Revised: 3/6/2020

Graphs II

Michael C. Hackett

Computer Science Department

Community College of Philadelphia

Lecture Topics

- Graph Traversals
  - Depth-First
  - Breadth-First
- Finding distance with breadth-first
- Weighted Graphs
  - Dijkstra's Algorithm

Graph Traversal

- Graphs can be traversed using depth-first or breadth-first traversals.

- Similar to traversing a tree, but we can begin at any node in the graph.

- Depth-first traversal uses a *stack*
- Breadth-first traversal uses a *queue*

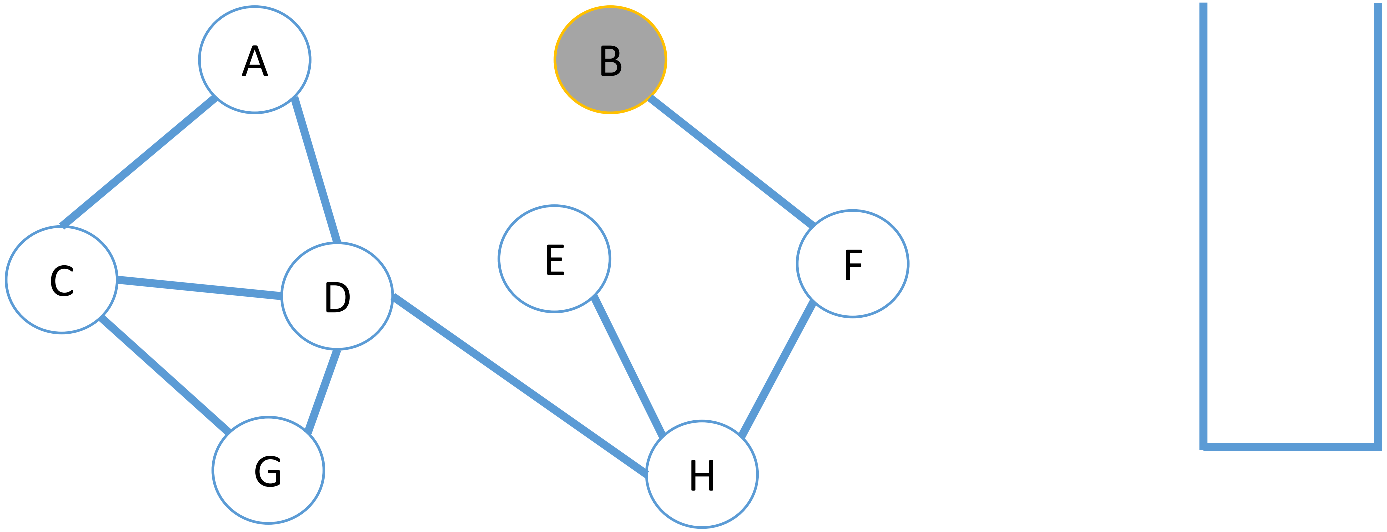
Depth-First Traversal

- Depth-first traversals work by coloring each node
  - White: Node has not been seen previously
  - Gray: Node has been seen previously
  - Black: Node has been visited and we are done with it

- This prevents getting stuck in a cycle, if one existed
  - The traversal is the same for bi-directional graphs or digraphs

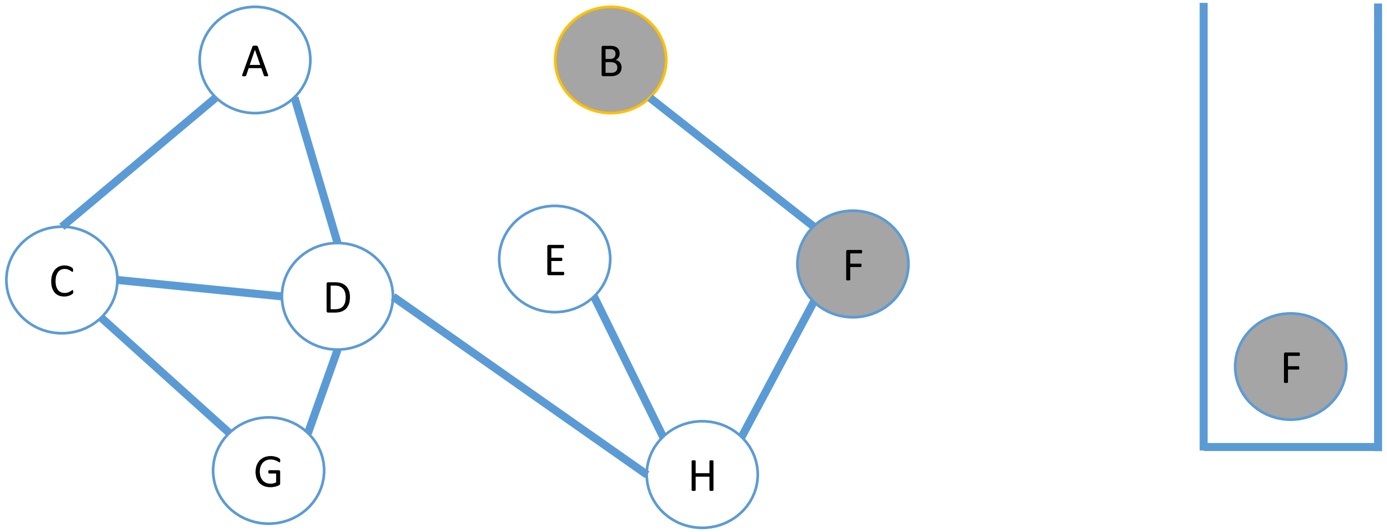
Depth-First Traversal

- Starting at B
- Order visited: N/A



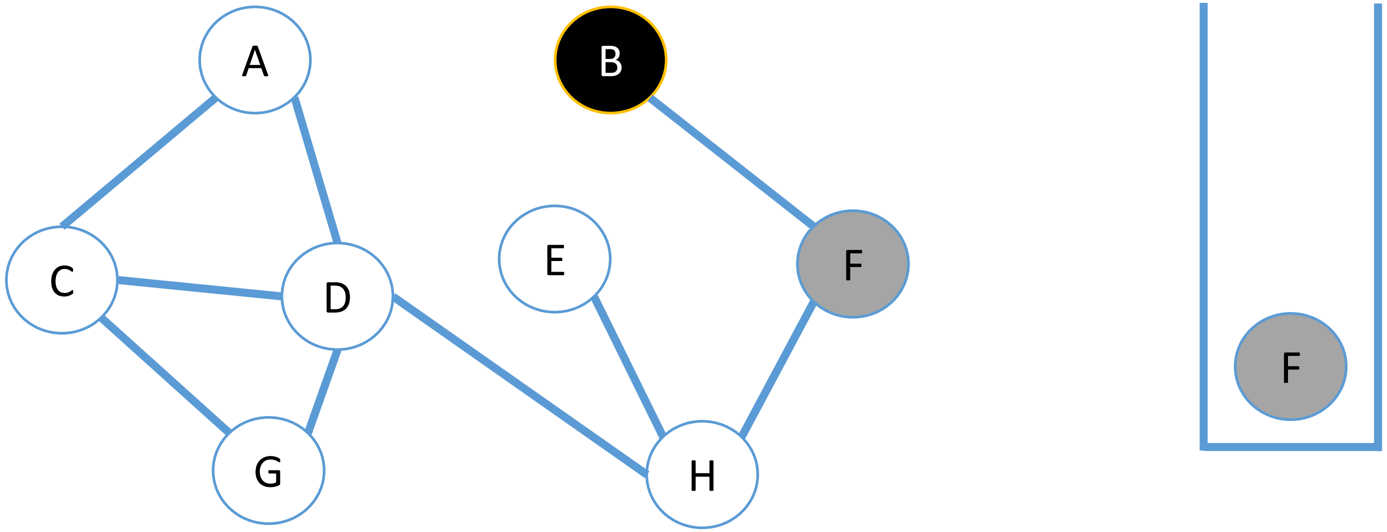
Depth-First Traversal

- Adds its adjacent white node(s) to the stack (turning the white node(s) gray)
- Order visited: N/A



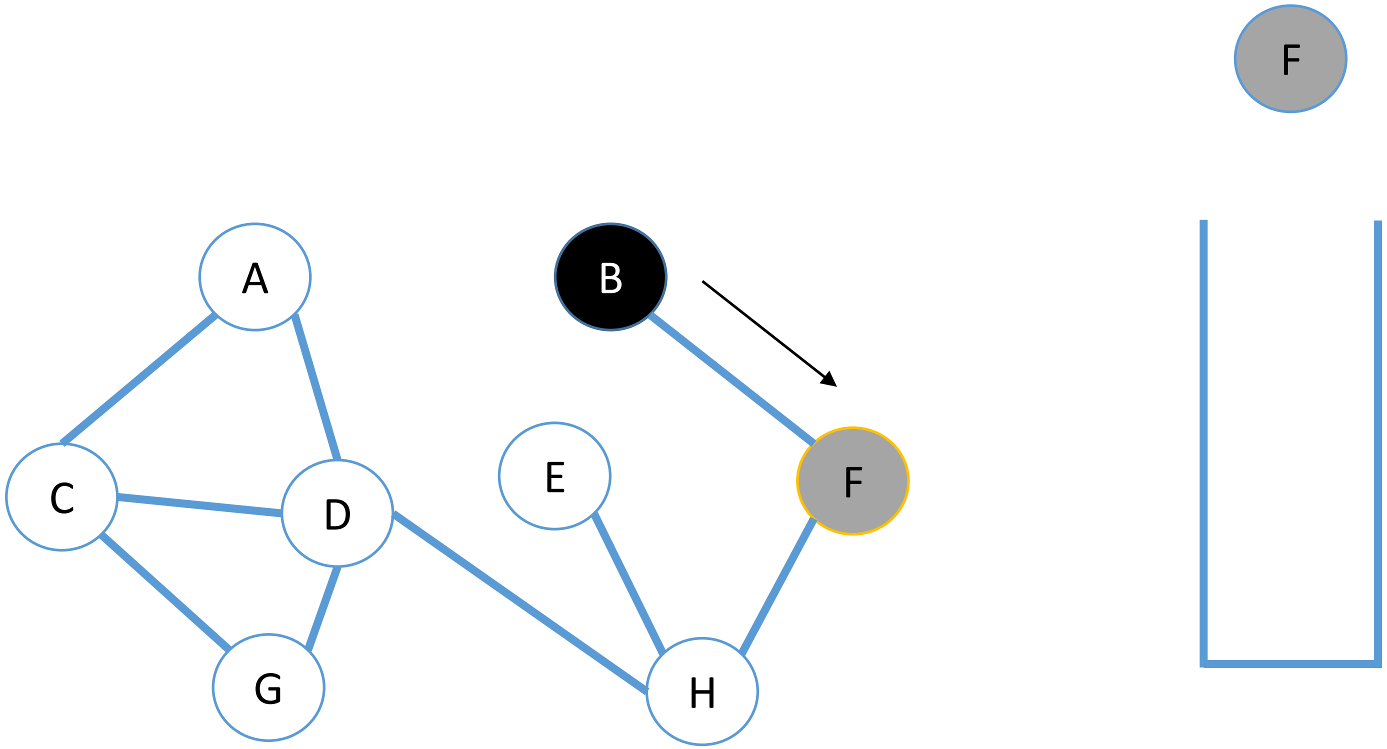
Depth-First Traversal

- B was visited
  - Changed to black
- Order visited: B



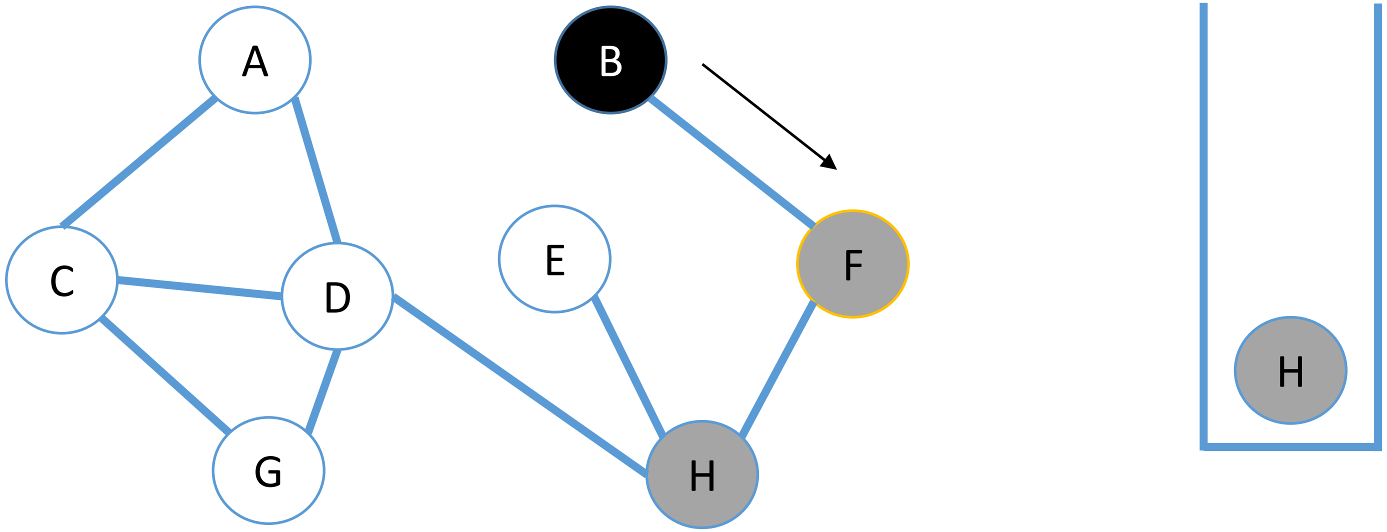
Depth-First Traversal

- Next node popped from stack
- Order visited: B



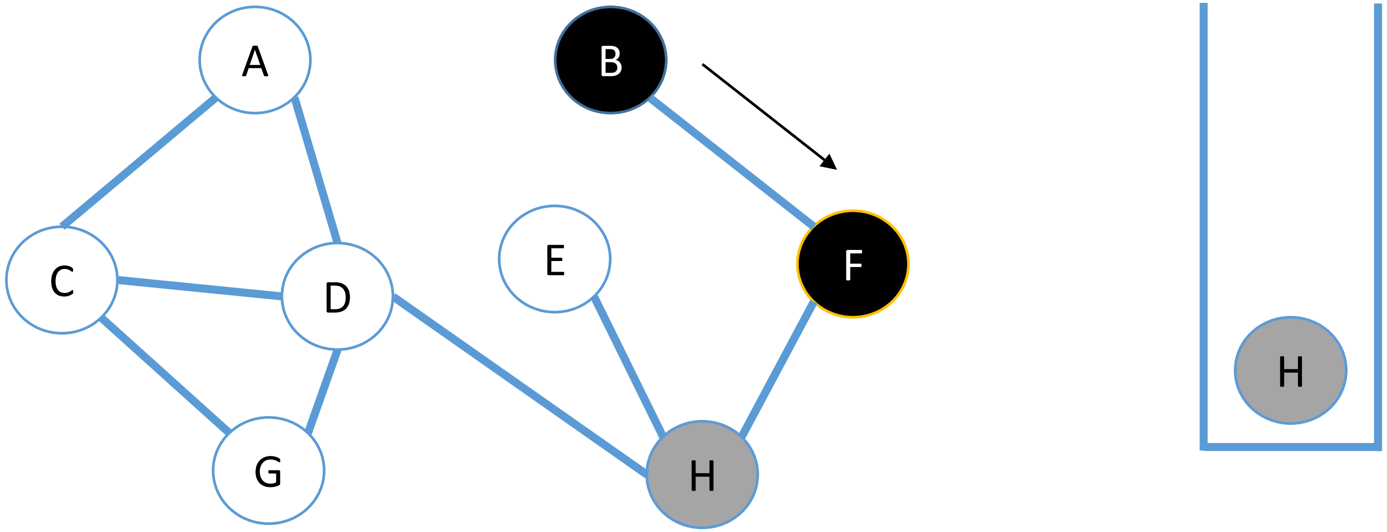
Depth-First Traversal

- Adds its adjacent white nodes to the stack (turning them gray)
- Order visited: B



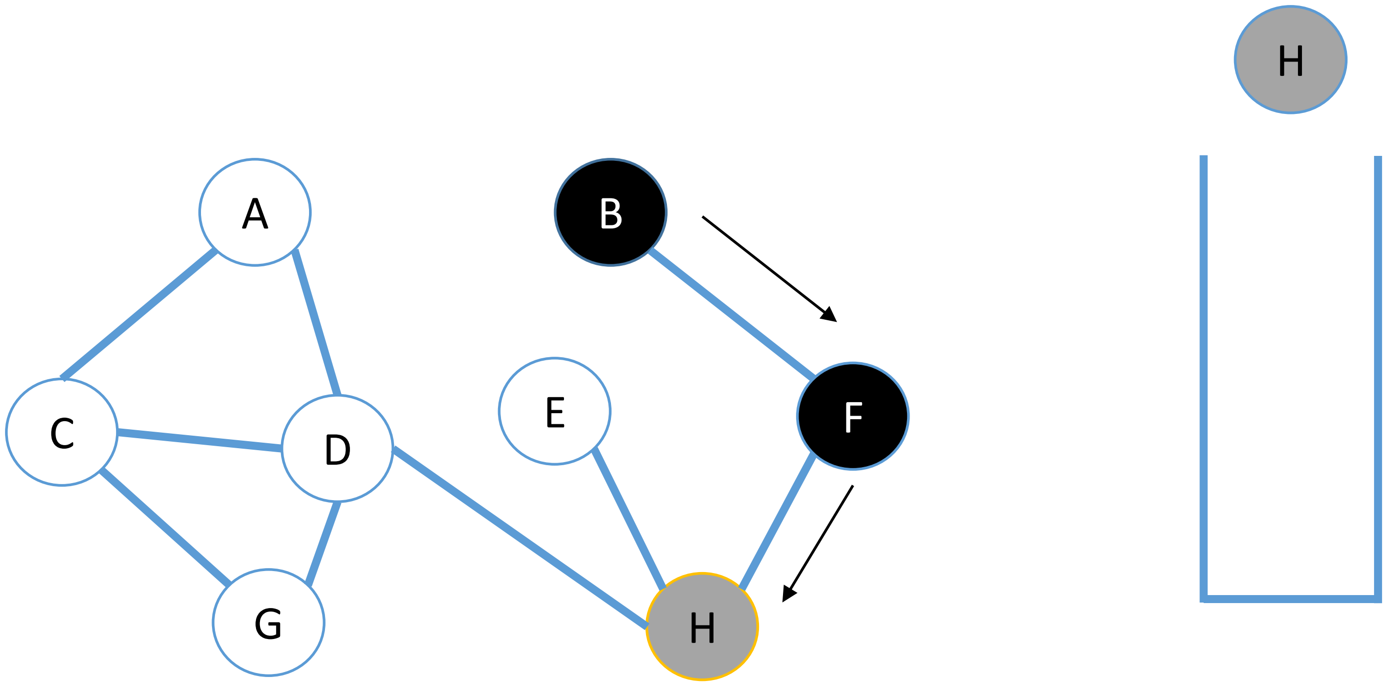
Depth-First Traversal

- F was visited
- Order visited: B, F



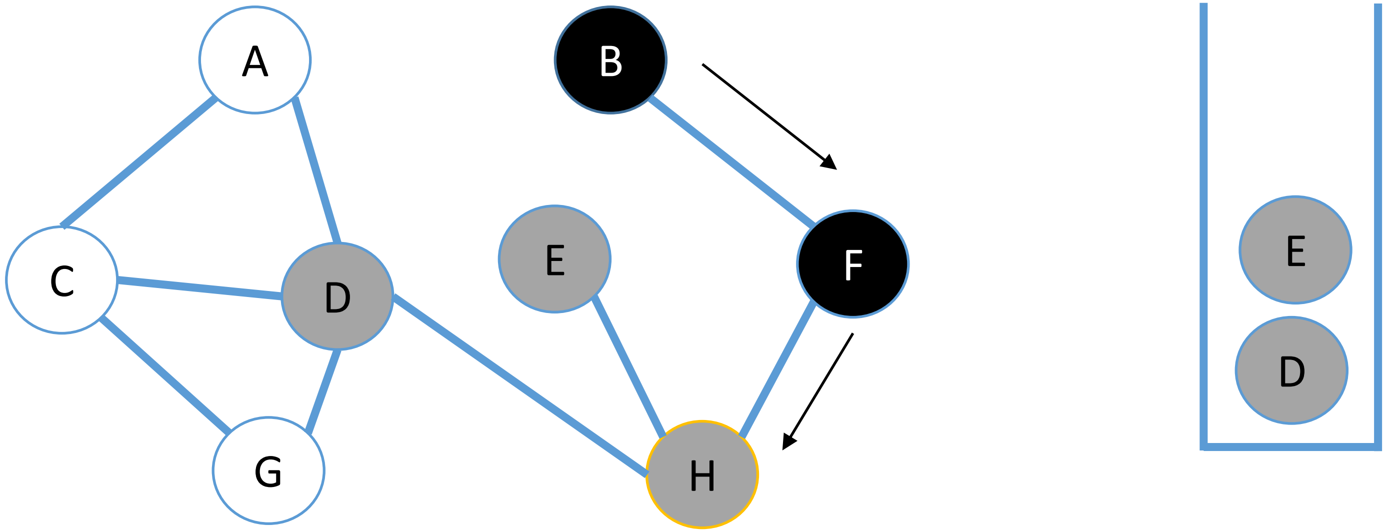
Depth-First Traversal

- Next node popped from stack
- Order visited: B, F



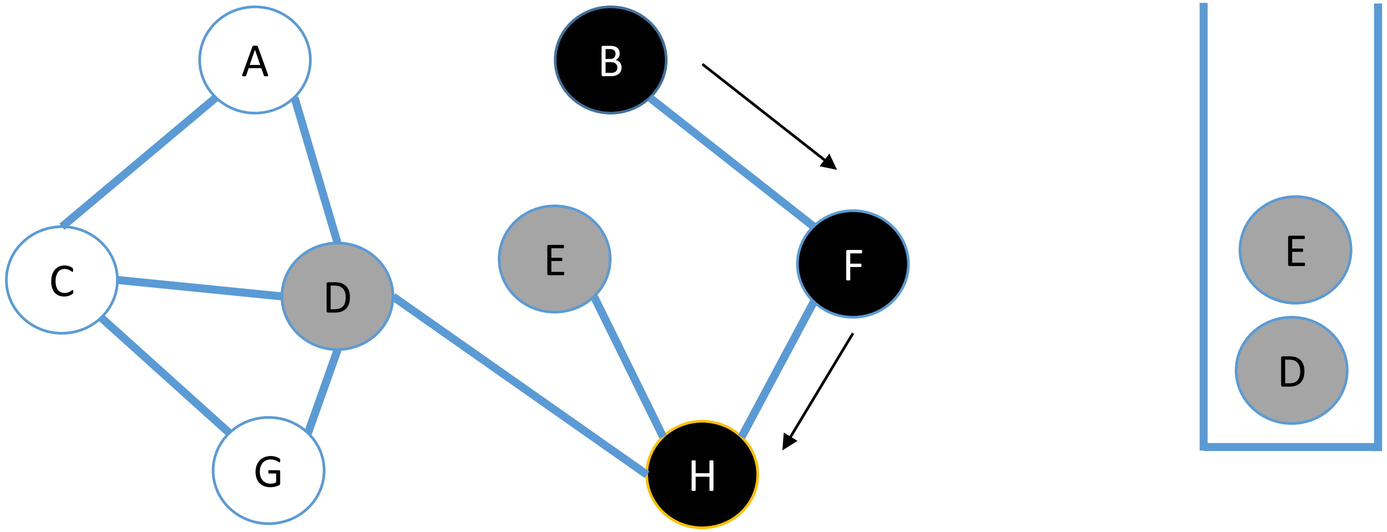
Depth-First Traversal

- Adds its adjacent white nodes to the stack
- Order visited: B, F



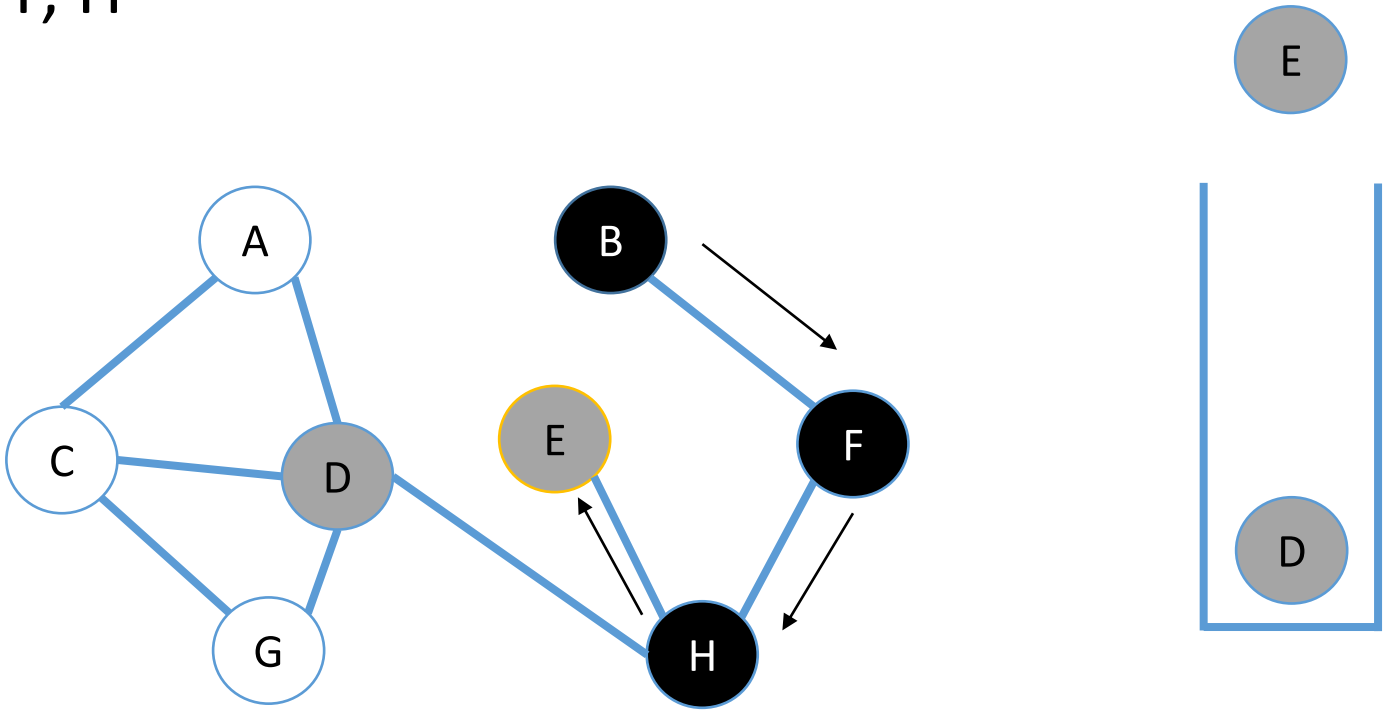
Depth-First Traversal

- H was visited
- Order visited: B, F, H



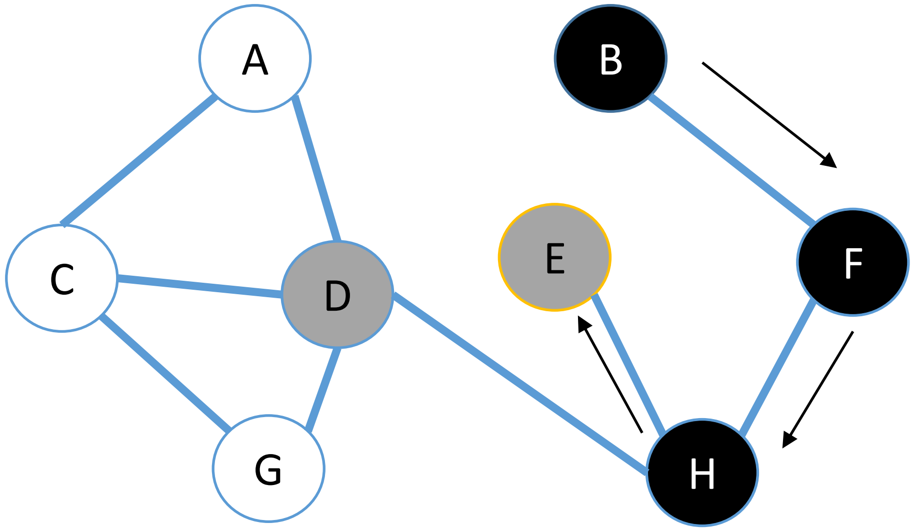
Depth-First Traversal

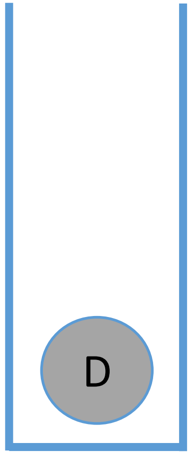
- Next node popped from stack
- Order visited: B, F, H



Depth-First Traversal

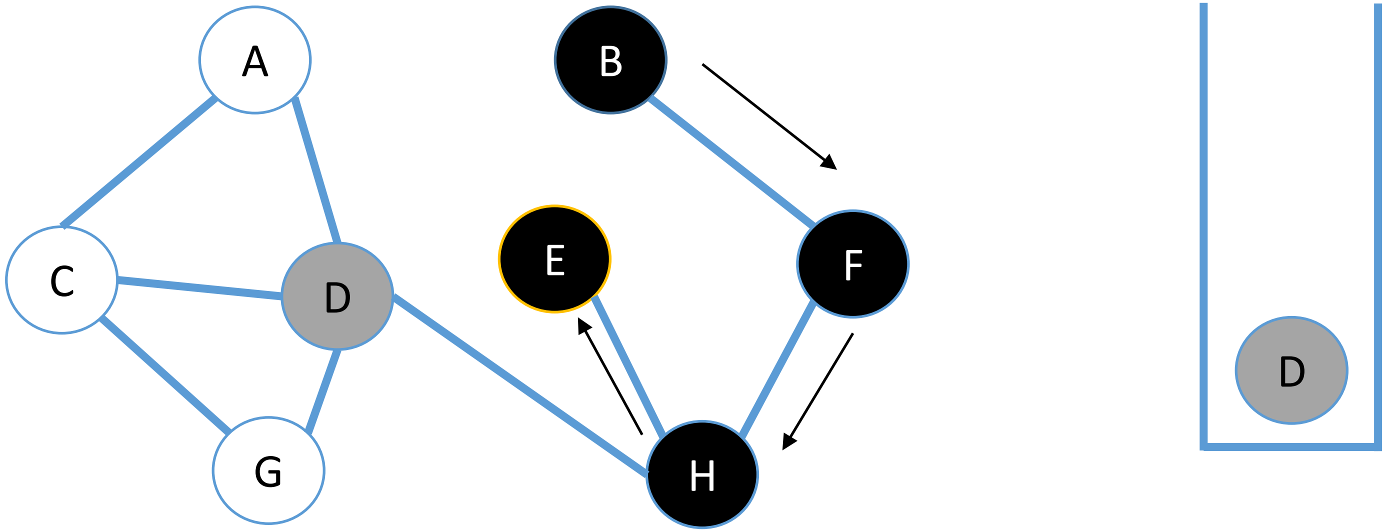
- Add its adjacent white nodes to the stack (there are none)
- Order visited: B, F, H





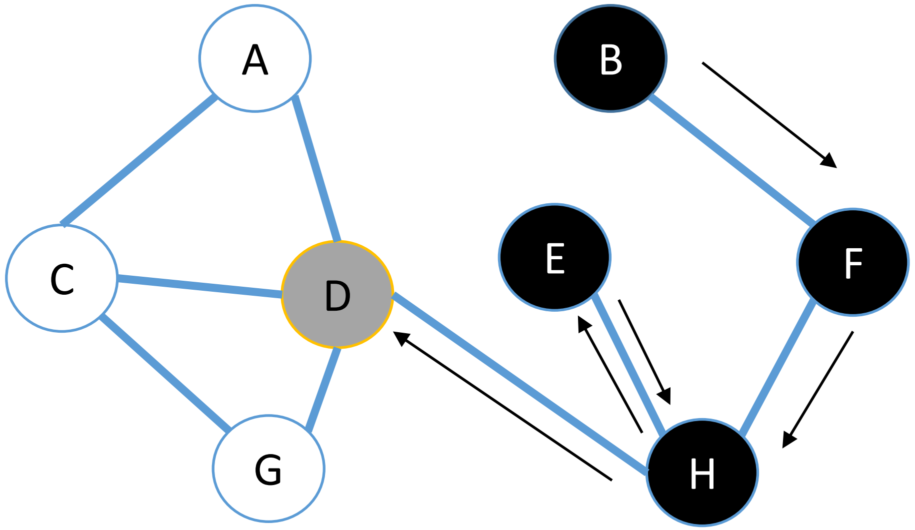
Depth-First Traversal

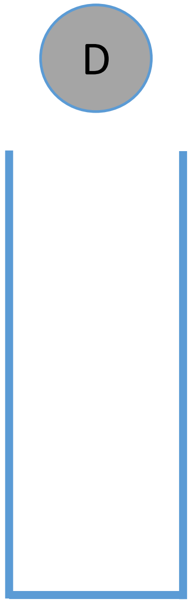
- E was visited
- Order visited: B, F, H, E



Depth-First Traversal

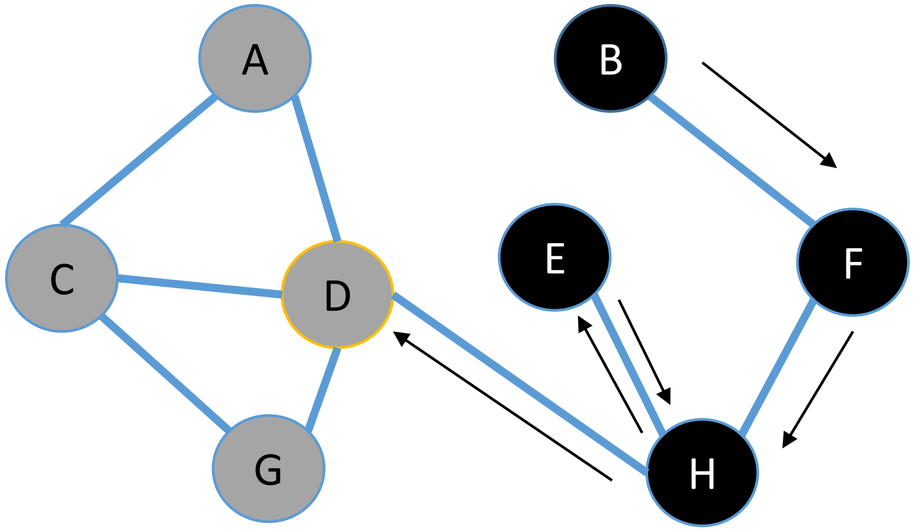
- Next node popped from stack
- Order visited: B, F, H, E

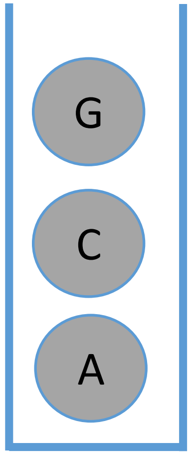




Depth-First Traversal

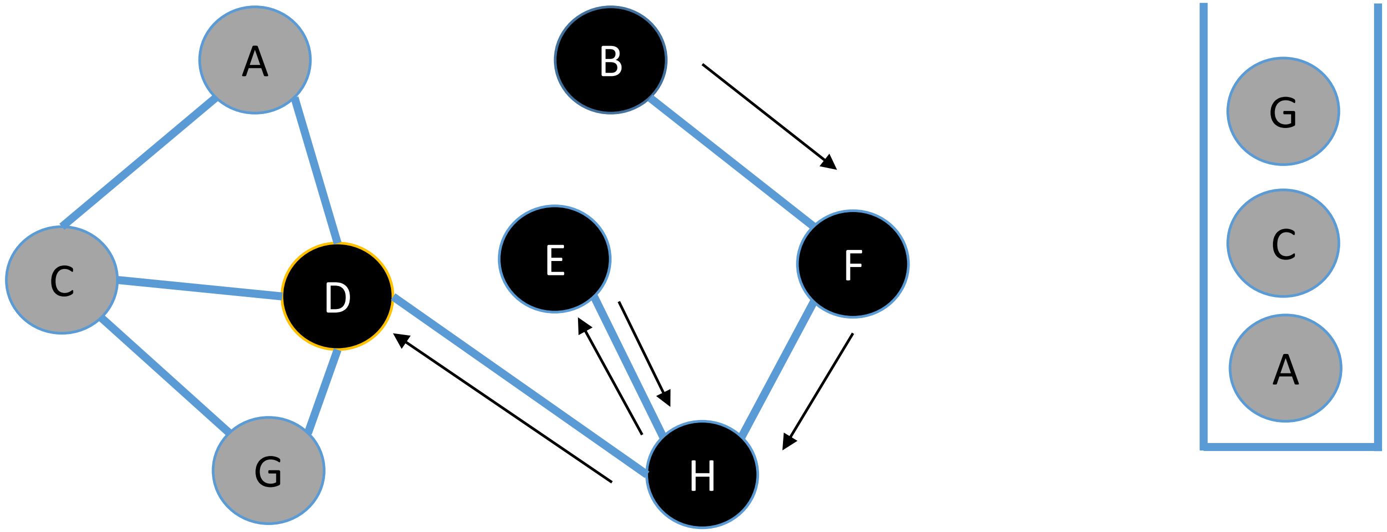
- Add its adjacent white nodes to the stack
- Order visited: B, F, H, E





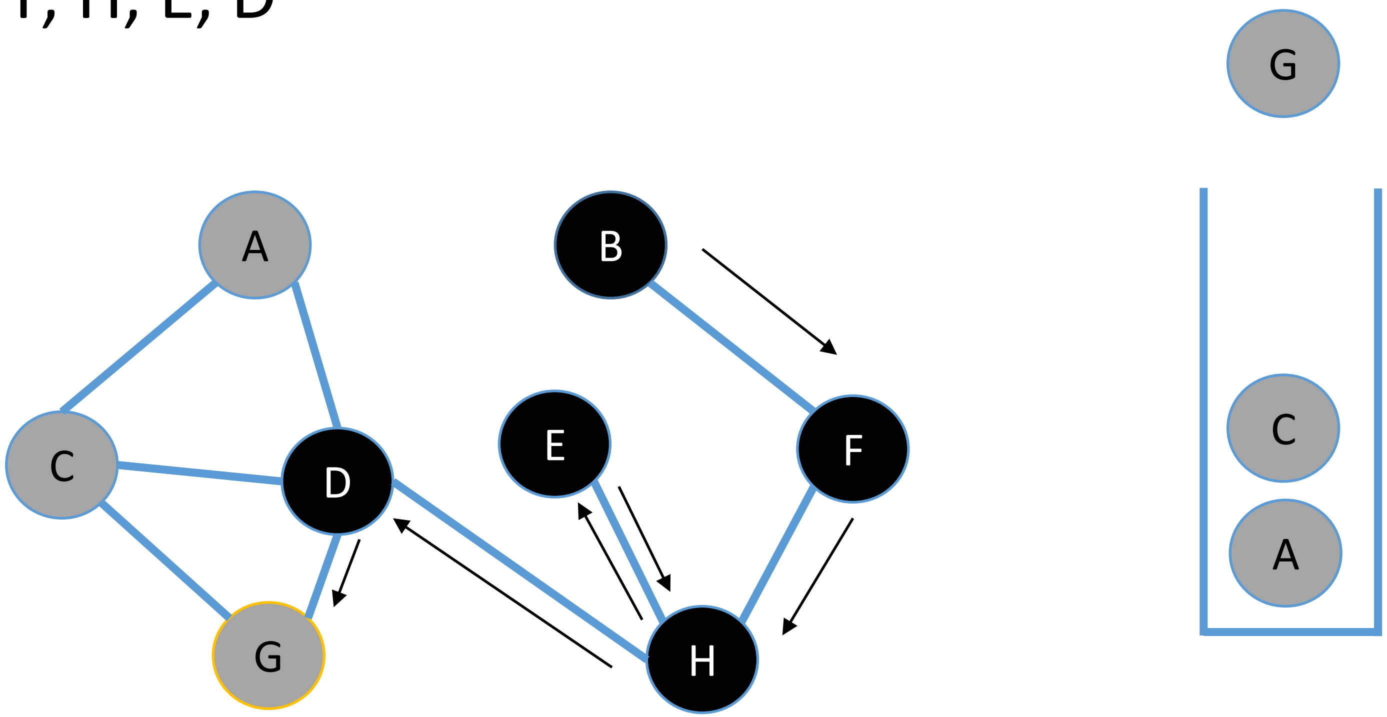
Depth-First Traversal

- D was visited
- Order visited: B, F, H, E, D



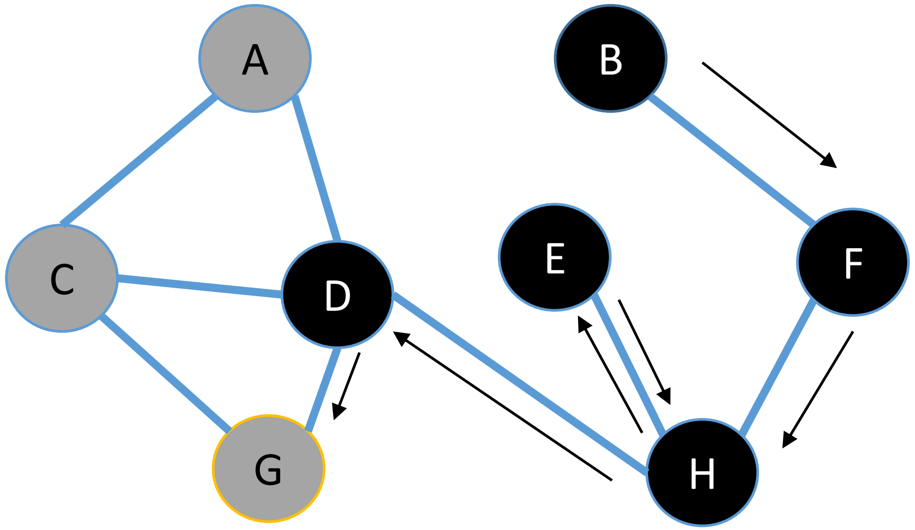
Depth-First Traversal

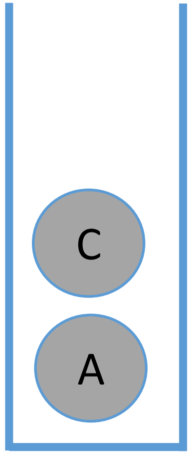
- Next node popped from the stack
- Order visited: B, F, H, E, D



Depth-First Traversal

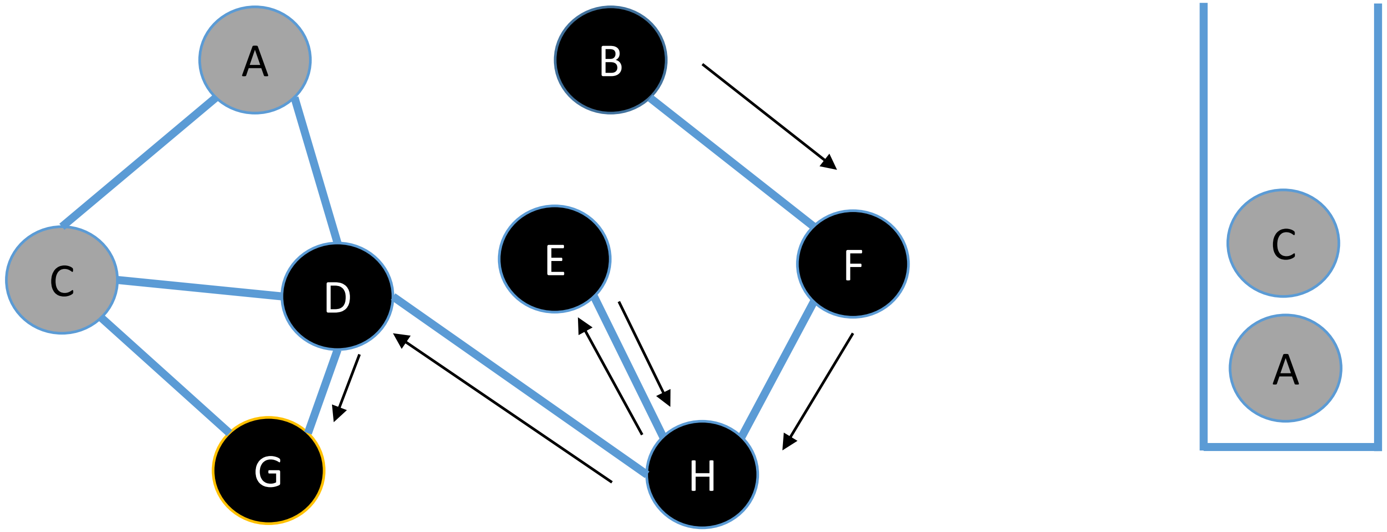
- Add its adjacent white nodes to the stack (there are none)
- Order visited: B, F, H, E, D





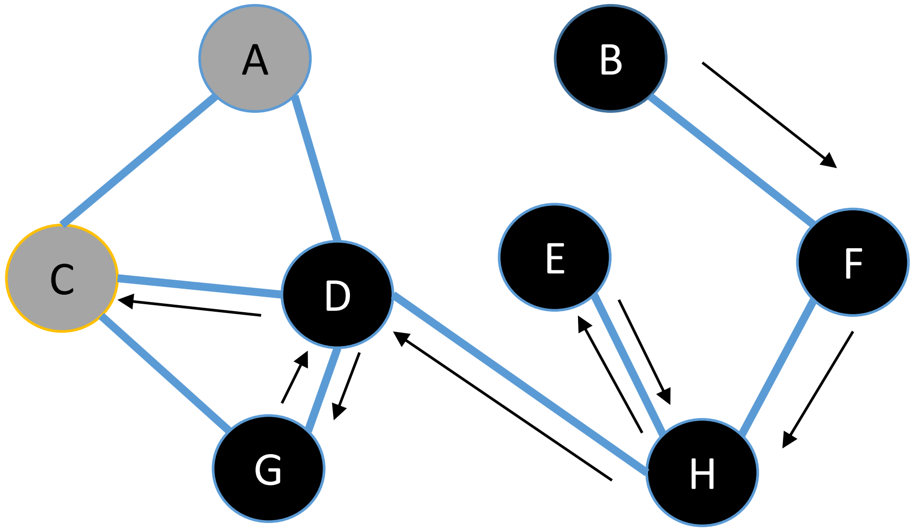
Depth-First Traversal

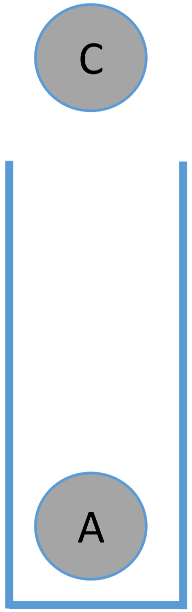
- G has been visited
- Order visited: B, F, H, E, D, G



Depth-First Traversal

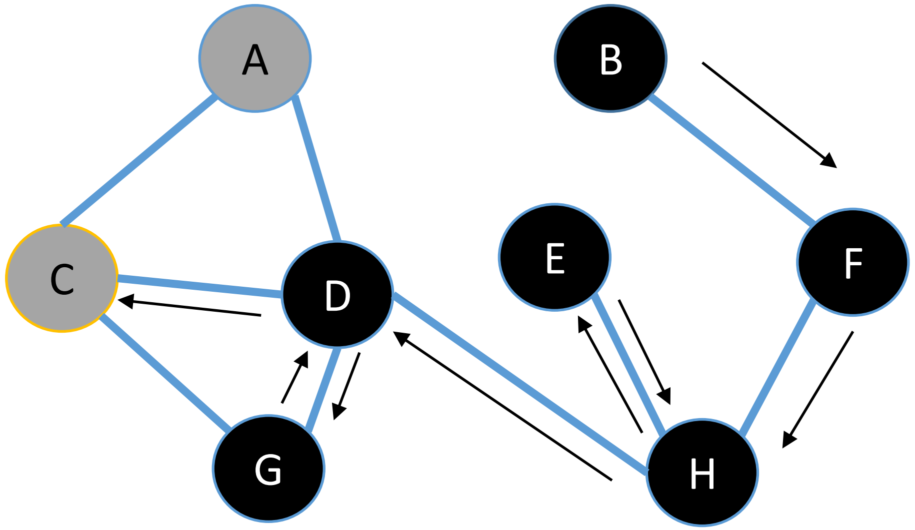
- Next node popped from the stack
- Order visited: B, F, H, E, D, G

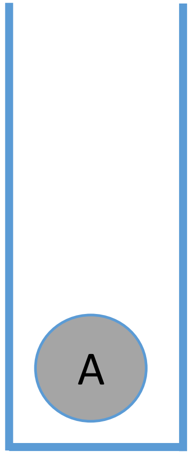




Depth-First Traversal

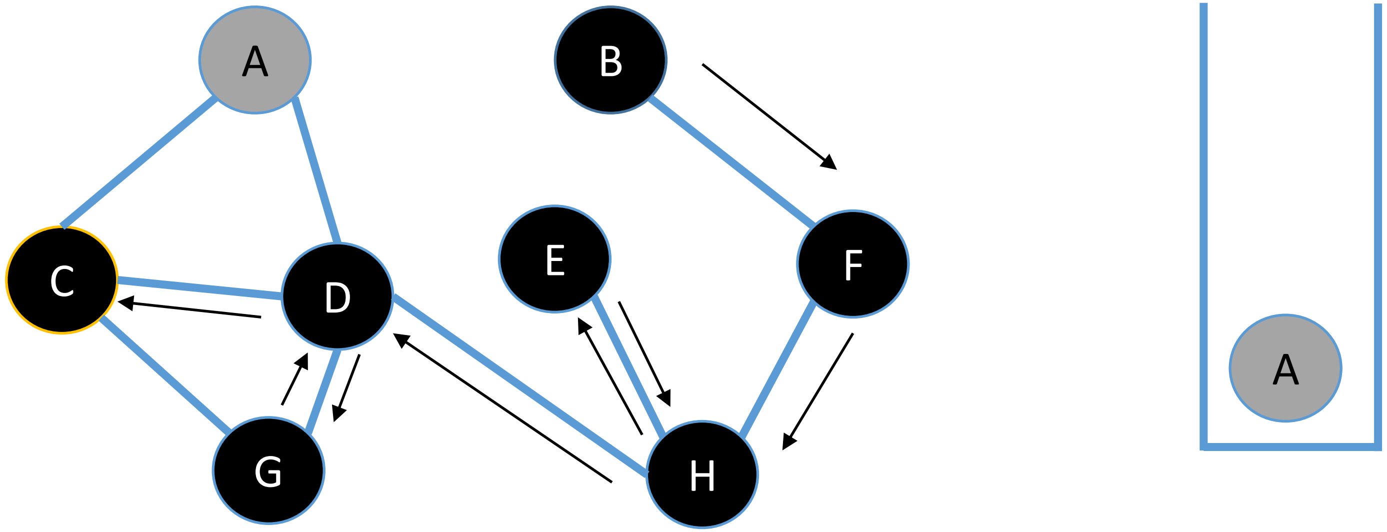
- Add its adjacent white nodes to the stack (there are none)
- Order visited: B, F, H, E, D, G





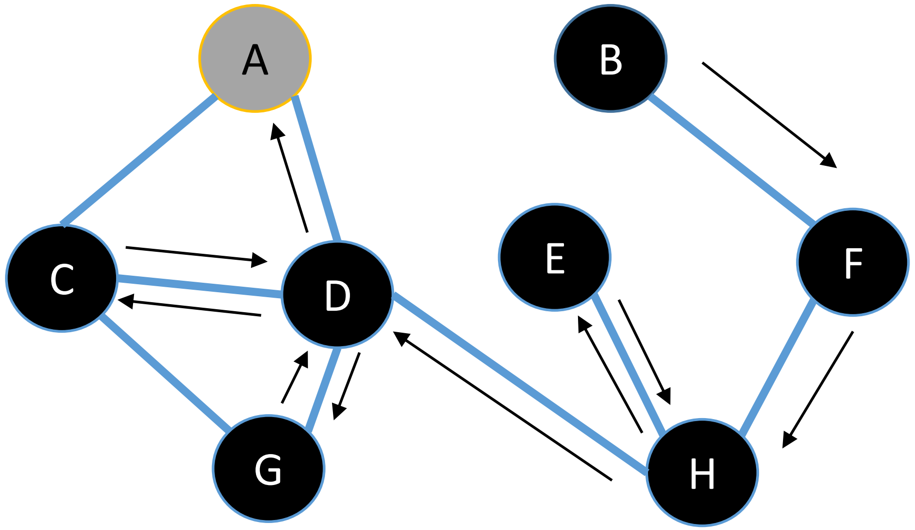
Depth-First Traversal

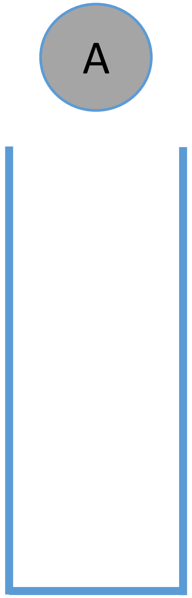
- C has been visited
- Order visited: B, F, H, E, D, G, C



Depth-First Traversal

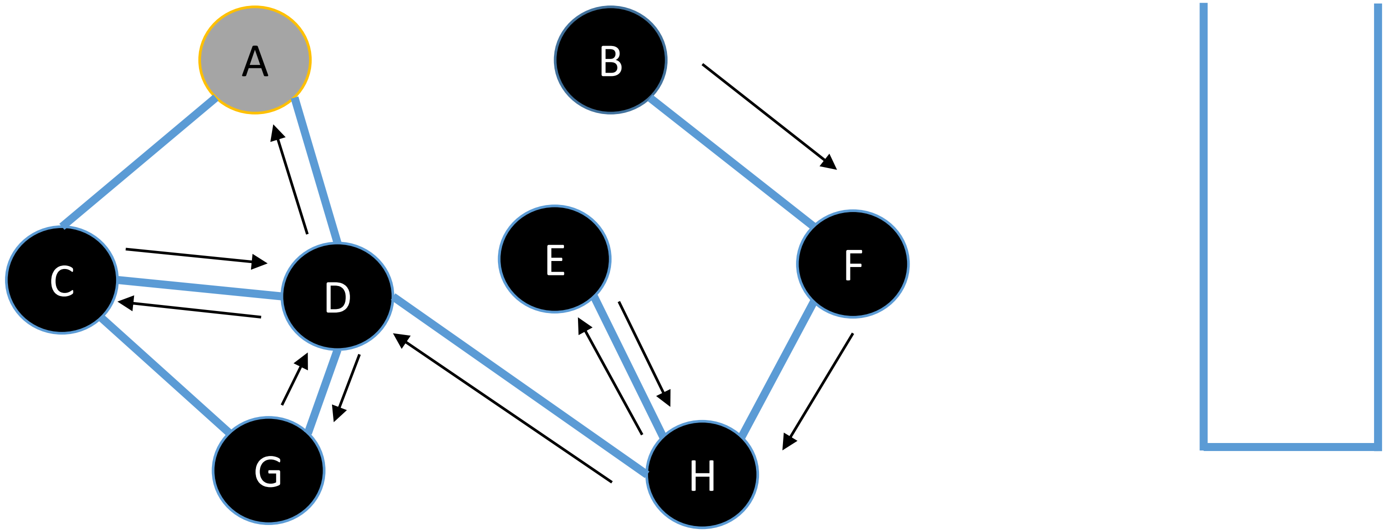
- Next node is popped from the stack
- Order visited: B, F, H, E, D, G, C





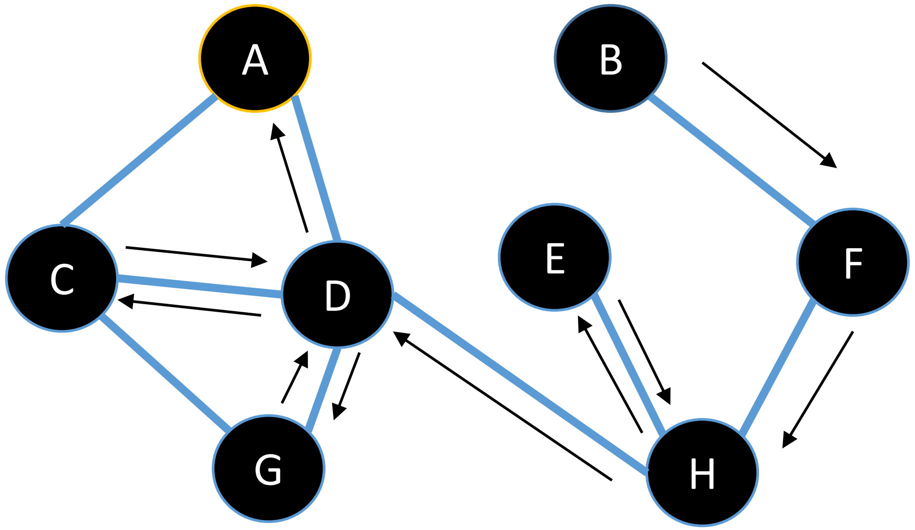
Depth-First Traversal

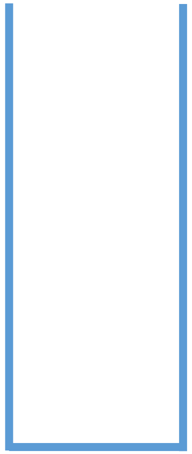
- Add its adjacent white nodes to the stack (there are none)
- Order visited: B, F, H, E, D, G, C



Depth-First Traversal

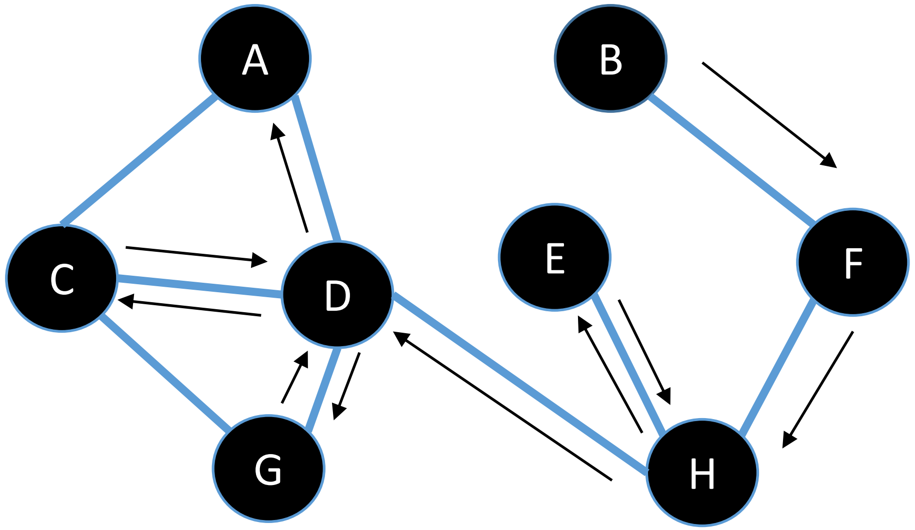
- A has been visited
- Order visited: B, F, H, E, D, G, C, A

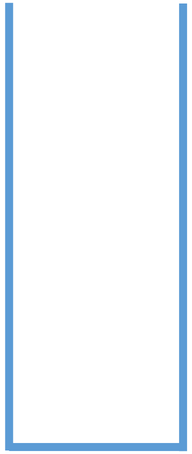




Depth-First Traversal

- Next node is popped from the stack (there are none)
  - Traversal is complete when the stack is empty
- Order visited: **B, F, H, E, D, G, C, A**



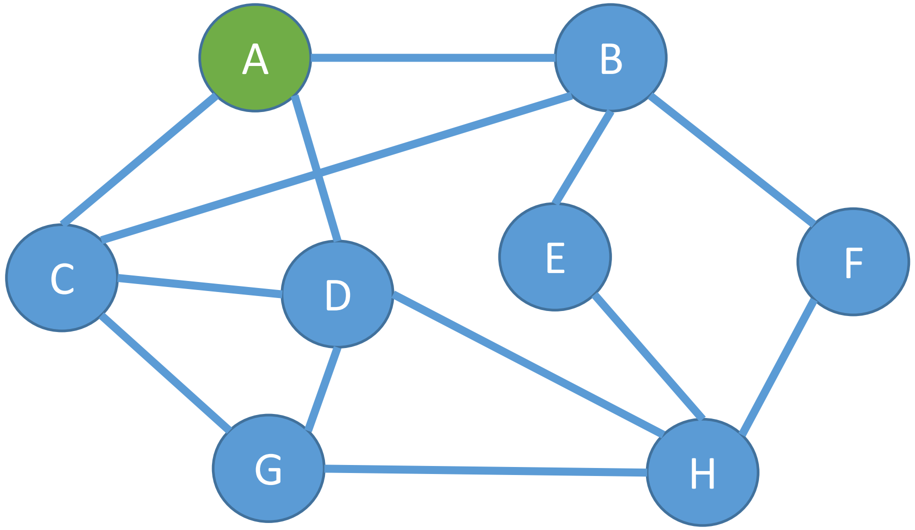


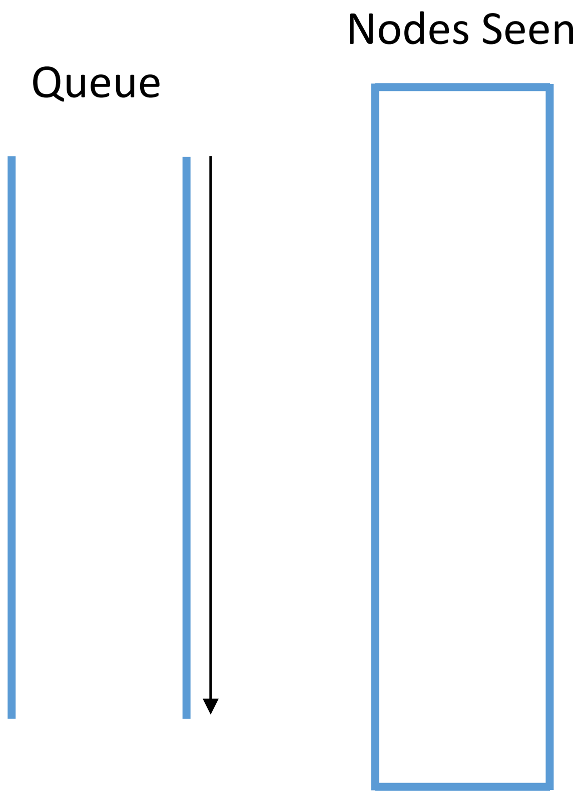
Breadth-First Traversal

- Breadth-first traversal of a graph is similar to the breadth-first traversal of a tree
- Uses a queue to manage the next nodes to visit
- Maintains an array of “seen” nodes to prevent getting stuck in a cycle, if one existed
  - The traversal process is the same for bi-directional graphs or digraphs

Breadth-First Traversal

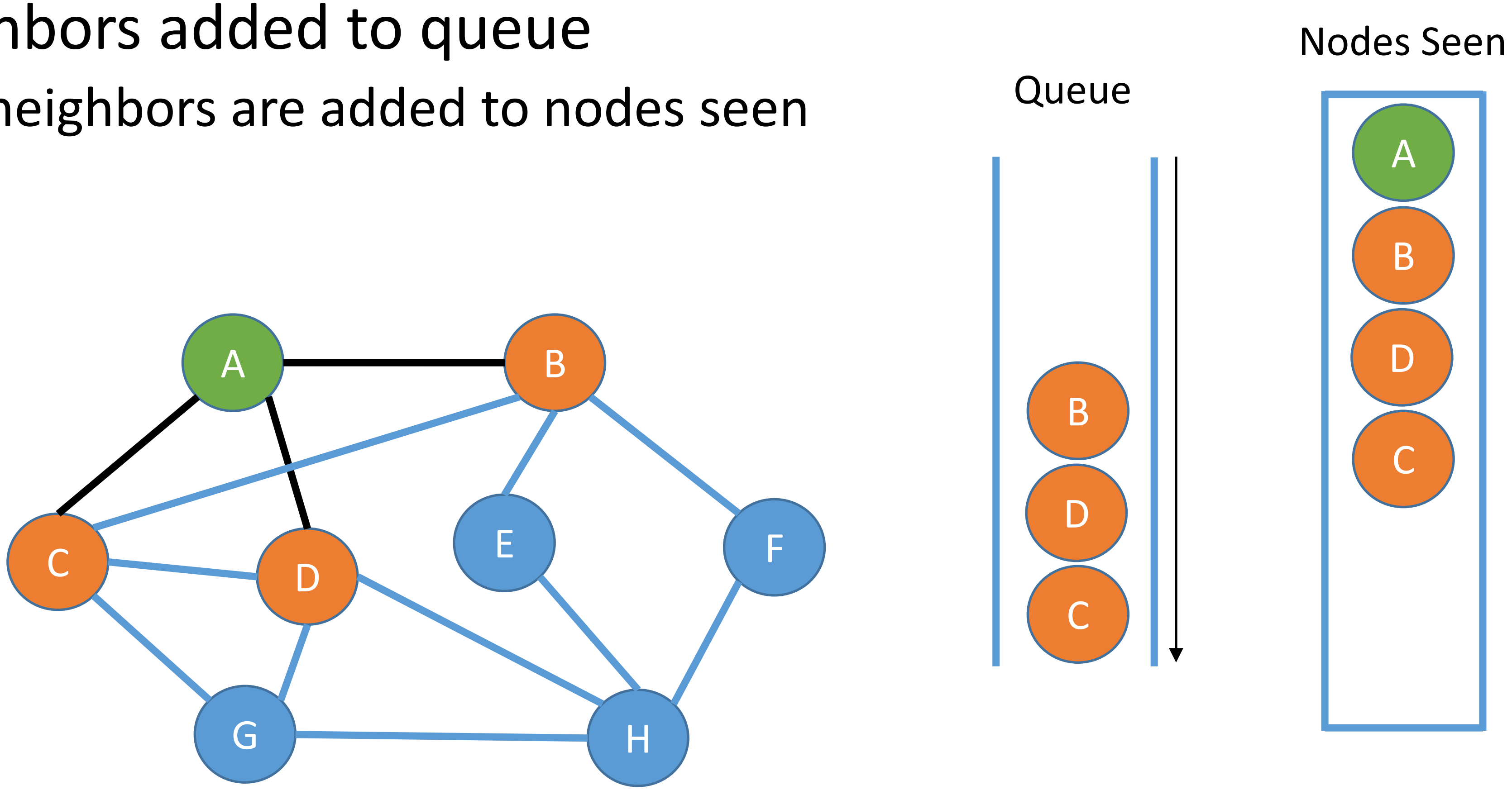
- Starting at A
- Order visited: N/A





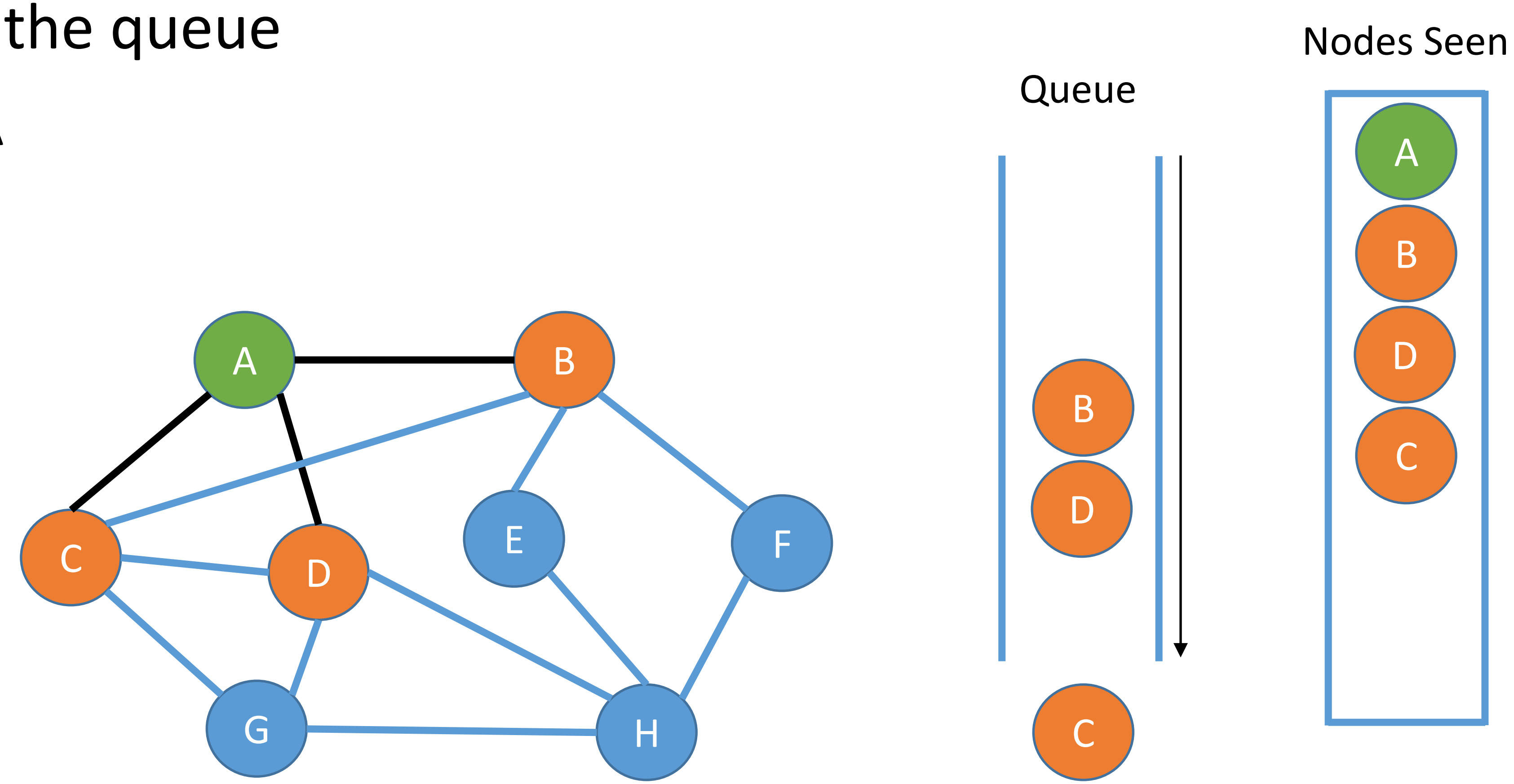
Breadth-First Traversal

- A's unseen neighbors added to queue
  - A and unseen neighbors are added to nodes seen
- Order visited: A



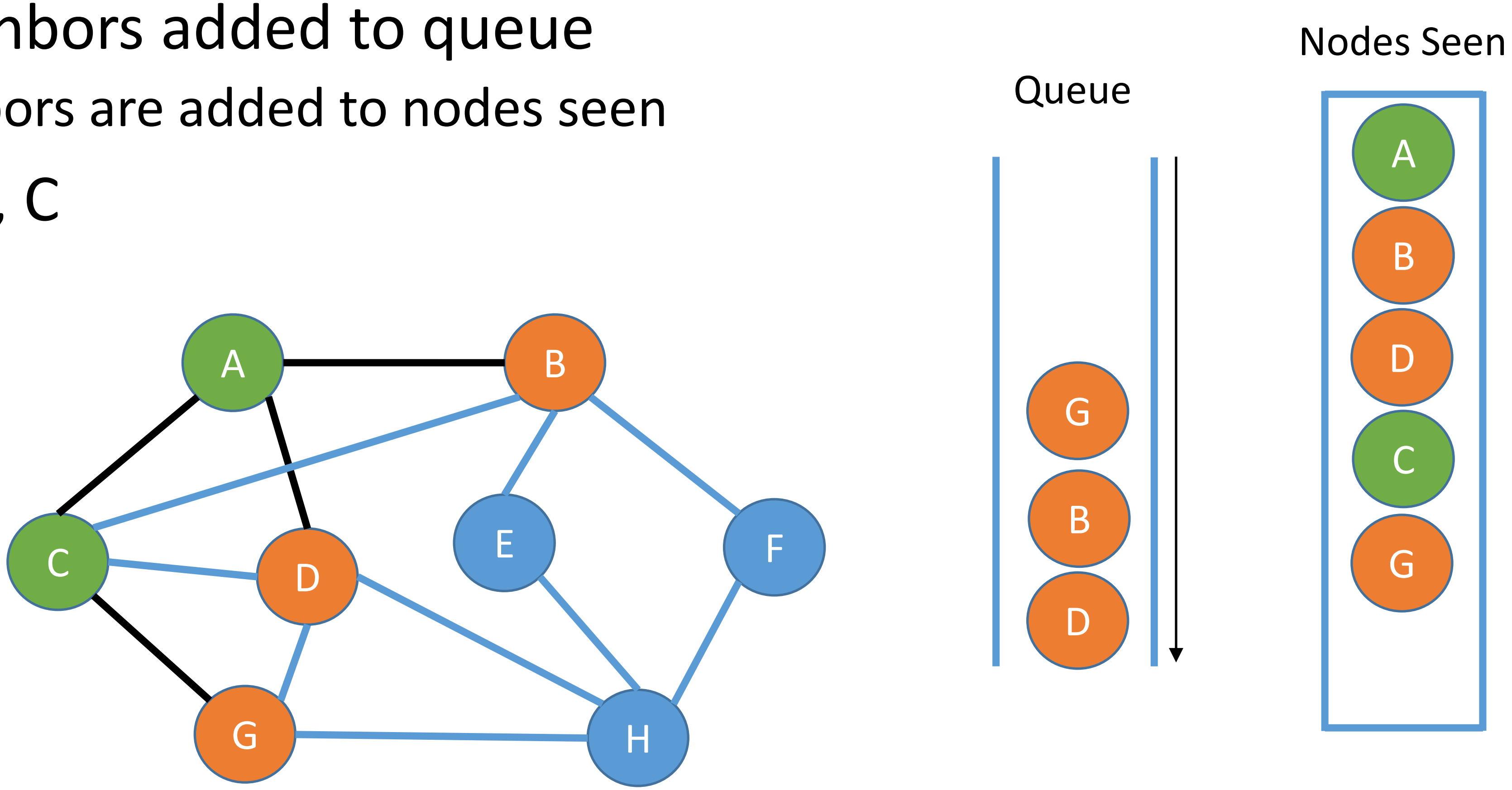
Breadth-First Traversal

- C popped from the queue
- Order visited: A



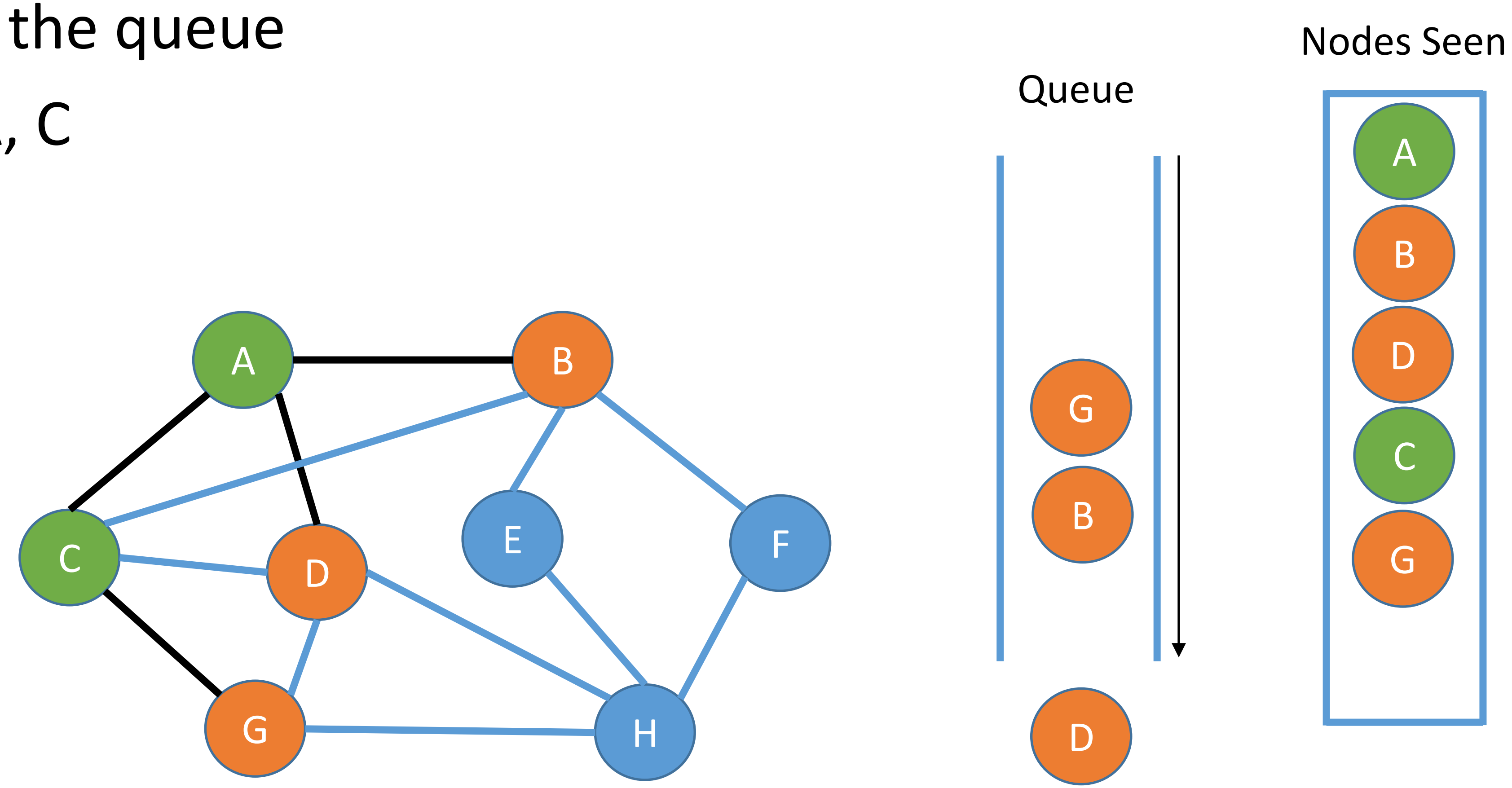
Breadth-First Traversal

- C's unseen neighbors added to queue
  - Unseen neighbors are added to nodes seen
- Order visited: A, C



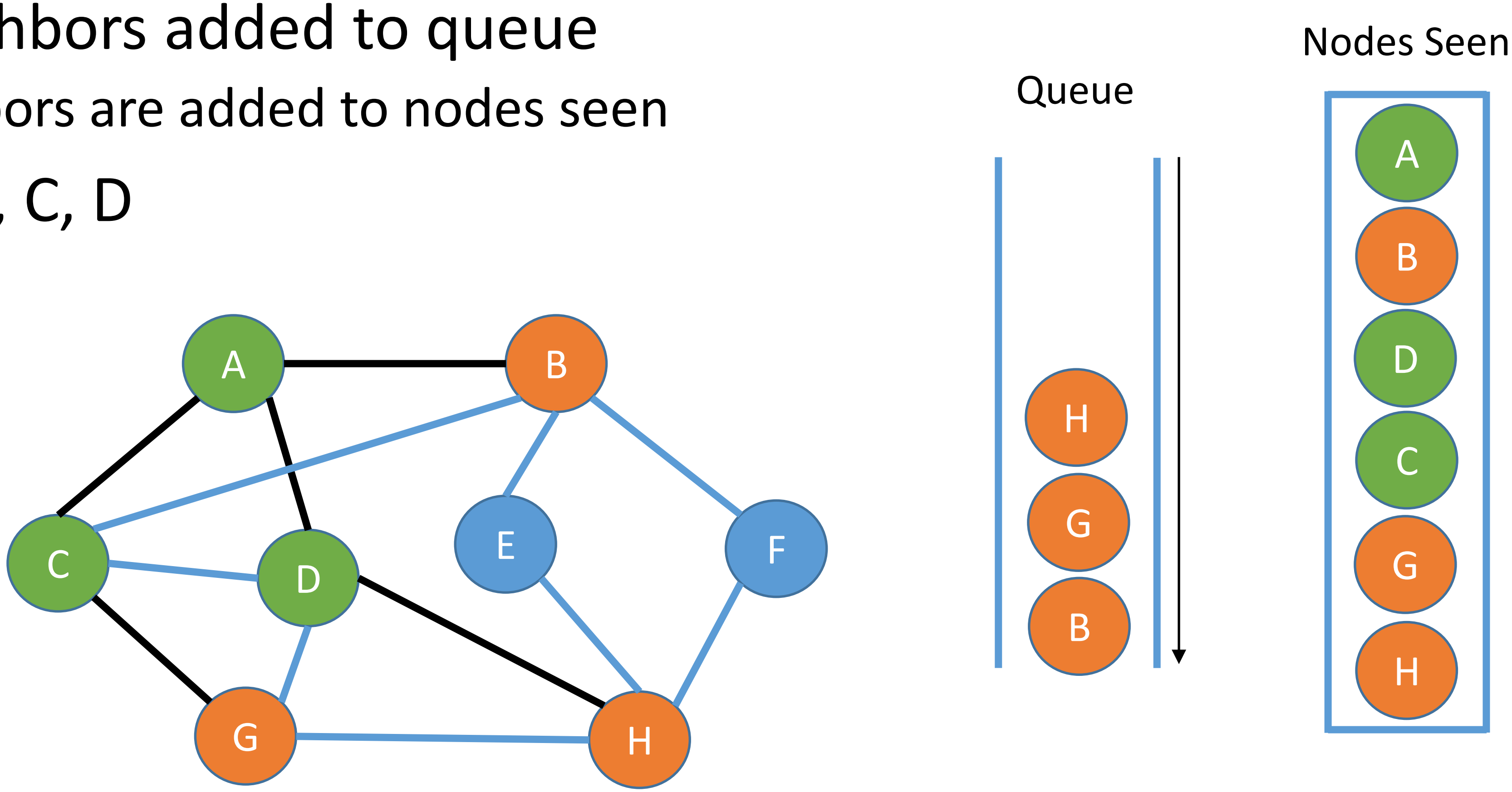
Breadth-First Traversal

- D popped from the queue
- Order visited: A, C



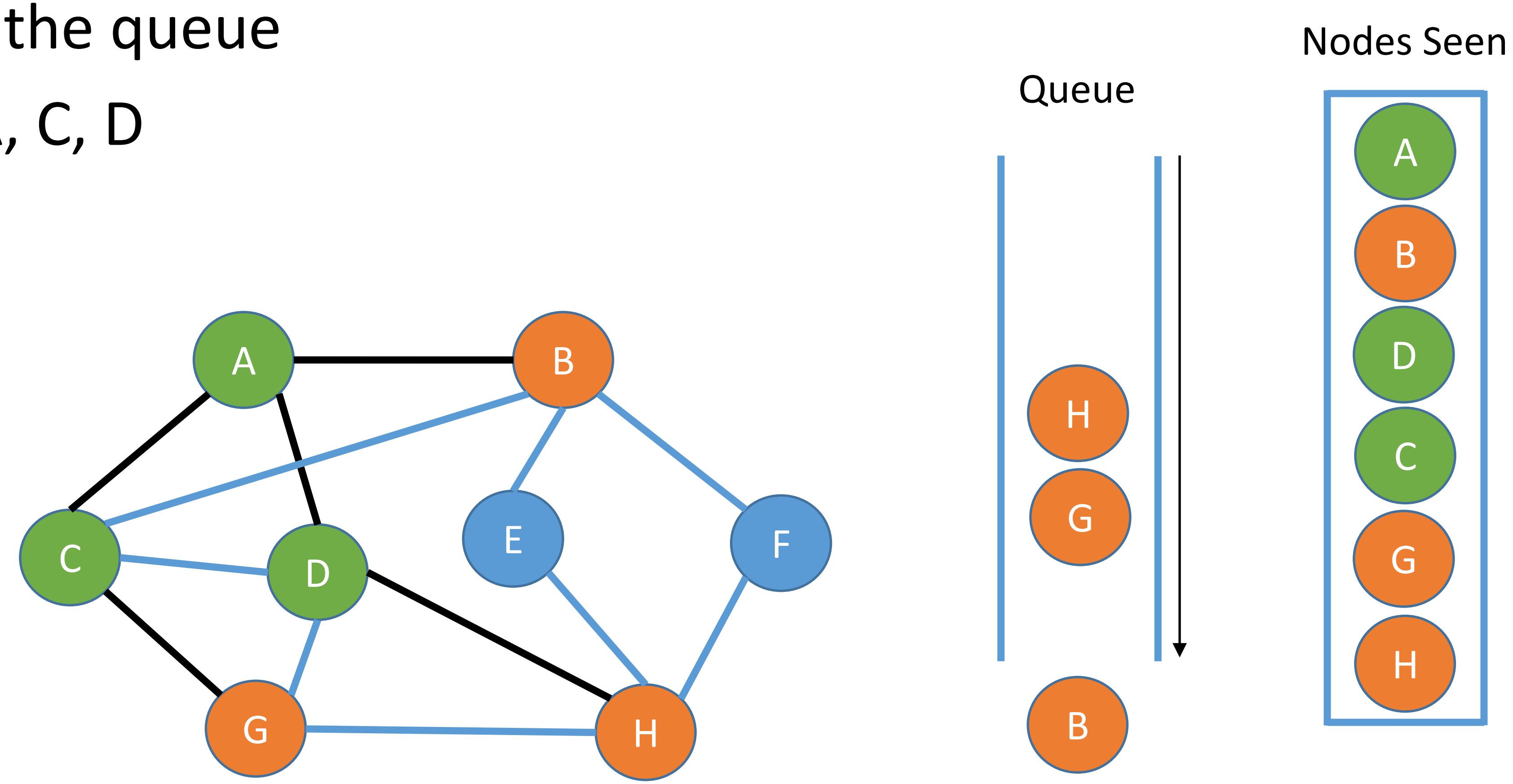
Breadth-First Traversal

- D's unseen neighbors added to queue
  - Unseen neighbors are added to nodes seen
- Order visited: A, C, D



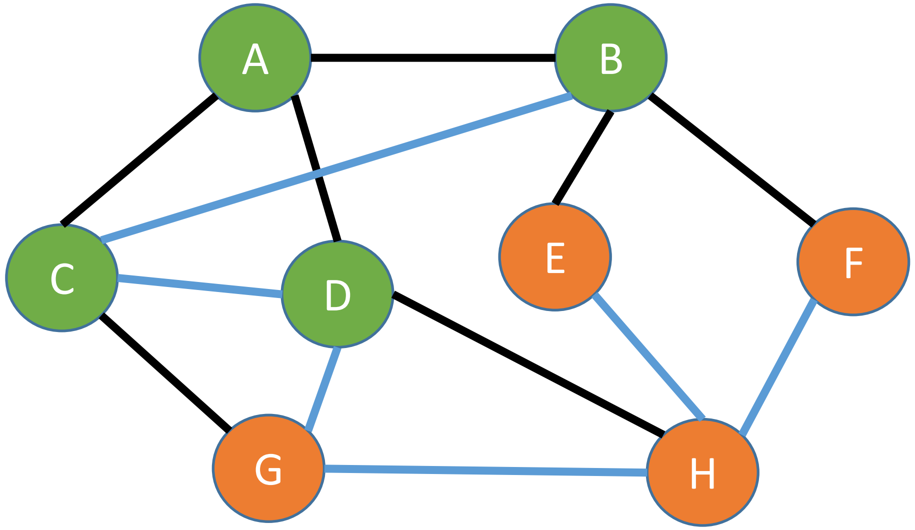
Breadth-First Traversal

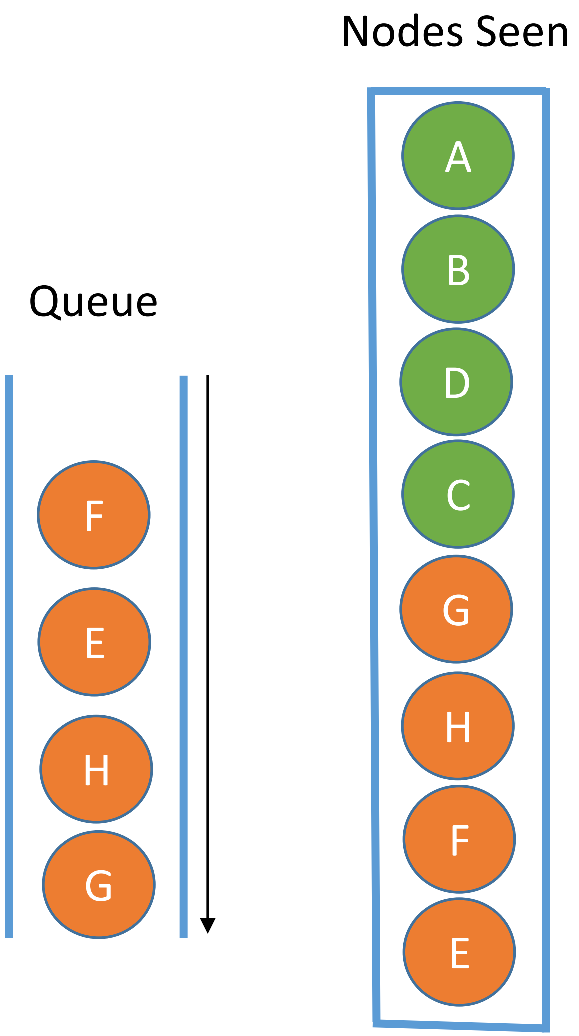
- B popped from the queue
- Order visited: A, C, D



Breadth-First Traversal

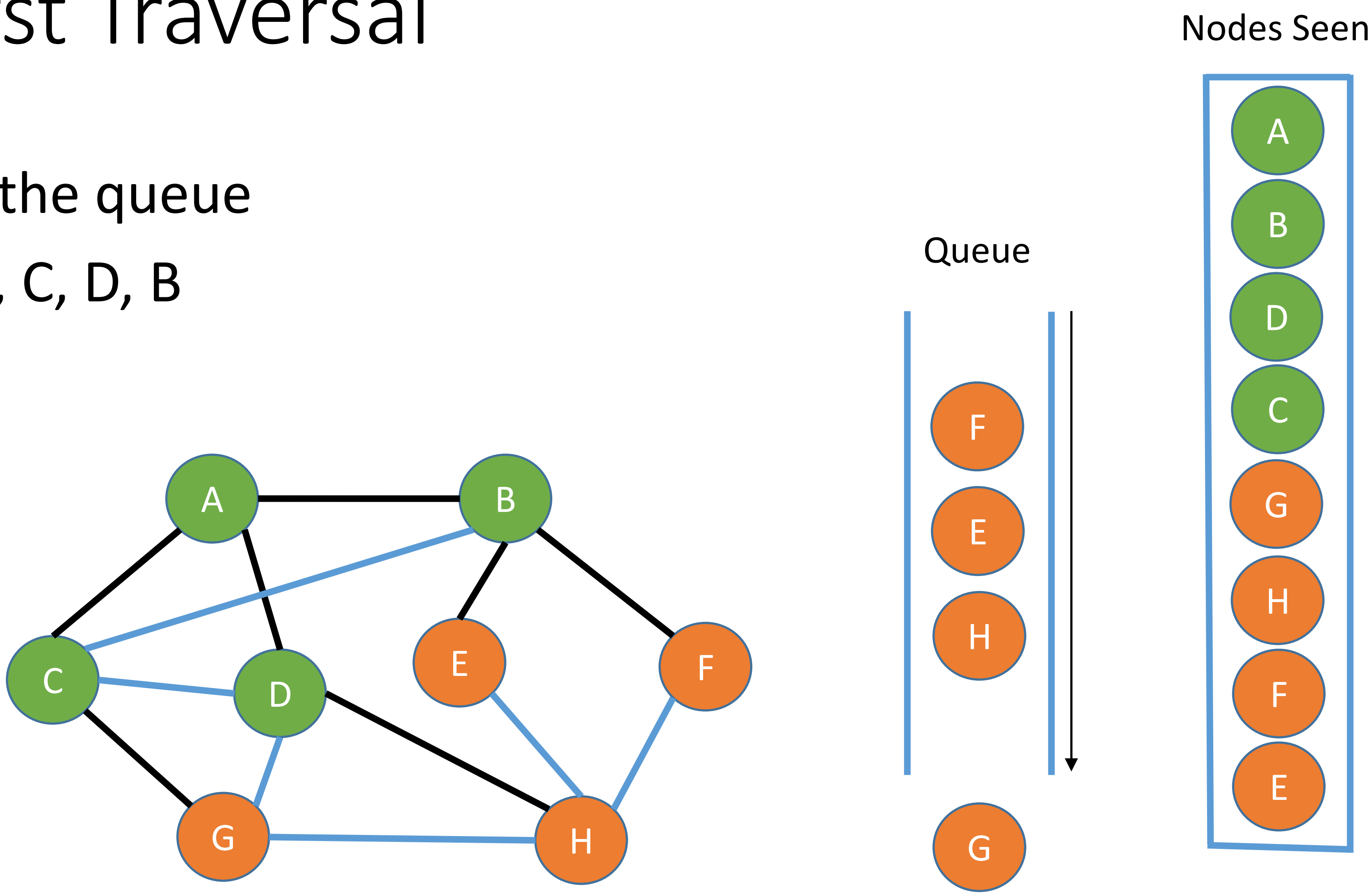
- B’s unseen neighbors added to queue
  - Unseen neighbors are added to nodes seen
- Order visited: A, C, D, B





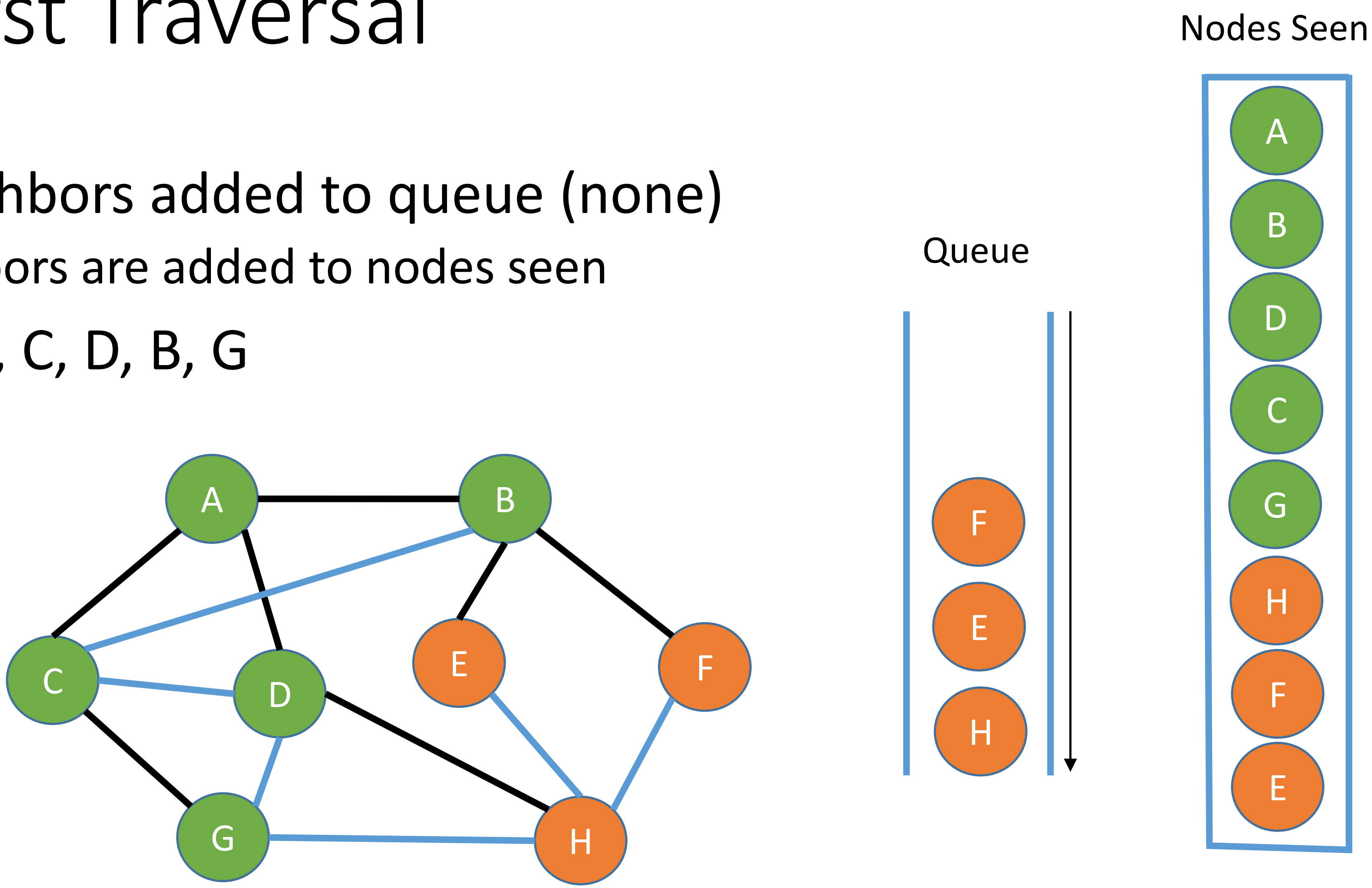
Breadth-First Traversal

- G popped from the queue
- Order visited: A, C, D, B



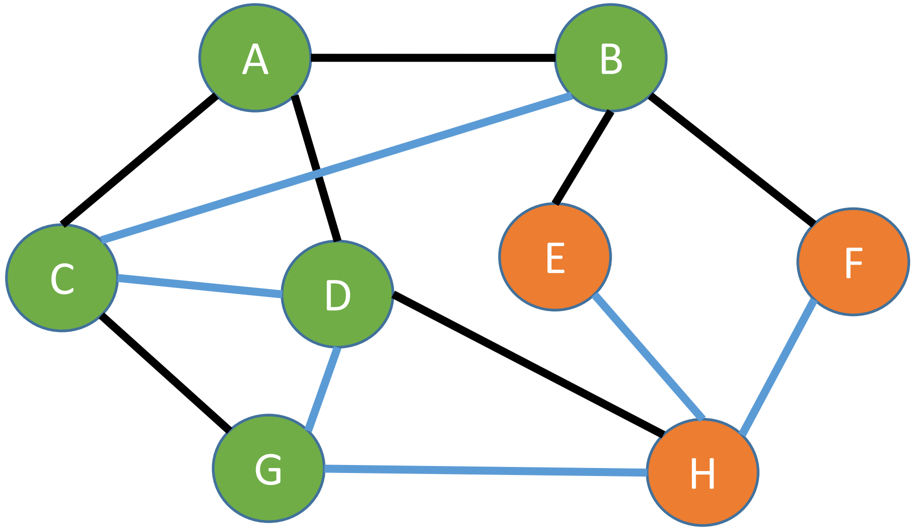
Breadth-First Traversal

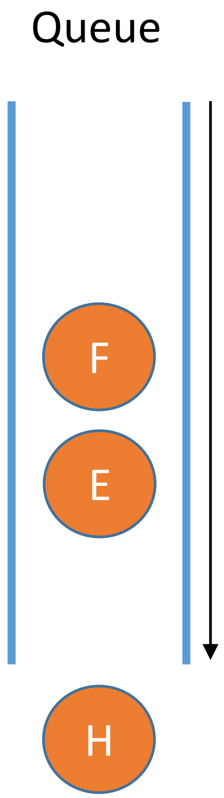
- G's unseen neighbors added to queue (none)
  - Unseen neighbors are added to nodes seen
- Order visited: A, C, D, B, G

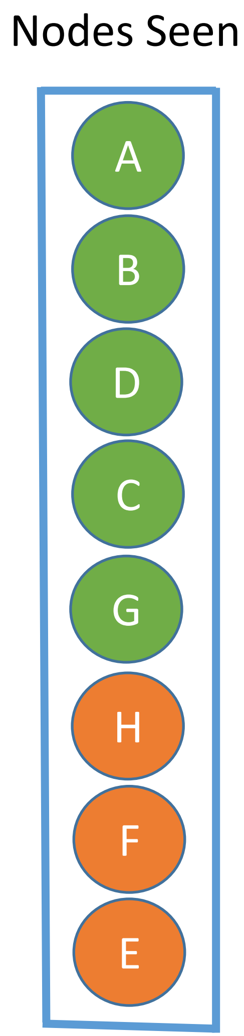


Breadth-First Traversal

- H popped from the queue
- Order visited: A, C, D, B, G

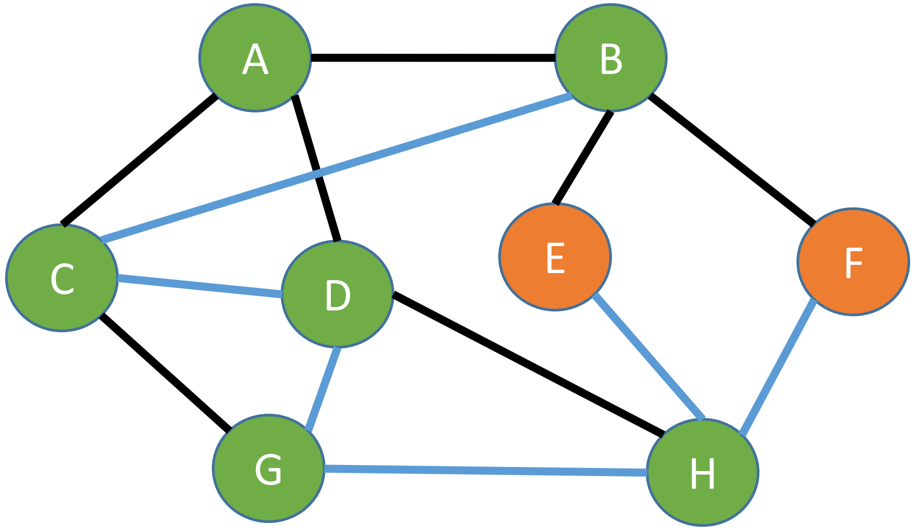


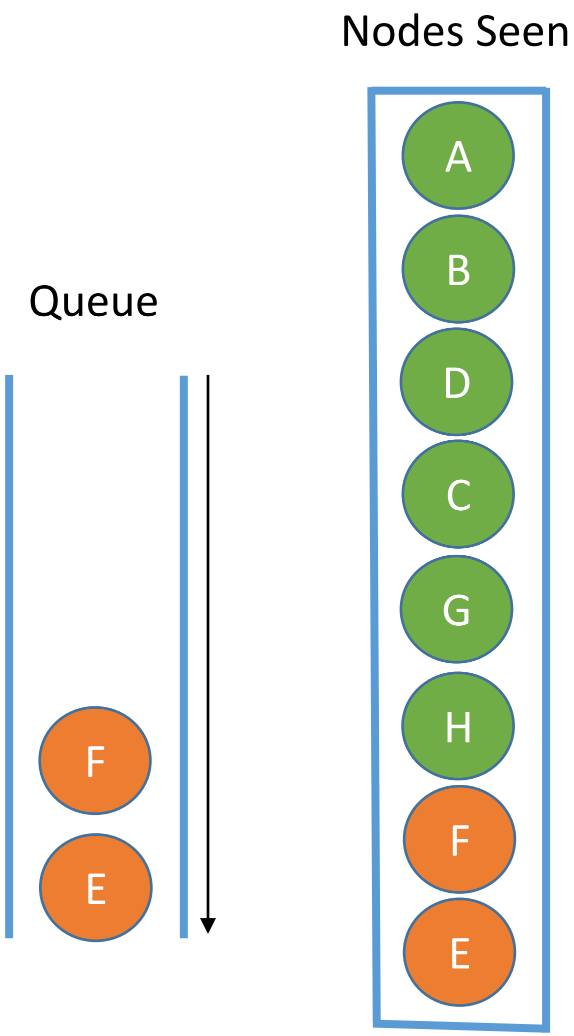




Breadth-First Traversal

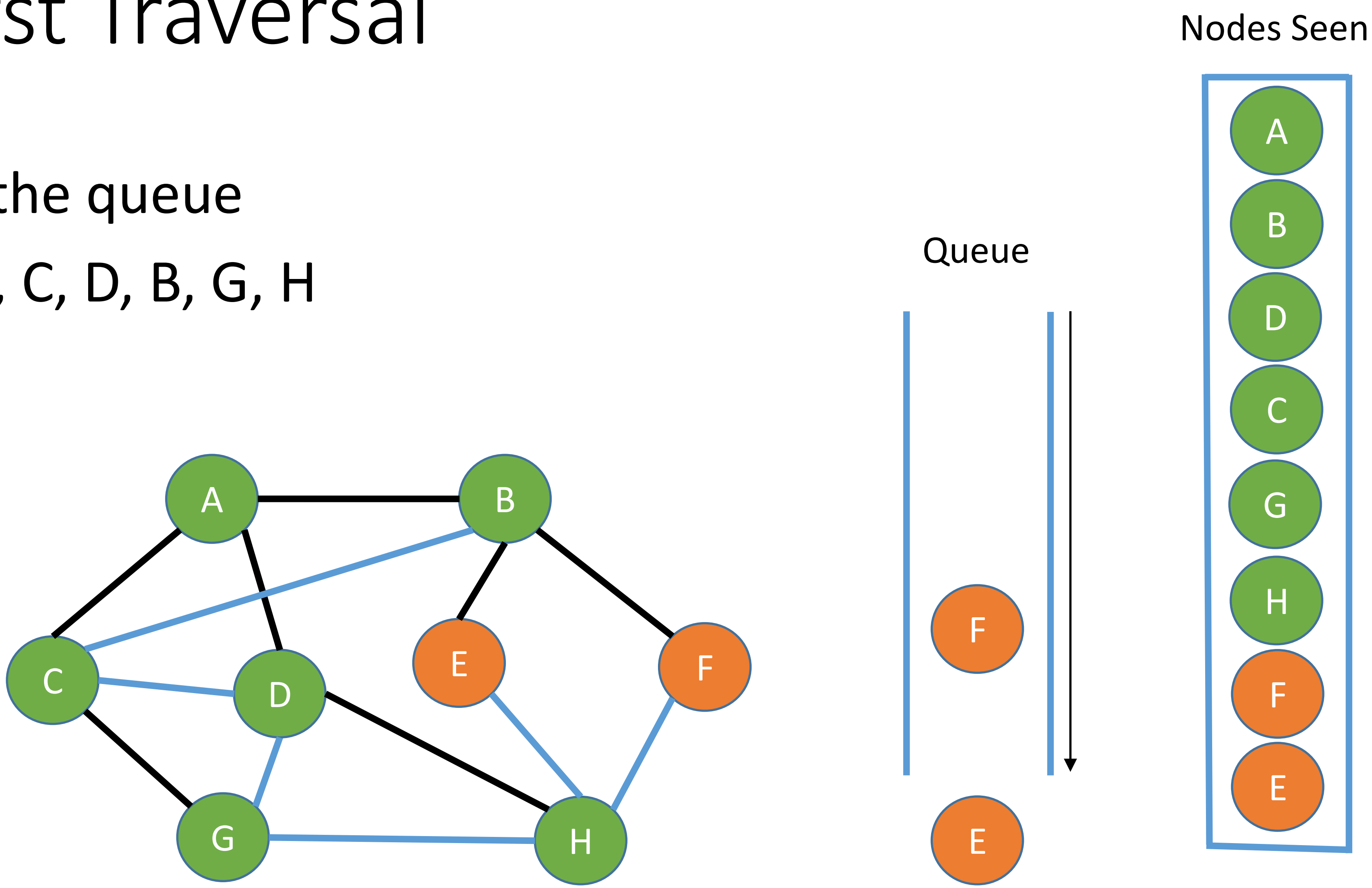
- H's unseen neighbors added to queue (none)
  - Unseen neighbors are added to nodes seen
- Order visited: A, C, D, B, G, H





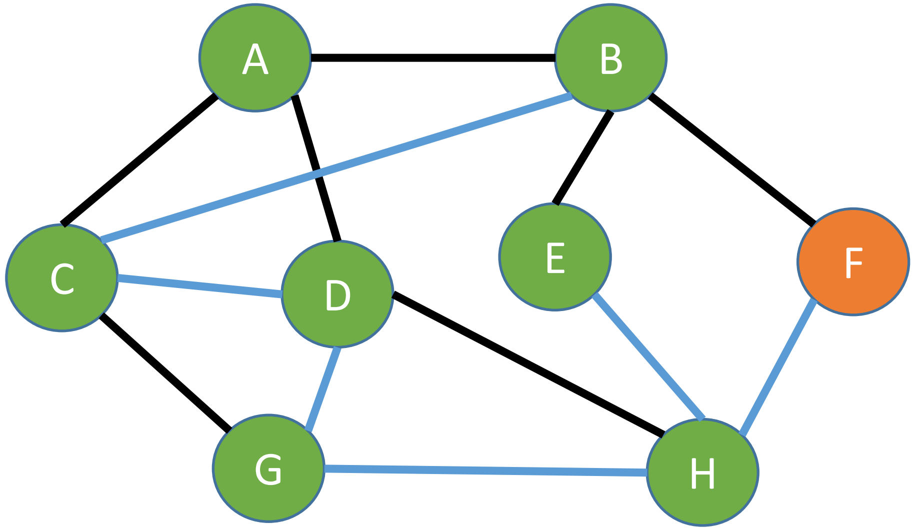
Breadth-First Traversal

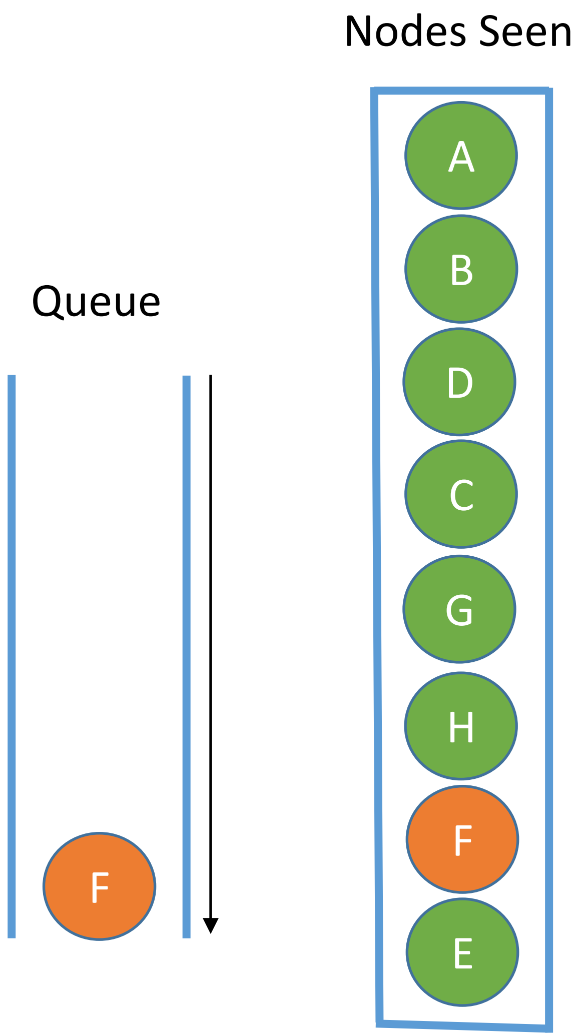
- E popped from the queue
- Order visited: A, C, D, B, G, H



Breadth-First Traversal

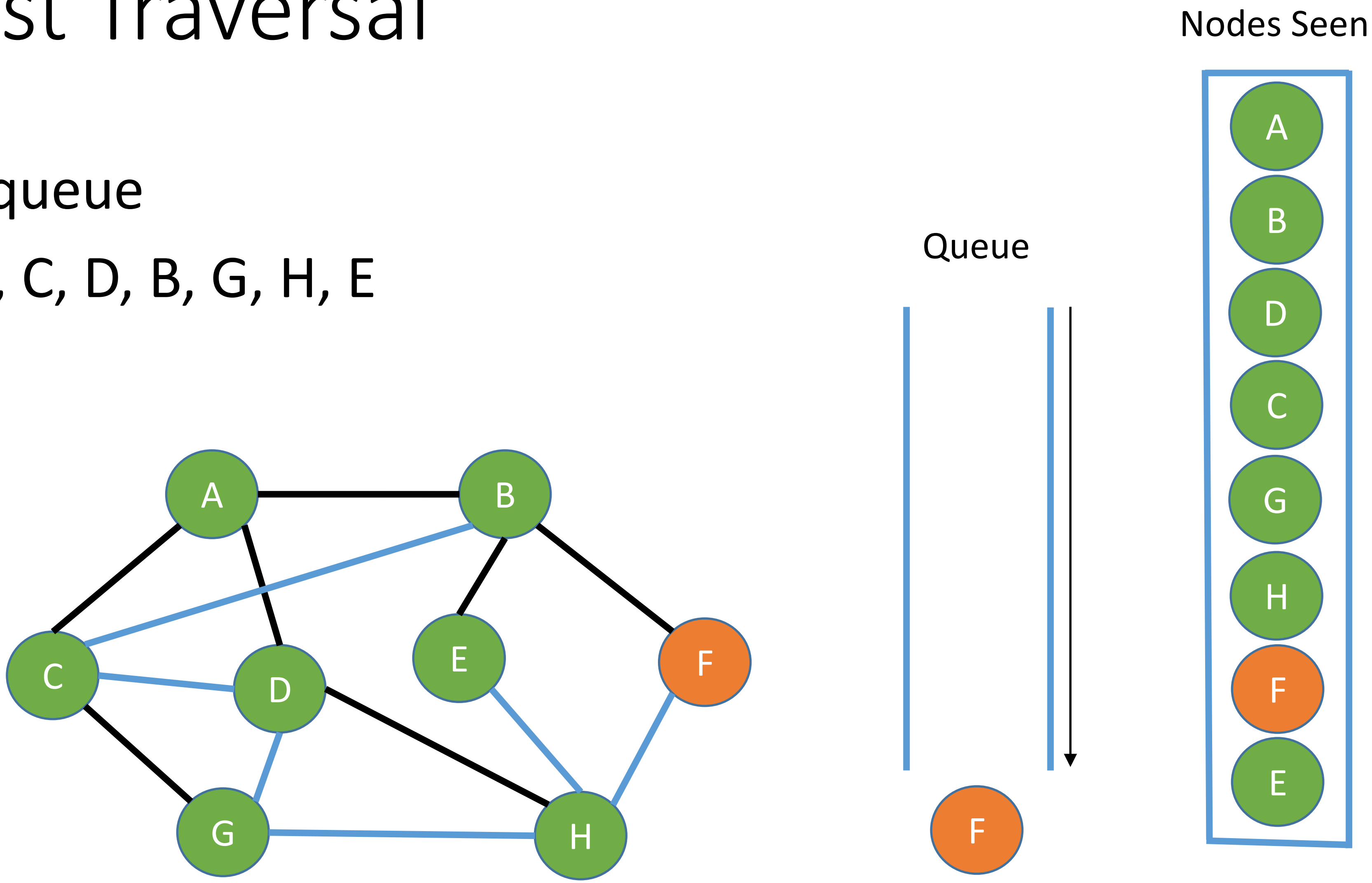
- E’s unseen neighbors added to queue (none)
  - Unseen neighbors are added to nodes seen
- Order visited: A, C, D, B, G, H, E





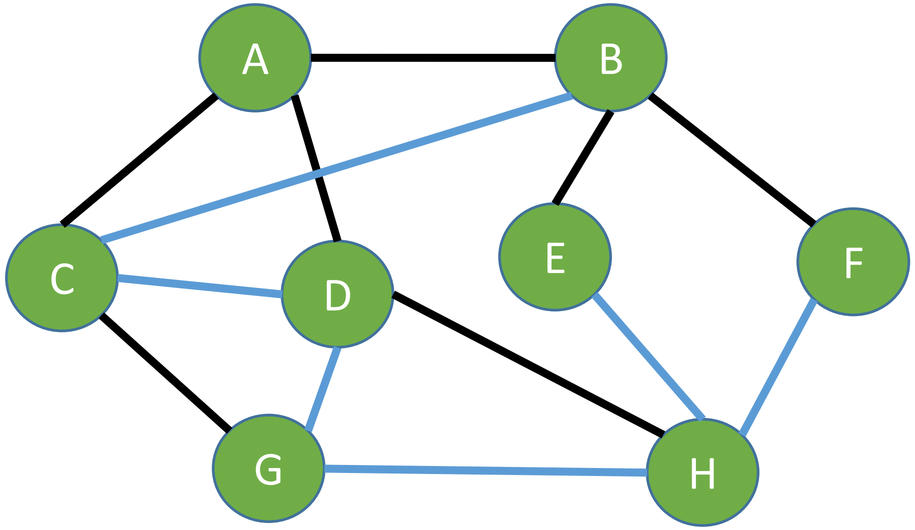
Breadth-First Traversal

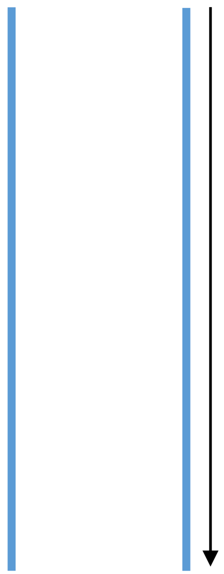
- F popped from queue
- Order visited: A, C, D, B, G, H, E

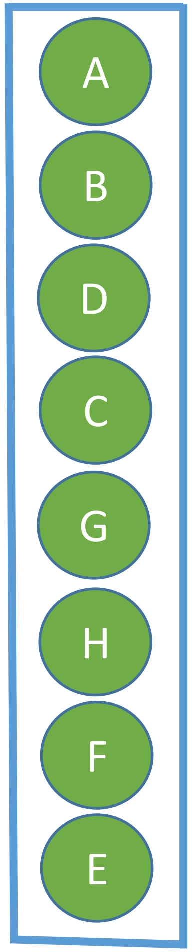


Breadth-First Traversal

- F’s unseen neighbors added to queue (none)
  - Unseen neighbors are added to nodes seen
- Order visited: A, C, D, B, G, H, E, F

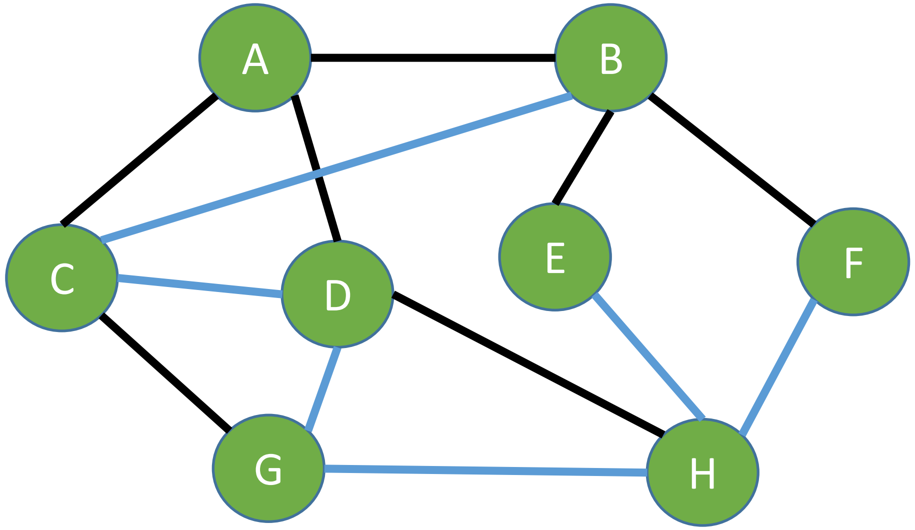


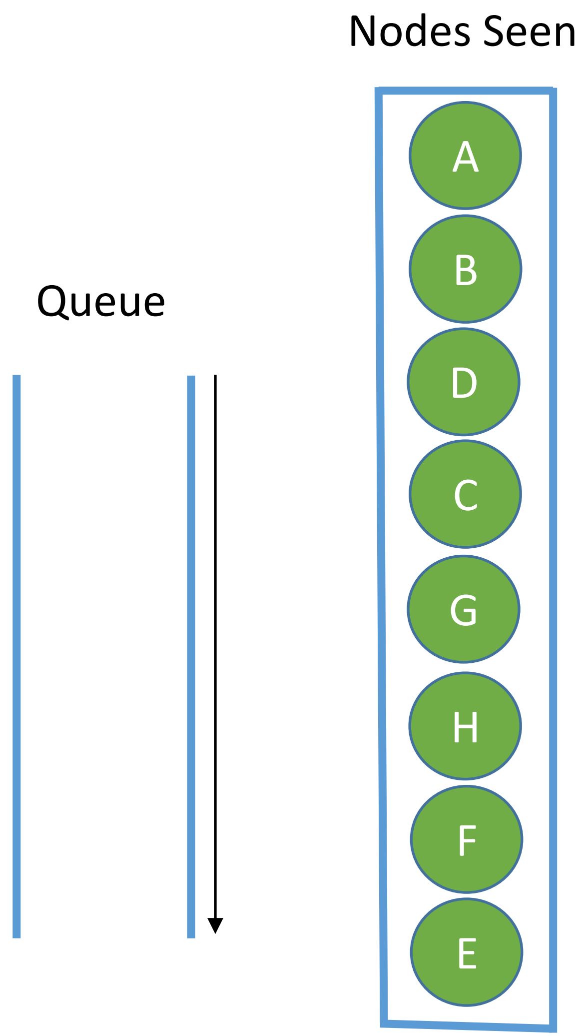




Breadth-First Traversal

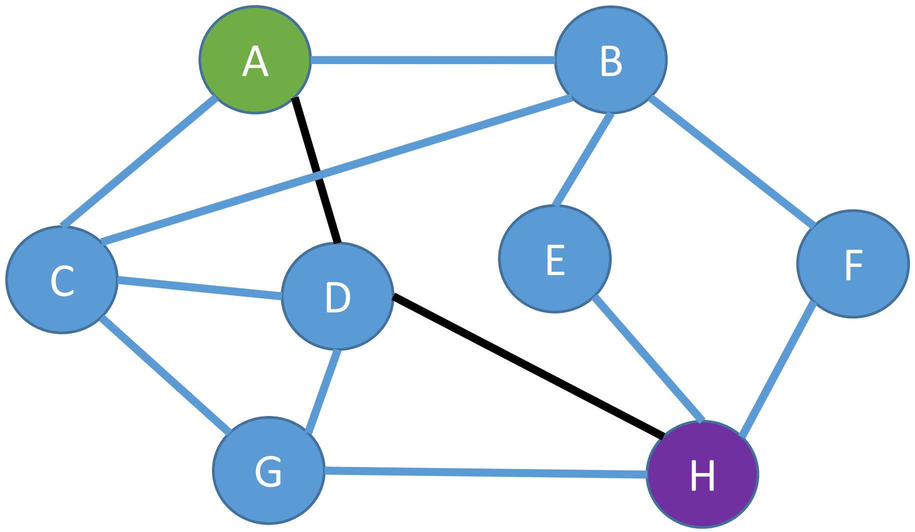
- Queue is empty
  - Traversal is complete
- Order visited: **A, C, D, B, G, H, E, F**





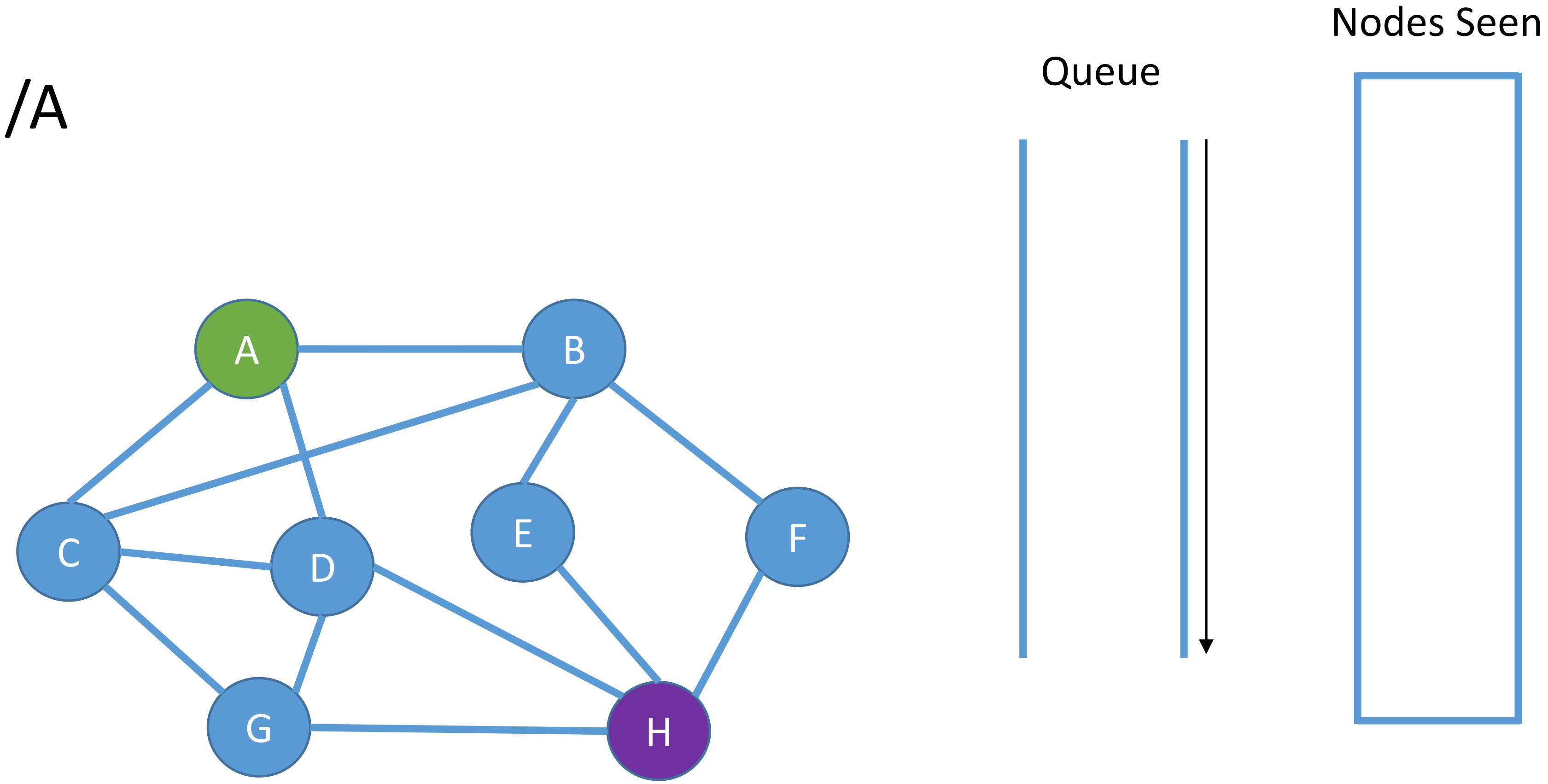
Finding Distance with Breadth-First

- Based on the last example, its easy to see a breadth-first traversal can be used to find the distance (shortest path) between two nodes
- As an example, we'll use a breadth-first traversal to find the distance between A and H



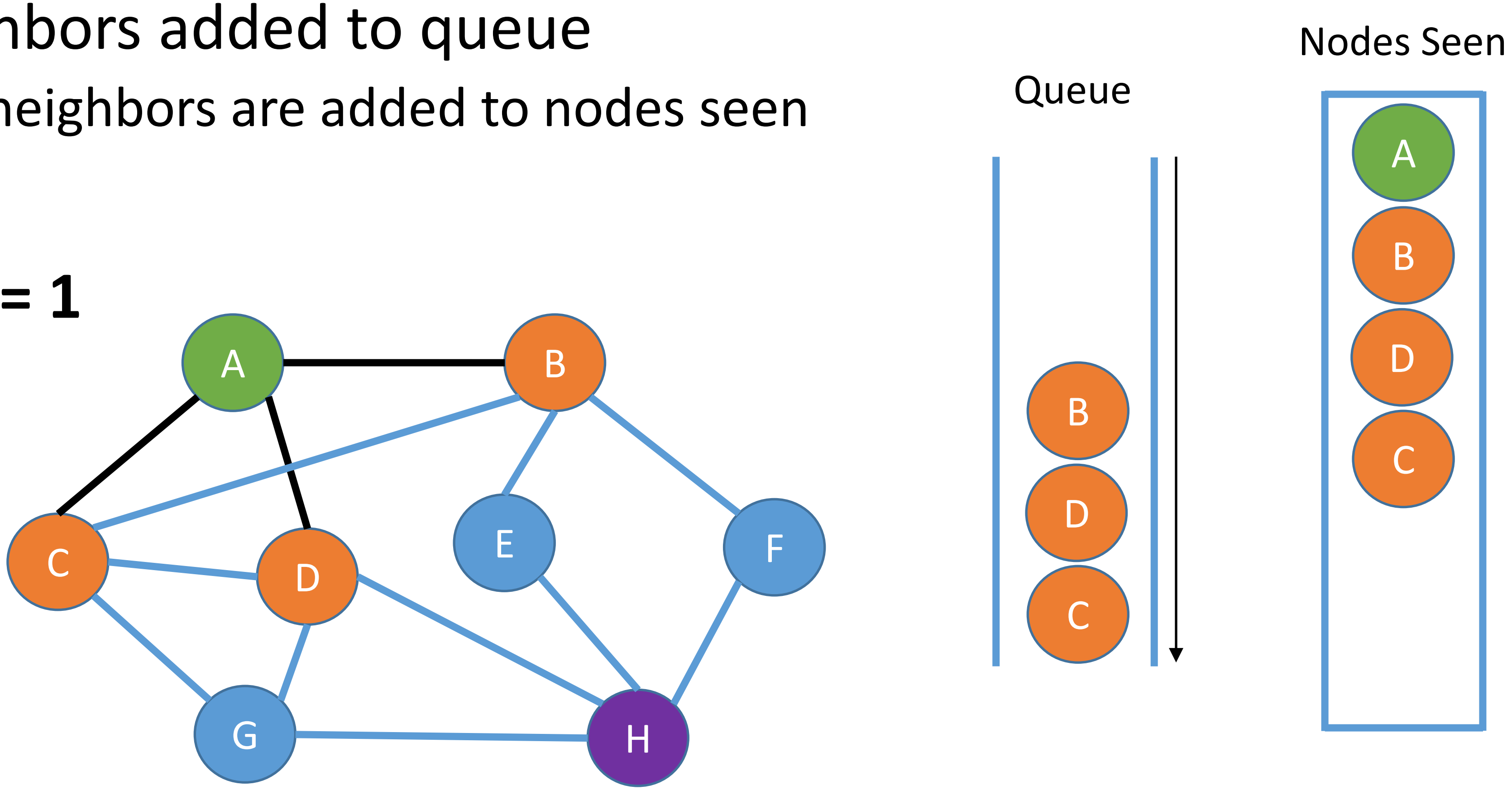
Finding Distance with Breadth-First

- Starting at A
- Order visited: N/A



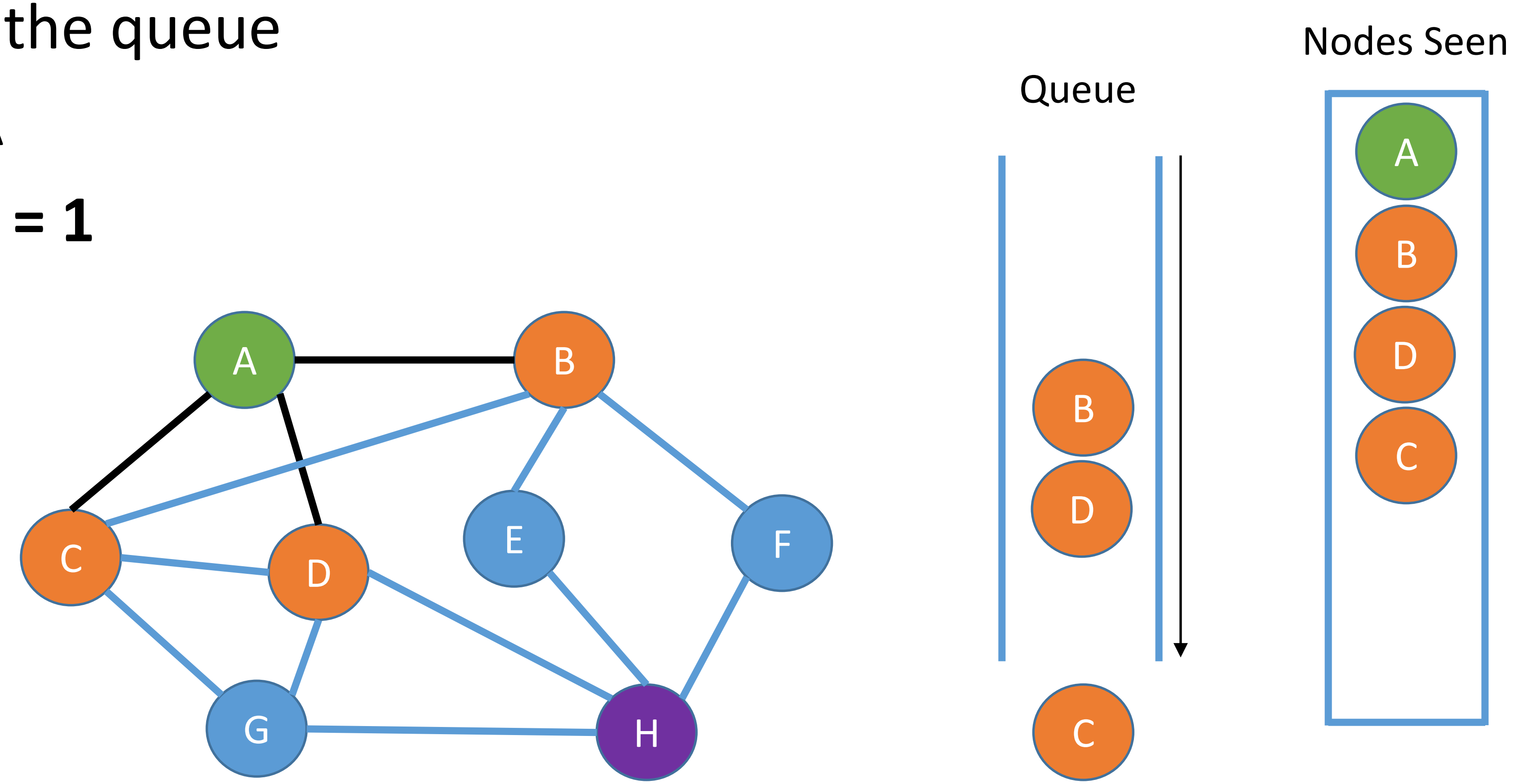
Finding Distance with Breadth-First

- A's unseen neighbors added to queue
  - A and unseen neighbors are added to nodes seen
- Order visited: A
- **DEEPEST LEVEL = 1**



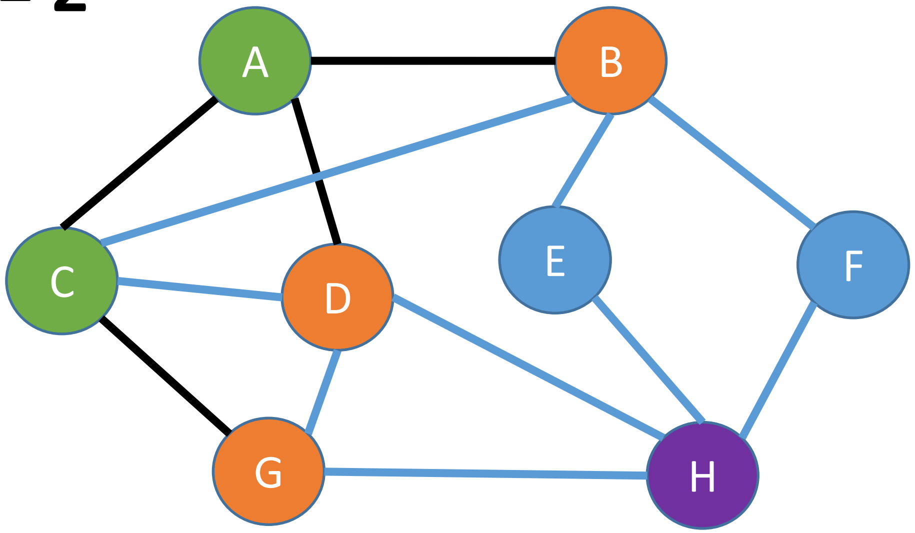
Finding Distance with Breadth-First

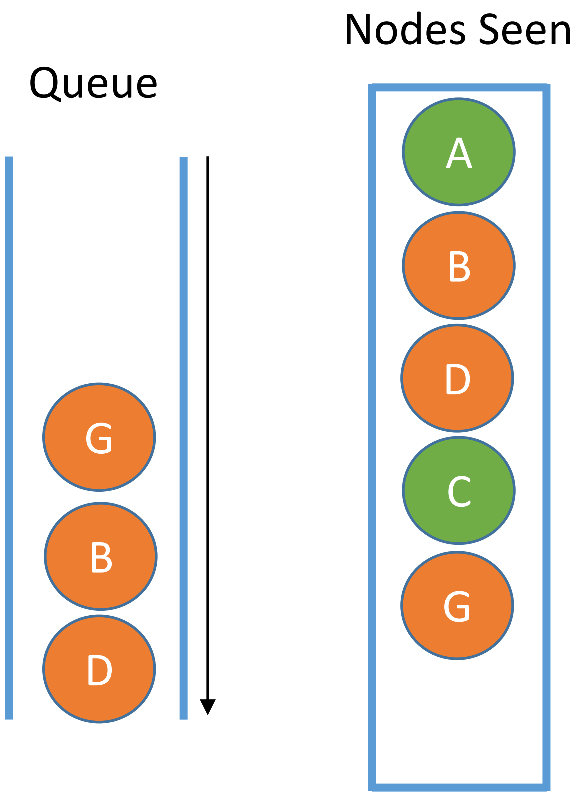
- C popped from the queue
- Order visited: A
- **DEEPEST LEVEL = 1**



Finding Distance with Breadth-First

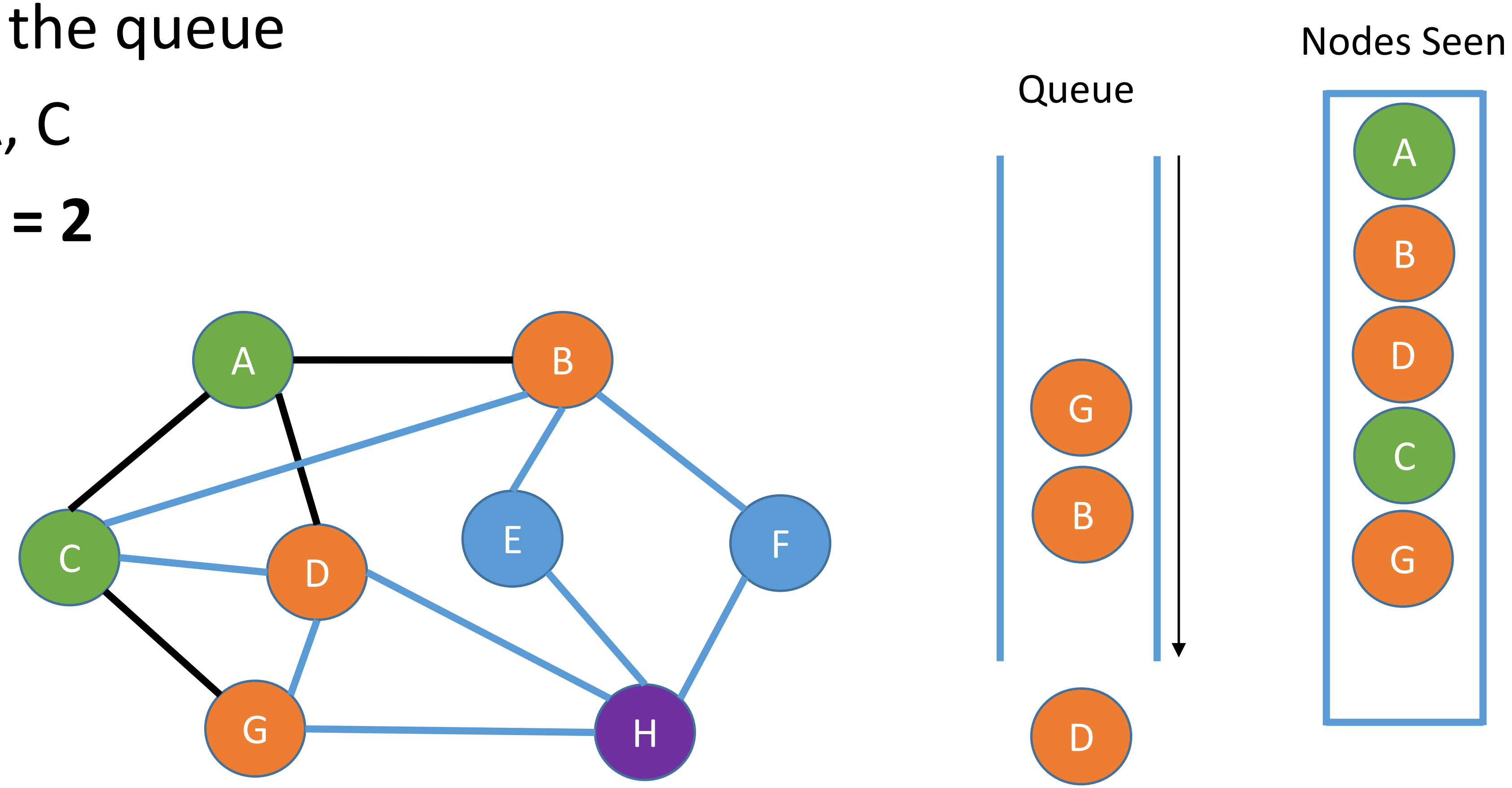
- C’s unseen neighbors added to queue
  - Unseen neighbors are added to nodes seen
- Order visited: A, C
- **DEEPEST LEVEL = 2**





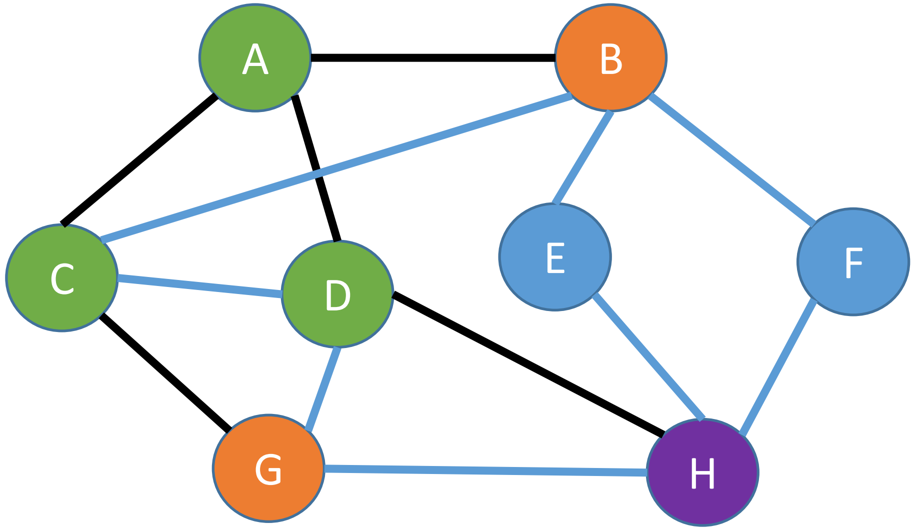
Finding Distance with Breadth-First

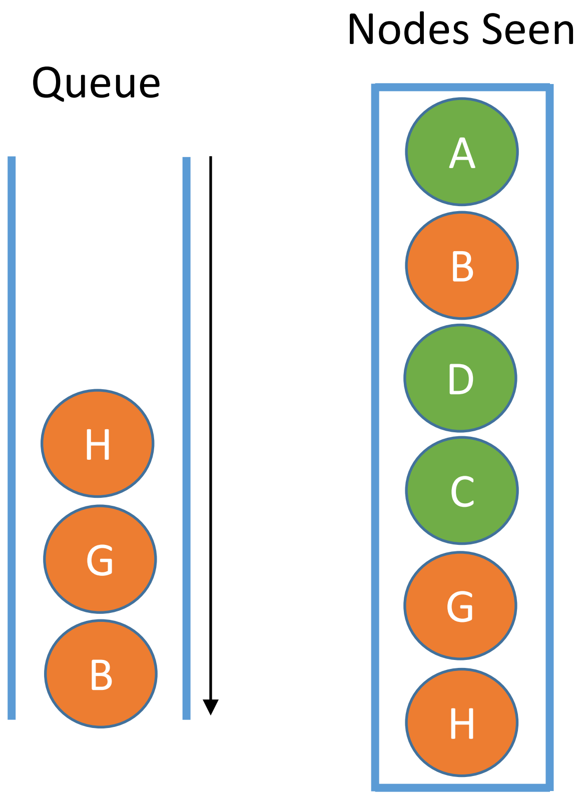
- D popped from the queue
- Order visited: A, C
- **DEEPEST LEVEL = 2**



Finding Distance with Breadth-First

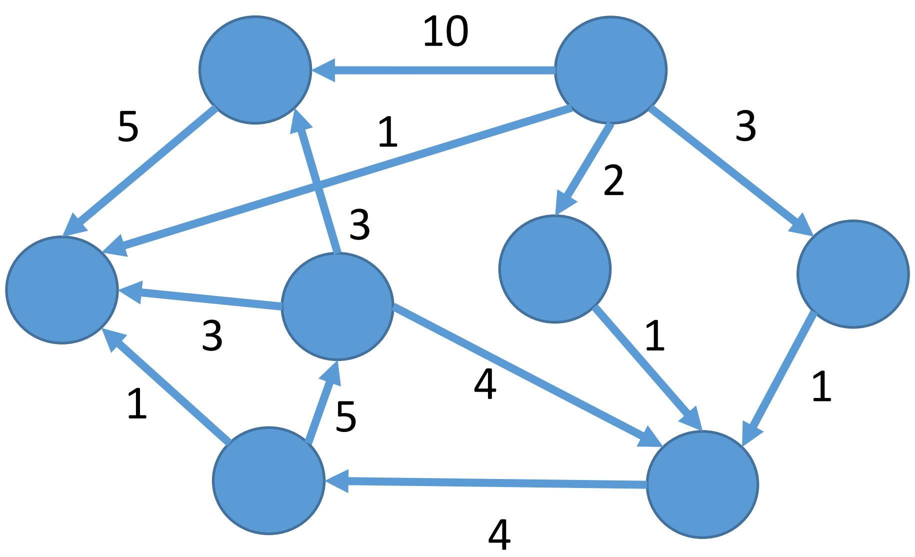
- D’s unseen neighbors added to queue
  - <u>H was reached</u>
  - Traversal stops
- **DEEPEST LEVEL = 2**
  - **<u>DISTANCE = 2</u>**





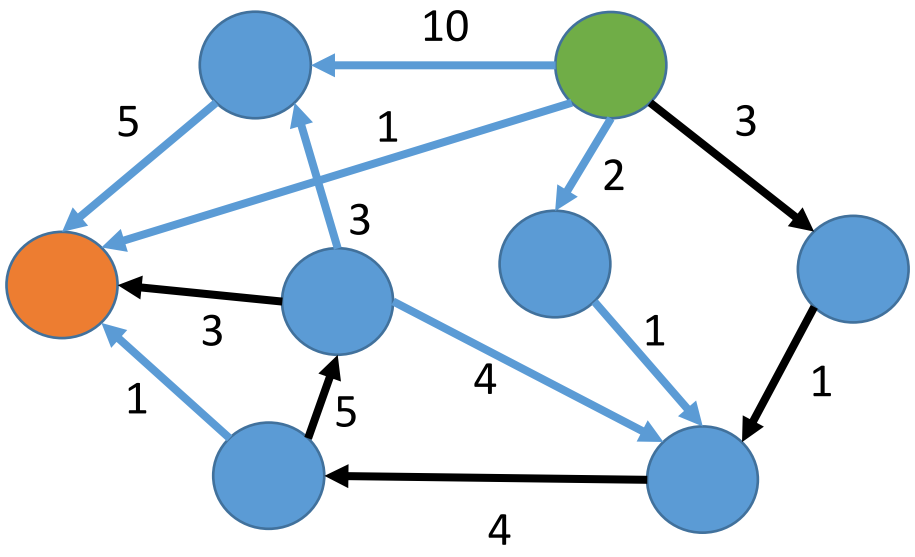
Weighted Graphs

- A **weighted graph** is a graph where each edge has a weight or cost.
  - Weighted graphs can be undirected or directed



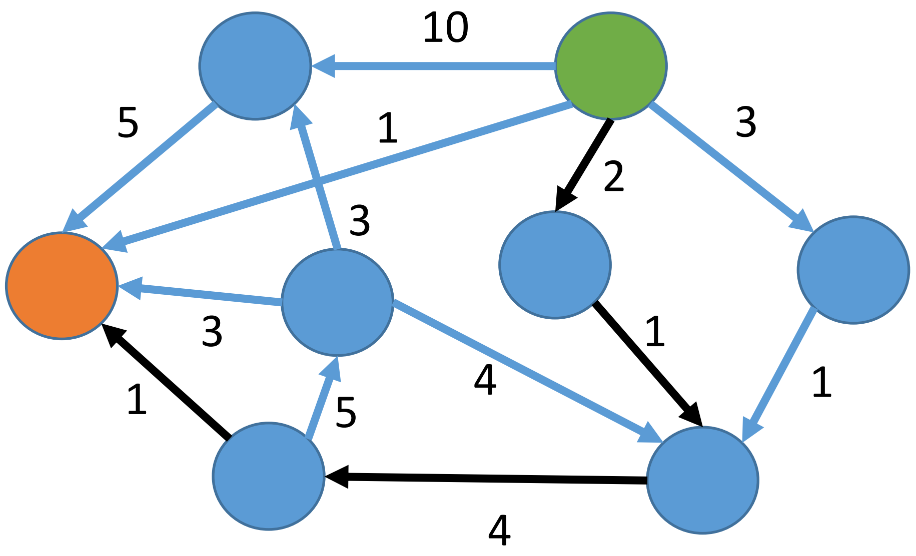
Weighted Graphs

- The path length of a weighted graph is the sum of the edge costs.
  - 3+1+4+5+3 = **16**

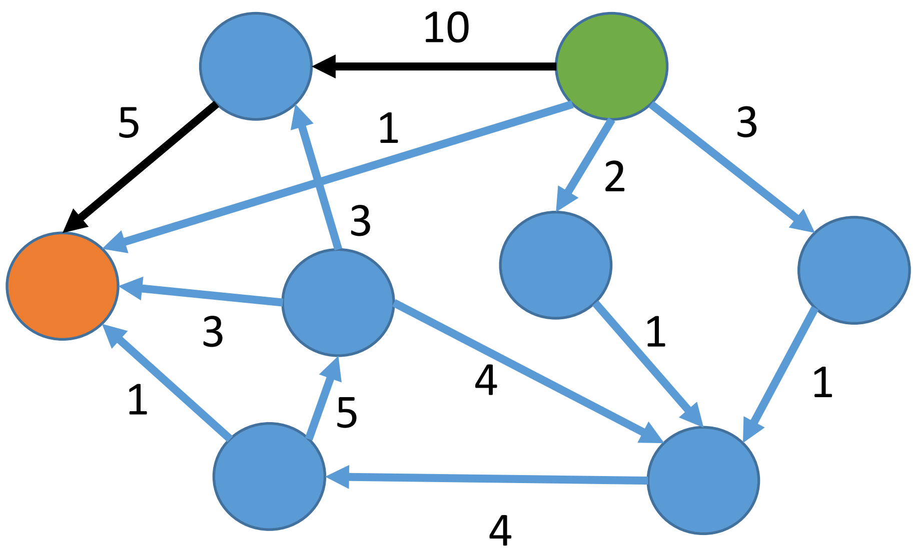


Weighted Graphs

Unweighted Length = 4
Weighted Length = 8



Unweighted Length = 2
Weighted Length = 15



- The first path is less costly than the second path, despite it being twice as long

Weighted Graphs

- A breadth-first traversal would not be useful for finding the path with the least cost

- Other algorithms are used to find the path with the least cost between two nodes

- The most well known is Djikstra’s Algorithm

Dijkstra’s Algorithm

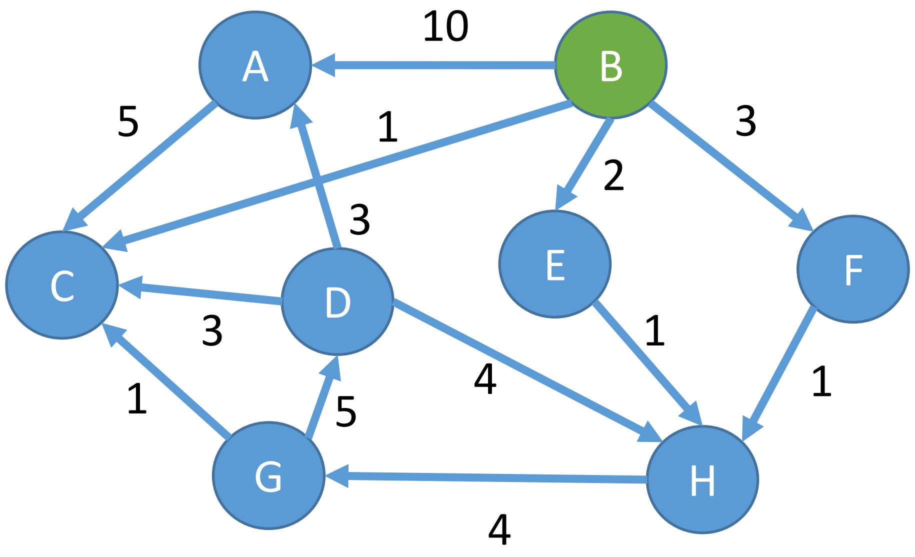
- This algorithm finds the shortest path between one node and **every other node** in a graph.

- *For each vertex*
  - The algorithm determines the vertex’s distance (shortest/least costly path) from the starting vertex
  - The algorithm determines the vertex’s predecessor pointer- the previous vertex with the shortest (or least costly) path from the starting vertex

- Can be used on:
  - Bi-directional and digraphs
  - Weighted graphs and unweighted graphs

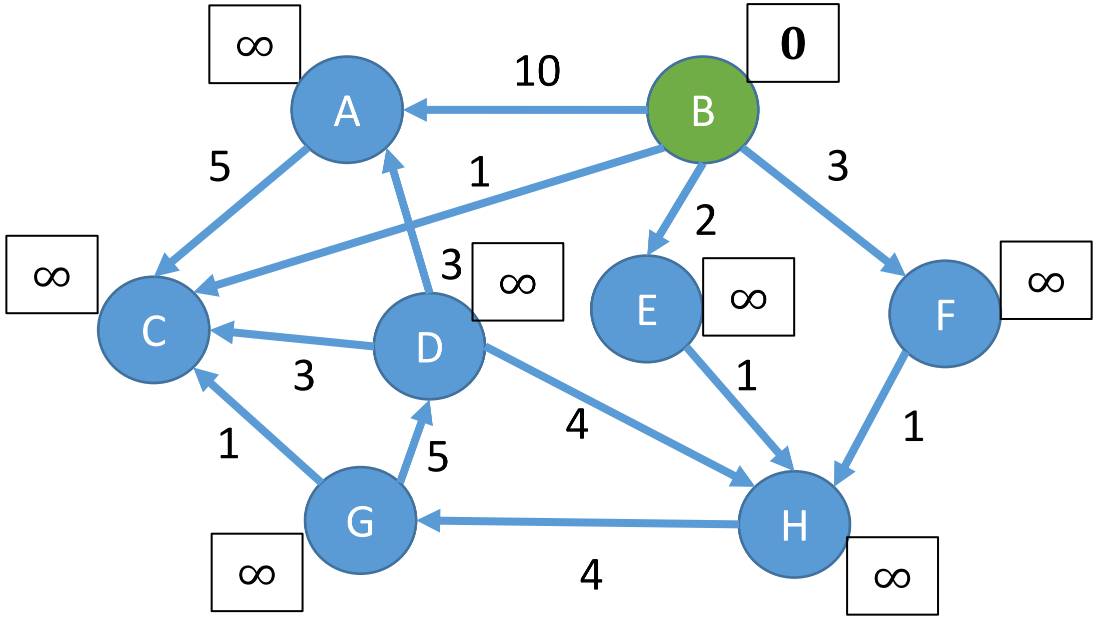
Dijkstra's Algorithm

- We can start with any node, but we'll start with node B since a path exists from B to all other nodes



Dijkstra's Algorithm

- We'll remember the cost from each node back to node B.
- B has a cost of 0; All other nodes are assumed to have a cost of infinity
  - Ensures the path found will be less than that

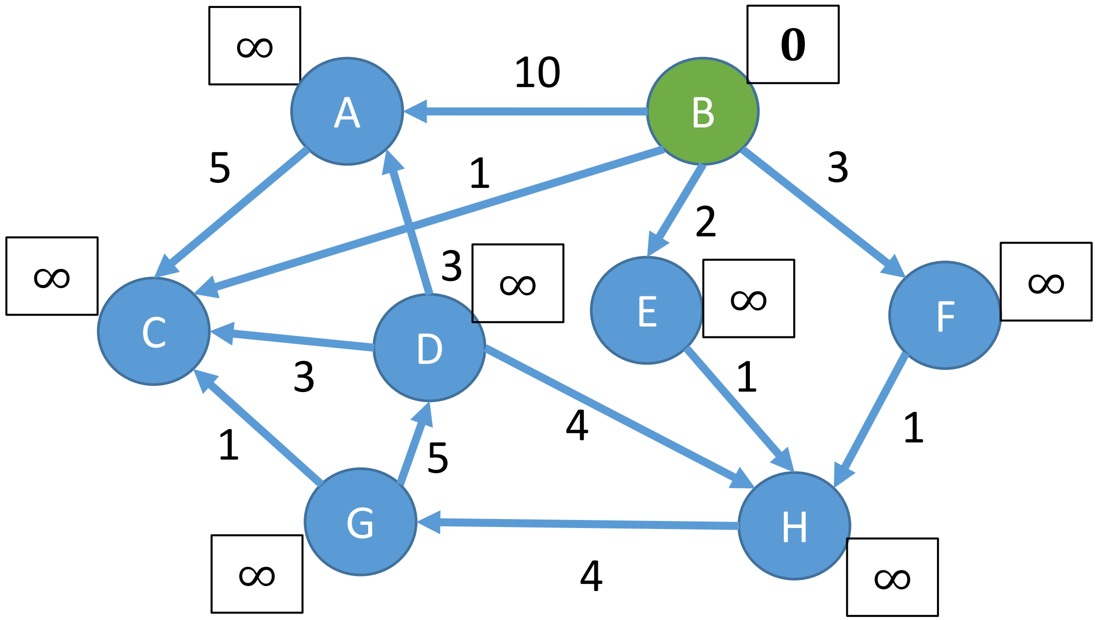


A cost = ∞
B cost = 0
C cost = ∞
D cost = ∞
E cost = ∞
F cost = ∞
G cost = ∞
H cost = ∞

Dijkstra's Algorithm

- A priority queue or min-heap is used to prioritize nodes with lower costs to be visited first

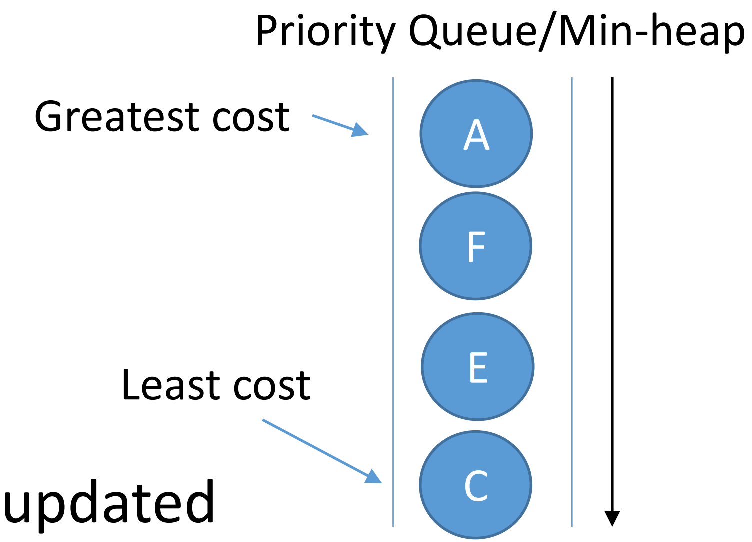
Priority Queue/Min-heap

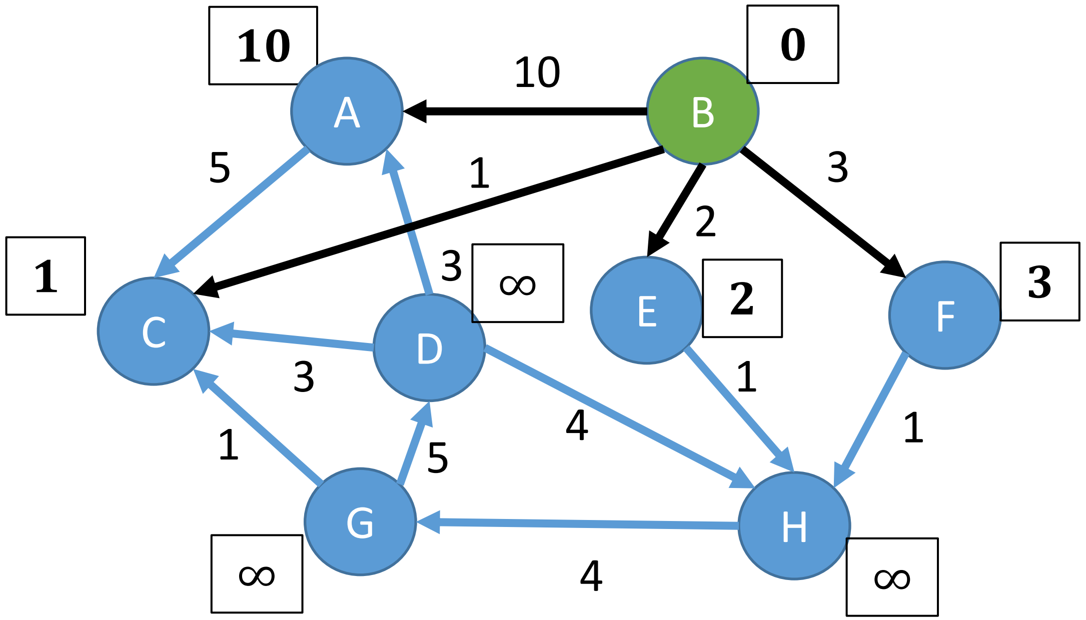


A cost = ∞
B cost = 0
C cost = ∞
D cost = ∞
E cost = ∞
F cost = ∞
G cost = ∞
H cost = ∞

Dijkstra's Algorithm

- We'll now look at the nodes adjacent to B.
- Cost of each is B's cost + cost of the edge
  - All of which are less than infinity so the costs are updated



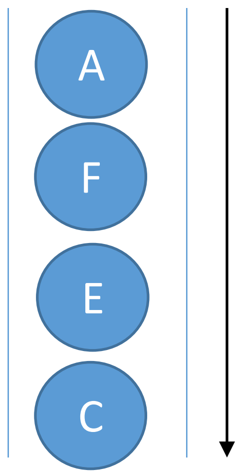


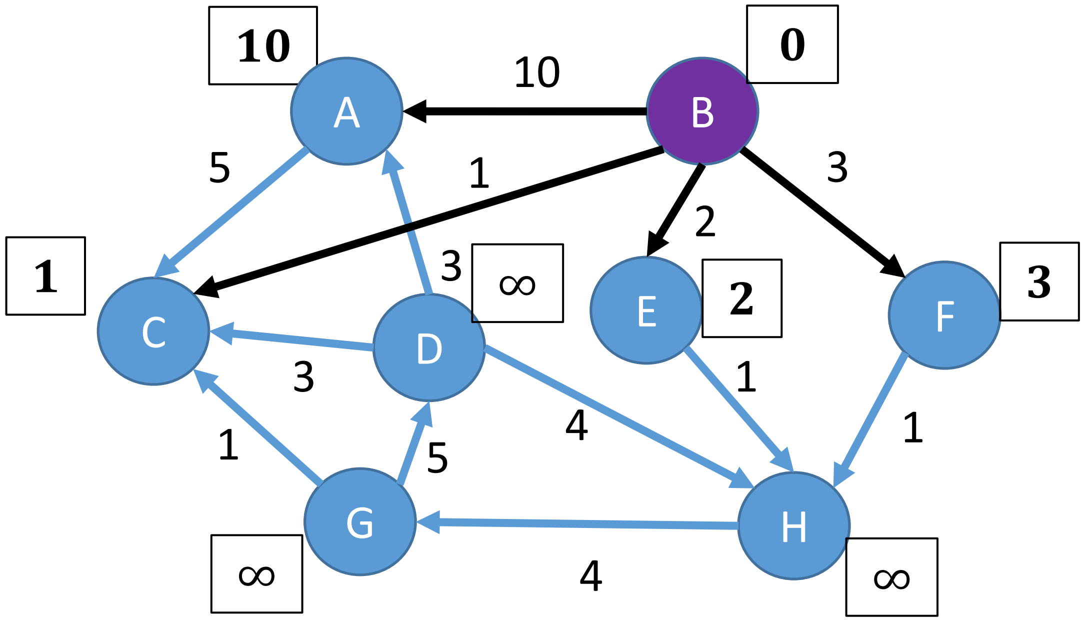
A cost = $0 + 10 = 10$
B cost = 0
C cost = $0 + 1 = 1$
D cost = ∞
E cost = $0 + 2 = 2$
F cost = $0 + 3 = 3$
G cost = ∞
H cost = ∞

Dijkstra's Algorithm

- We are finished with B

Priority Queue/Min-heap

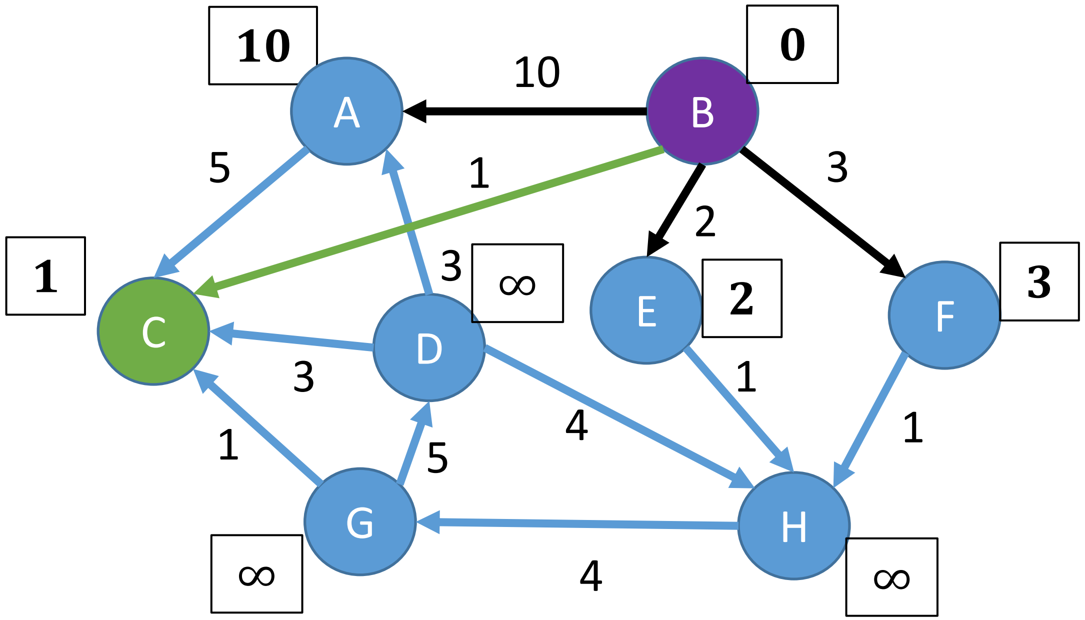




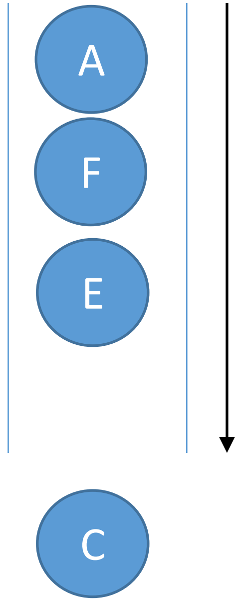
A cost = 10
B cost = 0
C cost = 1
D cost = ∞
E cost = 2
F cost = 3
G cost = ∞
H cost = ∞

Dijkstra’s Algorithm

- We move on to the next node in the priority queue



Priority Queue/Min-heap

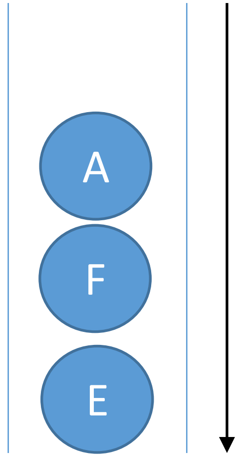


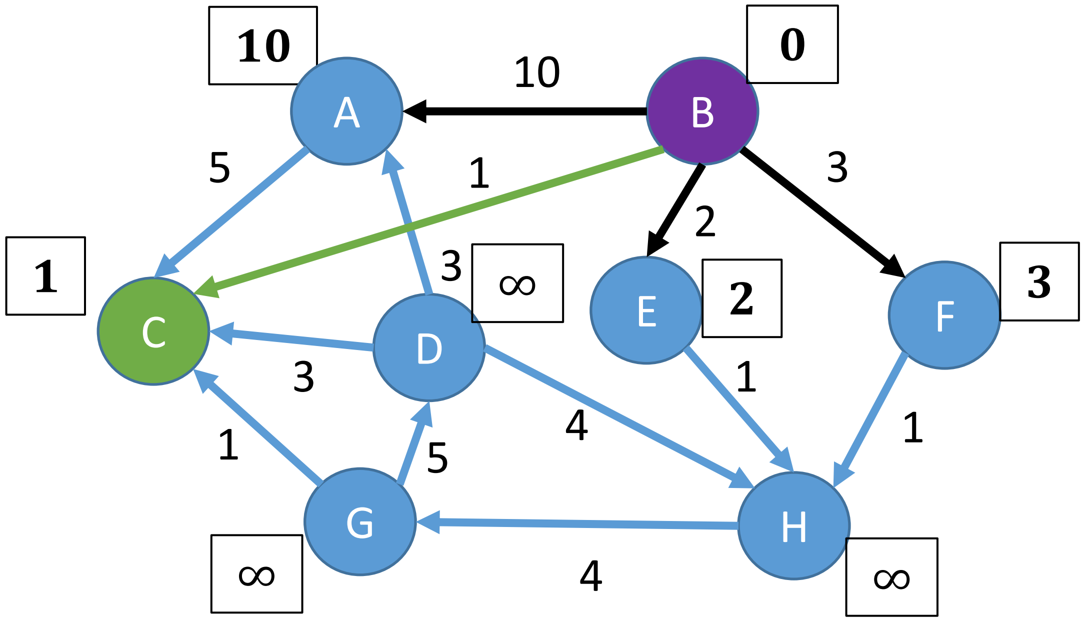
A cost = 10
B cost = 0
C cost = 1
D cost = ∞
E cost = 2
F cost = 3
G cost = ∞
H cost = ∞

Dijkstra’s Algorithm

- We’ll now look at the nodes adjacent to C.
  - There are none

Priority Queue/Min-heap

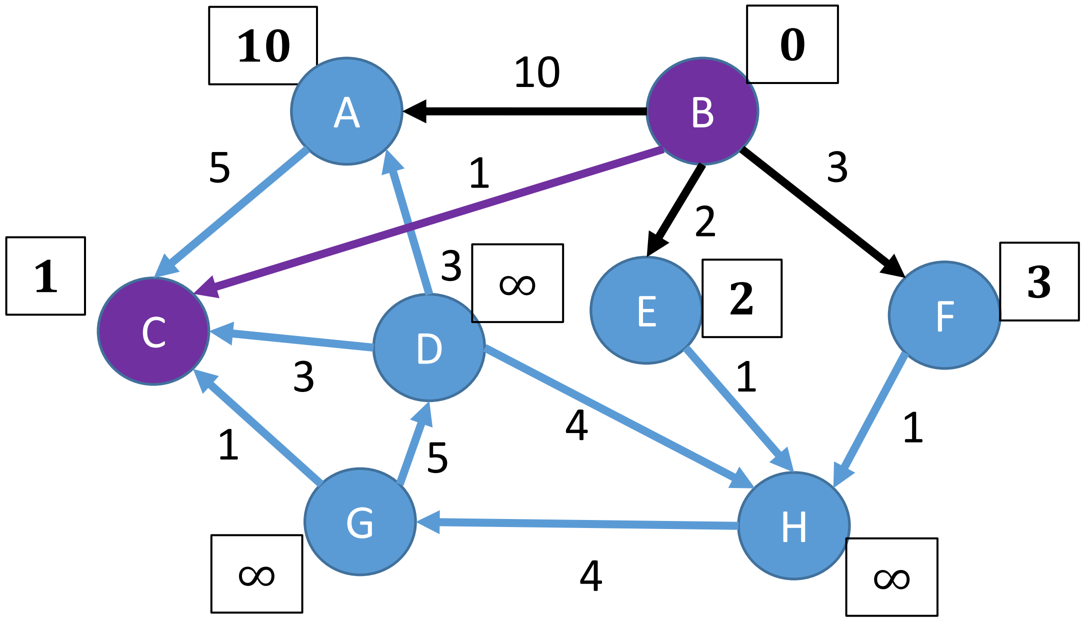




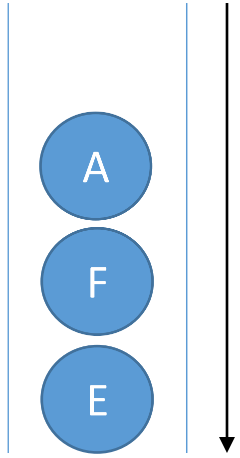
A cost = 10
B cost = 0
C cost = 1
D cost = ∞
E cost = 2
F cost = 3
G cost = ∞
H cost = ∞

Dijkstra's Algorithm

- We are finished with C



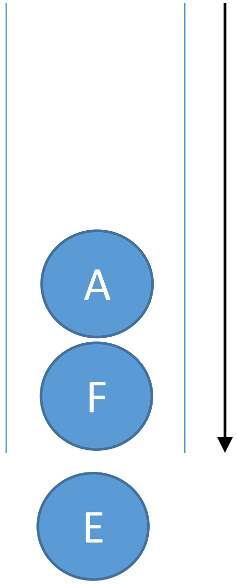
Priority Queue/Min-heap



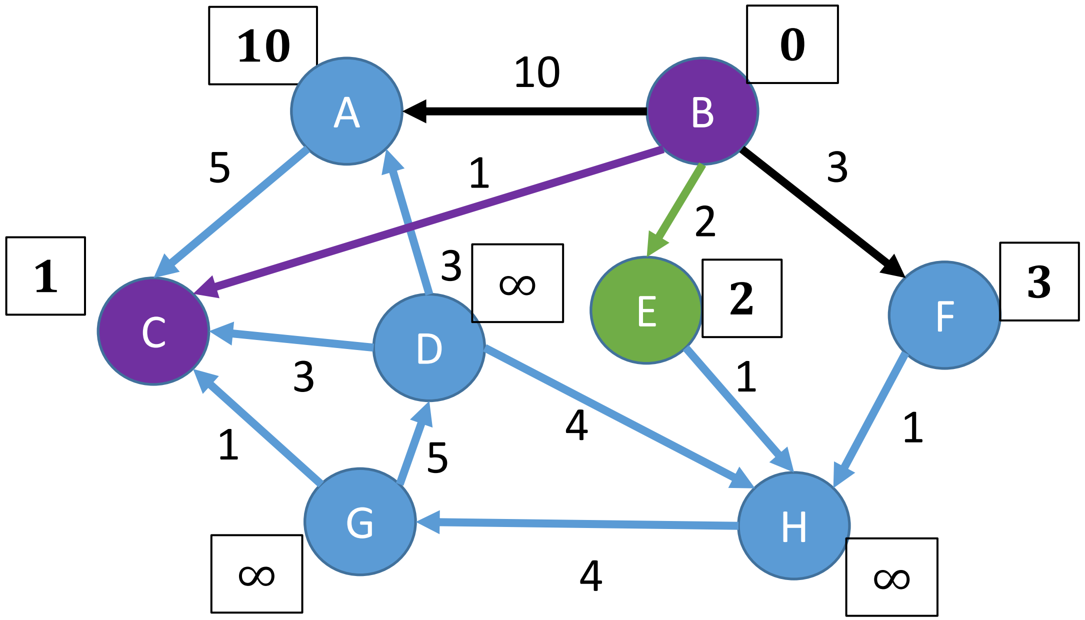
A cost = 10
B cost = 0
C cost = 1
D cost = ∞
E cost = 2
F cost = 3
G cost = ∞
H cost = ∞

Dijkstra’s Algorithm

Priority Queue/Min-heap



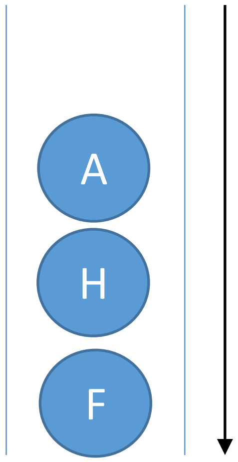
- We move on to the next node in the priority queue



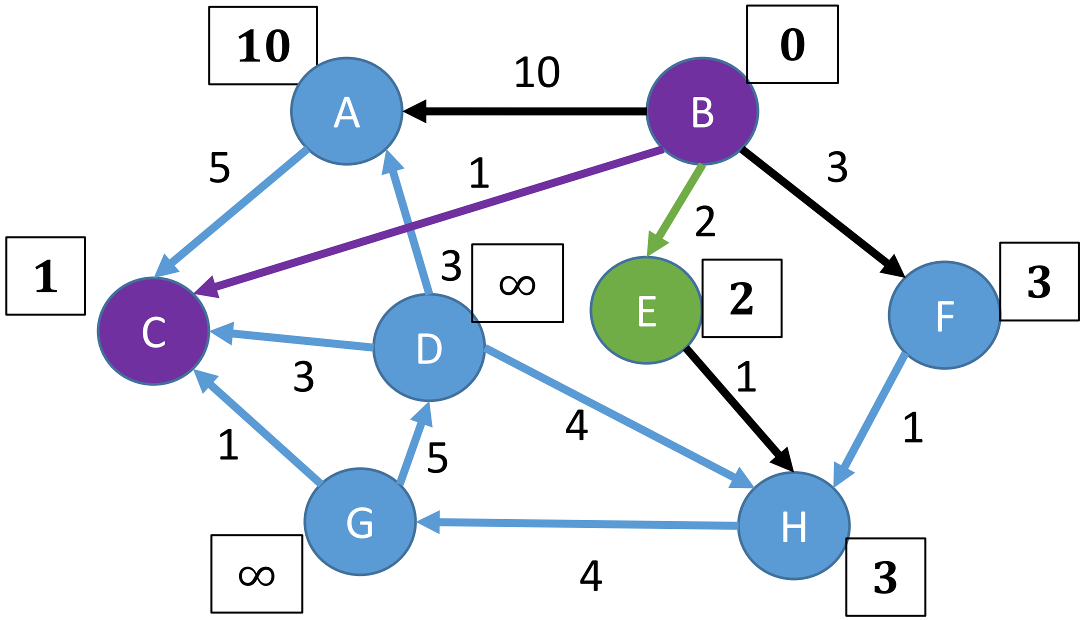
A cost = 10
B cost = 0
C cost = 1
D cost = ∞
E cost = 2
F cost = 3
G cost = ∞
H cost = ∞

Dijkstra's Algorithm

Priority Queue/Min-heap



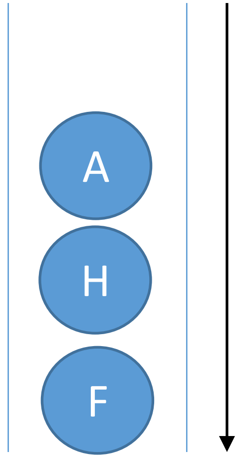
- We'll now look at the nodes adjacent to E.
  - Cost of each is E's cost + cost of the edge



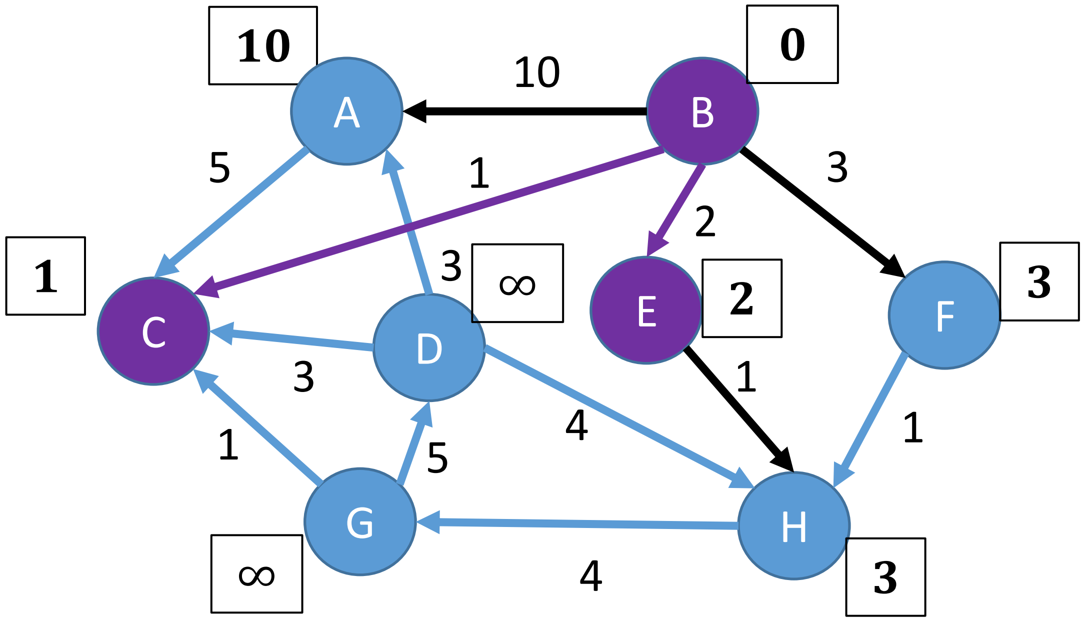
A cost = 10
B cost = 0
C cost = 1
D cost = ∞
E cost = 2
F cost = 3
G cost = ∞
H cost = $2 + 1 = 3$

Dijkstra's Algorithm

Priority Queue/Min-heap



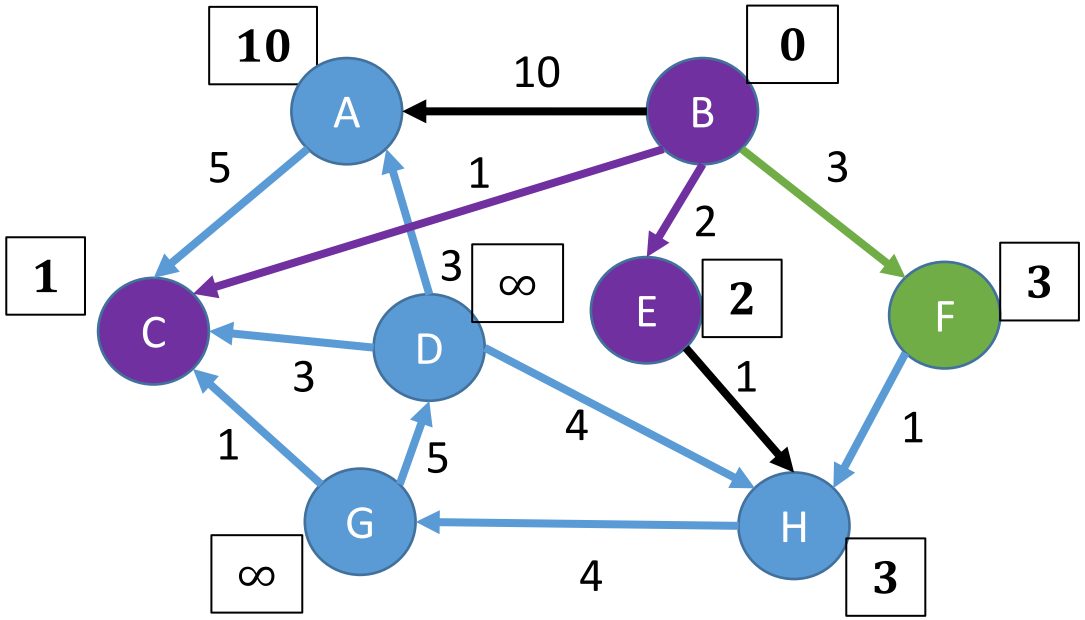
- We are finished with E.



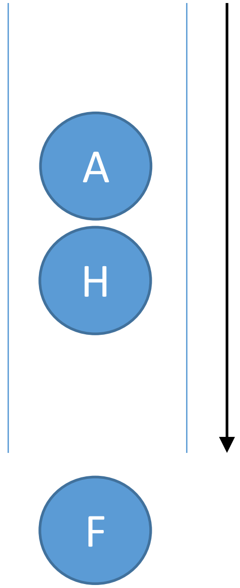
A cost = 10
B cost = 0
C cost = 1
D cost = ∞
E cost = 2
F cost = 3
G cost = ∞
H cost = 3

Dijkstra's Algorithm

- We move on to the next node in the priority queue

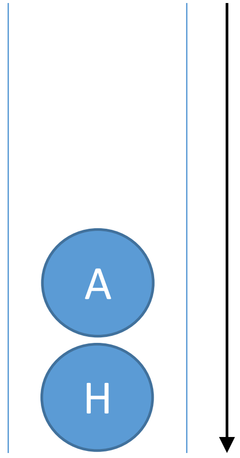


Priority Queue/Min-heap



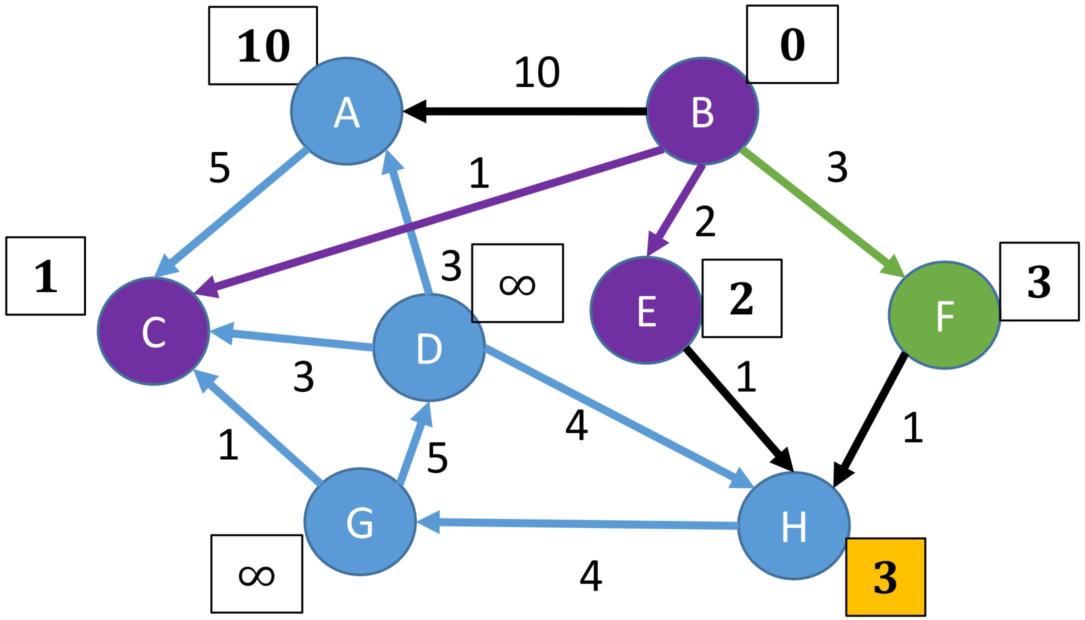
A cost = 10
B cost = 0
C cost = 1
D cost = ∞
E cost = 2
F cost = 3
G cost = ∞
H cost = 3

Priority Queue/Min-heap



Dijkstra's Algorithm

- We'll now look at the nodes adjacent to F.
  - Cost of each is F's cost + cost of the edge
  - 3 + 1 = **4** (NOT LESS THAN THE CURRENT COST OF H)

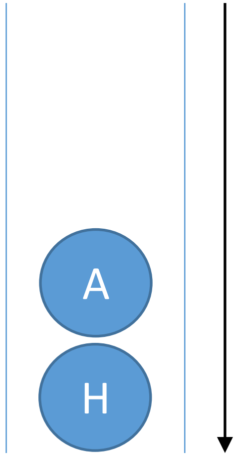


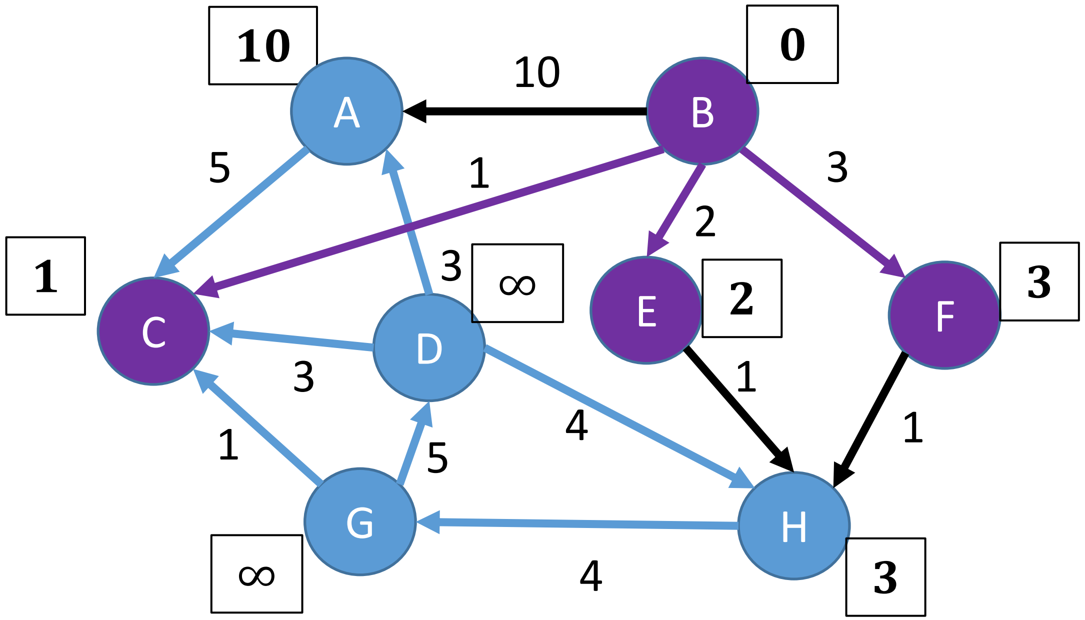
A cost = 10
B cost = 0
C cost = 1
D cost = ∞
E cost = 2
F cost = 3
G cost = ∞
H cost = **3**

Dijkstra's Algorithm

- We are finished with F

Priority Queue/Min-heap

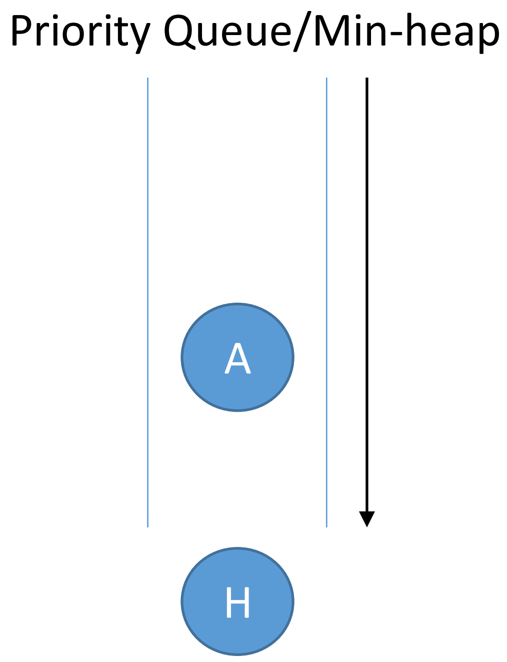


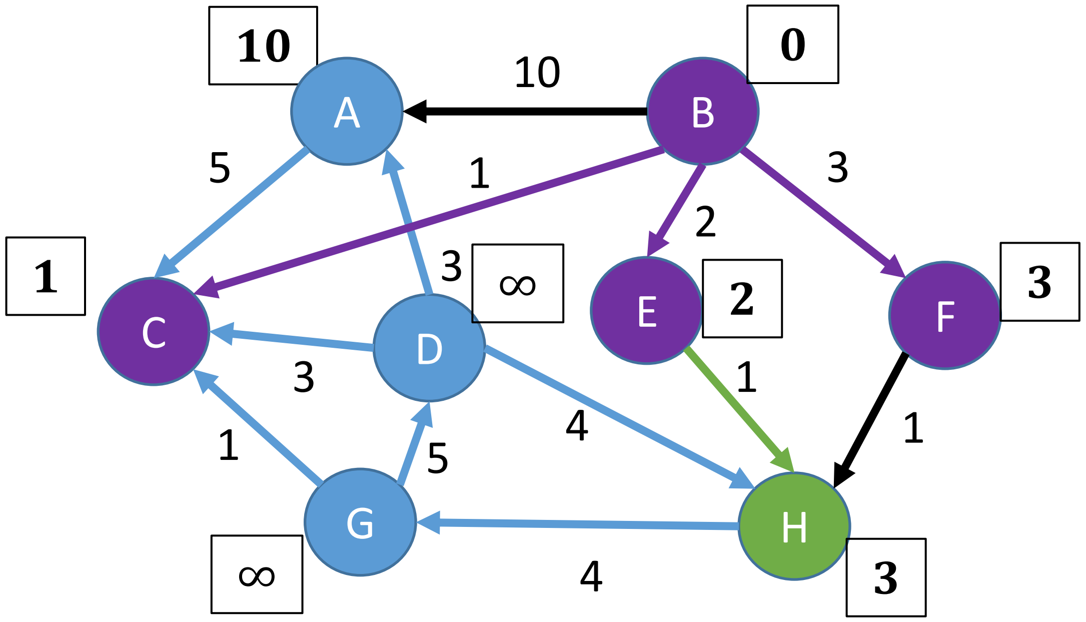


A cost = 10
B cost = 0
C cost = 1
D cost = ∞
E cost = 2
F cost = 3
G cost = ∞
H cost = 3

Dijkstra's Algorithm

- We move on to the next node in the priority queue



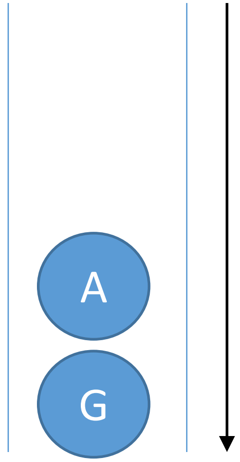


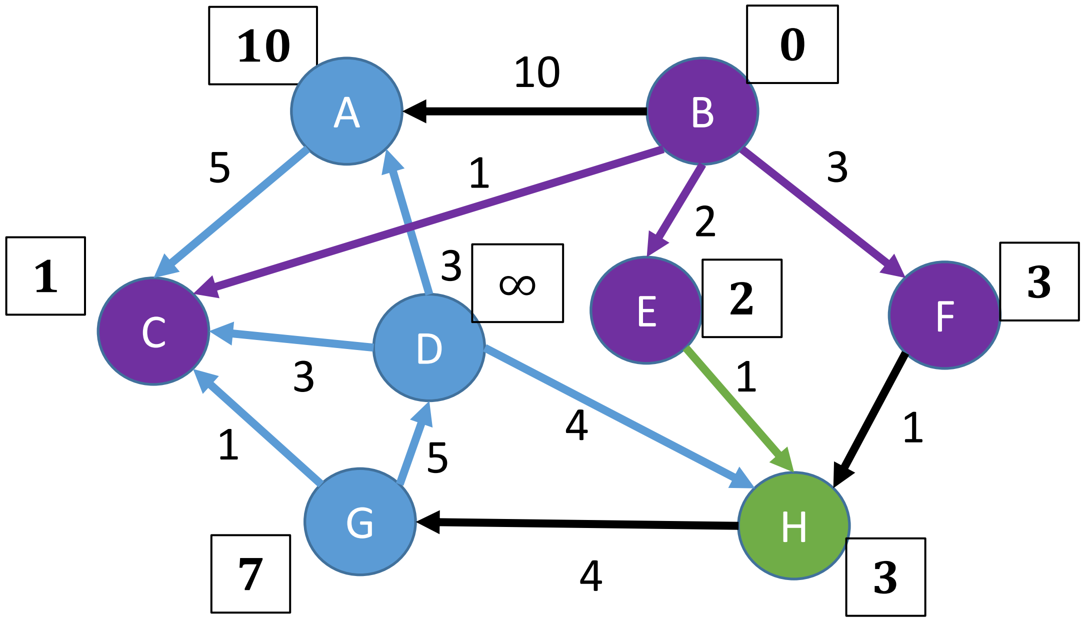
A cost = 10
B cost = 0
C cost = 1
D cost = ∞
E cost = 2
F cost = 3
G cost = ∞
H cost = 3

Dijkstra's Algorithm

- We'll now look at the nodes adjacent to H.
  - Cost of each is H's cost + cost of the edge

Priority Queue/Min-heap



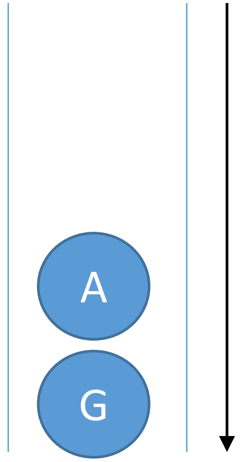


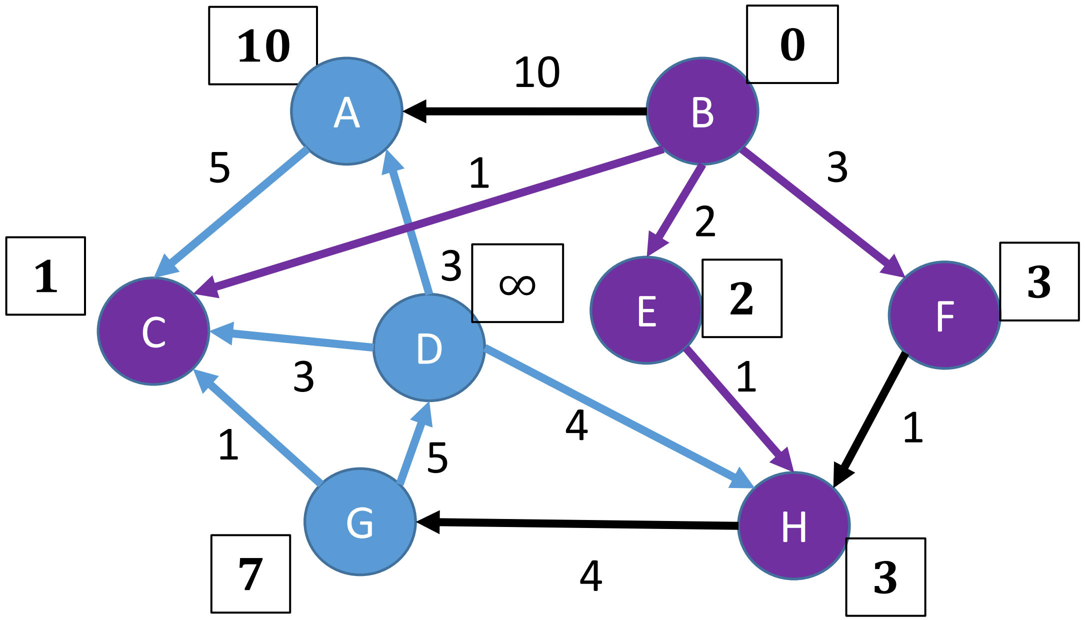
A cost = 10
B cost = 0
C cost = 1
D cost = ∞
E cost = 2
F cost = 3
G cost = $3 + 4 = 7$
H cost = 3

Dijkstra's Algorithm

- We are finished with H.

Priority Queue/Min-heap

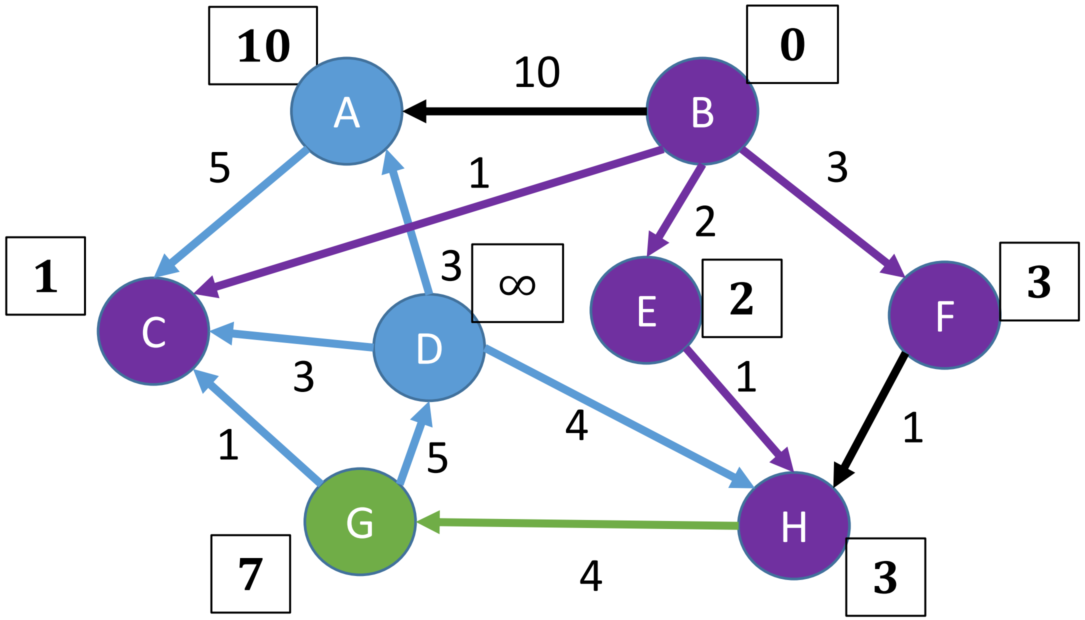




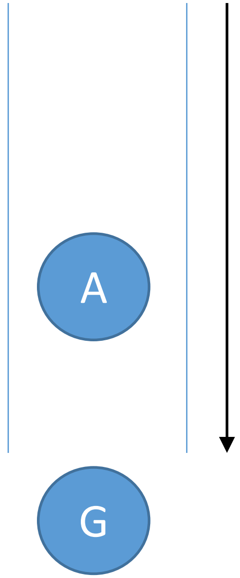
A cost = 10
B cost = 0
C cost = 1
D cost = ∞
E cost = 2
F cost = 3
G cost = 7
H cost = 3

Dijkstra’s Algorithm

- We move on to the next node in the priority queue



Priority Queue/Min-heap

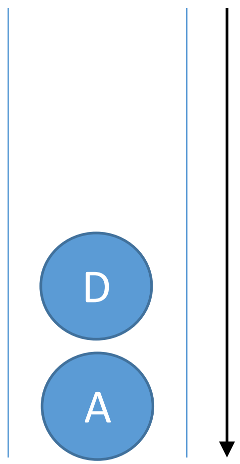


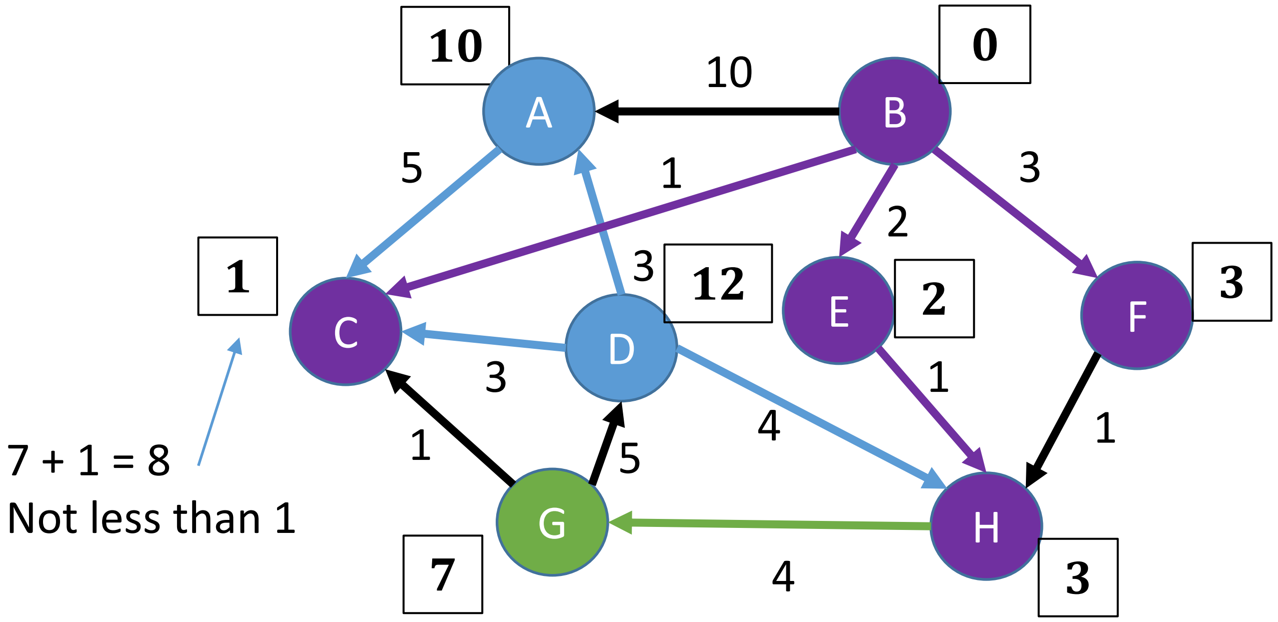
A cost = 10
B cost = 0
C cost = 1
D cost = ∞
E cost = 2
F cost = 3
G cost = 7
H cost = 3

Dijkstra's Algorithm

- We'll now look at the nodes adjacent to G.
  - Cost of each is G's cost + cost of the edge

Priority Queue/Min-heap



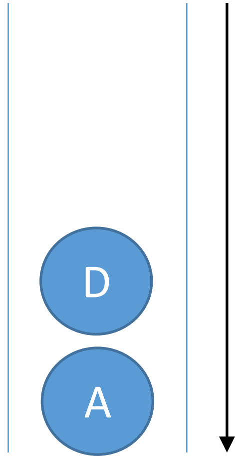


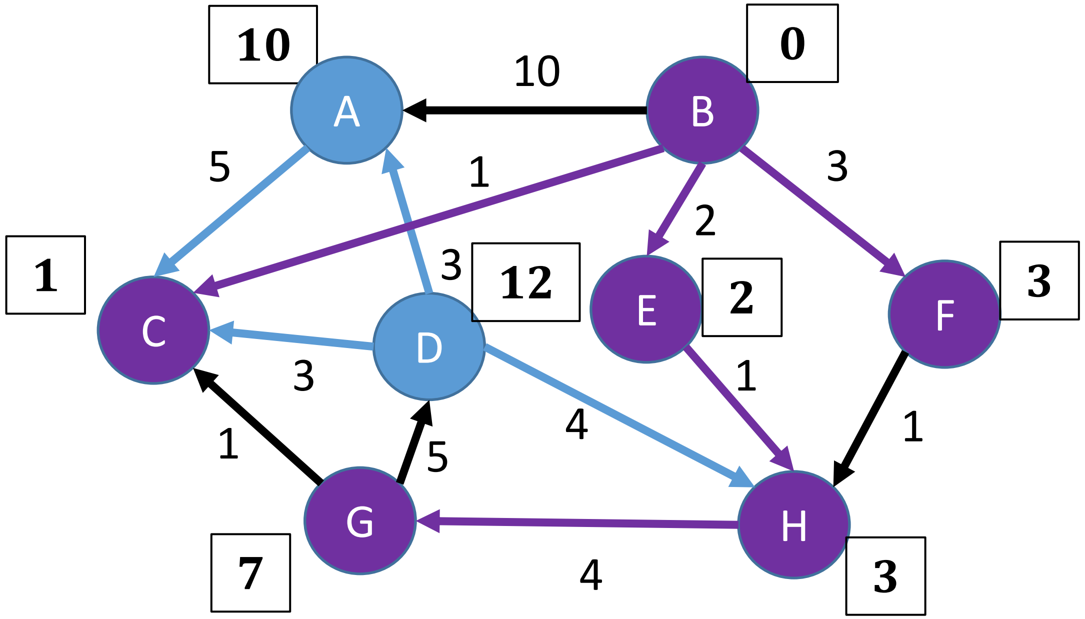
A cost = 10
B cost = 0
C cost = 1
D cost = 7 + 5 = 12
E cost = 2
F cost = 3
G cost = 7
H cost = 3

Dijkstra's Algorithm

- We are finished with G

Priority Queue/Min-heap

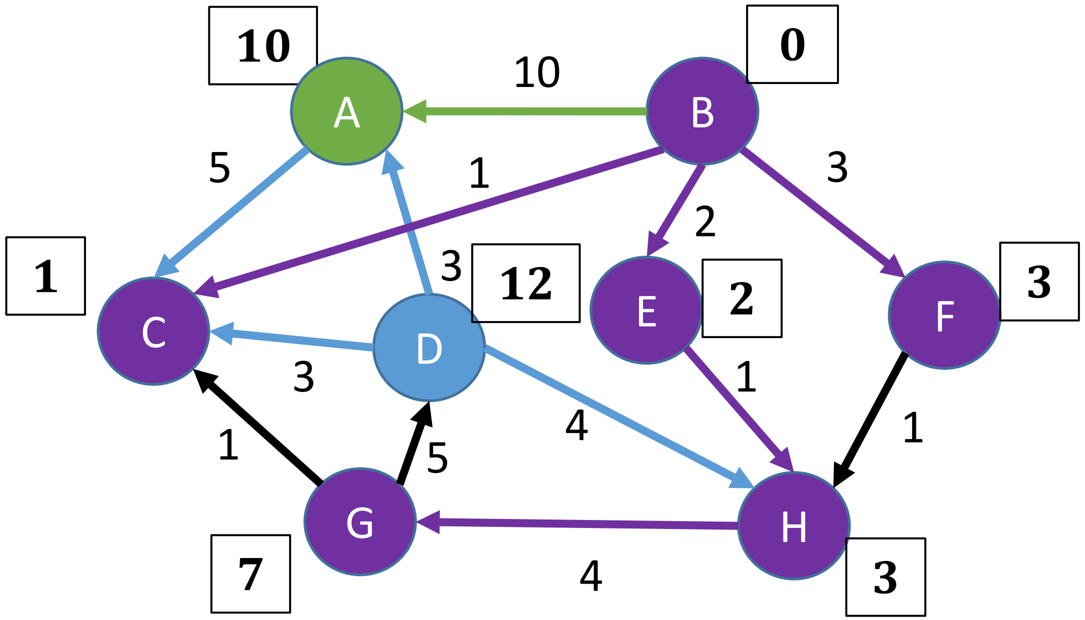




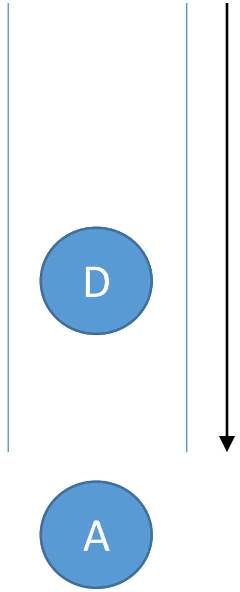
A cost = 10
B cost = 0
C cost = 1
D cost = 12
E cost = 2
F cost = 3
G cost = 7
H cost = 3

Dijkstra’s Algorithm

- We move on to the next node in the priority queue



Priority Queue/Min-heap

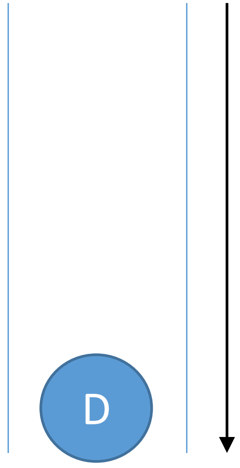


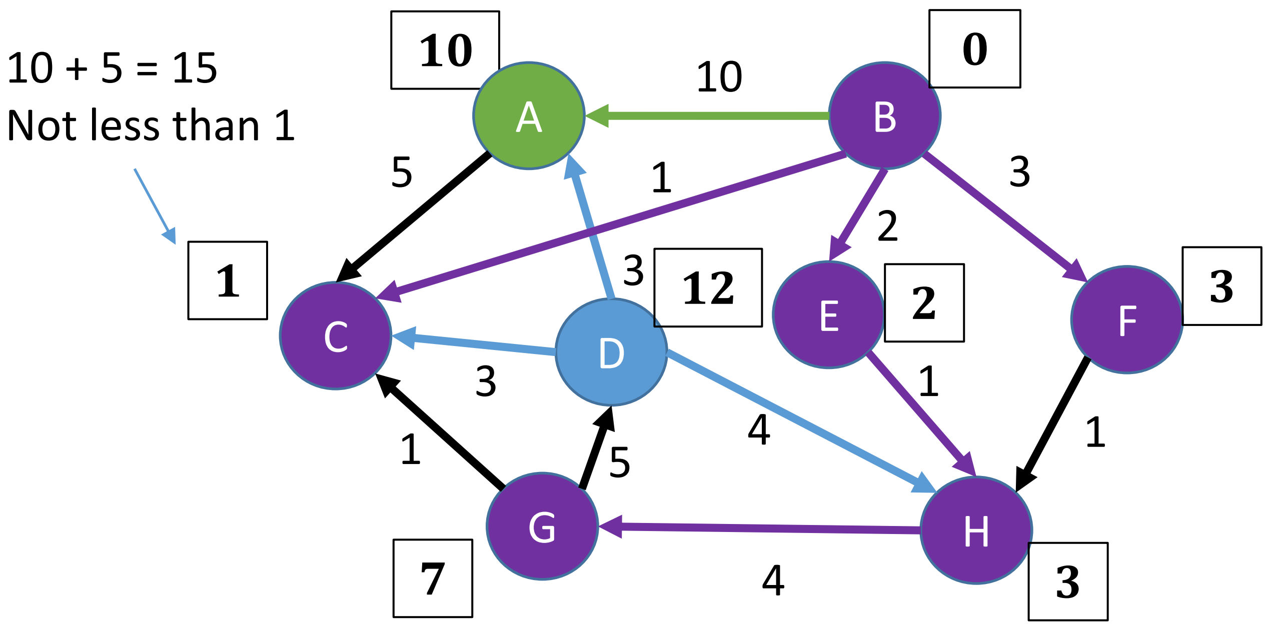
A cost = 10
B cost = 0
C cost = 1
D cost = 12
E cost = 2
F cost = 3
G cost = 7
H cost = 3

Dijkstra's Algorithm

- We'll now look at the nodes adjacent to A.
  - Cost of each is A's cost + cost of the edge

Priority Queue/Min-heap

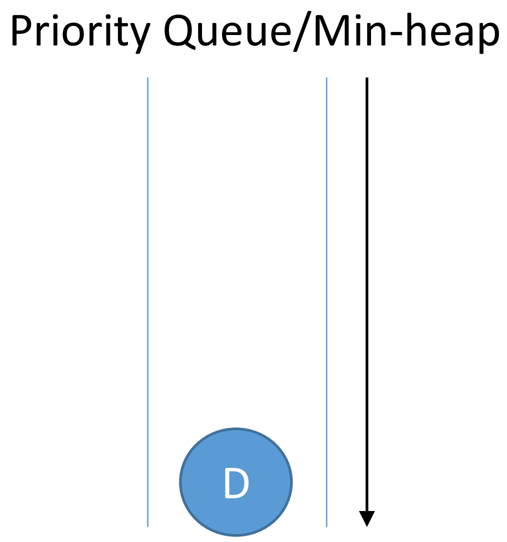


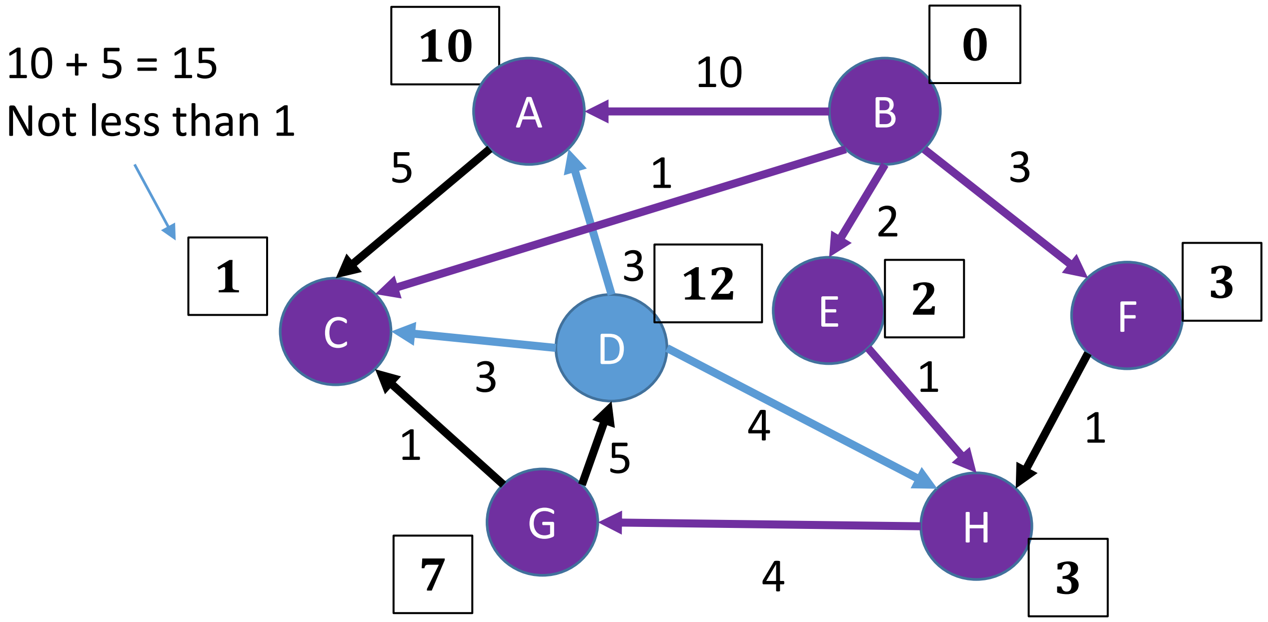


A cost = 10
B cost = 0
C cost = 1
D cost = 12
E cost = 2
F cost = 3
G cost = 7
H cost = 3

Dijkstra's Algorithm

- We are finished with A



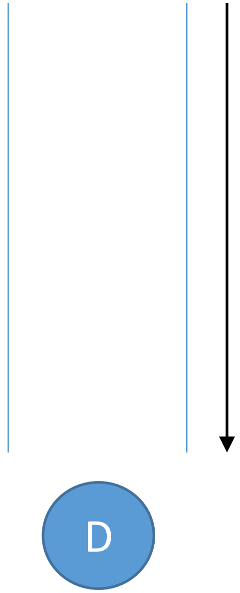


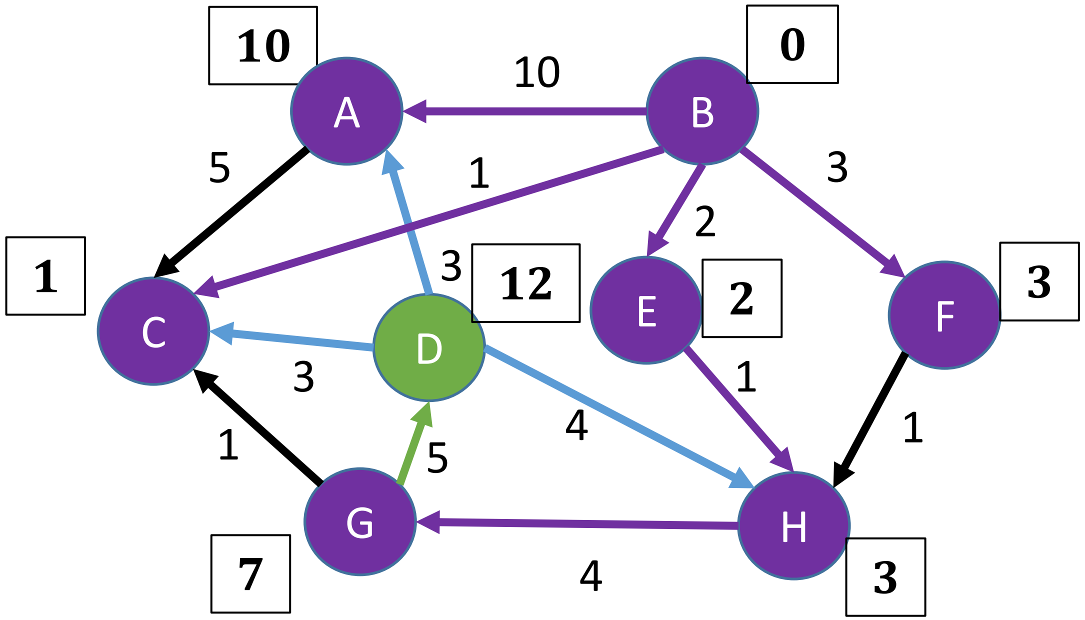
A cost = 10
B cost = 0
C cost = 1
D cost = 12
E cost = 2
F cost = 3
G cost = 7
H cost = 3

Dijkstra’s Algorithm

- We move on to the next node in the priority queue

Priority Queue/Min-heap



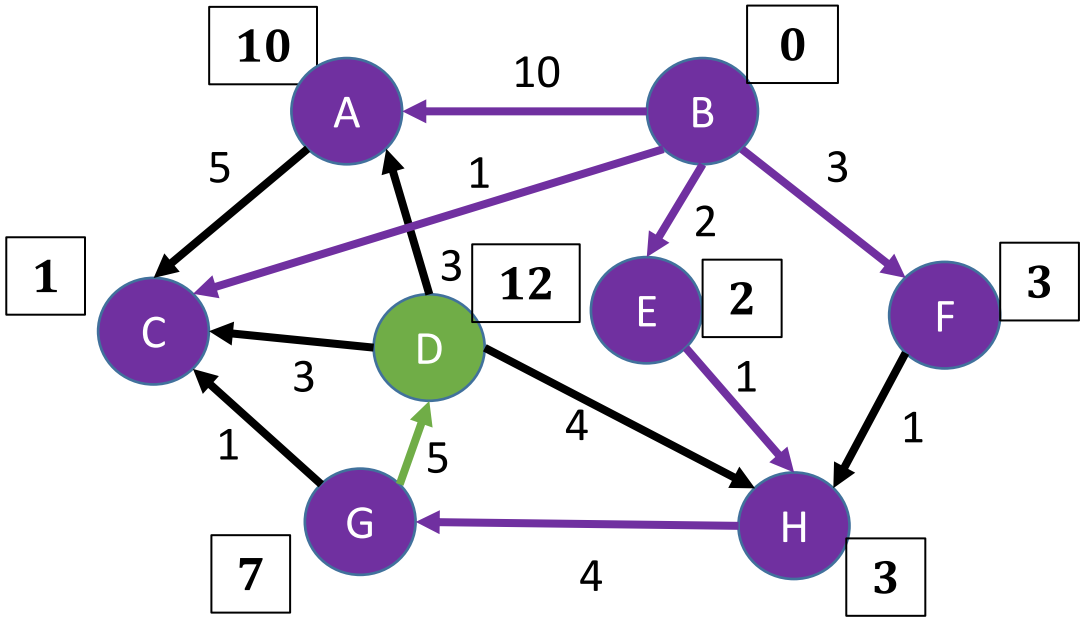


A cost = 10
B cost = 0
C cost = 1
D cost = 12
E cost = 2
F cost = 3
G cost = 7
H cost = 3

Dijkstra's Algorithm

- We'll now look at the nodes adjacent to D.
  - Cost of each is D's cost + cost of the edge

Priority Queue/Min-heap

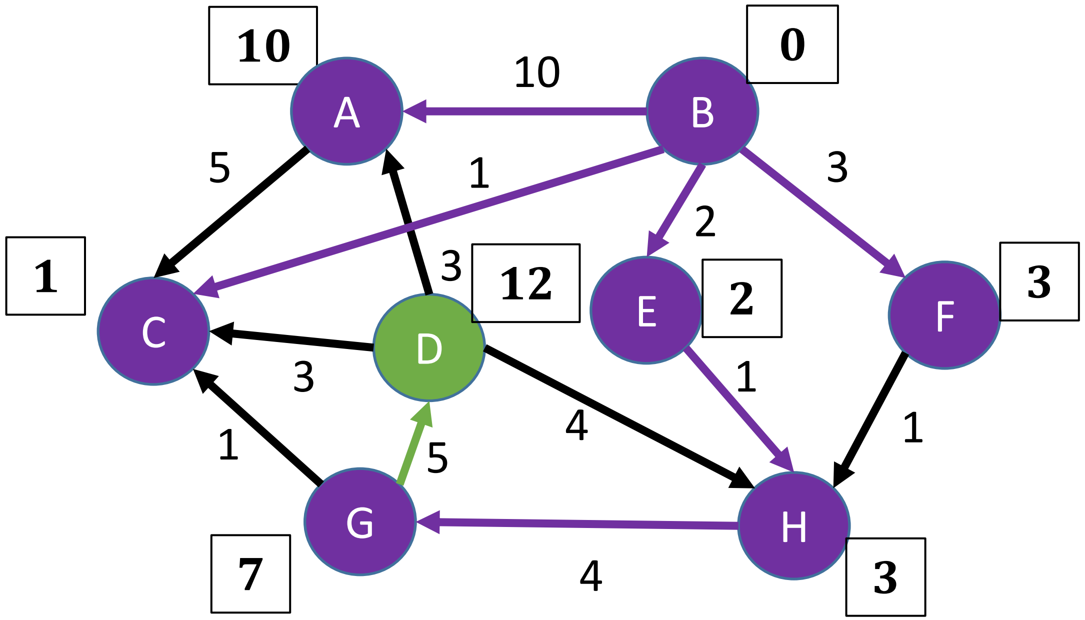


A cost = 10
B cost = 0
C cost = 1
D cost = 12
E cost = 2
F cost = 3
G cost = 7
H cost = 3

Dijkstra's Algorithm

Priority Queue/Min-heap

- D -> A = 12 + 3 = 15 (Not less than 10)
- D -> C = 12 + 3 = 15 (Not less than 1)
- D -> H = 12 + 4 = 16 (Not less than 3)

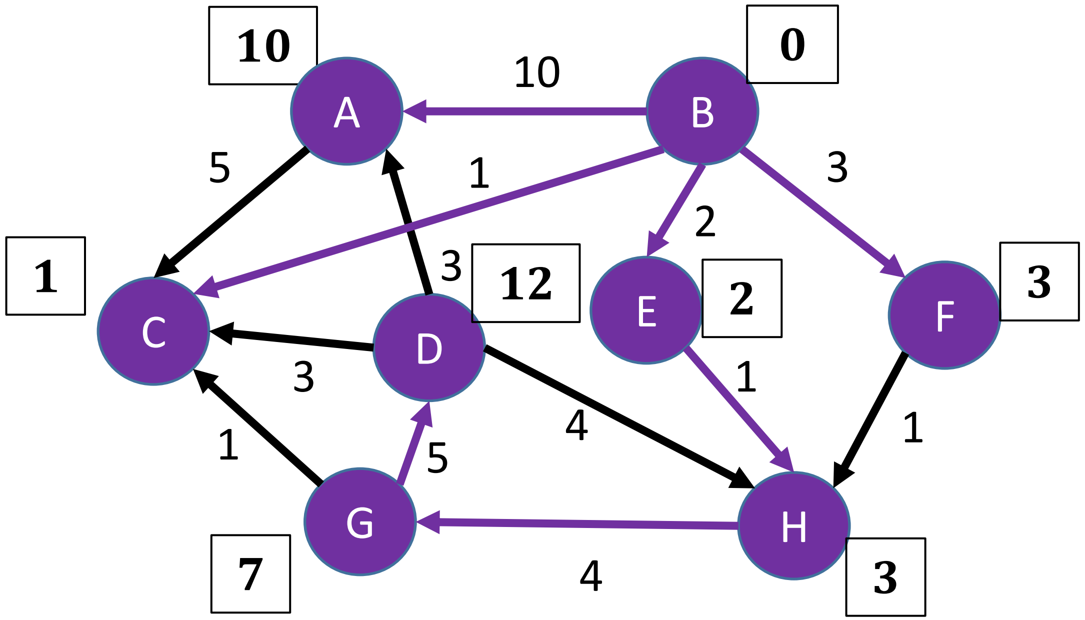


A cost = 10
B cost = 0
C cost = 1
D cost = 12
E cost = 2
F cost = 3
G cost = 7
H cost = 3

Priority Queue/Min-heap

Dijkstra's Algorithm

- We are finished with D

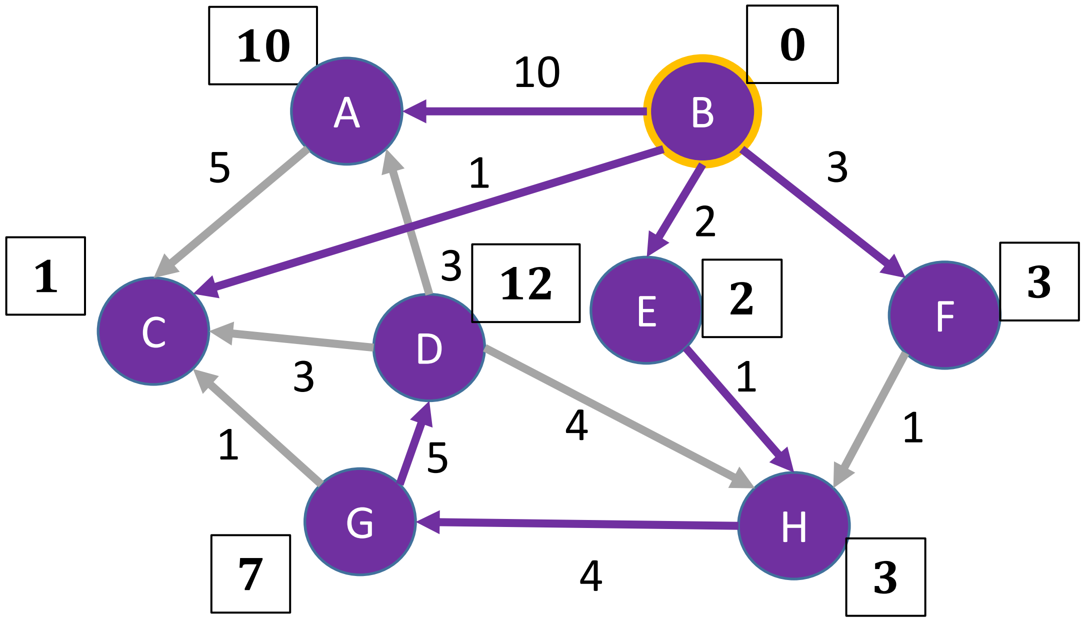


A cost = 10
B cost = 0
C cost = 1
D cost = 12
E cost = 2
F cost = 3
G cost = 7
H cost = 3

Priority Queue/Min-heap

Dijkstra's Algorithm

- Priority Queue is empty
  - Algorithm is complete



Least cost from B to all other nodes:

A cost = 10
B cost = 0
C cost = 1
D cost = 12
E cost = 2
F cost = 3
G cost = 7
H cost = 3