Chapter 14: Graphs

Data Structures

Basic terms | Degree, paths, cycles | Adjacency lists and matrices | BFS | DFS

Unweighted shortest paths | Dijkstra | Kruskal and Prim

Sercan KÜLCÜ · Giresun University · Computer Engineering

© Sercan Külcü. All rights reserved. · Interactive HTML version of these slides: https://sercankulcu.github.io/files/data_structures/slides/Chapter_14_Graphs.html

INTRODUCTION

Learning Outcomes

Basic terms

Explain vertices, edges, direction, weights, paths, and connectivity.

Representations

Compare adjacency matrices and adjacency lists by their costs.

Traversal and ordering

Run BFS, DFS, and topological sorting on DAGs, and interpret the results.

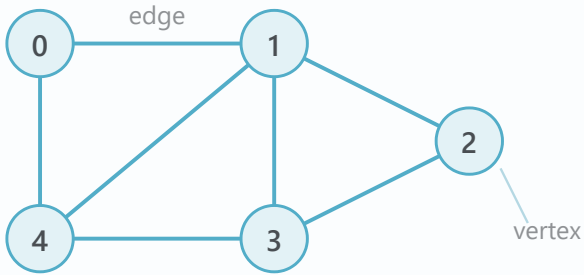
Shortest paths and MSTs

Match unweighted BFS, Dijkstra, Kruskal, and Prim to the problems they solve.

BASIC TERMS

Graphs

- A graph is a set of vertices (or nodes) together with edges that connect pairs of them.
- Graphs model complex relationships, networks, and structures.
- Graph theory studies the relationships between vertices and edges.



A graph is a pair **G = (V, E)**: **V** is the set of vertices, **E** the set of edges.
Example: V = {0, 1, 2, 3, 4}, E = {(0,1), (0,4), (1,2), (1,3), (1,4), (2,3), (3,4)} · n = |V| = 5, m = |E| = 7

Unlike a tree
No root, cycles allowed, and a vertex may be reachable along several paths.

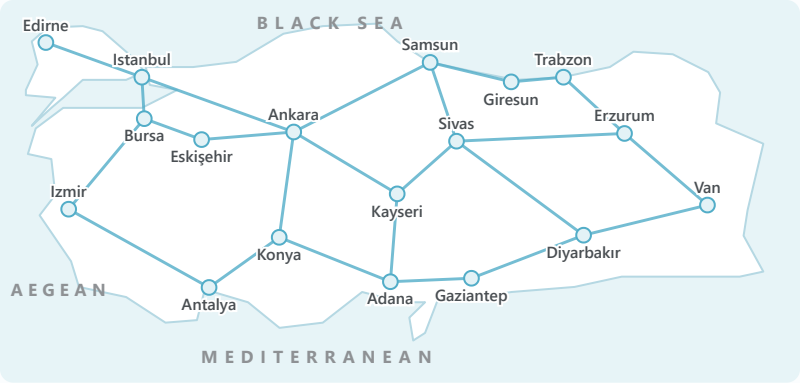
Kind of relation
A graph models many-to-many relationships.

Notation
n = number of vertices, m = number of edges.

BASIC TERMS

Vertices (Nodes)

- The basic building block of a graph.
- Represents an object such as a city, a person, or a computer.
- A vertex can store a value (a label, a name, a location…).
- **Example:** cities and intersections on a map.

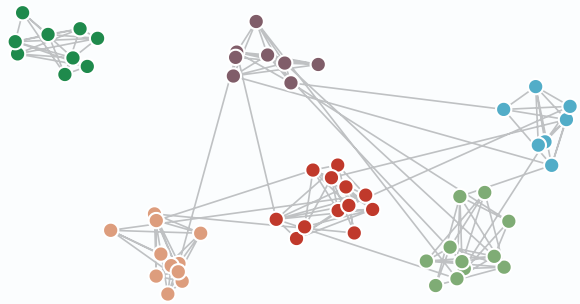


Road map: cities are vertices, roads are edges

Once the map is a graph, "What is the shortest route from Istanbul to Van?" becomes a **graph algorithm** problem (Dijkstra's algorithm, for instance).

Edges

- Edges are the connections between vertices.
- An edge represents a relationship between two vertices.
- If (v, w) is an edge, v and w are **adjacent**: each is a neighbor of the other, and the edge is **incident** to both.
- **Examples:** roads between cities, links in a computer network, friendships in a social network.

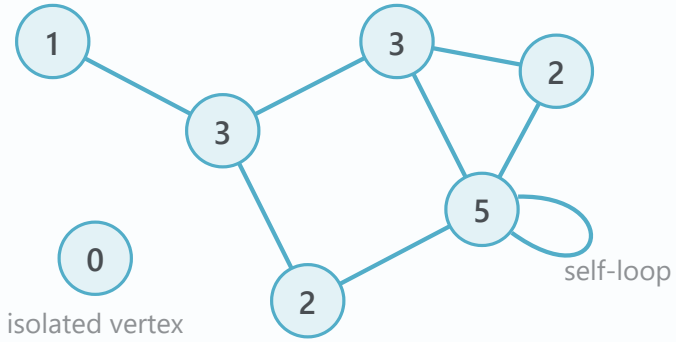


A social network: people are vertices, relationships are edges

BASIC TERMS

Degree

- The degree of a vertex is the number of edges incident to it: `deg(v)`.
- A vertex of degree 0 is called isolated.
- A self-loop adds **2** to the degree.
- **Example:** the number of roads meeting at an intersection.



Each vertex shows its own degree

Handshaking lemma

Every edge adds one to the degree of each end:

$\Sigma \deg(v) = 2m$

Here: 1+3+3+2+5+2+0 = 16 = 2·8 edges.

In a digraph

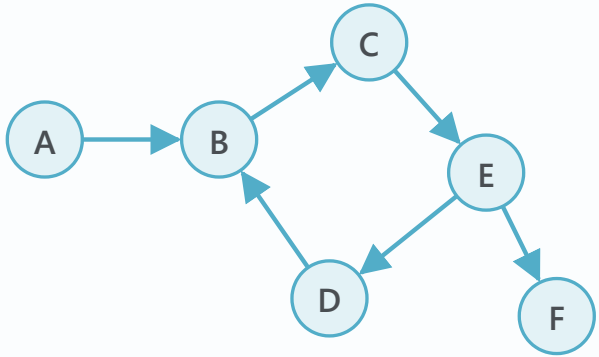
in-degree: number of incoming arrows; *out-degree*: number of outgoing arrows.

$\Sigma \text{ in} = \Sigma \text{ out} = m$

BASIC TERMS

Directed Graphs (Digraphs)

- Each edge points **in one direction**, from one vertex to another: (u, v) ≠ (v, u).
- There may be no way back: here, A cannot be reached from B.
- A directed graph is also called a *digraph*.
- **Examples:** web pages and their links, course prerequisites, one-way streets.

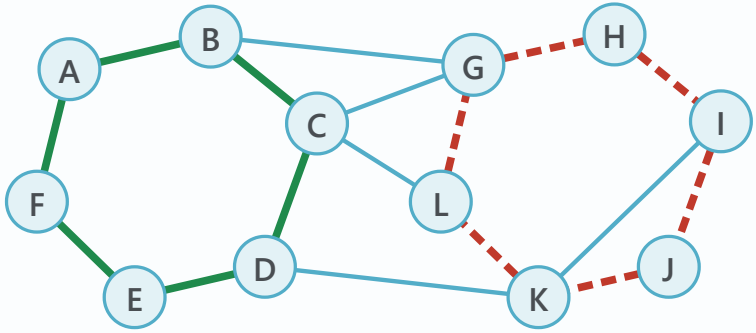


Vertex	A	B	C	D	E	F
in-degree	0	2	1	1	1	1
out-degree	1	1	1	1	2	0

BASIC TERMS

Paths and Cycles

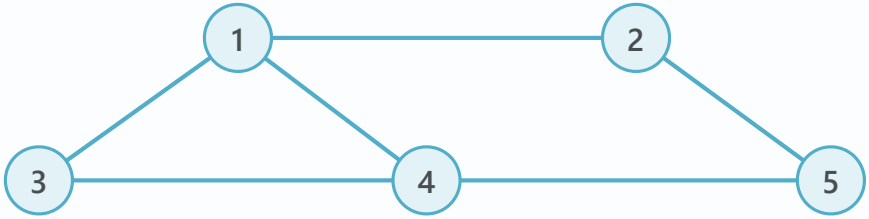
- **Path:** a sequence of vertices in which each consecutive pair is joined by an edge.
- **Simple path:** a path that repeats no vertex.
- **Cycle:** a path that starts and ends at the same vertex.
- **Simple cycle:** repeats no vertex except the start.
- Graphs may have cycles; **trees never do**.



Green: A-B-C-D-E-F-A · Red (dashed): G-H-I-J-K-L-G

Example: a flight route. Istanbul → Ankara → Izmir → Istanbul is a cycle; if it returns without visiting any city twice, it is a simple cycle.

An Example Graph

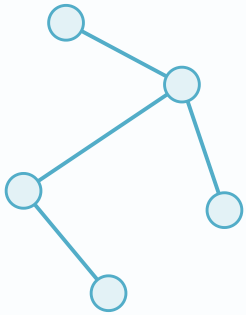


- **V** = {1, 2, 3, 4, 5}
- **E** = {(1,2), (1,3), (1,4), (2,5), (3,4), (4,5)}
- Vertices 1 and 2 are **adjacent**.
- deg(1) = 3, deg(4) = 3, all others 2 → total 12 = 2·6 ✓

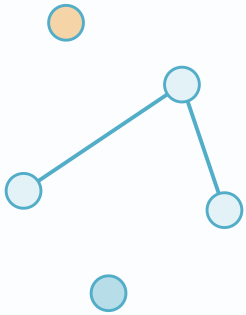
Sequence	Kind
1 → 2 → 5	simple path
1 → 3 → 4 → 1	simple cycle
1 → 3 → 4 → 1 → 2	non-simple path (1 repeats)
1 → 3 → 4 → 1 → 4 → 1	non-simple cycle

Connected and Complete Graphs

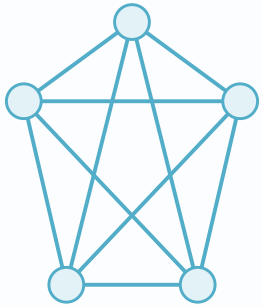
- **Connected graph:** there is a *path* between every pair of distinct vertices.
- **Connected component:** a maximal group of vertices that can reach one another.
- **Complete graph:** there is an *edge* between every pair of distinct vertices.



Connected: 1 component



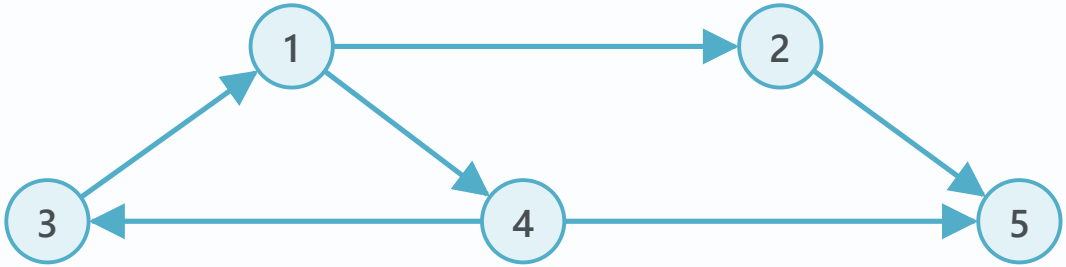
Disconnected: 3 components



Complete graph K_5: 10 edges

A simple undirected graph on n vertices has $m \le n(n-1)/2$, with equality for the complete graph. A connected graph has $m \ge n - 1$.

An Example Digraph



- **V** = {1, 2, 3, 4, 5}
- **E** = {(1,2), (1,4), (2,5), (4,5), (3,1), (4,3)}
- Vertex 2 is **adjacent** to vertex 1 (there is an edge 1 → 2).
- Vertex 1 is **not** adjacent to vertex 2!

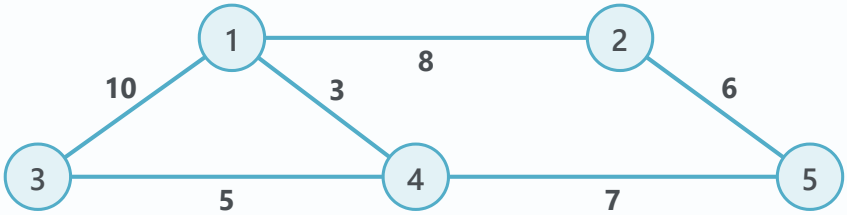
Sequence	Kind
1 → 2 → 5	directed path
1 → 4 → 3 → 1	directed cycle
5 → …	no edges leave 5

A directed edge (u, v) is an ordered pair: u is its tail (origin), v its head (destination).

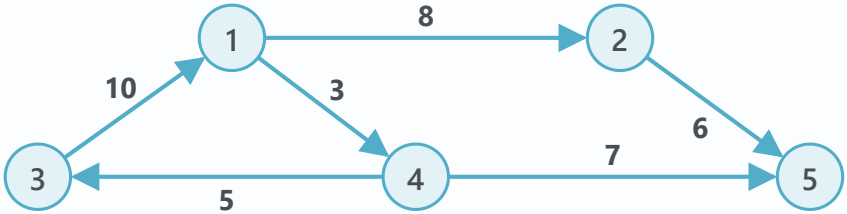
BASIC TERMS

Weighted Graphs

- Each edge carries a number, its **weight**: a distance, time, cost, capacity…
- The length of a path is the sum of its edge weights. Weighted graphs can be undirected or directed.



Weighted undirected: path $3 \to 4 \to 1$ is $5 + 3 = 8$



Weighted directed: path $1 \to 4 \to 3$ is $3 + 5 = 8$

The direct edge is not always the shortest path: in the undirected graph, edge 1–3 weighs 10, but the path $1 \to 4 \to 3$ is only **8**.

BASIC TERMS

The Graph ADT

The core operations of Goodrich's `Graph` interface (V: vertex, E: edge):

Operation	Description
`numVertices()`, `numEdges()`	Number of vertices n, number of edges m
`vertices()`, `edges()`	Iterate over all vertices / all edges
`getEdge(u, v)`	The edge from u to v, or null if there is none
`endVertices(e)`, `opposite(v, e)`	Both ends of an edge / the end of e other than v
`outDegree(v)`, `inDegree(v)`	Out-degree / in-degree (equal in an undirected graph)
`outgoingEdges(v)`, `incomingEdges(v)`	Edges leaving v / entering v
`insertVertex(x)`, `insertEdge(u, v, x)`	Add a new vertex / a new edge
`removeVertex(v)`, `removeEdge(e)`	Remove a vertex (with its incident edges) / an edge

What these operations cost depends on how the graph is stored in memory.

Storing a Graph: Three Representations

Edge list

All edges in a single list.

`[(A,B), (A,E), …]`

Adjacency list

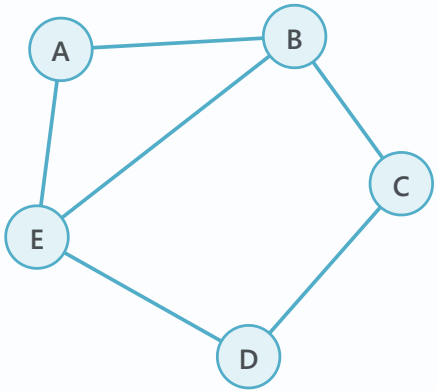
For each vertex, a list of its neighbors.

`V[i] = {j, k, l}`

Adjacency matrix

An n×n two-dimensional array; each cell says whether an edge exists.

`M[i][j] = 1`



The **edge list** is the simplest representation:

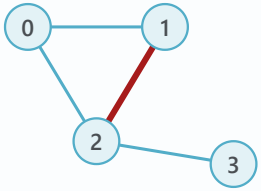
`(A, B)` `(A, E)` `(B, C)` `(B, E)` `(C, D)` `(D, E)`

- Space O(n + m); adding an edge is O(1).
- But "Are u and v adjacent?" or "Who are u's neighbors?" means scanning the whole list: **O(m)**.

GRAPH REPRESENTATIONS

Adjacency Matrix

- An n×n array: `matrix[i][j]` = 1 if there is an edge from vertex i to j, and 0 otherwise.
- **In a weighted graph**, `matrix[i][j]` holds the edge weight, or `∞` (or some special value) if there is no edge.
- **In an undirected graph**, the matrix is symmetric: `matrix[i][j] == matrix[j][i]`.
- Space is **O(n^2)**, so memory grows quickly with the number of vertices.



	0	1	2	3	
0	0	1	1	0	
1	1	0	1	0	← m[1][2]: one cell, O(1)
2	1	1	0	1	← neighbors of 2: whole row, O(n)
3	0	0	1	0	

symmetric: m[i][j] = m[j][i]

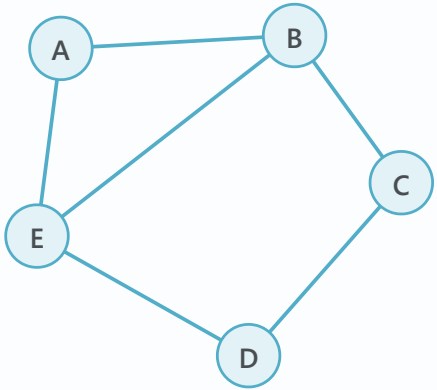
Easy

+ Is there an edge between two vertices? **O(1)**
+ Adding / removing an edge: O(1)

Hard

– Finding a vertex's neighbors: a whole row, **O(n)**
– In a sparse graph, most cells are empty (0)

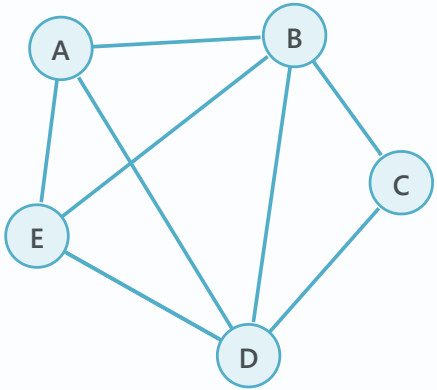
Adjacency Matrix: Three Examples



	A	B	C	D	E
A	0	**1**	0	0	**1**
B	**1**	0	**1**	0	**1**
C	0	**1**	0	**1**	0
D	0	0	**1**	0	**1**
E	**1**	**1**	0	**1**	0

Undirected graph: each edge fills two cells, so the matrix is **symmetric** about the diagonal. Row sum = degree (B: 3).

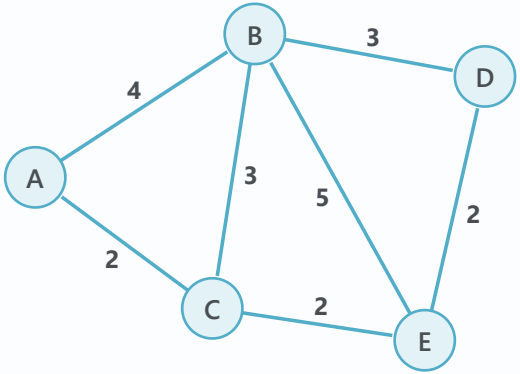
Adjacency Matrix: Three Examples



	A	B	C	D	E
A	0	**1**	0	0	**1**
B	0	0	**1**	**1**	0
C	0	0	0	0	0
D	**1**	0	**1**	0	**1**
E	0	**1**	0	**1**	0

Directed graph: row = origin, column = destination. Not symmetric; a row sum is an out-degree, a column sum an in-degree.

Adjacency Matrix: Three Examples

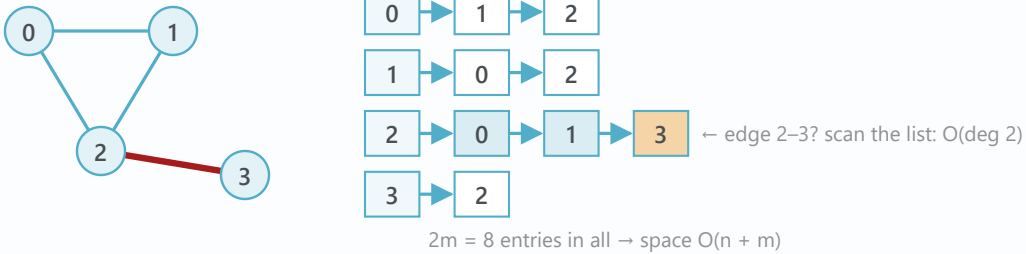


	A	B	C	D	E
A	0	**4**	∞	∞	∞
B	∞	0	**3**	**3**	∞
C	**2**	∞	0	∞	**2**
D	∞	∞	∞	0	∞
E	∞	**5**	∞	**2**	0

Weighted graph: a cell holds the weight, or ∞ if there is no edge (0 on the diagonal). Writing 0 in an empty cell could be confused with an edge of weight 0.

Adjacency List

- Keep n lists for n vertices (usually linked lists or dynamic arrays).
- `list[i]` holds the vertices adjacent to vertex i (with weights, if the graph is weighted).
- **In an undirected graph**, each edge (v, w) goes into both lists: w into v's list, v into w's.
- Space is **O(n + m)**, ideal for sparse graphs.



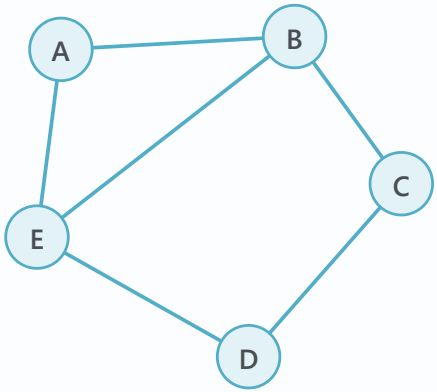
Easy

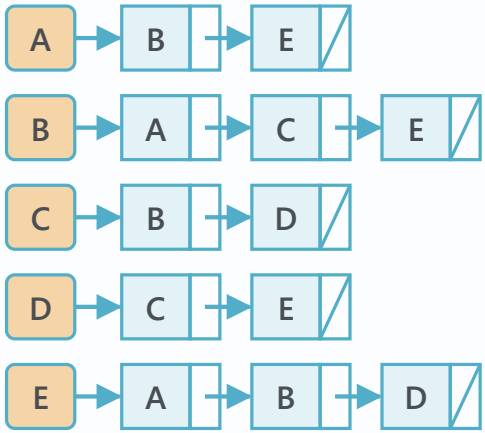
+ Visiting a vertex's neighbors: **O(deg(v))**
+ Adding an edge: O(1)

Hard

– Is there an edge between u and v? Scan the list: **O(deg(u))**

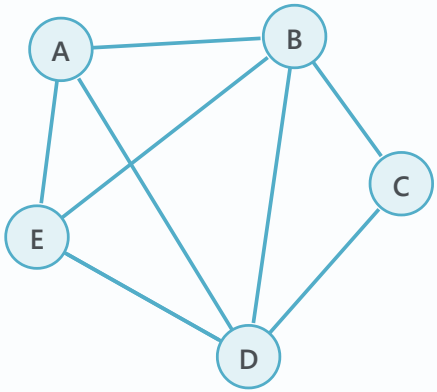
Adjacency List: Three Examples

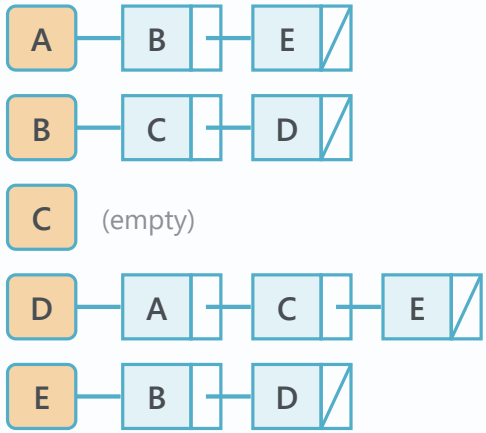




Undirected: 6 edges → 12 list entries (each edge twice). List length = degree.

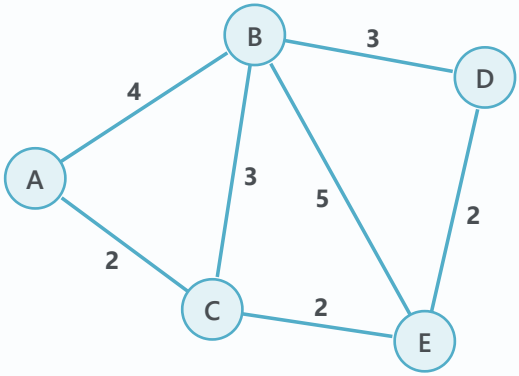
Adjacency List: Three Examples

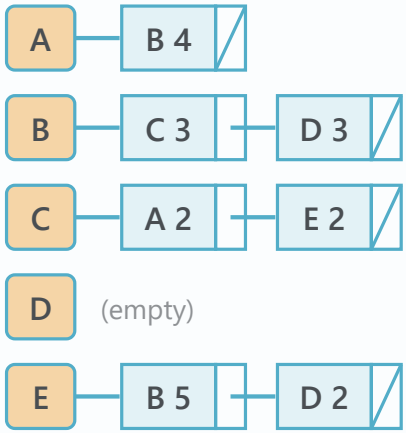




Directed: only outgoing edges are listed; 9 edges → 9 entries. No edge leaves C, so its list is empty.

Adjacency List: Three Examples





Weighted: each entry is a (neighbor, weight) pair; e.g. B's list: (C, 3), (D, 3).

Representations: Memory and Operation Costs

Operation (n vertices, m edges)	Edge list	Adjacency list	Adjacency matrix
Memory	O(n + m)	O(n + m)	$O(n^2)$
Is there an edge u–v? `getEdge`	O(m)	undirected: O(min(deg u, deg v)) directed u→v: O(outDeg u)	**O(1)**
Visit the neighbors of v	O(m)	**O(deg v)**	O(n)

Sparse graph (m ≈ n)

Road networks, social networks → *adjacency list*. With n = 10^6 vertices, a matrix needs 10^{12} cells!

Dense graph (m ≈ n^2)

Frequent "Is there an edge?" queries → *adjacency matrix*.

An Adjacency List in Java

```
class Graph {
    private final int n;                                    // vertices: 0..n-1
    private final List<List<Integer>> adj = new ArrayList<>();
    Graph(int n) {
        this.n = n;
        for (int i = 0; i < n; i++)
            adj.add(new ArrayList<>());
    }
    void addEdge(int u, int v) {
        adj.get(u).add(v);
        adj.get(v).add(u);                  // omit this line in a digraph
    }
    List<Integer> neighbors(int u) { return adj.get(u); }   // O(1)
    int degree(int u) { return adj.get(u).size(); }
}
```

Matrix version

`boolean[][] m = new boolean[n][n];` add an edge: `m[u][v] = m[v][u] = true;` query `m[u][v]` in O(1).

Weighted graph

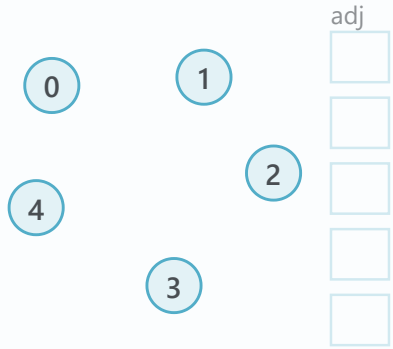
List entries are `int[]{v, w}` or a small `Edge` class; a matrix uses `int[][]` with `Integer.MAX_VALUE` for a missing edge.

Code Trace: Building an Adjacency List

```
1  Graph(int n) {
▶    this.n = n;
3    for (int i = 0; i < n; i++)
4      adj.add(new ArrayList<>());
5  }
6  void addEdge(int u, int v) {
7    adj.get(u).add(v);
8    adj.get(v).add(u);  // not in a digraph
9  }
10 // main:
11 Graph g = new Graph(5);
12 int[][] E = {{0,1}, {0,4}, {1,2},
13              {1,4}, {2,3}, {3,4}};
14 for (int[] e : E) g.addEdge(e[0], e[1]);
```

CALL STACK

- **Graph(5)**
- main()



`new Graph(5)` : the constructor runs and sets **n = 5**. `adj` is still an empty outer list.

Step 1 / 19

VARIABLES

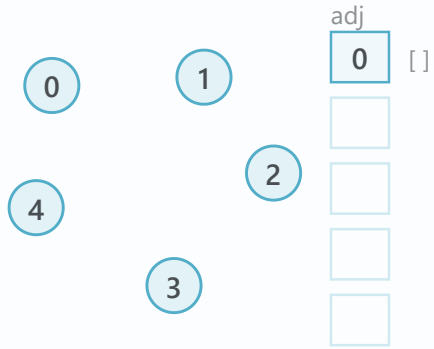
`n = 5`

Code Trace: Building an Adjacency List

```
1  Graph(int n) {
2    this.n = n;
▶    for (int i = 0; i < n; i++)
▶      adj.add(new ArrayList<>());
5  }
6  void addEdge(int u, int v) {
7    adj.get(u).add(v);
8    adj.get(v).add(u);  // not in a digraph
9  }
10 // main:
11 Graph g = new Graph(5);
12 int[][] E = {{0,1}, {0,4}, {1,2},
13              {1,4}, {2,3}, {3,4}};
14 for (int[] e : E) g.addEdge(e[0], e[1]);
```

CALL STACK

- **Graph(5)**
- main()



i = 0: add an **empty inner list** for vertex 0.

Step 2 / 19

VARIABLES

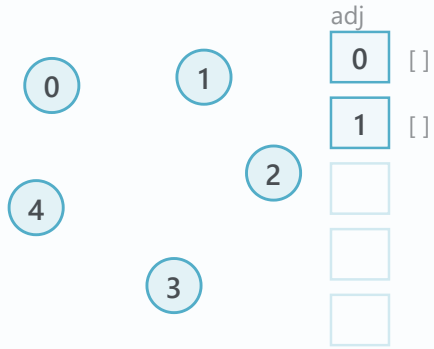
n = 5 i = 0

Code Trace: Building an Adjacency List

```
1  Graph(int n) {
2    this.n = n;
▶   for (int i = 0; i < n; i++)
▶     adj.add(new ArrayList<>());
5  }
6  void addEdge(int u, int v) {
7    adj.get(u).add(v);
8    adj.get(v).add(u);  // not in a digraph
9  }
10 // main:
11 Graph g = new Graph(5);
12 int[][] E = {{0,1}, {0,4}, {1,2},
13              {1,4}, {2,3}, {3,4}};
14 for (int[] e : E) g.addEdge(e[0], e[1]);
```

CALL STACK

- **Graph(5)**
- main()



i = 1: add an **empty inner list** for vertex 1.

Step 3 / 19

VARIABLES

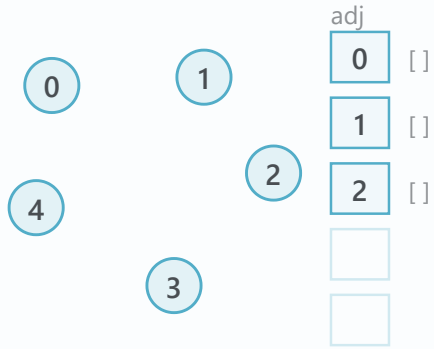
n = 5 i = 1

Code Trace: Building an Adjacency List

```
1  Graph(int n) {
2    this.n = n;
▶   for (int i = 0; i < n; i++)
▶     adj.add(new ArrayList<>());
5  }
6  void addEdge(int u, int v) {
7    adj.get(u).add(v);
8    adj.get(v).add(u);  // not in a digraph
9  }
10 // main:
11 Graph g = new Graph(5);
12 int[][] E = {{0,1}, {0,4}, {1,2},
13              {1,4}, {2,3}, {3,4}};
14 for (int[] e : E) g.addEdge(e[0], e[1]);
```

CALL STACK

- **Graph(5)**
- main()



i = 2: add an **empty inner list** for vertex 2.

Step 4 / 19

VARIABLES

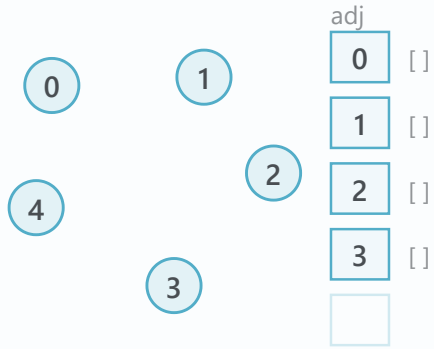
n = 5 i = 2

Code Trace: Building an Adjacency List

```
1  Graph(int n) {
2    this.n = n;
▶    for (int i = 0; i < n; i++)
▶      adj.add(new ArrayList<>());
5  }
6  void addEdge(int u, int v) {
7    adj.get(u).add(v);
8    adj.get(v).add(u);  // not in a digraph
9  }
10 // main:
11 Graph g = new Graph(5);
12 int[][] E = {{0,1}, {0,4}, {1,2},
13              {1,4}, {2,3}, {3,4}};
14 for (int[] e : E) g.addEdge(e[0], e[1]);
```

CALL STACK

- **Graph(5)**
- main()



i = 3: add an **empty inner list** for vertex 3.

Step 5 / 19

VARIABLES

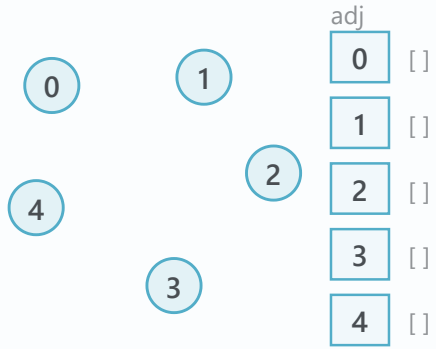
n = 5 i = 3

Code Trace: Building an Adjacency List

```
1  Graph(int n) {
2    this.n = n;
▶   for (int i = 0; i < n; i++)
▶     adj.add(new ArrayList<>());
5  }
6  void addEdge(int u, int v) {
7    adj.get(u).add(v);
8    adj.get(v).add(u);  // not in a digraph
9  }
10 // main:
11 Graph g = new Graph(5);
12 int[][] E = {{0,1}, {0,4}, {1,2},
13              {1,4}, {2,3}, {3,4}};
14 for (int[] e : E) g.addEdge(e[0], e[1]);
```

CALL STACK

- **Graph(5)**
- main()



i = 4: add an **empty inner list** for vertex 4. Every vertex now has an (empty) neighbor list.

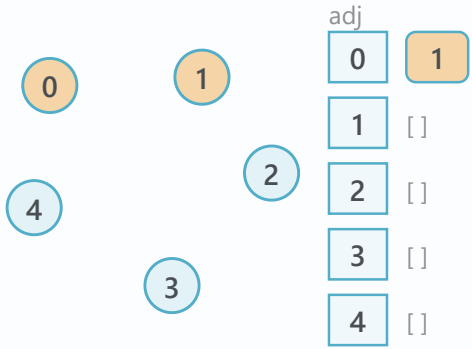
Step 6 / 19

VARIABLES

`n = 5` `i = 4`

Code Trace: Building an Adjacency List

```
1  Graph(int n) {
2    this.n = n;
3    for (int i = 0; i < n; i++)
4      adj.add(new ArrayList<>());
5  }
6  void addEdge(int u, int v) {
▶    adj.get(u).add(v);
8    adj.get(v).add(u);  // not in a digraph
9  }
10 // main:
11 Graph g = new Graph(5);
12 int[][] E = {{0,1}, {0,4}, {1,2},
13              {1,4}, {2,3}, {3,4}};
14 for (int[] e : E) g.addEdge(e[0], e[1]);
```



Edge 0–1: **1** is appended to 0's list.

Step 7 / 19

CALL STACK

- `addEdge(0, 1)`
- `main()`

VARIABLES

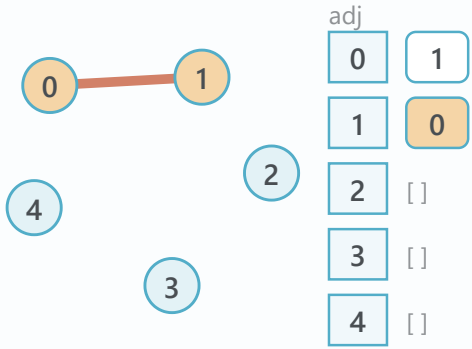
`u = 0` `v = 1`

Code Trace: Building an Adjacency List

```
1  Graph(int n) {
2    this.n = n;
3    for (int i = 0; i < n; i++)
4      adj.add(new ArrayList<>());
5  }
6  void addEdge(int u, int v) {
7    adj.get(u).add(v);
▶    adj.get(v).add(u);  // not in a digraph
9  }
10 // main:
11 Graph g = new Graph(5);
12 int[][] E = {{0,1}, {0,4}, {1,2},
13              {1,4}, {2,3}, {3,4}};
14 for (int[] e : E) g.addEdge(e[0], e[1]);
```

CALL STACK

- **addEdge(0, 1)**
- main()



Undirected: the same edge is added in reverse → **0** goes into 1's list.

Step 8 / 19

VARIABLES

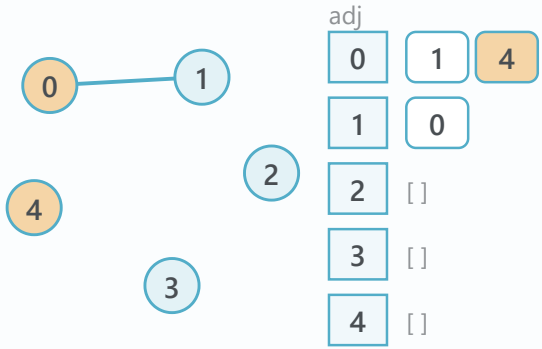
`u = 0` `v = 1`

Code Trace: Building an Adjacency List

```
1  Graph(int n) {
2    this.n = n;
3    for (int i = 0; i < n; i++)
4      adj.add(new ArrayList<>());
5  }
6  void addEdge(int u, int v) {
▶    adj.get(u).add(v);
8    adj.get(v).add(u);  // not in a digraph
9  }
10 // main:
11 Graph g = new Graph(5);
12 int[][] E = {{0,1}, {0,4}, {1,2},
13              {1,4}, {2,3}, {3,4}};
14 for (int[] e : E) g.addEdge(e[0], e[1]);
```

CALL STACK

- **addEdge(0, 4)**
- main()



Edge 0–4: **4** is appended to 0's list.

Step 9 / 19

VARIABLES

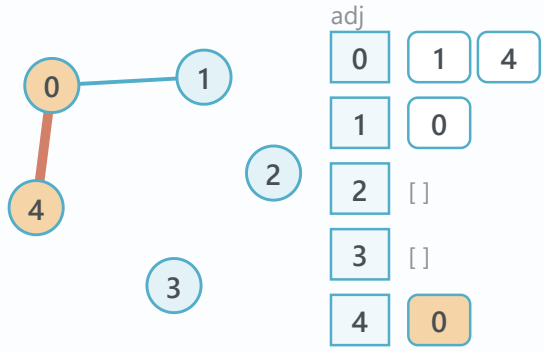
u = 0 v = 4

Code Trace: Building an Adjacency List

```
1  Graph(int n) {
2    this.n = n;
3    for (int i = 0; i < n; i++)
4      adj.add(new ArrayList<>());
5  }
6  void addEdge(int u, int v) {
7    adj.get(u).add(v);
▶    adj.get(v).add(u);  // not in a digraph
9  }
10 // main:
11 Graph g = new Graph(5);
12 int[][] E = {{0,1}, {0,4}, {1,2},
13              {1,4}, {2,3}, {3,4}};
14 for (int[] e : E) g.addEdge(e[0], e[1]);
```

CALL STACK

- **addEdge(0, 4)**
- main()



Undirected: the same edge is added in reverse → **0** goes into 4's list.

Step 10 / 19

VARIABLES

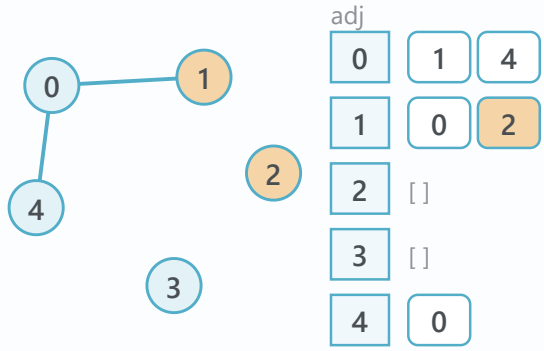
`u = 0` `v = 4`

Code Trace: Building an Adjacency List

```
1  Graph(int n) {
2    this.n = n;
3    for (int i = 0; i < n; i++)
4      adj.add(new ArrayList<>());
5  }
6  void addEdge(int u, int v) {
▶    adj.get(u).add(v);
8    adj.get(v).add(u);  // not in a digraph
9  }
10 // main:
11 Graph g = new Graph(5);
12 int[][] E = {{0,1}, {0,4}, {1,2},
13              {1,4}, {2,3}, {3,4}};
14 for (int[] e : E) g.addEdge(e[0], e[1]);
```

CALL STACK

- **addEdge(1, 2)**
- main()



Edge 1–2: **2** is appended to 1's list.

Step 11 / 19

VARIABLES

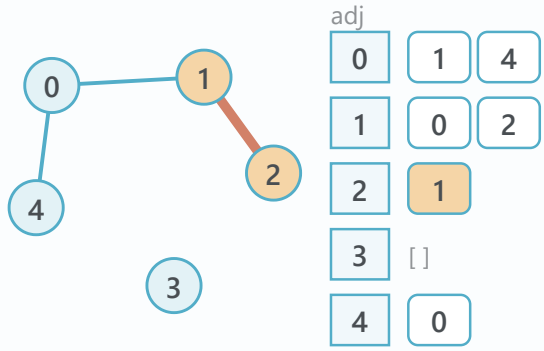
u = 1 v = 2

Code Trace: Building an Adjacency List

```
1  Graph(int n) {
2    this.n = n;
3    for (int i = 0; i < n; i++)
4      adj.add(new ArrayList<>());
5  }
6  void addEdge(int u, int v) {
7    adj.get(u).add(v);
▶    adj.get(v).add(u);  // not in a digraph
9  }
10 // main:
11 Graph g = new Graph(5);
12 int[][] E = {{0,1}, {0,4}, {1,2},
13              {1,4}, {2,3}, {3,4}};
14 for (int[] e : E) g.addEdge(e[0], e[1]);
```

CALL STACK

- **addEdge(1, 2)**
- main()



Undirected: the same edge is added in reverse → **1** goes into 2's list.

Step 12 / 19

VARIABLES

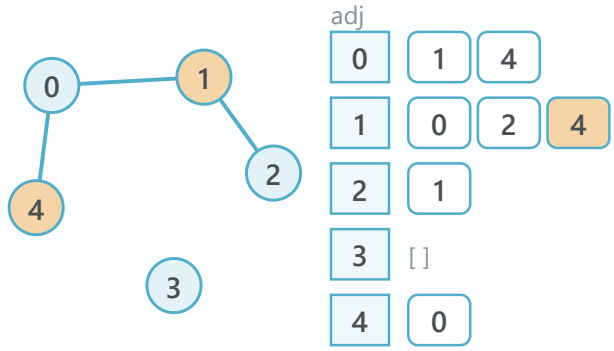
u = **1** v = **2**

Code Trace: Building an Adjacency List

```
1  Graph(int n) {
2    this.n = n;
3    for (int i = 0; i < n; i++)
4      adj.add(new ArrayList<>());
5  }
6  void addEdge(int u, int v) {
▶    adj.get(u).add(v);
8    adj.get(v).add(u);  // not in a digraph
9  }
10 // main:
11 Graph g = new Graph(5);
12 int[][] E = {{0,1}, {0,4}, {1,2},
13              {1,4}, {2,3}, {3,4}};
14 for (int[] e : E) g.addEdge(e[0], e[1]);
```

CALL STACK

- **addEdge(1, 4)**
- main()



Edge 1–4: **4** is appended to 1's list.

Step 13 / 19

VARIABLES

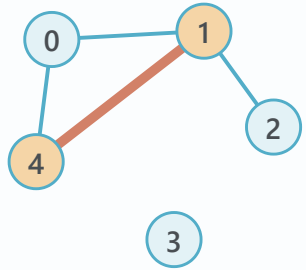
u = **1** v = **4**

Code Trace: Building an Adjacency List

```
1  Graph(int n) {
2    this.n = n;
3    for (int i = 0; i < n; i++)
4      adj.add(new ArrayList<>());
5  }
6  void addEdge(int u, int v) {
7    adj.get(u).add(v);
▶    adj.get(v).add(u);  // not in a digraph
9  }
10 // main:
11 Graph g = new Graph(5);
12 int[][] E = {{0,1}, {0,4}, {1,2},
13              {1,4}, {2,3}, {3,4}};
14 for (int[] e : E) g.addEdge(e[0], e[1]);
```

CALL STACK

- **addEdge(1, 4)**
- main()



adj

0	1	4	
1	0	2	4
2	1		
3	[]		
4	0	1	

Undirected: the same edge is added in reverse → **1** goes into 4's list.

Step 14 / 19

VARIABLES

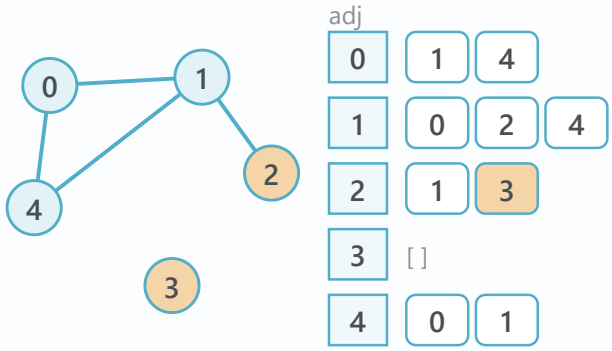
`u = 1` `v = 4`

Code Trace: Building an Adjacency List

```
1  Graph(int n) {
2    this.n = n;
3    for (int i = 0; i < n; i++)
4      adj.add(new ArrayList<>());
5  }
6  void addEdge(int u, int v) {
▶    adj.get(u).add(v);
8    adj.get(v).add(u);  // not in a digraph
9  }
10 // main:
11 Graph g = new Graph(5);
12 int[][] E = {{0,1}, {0,4}, {1,2},
13              {1,4}, {2,3}, {3,4}};
14 for (int[] e : E) g.addEdge(e[0], e[1]);
```

CALL STACK

- **addEdge(2, 3)**
- main()



Edge 2–3: **3** is appended to 2's list.

Step 15 / 19

VARIABLES

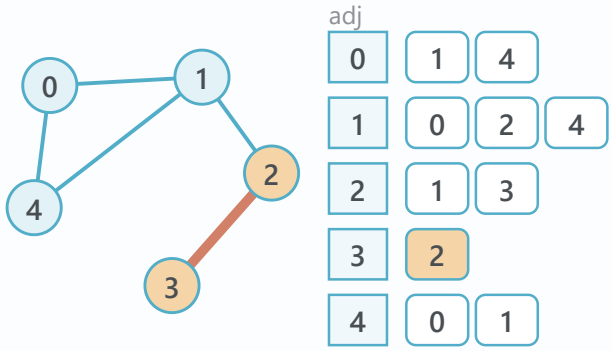
u = 2 v = 3

Code Trace: Building an Adjacency List

```
1  Graph(int n) {
2    this.n = n;
3    for (int i = 0; i < n; i++)
4      adj.add(new ArrayList<>());
5  }
6  void addEdge(int u, int v) {
7    adj.get(u).add(v);
▶    adj.get(v).add(u);  // not in a digraph
9  }
10 // main:
11 Graph g = new Graph(5);
12 int[][] E = {{0,1}, {0,4}, {1,2},
13              {1,4}, {2,3}, {3,4}};
14 for (int[] e : E) g.addEdge(e[0], e[1]);
```

CALL STACK

- **addEdge(2, 3)**
- main()



Undirected: the same edge is added in reverse → **2** goes into 3's list.

Step 16 / 19

VARIABLES

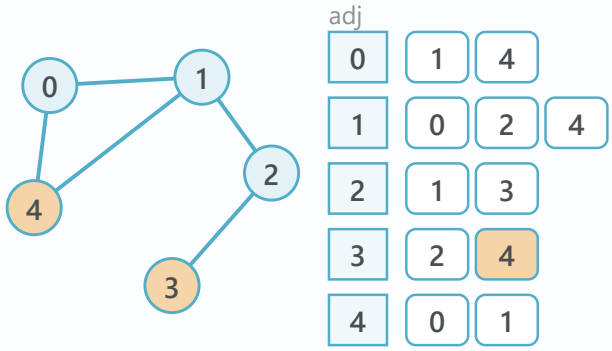
`u = 2` `v = 3`

Code Trace: Building an Adjacency List

```
1  Graph(int n) {
2    this.n = n;
3    for (int i = 0; i < n; i++)
4      adj.add(new ArrayList<>());
5  }
6  void addEdge(int u, int v) {
▶    adj.get(u).add(v);
8    adj.get(v).add(u);  // not in a digraph
9  }
10 // main:
11 Graph g = new Graph(5);
12 int[][] E = {{0,1}, {0,4}, {1,2},
13              {1,4}, {2,3}, {3,4}};
14 for (int[] e : E) g.addEdge(e[0], e[1]);
```

CALL STACK

- **addEdge(3, 4)**
- main()



Edge 3–4: **4** is appended to 3's list.

Step 17 / 19

VARIABLES

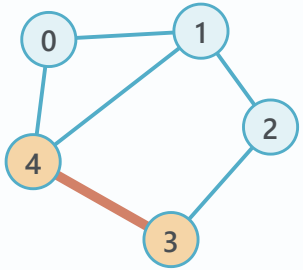
u = 3 v = 4

Code Trace: Building an Adjacency List

```
1  Graph(int n) {
2    this.n = n;
3    for (int i = 0; i < n; i++)
4      adj.add(new ArrayList<>());
5  }
6  void addEdge(int u, int v) {
7    adj.get(u).add(v);
▶    adj.get(v).add(u);  // not in a digraph
9  }
10 // main:
11 Graph g = new Graph(5);
12 int[][] E = {{0,1}, {0,4}, {1,2},
13              {1,4}, {2,3}, {3,4}};
14 for (int[] e : E) g.addEdge(e[0], e[1]);
```

CALL STACK

- **addEdge(3, 4)**
- main()



adj

0	1	4	
1	0	2	4
2	1	3	
3	2	4	
4	0	1	**3**

Undirected: the same edge is added in reverse → **3** goes into 4's list.

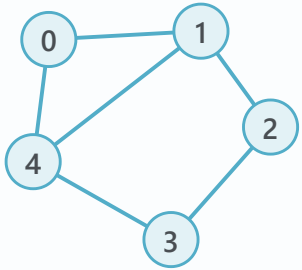
Step 18 / 19

VARIABLES

`u = 3` `v = 4`

Code Trace: Building an Adjacency List

```
1  Graph(int n) {
2    this.n = n;
3    for (int i = 0; i < n; i++)
4      adj.add(new ArrayList<>());
5  }
6  void addEdge(int u, int v) {
7    adj.get(u).add(v);
8    adj.get(v).add(u);  // not in a digraph
9  }
10 // main:
11 Graph g = new Graph(5);
12 int[][] E = {{0,1}, {0,4}, {1,2},
13              {1,4}, {2,3}, {3,4}};
▶  for (int[] e : E) g.addEdge(e[0], e[1]);
```



adj

0	1	4	
1	0	2	4
2	1	3	
3	2	4	
4	0	1	3

Done: 6 edges → **2·6 = 12** list entries. Each `addEdge` is O(1); memory O(n + m). degree(1) = list length = 3.

Step 19 / 19

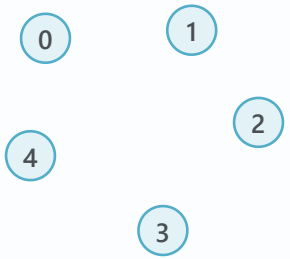
CALL STACK

`main()`

VARIABLES

—

Code Trace: Building an Adjacency Matrix



	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0	0	0
2	0	0	0	0	0
3	0	0	0	0	0
4	0	0	0	0	0

m[row][col] (1 = true)

A 5×5 **boolean** matrix is created; Java initializes every cell to **false** (0). Memory $O(n^2)$ = 25 cells.

Step 1 / 14

```
▶ boolean[][] m = new boolean[n][n];
2 void addEdge(int u, int v) {
3   m[u][v] = true;
4   m[v][u] = true;  // not in a digraph
5 }
6 // main: same 6 edges (n = 5)
7 for (int[] e : E) g.addEdge(e[0], e[1]);
```

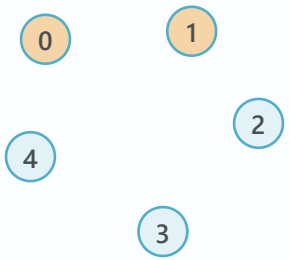
VARIABLES

`n = 5`

CALL STACK

`main()`

Code Trace: Building an Adjacency Matrix



	0	1	2	3	4
0	0	**1**	0	0	0
1	0	0	0	0	0
2	0	0	0	0	0
3	0	0	0	0	0
4	0	0	0	0	0

m[row][col] (1 = true)

Edge 0–1: **m[0][1] = true** (row 0, column 1).

Step 2 / 14

```
1 boolean[][] m = new boolean[n][n];
2 void addEdge(int u, int v) {
▶   m[u][v] = true;
4   m[v][u] = true;  // not in a digraph
5 }
6 // main: same 6 edges (n = 5)
7 for (int[] e : E) g.addEdge(e[0], e[1]);
```

VARIABLES

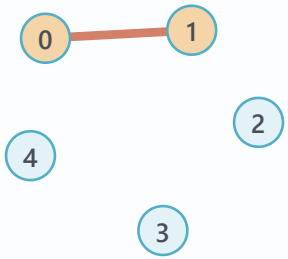
`u = 0` `v = 1`

CALL STACK

`addEdge(0, 1)`

`main()`

Code Trace: Building an Adjacency Matrix



	0	1	2	3	4
0	0	1	0	0	0
1	1	0	0	0	0
2	0	0	0	0	0
3	0	0	0	0	0
4	0	0	0	0	0

m[row][col] (1 = true)

Mirror cell: **m[1][0] = true**. An undirected graph's matrix is symmetric about the diagonal.

Step 3 / 14

```
1 boolean[][] m = new boolean[n][n];
2 void addEdge(int u, int v) {
3   m[u][v] = true;
▶   m[v][u] = true;  // not in a digraph
5 }
6 // main: same 6 edges (n = 5)
7 for (int[] e : E) g.addEdge(e[0], e[1]);
```

VARIABLES

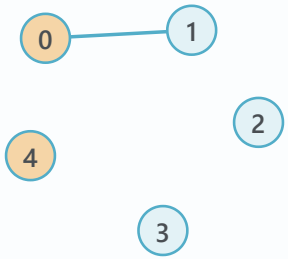
`u = 0` `v = 1`

CALL STACK

`addEdge(0, 1)`

`main()`

Code Trace: Building an Adjacency Matrix



	0	1	2	3	4
0	0	**1**	0	0	**1**
1	**1**	0	0	0	0
2	0	0	0	0	0
3	0	0	0	0	0
4	0	0	0	0	0

m[row][col] (1 = true)

Edge 0–4: **m[0][4] = true** (row 0, column 4).

Step 4 / 14

```
1 boolean[][] m = new boolean[n][n];
2 void addEdge(int u, int v) {
▶   m[u][v] = true;
4   m[v][u] = true;  // not in a digraph
5 }
6 // main: same 6 edges (n = 5)
7 for (int[] e : E) g.addEdge(e[0], e[1]);
```

VARIABLES

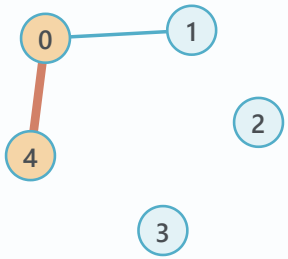
`u = 0` `v = 4`

CALL STACK

`addEdge(0, 4)`

`main()`

Code Trace: Building an Adjacency Matrix



	0	1	2	3	4
0	0	1	0	0	1
1	1	0	0	0	0
2	0	0	0	0	0
3	0	0	0	0	0
4	1	0	0	0	0

m[row][col] (1 = true)

Mirror cell: **m[4][0] = true**. An undirected graph's matrix is symmetric about the diagonal.

Step 5 / 14

```
1 boolean[][] m = new boolean[n][n];
2 void addEdge(int u, int v) {
3   m[u][v] = true;
▶   m[v][u] = true;  // not in a digraph
5 }
6 // main: same 6 edges (n = 5)
7 for (int[] e : E) g.addEdge(e[0], e[1]);
```

VARIABLES

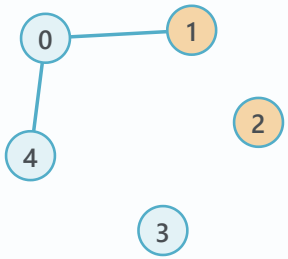
`u = 0` `v = 4`

CALL STACK

`addEdge(0, 4)`

`main()`

Code Trace: Building an Adjacency Matrix



	0	1	2	3	4
0	0	**1**	0	0	**1**
1	**1**	0	**1**	0	0
2	0	0	0	0	0
3	0	0	0	0	0
4	**1**	0	0	0	0

m[row][col] (1 = true)

Edge 1–2: **m[1][2] = true** (row 1, column 2).

Step 6 / 14

```
1 boolean[][] m = new boolean[n][n];
2 void addEdge(int u, int v) {
▶   m[u][v] = true;
4   m[v][u] = true;  // not in a digraph
5 }
6 // main: same 6 edges (n = 5)
7 for (int[] e : E) g.addEdge(e[0], e[1]);
```

VARIABLES

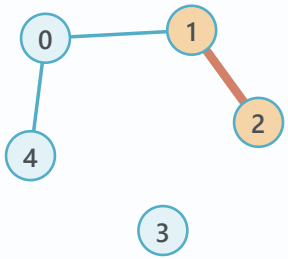
`u = 1` `v = 2`

CALL STACK

`addEdge(1, 2)`

`main()`

Code Trace: Building an Adjacency Matrix



	0	1	2	3	4
0	0	1	0	0	1
1	1	0	1	0	0
2	0	1	0	0	0
3	0	0	0	0	0
4	1	0	0	0	0

m[row][col] (1 = true)

Mirror cell: **m[2][1] = true**. An undirected graph's matrix is symmetric about the diagonal.

Step 7 / 14

```
1 boolean[][] m = new boolean[n][n];
2 void addEdge(int u, int v) {
3   m[u][v] = true;
▶   m[v][u] = true;  // not in a digraph
5 }
6 // main: same 6 edges (n = 5)
7 for (int[] e : E) g.addEdge(e[0], e[1]);
```

VARIABLES

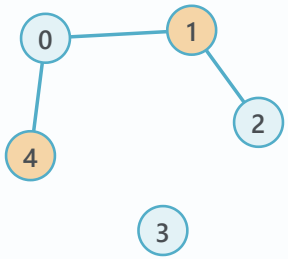
`u = 1` `v = 2`

CALL STACK

`addEdge(1, 2)`

`main()`

Code Trace: Building an Adjacency Matrix



	0	1	2	3	4
0	0	1	0	0	1
1	1	0	1	0	1
2	0	1	0	0	0
3	0	0	0	0	0
4	1	0	0	0	0

m[row][col] (1 = true)

Edge 1–4: **m[1][4] = true** (row 1, column 4).

Step 8 / 14

```
1 boolean[][] m = new boolean[n][n];
2 void addEdge(int u, int v) {
▶   m[u][v] = true;
4   m[v][u] = true;  // not in a digraph
5 }
6 // main: same 6 edges (n = 5)
7 for (int[] e : E) g.addEdge(e[0], e[1]);
```

VARIABLES

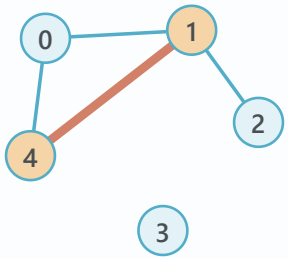
`u = 1` `v = 4`

CALL STACK

`addEdge(1, 4)`

`main()`

Code Trace: Building an Adjacency Matrix



	0	1	2	3	4
0	0	1	0	0	1
1	1	0	1	0	1
2	0	1	0	0	0
3	0	0	0	0	0
4	1	1	0	0	0

m[row][col] (1 = true)

Mirror cell: **m[4][1] = true**. An undirected graph's matrix is symmetric about the diagonal.

Step 9 / 14

```
1 boolean[][] m = new boolean[n][n];
2 void addEdge(int u, int v) {
3   m[u][v] = true;
▶   m[v][u] = true;  // not in a digraph
5 }
6 // main: same 6 edges (n = 5)
7 for (int[] e : E) g.addEdge(e[0], e[1]);
```

VARIABLES

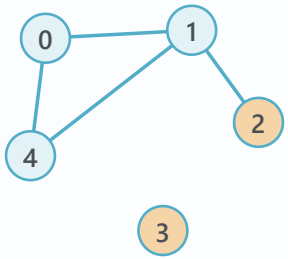
`u = 1` `v = 4`

CALL STACK

`addEdge(1, 4)`

`main()`

Code Trace: Building an Adjacency Matrix



	0	1	2	3	4
0	0	1	0	0	1
1	1	0	1	0	1
2	0	1	0	1	0
3	0	0	0	0	0
4	1	1	0	0	0

m[row][col] (1 = true)

Edge 2–3: **m[2][3] = true** (row 2, column 3).

Step 10 / 14

```
1 boolean[][] m = new boolean[n][n];
2 void addEdge(int u, int v) {
▶   m[u][v] = true;
4   m[v][u] = true;  // not in a digraph
5 }
6 // main: same 6 edges (n = 5)
7 for (int[] e : E) g.addEdge(e[0], e[1]);
```

VARIABLES

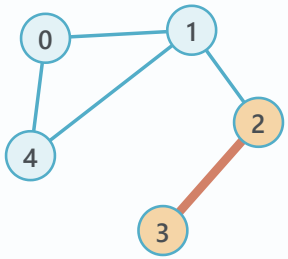
`u = 2` `v = 3`

CALL STACK

`addEdge(2, 3)`

`main()`

Code Trace: Building an Adjacency Matrix



	0	1	2	3	4
0	0	1	0	0	1
1	1	0	1	0	1
2	0	1	0	1	0
3	0	0	1	0	0
4	1	1	0	0	0

m[row][col] (1 = true)

Mirror cell: **m[3][2] = true**. An undirected graph's matrix is symmetric about the diagonal.

Step 11 / 14

```
1 boolean[][] m = new boolean[n][n];
2 void addEdge(int u, int v) {
3   m[u][v] = true;
▶   m[v][u] = true;  // not in a digraph
5 }
6 // main: same 6 edges (n = 5)
7 for (int[] e : E) g.addEdge(e[0], e[1]);
```

VARIABLES

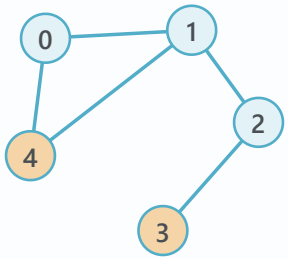
`u = 2` `v = 3`

CALL STACK

`addEdge(2, 3)`

`main()`

Code Trace: Building an Adjacency Matrix



	0	1	2	3	4
0	0	1	0	0	1
1	1	0	1	0	1
2	0	1	0	1	0
3	0	0	1	0	1
4	1	1	0	0	0

m[row][col] (1 = true)

Edge 3–4: **m[3][4] = true** (row 3, column 4).

Step 12 / 14

```
1 boolean[][] m = new boolean[n][n];
2 void addEdge(int u, int v) {
▶   m[u][v] = true;
4   m[v][u] = true;  // not in a digraph
5 }
6 // main: same 6 edges (n = 5)
7 for (int[] e : E) g.addEdge(e[0], e[1]);
```

VARIABLES

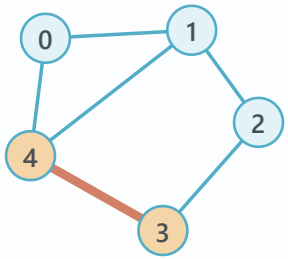
`u = 3` `v = 4`

CALL STACK

`addEdge(3, 4)`

`main()`

Code Trace: Building an Adjacency Matrix



	0	1	2	3	4
0	0	1	0	0	1
1	1	0	1	0	1
2	0	1	0	1	0
3	0	0	1	0	1
4	1	1	0	1	0

m[row][col] (1 = true)

Mirror cell: **m[4][3] = true**. An undirected graph's matrix is symmetric about the diagonal.

Step 13 / 14

```
1 boolean[][] m = new boolean[n][n];
2 void addEdge(int u, int v) {
3   m[u][v] = true;
▶   m[v][u] = true;  // not in a digraph
5 }
6 // main: same 6 edges (n = 5)
7 for (int[] e : E) g.addEdge(e[0], e[1]);
```

VARIABLES

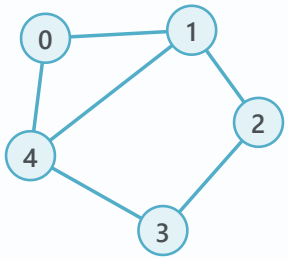
`u = 3` `v = 4`

CALL STACK

`addEdge(3, 4)`

`main()`

Code Trace: Building an Adjacency Matrix



	0	1	2	3	4
0	0	1	0	0	1
1	1	0	1	0	1
2	0	1	0	1	0
3	0	0	1	0	1
4	1	1	0	1	0

m[row][col] (1 = true)

Done: 12 cells are true, 13 are false and wasted. "Is there an edge 2–4?" → one access to `m[2][4]`, **O(1)**; but finding the neighbors of 2 means scanning a whole row: O(n).

Step 14 / 14

```
1 boolean[][] m = new boolean[n][n];
2 void addEdge(int u, int v) {
3   m[u][v] = true;
4   m[v][u] = true;  // not in a digraph
5 }
6 // main: same 6 edges (n = 5)
▶ for (int[] e : E) g.addEdge(e[0], e[1]);
```

VARIABLES

—

CALL STACK

`main()`

TRAVERSAL (BFS / DFS)

Traversal Algorithms

A traversal systematically visits **every vertex reachable from a start vertex, each exactly once**.

Breadth-First Search (BFS)

First all the nearby neighbors, then their neighbors... BFS advances level by level.

`helper structure: queue (FIFO)`

Depth-First Search (DFS)

DFS follows one path as deep as it can and backs up at a dead end.

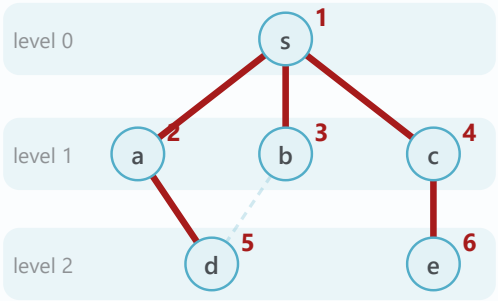
`helper structure: stack (LIFO) / recursion`

Unlike a tree, a graph can have cycles, and a vertex can be reached along several paths, so a **visited** mark (e.g. `boolean[] visited`) is essential; without it the algorithm can loop forever.

TRAVERSAL (BFS / DFS)

Breadth-First Search (BFS)

- Explores vertices **level by level**: first those 1 edge from the start, then those 2 edges away…
- Uses a **queue**: first discovered, first processed.
- Each vertex is visited exactly once.



Number = visit order; bold = BFS tree

Pros

+ Finds shortest paths (fewest edges) in unweighted graphs.
+ Discovers nearby vertices quickly.

Con

– Can use a lot of memory: a whole level waits in the queue.

Examples: "friends of my friends" in a social network (level 2); a web crawler exploring a site layer by layer.

The BFS Algorithm

Pseudocode

```
BFS(start):
  queue.add(start)
  mark start visited
  while queue is not empty:
    w = queue.remove()
    for each unvisited
        neighbor u of w:
      mark u visited
      queue.add(u)
```

A vertex is marked **when it enters** the queue, so no vertex enters the queue twice.

Java

```
void bfs(int s) {
  boolean[] visited = new boolean[n];
  Queue<Integer> q = new ArrayDeque<>();
  visited[s] = true;  q.add(s);
  while (!q.isEmpty()) {
    int w = q.poll();
    System.out.print(w + " ");
    for (int u : adj.get(w))
      if (!visited[u]) {
        visited[u] = true;
        q.add(u);
      }
  }
}
```

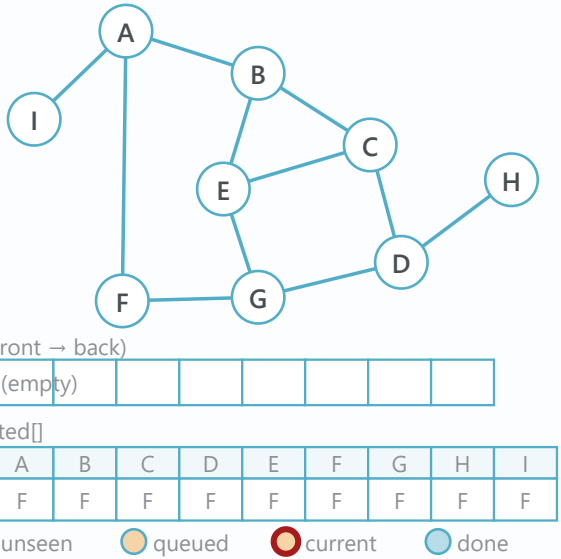
Code Trace: `bfs(A)`

```
1  void bfs(int s) {
▶    boolean[] visited = new boolean[n];
▶    Queue<Integer> q = new ArrayDeque<>();
4    visited[s] = true;  q.add(s);
5    while (!q.isEmpty()) {
6      int w = q.poll();
7      System.out.print(w + " ");
8      for (int u : adj.get(w))
9        if (!visited[u]) {
10         visited[u] = true;
11         q.add(u);
12       }
13   }
14 }
```

VARIABLES

`s = A`

OUTPUT



s = A (A=0, B=1, …, I=8; neighbors in alphabetical order). `visited` is all false; the queue is empty.

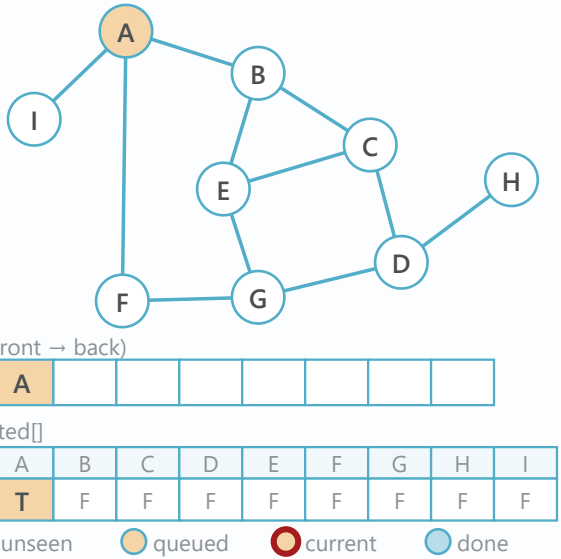
Code Trace: `bfs(A)`

```
1  void bfs(int s) {
2    boolean[] visited = new boolean[n];
3    Queue<Integer> q = new ArrayDeque<>();
▶    visited[s] = true;  q.add(s);
5    while (!q.isEmpty()) {
6      int w = q.poll();
7      System.out.print(w + " ");
8      for (int u : adj.get(w))
9        if (!visited[u]) {
10         visited[u] = true;
11         q.add(u);
12       }
13   }
14 }
```

VARIABLES

`s = A`

OUTPUT



A is marked as it enters the queue: `visited[A] = true`, q = [A].

Code Trace: `bfs(A)`

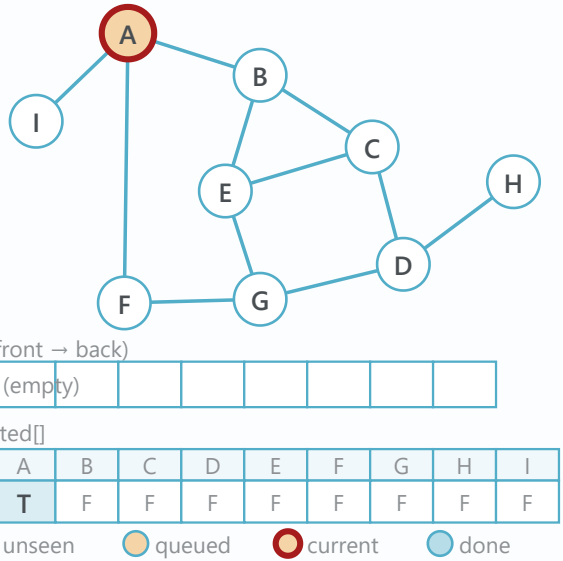
```
1  void bfs(int s) {
2    boolean[] visited = new boolean[n];
3    Queue<Integer> q = new ArrayDeque<>();
4    visited[s] = true;  q.add(s);
5    while (!q.isEmpty()) {
▶      int w = q.poll();
▶      System.out.print(w + " ");
8      for (int u : adj.get(w))
9        if (!visited[u]) {
10         visited[u] = true;
11         q.add(u);
12       }
13   }
14 }
```

VARIABLES

`w = A`

OUTPUT

```
A
```



Poll **A** from the front of the queue and print it. Now scan its neighbors B, F, I.

Code Trace: `bfs(A)`

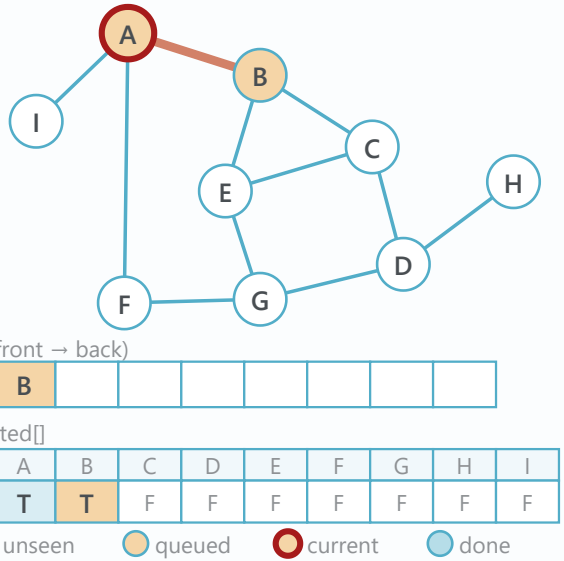
```
1  void bfs(int s) {
2    boolean[] visited = new boolean[n];
3    Queue<Integer> q = new ArrayDeque<>();
4    visited[s] = true;  q.add(s);
5    while (!q.isEmpty()) {
6      int w = q.poll();
7      System.out.print(w + " ");
8      for (int u : adj.get(w))
▶        if (!visited[u]) {
▶          visited[u] = true;
▶          q.add(u);
12       }
13   }
14 }
```

VARIABLES

`w = A` `u = B`

OUTPUT

```
A
```



u = B is unvisited → mark it and add it to the **back** of the queue (level 1).

Code Trace: `bfs(A)`

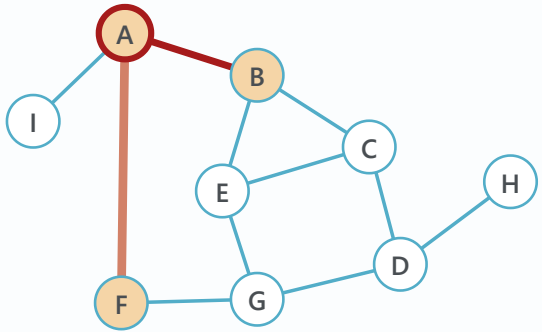
```
1  void bfs(int s) {
2    boolean[] visited = new boolean[n];
3    Queue<Integer> q = new ArrayDeque<>();
4    visited[s] = true;  q.add(s);
5    while (!q.isEmpty()) {
6      int w = q.poll();
7      System.out.print(w + " ");
8      for (int u : adj.get(w))
▶        if (!visited[u]) {
▶          visited[u] = true;
▶          q.add(u);
12       }
13   }
14 }
```

VARIABLES

`w = A` `u = F`

OUTPUT

```
A
```



q (front → back)

B	F						

visited[]

A	B	C	D	E	F	G	H	I
T	T	F	F	F	T	F	F	F

unseen queued current done

u = F is unvisited → mark it and add it to the **back** of the queue (level 1).

Code Trace: `bfs(A)`

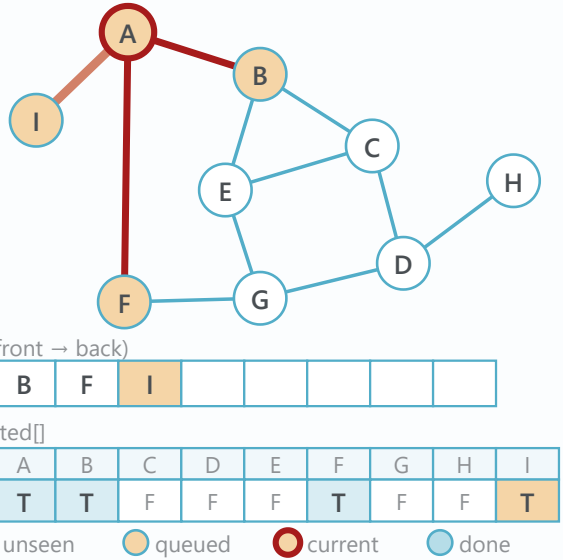
```
1  void bfs(int s) {
2    boolean[] visited = new boolean[n];
3    Queue<Integer> q = new ArrayDeque<>();
4    visited[s] = true;  q.add(s);
5    while (!q.isEmpty()) {
6      int w = q.poll();
7      System.out.print(w + " ");
8      for (int u : adj.get(w))
▶        if (!visited[u]) {
▶          visited[u] = true;
▶          q.add(u);
12       }
13   }
14 }
```

VARIABLES

`w = A` `u = I`

OUTPUT

```
A
```



u = I is unvisited → mark it and add it to the **back** of the queue (level 1).

Code Trace: `bfs(A)`

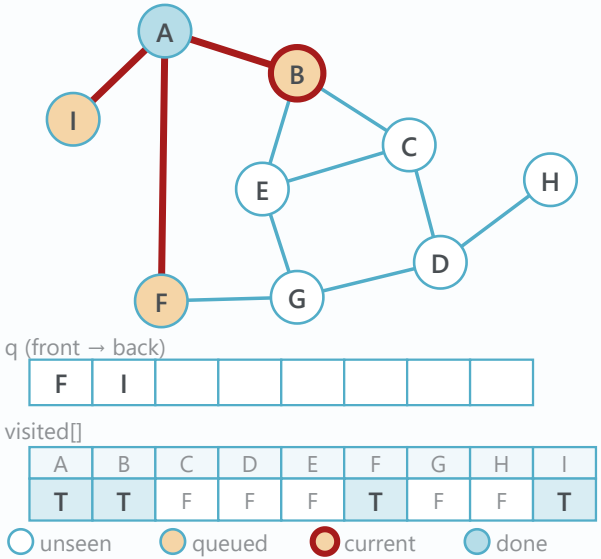
```
1  void bfs(int s) {
2    boolean[] visited = new boolean[n];
3    Queue<Integer> q = new ArrayDeque<>();
4    visited[s] = true;  q.add(s);
5    while (!q.isEmpty()) {
▶      int w = q.poll();
▶      System.out.print(w + " ");
8      for (int u : adj.get(w))
9        if (!visited[u]) {
10         visited[u] = true;
11         q.add(u);
12       }
13   }
14 }
```

VARIABLES

`w = B`

OUTPUT

```
A B
```



Poll **B** from the front of the queue and print it. Now scan its neighbors A, C, E.

Code Trace: `bfs(A)`

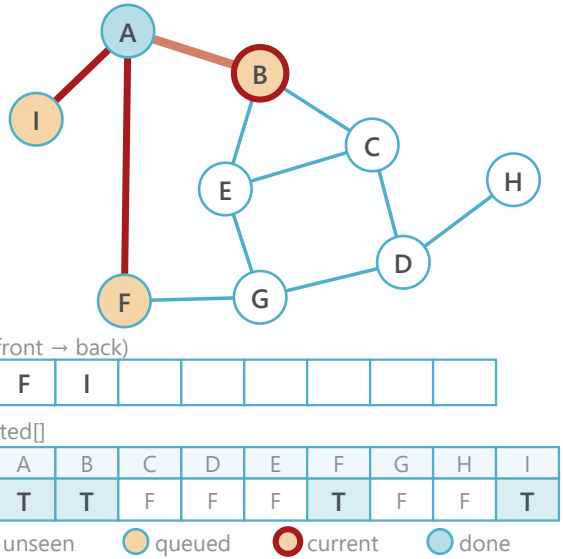
```
1  void bfs(int s) {
2    boolean[] visited = new boolean[n];
3    Queue<Integer> q = new ArrayDeque<>();
4    visited[s] = true;  q.add(s);
5    while (!q.isEmpty()) {
6      int w = q.poll();
7      System.out.print(w + " ");
8      for (int u : adj.get(w))
▶        if (!visited[u]) {
10         visited[u] = true;
11         q.add(u);
12       }
13   }
14 }
```

VARIABLES

`w = B` `u = A`

OUTPUT

```
A B
```



u = A: already visited → skip. Without the visited mark, A would enter the queue again.

Code Trace: `bfs(A)`

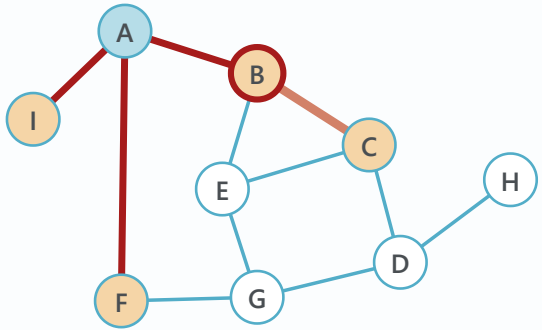
```
1  void bfs(int s) {
2    boolean[] visited = new boolean[n];
3    Queue<Integer> q = new ArrayDeque<>();
4    visited[s] = true;  q.add(s);
5    while (!q.isEmpty()) {
6      int w = q.poll();
7      System.out.print(w + " ");
8      for (int u : adj.get(w))
▶        if (!visited[u]) {
▶          visited[u] = true;
▶          q.add(u);
12       }
13   }
14 }
```

VARIABLES

`w = B` `u = C`

OUTPUT

```
A B
```



q (front → back)

F	I	C					

visited[]

A	B	C	D	E	F	G	H	I
T	T	T	F	F	T	F	F	T

unseen · queued · current · done

u = C is unvisited → mark it and add it to the **back** of the queue (level 2).

Code Trace: `bfs(A)`

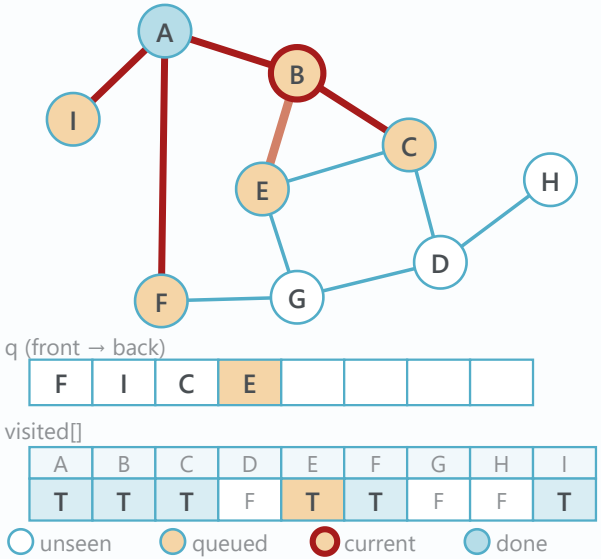
```
1  void bfs(int s) {
2    boolean[] visited = new boolean[n];
3    Queue<Integer> q = new ArrayDeque<>();
4    visited[s] = true;  q.add(s);
5    while (!q.isEmpty()) {
6      int w = q.poll();
7      System.out.print(w + " ");
8      for (int u : adj.get(w))
▶        if (!visited[u]) {
▶          visited[u] = true;
▶          q.add(u);
12       }
13   }
14 }
```

VARIABLES

`w = B` `u = E`

OUTPUT

```
A B
```



u = E is unvisited → mark it and add it to the **back** of the queue (level 2).

Code Trace: `bfs(A)`

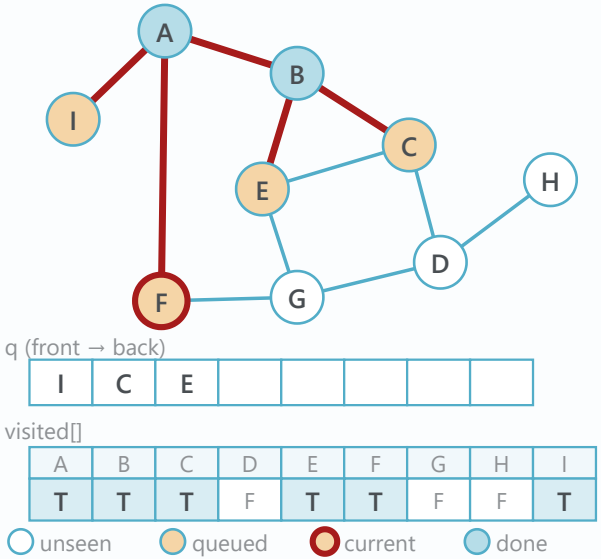
```
1  void bfs(int s) {
2    boolean[] visited = new boolean[n];
3    Queue<Integer> q = new ArrayDeque<>();
4    visited[s] = true;  q.add(s);
5    while (!q.isEmpty()) {
▶      int w = q.poll();
▶      System.out.print(w + " ");
8      for (int u : adj.get(w))
9        if (!visited[u]) {
10         visited[u] = true;
11         q.add(u);
12       }
13   }
14 }
```

VARIABLES

`w = F`

OUTPUT

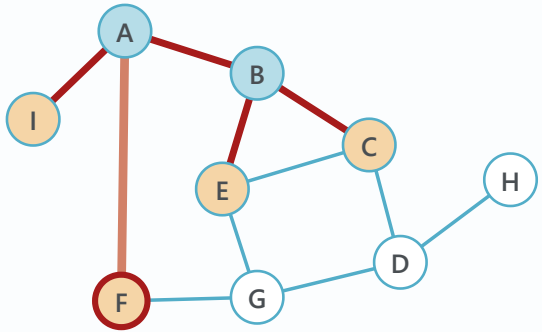
```
A B F
```



Poll **F** from the front of the queue and print it. Now scan its neighbors A, G.

Code Trace: `bfs(A)`

```
1  void bfs(int s) {
2    boolean[] visited = new boolean[n];
3    Queue<Integer> q = new ArrayDeque<>();
4    visited[s] = true;  q.add(s);
5    while (!q.isEmpty()) {
6      int w = q.poll();
7      System.out.print(w + " ");
8      for (int u : adj.get(w))
▶        if (!visited[u]) {
10         visited[u] = true;
11         q.add(u);
12       }
13   }
14 }
```



q (front → back)

I	C	E					

visited[]

A	B	C	D	E	F	G	H	I
T	T	T	F	T	T	F	F	T

unseen · queued · current · done

u = A: already visited → skip.

VARIABLES

`w = F` `u = A`

OUTPUT

```
A B F
```

Code Trace: `bfs(A)`

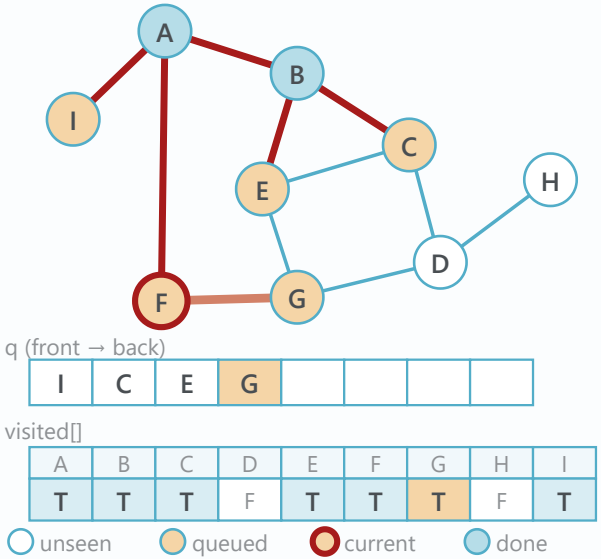
```
1  void bfs(int s) {
2    boolean[] visited = new boolean[n];
3    Queue<Integer> q = new ArrayDeque<>();
4    visited[s] = true;  q.add(s);
5    while (!q.isEmpty()) {
6      int w = q.poll();
7      System.out.print(w + " ");
8      for (int u : adj.get(w))
▶        if (!visited[u]) {
▶          visited[u] = true;
▶          q.add(u);
12       }
13   }
14 }
```

VARIABLES

`w = F` `u = G`

OUTPUT

```
A B F
```



u = G is unvisited → mark it and add it to the **back** of the queue (level 2).

Code Trace: `bfs(A)`

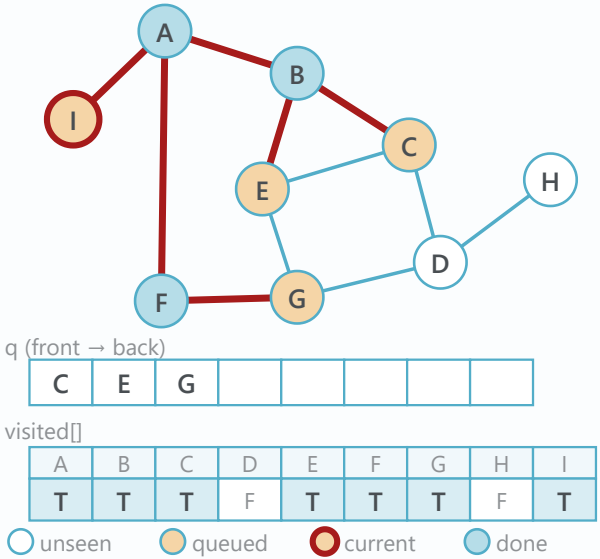
```
1  void bfs(int s) {
2    boolean[] visited = new boolean[n];
3    Queue<Integer> q = new ArrayDeque<>();
4    visited[s] = true;  q.add(s);
5    while (!q.isEmpty()) {
▶      int w = q.poll();
▶      System.out.print(w + " ");
8      for (int u : adj.get(w))
9        if (!visited[u]) {
10         visited[u] = true;
11         q.add(u);
12       }
13   }
14 }
```

VARIABLES

`w = I`

OUTPUT

```
A B F I
```



Poll **I** from the front of the queue and print it. Now scan its neighbor A.

Code Trace: `bfs(A)`

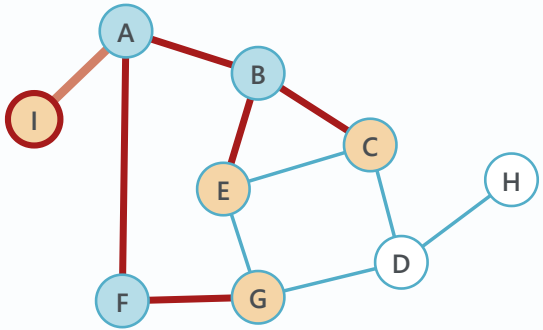
```
1  void bfs(int s) {
2    boolean[] visited = new boolean[n];
3    Queue<Integer> q = new ArrayDeque<>();
4    visited[s] = true;  q.add(s);
5    while (!q.isEmpty()) {
6      int w = q.poll();
7      System.out.print(w + " ");
8      for (int u : adj.get(w))
▶        if (!visited[u]) {
10         visited[u] = true;
11         q.add(u);
12       }
13   }
14 }
```

VARIABLES

`w = I` `u = A`

OUTPUT

```
A B F I
```



q (front → back)

C	E	G					

visited[]

A	B	C	D	E	F	G	H	I
T	T	T	F	T	T	T	F	T

unseen · queued · current · done

u = A: already visited → skip.

Code Trace: `bfs(A)` (continued)

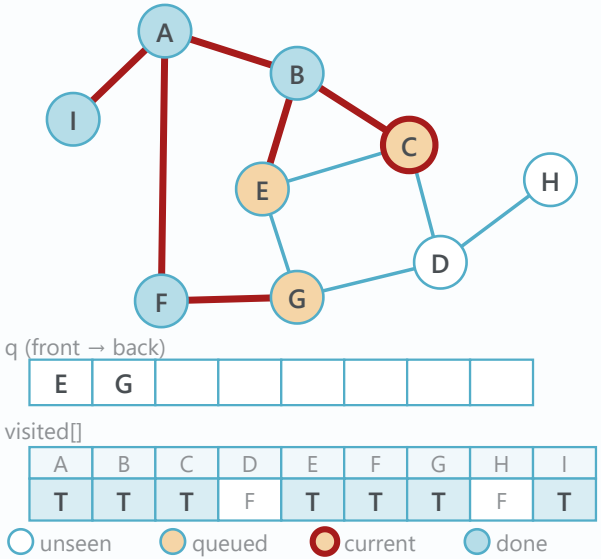
```
1  void bfs(int s) {
2    boolean[] visited = new boolean[n];
3    Queue<Integer> q = new ArrayDeque<>();
4    visited[s] = true;  q.add(s);
5    while (!q.isEmpty()) {
▶      int w = q.poll();
▶      System.out.print(w + " ");
8      for (int u : adj.get(w))
9        if (!visited[u]) {
10         visited[u] = true;
11         q.add(u);
12       }
13   }
14 }
```

VARIABLES

`w = C`

OUTPUT

```
A B F I C
```



Poll **C** from the front of the queue and print it. Now scan its neighbors B, D, E.

Code Trace: `bfs(A)` (continued)

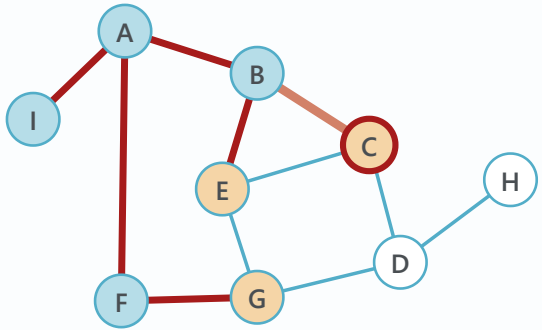
```
1  void bfs(int s) {
2    boolean[] visited = new boolean[n];
3    Queue<Integer> q = new ArrayDeque<>();
4    visited[s] = true;  q.add(s);
5    while (!q.isEmpty()) {
6      int w = q.poll();
7      System.out.print(w + " ");
8      for (int u : adj.get(w))
▶        if (!visited[u]) {
10         visited[u] = true;
11         q.add(u);
12       }
13   }
14 }
```

VARIABLES

`w = C` `u = B`

OUTPUT

```
A B F I C
```



q (front → back)

E	G						

visited[]

A	B	C	D	E	F	G	H	I
T	T	T	F	T	T	T	F	T

unseen · queued · current · done

u = B: already visited → skip.

Code Trace: `bfs(A)` (continued)

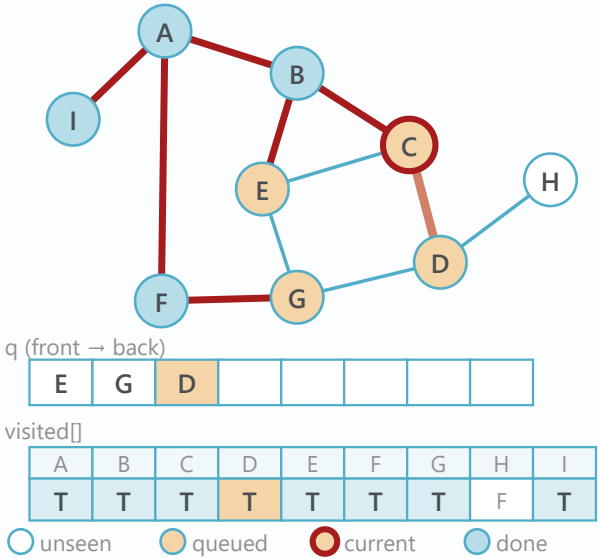
```
1  void bfs(int s) {
2    boolean[] visited = new boolean[n];
3    Queue<Integer> q = new ArrayDeque<>();
4    visited[s] = true;  q.add(s);
5    while (!q.isEmpty()) {
6      int w = q.poll();
7      System.out.print(w + " ");
8      for (int u : adj.get(w))
▶        if (!visited[u]) {
▶          visited[u] = true;
▶          q.add(u);
12       }
13   }
14 }
```

VARIABLES

`w = C` `u = D`

OUTPUT

```
A B F I C
```



u = D is unvisited → mark it and add it to the **back** of the queue (level 3).

Code Trace: `bfs(A)` (continued)

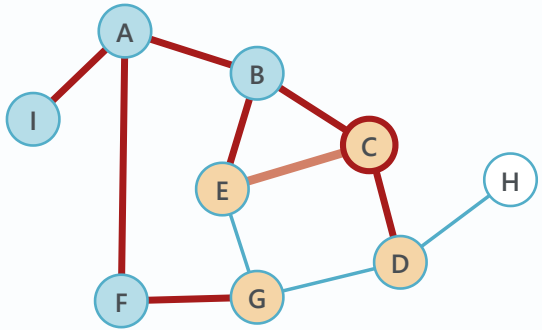
```
1  void bfs(int s) {
2    boolean[] visited = new boolean[n];
3    Queue<Integer> q = new ArrayDeque<>();
4    visited[s] = true;  q.add(s);
5    while (!q.isEmpty()) {
6      int w = q.poll();
7      System.out.print(w + " ");
8      for (int u : adj.get(w))
▶        if (!visited[u]) {
10         visited[u] = true;
11         q.add(u);
12       }
13   }
14 }
```

VARIABLES

`w = C` `u = E`

OUTPUT

```
A B F I C
```



q (front → back)

E	G	D					

visited[]

A	B	C	D	E	F	G	H	I
T	T	T	T	T	T	T	F	T

unseen queued current done

u = E: already visited → skip.

Code Trace: `bfs(A)` (continued)

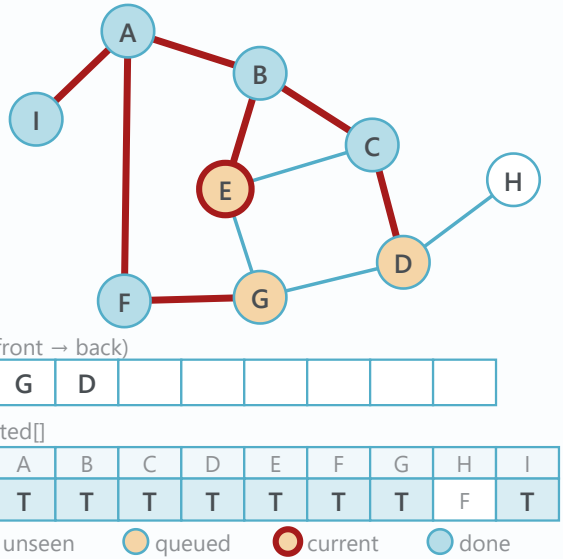
```
1  void bfs(int s) {
2    boolean[] visited = new boolean[n];
3    Queue<Integer> q = new ArrayDeque<>();
4    visited[s] = true;  q.add(s);
5    while (!q.isEmpty()) {
▶      int w = q.poll();
▶      System.out.print(w + " ");
8      for (int u : adj.get(w))
9        if (!visited[u]) {
10         visited[u] = true;
11         q.add(u);
12       }
13   }
14 }
```

VARIABLES

`w = E`

OUTPUT

```
A B F I C E
```



Poll **E** from the front of the queue and print it. Now scan its neighbors B, C, G.

Code Trace: `bfs(A)` (continued)

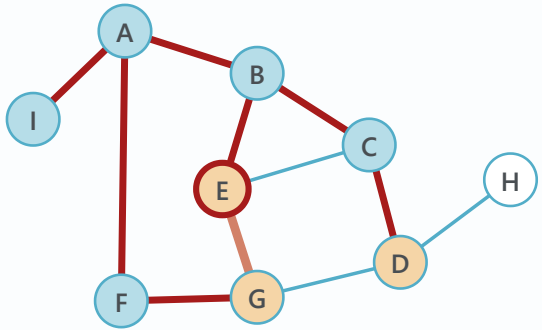
```
1  void bfs(int s) {
2    boolean[] visited = new boolean[n];
3    Queue<Integer> q = new ArrayDeque<>();
4    visited[s] = true;  q.add(s);
5    while (!q.isEmpty()) {
6      int w = q.poll();
7      System.out.print(w + " ");
8      for (int u : adj.get(w))
▶        if (!visited[u]) {
10         visited[u] = true;
11         q.add(u);
12       }
13   }
14 }
```

VARIABLES

`w = E` `u = G`

OUTPUT

```
A B F I C E
```



q (front → back)

G	D						

visited[]

A	B	C	D	E	F	G	H	I
T	T	T	T	T	T	T	F	T

unseen · queued · current · done

u = B, C, G: all already visited → skip.

Code Trace: `bfs(A)` (continued)

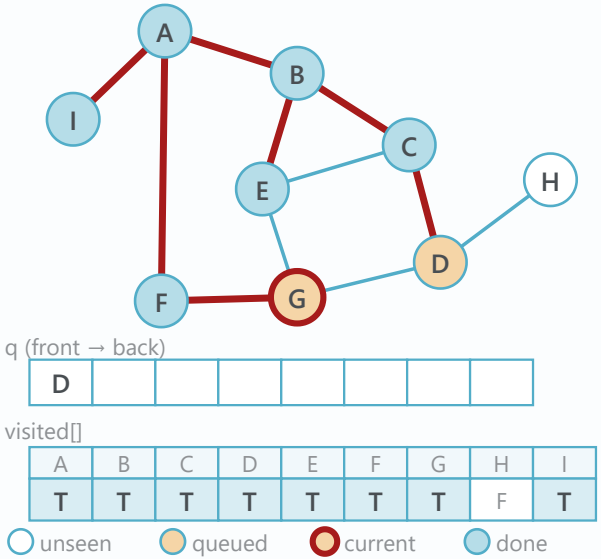
```
1  void bfs(int s) {
2    boolean[] visited = new boolean[n];
3    Queue<Integer> q = new ArrayDeque<>();
4    visited[s] = true;  q.add(s);
5    while (!q.isEmpty()) {
▶      int w = q.poll();
▶      System.out.print(w + " ");
8      for (int u : adj.get(w))
9        if (!visited[u]) {
10         visited[u] = true;
11         q.add(u);
12       }
13   }
14 }
```

VARIABLES

`w = G`

OUTPUT

```
A B F I C E G
```



Poll **G** from the front of the queue and print it. Now scan its neighbors D, E, F.

Code Trace: `bfs(A)` (continued)

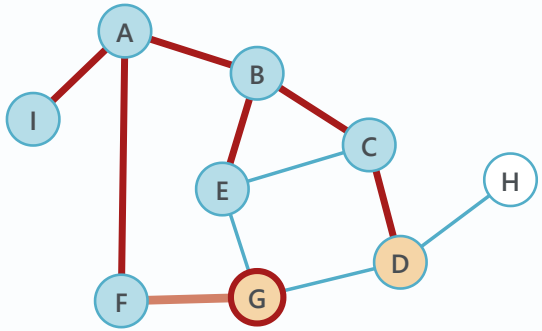
```
1  void bfs(int s) {
2    boolean[] visited = new boolean[n];
3    Queue<Integer> q = new ArrayDeque<>();
4    visited[s] = true;  q.add(s);
5    while (!q.isEmpty()) {
6      int w = q.poll();
7      System.out.print(w + " ");
8      for (int u : adj.get(w))
▶        if (!visited[u]) {
10         visited[u] = true;
11         q.add(u);
12       }
13   }
14 }
```

VARIABLES

`w = G` `u = F`

OUTPUT

```
A B F I C E G
```



q (front → back)

D							

visited[]

A	B	C	D	E	F	G	H	I
T	T	T	T	T	T	T	F	T

unseen · queued · current · done

u = D, E, F: all already visited → skip.

Code Trace: `bfs(A)` (continued)

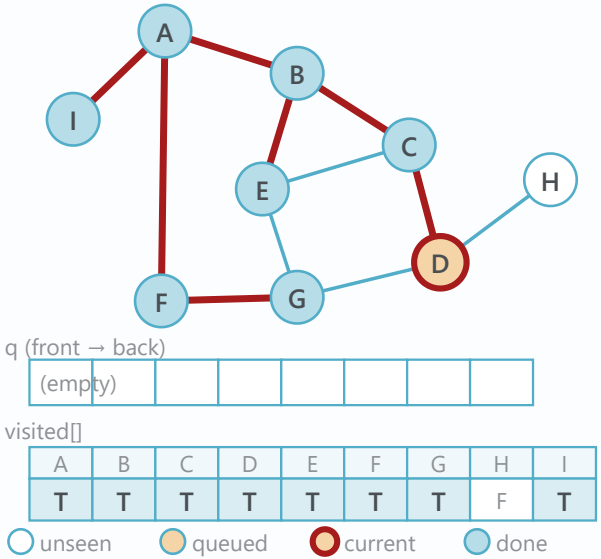
```
1  void bfs(int s) {
2    boolean[] visited = new boolean[n];
3    Queue<Integer> q = new ArrayDeque<>();
4    visited[s] = true;  q.add(s);
5    while (!q.isEmpty()) {
▶      int w = q.poll();
▶      System.out.print(w + " ");
8      for (int u : adj.get(w))
9        if (!visited[u]) {
10         visited[u] = true;
11         q.add(u);
12       }
13   }
14 }
```

VARIABLES

`w = D`

OUTPUT

```
A B F I C E G D
```



Poll **D** from the front of the queue and print it. Now scan its neighbors C, G, H.

Code Trace: `bfs(A)` (continued)

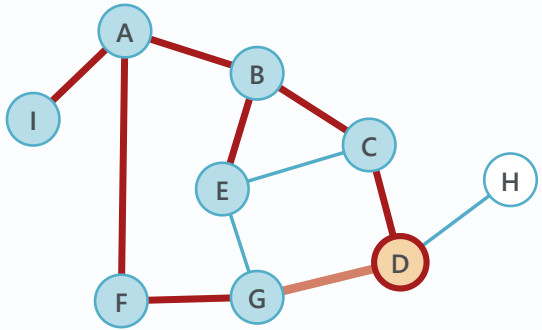
```
1  void bfs(int s) {
2    boolean[] visited = new boolean[n];
3    Queue<Integer> q = new ArrayDeque<>();
4    visited[s] = true;  q.add(s);
5    while (!q.isEmpty()) {
6      int w = q.poll();
7      System.out.print(w + " ");
8      for (int u : adj.get(w))
▶        if (!visited[u]) {
10         visited[u] = true;
11         q.add(u);
12       }
13   }
14 }
```

VARIABLES

`w = D` `u = G`

OUTPUT

```
A B F I C E G D
```



q (front → back)

(empty)							

visited[]

A	B	C	D	E	F	G	H	I
T	T	T	T	T	T	T	F	T

unseen · queued · current · done

u = C, G: all already visited → skip.

Code Trace: `bfs(A)` (continued)

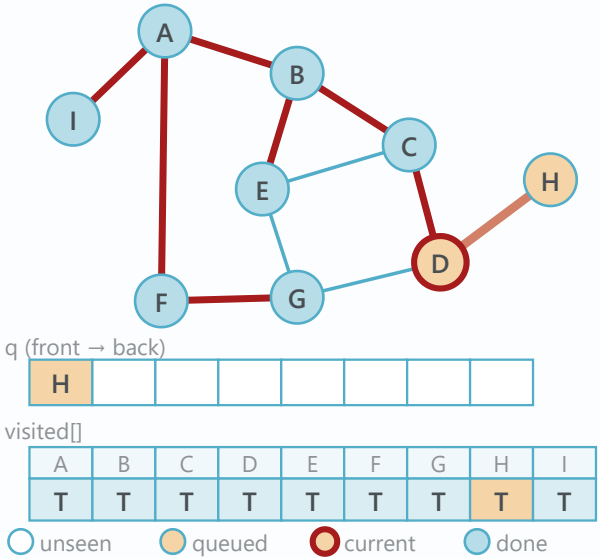
```
1  void bfs(int s) {
2    boolean[] visited = new boolean[n];
3    Queue<Integer> q = new ArrayDeque<>();
4    visited[s] = true;  q.add(s);
5    while (!q.isEmpty()) {
6      int w = q.poll();
7      System.out.print(w + " ");
8      for (int u : adj.get(w))
▶        if (!visited[u]) {
▶          visited[u] = true;
▶          q.add(u);
12       }
13   }
14 }
```

VARIABLES

`w = D` `u = H`

OUTPUT

```
A B F I C E G D
```



u = H is unvisited → mark it and add it to the **back** of the queue (level 4).

Code Trace: `bfs(A)` (continued)

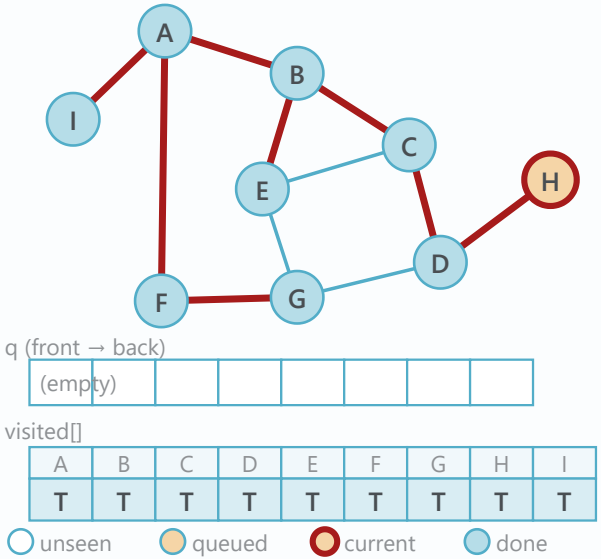
```
1  void bfs(int s) {
2    boolean[] visited = new boolean[n];
3    Queue<Integer> q = new ArrayDeque<>();
4    visited[s] = true;  q.add(s);
5    while (!q.isEmpty()) {
▶      int w = q.poll();
▶      System.out.print(w + " ");
8      for (int u : adj.get(w))
9        if (!visited[u]) {
10         visited[u] = true;
11         q.add(u);
12       }
13   }
14 }
```

VARIABLES

`w = H`

OUTPUT

```
A B F I C E G D H
```



Poll **H** from the front of the queue and print it. Now scan its neighbor D.

Code Trace: `bfs(A)` (continued)

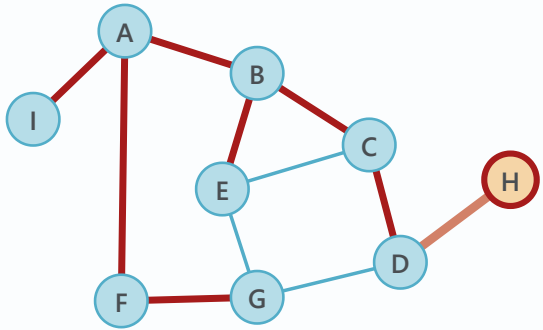
```
1  void bfs(int s) {
2    boolean[] visited = new boolean[n];
3    Queue<Integer> q = new ArrayDeque<>();
4    visited[s] = true;  q.add(s);
5    while (!q.isEmpty()) {
6      int w = q.poll();
7      System.out.print(w + " ");
8      for (int u : adj.get(w))
▶        if (!visited[u]) {
10         visited[u] = true;
11         q.add(u);
12       }
13   }
14 }
```

VARIABLES

`w = H` `u = D`

OUTPUT

```
A B F I C E G D H
```



q (front → back)

(empty)							

visited[]

A	B	C	D	E	F	G	H	I
T	T	T	T	T	T	T	T	T

unseen · queued · current · done

u = D: already visited → skip.

Code Trace: `bfs(A)` (continued)

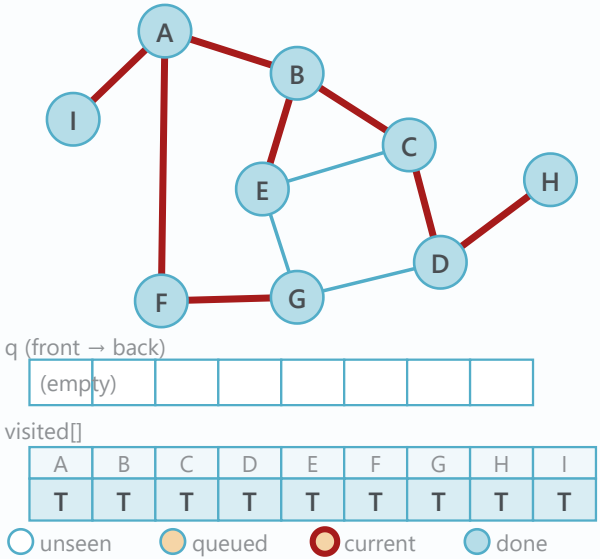
```
1  void bfs(int s) {
2    boolean[] visited = new boolean[n];
3    Queue<Integer> q = new ArrayDeque<>();
4    visited[s] = true;  q.add(s);
▶    while (!q.isEmpty()) {
6      int w = q.poll();
7      System.out.print(w + " ");
8      for (int u : adj.get(w))
9        if (!visited[u]) {
10         visited[u] = true;
11         q.add(u);
12       }
13   }
14 }
```

VARIABLES

—

OUTPUT

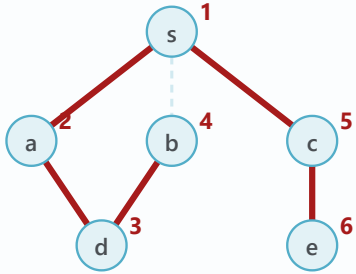
```
A B F I C E G D H
```



The queue is empty → done. Order **A B F I C E G D H**: level by level. Each vertex enters the queue once and each list is scanned once → **O(n + m)**.

Depth-First Search (DFS)

- Explores vertices **along a path**, going as deep as it can.
- At a dead end (no unvisited neighbors left) it **backtracks** and tries another branch.
- Uses a **stack** or recursive calls.



s→a→d→b goes deep; back up at the dead end, then c→e

Pros

+ Usually uses little memory (only the current path is on the stack).
+ Follows a path all the way to its end.

Con

– Does not guarantee shortest paths.

Uses: solving mazes, detecting cycles, finding connected components, and topological sorting of directed acyclic graphs.

The DFS Algorithm

Recursive

```
void dfs(int v, boolean[] visited) {
  visited[v] = true;
  System.out.print(v + " ");
  for (int u : adj.get(v))
    if (!visited[u])
      dfs(u, visited);
}
```

Recursion keeps the stack for us, as the **call stack**.

With an explicit stack

```
void dfsStack(int s) {
  boolean[] visited = new boolean[n];
  Deque<Integer> stack = new ArrayDeque<>();
  stack.push(s);  visited[s] = true;
  while (!stack.isEmpty()) {
    int v = stack.peek();
    Integer next = null;
    for (int u : adj.get(v))
      if (!visited[u]) { next = u; break; }
    if (next == null) stack.pop(); // back
    else { visited[next] = true;
           stack.push(next); }
  }
}
```

Code Trace: `dfs(A)` (recursive)

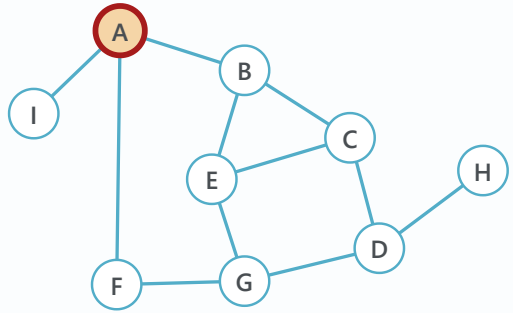
```
1 void dfs(int v, boolean[] visited) {
▶   visited[v] = true;
▶   System.out.print(v + " ");
4   for (int u : adj.get(v))
5     if (!visited[u])
6       dfs(u, visited);
7 }
```

CALL STACK

- `dfs(A)`
- `main()`

OUTPUT

```
A
```



visited[]

A	B	C	D	E	F	G	H	I
T	F	F	F	F	F	F	F	F

unseen · on stack · on top · finished

`dfs(A, visited)` is called (A=0, B=1, …, I=8; neighbors in alphabetical order). A is marked and printed.

Step 1 / 15

VARIABLES

v = A

Code Trace: `dfs(A)` (recursive)

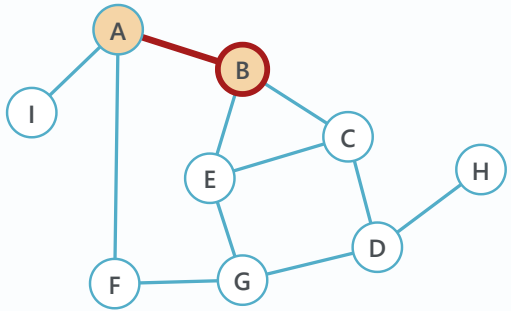
```
1 void dfs(int v, boolean[] visited) {
▶   visited[v] = true;
▶   System.out.print(v + " ");
4   for (int u : adj.get(v))
5     if (!visited[u])
6       dfs(u, visited);
7 }
```

CALL STACK

- `dfs(B)`
- `dfs(A)`
- `main()`

OUTPUT

```
A B
```



visited[]

A	B	C	D	E	F	G	H	I
T	T	F	F	F	F	F	F	F

unseen · on stack · on top · finished

Neighbor B of **A** is unvisited → call `dfs(B)` : a new frame opens; B is marked and printed.

VARIABLES

`v = B`

Code Trace: `dfs(A)` (recursive)

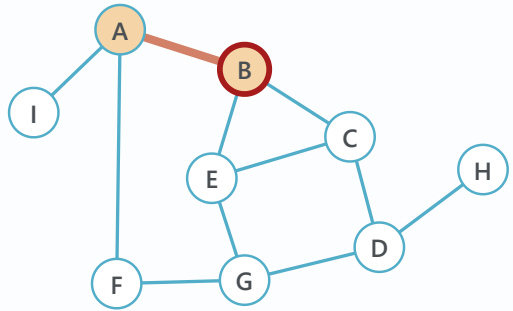
```
1 void dfs(int v, boolean[] visited) {
2   visited[v] = true;
3   System.out.print(v + " ");
▶   for (int u : adj.get(v))
▶     if (!visited[u])
6       dfs(u, visited);
7 }
```

CALL STACK

- **dfs(B)**
- dfs(A)
- main()

OUTPUT

```
A B
```



visited[]

A	B	C	D	E	F	G	H	I
T	T	F	F	F	F	F	F	F

unseen · on stack · on top · finished

B's neighbor A is already visited → skip.

Step 3 / 15

VARIABLES

v = B u = A

Code Trace: `dfs(A)` (recursive)

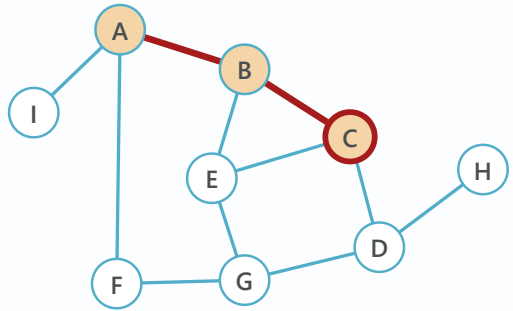
```
1 void dfs(int v, boolean[] visited) {
▶   visited[v] = true;
▶   System.out.print(v + " ");
4   for (int u : adj.get(v))
5     if (!visited[u])
6       dfs(u, visited);
7 }
```

CALL STACK

- `dfs(C)`
- `dfs(B)`
- `dfs(A)`
- `main()`

OUTPUT

```
A B C
```



visited[]

A	B	C	D	E	F	G	H	I
T	T	T	F	F	F	F	F	F

unseen · on stack · on top · finished

Neighbor C of **B** is unvisited → call `dfs(C)` : a new frame opens; C is marked and printed.

VARIABLES

v = **C**

Code Trace: `dfs(A)` (recursive)

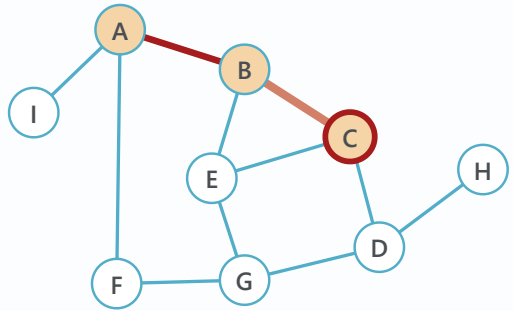
```
1 void dfs(int v, boolean[] visited) {
2   visited[v] = true;
3   System.out.print(v + " ");
▶   for (int u : adj.get(v))
▶     if (!visited[u])
6       dfs(u, visited);
7 }
```

CALL STACK

- `dfs(C)`
- `dfs(B)`
- `dfs(A)`
- `main()`

OUTPUT

```
A B C
```



visited[]

A	B	C	D	E	F	G	H	I
T	T	T	F	F	F	F	F	F

unseen · on stack · on top · finished

C's neighbor B is already visited → skip.

VARIABLES

`v = C` `u = B`

Code Trace: `dfs(A)` (recursive)

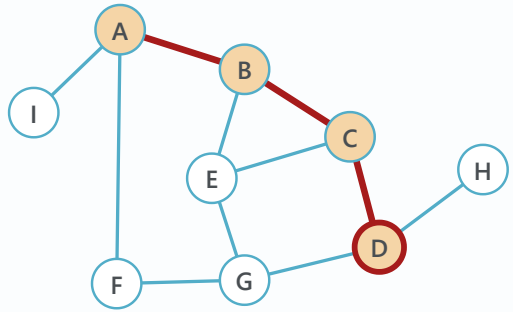
```
1 void dfs(int v, boolean[] visited) {
▶   visited[v] = true;
▶   System.out.print(v + " ");
4   for (int u : adj.get(v))
5     if (!visited[u])
6       dfs(u, visited);
7 }
```

CALL STACK

- `dfs(D)`
- `dfs(C)`
- `dfs(B)`
- `dfs(A)`
- `main()`

OUTPUT

```
A B C D
```



visited[]

A	B	C	D	E	F	G	H	I
T	T	T	T	F	F	F	F	F

unseen · on stack · on top · finished

Neighbor D of **C** is unvisited → call `dfs(D)` : a new frame opens; D is marked and printed.

VARIABLES

`v = D`

Code Trace: `dfs(A)` (recursive)

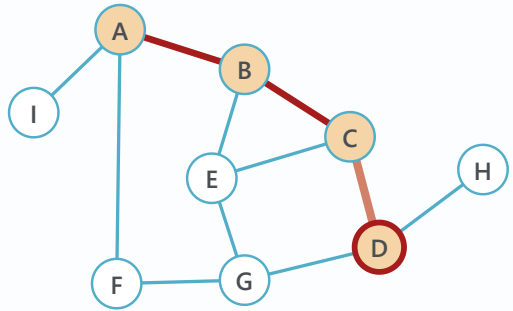
```
1 void dfs(int v, boolean[] visited) {
2   visited[v] = true;
3   System.out.print(v + " ");
▶   for (int u : adj.get(v))
▶     if (!visited[u])
6       dfs(u, visited);
7 }
```

CALL STACK

- `dfs(D)`
- `dfs(C)`
- `dfs(B)`
- `dfs(A)`
- `main()`

OUTPUT

```
A B C D
```



visited[]

A	B	C	D	E	F	G	H	I
T	T	T	T	F	F	F	F	F

unseen · on stack · on top · finished

D's neighbor C is already visited → skip.

Step 7 / 15

VARIABLES

`v = D` `u = C`

Code Trace: `dfs(A)` (recursive)

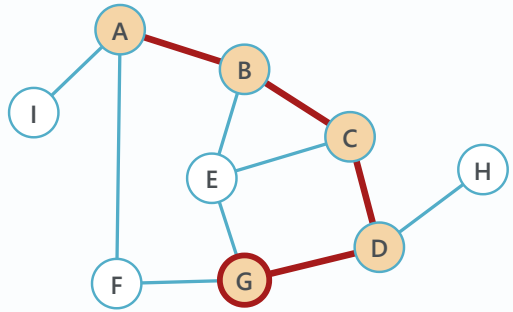
```
1 void dfs(int v, boolean[] visited) {
▶   visited[v] = true;
▶   System.out.print(v + " ");
4   for (int u : adj.get(v))
5     if (!visited[u])
6       dfs(u, visited);
7 }
```

CALL STACK

- `dfs(G)`
- `dfs(D)`
- `dfs(C)`
- `dfs(B)`
- `dfs(A)`
- `main()`

OUTPUT

```
A B C D G
```



visited[]

A	B	C	D	E	F	G	H	I
T	T	T	T	F	F	T	F	F

unseen · on stack · on top · finished

Neighbor G of **D** is unvisited → call `dfs(G)` : a new frame opens; G is marked and printed.

VARIABLES

`v = G`

Code Trace: `dfs(A)` (recursive)

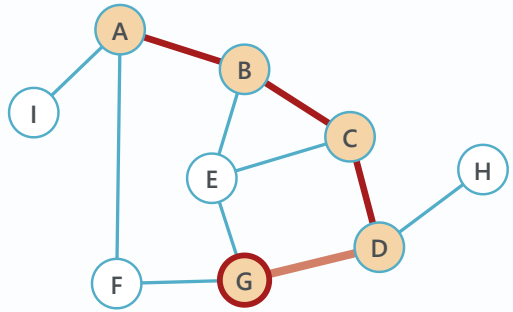
```
1 void dfs(int v, boolean[] visited) {
2   visited[v] = true;
3   System.out.print(v + " ");
▶   for (int u : adj.get(v))
▶     if (!visited[u])
6       dfs(u, visited);
7 }
```

CALL STACK

- `dfs(G)`
- `dfs(D)`
- `dfs(C)`
- `dfs(B)`
- `dfs(A)`
- `main()`

OUTPUT

```
A B C D G
```



visited[]

A	B	C	D	E	F	G	H	I
T	T	T	T	F	F	T	F	F

unseen · on stack · on top · finished

G's neighbor D is already visited → skip.

Step 9 / 15

VARIABLES

`v = G` `u = D`

Code Trace: `dfs(A)` (recursive)

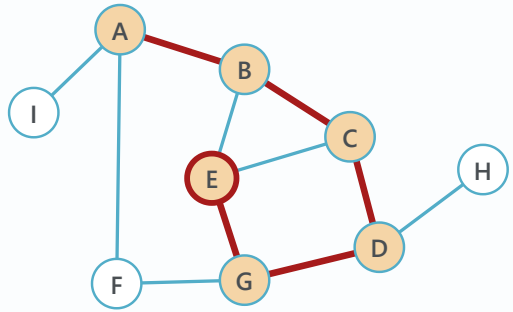
```
1 void dfs(int v, boolean[] visited) {
▶   visited[v] = true;
▶   System.out.print(v + " ");
4   for (int u : adj.get(v))
5     if (!visited[u])
6       dfs(u, visited);
7 }
```

CALL STACK

- `dfs(E)`
- `dfs(G)`
- `dfs(D)`
- `dfs(C)`
- `dfs(B)`
- `dfs(A)`
- `main()`

OUTPUT

```
A B C D G E
```



visited[]

A	B	C	D	E	F	G	H	I
T	T	T	T	T	F	T	F	F

unseen · on stack · on top · finished

Neighbor E of **G** is unvisited → call `dfs(E)` : a new frame opens; E is marked and printed.

VARIABLES

`v = E`

Code Trace: `dfs(A)` (recursive)

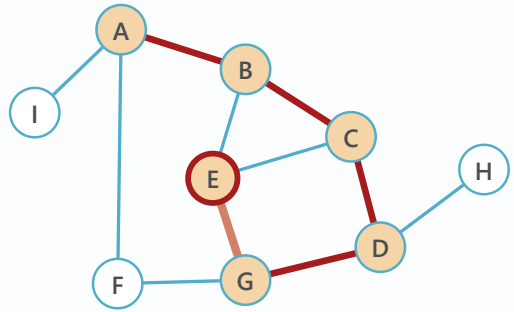
```
1 void dfs(int v, boolean[] visited) {
2   visited[v] = true;
3   System.out.print(v + " ");
▶   for (int u : adj.get(v))
▶     if (!visited[u])
6       dfs(u, visited);
7 }
```

CALL STACK

- dfs(E)
- dfs(G)
- dfs(D)
- dfs(C)
- dfs(B)
- dfs(A)
- main()

OUTPUT

```
A B C D G E
```



visited[]

A	B	C	D	E	F	G	H	I
T	T	T	T	T	F	T	F	F

unseen · on stack · on top · finished

E's neighbors B, C, G are already visited → skip.

VARIABLES

v = E u = G

Code Trace: `dfs(A)` (recursive)

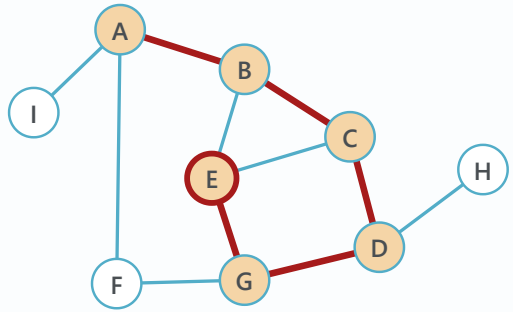
```
1 void dfs(int v, boolean[] visited) {
2   visited[v] = true;
3   System.out.print(v + " ");
4   for (int u : adj.get(v))
5     if (!visited[u])
6       dfs(u, visited);
▶ }
```

CALL STACK

- **dfs(E)**
- dfs(G)
- dfs(D)
- dfs(C)
- dfs(B)
- dfs(A)
- main()

OUTPUT

```
A B C D G E
```



visited[]

A	B	C	D	E	F	G	H	I
T	T	T	T	T	F	T	F	F

unseen · on stack · on top · finished

E has no neighbors left to visit → `dfs(E)` returns and its frame is popped (backtrack).

Step 12 / 15

VARIABLES

v = **E**

Code Trace: `dfs(A)` (recursive)

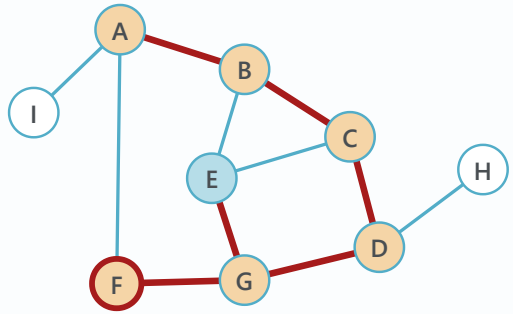
```
1 void dfs(int v, boolean[] visited) {
▶   visited[v] = true;
▶   System.out.print(v + " ");
4   for (int u : adj.get(v))
5     if (!visited[u])
6       dfs(u, visited);
7 }
```

CALL STACK

- `dfs(F)`
- `dfs(G)`
- `dfs(D)`
- `dfs(C)`
- `dfs(B)`
- `dfs(A)`
- `main()`

OUTPUT

```
A B C D G E F
```



visited[]

A	B	C	D	E	F	G	H	I
T	T	T	T	T	T	T	F	F

unseen · on stack · on top · finished

Neighbor F of **G** is unvisited → call `dfs(F)` : a new frame opens; F is marked and printed.

VARIABLES

`v = F`

Code Trace: `dfs(A)` (recursive)

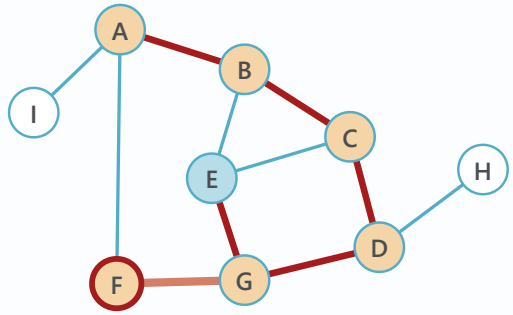
```
1 void dfs(int v, boolean[] visited) {
2   visited[v] = true;
3   System.out.print(v + " ");
▶   for (int u : adj.get(v))
▶     if (!visited[u])
6       dfs(u, visited);
7 }
```

CALL STACK

- `dfs(F)`
- `dfs(G)`
- `dfs(D)`
- `dfs(C)`
- `dfs(B)`
- `dfs(A)`
- `main()`

OUTPUT

```
A B C D G E F
```



visited[]

A	B	C	D	E	F	G	H	I
T	T	T	T	T	T	T	F	F

unseen · on stack · on top · finished

F's neighbors A, G are already visited → skip.

Step 14 / 15

VARIABLES

`v = F` `u = G`

Code Trace: `dfs(A)` (recursive)

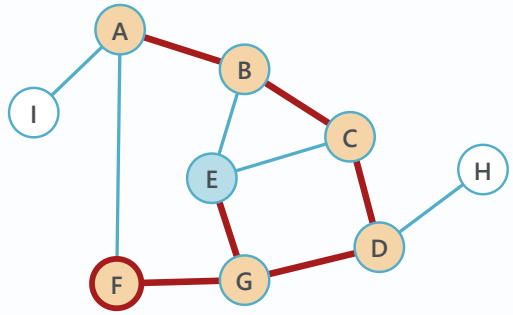
```
1 void dfs(int v, boolean[] visited) {
2   visited[v] = true;
3   System.out.print(v + " ");
4   for (int u : adj.get(v))
5     if (!visited[u])
6       dfs(u, visited);
▶ }
```

CALL STACK

- `dfs(F)`
- `dfs(G)`
- `dfs(D)`
- `dfs(C)`
- `dfs(B)`
- `dfs(A)`
- `main()`

OUTPUT

```
A B C D G E F
```



visited[]

A	B	C	D	E	F	G	H	I
T	T	T	T	T	T	T	F	F

unseen · on stack · on top · finished

F has no neighbors left to visit → `dfs(F)` returns and its frame is popped (backtrack).

Step 15 / 15

VARIABLES

v = **F**

Code Trace: `dfs(A)` (recursive, continued)

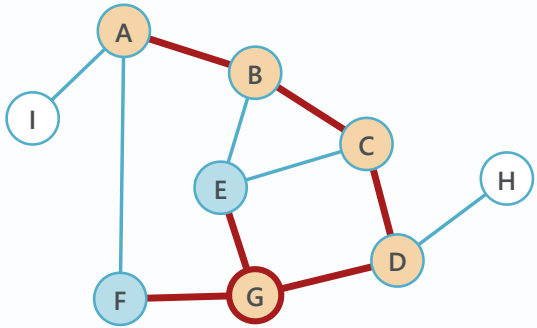
```
1 void dfs(int v, boolean[] visited) {
2   visited[v] = true;
3   System.out.print(v + " ");
4   for (int u : adj.get(v))
5     if (!visited[u])
6       dfs(u, visited);
▶ }
```

CALL STACK

- **`dfs(G)`**
- `dfs(D)`
- `dfs(C)`
- `dfs(B)`
- `dfs(A)`
- `main()`

OUTPUT

```
A B C D G E F
```



visited[]

A	B	C	D	E	F	G	H	I
T	T	T	T	T	T	T	F	F

unseen · on stack · on top · finished

G has no neighbors left to visit → `dfs(G)` returns and its frame is popped (backtrack).

Step 1 / 14

VARIABLES

v = **G**

Code Trace: `dfs(A)` (recursive, continued)

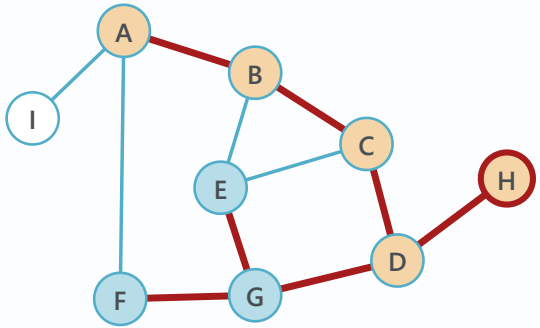
```
1 void dfs(int v, boolean[] visited) {
▶   visited[v] = true;
▶   System.out.print(v + " ");
4   for (int u : adj.get(v))
5     if (!visited[u])
6       dfs(u, visited);
7 }
```

CALL STACK

- `dfs(H)`
- `dfs(D)`
- `dfs(C)`
- `dfs(B)`
- `dfs(A)`
- `main()`

OUTPUT

```
A B C D G E F H
```



visited[]

A	B	C	D	E	F	G	H	I
T	T	T	T	T	T	T	T	F

unseen · on stack · on top · finished

Neighbor H of **D** is unvisited → call `dfs(H)`: a new frame opens; H is marked and printed.

VARIABLES

`v = H`

Code Trace: `dfs(A)` (recursive, continued)

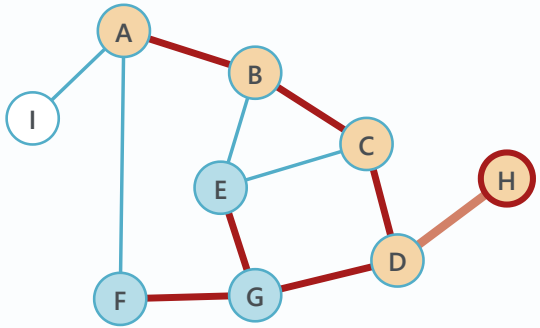
```
1 void dfs(int v, boolean[] visited) {
2   visited[v] = true;
3   System.out.print(v + " ");
▶   for (int u : adj.get(v))
▶     if (!visited[u])
6       dfs(u, visited);
7 }
```

CALL STACK

- `dfs(H)`
- `dfs(D)`
- `dfs(C)`
- `dfs(B)`
- `dfs(A)`
- `main()`

OUTPUT

```
A B C D G E F H
```



visited[]

A	B	C	D	E	F	G	H	I
T	T	T	T	T	T	T	T	F

unseen · on stack · on top · finished

H's neighbor D is already visited → skip.

Step 3 / 14

VARIABLES

`v = H` `u = D`

Code Trace: `dfs(A)` (recursive, continued)

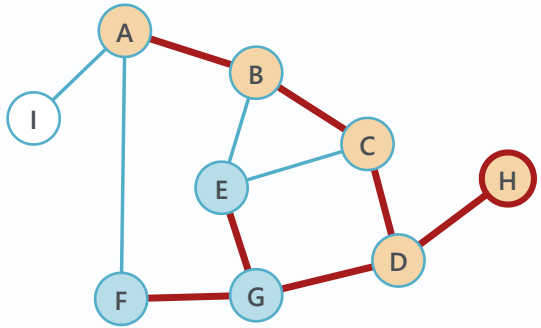
```
1 void dfs(int v, boolean[] visited) {
2   visited[v] = true;
3   System.out.print(v + " ");
4   for (int u : adj.get(v))
5     if (!visited[u])
6       dfs(u, visited);
▶ }
```

CALL STACK

- `dfs(H)`
- `dfs(D)`
- `dfs(C)`
- `dfs(B)`
- `dfs(A)`
- `main()`

OUTPUT

```
A B C D G E F H
```



visited[]

A	B	C	D	E	F	G	H	I
T	T	T	T	T	T	T	T	F

unseen · on stack · on top · finished

H has no neighbors left to visit → `dfs(H)` returns and its frame is popped (backtrack).

VARIABLES

`v = H`

Code Trace: `dfs(A)` (recursive, continued)

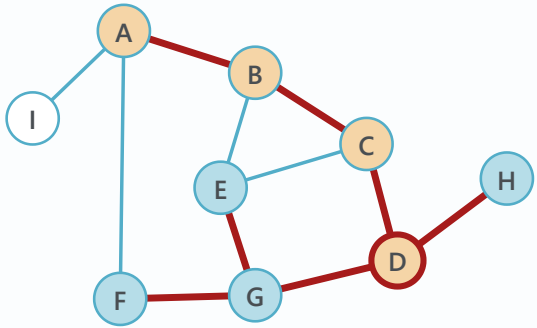
```
1 void dfs(int v, boolean[] visited) {
2   visited[v] = true;
3   System.out.print(v + " ");
4   for (int u : adj.get(v))
5     if (!visited[u])
6       dfs(u, visited);
▶ }
```

CALL STACK

- **`dfs(D)`**
- `dfs(C)`
- `dfs(B)`
- `dfs(A)`
- `main()`

OUTPUT

```
A B C D G E F H
```



visited[]

A	B	C	D	E	F	G	H	I
T	T	T	T	T	T	T	T	F

unseen · on stack · on top · finished

D has no neighbors left to visit → `dfs(D)` returns and its frame is popped (backtrack).

Step 5 / 14

VARIABLES

`v = D`

Code Trace: `dfs(A)` (recursive, continued)

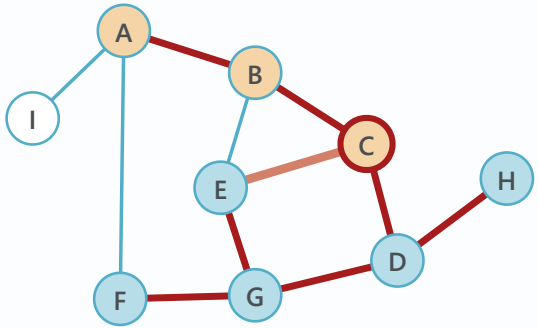
```
1 void dfs(int v, boolean[] visited) {
2   visited[v] = true;
3   System.out.print(v + " ");
▶   for (int u : adj.get(v))
▶     if (!visited[u])
6       dfs(u, visited);
7 }
```

CALL STACK

- `dfs(C)`
- `dfs(B)`
- `dfs(A)`
- `main()`

OUTPUT

```
A B C D G E F H
```



visited[]

A	B	C	D	E	F	G	H	I
T	T	T	T	T	T	T	T	F

unseen · on stack · on top · finished

C's neighbor E is already visited → skip.

Step 6 / 14

VARIABLES

`v = C` `u = E`

Code Trace: `dfs(A)` (recursive, continued)

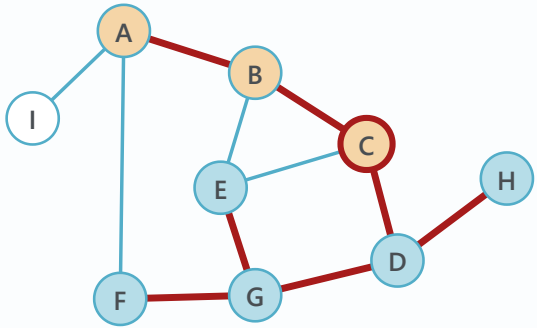
```
1 void dfs(int v, boolean[] visited) {
2   visited[v] = true;
3   System.out.print(v + " ");
4   for (int u : adj.get(v))
5     if (!visited[u])
6       dfs(u, visited);
▶ }
```

CALL STACK

- `dfs(C)`
- `dfs(B)`
- `dfs(A)`
- `main()`

OUTPUT

```
A B C D G E F H
```



visited[]

A	B	C	D	E	F	G	H	I
T	T	T	T	T	T	T	T	F

unseen · on stack · on top · finished

C has no neighbors left to visit → `dfs(C)` returns and its frame is popped (backtrack).

VARIABLES

`v = C`

Code Trace: `dfs(A)` (recursive, continued)

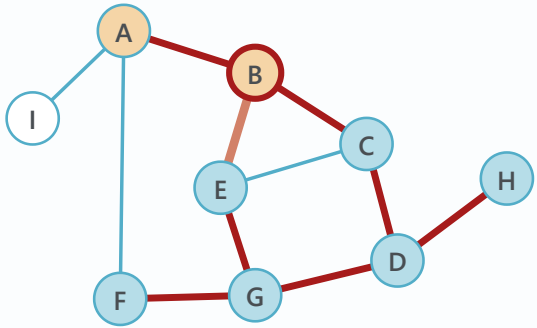
```
1 void dfs(int v, boolean[] visited) {
2   visited[v] = true;
3   System.out.print(v + " ");
▶   for (int u : adj.get(v))
▶     if (!visited[u])
6       dfs(u, visited);
7 }
```

CALL STACK

- **dfs(B)**
- dfs(A)
- main()

OUTPUT

```
A B C D G E F H
```



visited[]

A	B	C	D	E	F	G	H	I
T	T	T	T	T	T	T	T	F

unseen · on stack · on top · finished

B's neighbor E is already visited → skip.

VARIABLES

v = B u = E

Code Trace: `dfs(A)` (recursive, continued)

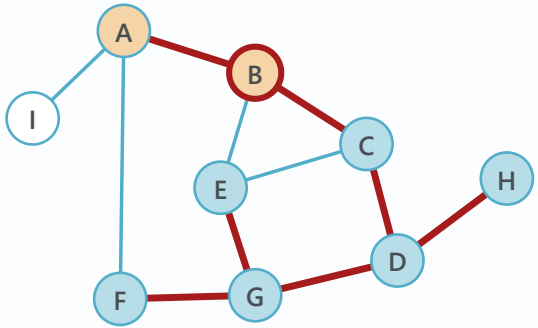
```
1 void dfs(int v, boolean[] visited) {
2   visited[v] = true;
3   System.out.print(v + " ");
4   for (int u : adj.get(v))
5     if (!visited[u])
6       dfs(u, visited);
▶ }
```

CALL STACK

- `dfs(B)`
- `dfs(A)`
- `main()`

OUTPUT

```
A B C D G E F H
```



visited[]

A	B	C	D	E	F	G	H	I
T	T	T	T	T	T	T	T	F

unseen · on stack · on top · finished

B has no neighbors left to visit → `dfs(B)` returns and its frame is popped (backtrack).

Step 9 / 14

VARIABLES

`v = B`

Code Trace: `dfs(A)` (recursive, continued)

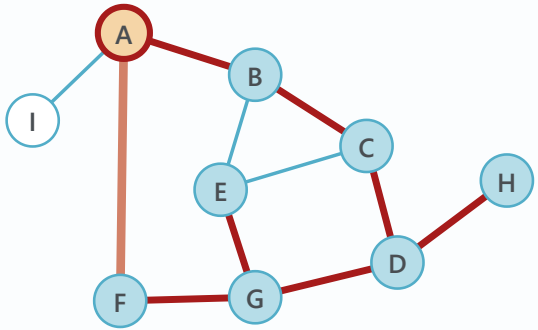
```
1 void dfs(int v, boolean[] visited) {
2   visited[v] = true;
3   System.out.print(v + " ");
▶   for (int u : adj.get(v))
▶     if (!visited[u])
6       dfs(u, visited);
7 }
```

CALL STACK

- `dfs(A)`
- `main()`

OUTPUT

```
A B C D G E F H
```



visited[]

A	B	C	D	E	F	G	H	I
T	T	T	T	T	T	T	T	F

unseen · on stack · on top · finished

A's neighbor F is already visited → skip.

VARIABLES

`v = A` `u = F`

Code Trace: `dfs(A)` (recursive, continued)

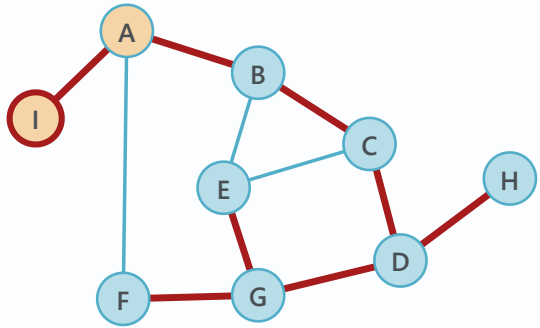
```
1 void dfs(int v, boolean[] visited) {
▶   visited[v] = true;
▶   System.out.print(v + " ");
4   for (int u : adj.get(v))
5     if (!visited[u])
6       dfs(u, visited);
7 }
```

CALL STACK

- **dfs(I)**
- dfs(A)
- main()

OUTPUT

```
A B C D G E F H I
```



visited[]

A	B	C	D	E	F	G	H	I
T	T	T	T	T	T	T	T	T

unseen · on stack · on top · finished

Neighbor I of **A** is unvisited → call `dfs(I)`: a new frame opens; I is marked and printed.

VARIABLES

`v = I`

Code Trace: `dfs(A)` (recursive, continued)

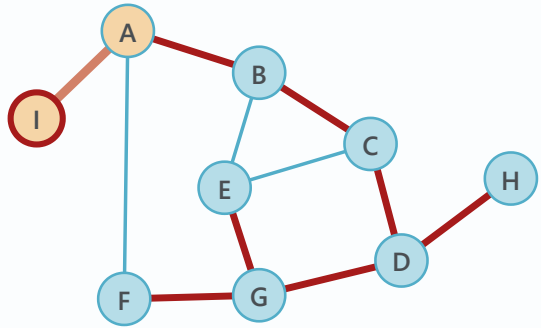
```
1 void dfs(int v, boolean[] visited) {
2   visited[v] = true;
3   System.out.print(v + " ");
▶   for (int u : adj.get(v))
▶     if (!visited[u])
6       dfs(u, visited);
7 }
```

CALL STACK

- **dfs(I)**
- dfs(A)
- main()

OUTPUT

```
A B C D G E F H I
```



visited[]

A	B	C	D	E	F	G	H	I
T	T	T	T	T	T	T	T	T

unseen · on stack · on top · finished

I's neighbor A is already visited → skip.

Step 12 / 14

VARIABLES

v = I u = A

Code Trace: `dfs(A)` (recursive, continued)

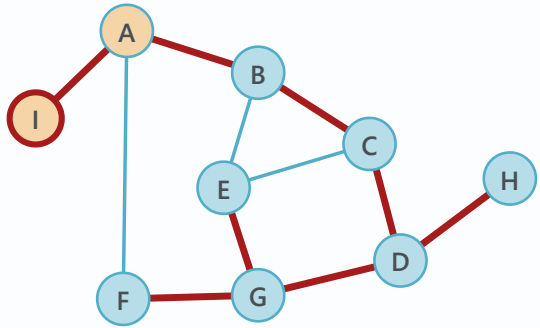
```
1 void dfs(int v, boolean[] visited) {
2   visited[v] = true;
3   System.out.print(v + " ");
4   for (int u : adj.get(v))
5     if (!visited[u])
6       dfs(u, visited);
▶ }
```

CALL STACK

- `dfs(I)`
- `dfs(A)`
- `main()`

OUTPUT

```
A B C D G E F H I
```



visited[]

A	B	C	D	E	F	G	H	I
T	T	T	T	T	T	T	T	T

○ unseen ● on stack ● on top ● finished

I has no neighbors left to visit → `dfs(I)` returns and its frame is popped (backtrack).

VARIABLES

`v = I`

Code Trace: `dfs(A)` (recursive, continued)

```
1 void dfs(int v, boolean[] visited) {
2   visited[v] = true;
3   System.out.print(v + " ");
4   for (int u : adj.get(v))
5     if (!visited[u])
6       dfs(u, visited);
▶ }
```

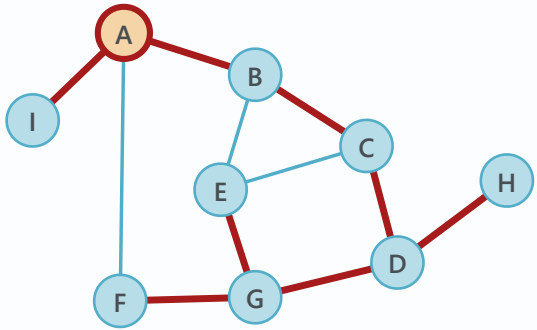
CALL STACK

`dfs(A)`

`main()`

OUTPUT

```
A B C D G E F H I
```



visited[]

A	B	C	D	E	F	G	H	I
T	T	T	T	T	T	T	T	T

unseen · on stack · on top · finished

`dfs(A)` returns and the stack is empty. Order: **A B C D G E F H I**.

VARIABLES

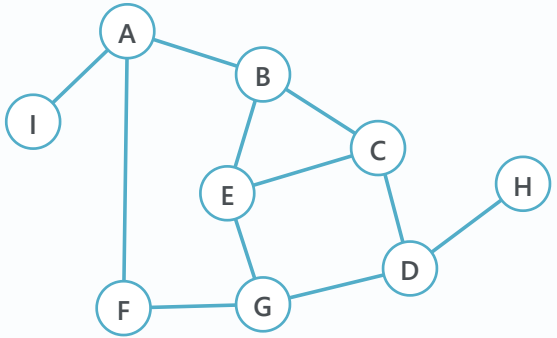
`v = A`

Code Trace: `dfsStack(A)`

```
1  void dfsStack(int s) {
2    boolean[] visited = new boolean[n];
3    Deque<Integer> stack = new ArrayDeque<>();
4    stack.push(s);  visited[s] = true;
5    while (!stack.isEmpty()) {
6      int v = stack.peek();
7      Integer next = null;
8      for (int u : adj.get(v))
9        if (!visited[u]) { next = u; break; }
10     if (next == null) stack.pop(); // back
11     else { visited[next] = true;
12            stack.push(next); }
13   }
14 }
```

VARIABLES

`s = A`



stack (bottom → top)

(empty)						

visited[]

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

unseen · on stack · on top · finished

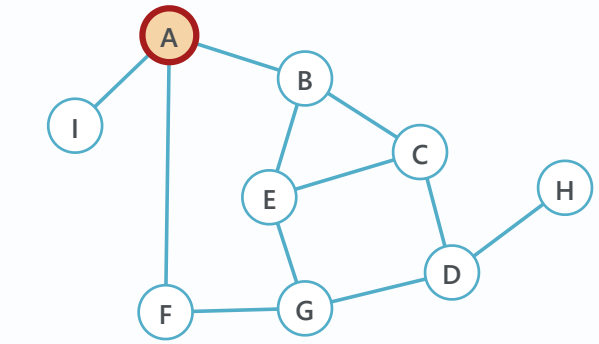
Start s = A. `visited` is all false; the stack is empty.

Code Trace: `dfsStack(A)`

```
1  void dfsStack(int s) {
2    boolean[] visited = new boolean[n];
3    Deque<Integer> stack = new ArrayDeque<>();
▶    stack.push(s);  visited[s] = true;
5    while (!stack.isEmpty()) {
6      int v = stack.peek();
7      Integer next = null;
8      for (int u : adj.get(v))
9        if (!visited[u]) { next = u; break; }
10     if (next == null) stack.pop(); // back
11     else { visited[next] = true;
12            stack.push(next); }
13   }
14 }
```

VARIABLES

`s = A`



stack (bottom → top)

A							

visited[]

	A	B	C	D	E	F	G	H	I
	T	F	F	F	F	F	F	F	F

unseen · on stack · on top · finished

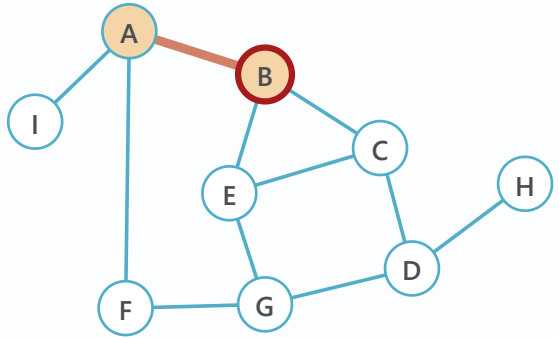
A is pushed and marked. The top of the stack = the vertex we are at.

Code Trace: `dfsStack(A)`

```
1  void dfsStack(int s) {
2    boolean[] visited = new boolean[n];
3    Deque<Integer> stack = new ArrayDeque<>();
4    stack.push(s);  visited[s] = true;
5    while (!stack.isEmpty()) {
▶      int v = stack.peek();
7      Integer next = null;
8      for (int u : adj.get(v))
▶        if (!visited[u]) { next = u; break; }
10     if (next == null) stack.pop(); // back
▶      else { visited[next] = true;
▶             stack.push(next); }
13   }
14 }
```

VARIABLES

`v = A` `next = B`



stack (bottom → top)

A	B						

visited[]

A	B	C	D	E	F	G	H	I
T	T	F	F	F	F	F	F	F

unseen · on stack · on top · finished

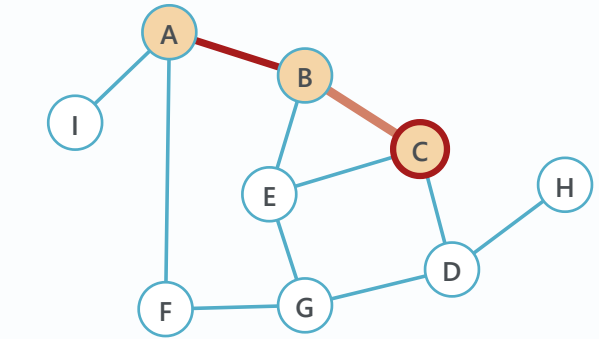
peek = **A**; first unvisited neighbor **B** (break) → mark it and push it.

Code Trace: `dfsStack(A)`

```
1  void dfsStack(int s) {
2    boolean[] visited = new boolean[n];
3    Deque<Integer> stack = new ArrayDeque<>();
4    stack.push(s);  visited[s] = true;
5    while (!stack.isEmpty()) {
▶      int v = stack.peek();
7      Integer next = null;
8      for (int u : adj.get(v))
▶        if (!visited[u]) { next = u; break; }
10     if (next == null) stack.pop(); // back
▶      else { visited[next] = true;
▶             stack.push(next); }
13   }
14 }
```

VARIABLES

`v = B` `next = C`



stack (bottom → top)

A	B	C					

visited[]

A	B	C	D	E	F	G	H	I
T	T	T	F	F	F	F	F	F

unseen · on stack · on top · finished

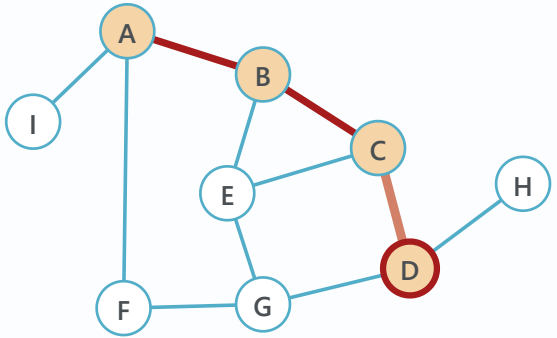
peek = **B**; first unvisited neighbor **C** (break) → mark it and push it.

Code Trace: `dfsStack(A)`

```
1  void dfsStack(int s) {
2    boolean[] visited = new boolean[n];
3    Deque<Integer> stack = new ArrayDeque<>();
4    stack.push(s);  visited[s] = true;
5    while (!stack.isEmpty()) {
▶      int v = stack.peek();
7      Integer next = null;
8      for (int u : adj.get(v))
▶        if (!visited[u]) { next = u; break; }
10     if (next == null) stack.pop(); // back
▶      else { visited[next] = true;
▶             stack.push(next); }
13   }
14 }
```

VARIABLES

`v = C` `next = D`



stack (bottom → top)

A	B	C	D				

visited[]

A	B	C	D	E	F	G	H	I
T	T	T	T	F	F	F	F	F

unseen · on stack · on top · finished

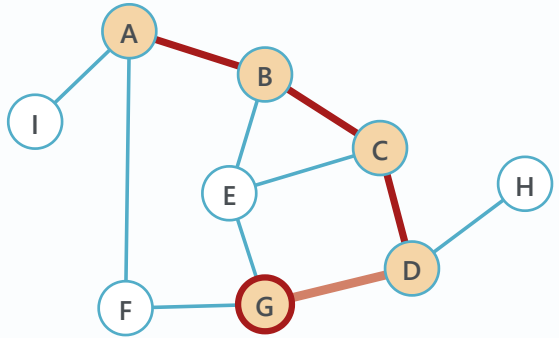
peek = **C**; first unvisited neighbor **D** (break) → mark it and push it.

Code Trace: `dfsStack(A)`

```
1  void dfsStack(int s) {
2    boolean[] visited = new boolean[n];
3    Deque<Integer> stack = new ArrayDeque<>();
4    stack.push(s);  visited[s] = true;
5    while (!stack.isEmpty()) {
▶      int v = stack.peek();
7      Integer next = null;
8      for (int u : adj.get(v))
▶        if (!visited[u]) { next = u; break; }
10     if (next == null) stack.pop(); // back
▶      else { visited[next] = true;
▶             stack.push(next); }
13   }
14 }
```

VARIABLES

`v = D` `next = G`



stack (bottom → top)

A	B	C	D	G			

visited[]

A	B	C	D	E	F	G	H	I
T	T	T	T	F	F	T	F	F

unseen · on stack · on top · finished

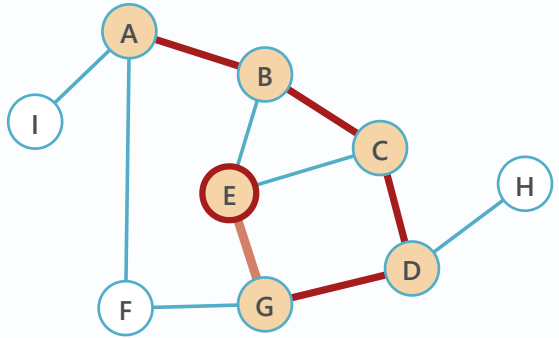
peek = **D**; first unvisited neighbor **G** (break) → mark it and push it.

Code Trace: `dfsStack(A)`

```
1  void dfsStack(int s) {
2    boolean[] visited = new boolean[n];
3    Deque<Integer> stack = new ArrayDeque<>();
4    stack.push(s);  visited[s] = true;
5    while (!stack.isEmpty()) {
▶      int v = stack.peek();
7      Integer next = null;
8      for (int u : adj.get(v))
▶        if (!visited[u]) { next = u; break; }
10     if (next == null) stack.pop(); // back
▶      else { visited[next] = true;
▶             stack.push(next); }
13   }
14 }
```

VARIABLES

`v = G` `next = E`



stack (bottom → top)

A	B	C	D	G	E		

visited[]

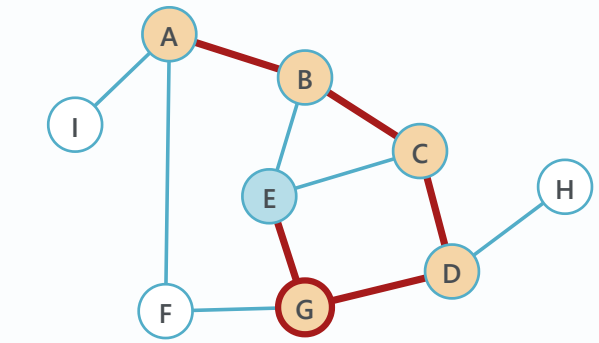
A	B	C	D	E	F	G	H	I
T	T	T	T	T	F	T	F	F

unseen · on stack · on top · finished

peek = **G**; first unvisited neighbor **E** (break) → mark it and push it.

Code Trace: `dfsStack(A)`

```
1  void dfsStack(int s) {
2    boolean[] visited = new boolean[n];
3    Deque<Integer> stack = new ArrayDeque<>();
4    stack.push(s);  visited[s] = true;
5    while (!stack.isEmpty()) {
▶      int v = stack.peek();
7      Integer next = null;
8      for (int u : adj.get(v))
9        if (!visited[u]) { next = u; break; }
▶      if (next == null) stack.pop(); // back
11     else { visited[next] = true;
12            stack.push(next); }
13   }
14 }
```



stack (bottom → top)

A	B	C	D	G			

visited[]

A	B	C	D	E	F	G	H	I
T	T	T	T	T	F	T	F	F

unseen · on stack · on top · finished

VARIABLES

`v = E` `next = null`

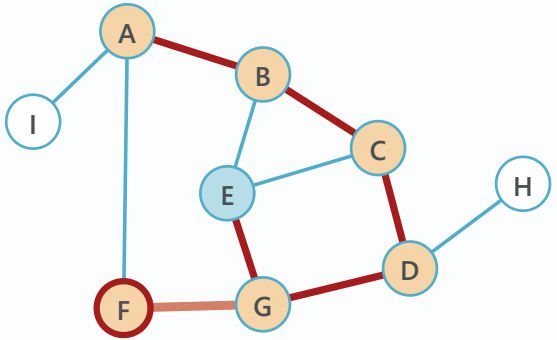
peek = **E**; no unvisited neighbor (next = null) → **pop** and back up.

Code Trace: `dfsStack(A)`

```
1  void dfsStack(int s) {
2    boolean[] visited = new boolean[n];
3    Deque<Integer> stack = new ArrayDeque<>();
4    stack.push(s);  visited[s] = true;
5    while (!stack.isEmpty()) {
▶      int v = stack.peek();
7      Integer next = null;
8      for (int u : adj.get(v))
▶        if (!visited[u]) { next = u; break; }
10     if (next == null) stack.pop(); // back
▶      else { visited[next] = true;
▶             stack.push(next); }
13   }
14 }
```

VARIABLES

`v = G` `next = F`



stack (bottom → top)

A	B	C	D	G	F		

visited[]

A	B	C	D	E	F	G	H	I
T	T	T	T	T	T	T	F	F

unseen · on stack · on top · finished

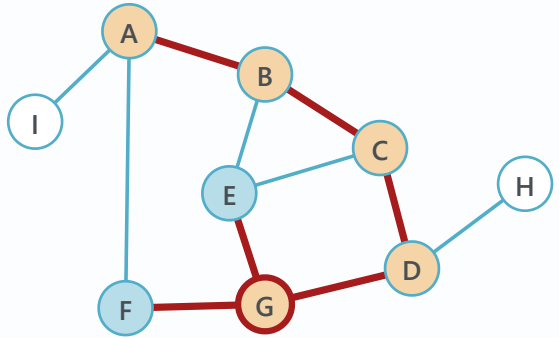
peek = **G**; first unvisited neighbor **F** (break) → mark it and push it.

Code Trace: `dfsStack(A)`

```
1  void dfsStack(int s) {
2    boolean[] visited = new boolean[n];
3    Deque<Integer> stack = new ArrayDeque<>();
4    stack.push(s);  visited[s] = true;
5    while (!stack.isEmpty()) {
▶      int v = stack.peek();
7      Integer next = null;
8      for (int u : adj.get(v))
9        if (!visited[u]) { next = u; break; }
▶      if (next == null) stack.pop(); // back
11     else { visited[next] = true;
12            stack.push(next); }
13   }
14 }
```

VARIABLES

`v = F` `next = null`



stack (bottom → top)

A	B	C	D	G			

visited[]

A	B	C	D	E	F	G	H	I
T	T	T	T	T	T	T	F	F

unseen · on stack · on top · finished

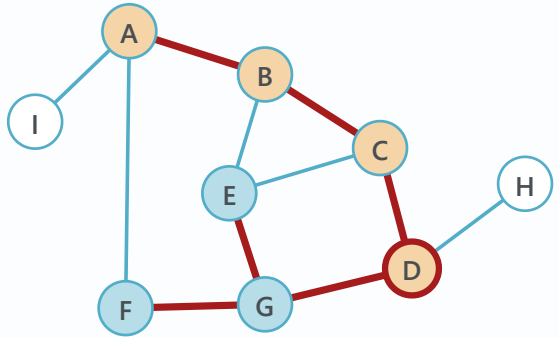
peek = **F**; no unvisited neighbor (next = null) → **pop** and back up.

Code Trace: `dfsStack(A)`

```
1  void dfsStack(int s) {
2    boolean[] visited = new boolean[n];
3    Deque<Integer> stack = new ArrayDeque<>();
4    stack.push(s);  visited[s] = true;
5    while (!stack.isEmpty()) {
▶      int v = stack.peek();
7      Integer next = null;
8      for (int u : adj.get(v))
9        if (!visited[u]) { next = u; break; }
▶      if (next == null) stack.pop(); // back
11     else { visited[next] = true;
12            stack.push(next); }
13   }
14 }
```

VARIABLES

`v = G` `next = null`



stack (bottom → top)

A	B	C	D				

visited[]

A	B	C	D	E	F	G	H	I
T	T	T	T	T	T	T	F	F

unseen · on stack · on top · finished

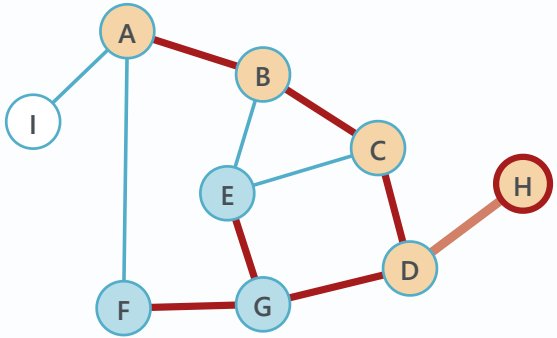
peek = **G**; no unvisited neighbor (next = null) → **pop** and back up.

Code Trace: `dfsStack(A)`

```
1  void dfsStack(int s) {
2    boolean[] visited = new boolean[n];
3    Deque<Integer> stack = new ArrayDeque<>();
4    stack.push(s);  visited[s] = true;
5    while (!stack.isEmpty()) {
▶      int v = stack.peek();
7      Integer next = null;
8      for (int u : adj.get(v))
▶        if (!visited[u]) { next = u; break; }
10     if (next == null) stack.pop(); // back
▶      else { visited[next] = true;
▶             stack.push(next); }
13   }
14 }
```

VARIABLES

`v = D` `next = H`



stack (bottom → top)

A	B	C	D	H			

visited[]

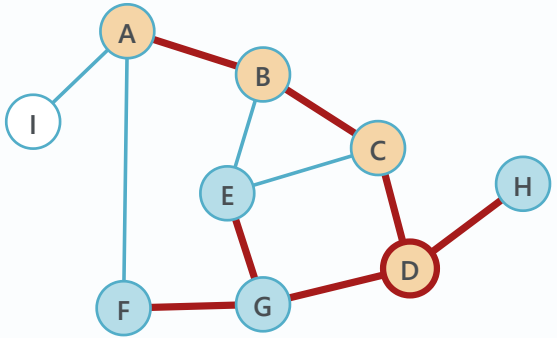
A	B	C	D	E	F	G	H	I
T	T	T	T	T	T	T	T	F

unseen · on stack · on top · finished

peek = **D**; first unvisited neighbor **H** (break) → mark it and push it.

Code Trace: `dfsStack(A)`

```
1  void dfsStack(int s) {
2    boolean[] visited = new boolean[n];
3    Deque<Integer> stack = new ArrayDeque<>();
4    stack.push(s);  visited[s] = true;
5    while (!stack.isEmpty()) {
▶      int v = stack.peek();
7      Integer next = null;
8      for (int u : adj.get(v))
9        if (!visited[u]) { next = u; break; }
▶      if (next == null) stack.pop(); // back
11     else { visited[next] = true;
12            stack.push(next); }
13   }
14 }
```



stack (bottom → top)

A	B	C	D				

visited[]

A	B	C	D	E	F	G	H	I
T	T	T	T	T	T	T	T	F

unseen · on stack · on top · finished

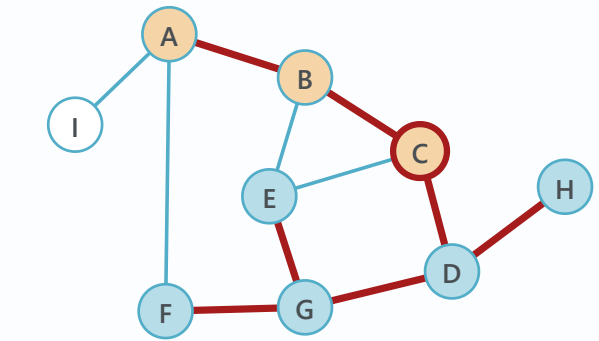
VARIABLES

`v = H` `next = null`

peek = **H**; no unvisited neighbor (next = null) → **pop** and back up.

Code Trace: `dfsStack(A)`

```
1  void dfsStack(int s) {
2    boolean[] visited = new boolean[n];
3    Deque<Integer> stack = new ArrayDeque<>();
4    stack.push(s);  visited[s] = true;
5    while (!stack.isEmpty()) {
▶      int v = stack.peek();
7      Integer next = null;
8      for (int u : adj.get(v))
9        if (!visited[u]) { next = u; break; }
▶      if (next == null) stack.pop(); // back
11     else { visited[next] = true;
12            stack.push(next); }
13   }
14 }
```



stack (bottom → top)

A	B	C					

visited[]

A	B	C	D	E	F	G	H	I
T	T	T	T	T	T	T	T	F

unseen · on stack · on top · finished

VARIABLES

`v = D` `next = null`

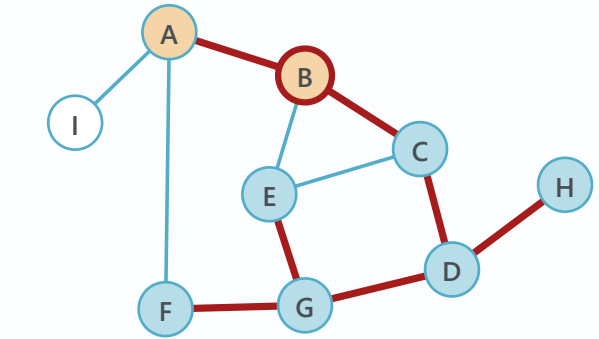
peek = **D**; no unvisited neighbor (next = null) → **pop** and back up.

Code Trace: `dfsStack(A)`

```
1  void dfsStack(int s) {
2    boolean[] visited = new boolean[n];
3    Deque<Integer> stack = new ArrayDeque<>();
4    stack.push(s);  visited[s] = true;
5    while (!stack.isEmpty()) {
▶      int v = stack.peek();
7      Integer next = null;
8      for (int u : adj.get(v))
9        if (!visited[u]) { next = u; break; }
▶      if (next == null) stack.pop(); // back
11     else { visited[next] = true;
12            stack.push(next); }
13   }
14 }
```

VARIABLES

`v = C` `next = null`



stack (bottom → top)

A	B						

visited[]

A	B	C	D	E	F	G	H	I
T	T	T	T	T	T	T	T	F

unseen · on stack · on top · finished

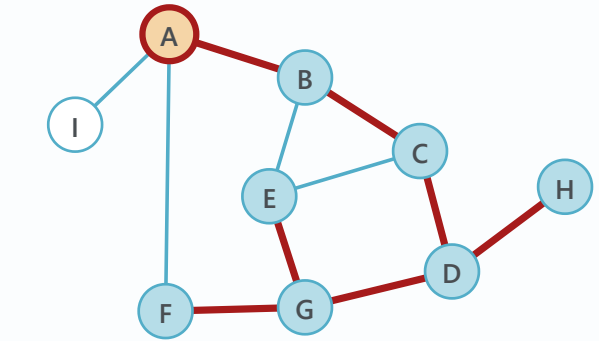
peek = **C**; no unvisited neighbor (next = null) → **pop** and back up.

Code Trace: `dfsStack(A)`

```
1  void dfsStack(int s) {
2    boolean[] visited = new boolean[n];
3    Deque<Integer> stack = new ArrayDeque<>();
4    stack.push(s);  visited[s] = true;
5    while (!stack.isEmpty()) {
▶      int v = stack.peek();
7      Integer next = null;
8      for (int u : adj.get(v))
9        if (!visited[u]) { next = u; break; }
▶      if (next == null) stack.pop(); // back
11     else { visited[next] = true;
12            stack.push(next); }
13   }
14 }
```

VARIABLES

`v = B` `next = null`



stack (bottom → top)

A							

visited[]

A	B	C	D	E	F	G	H	I
T	T	T	T	T	T	T	T	F

unseen · on stack · on top · finished

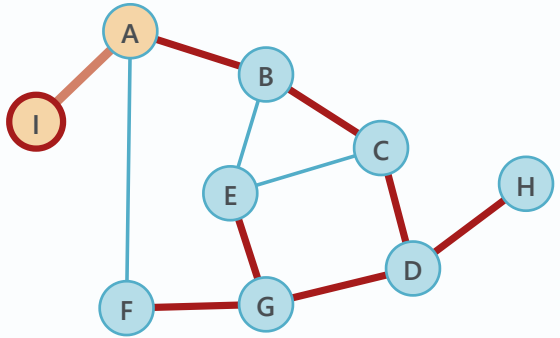
peek = **B**; no unvisited neighbor (next = null) → **pop** and back up.

Code Trace: `dfsStack(A)`

```
1  void dfsStack(int s) {
2    boolean[] visited = new boolean[n];
3    Deque<Integer> stack = new ArrayDeque<>();
4    stack.push(s);  visited[s] = true;
5    while (!stack.isEmpty()) {
▶      int v = stack.peek();
7      Integer next = null;
8      for (int u : adj.get(v))
▶        if (!visited[u]) { next = u; break; }
10     if (next == null) stack.pop(); // back
▶      else { visited[next] = true;
▶             stack.push(next); }
13   }
14 }
```

VARIABLES

`v = A` `next = I`



stack (bottom → top)

A	I						

visited[]

A	B	C	D	E	F	G	H	I
T	T	T	T	T	T	T	T	T

unseen · on stack · on top · finished

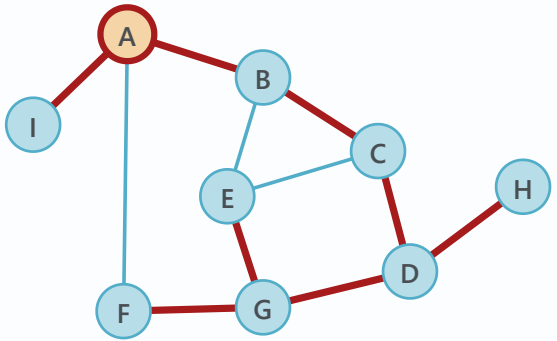
peek = **A**; first unvisited neighbor **I** (break) → mark it and push it.

Code Trace: `dfsStack(A)`

```
1  void dfsStack(int s) {
2    boolean[] visited = new boolean[n];
3    Deque<Integer> stack = new ArrayDeque<>();
4    stack.push(s);  visited[s] = true;
5    while (!stack.isEmpty()) {
▶      int v = stack.peek();
7      Integer next = null;
8      for (int u : adj.get(v))
9        if (!visited[u]) { next = u; break; }
▶      if (next == null) stack.pop(); // back
11     else { visited[next] = true;
12            stack.push(next); }
13   }
14 }
```

VARIABLES

`v = I` `next = null`



stack (bottom → top)

A							

visited[]

A	B	C	D	E	F	G	H	I
T	T	T	T	T	T	T	T	T

unseen · on stack · on top · finished

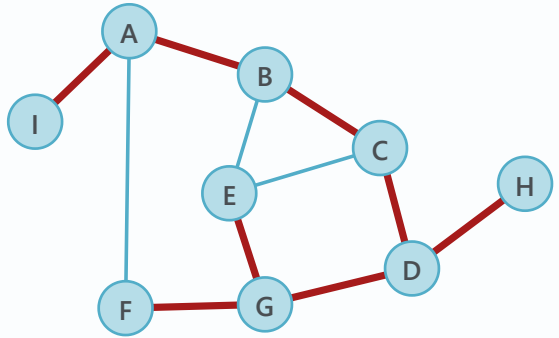
peek = **I**; no unvisited neighbor (next = null) → **pop** and back up.

Code Trace: `dfsStack(A)`

```
1  void dfsStack(int s) {
2    boolean[] visited = new boolean[n];
3    Deque<Integer> stack = new ArrayDeque<>();
4    stack.push(s);  visited[s] = true;
5    while (!stack.isEmpty()) {
▶      int v = stack.peek();
7      Integer next = null;
8      for (int u : adj.get(v))
9        if (!visited[u]) { next = u; break; }
▶      if (next == null) stack.pop(); // back
11     else { visited[next] = true;
12            stack.push(next); }
13   }
14 }
```

VARIABLES

`v = A` `next = null`



stack (bottom → top)

(empty)						

visited[]

A	B	C	D	E	F	G	H	I
T	T	T	T	T	T	T	T	T

unseen · on stack · on top · finished

peek = **A**; no unvisited neighbor (next = null) → **pop** and back up. The stack is empty: done. The visit order matches the recursive version.

BFS vs. DFS

	BFS	DFS
Helper structure	Queue (FIFO)	Stack (LIFO) / recursion
Time (adjacency list)	O(n + m)	O(n + m)
Extra memory	O(n): the widest level is in the queue	O(n): the longest path is on the stack
Shortest paths?	Yes (in unweighted graphs)	No
Typical uses	fewest steps, finding levels	cycle detection, components, mazes

Why O(n + m)? Each vertex is processed once and every adjacency entry is scanned. In an undirected graph each edge appears in two lists, but 2m is still O(m).

The traces assume neighbors are visited in alphabetical order; a different neighbor order gives a different visit order. Because `dfsStack` rescans the neighbor list from the start on every `peek`, it can take longer than O(n + m); remembering where each vertex left off restores O(n + m).

Code Trace: Numbering Components with DFS



Every vertex is **−1** (not yet in any component).

Step 1 / 18

```
1  int components(int[] comp) {
▶    Arrays.fill(comp, -1);
3    int count = 0;
4    for (int s = 0; s < n; s++)
5      if (comp[s] == -1) dfs(s, count++, comp);
6    return count;
7  }
8  void dfs(int v, int c, int[] comp) {
9    comp[v] = c;
10   for (int u : adj.get(v))
11     if (comp[u] == -1) dfs(u, c, comp);
12 }
```

VARIABLES

`comp = [-1, -1, -1, -1, -1, -1]`

CALL STACK

`components(comp)`

`main()`

Code Trace: Numbering Components with DFS



Component counter **count = 0**.

Step 2 / 18

```
1  int components(int[] comp) {
2    Arrays.fill(comp, -1);
▶    int count = 0;
4    for (int s = 0; s < n; s++)
5      if (comp[s] == -1) dfs(s, count++, comp);
6    return count;
7  }
8  void dfs(int v, int c, int[] comp) {
9    comp[v] = c;
10   for (int u : adj.get(v))
11     if (comp[u] == -1) dfs(u, c, comp);
12 }
```

VARIABLES

`count = 0`

`comp = [-1, -1, -1, -1, -1, -1]`

CALL STACK

`components(comp)`

`main()`

Code Trace: Numbering Components with DFS



comp[0] = −1: **new component**. `count++` yields 0 and count becomes 1; call dfs(0, 0).

Step 3 / 18

```
1  int components(int[] comp) {
2    Arrays.fill(comp, -1);
3    int count = 0;
▶    for (int s = 0; s < n; s++)
▶      if (comp[s] == -1) dfs(s, count++, comp);
6    return count;
7  }
8  void dfs(int v, int c, int[] comp) {
9    comp[v] = c;
10   for (int u : adj.get(v))
11     if (comp[u] == -1) dfs(u, c, comp);
12 }
```

VARIABLES

`s = 0` `count = 1`

`comp = [-1, -1, -1, -1, -1, -1]`

CALL STACK

`components(comp)`

`main()`

Code Trace: Numbering Components with DFS



v	0	1	2	3	4	5
comp	0	-1	-1	-1	-1	-1

5

comp[0] = 0: vertex 0 gets component number 0.

Step 4 / 18

```
1  int components(int[] comp) {
2    Arrays.fill(comp, -1);
3    int count = 0;
4    for (int s = 0; s < n; s++)
5      if (comp[s] == -1) dfs(s, count++, comp);
6    return count;
7  }
8  void dfs(int v, int c, int[] comp) {
▶    comp[v] = c;
10   for (int u : adj.get(v))
11     if (comp[u] == -1) dfs(u, c, comp);
12 }
```

VARIABLES

`s = 0` `count = 1` `v = 0` `c = 0`

`comp = [0, -1, -1, -1, -1, -1]`

CALL STACK

- `dfs(0, 0)`
- `components(comp)`
- `main()`

Code Trace: Numbering Components with DFS



comp[1] = 0: vertex 1 gets component number 0.

Step 5 / 18

```
1  int components(int[] comp) {
2    Arrays.fill(comp, -1);
3    int count = 0;
4    for (int s = 0; s < n; s++)
5      if (comp[s] == -1) dfs(s, count++, comp);
6    return count;
7  }
8  void dfs(int v, int c, int[] comp) {
▶    comp[v] = c;
10   for (int u : adj.get(v))
11     if (comp[u] == -1) dfs(u, c, comp);
12 }
```

VARIABLES

`s = 0` `count = 1` `v = 1` `c = 0`

`comp = [0, 0, -1, -1, -1, -1]`

CALL STACK

- **dfs(1, 0)**
- dfs(0, 0)
- components(comp)
- main()

Code Trace: Numbering Components with DFS



Neighbor 0 is already labeled (comp = 0) → skip. Each edge is examined twice: O(n + m).

Step 6 / 18

```
1  int components(int[] comp) {
2    Arrays.fill(comp, -1);
3    int count = 0;
4    for (int s = 0; s < n; s++)
5      if (comp[s] == -1) dfs(s, count++, comp);
6    return count;
7  }
8  void dfs(int v, int c, int[] comp) {
9    comp[v] = c;
▶    for (int u : adj.get(v))
▶      if (comp[u] == -1) dfs(u, c, comp);
12 }
```

VARIABLES

`s = 0` `count = 1` `v = 1` `c = 0` `u = 0`

`comp = [0, 0, -1, -1, -1, -1]`

CALL STACK

- **dfs(1, 0)**
- dfs(0, 0)
- components(comp)
- main()

Code Trace: Numbering Components with DFS



comp[2] = 0: vertex 2 gets component number 0.

Step 7 / 18

```
1  int components(int[] comp) {
2    Arrays.fill(comp, -1);
3    int count = 0;
4    for (int s = 0; s < n; s++)
5      if (comp[s] == -1) dfs(s, count++, comp);
6    return count;
7  }
8  void dfs(int v, int c, int[] comp) {
▶    comp[v] = c;
10   for (int u : adj.get(v))
11     if (comp[u] == -1) dfs(u, c, comp);
12 }
```

VARIABLES

`s = 0` `count = 1` `v = 2` `c = 0`

`comp = [0, 0, 0, -1, -1, -1]`

CALL STACK

- **dfs(2, 0)**
- dfs(0, 0)
- components(comp)
- main()

Code Trace: Numbering Components with DFS



Neighbor 0 is already labeled (comp = 0) → skip. Each edge is examined twice: O(n + m).

Step 8 / 18

```
1  int components(int[] comp) {
2    Arrays.fill(comp, -1);
3    int count = 0;
4    for (int s = 0; s < n; s++)
5      if (comp[s] == -1) dfs(s, count++, comp);
6    return count;
7  }
8  void dfs(int v, int c, int[] comp) {
9    comp[v] = c;
▶    for (int u : adj.get(v))
▶      if (comp[u] == -1) dfs(u, c, comp);
12 }
```

VARIABLES

`s = 0` `count = 1` `v = 2` `c = 0` `u = 0`

`comp = [0, 0, 0, -1, -1, -1]`

CALL STACK

- **dfs(2, 0)**
- dfs(0, 0)
- components(comp)
- main()

Code Trace: Numbering Components with DFS



comp[1] = 0: vertex 1 was labeled by an earlier DFS → skip.

Step 9 / 18

```
1  int components(int[] comp) {
2    Arrays.fill(comp, -1);
3    int count = 0;
▶    for (int s = 0; s < n; s++)
▶      if (comp[s] == -1) dfs(s, count++, comp);
6    return count;
7  }
8  void dfs(int v, int c, int[] comp) {
9    comp[v] = c;
10   for (int u : adj.get(v))
11     if (comp[u] == -1) dfs(u, c, comp);
12 }
```

VARIABLES

`s = 1` `count = 1`

`comp = [0, 0, 0, -1, -1, -1]`

CALL STACK

`components(comp)`

`main()`

Code Trace: Numbering Components with DFS



comp[2] = 0: vertex 2 was labeled by an earlier DFS → skip.

Step 10 / 18

```
1  int components(int[] comp) {
2    Arrays.fill(comp, -1);
3    int count = 0;
▶    for (int s = 0; s < n; s++)
▶      if (comp[s] == -1) dfs(s, count++, comp);
6    return count;
7  }
8  void dfs(int v, int c, int[] comp) {
9    comp[v] = c;
10   for (int u : adj.get(v))
11     if (comp[u] == -1) dfs(u, c, comp);
12 }
```

VARIABLES

`s = 2` `count = 1`

`comp = [0, 0, 0, -1, -1, -1]`

CALL STACK

`components(comp)`

`main()`

Code Trace: Numbering Components with DFS



comp[3] = −1: **new component**. `count++` yields 1 and count becomes 2; call dfs(3, 1).

Step 11 / 18

```
1  int components(int[] comp) {
2    Arrays.fill(comp, -1);
3    int count = 0;
▶    for (int s = 0; s < n; s++)
▶      if (comp[s] == -1) dfs(s, count++, comp);
6    return count;
7  }
8  void dfs(int v, int c, int[] comp) {
9    comp[v] = c;
10   for (int u : adj.get(v))
11     if (comp[u] == -1) dfs(u, c, comp);
12 }
```

VARIABLES

`s = 3` `count = 2`

`comp = [0, 0, 0, -1, -1, -1]`

CALL STACK

- `components(comp)`
- `main()`

Code Trace: Numbering Components with DFS



comp[3] = 1: vertex 3 gets component number 1.

Step 12 / 18

```
1  int components(int[] comp) {
2    Arrays.fill(comp, -1);
3    int count = 0;
4    for (int s = 0; s < n; s++)
5      if (comp[s] == -1) dfs(s, count++, comp);
6    return count;
7  }
8  void dfs(int v, int c, int[] comp) {
▶    comp[v] = c;
10   for (int u : adj.get(v))
11     if (comp[u] == -1) dfs(u, c, comp);
12 }
```

VARIABLES

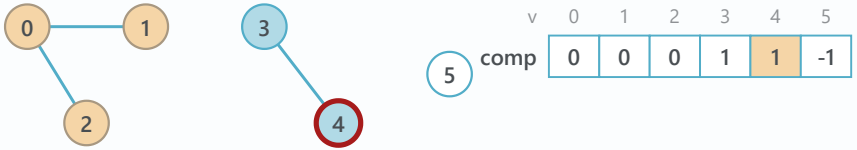
`s = 3` `count = 2` `v = 3` `c = 1`

`comp = [0, 0, 0, 1, -1, -1]`

CALL STACK

- **dfs(3, 1)**
- components(comp)
- main()

Code Trace: Numbering Components with DFS



comp[4] = 1: vertex 4 gets component number 1.

Step 13 / 18

```
1  int components(int[] comp) {
2    Arrays.fill(comp, -1);
3    int count = 0;
4    for (int s = 0; s < n; s++)
5      if (comp[s] == -1) dfs(s, count++, comp);
6    return count;
7  }
8  void dfs(int v, int c, int[] comp) {
▶    comp[v] = c;
10   for (int u : adj.get(v))
11     if (comp[u] == -1) dfs(u, c, comp);
12 }
```

VARIABLES

`s = 3` `count = 2` `v = 4` `c = 1`

`comp = [0, 0, 0, 1, 1, -1]`

CALL STACK

- **dfs(4, 1)**
- dfs(3, 1)
- components(comp)
- main()

Code Trace: Numbering Components with DFS



Neighbor 3 is already labeled (comp = 1) → skip. Each edge is examined twice: O(n + m).

Step 14 / 18

```
1  int components(int[] comp) {
2    Arrays.fill(comp, -1);
3    int count = 0;
4    for (int s = 0; s < n; s++)
5      if (comp[s] == -1) dfs(s, count++, comp);
6    return count;
7  }
8  void dfs(int v, int c, int[] comp) {
9    comp[v] = c;
▶    for (int u : adj.get(v))
▶      if (comp[u] == -1) dfs(u, c, comp);
12 }
```

VARIABLES

`s = 3` `count = 2` `v = 4` `c = 1`

`u = 3` `comp = [0, 0, 0, 1, 1, -1]`

CALL STACK

- **dfs(4, 1)**
- dfs(3, 1)
- components(comp)
- main()

Code Trace: Numbering Components with DFS



comp[4] = 1: vertex 4 was labeled by an earlier DFS → skip.

Step 15 / 18

```
1  int components(int[] comp) {
2    Arrays.fill(comp, -1);
3    int count = 0;
▶    for (int s = 0; s < n; s++)
▶      if (comp[s] == -1) dfs(s, count++, comp);
6    return count;
7  }
8  void dfs(int v, int c, int[] comp) {
9    comp[v] = c;
10   for (int u : adj.get(v))
11     if (comp[u] == -1) dfs(u, c, comp);
12 }
```

VARIABLES

`s = 4` `count = 2`

`comp = [0, 0, 0, 1, 1, -1]`

CALL STACK

`components(comp)`

`main()`

Code Trace: Numbering Components with DFS



comp[5] = −1: **new component**. `count++` yields 2 and count becomes 3; call dfs(5, 2).

Step 16 / 18

```
1  int components(int[] comp) {
2    Arrays.fill(comp, -1);
3    int count = 0;
▶    for (int s = 0; s < n; s++)
▶      if (comp[s] == -1) dfs(s, count++, comp);
6    return count;
7  }
8  void dfs(int v, int c, int[] comp) {
9    comp[v] = c;
10   for (int u : adj.get(v))
11     if (comp[u] == -1) dfs(u, c, comp);
12 }
```

VARIABLES

`s = 5` `count = 3`

`comp = [0, 0, 0, 1, 1, -1]`

CALL STACK

- **components(comp)**
- main()

Code Trace: Numbering Components with DFS



comp[5] = 2: vertex 5 gets component number 2.

Step 17 / 18

```
1  int components(int[] comp) {
2    Arrays.fill(comp, -1);
3    int count = 0;
4    for (int s = 0; s < n; s++)
5      if (comp[s] == -1) dfs(s, count++, comp);
6    return count;
7  }
8  void dfs(int v, int c, int[] comp) {
▶    comp[v] = c;
10   for (int u : adj.get(v))
11     if (comp[u] == -1) dfs(u, c, comp);
12 }
```

VARIABLES

`s = 5` `count = 3` `v = 5` `c = 2`

`comp = [0, 0, 0, 1, 1, 2]`

CALL STACK

- `dfs(5, 2)`
- `components(comp)`
- `main()`

Code Trace: Numbering Components with DFS



3 components: {0, 1, 2}, {3, 4}, {5}. Same comp = same component; the query `comp[u] == comp[v]` is O(1).

Step 18 / 18

```
1  int components(int[] comp) {
2    Arrays.fill(comp, -1);
3    int count = 0;
4    for (int s = 0; s < n; s++)
5      if (comp[s] == -1) dfs(s, count++, comp);
▶    return count;
7  }
8  void dfs(int v, int c, int[] comp) {
9    comp[v] = c;
10   for (int u : adj.get(v))
11     if (comp[u] == -1) dfs(u, c, comp);
12 }
```

VARIABLES

`count = 3` `return = 3`

`comp = [0, 0, 0, 1, 1, 2]`

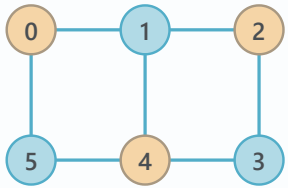
CALL STACK

`components(comp)`

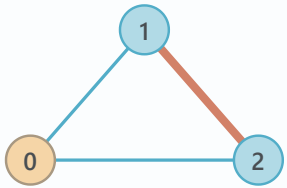
`main()`

Testing Whether a Graph Is Bipartite

- **Bipartite:** the vertices split into two groups, and every edge runs *between* the groups.
- Test = **2-coloring with BFS**: give each neighbor the opposite color; two adjacent vertices of the same color → not bipartite.
- Equivalently: **no odd-length cycle**.



bipartite ✓ (even cycles)



triangle ✗: 1 and 2 same color

```
// color[] starts at -1; BFS from s
color[s] = 0;  q.add(s);
while (!q.isEmpty()) {
  int w = q.poll();
  for (int u : adj.get(w))
    if (color[u] == -1) {
      color[u] = 1 - color[w];  q.add(u);
    } else if (color[u] == color[w]) return false;
}
return true;
```

Same cost as BFS: O(n + m). Uses: matching students to projects, two-shift scheduling, conflict-free grouping.

Topological Sorting: DAGs Only

- Orders the vertices of a digraph so that for every edge `u → v`, u comes before v.
- Possible only in **directed acyclic graphs** (DAGs).
- The result need not be unique: several vertices of in-degree 0 may be available at the same time.
- Uses: course prerequisites, task dependencies, build and workflow planning.

A graph with a cycle has no topological order: every task on the cycle waits for another task on it to be done first.

```
Kahn(G):
  q = vertices with inDegree 0
  result = []
  while q is not empty:
    u = remove from q
    append u to result
    for each edge (u, v) leaving u:
      inDegree[v]--
      if inDegree[v] == 0:
        add v to q
  if result.size < n:
    there is a cycle
  return result
```

Time

O(n + m)

Alternative

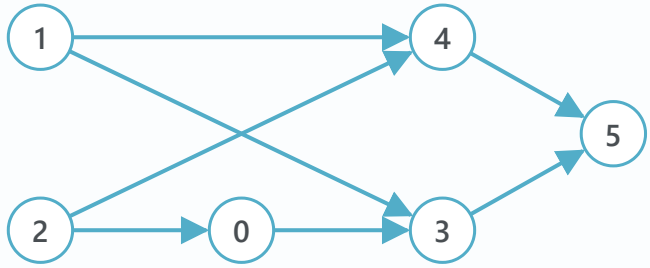
Reverse the DFS finishing order; finding a back edge means there is a cycle.

Code Trace: Topological Sorting (Kahn)

```
1  List<Integer> kahn(List<List<Integer>> adj)
   {
▶    int n = adj.size();
▶    int[] indeg = new int[n];
4    for (List<Integer> l : adj)
5      for (int v : l) indeg[v]++;
6    Queue<Integer> q = new ArrayDeque<>();
7    for (int v = 0; v < n; v++)
8      if (indeg[v] == 0) q.add(v);
9    List<Integer> result = new ArrayList<>();
10   while (!q.isEmpty()) {
11     int u = q.poll();
12     result.add(u);
13     for (int v : adj.get(u))
14       if (--indeg[v] == 0) q.add(v);
15   }
16   return result;  // size < n → cycle
17 }
```

VARIABLES

`n = 6`



indeg[] (in-degree)

	0	1	2	3	4	5
indeg	0	0	0	0	0	0

q	(empty)					
result	[ ]					

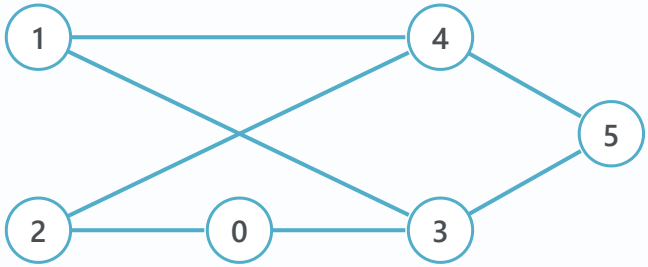
A course-prerequisite example: 6 vertices, 7 directed edges (u → v: u before v). `indeg[]` starts at all zeros.

Code Trace: Topological Sorting (Kahn)

```
1  List<Integer> kahn(List<List<Integer>> adj)
   {
2    int n = adj.size();
3    int[] indeg = new int[n];
▶    for (List<Integer> l : adj)
▶      for (int v : l) indeg[v]++;
6    Queue<Integer> q = new ArrayDeque<>();
7    for (int v = 0; v < n; v++)
8      if (indeg[v] == 0) q.add(v);
9    List<Integer> result = new ArrayList<>();
10   while (!q.isEmpty()) {
11     int u = q.poll();
12     result.add(u);
13     for (int v : adj.get(u))
14       if (--indeg[v] == 0) q.add(v);
15   }
16   return result;  // size < n → cycle
17 }
```

VARIABLES

`n = 6` `l = adj[0]`



indeg[] (in-degree)

	0	1	2	3	4	5
indeg	0	0	0	1	0	0

q: (empty)

result: [ ]

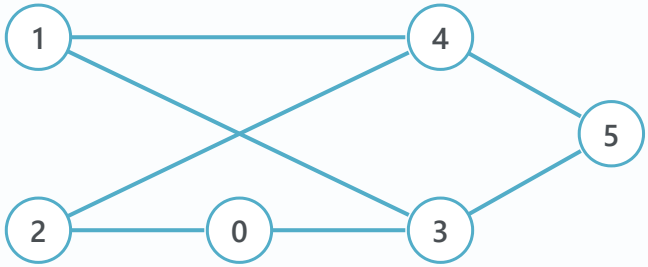
Edges out of 0 (0→3): each target's in-degree goes up by one.

Code Trace: Topological Sorting (Kahn)

```
1  List<Integer> kahn(List<List<Integer>> adj)
   {
2    int n = adj.size();
3    int[] indeg = new int[n];
▶    for (List<Integer> l : adj)
▶      for (int v : l) indeg[v]++;
6    Queue<Integer> q = new ArrayDeque<>();
7    for (int v = 0; v < n; v++)
8      if (indeg[v] == 0) q.add(v);
9    List<Integer> result = new ArrayList<>();
10   while (!q.isEmpty()) {
11     int u = q.poll();
12     result.add(u);
13     for (int v : adj.get(u))
14       if (--indeg[v] == 0) q.add(v);
15   }
16   return result;  // size < n → cycle
17 }
```

VARIABLES

`n = 6` `l = adj[1]`



indeg[] (in-degree)

	0	1	2	3	4	5
indeg	0	0	0	2	1	0

q: (empty)

result: [ ]

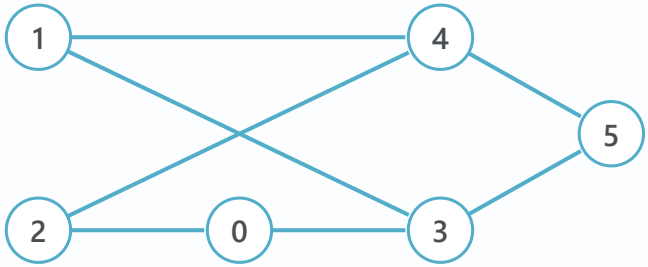
Edges out of 1 (1→3, 1→4): each target's in-degree goes up by one.

Code Trace: Topological Sorting (Kahn)

```
1  List<Integer> kahn(List<List<Integer>> adj)
   {
2    int n = adj.size();
3    int[] indeg = new int[n];
▶    for (List<Integer> l : adj)
▶      for (int v : l) indeg[v]++;
6    Queue<Integer> q = new ArrayDeque<>();
7    for (int v = 0; v < n; v++)
8      if (indeg[v] == 0) q.add(v);
9    List<Integer> result = new ArrayList<>();
10   while (!q.isEmpty()) {
11     int u = q.poll();
12     result.add(u);
13     for (int v : adj.get(u))
14       if (--indeg[v] == 0) q.add(v);
15   }
16   return result;  // size < n → cycle
17 }
```

VARIABLES

`n = 6` `l = adj[2]`



indeg[] (in-degree)

	0	1	2	3	4	5
indeg	1	0	0	2	2	0

q	(empty)					
result	[ ]					

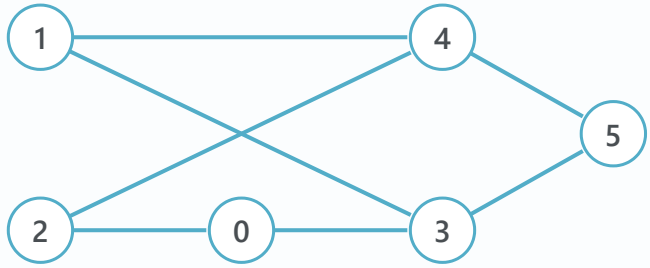
Edges out of 2 (2→0, 2→4): each target's in-degree goes up by one.

Code Trace: Topological Sorting (Kahn)

```
1  List<Integer> kahn(List<List<Integer>> adj)
   {
2    int n = adj.size();
3    int[] indeg = new int[n];
▶    for (List<Integer> l : adj)
▶      for (int v : l) indeg[v]++;
6    Queue<Integer> q = new ArrayDeque<>();
7    for (int v = 0; v < n; v++)
8      if (indeg[v] == 0) q.add(v);
9    List<Integer> result = new ArrayList<>();
10   while (!q.isEmpty()) {
11     int u = q.poll();
12     result.add(u);
13     for (int v : adj.get(u))
14       if (--indeg[v] == 0) q.add(v);
15   }
16   return result;  // size < n → cycle
17 }
```

VARIABLES

`n = 6` `l = adj[3]`



indeg[] (in-degree)

	0	1	2	3	4	5
indeg	1	0	0	2	2	1

q	(empty)					
result	[ ]					

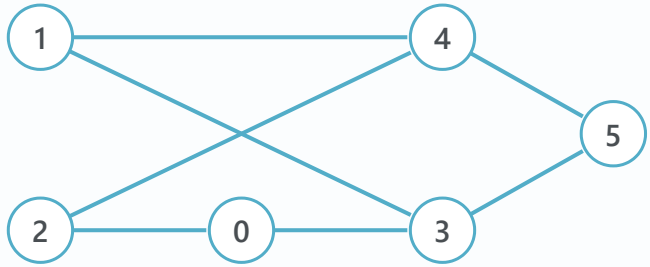
Edges out of 3 (3→5): each target's in-degree goes up by one.

Code Trace: Topological Sorting (Kahn)

```
1  List<Integer> kahn(List<List<Integer>> adj)
   {
2    int n = adj.size();
3    int[] indeg = new int[n];
▶    for (List<Integer> l : adj)
▶      for (int v : l) indeg[v]++;
6    Queue<Integer> q = new ArrayDeque<>();
7    for (int v = 0; v < n; v++)
8      if (indeg[v] == 0) q.add(v);
9    List<Integer> result = new ArrayList<>();
10   while (!q.isEmpty()) {
11     int u = q.poll();
12     result.add(u);
13     for (int v : adj.get(u))
14       if (--indeg[v] == 0) q.add(v);
15   }
16   return result;  // size < n → cycle
17 }
```

VARIABLES

`n = 6` `l = adj[4]`



indeg[] (in-degree)

	0	1	2	3	4	5
indeg	1	0	0	2	2	2

q: (empty)

result: [ ]

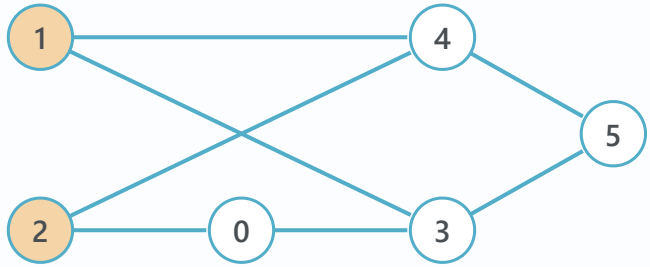
Edges out of 4 (4→5): each target's in-degree goes up by one.

Code Trace: Topological Sorting (Kahn)

```
1  List<Integer> kahn(List<List<Integer>> adj)
   {
2    int n = adj.size();
3    int[] indeg = new int[n];
4    for (List<Integer> l : adj)
5      for (int v : l) indeg[v]++;
▶    Queue<Integer> q = new ArrayDeque<>();
▶    for (int v = 0; v < n; v++)
▶      if (indeg[v] == 0) q.add(v);
▶    List<Integer> result = new ArrayList<>();
10   while (!q.isEmpty()) {
11     int u = q.poll();
12     result.add(u);
13     for (int v : adj.get(u))
14       if (--indeg[v] == 0) q.add(v);
15   }
16   return result;  // size < n → cycle
17 }
```

VARIABLES

`n = 6`



indeg[] (in-degree)

	0	1	2	3	4	5
indeg	1	0	0	2	2	2

q	1	2				
result	[ ]					

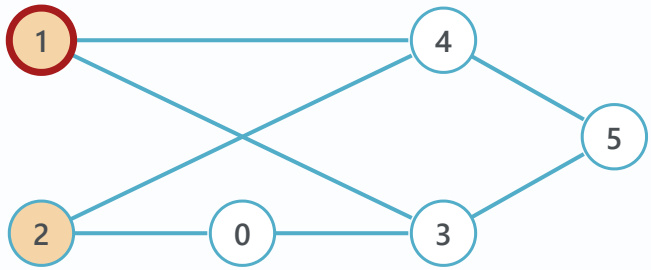
The vertices of in-degree 0 (1, 2) depend on nothing → enqueue them. (5's list is empty, so its counting round changed nothing.)

Code Trace: Topological Sorting (Kahn)

```
1  List<Integer> kahn(List<List<Integer>> adj)
   {
2    int n = adj.size();
3    int[] indeg = new int[n];
4    for (List<Integer> l : adj)
5      for (int v : l) indeg[v]++;
6    Queue<Integer> q = new ArrayDeque<>();
7    for (int v = 0; v < n; v++)
8      if (indeg[v] == 0) q.add(v);
9    List<Integer> result = new ArrayList<>();
10   while (!q.isEmpty()) {
▶      int u = q.poll();
▶      result.add(u);
13     for (int v : adj.get(u))
14       if (--indeg[v] == 0) q.add(v);
15   }
16   return result;  // size < n → cycle
17 }
```

VARIABLES

`u = 1`



indeg[] (in-degree)

	0	1	2	3	4	5
indeg	1	0	0	2	2	2

q	2					
result	1					

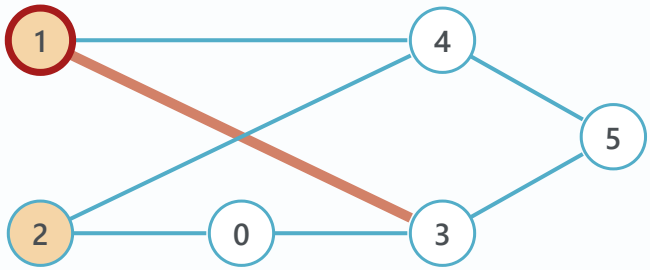
1 leaves the queue and joins the result: all its prerequisites are already there.

Code Trace: Topological Sorting (Kahn)

```
1  List<Integer> kahn(List<List<Integer>> adj)
   {
2    int n = adj.size();
3    int[] indeg = new int[n];
4    for (List<Integer> l : adj)
5      for (int v : l) indeg[v]++;
6    Queue<Integer> q = new ArrayDeque<>();
7    for (int v = 0; v < n; v++)
8      if (indeg[v] == 0) q.add(v);
9    List<Integer> result = new ArrayList<>();
10   while (!q.isEmpty()) {
11     int u = q.poll();
12     result.add(u);
13     for (int v : adj.get(u))
▶        if (--indeg[v] == 0) q.add(v);
15   }
16   return result;  // size < n → cycle
17 }
```

VARIABLES

`u = 1` `v = 3`



indeg[] (in-degree)

	0	1	2	3	4	5
indeg	1	0	0	1	2	2

q						
	2					

result						
	1					

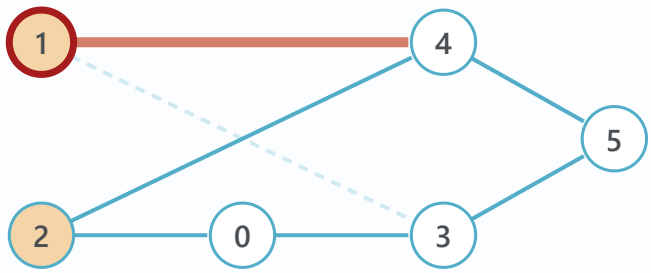
Count 1→3 as deleted: indeg[3] = 1; 3 still waits for another prerequisite.

Code Trace: Topological Sorting (Kahn)

```
1  List<Integer> kahn(List<List<Integer>> adj)
   {
2    int n = adj.size();
3    int[] indeg = new int[n];
4    for (List<Integer> l : adj)
5      for (int v : l) indeg[v]++;
6    Queue<Integer> q = new ArrayDeque<>();
7    for (int v = 0; v < n; v++)
8      if (indeg[v] == 0) q.add(v);
9    List<Integer> result = new ArrayList<>();
10   while (!q.isEmpty()) {
11     int u = q.poll();
12     result.add(u);
13     for (int v : adj.get(u))
▶        if (--indeg[v] == 0) q.add(v);
15   }
16   return result;  // size < n → cycle
17 }
```

VARIABLES

`u = 1` `v = 4`



indeg[] (in-degree)

	0	1	2	3	4	5
indeg	1	0	0	1	1	2

q	2					

result	1					

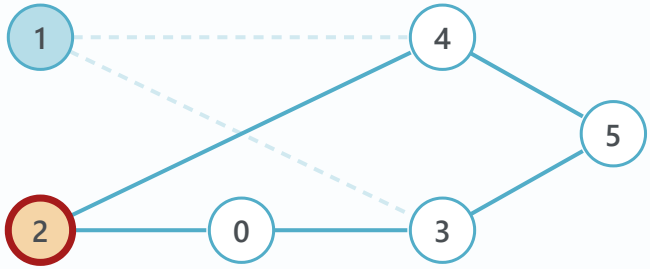
Count 1→4 as deleted: indeg[4] = 1; 4 still waits for another prerequisite.

Code Trace: Topological Sorting (Kahn)

```
1  List<Integer> kahn(List<List<Integer>> adj)
   {
2    int n = adj.size();
3    int[] indeg = new int[n];
4    for (List<Integer> l : adj)
5      for (int v : l) indeg[v]++;
6    Queue<Integer> q = new ArrayDeque<>();
7    for (int v = 0; v < n; v++)
8      if (indeg[v] == 0) q.add(v);
9    List<Integer> result = new ArrayList<>();
10   while (!q.isEmpty()) {
▶      int u = q.poll();
▶      result.add(u);
13     for (int v : adj.get(u))
14       if (--indeg[v] == 0) q.add(v);
15   }
16   return result;  // size < n → cycle
17 }
```

VARIABLES

`u = 2`



indeg[] (in-degree)

	0	1	2	3	4	5
indeg	1	0	0	1	1	2

q	(empty)					
result	1	2				

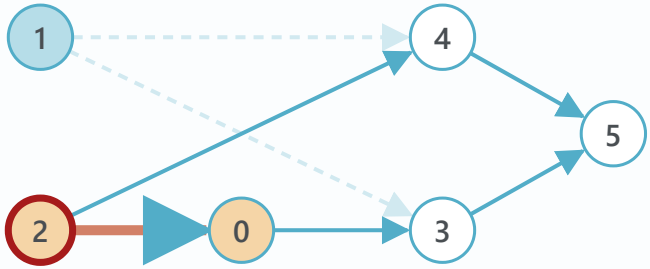
2 leaves the queue and joins the result: all its prerequisites are already there.

Code Trace: Topological Sorting (Kahn, continued)

```
1  List<Integer> kahn(List<List<Integer>> adj)
   {
2    int n = adj.size();
3    int[] indeg = new int[n];
4    for (List<Integer> l : adj)
5      for (int v : l) indeg[v]++;
6    Queue<Integer> q = new ArrayDeque<>();
7    for (int v = 0; v < n; v++)
8      if (indeg[v] == 0) q.add(v);
9    List<Integer> result = new ArrayList<>();
10   while (!q.isEmpty()) {
11     int u = q.poll();
12     result.add(u);
13     for (int v : adj.get(u))
▶        if (--indeg[v] == 0) q.add(v);
15   }
16   return result;  // size < n → cycle
17 }
```

VARIABLES

`u = 2` `v = 0`



indeg[] (in-degree)

	0	1	2	3	4	5
indeg	0	0	0	1	1	2

q						
	0					

result						
	1	2				

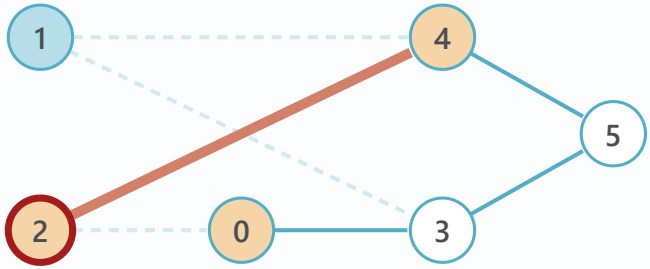
Count 2→0 as deleted: indeg[0] = **0** → 0 is ready and joins the queue.

Code Trace: Topological Sorting (Kahn, continued)

```
1  List<Integer> kahn(List<List<Integer>> adj)
   {
2    int n = adj.size();
3    int[] indeg = new int[n];
4    for (List<Integer> l : adj)
5      for (int v : l) indeg[v]++;
6    Queue<Integer> q = new ArrayDeque<>();
7    for (int v = 0; v < n; v++)
8      if (indeg[v] == 0) q.add(v);
9    List<Integer> result = new ArrayList<>();
10   while (!q.isEmpty()) {
11     int u = q.poll();
12     result.add(u);
13     for (int v : adj.get(u))
▶        if (--indeg[v] == 0) q.add(v);
15   }
16   return result;  // size < n → cycle
17 }
```

VARIABLES

`u = 2` `v = 4`



indeg[] (in-degree)

	0	1	2	3	4	5
indeg	0	0	0	1	0	2

q						
	0	4				

result						
	1	2				

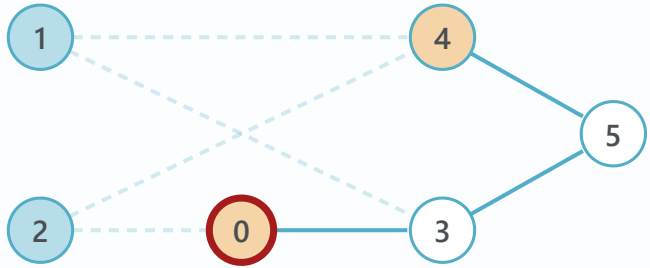
Count 2→4 as deleted: indeg[4] = **0** → 4 is ready and joins the queue.

Code Trace: Topological Sorting (Kahn, continued)

```
1  List<Integer> kahn(List<List<Integer>> adj)
   {
2    int n = adj.size();
3    int[] indeg = new int[n];
4    for (List<Integer> l : adj)
5      for (int v : l) indeg[v]++;
6    Queue<Integer> q = new ArrayDeque<>();
7    for (int v = 0; v < n; v++)
8      if (indeg[v] == 0) q.add(v);
9    List<Integer> result = new ArrayList<>();
10   while (!q.isEmpty()) {
▶      int u = q.poll();
▶      result.add(u);
13     for (int v : adj.get(u))
14       if (--indeg[v] == 0) q.add(v);
15   }
16   return result;  // size < n → cycle
17 }
```

VARIABLES

`u = 0`



indeg[] (in-degree)

	0	1	2	3	4	5
indeg	0	0	0	1	0	2

q	4					
result	1	2	0			

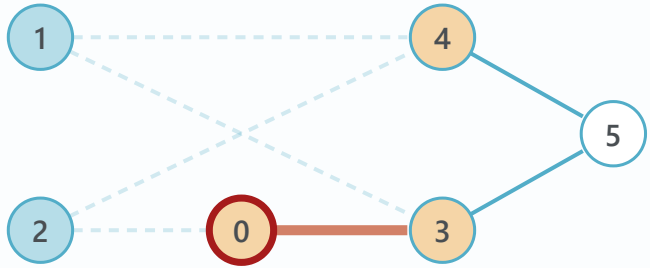
0 leaves the queue and joins the result: all its prerequisites are already there.

Code Trace: Topological Sorting (Kahn, continued)

```
1  List<Integer> kahn(List<List<Integer>> adj)
   {
2    int n = adj.size();
3    int[] indeg = new int[n];
4    for (List<Integer> l : adj)
5      for (int v : l) indeg[v]++;
6    Queue<Integer> q = new ArrayDeque<>();
7    for (int v = 0; v < n; v++)
8      if (indeg[v] == 0) q.add(v);
9    List<Integer> result = new ArrayList<>();
10   while (!q.isEmpty()) {
11     int u = q.poll();
12     result.add(u);
13     for (int v : adj.get(u))
▶        if (--indeg[v] == 0) q.add(v);
15   }
16   return result;  // size < n → cycle
17 }
```

VARIABLES

`u = 0` `v = 3`



indeg[] (in-degree)

	0	1	2	3	4	5
indeg	0	0	0	0	0	2

q	4	3				
result	1	2	0			

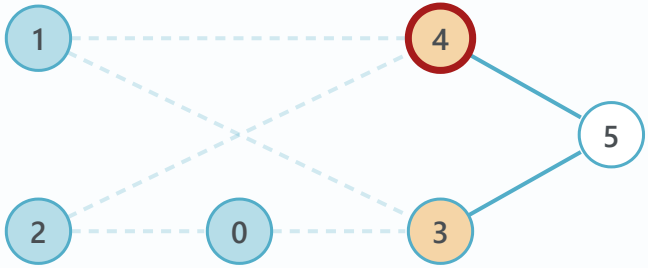
Count 0→3 as deleted: indeg[3] = **0** → 3 is ready and joins the queue.

Code Trace: Topological Sorting (Kahn, continued)

```
1  List<Integer> kahn(List<List<Integer>> adj)
   {
2    int n = adj.size();
3    int[] indeg = new int[n];
4    for (List<Integer> l : adj)
5      for (int v : l) indeg[v]++;
6    Queue<Integer> q = new ArrayDeque<>();
7    for (int v = 0; v < n; v++)
8      if (indeg[v] == 0) q.add(v);
9    List<Integer> result = new ArrayList<>();
10   while (!q.isEmpty()) {
▶      int u = q.poll();
▶      result.add(u);
13     for (int v : adj.get(u))
14       if (--indeg[v] == 0) q.add(v);
15   }
16   return result;  // size < n → cycle
17 }
```

VARIABLES

`u = 4`



indeg[] (in-degree)

	0	1	2	3	4	5
indeg	0	0	0	0	0	2

q						
	3					

result						
	1	2	0	4		

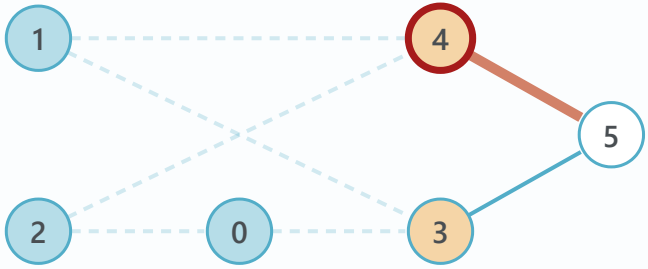
4 leaves the queue and joins the result: all its prerequisites are already there.

Code Trace: Topological Sorting (Kahn, continued)

```
1  List<Integer> kahn(List<List<Integer>> adj)
   {
2    int n = adj.size();
3    int[] indeg = new int[n];
4    for (List<Integer> l : adj)
5      for (int v : l) indeg[v]++;
6    Queue<Integer> q = new ArrayDeque<>();
7    for (int v = 0; v < n; v++)
8      if (indeg[v] == 0) q.add(v);
9    List<Integer> result = new ArrayList<>();
10   while (!q.isEmpty()) {
11     int u = q.poll();
12     result.add(u);
13     for (int v : adj.get(u))
▶        if (--indeg[v] == 0) q.add(v);
15   }
16   return result;  // size < n → cycle
17 }
```

VARIABLES

`u = 4` `v = 5`



indeg[] (in-degree)

	0	1	2	3	4	5
indeg	0	0	0	0	0	1

q						
	3					

result						
	1	2	0	4		

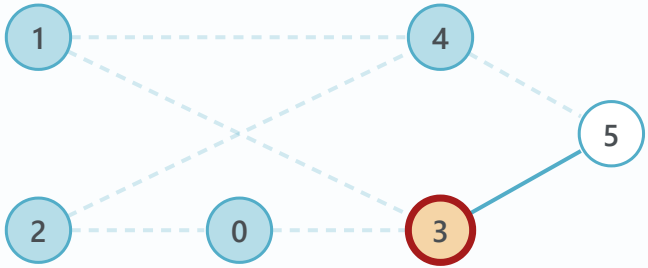
Count 4→5 as deleted: indeg[5] = 1; 5 still waits for another prerequisite.

Code Trace: Topological Sorting (Kahn, continued)

```
1  List<Integer> kahn(List<List<Integer>> adj)
   {
2    int n = adj.size();
3    int[] indeg = new int[n];
4    for (List<Integer> l : adj)
5      for (int v : l) indeg[v]++;
6    Queue<Integer> q = new ArrayDeque<>();
7    for (int v = 0; v < n; v++)
8      if (indeg[v] == 0) q.add(v);
9    List<Integer> result = new ArrayList<>();
10   while (!q.isEmpty()) {
▶      int u = q.poll();
▶      result.add(u);
13     for (int v : adj.get(u))
14       if (--indeg[v] == 0) q.add(v);
15   }
16   return result;  // size < n → cycle
17 }
```

VARIABLES

`u = 3`



indeg[] (in-degree)

	0	1	2	3	4	5
indeg	0	0	0	0	0	1

q	(empty)					

result	1	2	0	4	3	

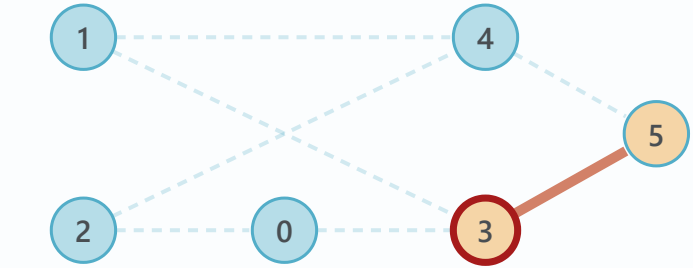
3 leaves the queue and joins the result: all its prerequisites are already there.

Code Trace: Topological Sorting (Kahn, continued)

```
1  List<Integer> kahn(List<List<Integer>> adj)
   {
2    int n = adj.size();
3    int[] indeg = new int[n];
4    for (List<Integer> l : adj)
5      for (int v : l) indeg[v]++;
6    Queue<Integer> q = new ArrayDeque<>();
7    for (int v = 0; v < n; v++)
8      if (indeg[v] == 0) q.add(v);
9    List<Integer> result = new ArrayList<>();
10   while (!q.isEmpty()) {
11     int u = q.poll();
12     result.add(u);
13     for (int v : adj.get(u))
▶        if (--indeg[v] == 0) q.add(v);
15   }
16   return result;  // size < n → cycle
17 }
```

VARIABLES

u = 3 v = 5



indeg[] (in-degree)

	0	1	2	3	4	5
indeg	0	0	0	0	0	0

q						
	5					

result						
	1	2	0	4	3	

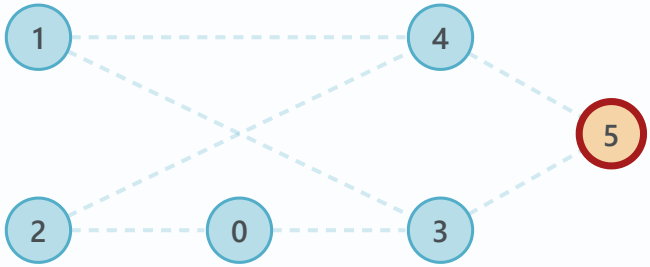
Count 3→5 as deleted: indeg[5] = **0** → 5 is ready and joins the queue.

Code Trace: Topological Sorting (Kahn, continued)

```
1  List<Integer> kahn(List<List<Integer>> adj)
   {
2    int n = adj.size();
3    int[] indeg = new int[n];
4    for (List<Integer> l : adj)
5      for (int v : l) indeg[v]++;
6    Queue<Integer> q = new ArrayDeque<>();
7    for (int v = 0; v < n; v++)
8      if (indeg[v] == 0) q.add(v);
9    List<Integer> result = new ArrayList<>();
10   while (!q.isEmpty()) {
▶      int u = q.poll();
▶      result.add(u);
13     for (int v : adj.get(u))
14       if (--indeg[v] == 0) q.add(v);
15   }
16   return result;  // size < n → cycle
17 }
```

VARIABLES

`u = 5`



indeg[] (in-degree)

	0	1	2	3	4	5
indeg	0	0	0	0	0	0

q: (empty)

result: 1 2 0 4 3 5

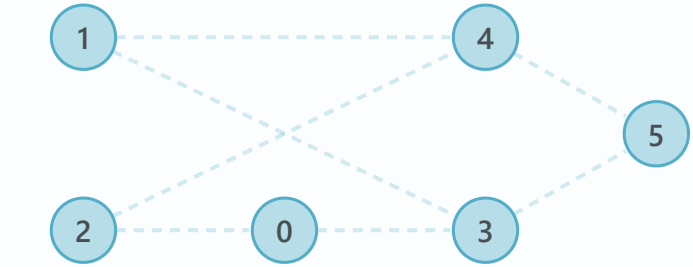
5 leaves the queue and joins the result: all its prerequisites are already there.

Code Trace: Topological Sorting (Kahn, continued)

```
1  List<Integer> kahn(List<List<Integer>> adj)
   {
2    int n = adj.size();
3    int[] indeg = new int[n];
4    for (List<Integer> l : adj)
5      for (int v : l) indeg[v]++;
6    Queue<Integer> q = new ArrayDeque<>();
7    for (int v = 0; v < n; v++)
8      if (indeg[v] == 0) q.add(v);
9    List<Integer> result = new ArrayList<>();
10   while (!q.isEmpty()) {
11     int u = q.poll();
12     result.add(u);
13     for (int v : adj.get(u))
14       if (--indeg[v] == 0) q.add(v);
15   }
▶    return result;  // size < n → cycle
17 }
```

VARIABLES

—



indeg[] (in-degree)

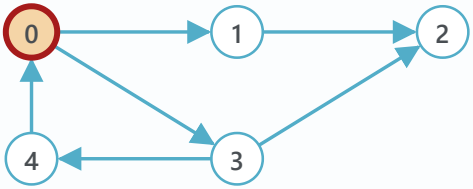
	0	1	2	3	4	5
indeg	0	0	0	0	0	0

q	(empty)					

result	1	2	0	4	3	5

The queue is empty; the result is **1 2 0 4 3 5**, and its size 6 = n → no cycle. Each edge caused one decrement: **O(n + m)**.

Code Trace: Cycles in a Digraph (Three-Color DFS)



v	0	1	2	3	4
color	1	0	0	0	0

0 white · 1 gray · 2 black

color[0] = 1 (gray): 0 is on the current DFS path.

Step 1 / 15

```
1  // color: 0 white, 1 gray (on path), 2 black
2  boolean hasCycle(int v, int[] color) {
▶    color[v] = 1;
4    for (int u : adj.get(v)) {
5      if (color[u] == 1) return true;  // back edge
6      if (color[u] == 0 && hasCycle(u, color))
7        return true;
8    }
9    color[v] = 2;
10   return false;
11 }
```

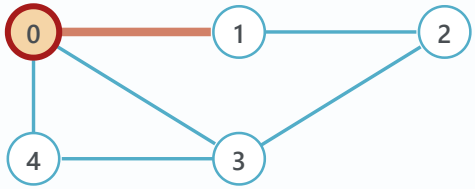
VARIABLES

`v = 0` `color = [1, 0, 0, 0, 0]`

CALL STACK

- `hasCycle(0)`
- `main()`

Code Trace: Cycles in a Digraph (Three-Color DFS)



v	0	1	2	3	4
color	1	0	0	0	0

0 white · 1 gray · 2 black

Edge 0 → 1: 1 is white → recursive call.

Step 2 / 15

```
1  // color: 0 white, 1 gray (on path), 2 black
2  boolean hasCycle(int v, int[] color) {
3    color[v] = 1;
4    for (int u : adj.get(v)) {
5      if (color[u] == 1) return true;  // back edge
▶      if (color[u] == 0 && hasCycle(u, color))
7        return true;
8    }
9    color[v] = 2;
10   return false;
11 }
```

VARIABLES

`v = 0` `u = 1`

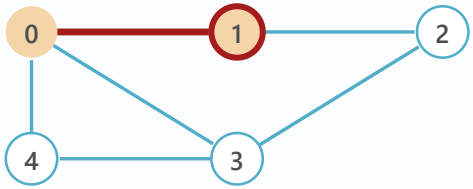
`color = [1, 0, 0, 0, 0]`

CALL STACK

`hasCycle(0)`

`main()`

Code Trace: Cycles in a Digraph (Three-Color DFS)



v	0	1	2	3	4
color	1	1	0	0	0

0 white · 1 gray · 2 black

color[1] = 1 (gray): 1 is on the current DFS path.

Step 3 / 15

```
1  // color: 0 white, 1 gray (on path), 2 black
2  boolean hasCycle(int v, int[] color) {
▶    color[v] = 1;
4    for (int u : adj.get(v)) {
5      if (color[u] == 1) return true;  // back edge
6      if (color[u] == 0 && hasCycle(u, color))
7        return true;
8    }
9    color[v] = 2;
10   return false;
11 }
```

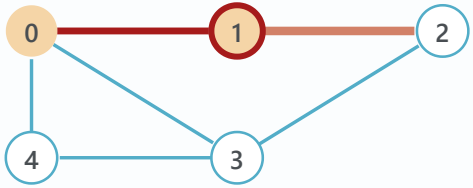
VARIABLES

`v = 1` `color = [1, 1, 0, 0, 0]`

CALL STACK

- `hasCycle(1)`
- `hasCycle(0)`
- `main()`

Code Trace: Cycles in a Digraph (Three-Color DFS)



v	0	1	2	3	4
color	1	1	0	0	0

0 white · 1 gray · 2 black

Edge 1 → 2: 2 is white → recursive call.

Step 4 / 15

```
1  // color: 0 white, 1 gray (on path), 2 black
2  boolean hasCycle(int v, int[] color) {
3    color[v] = 1;
4    for (int u : adj.get(v)) {
5      if (color[u] == 1) return true;  // back edge
▶      if (color[u] == 0 && hasCycle(u, color))
7        return true;
8    }
9    color[v] = 2;
10   return false;
11 }
```

VARIABLES

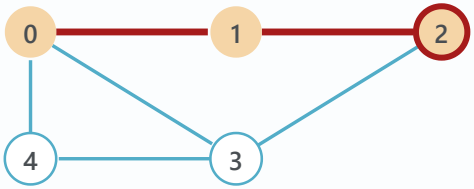
`v = 1` `u = 2`

`color = [1, 1, 0, 0, 0]`

CALL STACK

- `hasCycle(1)`
- `hasCycle(0)`
- `main()`

Code Trace: Cycles in a Digraph (Three-Color DFS)



v	0	1	2	3	4
color	1	1	1	0	0

0 white · 1 gray · 2 black

color[2] = 1 (gray): 2 is on the current DFS path.

Step 5 / 15

```
1  // color: 0 white, 1 gray (on path), 2 black
2  boolean hasCycle(int v, int[] color) {
▶    color[v] = 1;
4    for (int u : adj.get(v)) {
5      if (color[u] == 1) return true;  // back edge
6      if (color[u] == 0 && hasCycle(u, color))
7        return true;
8    }
9    color[v] = 2;
10   return false;
11 }
```

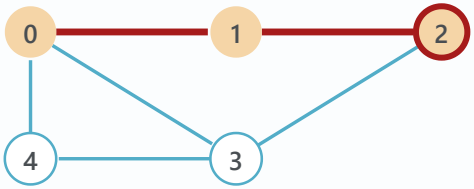
VARIABLES

`v = 2` `color = [1, 1, 1, 0, 0]`

CALL STACK

- `hasCycle(2)`
- `hasCycle(1)`
- `hasCycle(0)`
- `main()`

Code Trace: Cycles in a Digraph (Three-Color DFS)



v	0	1	2	3	4
color	1	1	2	0	0

0 white · 1 gray · 2 black

2 has no outgoing edges: **color[2] = 2 (black)**; return false and pop the frame.

Step 6 / 15

```
1  // color: 0 white, 1 gray (on path), 2 black
2  boolean hasCycle(int v, int[] color) {
3    color[v] = 1;
4    for (int u : adj.get(v)) {
5      if (color[u] == 1) return true;  // back edge
6      if (color[u] == 0 && hasCycle(u, color))
7        return true;
8    }
▶    color[v] = 2;
▶    return false;
11 }
```

VARIABLES

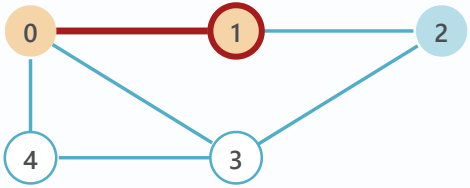
`v = 2` `return = false`

`color = [1, 1, 2, 0, 0]`

CALL STACK

- **hasCycle(2)**
- hasCycle(1)
- hasCycle(0)
- main()

Code Trace: Cycles in a Digraph (Three-Color DFS)



v	0	1	2	3	4
color	1	2	2	0	0

0 white · 1 gray · 2 black

1's neighbors are all done: **color[1] = 2 (black)**; return false and pop the frame.

Step 7 / 15

```
1  // color: 0 white, 1 gray (on path), 2 black
2  boolean hasCycle(int v, int[] color) {
3    color[v] = 1;
4    for (int u : adj.get(v)) {
5      if (color[u] == 1) return true;  // back edge
6      if (color[u] == 0 && hasCycle(u, color))
7        return true;
8    }
▶    color[v] = 2;
▶    return false;
11 }
```

VARIABLES

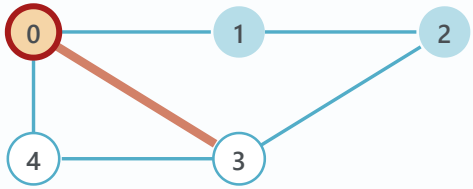
`v = 1` `return = false`

`color = [1, 2, 2, 0, 0]`

CALL STACK

- **hasCycle(1)**
- hasCycle(0)
- main()

Code Trace: Cycles in a Digraph (Three-Color DFS)



v	0	1	2	3	4
color	1	2	2	0	0

0 white · 1 gray · 2 black

Edge 0 → 3: 3 is white → recursive call.

Step 8 / 15

```
1  // color: 0 white, 1 gray (on path), 2 black
2  boolean hasCycle(int v, int[] color) {
3    color[v] = 1;
4    for (int u : adj.get(v)) {
5      if (color[u] == 1) return true;  // back edge
▶      if (color[u] == 0 && hasCycle(u, color))
7        return true;
8    }
9    color[v] = 2;
10   return false;
11 }
```

VARIABLES

`v = 0` `u = 3`

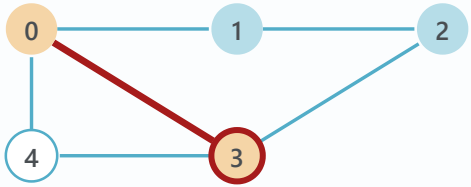
`color = [1, 2, 2, 0, 0]`

CALL STACK

`hasCycle(0)`

`main()`

Code Trace: Cycles in a Digraph (Three-Color DFS)



v	0	1	2	3	4
color	1	2	2	1	0

0 white · 1 gray · 2 black

color[3] = 1 (gray): 3 is on the current DFS path.

Step 9 / 15

```
1  // color: 0 white, 1 gray (on path), 2 black
2  boolean hasCycle(int v, int[] color) {
▶    color[v] = 1;
4    for (int u : adj.get(v)) {
5      if (color[u] == 1) return true;  // back edge
6      if (color[u] == 0 && hasCycle(u, color))
7        return true;
8    }
9    color[v] = 2;
10   return false;
11 }
```

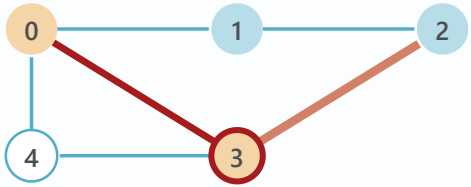
VARIABLES

`v = 3` `color = [1, 2, 2, 1, 0]`

CALL STACK

- `hasCycle(3)`
- `hasCycle(0)`
- `main()`

Code Trace: Cycles in a Digraph (Three-Color DFS)



v	0	1	2	3	4
color	1	2	2	1	0

0 white · 1 gray · 2 black

Edge 3 → 2: 2 is **black** (finished). Not a cycle: there is no path from 2 back to 3. Only a gray vertex signals a cycle.

Step 10 / 15

```
1  // color: 0 white, 1 gray (on path), 2 black
2  boolean hasCycle(int v, int[] color) {
3    color[v] = 1;
4    for (int u : adj.get(v)) {
▶      if (color[u] == 1) return true;  // back edge
▶      if (color[u] == 0 && hasCycle(u, color))
7        return true;
8    }
9    color[v] = 2;
10   return false;
11 }
```

VARIABLES

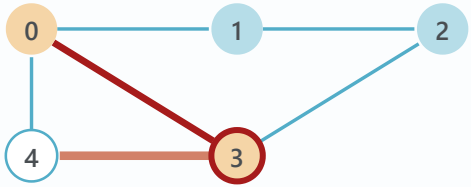
`v = 3` `u = 2`

`color = [1, 2, 2, 1, 0]`

CALL STACK

- `hasCycle(3)`
- `hasCycle(0)`
- `main()`

Code Trace: Cycles in a Digraph (Three-Color DFS)



v	0	1	2	3	4
color	1	2	2	1	0

0 white · 1 gray · 2 black

Edge 3 → 4: 4 is white → recursive call.

Step 11 / 15

```
1  // color: 0 white, 1 gray (on path), 2 black
2  boolean hasCycle(int v, int[] color) {
3    color[v] = 1;
4    for (int u : adj.get(v)) {
5      if (color[u] == 1) return true;  // back edge
▶      if (color[u] == 0 && hasCycle(u, color))
7        return true;
8    }
9    color[v] = 2;
10   return false;
11 }
```

VARIABLES

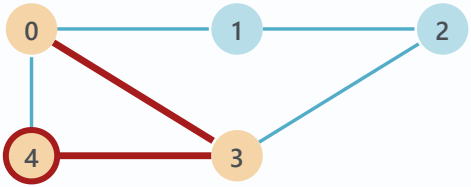
`v = 3` `u = 4`

`color = [1, 2, 2, 1, 0]`

CALL STACK

- `hasCycle(3)`
- `hasCycle(0)`
- `main()`

Code Trace: Cycles in a Digraph (Three-Color DFS)



v	0	1	2	3	4
color	1	2	2	1	1

0 white · 1 gray · 2 black

color[4] = 1 (gray): 4 is on the current DFS path.

Step 12 / 15

```
1  // color: 0 white, 1 gray (on path), 2 black
2  boolean hasCycle(int v, int[] color) {
▶    color[v] = 1;
4    for (int u : adj.get(v)) {
5      if (color[u] == 1) return true;  // back edge
6      if (color[u] == 0 && hasCycle(u, color))
7        return true;
8    }
9    color[v] = 2;
10   return false;
11 }
```

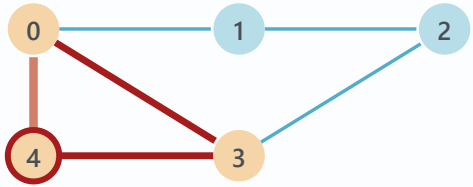
VARIABLES

`v = 4` `color = [1, 2, 2, 1, 1]`

CALL STACK

- `hasCycle(4)`
- `hasCycle(3)`
- `hasCycle(0)`
- `main()`

Code Trace: Cycles in a Digraph (Three-Color DFS)



v	0	1	2	3	4
color	1	2	2	1	1

0 white · 1 gray · 2 black

back edge → CYCLE

Edge 4 → 0: 0 is **gray**, so it is on the path! This is a **back edge** → cycle 0 → 3 → 4 → 0 found.

Step 13 / 15

```
1  // color: 0 white, 1 gray (on path), 2 black
2  boolean hasCycle(int v, int[] color) {
3    color[v] = 1;
4    for (int u : adj.get(v)) {
▶      if (color[u] == 1) return true;  // back edge
6      if (color[u] == 0 && hasCycle(u, color))
7        return true;
8    }
9    color[v] = 2;
10   return false;
11 }
```

VARIABLES

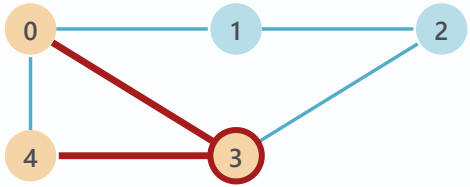
`v = 4` `u = 0`

`color = [1, 2, 2, 1, 1]`

CALL STACK

- `hasCycle(4)`
- `hasCycle(3)`
- `hasCycle(0)`
- `main()`

Code Trace: Cycles in a Digraph (Three-Color DFS)



v	0	1	2	3	4
color	1	2	2	1	1

0 white · 1 gray · 2 black

back edge → CYCLE

The inner call returned **true** → 3 returns true too.

Step 14 / 15

```
1  // color: 0 white, 1 gray (on path), 2 black
2  boolean hasCycle(int v, int[] color) {
3    color[v] = 1;
4    for (int u : adj.get(v)) {
5      if (color[u] == 1) return true;  // back edge
6      if (color[u] == 0 && hasCycle(u, color))
▶        return true;
8    }
9    color[v] = 2;
10   return false;
11 }
```

VARIABLES

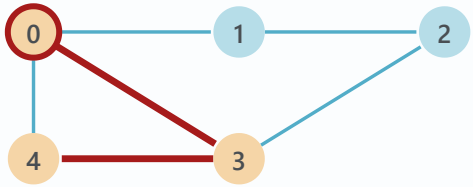
`v = 3` `u = 4` `return = true`

`color = [1, 2, 2, 1, 1]`

CALL STACK

- `hasCycle(3)`
- `hasCycle(0)`
- `main()`

Code Trace: Cycles in a Digraph (Three-Color DFS)



v	0	1	2	3	4
color	1	2	2	1	1

0 white · 1 gray · 2 black

back edge → CYCLE

The inner call returned **true** → 0 returns true too.

Step 15 / 15

```
1  // color: 0 white, 1 gray (on path), 2 black
2  boolean hasCycle(int v, int[] color) {
3    color[v] = 1;
4    for (int u : adj.get(v)) {
5      if (color[u] == 1) return true;  // back edge
6      if (color[u] == 0 && hasCycle(u, color))
▶        return true;
8    }
9    color[v] = 2;
10   return false;
11 }
```

VARIABLES

`v = 0` `u = 3` `return = true`

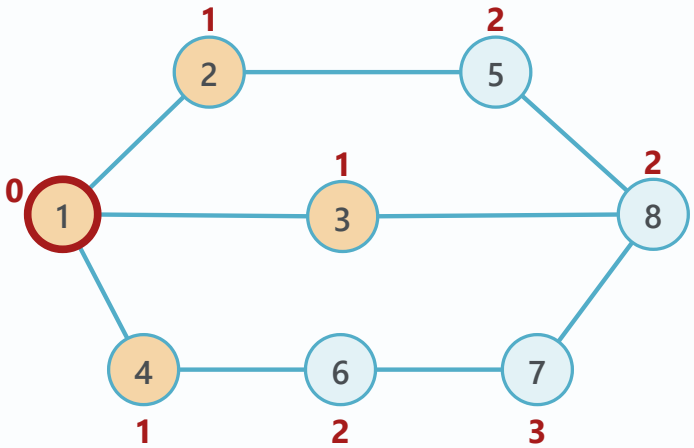
`color = [1, 2, 2, 1, 1]`

CALL STACK

- `hasCycle(0)`
- `main()`

The Unweighted Shortest-Path Problem

- Find shortest paths from a source vertex to **every other vertex** (single-source shortest paths).
- Every edge is assumed to have the same weight (1): path length = number of edges.
- Solution: **BFS**. A vertex on level k is exactly k edges away.



Source 1; the number next to each vertex is its distance from 1

There are two shortest paths to 7: 1 → 4 → 6 → 7 and 1 → 3 → 8 → 7 (3 edges each). BFS finds one of them; either way, the length is 3.

The Unweighted Shortest-Path Algorithm

Pseudocode

```
dist[every v] = ∞
dist[s] = 0
queue.add(s)
while queue is not empty:
  w = queue.remove()
  for each neighbor u of w:
    if dist[u] == ∞:
      dist[u] = dist[w] + 1
      queue.add(u)
return dist
```

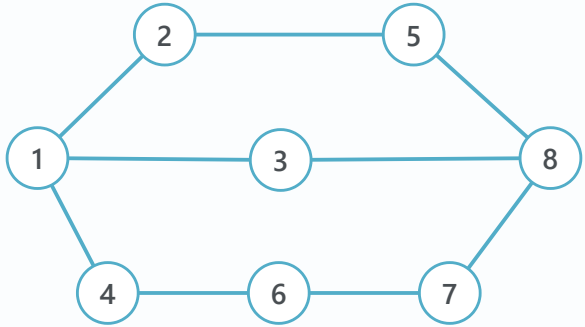
"dist = ∞" also means "not visited", so no separate array is needed. In Java, `graph` maps each vertex to its neighbor list (`Map<Integer, List<Integer>>`), because the vertices are numbered from 1.

Java · O(n + m)

```
Map<Integer, Integer> shortestPath(int s) {
  final int INF = Integer.MAX_VALUE;  // ∞
  Map<Integer, Integer> dist =
      new HashMap<>();
  for (int v : graph.keySet())
    dist.put(v, INF);
  dist.put(s, 0);
  Queue<Integer> q = new ArrayDeque<>();
  q.offer(s);
  while (!q.isEmpty()) {
    int w = q.poll();
    for (int u : graph.get(w))
      if (dist.get(u) == INF) {
        dist.put(u, dist.get(w) + 1);
        q.offer(u);
      }
  }
  return dist;
}
```

Code Trace: `shortestPath(1)`

```
1  Map<Integer, Integer> shortestPath(int s) {
2    final int INF = Integer.MAX_VALUE;  // ∞
3    Map<Integer, Integer> dist =
4        new HashMap<>();
▶    for (int v : graph.keySet())
▶      dist.put(v, INF);
7    dist.put(s, 0);
8    Queue<Integer> q = new ArrayDeque<>();
9    q.offer(s);
10   while (!q.isEmpty()) {
11     int w = q.poll();
12     for (int u : graph.get(w))
13       if (dist.get(u) == INF) {
14         dist.put(u, dist.get(w) + 1);
15         q.offer(u);
16       }
17   }
18   return dist;
19 }
```



	1	2	3	4	5	6	7	8
dist	∞	∞	∞	∞	∞	∞	∞	∞

q	(empty)					

Source s = 1. All distances are **INF** (∞): "not reached yet" = "not visited".

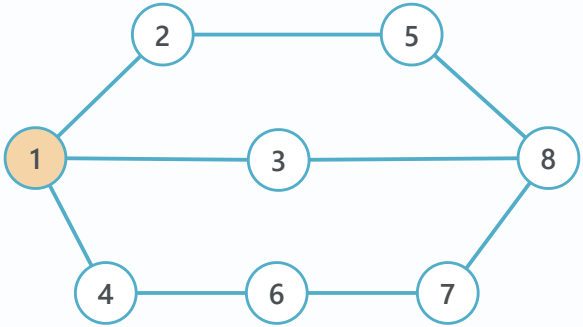
Step 1 / 13

VARIABLES

`s = 1`

Code Trace: `shortestPath(1)`

```
1  Map<Integer, Integer> shortestPath(int s) {
2    final int INF = Integer.MAX_VALUE;  // ∞
3    Map<Integer, Integer> dist =
4        new HashMap<>();
5    for (int v : graph.keySet())
6      dist.put(v, INF);
▶    dist.put(s, 0);
▶    Queue<Integer> q = new ArrayDeque<>();
▶    q.offer(s);
10   while (!q.isEmpty()) {
11     int w = q.poll();
12     for (int u : graph.get(w))
13       if (dist.get(u) == INF) {
14         dist.put(u, dist.get(w) + 1);
15         q.offer(u);
16       }
17   }
18   return dist;
19 }
```



	1	2	3	4	5	6	7	8
dist	0	∞	∞	∞	∞	∞	∞	∞
q	1							

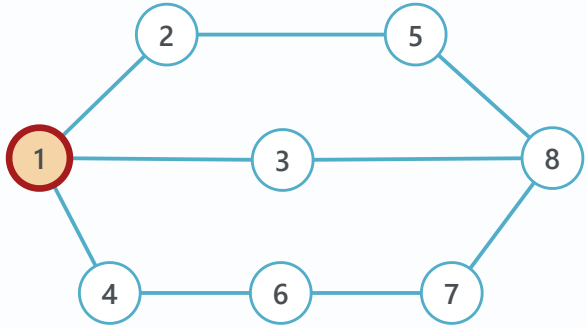
dist[1] = 0, and 1 enters the queue.

VARIABLES

`s = 1`

Code Trace: `shortestPath(1)`

```
1  Map<Integer, Integer> shortestPath(int s) {
2    final int INF = Integer.MAX_VALUE;  // ∞
3    Map<Integer, Integer> dist =
4        new HashMap<>();
5    for (int v : graph.keySet())
6      dist.put(v, INF);
7    dist.put(s, 0);
8    Queue<Integer> q = new ArrayDeque<>();
9    q.offer(s);
10   while (!q.isEmpty()) {
▶      int w = q.poll();
12     for (int u : graph.get(w))
13       if (dist.get(u) == INF) {
14         dist.put(u, dist.get(w) + 1);
15         q.offer(u);
16       }
17   }
18   return dist;
19 }
```



	1	2	3	4	5	6	7	8
dist	0	∞	∞	∞	∞	∞	∞	∞
q	(empty)							

Dequeue **1**; dist[1] = 0. Examine its neighbors 2, 3, 4.

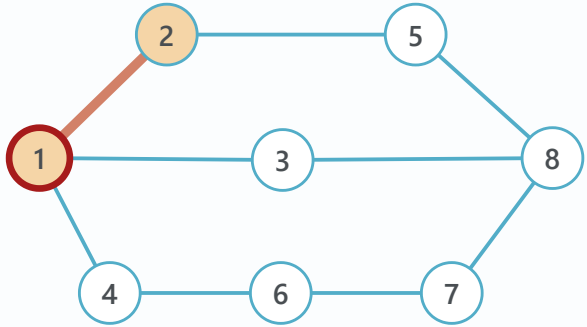
Step 3 / 13

VARIABLES

`s = 1` `w = 1`

Code Trace: `shortestPath(1)`

```
1  Map<Integer, Integer> shortestPath(int s) {
2    final int INF = Integer.MAX_VALUE;  // ∞
3    Map<Integer, Integer> dist =
4        new HashMap<>();
5    for (int v : graph.keySet())
6      dist.put(v, INF);
7    dist.put(s, 0);
8    Queue<Integer> q = new ArrayDeque<>();
9    q.offer(s);
10   while (!q.isEmpty()) {
11     int w = q.poll();
12     for (int u : graph.get(w))
▶        if (dist.get(u) == INF) {
▶          dist.put(u, dist.get(w) + 1);
▶          q.offer(u);
16       }
17   }
18   return dist;
19 }
```



	1	2	3	4	5	6	7	8
dist	0	1	∞	∞	∞	∞	∞	∞
q	2							

dist[2] = INF → **dist[2] = 0 + 1 = 1**; enqueue 2.

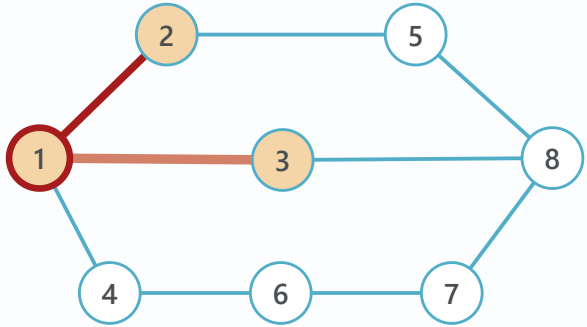
Step 4 / 13

VARIABLES

`s = 1` `w = 1` `u = 2`

Code Trace: `shortestPath(1)`

```
1  Map<Integer, Integer> shortestPath(int s) {
2    final int INF = Integer.MAX_VALUE;  // ∞
3    Map<Integer, Integer> dist =
4        new HashMap<>();
5    for (int v : graph.keySet())
6      dist.put(v, INF);
7    dist.put(s, 0);
8    Queue<Integer> q = new ArrayDeque<>();
9    q.offer(s);
10   while (!q.isEmpty()) {
11     int w = q.poll();
12     for (int u : graph.get(w))
▶        if (dist.get(u) == INF) {
▶          dist.put(u, dist.get(w) + 1);
▶          q.offer(u);
16       }
17   }
18   return dist;
19 }
```



	1	2	3	4	5	6	7	8
dist	0	1	1	∞	∞	∞	∞	∞
q	2	3						

dist[3] = INF → **dist[3] = 0 + 1 = 1**; enqueue 3.

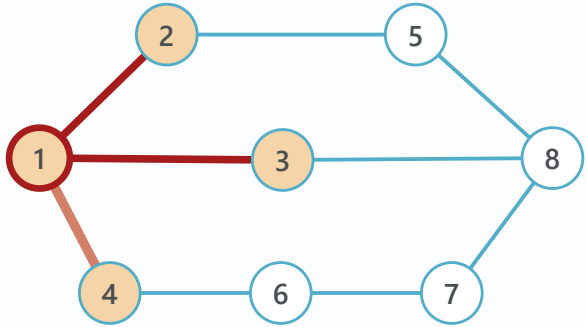
Step 5 / 13

VARIABLES

`s = 1` `w = 1` `u = 3`

Code Trace: `shortestPath(1)`

```
1  Map<Integer, Integer> shortestPath(int s) {
2    final int INF = Integer.MAX_VALUE;  // ∞
3    Map<Integer, Integer> dist =
4        new HashMap<>();
5    for (int v : graph.keySet())
6      dist.put(v, INF);
7    dist.put(s, 0);
8    Queue<Integer> q = new ArrayDeque<>();
9    q.offer(s);
10   while (!q.isEmpty()) {
11     int w = q.poll();
12     for (int u : graph.get(w))
▶        if (dist.get(u) == INF) {
▶          dist.put(u, dist.get(w) + 1);
▶          q.offer(u);
16       }
17   }
18   return dist;
19 }
```



	1	2	3	4	5	6	7	8
dist	0	1	1	1	∞	∞	∞	∞
q	2	3	4					

dist[4] = INF → **dist[4] = 0 + 1 = 1**; enqueue 4.

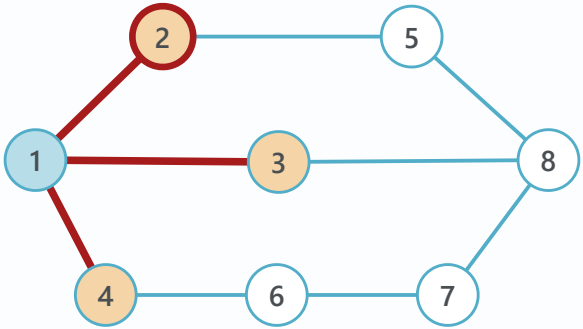
Step 6 / 13

VARIABLES

`s = 1` `w = 1` `u = 4`

Code Trace: `shortestPath(1)`

```
1  Map<Integer, Integer> shortestPath(int s) {
2    final int INF = Integer.MAX_VALUE;  // ∞
3    Map<Integer, Integer> dist =
4        new HashMap<>();
5    for (int v : graph.keySet())
6      dist.put(v, INF);
7    dist.put(s, 0);
8    Queue<Integer> q = new ArrayDeque<>();
9    q.offer(s);
10   while (!q.isEmpty()) {
▶      int w = q.poll();
12     for (int u : graph.get(w))
13       if (dist.get(u) == INF) {
14         dist.put(u, dist.get(w) + 1);
15         q.offer(u);
16       }
17   }
18   return dist;
19 }
```



	1	2	3	4	5	6	7	8
dist	0	1	1	1	∞	∞	∞	∞

q	3	4					

Dequeue **2**; dist[2] = 1. Examine its neighbors 1, 5.

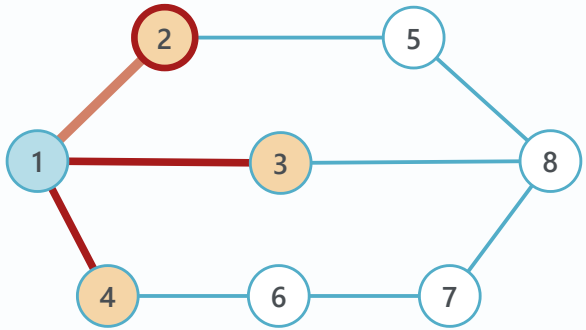
Step 7 / 13

VARIABLES

`s = 1` `w = 2`

Code Trace: `shortestPath(1)`

```
1  Map<Integer, Integer> shortestPath(int s) {
2    final int INF = Integer.MAX_VALUE;  // ∞
3    Map<Integer, Integer> dist =
4        new HashMap<>();
5    for (int v : graph.keySet())
6      dist.put(v, INF);
7    dist.put(s, 0);
8    Queue<Integer> q = new ArrayDeque<>();
9    q.offer(s);
10   while (!q.isEmpty()) {
11     int w = q.poll();
12     for (int u : graph.get(w))
▶        if (dist.get(u) == INF) {
14         dist.put(u, dist.get(w) + 1);
15         q.offer(u);
16       }
17   }
18   return dist;
19 }
```



	1	2	3	4	5	6	7	8
dist	0	1	1	1	∞	∞	∞	∞

q							
	3	4					

1: distance already known (0) → unchanged. In BFS, the first distance found is the shortest.

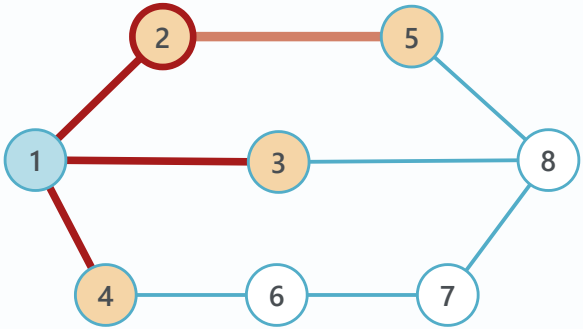
Step 8 / 13

VARIABLES

`s = 1` `w = 2` `u = 1`

Code Trace: `shortestPath(1)`

```
1  Map<Integer, Integer> shortestPath(int s) {
2    final int INF = Integer.MAX_VALUE;  // ∞
3    Map<Integer, Integer> dist =
4        new HashMap<>();
5    for (int v : graph.keySet())
6      dist.put(v, INF);
7    dist.put(s, 0);
8    Queue<Integer> q = new ArrayDeque<>();
9    q.offer(s);
10   while (!q.isEmpty()) {
11     int w = q.poll();
12     for (int u : graph.get(w))
▶        if (dist.get(u) == INF) {
▶          dist.put(u, dist.get(w) + 1);
▶          q.offer(u);
16       }
17   }
18   return dist;
19 }
```



	1	2	3	4	5	6	7	8
dist	0	1	1	1	2	∞	∞	∞
q	3	4	5					

dist[5] = INF → **dist[5] = 1 + 1 = 2**; enqueue 5.

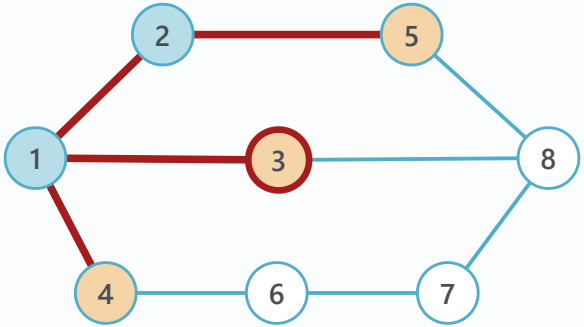
Step 9 / 13

VARIABLES

`s = 1` `w = 2` `u = 5`

Code Trace: `shortestPath(1)`

```
1  Map<Integer, Integer> shortestPath(int s) {
2    final int INF = Integer.MAX_VALUE;  // ∞
3    Map<Integer, Integer> dist =
4        new HashMap<>();
5    for (int v : graph.keySet())
6      dist.put(v, INF);
7    dist.put(s, 0);
8    Queue<Integer> q = new ArrayDeque<>();
9    q.offer(s);
10   while (!q.isEmpty()) {
▶      int w = q.poll();
12     for (int u : graph.get(w))
13       if (dist.get(u) == INF) {
14         dist.put(u, dist.get(w) + 1);
15         q.offer(u);
16       }
17   }
18   return dist;
19 }
```



	1	2	3	4	5	6	7	8
dist	0	1	1	1	2	∞	∞	∞

q	4	5					

Dequeue **3**; dist[3] = 1. Examine its neighbors 1, 8.

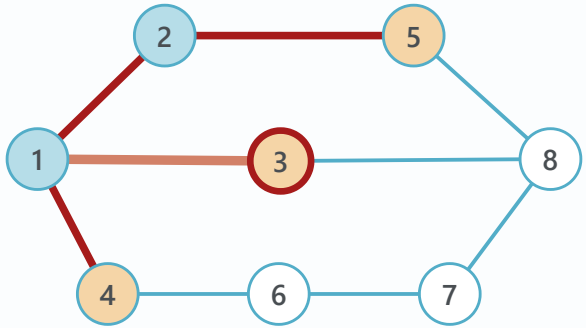
Step 10 / 13

VARIABLES

`s = 1` `w = 3`

Code Trace: `shortestPath(1)`

```
1  Map<Integer, Integer> shortestPath(int s) {
2    final int INF = Integer.MAX_VALUE;  // ∞
3    Map<Integer, Integer> dist =
4        new HashMap<>();
5    for (int v : graph.keySet())
6      dist.put(v, INF);
7    dist.put(s, 0);
8    Queue<Integer> q = new ArrayDeque<>();
9    q.offer(s);
10   while (!q.isEmpty()) {
11     int w = q.poll();
12     for (int u : graph.get(w))
▶        if (dist.get(u) == INF) {
14         dist.put(u, dist.get(w) + 1);
15         q.offer(u);
16       }
17   }
18   return dist;
19 }
```



	1	2	3	4	5	6	7	8
dist	0	1	1	1	2	∞	∞	∞

q							
	4	5					

1: distance already known (0) → unchanged. In BFS, the first distance found is the shortest.

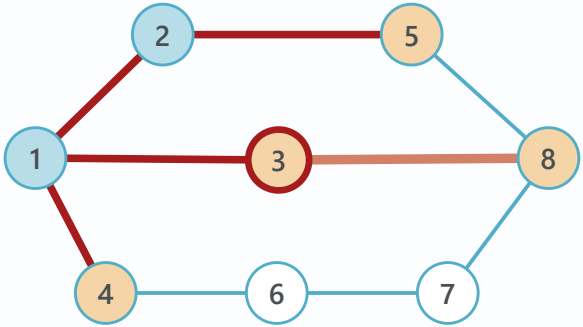
Step 11 / 13

VARIABLES

`s = 1` `w = 3` `u = 1`

Code Trace: `shortestPath(1)`

```
1  Map<Integer, Integer> shortestPath(int s) {
2    final int INF = Integer.MAX_VALUE;  // ∞
3    Map<Integer, Integer> dist =
4        new HashMap<>();
5    for (int v : graph.keySet())
6      dist.put(v, INF);
7    dist.put(s, 0);
8    Queue<Integer> q = new ArrayDeque<>();
9    q.offer(s);
10   while (!q.isEmpty()) {
11     int w = q.poll();
12     for (int u : graph.get(w))
▶        if (dist.get(u) == INF) {
▶          dist.put(u, dist.get(w) + 1);
▶          q.offer(u);
16       }
17   }
18   return dist;
19 }
```



	1	2	3	4	5	6	7	8
dist	0	1	1	1	2	∞	∞	2

q	4	5	8				

dist[8] = INF → **dist[8] = 1 + 1 = 2**; enqueue 8.

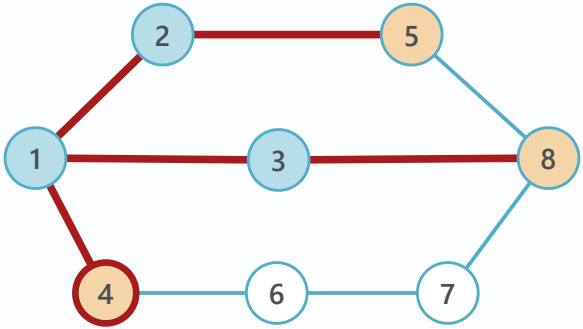
Step 12 / 13

VARIABLES

`s = 1` `w = 3` `u = 8`

Code Trace: `shortestPath(1)`

```
1  Map<Integer, Integer> shortestPath(int s) {
2    final int INF = Integer.MAX_VALUE;  // ∞
3    Map<Integer, Integer> dist =
4        new HashMap<>();
5    for (int v : graph.keySet())
6      dist.put(v, INF);
7    dist.put(s, 0);
8    Queue<Integer> q = new ArrayDeque<>();
9    q.offer(s);
10   while (!q.isEmpty()) {
▶      int w = q.poll();
12     for (int u : graph.get(w))
13       if (dist.get(u) == INF) {
14         dist.put(u, dist.get(w) + 1);
15         q.offer(u);
16       }
17   }
18   return dist;
19 }
```



	1	2	3	4	5	6	7	8
dist	0	1	1	1	2	∞	∞	2

q	5	8					

Dequeue **4**; dist[4] = 1. Examine its neighbors 1, 6.

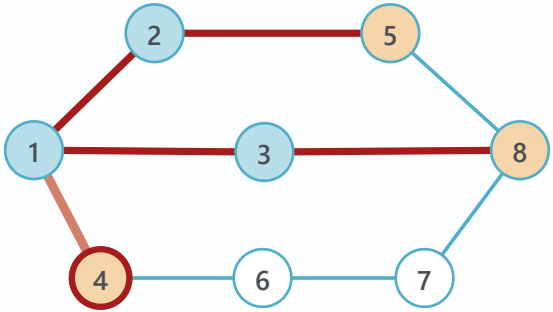
Step 13 / 13

VARIABLES

`s = 1` `w = 4`

Code Trace: `shortestPath(1)` (continued)

```
1  Map<Integer, Integer> shortestPath(int s) {
2    final int INF = Integer.MAX_VALUE;  // ∞
3    Map<Integer, Integer> dist =
4        new HashMap<>();
5    for (int v : graph.keySet())
6      dist.put(v, INF);
7    dist.put(s, 0);
8    Queue<Integer> q = new ArrayDeque<>();
9    q.offer(s);
10   while (!q.isEmpty()) {
11     int w = q.poll();
12     for (int u : graph.get(w))
▶        if (dist.get(u) == INF) {
14         dist.put(u, dist.get(w) + 1);
15         q.offer(u);
16       }
17   }
18   return dist;
19 }
```



	1	2	3	4	5	6	7	8
dist	0	1	1	1	2	∞	∞	2

q	5	8					

1: distance already known (0) → unchanged. In BFS, the first distance found is the shortest.

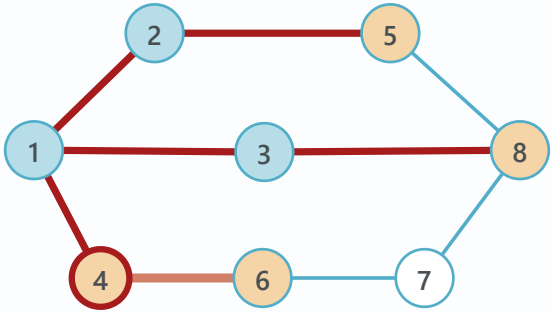
Step 1 / 12

VARIABLES

`s = 1` `w = 4` `u = 1`

Code Trace: `shortestPath(1)` (continued)

```
1  Map<Integer, Integer> shortestPath(int s) {
2    final int INF = Integer.MAX_VALUE;  // ∞
3    Map<Integer, Integer> dist =
4        new HashMap<>();
5    for (int v : graph.keySet())
6      dist.put(v, INF);
7    dist.put(s, 0);
8    Queue<Integer> q = new ArrayDeque<>();
9    q.offer(s);
10   while (!q.isEmpty()) {
11     int w = q.poll();
12     for (int u : graph.get(w))
▶        if (dist.get(u) == INF) {
▶          dist.put(u, dist.get(w) + 1);
▶          q.offer(u);
16       }
17   }
18   return dist;
19 }
```



	1	2	3	4	5	6	7	8
dist	0	1	1	1	2	2	∞	2

q							
	5	8	6				

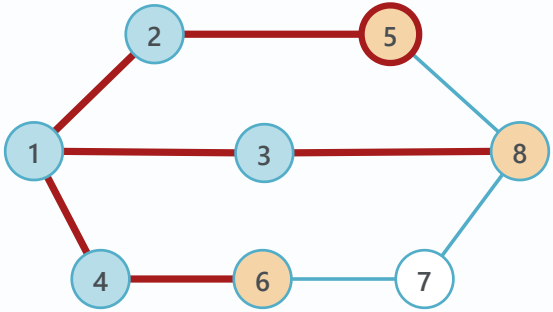
dist[6] = INF → **dist[6] = 1 + 1 = 2**; enqueue 6.

VARIABLES

`s = 1` `w = 4` `u = 6`

Code Trace: `shortestPath(1)` (continued)

```
1  Map<Integer, Integer> shortestPath(int s) {
2    final int INF = Integer.MAX_VALUE;  // ∞
3    Map<Integer, Integer> dist =
4        new HashMap<>();
5    for (int v : graph.keySet())
6      dist.put(v, INF);
7    dist.put(s, 0);
8    Queue<Integer> q = new ArrayDeque<>();
9    q.offer(s);
10   while (!q.isEmpty()) {
▶      int w = q.poll();
12     for (int u : graph.get(w))
13       if (dist.get(u) == INF) {
14         dist.put(u, dist.get(w) + 1);
15         q.offer(u);
16       }
17   }
18   return dist;
19 }
```



	1	2	3	4	5	6	7	8
dist	0	1	1	1	2	2	∞	2

q							
8	6						

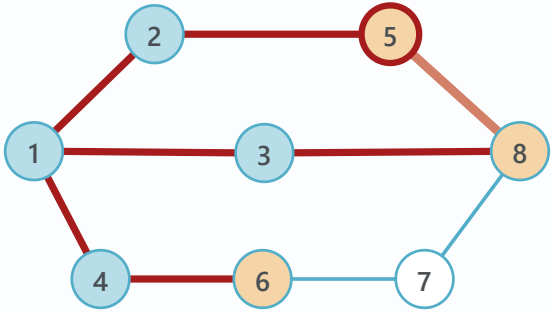
Dequeue **5**; dist[5] = 2. Examine its neighbors 2, 8.

VARIABLES

`s = 1` `w = 5`

Code Trace: `shortestPath(1)` (continued)

```
1  Map<Integer, Integer> shortestPath(int s) {
2    final int INF = Integer.MAX_VALUE;  // ∞
3    Map<Integer, Integer> dist =
4        new HashMap<>();
5    for (int v : graph.keySet())
6      dist.put(v, INF);
7    dist.put(s, 0);
8    Queue<Integer> q = new ArrayDeque<>();
9    q.offer(s);
10   while (!q.isEmpty()) {
11     int w = q.poll();
12     for (int u : graph.get(w))
▶        if (dist.get(u) == INF) {
14         dist.put(u, dist.get(w) + 1);
15         q.offer(u);
16       }
17   }
18   return dist;
19 }
```



	1	2	3	4	5	6	7	8
dist	0	1	1	1	2	2	∞	2

q							
8	6						

2, 8: distance already known (1, 2) → unchanged. In BFS, the first distance found is the shortest.

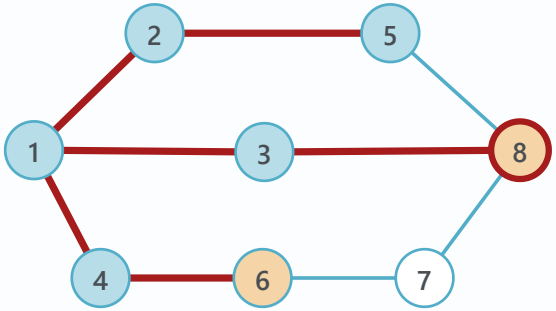
Step 4 / 12

VARIABLES

`s = 1` `w = 5` `u = 8`

Code Trace: `shortestPath(1)` (continued)

```
1  Map<Integer, Integer> shortestPath(int s) {
2    final int INF = Integer.MAX_VALUE;  // ∞
3    Map<Integer, Integer> dist =
4        new HashMap<>();
5    for (int v : graph.keySet())
6      dist.put(v, INF);
7    dist.put(s, 0);
8    Queue<Integer> q = new ArrayDeque<>();
9    q.offer(s);
10   while (!q.isEmpty()) {
▶      int w = q.poll();
12     for (int u : graph.get(w))
13       if (dist.get(u) == INF) {
14         dist.put(u, dist.get(w) + 1);
15         q.offer(u);
16       }
17   }
18   return dist;
19 }
```



	1	2	3	4	5	6	7	8
dist	0	1	1	1	2	2	∞	2

q	6						

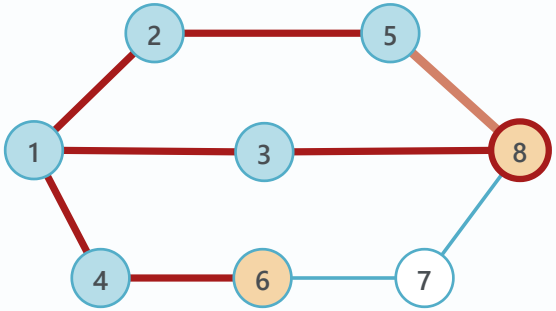
Dequeue **8**; dist[8] = 2. Examine its neighbors 3, 5, 7.

VARIABLES

`s = 1` `w = 8`

Code Trace: `shortestPath(1)` (continued)

```
1  Map<Integer, Integer> shortestPath(int s) {
2    final int INF = Integer.MAX_VALUE;  // ∞
3    Map<Integer, Integer> dist =
4        new HashMap<>();
5    for (int v : graph.keySet())
6      dist.put(v, INF);
7    dist.put(s, 0);
8    Queue<Integer> q = new ArrayDeque<>();
9    q.offer(s);
10   while (!q.isEmpty()) {
11     int w = q.poll();
12     for (int u : graph.get(w))
▶        if (dist.get(u) == INF) {
14         dist.put(u, dist.get(w) + 1);
15         q.offer(u);
16       }
17   }
18   return dist;
19 }
```



	1	2	3	4	5	6	7	8
dist	0	1	1	1	2	2	∞	2

q	6						

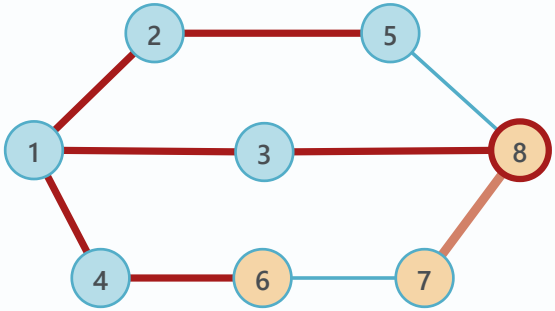
3, 5: distance already known (1, 2) → unchanged. In BFS, the first distance found is the shortest.

VARIABLES

`s = 1` `w = 8` `u = 5`

Code Trace: `shortestPath(1)` (continued)

```
1  Map<Integer, Integer> shortestPath(int s) {
2    final int INF = Integer.MAX_VALUE;  // ∞
3    Map<Integer, Integer> dist =
4        new HashMap<>();
5    for (int v : graph.keySet())
6      dist.put(v, INF);
7    dist.put(s, 0);
8    Queue<Integer> q = new ArrayDeque<>();
9    q.offer(s);
10   while (!q.isEmpty()) {
11     int w = q.poll();
12     for (int u : graph.get(w))
▶        if (dist.get(u) == INF) {
▶          dist.put(u, dist.get(w) + 1);
▶          q.offer(u);
16       }
17   }
18   return dist;
19 }
```



	1	2	3	4	5	6	7	8
dist	0	1	1	1	2	2	3	2

q							
	6	7					

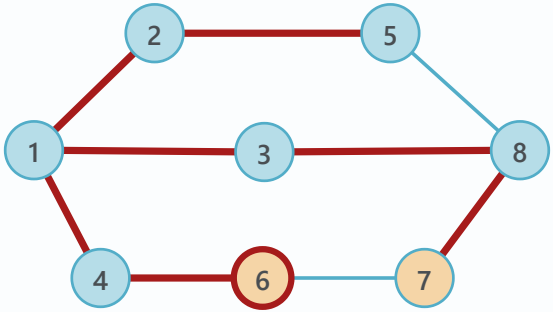
dist[7] = INF → **dist[7] = 2 + 1 = 3**; enqueue 7.

VARIABLES

`s = 1` `w = 8` `u = 7`

Code Trace: `shortestPath(1)` (continued)

```
1  Map<Integer, Integer> shortestPath(int s) {
2    final int INF = Integer.MAX_VALUE;  // ∞
3    Map<Integer, Integer> dist =
4        new HashMap<>();
5    for (int v : graph.keySet())
6      dist.put(v, INF);
7    dist.put(s, 0);
8    Queue<Integer> q = new ArrayDeque<>();
9    q.offer(s);
10   while (!q.isEmpty()) {
▶      int w = q.poll();
12     for (int u : graph.get(w))
13       if (dist.get(u) == INF) {
14         dist.put(u, dist.get(w) + 1);
15         q.offer(u);
16       }
17   }
18   return dist;
19 }
```



	1	2	3	4	5	6	7	8
dist	0	1	1	1	2	2	3	2

q	7						

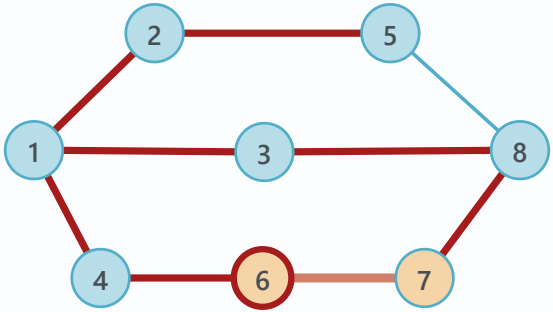
Dequeue **6**; dist[6] = 2. Examine its neighbors 4, 7.

VARIABLES

`s = 1` `w = 6`

Code Trace: `shortestPath(1)` (continued)

```
1  Map<Integer, Integer> shortestPath(int s) {
2    final int INF = Integer.MAX_VALUE;  // ∞
3    Map<Integer, Integer> dist =
4        new HashMap<>();
5    for (int v : graph.keySet())
6      dist.put(v, INF);
7    dist.put(s, 0);
8    Queue<Integer> q = new ArrayDeque<>();
9    q.offer(s);
10   while (!q.isEmpty()) {
11     int w = q.poll();
12     for (int u : graph.get(w))
▶        if (dist.get(u) == INF) {
14         dist.put(u, dist.get(w) + 1);
15         q.offer(u);
16       }
17   }
18   return dist;
19 }
```



	1	2	3	4	5	6	7	8
dist	0	1	1	1	2	2	3	2

q	7						

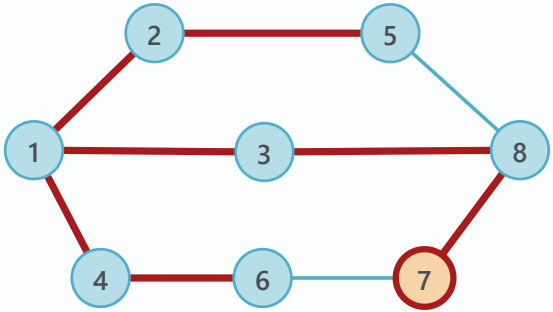
4, 7: distance already known (1, 3) → unchanged. In BFS, the first distance found is the shortest.

VARIABLES

`s = 1` `w = 6` `u = 7`

Code Trace: `shortestPath(1)` (continued)

```
1  Map<Integer, Integer> shortestPath(int s) {
2    final int INF = Integer.MAX_VALUE;  // ∞
3    Map<Integer, Integer> dist =
4        new HashMap<>();
5    for (int v : graph.keySet())
6      dist.put(v, INF);
7    dist.put(s, 0);
8    Queue<Integer> q = new ArrayDeque<>();
9    q.offer(s);
10   while (!q.isEmpty()) {
▶      int w = q.poll();
12     for (int u : graph.get(w))
13       if (dist.get(u) == INF) {
14         dist.put(u, dist.get(w) + 1);
15         q.offer(u);
16       }
17   }
18   return dist;
19 }
```



	1	2	3	4	5	6	7	8
dist	0	1	1	1	2	2	3	2

q	(empty)						

Dequeue **7**; dist[7] = 3. Examine its neighbors 6, 8.

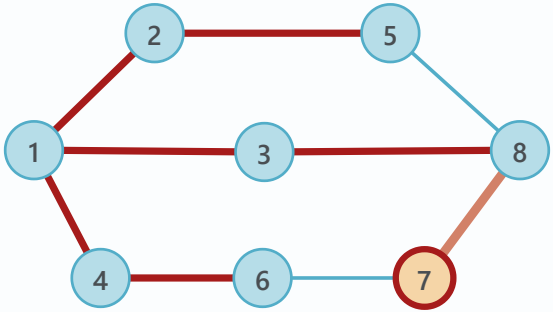
Step 10 / 12

VARIABLES

`s = 1` `w = 7`

Code Trace: `shortestPath(1)` (continued)

```
1  Map<Integer, Integer> shortestPath(int s) {
2    final int INF = Integer.MAX_VALUE;  // ∞
3    Map<Integer, Integer> dist =
4        new HashMap<>();
5    for (int v : graph.keySet())
6      dist.put(v, INF);
7    dist.put(s, 0);
8    Queue<Integer> q = new ArrayDeque<>();
9    q.offer(s);
10   while (!q.isEmpty()) {
11     int w = q.poll();
12     for (int u : graph.get(w))
▶        if (dist.get(u) == INF) {
14         dist.put(u, dist.get(w) + 1);
15         q.offer(u);
16       }
17   }
18   return dist;
19 }
```



	1	2	3	4	5	6	7	8
dist	0	1	1	1	2	2	3	2

q	(empty)						

6, 8: distance already known (2, 2) → unchanged. In BFS, the first distance found is the shortest.

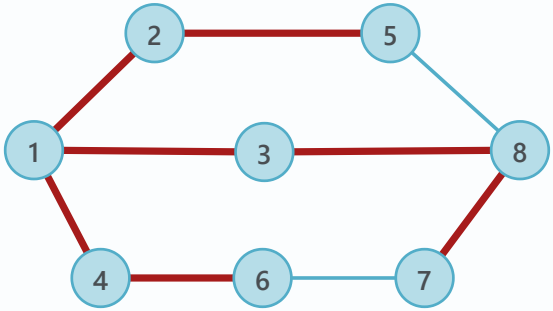
Step 11 / 12

VARIABLES

`s = 1` `w = 7` `u = 8`

Code Trace: `shortestPath(1)` (continued)

```
1  Map<Integer, Integer> shortestPath(int s) {
2    final int INF = Integer.MAX_VALUE;  // ∞
3    Map<Integer, Integer> dist =
4        new HashMap<>();
5    for (int v : graph.keySet())
6      dist.put(v, INF);
7    dist.put(s, 0);
8    Queue<Integer> q = new ArrayDeque<>();
9    q.offer(s);
10   while (!q.isEmpty()) {
11     int w = q.poll();
12     for (int u : graph.get(w))
13       if (dist.get(u) == INF) {
14         dist.put(u, dist.get(w) + 1);
15         q.offer(u);
16       }
17   }
▶    return dist;
19 }
```



	1	2	3	4	5	6	7	8
dist	0	1	1	1	2	2	3	2

q	(empty)						

The queue is empty → `return dist`. The distance to 7 is 3 (1 → 3 → 8 → 7 was found; 1 → 4 → 6 → 7 also has 3 edges). O(n + m).

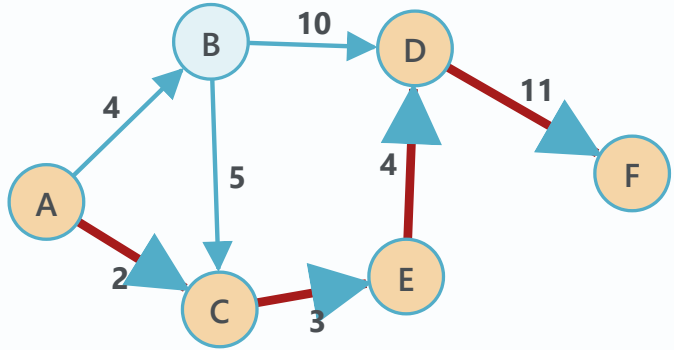
Step 12 / 12

VARIABLES

`s = 1`

The Weighted Shortest-Path Problem

- Find shortest paths from a source vertex to every other vertex.
- Each edge may have its own weight: a distance, time, cost…
- Path length = sum of the edge weights.



Shortest path from A to F: A → C → E → D → F = 2 + 3 + 4 + 11 = **20**

BFS is not enough here

The path with the fewest edges is A → B → D → F (3 edges), but its length is 4 + 10 + 11 = **25**

.

Solution: Dijkstra's algorithm

Edsger W. Dijkstra, 1956. Weights **must not be negative**

.

Dijkstra's Algorithm: Idea and Pseudocode

- Pick the unsettled vertex with the smallest distance.
- Update its neighbors' distances by **relaxation**.
- Use a priority queue for the selection and `prev[]` to rebuild the path.

```
Dijkstra(G, s):
  for each vertex v:
    dist[v] = ∞
    prev[v] = undefined
  dist[s] = 0
  Q = all vertices  // priority queue
  while Q is not empty:
    u = Q.removeMin()   // smallest dist[u]
    for each neighbor v of u in Q:
      alt = dist[u] + weight(u, v)
      if alt < dist[v]:
        dist[v] = alt
        prev[v] = u
        // decrease v's priority in Q
```

SHORTEST PATHS

Dijkstra's Algorithm in Java

```
long[] dijkstra(List<List<int[]>> adj, int s, int[] prev) { // weights ≥ 0
    long[] d = new long[adj.size()];
    Arrays.fill(d, Long.MAX_VALUE);  d[s] = 0;  Arrays.fill(prev, -1);
    PriorityQueue<long[]> pq =                      // {vertex, dist}
        new PriorityQueue<>((a, b) -> Long.compare(a[1], b[1]));
    pq.add(new long[]{s, 0});
    while (!pq.isEmpty()) {
        long[] e = pq.poll();  int u = (int) e[0];
        if (e[1] > d[u]) continue;                  // stale entry: skip
        for (int[] k : adj.get(u)) {
            int v = k[0];  long alt = d[u] + k[1];
            if (alt < d[v]) { d[v] = alt; prev[v] = u; pq.add(new long[]{v, alt}); }
        }
    }
    return d;
}
```

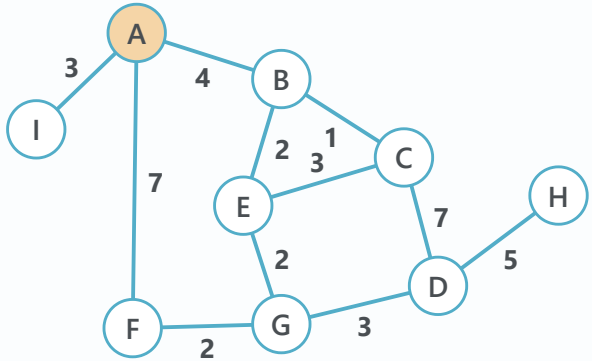
Priority = distance (not the vertex number). Java's `PriorityQueue` has no "decrease key": we add a new entry, and the old one is skipped as *stale* when it comes out. `prev[v] = u` records the shortest-path tree; a path is traced back from the target.

Code Trace: `dijkstra(adj, A)`

```
// d[] = ∞, d[s] = 0, prev[] = -1
pq.add(new long[]{s, 0});
while (!pq.isEmpty()) {
  long[] e = pq.poll();
  int u = (int) e[0];
  if (e[1] > d[u]) continue;  // stale
  for (int[] k : adj.get(u)) {
    int v = k[0];
    long alt = d[u] + k[1];
    if (alt < d[v]) {
      d[v] = alt;
      prev[v] = u;
      pq.add(new long[]{v, alt});
    }
  }
}
```

VARIABLES

`s = A`



	A	B	C	D	E	F	G	H	I
d[]	0	∞	∞	∞	∞	∞	∞	∞	∞
prev	–	–	–	–	–	–	–	–	–

pq (shown sorted by distance; faded = stale)

A,0

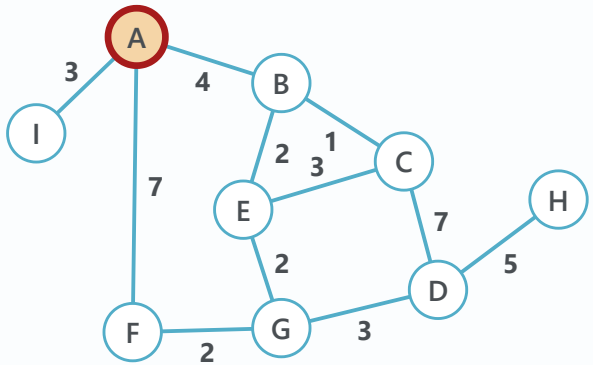
Source A: d[A] = 0, all others ∞. Add {A, 0} to the queue. Priority = **distance**.

Code Trace: `dijkstra(adj, A)`

```
1  // d[] = ∞, d[s] = 0, prev[] = -1
2  pq.add(new long[]{s, 0});
3  while (!pq.isEmpty()) {
▶    long[] e = pq.poll();
▶    int u = (int) e[0];
▶    if (e[1] > d[u]) continue;  // stale
7    for (int[] k : adj.get(u)) {
8      int v = k[0];
9      long alt = d[u] + k[1];
10     if (alt < d[v]) {
11       d[v] = alt;
12       prev[v] = u;
13       pq.add(new long[]{v, alt});
14     }
15   }
16 }
```

VARIABLES

`e = {A,0}` `u = A`



	A	B	C	D	E	F	G	H	I
d[]	0	∞	∞	∞	∞	∞	∞	∞	∞
prev	–	–	–	–	–	–	–	–	–

pq (shown sorted by distance; faded = stale)

(empty)

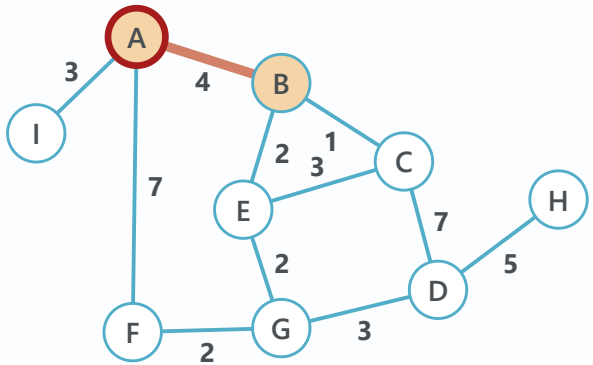
Poll the smallest entry {A, 0}; 0 = d[A] (current) → **A is settled**. Relax its neighbors B, F, I.

Code Trace: `dijkstra(adj, A)`

```
1  // d[] = ∞, d[s] = 0, prev[] = -1
2  pq.add(new long[]{s, 0});
3  while (!pq.isEmpty()) {
4    long[] e = pq.poll();
5    int u = (int) e[0];
6    if (e[1] > d[u]) continue;  // stale
7    for (int[] k : adj.get(u)) {
       int v = k[0];
       long alt = d[u] + k[1];
       if (alt < d[v]) {
         d[v] = alt;
         prev[v] = u;
         pq.add(new long[]{v, alt});
14     }
15   }
16 }
```

VARIABLES

`e = {A,0}` `u = A` `v = B` `alt = 4`



	A	B	C	D	E	F	G	H	I
d[]	0	4	∞	∞	∞	∞	∞	∞	∞
prev	–	A	–	–	–	–	–	–	–

pq (shown sorted by distance; faded = stale)

B,4

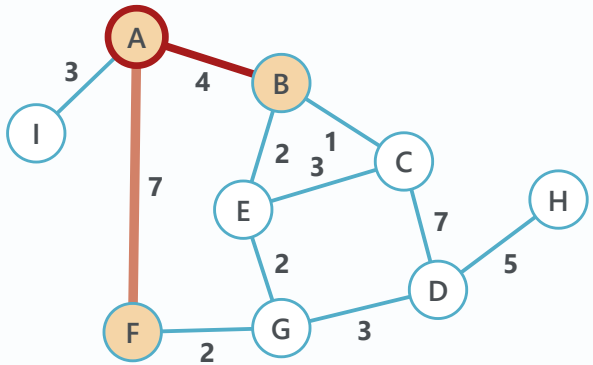
v = B: alt = 0 + 4 = **4** < ∞ → d[B] = 4, prev[B] = A, {B, 4} enters the queue.

Code Trace: `dijkstra(adj, A)`

```
1  // d[] = ∞, d[s] = 0, prev[] = -1
2  pq.add(new long[]{s, 0});
3  while (!pq.isEmpty()) {
4    long[] e = pq.poll();
5    int u = (int) e[0];
6    if (e[1] > d[u]) continue;  // stale
7    for (int[] k : adj.get(u)) {
       int v = k[0];
       long alt = d[u] + k[1];
       if (alt < d[v]) {
         d[v] = alt;
         prev[v] = u;
         pq.add(new long[]{v, alt});
14     }
15   }
16 }
```

VARIABLES

`e = {A,0}` `u = A` `v = F` `alt = 7`



	A	B	C	D	E	F	G	H	I
d[]	0	4	∞	∞	∞	7	∞	∞	∞
prev	–	A	–	–	–	A	–	–	–

pq (shown sorted by distance; faded = stale)

B,4 F,7

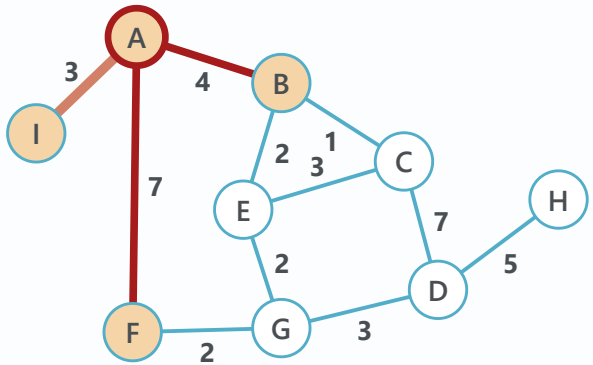
v = F: alt = 0 + 7 = **7** < ∞ → d[F] = 7, prev[F] = A, {F, 7} enters the queue.

Code Trace: `dijkstra(adj, A)`

```
1  // d[] = ∞, d[s] = 0, prev[] = -1
2  pq.add(new long[]{s, 0});
3  while (!pq.isEmpty()) {
4    long[] e = pq.poll();
5    int u = (int) e[0];
6    if (e[1] > d[u]) continue;  // stale
7    for (int[] k : adj.get(u)) {
       int v = k[0];
       long alt = d[u] + k[1];
       if (alt < d[v]) {
         d[v] = alt;
         prev[v] = u;
         pq.add(new long[]{v, alt});
14     }
15   }
16 }
```

VARIABLES

`e = {A,0}` `u = A` `v = I` `alt = 3`



	A	B	C	D	E	F	G	H	I
d[]	0	4	∞	∞	∞	7	∞	∞	3
prev	–	A	–	–	–	A	–	–	A

pq (shown sorted by distance; faded = stale)

I,3 | B,4 | F,7

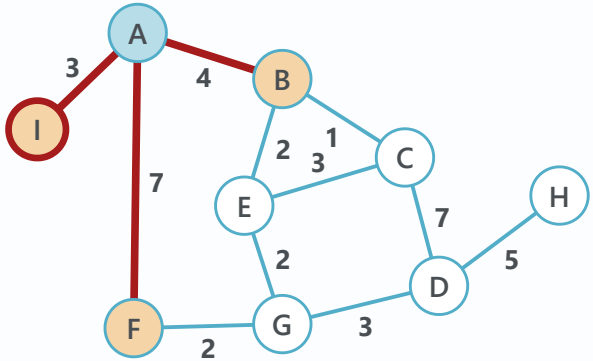
v = I: alt = 0 + 3 = **3** < ∞ → d[I] = 3, prev[I] = A, {I, 3} enters the queue.

Code Trace: `dijkstra(adj, A)`

```
1  // d[] = ∞, d[s] = 0, prev[] = -1
2  pq.add(new long[]{s, 0});
3  while (!pq.isEmpty()) {
▶    long[] e = pq.poll();
▶    int u = (int) e[0];
▶    if (e[1] > d[u]) continue;  // stale
7    for (int[] k : adj.get(u)) {
8      int v = k[0];
9      long alt = d[u] + k[1];
10     if (alt < d[v]) {
11       d[v] = alt;
12       prev[v] = u;
13       pq.add(new long[]{v, alt});
14     }
15   }
16 }
```

VARIABLES

`e = {I,3}` `u = I`



	A	B	C	D	E	F	G	H	I
d[]	0	4	∞	∞	∞	7	∞	∞	3
prev	–	A	–	–	–	A	–	–	A

pq (shown sorted by distance; faded = stale)

B,4 F,7

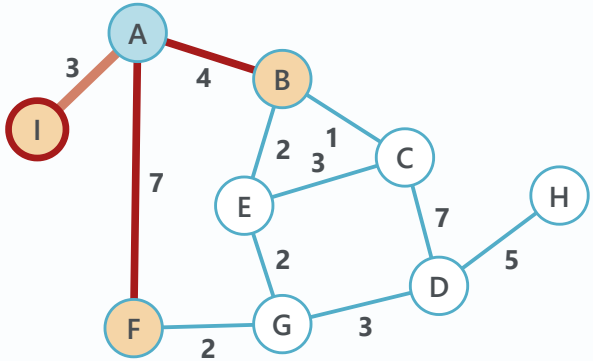
Poll the smallest entry {I, 3}; 3 = d[I] (current) → **I is settled**. Relax its neighbor A.

Code Trace: `dijkstra(adj, A)`

```
1  // d[] = ∞, d[s] = 0, prev[] = -1
2  pq.add(new long[]{s, 0});
3  while (!pq.isEmpty()) {
4    long[] e = pq.poll();
5    int u = (int) e[0];
6    if (e[1] > d[u]) continue;  // stale
7    for (int[] k : adj.get(u)) {
▶      int v = k[0];
▶      long alt = d[u] + k[1];
▶      if (alt < d[v]) {
11       d[v] = alt;
12       prev[v] = u;
13       pq.add(new long[]{v, alt});
14     }
15   }
16 }
```

VARIABLES

`e = {I,3}` `u = I` `v = A` `alt = 6`



	A	B	C	D	E	F	G	H	I
d[]	0	4	∞	∞	∞	7	∞	∞	3
prev	–	A	–	–	–	A	–	–	A

pq (shown sorted by distance; faded = stale)

B,4 F,7

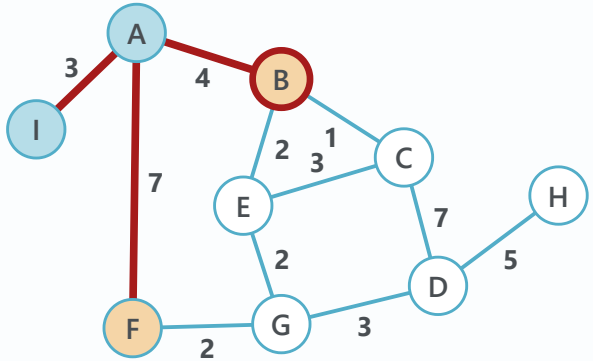
v = A: alt = 6 ≥ d[A] = 0 → no update.

Code Trace: `dijkstra(adj, A)`

```
1  // d[] = ∞, d[s] = 0, prev[] = -1
2  pq.add(new long[]{s, 0});
3  while (!pq.isEmpty()) {
▶    long[] e = pq.poll();
▶    int u = (int) e[0];
▶    if (e[1] > d[u]) continue;  // stale
7    for (int[] k : adj.get(u)) {
8      int v = k[0];
9      long alt = d[u] + k[1];
10     if (alt < d[v]) {
11       d[v] = alt;
12       prev[v] = u;
13       pq.add(new long[]{v, alt});
14     }
15   }
16 }
```

VARIABLES

`e = {B,4}` `u = B`



	A	B	C	D	E	F	G	H	I
d[]	0	4	∞	∞	∞	7	∞	∞	3
prev	–	A	–	–	–	A	–	–	A

pq (shown sorted by distance; faded = stale)

F,7

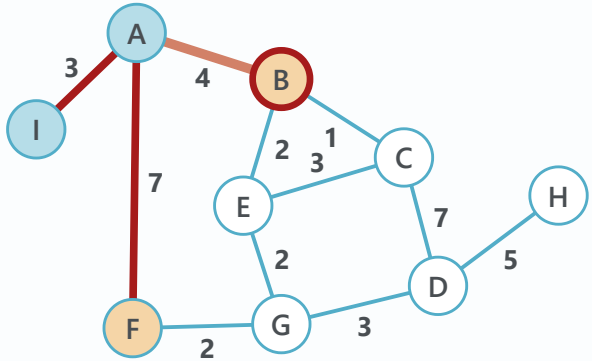
Poll the smallest entry {B, 4}; 4 = d[B] (current) → **B is settled**. Relax its neighbors A, C, E.

Code Trace: `dijkstra(adj, A)`

```
1  // d[] = ∞, d[s] = 0, prev[] = -1
2  pq.add(new long[]{s, 0});
3  while (!pq.isEmpty()) {
4    long[] e = pq.poll();
5    int u = (int) e[0];
6    if (e[1] > d[u]) continue;  // stale
7    for (int[] k : adj.get(u)) {
▶      int v = k[0];
▶      long alt = d[u] + k[1];
▶      if (alt < d[v]) {
11       d[v] = alt;
12       prev[v] = u;
13       pq.add(new long[]{v, alt});
14     }
15   }
16 }
```

VARIABLES

`e = {B,4}` `u = B` `v = A` `alt = 8`



	A	B	C	D	E	F	G	H	I
d[]	0	4	∞	∞	∞	7	∞	∞	3
prev	–	A	–	–	–	A	–	–	A

pq (shown sorted by distance; faded = stale)

F,7

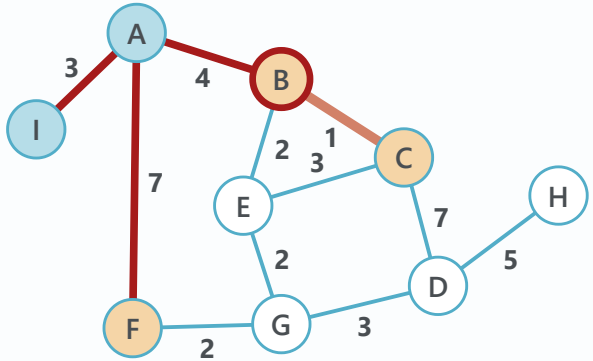
v = A: alt = 8 ≥ d[A] = 0 → no update.

Code Trace: `dijkstra(adj, A)`

```
1  // d[] = ∞, d[s] = 0, prev[] = -1
2  pq.add(new long[]{s, 0});
3  while (!pq.isEmpty()) {
4    long[] e = pq.poll();
5    int u = (int) e[0];
6    if (e[1] > d[u]) continue;  // stale
7    for (int[] k : adj.get(u)) {
       int v = k[0];
       long alt = d[u] + k[1];
       if (alt < d[v]) {
         d[v] = alt;
         prev[v] = u;
         pq.add(new long[]{v, alt});
14     }
15   }
16 }
```

VARIABLES

`e = {B,4}` `u = B` `v = C` `alt = 5`



	A	B	C	D	E	F	G	H	I
d[]	0	4	5	∞	∞	7	∞	∞	3
prev	–	A	B	–	–	A	–	–	A

pq (shown sorted by distance; faded = stale)

C,5 F,7

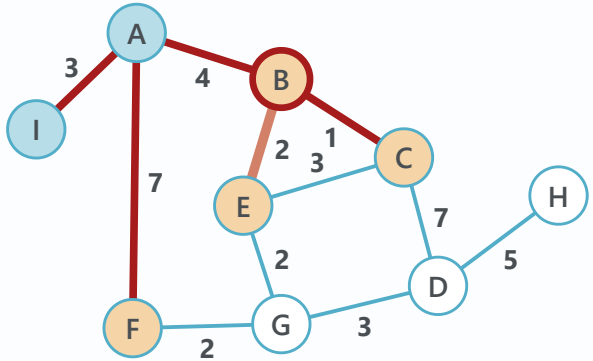
v = C: alt = 4 + 1 = **5** < ∞ → d[C] = 5, prev[C] = B, {C, 5} enters the queue.

Code Trace: `dijkstra(adj, A)`

```
1  // d[] = ∞, d[s] = 0, prev[] = -1
2  pq.add(new long[]{s, 0});
3  while (!pq.isEmpty()) {
4    long[] e = pq.poll();
5    int u = (int) e[0];
6    if (e[1] > d[u]) continue;  // stale
7    for (int[] k : adj.get(u)) {
       int v = k[0];
       long alt = d[u] + k[1];
       if (alt < d[v]) {
         d[v] = alt;
         prev[v] = u;
         pq.add(new long[]{v, alt});
14     }
15   }
16 }
```

VARIABLES

`e = {B,4}` `u = B` `v = E` `alt = 6`



	A	B	C	D	E	F	G	H	I
d[]	0	4	5	∞	6	7	∞	∞	3
prev	–	A	B	–	B	A	–	–	A

pq (shown sorted by distance; faded = stale)

C,5 | E,6 | F,7

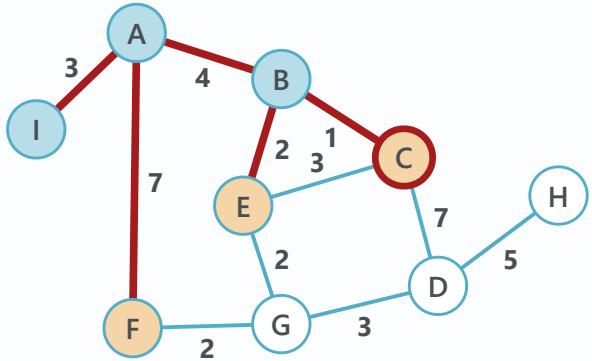
v = E: alt = 4 + 2 = **6** < ∞ → d[E] = 6, prev[E] = B, {E, 6} enters the queue.

Code Trace: `dijkstra(adj, A)`

```
1  // d[] = ∞, d[s] = 0, prev[] = -1
2  pq.add(new long[]{s, 0});
3  while (!pq.isEmpty()) {
▶    long[] e = pq.poll();
▶    int u = (int) e[0];
▶    if (e[1] > d[u]) continue;  // stale
7    for (int[] k : adj.get(u)) {
8      int v = k[0];
9      long alt = d[u] + k[1];
10     if (alt < d[v]) {
11       d[v] = alt;
12       prev[v] = u;
13       pq.add(new long[]{v, alt});
14     }
15   }
16 }
```

VARIABLES

`e = {C,5}` `u = C`



	A	B	C	D	E	F	G	H	I
d[]	0	4	5	∞	6	7	∞	∞	3
prev	–	A	B	–	B	A	–	–	A

pq (shown sorted by distance; faded = stale)

E,6 F,7

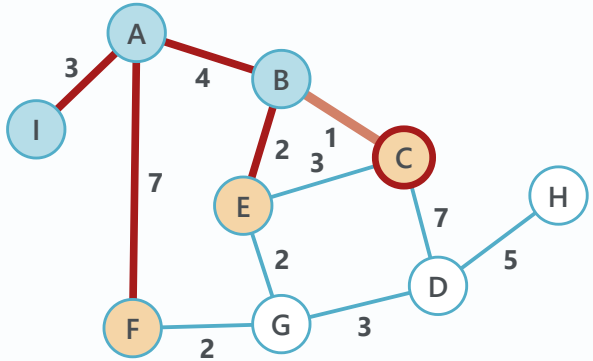
Poll the smallest entry {C, 5}; 5 = d[C] (current) → **C is settled**. Relax its neighbors B, D, E.

Code Trace: `dijkstra(adj, A)`

```
1  // d[] = ∞, d[s] = 0, prev[] = -1
2  pq.add(new long[]{s, 0});
3  while (!pq.isEmpty()) {
4    long[] e = pq.poll();
5    int u = (int) e[0];
6    if (e[1] > d[u]) continue;  // stale
7    for (int[] k : adj.get(u)) {
▶      int v = k[0];
▶      long alt = d[u] + k[1];
▶      if (alt < d[v]) {
11       d[v] = alt;
12       prev[v] = u;
13       pq.add(new long[]{v, alt});
14     }
15   }
16 }
```

VARIABLES

`e = {C,5}` `u = C` `v = B` `alt = 6`



	A	B	C	D	E	F	G	H	I
d[]	0	4	5	∞	6	7	∞	∞	3
prev	–	A	B	–	B	A	–	–	A

pq (shown sorted by distance; faded = stale)

E,6 F,7

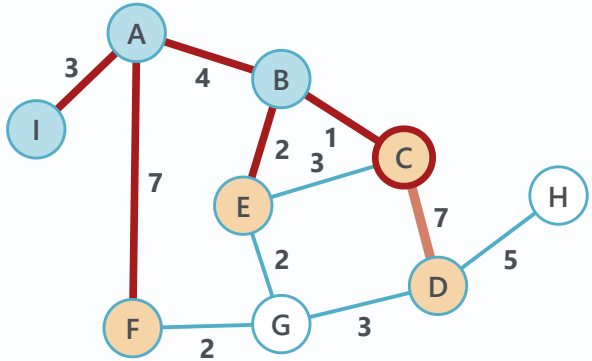
v = B: alt = 6 ≥ d[B] = 4 → no update.

Code Trace: `dijkstra(adj, A)`

```
1  // d[] = ∞, d[s] = 0, prev[] = -1
2  pq.add(new long[]{s, 0});
3  while (!pq.isEmpty()) {
4    long[] e = pq.poll();
5    int u = (int) e[0];
6    if (e[1] > d[u]) continue;  // stale
7    for (int[] k : adj.get(u)) {
       int v = k[0];
       long alt = d[u] + k[1];
       if (alt < d[v]) {
         d[v] = alt;
         prev[v] = u;
         pq.add(new long[]{v, alt});
14     }
15   }
16 }
```

VARIABLES

`e = {C,5}` `u = C` `v = D` `alt = 12`



	A	B	C	D	E	F	G	H	I
d[]	0	4	5	12	6	7	∞	∞	3
prev	–	A	B	C	B	A	–	–	A

pq (shown sorted by distance; faded = stale)

E,6 | F,7 | D,12

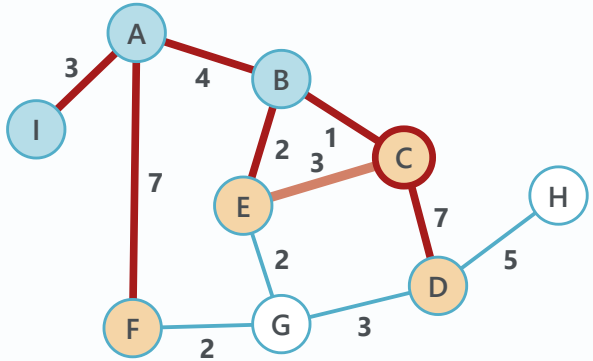
v = D: alt = 5 + 7 = **12** < ∞ → d[D] = 12, prev[D] = C, {D, 12} enters the queue.

Code Trace: `dijkstra(adj, A)`

```
1  // d[] = ∞, d[s] = 0, prev[] = -1
2  pq.add(new long[]{s, 0});
3  while (!pq.isEmpty()) {
4    long[] e = pq.poll();
5    int u = (int) e[0];
6    if (e[1] > d[u]) continue;  // stale
7    for (int[] k : adj.get(u)) {
▶      int v = k[0];
▶      long alt = d[u] + k[1];
▶      if (alt < d[v]) {
11       d[v] = alt;
12       prev[v] = u;
13       pq.add(new long[]{v, alt});
14     }
15   }
16 }
```

VARIABLES

`e = {C,5}` `u = C` `v = E` `alt = 8`



	A	B	C	D	E	F	G	H	I
d[]	0	4	5	12	6	7	∞	∞	3
prev	–	A	B	C	B	A	–	–	A

pq (shown sorted by distance; faded = stale)

E,6 | F,7 | D,12

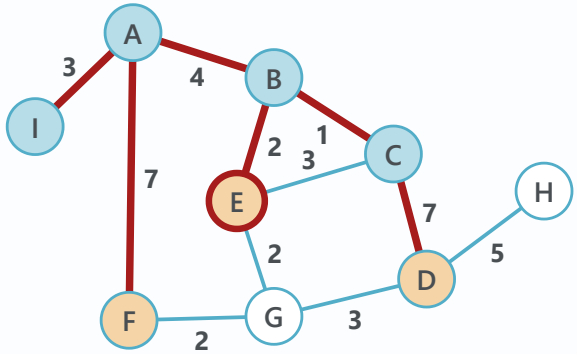
v = E: alt = 8 ≥ d[E] = 6 → no update.

Code Trace: `dijkstra(adj, A)` (continued)

```
1  // d[] = ∞, d[s] = 0, prev[] = -1
2  pq.add(new long[]{s, 0});
3  while (!pq.isEmpty()) {
▶    long[] e = pq.poll();
▶    int u = (int) e[0];
▶    if (e[1] > d[u]) continue;  // stale
7    for (int[] k : adj.get(u)) {
8      int v = k[0];
9      long alt = d[u] + k[1];
10     if (alt < d[v]) {
11       d[v] = alt;
12       prev[v] = u;
13       pq.add(new long[]{v, alt});
14     }
15   }
16 }
```

VARIABLES

`e = {E,6}` `u = E`



	A	B	C	D	E	F	G	H	I
d[]	0	4	5	12	6	7	∞	∞	3
prev	–	A	B	C	B	A	–	–	A

pq (shown sorted by distance; faded = stale)

F,7 D,12

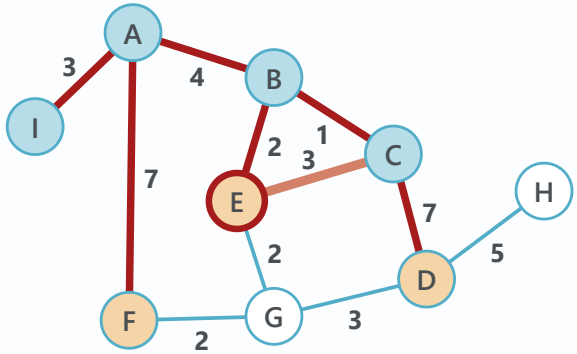
Poll the smallest entry {E, 6}; 6 = d[E] (current) → **E is settled**. Relax its neighbors B, C, G.

Code Trace: `dijkstra(adj, A)` (continued)

```
1  // d[] = ∞, d[s] = 0, prev[] = -1
2  pq.add(new long[]{s, 0});
3  while (!pq.isEmpty()) {
4    long[] e = pq.poll();
5    int u = (int) e[0];
6    if (e[1] > d[u]) continue;  // stale
7    for (int[] k : adj.get(u)) {
▶      int v = k[0];
▶      long alt = d[u] + k[1];
▶      if (alt < d[v]) {
11       d[v] = alt;
12       prev[v] = u;
13       pq.add(new long[]{v, alt});
14     }
15   }
16 }
```

VARIABLES

`e = {E,6}` `u = E` `v = C` `alt = 9`



	A	B	C	D	E	F	G	H	I
d[]	0	4	5	12	6	7	∞	∞	3
prev	–	A	B	C	B	A	–	–	A

pq (shown sorted by distance; faded = stale)

F,7 D,12

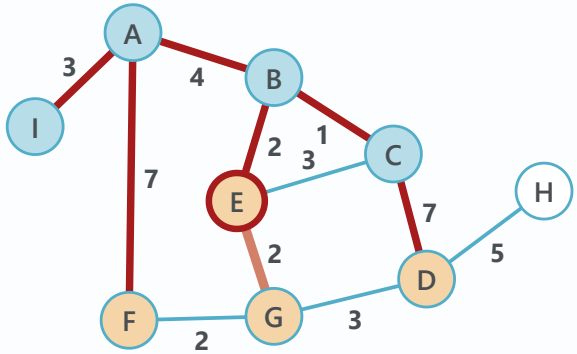
v = B: alt = 8 ≥ d[B] = 4; v = C: alt = 9 ≥ d[C] = 5 → no update.

Code Trace: `dijkstra(adj, A)` (continued)

```
1  // d[] = ∞, d[s] = 0, prev[] = -1
2  pq.add(new long[]{s, 0});
3  while (!pq.isEmpty()) {
4    long[] e = pq.poll();
5    int u = (int) e[0];
6    if (e[1] > d[u]) continue;  // stale
7    for (int[] k : adj.get(u)) {
▶      int v = k[0];
▶      long alt = d[u] + k[1];
▶      if (alt < d[v]) {
▶        d[v] = alt;
▶        prev[v] = u;
▶        pq.add(new long[]{v, alt});
14     }
15   }
16 }
```

VARIABLES

`e = {E,6}` `u = E` `v = G` `alt = 8`



	A	B	C	D	E	F	G	H	I
d[]	0	4	5	12	6	7	8	∞	3
prev	–	A	B	C	B	A	E	–	A

pq (shown sorted by distance; faded = stale)

F,7 | G,8 | D,12

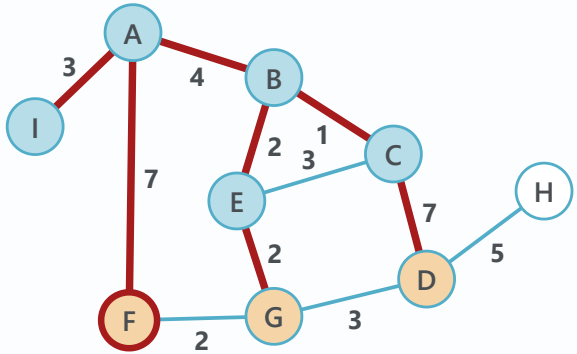
v = G: alt = 6 + 2 = **8** < ∞ → d[G] = 8, prev[G] = E, {G, 8} enters the queue.

Code Trace: `dijkstra(adj, A)` (continued)

```
1  // d[] = ∞, d[s] = 0, prev[] = -1
2  pq.add(new long[]{s, 0});
3  while (!pq.isEmpty()) {
▶    long[] e = pq.poll();
▶    int u = (int) e[0];
▶    if (e[1] > d[u]) continue;  // stale
7    for (int[] k : adj.get(u)) {
8      int v = k[0];
9      long alt = d[u] + k[1];
10     if (alt < d[v]) {
11       d[v] = alt;
12       prev[v] = u;
13       pq.add(new long[]{v, alt});
14     }
15   }
16 }
```

VARIABLES

`e = {F,7}` `u = F`



	A	B	C	D	E	F	G	H	I
d[]	0	4	5	12	6	7	8	∞	3
prev	–	A	B	C	B	A	E	–	A

pq (shown sorted by distance; faded = stale)

G,8 D,12

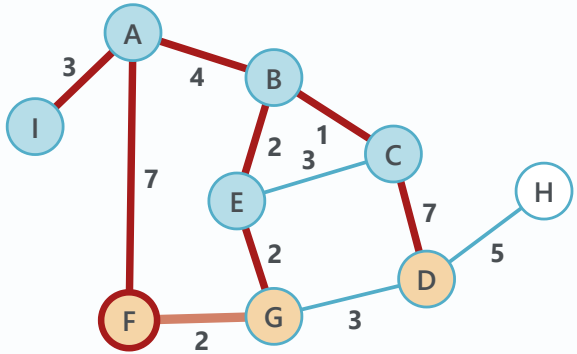
Poll the smallest entry {F, 7}; 7 = d[F] (current) → **F is settled**. Relax its neighbors A, G.

Code Trace: `dijkstra(adj, A)` (continued)

```
1  // d[] = ∞, d[s] = 0, prev[] = -1
2  pq.add(new long[]{s, 0});
3  while (!pq.isEmpty()) {
4    long[] e = pq.poll();
5    int u = (int) e[0];
6    if (e[1] > d[u]) continue;  // stale
7    for (int[] k : adj.get(u)) {
▶      int v = k[0];
▶      long alt = d[u] + k[1];
▶      if (alt < d[v]) {
11       d[v] = alt;
12       prev[v] = u;
13       pq.add(new long[]{v, alt});
14     }
15   }
16 }
```

VARIABLES

`e = {F,7}` `u = F` `v = G` `alt = 9`



	A	B	C	D	E	F	G	H	I
d[]	0	4	5	12	6	7	8	∞	3
prev	–	A	B	C	B	A	E	–	A

pq (shown sorted by distance; faded = stale)

G,8 D,12

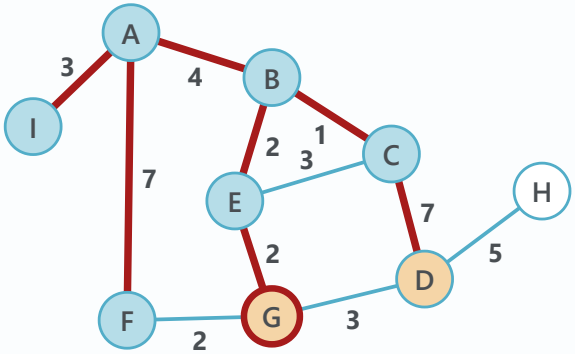
v = A: alt = 14 ≥ d[A] = 0; v = G: alt = 9 ≥ d[G] = 8 → no update.

Code Trace: `dijkstra(adj, A)` (continued)

```
1  // d[] = ∞, d[s] = 0, prev[] = -1
2  pq.add(new long[]{s, 0});
3  while (!pq.isEmpty()) {
▶    long[] e = pq.poll();
▶    int u = (int) e[0];
▶    if (e[1] > d[u]) continue;  // stale
7    for (int[] k : adj.get(u)) {
8      int v = k[0];
9      long alt = d[u] + k[1];
10     if (alt < d[v]) {
11       d[v] = alt;
12       prev[v] = u;
13       pq.add(new long[]{v, alt});
14     }
15   }
16 }
```

VARIABLES

`e = {G,8}` `u = G`



	A	B	C	D	E	F	G	H	I
d[]	0	4	5	12	6	7	8	∞	3
prev	–	A	B	C	B	A	E	–	A

pq (shown sorted by distance; faded = stale)

D,12

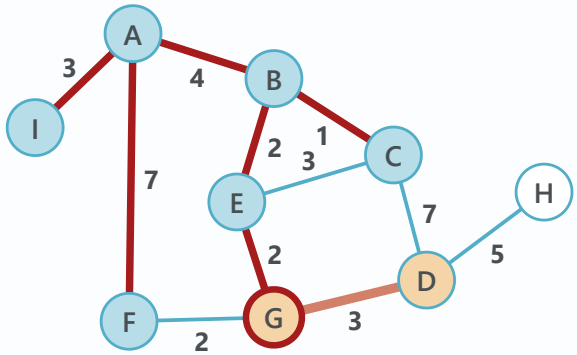
Poll the smallest entry {G, 8}; 8 = d[G] (current) → **G is settled**. Relax its neighbors D, E, F.

Code Trace: `dijkstra(adj, A)` (continued)

```
1  // d[] = ∞, d[s] = 0, prev[] = -1
2  pq.add(new long[]{s, 0});
3  while (!pq.isEmpty()) {
4    long[] e = pq.poll();
5    int u = (int) e[0];
6    if (e[1] > d[u]) continue;  // stale
7    for (int[] k : adj.get(u)) {
       int v = k[0];
       long alt = d[u] + k[1];
       if (alt < d[v]) {
         d[v] = alt;
         prev[v] = u;
         pq.add(new long[]{v, alt});
14     }
15   }
16 }
```

VARIABLES

`e = {G,8}` `u = G` `v = D` `alt = 11`



	A	B	C	D	E	F	G	H	I
d[]	0	4	5	11	6	7	8	∞	3
prev	–	A	B	G	B	A	E	–	A

pq (shown sorted by distance; faded = stale)

D,11 ~~D,12~~

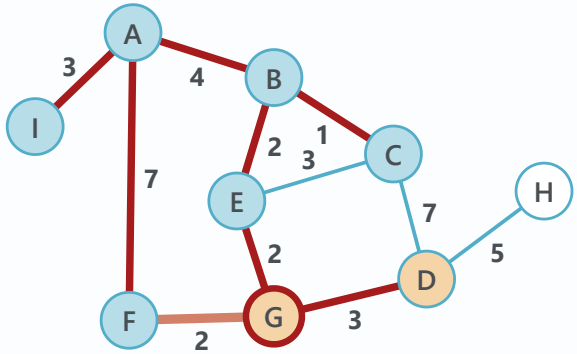
v = D: alt = 8 + 3 = **11** < 12 → d[D] = 11, prev[D] = G, {D, 11} enters the queue. The old entry {D, 12} stays in the queue as **stale**.

Code Trace: `dijkstra(adj, A)` (continued)

```
1  // d[] = ∞, d[s] = 0, prev[] = -1
2  pq.add(new long[]{s, 0});
3  while (!pq.isEmpty()) {
4    long[] e = pq.poll();
5    int u = (int) e[0];
6    if (e[1] > d[u]) continue;  // stale
7    for (int[] k : adj.get(u)) {
▶      int v = k[0];
▶      long alt = d[u] + k[1];
▶      if (alt < d[v]) {
11       d[v] = alt;
12       prev[v] = u;
13       pq.add(new long[]{v, alt});
14     }
15   }
16 }
```

VARIABLES

`e = {G,8}` `u = G` `v = F` `alt = 10`



	A	B	C	D	E	F	G	H	I
d[]	0	4	5	11	6	7	8	∞	3
prev	–	A	B	G	B	A	E	–	A

pq (shown sorted by distance; faded = stale)

D,11 ~~D,12~~

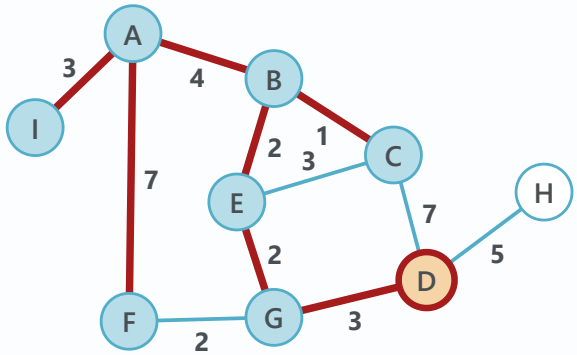
v = E: alt = 10 ≥ d[E] = 6; v = F: alt = 10 ≥ d[F] = 7 → no update.

Code Trace: `dijkstra(adj, A)` (continued)

```
1  // d[] = ∞, d[s] = 0, prev[] = -1
2  pq.add(new long[]{s, 0});
3  while (!pq.isEmpty()) {
▶    long[] e = pq.poll();
▶    int u = (int) e[0];
▶    if (e[1] > d[u]) continue;  // stale
7    for (int[] k : adj.get(u)) {
8      int v = k[0];
9      long alt = d[u] + k[1];
10     if (alt < d[v]) {
11       d[v] = alt;
12       prev[v] = u;
13       pq.add(new long[]{v, alt});
14     }
15   }
16 }
```

VARIABLES

`e = {D,11}` `u = D`



	A	B	C	D	E	F	G	H	I
d[]	0	4	5	11	6	7	8	∞	3
prev	–	A	B	G	B	A	E	–	A

pq (shown sorted by distance; faded = stale)

~~D,12~~

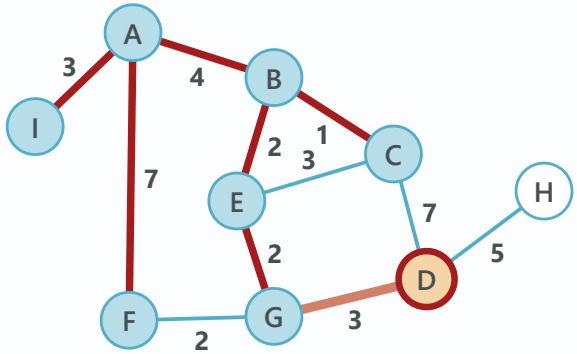
Poll the smallest entry {D, 11}; 11 = d[D] (current) → **D is settled**. Relax its neighbors C, G, H.

Code Trace: `dijkstra(adj, A)` (continued)

```
1  // d[] = ∞, d[s] = 0, prev[] = -1
2  pq.add(new long[]{s, 0});
3  while (!pq.isEmpty()) {
4    long[] e = pq.poll();
5    int u = (int) e[0];
6    if (e[1] > d[u]) continue;  // stale
7    for (int[] k : adj.get(u)) {
▶      int v = k[0];
▶      long alt = d[u] + k[1];
▶      if (alt < d[v]) {
11       d[v] = alt;
12       prev[v] = u;
13       pq.add(new long[]{v, alt});
14     }
15   }
16 }
```

VARIABLES

`e = {D,11}` `u = D` `v = G` `alt = 14`



	A	B	C	D	E	F	G	H	I
d[]	0	4	5	11	6	7	8	∞	3
prev	–	A	B	G	B	A	E	–	A

pq (shown sorted by distance; faded = stale)

~~D,12~~

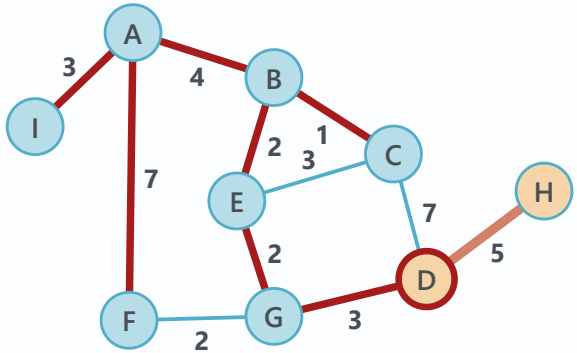
v = C: alt = 18 ≥ d[C] = 5; v = G: alt = 14 ≥ d[G] = 8 → no update.

Code Trace: `dijkstra(adj, A)` (continued)

```
1  // d[] = ∞, d[s] = 0, prev[] = -1
2  pq.add(new long[]{s, 0});
3  while (!pq.isEmpty()) {
4    long[] e = pq.poll();
5    int u = (int) e[0];
6    if (e[1] > d[u]) continue;  // stale
7    for (int[] k : adj.get(u)) {
▶      int v = k[0];
▶      long alt = d[u] + k[1];
▶      if (alt < d[v]) {
▶        d[v] = alt;
▶        prev[v] = u;
▶        pq.add(new long[]{v, alt});
14     }
15   }
16 }
```

VARIABLES

`e = {D,11}` `u = D` `v = H` `alt = 16`



	A	B	C	D	E	F	G	H	I
d[]	0	4	5	11	6	7	8	16	3
prev	–	A	B	G	B	A	E	D	A

pq (shown sorted by distance; faded = stale)

~~D,12~~ H,16

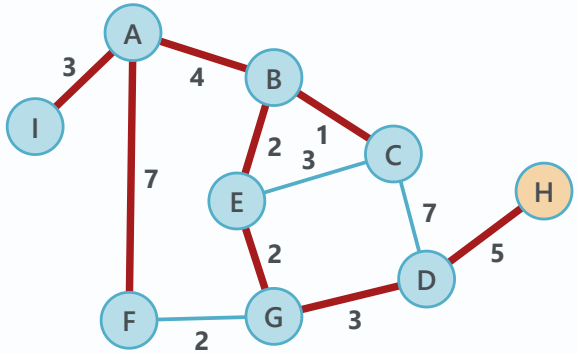
v = H: alt = 11 + 5 = **16** < ∞ → d[H] = 16, prev[H] = D, {H, 16} enters the queue.

Code Trace: `dijkstra(adj, A)` (continued)

```
1  // d[] = ∞, d[s] = 0, prev[] = -1
2  pq.add(new long[]{s, 0});
3  while (!pq.isEmpty()) {
▶    long[] e = pq.poll();
▶    int u = (int) e[0];
▶    if (e[1] > d[u]) continue;  // stale
7    for (int[] k : adj.get(u)) {
8      int v = k[0];
9      long alt = d[u] + k[1];
10     if (alt < d[v]) {
11       d[v] = alt;
12       prev[v] = u;
13       pq.add(new long[]{v, alt});
14     }
15   }
16 }
```

VARIABLES

`e = {D,12}` `u = D`



	A	B	C	D	E	F	G	H	I
d[]	0	4	5	11	6	7	8	16	3
prev	–	A	B	G	B	A	E	D	A

pq (shown sorted by distance; faded = stale)

H,16

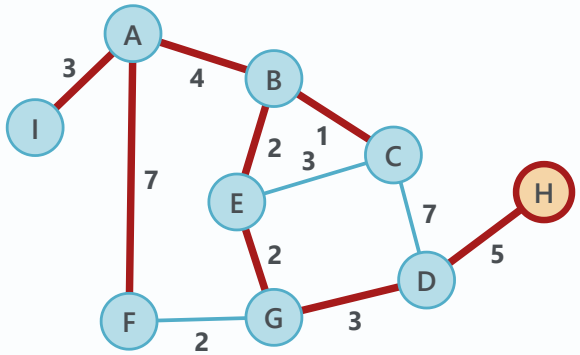
{D, 12} comes out, but d[D] = 11 < 12: a **stale entry** (a better path was found later) → `continue` skips it.

Code Trace: `dijkstra(adj, A)` (continued)

```
1  // d[] = ∞, d[s] = 0, prev[] = -1
2  pq.add(new long[]{s, 0});
3  while (!pq.isEmpty()) {
▶    long[] e = pq.poll();
▶    int u = (int) e[0];
▶    if (e[1] > d[u]) continue;  // stale
7    for (int[] k : adj.get(u)) {
8      int v = k[0];
9      long alt = d[u] + k[1];
10     if (alt < d[v]) {
11       d[v] = alt;
12       prev[v] = u;
13       pq.add(new long[]{v, alt});
14     }
15   }
16 }
```

VARIABLES

`e = {H,16}` `u = H`



	A	B	C	D	E	F	G	H	I
d[]	0	4	5	11	6	7	8	16	3
prev	–	A	B	G	B	A	E	D	A

pq (shown sorted by distance; faded = stale)

(empty)

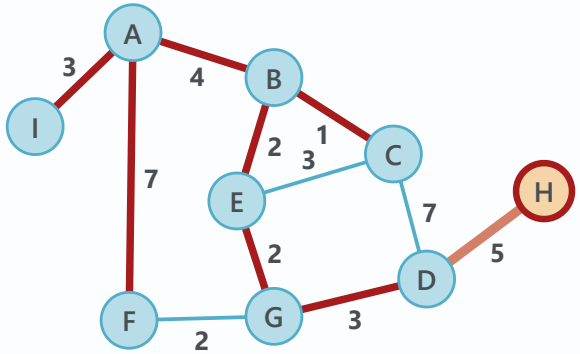
Poll the smallest entry {H, 16}; 16 = d[H] (current) → **H is settled**. Relax its neighbor D.

Code Trace: `dijkstra(adj, A)` (continued)

```
1  // d[] = ∞, d[s] = 0, prev[] = -1
2  pq.add(new long[]{s, 0});
3  while (!pq.isEmpty()) {
4    long[] e = pq.poll();
5    int u = (int) e[0];
6    if (e[1] > d[u]) continue;  // stale
7    for (int[] k : adj.get(u)) {
▶      int v = k[0];
▶      long alt = d[u] + k[1];
▶      if (alt < d[v]) {
11       d[v] = alt;
12       prev[v] = u;
13       pq.add(new long[]{v, alt});
14     }
15   }
16 }
```

VARIABLES

`e = {H,16}` `u = H` `v = D` `alt = 21`



	A	B	C	D	E	F	G	H	I
d[]	0	4	5	11	6	7	8	16	3
prev	–	A	B	G	B	A	E	D	A

pq (shown sorted by distance; faded = stale)

(empty)

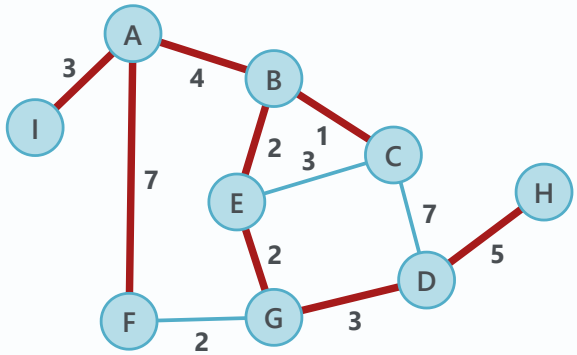
v = D: alt = 21 ≥ d[D] = 11 → no update.

Code Trace: `dijkstra(adj, A)` (continued)

```
1  // d[] = ∞, d[s] = 0, prev[] = -1
2  pq.add(new long[]{s, 0});
▶  while (!pq.isEmpty()) {
4    long[] e = pq.poll();
5    int u = (int) e[0];
6    if (e[1] > d[u]) continue;  // stale
7    for (int[] k : adj.get(u)) {
8      int v = k[0];
9      long alt = d[u] + k[1];
10     if (alt < d[v]) {
11       d[v] = alt;
12       prev[v] = u;
13       pq.add(new long[]{v, alt});
14     }
15   }
16 }
```

VARIABLES

—



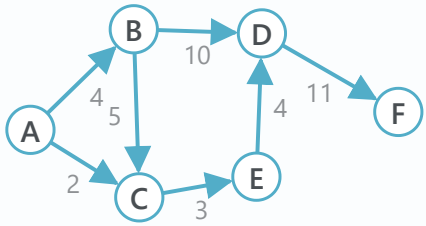
	A	B	C	D	E	F	G	H	I
d[]	0	4	5	11	6	7	8	16	3
prev	–	A	B	G	B	A	E	D	A

pq (shown sorted by distance; faded = stale)

(empty)

The queue is empty → done. Trace prev[] back: H ← D ← G ← E ← B ← A. The shortest path is **A → B → E → G → D → H = 16**.

Code Trace: Printing the Shortest Path with `prev[]`



v	A	B	C	D	E	F
prev	–	A	A	E	C	D
y						

addFirst → from target back to source, insert at front

The `prev[]` array Dijkstra leaves behind: for each vertex, the vertex *just before it* on its shortest path. List p starts empty.

Step 1 / 14

```
1 List<Integer> path(int[] prev, int s, int t) {
▶   LinkedList<Integer> p = new LinkedList<>();
3   for (int v = t; v != -1; v = prev[v])
4     p.addFirst(v);
5   if (p.getFirst() != s) return List.of();
6   return p;                  // s ... t
7 }
```

VARIABLES

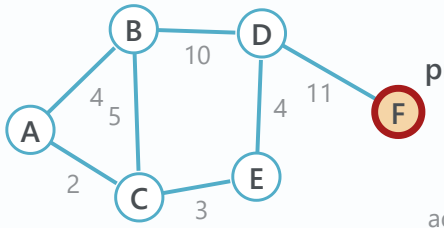
`prev = [–, A, A, E, C, D]` `p = []`

CALL STACK

`path(prev, A, F)`

`main()`

Code Trace: Printing the Shortest Path with `prev[]`



v	A	B	C	D	E	F
prev	–	A	A	E	C	D
y						

addFirst → from target back to source, insert at front

Start at the target: **v = F**.

Step 2 / 14

```
1  List<Integer> path(int[] prev, int s, int t) {
2    LinkedList<Integer> p = new LinkedList<>();
▶    for (int v = t; v != -1; v = prev[v])
4      p.addFirst(v);
5    if (p.getFirst() != s) return List.of();
6    return p;                    // s ... t
7  }
```

VARIABLES

`prev = [–, A, A, E, C, D]` `v = F`

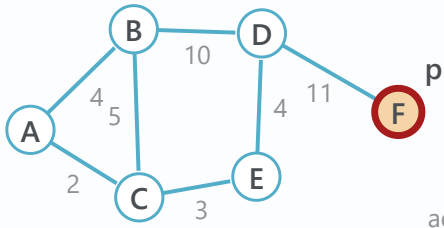
`p = []`

CALL STACK

`path(prev, A, F)`

`main()`

Code Trace: Printing the Shortest Path with `prev[]`



v	A	B	C	D	E	F
prev	–	A	A	E	C	D
y	F					

addFirst → from target back to source, insert at front

F goes to the **front** of the list (vertices added later come earlier on the path).

Step 3 / 14

```
1 List<Integer> path(int[] prev, int s, int t) {
2   LinkedList<Integer> p = new LinkedList<>();
3   for (int v = t; v != -1; v = prev[v])
▶     p.addFirst(v);
5   if (p.getFirst() != s) return List.of();
6   return p;                    // s ... t
7 }
```

VARIABLES

`prev = [–, A, A, E, C, D]` `v = F`

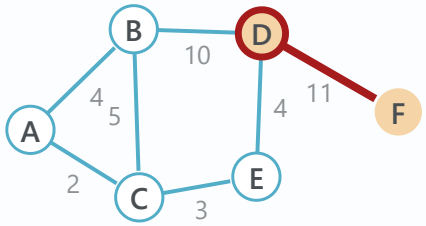
`p = [F]`

CALL STACK

`path(prev, A, F)`

`main()`

Code Trace: Printing the Shortest Path with `prev[]`



v	A	B	C	D	E	F
prev	–	A	A	E	C	D
y	F					

addFirst → from target back to source, insert at front

v = prev[...] = D: one step back.

Step 4 / 14

```
1 List<Integer> path(int[] prev, int s, int t) {
2   LinkedList<Integer> p = new LinkedList<>();
▶   for (int v = t; v != -1; v = prev[v])
4     p.addFirst(v);
5   if (p.getFirst() != s) return List.of();
6   return p;                    // s ... t
7 }
```

VARIABLES

`prev = [–, A, A, E, C, D]` `v = D`

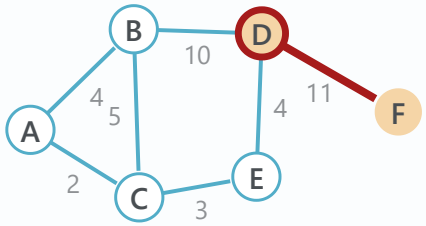
`p = [F]`

CALL STACK

`path(prev, A, F)`

`main()`

Code Trace: Printing the Shortest Path with `prev[]`



v	A	B	C	D	E	F
prev	–	A	A	E	C	D

y	D	F				

addFirst → from target back to source, insert at front

D goes to the **front** of the list (vertices added later come earlier on the path).

Step 5 / 14

```
1 List<Integer> path(int[] prev, int s, int t) {
2   LinkedList<Integer> p = new LinkedList<>();
3   for (int v = t; v != -1; v = prev[v])
▶     p.addFirst(v);
5   if (p.getFirst() != s) return List.of();
6   return p;                    // s ... t
7 }
```

VARIABLES

`prev = [–, A, A, E, C, D]` `v = D`

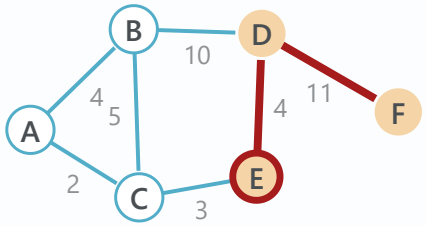
`p = [D, F]`

CALL STACK

`path(prev, A, F)`

`main()`

Code Trace: Printing the Shortest Path with `prev[]`



v	A	B	C	D	E	F
prev	–	A	A	E	C	D

y	D	F				

addFirst → from target back to source, insert at front

v = prev[...] = E: one step back.

Step 6 / 14

```
1 List<Integer> path(int[] prev, int s, int t) {
2   LinkedList<Integer> p = new LinkedList<>();
▶   for (int v = t; v != -1; v = prev[v])
4     p.addFirst(v);
5   if (p.getFirst() != s) return List.of();
6   return p;                    // s ... t
7 }
```

VARIABLES

`prev = [–, A, A, E, C, D]` `v = E`

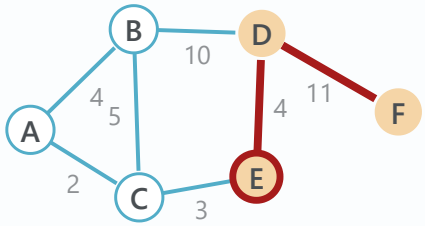
`p = [D, F]`

CALL STACK

`path(prev, A, F)`

`main()`

Code Trace: Printing the Shortest Path with `prev[]`



v	A	B	C	D	E	F
prev	–	A	A	E	C	D
y	E	D	F			

addFirst → from target back to source, insert at front

E goes to the **front** of the list (vertices added later come earlier on the path).

Step 7 / 14

```
1 List<Integer> path(int[] prev, int s, int t) {
2   LinkedList<Integer> p = new LinkedList<>();
3   for (int v = t; v != -1; v = prev[v])
▶     p.addFirst(v);
5   if (p.getFirst() != s) return List.of();
6   return p;                    // s ... t
7 }
```

VARIABLES

`prev = [–, A, A, E, C, D]` `v = E`

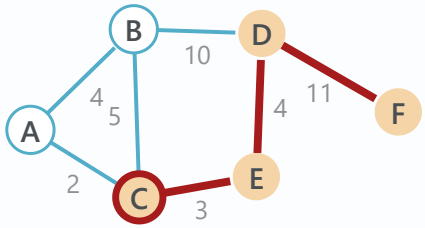
`p = [E, D, F]`

CALL STACK

`path(prev, A, F)`

`main()`

Code Trace: Printing the Shortest Path with `prev[]`



v	A	B	C	D	E	F
prev	–	A	A	E	C	D
y	E	D	F			

addFirst → from target back to source, insert at front

v = prev[...] = C: one step back.

Step 8 / 14

```
1 List<Integer> path(int[] prev, int s, int t) {
2   LinkedList<Integer> p = new LinkedList<>();
▶   for (int v = t; v != -1; v = prev[v])
4     p.addFirst(v);
5   if (p.getFirst() != s) return List.of();
6   return p;                    // s ... t
7 }
```

VARIABLES

`prev = [–, A, A, E, C, D]` `v = C`

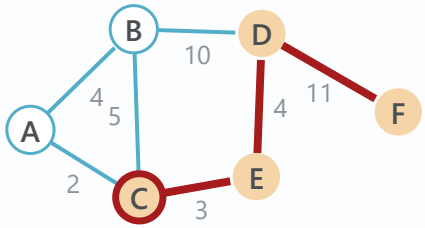
`p = [E, D, F]`

CALL STACK

`path(prev, A, F)`

`main()`

Code Trace: Printing the Shortest Path with `prev[]`



v	A	B	C	D	E	F
prev	–	A	A	E	C	D

y	C	E	D	F		

addFirst → from target back to source, insert at front

C goes to the **front** of the list (vertices added later come earlier on the path).

Step 9 / 14

```
1 List<Integer> path(int[] prev, int s, int t) {
2   LinkedList<Integer> p = new LinkedList<>();
3   for (int v = t; v != -1; v = prev[v])
▶     p.addFirst(v);
5   if (p.getFirst() != s) return List.of();
6   return p;                    // s ... t
7 }
```

VARIABLES

`prev = [–, A, A, E, C, D]` `v = C`

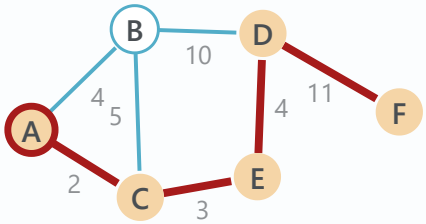
`p = [C, E, D, F]`

CALL STACK

`path(prev, A, F)`

`main()`

Code Trace: Printing the Shortest Path with `prev[]`



v	A	B	C	D	E	F
prev	–	A	A	E	C	D

y	C	E	D	F		

addFirst → from target back to source, insert at front

v = prev[...] = A: one step back.

Step 10 / 14

```
1 List<Integer> path(int[] prev, int s, int t) {
2   LinkedList<Integer> p = new LinkedList<>();
▶   for (int v = t; v != -1; v = prev[v])
4     p.addFirst(v);
5   if (p.getFirst() != s) return List.of();
6   return p;                    // s ... t
7 }
```

VARIABLES

`prev = [–, A, A, E, C, D]` `v = A`

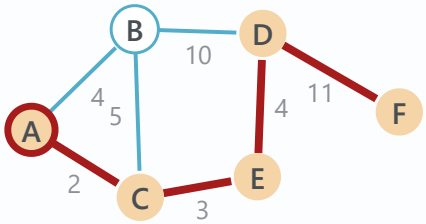
`p = [C, E, D, F]`

CALL STACK

`path(prev, A, F)`

`main()`

Code Trace: Printing the Shortest Path with `prev[]`



v	A	B	C	D	E	F
prev	–	A	A	E	C	D

y	A	C	E	D	F	

addFirst → from target back to source, insert at front

A goes to the **front** of the list (vertices added later come earlier on the path).

Step 11 / 14

```
1 List<Integer> path(int[] prev, int s, int t) {
2   LinkedList<Integer> p = new LinkedList<>();
3   for (int v = t; v != -1; v = prev[v])
▶     p.addFirst(v);
5   if (p.getFirst() != s) return List.of();
6   return p;                    // s ... t
7 }
```

VARIABLES

`prev = [–, A, A, E, C, D]` `v = A`

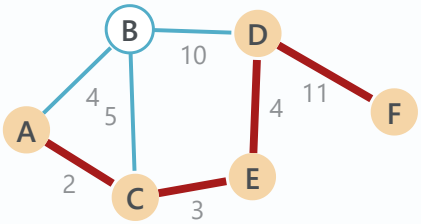
`p = [A, C, E, D, F]`

CALL STACK

`path(prev, A, F)`

`main()`

Code Trace: Printing the Shortest Path with `prev[]`



v	A	B	C	D	E	F
prev	–	A	A	E	C	D

y	A	C	E	D	F	

addFirst → from target back to source, insert at front

prev[A] = −1: we have reached the source, so the loop ends. 5 rounds = vertices on the path → O(path length).

Step 12 / 14

```
1 List<Integer> path(int[] prev, int s, int t) {
2   LinkedList<Integer> p = new LinkedList<>();
▶ for (int v = t; v != -1; v = prev[v])
4     p.addFirst(v);
5   if (p.getFirst() != s) return List.of();
6   return p;                  // s ... t
7 }
```

VARIABLES

`prev = [–, A, A, E, C, D]` `v = -1`

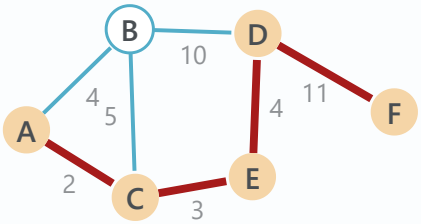
`p = [A, C, E, D, F]`

CALL STACK

`path(prev, A, F)`

`main()`

Code Trace: Printing the Shortest Path with `prev[]`



v	A	B	C	D	E	F
prev	–	A	A	E	C	D

y	A	C	E	D	F	

addFirst → from target back to source, insert at front

Is the first element the source A? Yes → t is reachable. (For an unreachable t, the chain never reaches A and an empty list is returned.)

Step 13 / 14

```
1 List<Integer> path(int[] prev, int s, int t) {
2   LinkedList<Integer> p = new LinkedList<>();
3   for (int v = t; v != -1; v = prev[v])
4     p.addFirst(v);
▶   if (p.getFirst() != s) return List.of();
6   return p;                    // s ... t
7 }
```

VARIABLES

`prev = [–, A, A, E, C, D]`

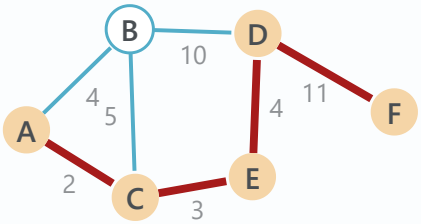
`p = [A, C, E, D, F]`

CALL STACK

`path(prev, A, F)`

`main()`

Code Trace: Printing the Shortest Path with `prev[]`



v	A	B	C	D	E	F
prev	–	A	A	E	C	D

y	A	C	E	D	F	

addFirst → from target back to source, insert at front

Result: **A** → **C** → **E** → **D** → **F**, length d[F] = 2 + 3 + 4 + 11 = **20**.

Step 14 / 14

```
1 List<Integer> path(int[] prev, int s, int t) {
2   LinkedList<Integer> p = new LinkedList<>();
3   for (int v = t; v != -1; v = prev[v])
4     p.addFirst(v);
5   if (p.getFirst() != s) return List.of();
▶   return p;                    // s ... t
7 }
```

VARIABLES

`prev = [–, A, A, E, C, D]`

`return = [A, C, E, D, F]`

CALL STACK

`path(prev, A, F)`

`main()`

Analyzing Dijkstra's Algorithm

Implementation	Complexity	Why?
Priority queue (binary heap)	$O((n + m) \log n)$	n remove-mins and at most m updates/inserts, each $O(\log n)$
No queue (array scan)	$O(n^2 + m) = O(n^2)$	Each step scans n vertices for the smallest distance; neighbor checks total m

Which one when?

For a sparse graph ($m \approx n$), the heap gives $O(n \log n)$. For a very dense graph ($m \approx n^2$), the $O(n^2)$ array scan can be better.

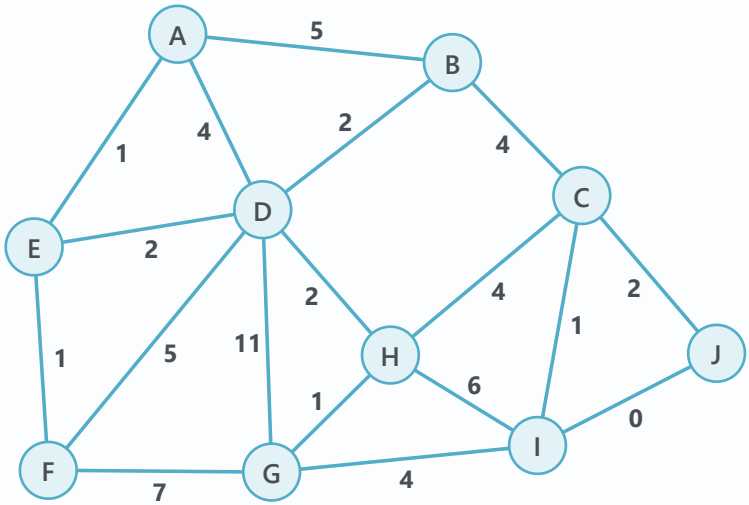
Negative weights

A settled distance could later shrink through a negative edge → the pseudocode version of Dijkstra gives wrong answers. The Java version with stale entries can reprocess a vertex and still find the right answer, but loses its time bound and never finishes on a negative cycle. Use Bellman-Ford instead.

Why does it work? With weights ≥ 0, the vertex with the smallest distance in the queue cannot be reached more cheaply through some other vertex: every other route is already at least that long.

Minimum Spanning Trees

- **Spanning tree:** a subgraph of a connected, weighted graph that contains **every vertex** and no cycle; it has n − 1 edges for n vertices.
- **Minimum spanning tree** (MST): a spanning tree with the **smallest** total edge weight.
- Application: connecting cities with the least cable or pipe.
- Kruskal's algorithm answers "Does this edge form a cycle?" with **Union-Find** (disjoint sets): `find(x)` returns the representative of x's set; `union(u, v)` merges two different sets and returns `true`, or returns `false` if they are already the same set.



Kruskal's Algorithm

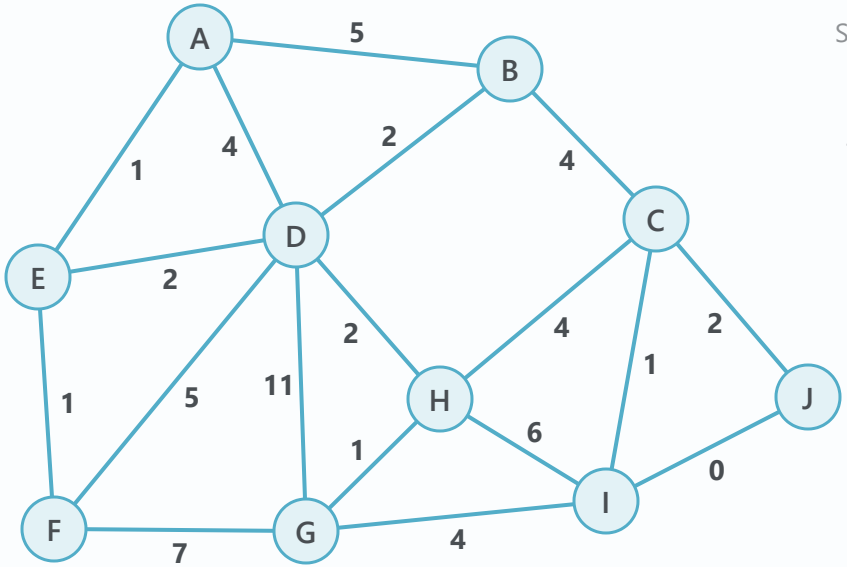
```
Kruskal(G):
  T = ∅          // chosen edges
  for each vertex v: makeSet(v)
  sort the edges by weight
  for each (u, v), lightest first:
    if find(u) != find(v): // no cycle
      T.add((u, v))
      union(u, v)
    if |T| == n - 1: break
  return T  // MST (if disconnected: forest)
```

- **Greedy**: always take the lightest edge that does not form a cycle.
- If u and v are already in the same set, the edge closes a **cycle** → skip it.
- Start with n disjoint sets; each accepted edge merges two of them.
- If the graph is disconnected, n−1 edges are never reached; the algorithm returns a **minimum spanning forest**, one tree per connected component.

Sorting	O(m log m)
2m find + (n−1) union	O(m · α(n))
Total	**O(m log n)**

Kruskal's Algorithm Step by Step

```
I–J = 0
A–E = 1
C–I = 1
E–F = 1
G–H = 1
B–D = 2
C–J = 2
D–E = 2
D–H = 2
A–D = 4
B–C = 4
C–H = 4
G–I = 4
A–B = 5
D–F = 5
H–I = 6
F–G = 7
D–G = 11
```



Selected edge

0 / 9

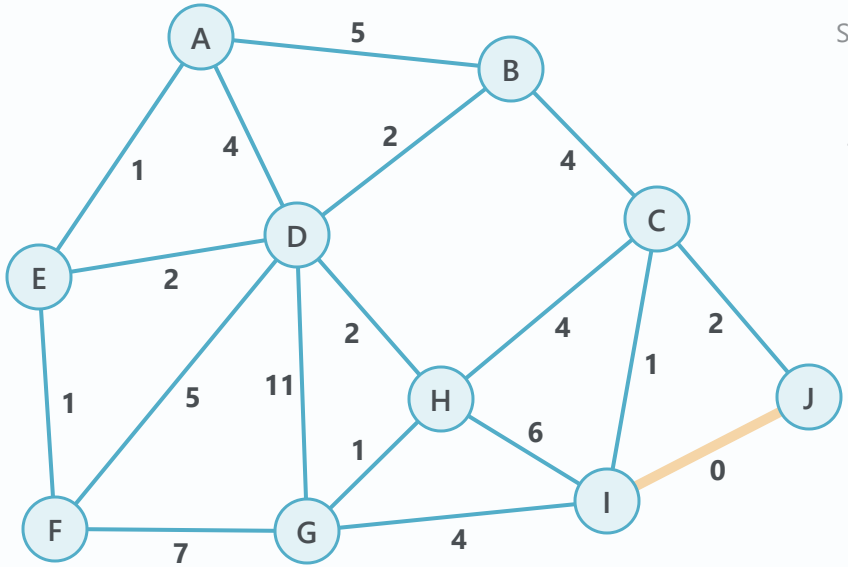
Total weight

0

The edges are sorted by weight (left). Initially 10 vertices = 10 separate sets; T is empty.

Kruskal's Algorithm Step by Step

```
I–J = 0 ✓
A–E = 1
C–I = 1
E–F = 1
G–H = 1
B–D = 2
C–J = 2
D–E = 2
D–H = 2
A–D = 4
B–C = 4
C–H = 4
G–I = 4
A–B = 5
D–F = 5
H–I = 6
F–G = 7
D–G = 11
```



Selected edge

1 / 9

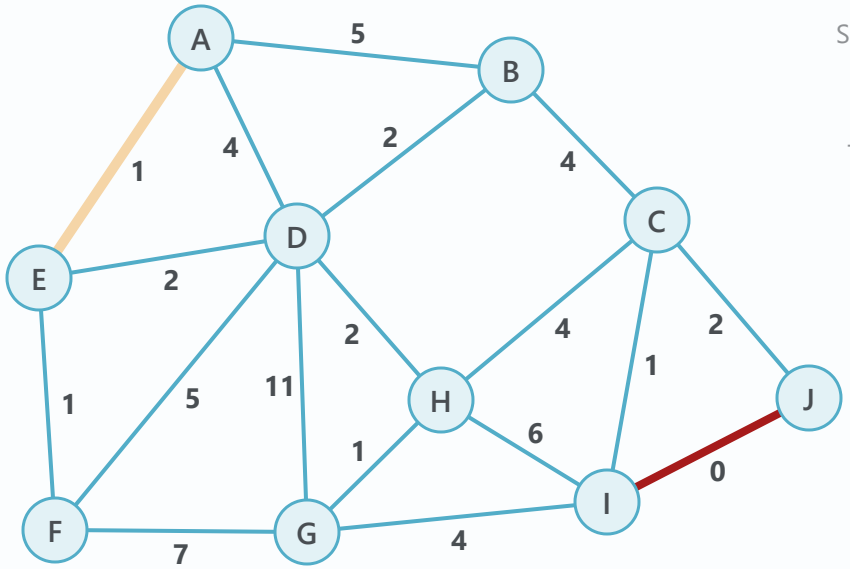
Total weight

0

I–J (0): find(I) ≠ find(J) → accept; the sets merge.

Kruskal's Algorithm Step by Step

```
I–J = 0 ✓
A–E = 1 ✓
C–I = 1
E–F = 1
G–H = 1
B–D = 2
C–J = 2
D–E = 2
D–H = 2
A–D = 4
B–C = 4
C–H = 4
G–I = 4
A–B = 5
D–F = 5
H–I = 6
F–G = 7
D–G = 11
```



Selected edge

2 / 9

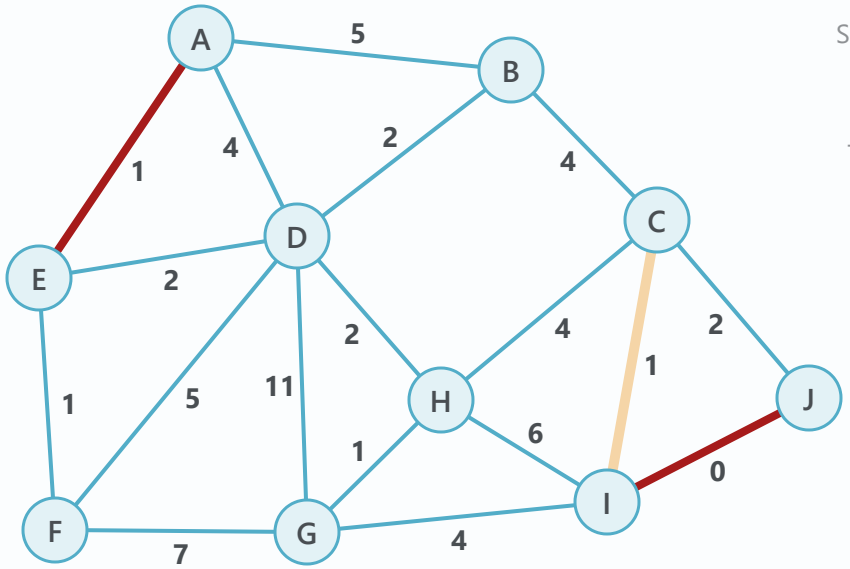
Total weight

1

A–E (1): find(A) ≠ find(E) → accept; the sets merge.

Kruskal's Algorithm Step by Step

I–J = 0 ✓
A–E = 1 ✓
C–I = 1 ✓
E–F = 1
G–H = 1
B–D = 2
C–J = 2
D–E = 2
D–H = 2
A–D = 4
B–C = 4
C–H = 4
G–I = 4
A–B = 5
D–F = 5
H–I = 6
F–G = 7
D–G = 11



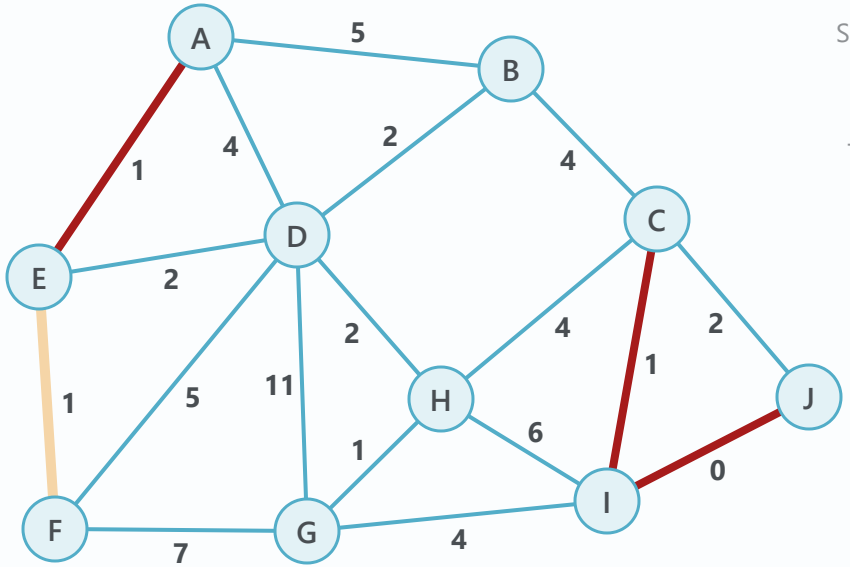
Selected edge
3 / 9

Total weight
2

C–I (1): find(C) ≠ find(I) → accept; the sets merge.

Kruskal's Algorithm Step by Step

I–J = 0 ✓
A–E = 1 ✓
C–I = 1 ✓
E–F = 1 ✓
G–H = 1
B–D = 2
C–J = 2
D–E = 2
D–H = 2
A–D = 4
B–C = 4
C–H = 4
G–I = 4
A–B = 5
D–F = 5
H–I = 6
F–G = 7
D–G = 11



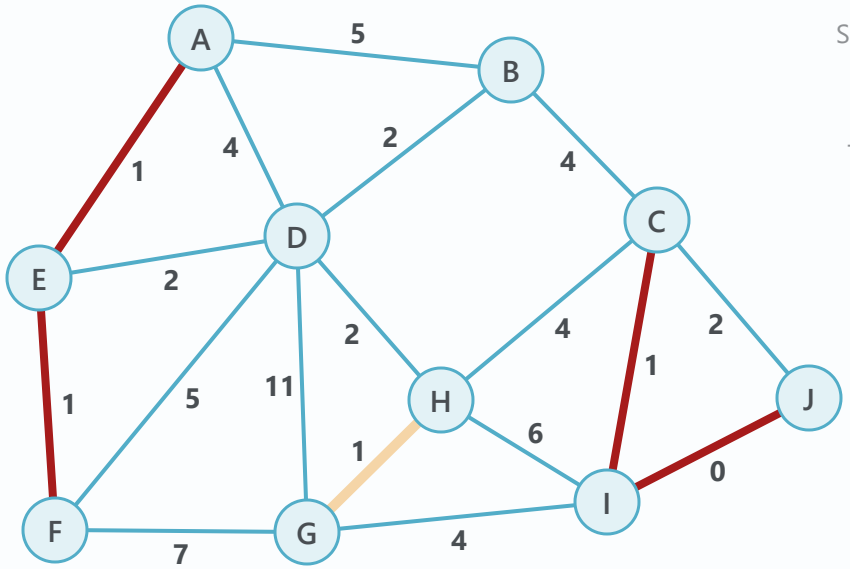
Selected edge
4 / 9

Total weight
3

E–F (1): find(E) ≠ find(F) → accept; the sets merge.

Kruskal's Algorithm Step by Step

I–J = 0 ✓
A–E = 1 ✓
C–I = 1 ✓
E–F = 1 ✓
G–H = 1 ✓
B–D = 2
C–J = 2
D–E = 2
D–H = 2
A–D = 4
B–C = 4
C–H = 4
G–I = 4
A–B = 5
D–F = 5
H–I = 6
F–G = 7
D–G = 11



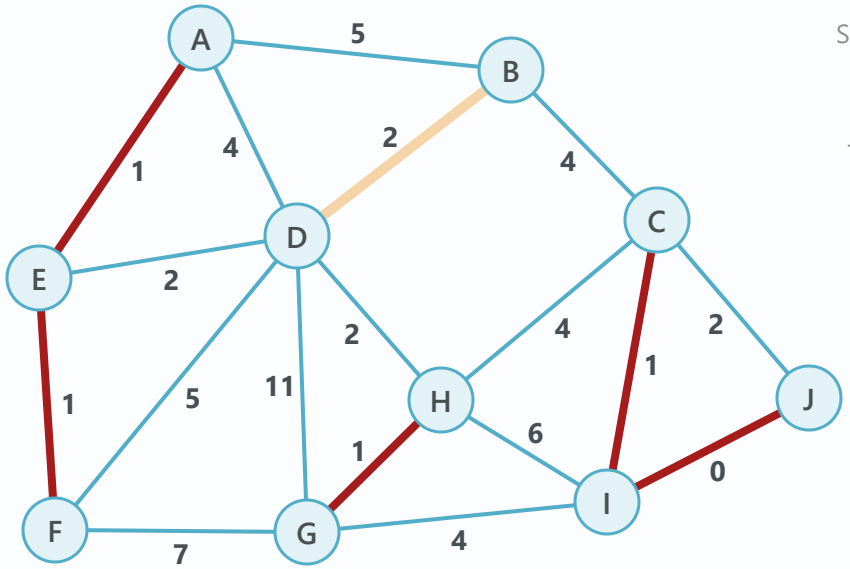
Selected edge
5 / 9

Total weight
4

G–H (1): find(G) ≠ find(H) → accept; the sets merge.

Kruskal's Algorithm Step by Step

I–J = 0 ✓
A–E = 1 ✓
C–I = 1 ✓
E–F = 1 ✓
G–H = 1 ✓
B–D = 2 ✓
C–J = 2
D–E = 2
D–H = 2
A–D = 4
B–C = 4
C–H = 4
G–I = 4
A–B = 5
D–F = 5
H–I = 6
F–G = 7
D–G = 11



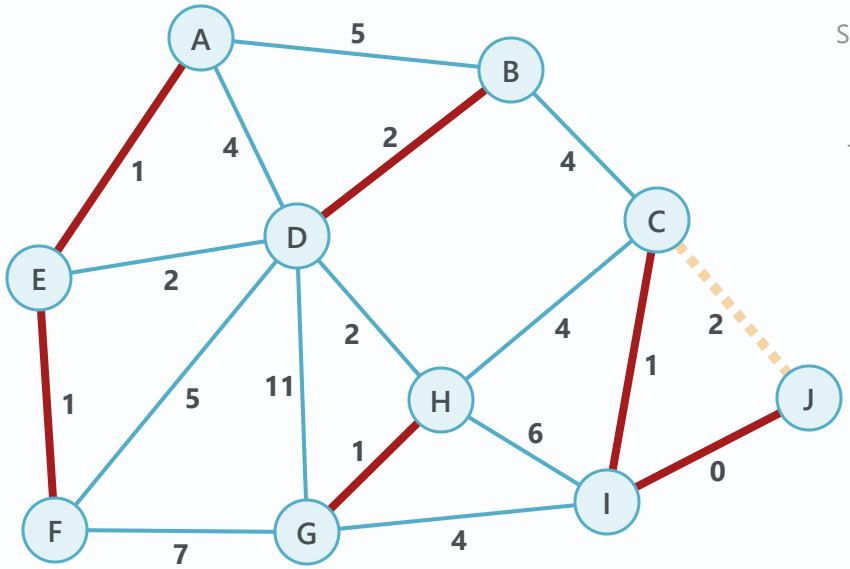
Selected edge
6 / 9

Total weight
6

B–D (2): find(B) ≠ find(D) → accept; the sets merge.

Kruskal's Algorithm Step by Step

I–J = 0 ✓
A–E = 1 ✓
C–I = 1 ✓
E–F = 1 ✓
G–H = 1 ✓
B–D = 2 ✓
~~C–J = 2 ✗~~
D–E = 2
D–H = 2
A–D = 4
B–C = 4
C–H = 4
G–I = 4
A–B = 5
D–F = 5
H–I = 6
F–G = 7
D–G = 11



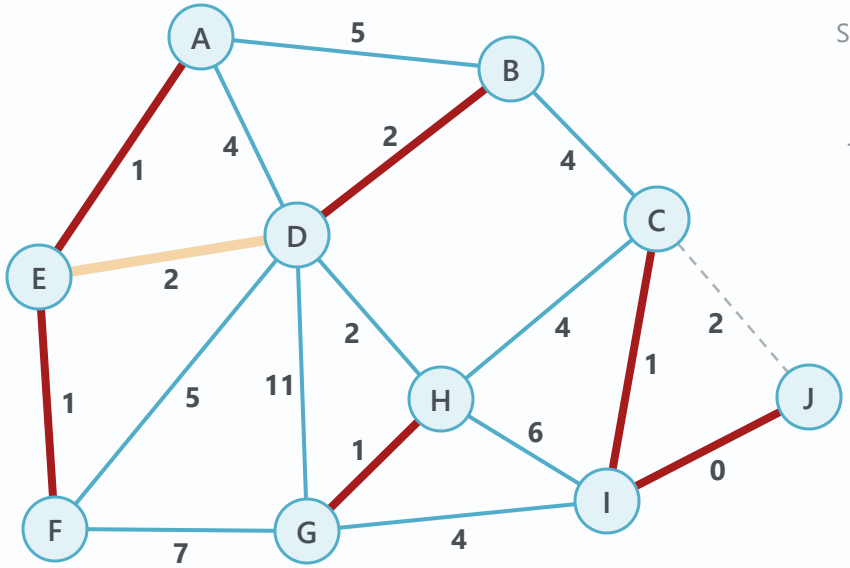
Selected edge
6 / 9

Total weight
6

C–J (2): C and J are already in the same set → it would form a cycle, so it is **rejected**.

Kruskal's Algorithm Step by Step

I–J = 0 ✓
A–E = 1 ✓
C–I = 1 ✓
E–F = 1 ✓
G–H = 1 ✓
B–D = 2 ✓
~~C–J = 2 ✗~~
D–E = 2 ✓
D–H = 2
A–D = 4
B–C = 4
C–H = 4
G–I = 4
A–B = 5
D–F = 5
H–I = 6
F–G = 7
D–G = 11



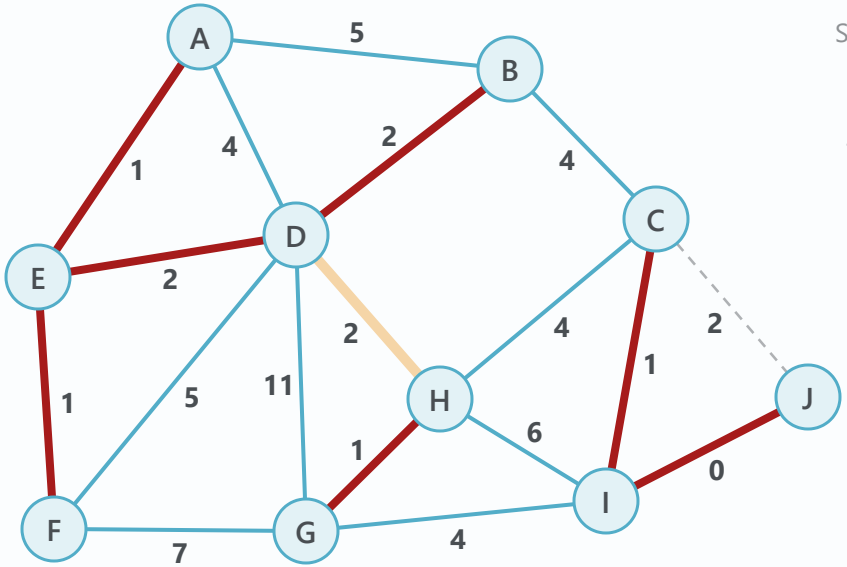
Selected edge
7 / 9

Total weight
8

D–E (2): find(D) ≠ find(E) → accept; the sets merge.

Kruskal's Algorithm Step by Step

I–J = 0 ✓
A–E = 1 ✓
C–I = 1 ✓
E–F = 1 ✓
G–H = 1 ✓
B–D = 2 ✓
~~C–J = 2 ✗~~
D–E = 2 ✓
D–H = 2 ✓
A–D = 4
B–C = 4
C–H = 4
G–I = 4
A–B = 5
D–F = 5
H–I = 6
F–G = 7
D–G = 11



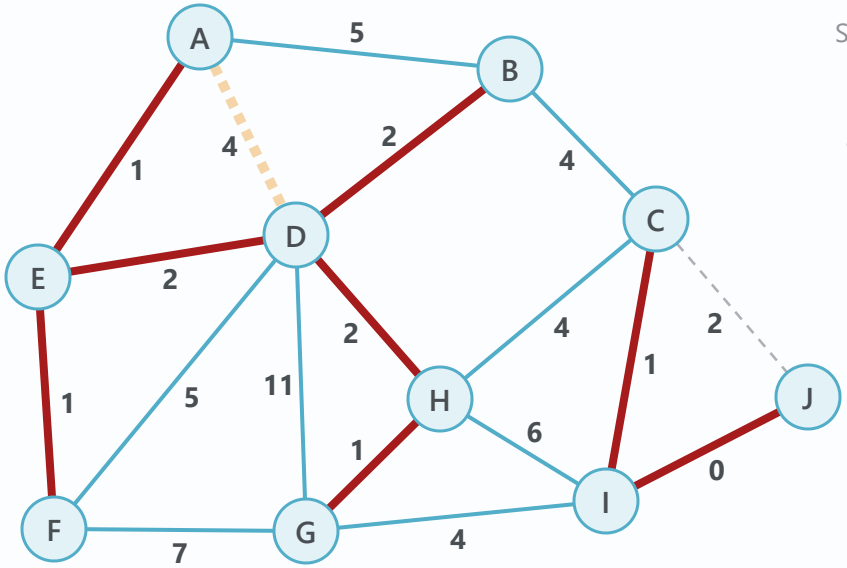
Selected edge
8 / 9

Total weight
10

D–H (2): find(D) ≠ find(H) → accept; the sets merge.

Kruskal's Algorithm Step by Step

I–J = 0 ✓
A–E = 1 ✓
C–I = 1 ✓
E–F = 1 ✓
G–H = 1 ✓
B–D = 2 ✓
~~C–J = 2 ✗~~
D–E = 2 ✓
D–H = 2 ✓
~~A–D = 4 ✗~~
B–C = 4
C–H = 4
G–I = 4
A–B = 5
D–F = 5
H–I = 6
F–G = 7
D–G = 11



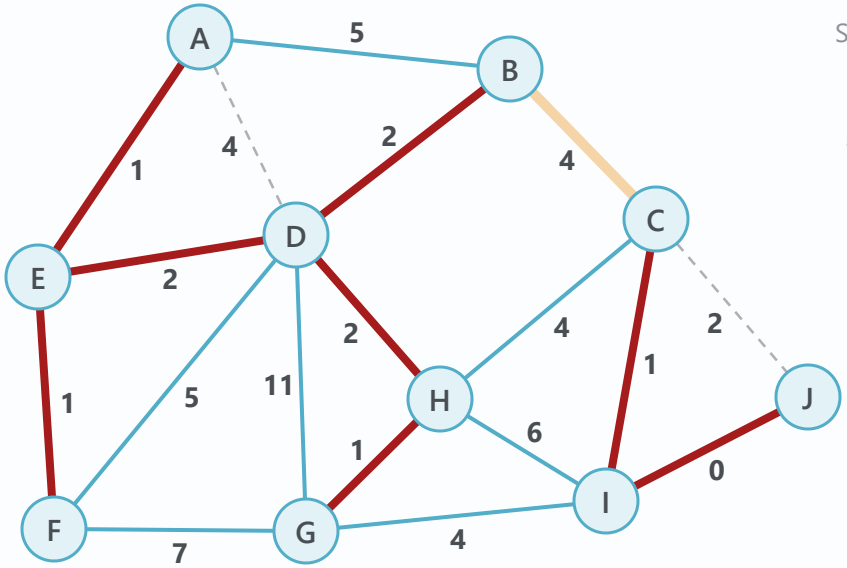
Selected edge
8 / 9

Total weight
10

A–D (4): A and D are already in the same set → it would form a cycle, so it is **rejected**.

Kruskal's Algorithm Step by Step

I–J = 0 ✓
A–E = 1 ✓
C–I = 1 ✓
E–F = 1 ✓
G–H = 1 ✓
B–D = 2 ✓
~~C–J = 2 ✗~~
D–E = 2 ✓
D–H = 2 ✓
~~A–D = 4 ✗~~
B–C = 4 ✓
C–H = 4
G–I = 4
A–B = 5
D–F = 5
H–I = 6
F–G = 7
D–G = 11



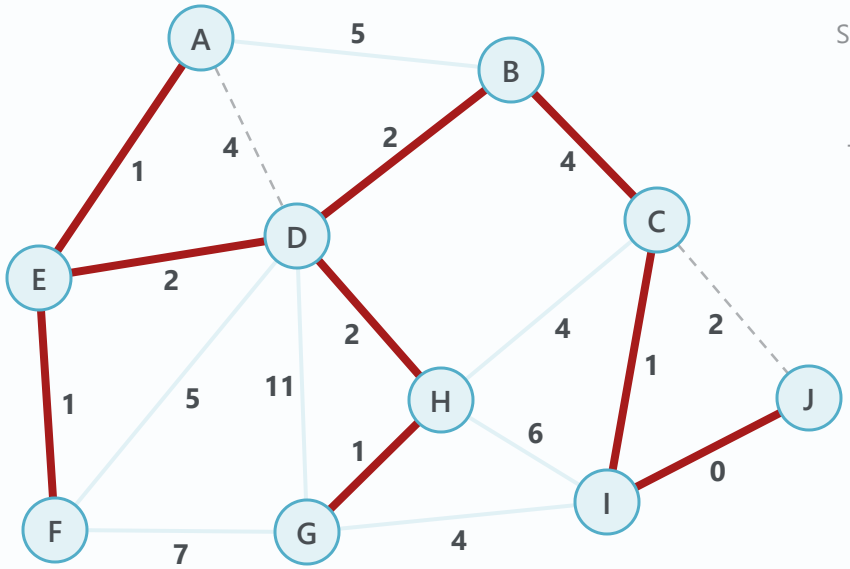
Selected edge
9 / 9

Total weight
14

B–C (4): find(B) ≠ find(C) → accept; the sets merge. 9 = n − 1 edges chosen: **done**, total weight 14.

Kruskal's Algorithm Step by Step

I–J = 0 ✓
A–E = 1 ✓
C–I = 1 ✓
E–F = 1 ✓
G–H = 1 ✓
B–D = 2 ✓
~~C–J = 2 ✗~~
D–E = 2 ✓
D–H = 2 ✓
~~A–D = 4 ✗~~
B–C = 4 ✓
C–H = 4
G–I = 4
A–B = 5
D–F = 5
H–I = 6
F–G = 7
D–G = 11



Selected edge
9 / 9

Total weight
14

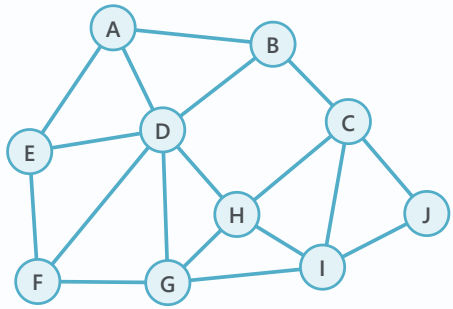
Minimum spanning tree: A–E, E–F, D–E, B–D, D–H, G–H, B–C, C–I, I–J. The remaining edges need not be examined.

Code Trace: `kruskal()` + Union-Find

```
1  List<int[]> kruskal(int n, List<int[]> E) {
▶    E.sort((a, b) ->          // {u, v, w}
▶        Integer.compare(a[2], b[2]));
4    UnionFind uf = new UnionFind(n);
5    List<int[]> T = new ArrayList<>();
6    for (int[] e : E) {
7      if (uf.union(e[0], e[1]))  // no cycle
8        T.add(e);
9      if (T.size() == n - 1) break;
10   }
11   return T;
12 }
```

VARIABLES

`n = 10`



uf forest (arrow → parent)



edges (sorted)

```
I–J 0
A–E 1
C–I 1
E–F 1
G–H 1
B–D 2
C–J 2
D–E 2
D–H 2
A–D 4
B–C 4
C–H 4
G–I 4
A–B 5
D–F 5
H–I 6
F–G 7
D–G 11
```

|T|=0, weight=0

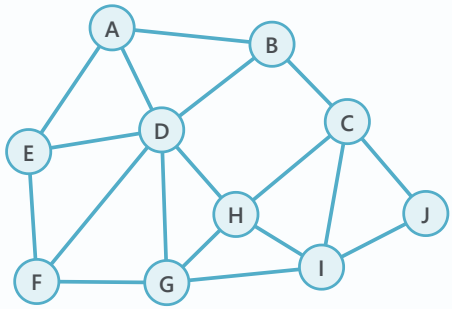
Sort the edges by weight (A=0 … J=9; `sort` is stable, so equal weights keep their input order). O(m log m).

Code Trace: `kruskal()` + Union-Find

```
1  List<int[]> kruskal(int n, List<int[]> E) {
2    E.sort((a, b) ->            // {u, v, w}
3        Integer.compare(a[2], b[2]));
▶    UnionFind uf = new UnionFind(n);
▶    List<int[]> T = new ArrayList<>();
6    for (int[] e : E) {
7      if (uf.union(e[0], e[1]))  // no cycle
8        T.add(e);
9      if (T.size() == n - 1) break;
10   }
11   return T;
12 }
```

VARIABLES

`n = 10`



uf forest (arrow → parent)



edges (sorted)

```
I–J 0
A–E 1
C–I 1
E–F 1
G–H 1
B–D 2
C–J 2
D–E 2
D–H 2
A–D 4
B–C 4
C–H 4
G–I 4
A–B 5
D–F 5
H–I 6
F–G 7
D–G 11
```

|T|=0, weight=0

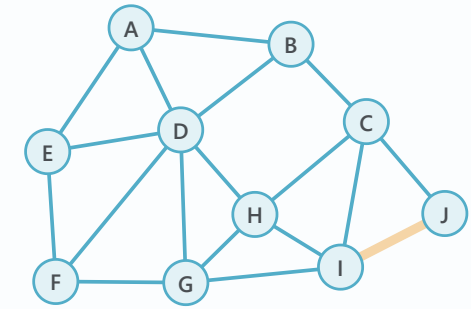
`new UnionFind(10)` : each vertex is its own set (10 roots). T is empty.

Code Trace: `kruskal()` + Union-Find

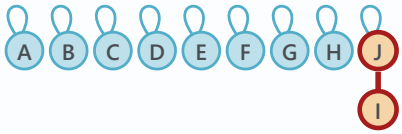
```
1  List<int[]> kruskal(int n, List<int[]> E) {
2    E.sort((a, b) ->            // {u, v, w}
3        Integer.compare(a[2], b[2]));
4    UnionFind uf = new UnionFind(n);
5    List<int[]> T = new ArrayList<>();
6    for (int[] e : E) {
▶      if (uf.union(e[0], e[1]))  // no cycle
▶        T.add(e);
9      if (T.size() == n - 1) break;
10   }
11   return T;
12 }
```

VARIABLES

`e = {I,J,0}` `|T| = 1`



uf forest (arrow → parent)



edges (sorted)

- **I–J 0 ✓**
- A–E 1
- C–I 1
- E–F 1
- G–H 1
- B–D 2
- C–J 2
- D–E 2
- D–H 2
- A–D 4
- B–C 4
- C–H 4
- G–I 4
- A–B 5
- D–F 5
- H–I 6
- F–G 7
- D–G 11

|T|=1, weight=0

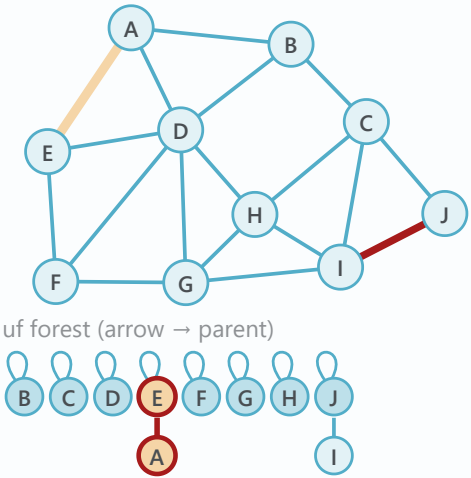
I–J (0): roots I ≠ J → `union` returns true: the trees merge (I → J) and the edge joins T.

Code Trace: `kruskal()` + Union-Find

```
1  List<int[]> kruskal(int n, List<int[]> E) {
2    E.sort((a, b) ->            // {u, v, w}
3        Integer.compare(a[2], b[2]));
4    UnionFind uf = new UnionFind(n);
5    List<int[]> T = new ArrayList<>();
6    for (int[] e : E) {
▶      if (uf.union(e[0], e[1]))  // no cycle
▶        T.add(e);
9      if (T.size() == n - 1) break;
10   }
11   return T;
12 }
```

VARIABLES

`e = {A,E,1}` `|T| = 2`



edges (sorted)

- **I–J 0 ✓**
- **A–E 1 ✓**
- C–I 1
- E–F 1
- G–H 1
- B–D 2
- C–J 2
- D–E 2
- D–H 2
- A–D 4
- B–C 4
- C–H 4
- G–I 4
- A–B 5
- D–F 5
- H–I 6
- F–G 7
- D–G 11

|T|=2, weight=1

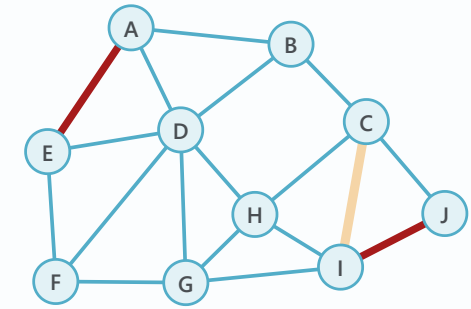
A–E (1): roots A ≠ E → `union` returns true: the trees merge (A → E) and the edge joins T.

Code Trace: `kruskal()` + Union-Find

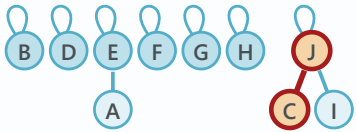
```
1  List<int[]> kruskal(int n, List<int[]> E) {
2    E.sort((a, b) ->            // {u, v, w}
3        Integer.compare(a[2], b[2]));
4    UnionFind uf = new UnionFind(n);
5    List<int[]> T = new ArrayList<>();
6    for (int[] e : E) {
▶      if (uf.union(e[0], e[1]))  // no cycle
▶        T.add(e);
9      if (T.size() == n - 1) break;
10   }
11   return T;
12 }
```

VARIABLES

`e = {C,I,1}` `|T| = 3`



uf forest (arrow → parent)



edges (sorted)

I–J 0 ✓
A–E 1 ✓
C–I 1 ✓
E–F 1
G–H 1
B–D 2
C–J 2
D–E 2
D–H 2
A–D 4
B–C 4
C–H 4
G–I 4
A–B 5
D–F 5
H–I 6
F–G 7
D–G 11

|T|=3, weight=2

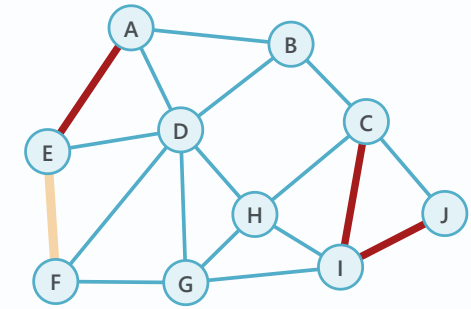
C–I (1): roots C ≠ J → `union` returns true: the trees merge (C → J) and the edge joins T.

Code Trace: `kruskal()` + Union-Find

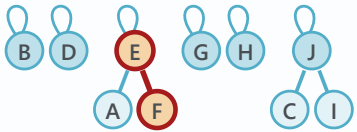
```
1  List<int[]> kruskal(int n, List<int[]> E) {
2    E.sort((a, b) ->            // {u, v, w}
3        Integer.compare(a[2], b[2]));
4    UnionFind uf = new UnionFind(n);
5    List<int[]> T = new ArrayList<>();
6    for (int[] e : E) {
▶      if (uf.union(e[0], e[1]))  // no cycle
▶        T.add(e);
9      if (T.size() == n - 1) break;
10   }
11   return T;
12 }
```

VARIABLES

`e = {E,F,1}` `|T| = 4`



uf forest (arrow → parent)



edges (sorted)

I–J 0 ✓
A–E 1 ✓
C–I 1 ✓
E–F 1 ✓
G–H 1
B–D 2
C–J 2
D–E 2
D–H 2
A–D 4
B–C 4
C–H 4
G–I 4
A–B 5
D–F 5
H–I 6
F–G 7
D–G 11

|T|=4, weight=3

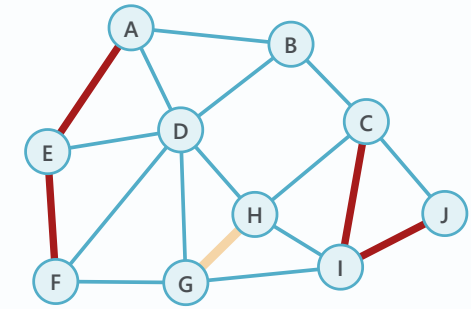
E–F (1): roots E ≠ F → `union` returns true: the trees merge (F → E) and the edge joins T.

Code Trace: `kruskal()` + Union-Find

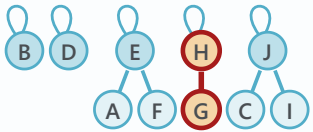
```
1  List<int[]> kruskal(int n, List<int[]> E) {
2    E.sort((a, b) ->            // {u, v, w}
3        Integer.compare(a[2], b[2]));
4    UnionFind uf = new UnionFind(n);
5    List<int[]> T = new ArrayList<>();
6    for (int[] e : E) {
▶      if (uf.union(e[0], e[1]))  // no cycle
▶        T.add(e);
9      if (T.size() == n - 1) break;
10   }
11   return T;
12 }
```

VARIABLES

`e = {G,H,1}` `|T| = 5`



uf forest (arrow → parent)



edges (sorted)

I–J 0 ✓
A–E 1 ✓
C–I 1 ✓
E–F 1 ✓
G–H 1 ✓
B–D 2
C–J 2
D–E 2
D–H 2
A–D 4
B–C 4
C–H 4
G–I 4
A–B 5
D–F 5
H–I 6
F–G 7
D–G 11

|T|=5, weight=4

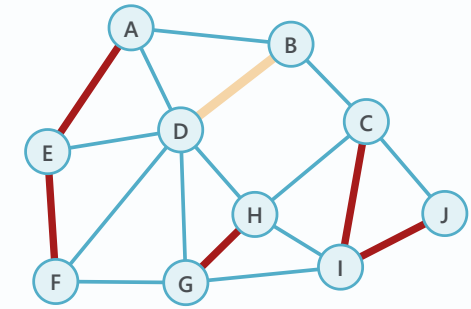
G–H (1): roots G ≠ H → `union` returns true: the trees merge (G → H) and the edge joins T.

Code Trace: `kruskal()` + Union-Find

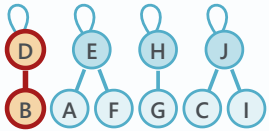
```
1  List<int[]> kruskal(int n, List<int[]> E) {
2    E.sort((a, b) ->              // {u, v, w}
3        Integer.compare(a[2], b[2]));
4    UnionFind uf = new UnionFind(n);
5    List<int[]> T = new ArrayList<>();
6    for (int[] e : E) {
▶      if (uf.union(e[0], e[1]))  // no cycle
▶        T.add(e);
9      if (T.size() == n - 1) break;
10   }
11   return T;
12 }
```

VARIABLES

`e = {B,D,2}` `|T| = 6`



uf forest (arrow → parent)



edges (sorted)

- **I–J 0** ✓
- **A–E 1** ✓
- **C–I 1** ✓
- **E–F 1** ✓
- **G–H 1** ✓
- **B–D 2** ✓
- C–J 2
- D–E 2
- D–H 2
- A–D 4
- B–C 4
- C–H 4
- G–I 4
- A–B 5
- D–F 5
- H–I 6
- F–G 7
- D–G 11

|T|=6, weight=6

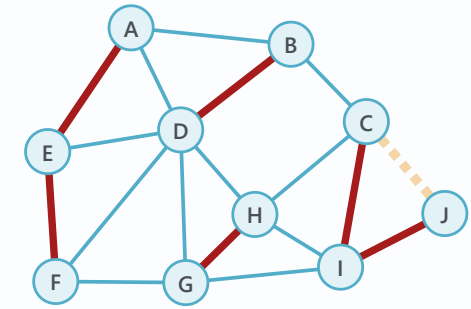
B–D (2): roots B ≠ D → `union` returns true: the trees merge (B → D) and the edge joins T.

Code Trace: `kruskal()` + Union-Find

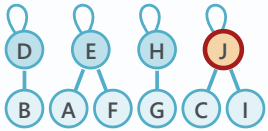
```
1  List<int[]> kruskal(int n, List<int[]> E) {
2    E.sort((a, b) ->              // {u, v, w}
3        Integer.compare(a[2], b[2]));
4    UnionFind uf = new UnionFind(n);
5    List<int[]> T = new ArrayList<>();
6    for (int[] e : E) {
▶      if (uf.union(e[0], e[1]))  // no cycle
8        T.add(e);
9      if (T.size() == n - 1) break;
10   }
11   return T;
12 }
```

VARIABLES

`e = {C,J,2}` `|T| = 6`



uf forest (arrow → parent)



edges (sorted)

I–J 0 ✓
A–E 1 ✓
C–I 1 ✓
E–F 1 ✓
G–H 1 ✓
B–D 2 ✓
~~C–J 2 ✗~~
D–E 2
D–H 2
A–D 4
B–C 4
C–H 4
G–I 4
A–B 5
D–F 5
H–I 6
F–G 7
D–G 11

|T|=6, weight=6

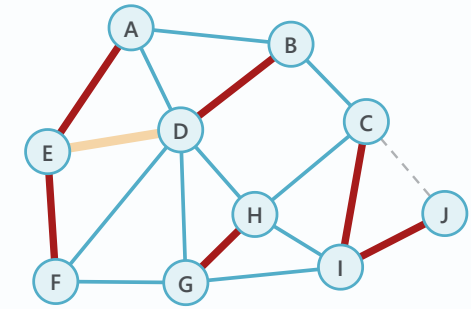
C–J (2): find(C) = find(J) = J → same tree; `union` returns false: the edge **would close a cycle and is skipped**.

Code Trace: `kruskal()` + Union-Find

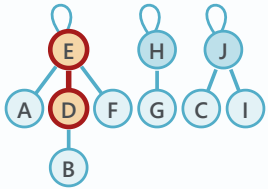
```
1  List<int[]> kruskal(int n, List<int[]> E) {
2    E.sort((a, b) ->            // {u, v, w}
3        Integer.compare(a[2], b[2]));
4    UnionFind uf = new UnionFind(n);
5    List<int[]> T = new ArrayList<>();
6    for (int[] e : E) {
▶      if (uf.union(e[0], e[1]))  // no cycle
▶        T.add(e);
9      if (T.size() == n - 1) break;
10   }
11   return T;
12 }
```

VARIABLES

`e = {D,E,2}` `|T| = 7`



uf forest (arrow → parent)



edges (sorted)

- **I–J 0** ✓
- **A–E 1** ✓
- **C–I 1** ✓
- **E–F 1** ✓
- **G–H 1** ✓
- **B–D 2** ✓
- ~~C–J 2 ✗~~
- **D–E 2** ✓
- D–H 2
- A–D 4
- B–C 4
- C–H 4
- G–I 4
- A–B 5
- D–F 5
- H–I 6
- F–G 7
- D–G 11

|T|=7, weight=8

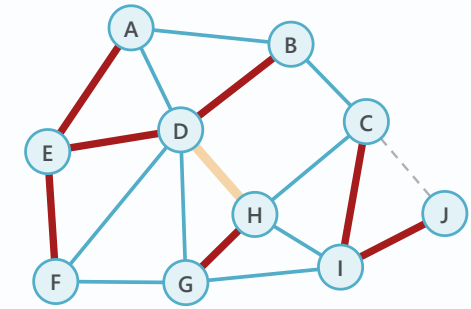
D–E (2): roots D ≠ E → `union` returns true: the trees merge (D → E) and the edge joins T.

Code Trace: `kruskal()` + Union-Find

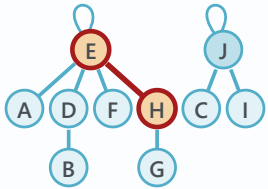
```
1  List<int[]> kruskal(int n, List<int[]> E) {
2    E.sort((a, b) ->          // {u, v, w}
3        Integer.compare(a[2], b[2]));
4    UnionFind uf = new UnionFind(n);
5    List<int[]> T = new ArrayList<>();
6    for (int[] e : E) {
▶      if (uf.union(e[0], e[1]))  // no cycle
▶        T.add(e);
9      if (T.size() == n - 1) break;
10   }
11   return T;
12 }
```

VARIABLES

`e = {D,H,2}` `|T| = 8`



uf forest (arrow → parent)



edges (sorted)

- **I–J 0** ✓
- **A–E 1** ✓
- **C–I 1** ✓
- **E–F 1** ✓
- **G–H 1** ✓
- **B–D 2** ✓
- ~~C–J 2 ✗~~
- **D–E 2** ✓
- **D–H 2** ✓
- A–D 4
- B–C 4
- C–H 4
- G–I 4
- A–B 5
- D–F 5
- H–I 6
- F–G 7
- D–G 11

|T|=8, weight=10

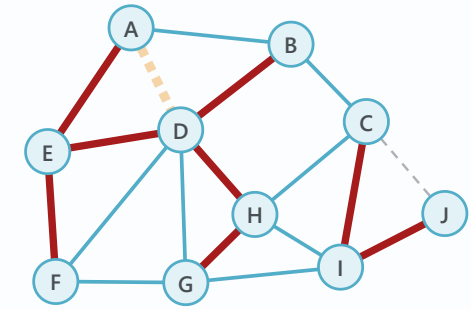
D–H (2): roots E ≠ H → `union` returns true: the trees merge (H → E) and the edge joins T.

Code Trace: `kruskal()` + Union-Find

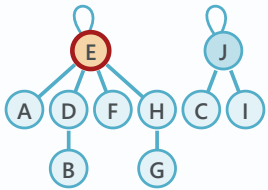
```
1  List<int[]> kruskal(int n, List<int[]> E) {
2    E.sort((a, b) ->            // {u, v, w}
3        Integer.compare(a[2], b[2]));
4    UnionFind uf = new UnionFind(n);
5    List<int[]> T = new ArrayList<>();
6    for (int[] e : E) {
▶      if (uf.union(e[0], e[1]))  // no cycle
8        T.add(e);
9      if (T.size() == n - 1) break;
10   }
11   return T;
12 }
```

VARIABLES

`e = {A,D,4}` `|T| = 8`



uf forest (arrow → parent)



edges (sorted)

~~I–J 0 ✓~~
~~A–E 1 ✓~~
~~C–I 1 ✓~~
~~E–F 1 ✓~~
~~G–H 1 ✓~~
~~B–D 2 ✓~~
~~C–J 2 ✗~~
~~D–E 2 ✓~~
~~D–H 2 ✓~~
~~A–D 4 ✗~~
B–C 4
C–H 4
G–I 4
A–B 5
D–F 5
H–I 6
F–G 7
D–G 11

|T|=8, weight=10

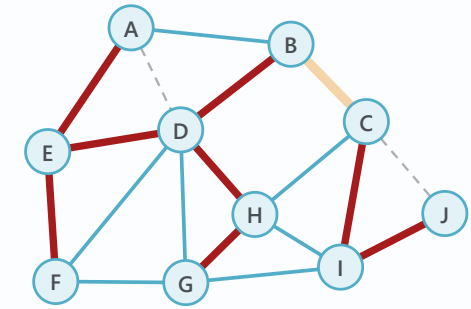
A–D (4): find(A) = find(D) = E → same tree; `union` returns false: the edge **would close a cycle and is skipped**.

Code Trace: `kruskal()` + Union-Find

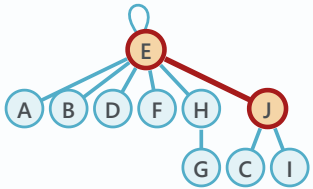
```
1  List<int[]> kruskal(int n, List<int[]> E) {
2    E.sort((a, b) ->          // {u, v, w}
3        Integer.compare(a[2], b[2]));
4    UnionFind uf = new UnionFind(n);
5    List<int[]> T = new ArrayList<>();
6    for (int[] e : E) {
▶      if (uf.union(e[0], e[1]))  // no cycle
▶        T.add(e);
▶      if (T.size() == n - 1) break;
10   }
11   return T;
12 }
```

VARIABLES

`e = {B,C,4}` `|T| = 9`



uf forest (arrow → parent)



edges (sorted)

- **I–J 0** ✓
- **A–E 1** ✓
- **C–I 1** ✓
- **E–F 1** ✓
- **G–H 1** ✓
- **B–D 2** ✓
- ~~C–J 2 ✗~~
- **D–E 2** ✓
- **D–H 2** ✓
- ~~A–D 4 ✗~~
- **B–C 4** ✓
- C–H 4
- G–I 4
- A–B 5
- D–F 5
- H–I 6
- F–G 7
- D–G 11

|T|=9, weight=14

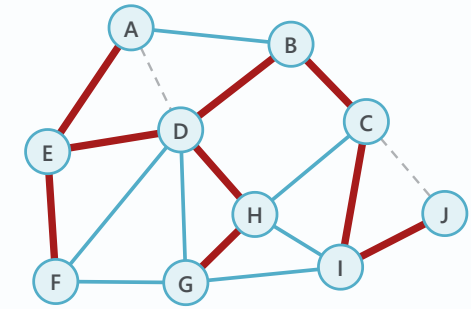
B–C (4): roots E ≠ J → `union` returns true: the trees merge (J → E) and the edge joins T. |T| = 9 = n − 1 → **break**.

Code Trace: `kruskal()` + Union-Find

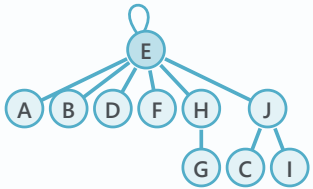
```
1  List<int[]> kruskal(int n, List<int[]> E) {
2    E.sort((a, b) ->          // {u, v, w}
3        Integer.compare(a[2], b[2]));
4    UnionFind uf = new UnionFind(n);
5    List<int[]> T = new ArrayList<>();
6    for (int[] e : E) {
7      if (uf.union(e[0], e[1]))  // no cycle
8        T.add(e);
9      if (T.size() == n - 1) break;
10   }
▶    return T;
12 }
```

VARIABLES

`|T| = 9`



uf forest (arrow → parent)



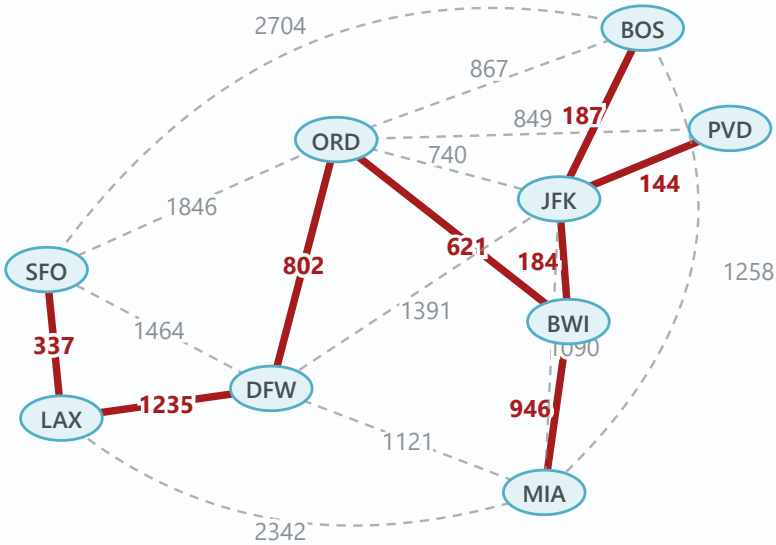
edges (sorted)

- **I–J 0** ✓
- **A–E 1** ✓
- **C–I 1** ✓
- **E–F 1** ✓
- **G–H 1** ✓
- **B–D 2** ✓
- ~~C–J 2 ✗~~
- **D–E 2** ✓
- **D–H 2** ✓
- ~~A–D 4 ✗~~
- **B–C 4** ✓
- C–H 4
- G–I 4
- A–B 5
- D–F 5
- H–I 6
- F–G 7
- D–G 11

|T|=9, weight=14

`return T`: 9 edges, total weight **14**. A single tree remains. The other 7 edges were never examined.

A Second Example: Airports



Goodrich's example: 9 airports, 19 flight routes (miles). Kruskal's order:

144, 184, 187, 337, 621	accept
740 ORD–JFK	reject (cycle)
802	accept
849, 867	reject
946	accept
1090, 1121	reject
1235 LAX–DFW	accept → 8 edges

Total: **4,456** miles.

Prim's Algorithm

```
Prim(G, s):
  tree = {s}, T = ∅
  add s's edges to the queue
  while |T| < n − 1:
    (u, v) = lightest edge  // u in tree
    if v in tree: skip       // cycle
    add v to tree, (u, v) to T
    add v's edges leaving the tree
      to the queue
  return T
```

- The tree starts from **a single vertex** and grows by one vertex per step.
- Each step takes the **lightest** edge joining a tree vertex to a vertex outside the tree.
- The lightest edge leaving the tree always belongs to some minimum spanning tree: the tree must connect to the outside somehow, and this edge is the cheapest connection.
- Similar to Dijkstra; the difference is the queue key: **the weight of a single edge**, not the distance from the source.
- Unlike Kruskal, there is always **a single tree** along the way, so no Union-Find is needed.

Prim in Java (Priority Queue, Lazy Version)

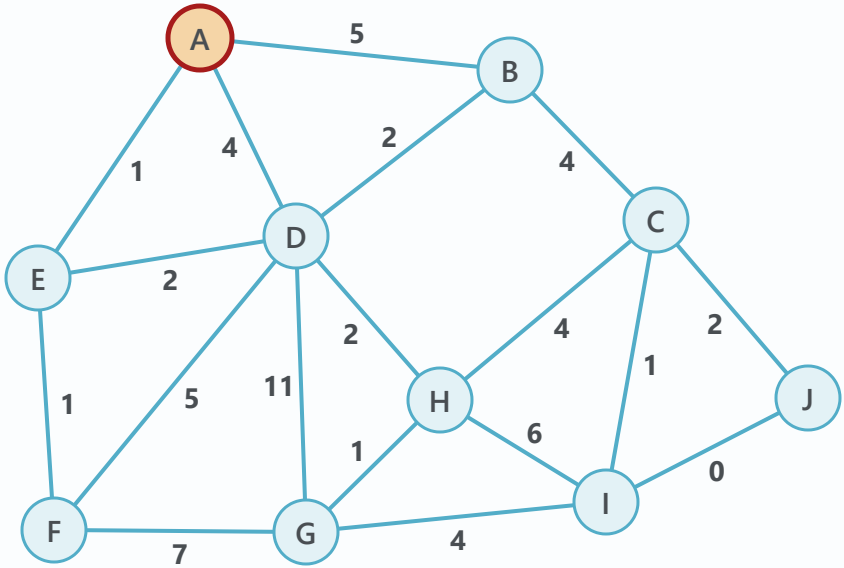
```
List<int[]> prim(List<List<int[]>> adj, int s) {
  int n = adj.size();                 // adj.get(u): {v, w}
  boolean[] inTree = new boolean[n];
  PriorityQueue<int[]> pq = new PriorityQueue<>(
      (a, b) -> Integer.compare(a[2], b[2]));  // {u, v, w}
  List<int[]> T = new ArrayList<>();
  inTree[s] = true;
  for (int[] e : adj.get(s)) pq.add(new int[]{s, e[0], e[1]});
  while (!pq.isEmpty() && T.size() < n - 1) {
    int[] e = pq.poll();               // lightest candidate
    if (inTree[e[1]]) continue;        // stale: closes a cycle
    inTree[e[1]] = true;
    T.add(e);
    for (int[] f : adj.get(e[1]))
      if (!inTree[f[0]]) pq.add(new int[]{e[1], f[0], f[1]});
  }
  return T;
}
```

- The queue holds entries {u, v, w}: u is in the tree, v is a candidate.
- **Lazy** version: when v joins the tree, old entries leading to v are not removed; when polled, they are skipped with `continue` (stale entries).
- Each edge enters the queue at most once (from whichever end joins the tree first; when the other end joins, the first end is already in the tree): at most m inserts and removals, each O(log m).
- Total **O(m log m) = O(m log n)**; extra memory O(m).

Prim's Algorithm Step by Step (from A)

priority queue

```
A–E = 1
A–D = 4
A–B = 5
```



Tree vertex

1 / 10

Total weight

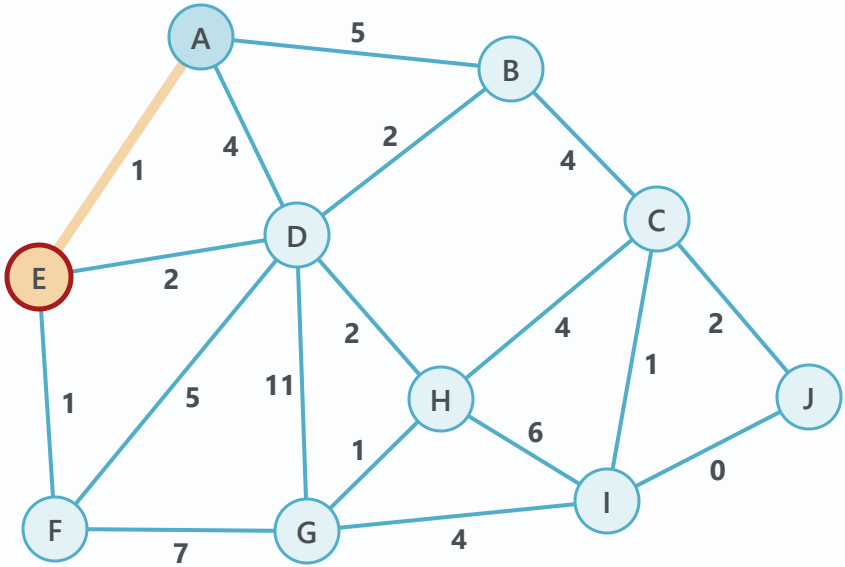
0

Start: only **A** is in the tree. A's edges enter the queue: A–E 1, A–D 4, A–B 5. Ties are broken alphabetically.

Prim's Algorithm Step by Step (from A)

priority queue

```
E–F = 1
E–D = 2
A–D = 4
A–B = 5
```



Tree vertex

2 / 10

Total weight

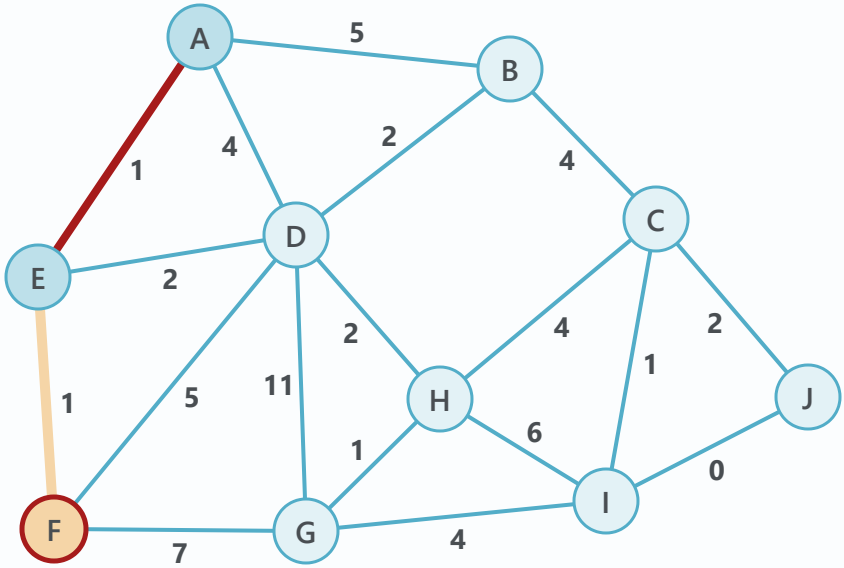
1

A–E (1): lightest edge leaving the tree → **E** joins the tree. New candidates: E–F, E–D.

Prim's Algorithm Step by Step (from A)

priority queue

- E–D = 2
- A–D = 4
- A–B = 5
- F–D = 5
- F–G = 7



Tree vertex

3 / 10

Total weight

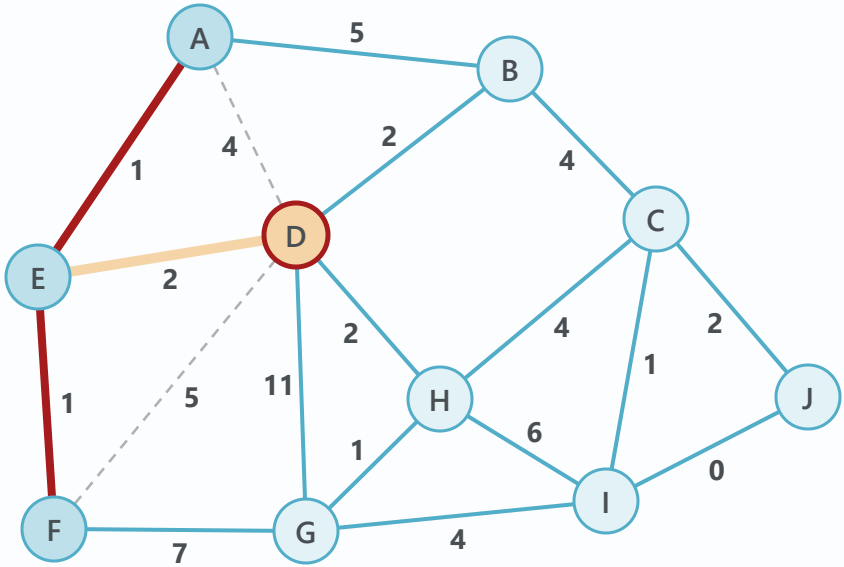
2

E–F (1): lightest edge leaving the tree → **F** joins the tree. New candidates: F–D, F–G.

Prim's Algorithm Step by Step (from A)

priority queue

- D–B = 2
- D–H = 2
- ~~A–D = 4~~
- A–B = 5
- ~~F–D = 5~~
- F–G = 7
- D–G = 11



Tree vertex

4 / 10

Total weight

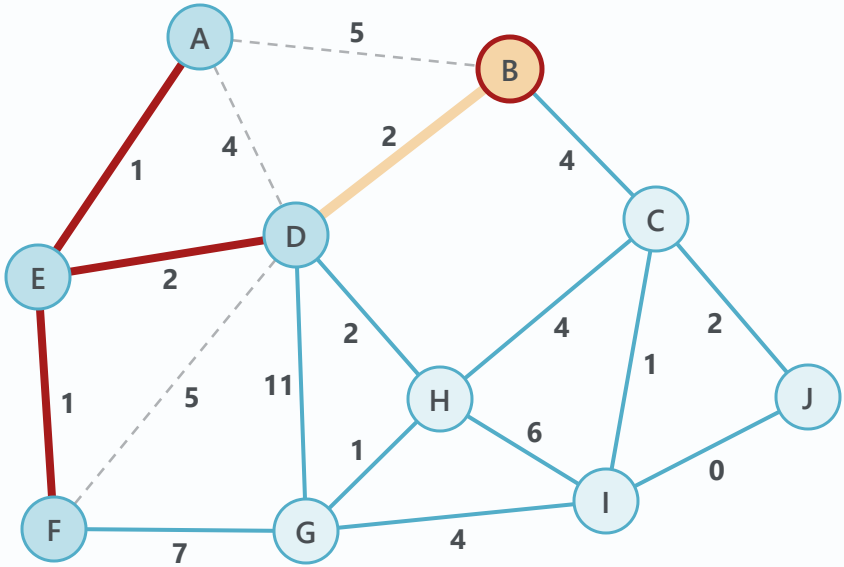
4

E–D (2): lightest edge leaving the tree → **D** joins the tree. New candidates: D–B, D–H, D–G. Struck-out entries are now stale (both ends in the tree); they will be skipped when polled.

Prim's Algorithm Step by Step (from A)

priority queue

- D–H = 2
- B–C = 4
- ~~A–D = 4~~
- ~~A–B = 5~~
- ~~F–D = 5~~
- F–G = 7
- D–G = 11



Tree vertex

5 / 10

Total weight

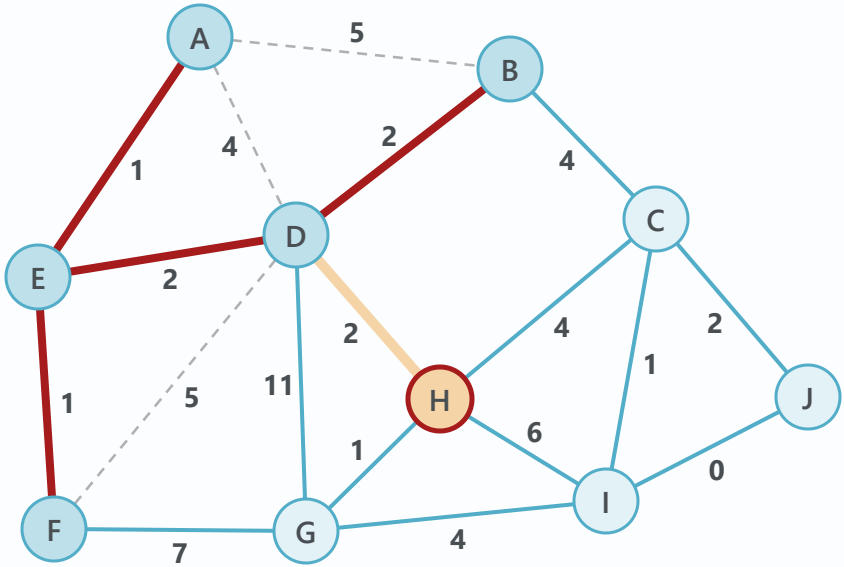
6

D–B (2): lightest edge leaving the tree → **B** joins the tree. New candidates: B–C.

Prim's Algorithm Step by Step (from A)

priority queue

- H–G = 1
- B–C = 4
- H–C = 4
- ~~A–D = 4~~
- ~~A–B = 5~~
- ~~F–D = 5~~
- H–I = 6
- F–G = 7
- D–G = 11



Tree vertex

6 / 10

Total weight

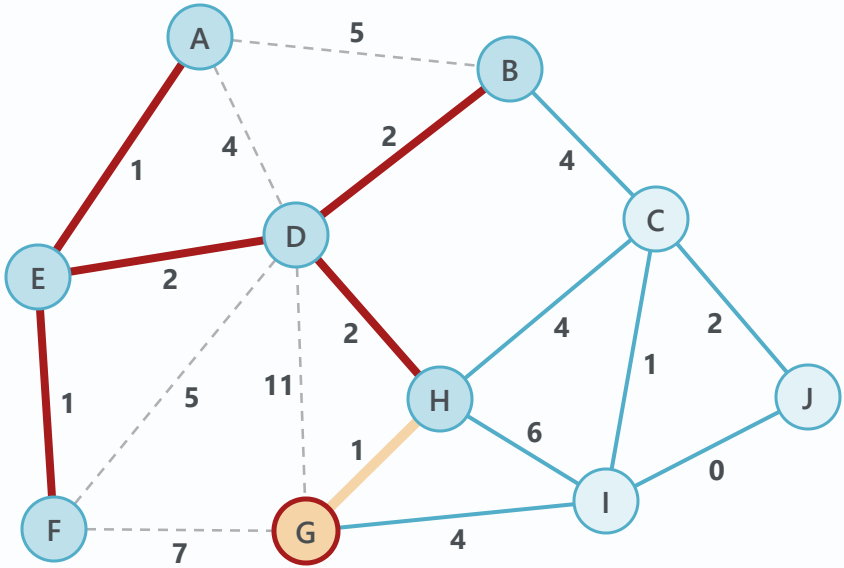
8

D–H (2): lightest edge leaving the tree → **H** joins the tree. New candidates: H–G, H–C, H–I.

Prim's Algorithm Step by Step (from A)

priority queue

- B–C = 4
- H–C = 4
- ~~A–D = 4~~
- G–I = 4
- ~~A–B = 5~~
- ~~F–D = 5~~
- H–I = 6
- ~~F–G = 7~~
- ~~D–G = 11~~



Tree vertex

7 / 10

Total weight

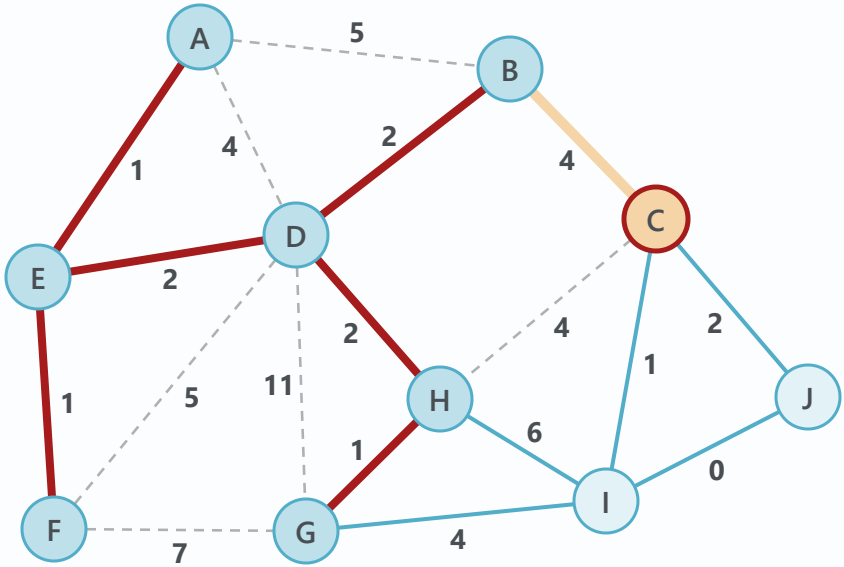
9

H–G (1): lightest edge leaving the tree → **G** joins the tree. New candidates: G–I. Struck-out entries are now stale (both ends in the tree); they will be skipped when polled.

Prim's Algorithm Step by Step (from A)

priority queue

- C–I = 1
- C–J = 2
- ~~H–C = 4~~
- ~~A–D = 4~~
- G–I = 4
- ~~A–B = 5~~
- ~~F–D = 5~~
- H–I = 6
- ~~F–G = 7~~
- ~~D–G = 11~~



Tree vertex

8 / 10

Total weight

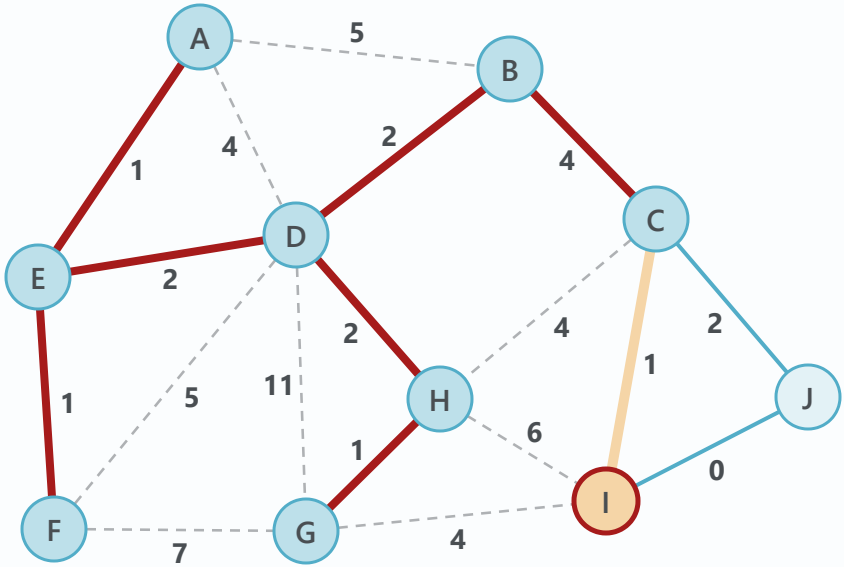
13

B–C (4): lightest edge leaving the tree → **C** joins the tree. New candidates: C–I, C–J.

Prim's Algorithm Step by Step (from A)

priority queue

- I–J = 0
- C–J = 2
- ~~H–C = 4~~
- ~~A–D = 4~~
- ~~G–I = 4~~
- ~~A–B = 5~~
- ~~F–D = 5~~
- ~~H–I = 6~~
- ~~F–G = 7~~
- ~~D–G = 11~~



Tree vertex

9 / 10

Total weight

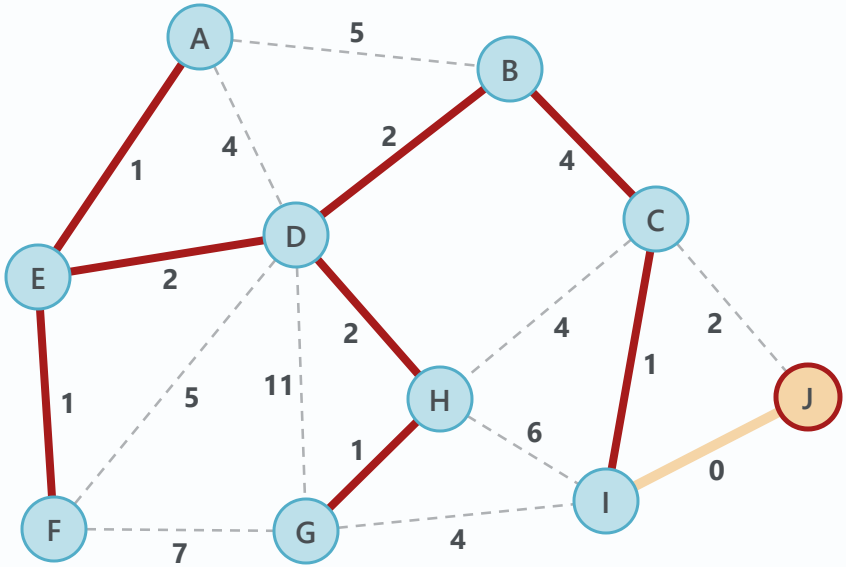
14

C–I (1): lightest edge leaving the tree → **I** joins the tree. New candidates: I–J.

Prim's Algorithm Step by Step (from A)

priority queue

~~C–J = 2~~
~~H–C = 4~~
~~A–D = 4~~
~~G–I = 4~~
~~A–B = 5~~
~~F–D = 5~~
~~H–I = 6~~
~~F–G = 7~~
~~D–G = 11~~



Tree vertex
10 / 10

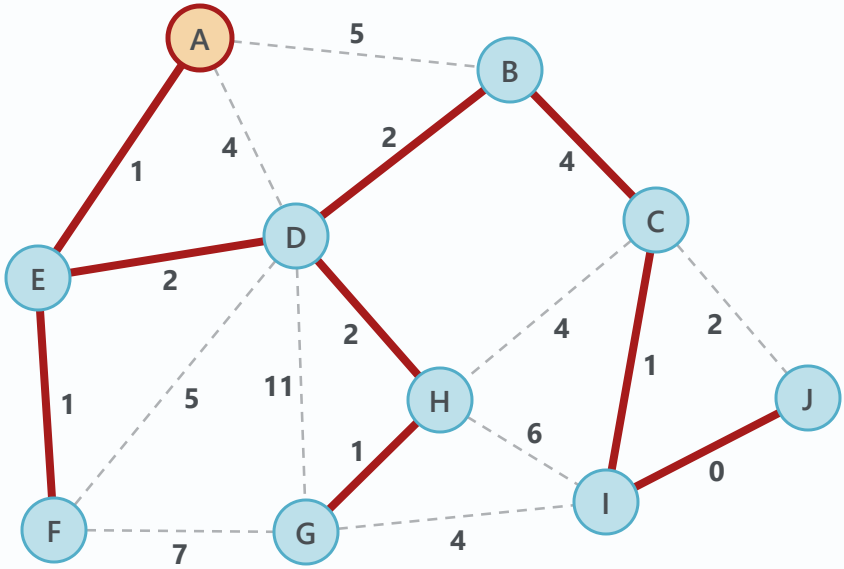
Total weight
14

I–J (0): lightest edge leaving the tree → **J** joins the tree. 9 = n − 1 edges: **done**, total weight **14**.

Prim's Algorithm Step by Step (from A)

priority queue

~~C–J = 2~~
~~H–C = 4~~
~~A–D = 4~~
~~G–I = 4~~
~~A–B = 5~~
~~F–D = 5~~
~~H–I = 6~~
~~F–G = 7~~
~~D–G = 11~~



Tree vertex
10 / 10

Total weight
14

Prim's tree: A–E, E–F, E–D, D–B, D–H, H–G, B–C, C–I, I–J. **The same total weight (14)** as Kruskal. If equal weights come out in another order (e.g. H–C instead of B–C), the tree may differ but still weighs 14.

Kruskal vs. Prim

	Kruskal	Prim
Idea	tries edges from lightest to heaviest, merging trees	grows a single tree from one vertex
Along the way	a forest (several trees)	a single tree
Data structures	sorted edge list + Union-Find	priority queue + marker array, adjacency list
Cycle check	find(u) == find(v)	is v already in the tree?
Cost	O(m log n) (sorting dominates)	O(m log n) (with a heap); $O(n^2)$ with a matrix
When?	sparse graph, edges already in a list	adjacency list at hand; $O(n^2)$ version for very dense graphs

Both are greedy and find the same **total weight** on the same graph (14 in the example). With equal edge weights, the trees they find may differ.

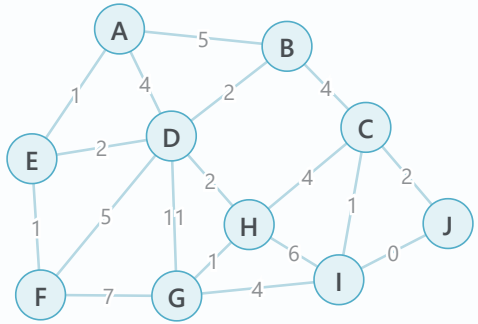
Lab: Minimum Spanning Trees

Kruskal: 10-vertex example graph

Input (edges: A-B:4, ...) `A-B:5, A-D:4, A-E:1, B-C:4, B-D:2, C-H:4, C-I:1, C-J:2, D-E:2, D-F:5, D-G`

CODE · **step 1 / 47**

```
1  List<int[]> kruskal(int n, List<int[]> …
2    E.sort((a, b) ->           // {u, v, w}
3        Integer.compare(a[2], b[2]));
4    UnionFind uf = new UnionFind(n);
5    List<int[]> T = new ArrayList<>();
6    for (int[] e : E) {
7      if (uf.union(e[0], e[1]))  // no cy…
8        T.add(e);
9      if (T.size() == n - 1) break;
10   }
11   return T;
12 }
```



E sorted empty

`T.size() = 0` `weight = 0`

RETURN VALUE –

kruskal(n = 10, E) is called; the graph has 10 vertices and 18 edges.

SUMMARY

Summary

Terms
Vertex, edge, degree, direction, weight, path, cycle, and connected component.

Representations
An adjacency list for sparse graphs; a matrix for fast edge queries.

Algorithm	Structure	Time (with a list)	What it gives
BFS	queue	O(n + m)	level order, unweighted shortest paths
DFS	stack / recursion	O(n + m)	depth-first traversal, cycles, components
Topological	queue / DFS	O(n + m)	dependency order for a DAG
Dijkstra	priority queue	O((n + m) log n)	weighted shortest paths (weights ≥ 0)
Kruskal	sorting + Union-Find	O(m log n)	minimum spanning tree
Prim	priority queue	O(m log n)	minimum spanning tree

The data structure shapes the algorithm: BFS uses a queue, DFS a stack, and Dijkstra and Prim a priority queue; topological sorting is defined only for DAGs.

SUMMARY

Interactive Simulators

Undirected Graph Simulator

Add and remove vertices and edges, run BFS and DFS, see connected components, and view the adjacency list and matrix.

Directed Graph Simulator

Arrowed edges, in- and out-degrees, and directed traversal.

Dijkstra's Algorithm Simulator

Watch shortest paths in a weighted graph being found step by step.

Application: Course Prerequisite Planner → Plans courses semester by semester with Kahn's algorithm and catches prerequisite cycles.

SUMMARY

Sample Problems

1. At most how many edges can a simple undirected graph on 10 vertices have? If the degrees add up to 30, how many edges are there?
2. Which representation would you use for a road network with n = 10,000 vertices and m = 30,000 edges? Why?
3. Run BFS and DFS on the traversal example graph starting from **H** (neighbors in alphabetical order).
4. If we write DFS with a queue instead of a stack, which algorithm do we get?
5. In the weighted shortest-path example, if edge B → C weighed 1 instead of 5, what would the shortest path from A to D be?
6. Why was edge C–J rejected in the Kruskal example?
7. If Prim starts from J instead of A on the same graph, what are the first three edges chosen and the total weight?

▶ **Show answers**

SUMMARY

Exercises

1. Undirected edges (0,1), (0,2), (1,3), (2,3), (3,4), (4,5): what are the BFS order and distances from 0? (neighbors in increasing order)
2. How many cells of this graph's adjacency matrix are 1? How many entries does its adjacency list hold in total?
3. For the digraph 0→1, 1→2, 2→0, 2→3, while examining which edge does `hasCycle(0)` return true?
4. How many topological orders does the DAG 0→1, 0→2, 1→3, 2→3 have? List them.
5. If `prev = [-1, 0, 0, 1, 3]`, what is the path from 0 to 4? From 2 to 4?
6. Is a 5-vertex cycle (C_5) bipartite? How many connected components does it have?
7. In lazy Prim, at most how many entries enter the queue? When does an entry become stale?

▶ **Answers / hints**

SUMMARY

Further Reading

- Goodrich, Tamassia, and Goldwasser, *Data Structures and Algorithms in Java*, 6th ed., 2014.
- Cormen, Leiserson, Rivest, and Stein, *Introduction to Algorithms*, 4th ed., 2022.
- Sedgewick and Wayne, *Algorithms*, 4th ed., 2011.

The End

Chapter 14: Graphs

Study questions · Simulators: Undirected Graph Simulator, Directed Graph Simulator, Dijkstra's Algorithm Simulator, Course Prerequisite Planner · Glossary