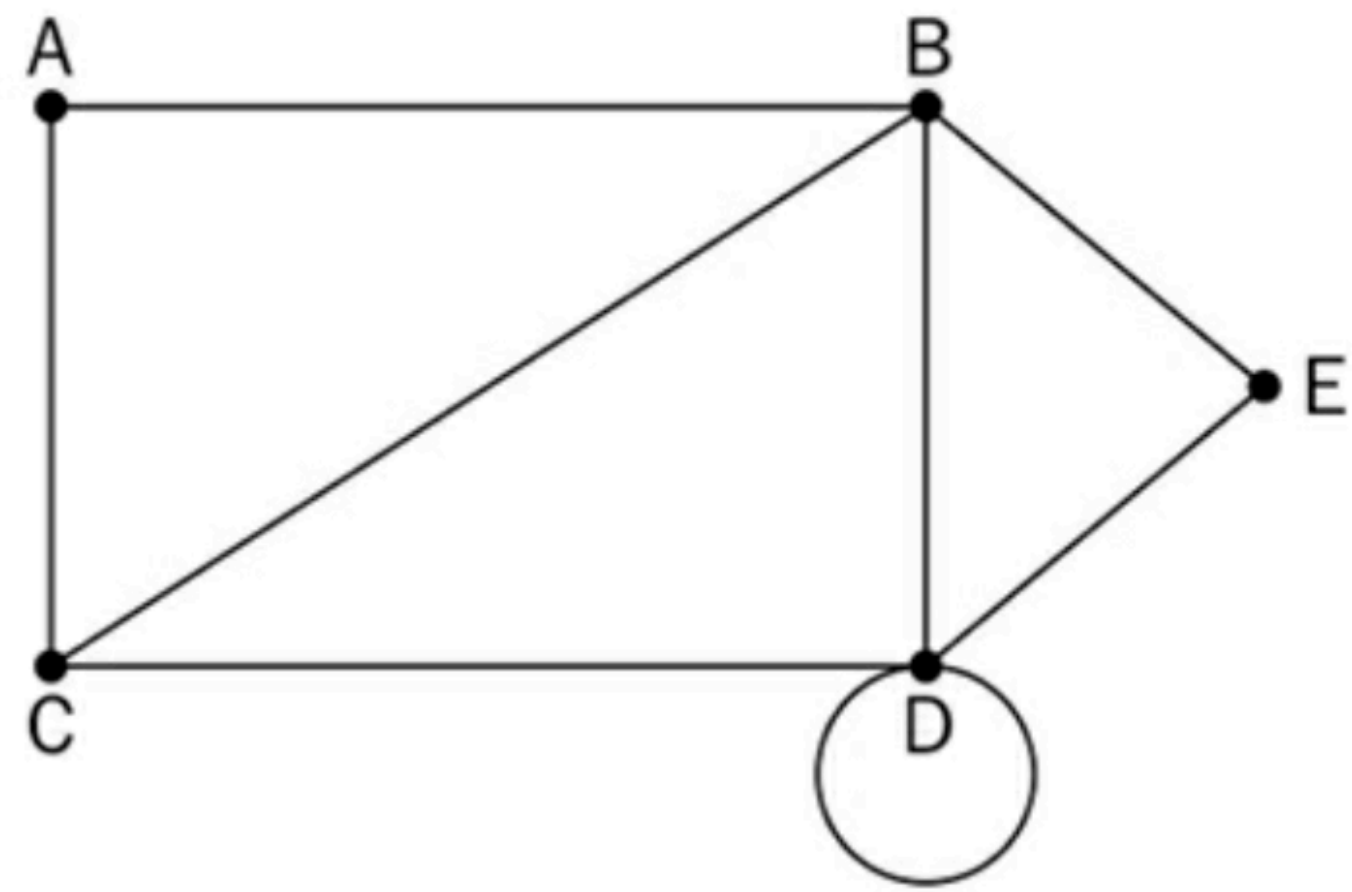
# 9 Graphs and Algorithms

## For COMSC 132

Sam Bowne

Revised Oct 28, 2024

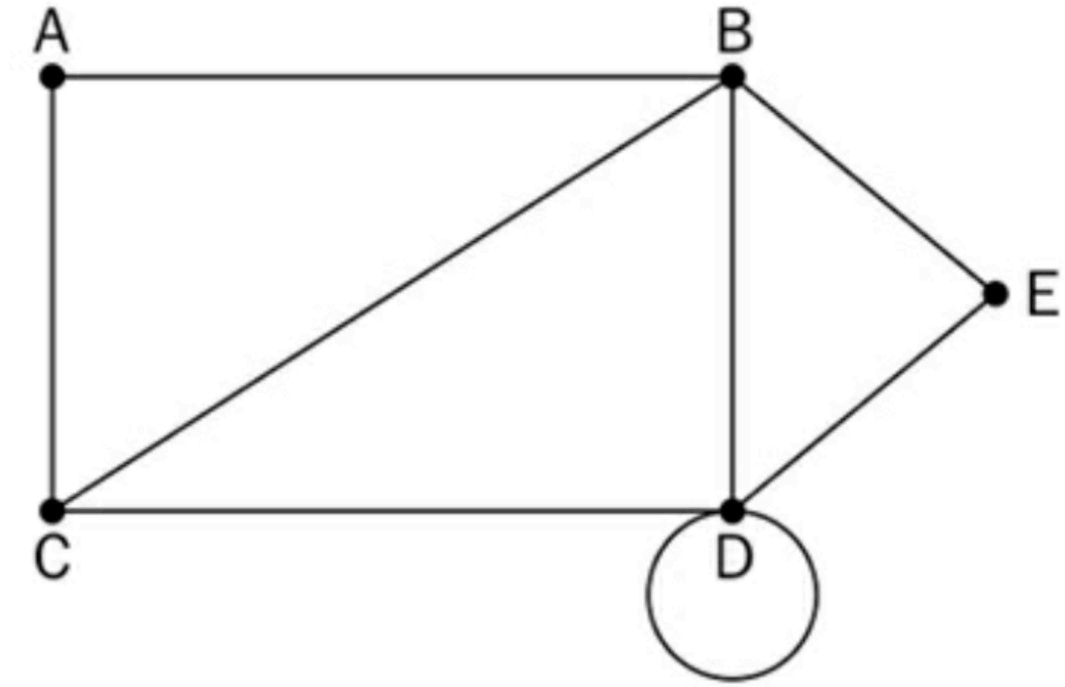
# Graph Example




The graph `G = (V, E)` in *Figure 9.1* can be described as below:

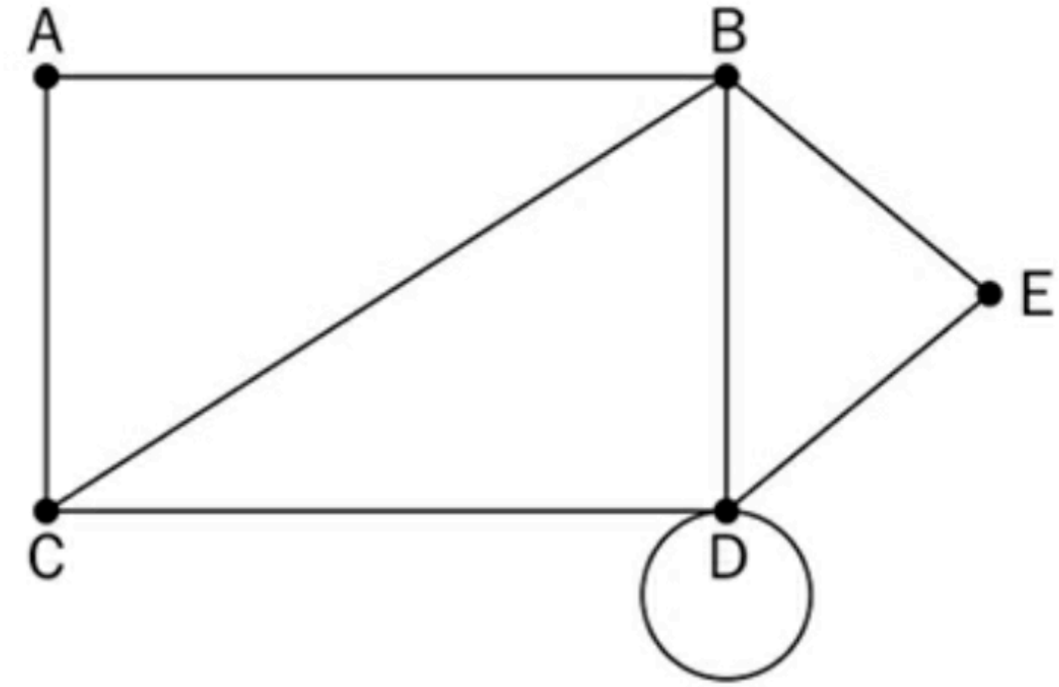
- `V = {A, B, C, D, E}`
- `E = {{A, B}, {A, C}, {B, C}, {B, D}, {C, D}, {D, D}, {B, E}, {D, E}}`
- `G = (V, E)`

# Graph terms




- **Node** or **vertex**
  - Endpoints, marked with dots
- **Edge**: connects two vertices
- **Loop**: an edge from a node returns to itself
  - See node D
- **Degree of a vertex/node**
  - Total number of edges incidental
  - Degree of B is 4

# Graph terms



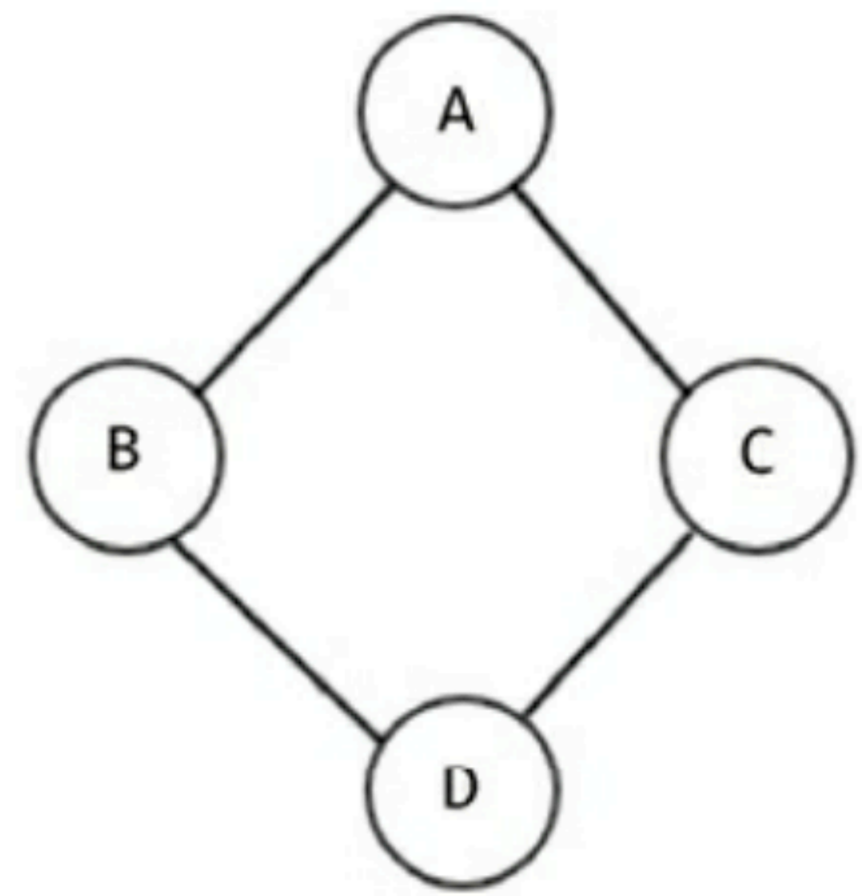

- **Adjacency**
  - Connection between any two nodes
  - C is adjacent to A
- **Path**: a sequence of vertices and edges between two nodes
  - CABE is a path from C to E
- **Leaf vertex**: has degree one

# Types of graphs

- Directed
- Undirected
- Directed acyclic
- Weighted
- Bipartite

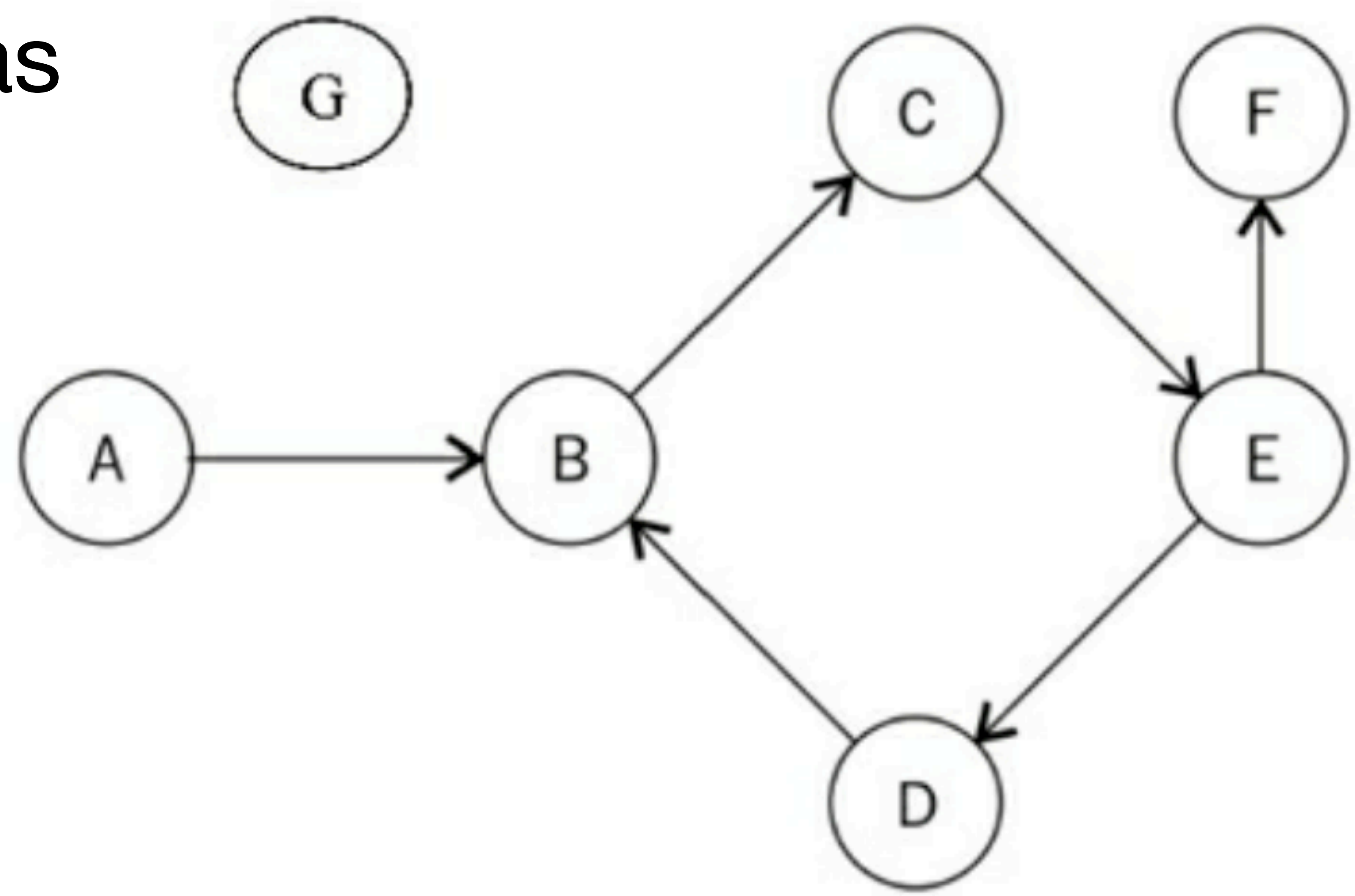
# Undirected graph

- Edges are simple lines
- No additional information other than that the nodes are connected



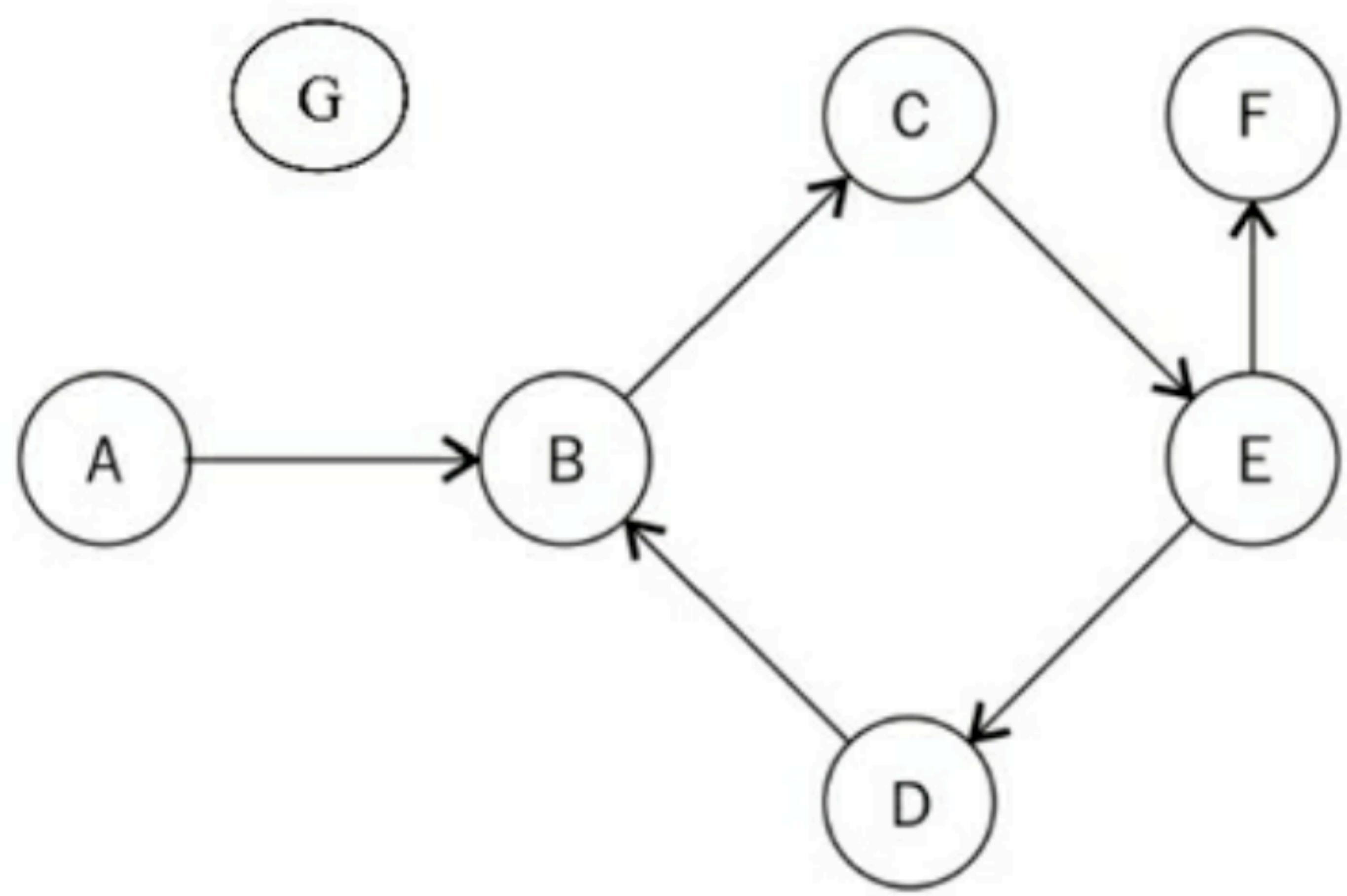
# Directed graph

- Edges have a direction
- One can only move in the direction of the arrow
- Each node has an **indegree** and an **outdegree**



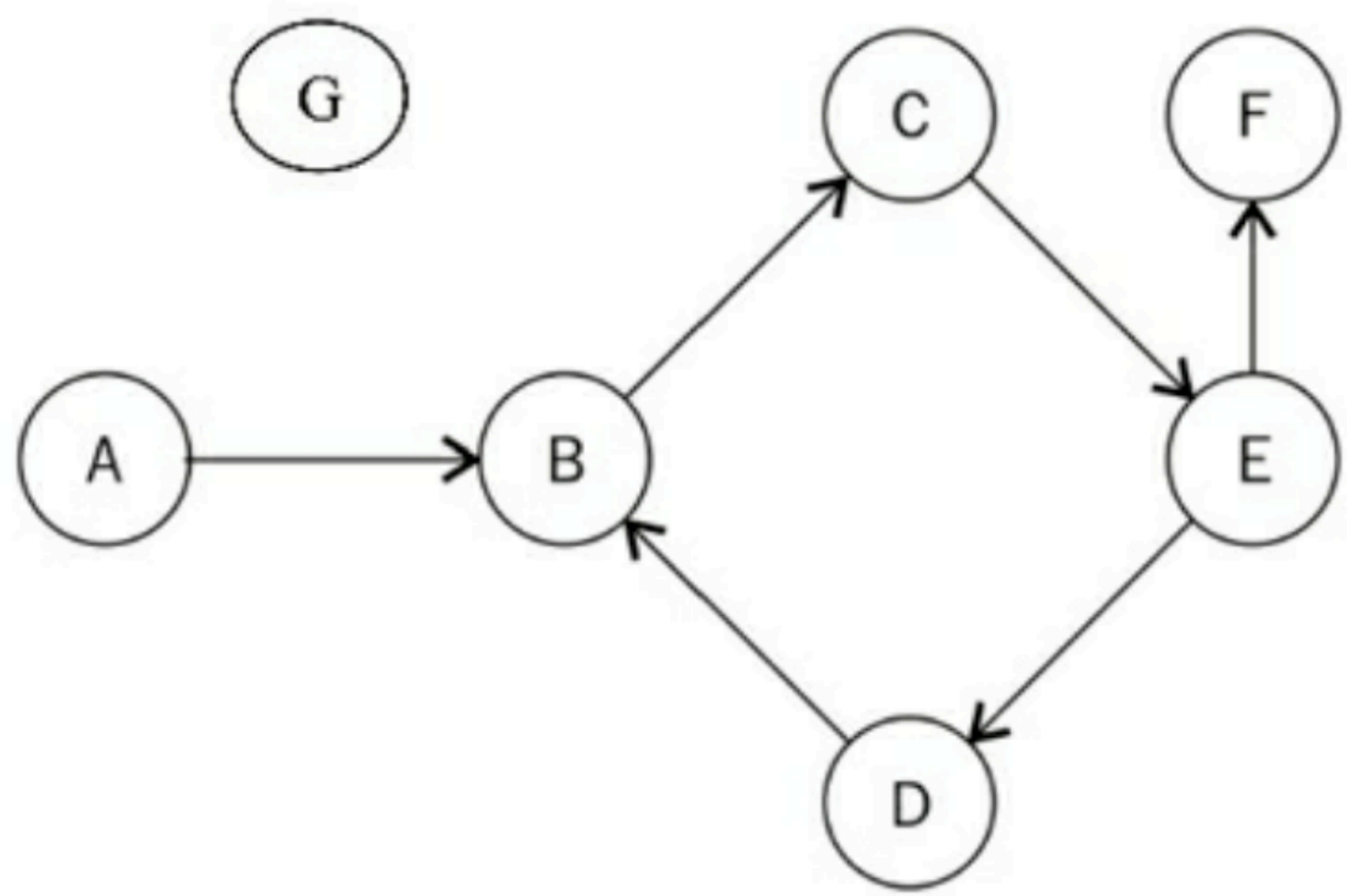
# Directed graph

- **Indegree**: number of edges coming in
  - E has indegree 1
- **Outdegree**: edges going out
  - E has outdegree 2



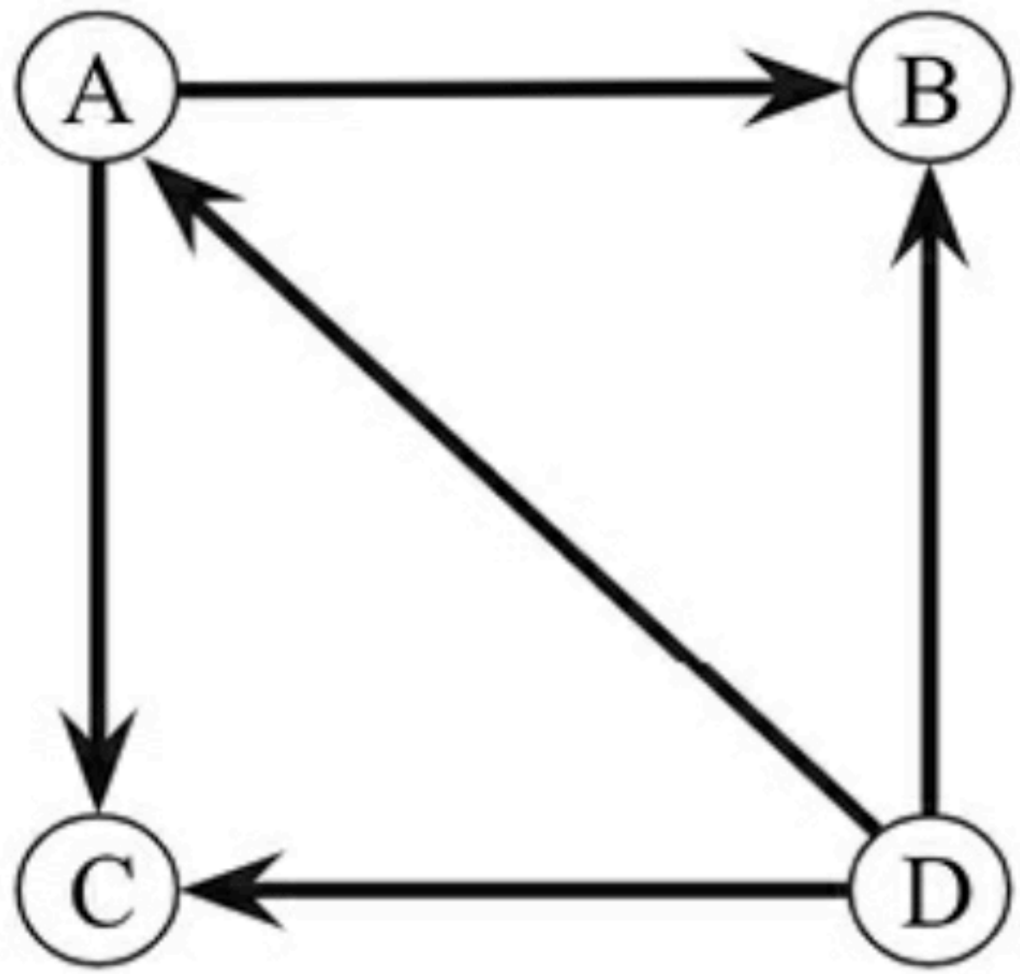
# Directed graph

- **Isolated vertex** has degree 0, like G
- **Source vertex** has indegree 0, like A
- **Sink vertex** has outdegree 0, like F



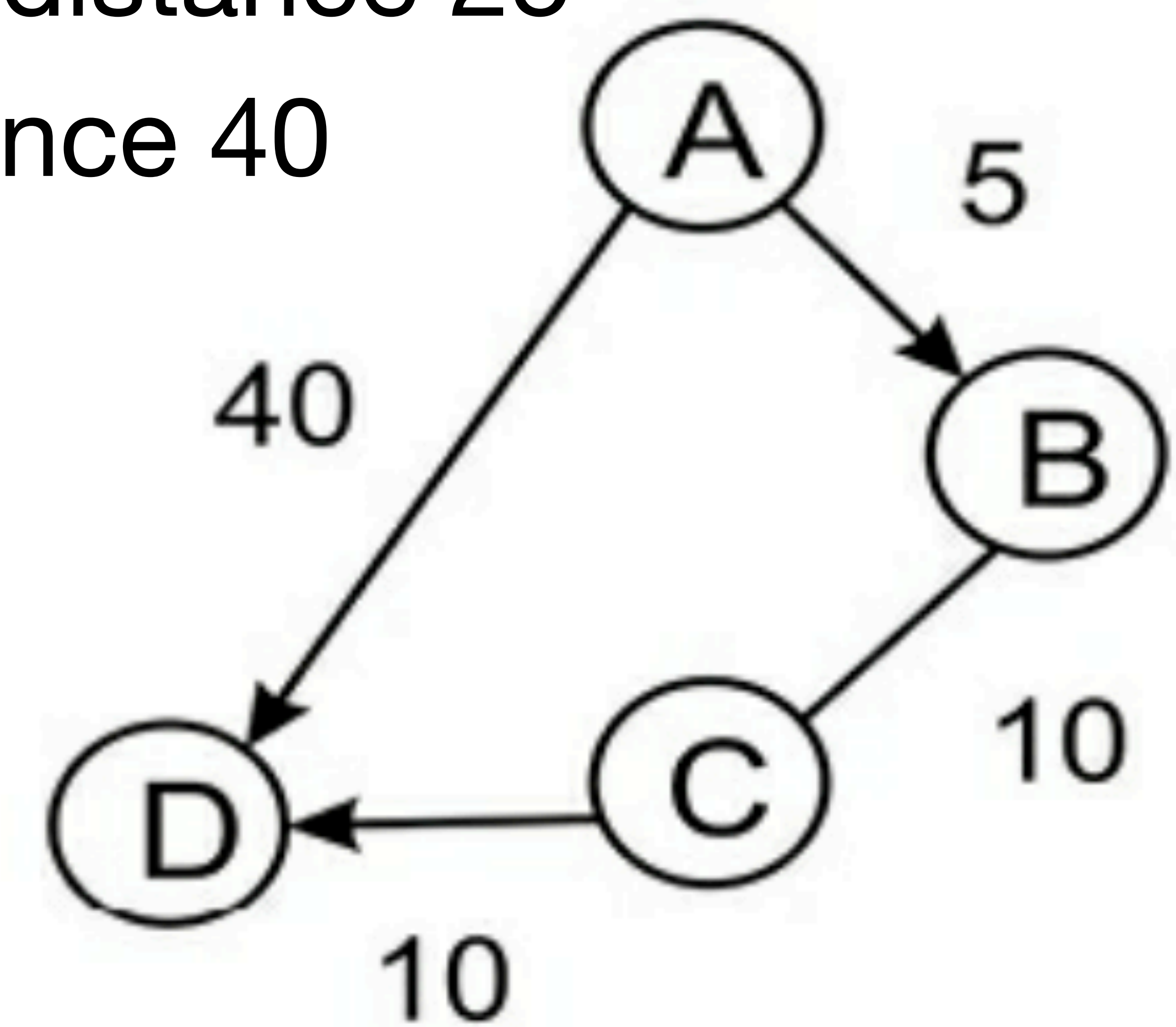
# Directed acyclic graph

- No **cycles** -- paths that end at starting node



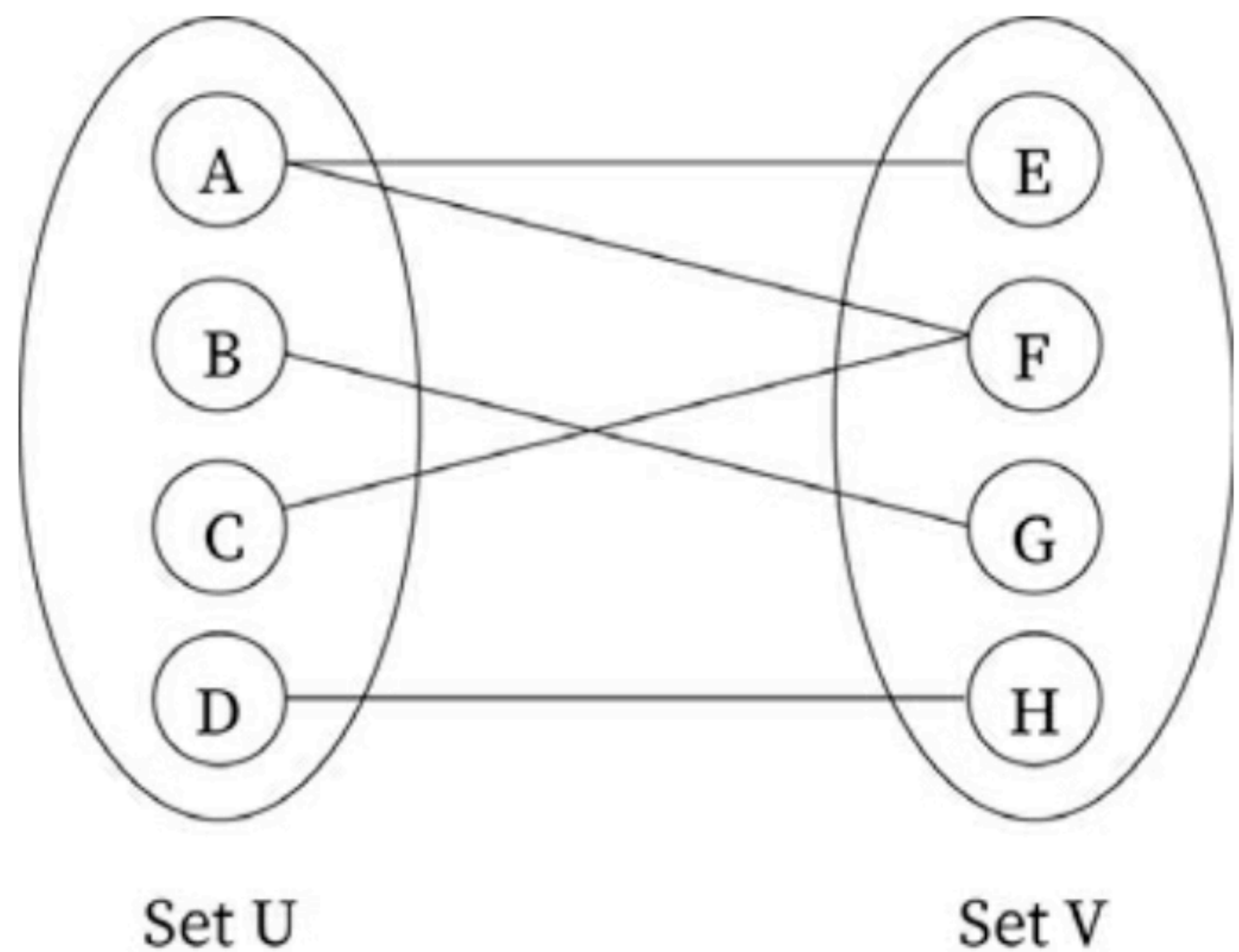
# Weighted graph

- Numeric **weight** on each edge
- Can be directed or undirected
- Path A-B-C-D has distance 25
- Path A-D has distance 40



# Bipartite graph

- Two sets of nodes
- No edge connects nodes of the same set
- Can represent relationships between different types of objects
- Like applicants and jobs



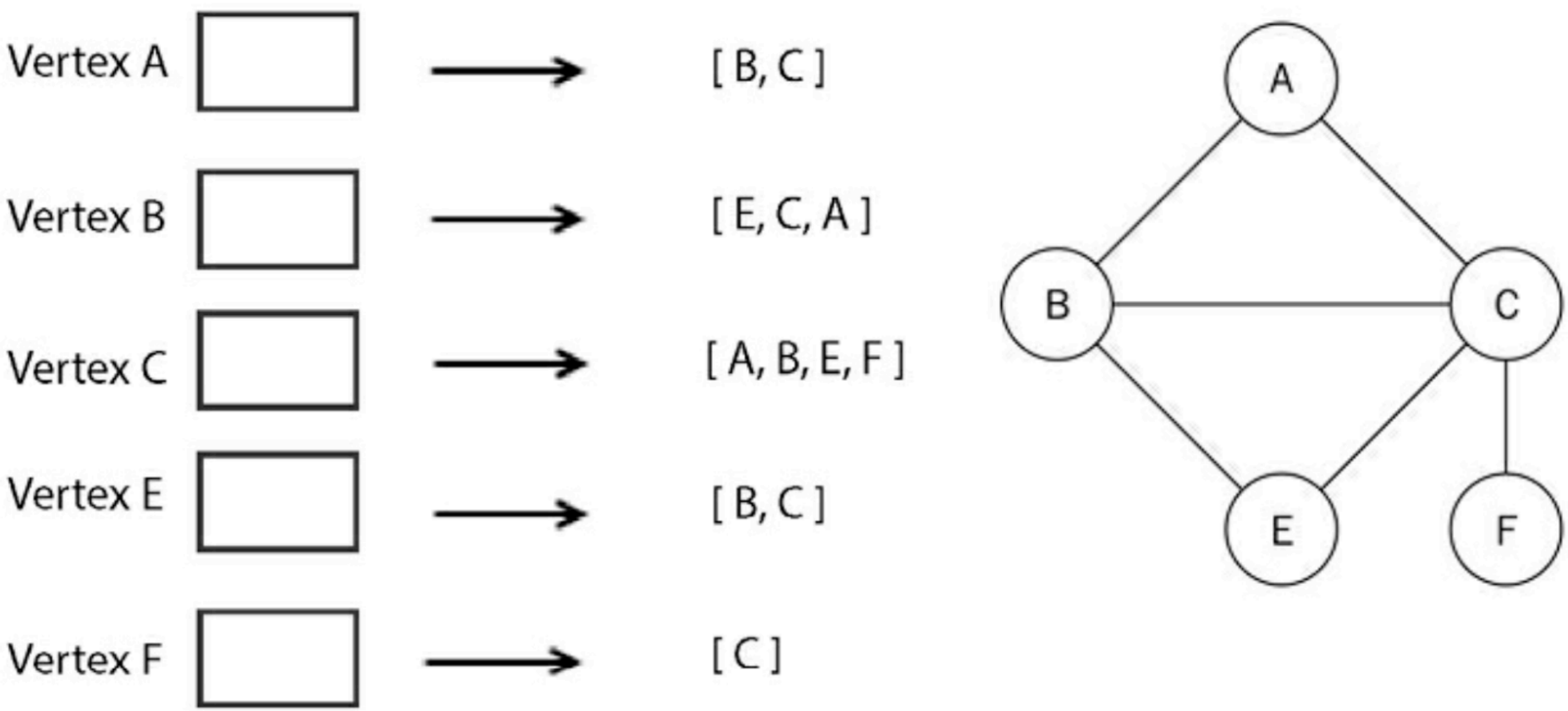
# Graph representations

# Graph representations

- How a graph is stored in memory
- Two ways
  - Adjacency list
    - Preferable for sparse graph with few edges
  - Adjacency matrix
    - Preferred for graphs with a lot of edges

# Adjacency list

- Storing this in a Python list means we can't directly use the vertex labels

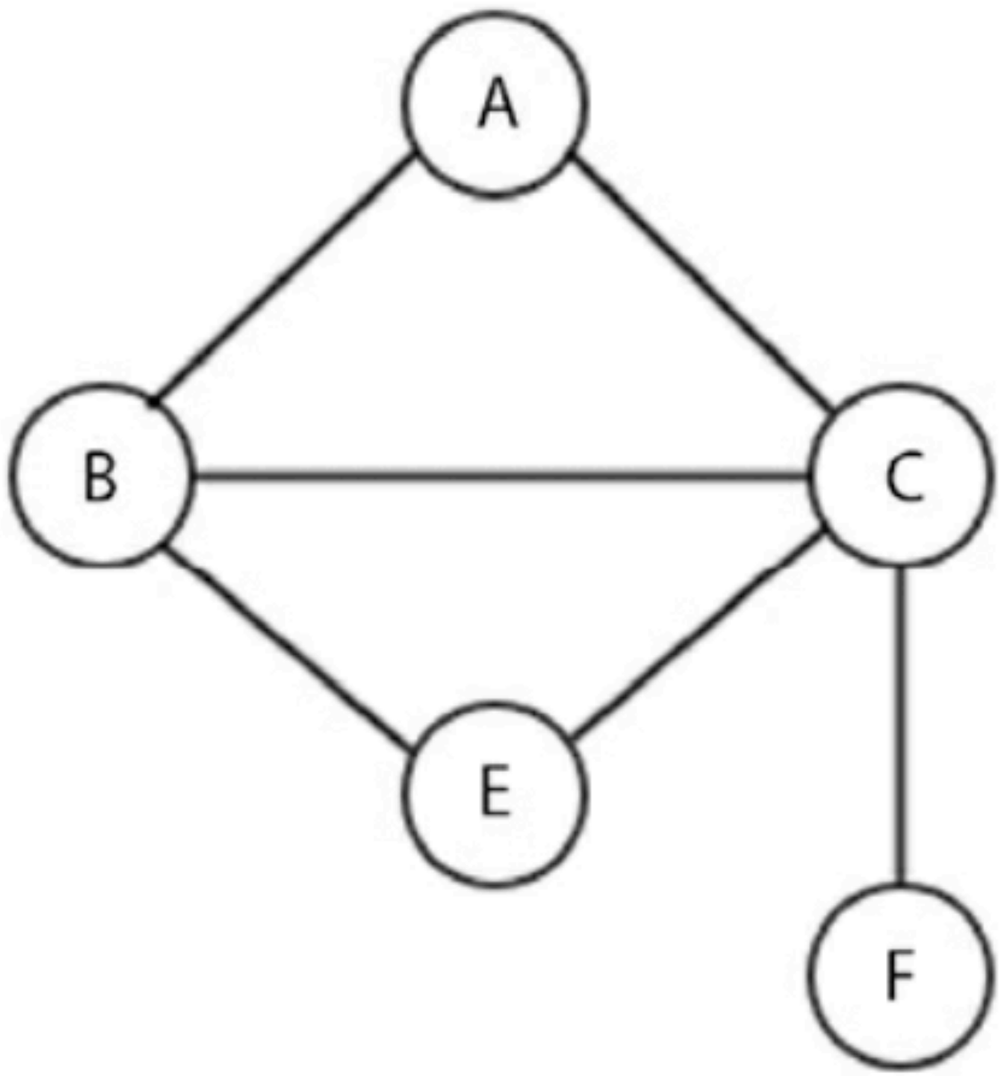


# Adjacency list

- Better to store it as a dictionary
- Easy to add and delete nodes
- But it's difficult to check whether an edge is present
  - Such as CF

```
graph = dict()
graph['A'] = ['B', 'C']
graph['B'] = ['E','C', 'A']
graph['C'] = ['A', 'B', 'E','F']
graph['E'] = ['B', 'C']
graph['F'] = ['C']
```

# Adjacency matrix




Adjacency Matrix

| | A | B | C | E | F |
|---|---|---|---|---|---|
| A | 0 | 1 | 1 | 0 | 0 |
| B | 1 | 0 | 1 | 1 | 0 |
| C | 1 | 1 | 0 | 1 | 1 |
| E | 0 | 1 | 1 | 0 | 0 |
| F | 0 | 0 | 1 | 0 | 0 |

# Adjacency matrix

```
matrix_elements = sorted(graph.keys())
cols = rows = len(matrix_elements)
```

```
adjacency_matrix = [[0 for x in range(rows)] for y in range(cols)]
edges_list = []
```

```
for key in matrix_elements:
    for neighbor in graph[key]:
        edges_list.append((key, neighbor))
print(edges_list)
```

```
[('A', 'B'), ('A', 'C'), ('B', 'E'), ('B', 'C'), ('B', 'A'),
('C', 'A'), ('C', 'B'), ('C', 'E'), ('C', 'F'), ('E', 'B'),
('E', 'C'), ('F', 'C')]
```

# Adjacency matrix

```
for edge in edges_list:
    index_of_first_vertex = matrix_elements.index(edge[0])
    index_of_second_vertex = matrix_elements.index(edge[1])
    adjacency_matrix[index_of_first_vertex][index_of_second_vertex] = 1
print(adjacency_matrix)
```

- Suitable when we frequently need to look up and check presence of an edge between two nodes
- Not suitable if we frequently add or delete nodes

```
[0, 1, 1, 0, 0]
[1, 0, 0, 1, 0]
[1, 1, 0, 1, 1]
[0, 1, 1, 0, 0]
[0, 0, 1, 0, 0]
```

# Graph traversals

# Graph traversal

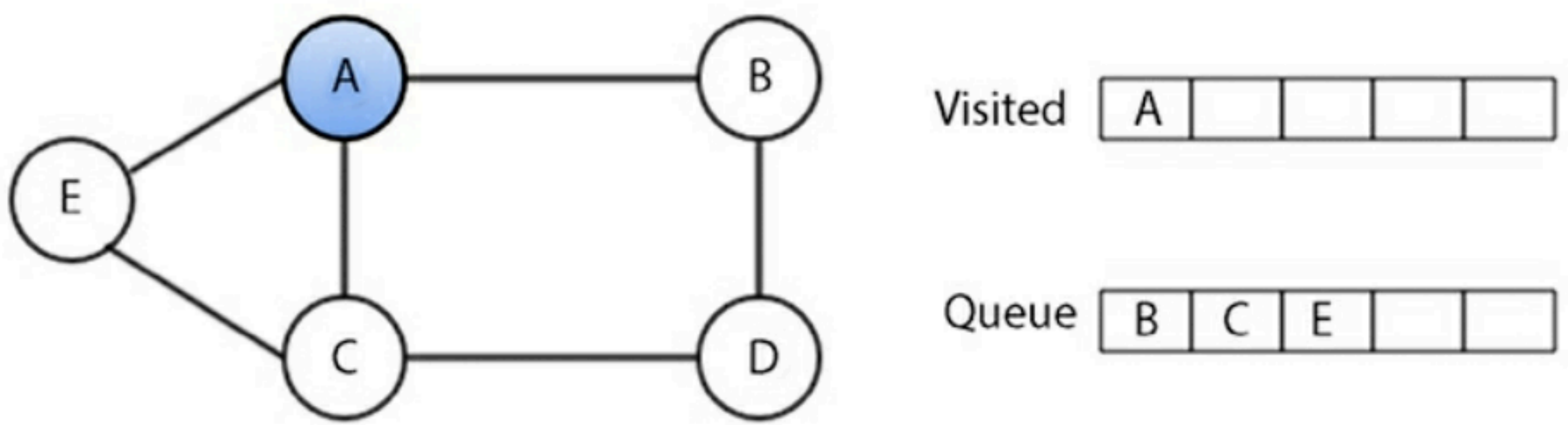
- List all vertices
  - While keeping track which vertices have been visited
- Similar to tree traversal

# Breadth-first search (BFS)

- First visit root node
- Then all nodes connected to root
- Then nodes 2 hops from the root
- etc.

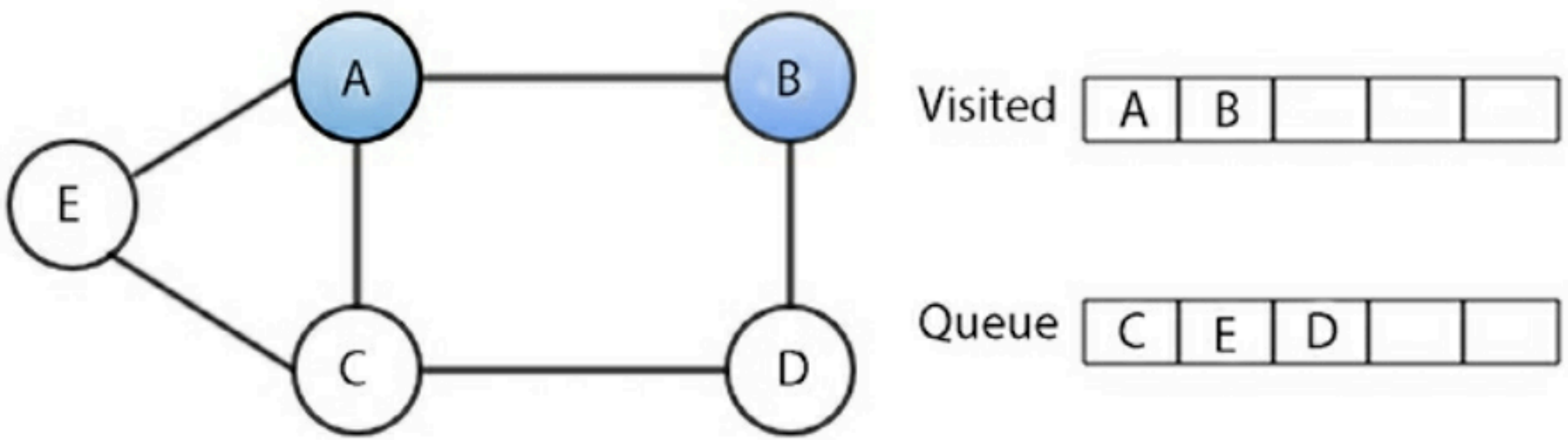
# Breadth-first search (BFS)

- Visit root **A**
- Load queue with adjacent vertices
  - In any order
  - Here, alphabetical order



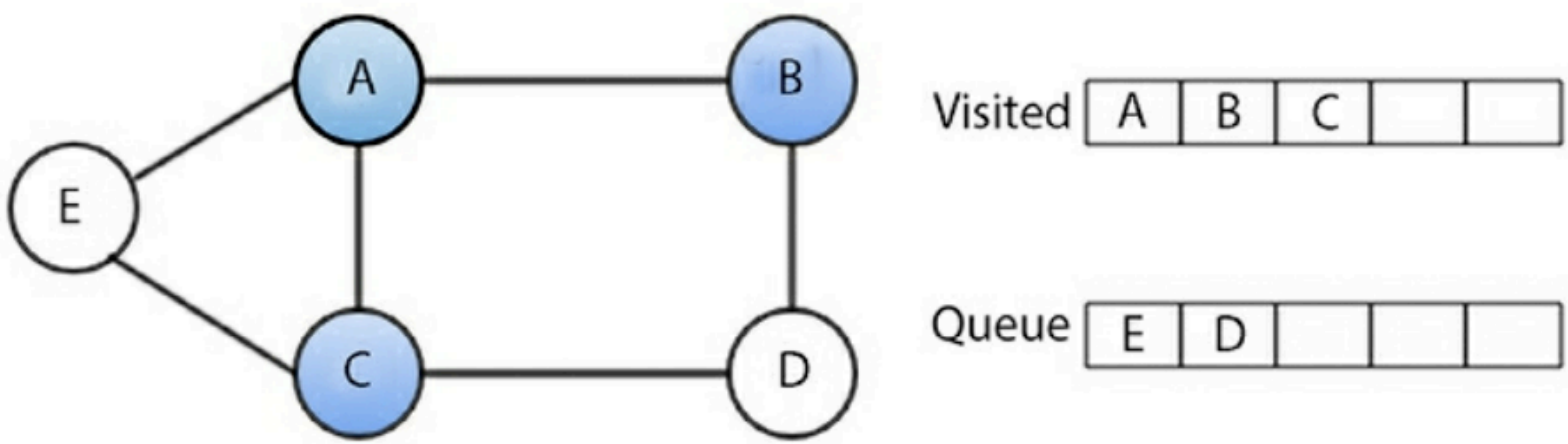
# Breadth-first search (BFS)

- Visit next node in queue: **B**
- Add adjacent vertices to the queue



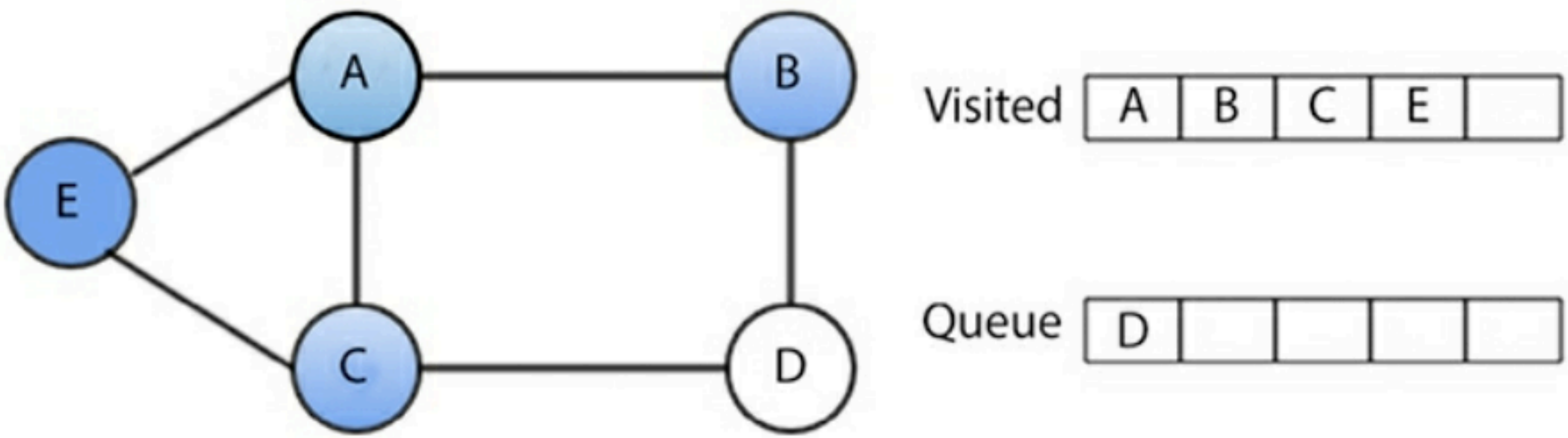
# Breadth-first search (BFS)

- Visit next node in queue: **C**
- Add adjacent vertices to the queue
  - Excluding ones already visited or queued
  - No new nodes added



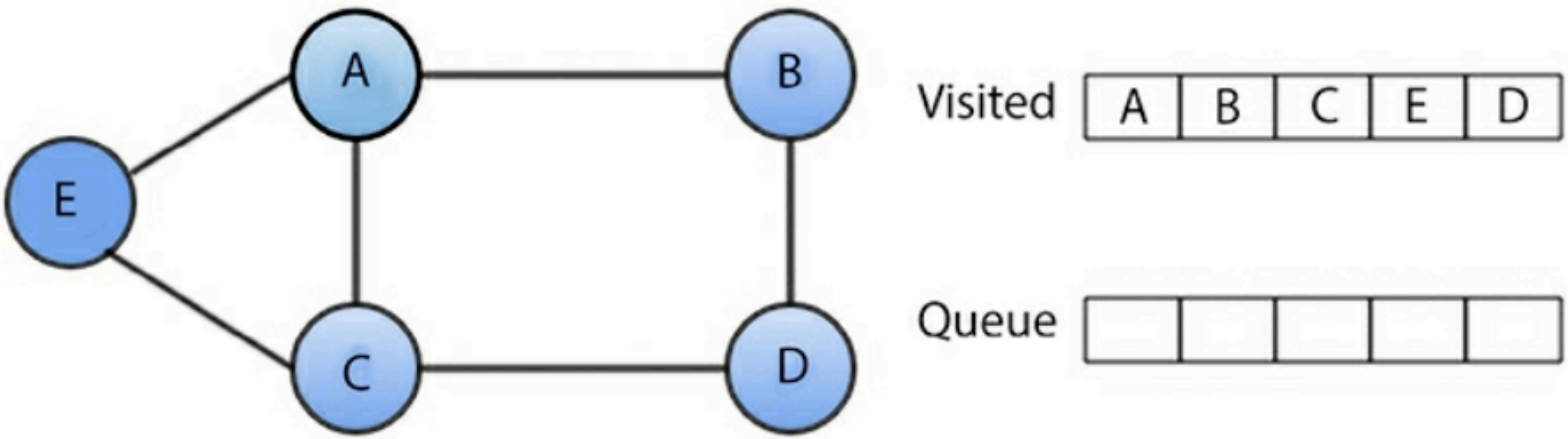
# Breadth-first search (BFS)

- Visit next node in queue: **E**
  - No new nodes to queue



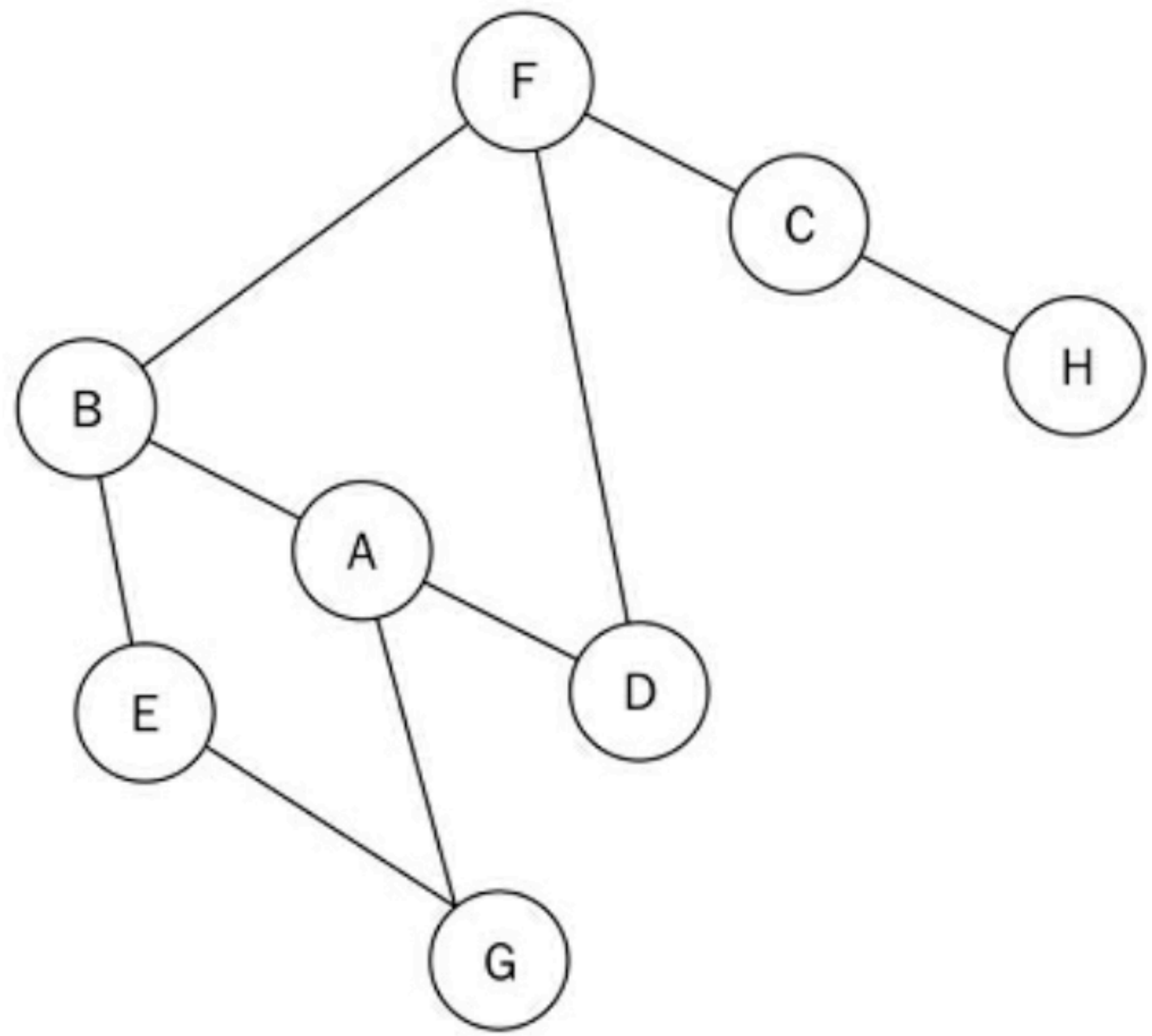
# Breadth-first search (BFS)

- Visit next node in queue: **D**
  - No new nodes to queue



# Breadth-first search (BFS)

```
graph = dict()
graph['A'] = ['B', 'G', 'D']
graph['B'] = ['A', 'F', 'E']
graph['C'] = ['F', 'H']
graph['D'] = ['F', 'A']
graph['E'] = ['B', 'G']
graph['F'] = ['B', 'D', 'C']
graph['G'] = ['A', 'E']
graph['H'] = ['C']
```

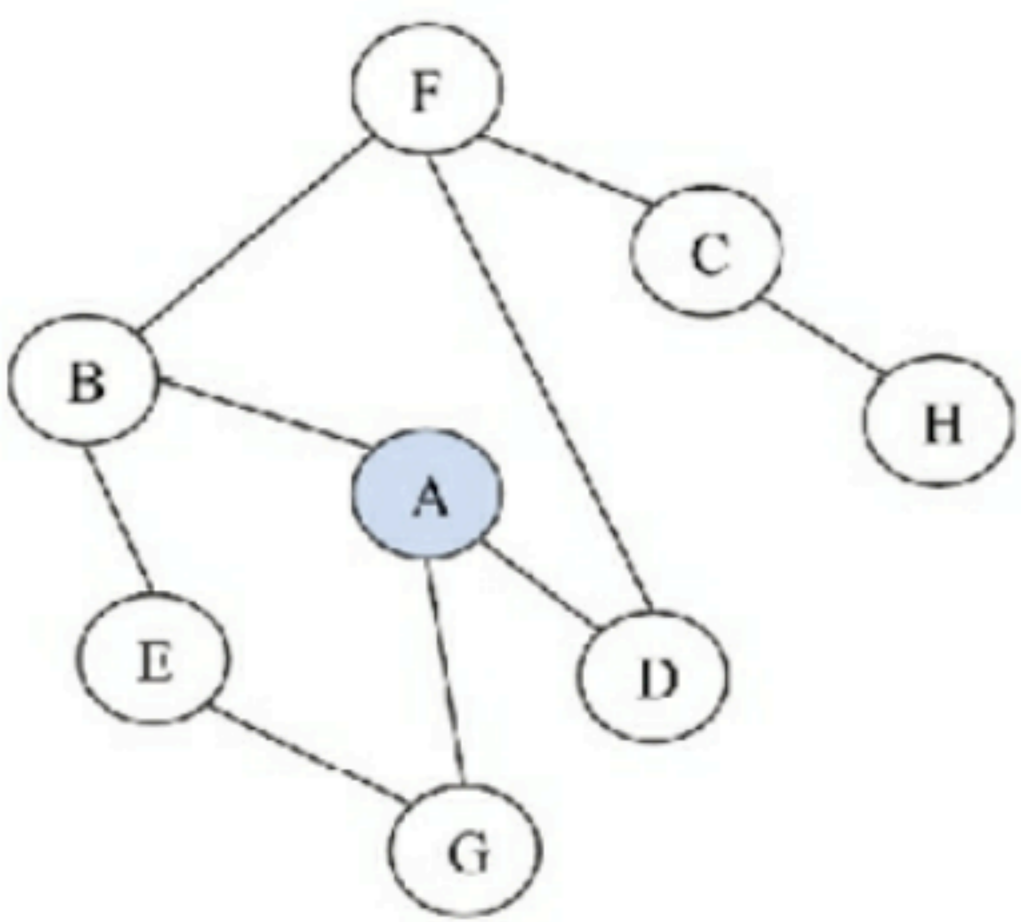


# Breadth-first search (BFS)

```
from collections import deque
def breadth_first_search(graph, root):
    visited_vertices = list()
    graph_queue = deque([root])
    visited_vertices.append(root)
    node = root
    while len(graph_queue) > 0:
        node = graph_queue.popleft()
        adj_nodes = graph[node]
        remaining_elements = set(adj_nodes).difference(set(visited_vertices))
        if len(remaining_elements) > 0:
             for elem in sorted(remaining_elements):
                 visited_vertices.append(elem)
                 graph_queue.append(elem)
    return visited_vertices
```

# Breadth-first search (BFS)

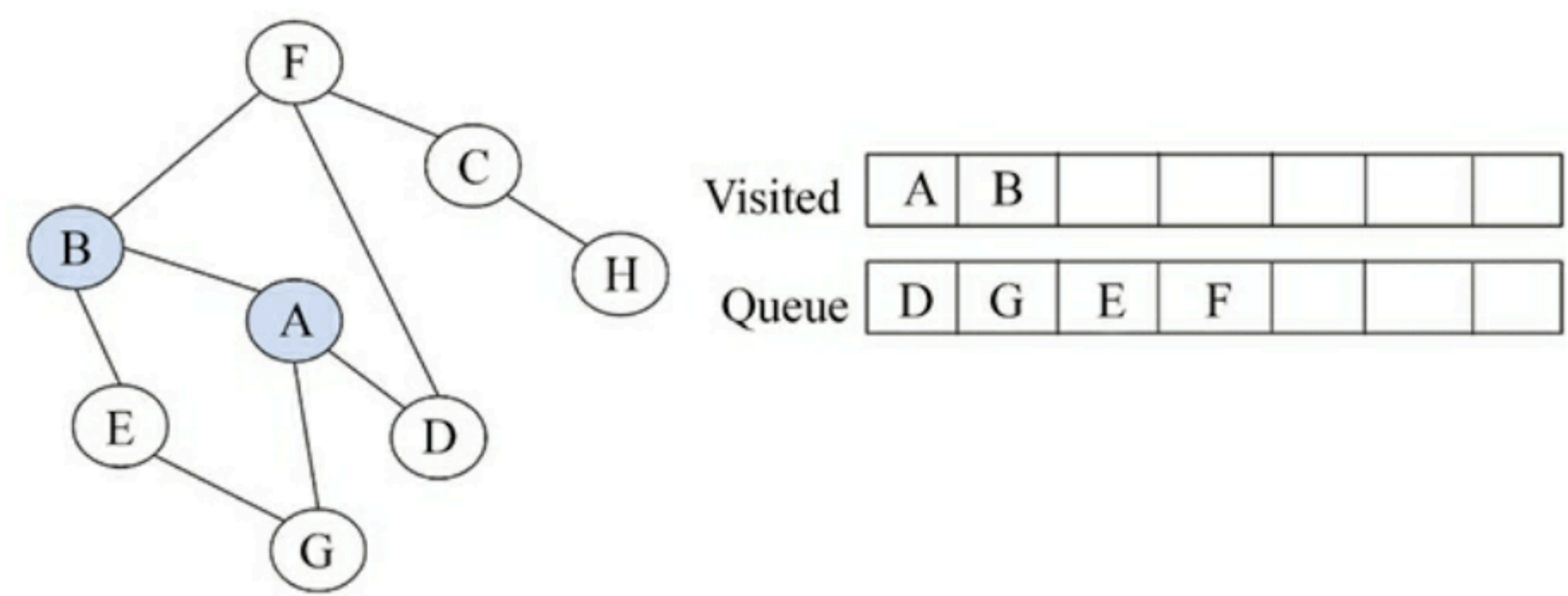
- Visit root **A**
- Add adjacent nodes to queue




| | | | | | | | |
|---|---|---|---|---|---|---|---|
| Visited | A | | | | | | |
| Queue | B | D | G | | | | |

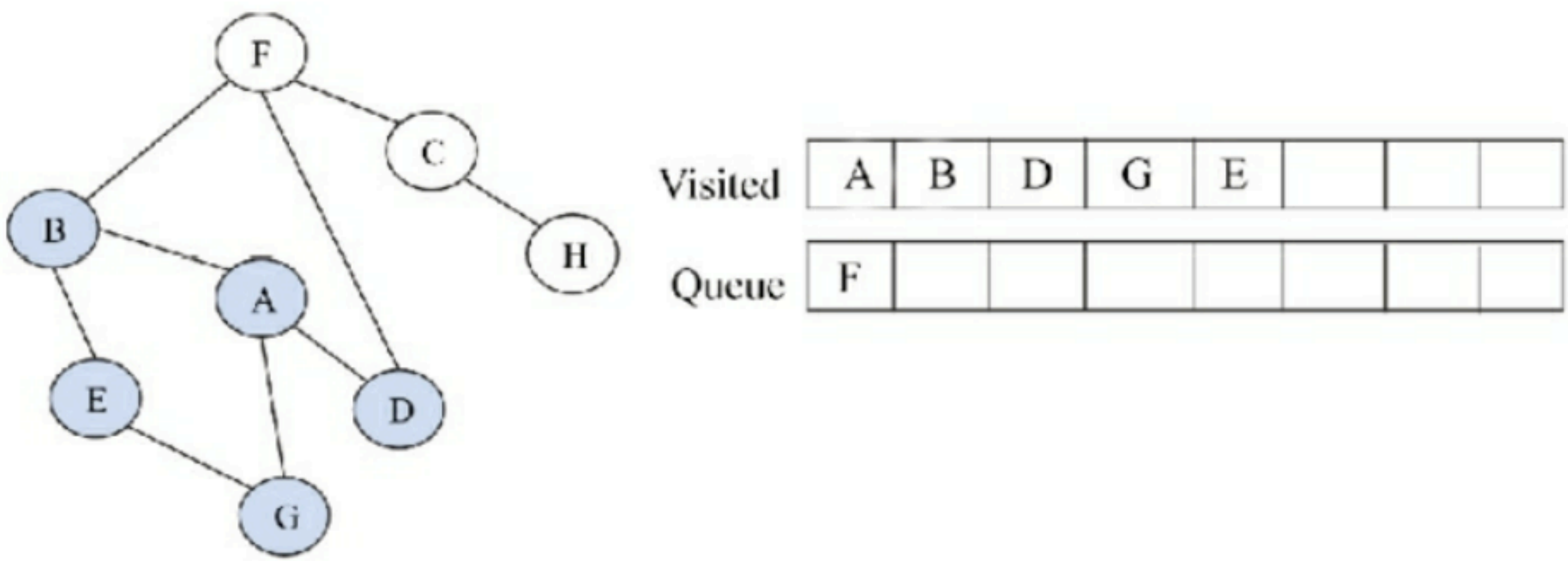
# Breadth-first search (BFS)

- Visit **B**
- Add new nodes adjacent to B to the queue



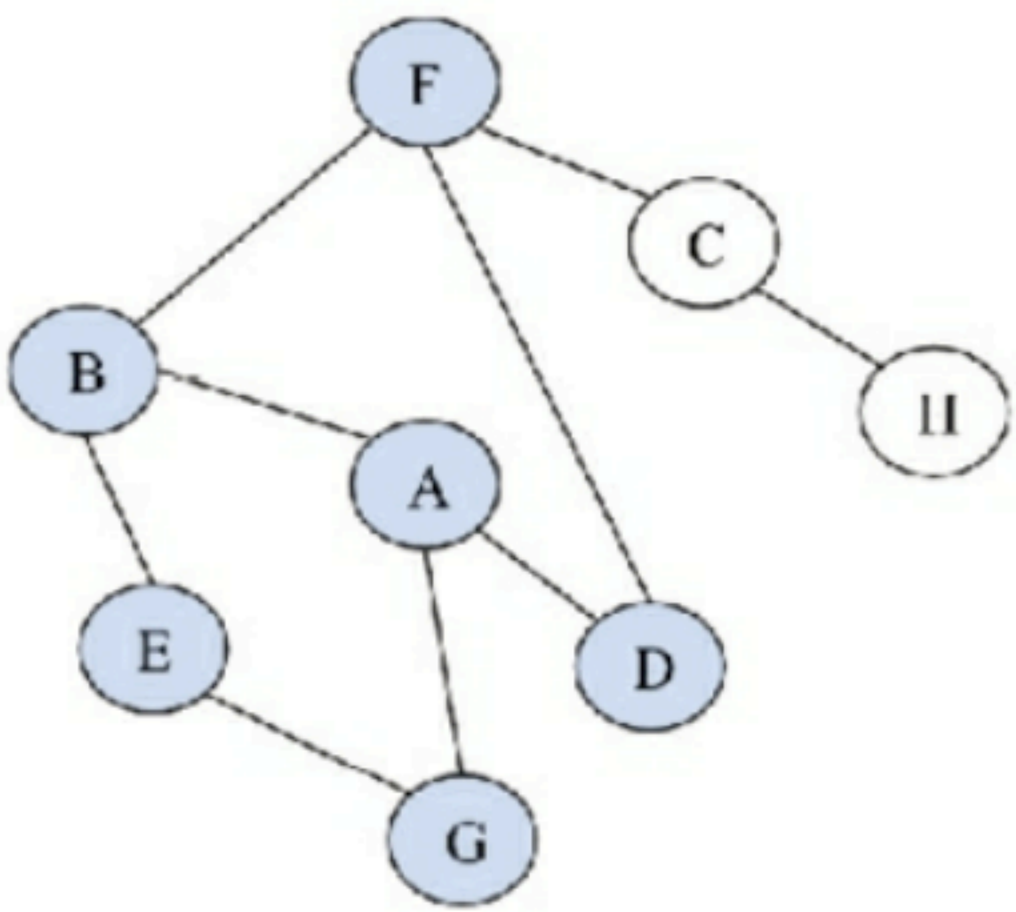
# Breadth-first search (BFS)

- Visit **D**, **G**, **E**
- No new nodes adjacent to them



# Breadth-first search (BFS)

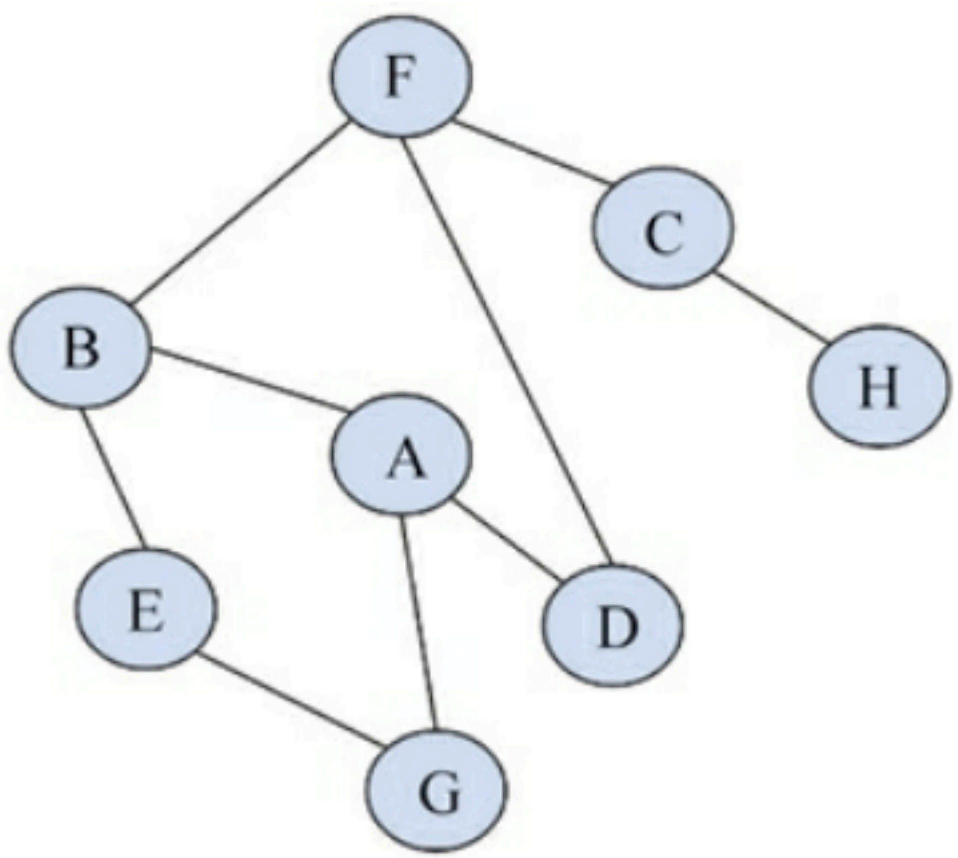
- Visit **F**
- Add **C** to queue




| | | | | | | | |
|---|---|---|---|---|---|---|---|
| Visited | A | B | D | G | E | F | |
| Queue | C | | | | | | |

# Breadth-first search (BFS)

- Visit **C**
- Add **H** to queue
- Visit **H**




| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Visited | A | B | D | G | E | F | C | H |
| Queue | | | | | | | | |

# Breadth-first search (BFS)

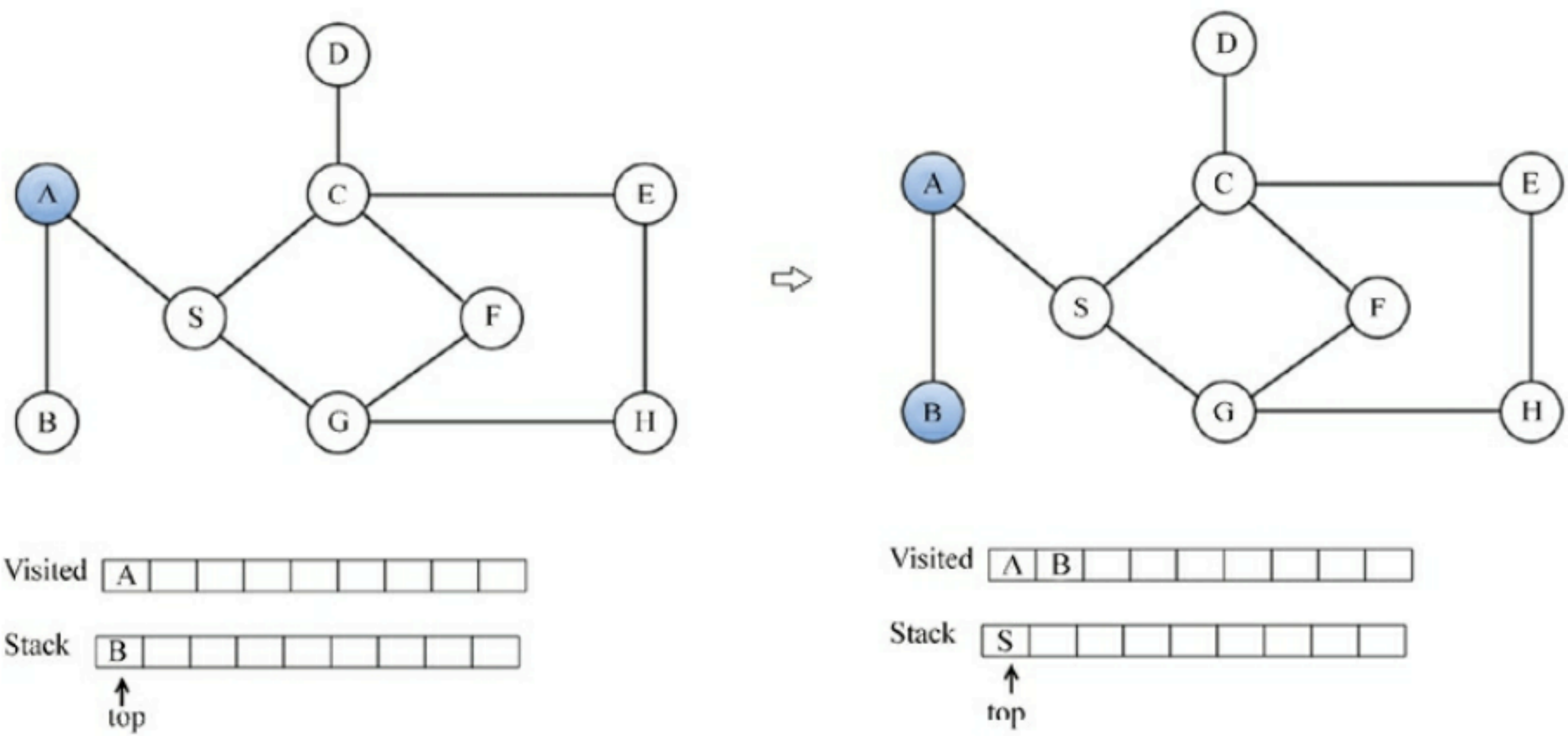
- Time complexity is O(|V| + |E|)
  - |V| is the number of vertices
  - |E| is the number of edges
- Useful for constructing the shortest path traversal in a graph with minimal iterations
- Can create an efficient web crawler
- And for a navigation system

# Depth-first search (DFS)

- Child nodes are visited before siblings
- Start at root
- Visit a new adjacent node
- Repeat until a dead end
- Then backtrack to previous nodes
- End when backtracking hits the root node

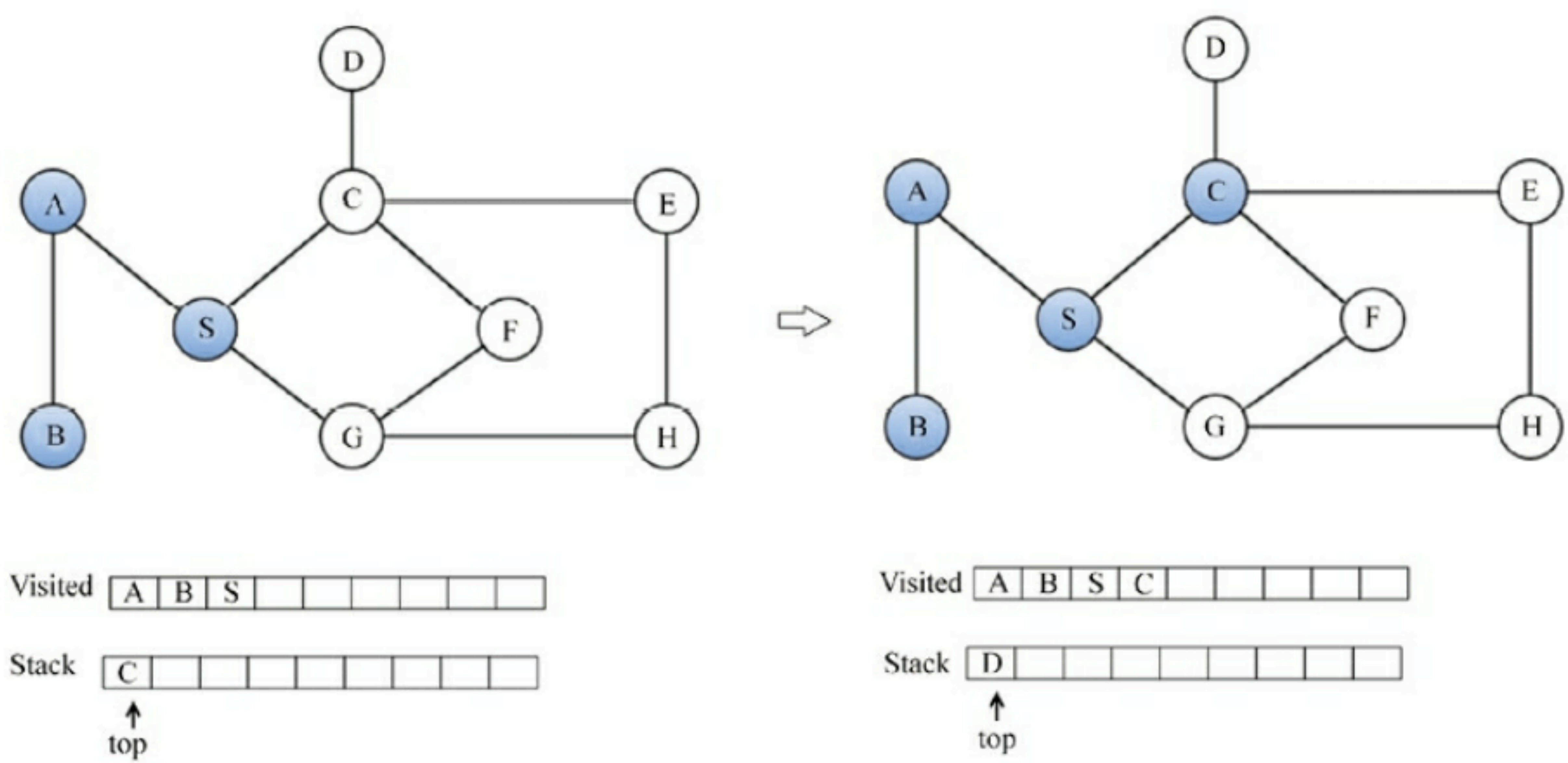
# Depth-first search (DFS)

- Visit root **A**
- Visit a neighbor **B**



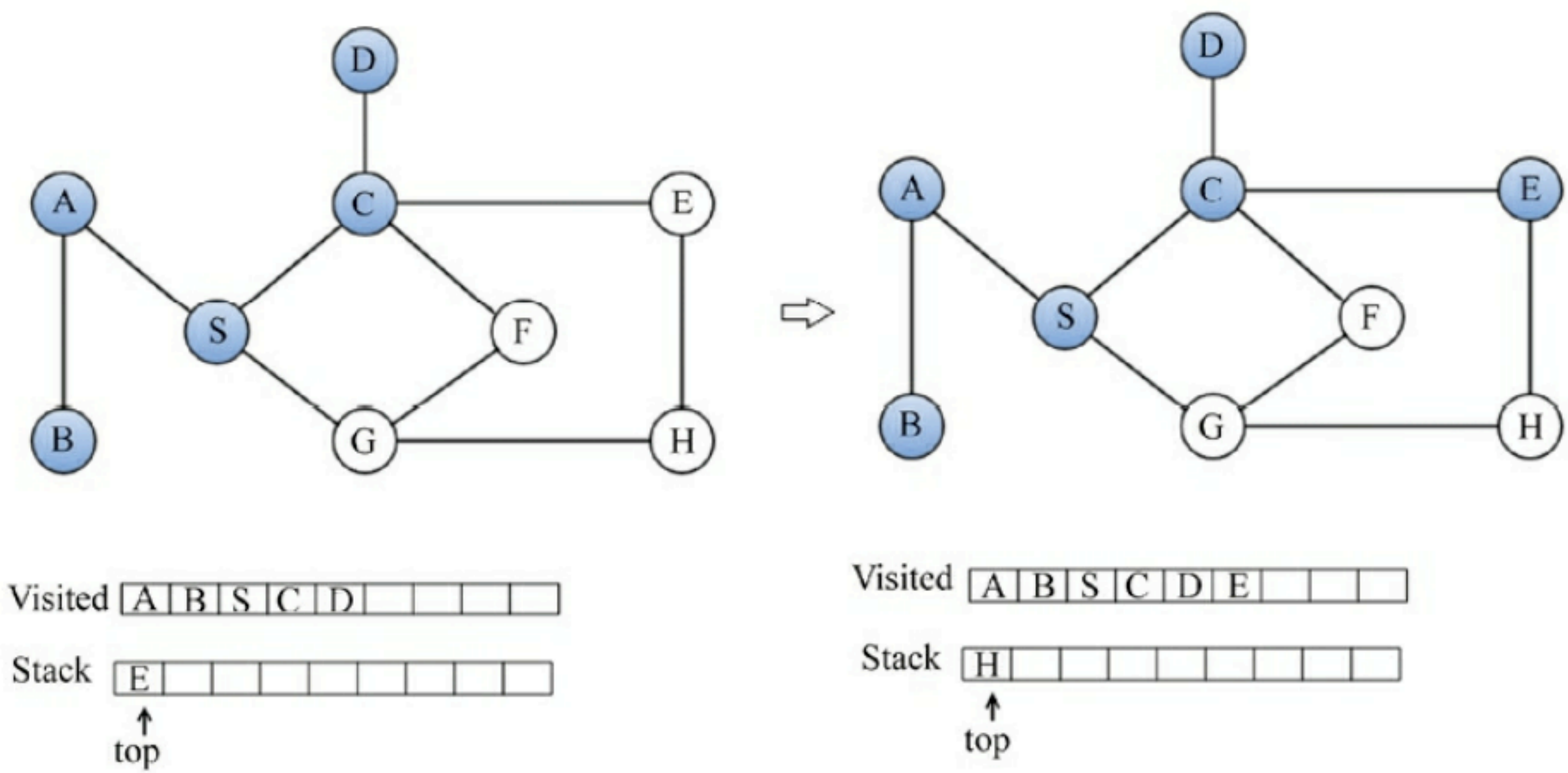
# Depth-first search (DFS)

- Visit other neighbor of root **S**
- Visit a new neighbor of S: **C**



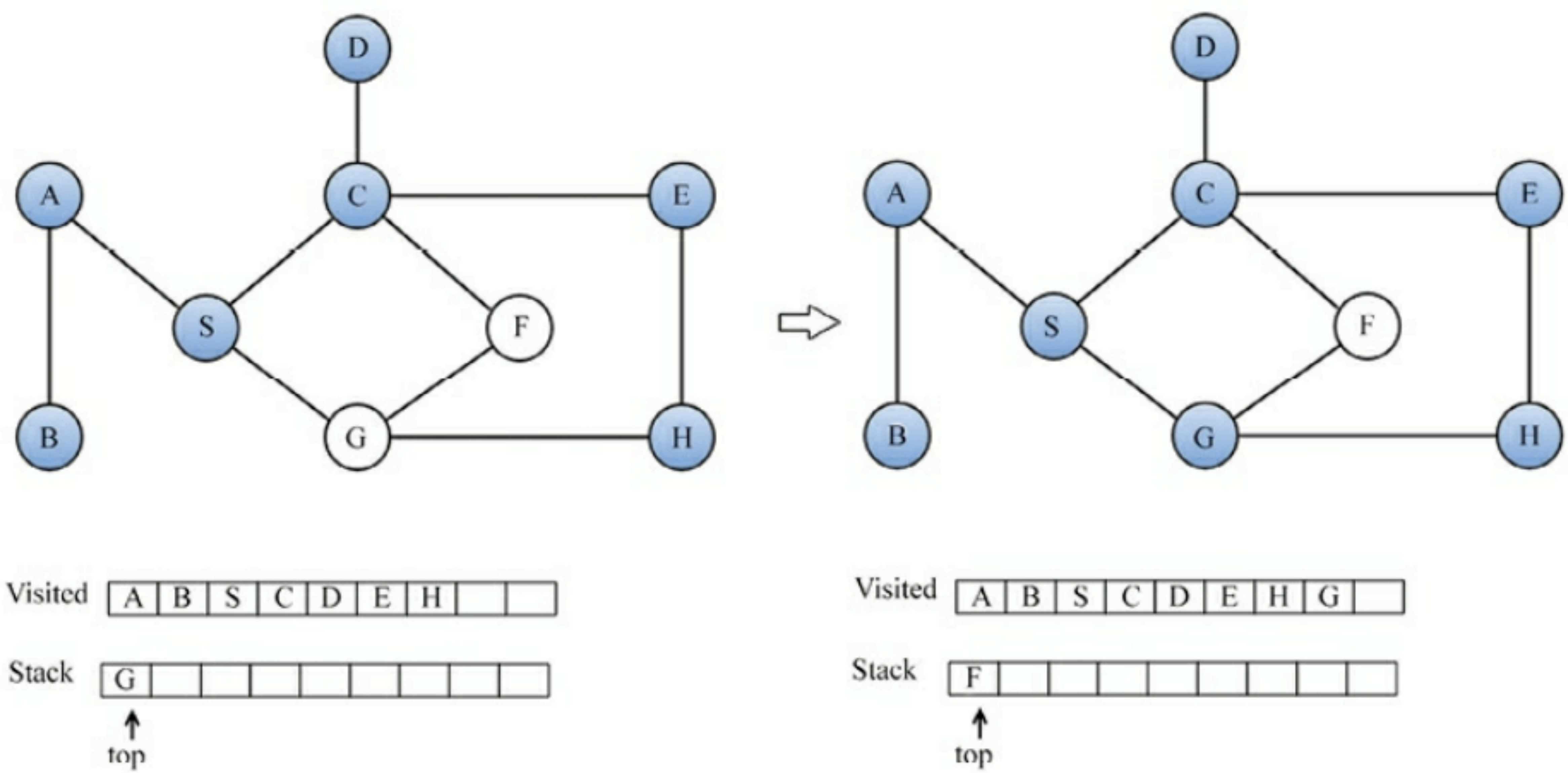
# Depth-first search (DFS)

- Visit neighbors of C: **D E**



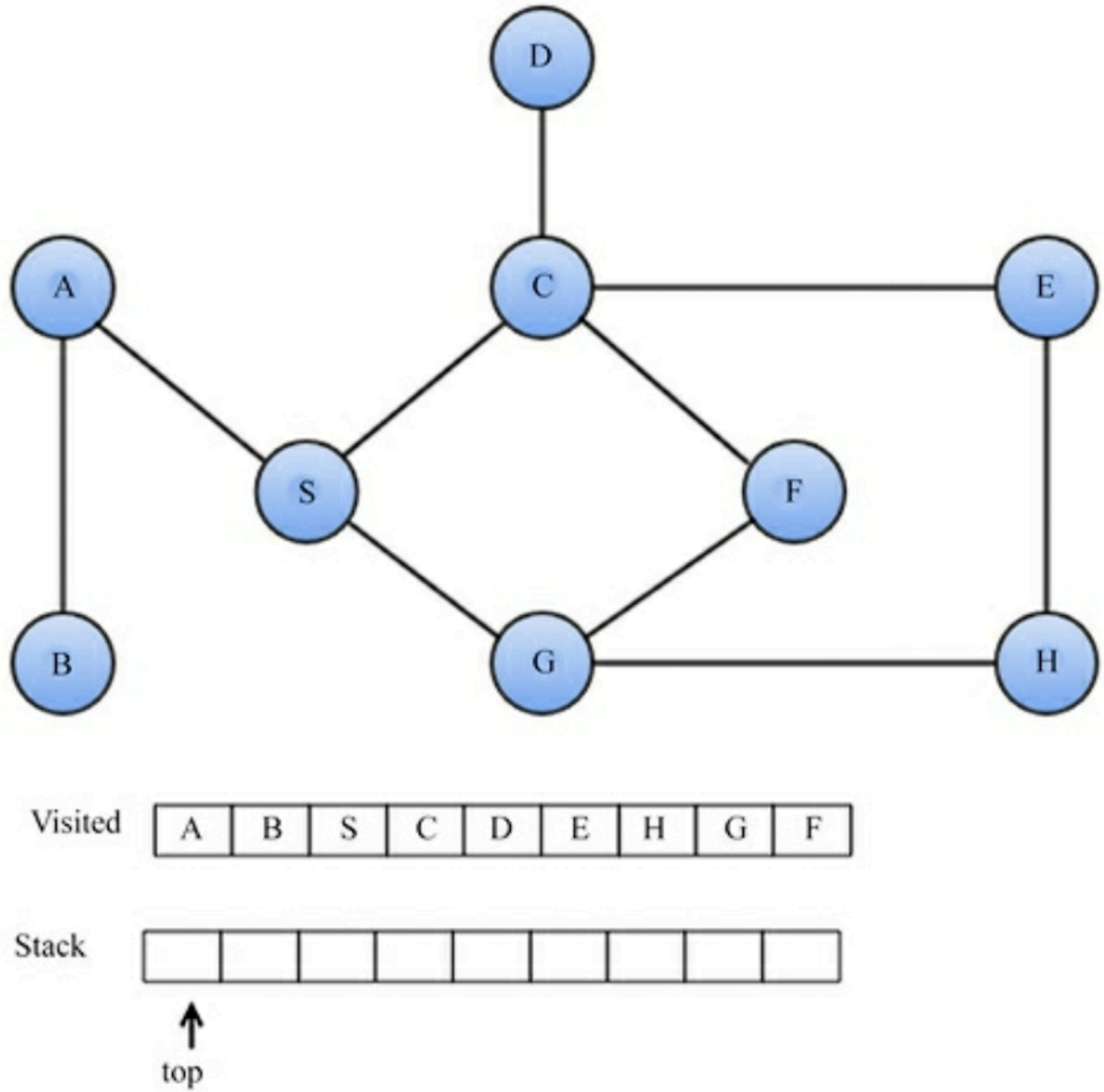
# Depth-first search (DFS)

- Visit neighbor of E: **H**
- Visit neighbor of H: **G**



# Depth-first search (DFS)

- Finally, visit **F**



# Depth-first search (DFS)

- Time complexity of DFS is
  - $O(V + E)$ when we use an adjacency list
  - $O(V^2)$ when we use an adjacency matrix

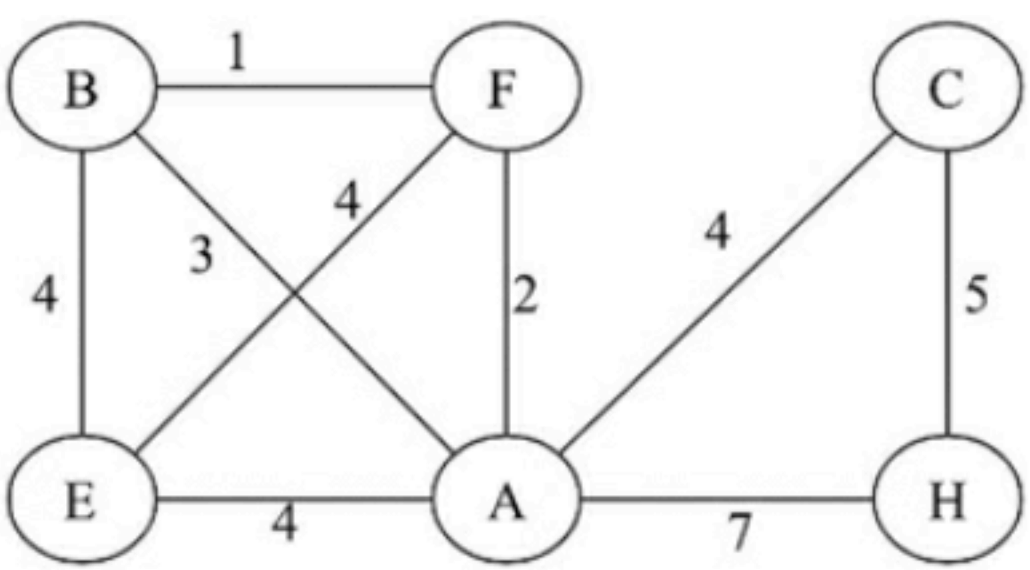
# Depth-first search (DFS)

- DFS applications
  - Solving maze problems
  - Finding connected components
  - Cycle detection in graphs
  - Finding bridges of a graph
    - Removing a "bridge" edge disconnects the graph

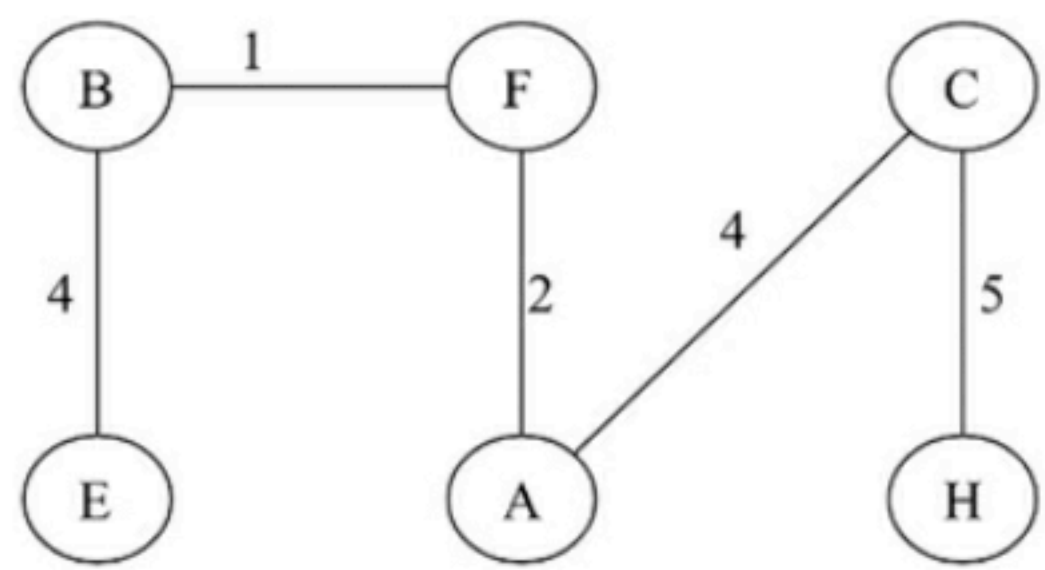
# Minimum Spanning Tree

# Minimum spanning tree

- A subset of the edges of a connected graph
- Connects all the nodes
- Lowest possible edge weight
- No cycle




A graph




A minimum spanning tree

# Kruskal's minimum spanning tree algorithm

- Greedy approach
- Find the edge with lowest weight
  - Add it to the tree
- With each iteration, repeat this process
  - Avoiding forming a cycle

# Kruskal's minimum spanning tree algorithm

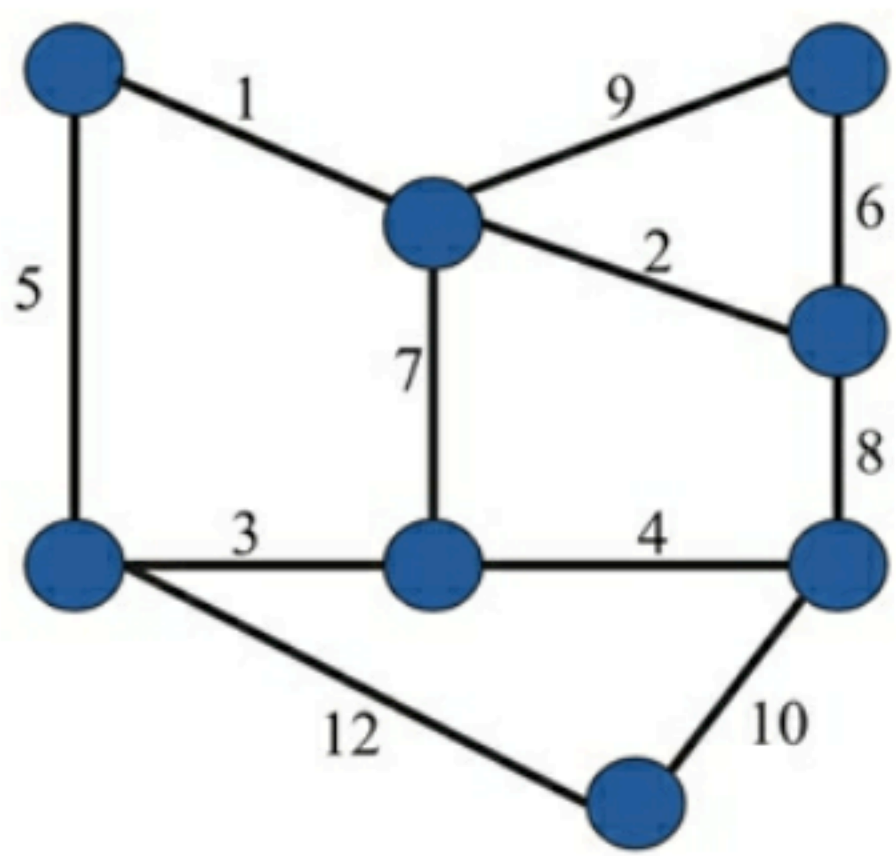
1. Initialize an empty MST (M) with zero edges
2. Sort all the edges according to their weights
3. For each edge from the sorted list, we add them one by one to the MST (M) in such a way that it does not form a cycle

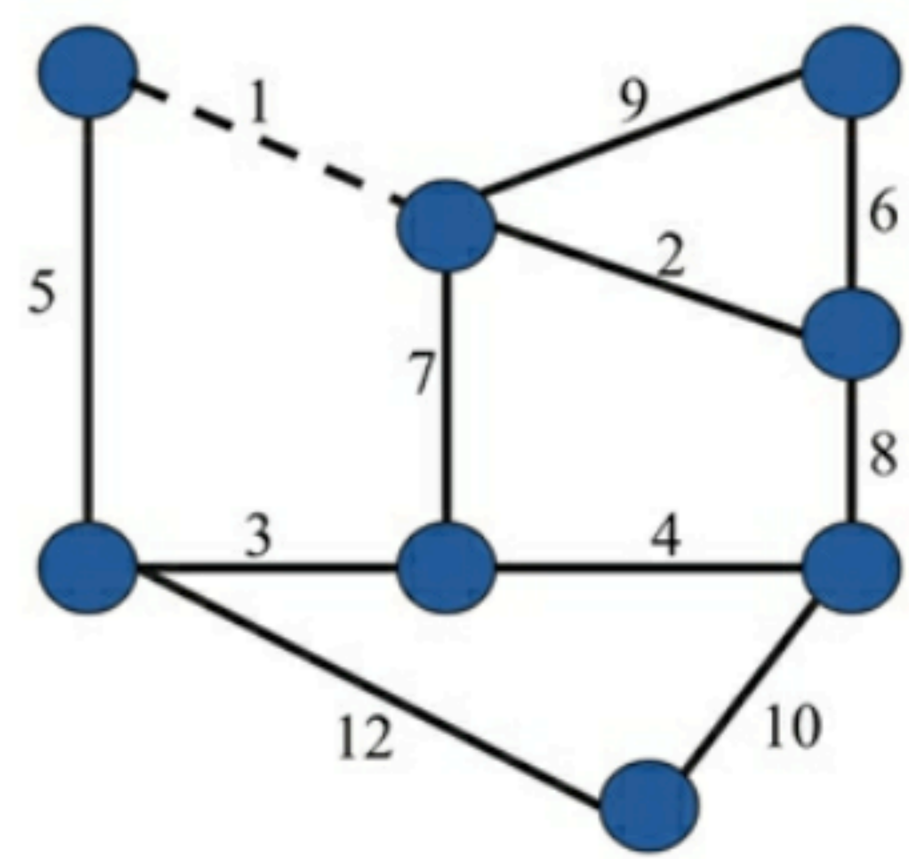
# Kruskal's minimum spanning tree algorithm

- Greedy approach
- Find the edge with lowest weight
  - Add it to the tree
- With each iteration, repeat this process
  - Avoiding forming a cycle

# Kruskal's minimum spanning tree algorithm

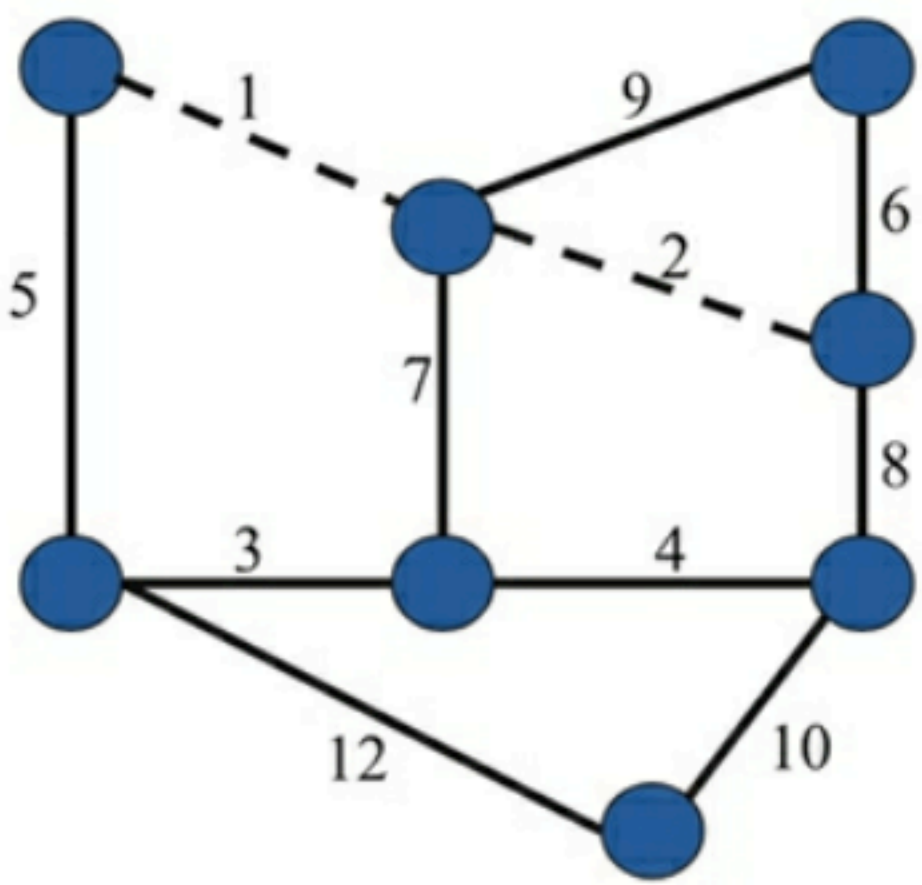
- Add dotted line to MST

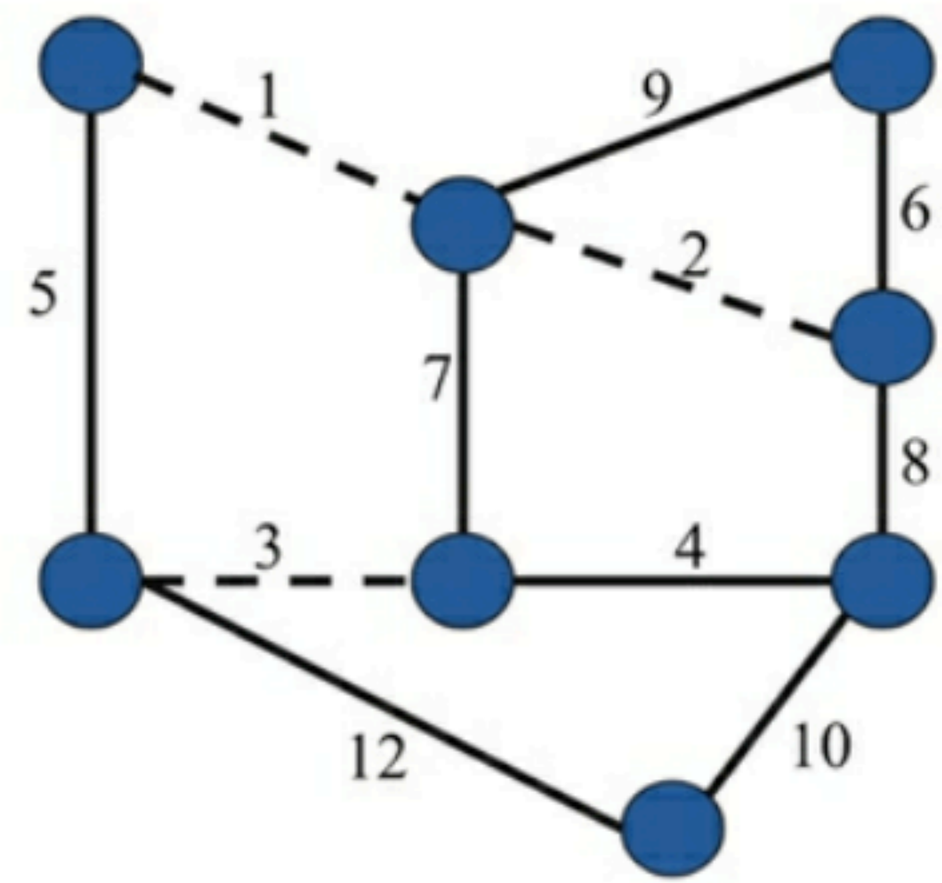


# Kruskal's minimum spanning tree algorithm

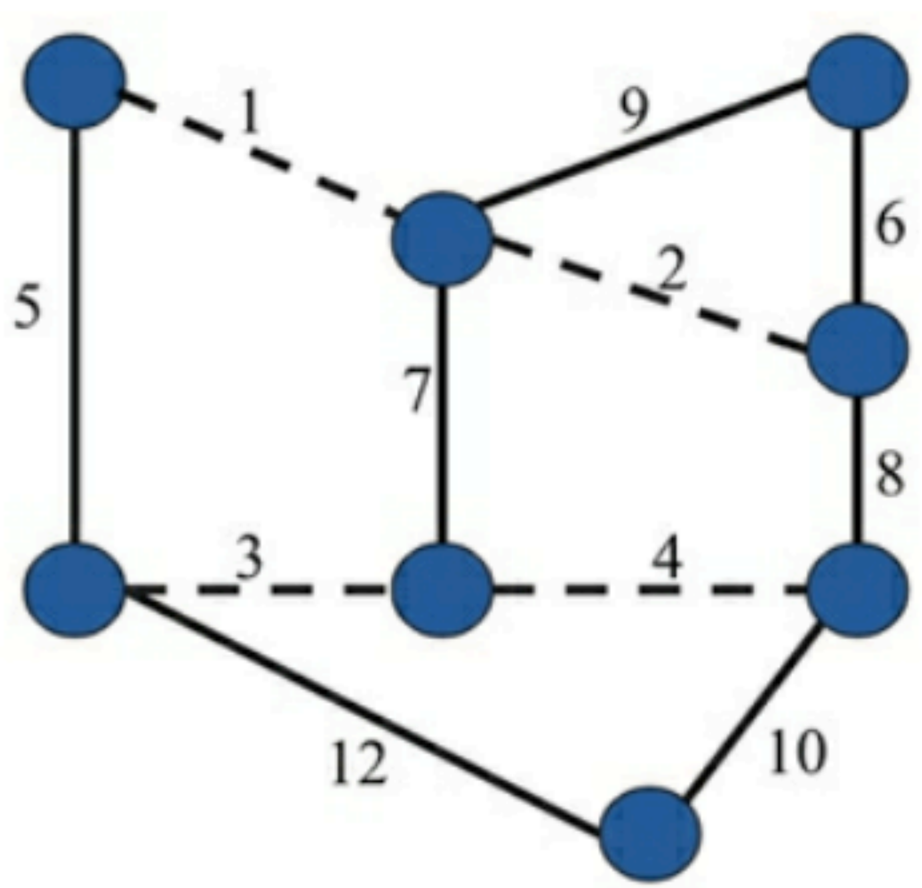
- Add lines 2 and 3

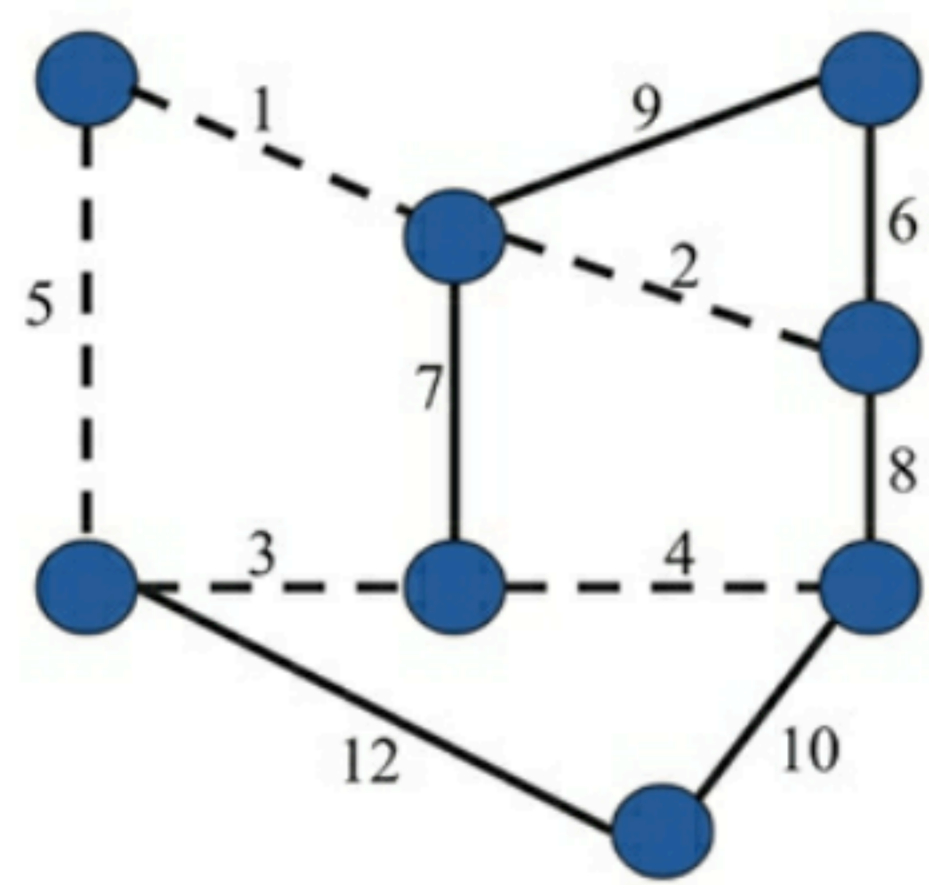


# Kruskal's minimum spanning tree algorithm

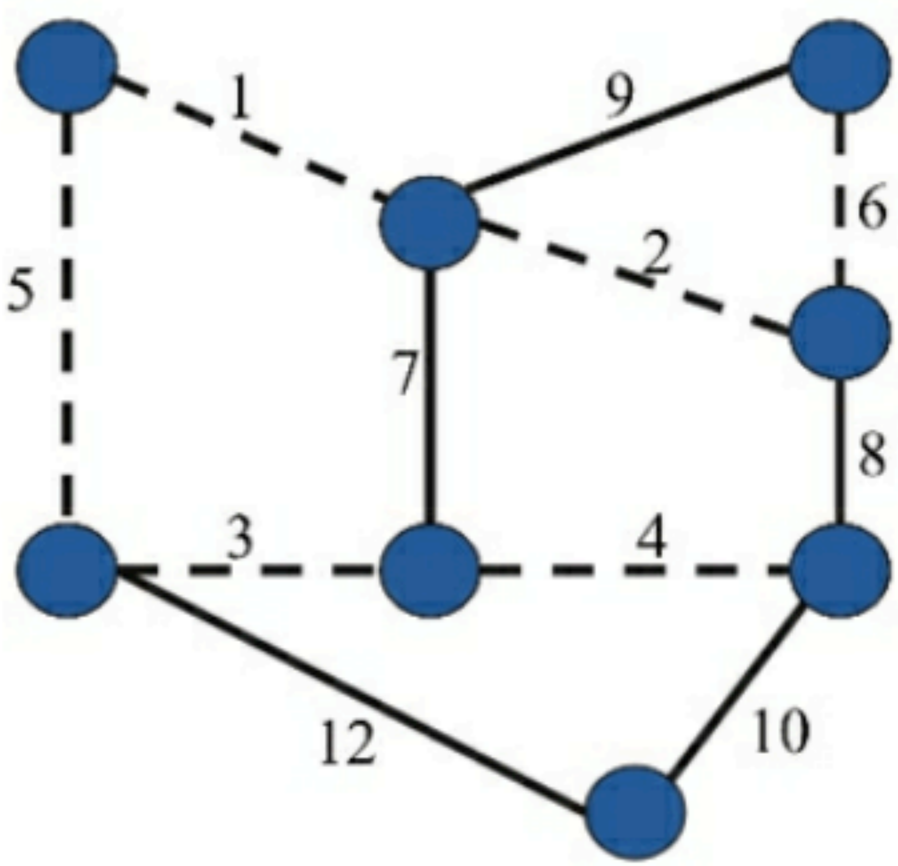
- Add lines 4 and 5

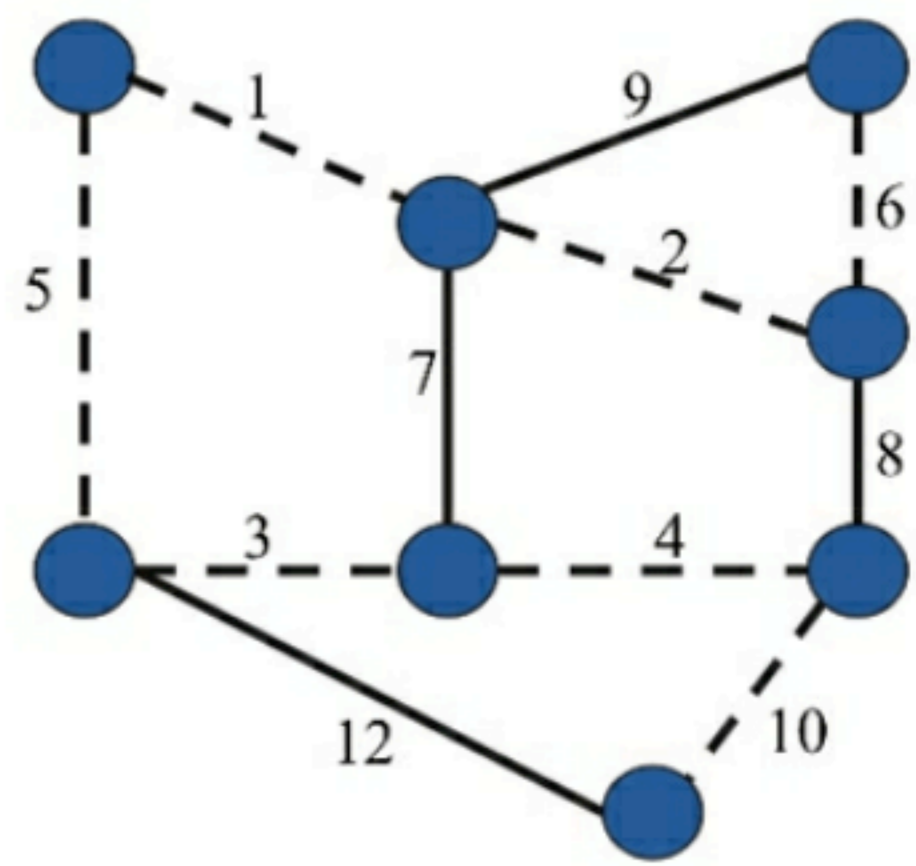


# Kruskal's minimum spanning tree algorithm

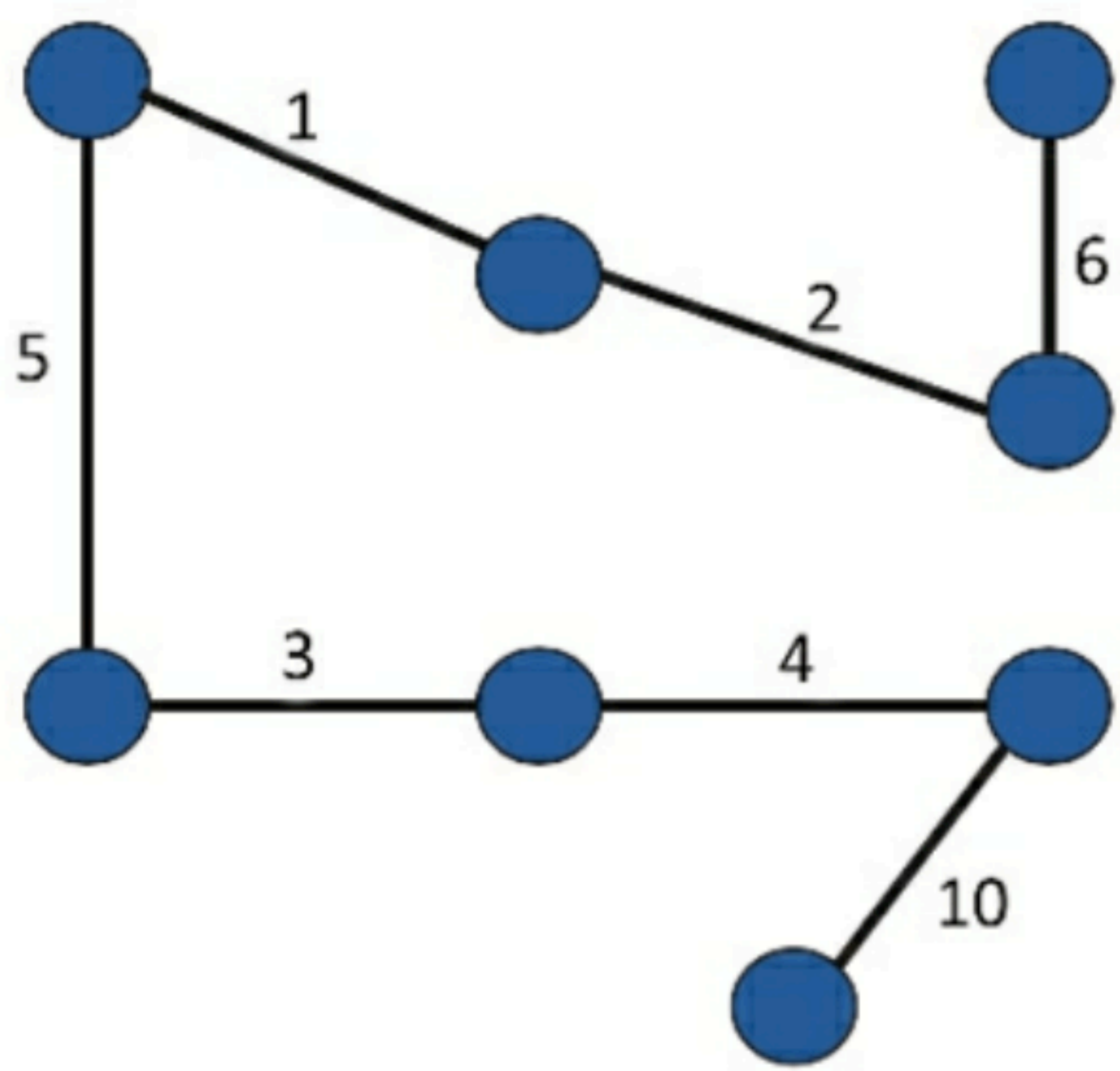
- Add line 6
- Lines 7, 8, and 9 would form a cycle, so skip them
- Add line 10



# Kruskal's minimum spanning tree algorithm

- Final spanning tree



# Kruskal's minimum spanning tree algorithm

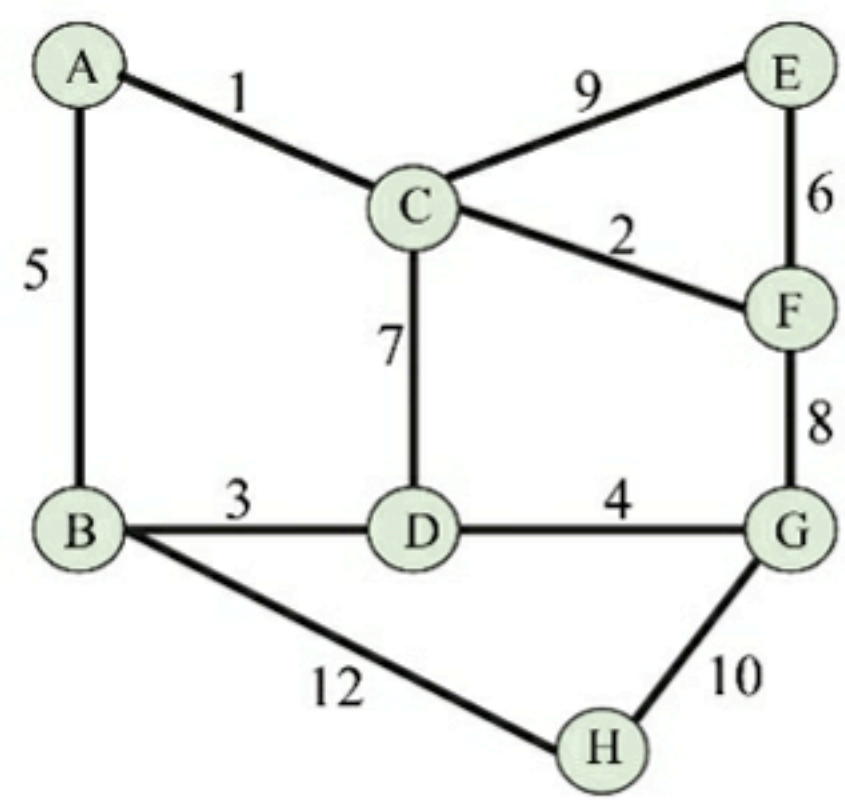
- Applications
  - Traveling salesman problem
  - TV networks
  - Tour operations
  - LAN networks
  - Electric grids
- Time complexity
  - O(E log E) or O(E log(V))

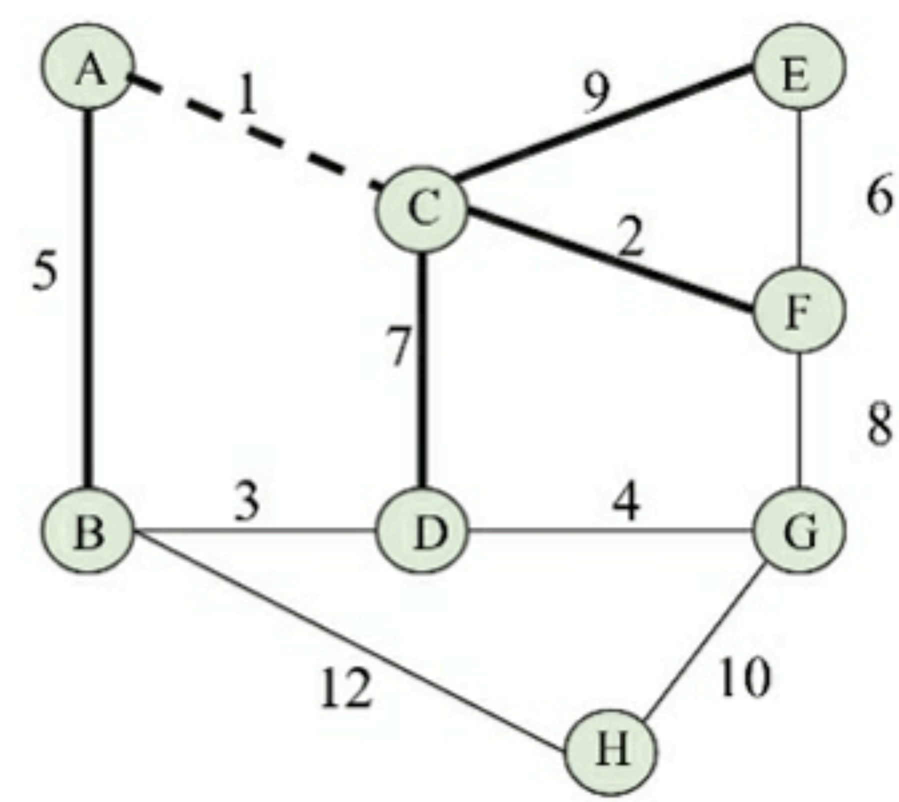
# Prim's minimum spanning tree algorithm

1. Create a dictionary that holds all the edges and their weights
2. Get the edges, one by one, that have the lowest cost from the dictionary and grow the tree in such a way that the cycle is not formed
3. Repeat step 2 until all the vertices are visited

# Prim's minimum spanning tree algorithm

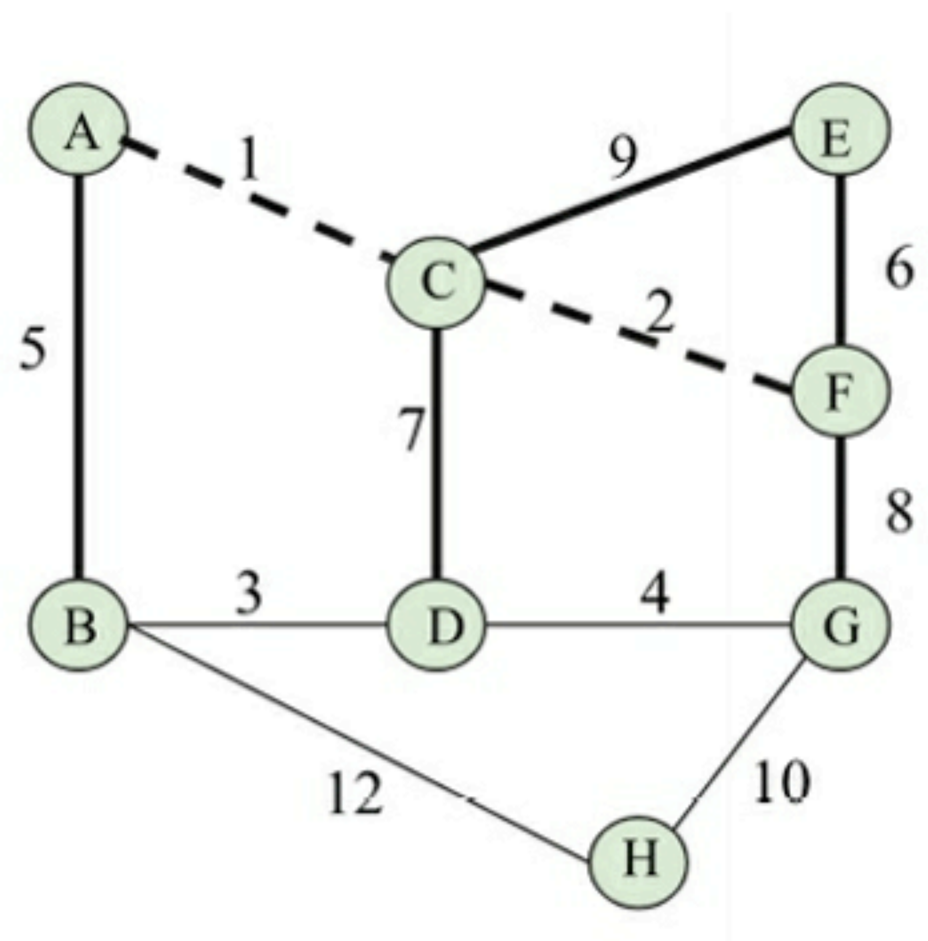
- Add shortest path from A: **AC**

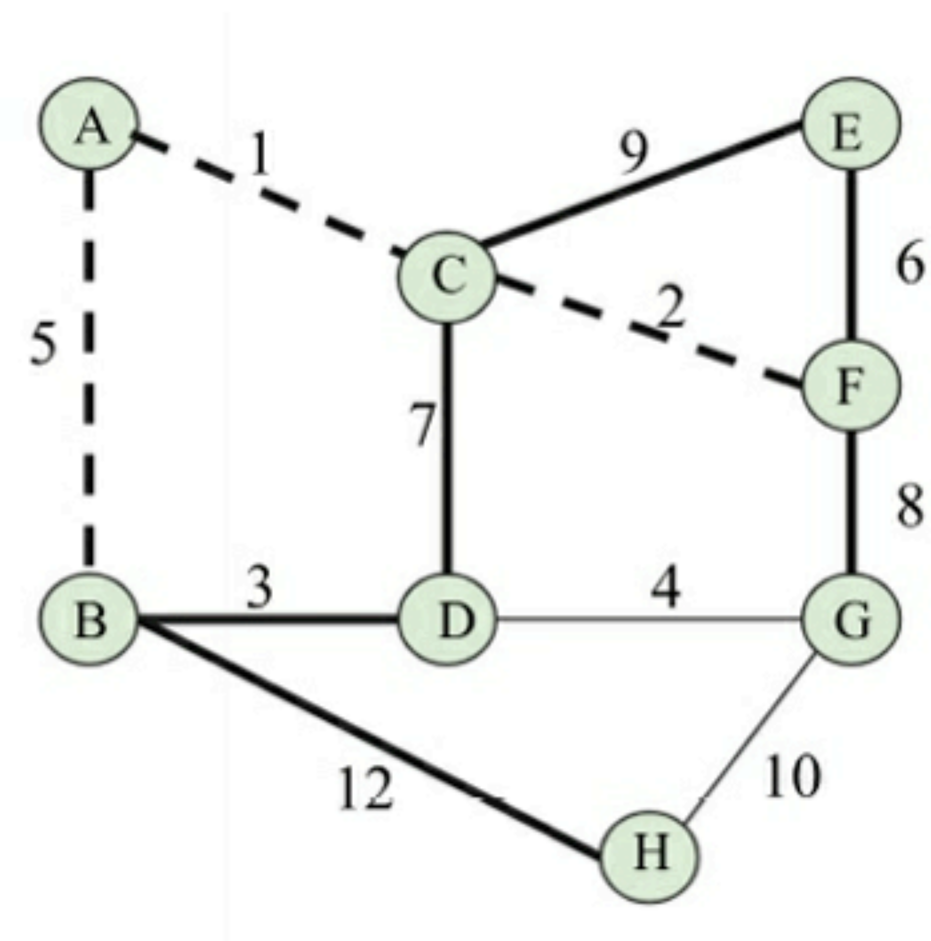


# Prim's minimum spanning tree algorithm

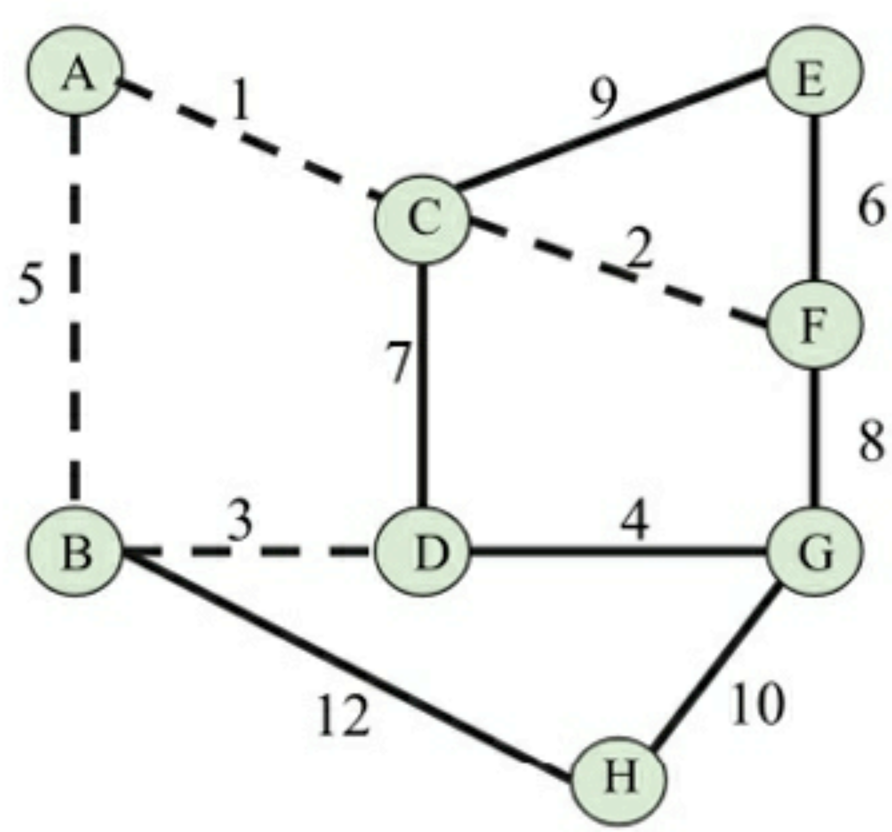
- Add shortest path from edge AC: **CF**
- Add shortest path from tree: **AB**

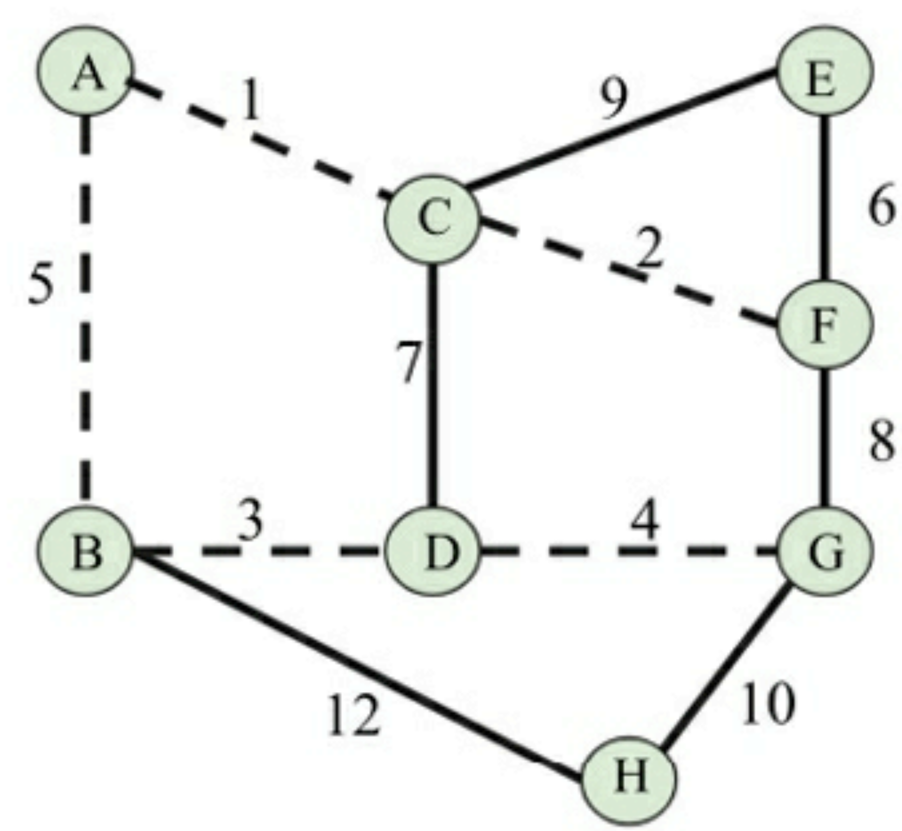


# Prim's minimum spanning tree algorithm

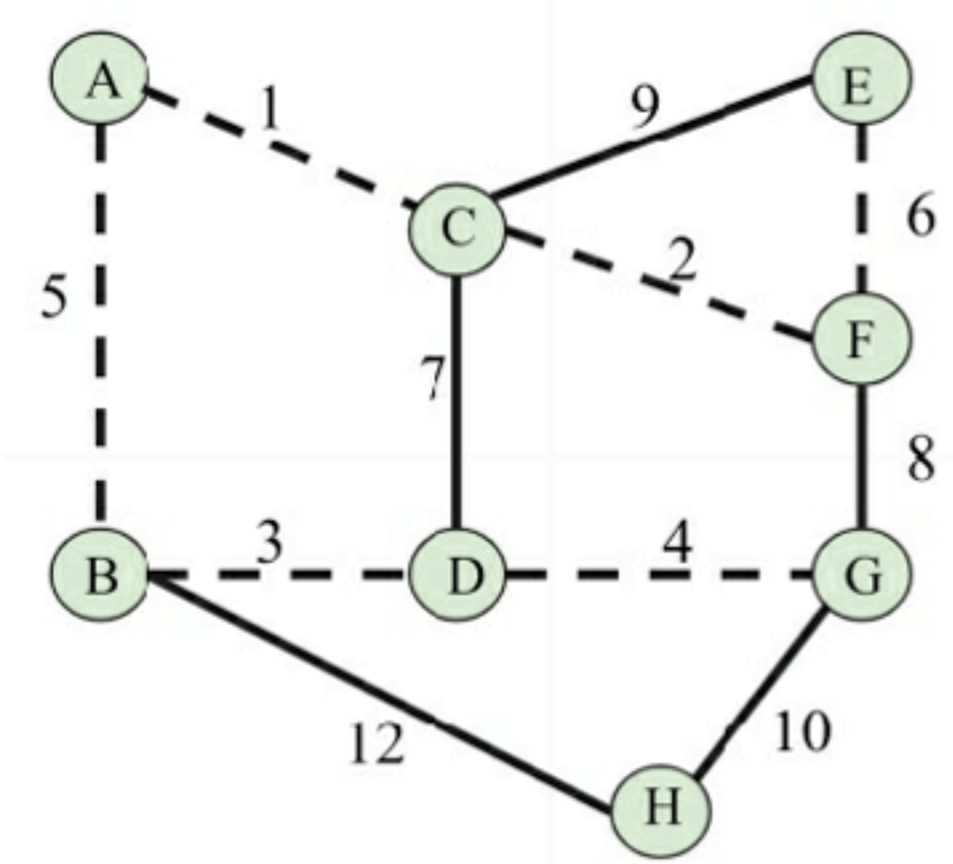
- Add shortest path from tree: **BD**
- Add shortest path from tree: **DG**

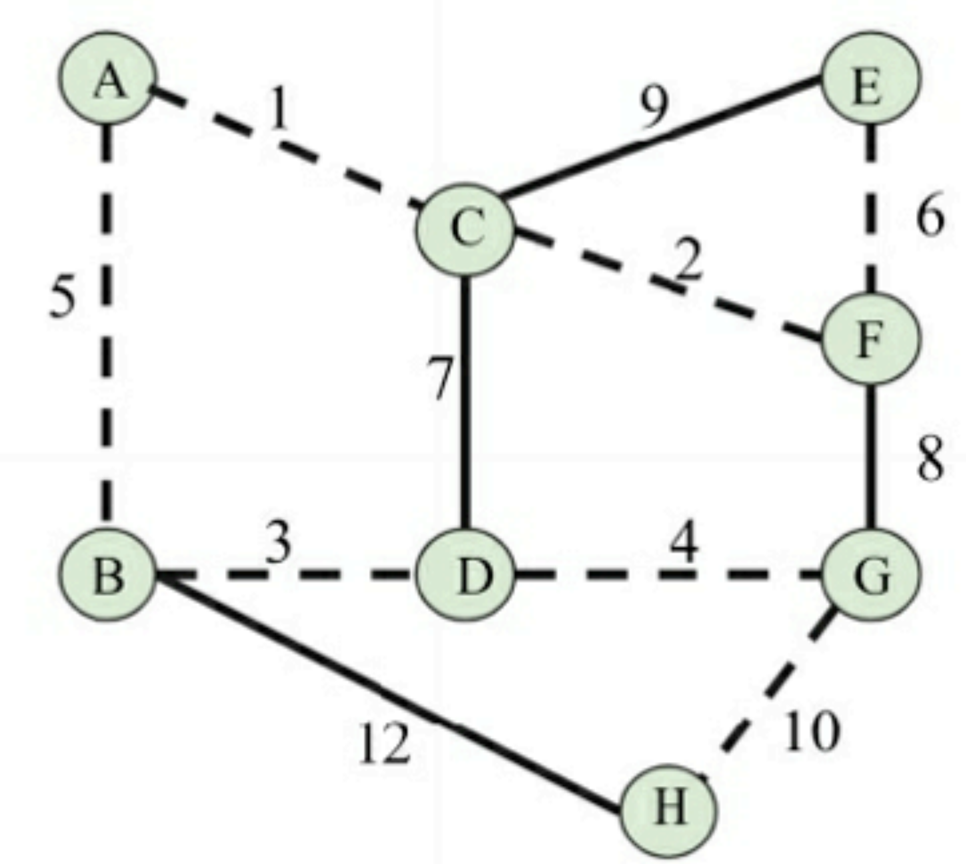


# Prim's minimum spanning tree algorithm

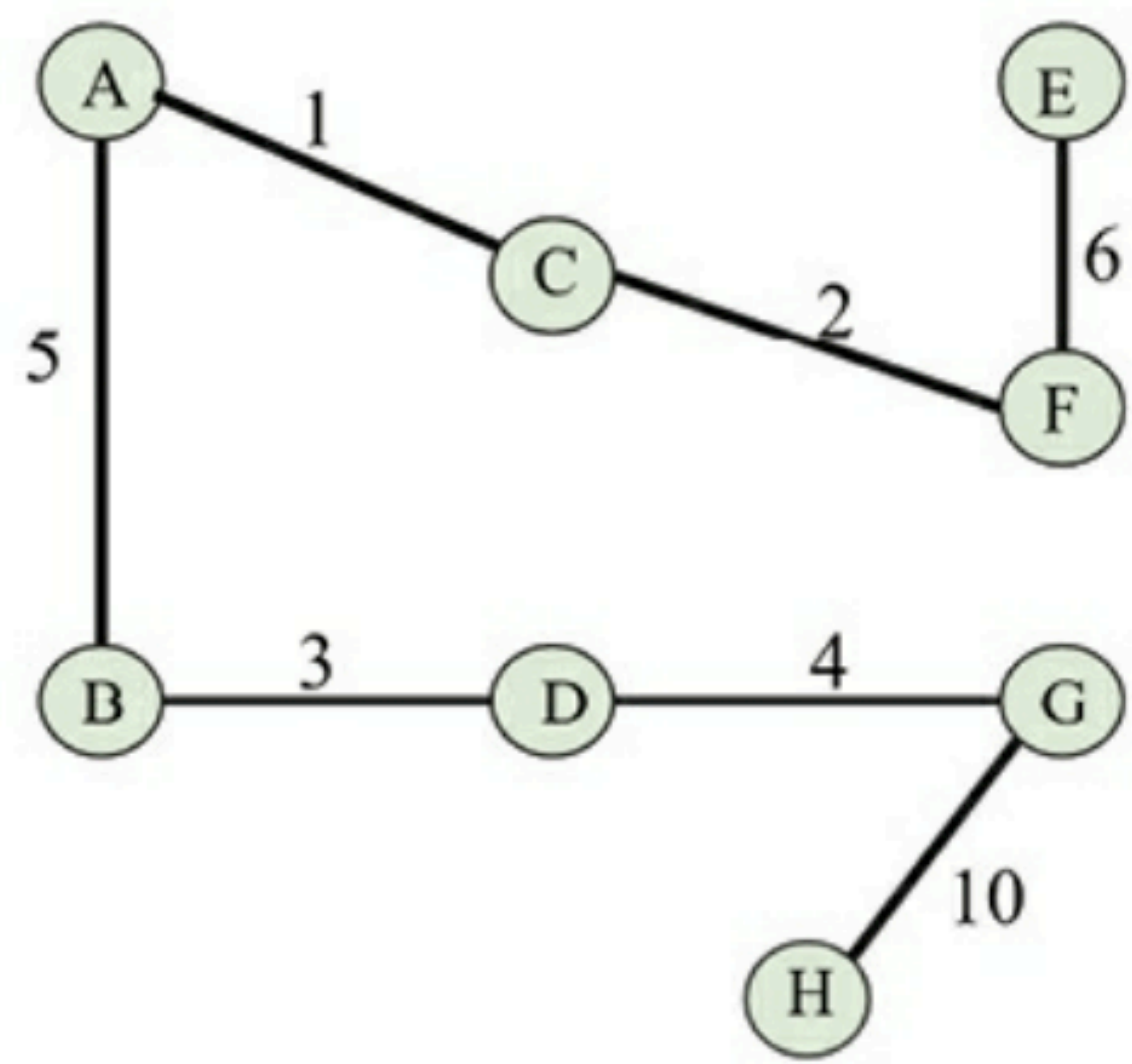
- Add shortest path from tree: **FE**
- Add shortest path from tree: **GH**
  - All remaining paths form cycles



# Prim's minimum spanning tree algorithm

- Final spanning tree



# Comparing algorithms

- Kruskal's: **O(E log V)**
- Prim's: **O(E + V log V)**
- For a dense graph, E > V, so Prim's is better
- For a sparse graph, E is nearly equal to V, so Kruskal's is better

Kahoot!

Ch 9