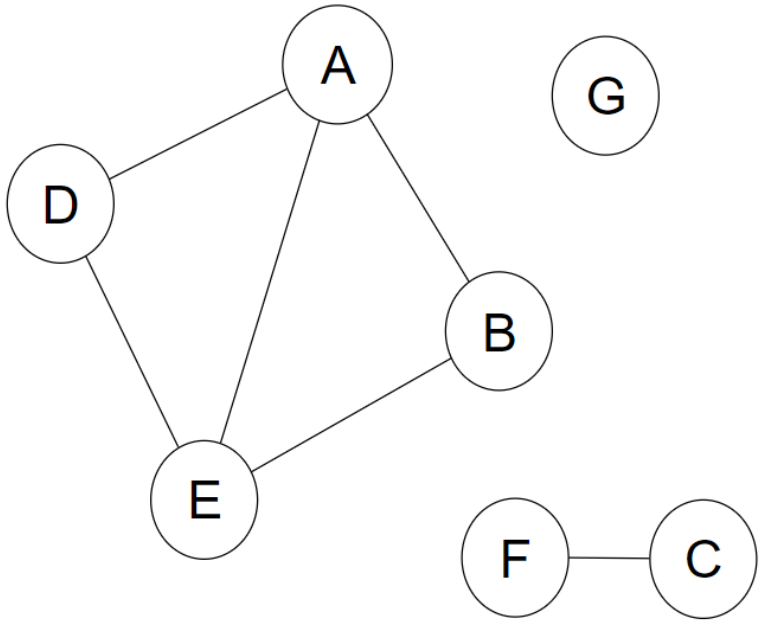


connected
complete



Example: Not Connected

not connected
not complete



Connectedness and Digraphs

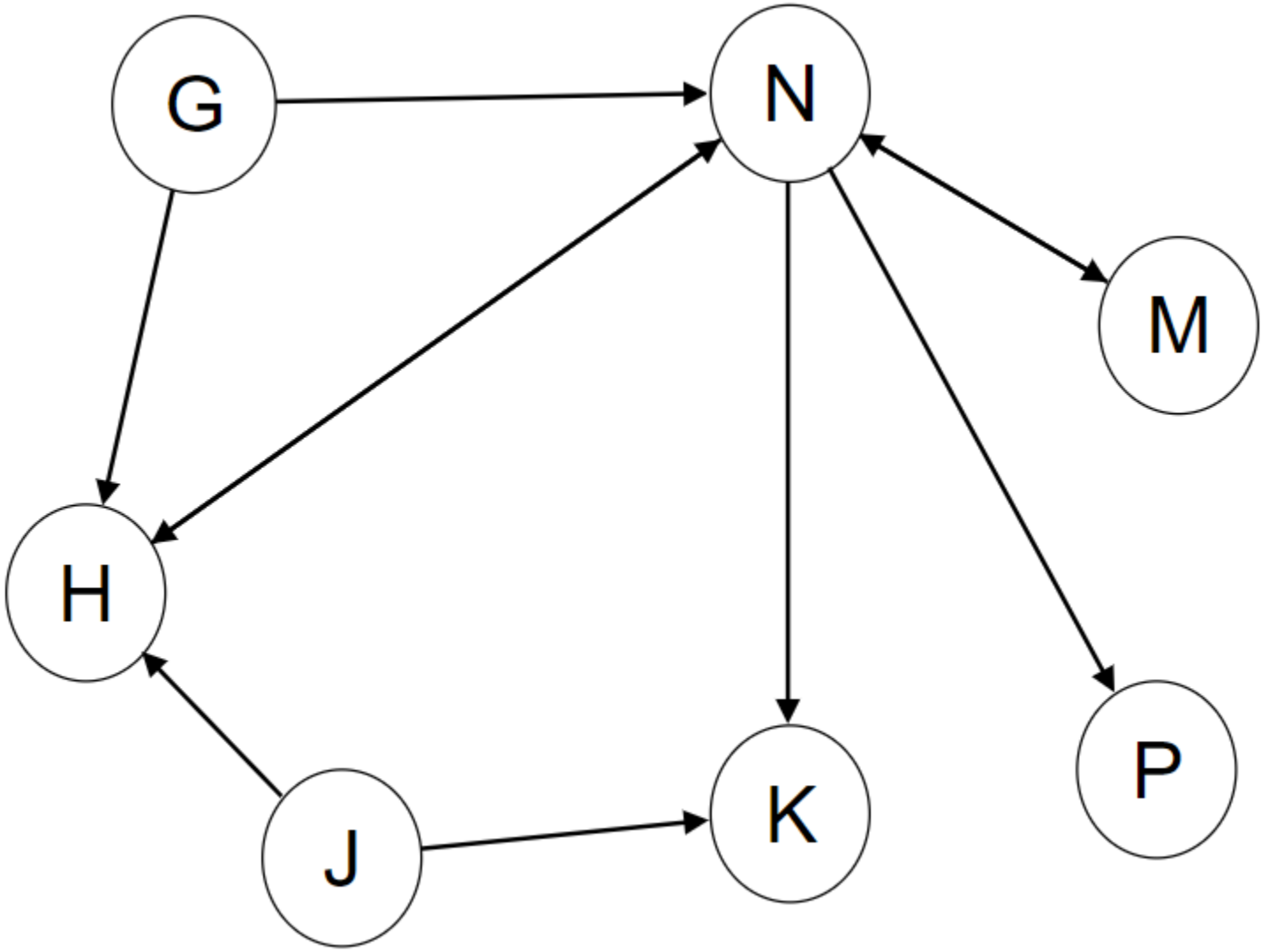
- A **connected** graph has a **path** to and from every vertex.
- Weakly connected: a path to/from each vertex but only if you ignore the directionality.
- Strongly connected: a path to/from each vertex following the directionality of the edges.

- Most often, when discussing connectedness in a digraph, we are referring to strongly connected. But it's always best to make sure!

Example: Weakly Connected

weakly connected

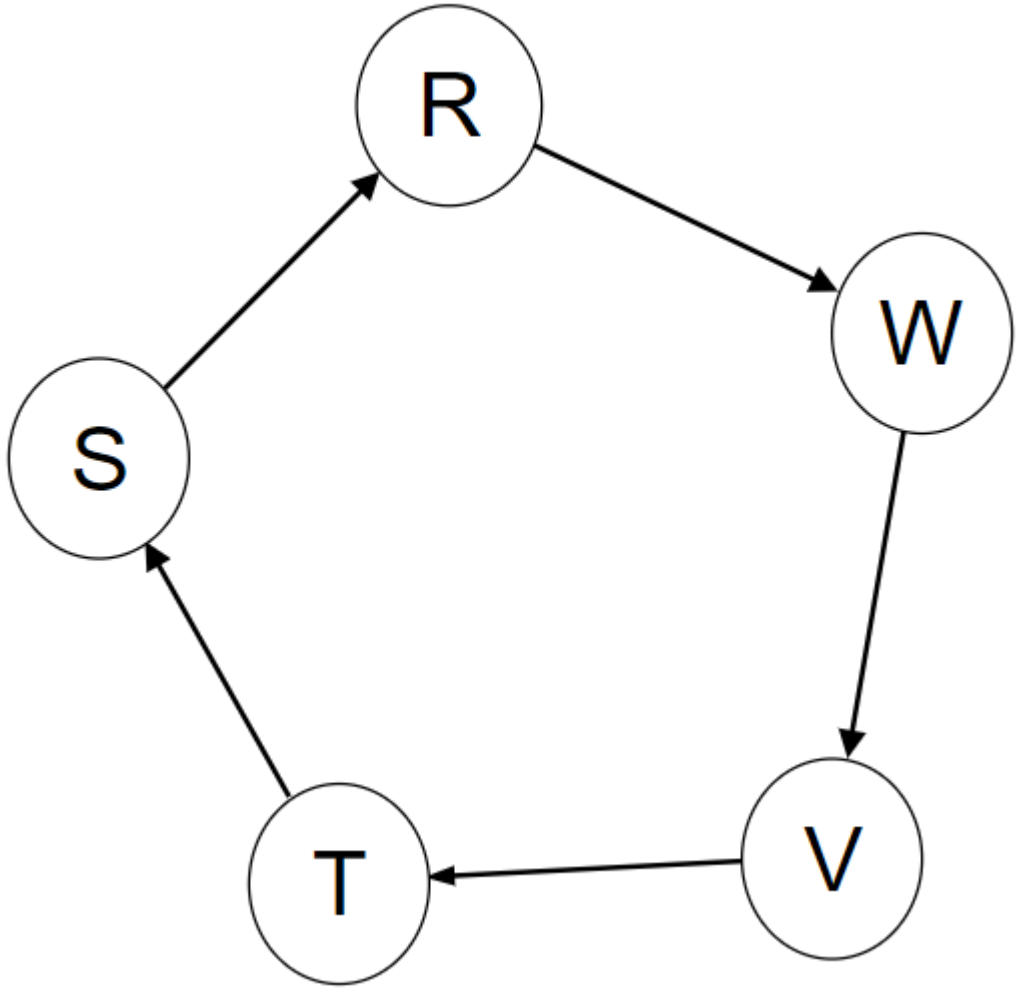
not strongly connected:
no path from G to J
(and others)



Example: Strongly Connected

strongly connected

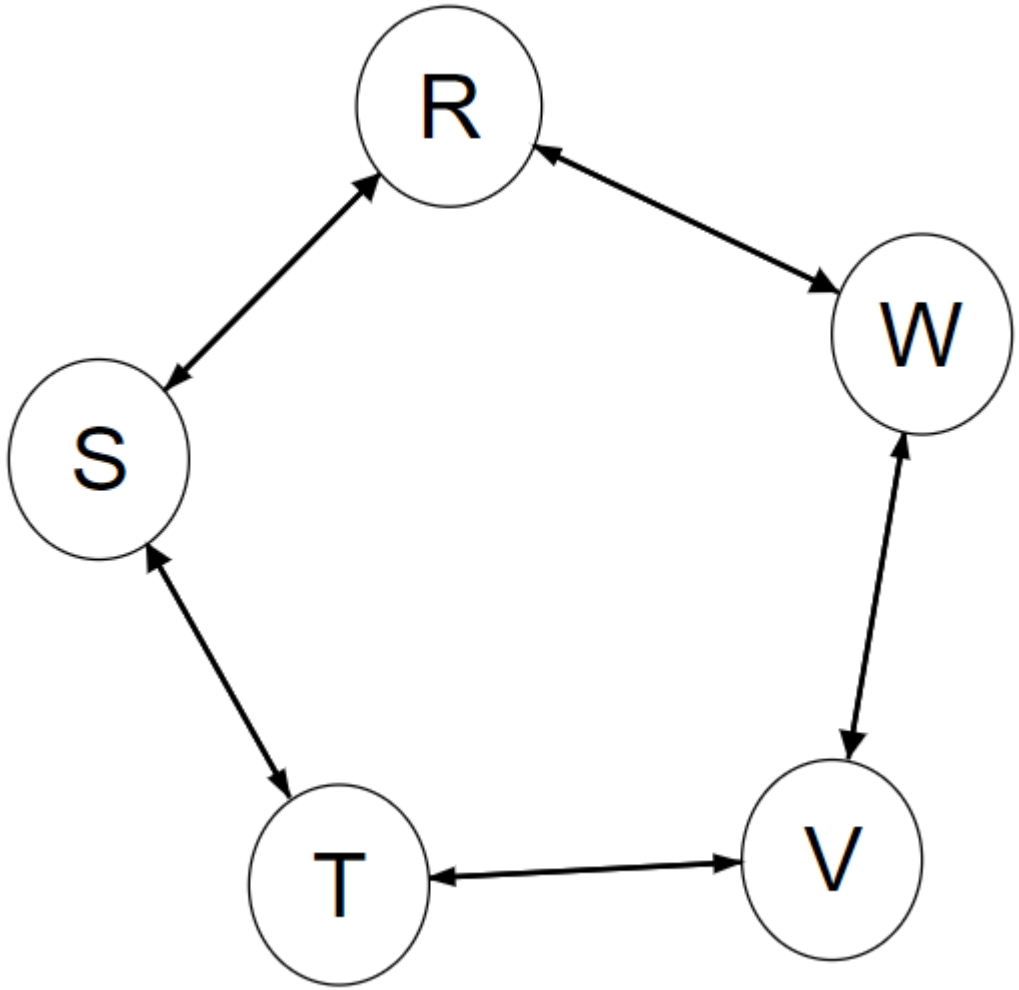
not complete



Example: Graph

connected?

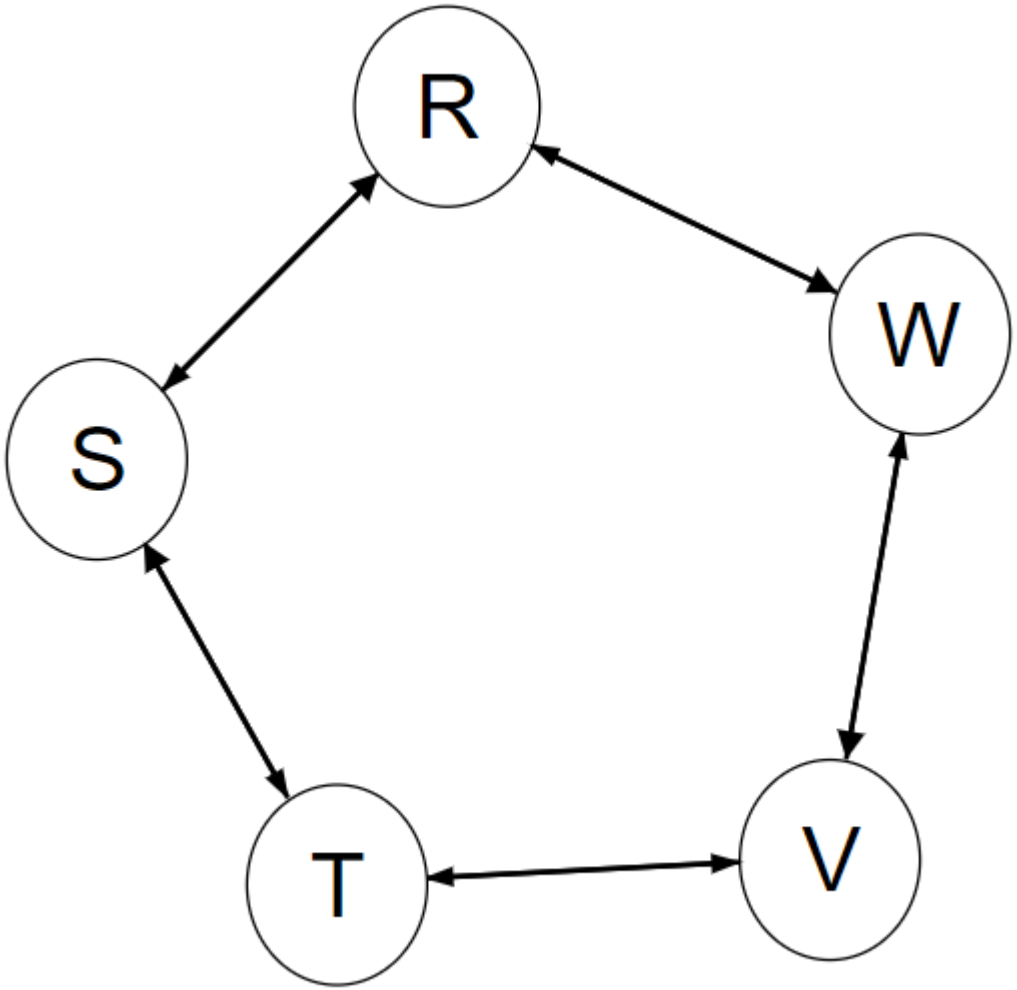
complete?



Example: Strongly Connected

strongly connected

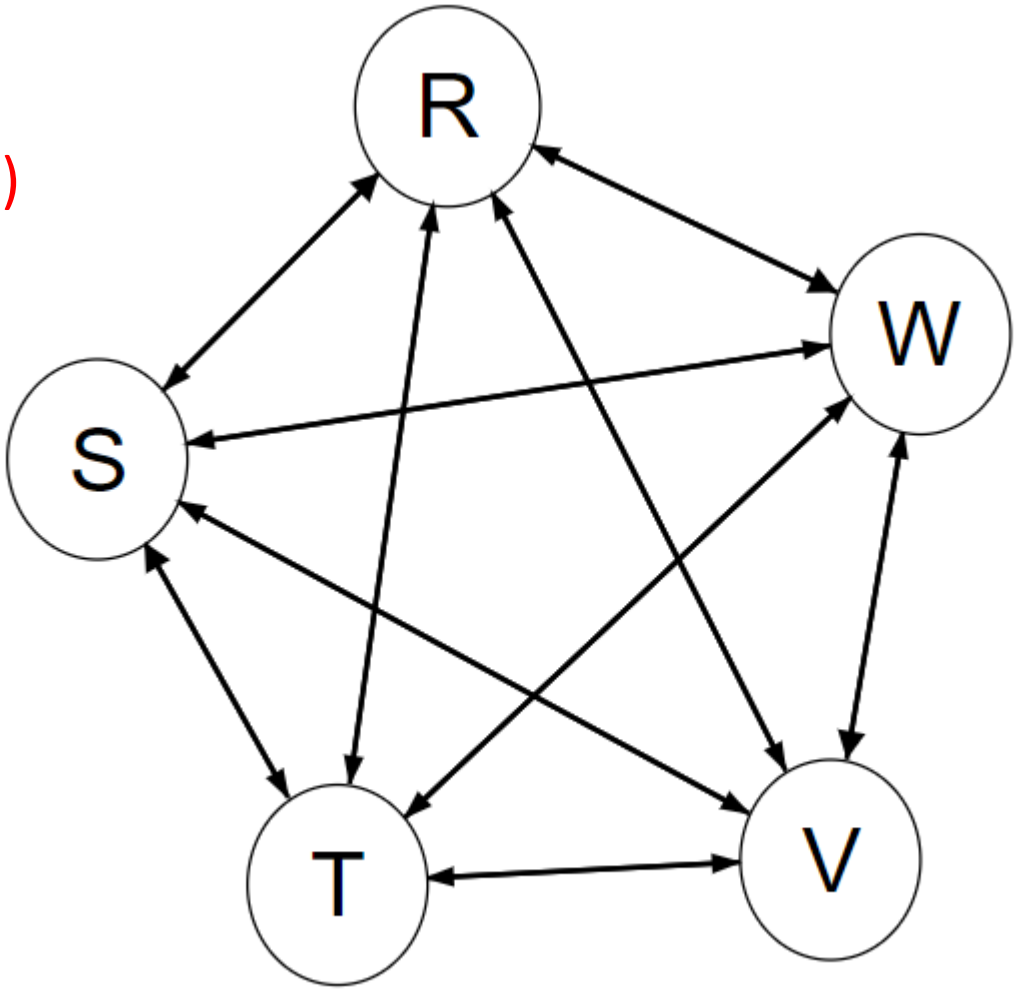
not complete



Example: Complete

complete

(thus also strongly connected)



GRAPH TRAVERSALS

Graph Traversal

- Traversal is a way to access the data in a graph.
- Two types of traversals: depth-first and breadth-first

Graph Traversal- The Origin

- Graph traversals are dependent on the starting vertex- called the *origin.*
- There is no "root vertex" like in a tree- *any* vertex can be the origin.

- The choice of origin affects the order of a traversal.
- If a graph is not connected, a traversal might not include all vertices.

Graph Traversal- Visitation Order

- The order in which you visit vertices is not part of the breadth-first or depth-first algorithm.
- It must be specified but is separate from the algorithm.

- We'll use alphabetic order.

BREADTH-FIRST TRAVERSAL

Breadth-First Traversal

- Visit all neighbors of a vertex, then visit all neighbors of neighbors, then all neighbors of neighbors, and so on.
- View an entire level before moving down to the next level.

- Remember: origin matters!
- Remember: order of visitation is not part of the algorithm.

Breadth-First Traversal

- Typically implemented with a queue
- General approach:
  - visit a vertex
  - put all of the vertex's neighbors in the back of the queue
  - repeat

Breadth-First Traversal Algorithm

enqueue the origin onto the vertexQueue and mark as visited

while the vertexQueue is not empty

    dequeue a vertex from the vertexQueue

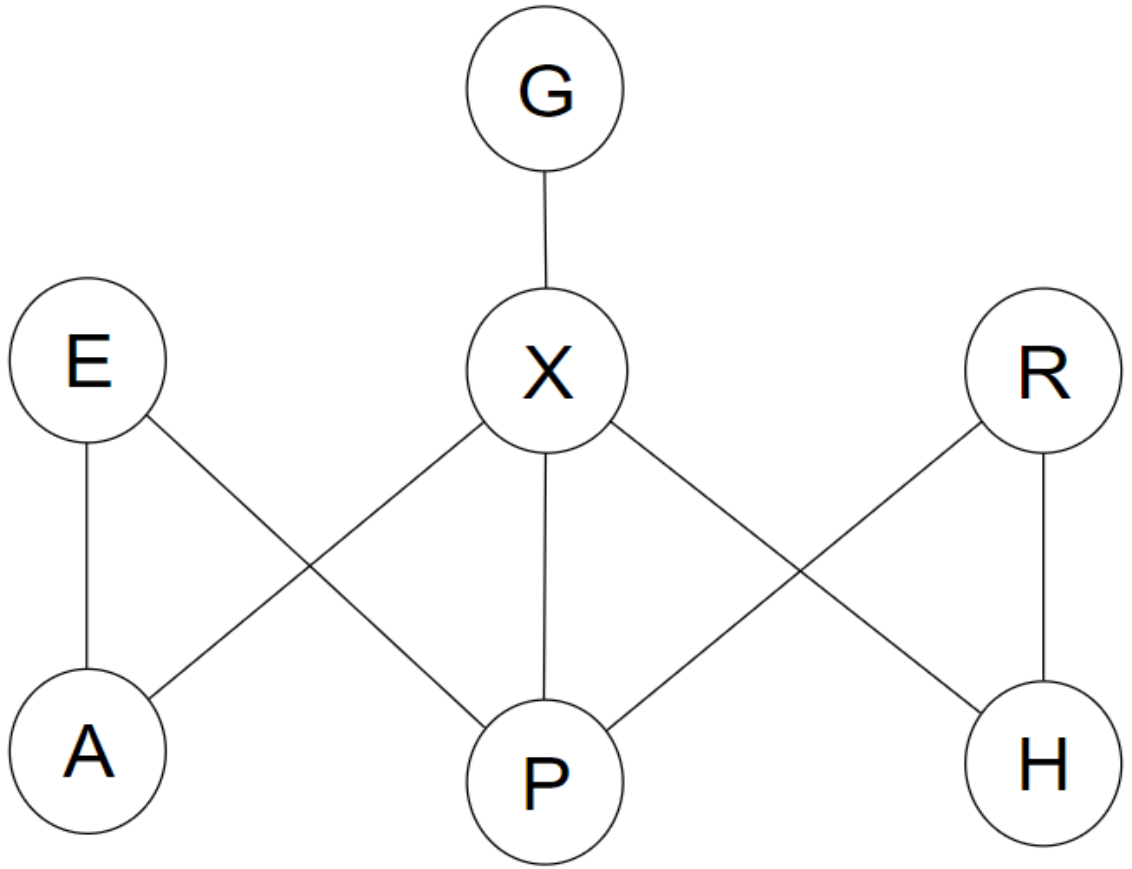
    enqueue the vertex into the traversalOrder queue

    enqueue its non-visited neighbors onto the vertexQueue

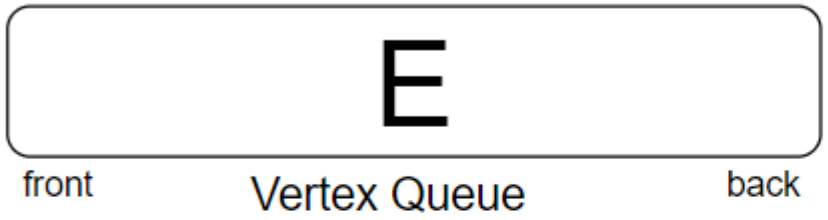
    mark each of those neighbors as visited

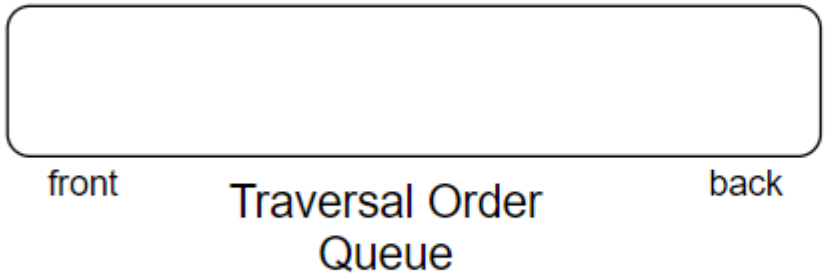
Breadth-First Example

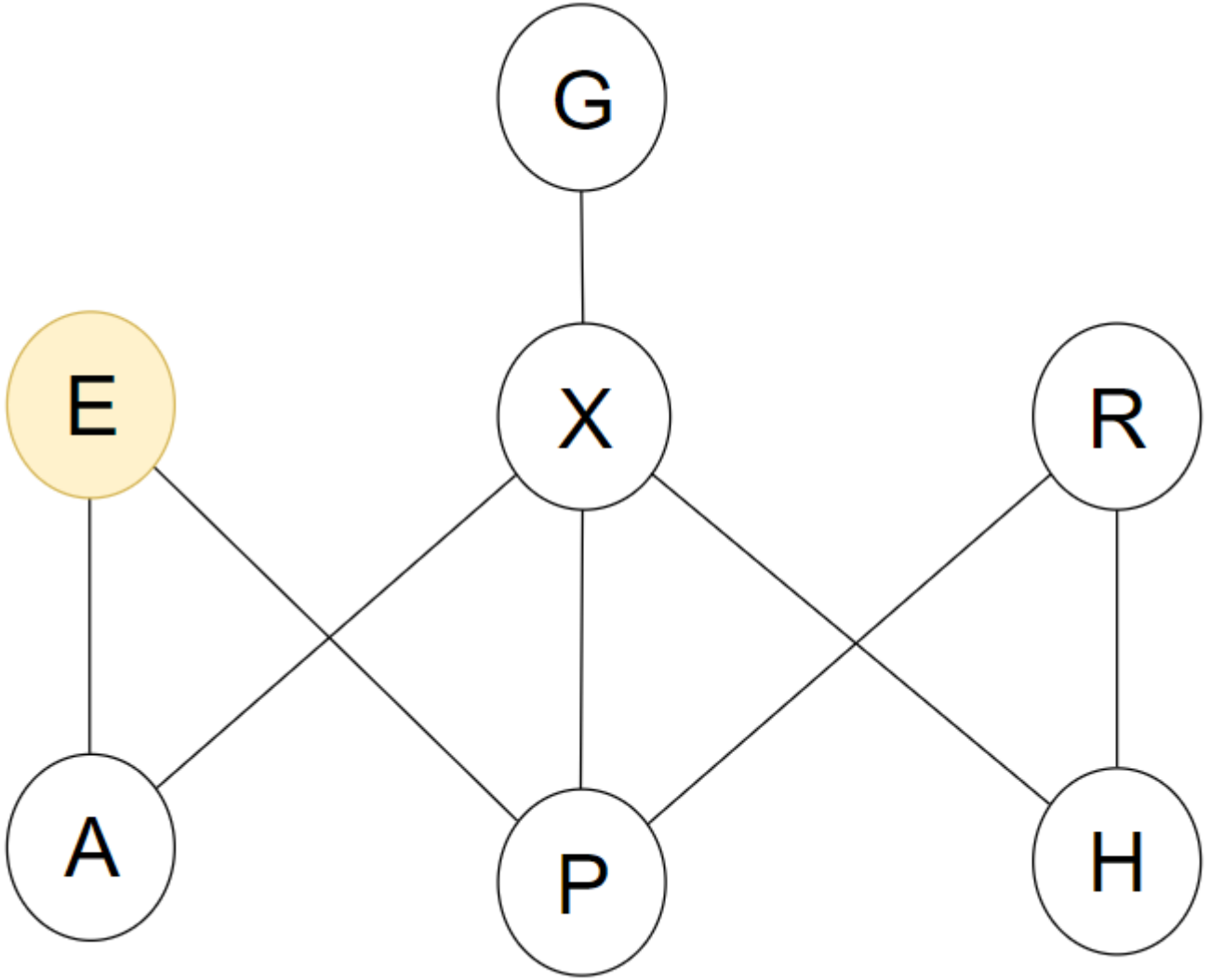
- Trace a breadth first traversal starting at vertex E.



enqueue the origin (E) onto the vertexQueue and mark as visited

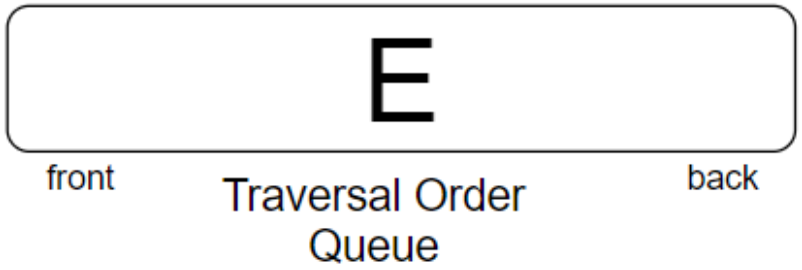


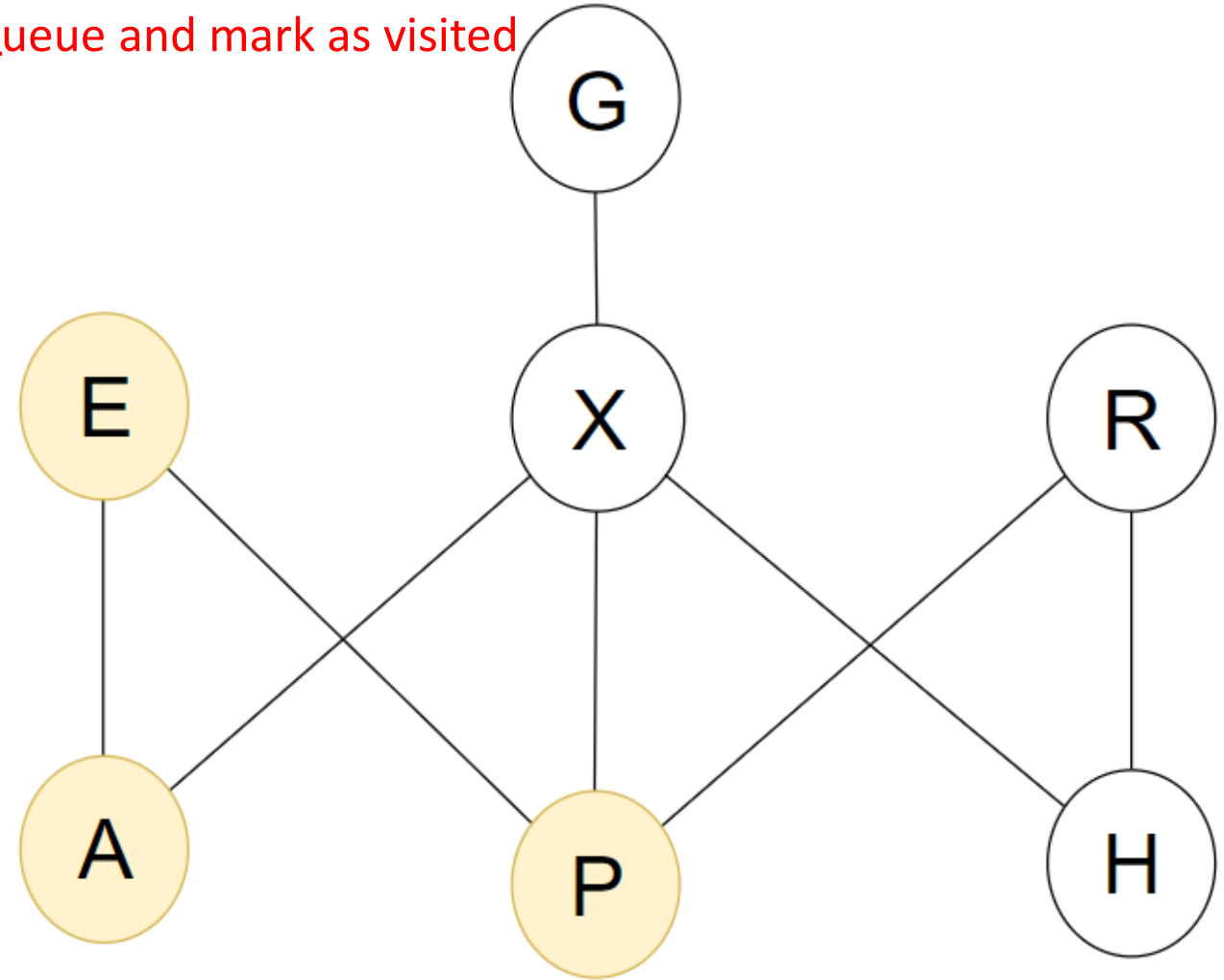




dequeue E and enqueue onto traversalOrder

enqueue E's neighbors (A and P) onto vertexQueue and mark as visited





dequeue A and enqueue onto traversalOrder

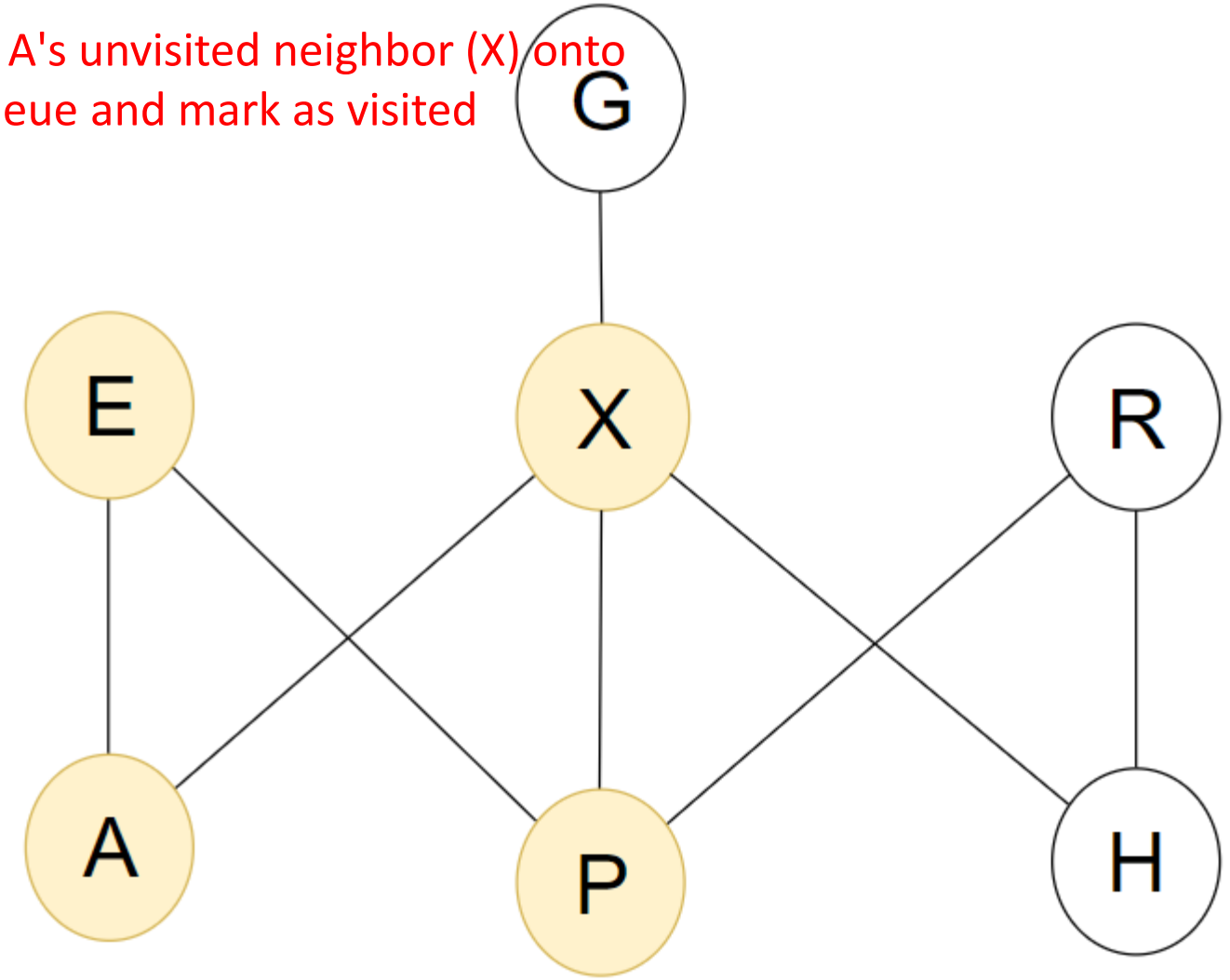
enqueue A's unvisited neighbor (X) onto vertexQueue and mark as visited

P X

front Vertex Queue back

E A

front Traversal Order Queue back



dequeue P and enqueue onto traversalOrder

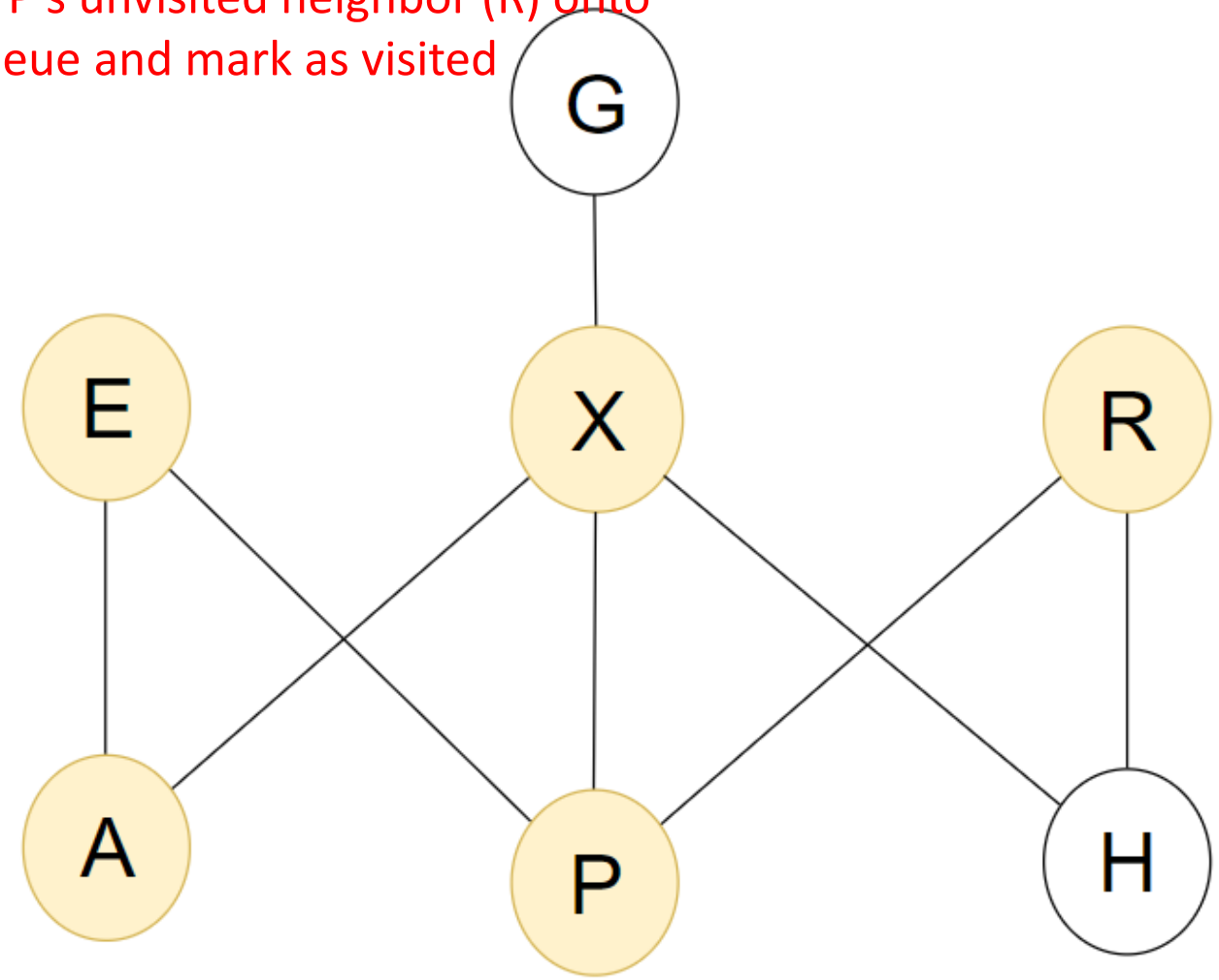
enqueue P's unvisited neighbor (R) onto vertexQueue and mark as visited

X R

front Vertex Queue back

E A P

front Traversal Order Queue back



dequeue X and enqueue onto traversalOrder

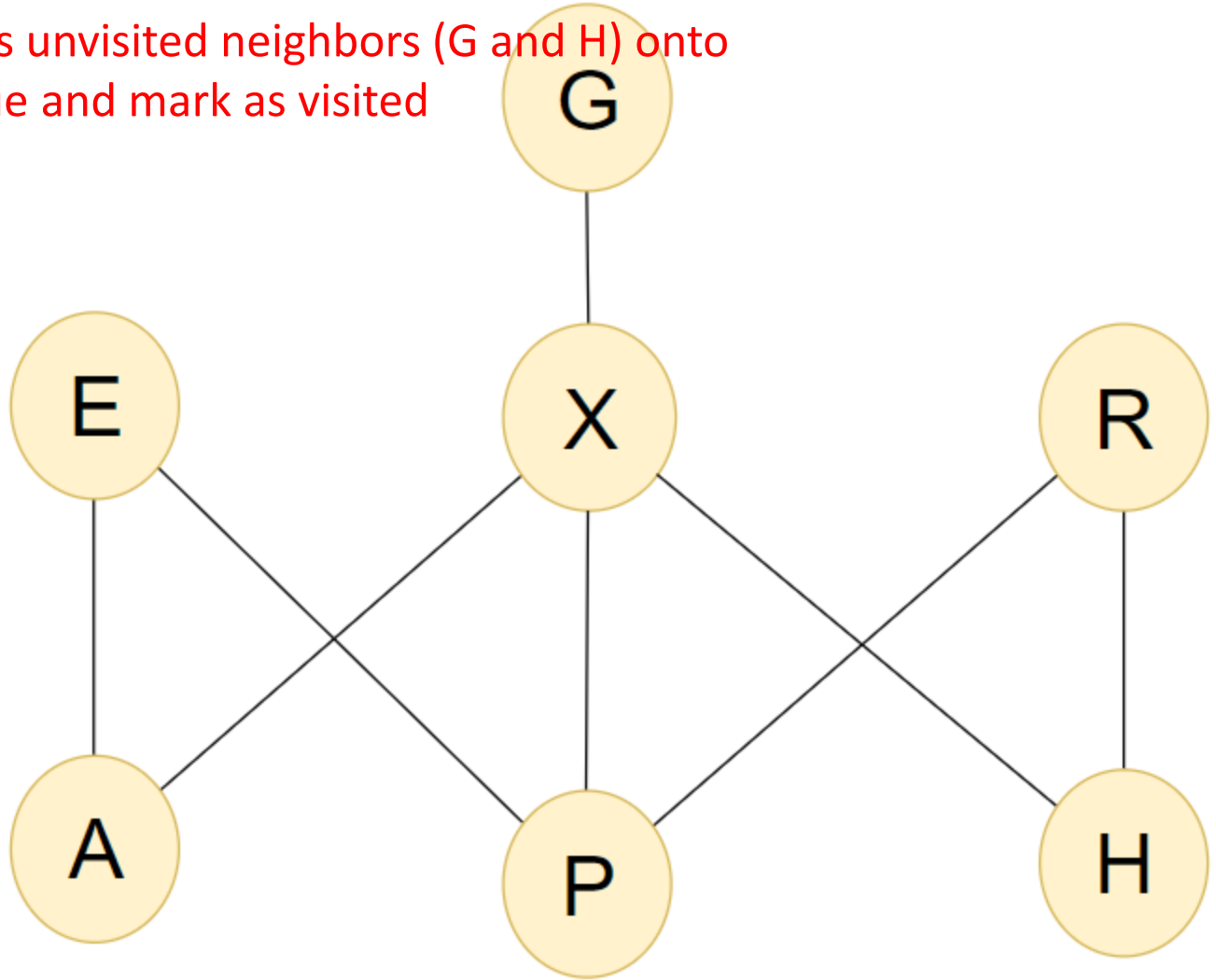
enqueue X's unvisited neighbors (G and H) onto vertexQueue and mark as visited

R G H

front Vertex Queue back

E A P X

front Traversal Order Queue back



dequeue R and enqueue onto traversalOrder

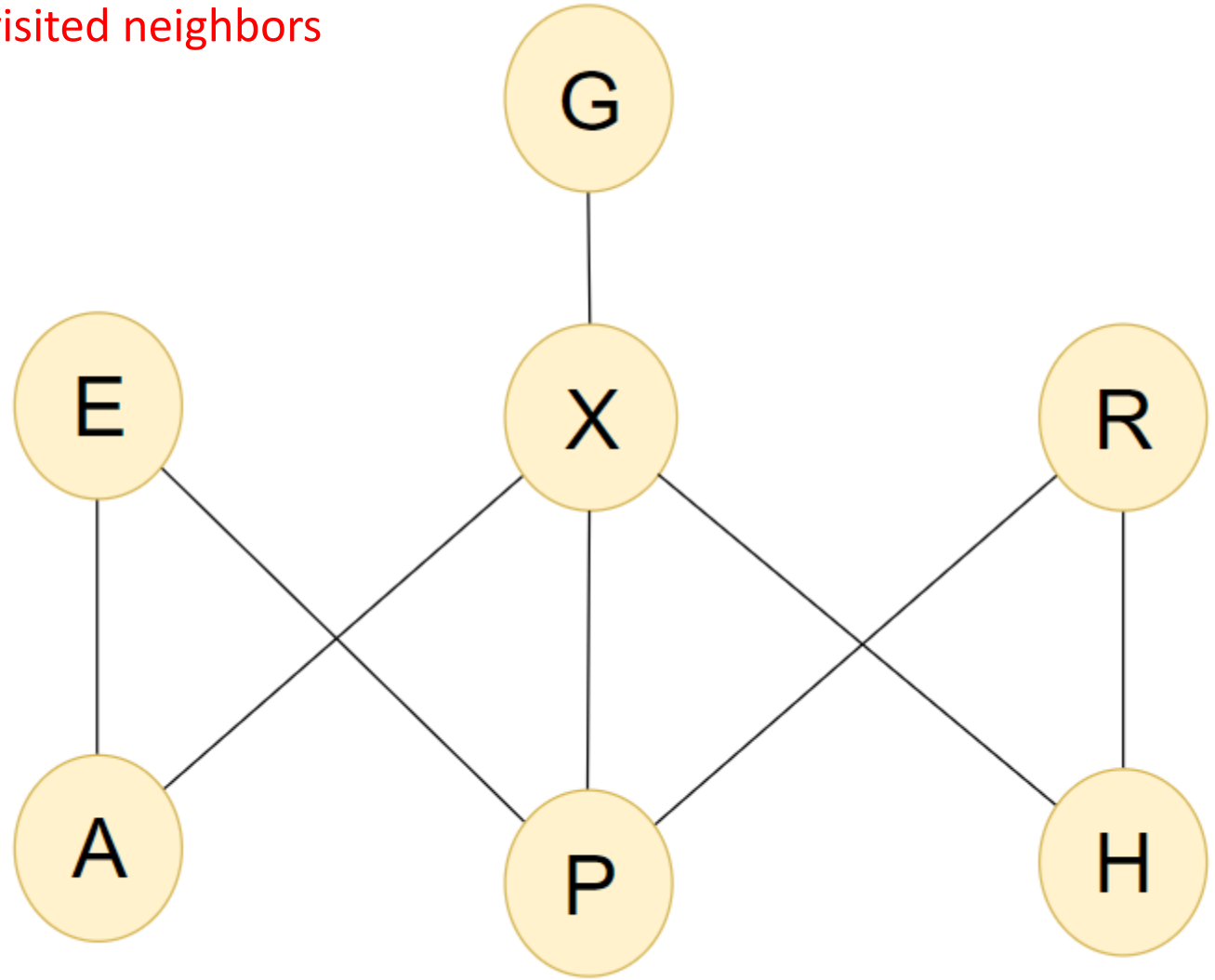
R has no unvisited neighbors

G H

front Vertex Queue back

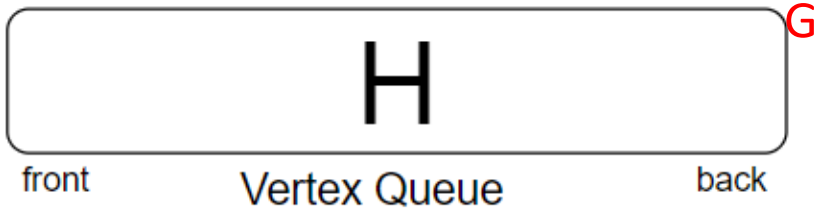
E A P X R

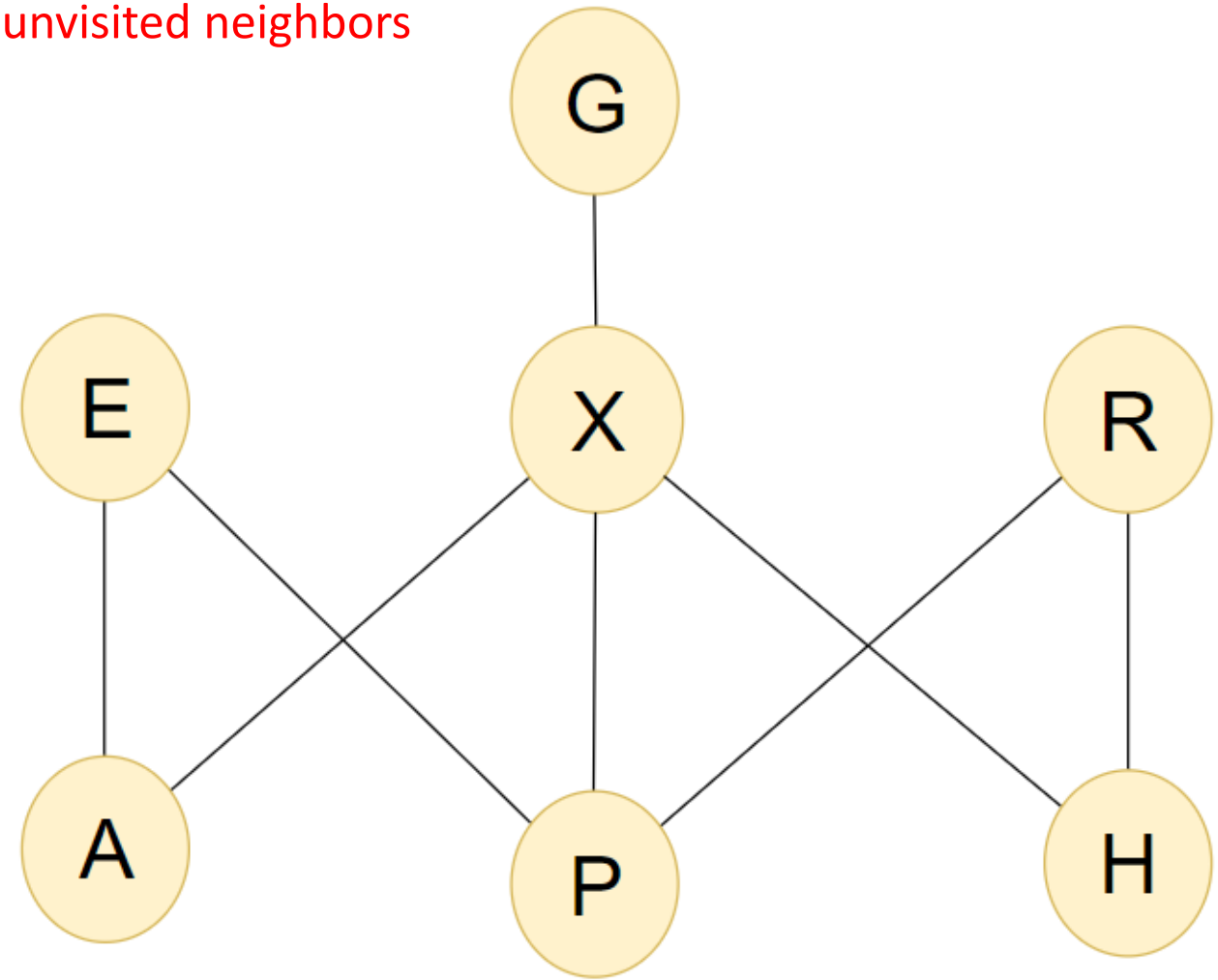
front Traversal Order Queue back



dequeue G and enqueue onto traversalOrder

G has no unvisited neighbors





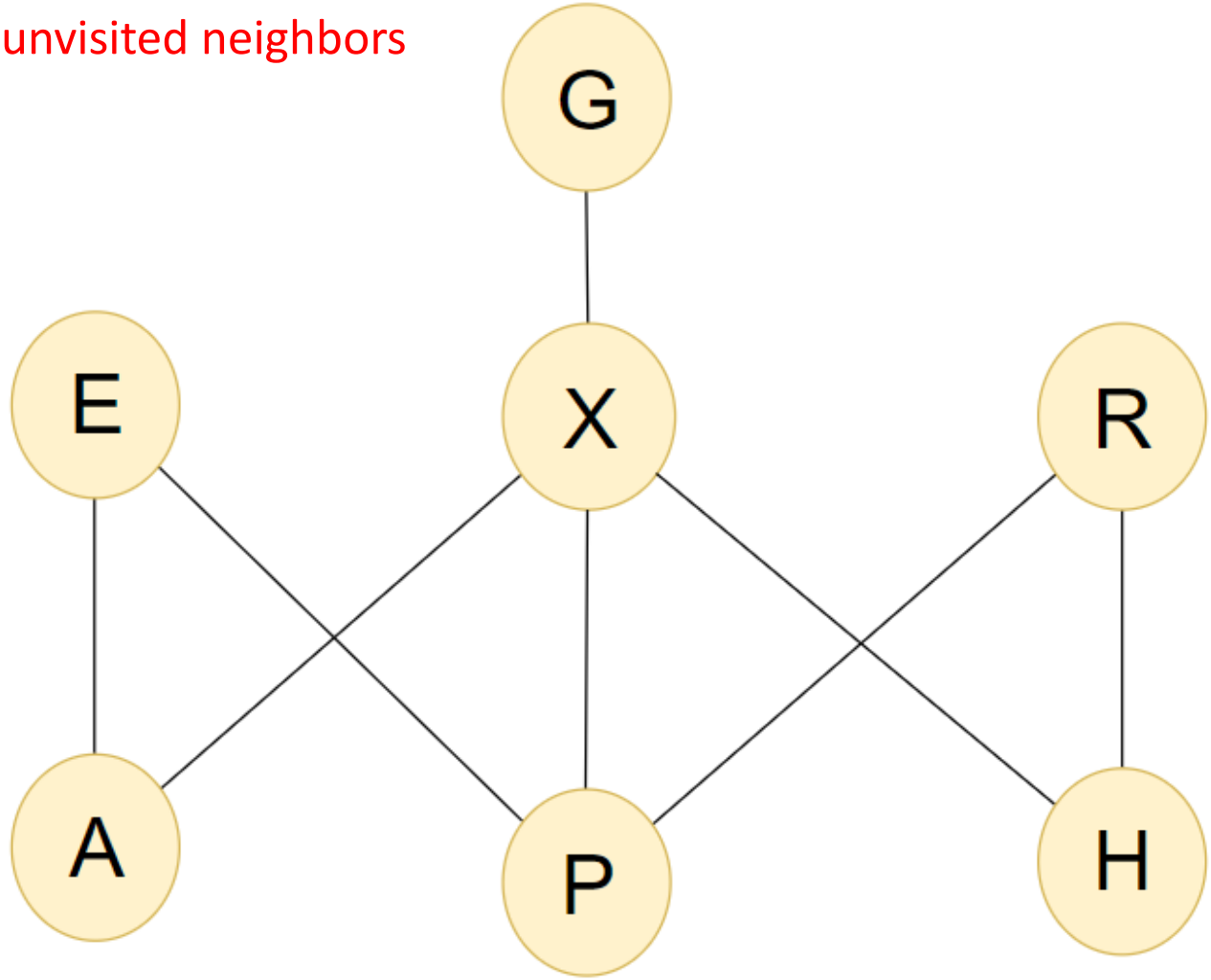
dequeue H and enqueue onto traversalOrder

H has no unvisited neighbors

front Vertex Queue back

E A P X R G H

front Traversal Order Queue back

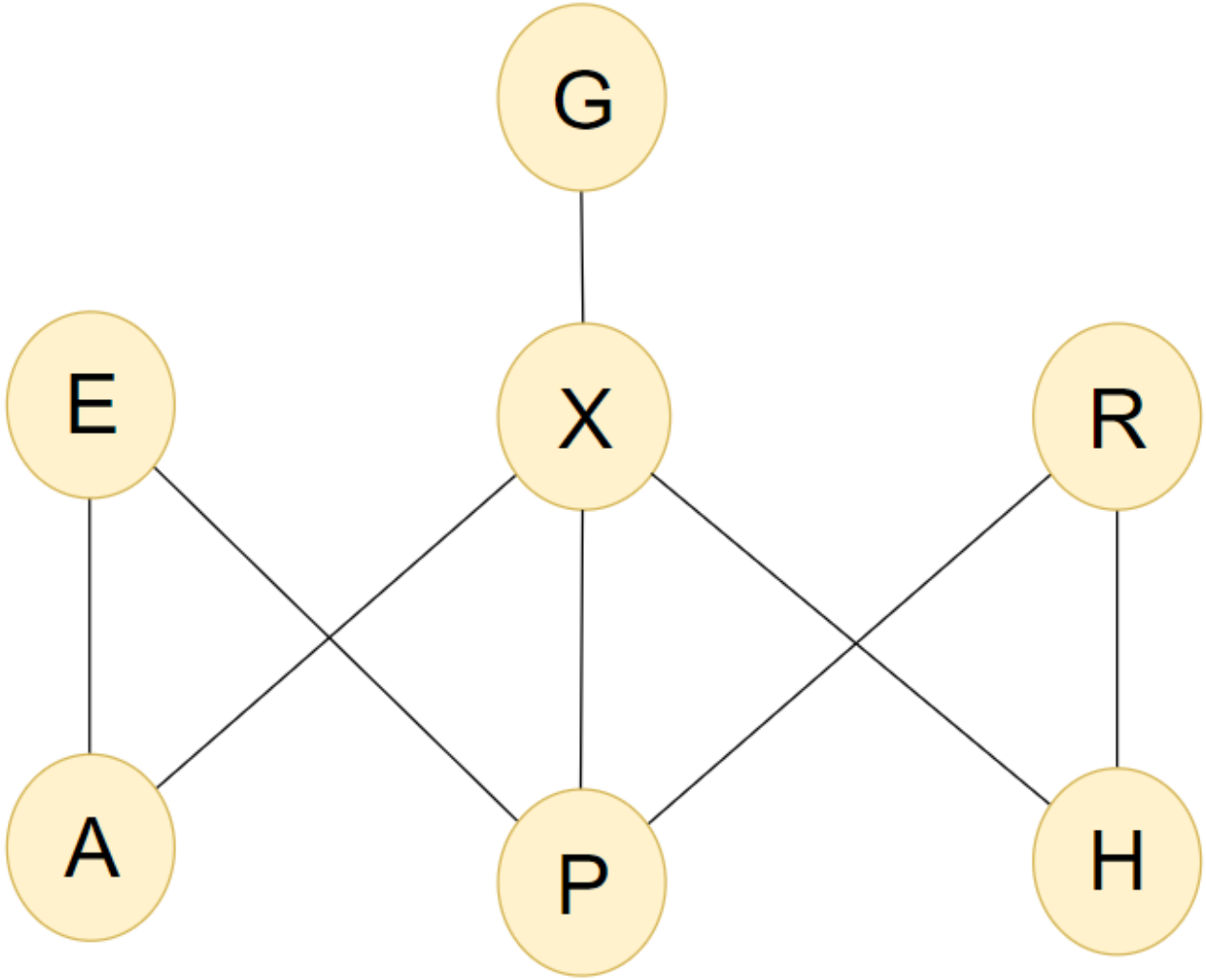


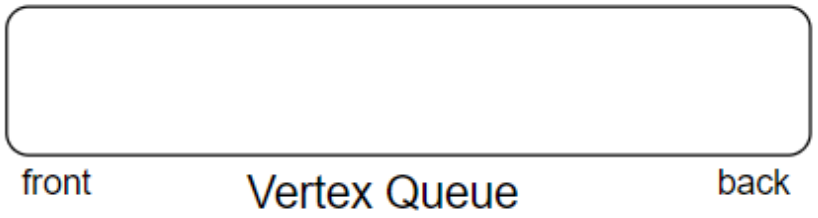
the traversal order is: EAPXRGH

front Vertex Queue back

E A P X R G H

front Traversal Order Queue back

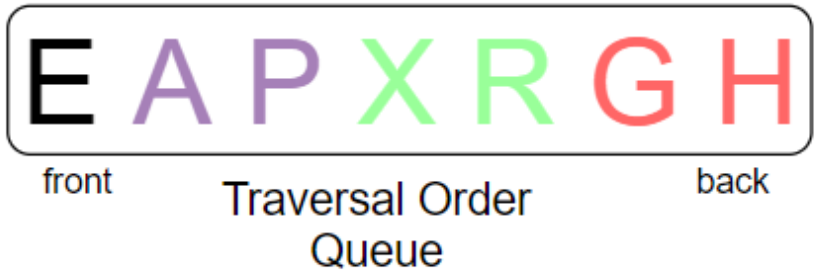


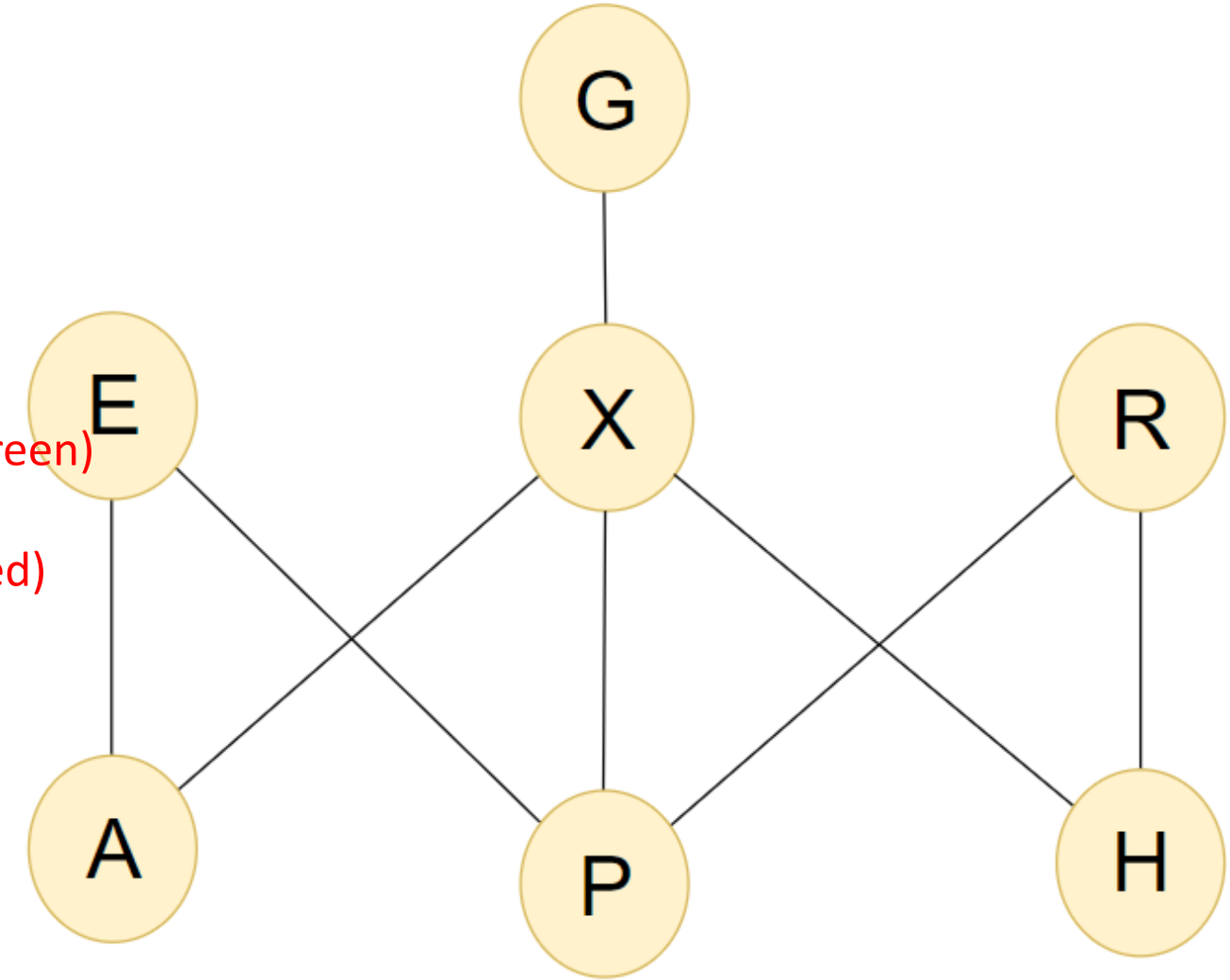


all of E's neighbors come first (in purple)

next come vertices one edges away from E (in green)

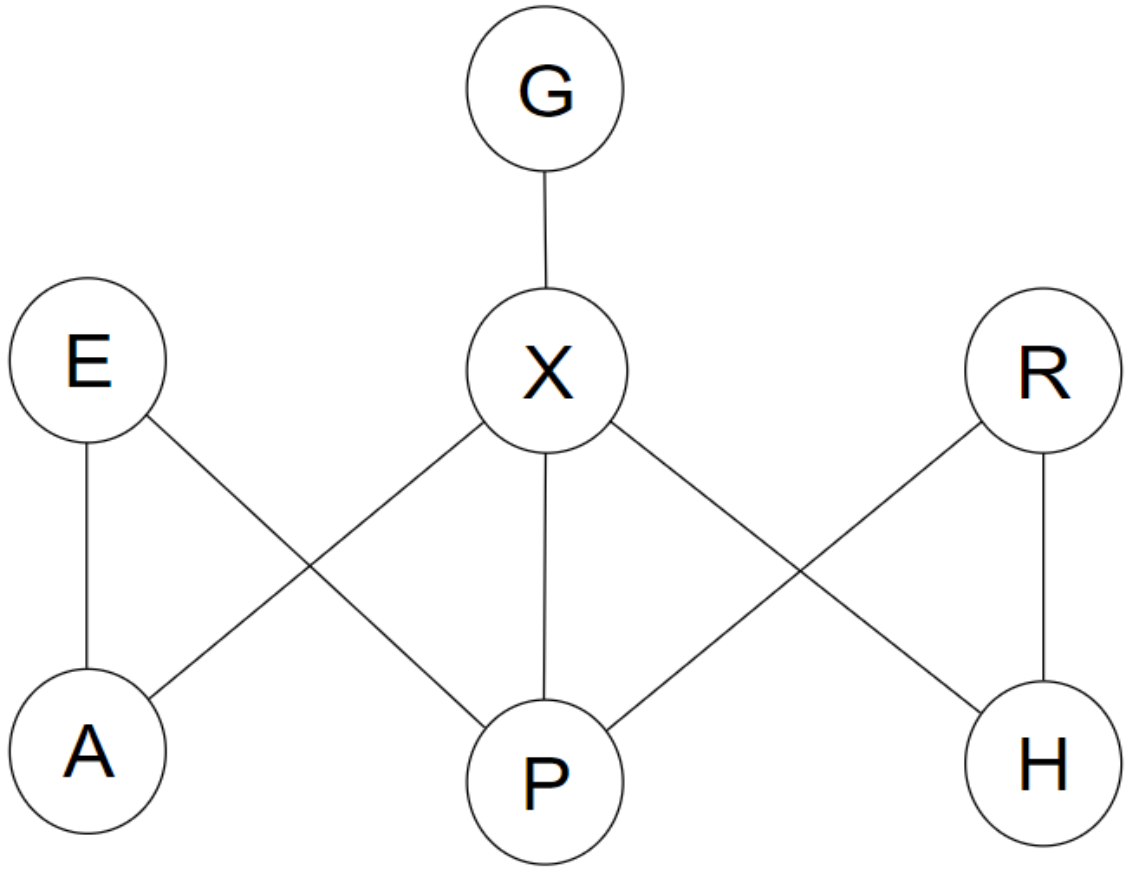
next come vertices two edges away from E (in red)





Breadth-First Example

- Trace a breadth first traversal starting at vertex H.

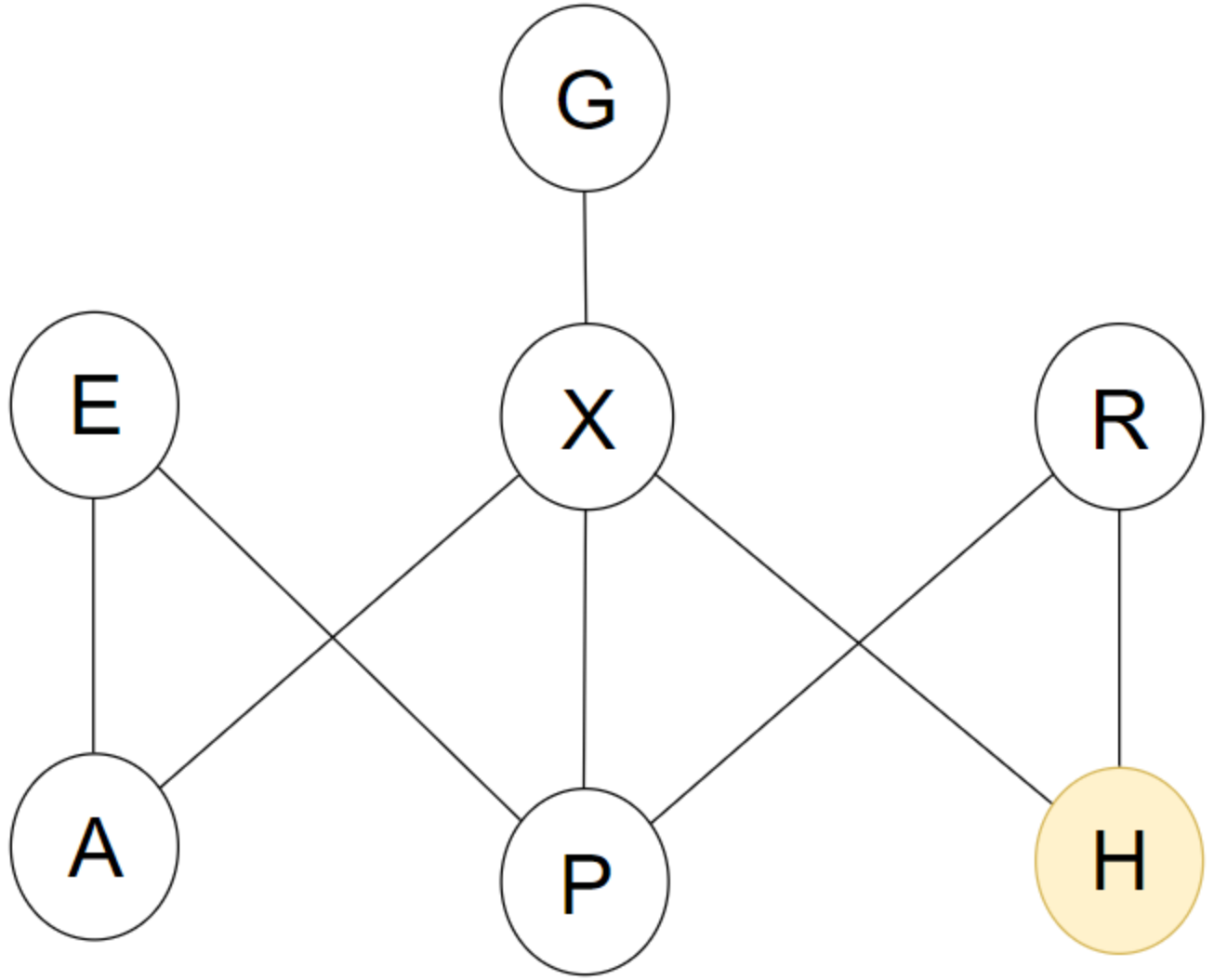


enqueue the origin (H) onto the vertexQueue and mark as visited

front Vertex Queue back

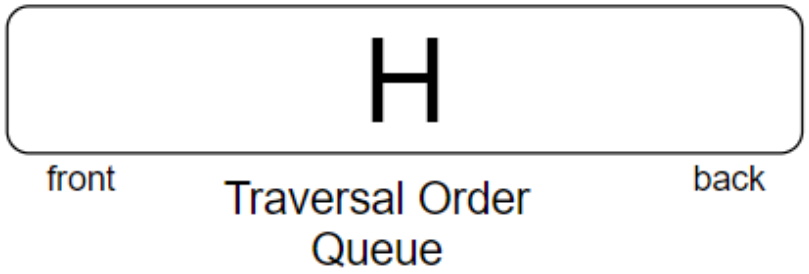
front H back

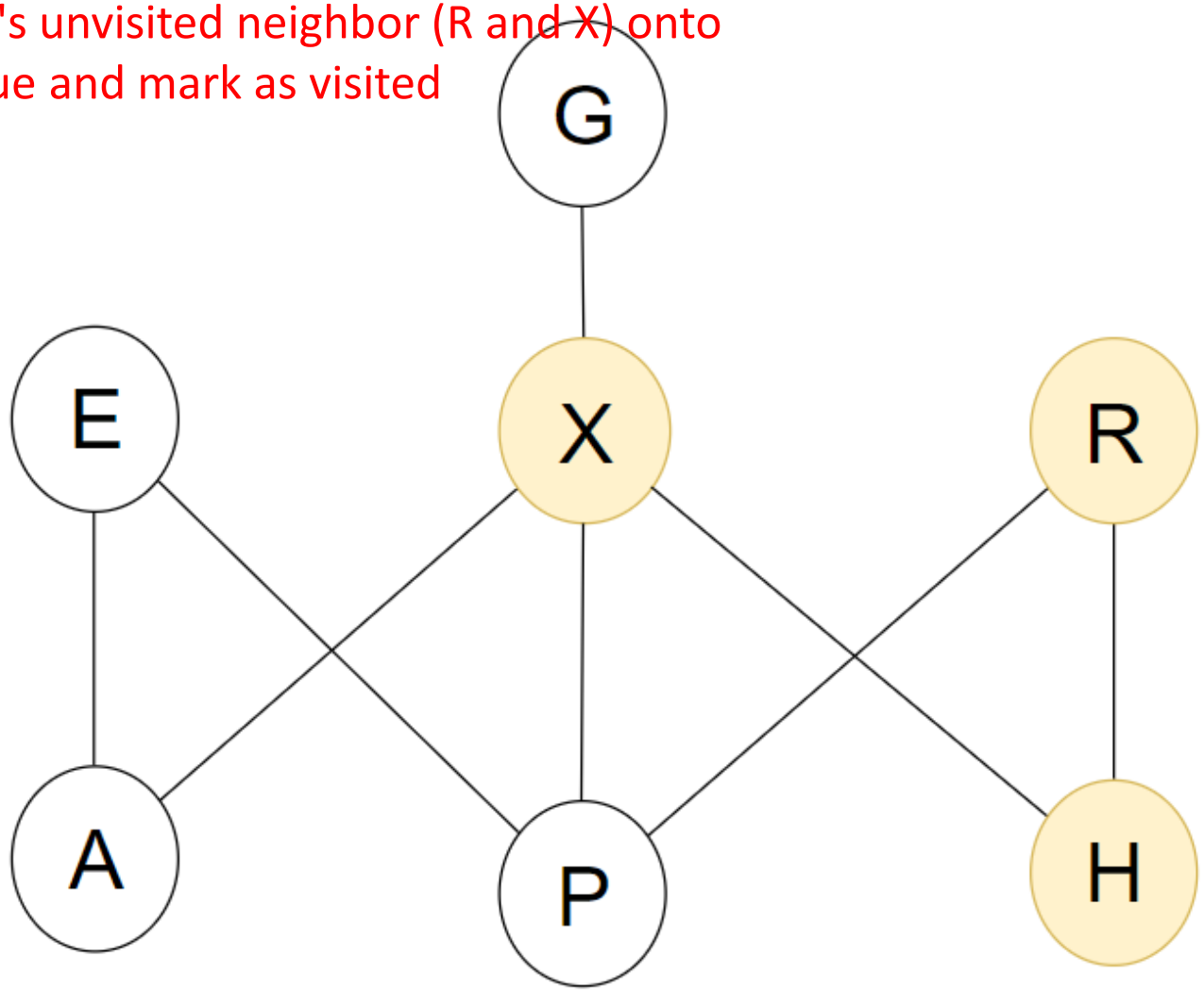
Traversal Order Queue



dequeue H and enqueue onto traversalOrder

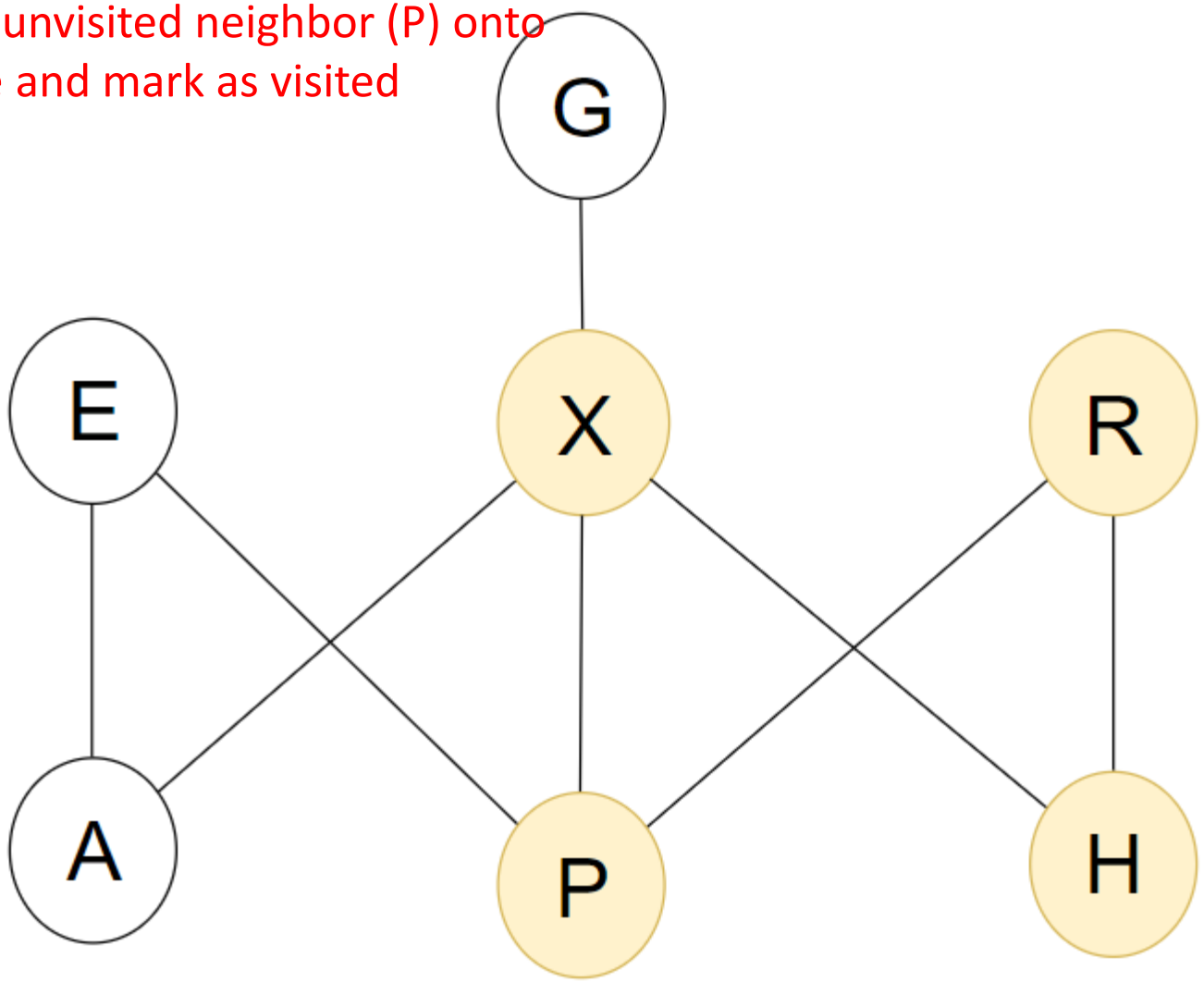
enqueue H's unvisited neighbor (R and X) onto vertexQueue and mark as visited





dequeue R and enqueue onto traversalOrder

enqueue R's unvisited neighbor (P) onto vertexQueue and mark as visited



dequeue X and enqueue onto traversalOrder

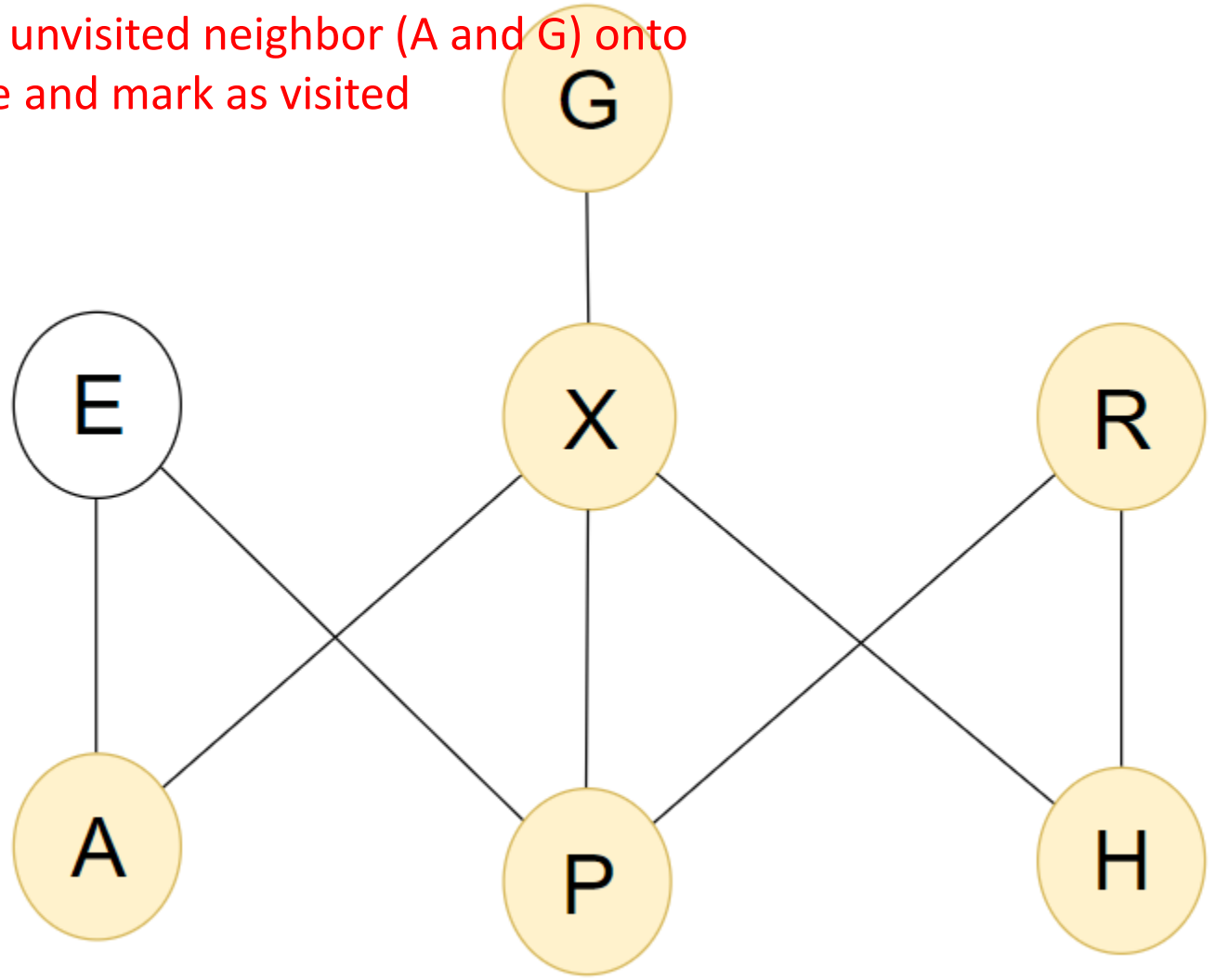
enqueue X's unvisited neighbor (A and G) onto vertexQueue and mark as visited

P A G

front Vertex Queue back

H R X

front Traversal Order Queue back



dequeue P and enqueue onto traversalOrder

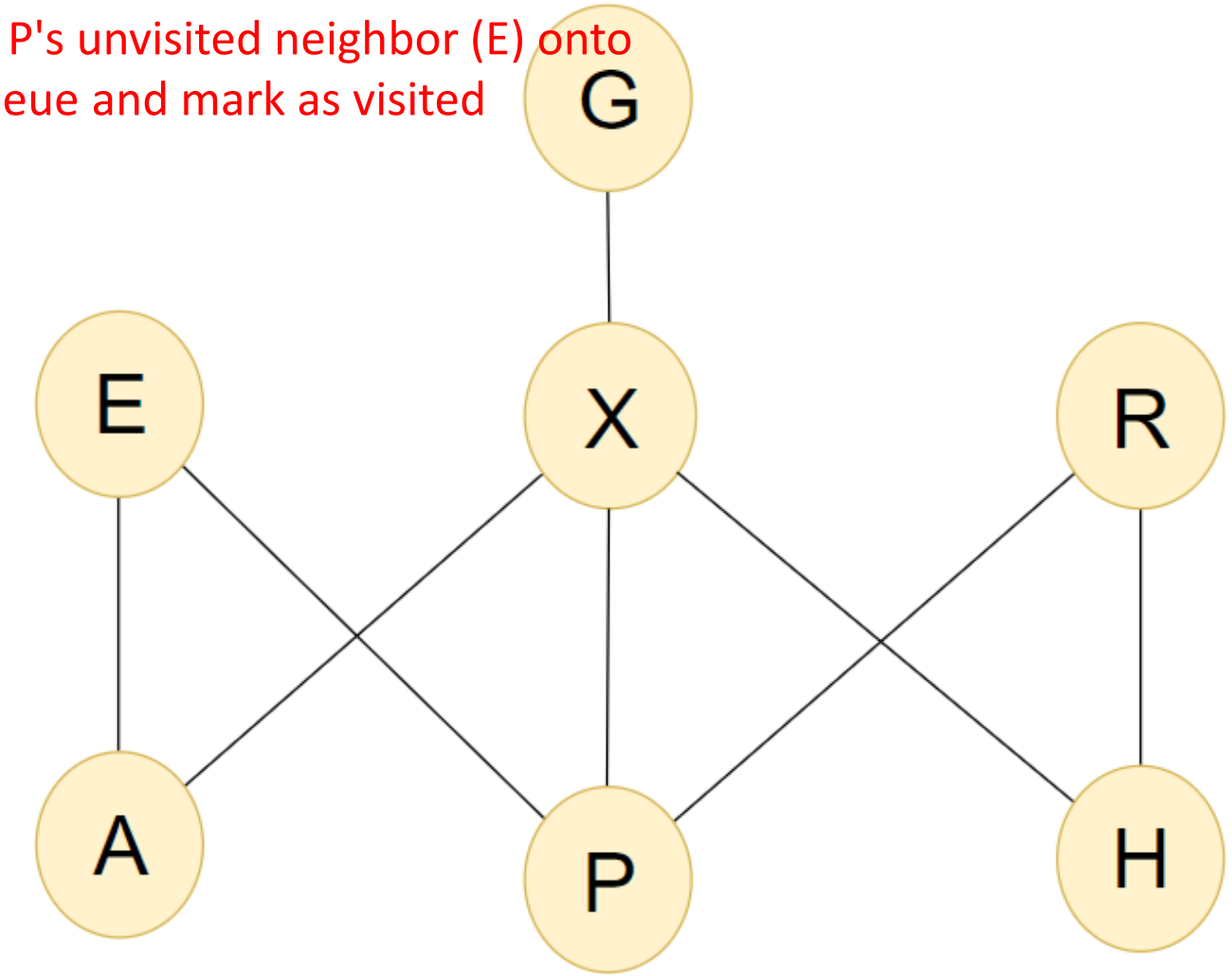
enqueue P's unvisited neighbor (E) onto vertexQueue and mark as visited

A G E

front Vertex Queue back

H R X P

front Traversal Order Queue back



dequeue A and enqueue onto traversalOrder

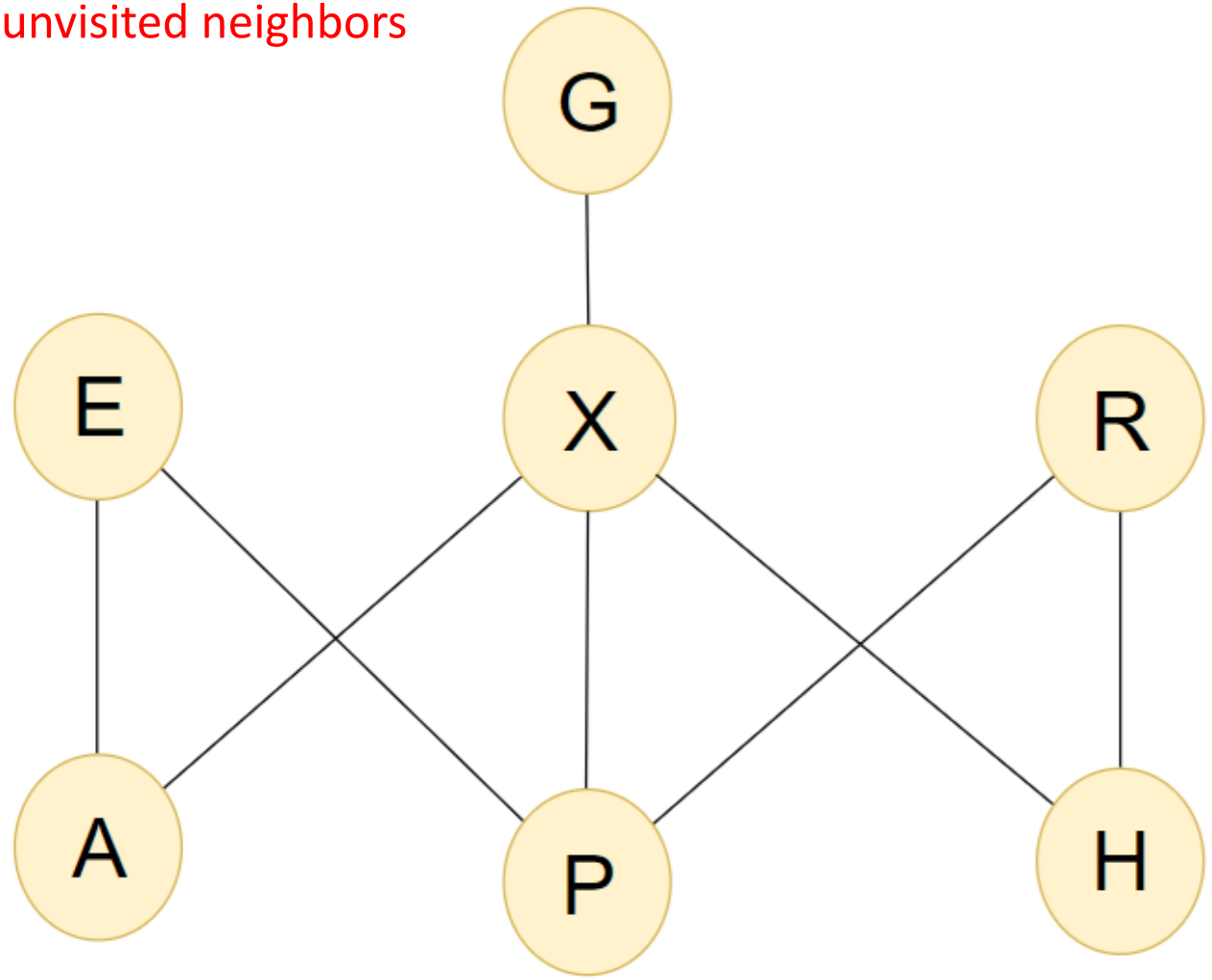
A has no unvisited neighbors

G E

front Vertex Queue back

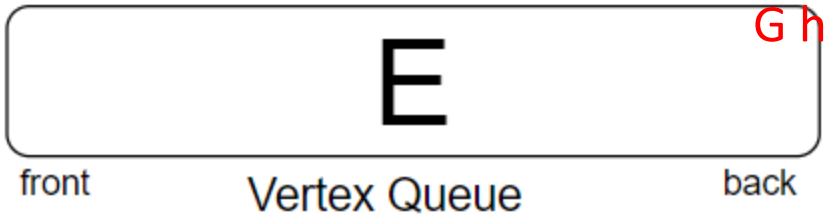
H R X P A

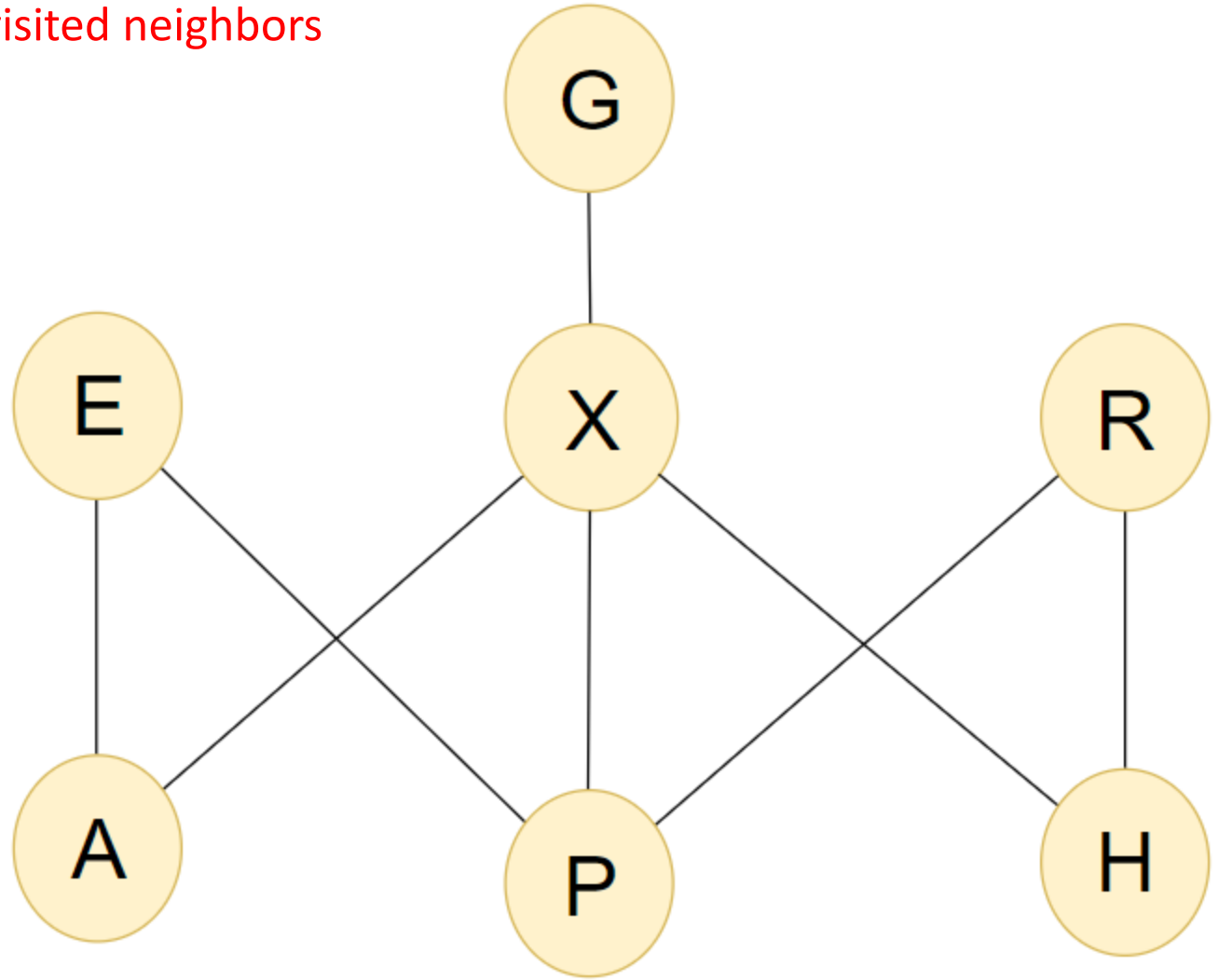
front Traversal Order Queue back



dequeue G and enqueue onto traversalOrder

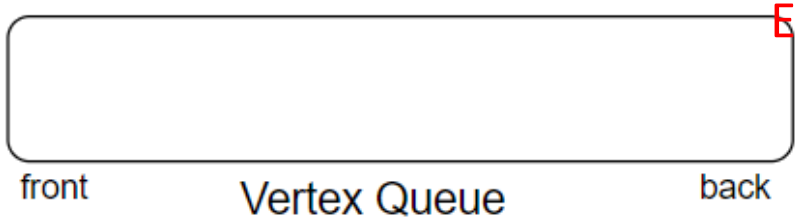
G has no unvisited neighbors

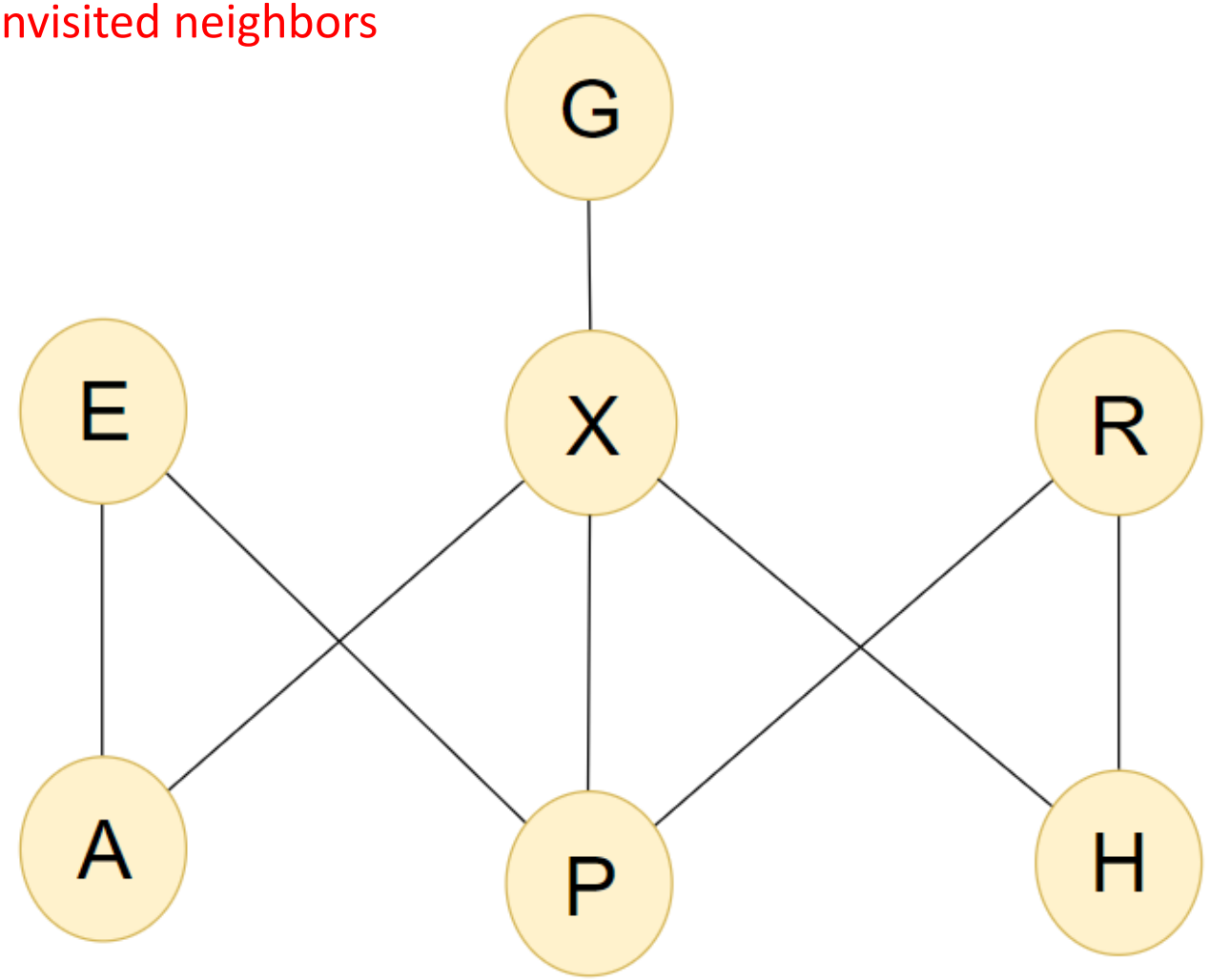




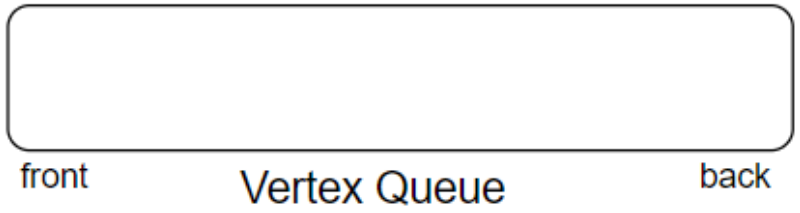
dequeue E and enqueue onto traversalOrder

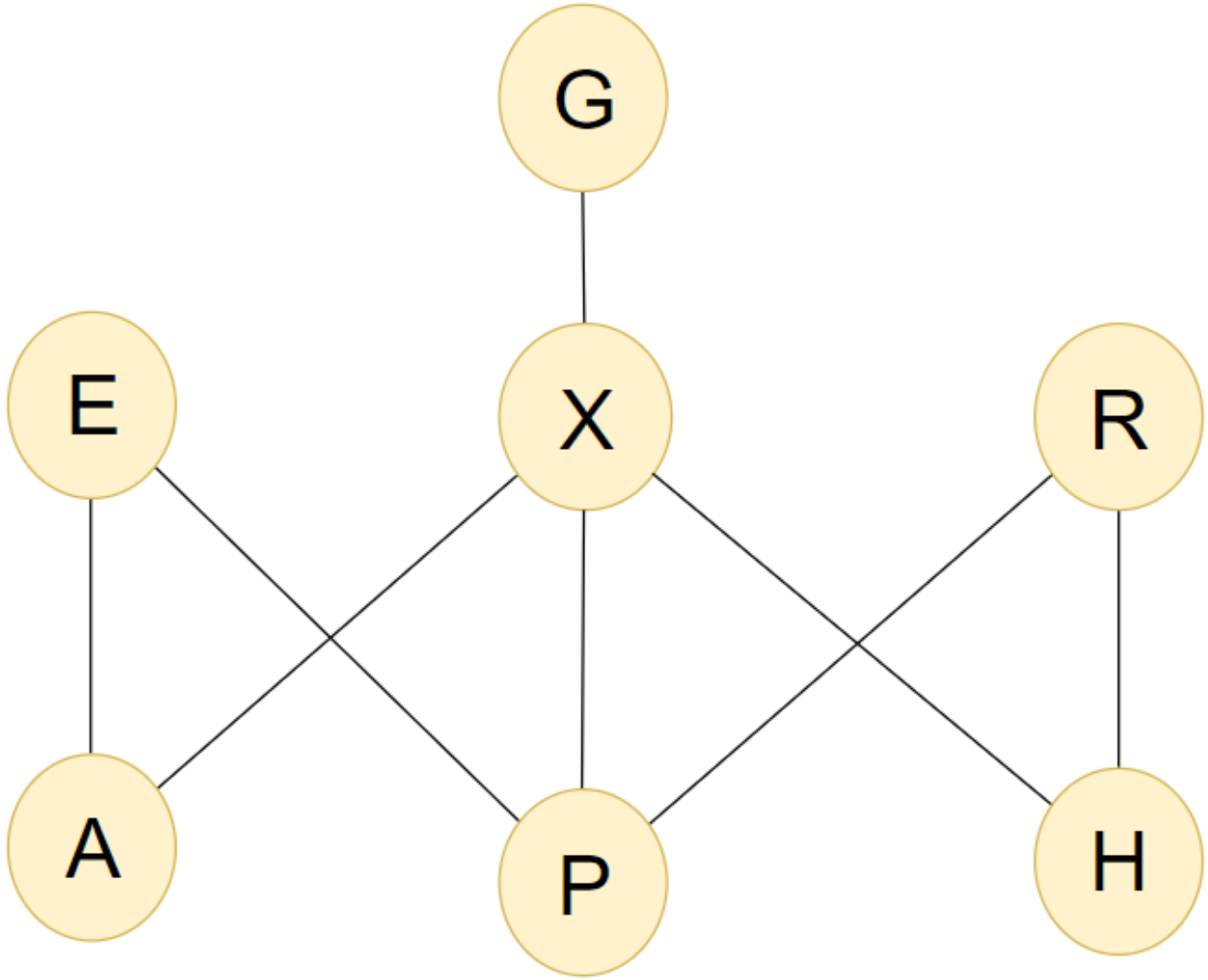
E has no unvisited neighbors





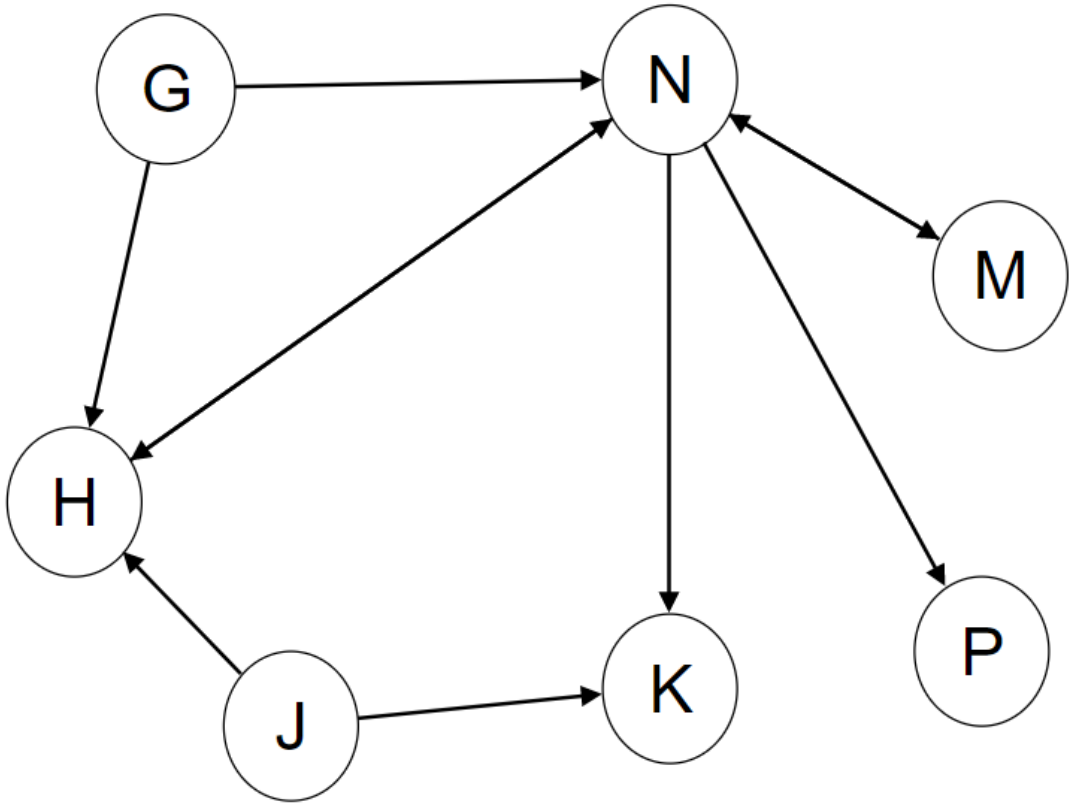
the traversal order is: HRXPAGE



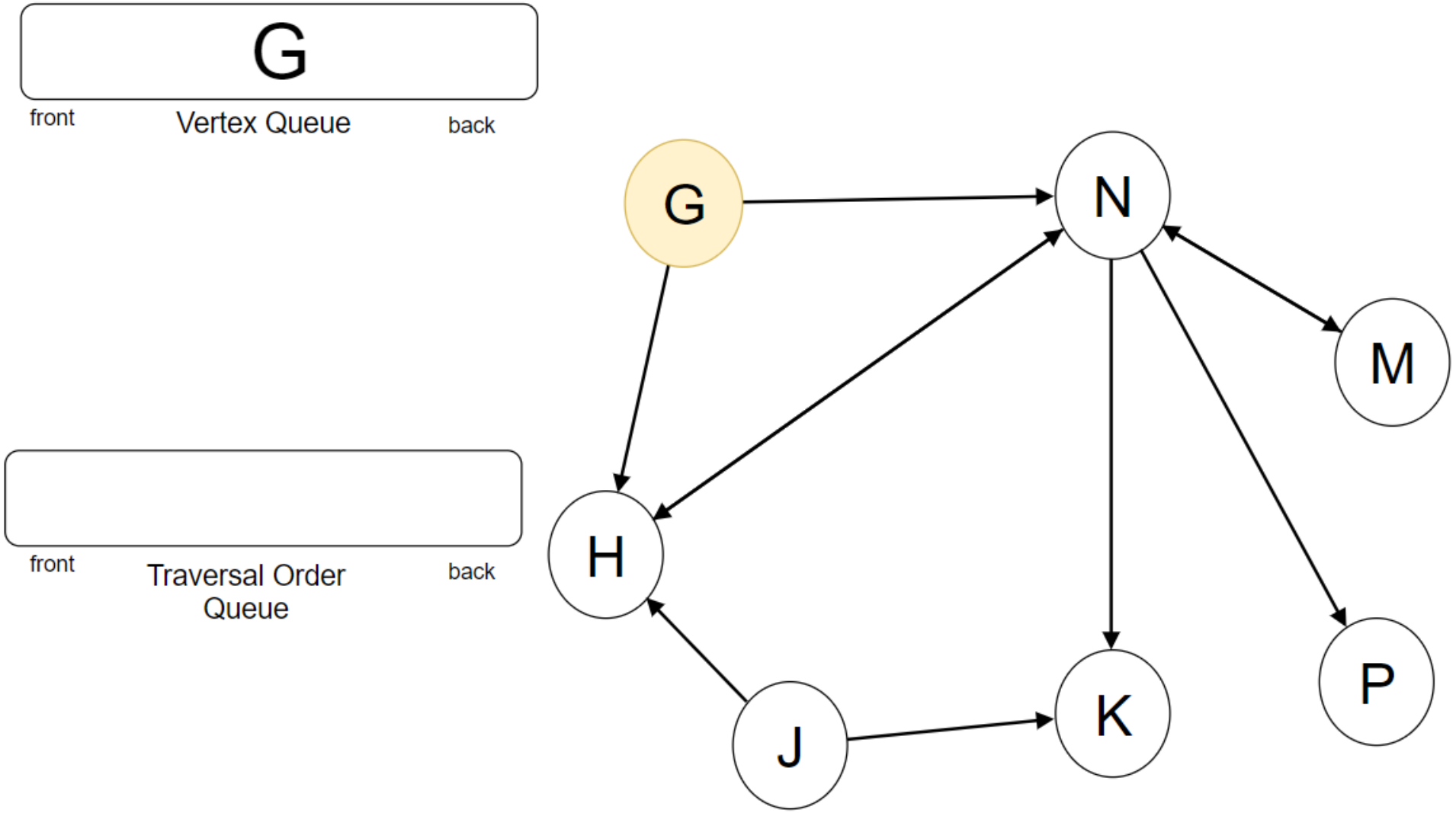


Breadth-First Example

- Trace a breadth first traversal starting at vertex G.



enqueue the origin (G) onto the vertexQueue and mark as visited



dequeue G and enqueue onto traversalOrder

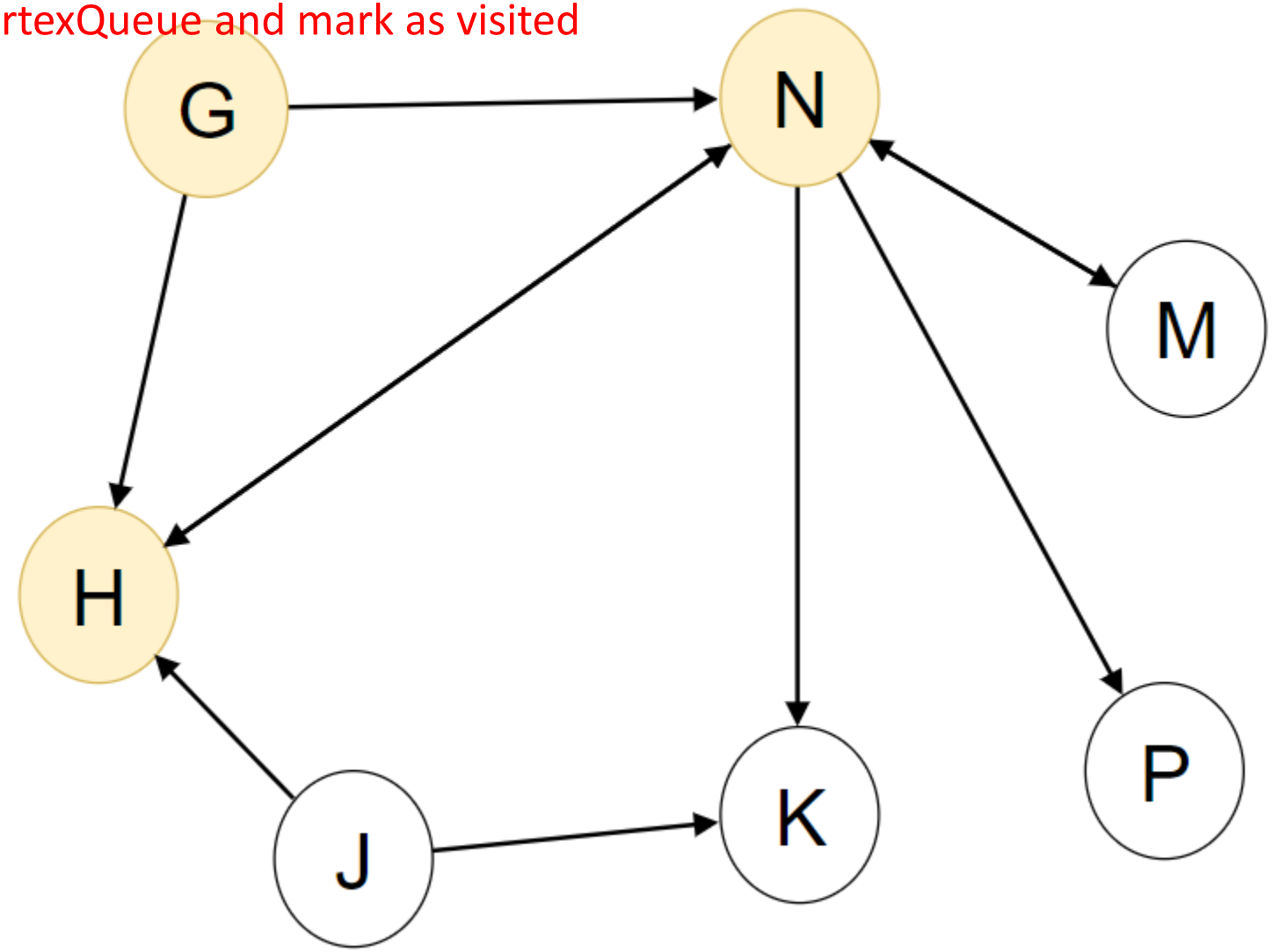
enqueue G's unvisited neighbor (H and N) onto vertexQueue and mark as visited

H	N		

front — Vertex Queue — back

G			

front — Traversal Order Queue — back



dequeue X and enqueue onto traversalOrder

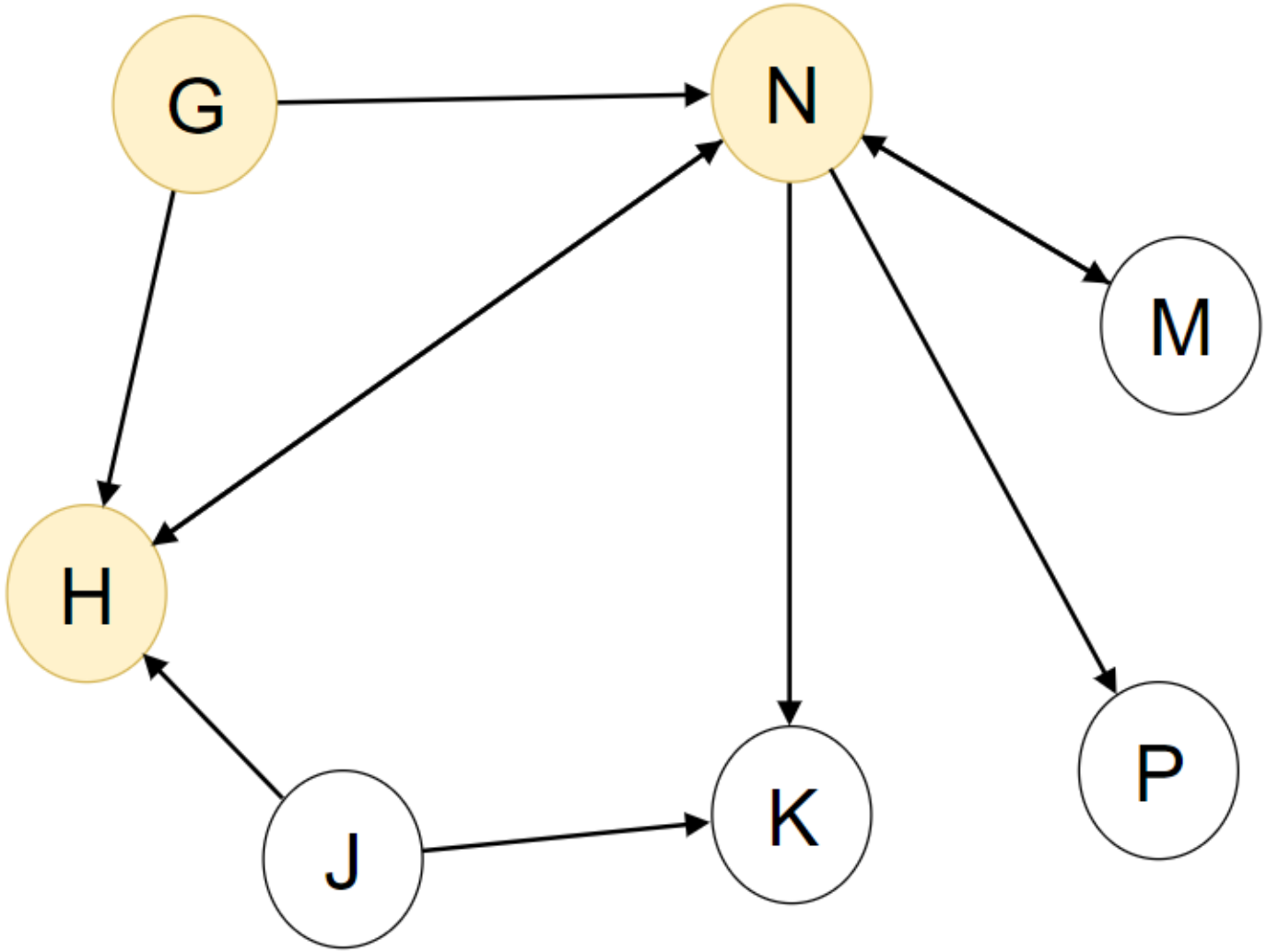
H has no unvisited neighbors

N

front Vertex Queue back

G H

front Traversal Order Queue back



dequeue N and enqueue onto traversalOrder

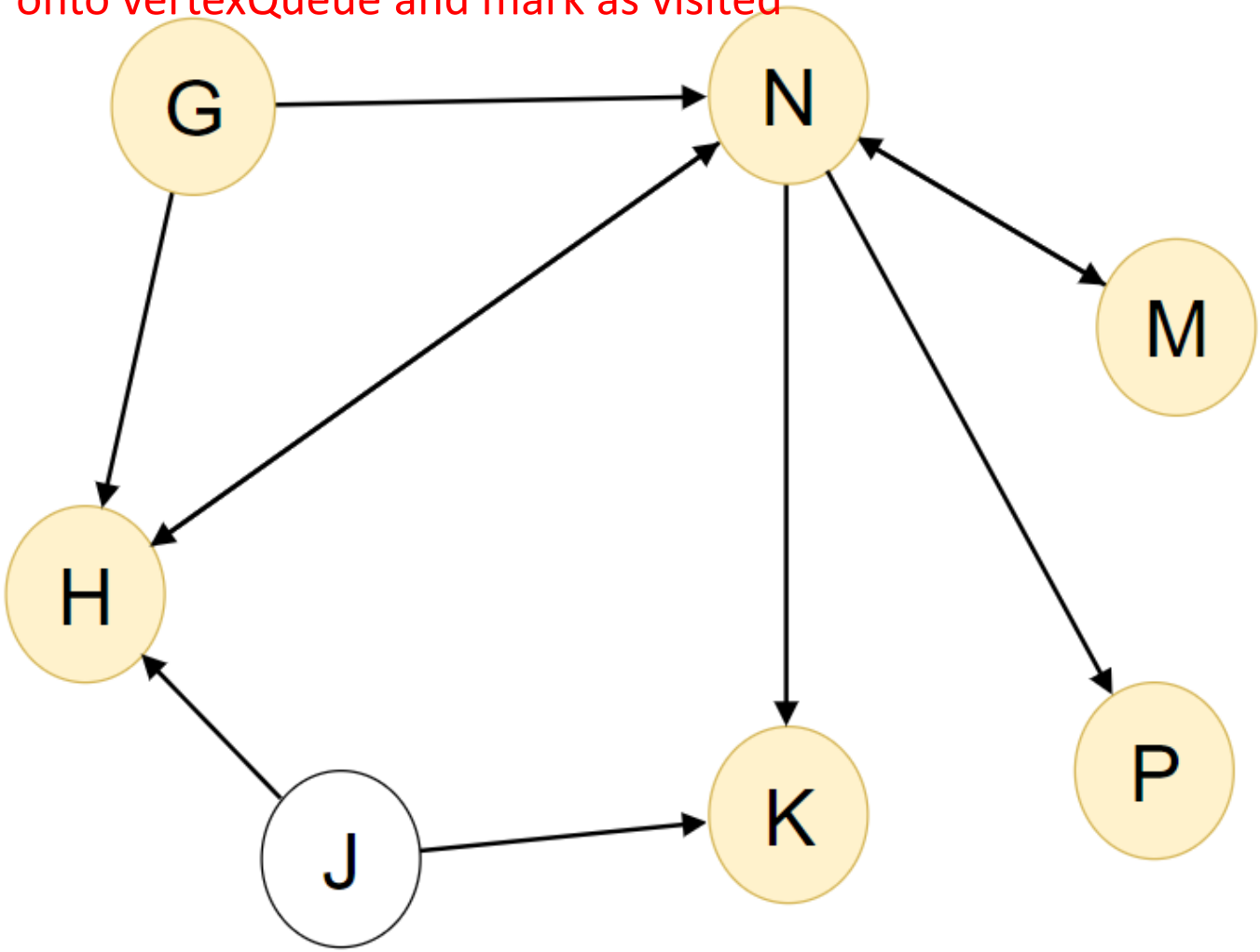
enqueue N's unvisited neighbor (K, M, and P) onto vertexQueue and mark as visited

K M P

front — Vertex Queue — back

G H N

front — Traversal Order Queue — back



dequeue K and enqueue onto traversalOrder

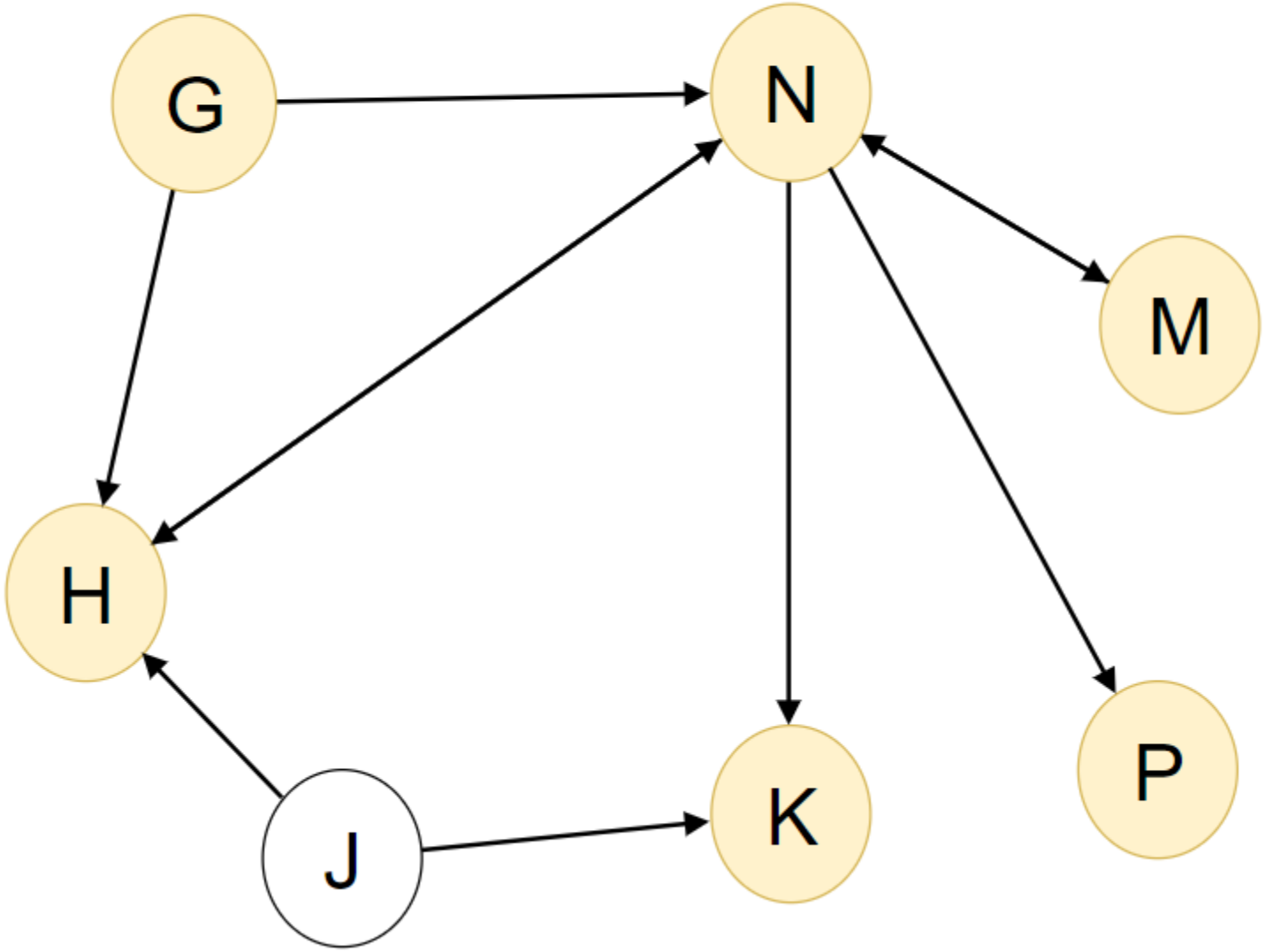
K has no unvisited neighbors

M P

front Vertex Queue back

G H N K

front Traversal Order Queue back



dequeue M and enqueue onto traversalOrder

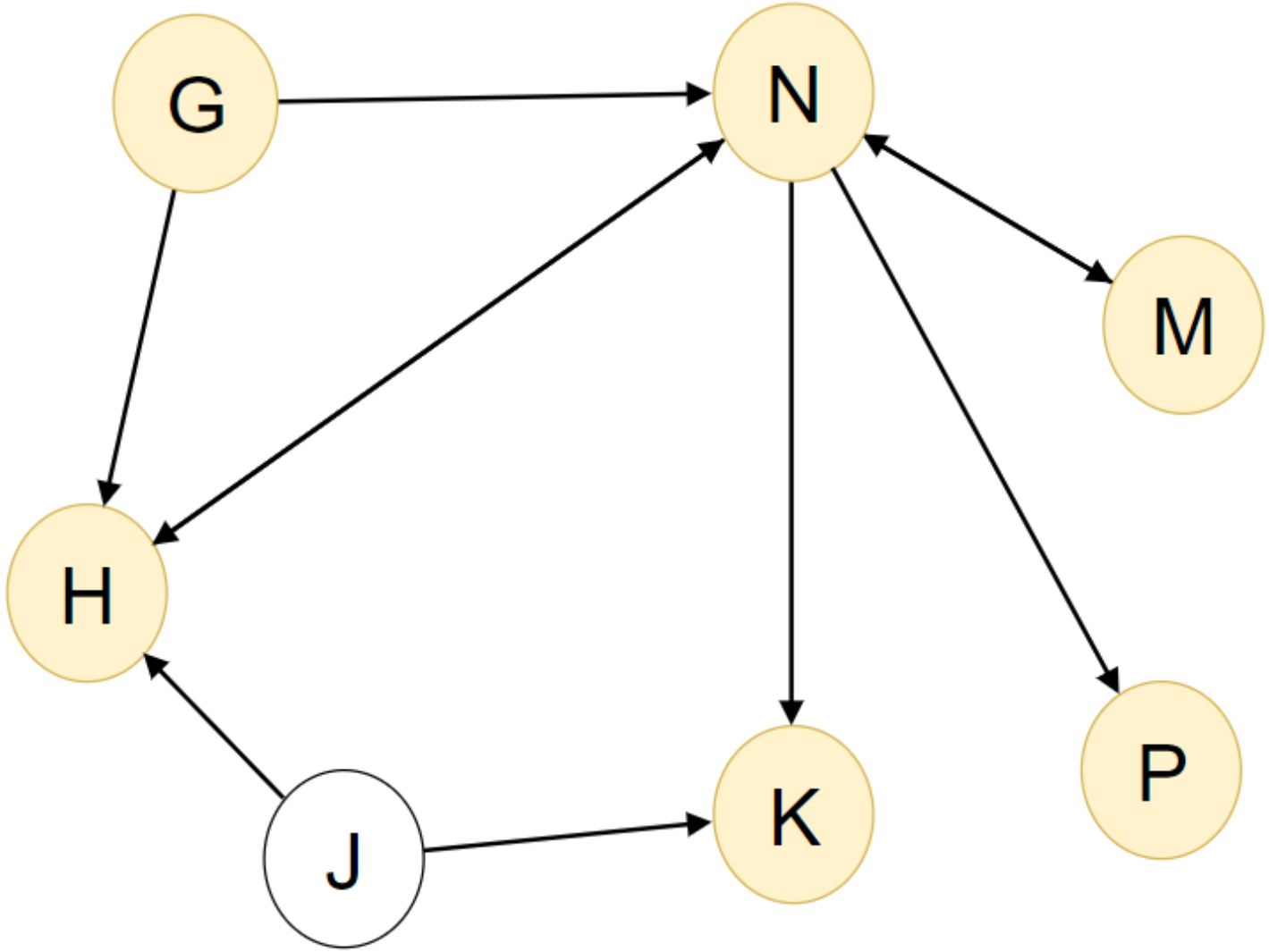
M has no unvisited neighbors

P

front Vertex Queue back

G H N K M

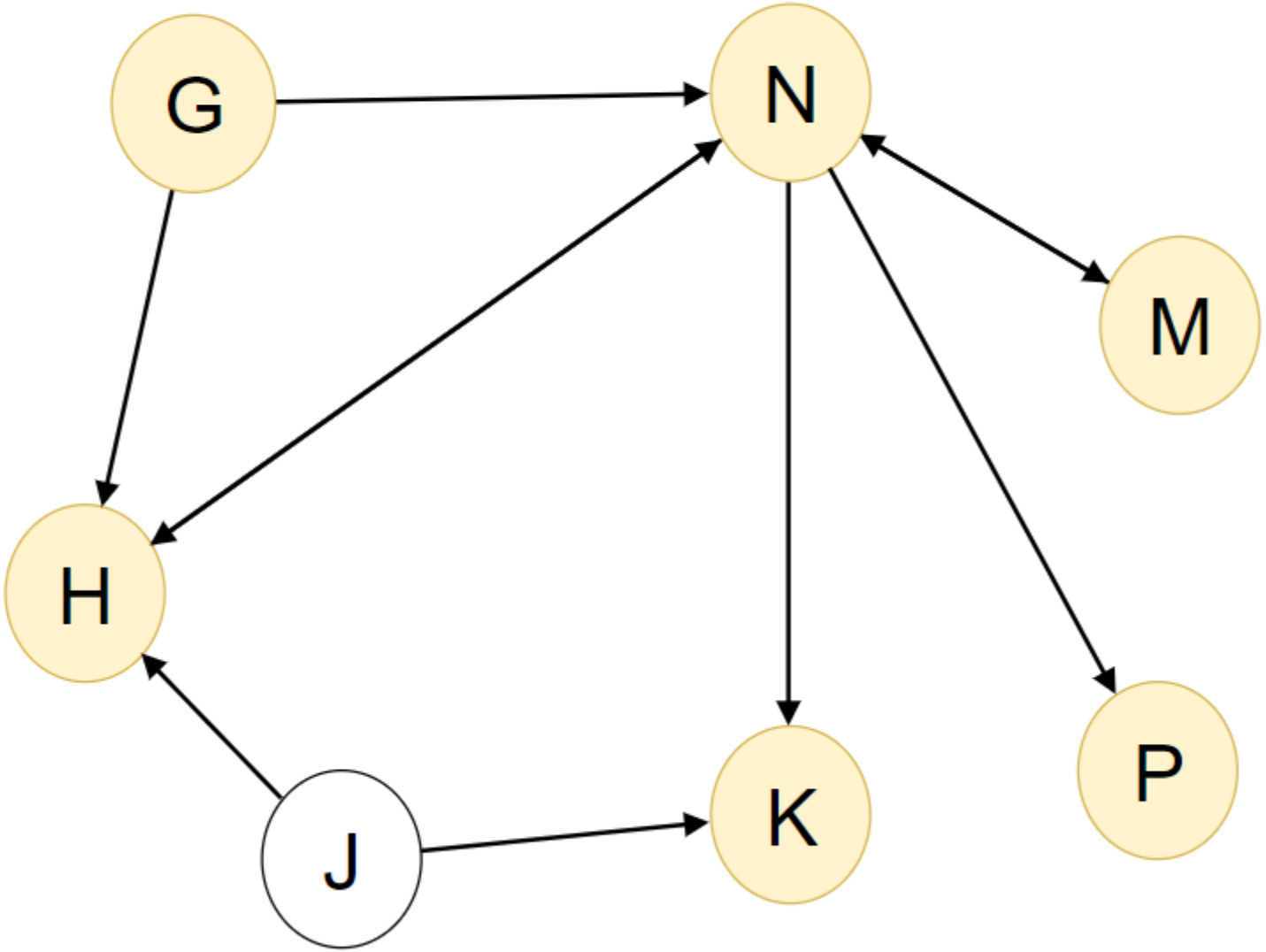
front Traversal Order Queue back



dequeue P and enqueue onto traversalOrder

P has no unvisited neighbors

front Vertex Queue back



G H N K M P

front Traversal Order Queue back

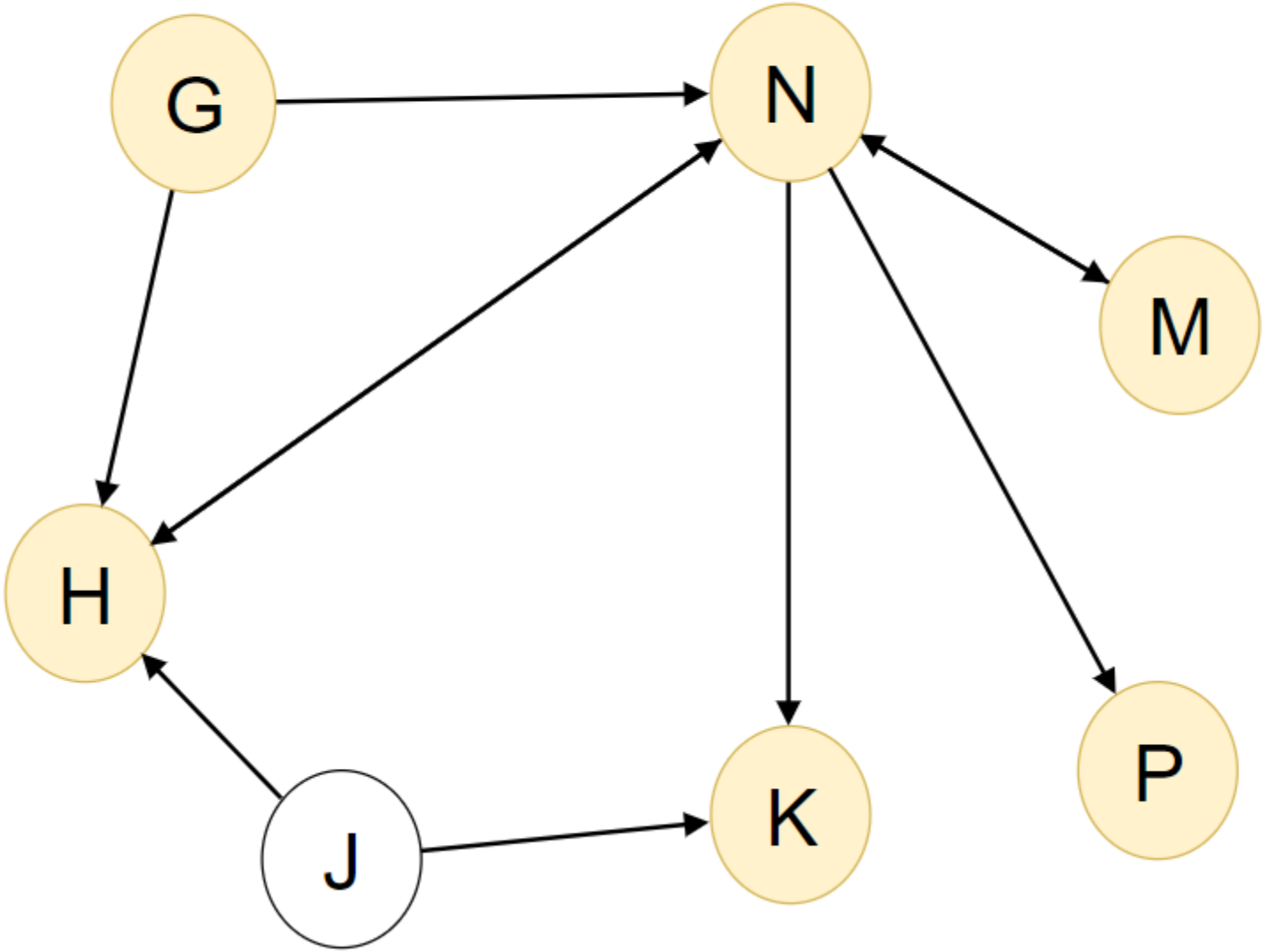
front Vertex Queue back

the traversal order is: GHNKMP

note: J is **not** part of the traversal!

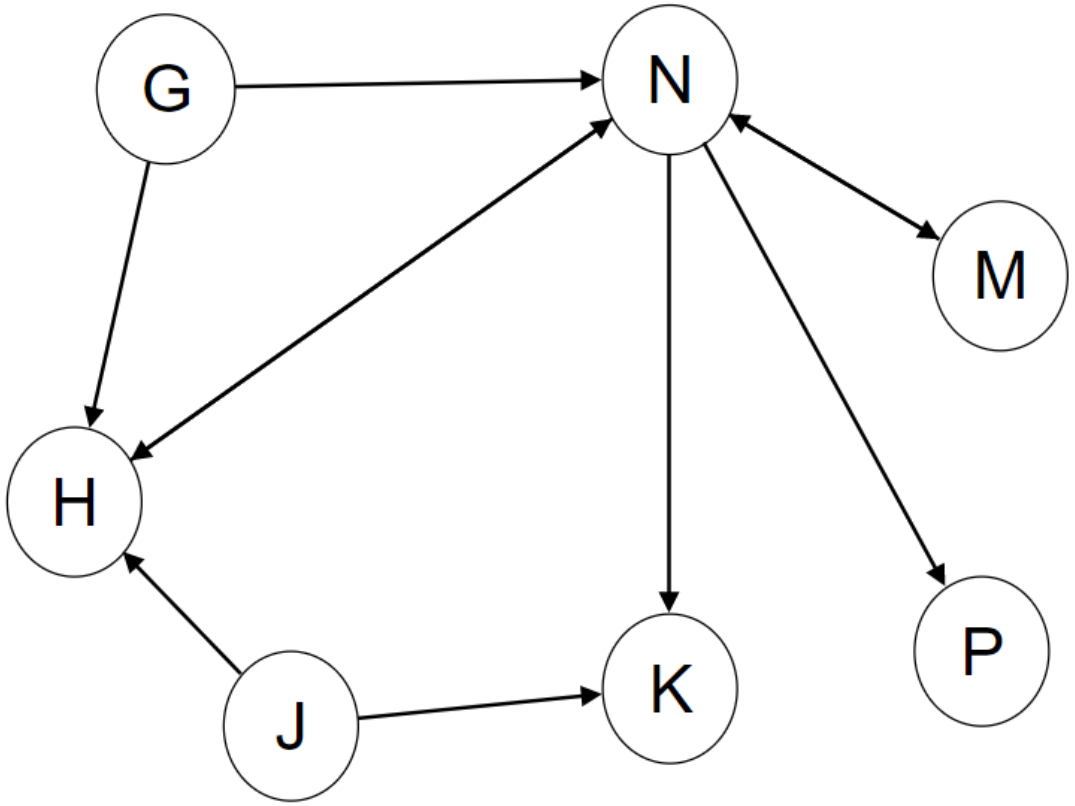
G H N K M P

front Traversal Order Queue back



Breadth-First Example

- Trace a breadth first traversal starting at vertex K.

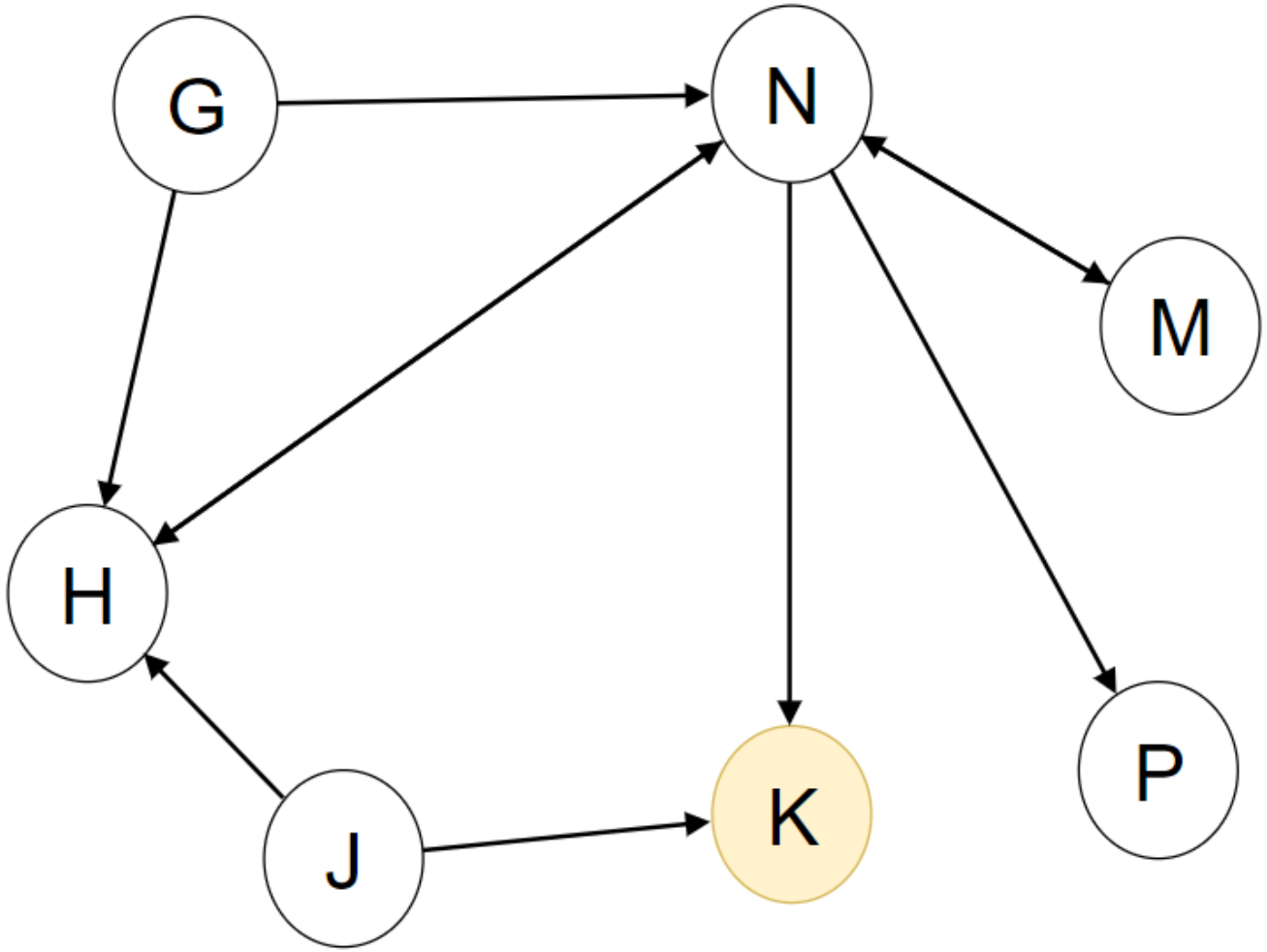


front Vertex Queue back

the traversal order is: K

K

front Traversal Order Queue back



Paths

- We can determine if there is a path between two vertices with a traversal.
  - Each time you reach a vertex, check if it's the destination.

Shortest Path

- There can be more than one path between vertices.
- We might not want just *any* path, but the path with the shortest length (the fewest edges).
- We can use a modified breadth-first traversal to find the shortest path.

Finding Shortest Path

- In breadth-first, each vertex is placed into a queue as it is visited.
- To modify to find the shortest path:
  - keep track of the predecessor vertex (where we came from)
  - keep track of the path length (1 + the length to the predecessor)

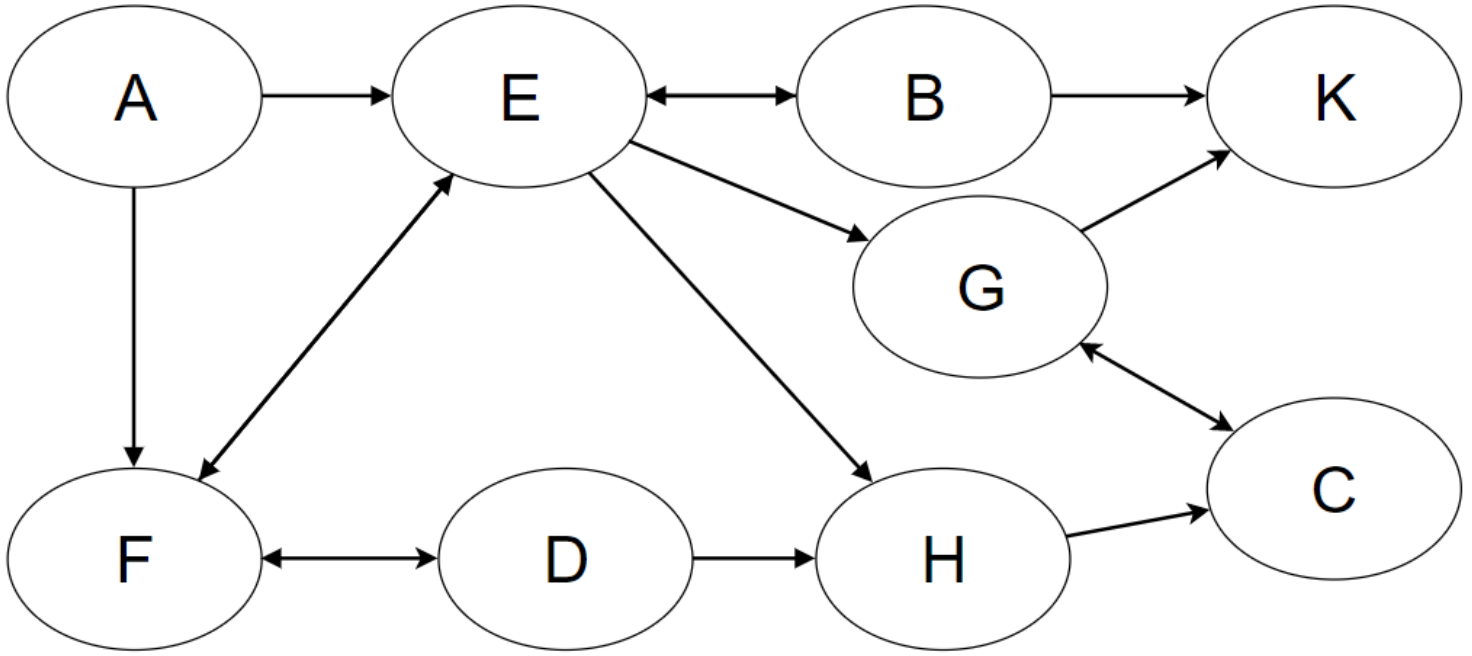
Shortest Path Algorithm

enqueue the origin onto the vertexQueue and mark as visited
while we haven't reached the endVertex and the vertexQueue is not empty
dequeue a vertex from the vertexQueue (frontVertex)
for all of frontVertex's ***unvisited*** *neighbors*
mark as visited
set the length of the path to the neighbor to 1 + length to frontIndex
set the predecessor of the neighbor to frontVertex
enqueue onto the vertexQueue
if the neighbor is the destination, we are done!

to find the path:
push the endVertex onto a stack
while the top of the stack has a predecessor
push predecessor onto stack
the contents of the stack are the path

Shortest Path Example

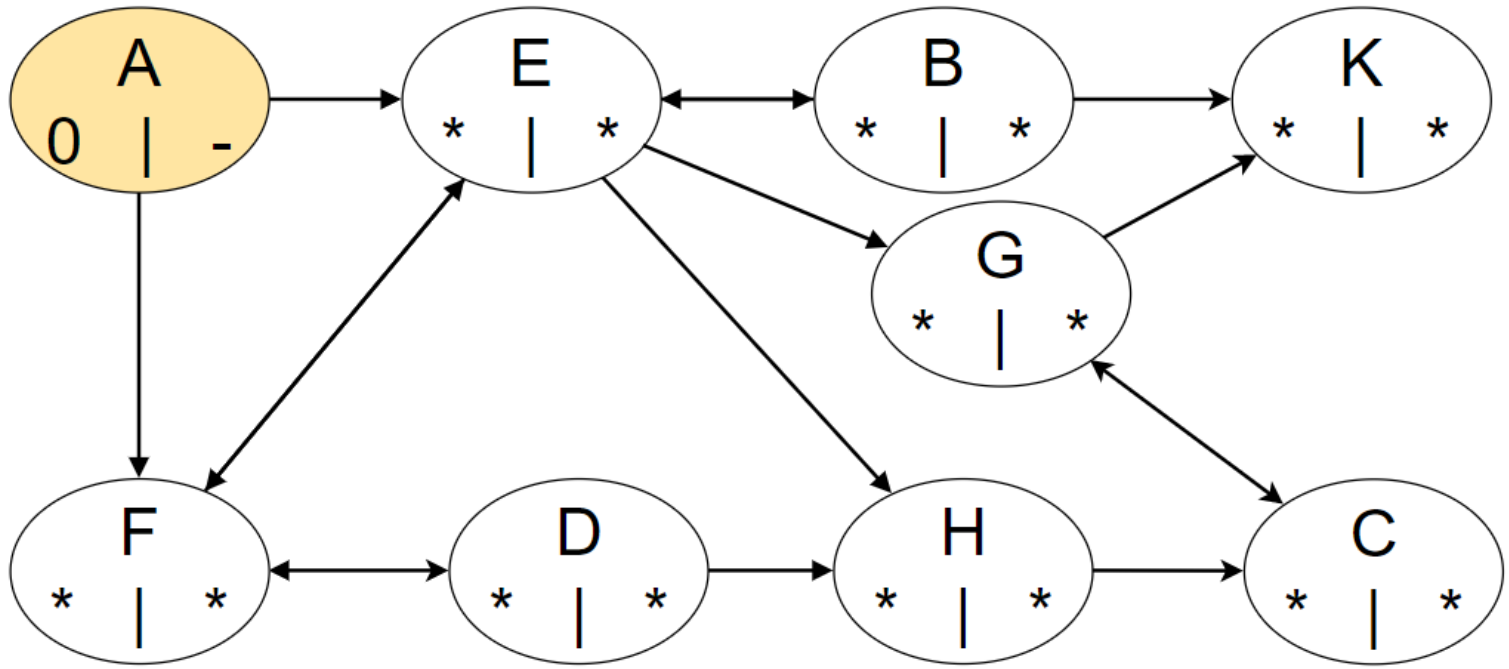
- Find a shortest path from vertex A to C.



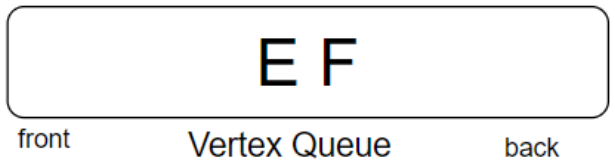
front Vertex Queue back

frontVertex: A

dequeue frontVertex from the vertexQueue
for all of frontVertex's unvisited neighbors
set the length of the path
set the predecessor of the neighbor
enqueue onto the vertexQueue

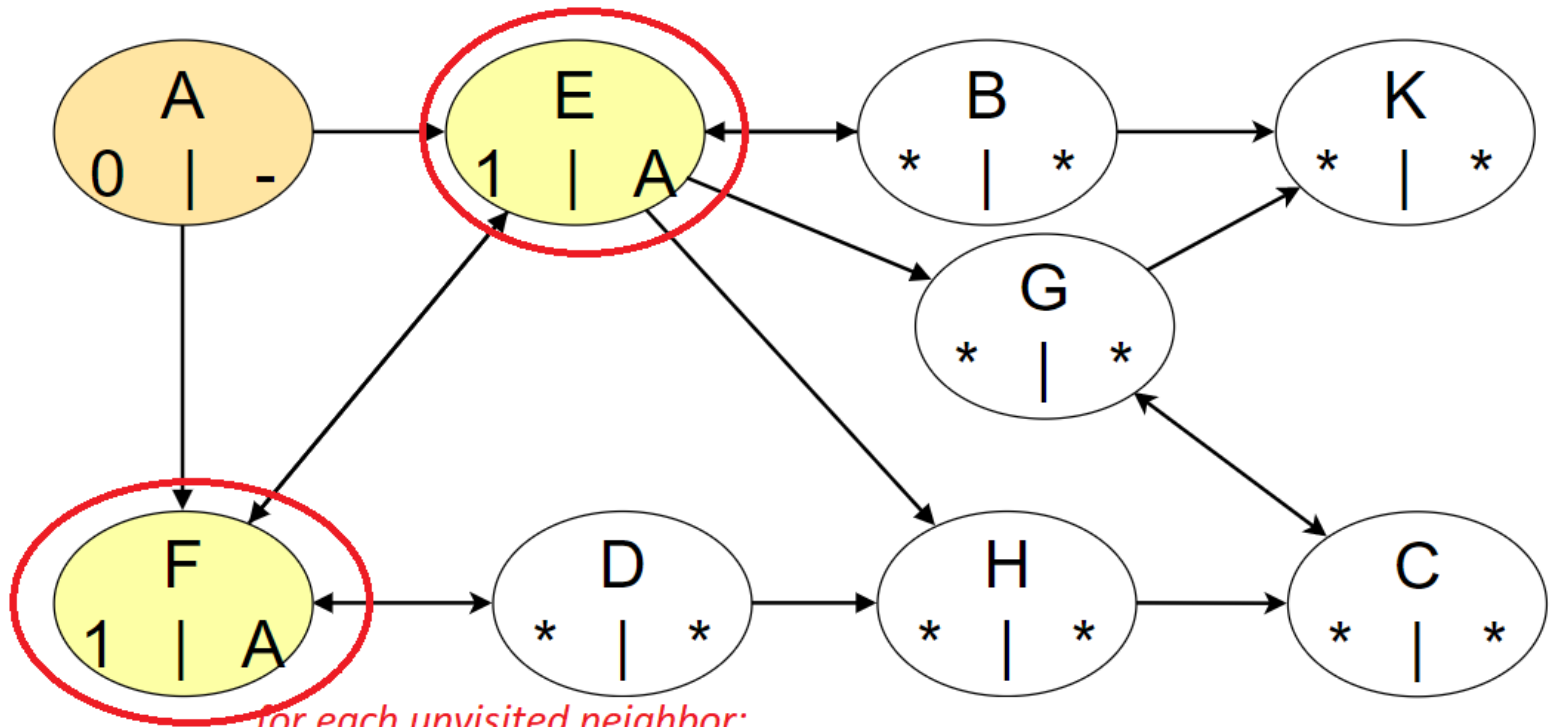


mark the origin as visited
the origin is frontVertext

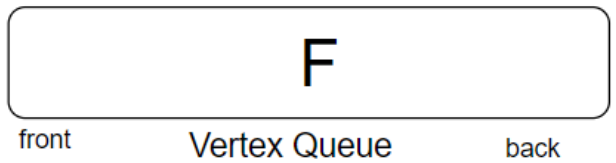


frontVertex: A

dequeue frontVertex from the vertexQueue
for all of frontVertex's unvisited neighbors
set the length of the path
set the predecessor of the neighbor
enqueue onto the vertexQueue

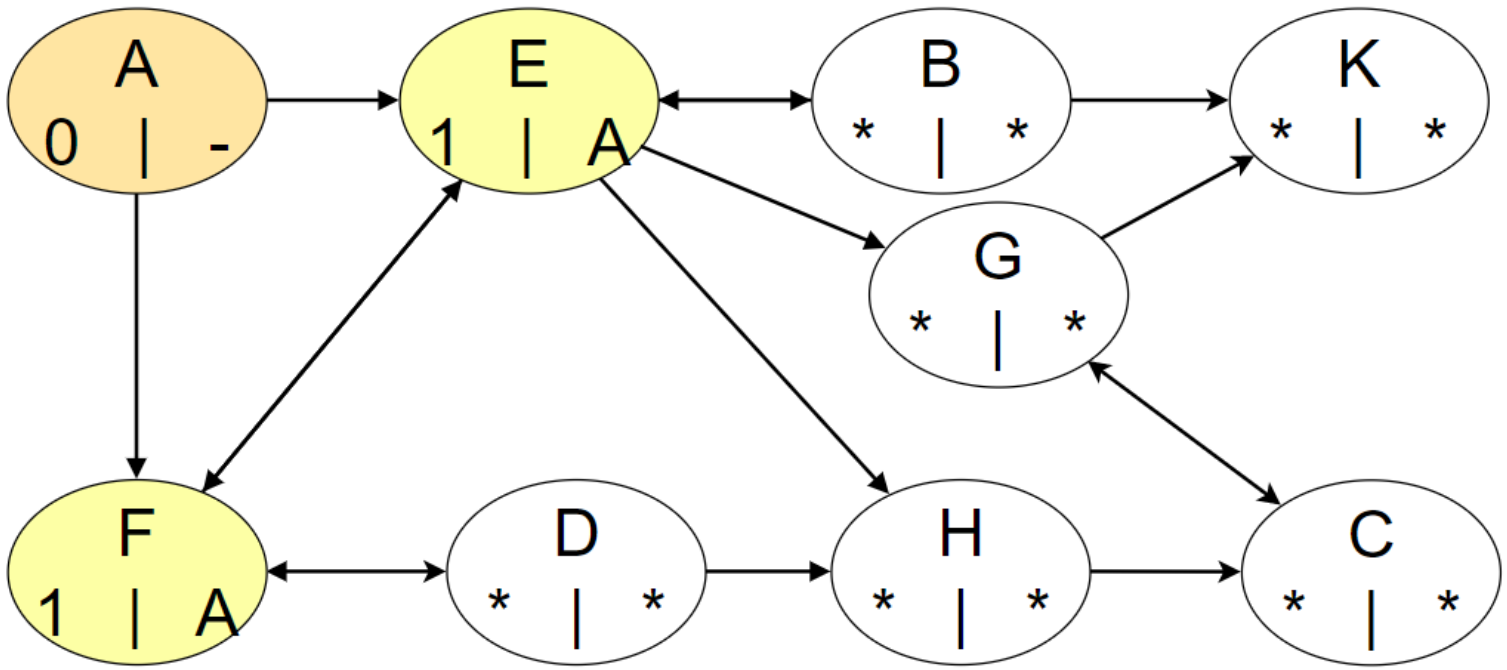


for each unvisited neighbor:
set the path length to the length of frontVertex +1
set the predecessor to the frontVertext
mark the neighbor as visited and enqueue



frontVertex: E

dequeue frontVertex from the vertexQueue
for all of frontVertex's unvisited neighbors
set the length of the path
set the predecessor of the neighbor
enqueue onto the vertexQueue



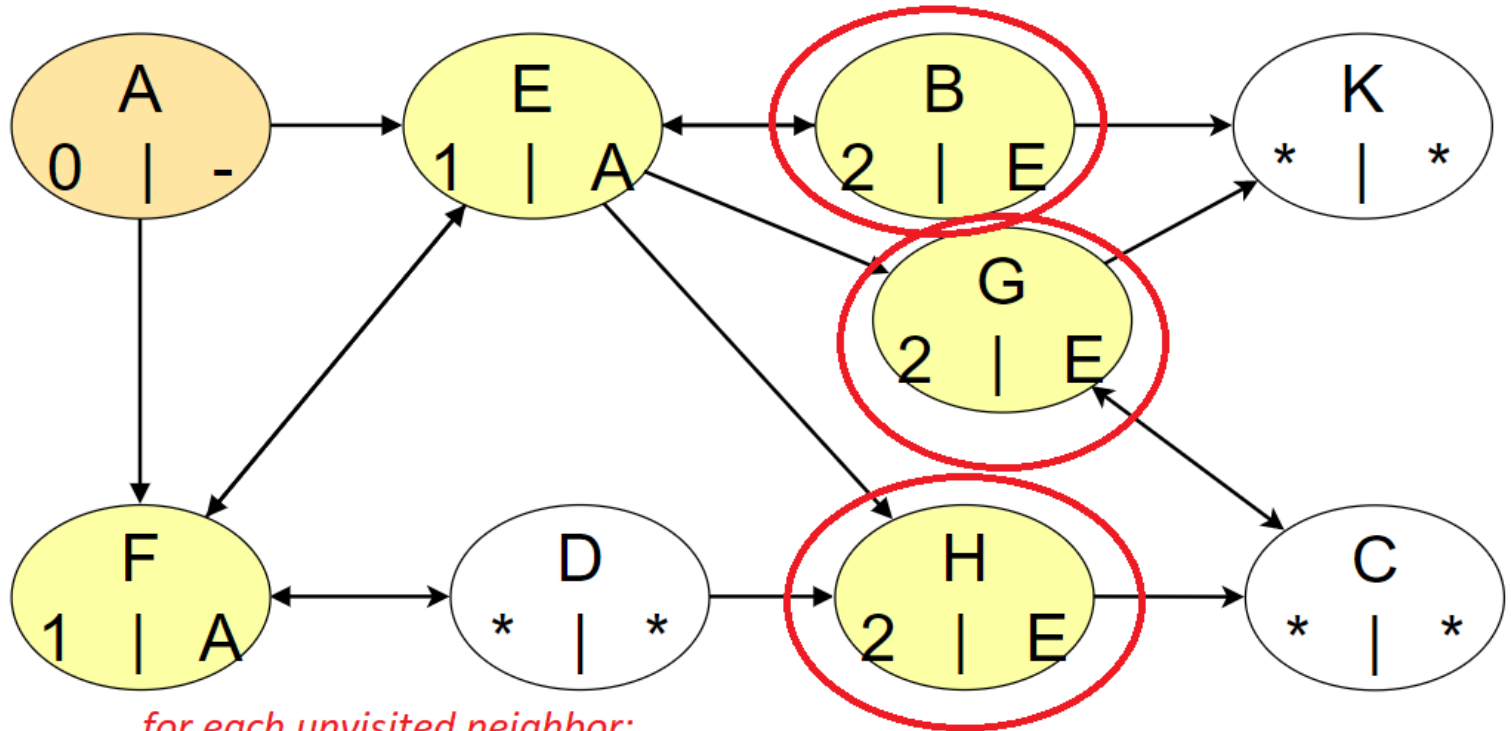
dequeue to get the new frontVertext

FBGH

front Vertex Queue back

frontVertex: E

dequeue frontVertex from the vertexQueue
for all of frontVertex's unvisited neighbors
set the length of the path
set the predecessor of the neighbor
enqueue onto the vertexQueue



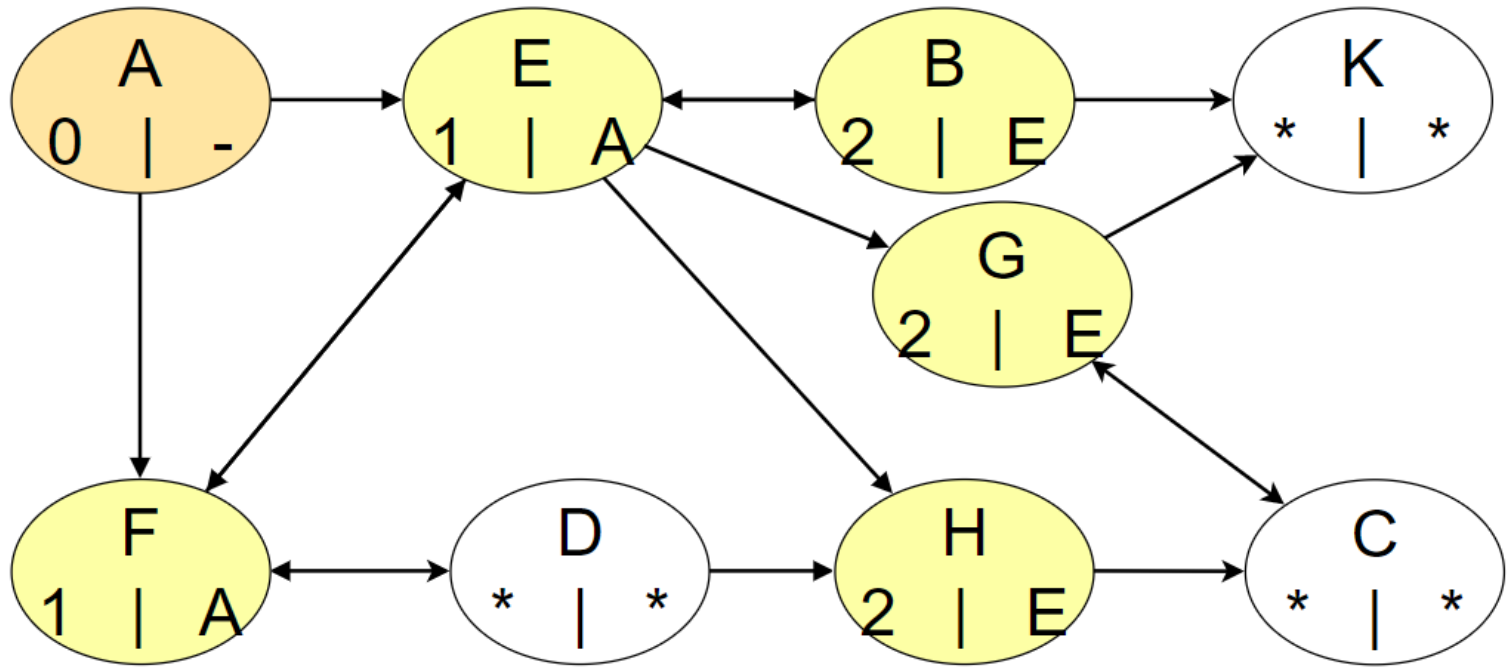
for each unvisited neighbor:
set the path length to the length of frontVertex +1
set the predecessor to the frontVertext
mark the neighbor as visited and enqueue

BGH

front Vertex Queue back

frontVertex: F

dequeue frontVertex from the vertexQueue
for all of frontVertex's unvisited neighbors
set the length of the path
set the predecessor of the neighbor
enqueue onto the vertexQueue



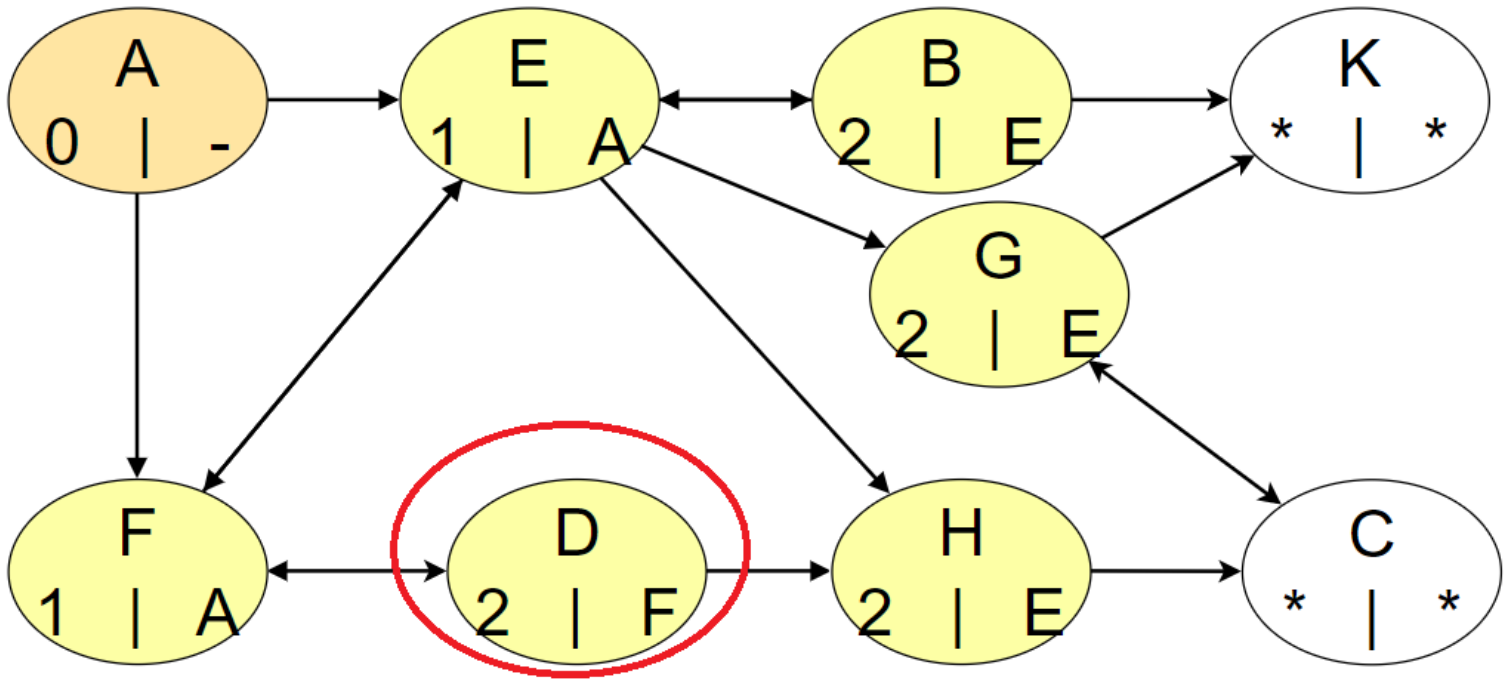
dequeue to get the new frontVertext

BGHD

front Vertex Queue back

frontVertex: F

dequeue frontVertex from the vertexQueue
for all of frontVertex's unvisited neighbors
set the length of the path
set the predecessor of the neighbor
enqueue onto the vertexQueue



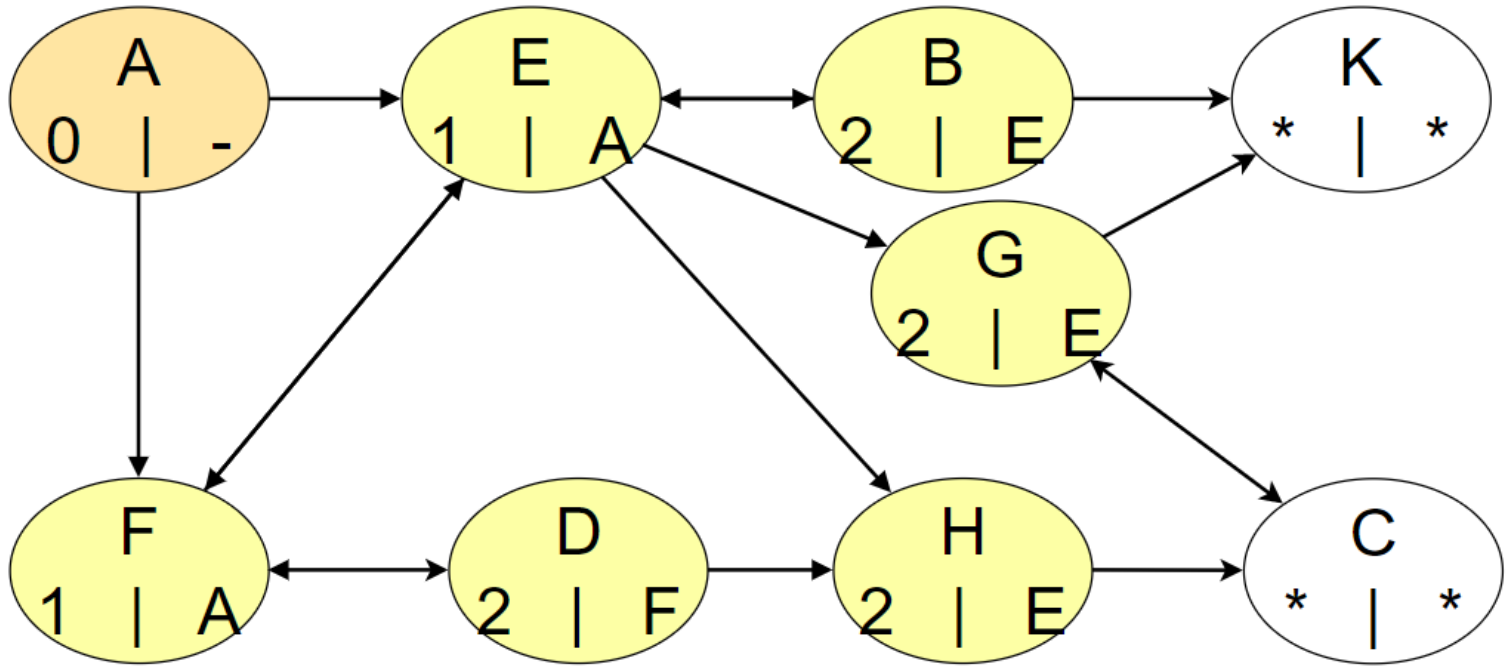
for each unvisited neighbor:
set the path length to the length of frontVertex +1
set the predecessor to the frontVertext
mark the neighbor as visited and enqueue

GHD

front Vertex Queue back

frontVertex: B

dequeue frontVertex from the vertexQueue
for all of frontVertex's unvisited neighbors
set the length of the path
set the predecessor of the neighbor
enqueue onto the vertexQueue



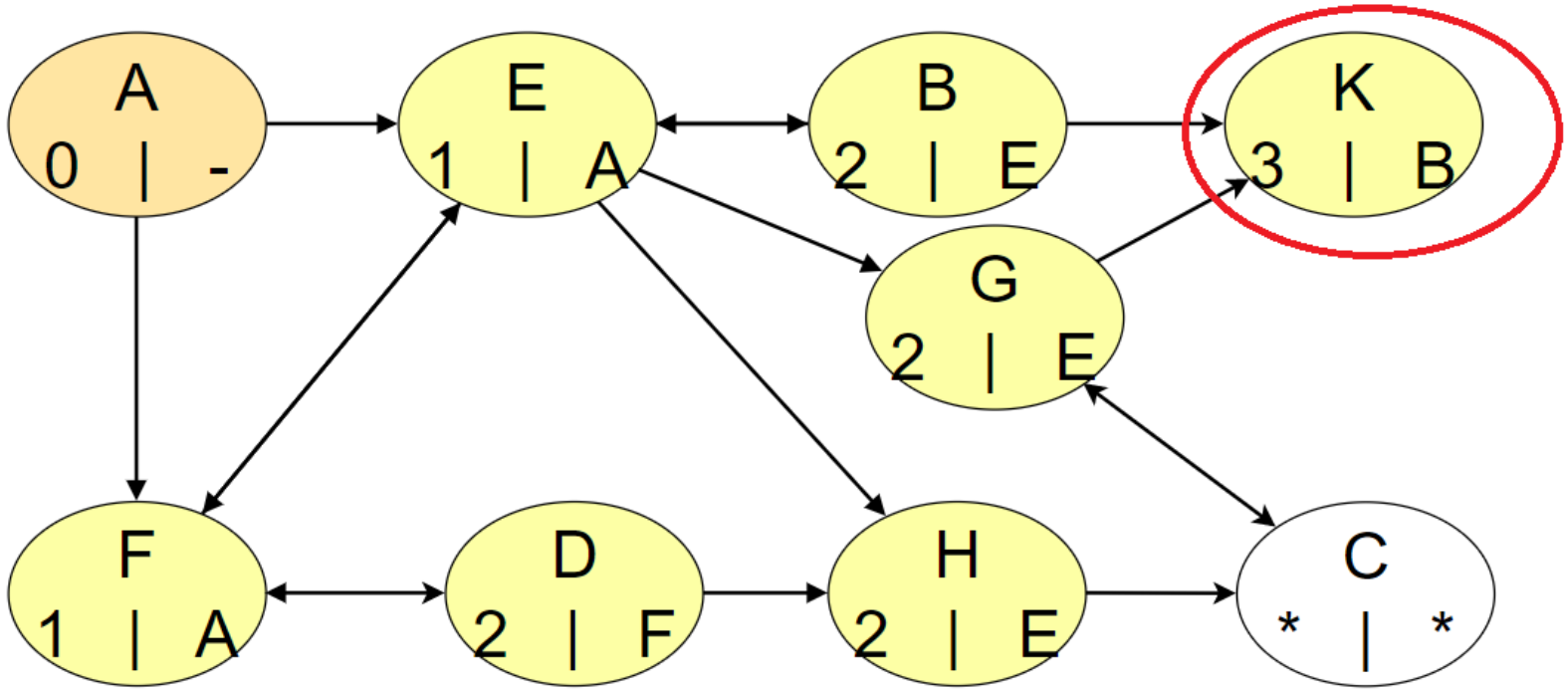
dequeue to get the new frontVertext

GHDK

front Vertex Queue back

frontVertex: B

dequeue frontVertex from the vertexQueue
for all of frontVertex's unvisited neighbors
set the length of the path
set the predecessor of the neighbor
enqueue onto the vertexQueue



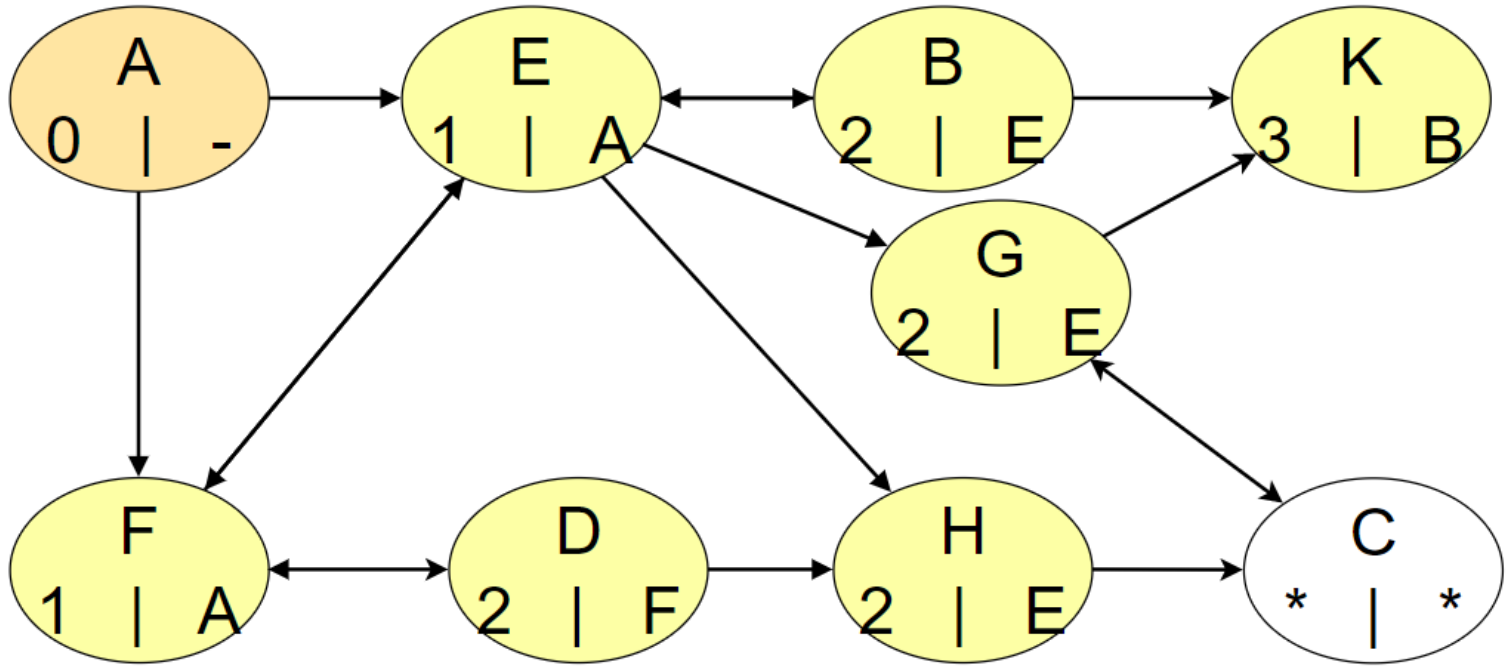
for each unvisited neighbor:
set the path length to the length of frontVertex +1
set the predecessor to the frontVertext
mark the neighbor as visited and enqueue

HDK

front Vertex Queue back

frontVertex: G

dequeue frontVertex from the vertexQueue
for all of frontVertex's unvisited neighbors
set the length of the path
set the predecessor of the neighbor
enqueue onto the vertexQueue



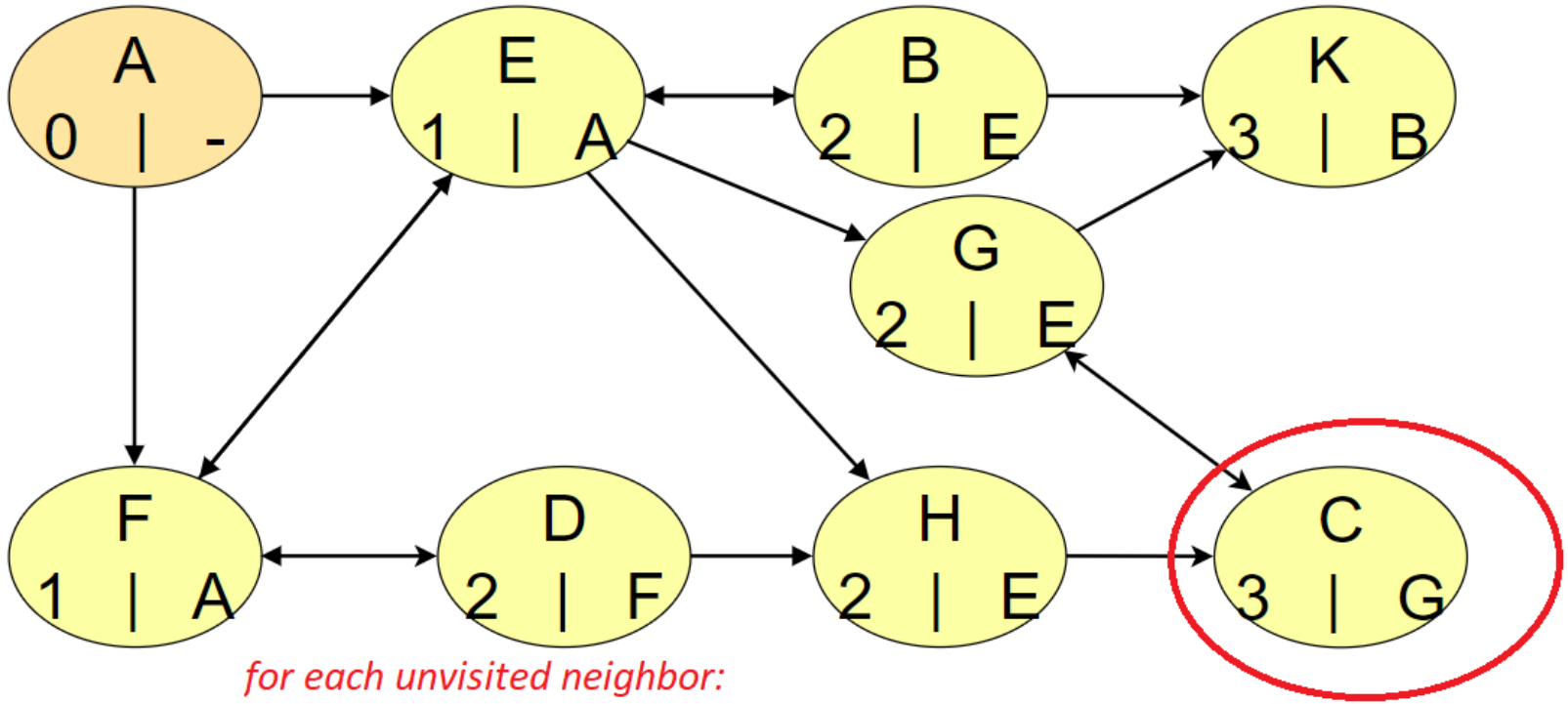
dequeue to get the new frontVertext

HDKC

front Vertex Queue back

frontVertex: G

dequeue frontVertex from the vertexQueue
for all of frontVertex's unvisited neighbors
set the length of the path
set the predecessor of the neighbor
enqueue onto the vertexQueue



for each unvisited neighbor:
set the path length to the length of frontVertex +1
set the predecessor to the frontVertext
mark the neighbor as visited and enqueue

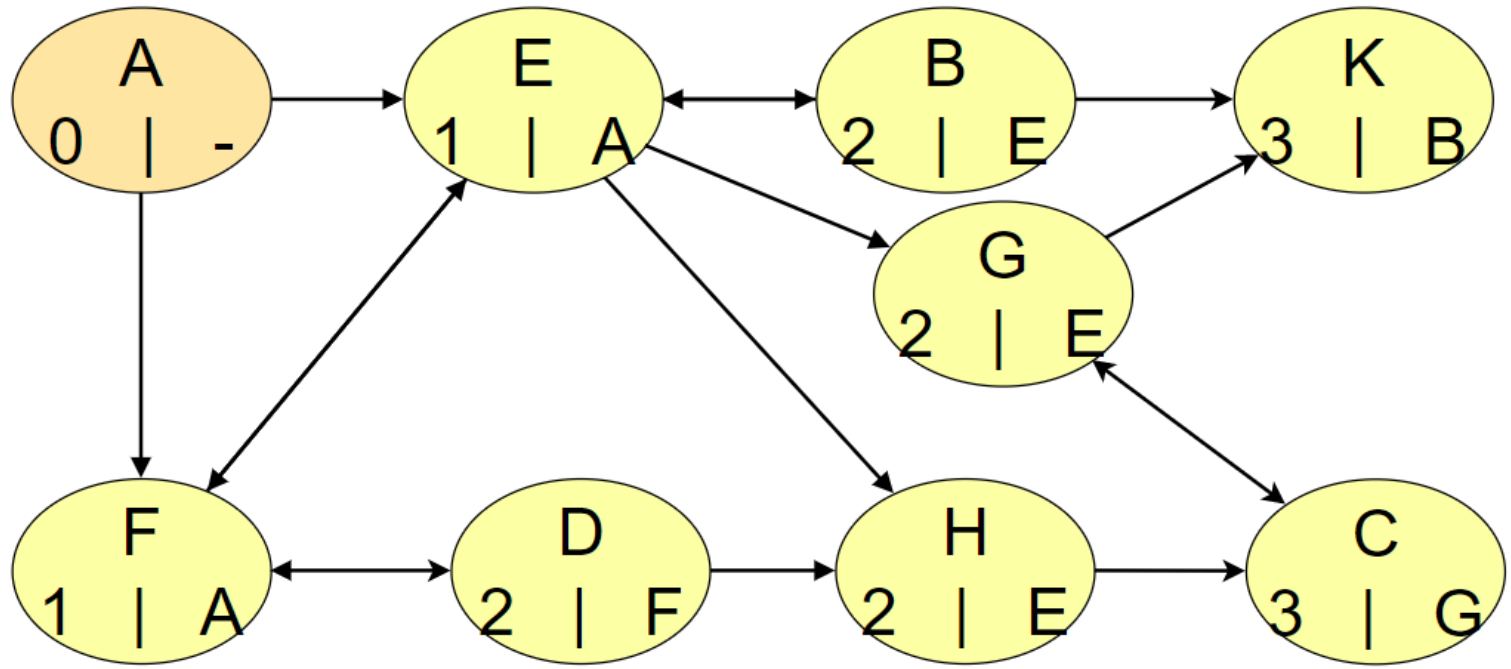
HDKC

front Vertex Queue back

frontVertex: G

dequeue frontVertex from the vertexQueue
for all of frontVertex's unvisited neighbors
set the length of the path
set the predecessor of the neighbor
enqueue onto the vertexQueue

C is the destination so we exit the loop and are done!
C is now the endVertex that we use to build the stack



HDKC

front Vertex Queue back

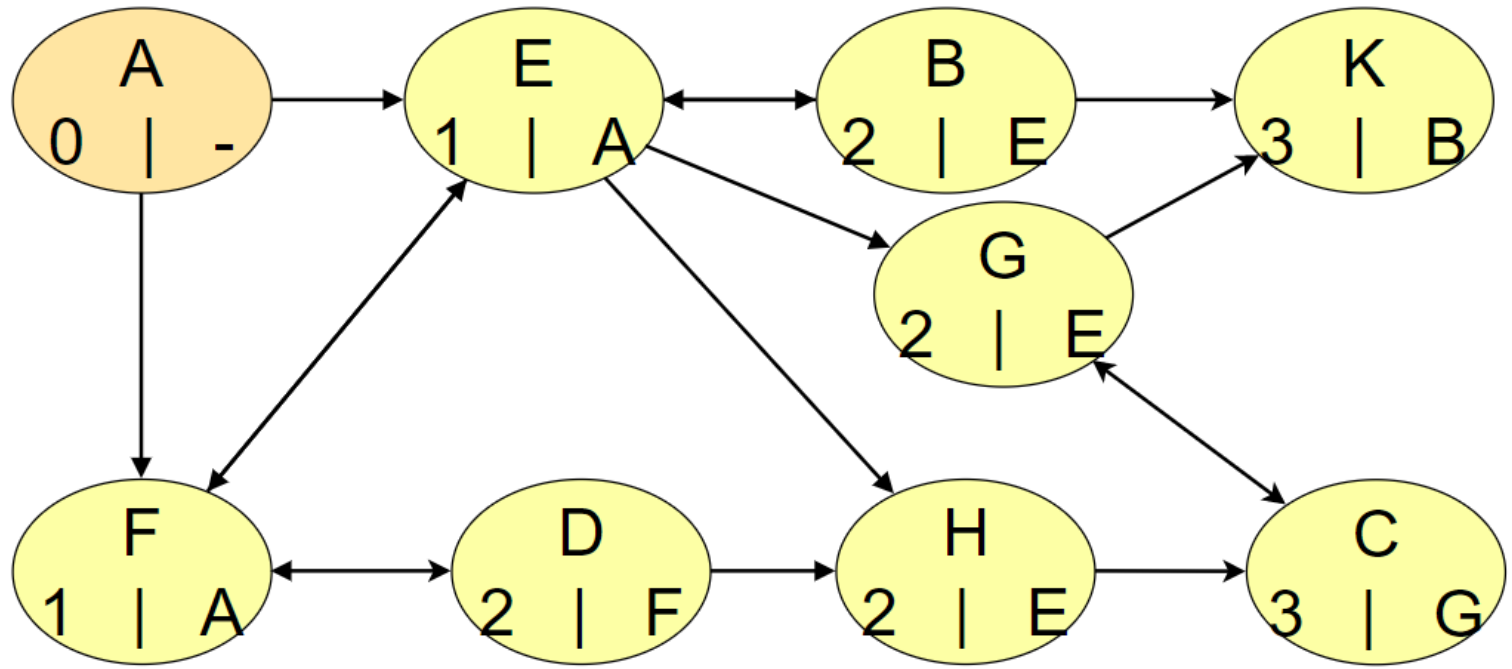
frontVertex: G

Shortest Path Stack

C

dequeue frontVertex from the vertexQueue
for all of frontVertex's unvisited neighbors
set the length of the path
set the predecessor of the neighbor
enqueue onto the vertexQueue

push the endVertex on the stack



HDKC

front Vertex Queue back

frontVertex: G

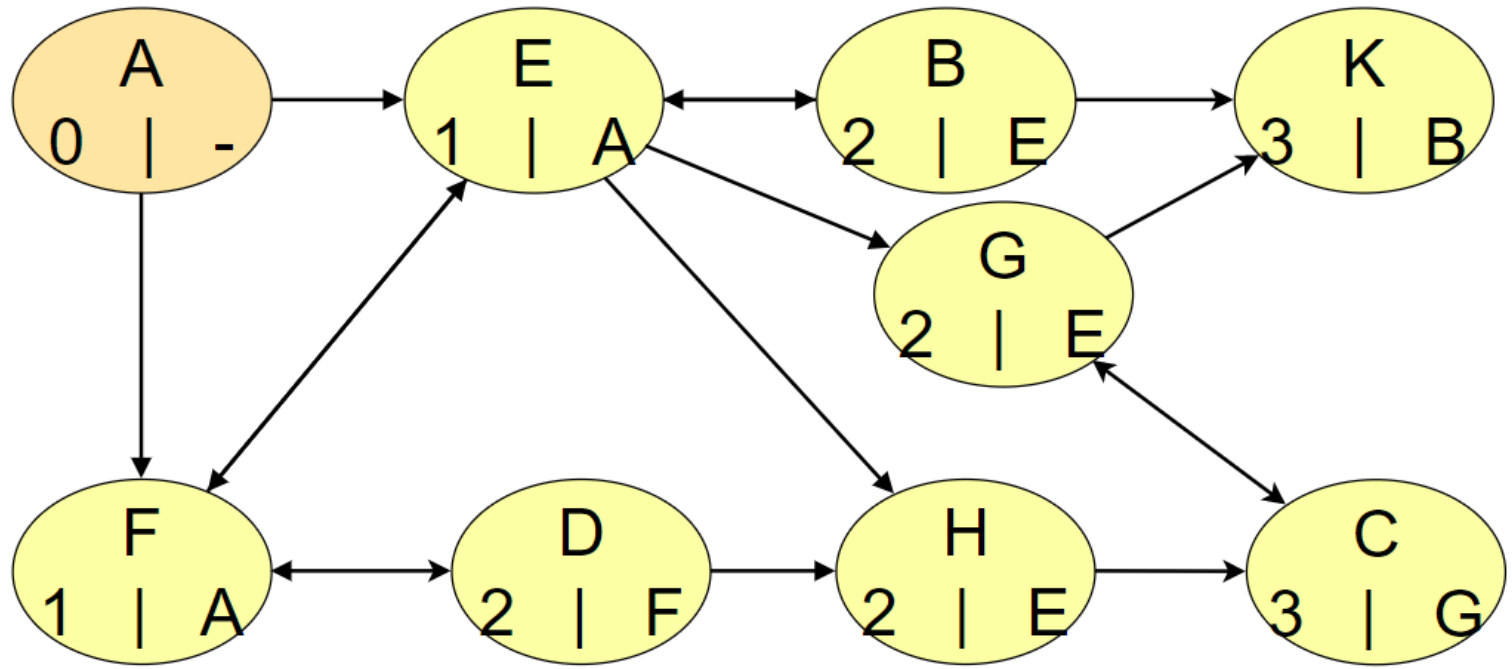
Shortest Path Stack

G

C

dequeue frontVertex from the vertexQueue
for all of frontVertex's unvisited neighbors
set the length of the path
set the predecessor of the neighbor
enqueue onto the vertexQueue

peek the stack; put that vertex's predecessor on the stack



HDKC

front Vertex Queue back

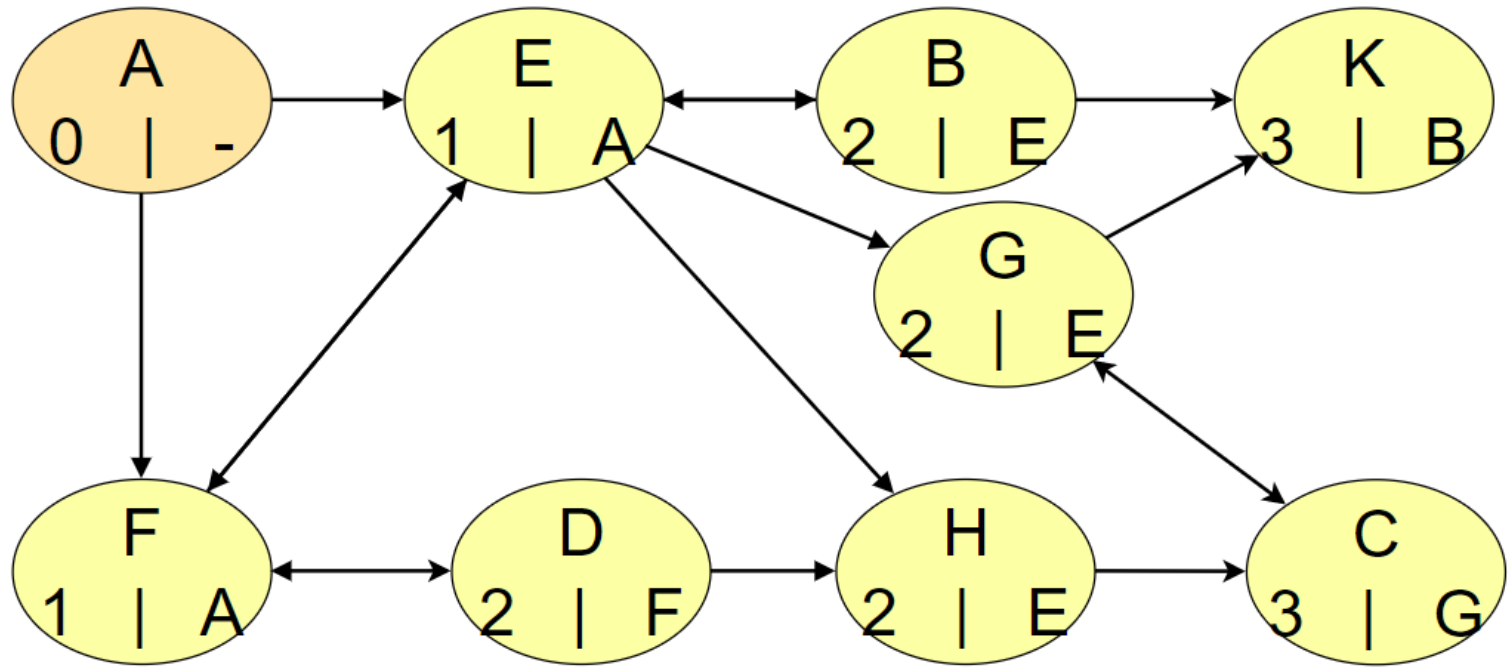
frontVertex: G

Shortest Path Stack

E
G
C

dequeue frontVertex from the vertexQueue
for all of frontVertex's unvisited neighbors
set the length of the path
set the predecessor of the neighbor
enqueue onto the vertexQueue

peek the stack; put that vertex's predecessor on the stack



HDKC

front Vertex Queue back

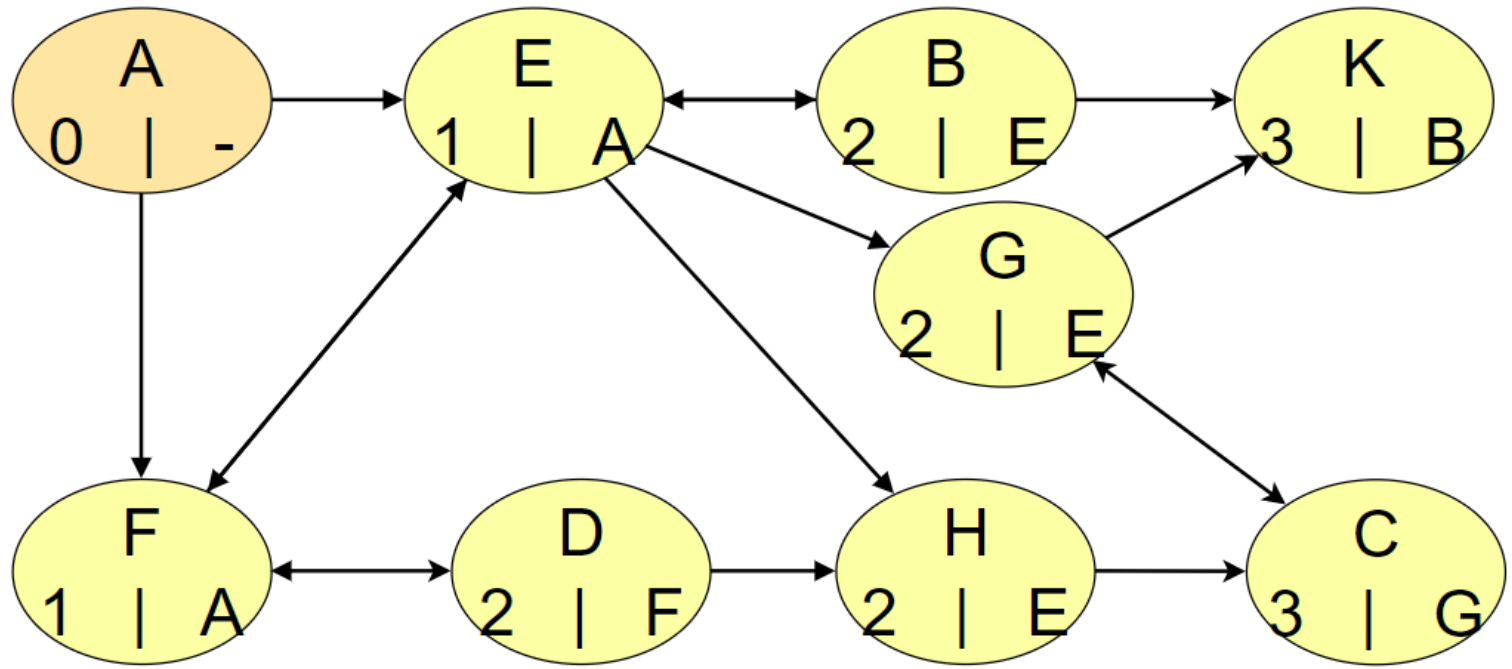
frontVertex: G

Shortest Path Stack

A
E
G
C

dequeue frontVertex from the vertexQueue
for all of frontVertex's unvisited neighbors
set the length of the path
set the predecessor of the neighbor
enqueue onto the vertexQueue

peek the stack; put that vertex's predecessor on the stack



HDKC

front Vertex Queue back

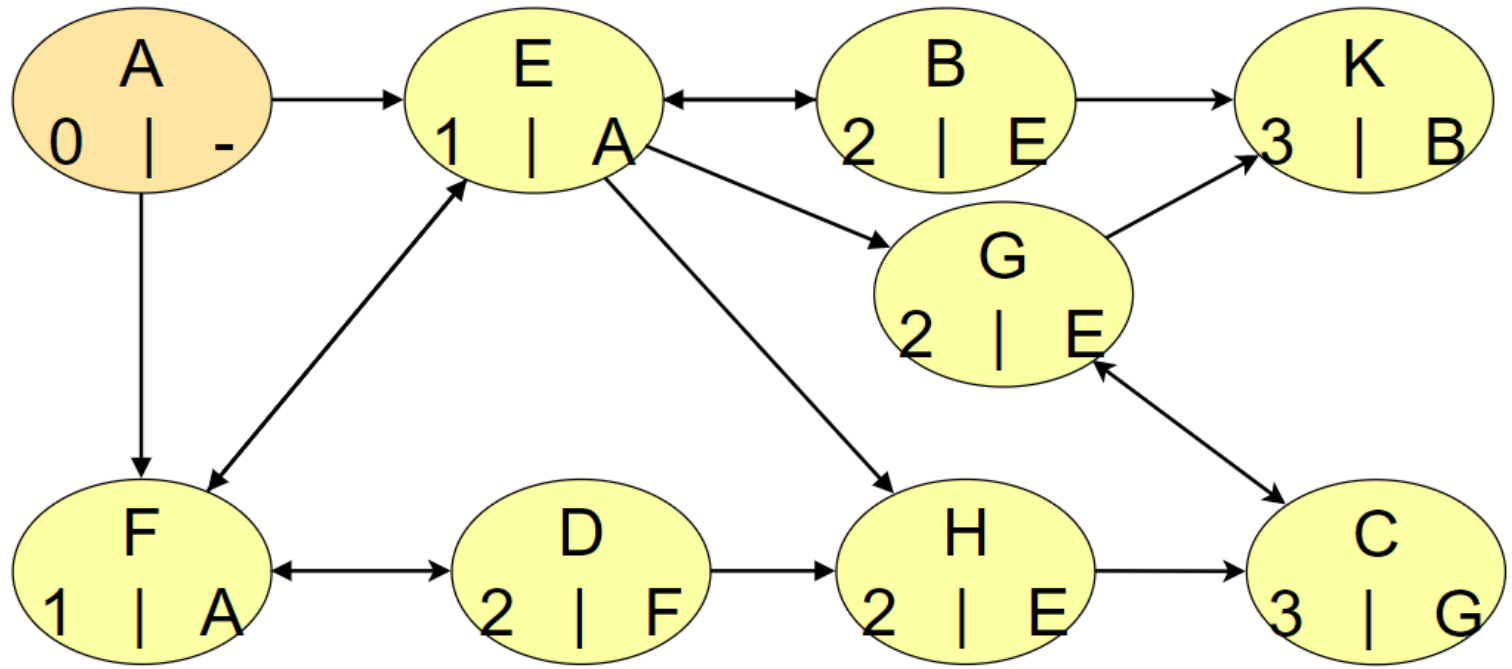
frontVertex: G

Shortest Path Stack

A
E
G
C

dequeue frontVertex from the vertexQueue
for all of frontVertex's unvisited neighbors
set the length of the path
set the predecessor of the neighbor
enqueue onto the vertexQueue

the stack contains the shortest path



Shortest Path in a Weighted Graph

- The algorithm is similar but takes edge weights into account and uses a priority queue.
- The shortest path is not necessarily the one with the fewest edges.
- The shortest path is the one with the **smallest sum of edge-weights**.

- The shortest path in a weight graph is often called the ***cheapest path***.
- This algorithm also uses a modified breadth first search.

Finding Cheapest Path

- In breadth-first, each vertex is placed into a queue as it is visited.
- To modify to find the cheapest path, we need more information.
  - keep track of the predecessor vertex (where we came from)
  - keep track of the cost of the path to that vertex
- We'll create a new object to keep track of this information: VertexCPData
  - holds a vertex, cost of the path to that vertex, and a predecessor vertex
- We use a priority queue of VertexCPData objects.
  - The priority is the cost of the path.
  - Smallest cost has highest priority.
  - When a new object is added, it is put in order based on cost.
  - If there are other objects with the same cost, it is put at the back of that section.

Cheapest Path Algorithm

enqueue a VertexCPData with [origin, 0, null] onto the priorityQueue
while we haven't reached the endVertex and the priorityQueue is not empty
    dequeue vertex data from the priorityQueue (data is for frontVertex)
    if(frontVertex is not visited)
        mark as visited
        set the cost of the path and predecessor based on the data
        if(frontVertex is the endVertex)
            we are done
        else
            for all of frontVertex's neighbors
                if the neighbor is not visited, put vertex data on the priority queue (order based on cost)
                    cost = cost of path to frontVertex + weight of edge to that neighbor
                    predecessor = frontVertex

Cheapest Path Algorithm

to find the path:

cost = cost of path to endVertex

push the endVertex onto a stack

while the top of the stack has a predecessor

    push predecessor onto stack

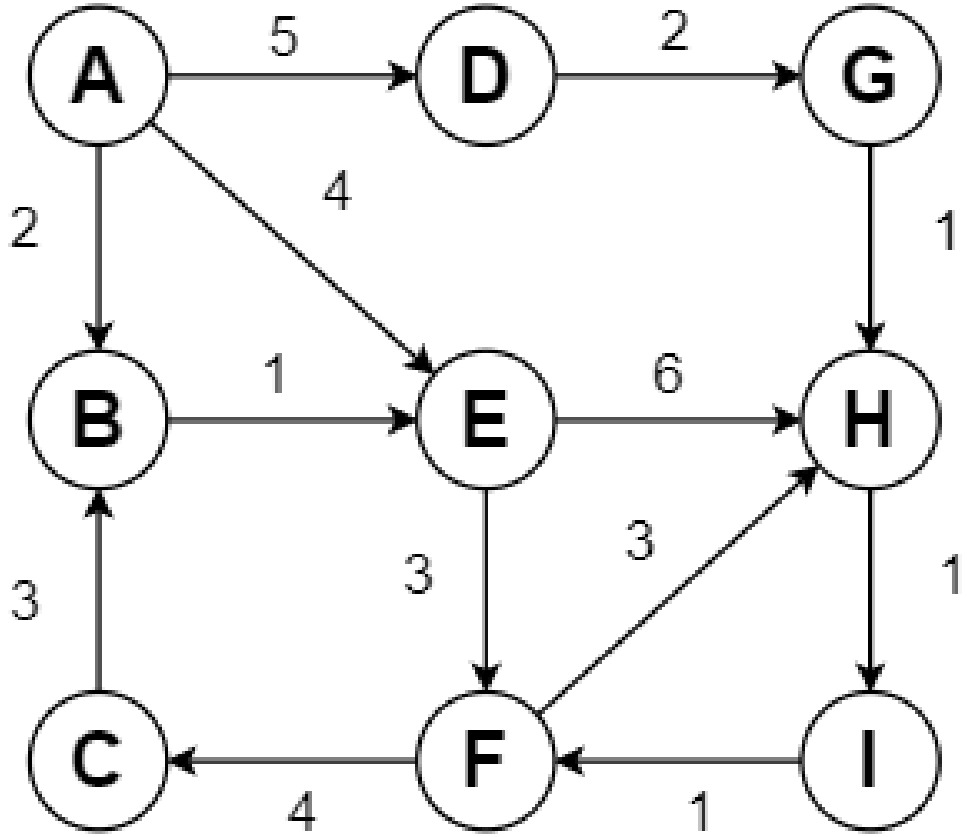
the contents of the stack are the path

Cheapest Path Algorithm

- This algorithm is based on Dijkstra's algorithm.
- Dijkstra's algorithm finds the shortest paths from an origin to all other vertices, which creates a *shortest path tree*.

Cheapest Path Example

- Find the cheapest path from A to H.



DEPTH-FIRST TRAVERSAL

Depth-First Traversal

- Start at a vertex and descend as deep as possible and then back up one level and follow another path as deep as possible
- Preorder, postorder, and inorder tree traversals are all examples of depth-first traversals

- Remember: origin matters!
- Remember: order of visitation is not part of the algorithm.

Depth-First Traversal

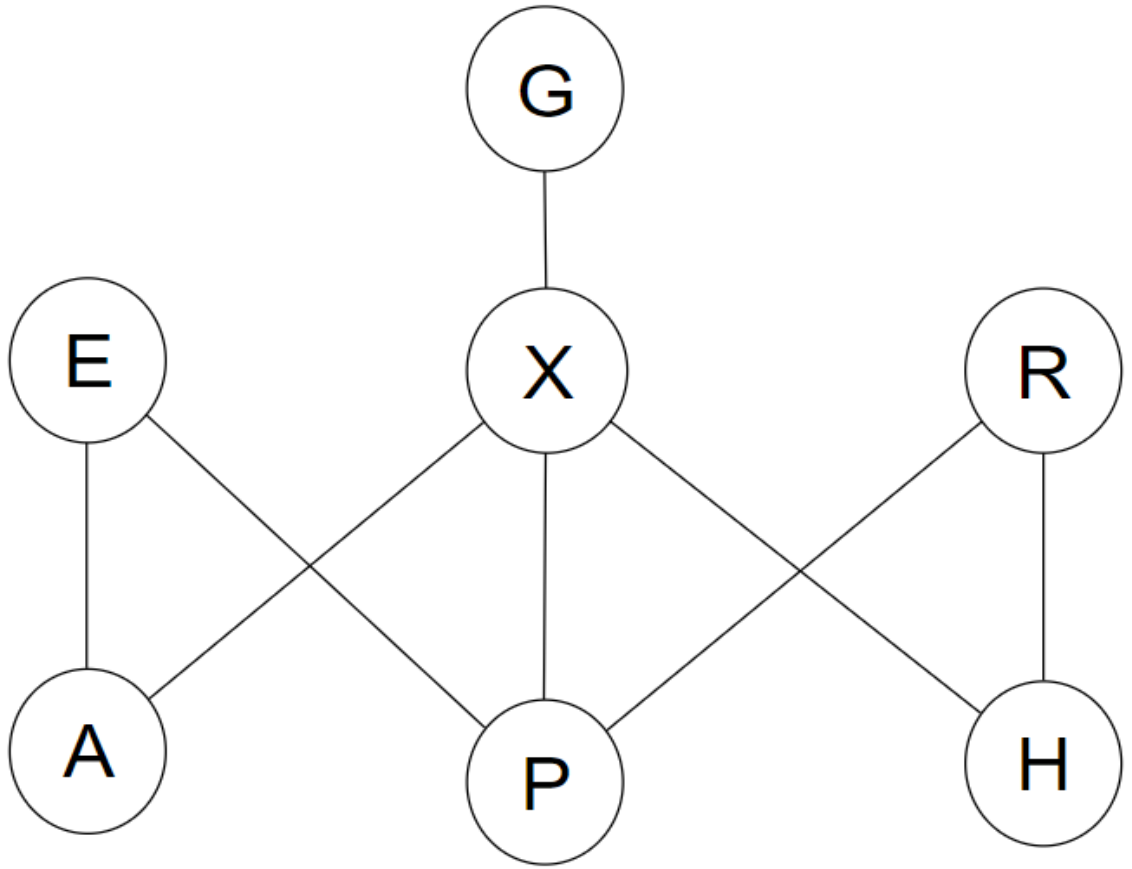
- Typically implemented with a stack
  - Has a "recursive feel"
- General approach:
  - visit a vertex
  - put all of the vertex's neighbors on the stack
  - repeat

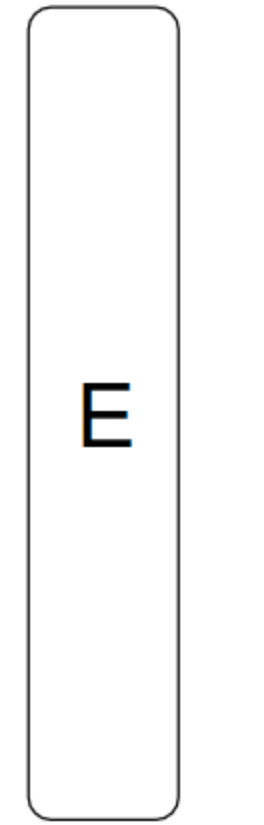
Depth-First Traversal Algorithm

enqueue the origin onto the traversalOrderQueue
mark the origin as visited
push the origin onto the vertexStack
while the vertexStack is not empty
    peek at a vertex
    if the vertex has unvisited neighbors
        mark one of the unvisited neighbors as visited
        enqueue the vertex onto the traversalOrderQueue
        push the vertex onto the vertexStack
    else pop the vertex

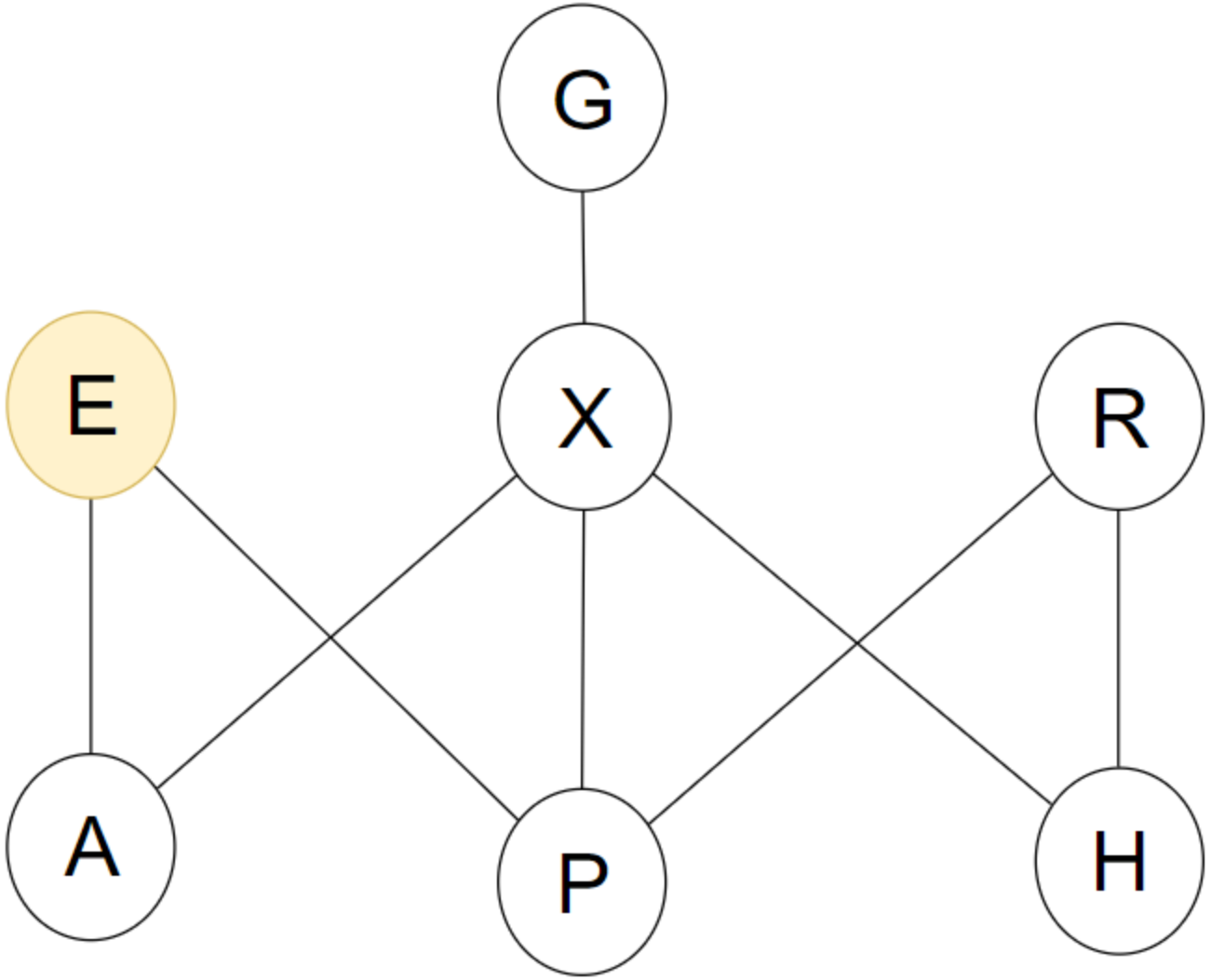
Depth-First Example

- Trace a depth-first traversal starting at vertex E.





Vertex Stack

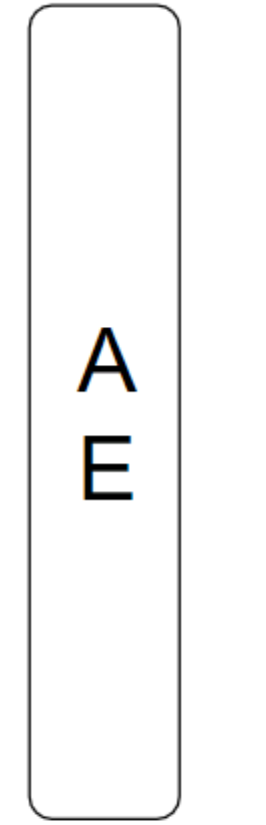


front

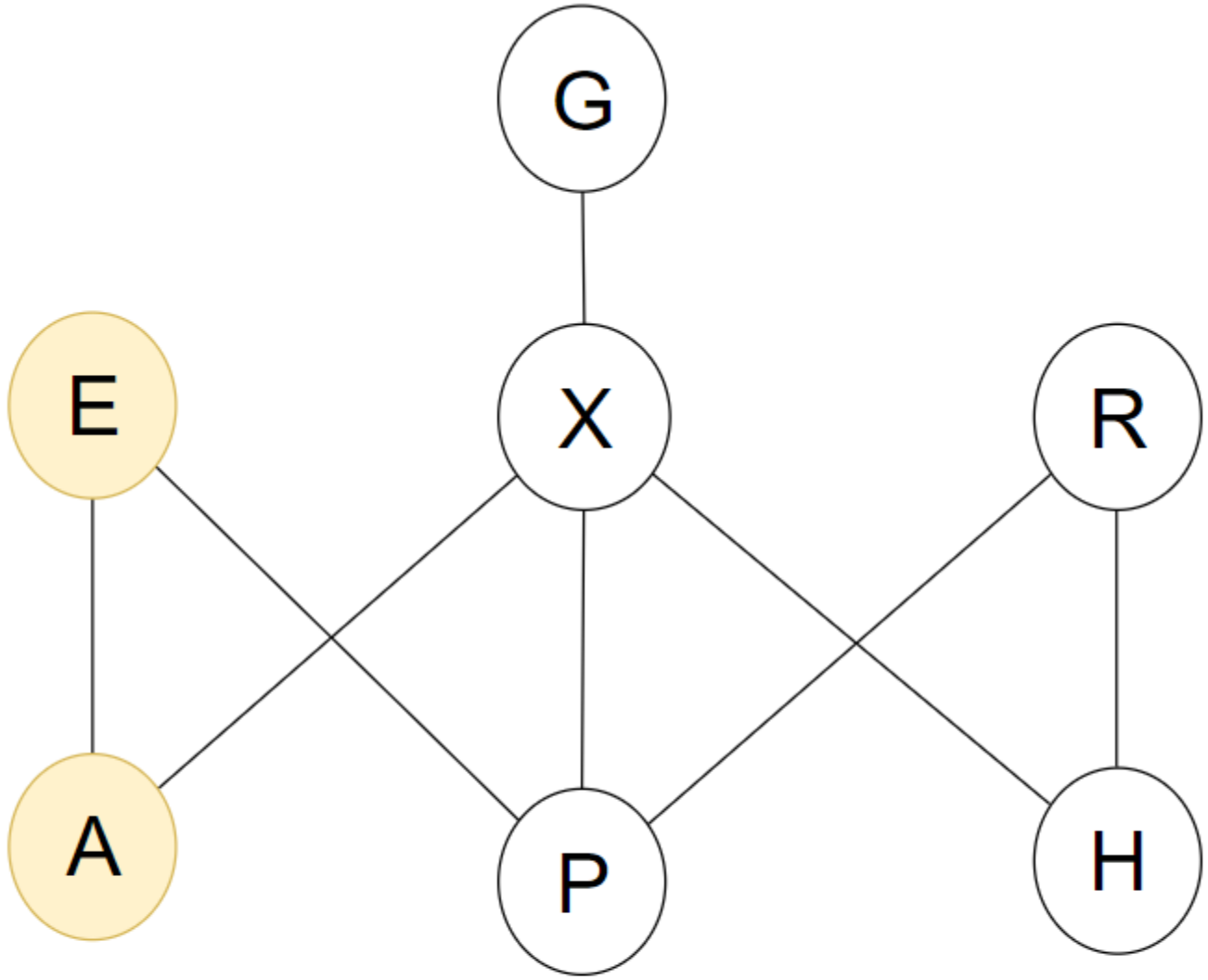
Traversal Order Queue

back

mark the origin as visited
enqueue the origin
push the origin on the stack



Vertex Stack

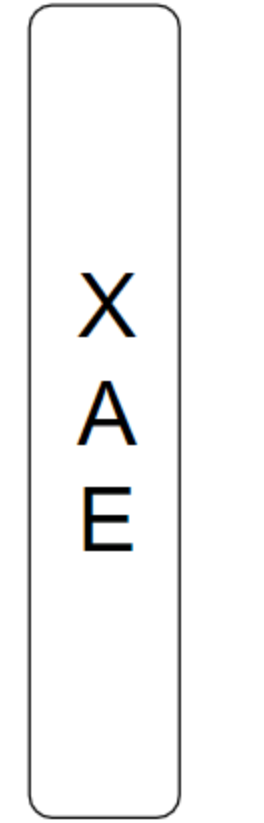


front

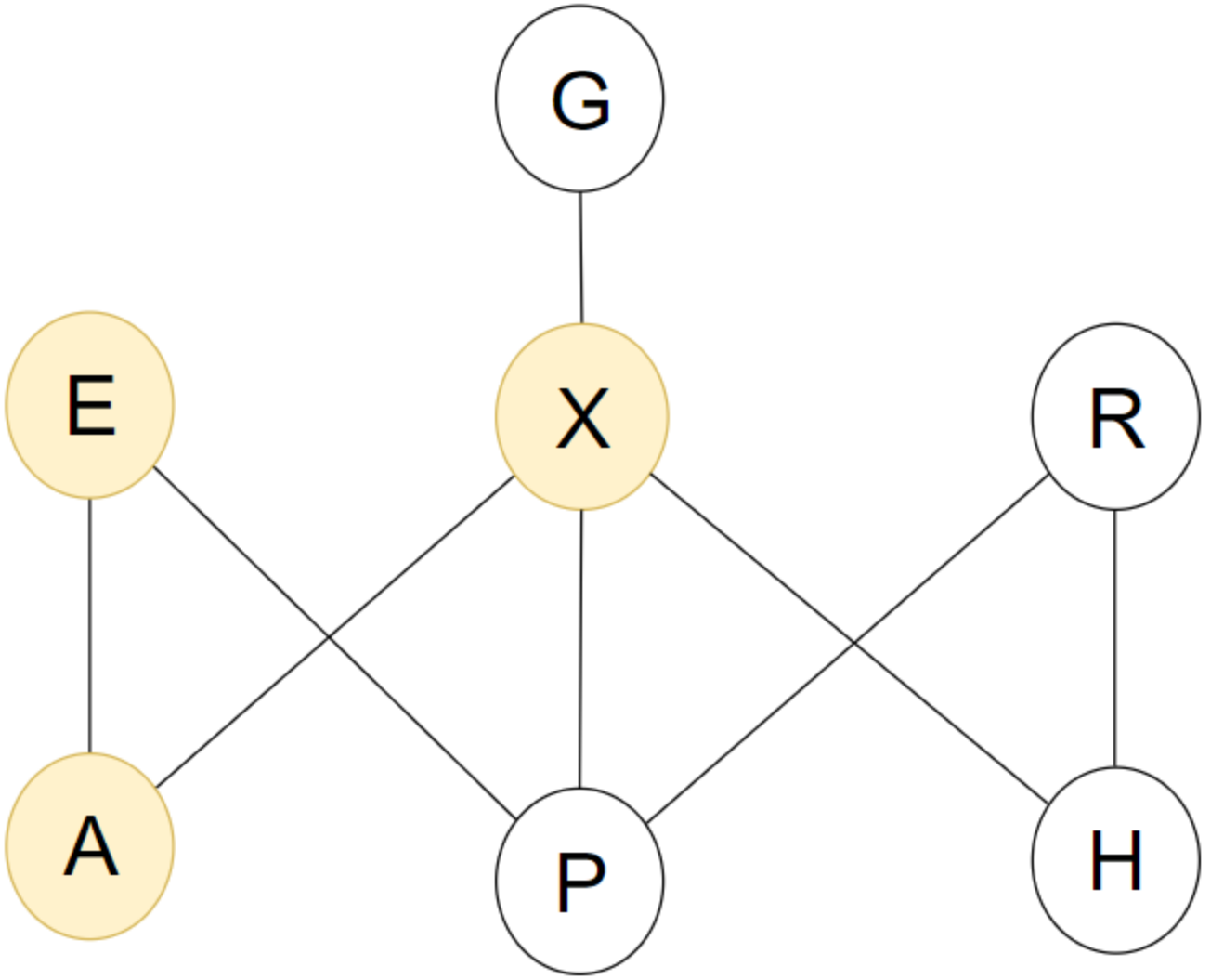
Traversal Order Queue

back

peek at the stack to see a vertex (E)
mark one of that vertex's unvisited neighbors as visited (A)
put that vertex (A) on the traversal order queue
push that vertex (A) onto the stack



Vertex Stack

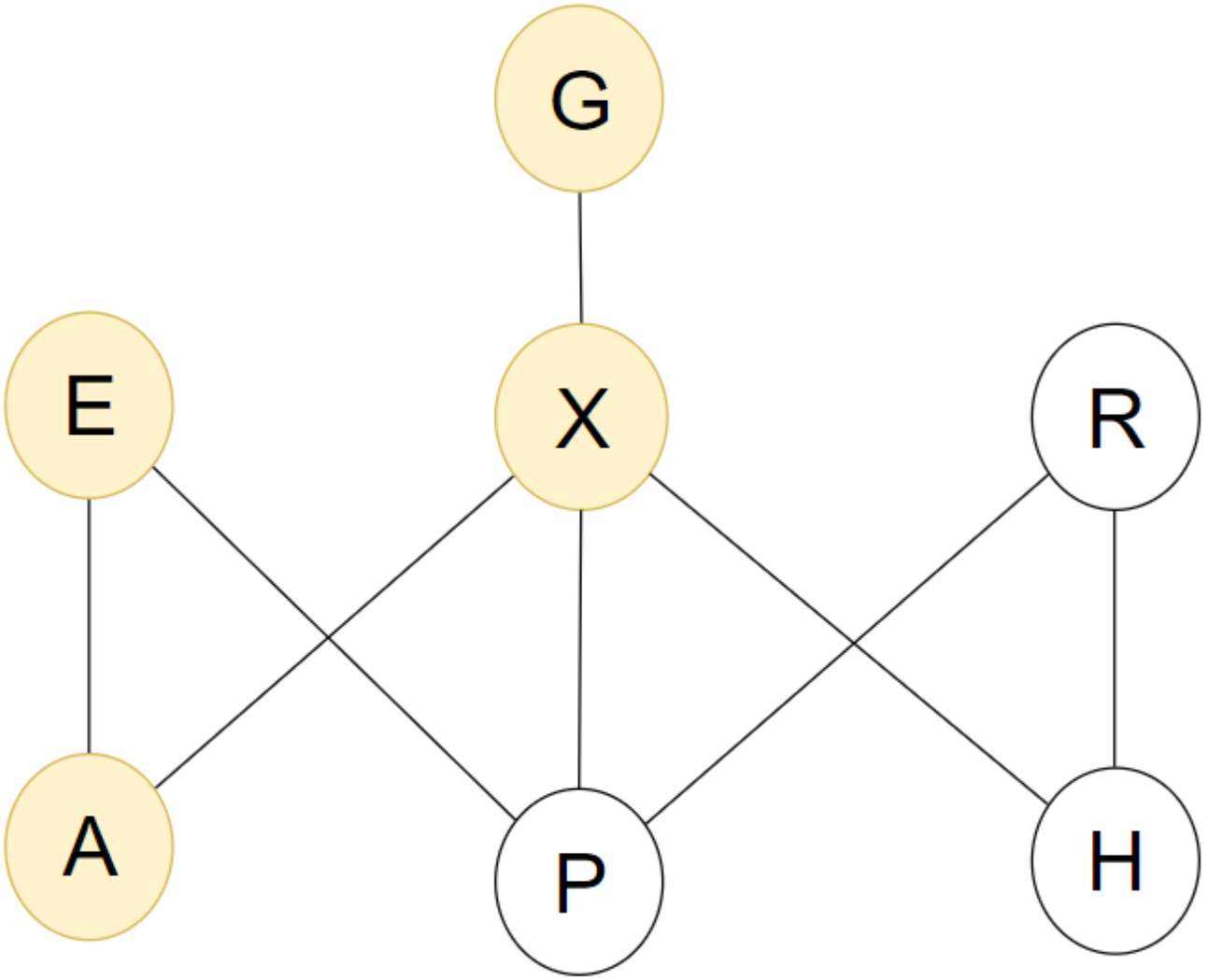


front

Traversal Order Queue

back

peek at the stack to see a vertex (A)
mark one of that vertex's unvisited neighbors as visited (X)
put that vertex (X) on the traversal order queue
push that vertex (X) onto the stack



G
X
A
E

Vertex Stack

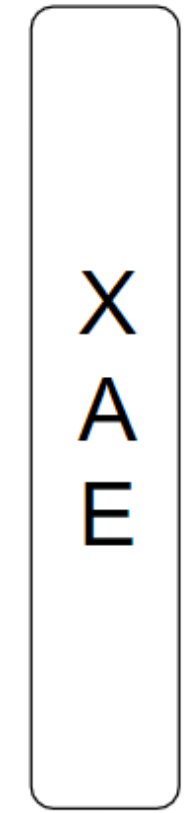
E A X G

front

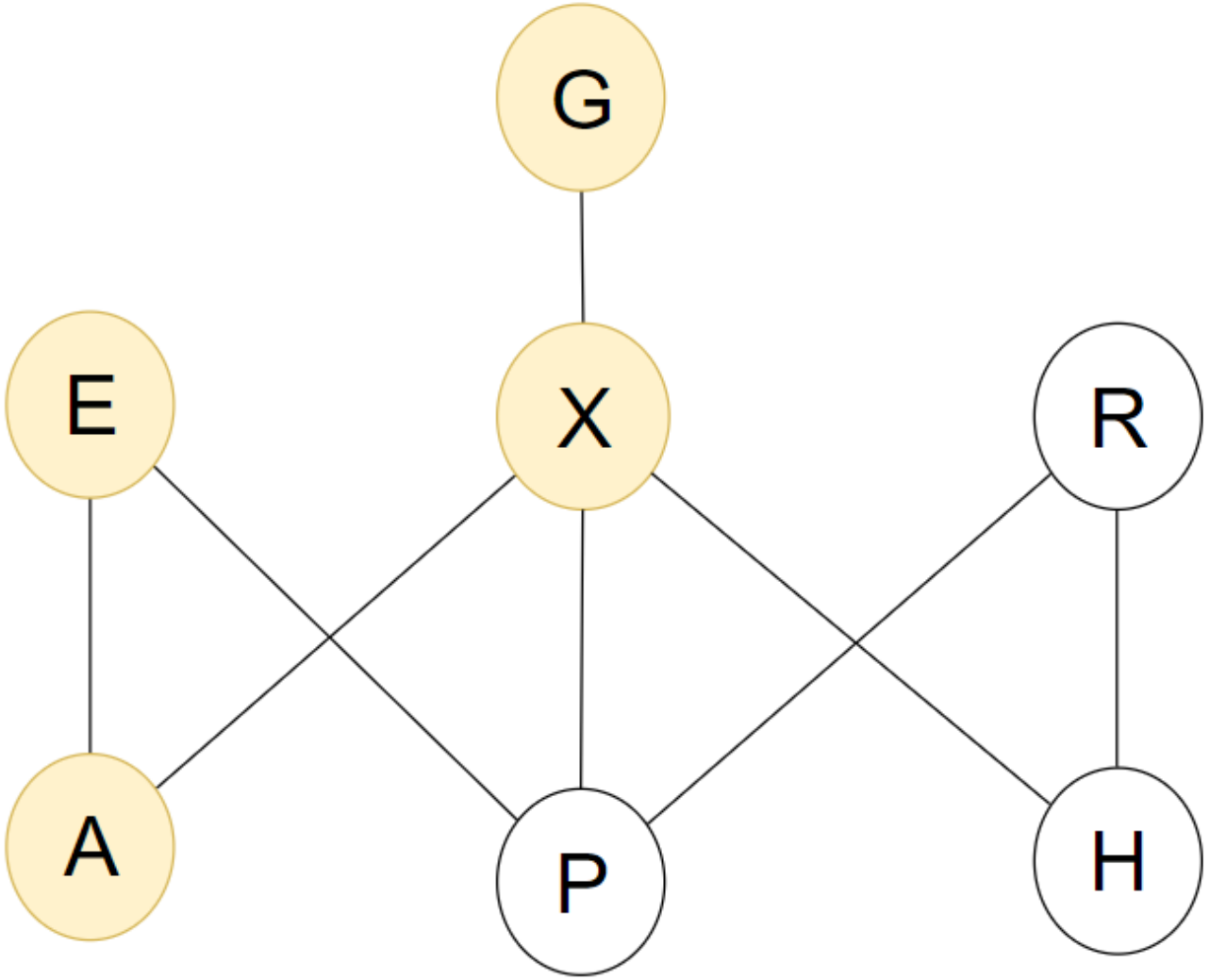
Traversal Order Queue

back

peek at the stack to see a vertex (X)
mark one of that vertex's unvisited neighbors as visited (G)
put that vertex (G) on the traversal order queue
push that vertex (G) onto the stack



Vertex Stack

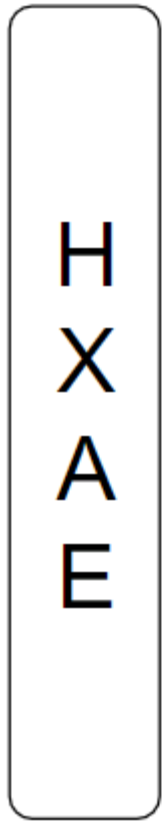


front

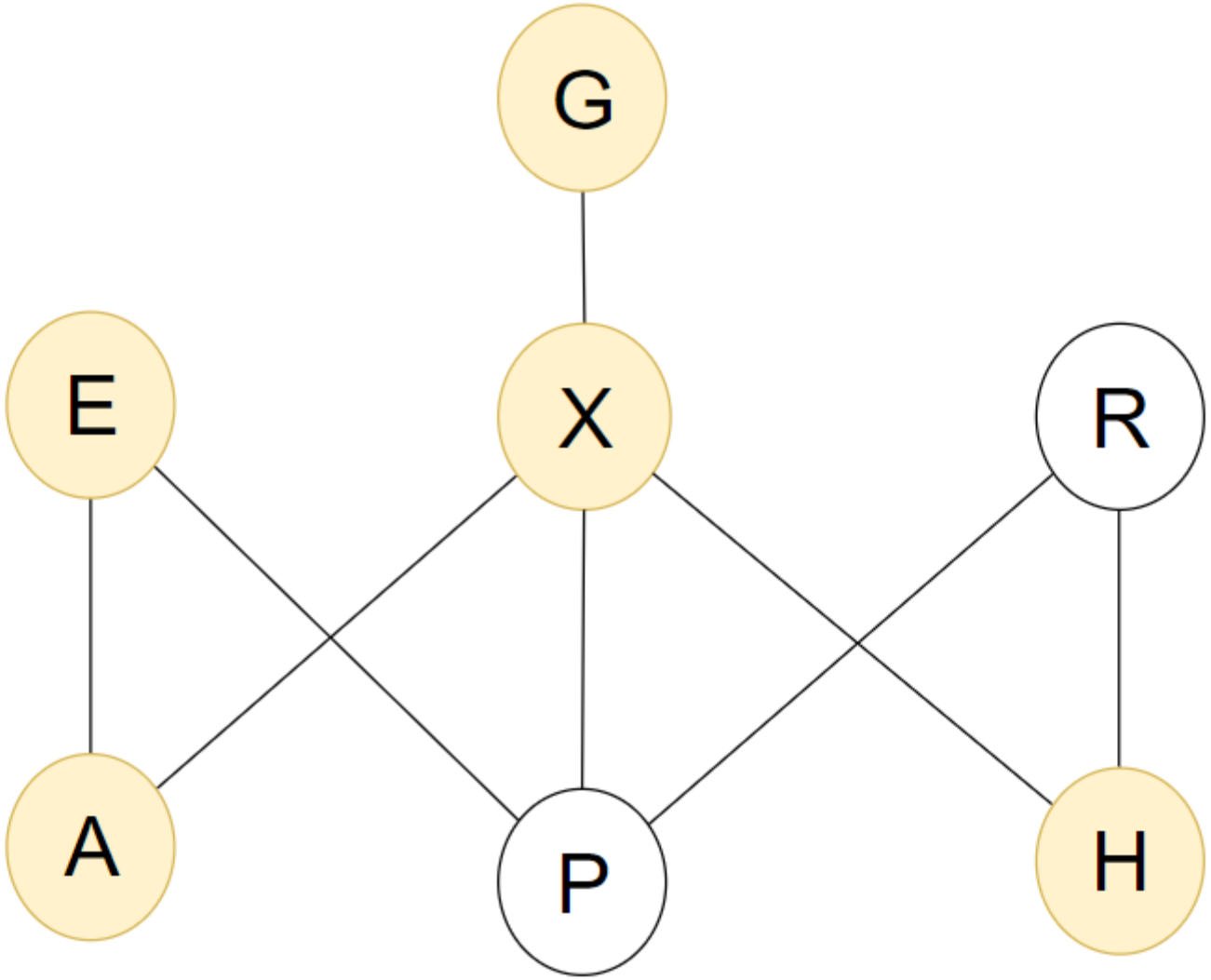
Traversal Order
Queue

back

peek at the stack to see a vertex (G)
that vertex has no unvisited neighbors, so pop from the stack

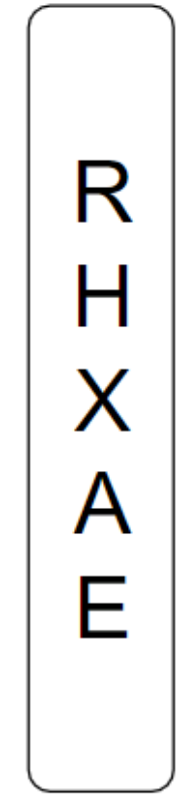


Vertex Stack

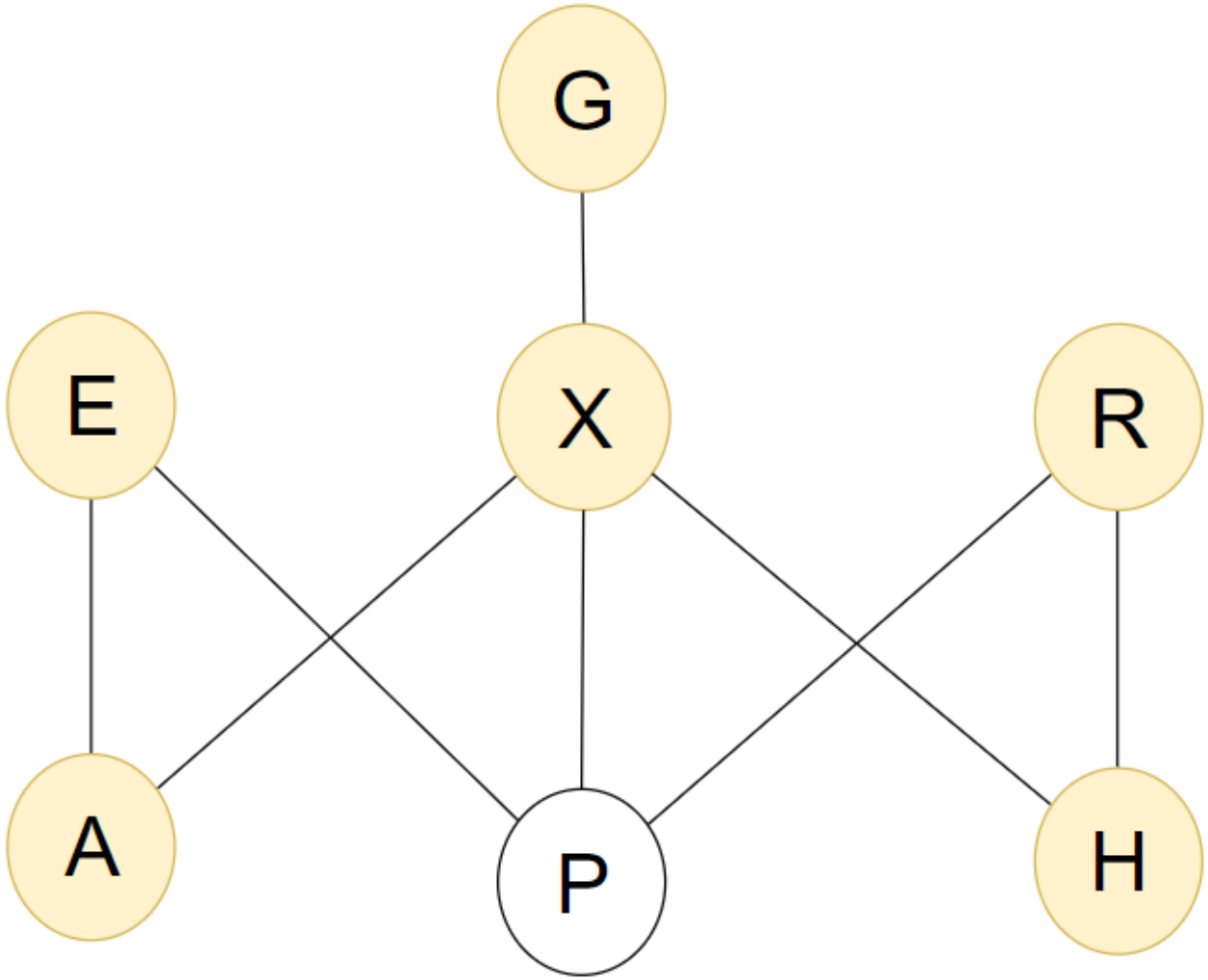


front Traversal Order Queue back

peek at the stack to see a vertex (X)
mark one of that vertex's unvisited neighbors as visited (H)
put that vertex (H) on the traversal order queue
push that vertex (H) onto the stack



Vertex Stack



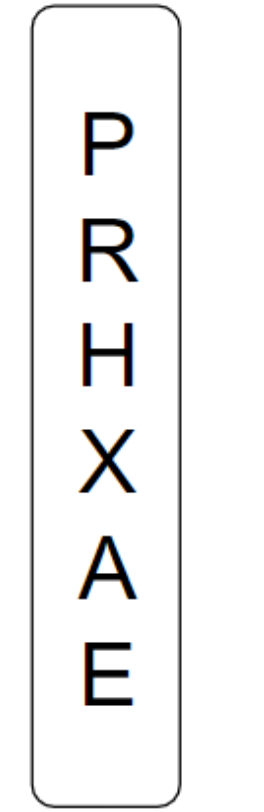
E A X G H R

front

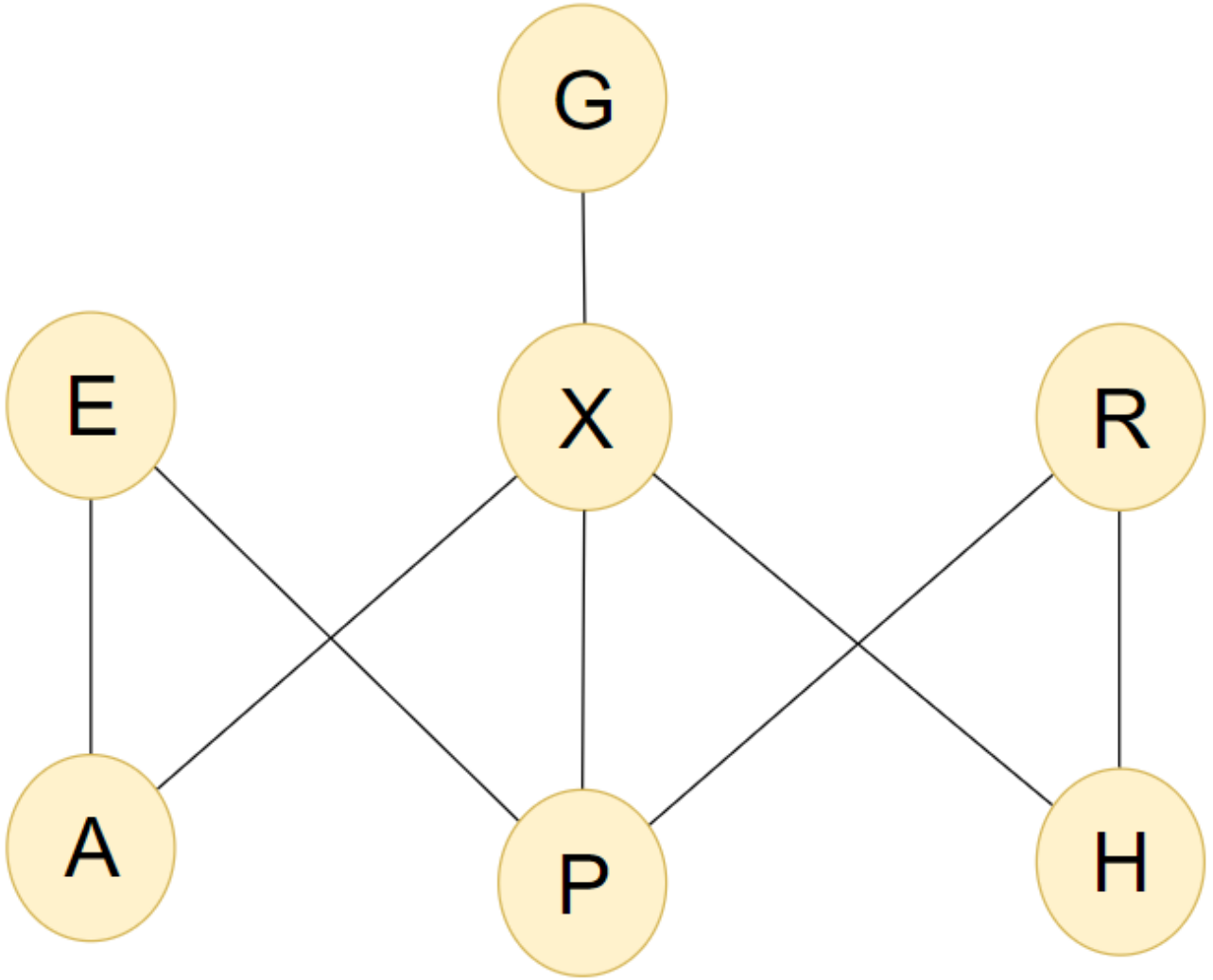
back

Traversal Order Queue

peek at the stack to see a vertex (H)
mark one of that vertex's unvisited neighbors as visited (R)
put that vertex (R) on the traversal order queue
push that vertex (R) onto the stack



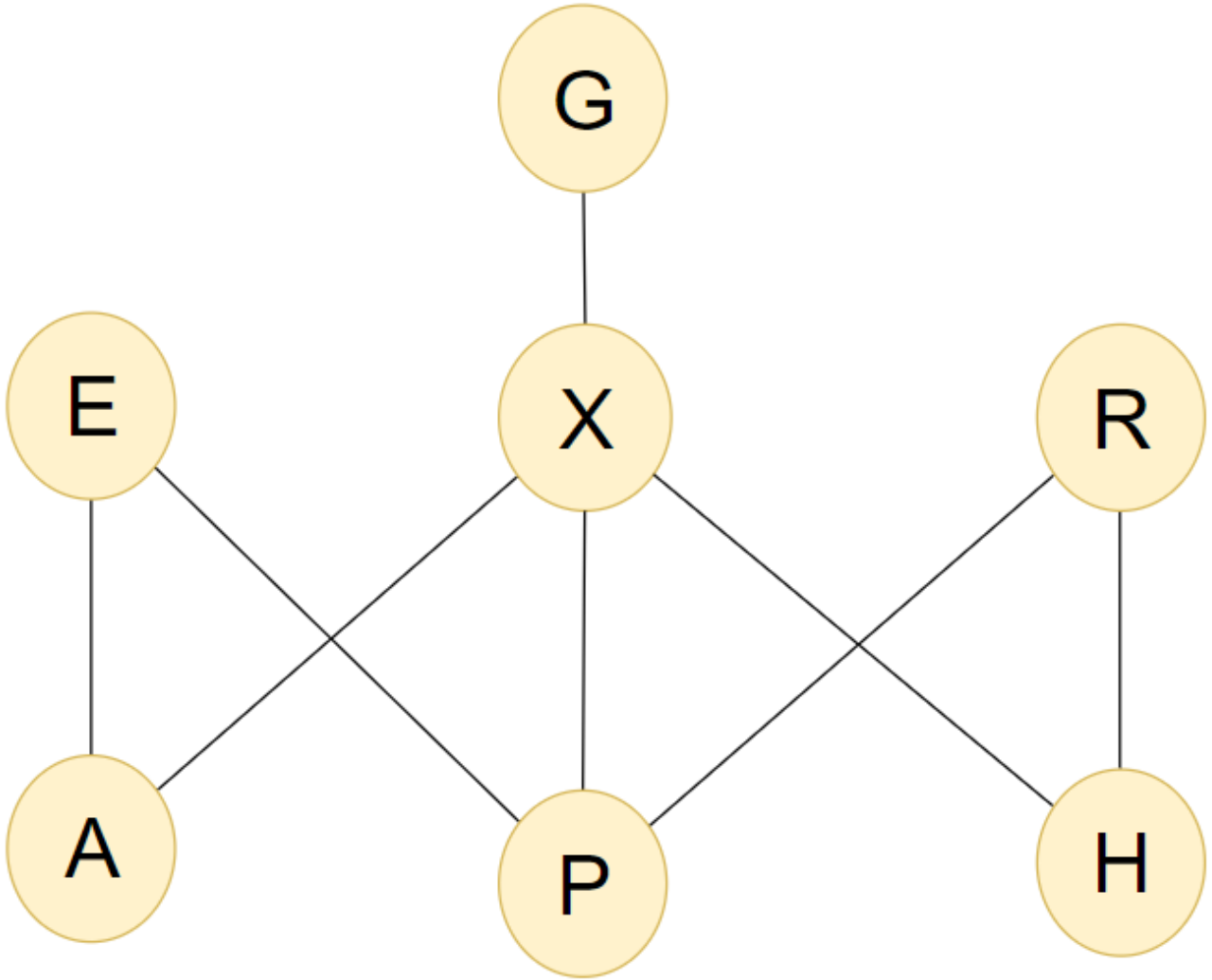
Vertex Stack



front Traversal Order Queue back

peek at the stack to see a vertex (R)
mark one of that vertex's unvisited neighbors as visited (P)
put that vertex (P) on the traversal order queue
push that vertex (P) onto the stack

Vertex Stack



E A X G H R P

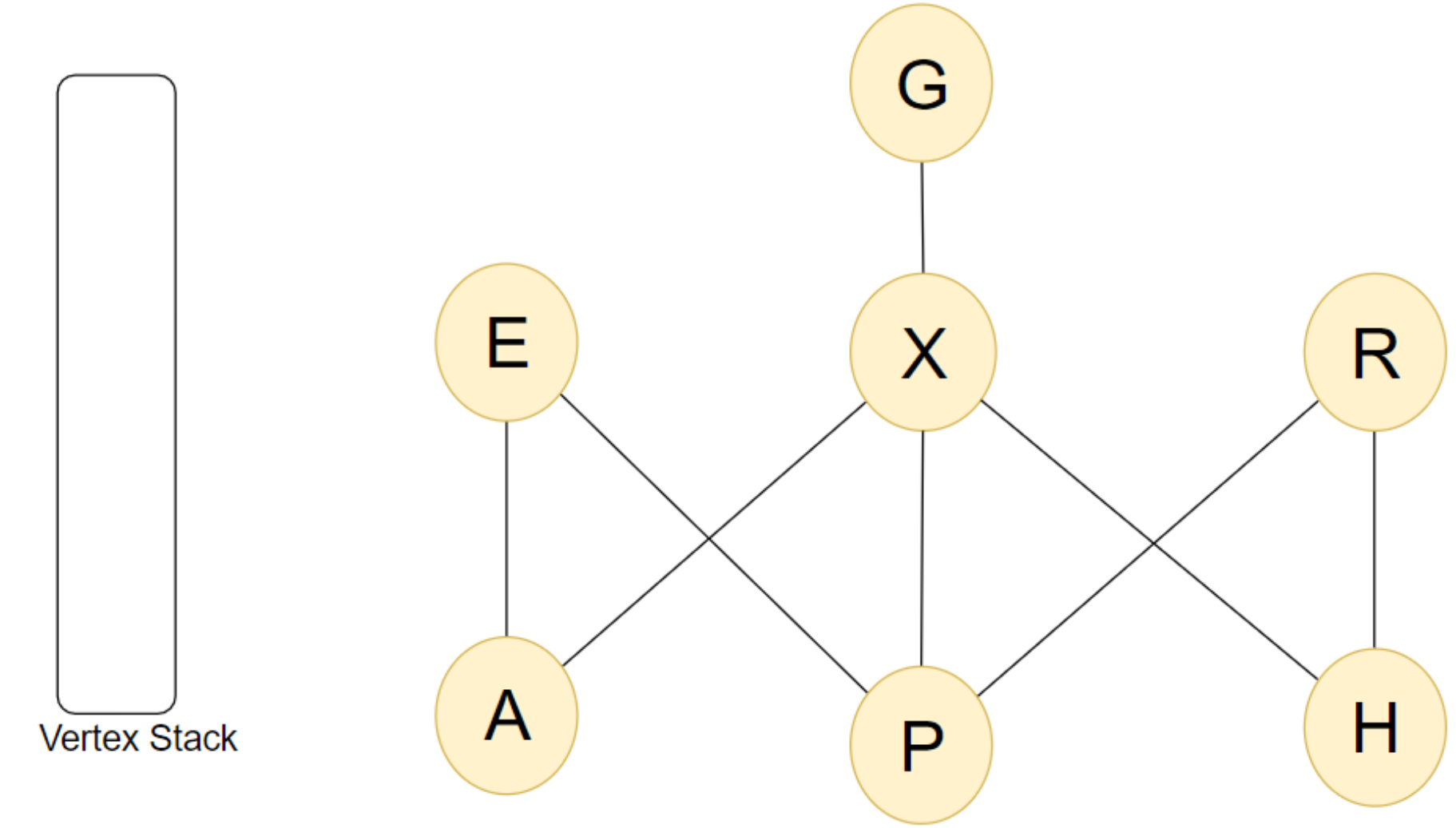
front

Traversal Order
Queue

back

peek at the stack to see a vertex (P)
that vertex has no unvisited neighbors, so pop from the stack

this continues until the stack is empty (because all vertices hav
been visited)



Vertex Stack

E A X G H R P

front

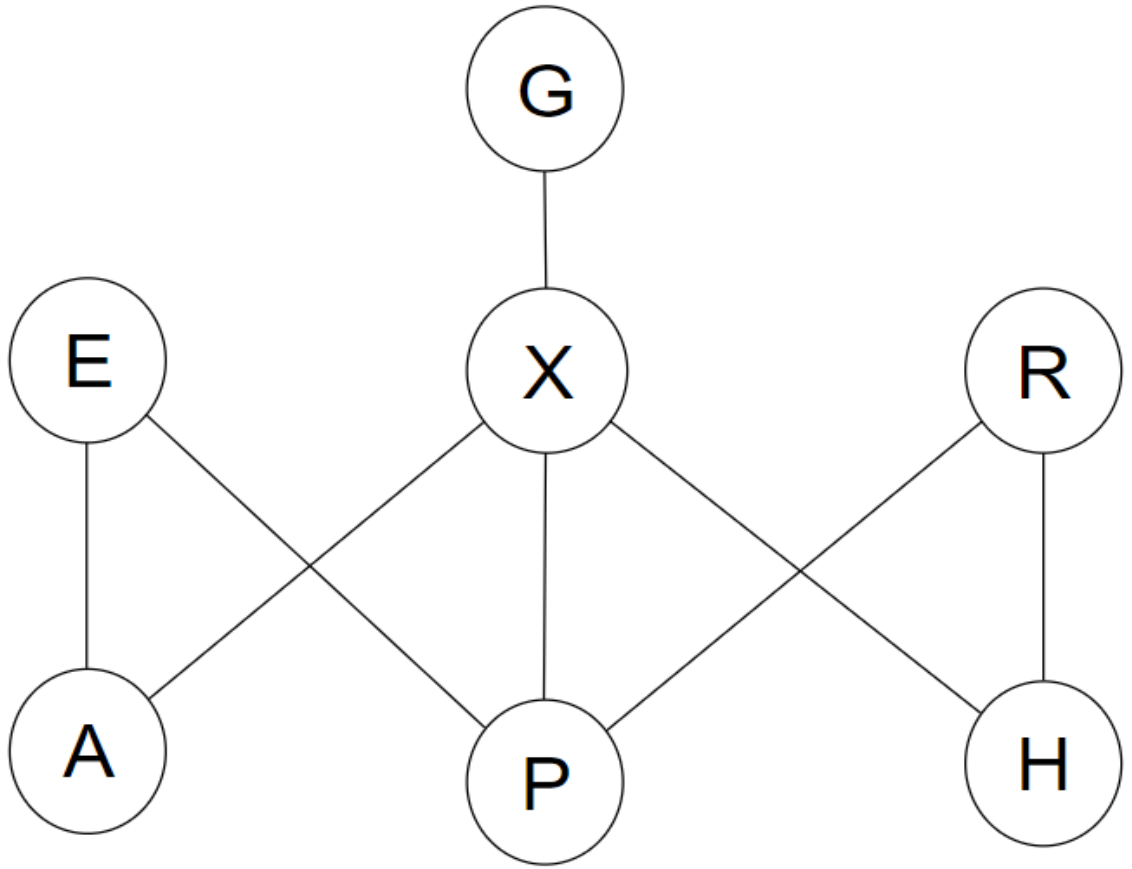
Traversal Order Queue

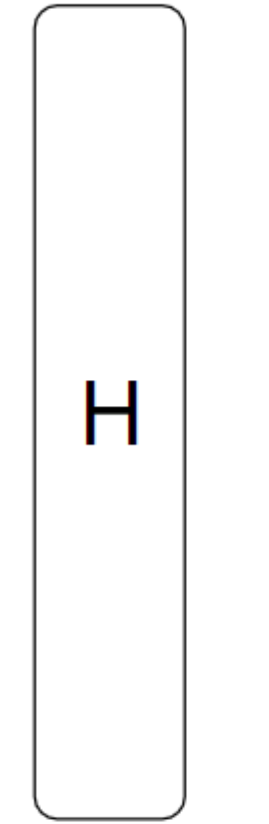
back

the traversal order is: EAXGHRP

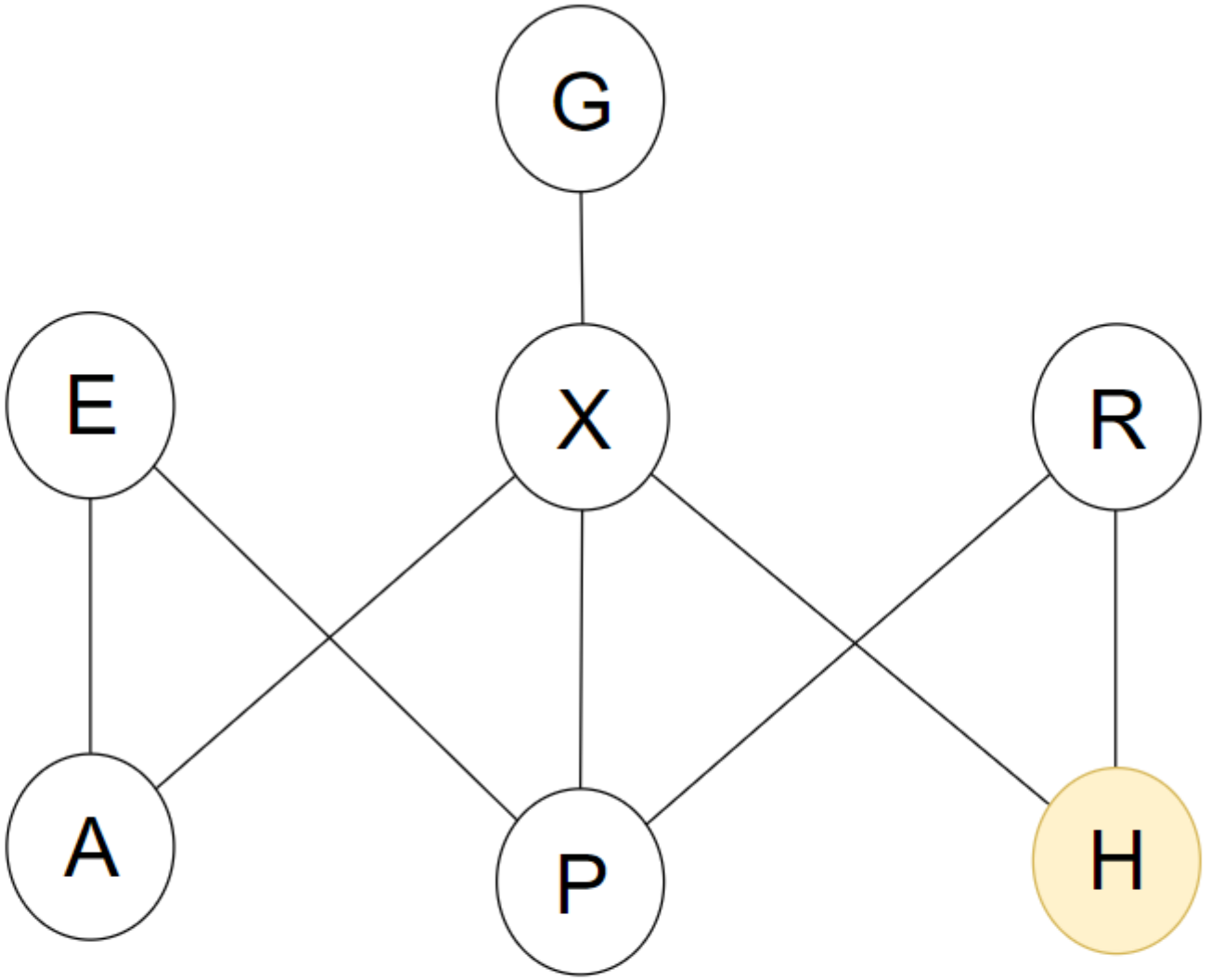
Depth-First Example

- Trace a depth-first traversal starting at vertex H.





Vertex Stack

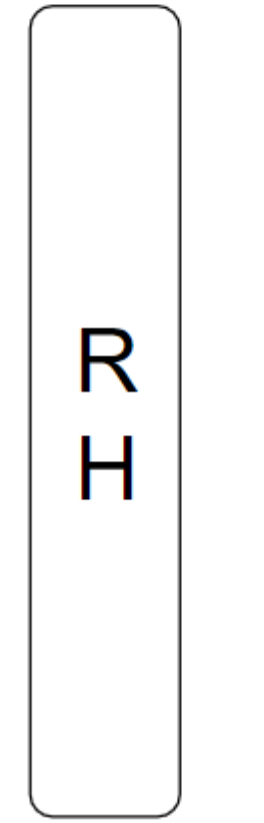


front

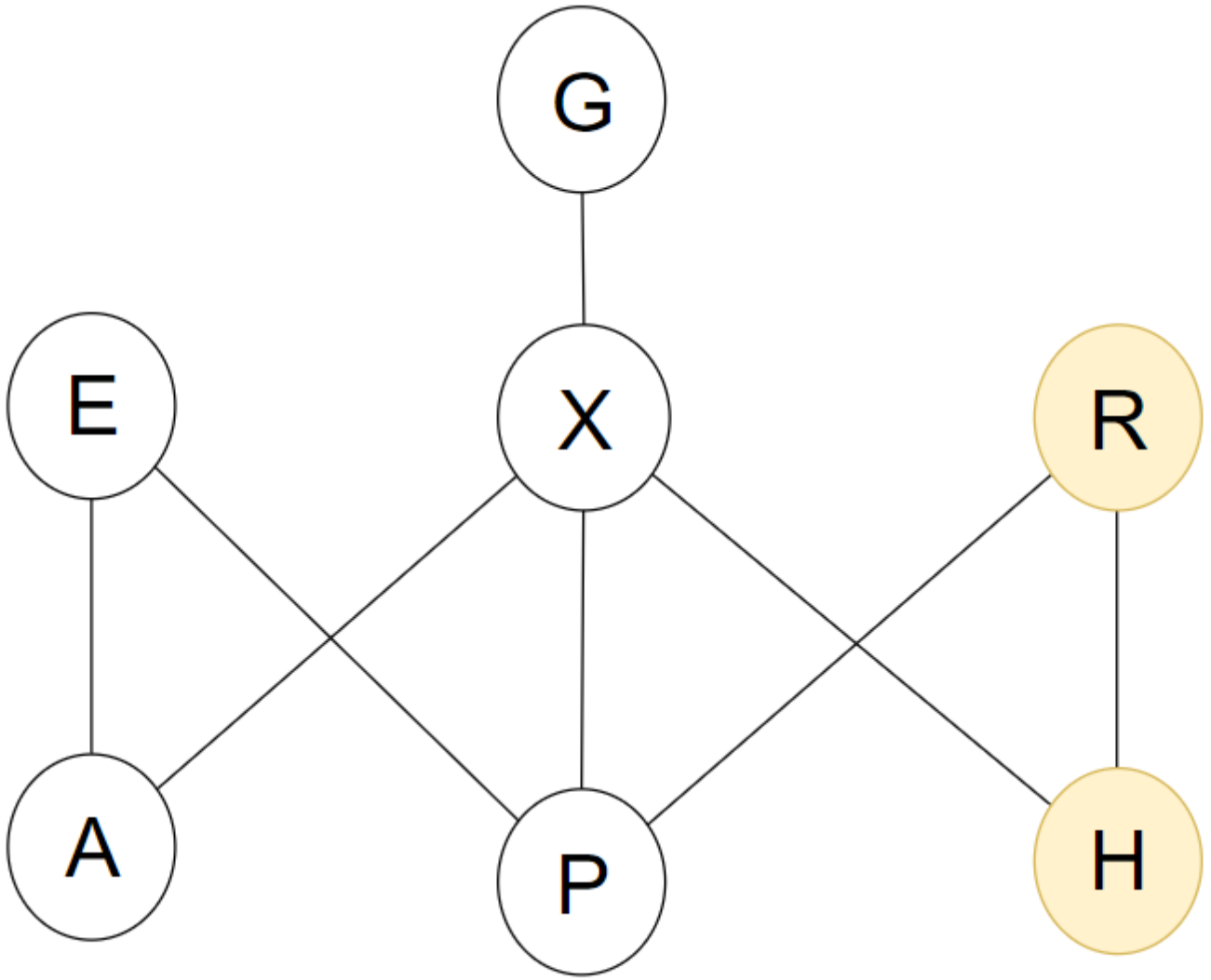
Traversal Order Queue

back

mark the origin as visited
enqueue the origin
push the origin on the stack



Vertex Stack

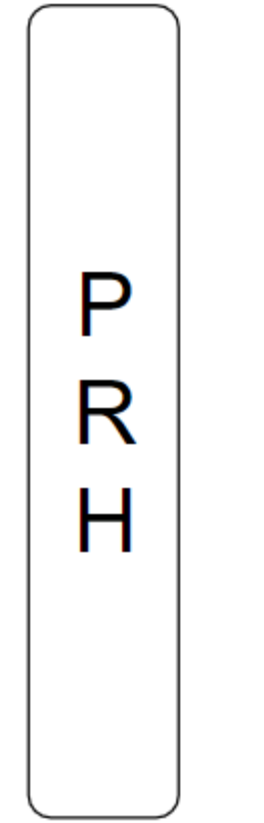


front

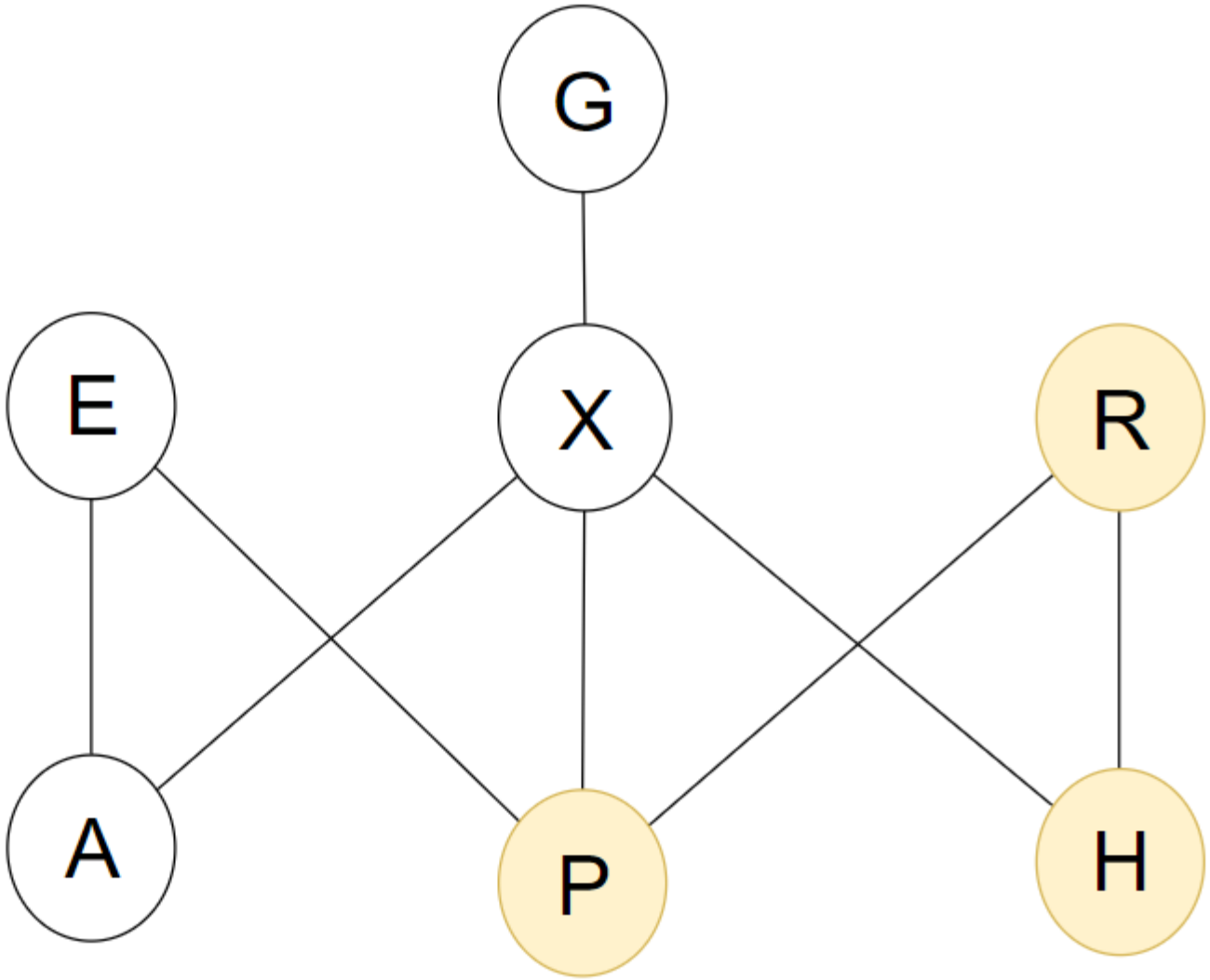
Traversal Order Queue

back

peek at the stack to see a vertex (H)
mark one of that vertex's unvisited neighbors as visited (R)
put that vertex (R) on the traversal order queue
push that vertex (R) onto the stack



Vertex Stack

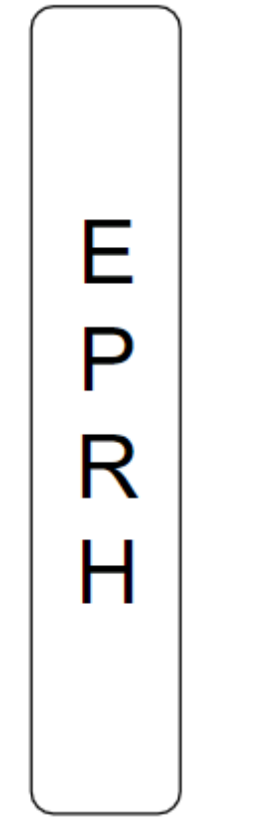


front

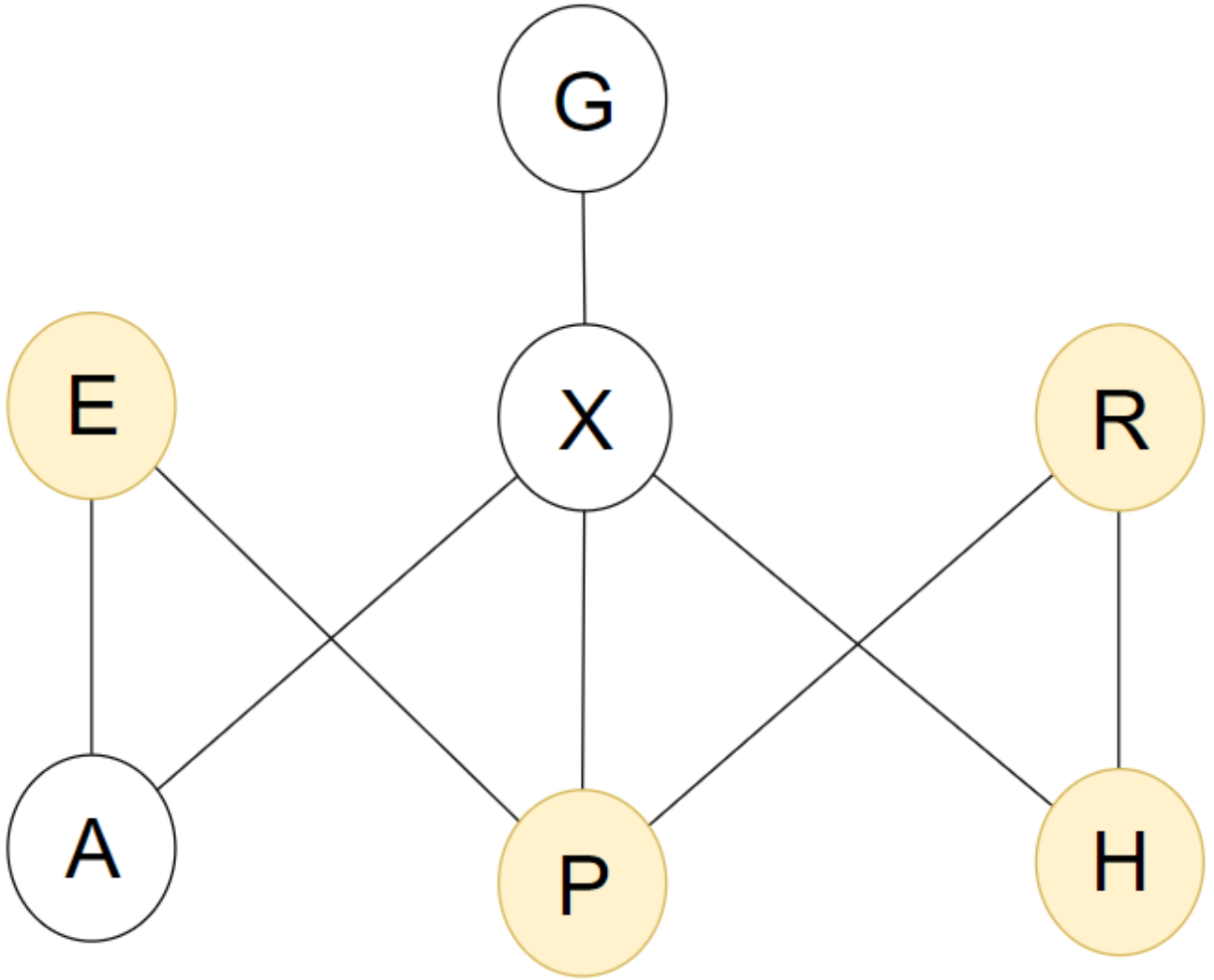
Traversal Order Queue

back

peek at the stack to see a vertex (R)
mark one of that vertex's unvisited neighbors as visited (P)
put that vertex (P) on the traversal order queue
push that vertex (P) onto the stack



Vertex Stack

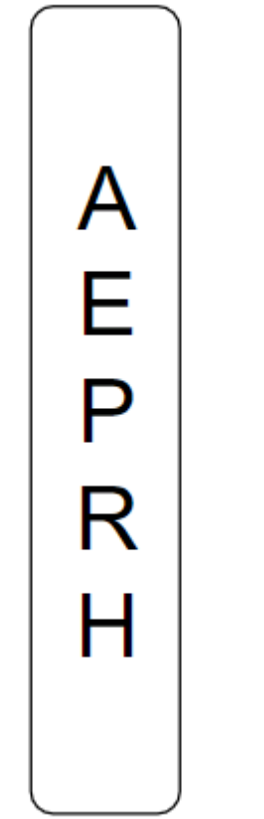


front

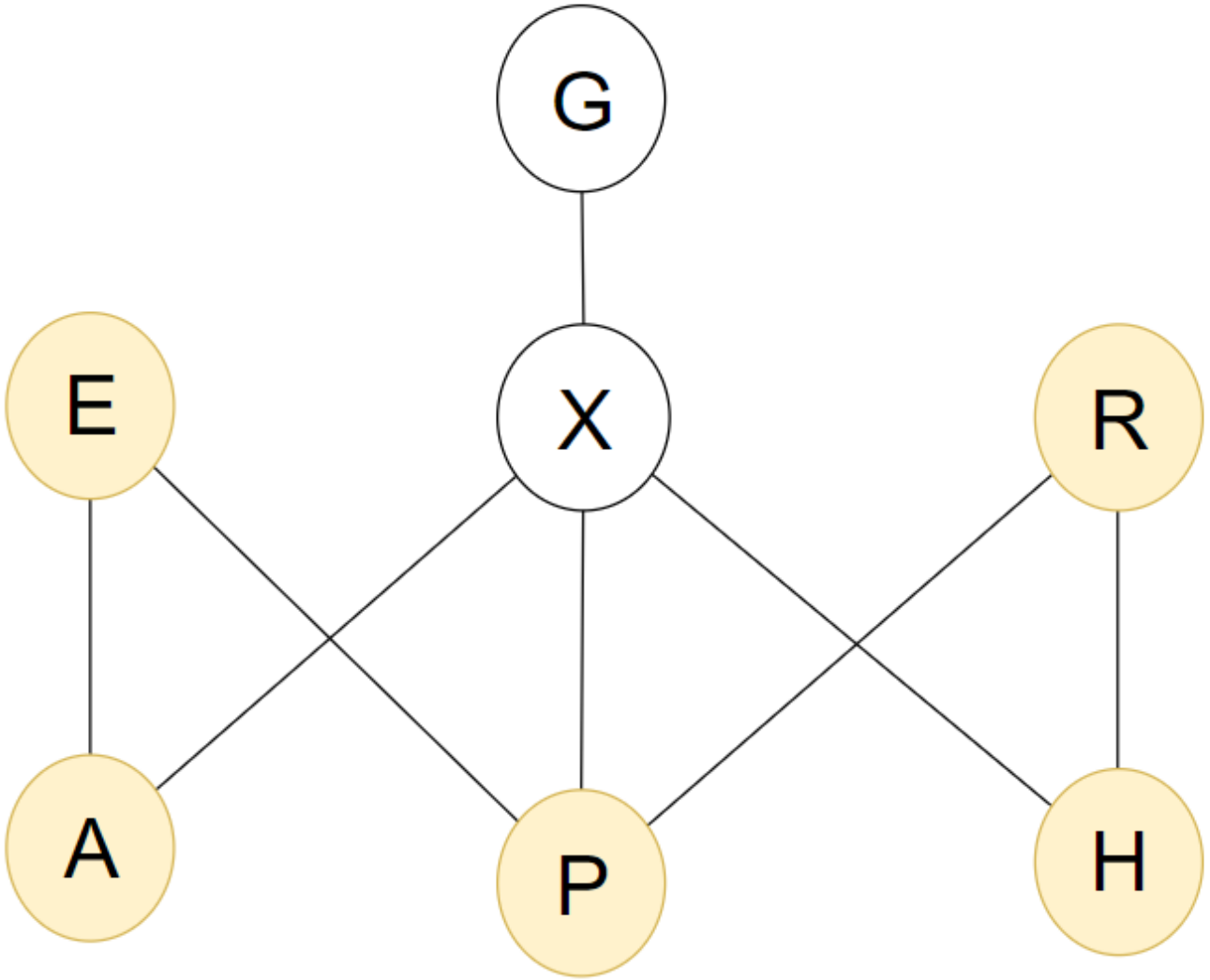
Traversal Order Queue

back

peek at the stack to see a vertex (P)
mark one of that vertex's unvisited neighbors as visited (E)
put that vertex (E) on the traversal order queue
push that vertex (E) onto the stack



Vertex Stack

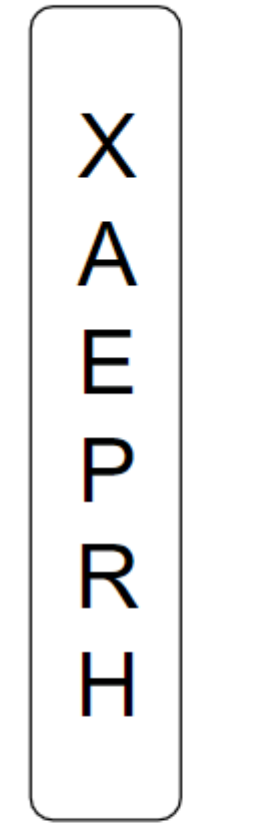


front

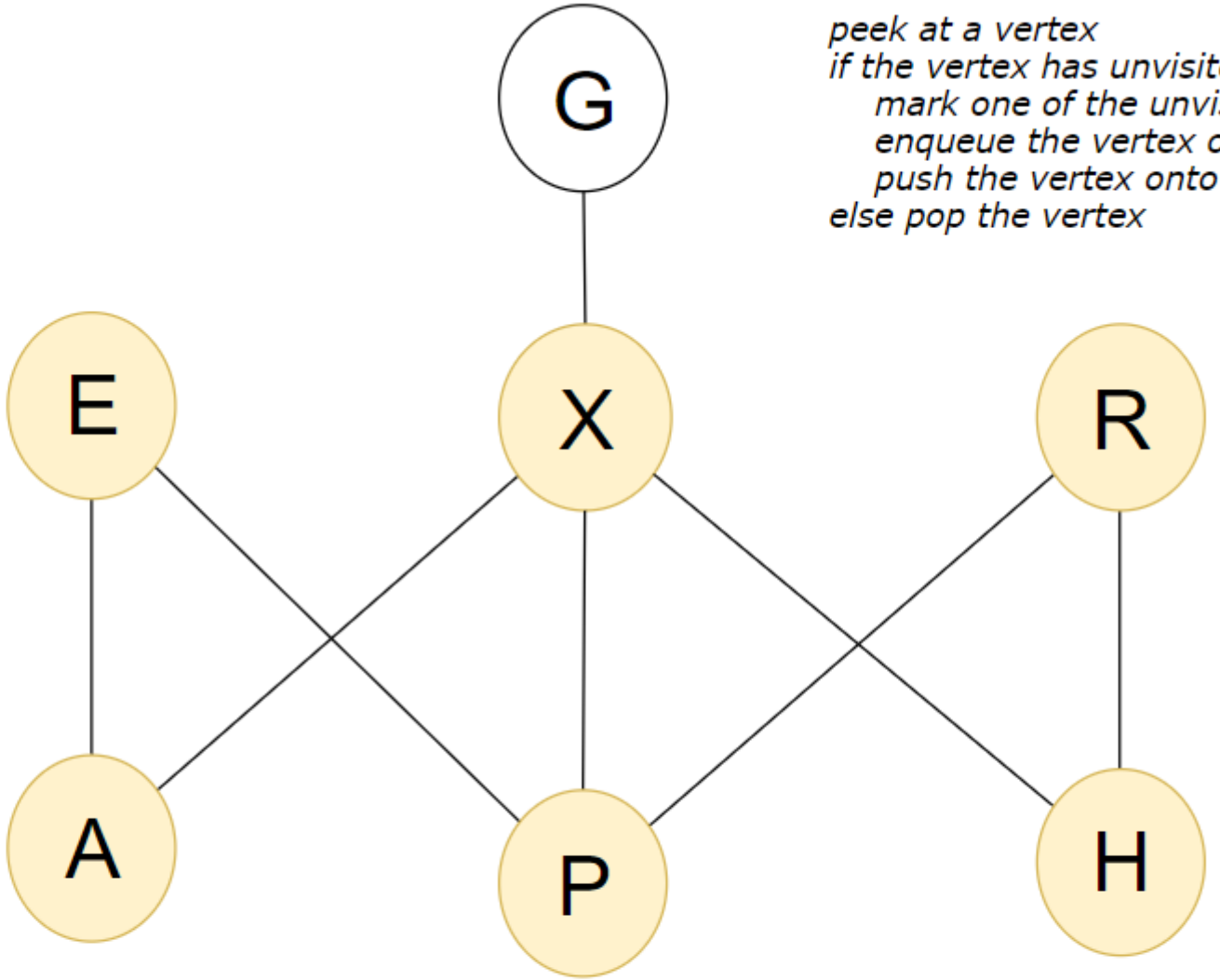
Traversal Order Queue

back

peek at the stack to see a vertex (E)
mark one of that vertex's unvisited neighbors as visited (A)
put that vertex (A) on the traversal order queue
push that vertex (A) onto the stack



Vertex Stack



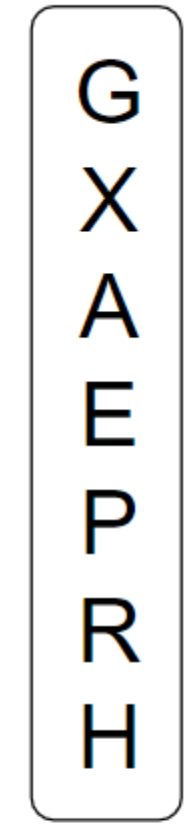
peek at a vertex
if the vertex has unvisited neigh
mark one of the unvisited ne
enqueue the vertex onto the
push the vertex onto the ver
else pop the vertex

front

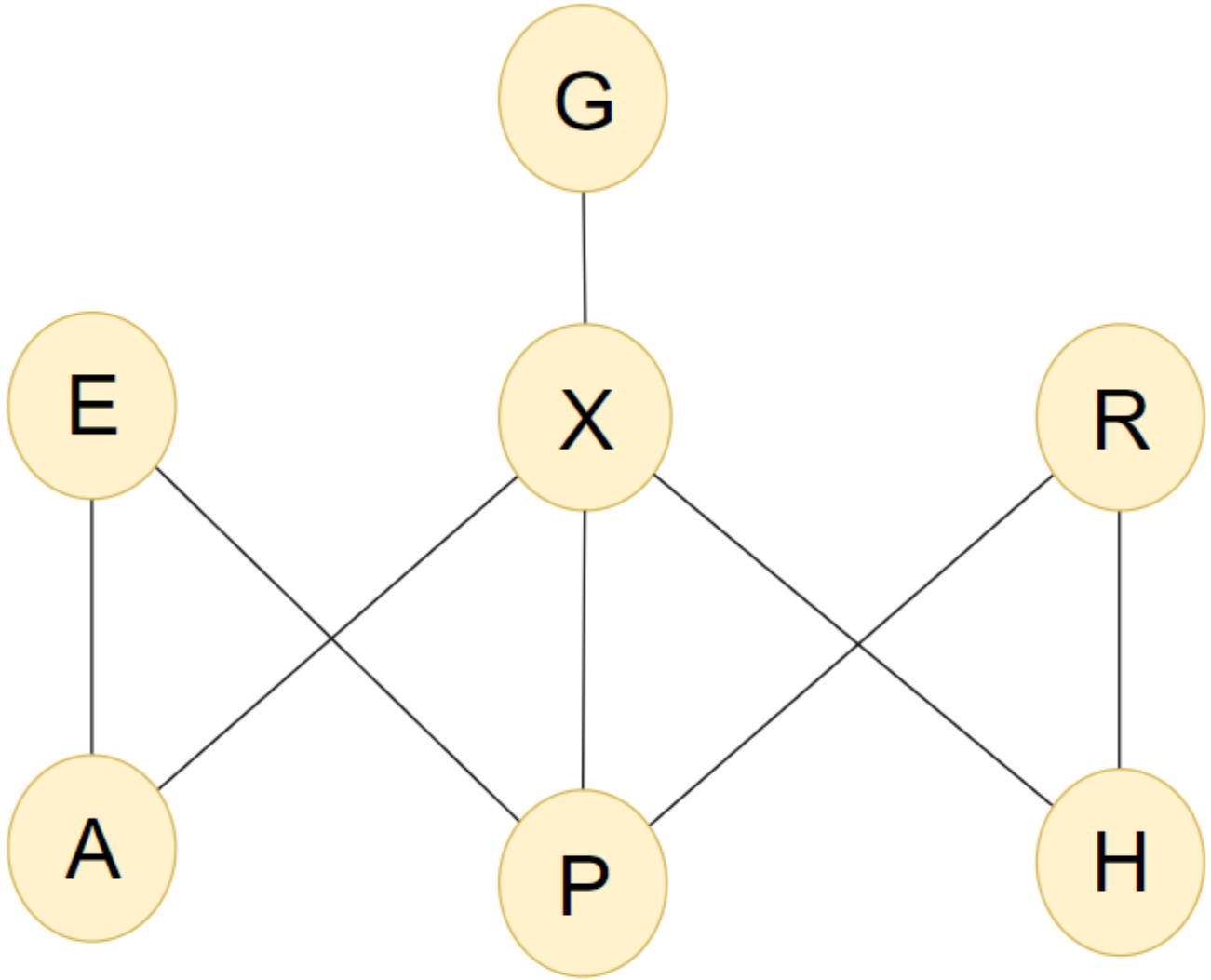
Traversal Order Queue

back

peek at the stack to see a vertex (A)
mark one of that vertex's unvisited neighbors as visited (X)
put that vertex (X) on the traversal order queue
push that vertex (X) onto the stack



Vertex Stack



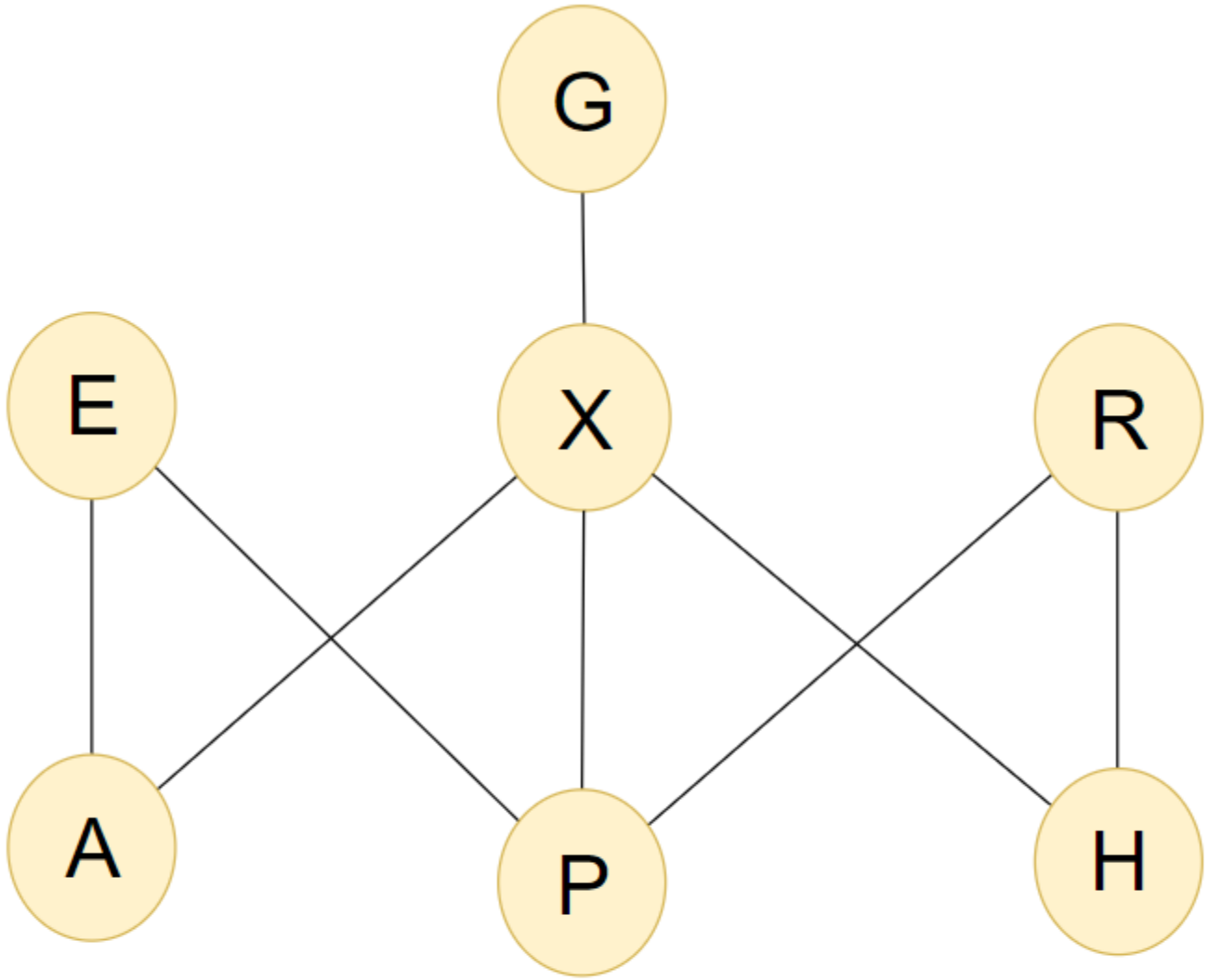
front

Traversal Order Queue

back

peek at the stack to see a vertex (X)
mark one of that vertex's unvisited neighbors as visited (G)
put that vertex (G) on the traversal order queue
push that vertex (G) onto the stack

Vertex Stack



H R P E A X G

front

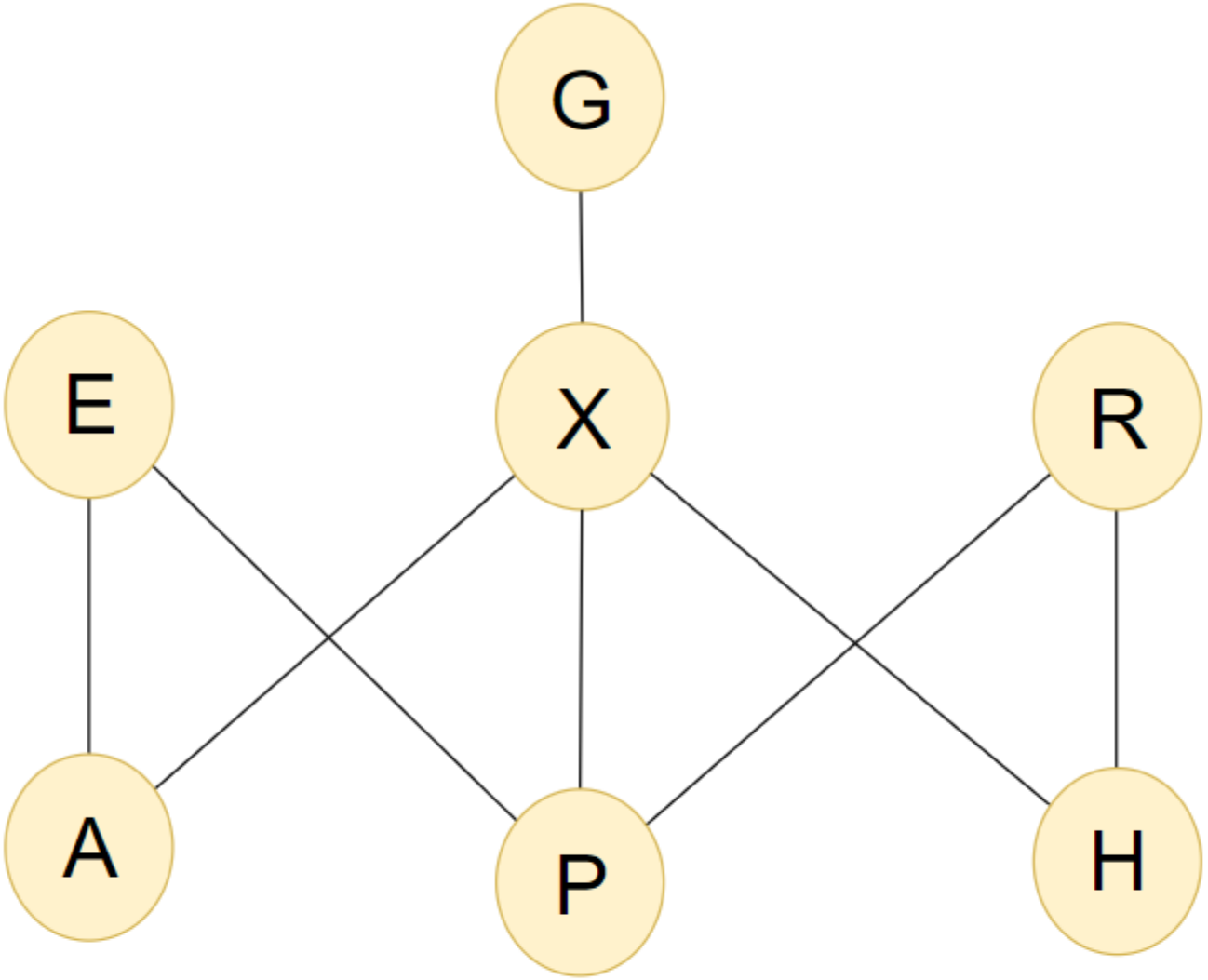
back

Traversal Order Queue

peek at the stack to see a vertex (G)
that vertex has no unvisited neighbors, so pop from the stack

this continues until the stack is empty (because all vertices hav
been visited)

Vertex Stack



H R P E A X G

front

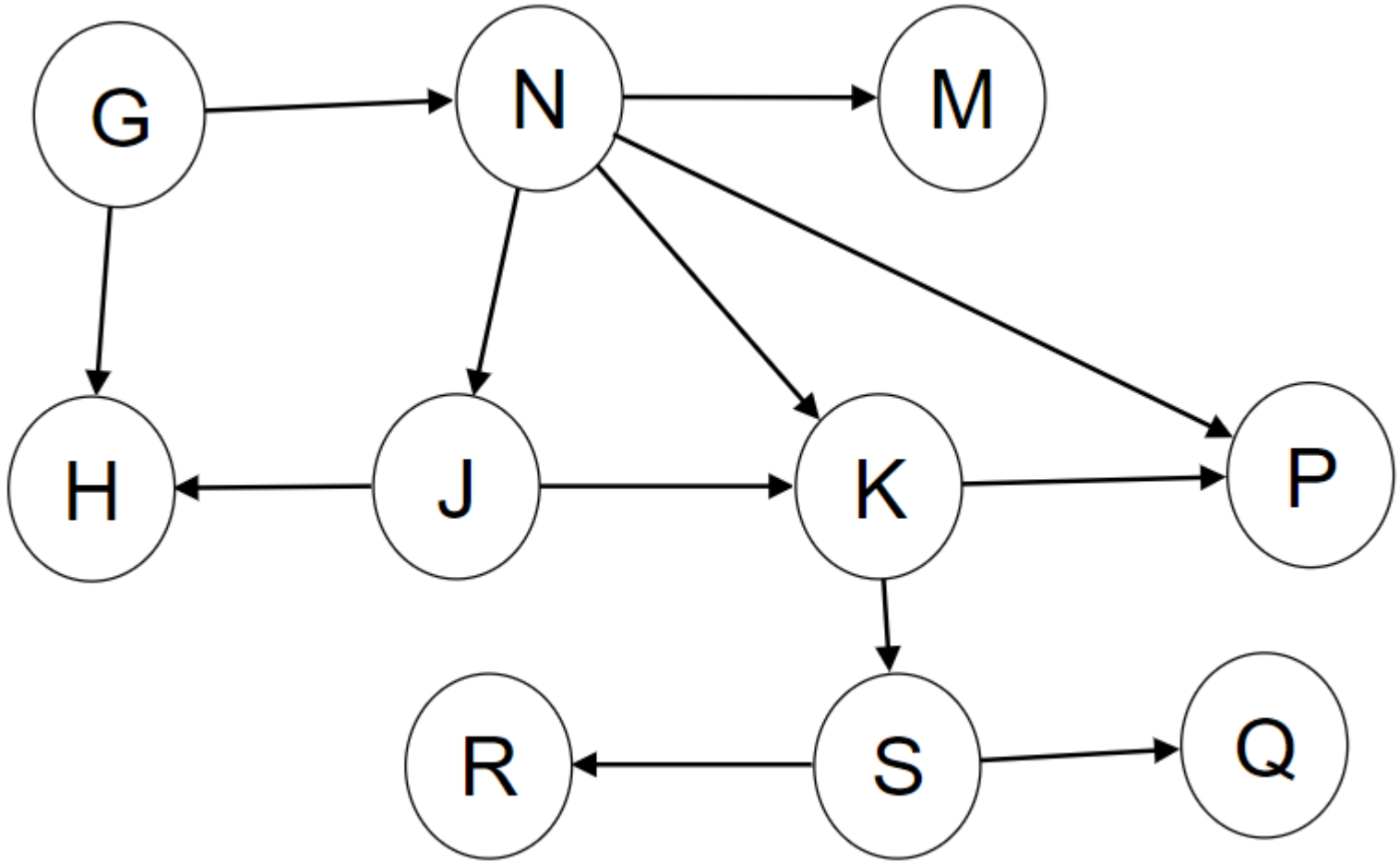
Traversal Order Queue

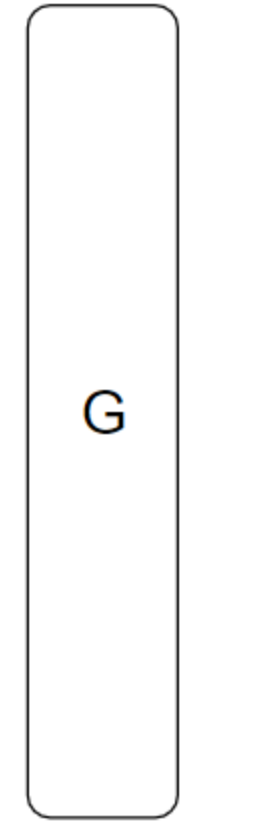
back

the traversal order is: HRPEAXG

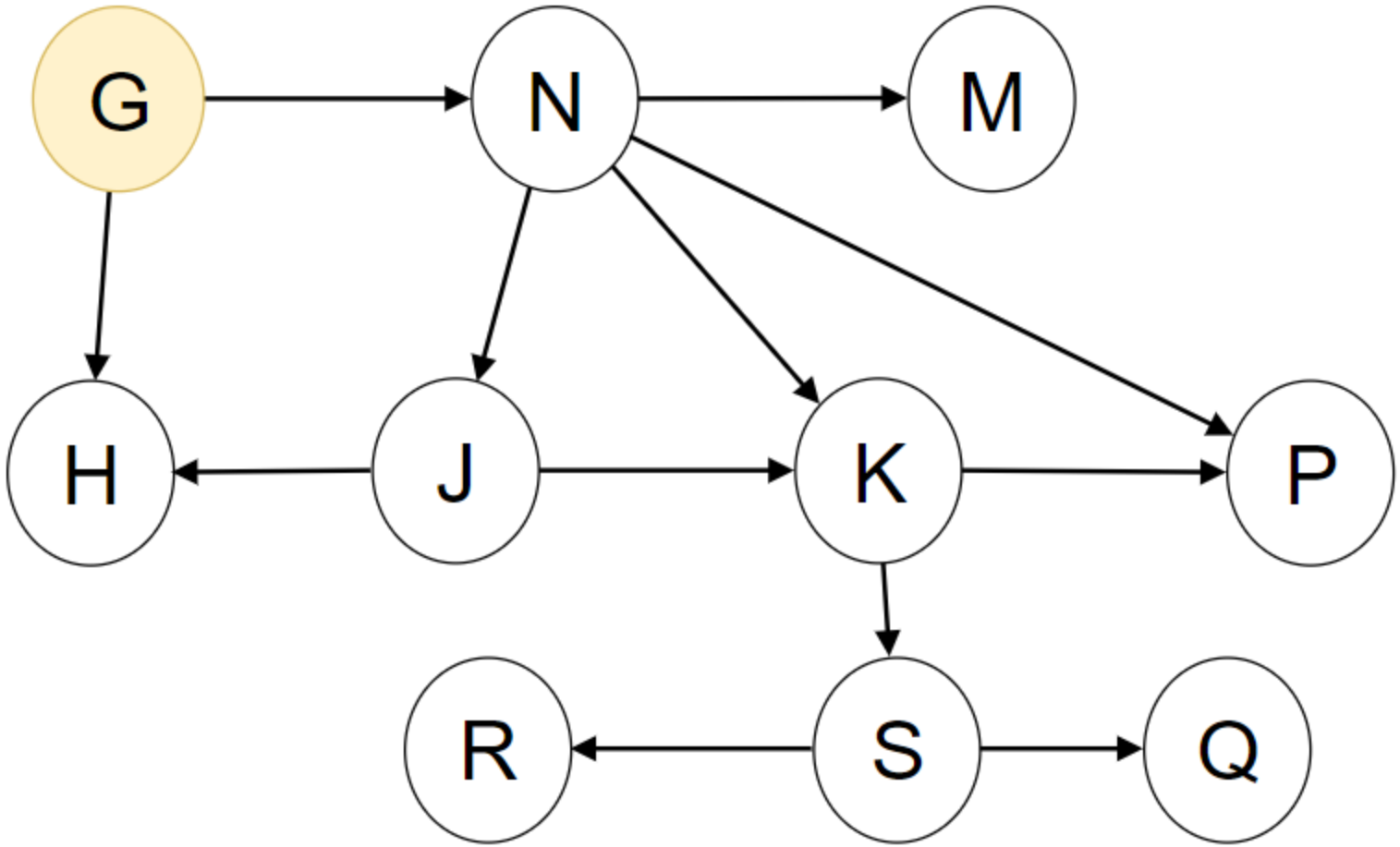
Depth-First Example

- Trace a depth first traversal starting at vertex G.





Vertex Stack

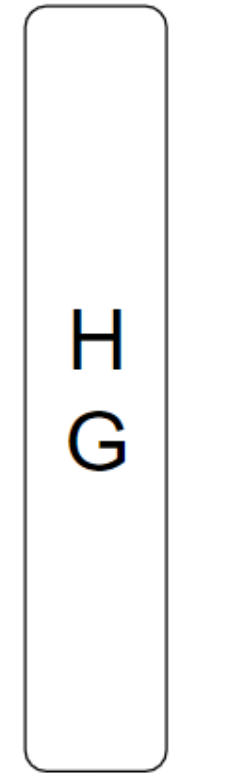


front

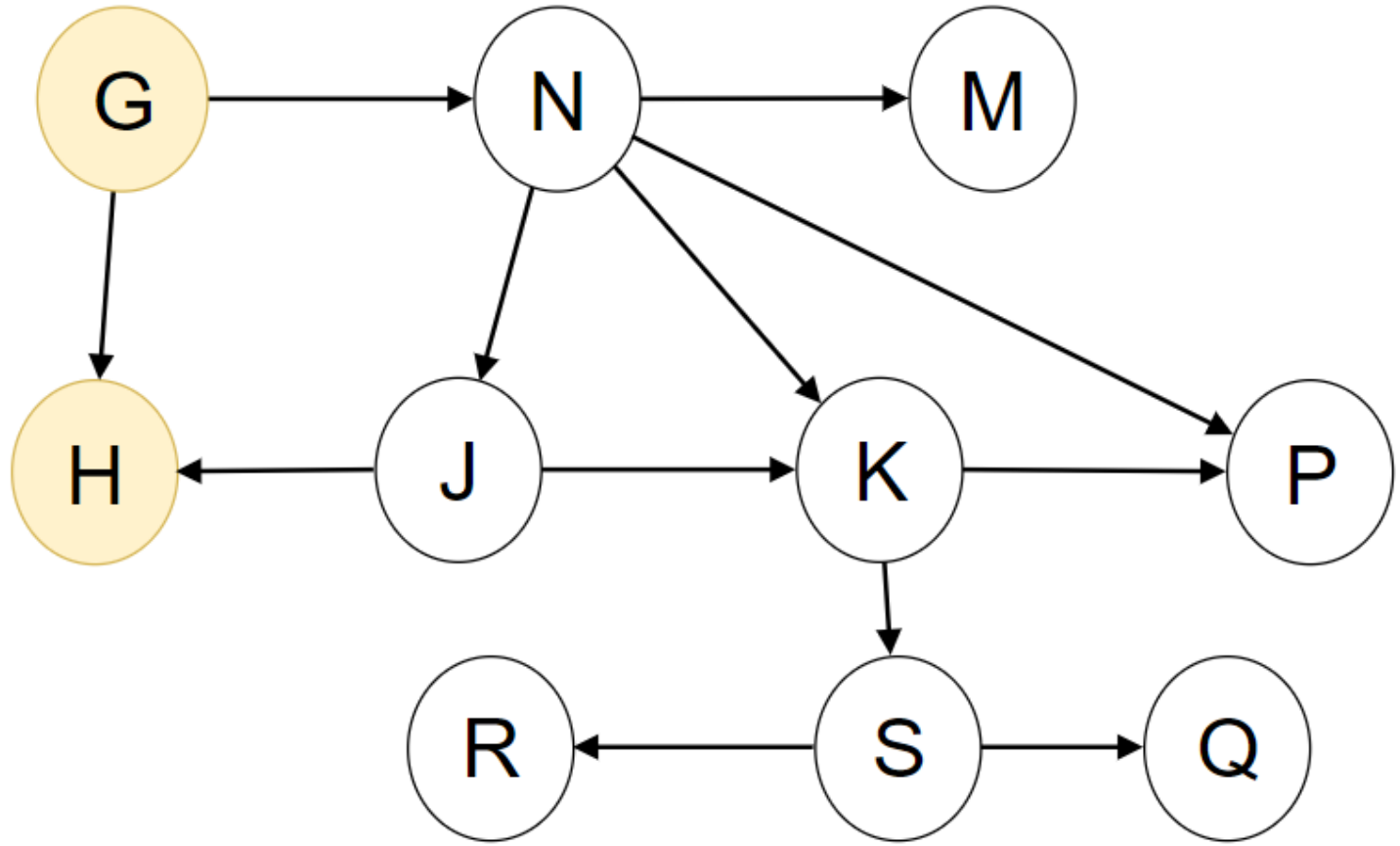
Traversal Order

back

mark the origin as visited
enqueue the origin
push the origin on the stack



Vertex Stack

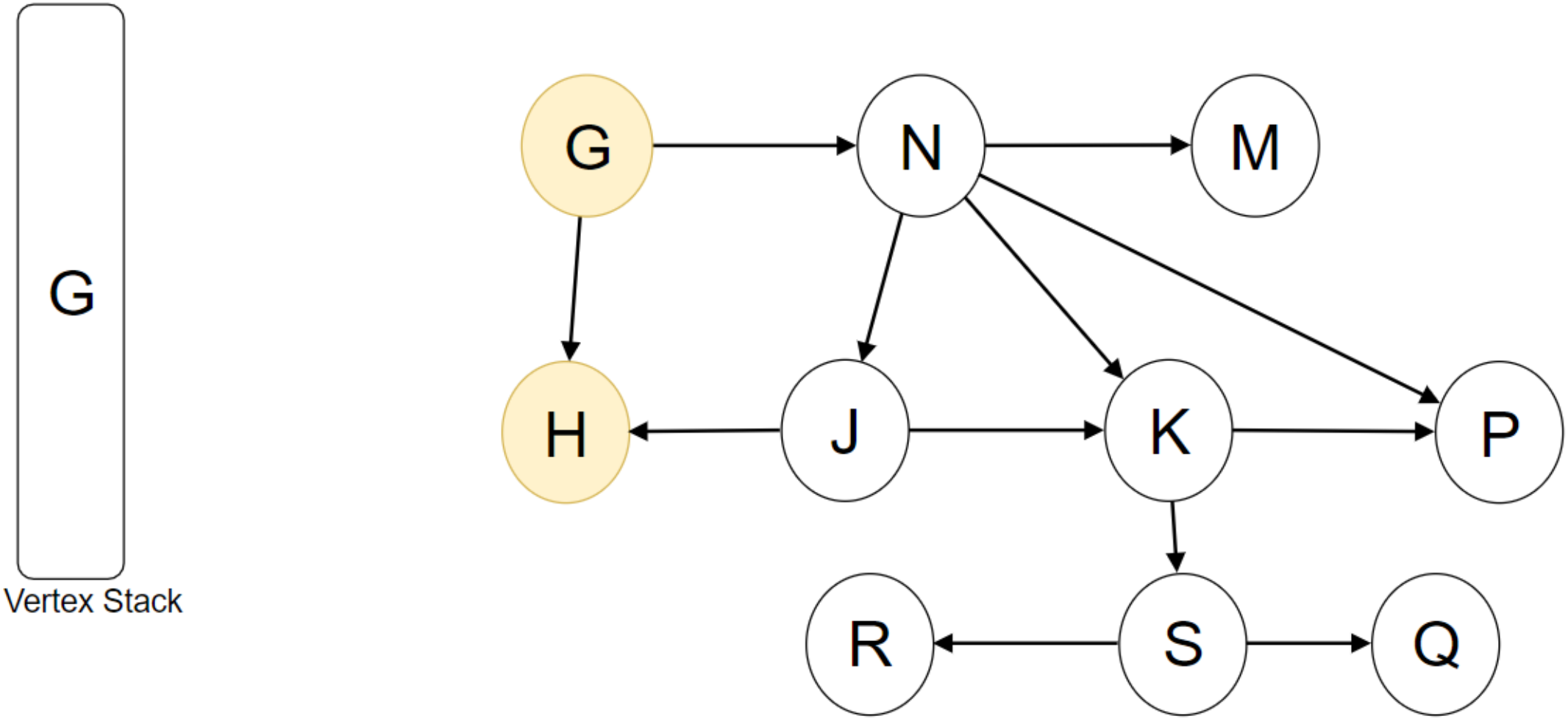


front

back

Traversal Order Queue

peek at the stack to see a vertex (G)
mark one of the unvisited neighbors as visited (H)
put that vertex (H) on the traversal order queue
push that vertex (H) onto the stack



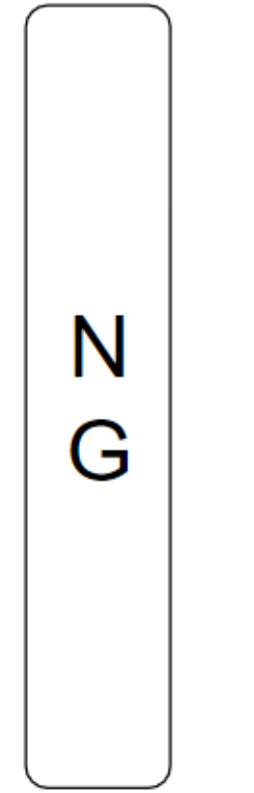
G H

front

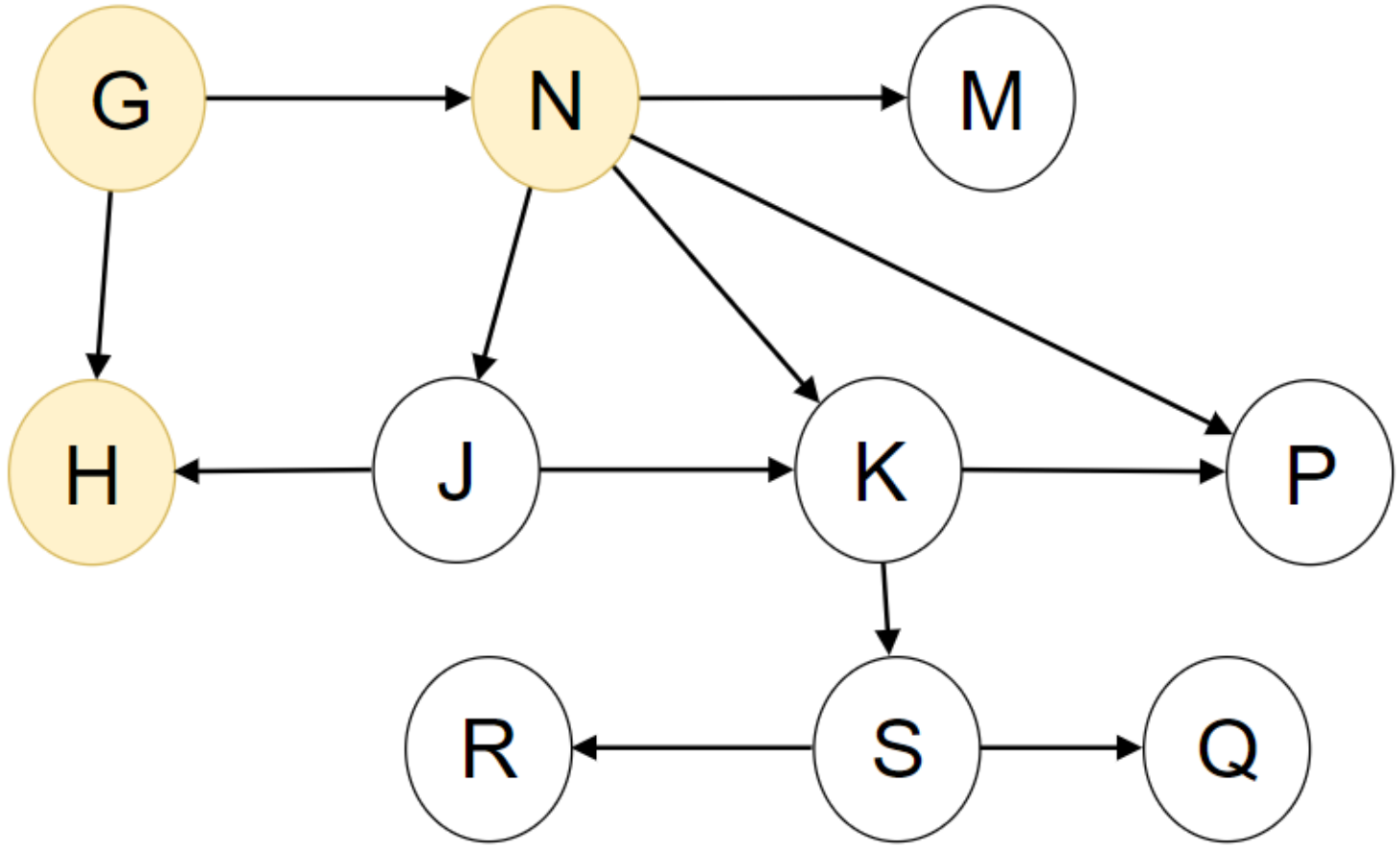
back

Traversal Order
Queue

peek at the stack to see a vertex (H)
no unvisited neighbors, so pop from the stack



Vertex Stack

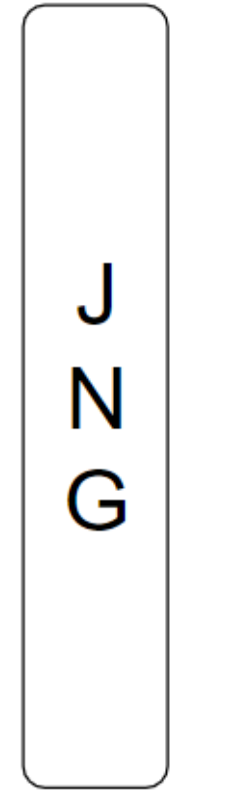


front

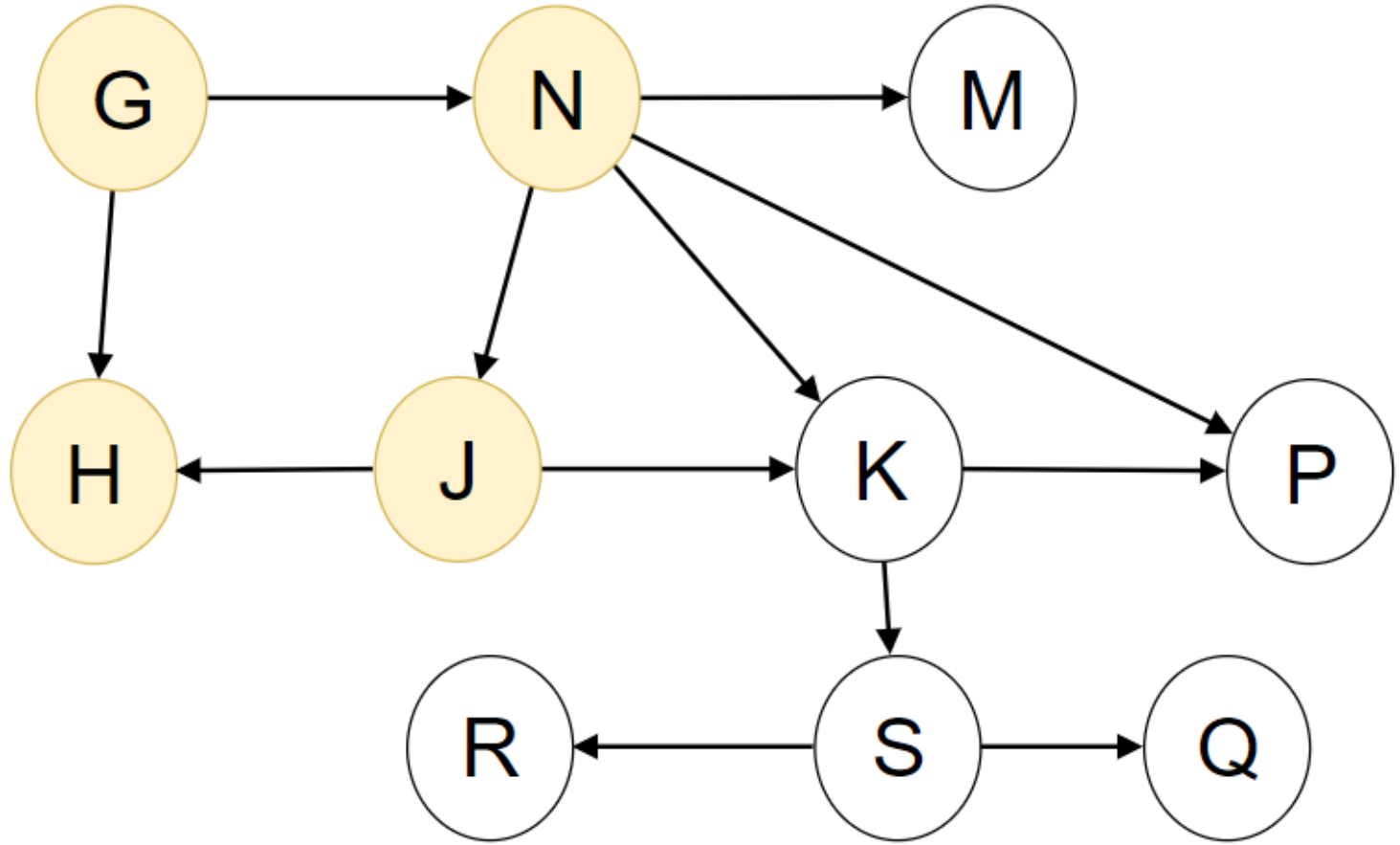
Traversal Order Queue

back

peek at the stack to see a vertex (G)
mark one of the unvisited neighbors as visited (N)
put that vertex (N) on the traversal order queue
push that vertex (N) onto the stack



Vertex Stack

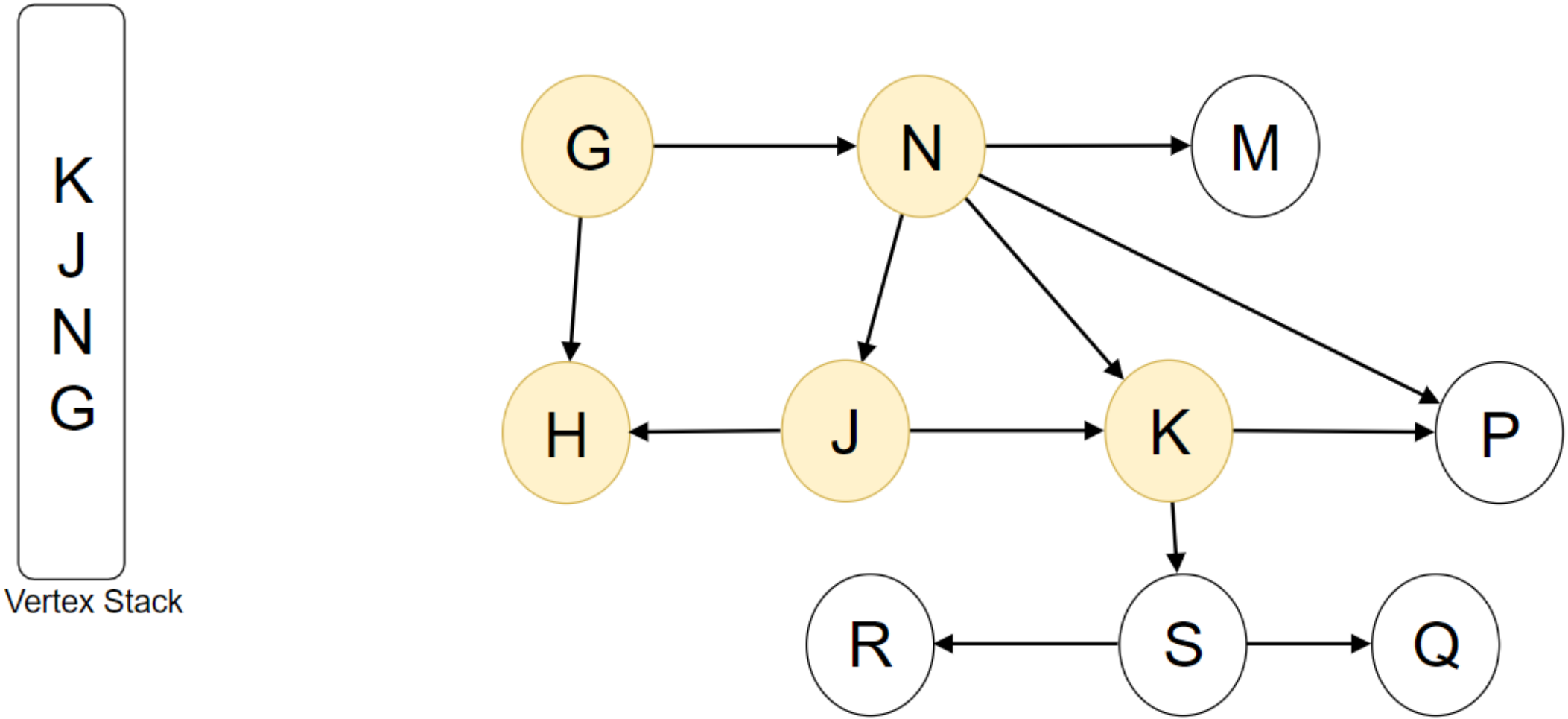


front

Traversal Order Queue

back

peek at the stack to see a vertex (N)
mark one of the unvisited neighbors as visited (J)
put that vertex (J) on the traversal order queue
push that vertex (J) onto the stack



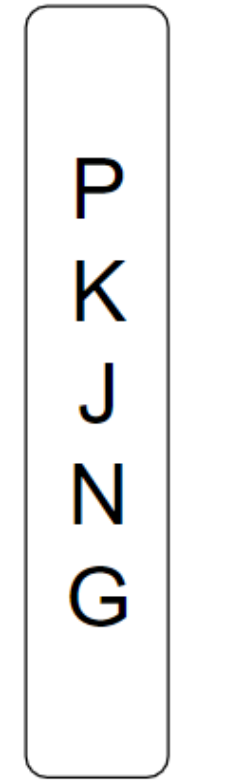
G H N J K

front

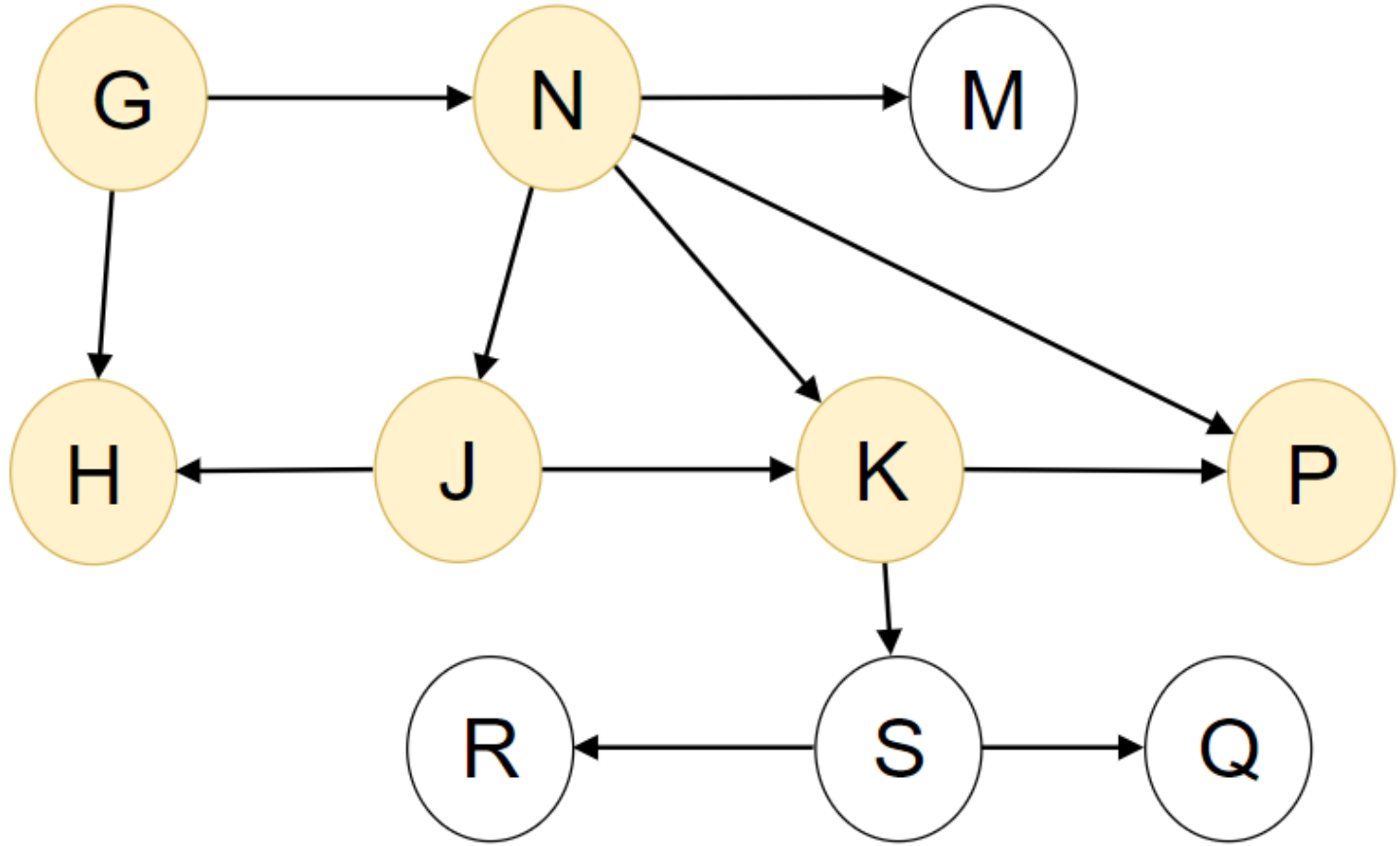
Traversal Order Queue

back

peek at the stack to see a vertex (J)
mark one of the unvisited neighbors as visited (K)
put that vertex (K) on the traversal order queue
push that vertex (K) onto the stack



Vertex Stack

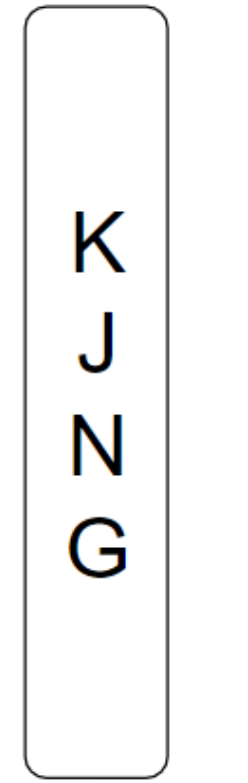


front

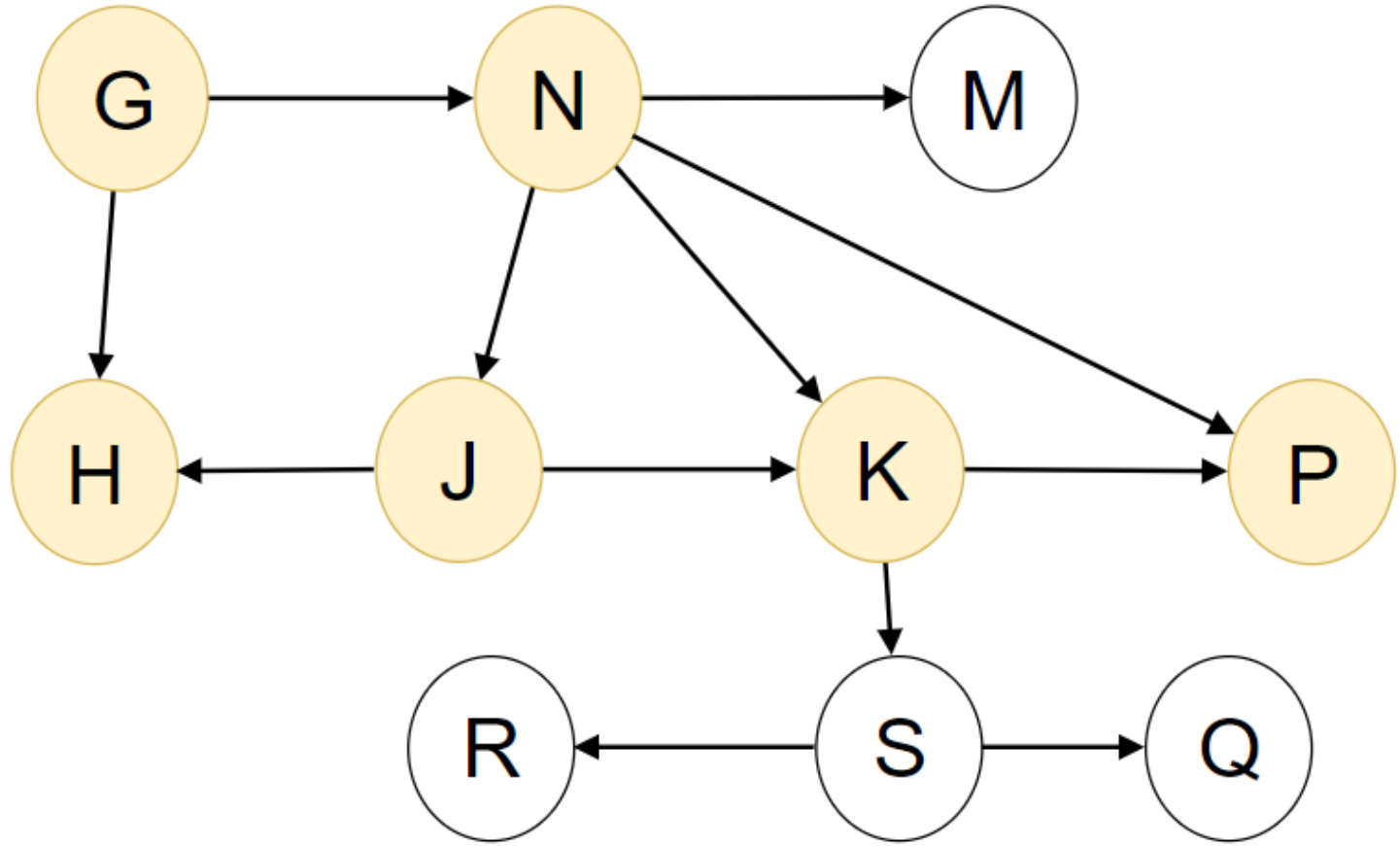
Traversal Order Queue

back

peek at the stack to see a vertex (K)
mark one of the unvisited neighbors as visited (P)
put that vertex (P) on the traversal order queue
push that vertex (P) onto the stack



Vertex Stack



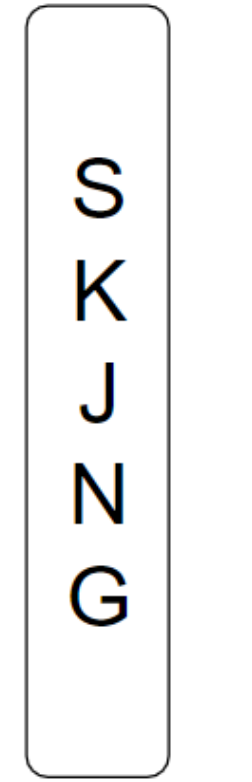
G H N J K P

front

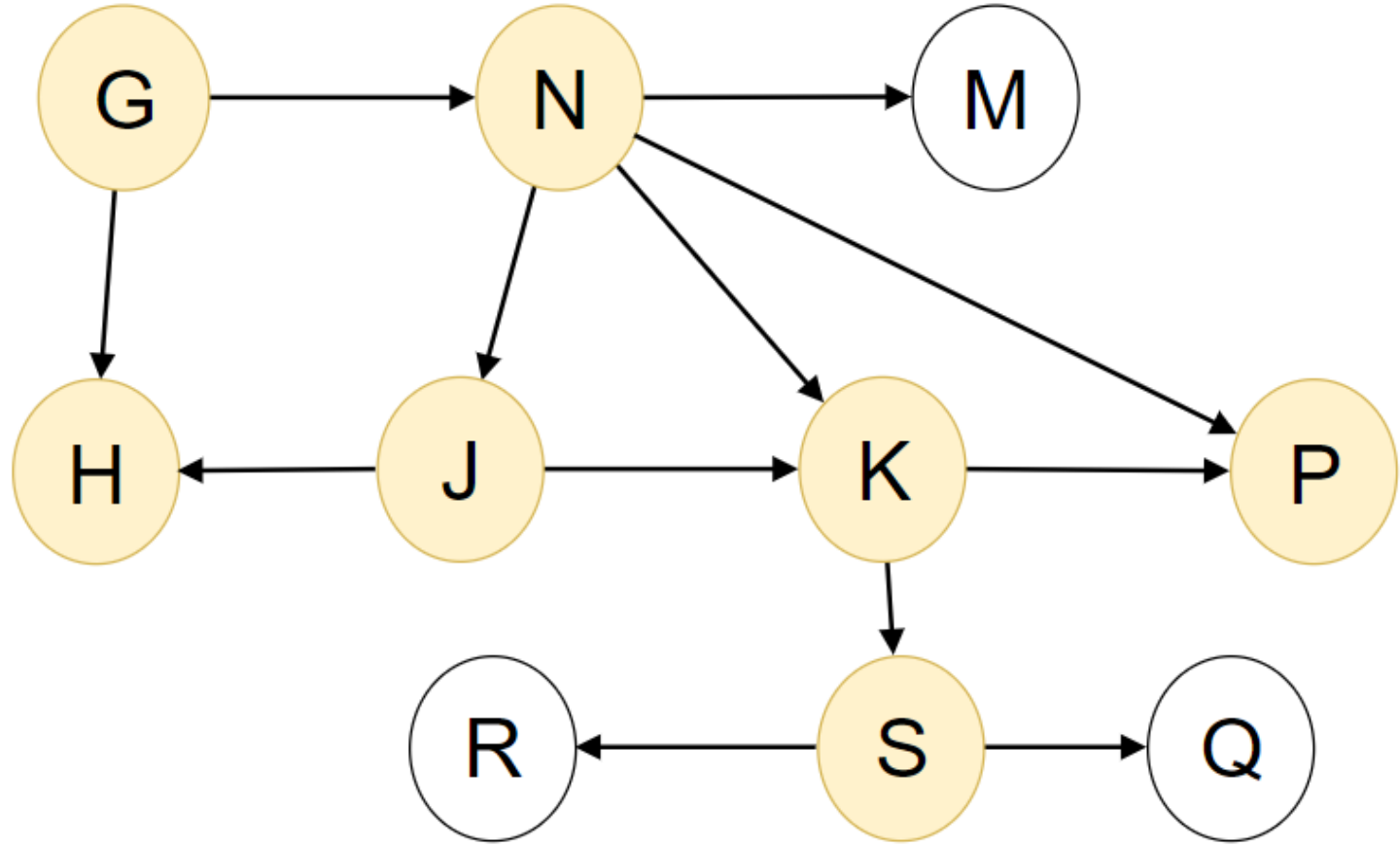
Traversal Order
Queue

back

peek at the stack to see a vertex (P)
no unvisited neighbors, so pop from the stack



Vertex Stack

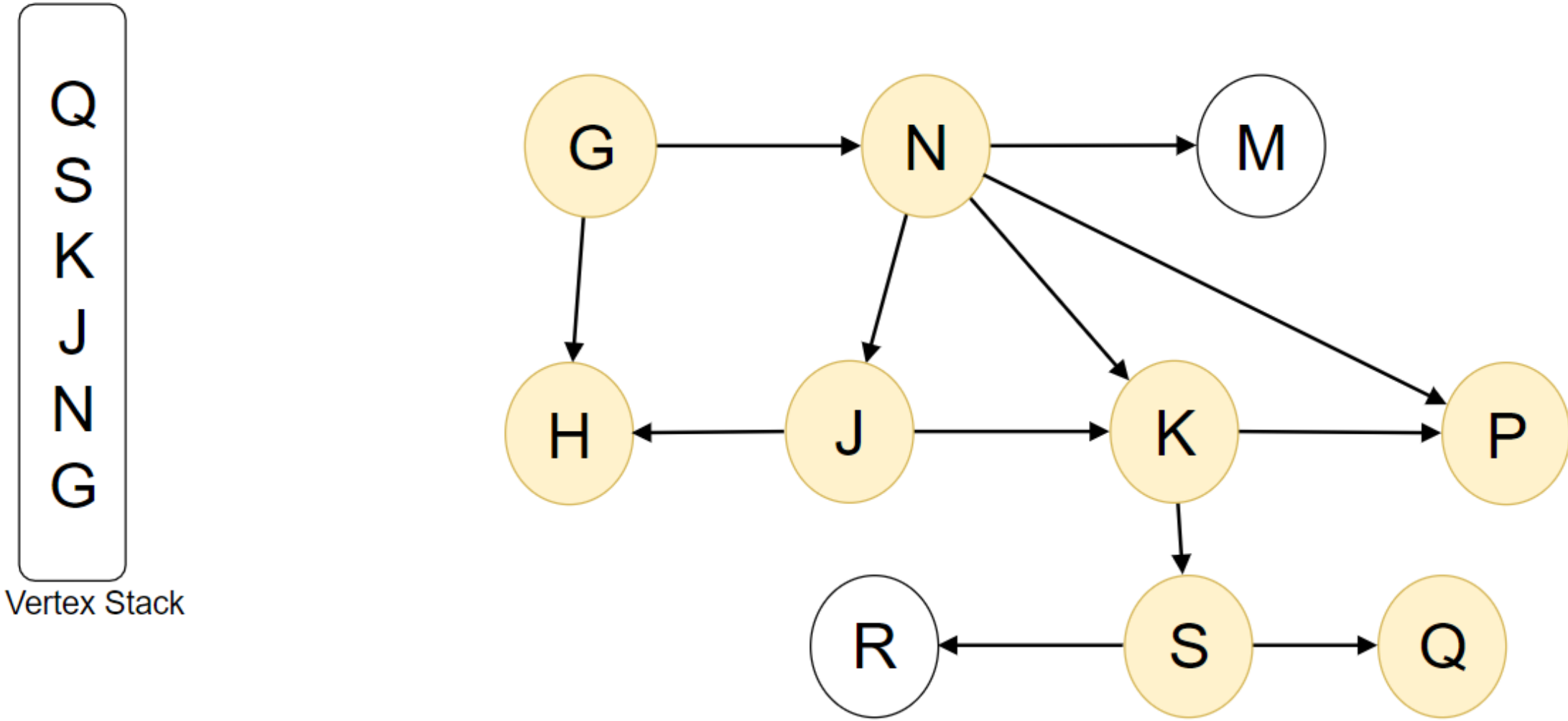


front

Traversal Order Queue

back

peek at the stack to see a vertex (K)
mark one of the unvisited neighbors as visited (S)
put that vertex (S) on the traversal order queue
push that vertex (S) onto the stack



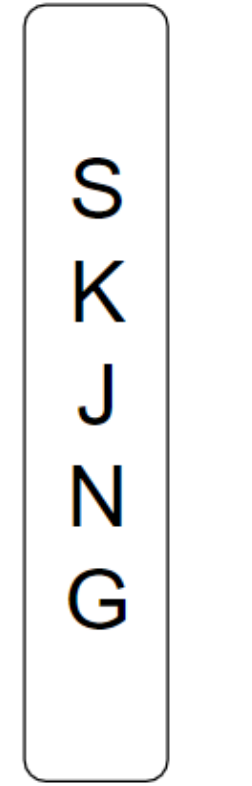
peek at the stack to see a vertex (S)
mark one of the unvisited neighbors as visited (Q)
put that vertex (Q) on the traversal order queue
push that vertex (Q) onto the stack

G H N J K P S Q

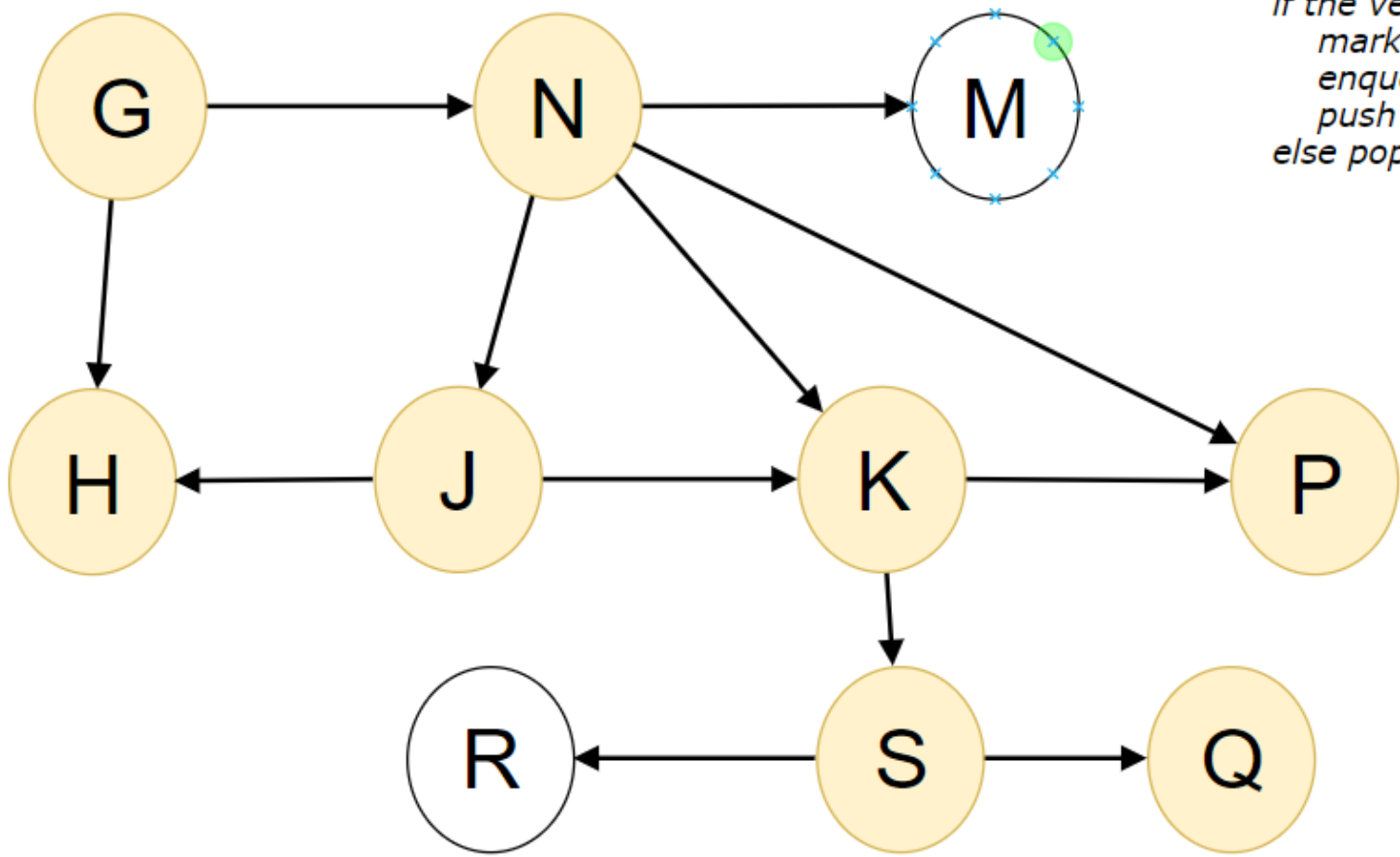
front

Traversal Order Queue

back



Vertex Stack



peek at a
if the vert
mark o
enqueu
push th
else pop t

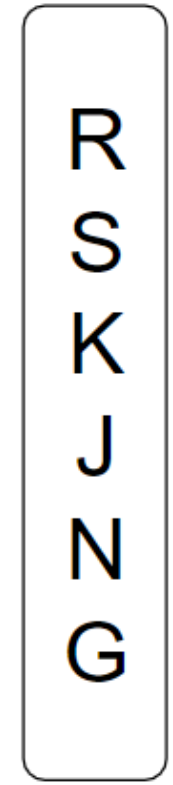
G H N J K P S Q

front

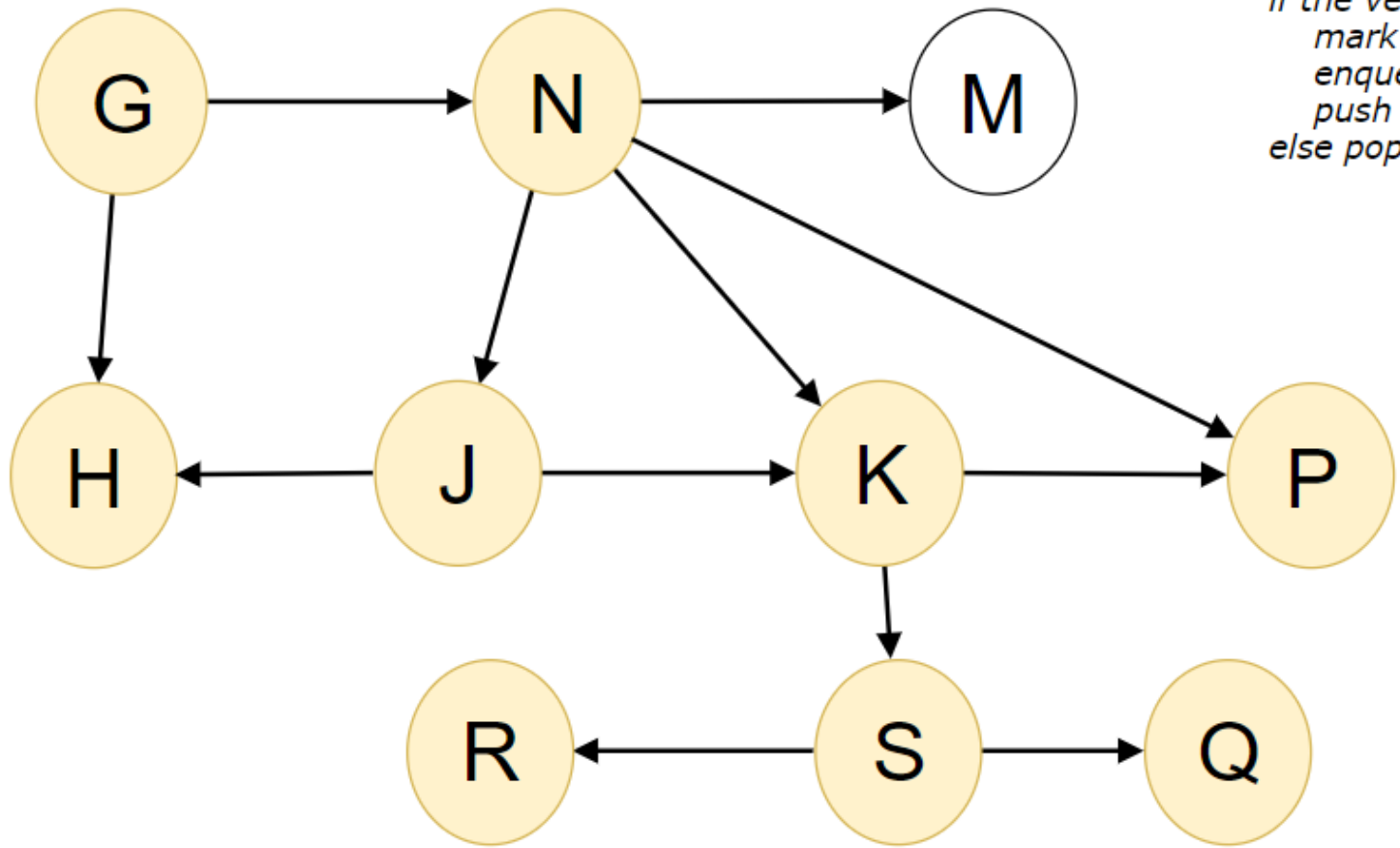
Traversal Order
Queue

back

peek at the stack to see a vertex (Q)
no unvisited neighbors, so pop from the stack



Vertex Stack



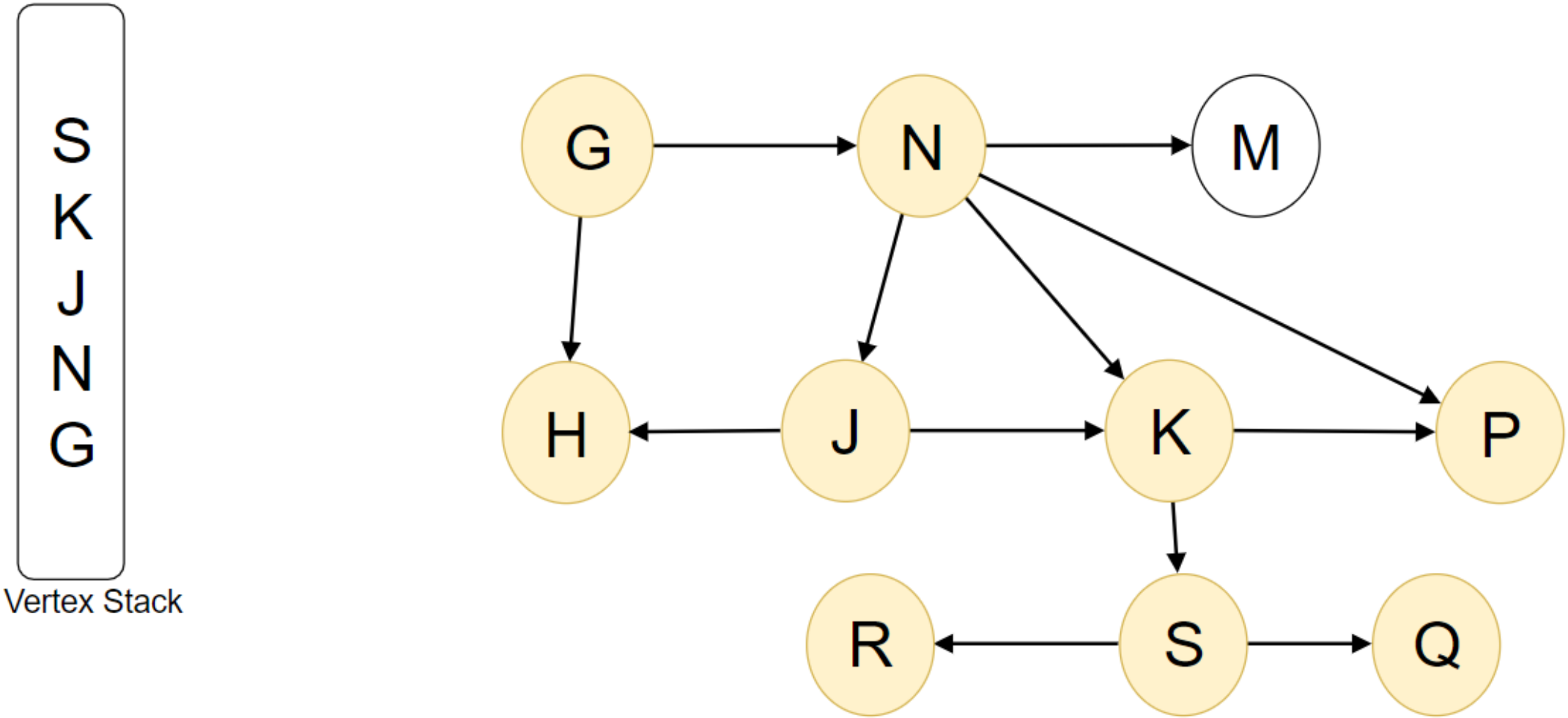
peek at a v
if the verte
mark on
enqueue
push the
else pop th

front

Traversal Order Queue

back

peek at the stack to see a vertex (S)
mark one of the unvisited neighbors as visited (R)
put that vertex (R) on the traversal order queue
push that vertex (R) onto the stack



G	H	N	J	K	P	S	Q	R

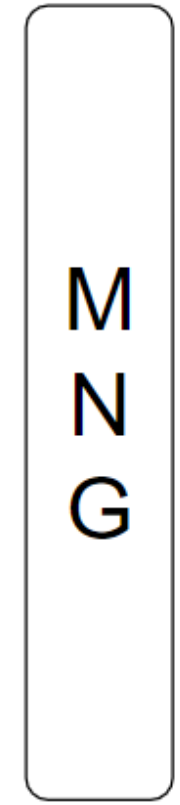
front

Traversal Order Queue

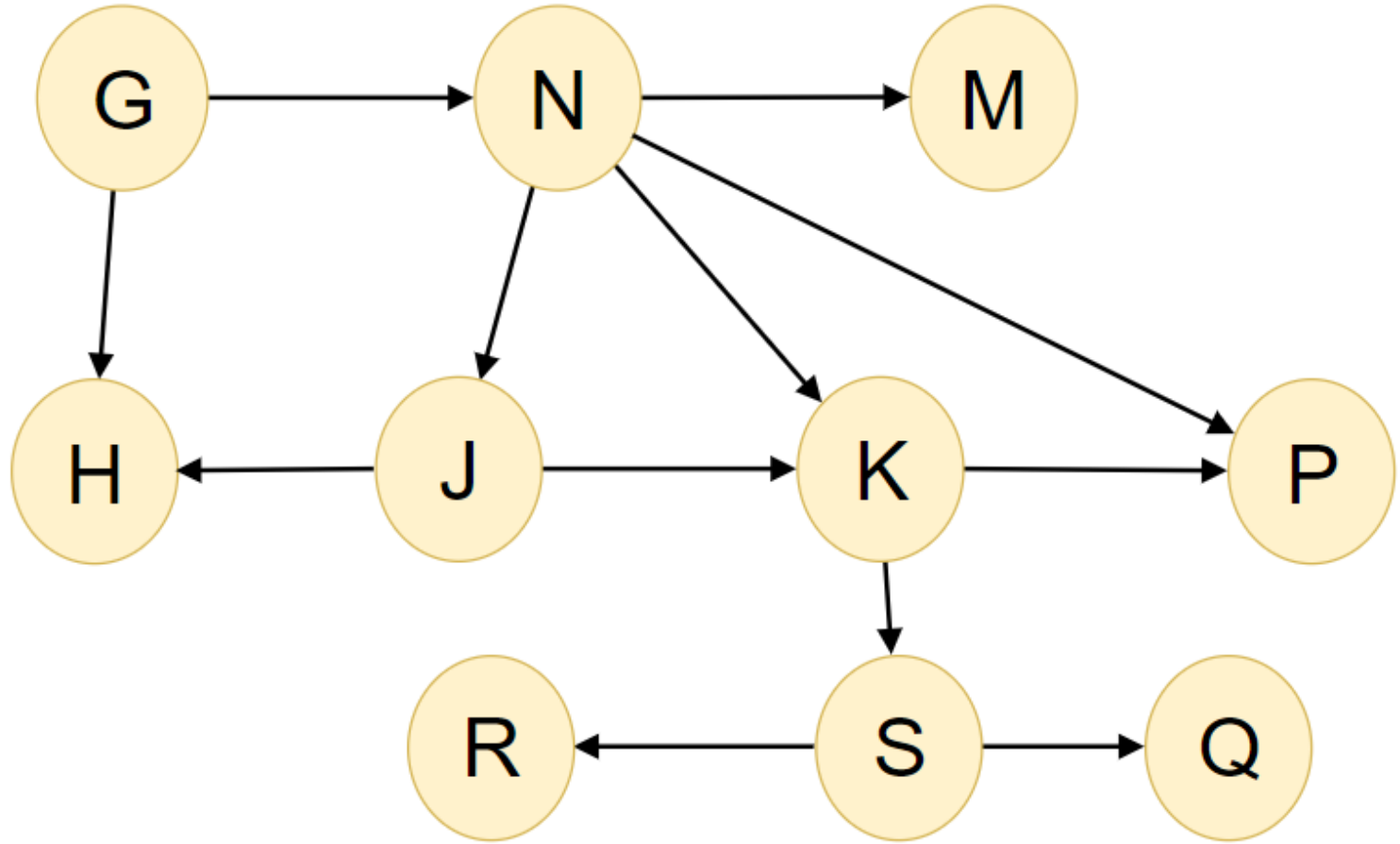
back

peek at the stack to see a vertex (R)
no unvisited neighbors, so pop from the stack

S, K, and J also popped because they have no unvisited neighbors



Vertex Stack



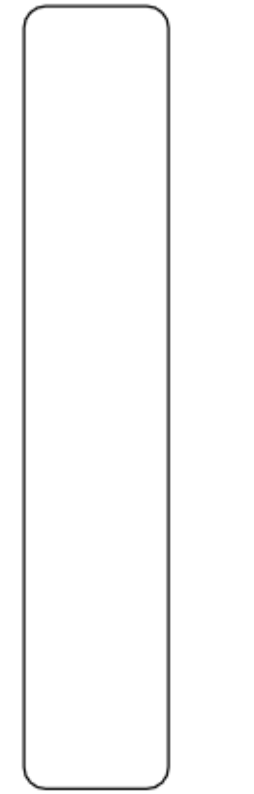
G	H	N	J	K	P	S	Q	R	M

front

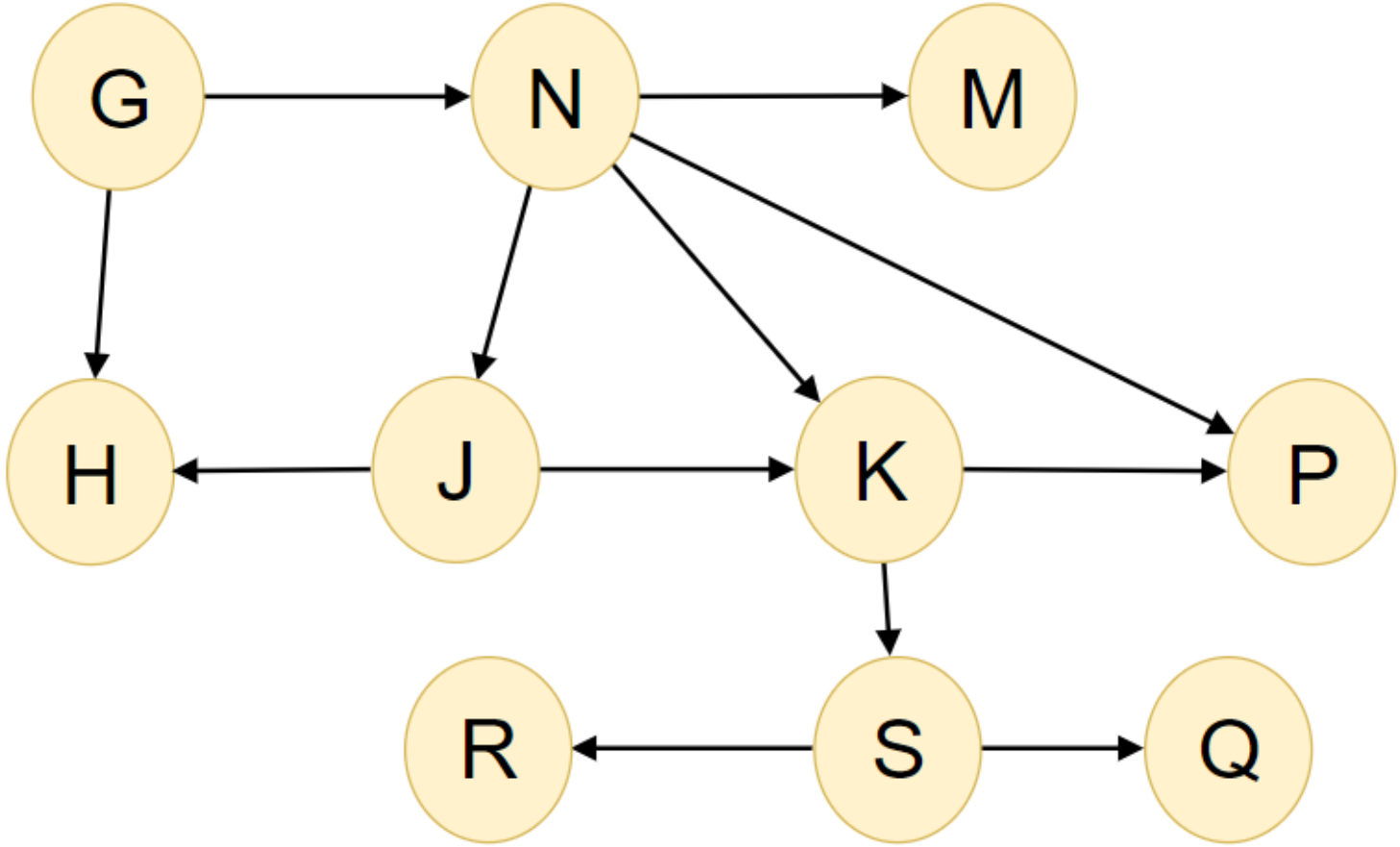
Traversal Order Queue

back

peek at the stack to see a vertex (N)
mark one of the unvisited neighbors as visited (M)
put that vertex (M) on the traversal order queue
push that vertex (M) onto the stack



Vertex Stack



G	H	N	J	K	P	S	Q	R	M

front

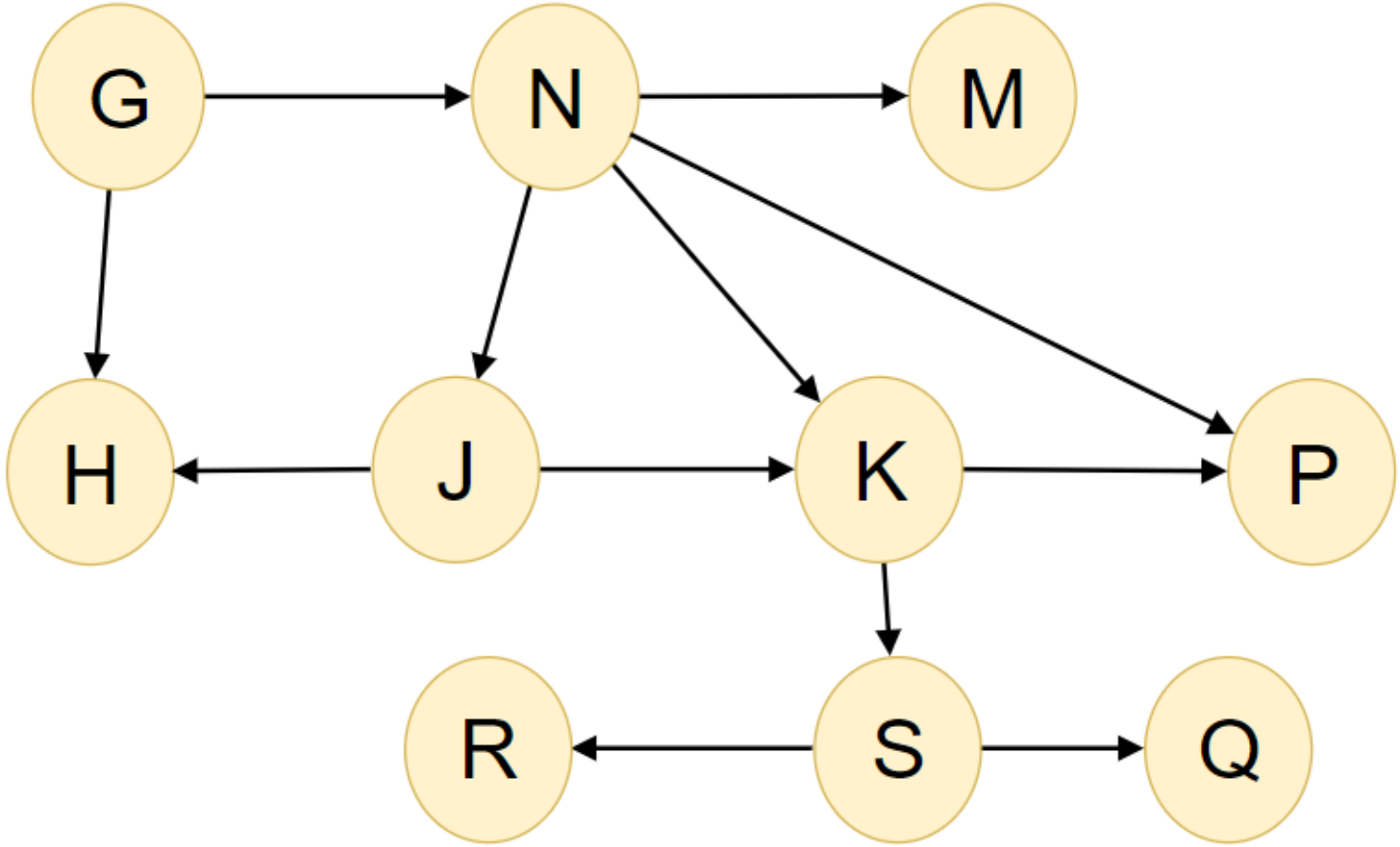
Traversal Order Queue

back

peek at the stack to see a vertex (M)
that vertex has no unvisited neighbors, so pop from the stack

this continues until the stack is empty (because all vertices have been visited)

Vertex Stack



G	H	N	J	K	P	S	Q	R	M

front

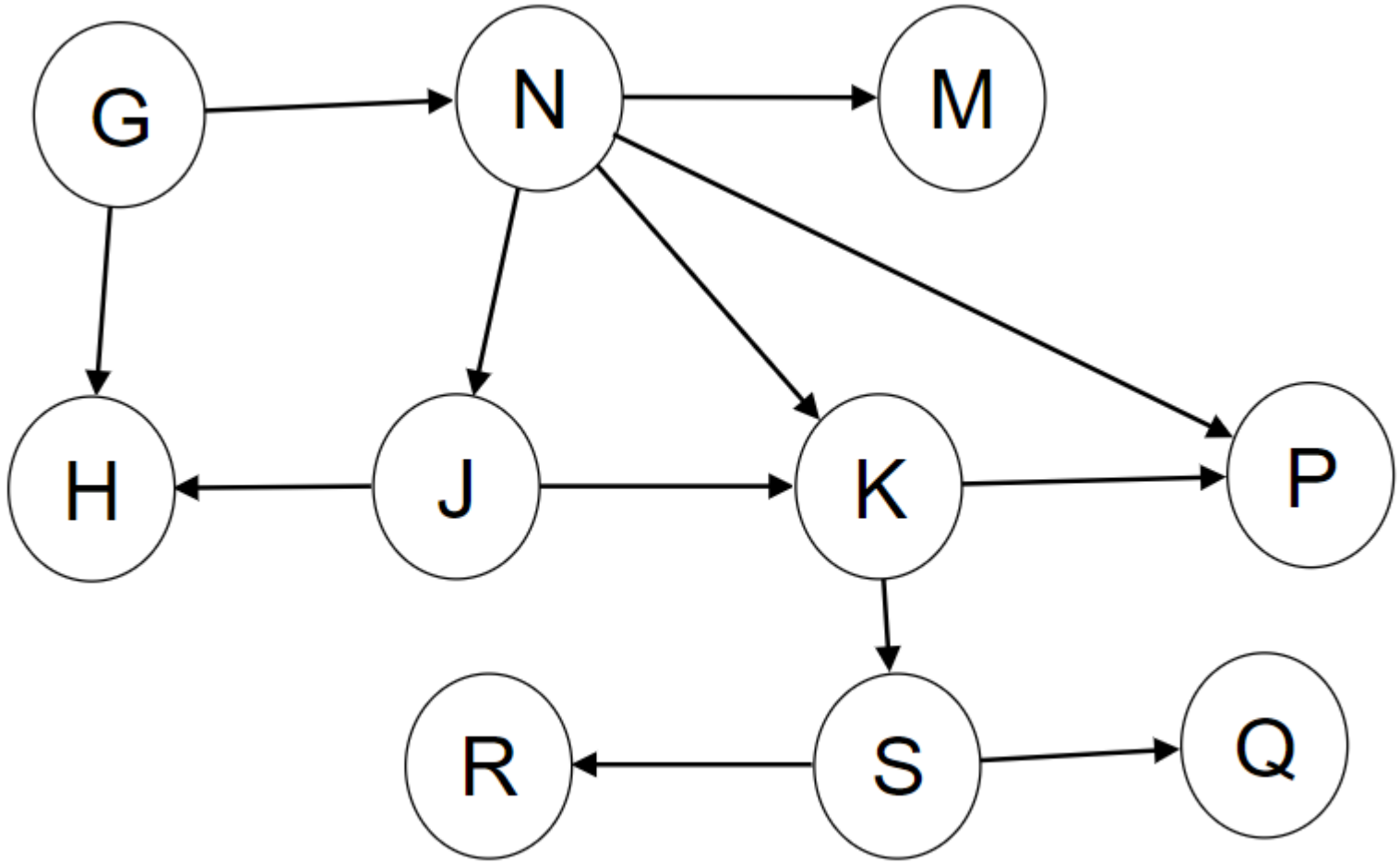
back

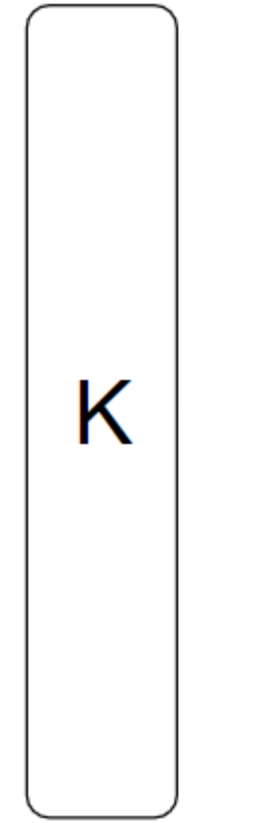
Traversal Order
Queue

the traversal order is: GHNJKPSQRM

Depth-First Example

- Trace a depth first traversal starting at vertex K.



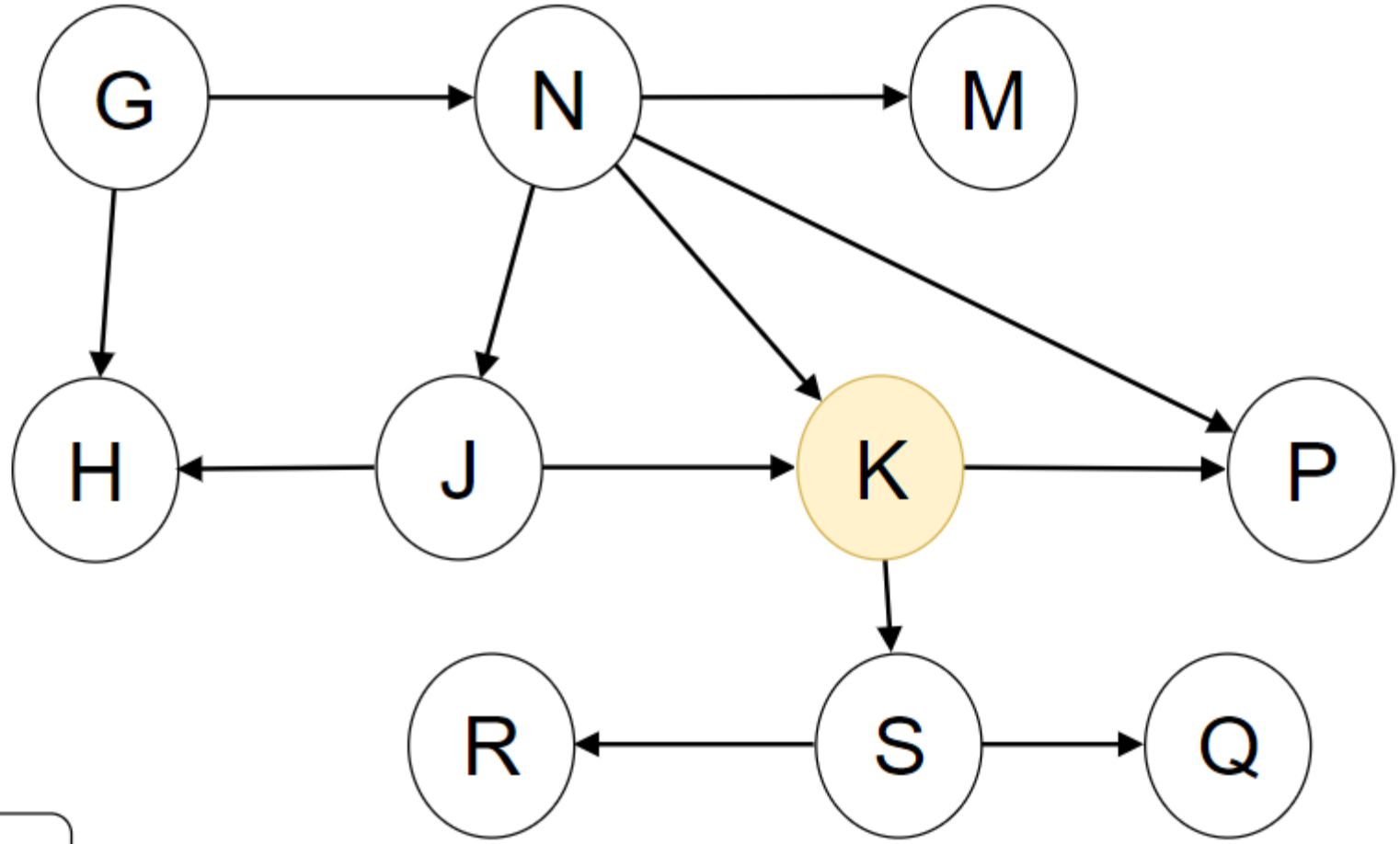


Vertex Stack

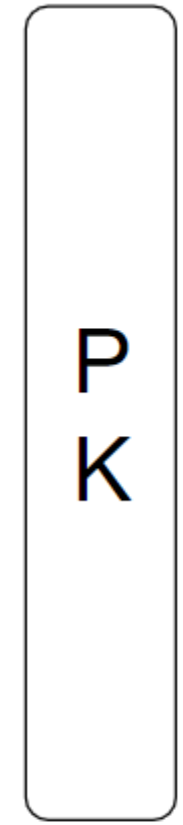
front

Traversal Order Queue

back



mark the origin as visited
enqueue the origin
push the origin on the stack

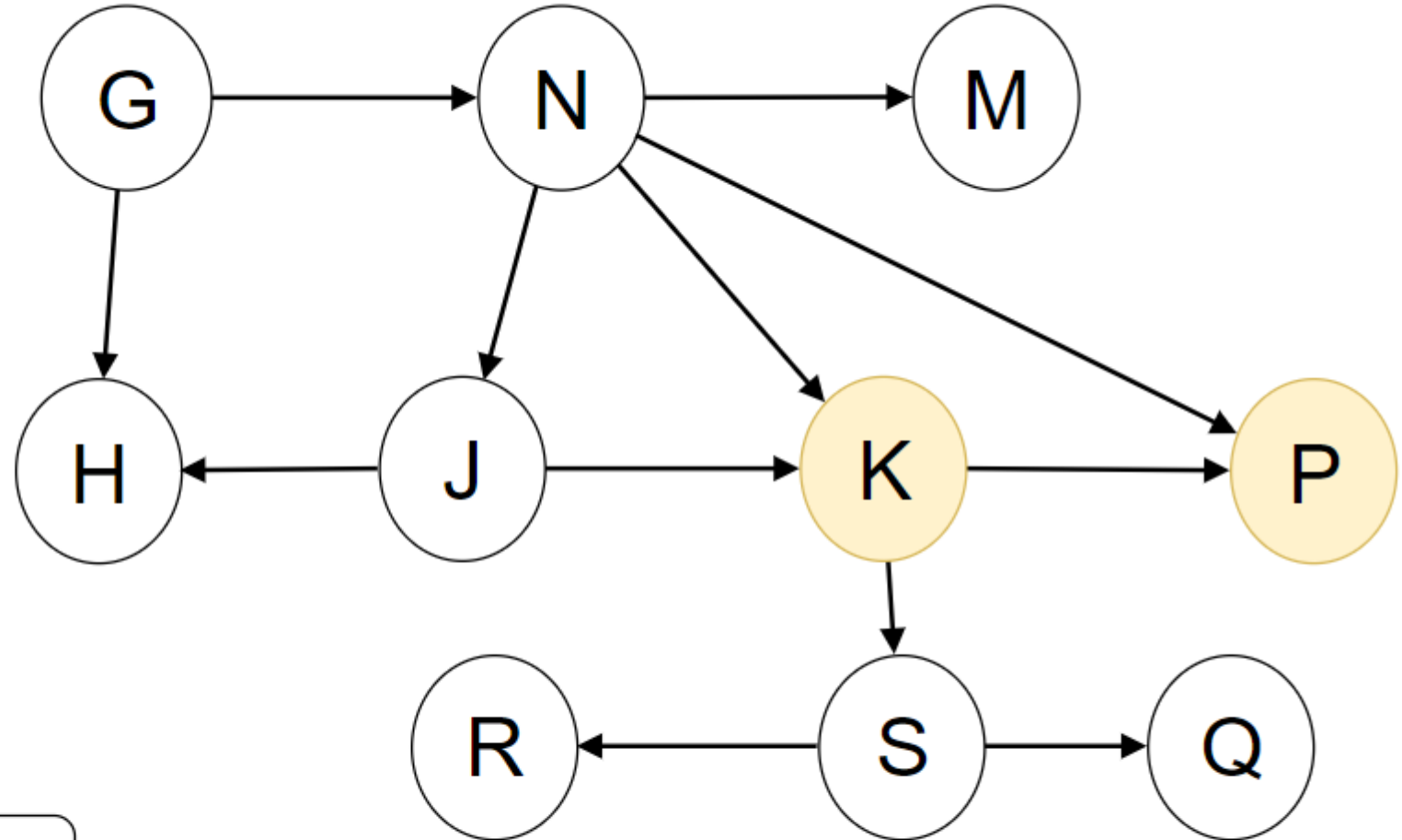


Vertex Stack

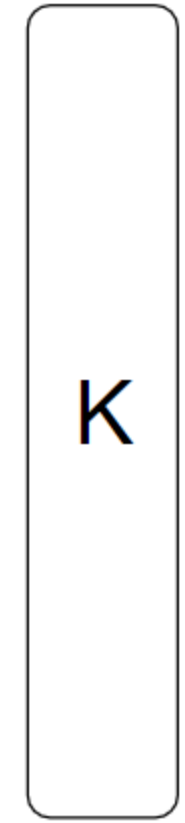
front

Traversal Order Queue

back



peek at the stack to see a vertex (K)
mark one of the unvisited neighbors as visited (P)
put that vertex (P) on the traversal order queue
push that vertex (P) onto the stack

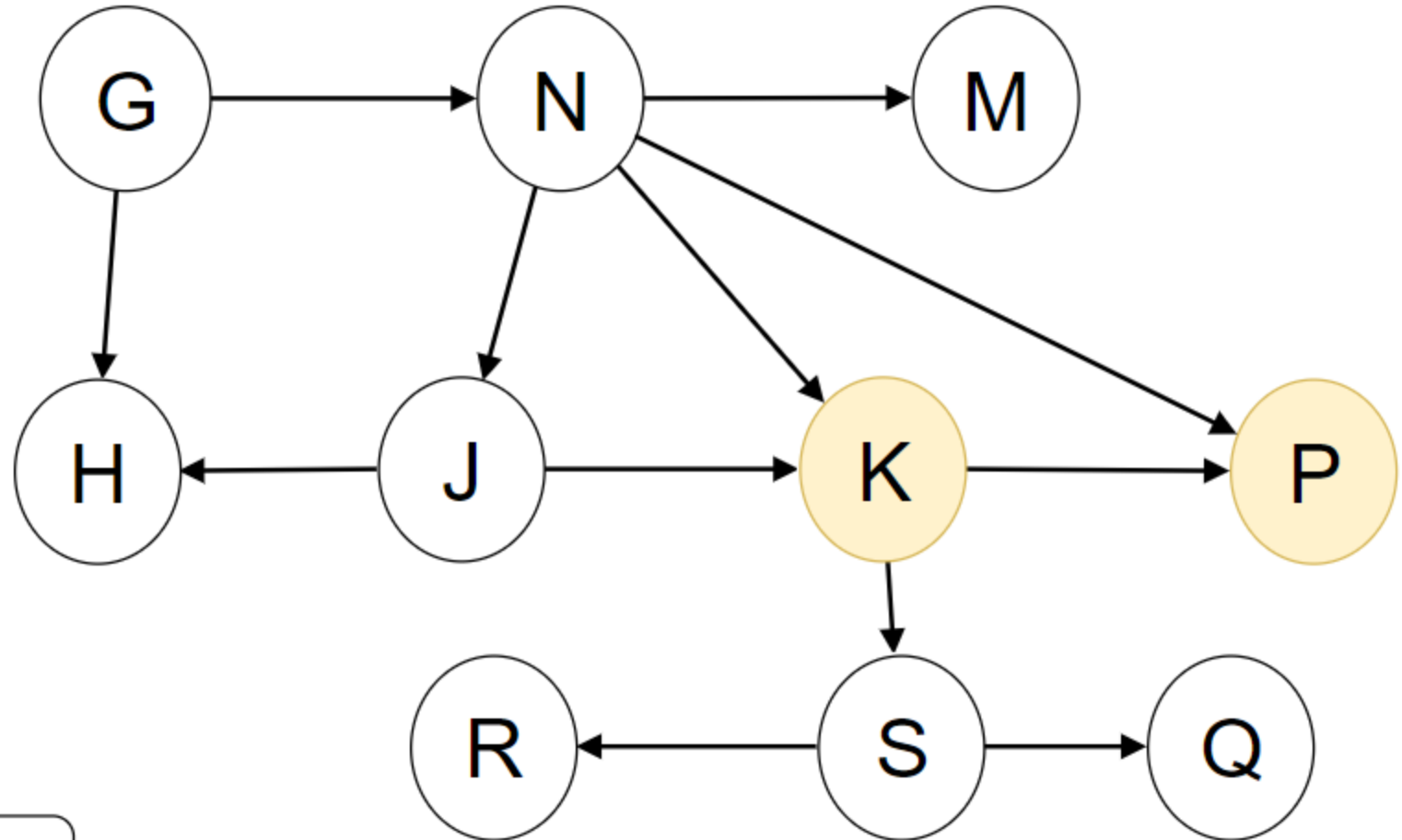


Vertex Stack

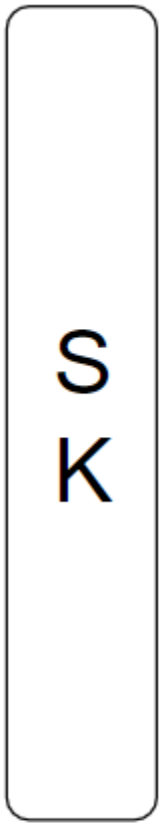
front

Traversal Order Queue

back

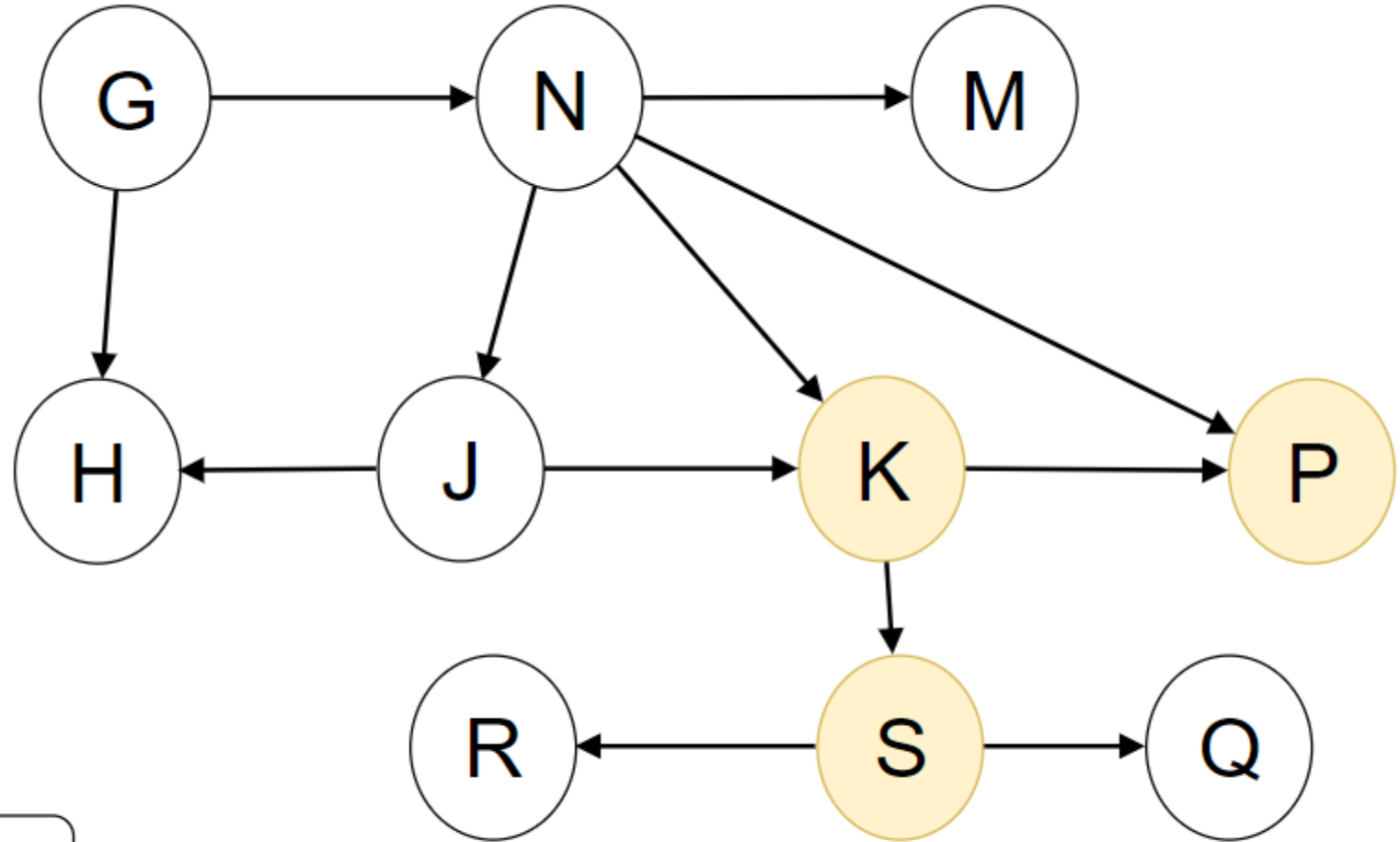


peek at the stack to see a vertex (P)
no unvisited neighbors, so pop from the stack

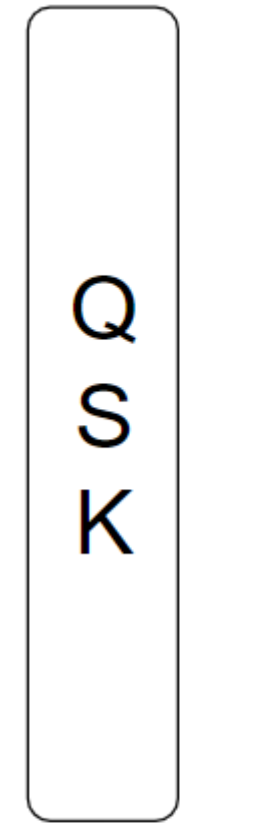


Vertex Stack

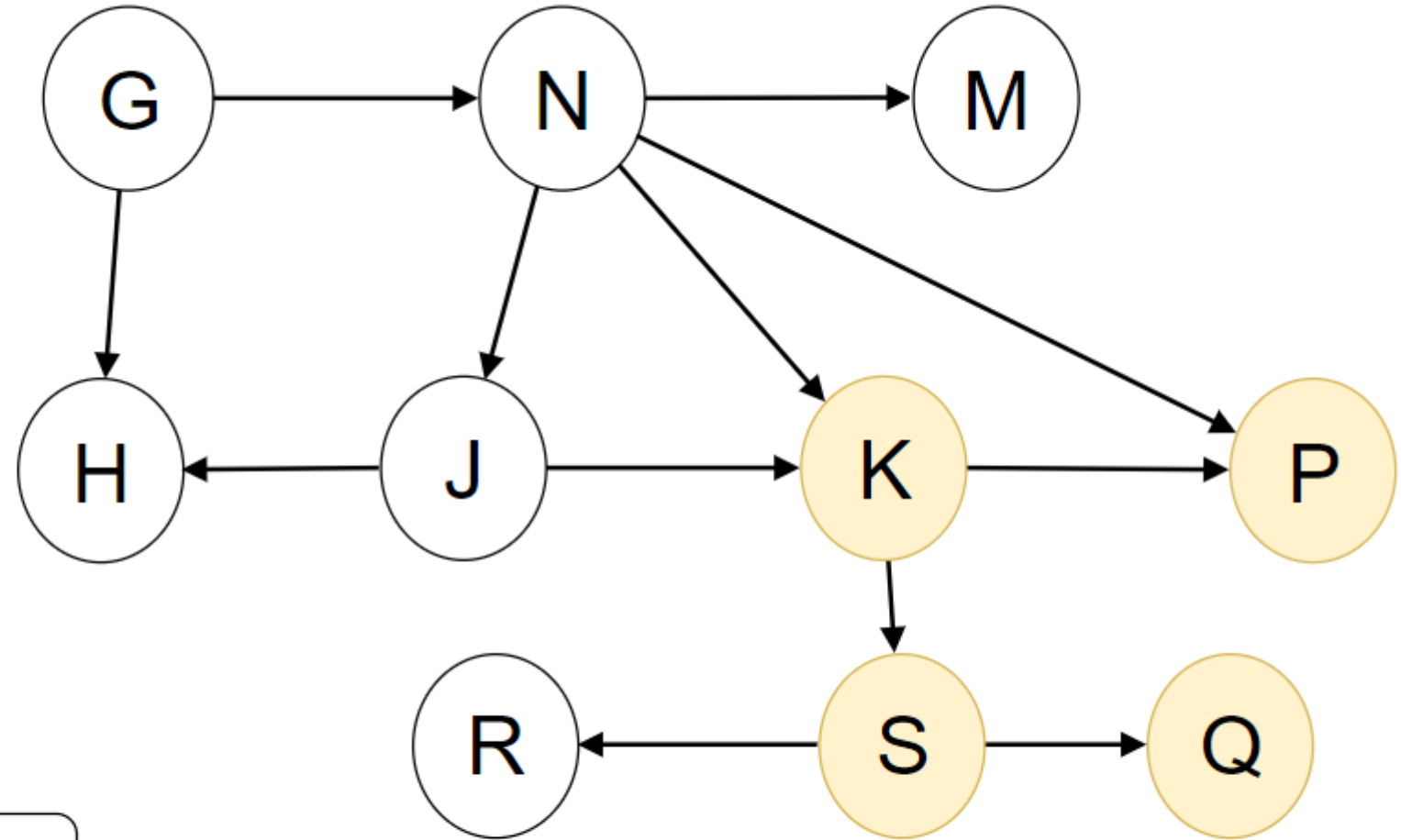
front Traversal Order Queue back



peek at the stack to see a vertex (K)
mark one of the unvisited neighbors as visited (S)
put that vertex (S) on the traversal order queue
push that vertex (S) onto the stack

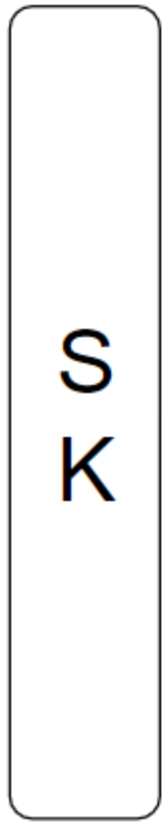


Vertex Stack

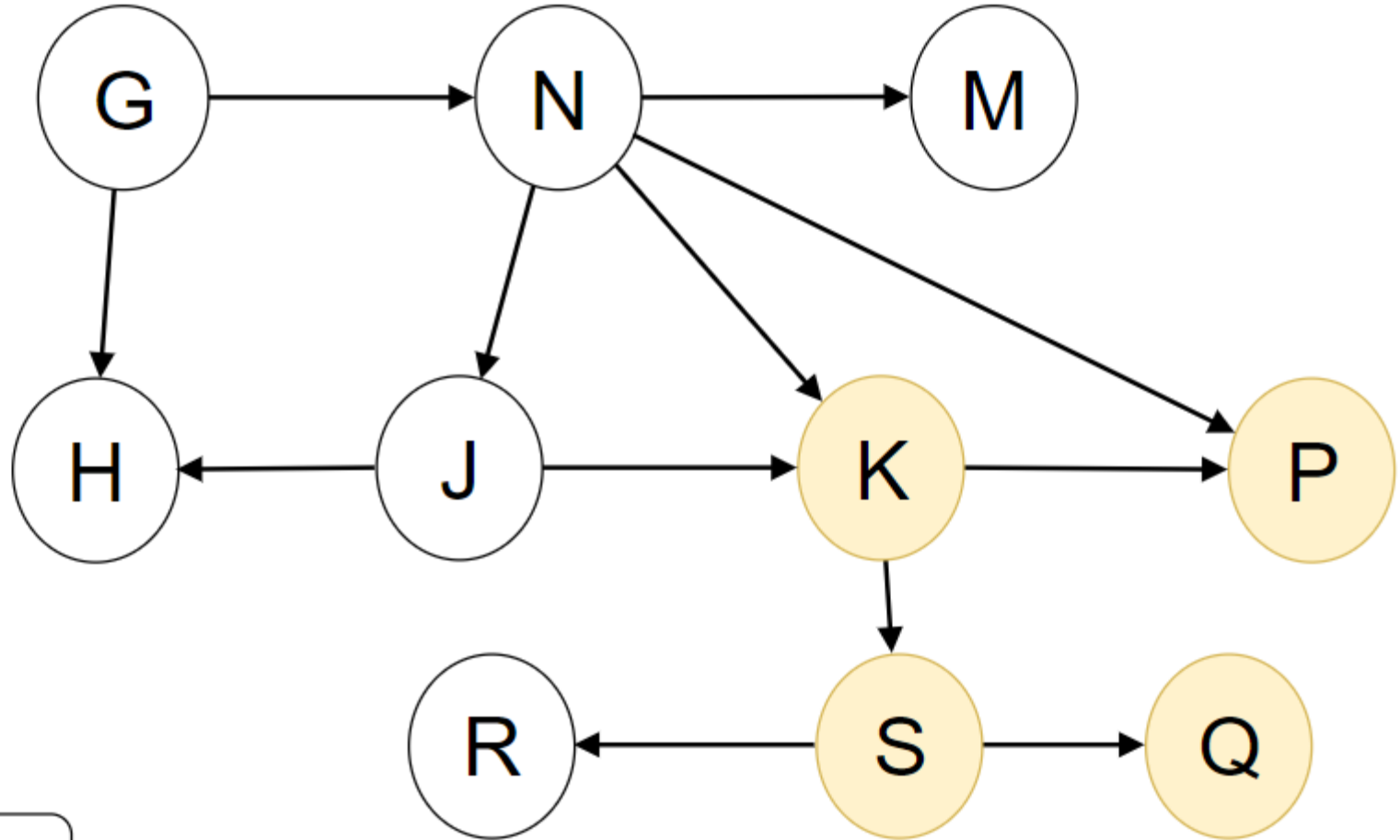


front Traversal Order Queue back

peek at the stack to see a vertex (S)
mark one of the unvisited neighbors as visited (Q)
put that vertex (Q) on the traversal order queue
push that vertex (Q) onto the stack



Vertex Stack

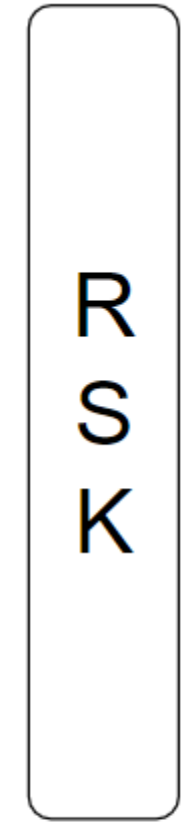


front

Traversal Order
Queue

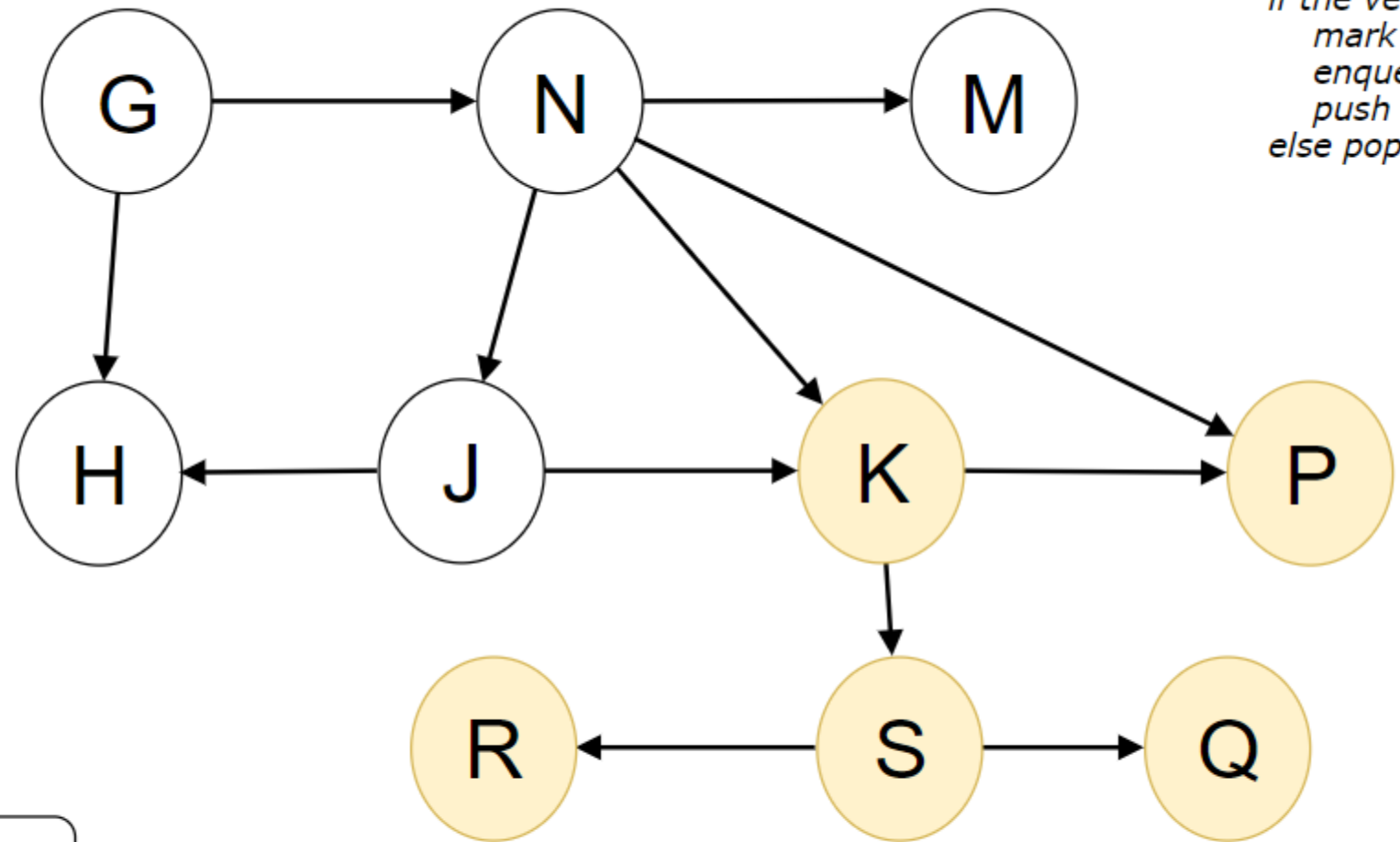
back

peek at the stack to see a vertex (Q)
no unvisited neighbors, so pop from the stack



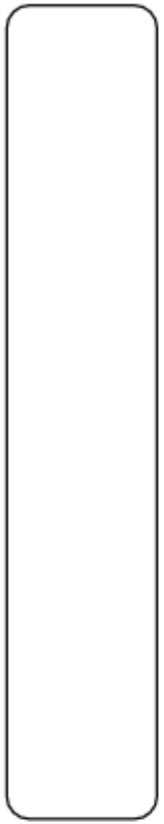
Vertex Stack

front Traversal Order Queue back

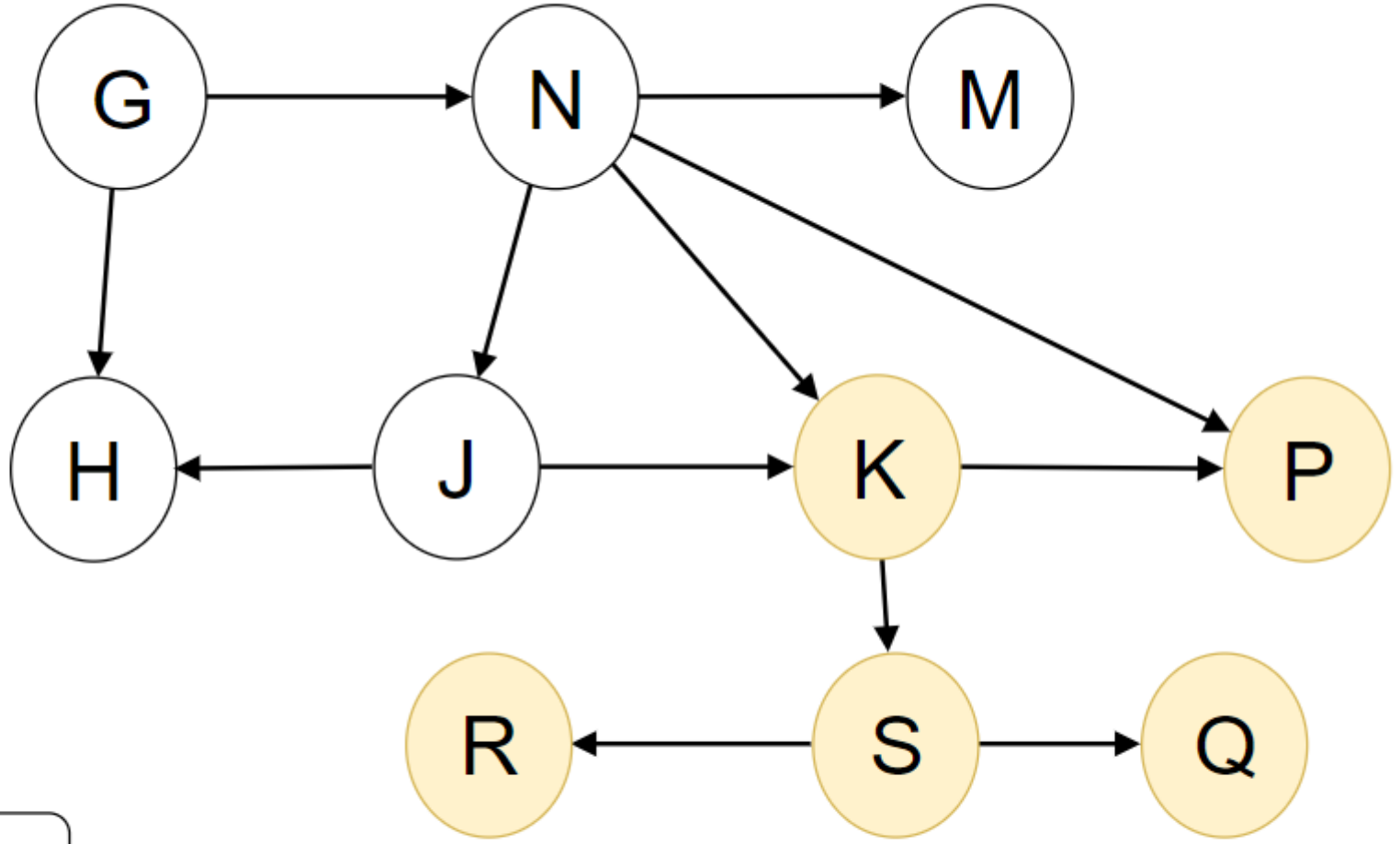


peek at a v
if the verte
mark on
enqueue
push the
else pop th

peek at the stack to see a vertex (S)
mark one of the unvisited neighbors as visited (R)
put that vertex (R) on the traversal order queue
push that vertex RS) onto the stack



Vertex Stack



K P S Q R

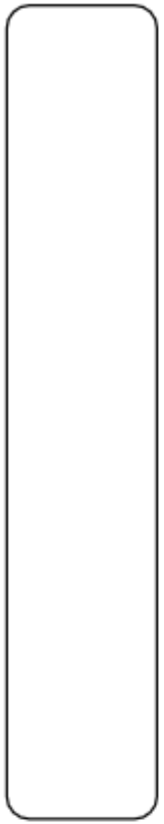
front

back

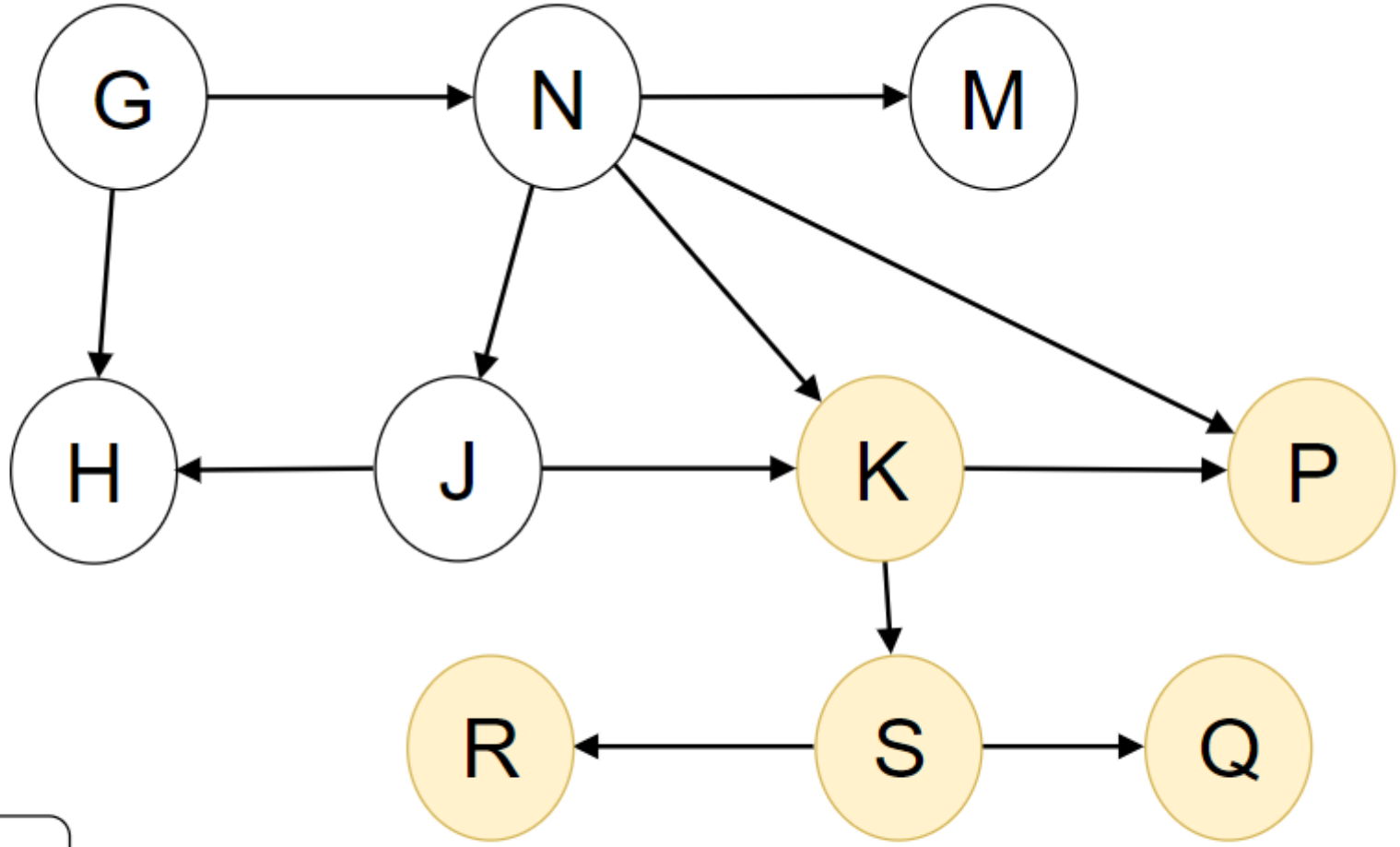
Traversal Order Queue

peek at the stack to see a vertex (R)
that vertex has no unvisited neighbors, so pop from the stack

this continues until the stack is empty (because all vertices have been visited)



Vertex Stack



K	P	S	Q	R

front

Traversal Order Queue

back

the traversal order is: KPSQR

note: not all vertices are in the traversal!

BREADTH AND DEPTH FIRST SEARCH

Breadth and Depth First

- Breadth-first visits neighbors, neighbors of neighbors, etc.
- Depth-first goes as far down a path as possible., backtrack and go down again

Using Breadth-First Traversal

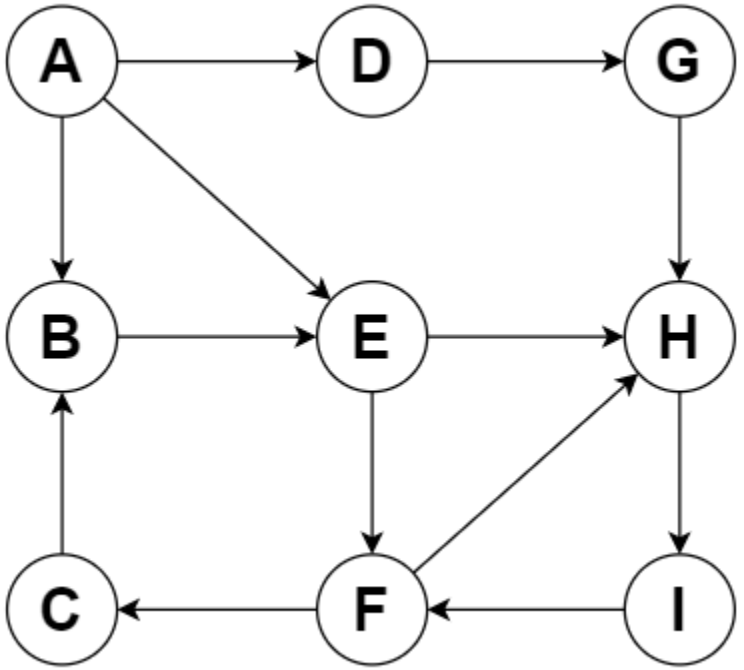
- Find shortest path or cheapest path
- GPS navigation systems
- Find neighbors in a peer to peer network (e.g., BitTorrent)
- Web crawlers (used in some search engines)
  - start from a source page and follow all links from that page
- Social networks
  - e.g., LinkedIn: 1st, 2nd, 3rd level connection
- Garbage collection algorithms

Using Depth-First Traversal

- Solving puzzles/mazes with only one (or more) solution
- Finding connectivity or connected components
- Cycle detection
- Find minimum spanning tree
- Path finding
- Network flow algorithms
- Graph matching algorithms
- Topological sorting
  - Used for dependency resolution
  - Used for job scheduling

Practice

- Review a breadth-first and depth-first traversal of the graph below (from the textbook) starting at vertex E.



TOPOLOGICAL ORDER

Topological Ordering

- The *topological ordering* of a graph is an ordering of vertices such that a vertex *a* comes before vertex *b* in the order if there is a directed edge from *a->b*
- Also called a *topological sort*

Topological Ordering

- Topological ordering is useful for finding valid sequences of tasks or instructions to be performed
- Example:
  - vertices of a graph are tasks and edges represent precedence
  - A -> B means task A must be completed before task B
  - The topological ordering gives you a valid order for executing all tasks in a way that satisfies all precedence rules

Topological Ordering

- Only directed, acyclic graphs (DAGs) can have a topological ordering
  - Why?

- A graph can have more than one topological ordering.
- Similar to the traversals, the order of visited vertices is not part of the algorithm.
  - We'll use alphabetic ordering.

- **All** vertices are part of a topological ordering- even if they are disconnected from the other parts of a graph.

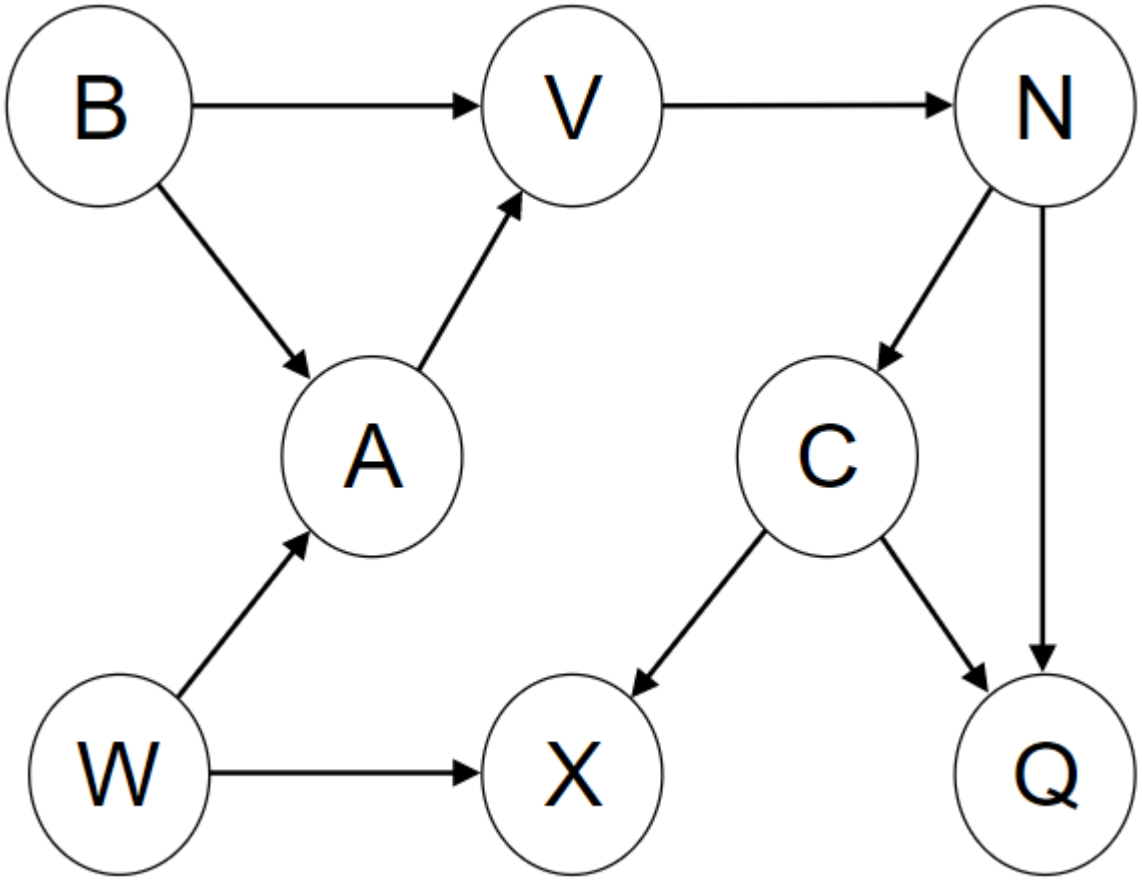
Topological Order Algorithm: Neighbors

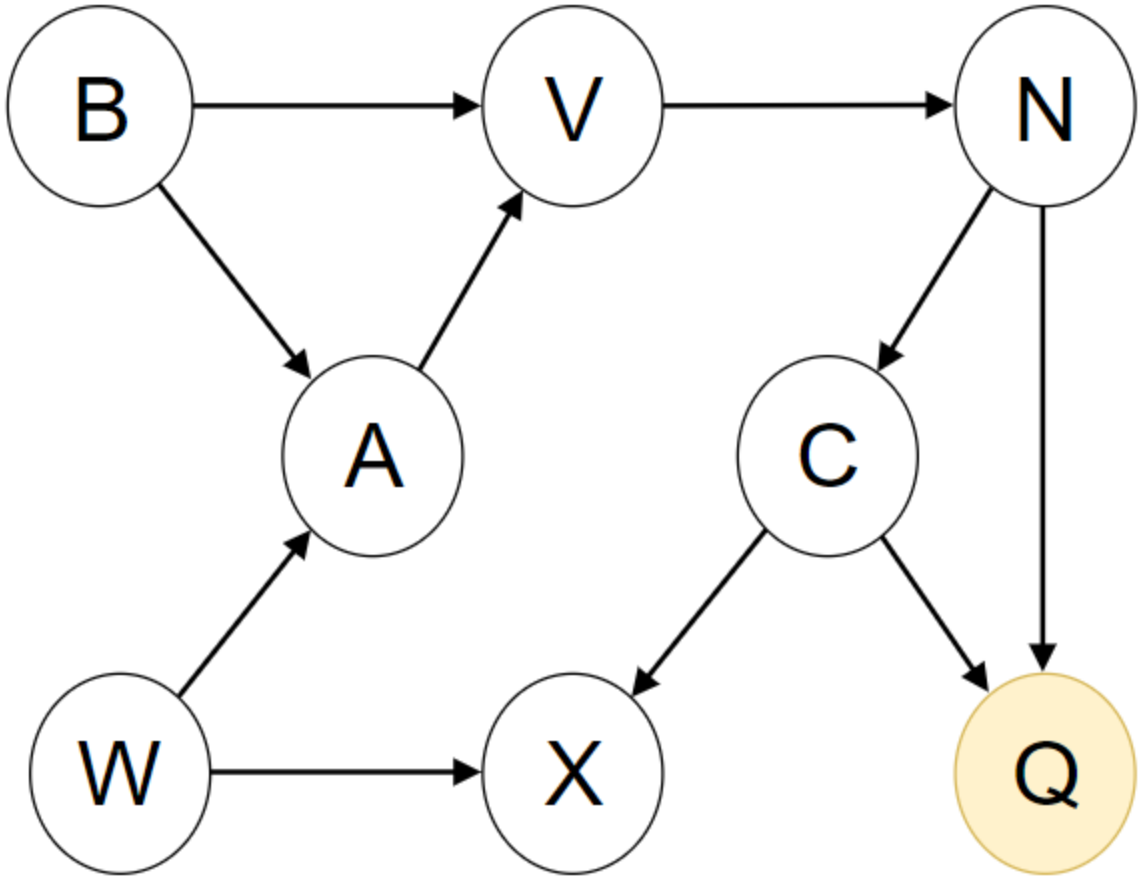
- Find the topological order by looking at adjacent vertices.
  - Remember: in a directed graph, a is adjacent to b if b->a
- Algorithm:
  - pick an unvisited vertex with no unvisited neighbors
    - this means an unvisited vertex whose neighbors have all been visited
    - this means no edges coming ***out of*** the vertex and into an unvisited neighbor
  - mark this vertex as visited
  - push the vertex onto the vertexStack
  - after all vertices are visited, pop from the stack for the order (order is top to bottom)

Example: Topological Order by Neighbors

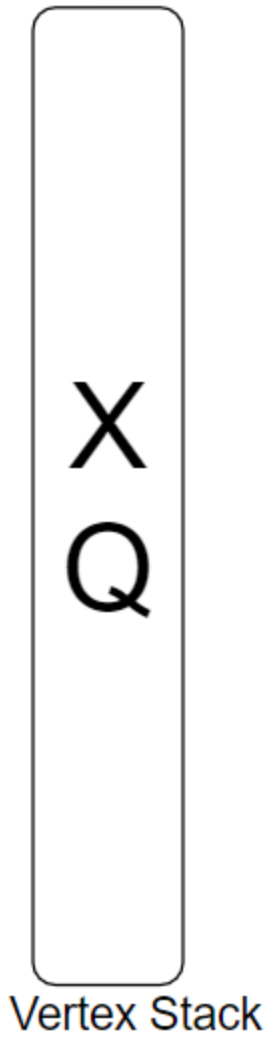
- Find a topological order of this DAG.

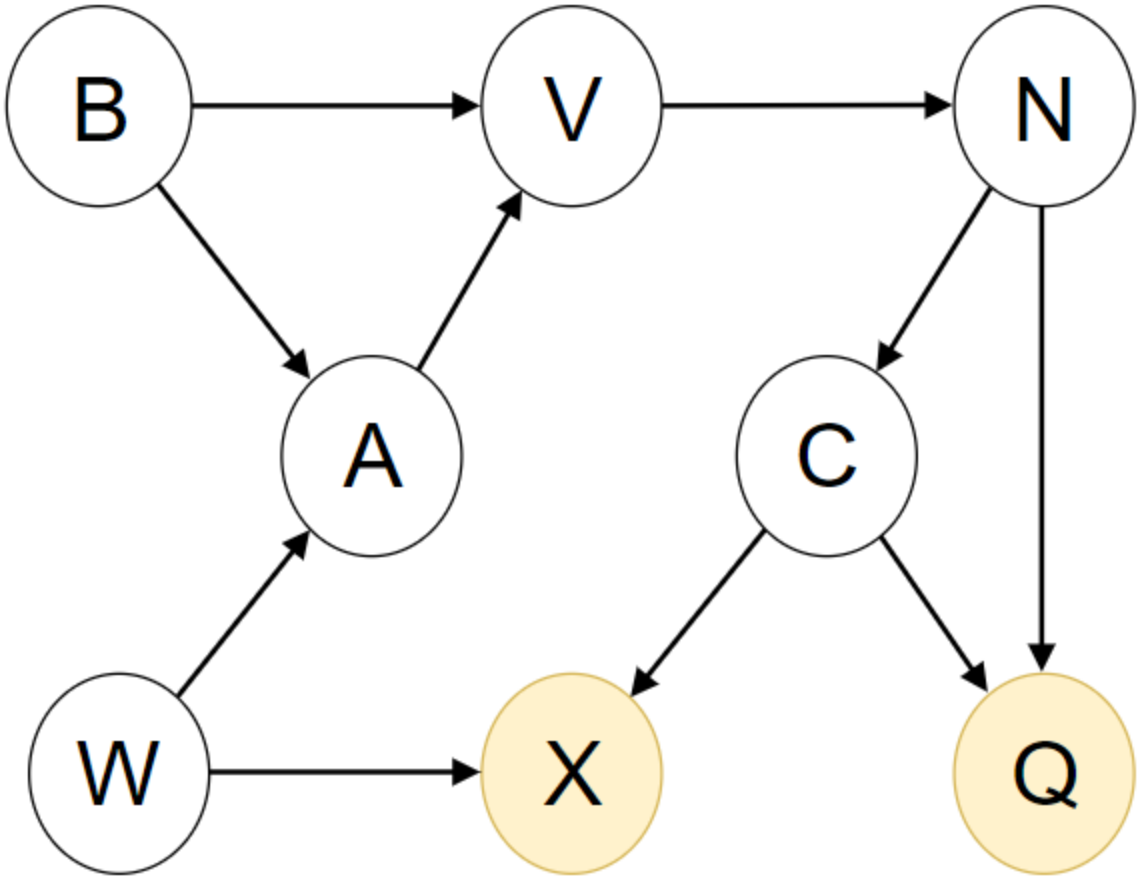
directed, acyclic graphs (DAG)



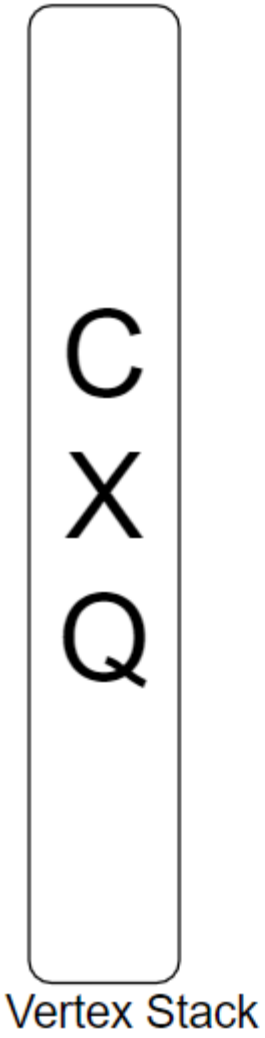


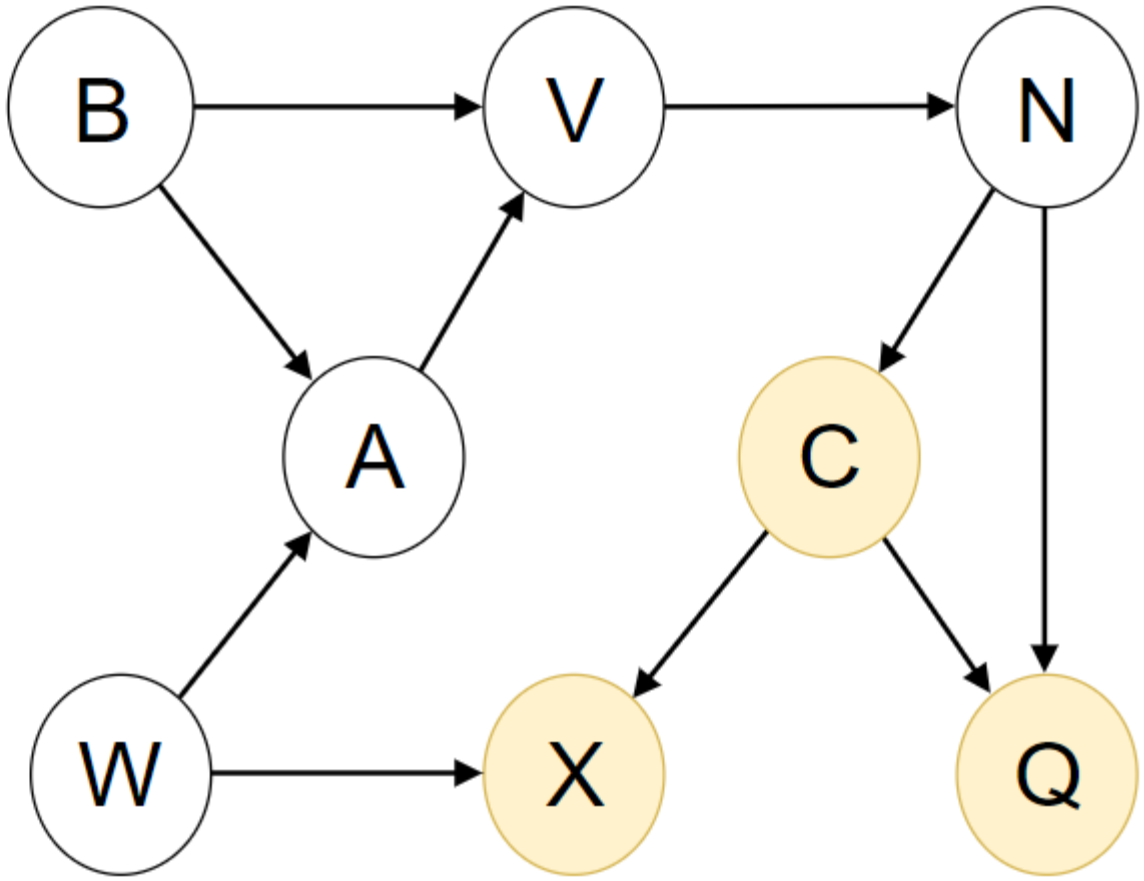
choose an unvisited vertex with no unvisited neighbors (Q or X; we choose Q)
mark visited and push on the stack



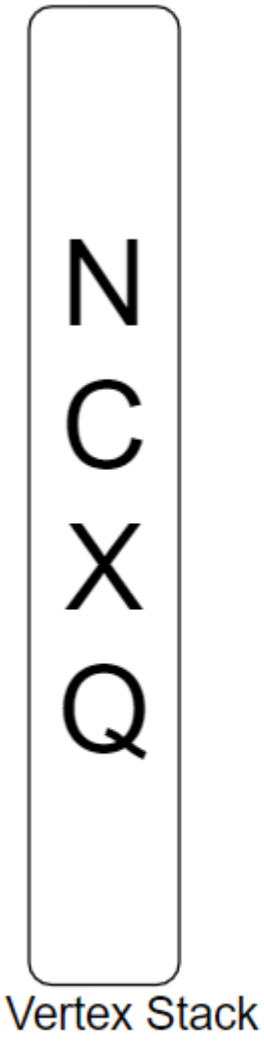


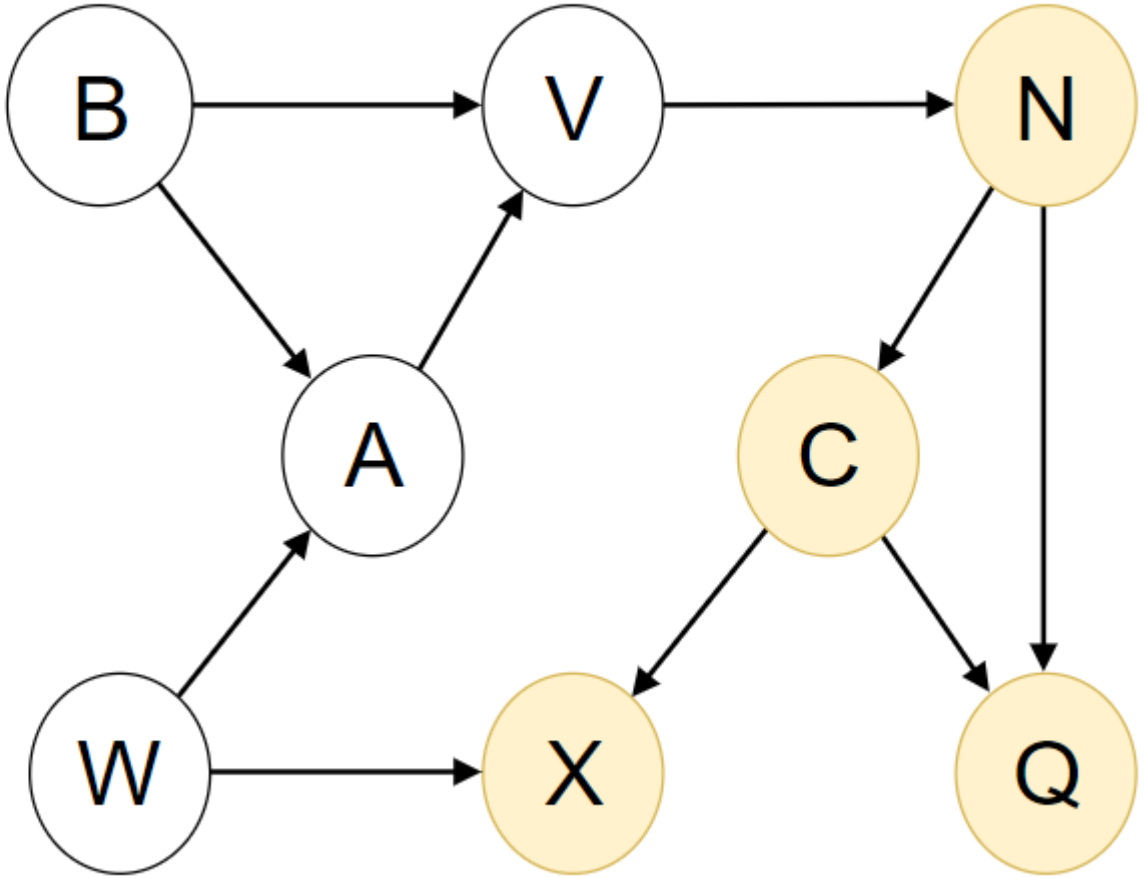
choose an unvisited vertex with no unvisited neighbors (X)
mark visited and push on the stack





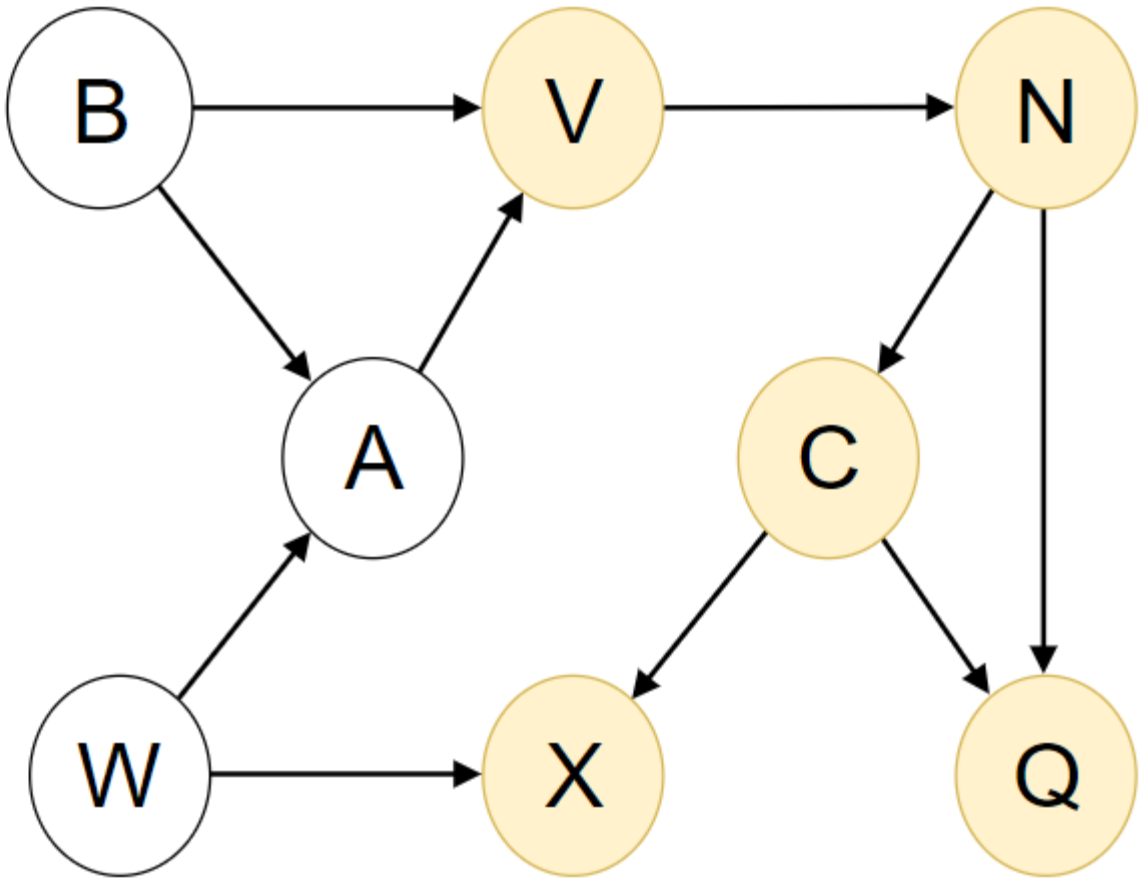
choose an unvisited vertex with no unvisited neighbors (C)
mark visited and push on the stack





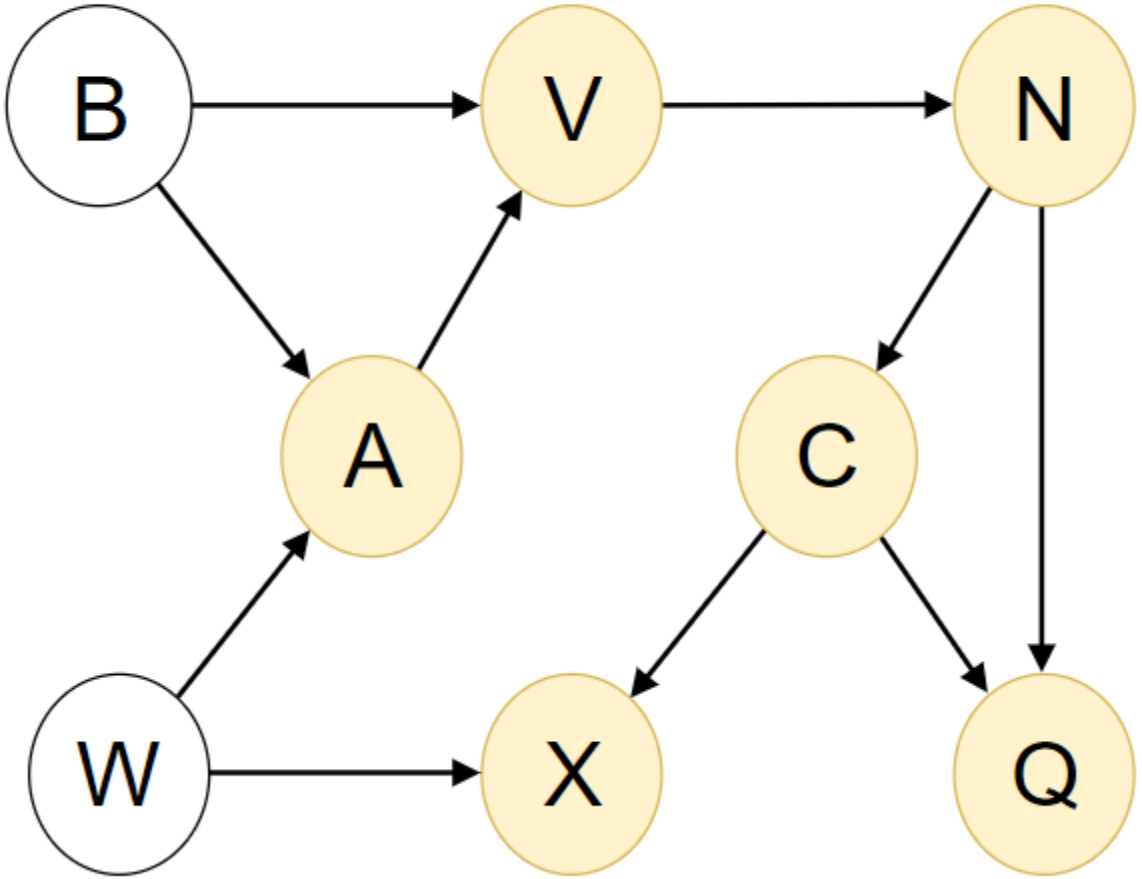
choose an unvisited vertex with no unvisited neighbors (N)
mark visited and push on the stack

Vertex Stack

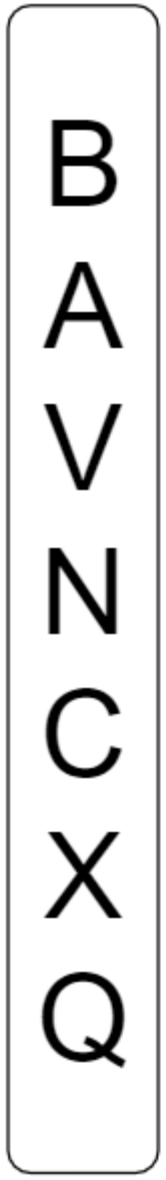


choose an unvisited vertex with no unvisited neighbors (V)
mark visited and push on the stack

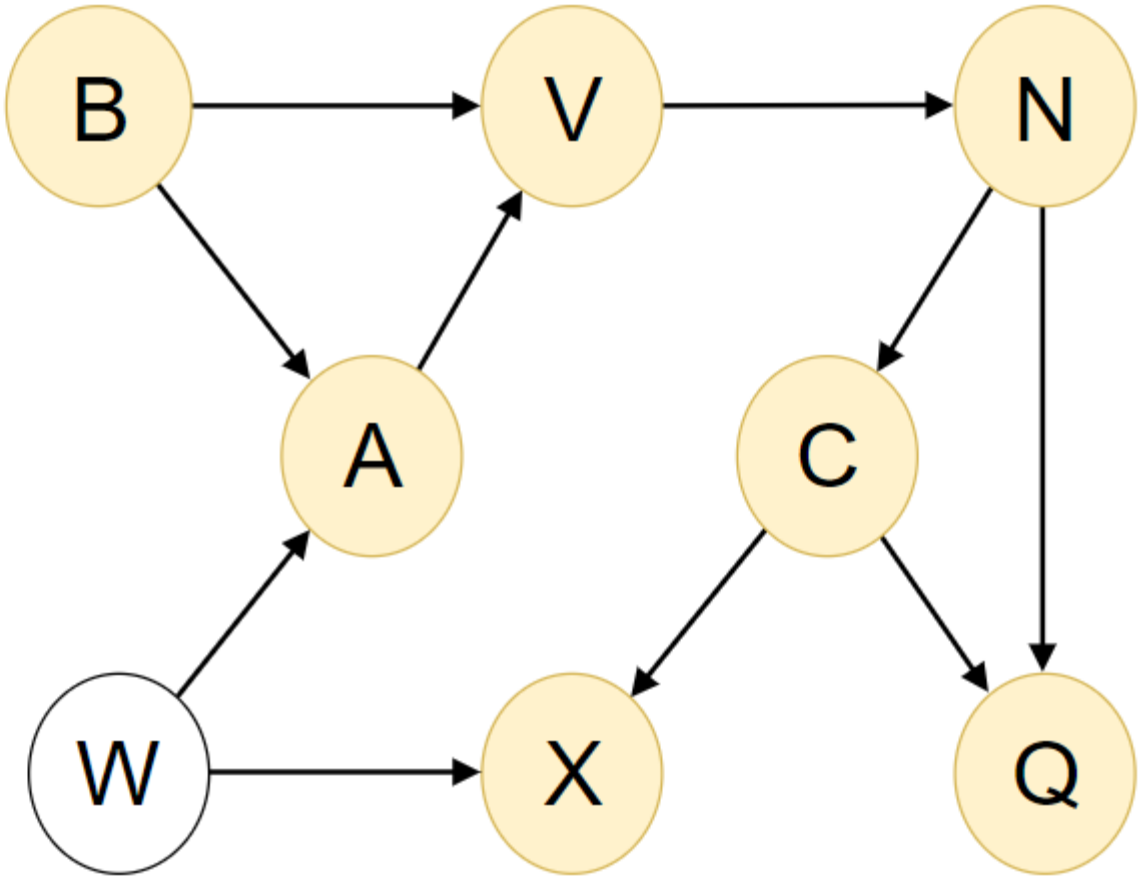
Vertex Stack



choose an unvisited vertex with no unvisited neighbors (A)
mark visited and push on the stack

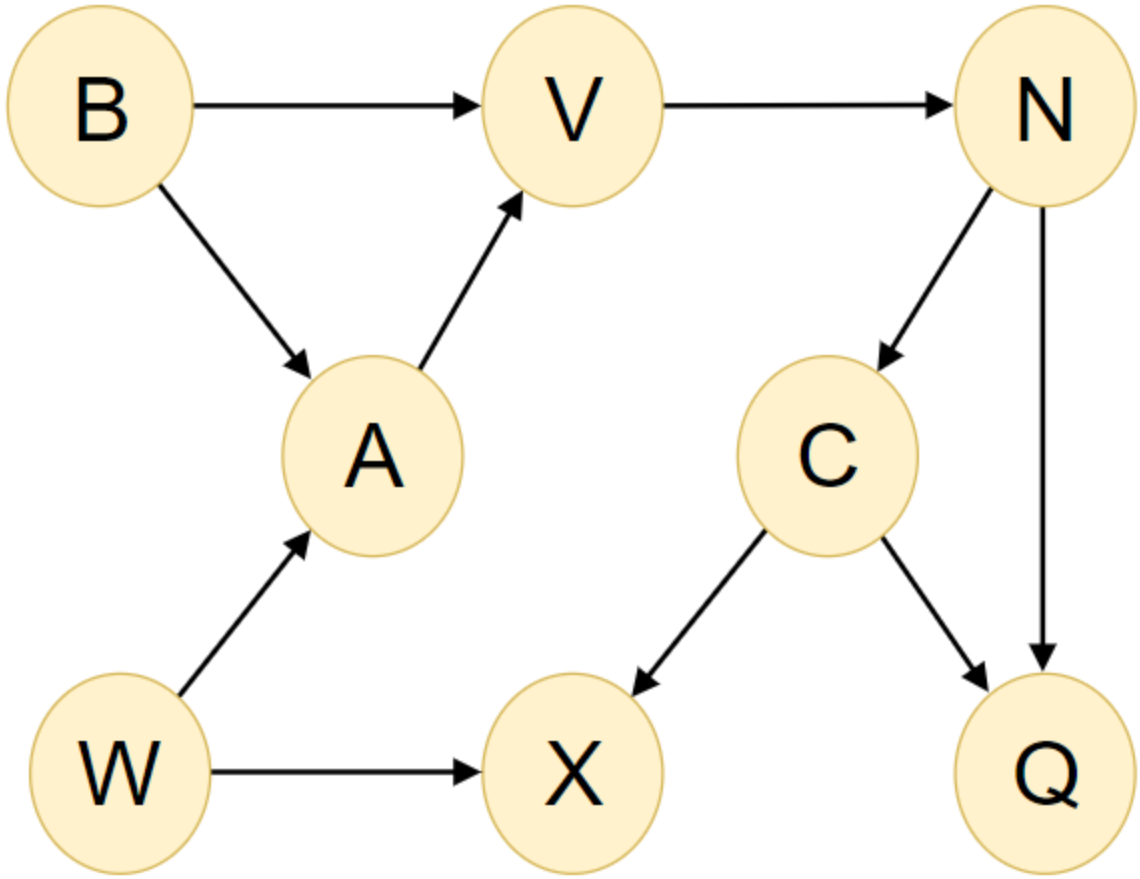


Vertex Stack



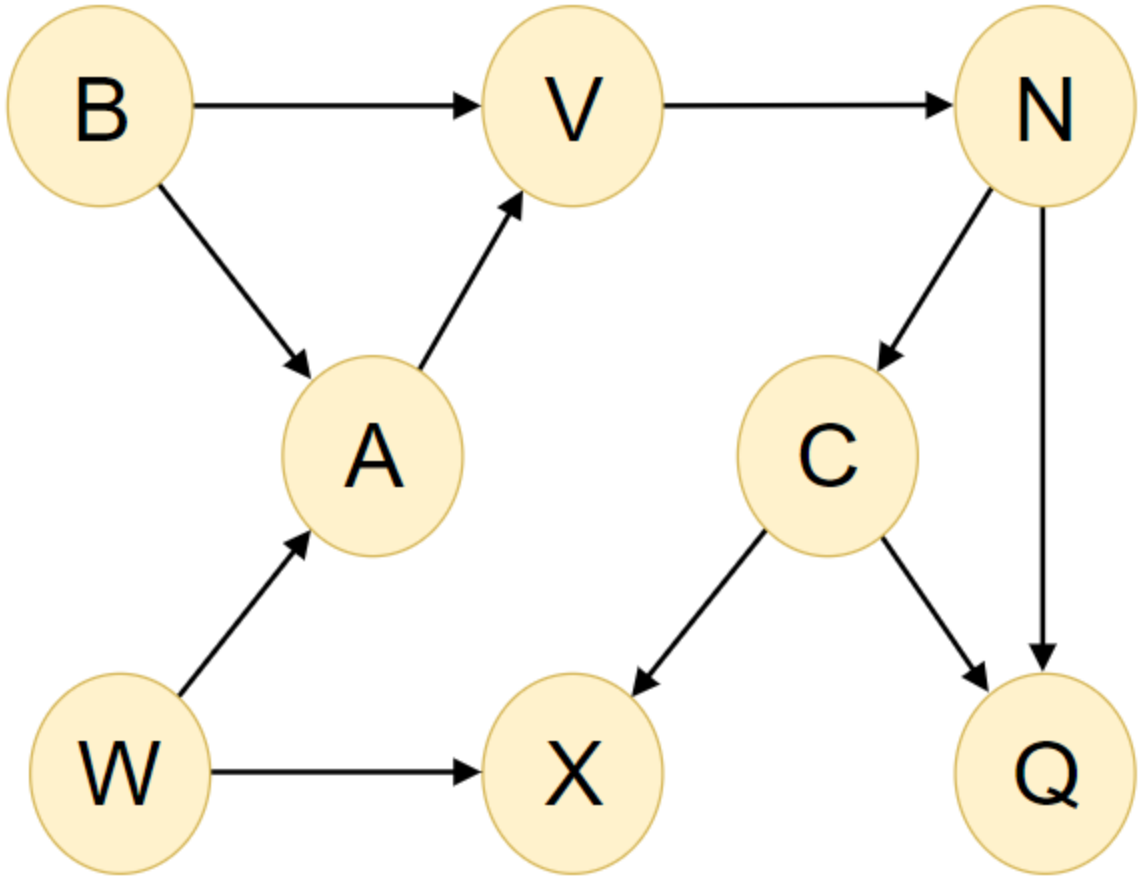
choose an unvisited vertex with no unvisited neighbors (B or W; we choose B)
mark visited and push on the stack

Vertex Stack



choose an unvisited vertex with no unvisited neighbors (W)
mark visited and push on the stack

Vertex Stack



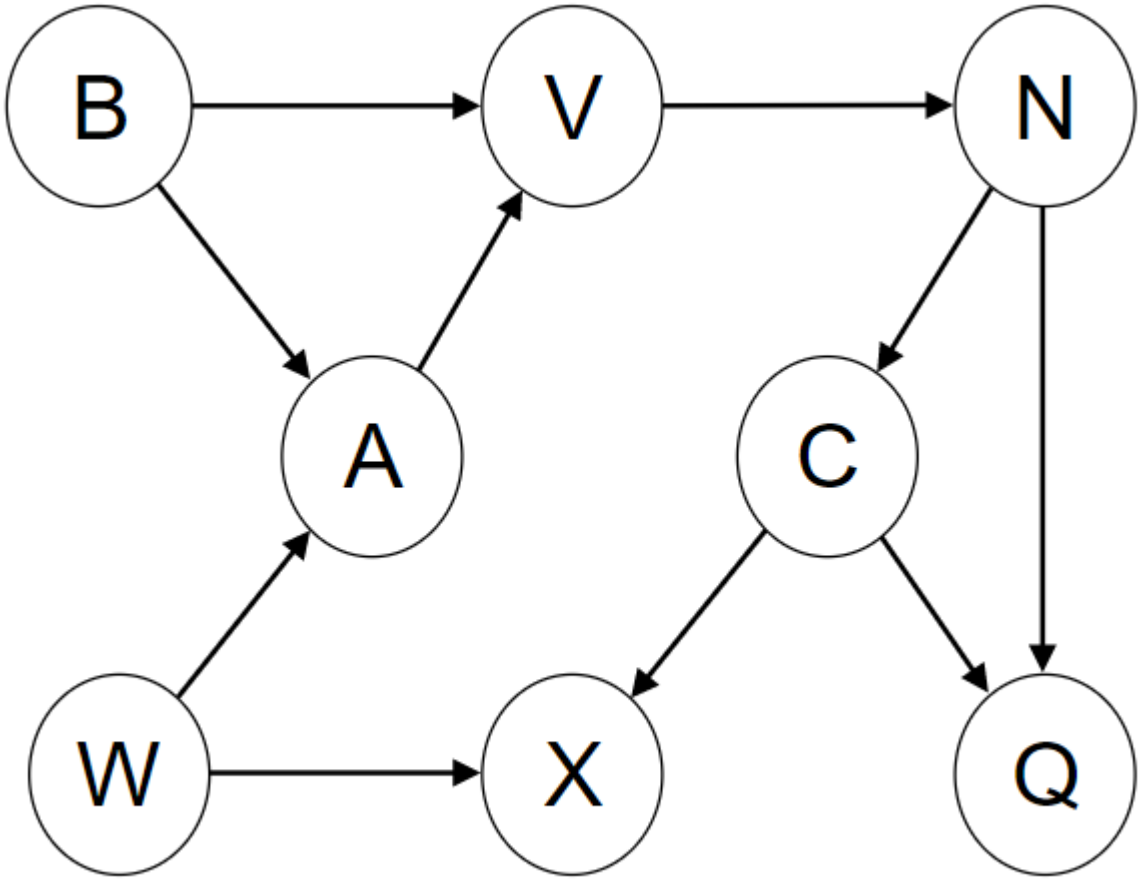
Topological Order: W B A V N C X Q
(top of stack to bottom)

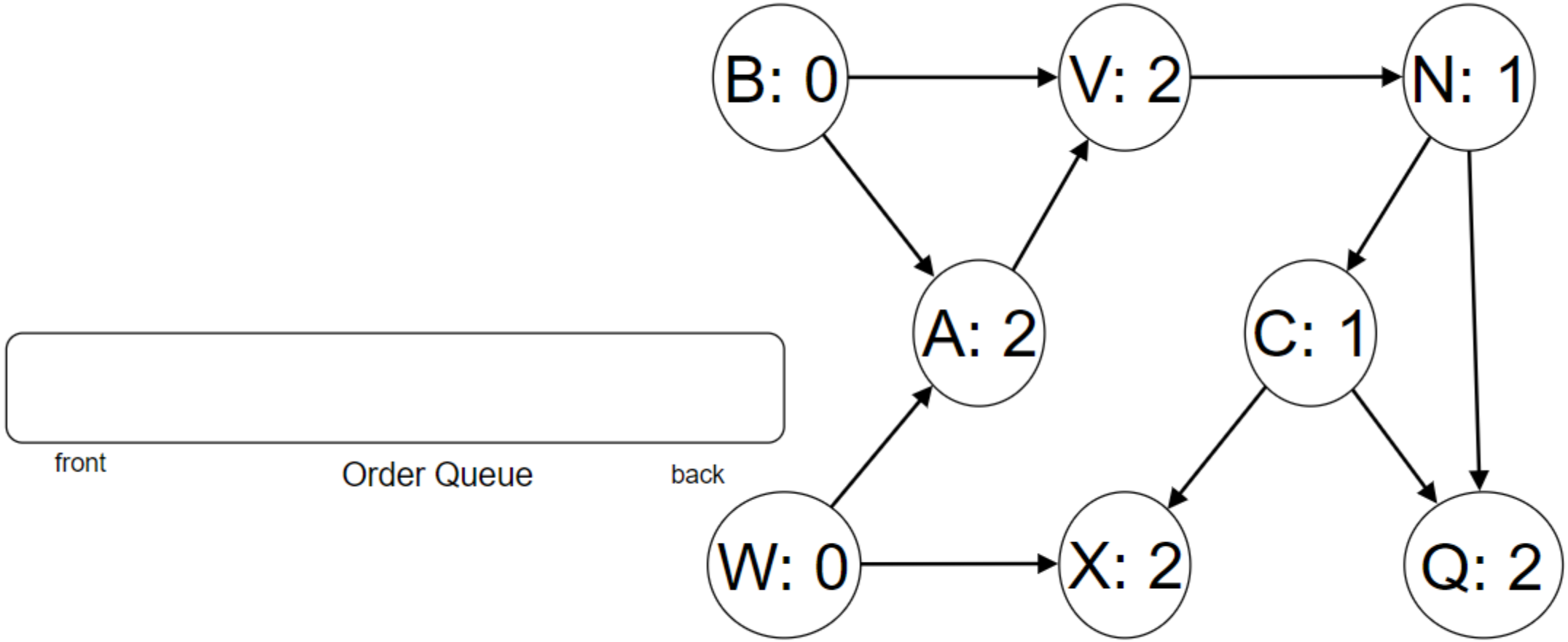
Topological Order Algorithm: Indegree

- Another way to find the topological order is to keep track of the *indegree* of each vertex.
  - The *indegree* is the number of edges coming **into** the vertex.
- Algorithm:
  - pick a vertex with indegree = 0
  - add that vertex to the order
  - remove that vertex and all edges coming from that vertex from the graph
  - update the indegree values
  - repeat

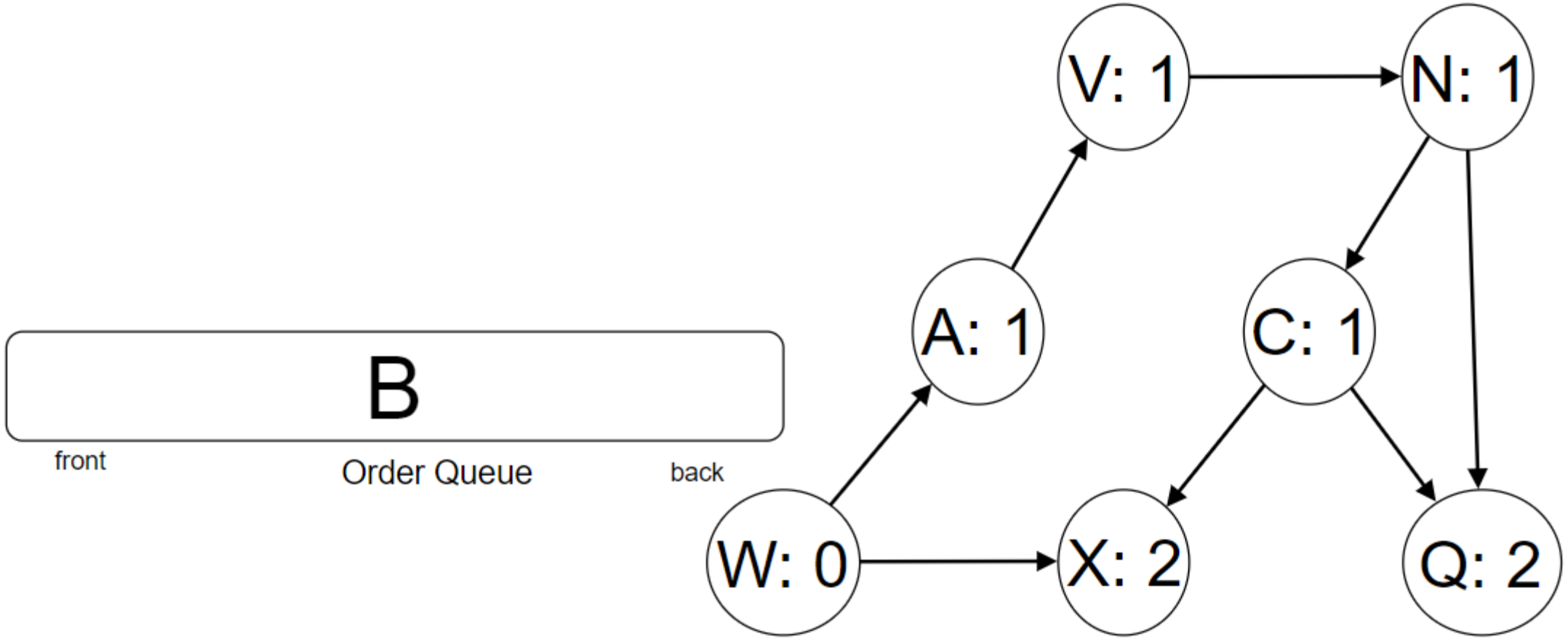
Example: Topological Order by Indegree

- Find a topological order of this DAG.

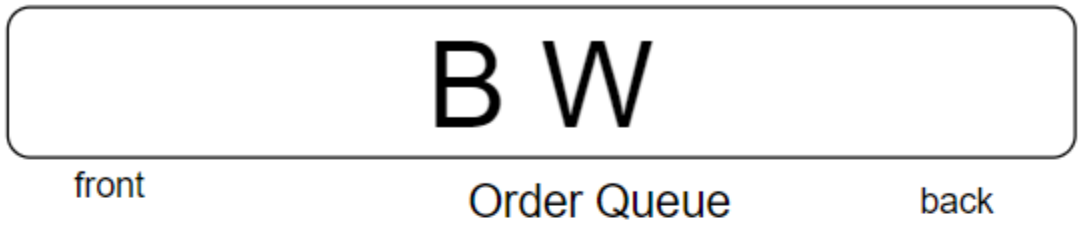


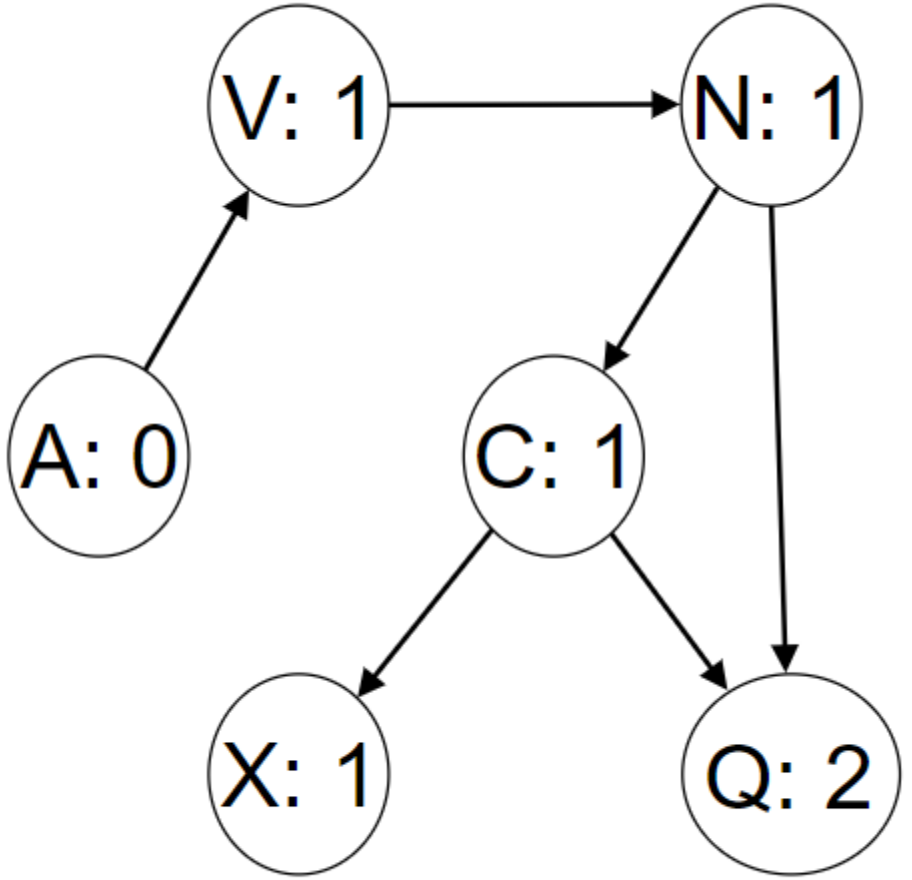


determine the indegree of each vertex (the number of edges coming INTO the vertex)

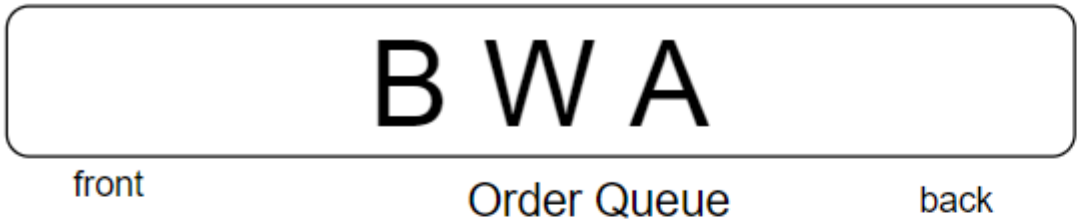


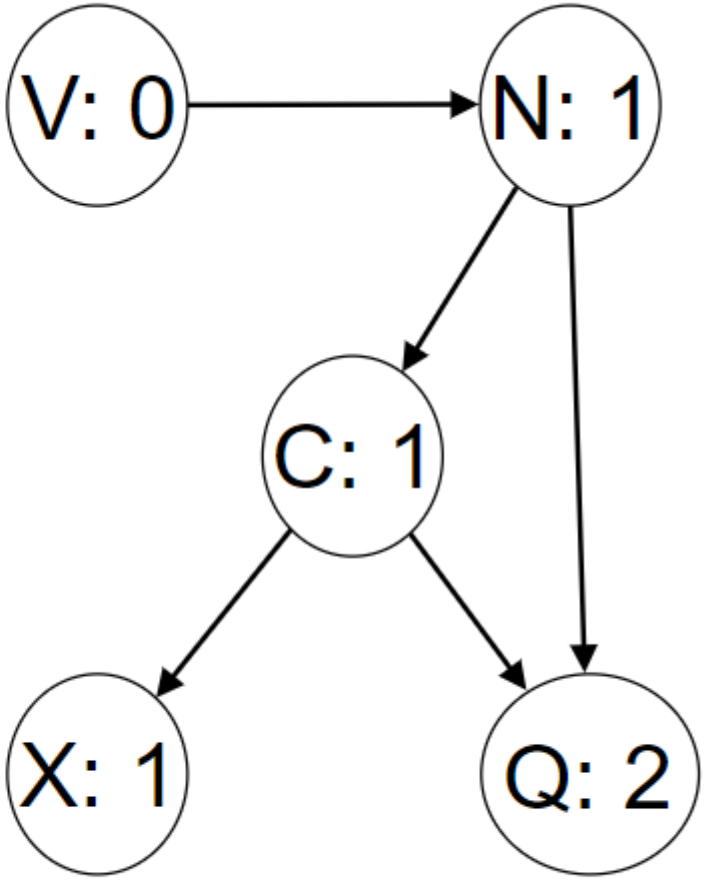
choose a vertex with indegree=0 (B or W; we choose B)
enqueue the vertex
remove the vertex and update any neighbor indegrees



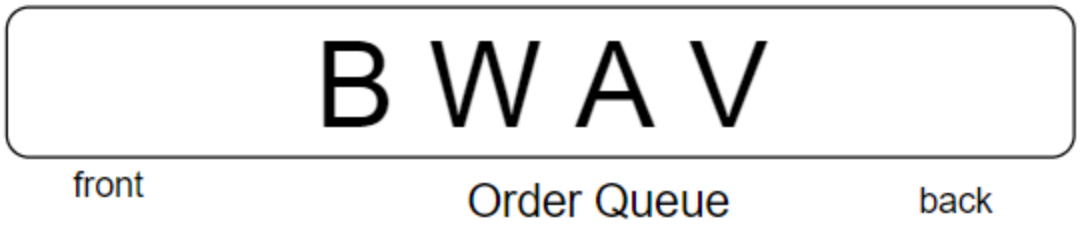


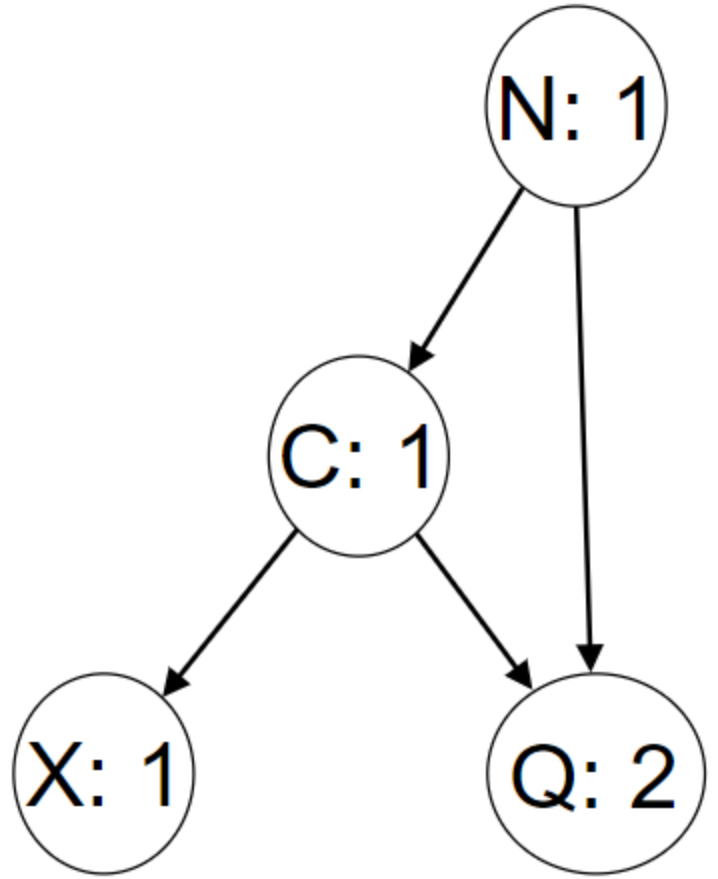
choose a vertex with indegree=0 (W)
enqueue the vertex
remove the vertex and update any neighbor indegrees



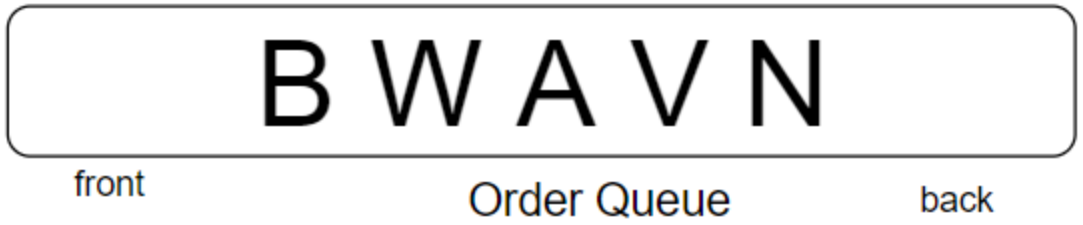


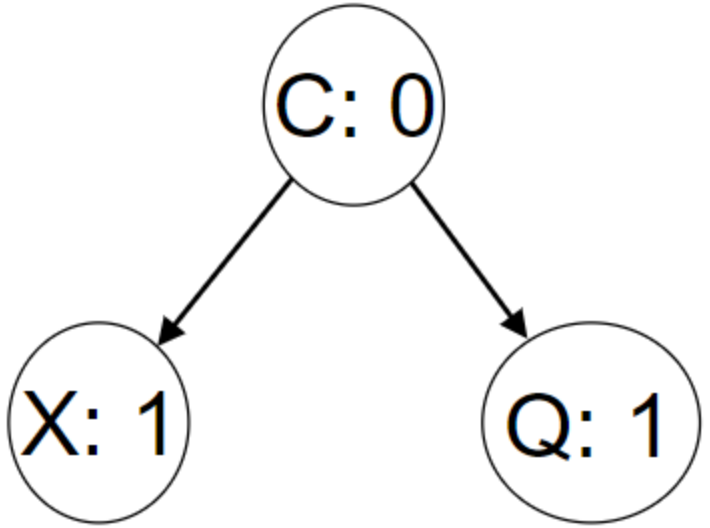
choose a vertex with indegree=0 (A)
enqueue the vertex
remove the vertex and update any neighbor indegrees



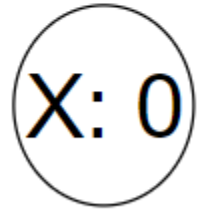


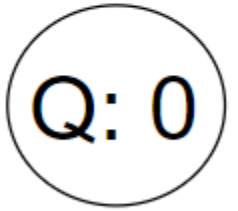
choose a vertex with indegree=0 (V)
enqueue the vertex
remove the vertex and update any neighbor indegrees



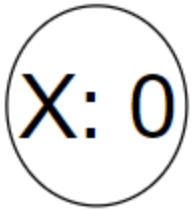


choose a vertex with indegree=0 (N)
enqueue the vertex
remove the vertex and update any neighbor indegrees





choose a vertex with indegree=0 (C)
enqueue the vertex
remove the vertex and update any neighbor indegrees



choose a vertex with indegree=0 (Q or X; we choose Q)
enqueue the vertex
remove the vertex and update any neighbor indegrees

choose a vertex with indegree=0 (X)
enqueue the vertex
remove the vertex and update any neighbor indegrees

Topological Order: BWAVNCQX
(queue from front to back)

GRAPH IMPLEMENTATIONS

Graph Implementations

- Graphs can be implemented using an *adjacency matrix* or with an *adjacency list*

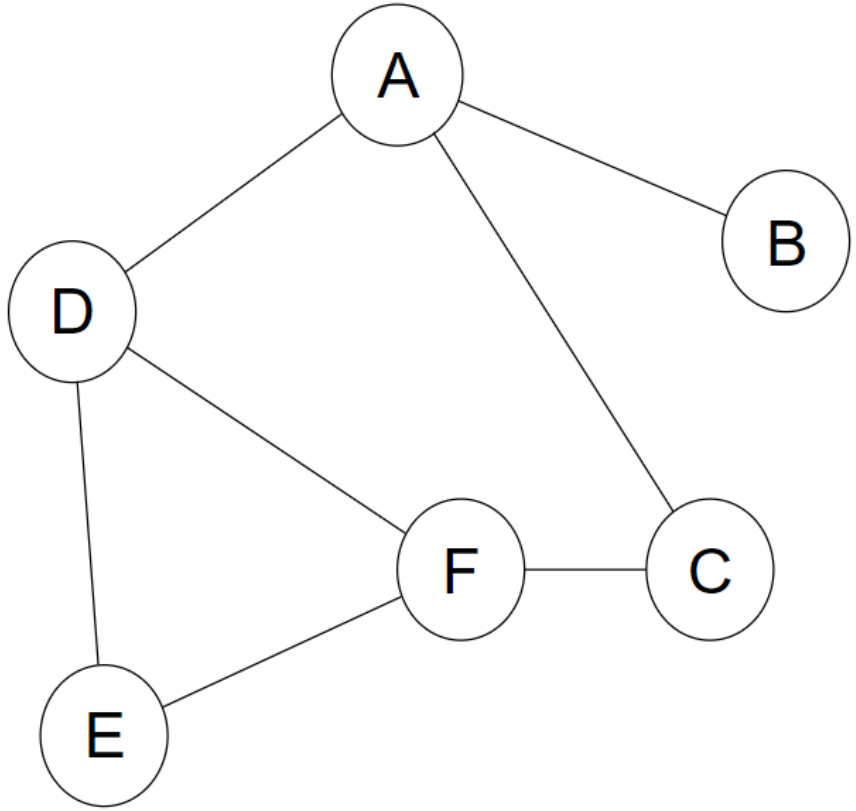
Adjacency Matrix

- A matrix / table / 2-dimensional array
- For a graph with n vertices, the matrix is n rows * n columns
- If there is an edge from vertex i to vertex j, there is an element in row i, column j in the matrix
  - cell (i, j) has a value if there is an edge from i->j
- For unweighted graphs, a boolean value can be used as the value
- For weighted graphs, the weight should be used as the value
- For undirected graphs, the matrix will be symmetrical:
  - cell (i, j) and (j, i) have the same value

Example: Undirected, Unweighted

	A	B	C	D	E	F
A		T	T	T		
B	T					
C	T					T
D	T				T	T
E				T		T
F			T	T	T	

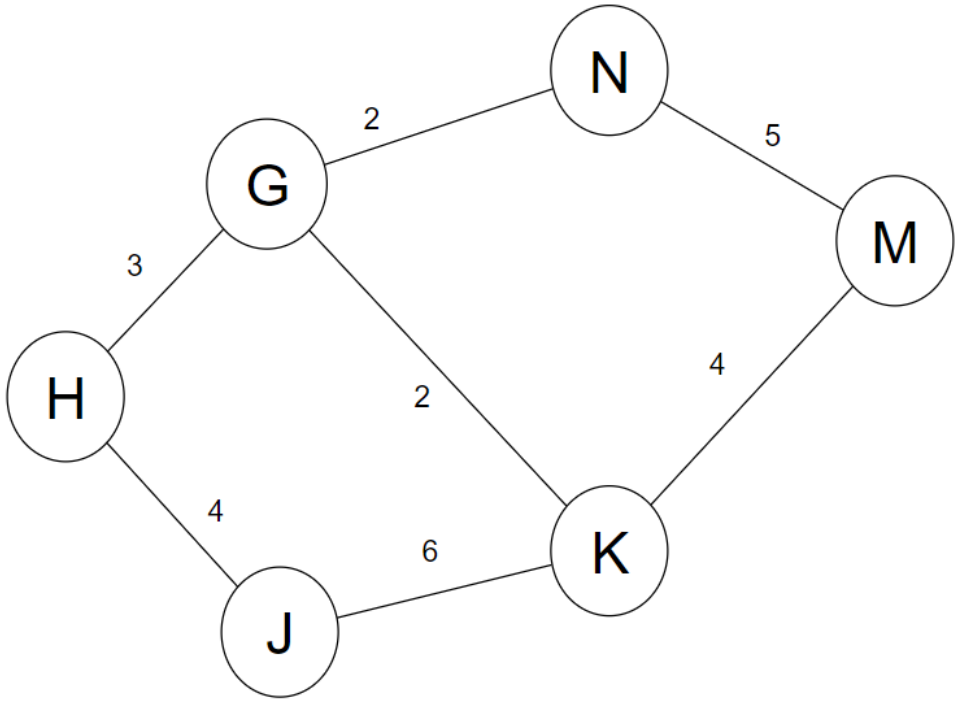
symmetrical!



Example: Undirected, Weighted

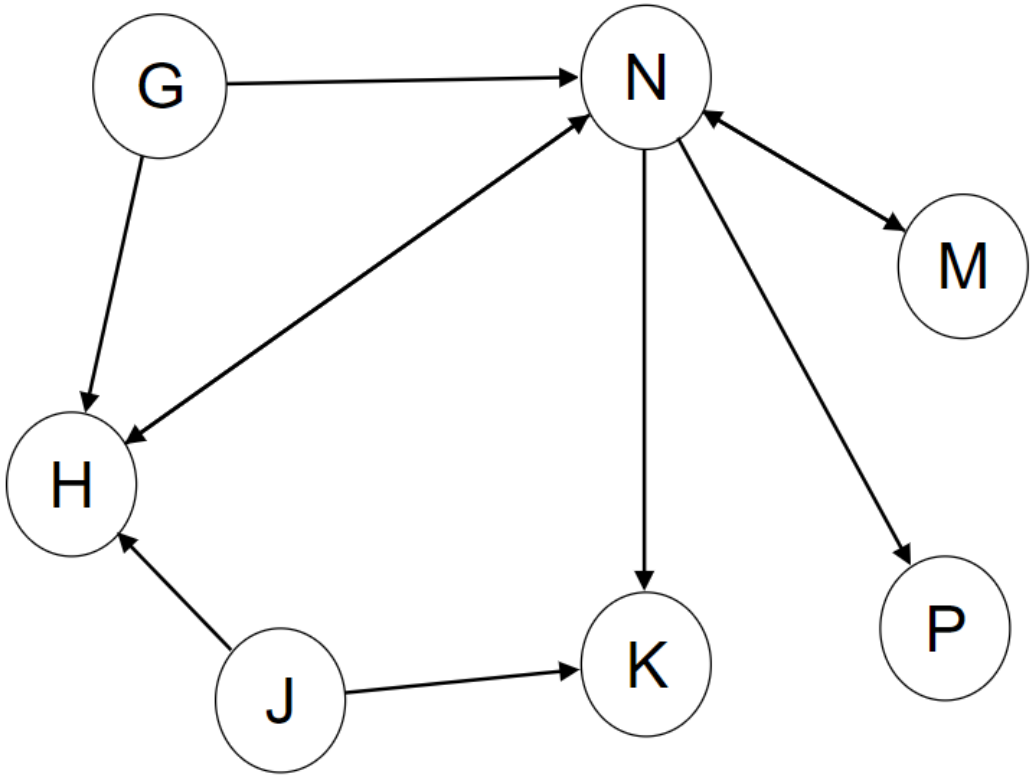
	G	H	J	K	N	M
G		3		2	2	
H	3		4			
J		4		6		
K	2		6			4
N	2					5
M				4	5	

symmetrical!



Example: Directed, Unweighted

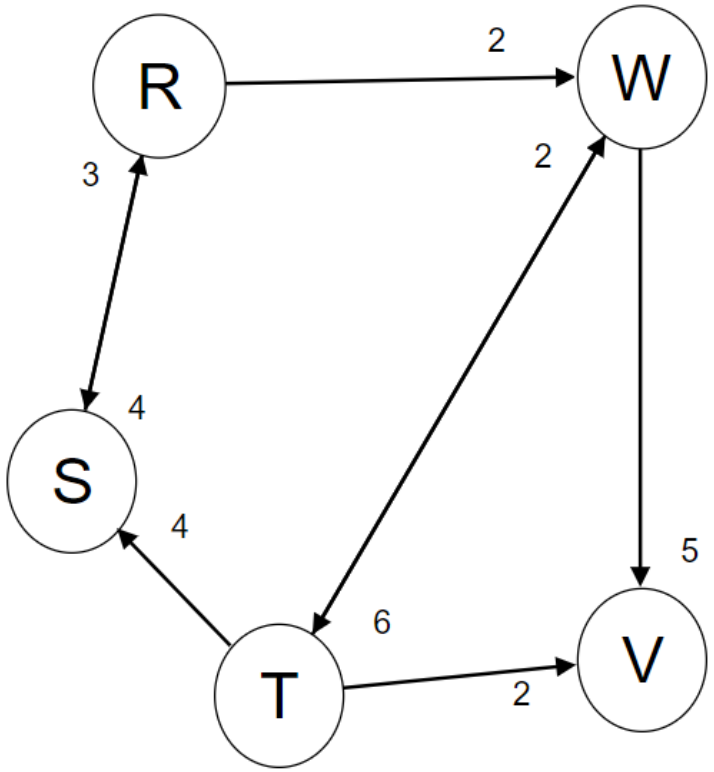
	G	H	J	K	M	N	P
G		T				T	
H						T	
J		T		T			
K							
M						T	
N		T		T	T		T
P							



originating vertex is the row; neighbor is the column!

Example: Directed, Weighted

	R	S	T	V	W
R		4			2
S	3				
T		4		2	2
V					
W			6	5	



originating vertex is the row; neighbor is the column!

Adjacency Matrix

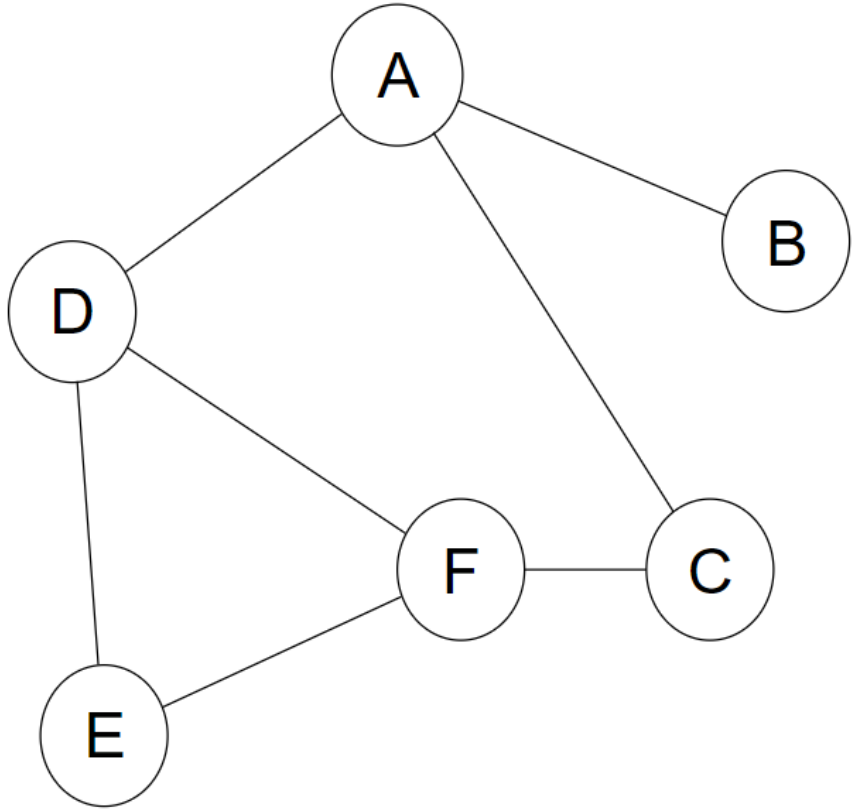
- Determining whether there is an edge between two vertices is fast!
  - O(1)
- Finding all neighbors of a vertex means you have to look across an entire row
  - O(n)
- A matrix is often not space-efficient, especially for sparse graphs

Adjacency List

- A list represents all neighbors of a vertex

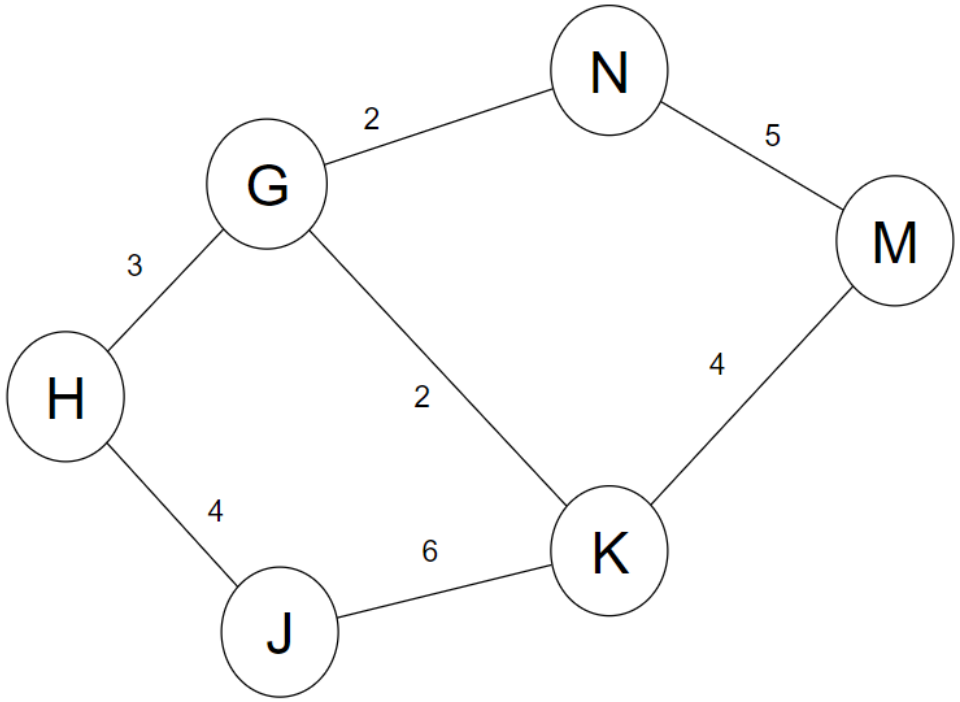
Example: Undirected, Unweighted

- A -> B C D
- B -> A
- C -> A F
- D -> A E F
- E -> D F
- F -> C D E



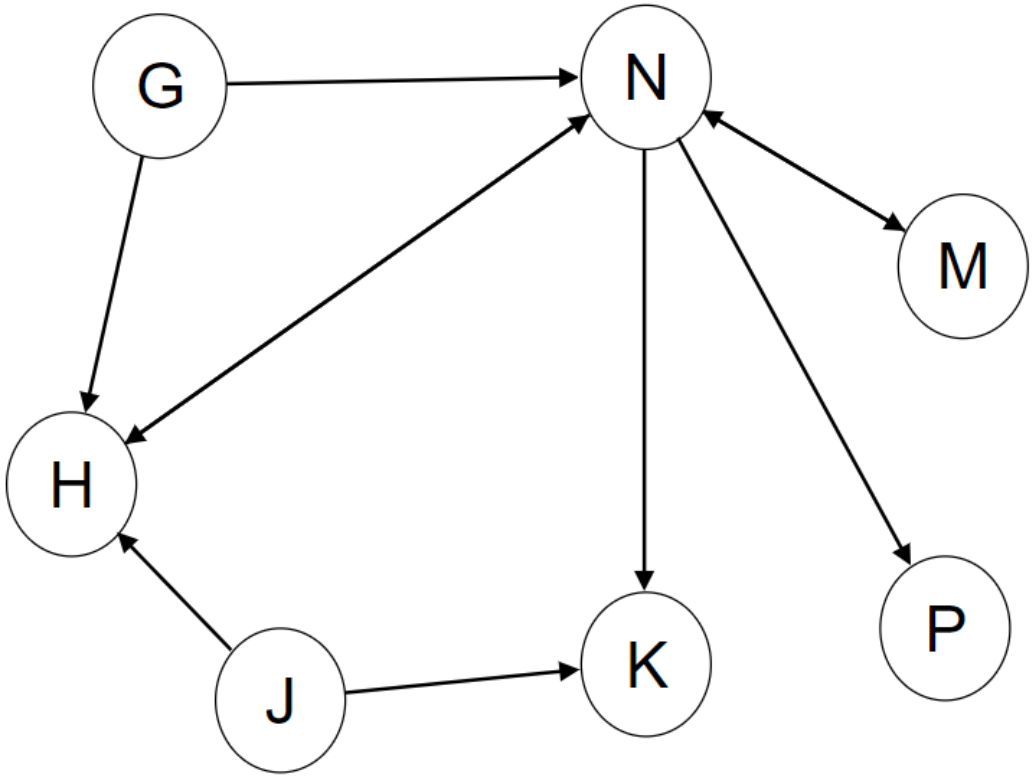
Example: Undirected, Weighted

- G -> H(3) K(2) N(2)
- H -> G(3) J(4)
- J -> H(4) K(6)
- K -> G(2) J(6) M(4)
- M-> K(4) N(5)
- N -> G(2) M(5)



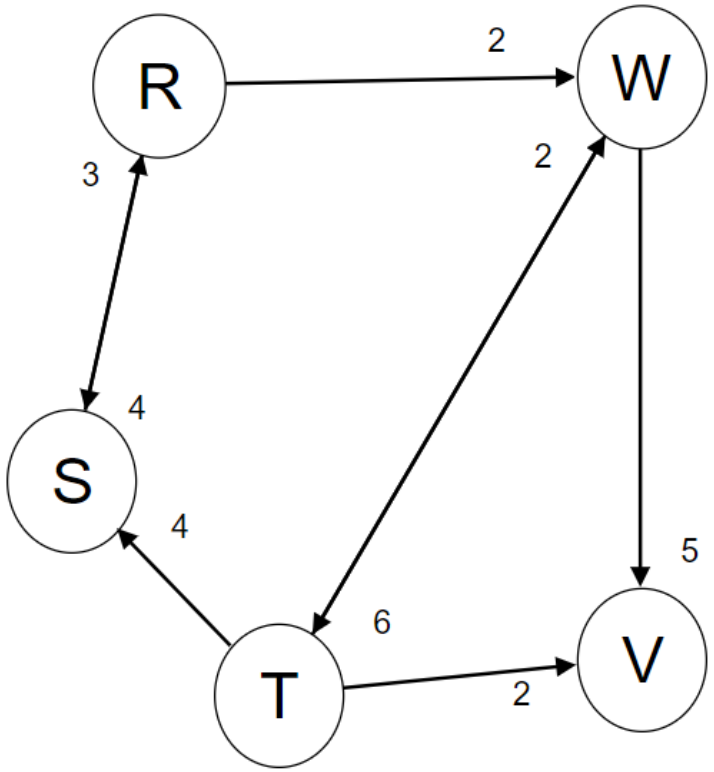
Example: Directed, Unweighted

- G -> H N
- H -> N
- J -> H K
- K ->
- M -> N
- N -> H K M P
- P ->



Example: Directed, Weighted

- R -> S(4) W(2)
- S -> R(3)
- T -> S(4) V(2) W(2)
- W -> T(6) V(5)
- V ->



Adjacency List

- Determining whether there is an edge between two vertices requires you to search an entire list
  - O(n) in the worst case
- Finding all neighbors of a vertex means you have to look across the list
  - O(n) in the worst case, but more efficient in an average and best case
- A list is space-efficient, especially for sparse graphs

Graph Implementations

- Choose an implementation based on:
  - the nature of your data (sparse/dense)
  - how the data will be used (are you most often finding neighbors, checking if edges exist, etc.)