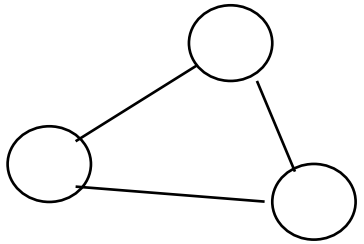
Introduction to Graphs

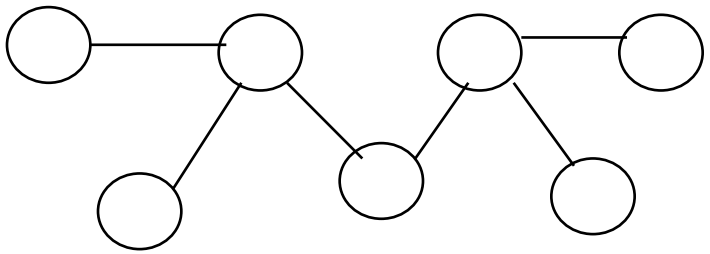
Dr. Chung-Wen Albert Tsao

Graphs and Trees

- A path that begins and ends at the same node is called a *cycle*.
  Example:

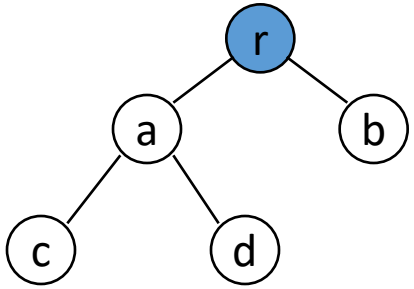


- *Two nodes are connected* if there is a path between them.
- *A graph is connected* if every pair of its nodes is connected.
- A graph is *acyclic* if it doesn't have any cycle.
- *tree* is connected, acyclic graph. |E| = |V| -1



Rooted Trees

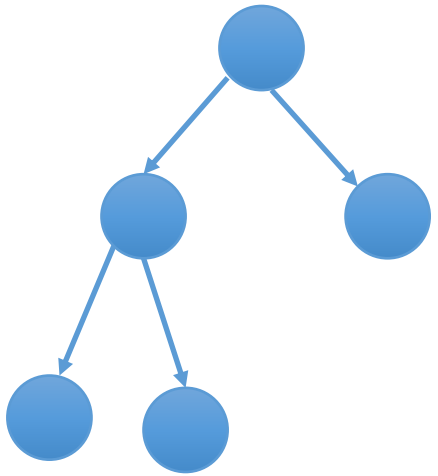
- **Definition**: A rooted tree is a tree in which one vertex is distinguished from the others and is called the root.



- The level of a vertex is the number of edges along the unique path between it and the root.
- The height of a rooted tree is the maximum level to any vertex of the tree.
- The children of a vertex *v* are those vertices that are adjacent to *v* and one level farther away from the root than *v*.

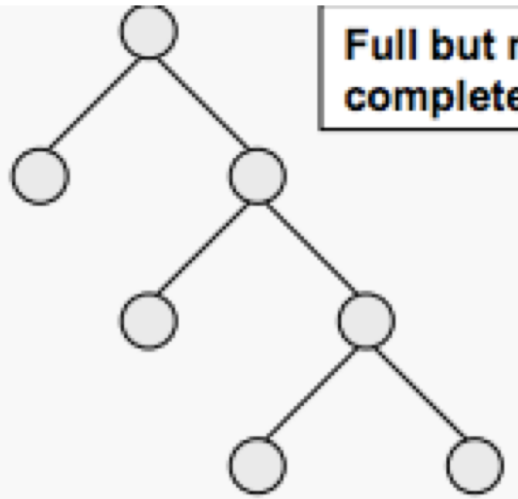
Binary Tree

- Tree with at most 2 children for each node
- Usually draw root at the top
- Can be complete, or full



Binary Tree

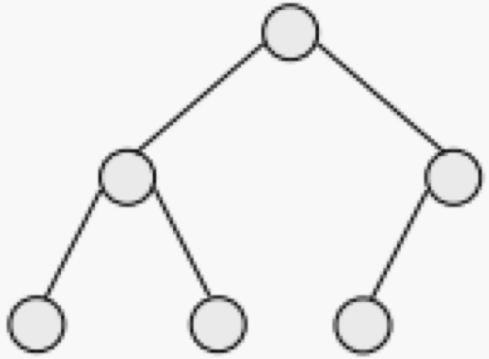
Definition: a binary tree T is *full* if each node is either a leaf or possesses exactly two child nodes.



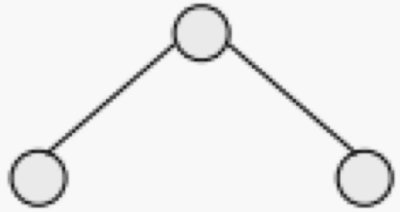
Full but not complete.

Neither complete nor full.

Definition: a binary tree T with n levels is *complete* if all levels except possibly the last are completely full, and the last level has all its nodes to the left side.



Complete but not full.



Full and complete.

http://courses.cs.vt.edu/~cs3114/Fall09/wmcquain/Notes/T03a.BinaryTreeTheorems.pdf

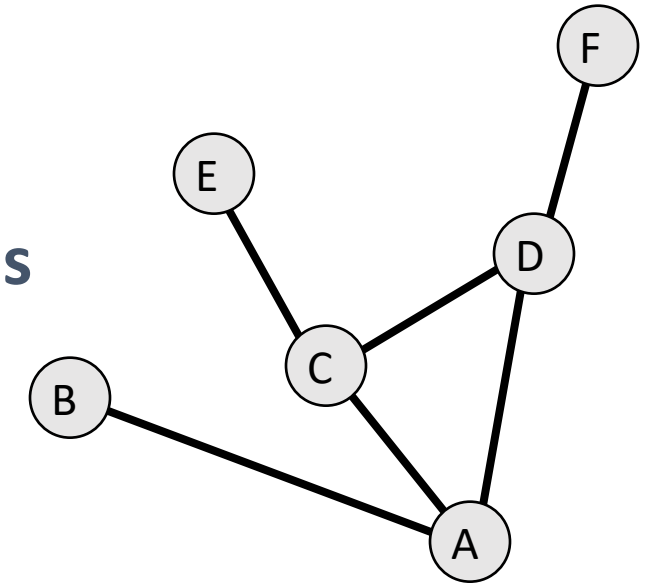
Outline

- Graph definition
- Graph representations
  - Adjacency matrix
  - Adjacency list
- Graph search
  - Breadth first search (BFS)
  - Depth first search (DFS)
- Topological sort

- Part of the slides are based on material from Prof. Jianhua Ruan, The University of Texas at San Antonio

Graph

- A graph is a structure that consists of a set of vertices and a set of edges between pairs of vertices
- A graph is a pair (V, E), where
  - V is a set of **vertices**
  - E is a set of **pairs of vertices**, called **edges**



- Example:
  - V = {A, B, C, D, E, F}
  - E = {(A,B), (A,C), (A,D), (C,D), (C,E), (D, F)}

Graphs

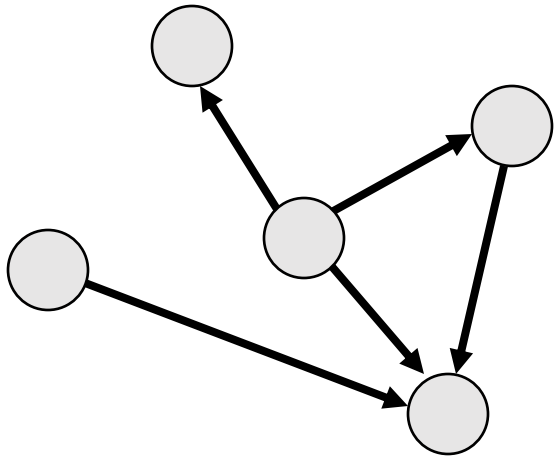
- We will typically express running times in terms of **|E|** and **|V|** (often dropping the |'s)
  - If $|E| \approx |V|^2$ the graph is ***dense***
  - If $|E| \approx |V|$ the graph is ***sparse***
- If you know you are dealing with dense or sparse graphs, different data structures may make sense

Graph Variations

- A ***connected*** *graph* has a path from every vertex to every other
- In an ***undirected*** *graph:*
  - Edge (u,v) = edge (v,u)
  - No self-loops
- In a ***directed*** graph:
  - Edge (u,v) goes from vertex u to vertex v, notated u→v
- A ***weighted*** *graph* associates weights with either the edges or the vertices
  - E.g., a road map: edges might be weighted w/ distance
- A ***multigraph*** allows multiple edges between the same vertices
  - E.g., the call graph in a program (a function can get called from multiple points in another function)

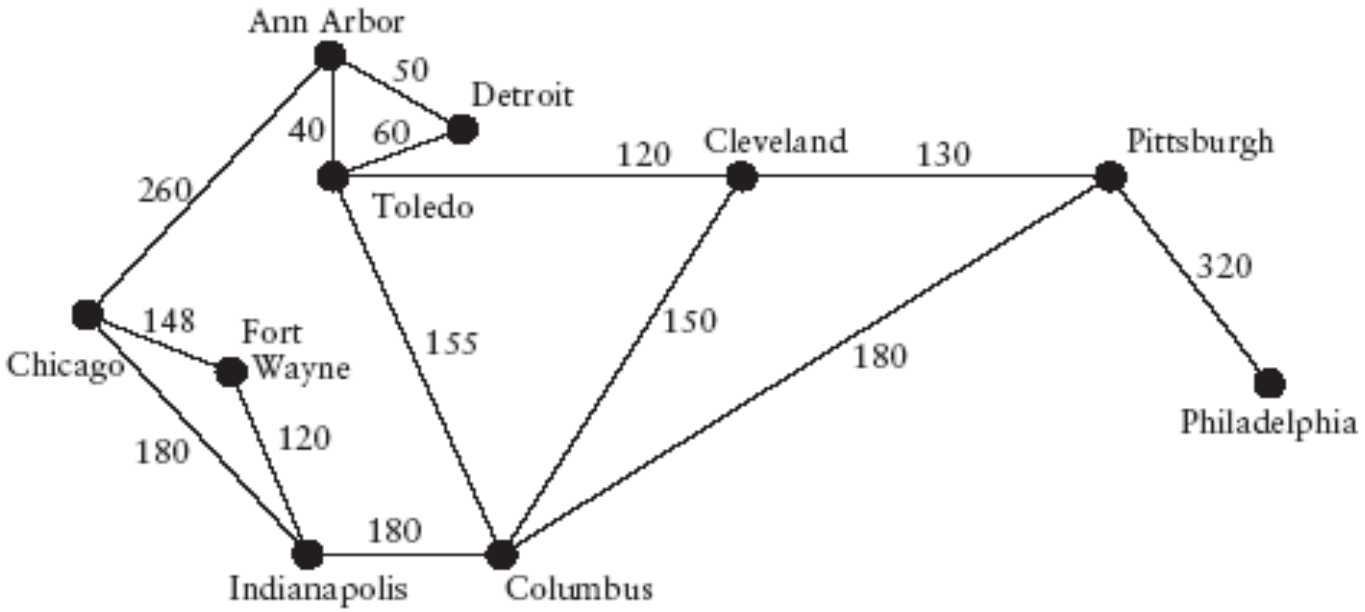
Directed graph (digraph)

- **Directed** edge
  - ordered pair of vertices (u,v)
- **Undirected** edge
  - unordered pair of vertices (u,v)
- A graph with directed edges is called a **directed graph** or **digraph**
- A graph with undirected edges is an **undirected graph** or simply a **graph**



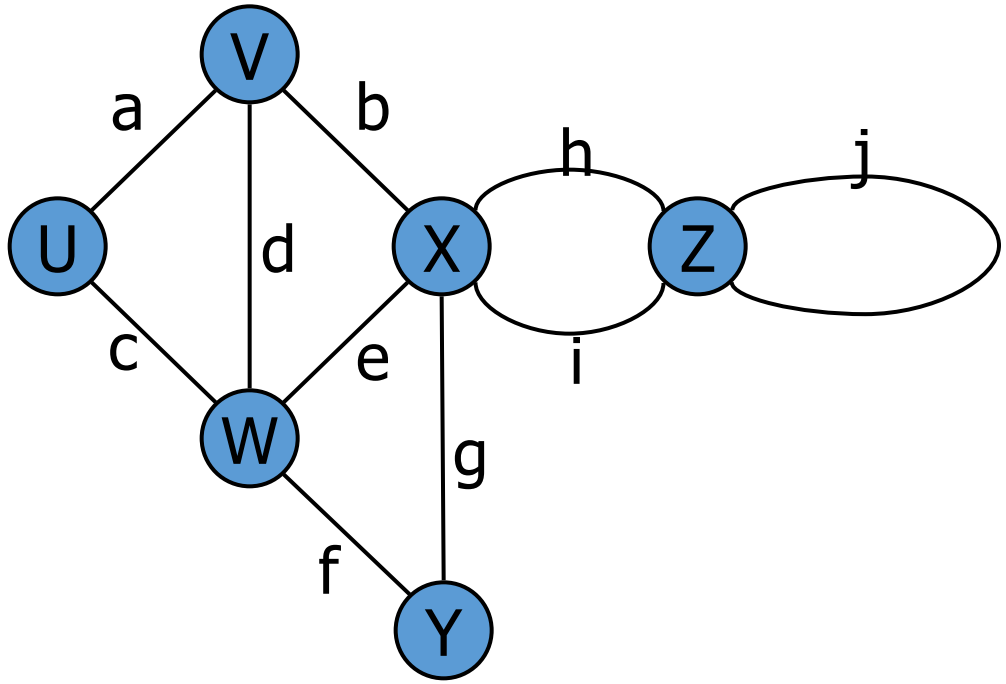
Weighted Graphs

- The edges in a graph may have values associated with them known as their **weights**
- A **graph with weighted edges** is known as a **weighted** graph



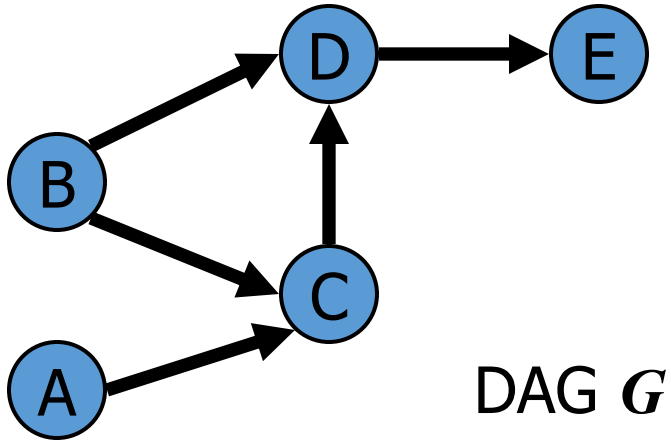
Terminology

- End **vertices** (or endpoints) of an edge
  - U and V are the endpoints of a
- **Edges** incident on a vertex
  - a, d, and b are incident on V
- **Adjacent** vertices
  - U and V are adjacent
- **Degree** of a vertex
  - X has degree 5
- Parallel edges
  - h and i are parallel edges
- Self-loop
  - j is a self-loop



DAG

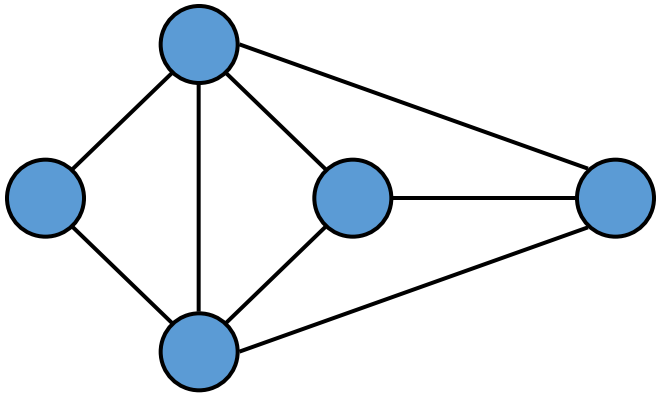
- A **Directed Acyclic Graph** (DAG) is a digraph that has **no** directed **cycles**



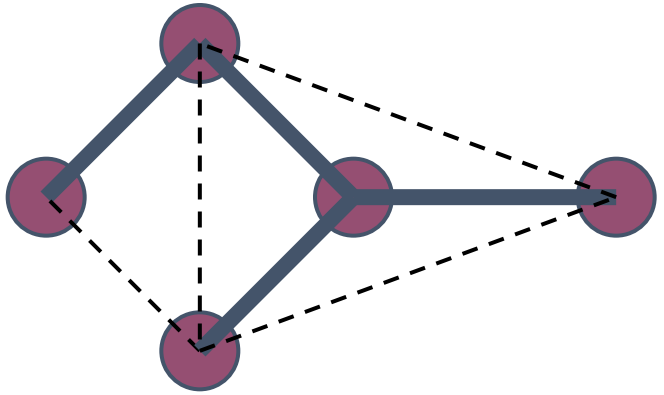
DAG G

Spanning Trees and Forests

- A spanning **tree** of a connected graph is a **spanning** subgraph that is a **tree**
- A spanning tree is not **unique** unless the graph is a tree
- A spanning forest of a graph is a spanning subgraph that is a forest



Graph



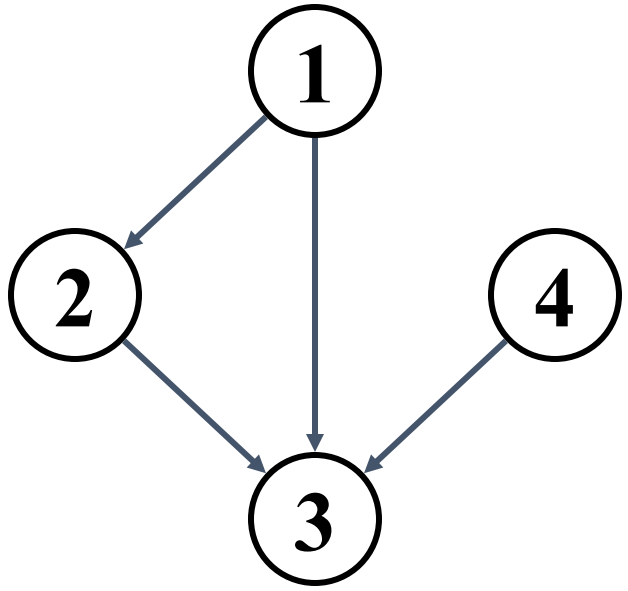
Spanning tree

Representing Graphs

- Assume **V** = {1, 2, …, *n*}
- An ***adjacency matrix*** represents the graph as a *n* x *n* matrix A:
  - $A[i, j]$ = 1 if edge $(i, j) \in E$
    = 0 if edge $(i, j) \notin E$
- For weighted graph
  - $A[i, j]$ = w_{ij} if edge $(i, j) \in E$
    = 0 if edge $(i, j) \notin E$
- For undirected graph
  - Matrix is symmetric: $A[i, j] = A[j, i]$

Directed Graphs: Adjacency **Matrix**

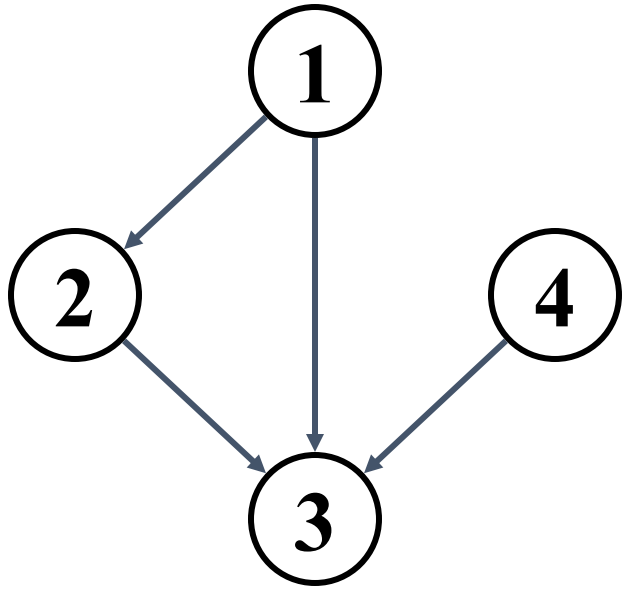
- Example:



A	1	2	3	4
1				
2				
3			??	
4				

Directed Graphs: Adjacency Matrix

- Example:



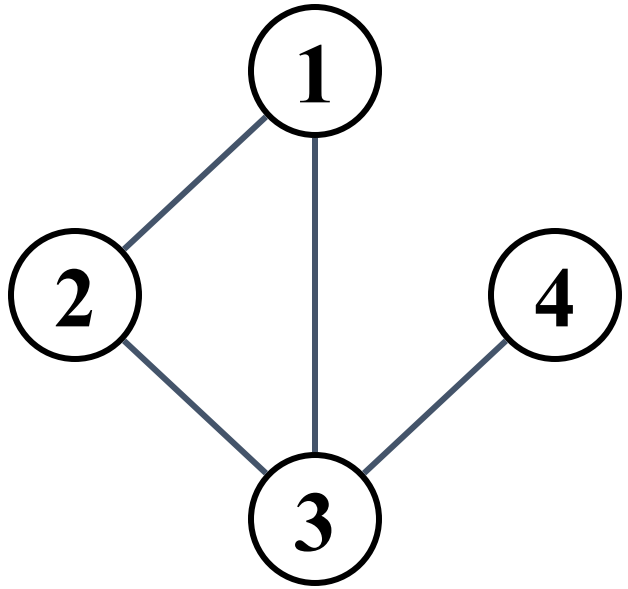
A	1	2	3	4
1	0	1	1	0
2	0	0	1	0
3	0	0	0	0
4	0	0	1	0

How much storage does the adjacency matrix require?

A: $O(V^2)$

(**Undirected**) Graphs: Adjacency Matrix

- Example:

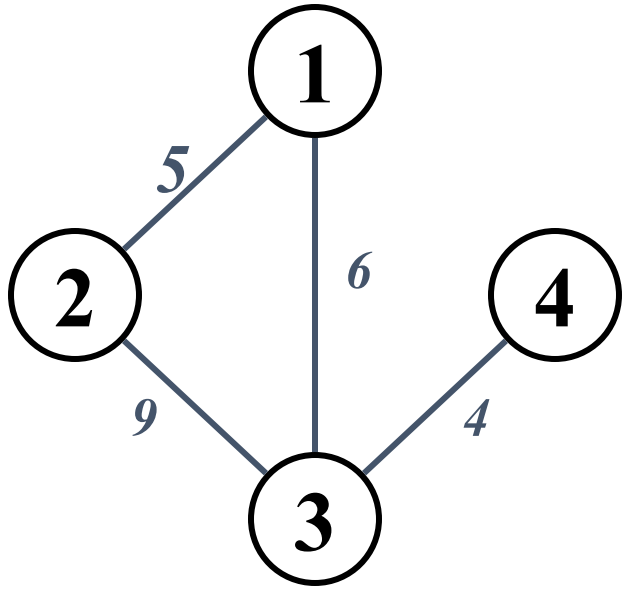


A	1	2	3	4
1	0	1	1	0
2	1	0	1	0
3	1	1	0	1
4	0	0	1	0

- Undirected graph → matrix is **symmetric**
- No self-loops → don't need diagonal

(Undirected) Graphs: Adjacency Matrix

- Example:



Weighted graph

A	1	2	3	4
1	0	**5**	6	0
2	**5**	0	9	0
3	6	9	0	4
4	0	0	4	0

Graphs: Adjacency Matrix

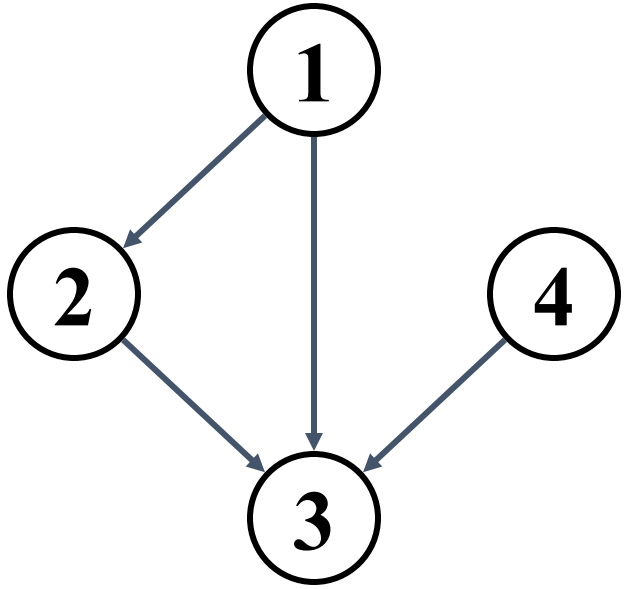
- Time to answer if there is an edge between vertex u and v: $\Theta(1)$
- Memory required: $\Theta(n^2)$ regardless of $|E|$
  - Usually too much storage for large graphs
  - But can be very efficient for small graphs
- Most large interesting graphs are sparse
  - E.g., road networks (due to limit on junctions)
  - For this reason the ***adjacency list*** is often a more appropriate representation

Graphs: Adjacency Matrix

- The adjacency matrix is a dense representation
  - Usually too much storage for large graphs
  - But can be very efficient for small graphs
- Most large interesting graphs are sparse
  - E.g., planar graphs, in which no edges cross, have $|E| = O(|V|)$ by Euler's formula
  - For this reason the ***adjacency list*** is often a more appropriate representation

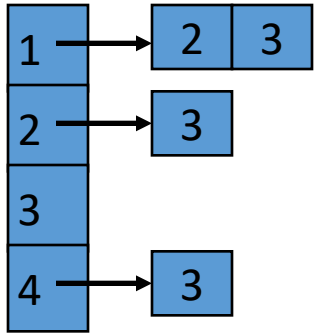
Graphs: **Adjacency List**

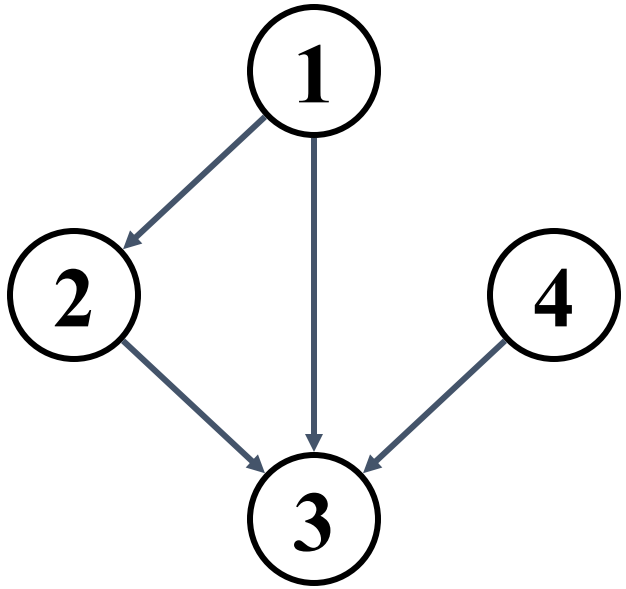
- Adjacency list: for each vertex $v \in V$, store a list of vertices adjacent to v
- Example:
  - **Adj**[1] = {**2,3**}
  - Adj[**2**] = {3}
  - Adj[3] = {}
  - Adj[4] = {3}
- Variation: can also keep a list of edges coming *into* vertex



Graph representations

- Adjacency list



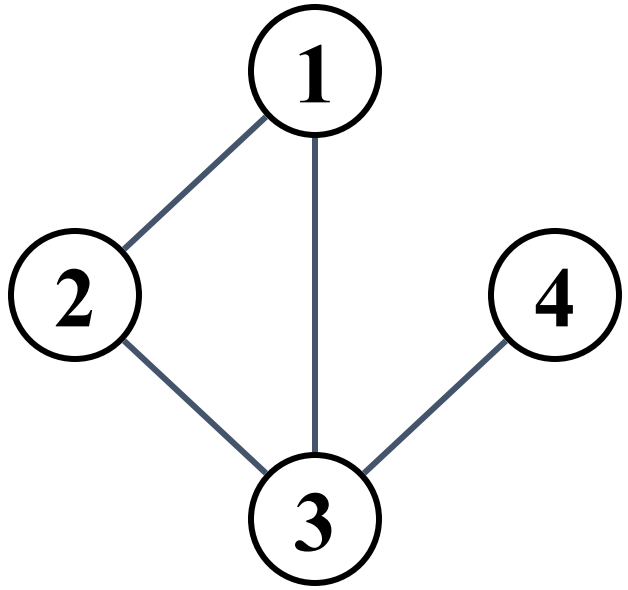


How much storage does the adjacency list require?

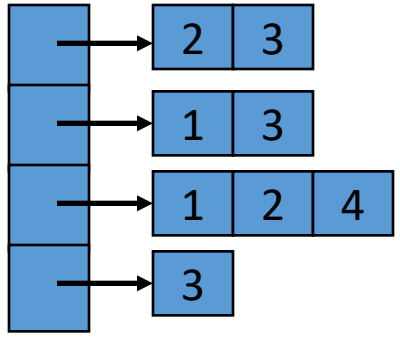
A: O(V+E)

Graph representations

- Undirected graph

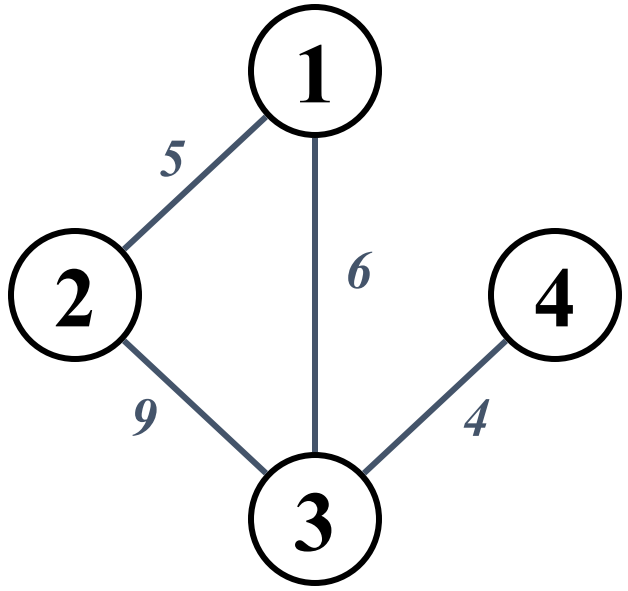


A	1	2	3	4
1	0	1	1	0
2	1	0	1	0
3	1	1	0	1
4	0	0	1	0

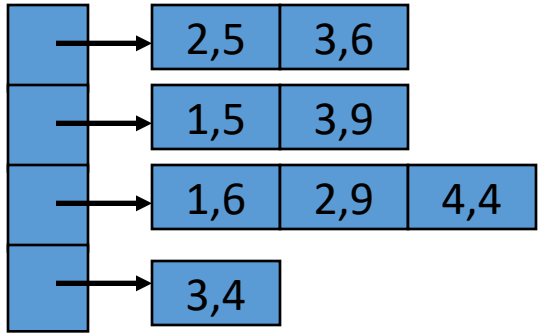


Graph representations

- Weighted graph



A	1	2	3	4
1	0	5	6	0
2	5	0	9	0
3	6	9	0	4
4	0	0	4	0



Graphs: Adjacency List

- How much storage is required?
- For directed graphs
  - |adj[v]| = out-degree(v)
  - Total # of items in adjacency lists is
    Σ out-degree(v) = |E|
- For undirected graphs
  - |adj[v]| = degree(v)
  - # items in adjacency lists is
    Σ degree(v) = 2 |E|
- So: Adjacency lists take Θ(V+E) storage
- Time needed to test if edge (u, v) $\in$ E is O(n)

Tradeoffs between the two representations

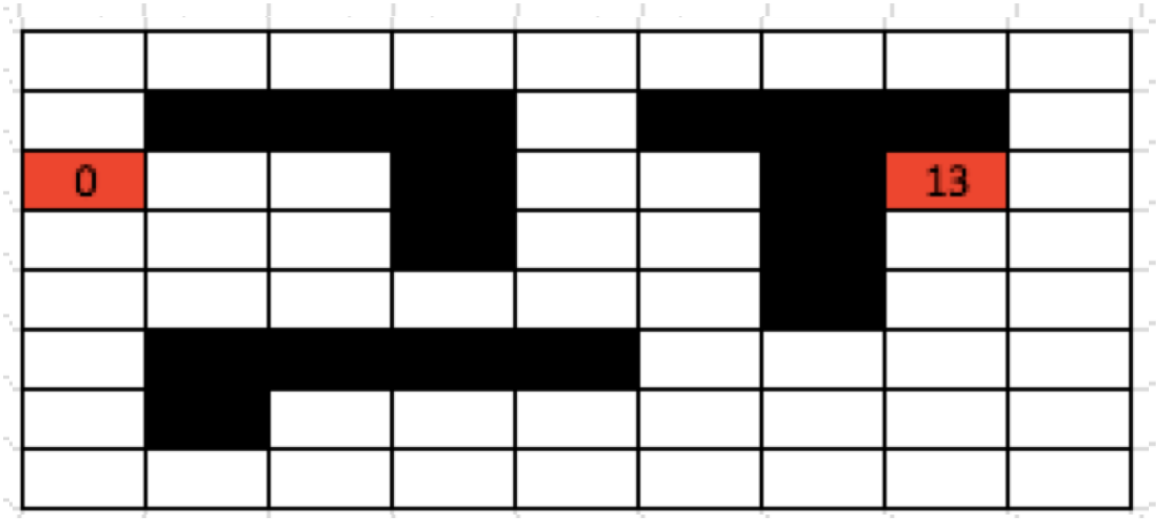
$|V| = n$, $|E| = m$

	Adj Matrix	Adj List
test $(u, v) \in E$	$\Theta(1)$	$O(n)$
Degree(u)	$\Theta(n)$	$O(n)$
Memory	$\Theta(n^2)$	$\Theta(n+m)$
Edge insertion	$\Theta(1)$	$\Theta(1)$
Edge deletion	$\Theta(1)$	$O(n)$
Graph traversal	$\Theta(n^2)$	$\Theta(n+m)$

Both representations are very useful and have different properties.

Breadth-First Search (BFS)

- "Explore" a graph, turning it into a **tree**
  - One vertex at a time
  - Expand frontier of explored vertices across the *breadth* of the frontier
- Builds a (shortest-Path) tree over the graph
  - Pick a *source vertex* to be the root
  - Find ("discover") its children, then their children, etc.



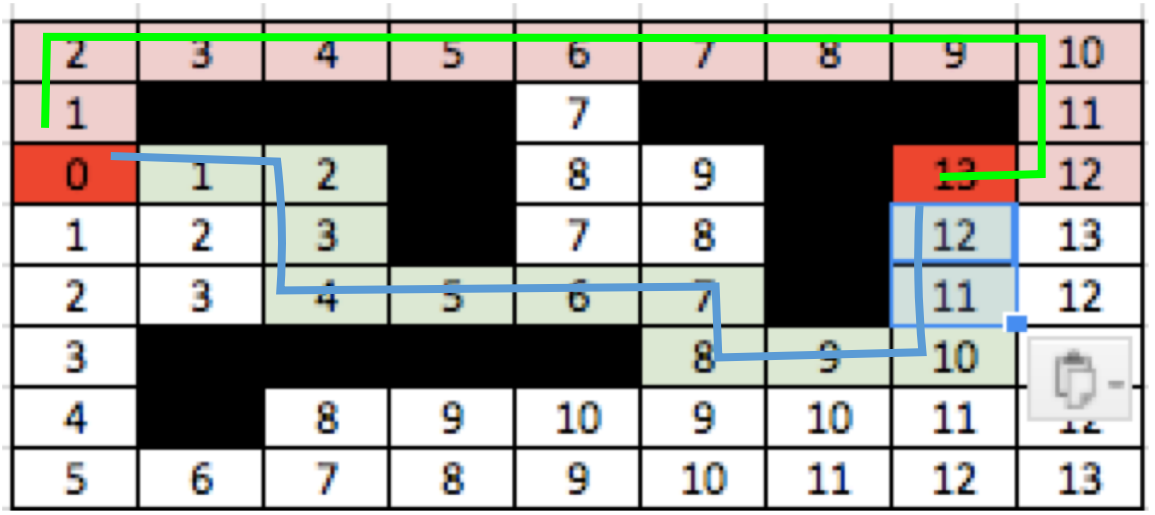
Breadth-First Search (BFS)

- "Explore" a graph, turning it into a tree
  - One vertex at a time
  - Expand frontier of explored vertices across the *breadth* of the frontier
- Builds a (shortest-Path) tree over the graph
  - Pick a *source vertex* to be the root
  - Find ("discover") its children, then their children, etc.

2	3	4	5	6	7	8	9	10
1				7				11
0	1	2		8	9		13	12
1	2			7	8		12	13
2	3		5	6	7		11	12
3					8	9	10	11
4		8	9	10	9	10	11	12
5	6	7	8	9	10	11	12	13

Breadth-First Search (BFS)

- "Explore" a graph, turning it into a tree
  - One vertex at a time
  - Expand frontier of explored vertices across the *breadth* of the frontier
- Builds a (shortest-Path) tree over the graph
  - Find out the shortest path by back tracking



Breadth-First Search

- Associate vertex “colors” to guide the algorithm
  - **White** vertices have not been discovered
    - All vertices start out white
  - **Gray** vertices are discovered but not fully explored
    - They may be adjacent to white vertices
  - **Yellow** vertices are discovered and fully explored
    - They are adjacent only to black and gray vertices
- Explore vertices by scanning adjacency list of gray vertices

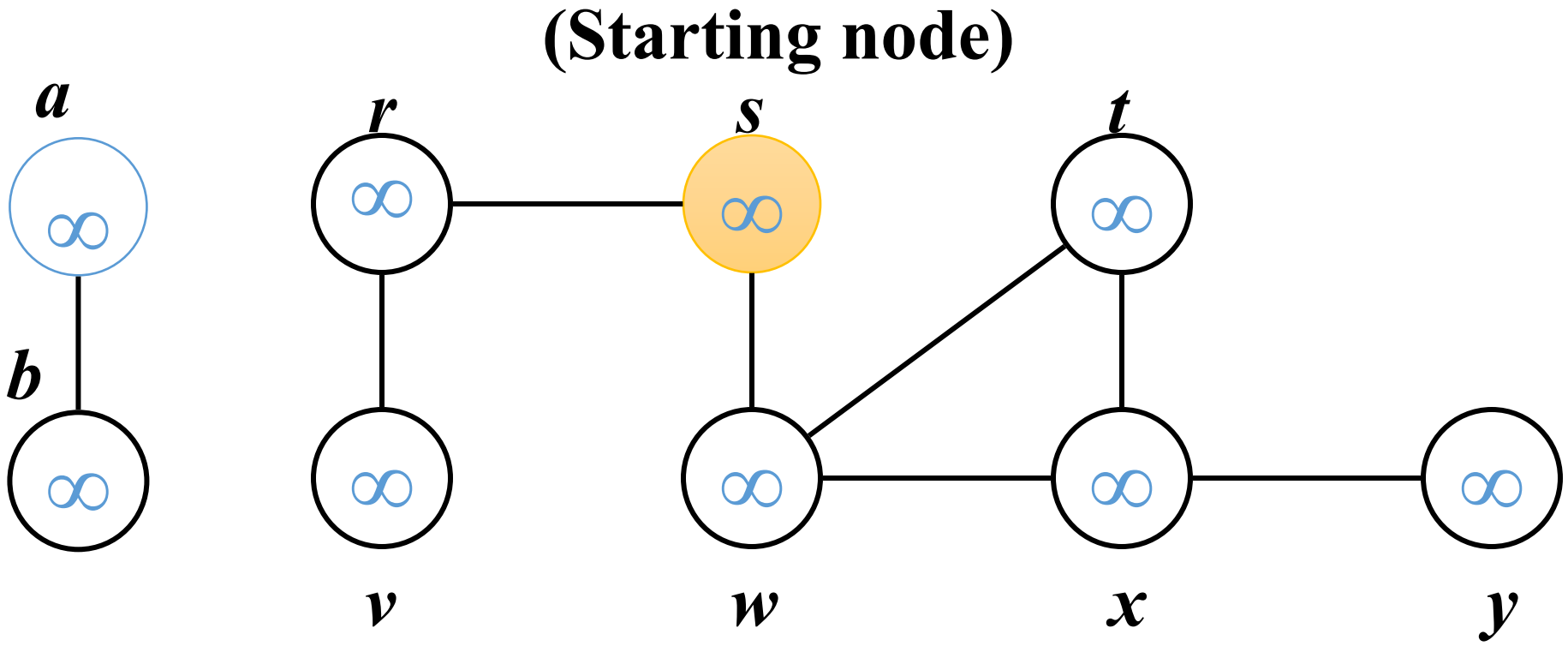
Breadth-First Search

```
BFS(G, s) {
    initialize vertices;
    Q = {s};         // Q is a queue, initialize to s
    while (Q not empty) {
        u = RemoveTop(Q);
        for each v ∈ u->adj {
            if (v->color == WHITE)
                v->color = GREY;
                v->d = u->d + 1;
                v->p = u;
                Enqueue(Q, v);
        }
        u->color = BLACK;
    }
}
```

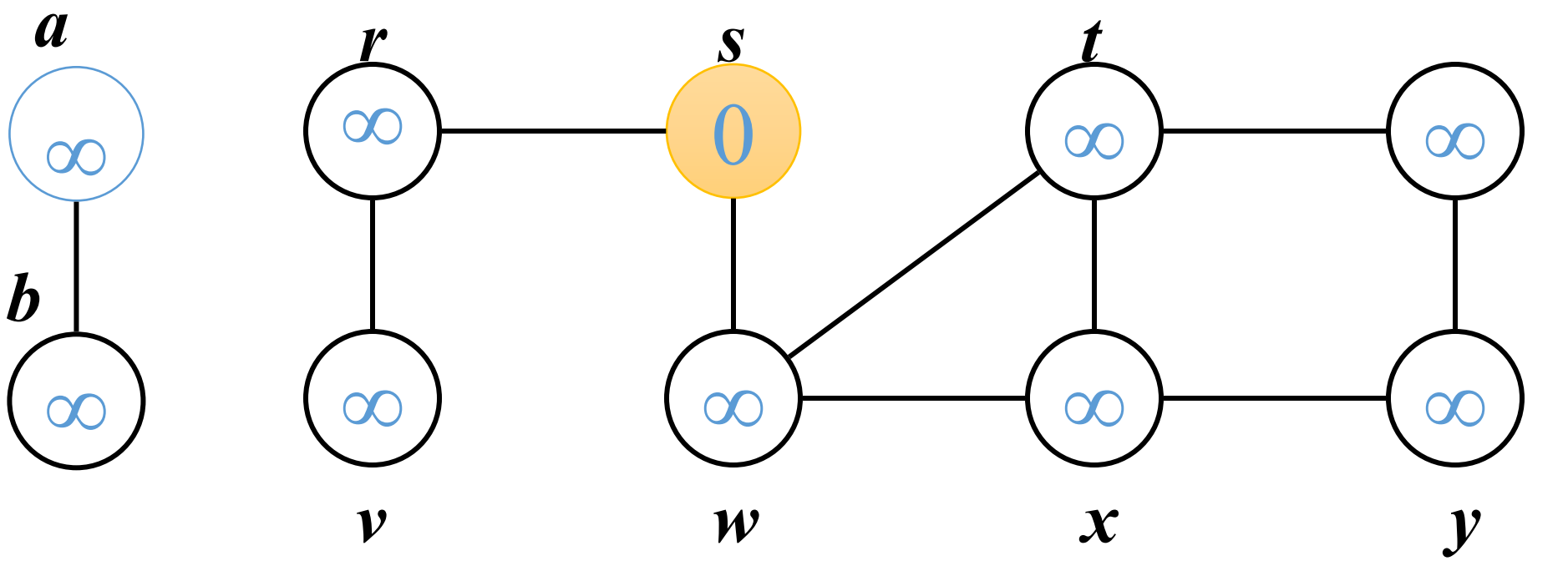
What does `v->d` represent? (distance, #steps)

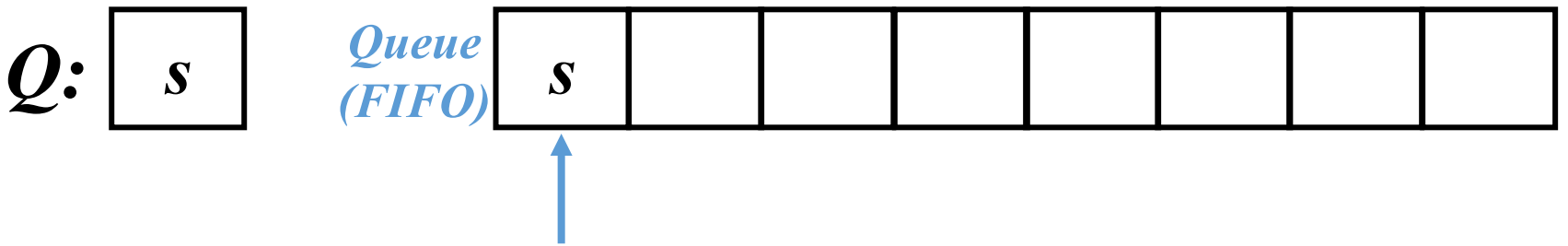
What does `v->p` represent? (preceding or parent node)

Breadth-First Search: Example

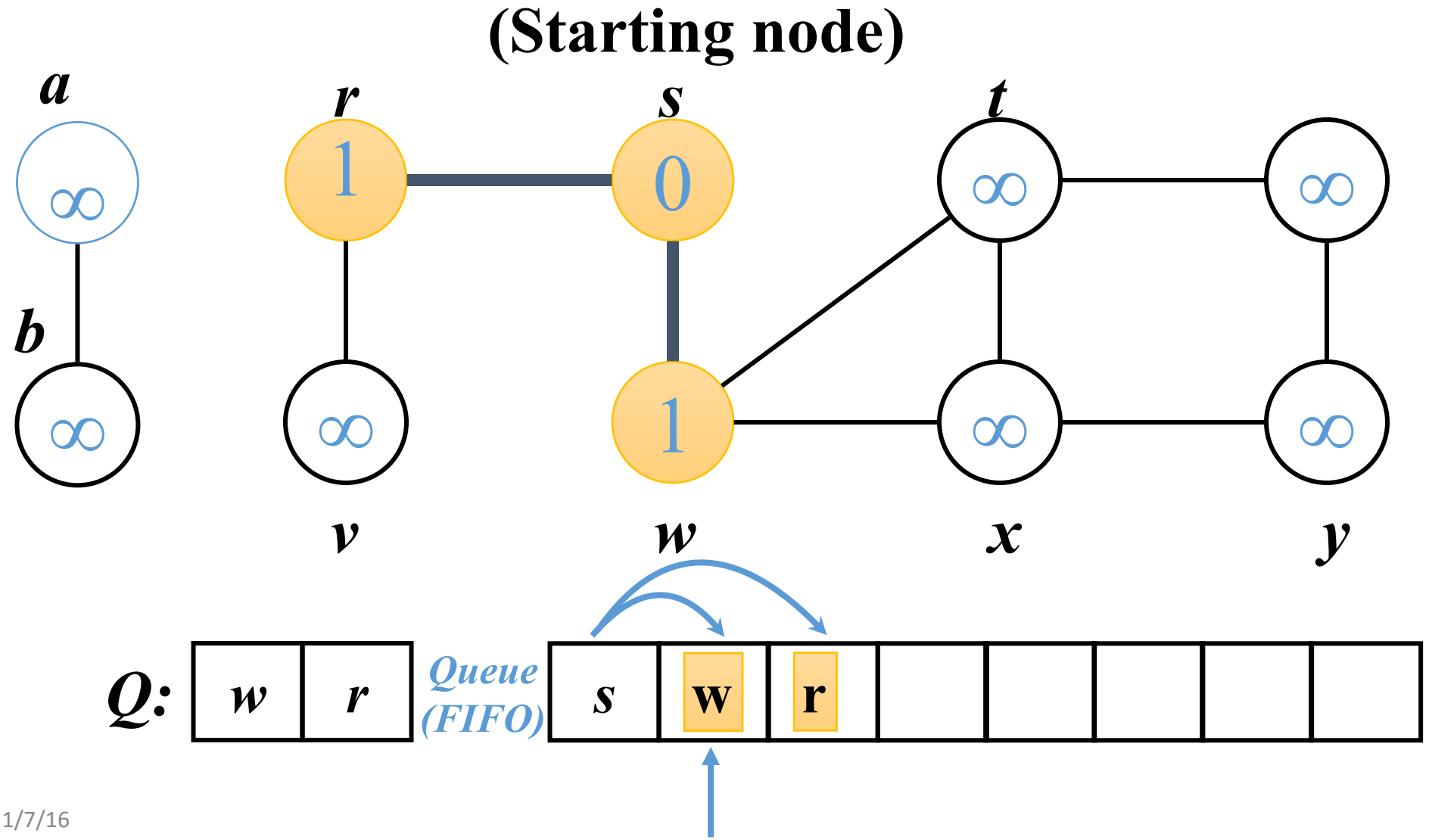


Breadth-First Search: Example

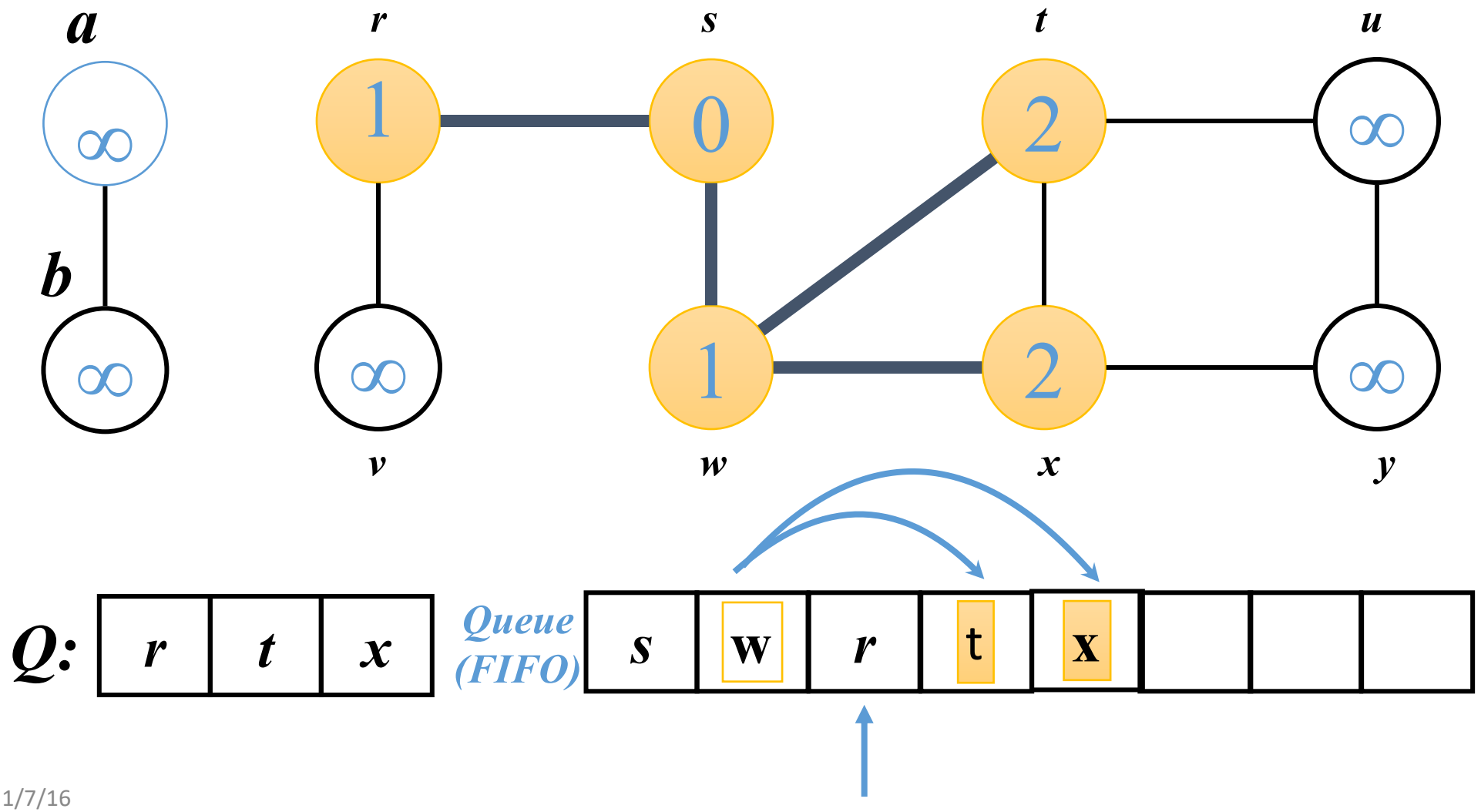




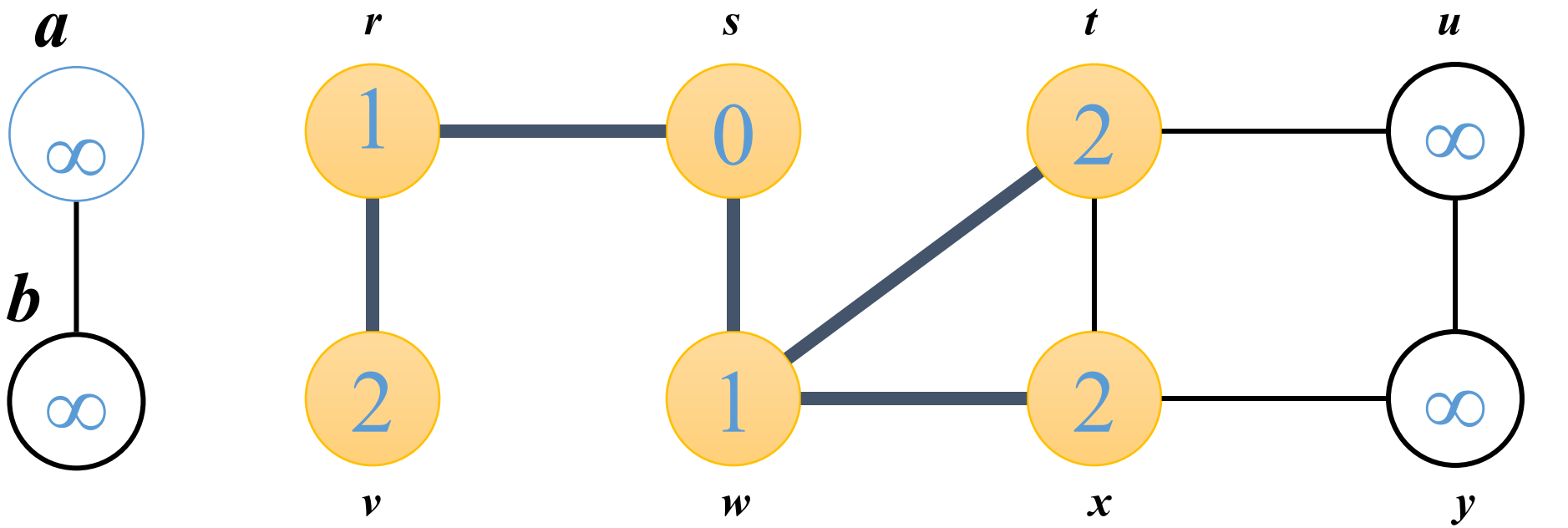
Breadth-First Search: Example

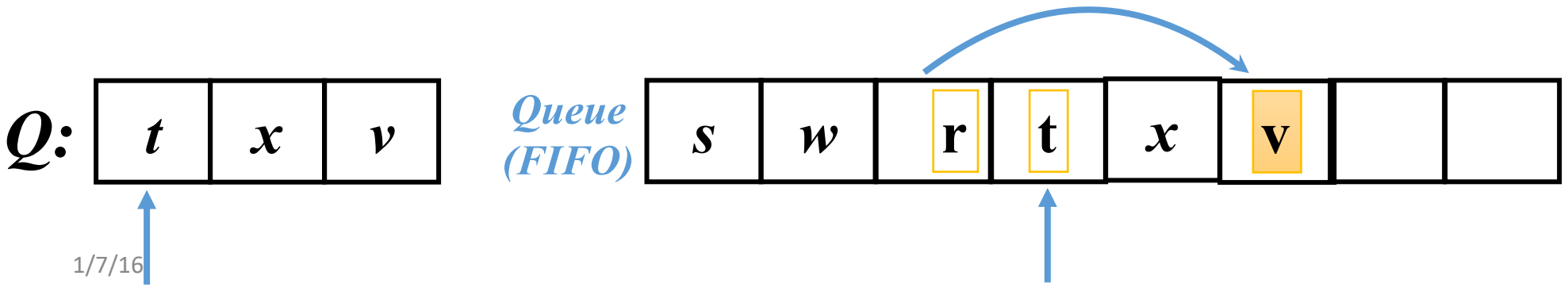


Breadth-First Search: Example

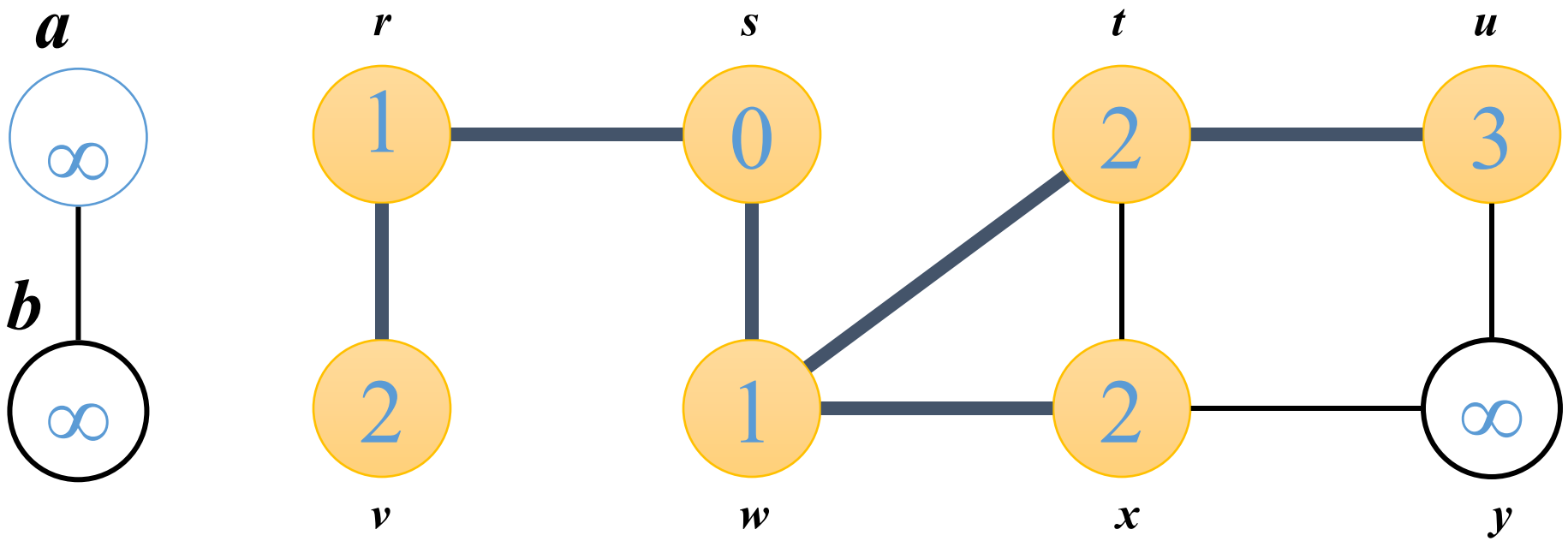


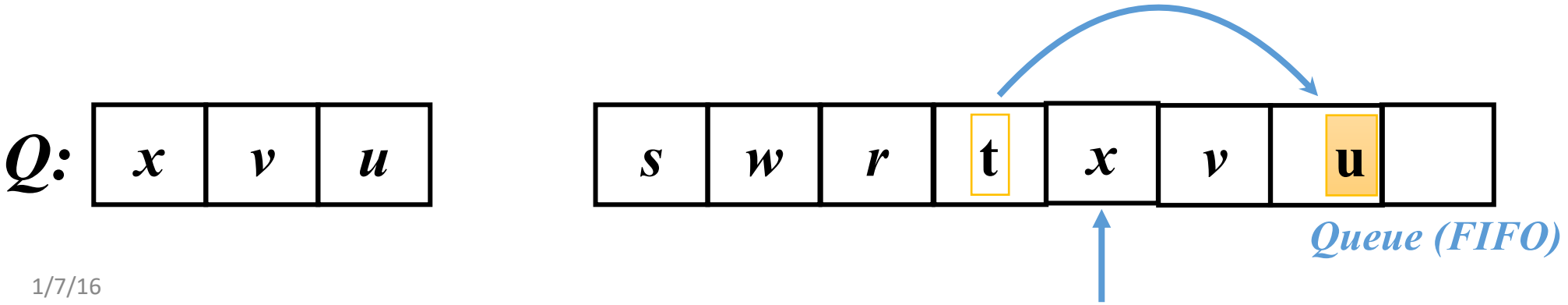
Breadth-First Search: Example



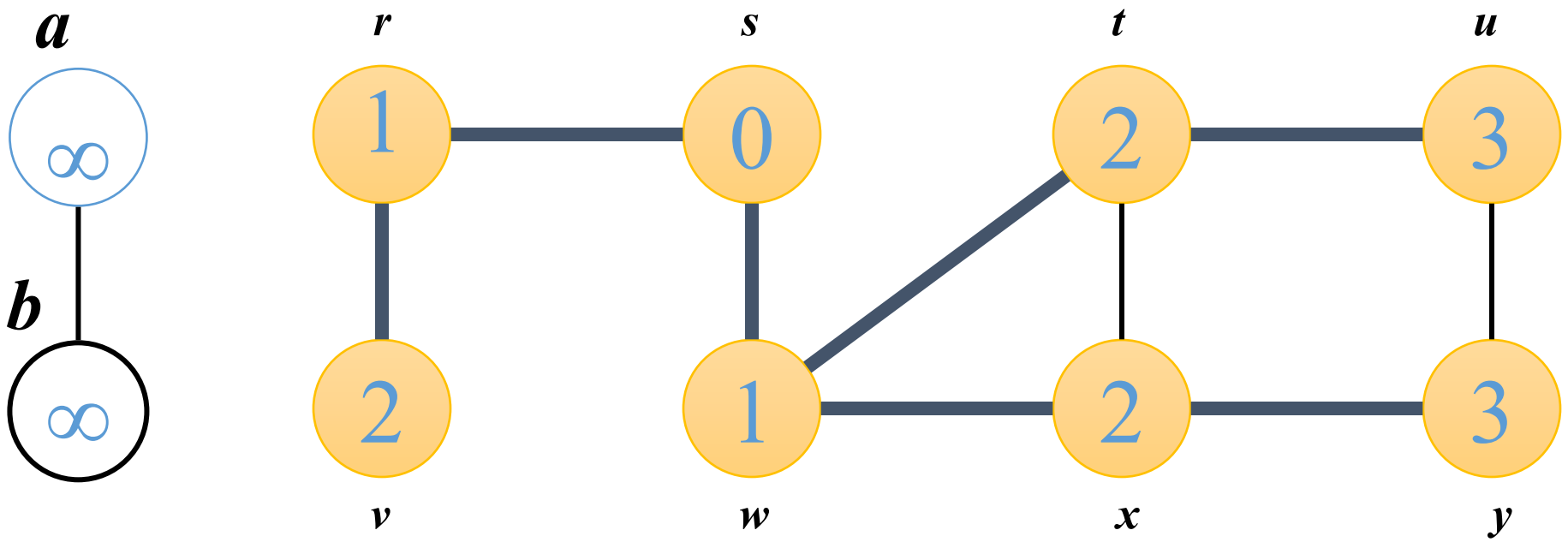


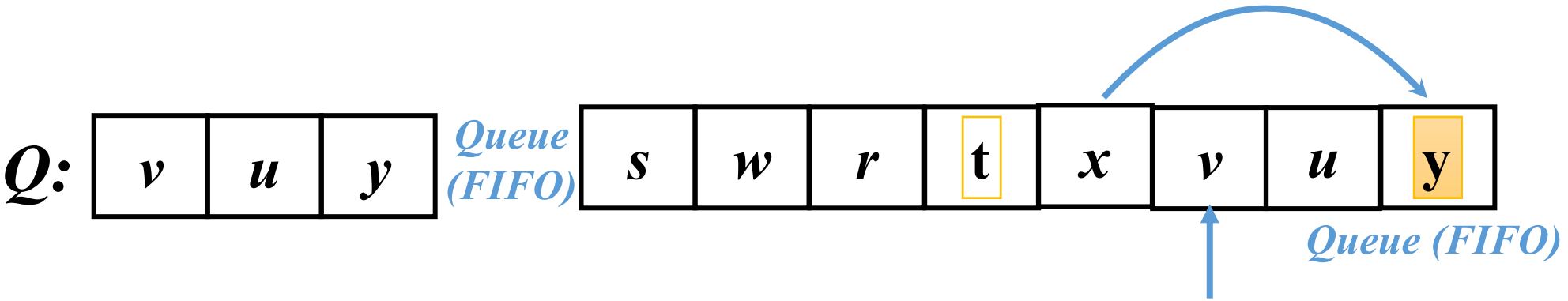
Breadth-First Search: Example



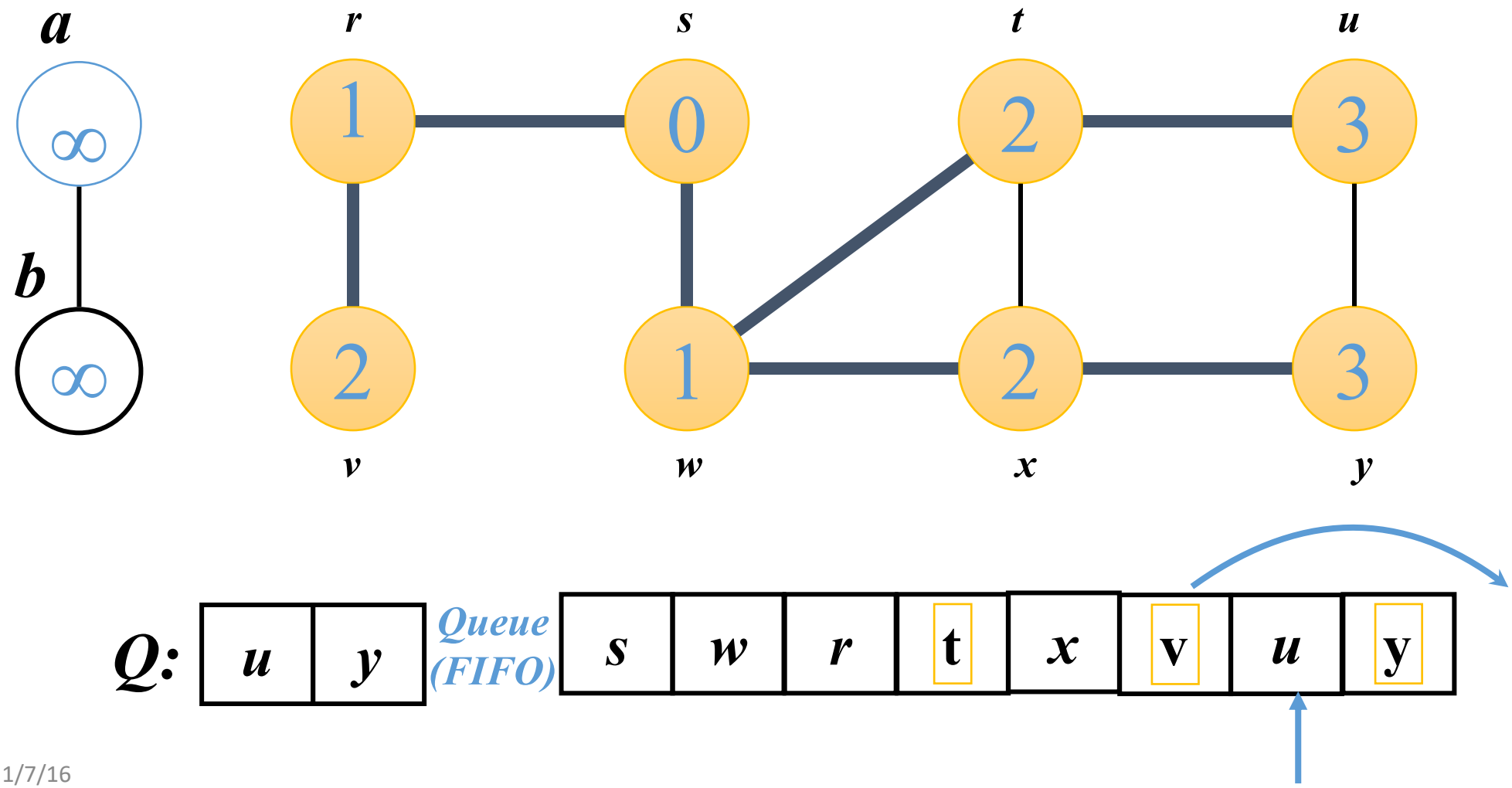


Breadth-First Search: Example

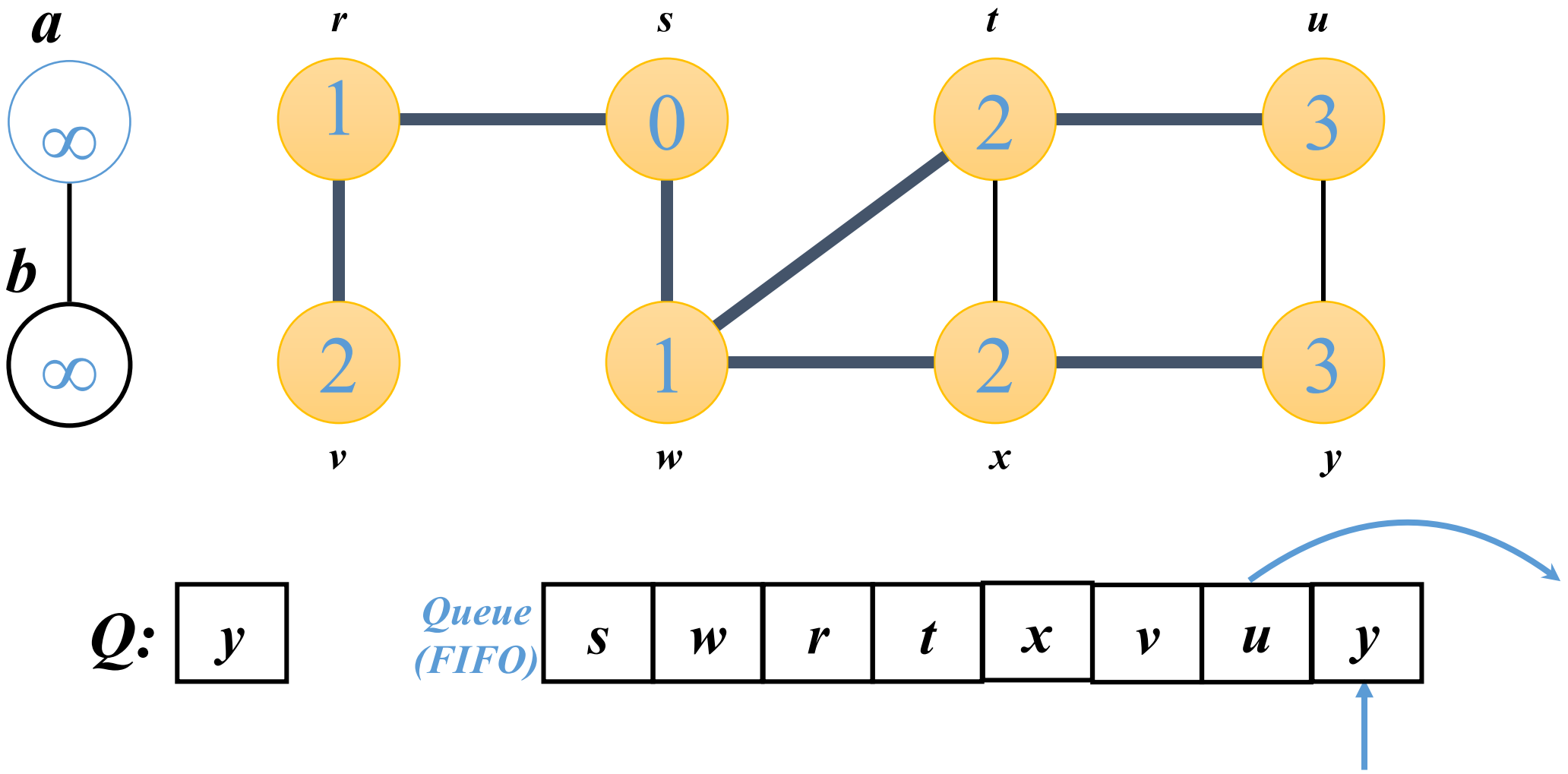




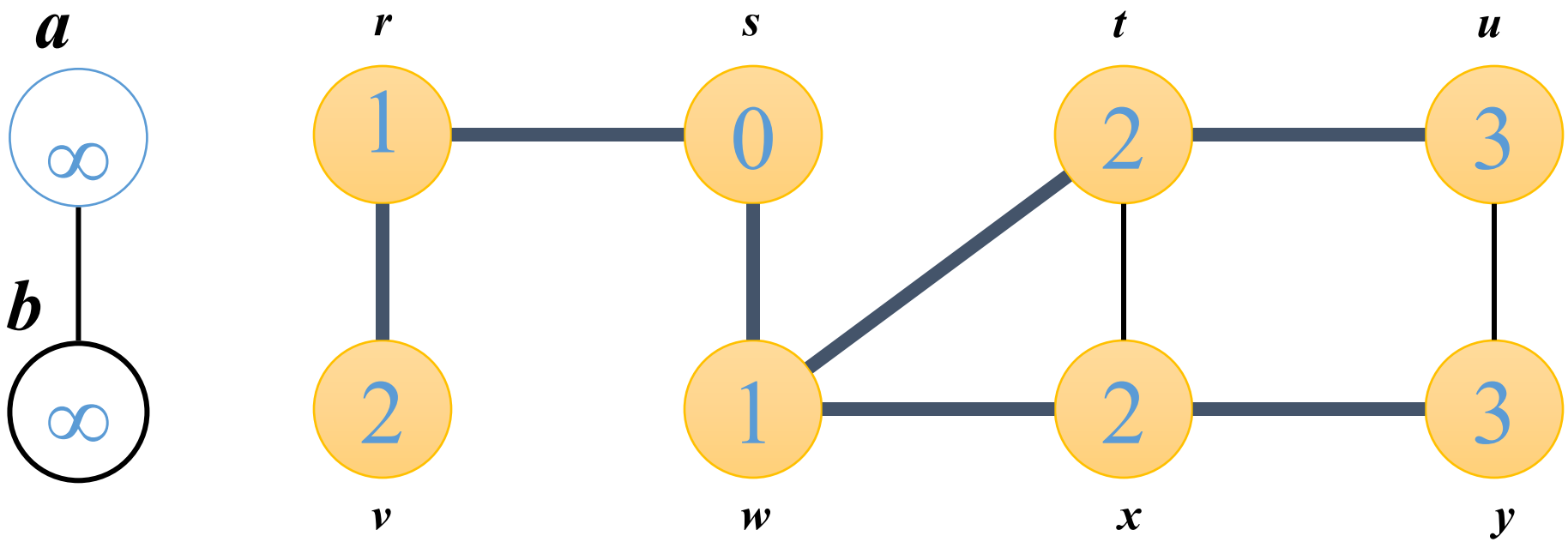
Breadth-First Search: Example



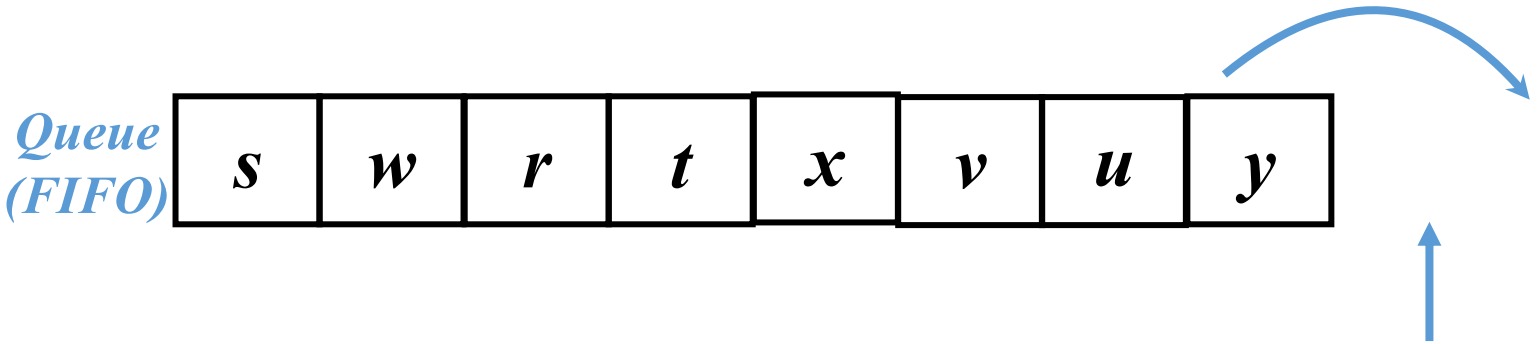
Breadth-First Search: Example



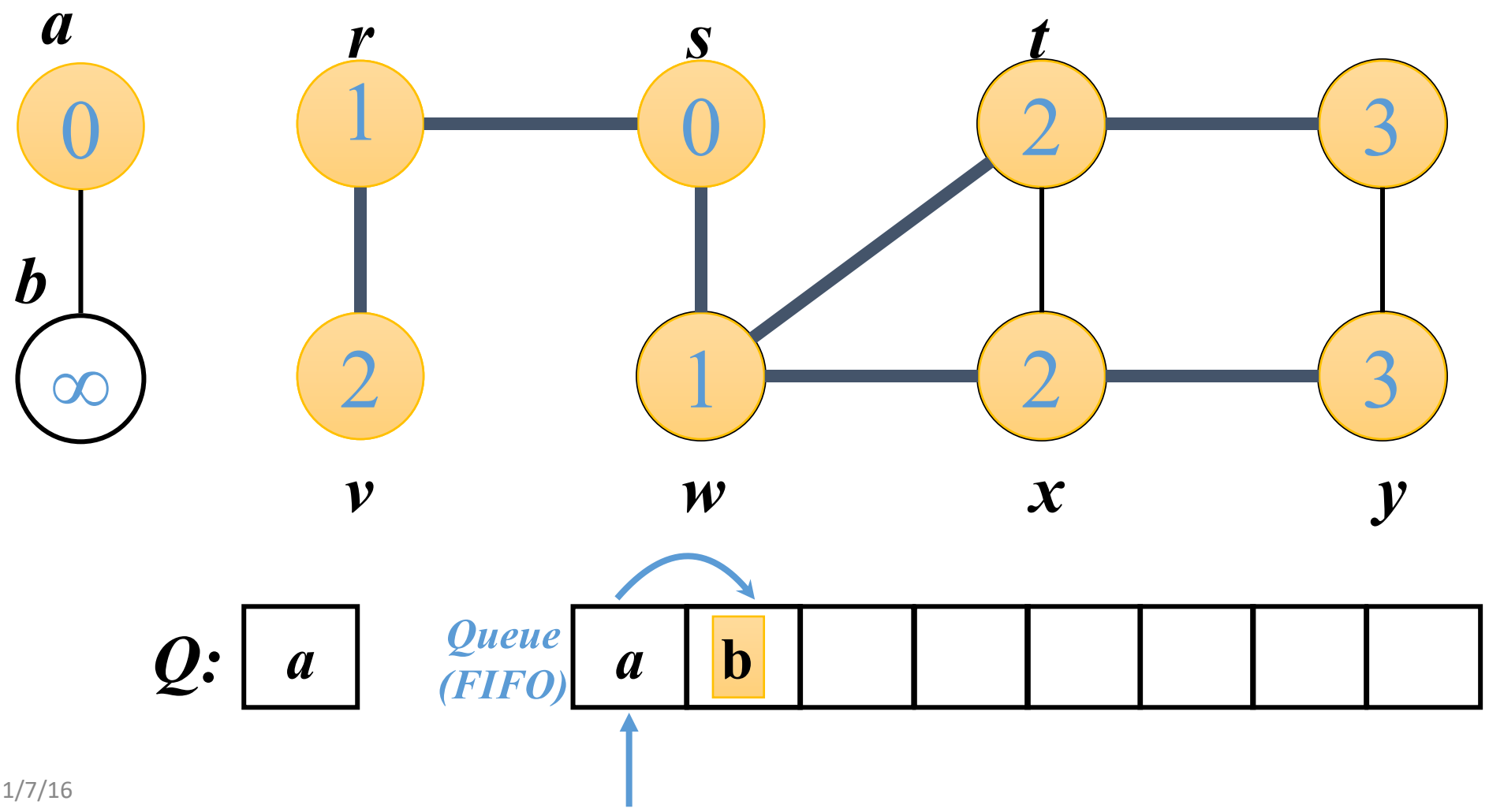
Breadth-First Search: Example



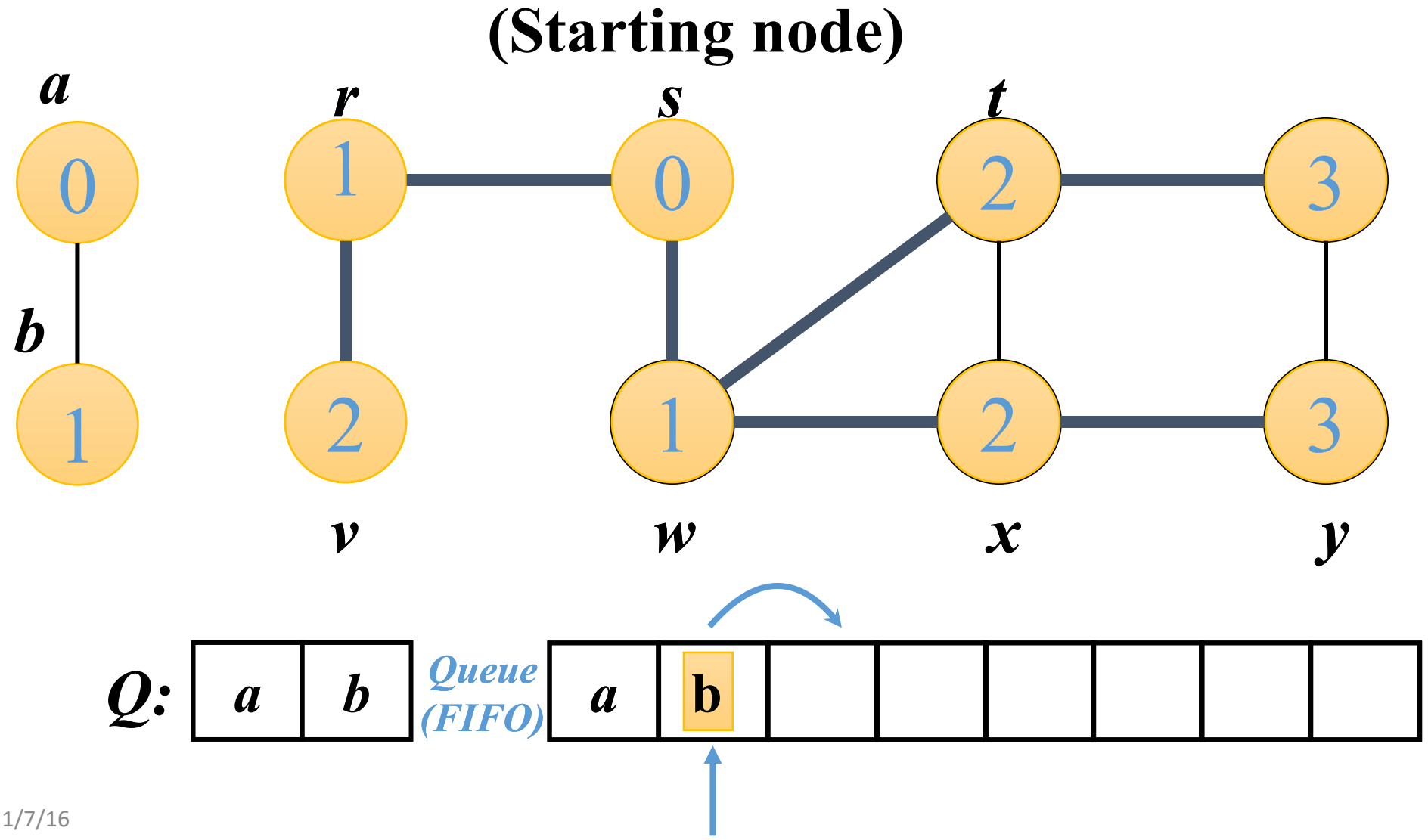
Q: Ø



Breadth-First Search: Example



Breadth-First Search: Example

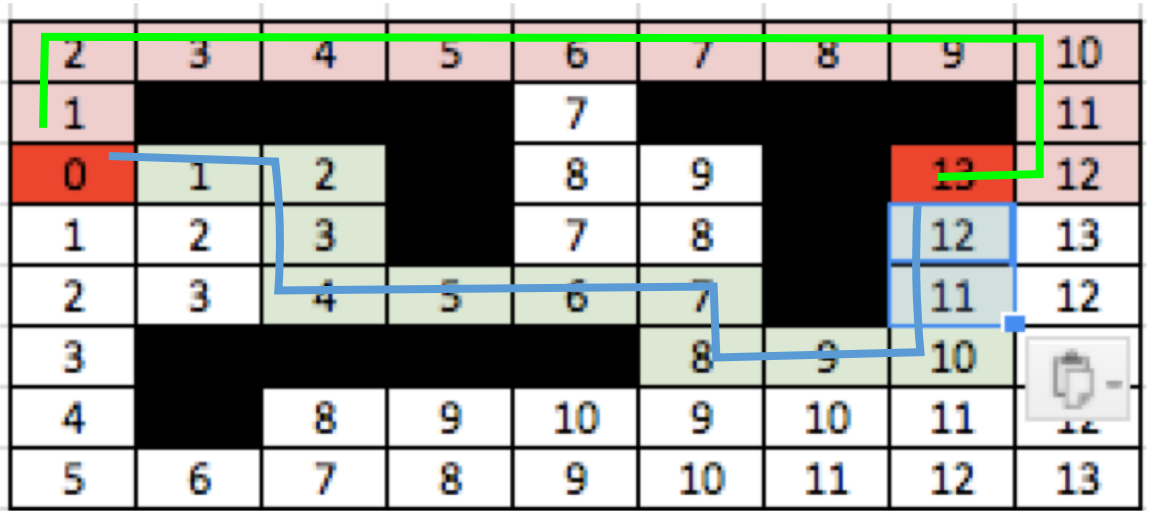


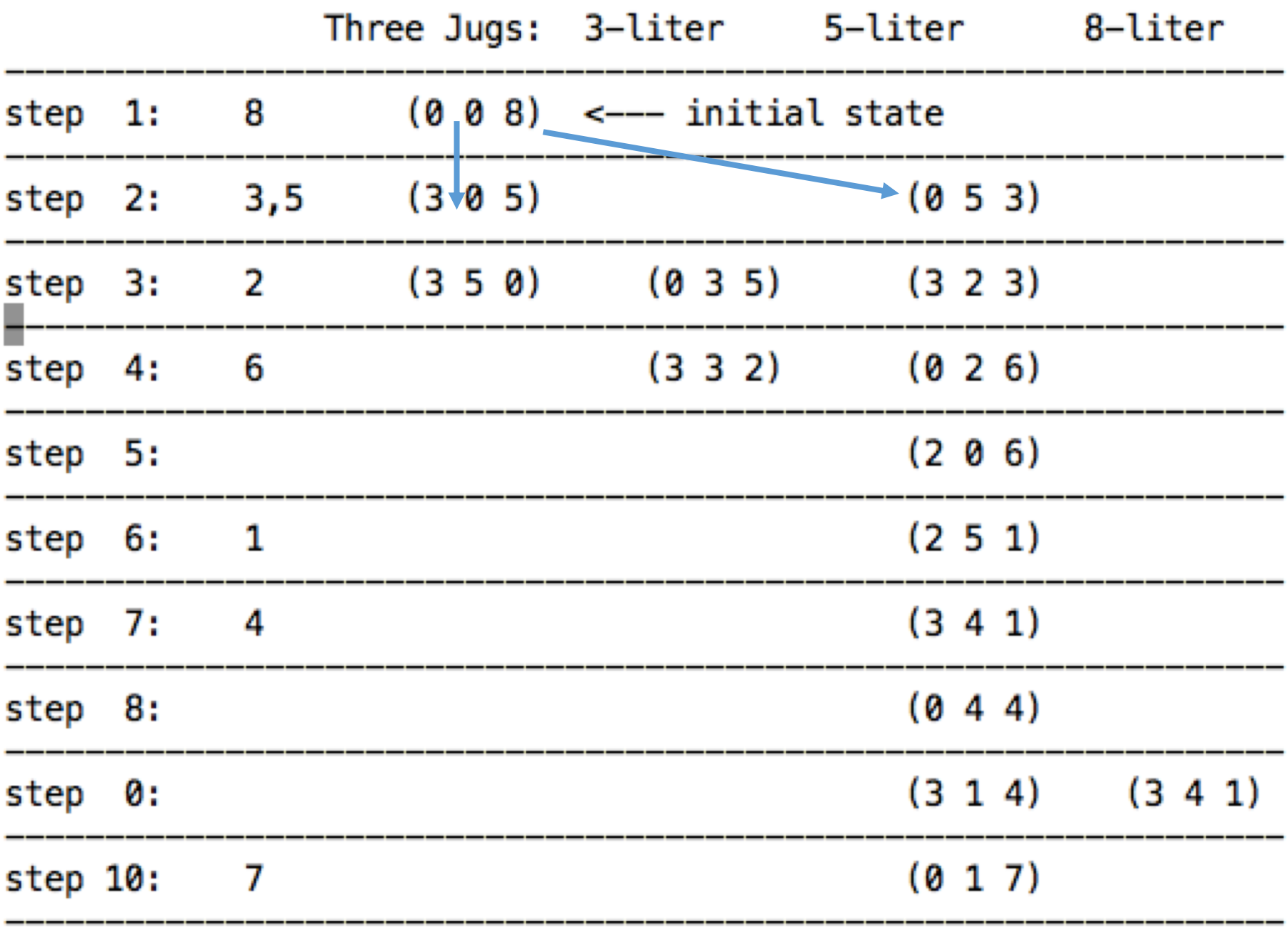
Breadth-First Search: Properties

- **Total running time: O(V+E)**
- **BFS** calculates the ***shortest-path distance*** from source to each node
  - Edge weight is assume to be **identical**
  - Shortest-path distance $\delta(s,v)$ = minimum number of edges from s to v, or ∞ if v not reachable from s
  - Proof given in the book (p. 472-5)
- **BFS** builds ***breadth-first tree***, in which paths to root represent shortest paths in G
  - Thus can use BFS to calculate shortest path from one vertex to another in O(V+E) time

Breadth-First Search (BFS)

- "Explore" a graph, turning it into a tree
  - One vertex at a time
  - Expand frontier of explored vertices across the *breadth* of the frontier
- Builds a (shortest-Path) tree over the graph
  - Find out the shortest path by back tracking




Three Jugs: 3-liter 5-liter 8-liter
step 1: 8 (0 0 8) <--- initial state
step 2: 3,5 (3 0 5) (0 5 3)
step 3: 2 (3 5 0) (0 3 5) (3 2 3)
step 4: 6 (3 3 2) (0 2 6)
step 5: (2 0 6)
step 6: 1 (2 5 1)
step 7: 4 (3 4 1)
step 8: (0 4 4)
step 0: (3 1 4) (3 4 1)
step 10: 7 (0 1 7)

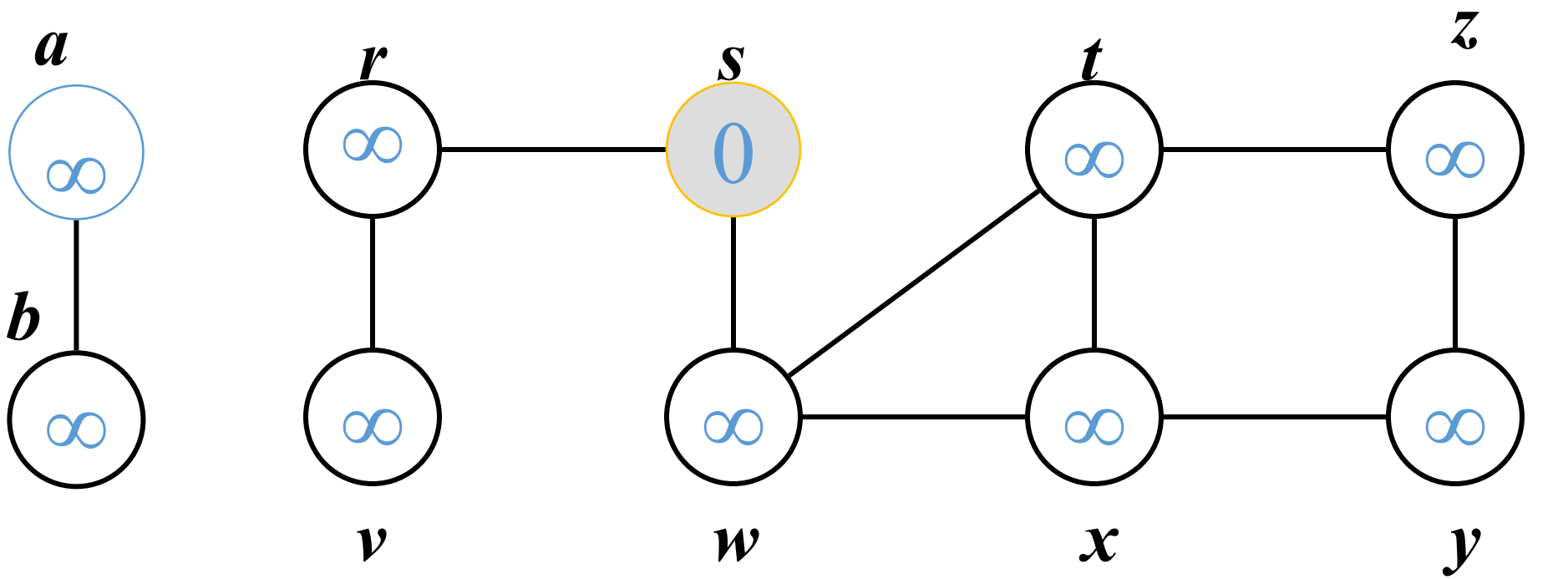
The Frog Puzzle

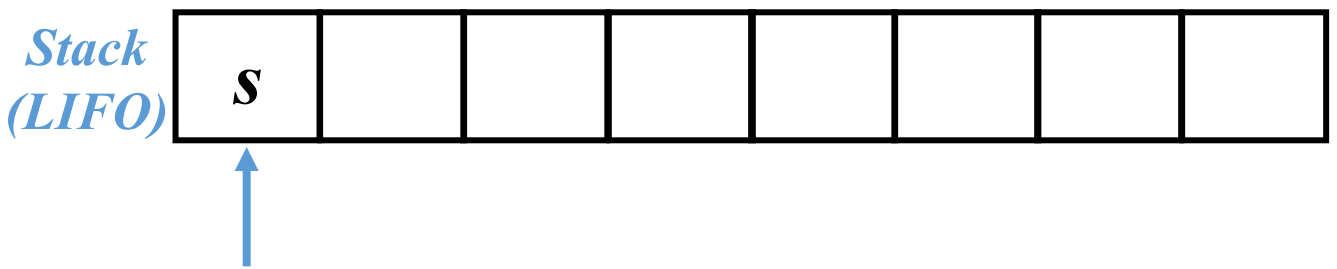
http://britton.disted.camosun.bc.ca/frog_puzzle.htm

```
------------------------------------------------------------
        Frog A moves to left only.       Frog Z moves to right only.
------------------------------------------------------------
0: ZZZ-AAA
 1: Z-ZZAAA   ZZ-ZAAA   ZZZA-AA   ZZZAA-A
 2: -ZZZAAA   ZZAZ-AA   ZZ-AZAA   ZZZAAA-
 3: ZZA-ZAA   ZZAZA-A   ZZAZAA-   -ZZAZAA   Z-ZAZAA
 4: Z-AZZAA   ZZAAZ-A   ZZA-AZA   ZAZ-ZAA
 5: -ZAZZAA   ZA-ZZAA   ZZAA-ZA   ZZAAZA-   Z-AZAZA   ZAZAZ-A
 6: AZ-ZZAA   -AZZZAA   ZZAAAZ-   ZZAA-AZ   -ZAZAZA   ZA-ZAZA   ZAZA-ZA   ZAZAZA-
 7: A-ZZZAA   ZZAAA-Z   AZ-ZAZA   -AZZAZA   ZAAZ-ZA   ZA-AZZA   ZAZAAZ-   ZAZA-AZ
 8: A-ZZAZA   AZAZ-ZA   ZAA-ZZA   ZAAZAZ-   -AZAZZA   ZAZAA-Z   ZA-AZAZ
 9: AZA-ZZA   AZAZAZ-   ZAAZA-Z   A-ZAZZA   -AZAZAZ   ZAA-ZAZ
10: A-AZZZA   AZAZA-Z   ZAA-AZZ   AAZ-ZZA   A-ZAZAZ   ZAAAZ-Z
11: AA-ZZZA   AZA-AZZ   ZAAA-ZZ   AAZ-ZAZ
12: A-AZAZZ   AZAA-ZZ   AA-ZZAZ   AAZAZ-Z
13: AA-ZAZZ   AAZA-ZZ
14: AAAZ-ZZ   AA-AZZZ
15: AAA-ZZZ
```

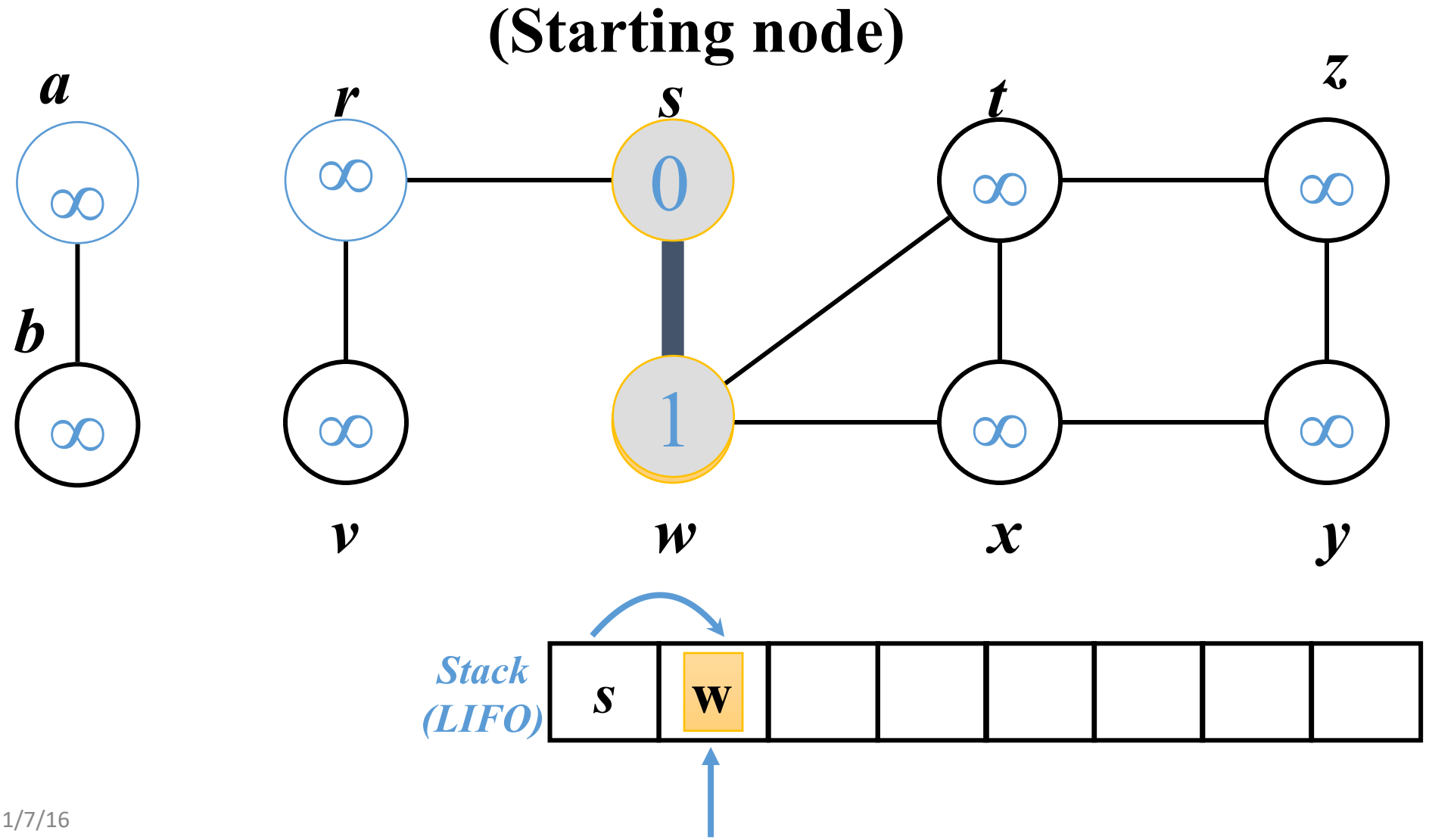
How Google's AlphaGo Beat a Go World Champion (BFS or DFS?)

Depth-First Search: Example

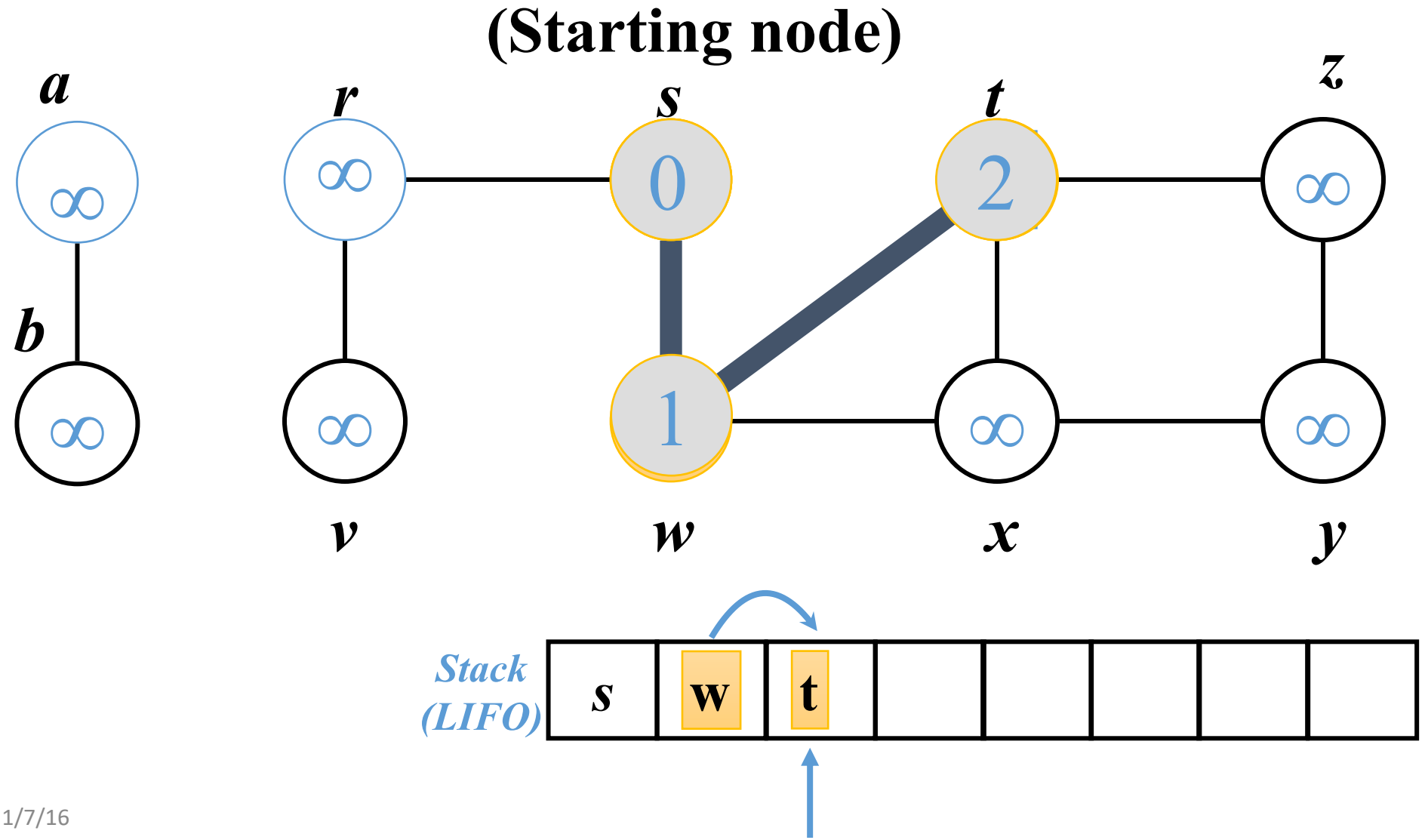




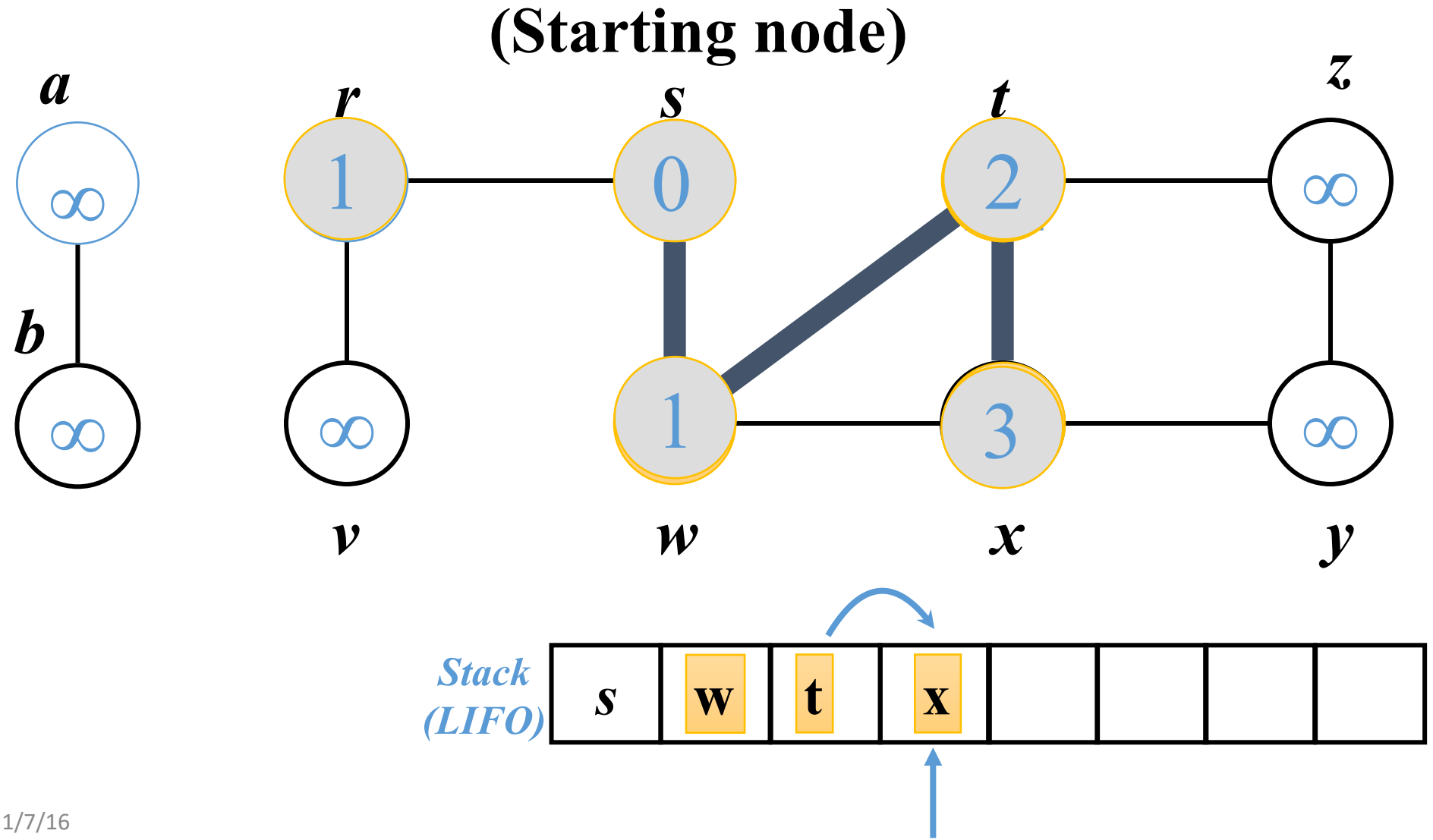
Depth-First Search: Example



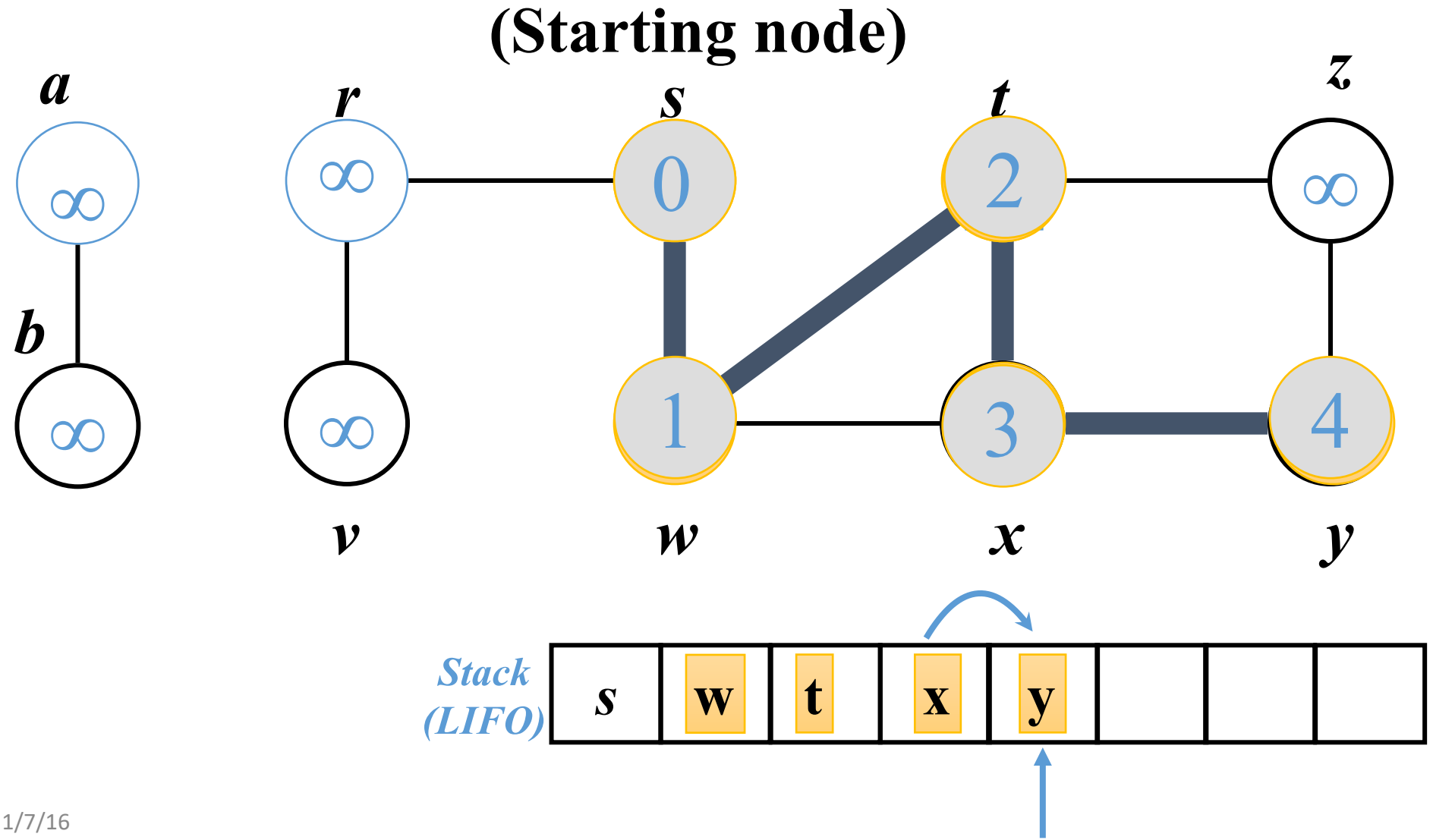
Depth-First Search: Example



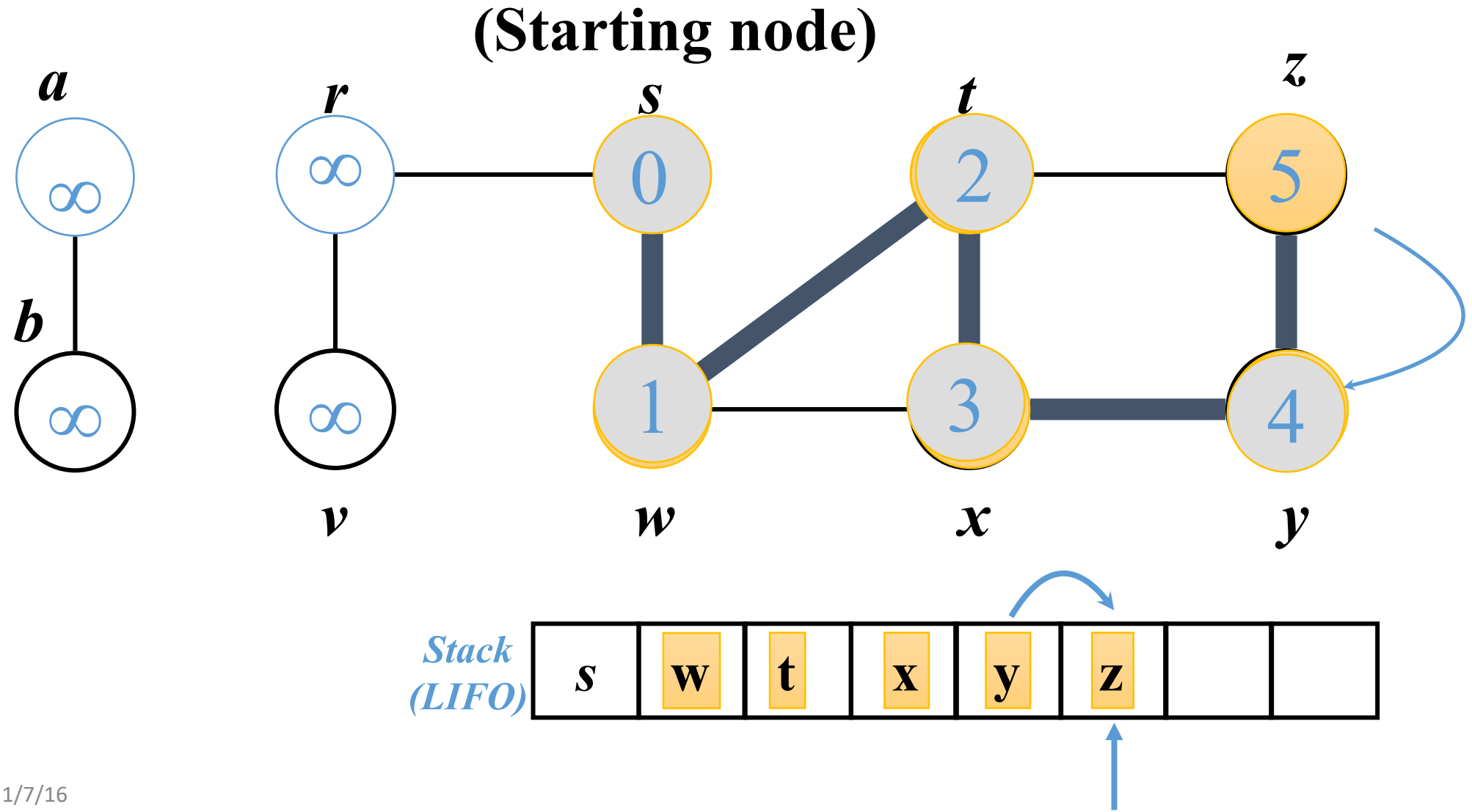
Depth-First Search: Example



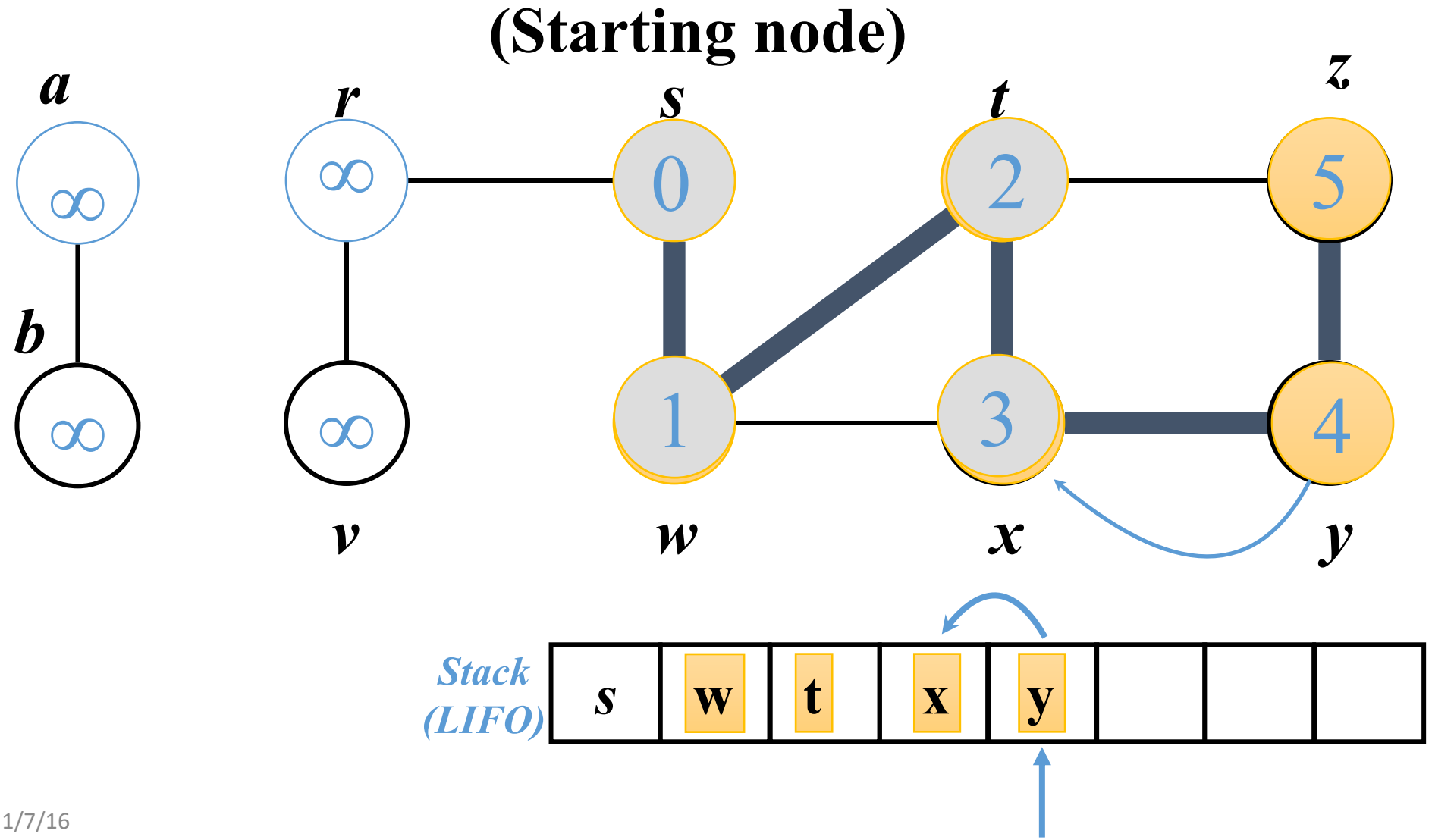
Depth-First Search: Example



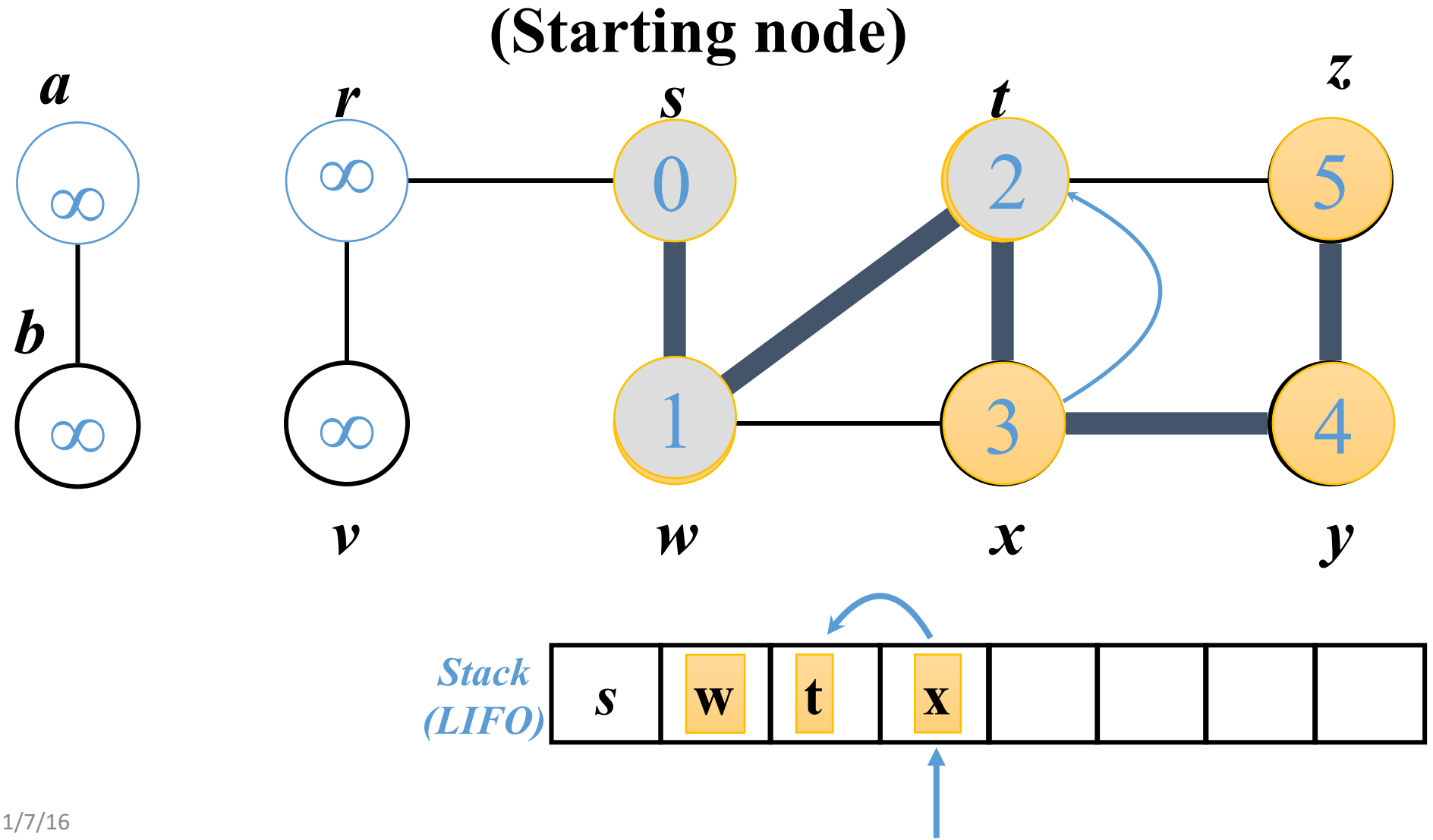
Depth-First Search: Example



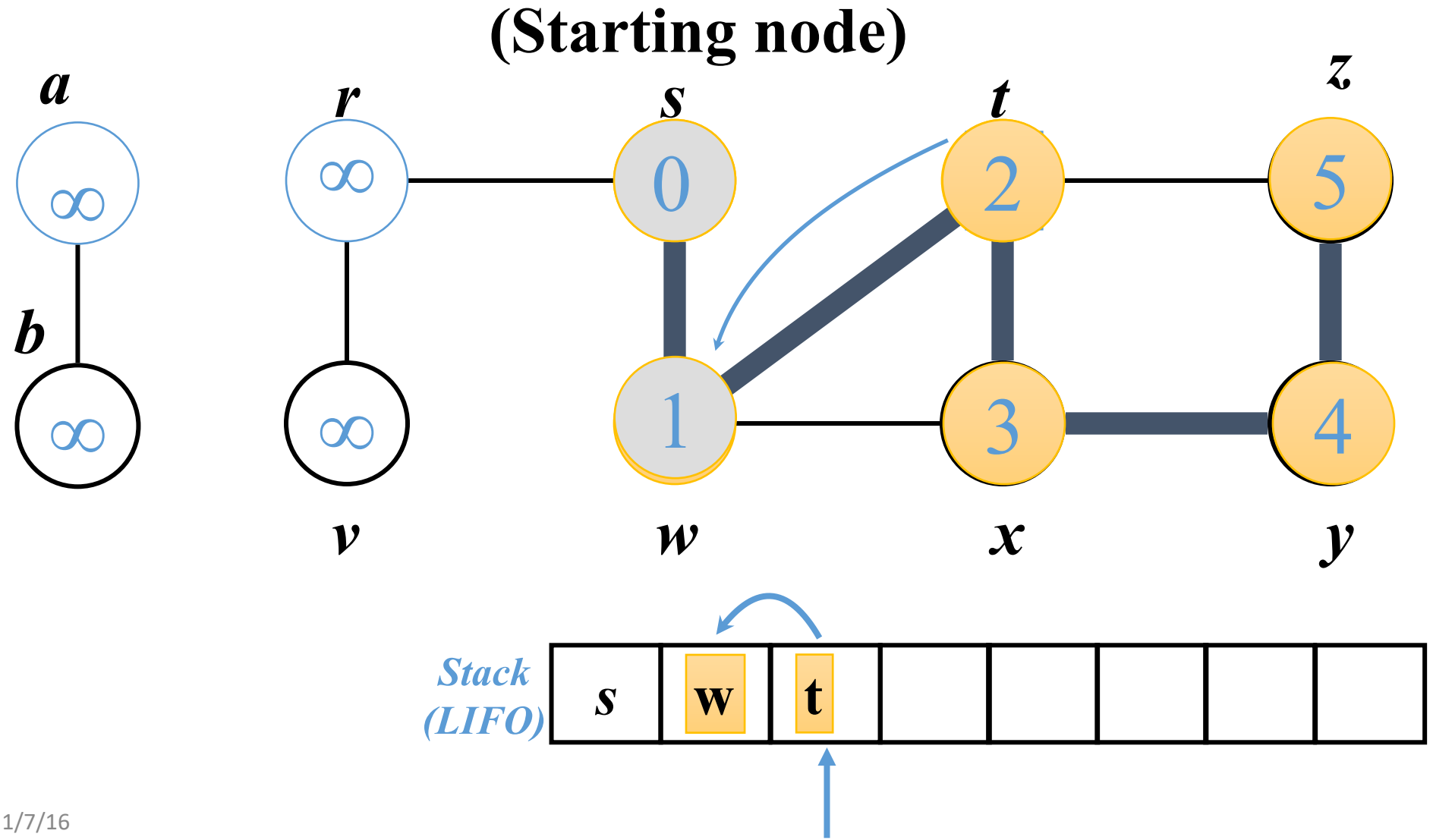
Depth-First Search: Example



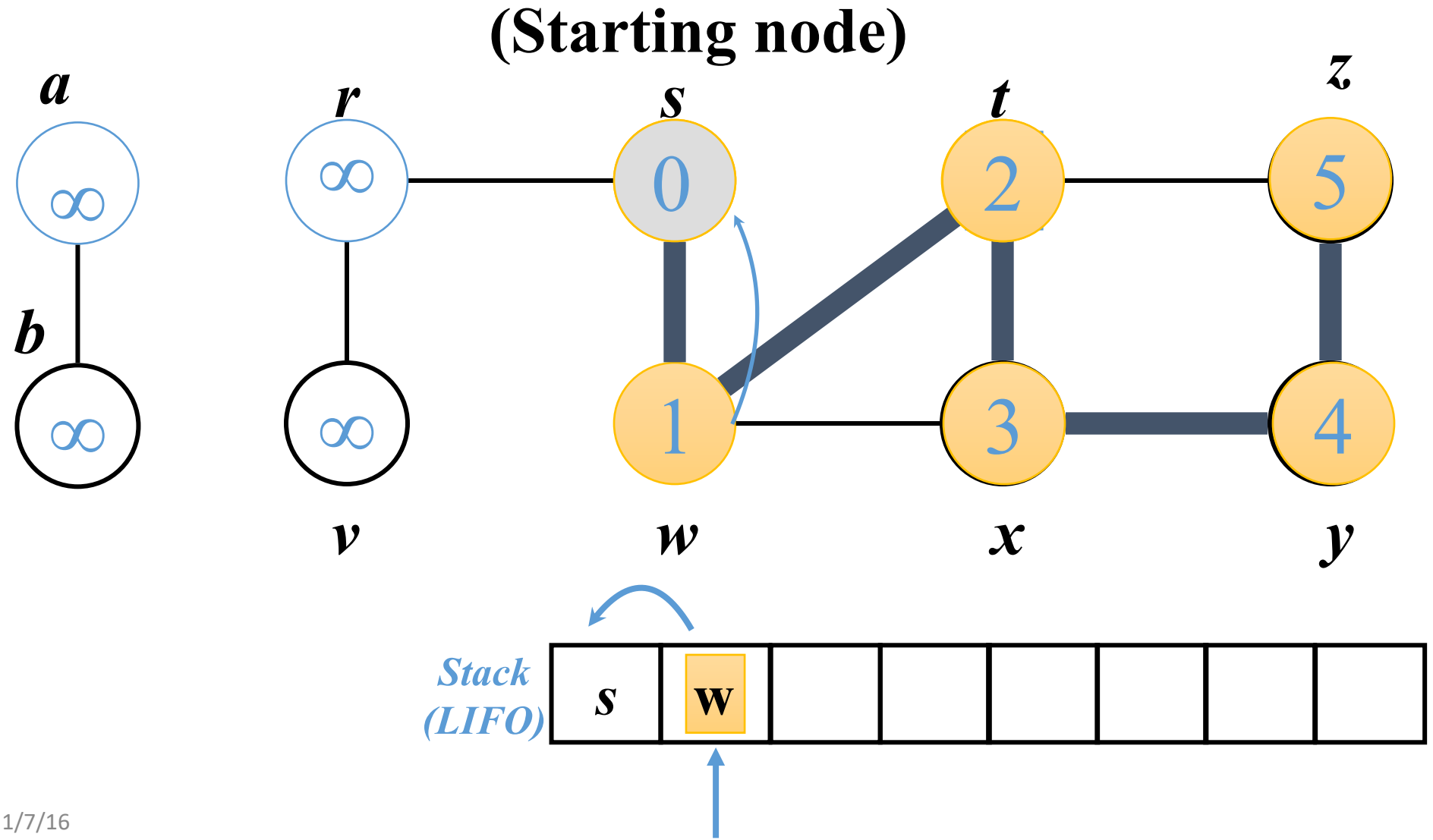
Depth-First Search: Example



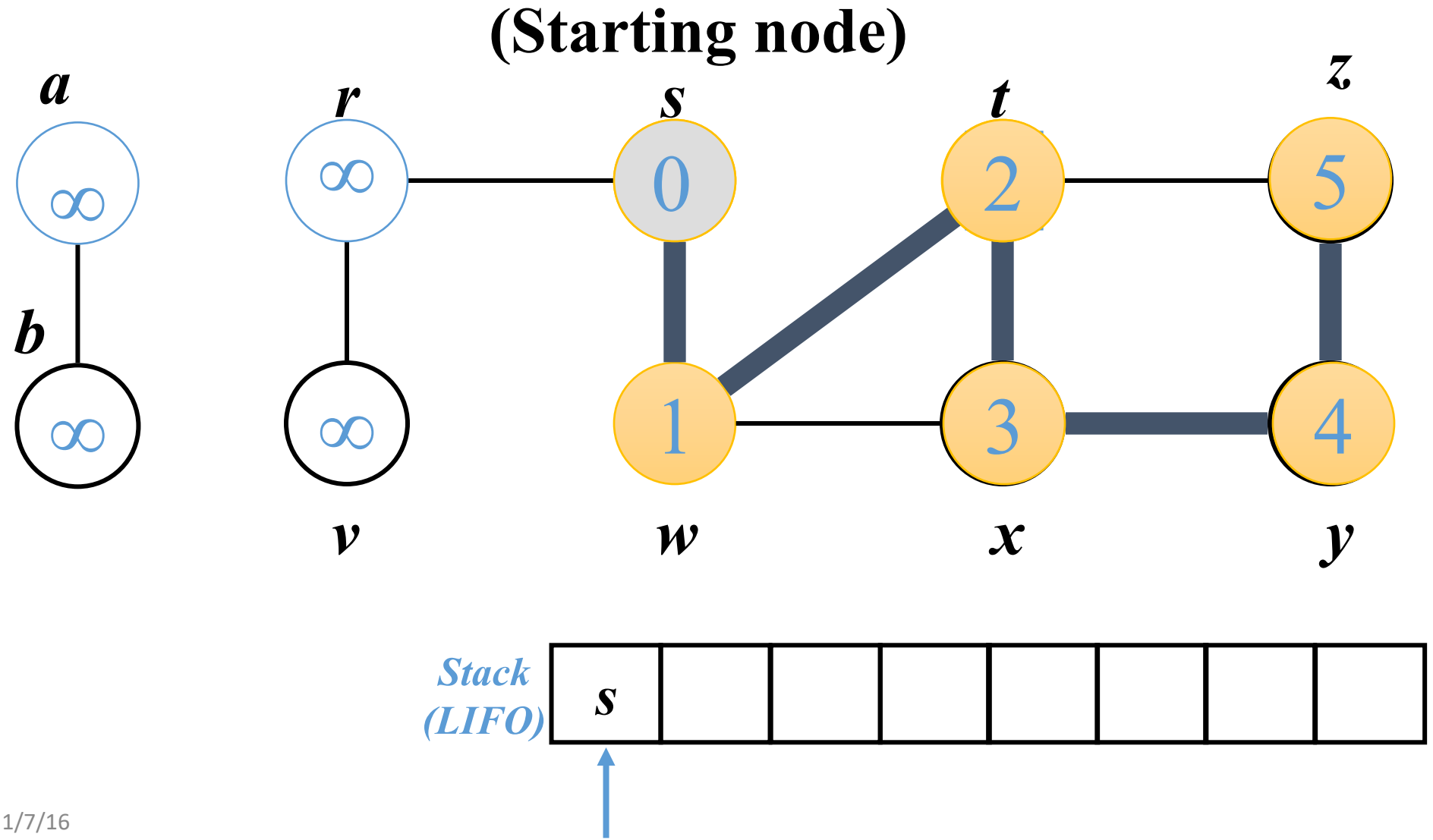
Depth-First Search: Example



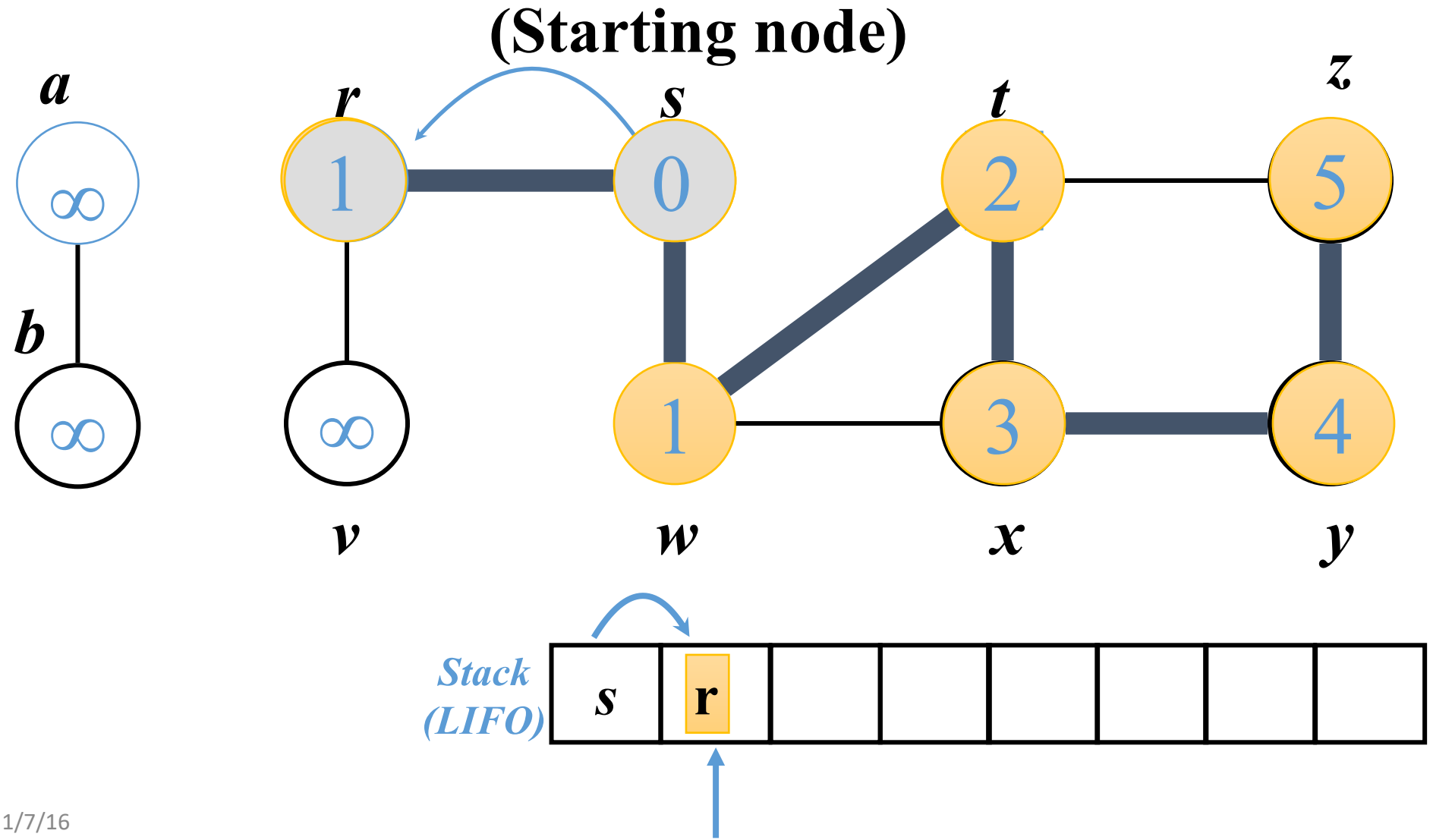
Depth-First Search: Example



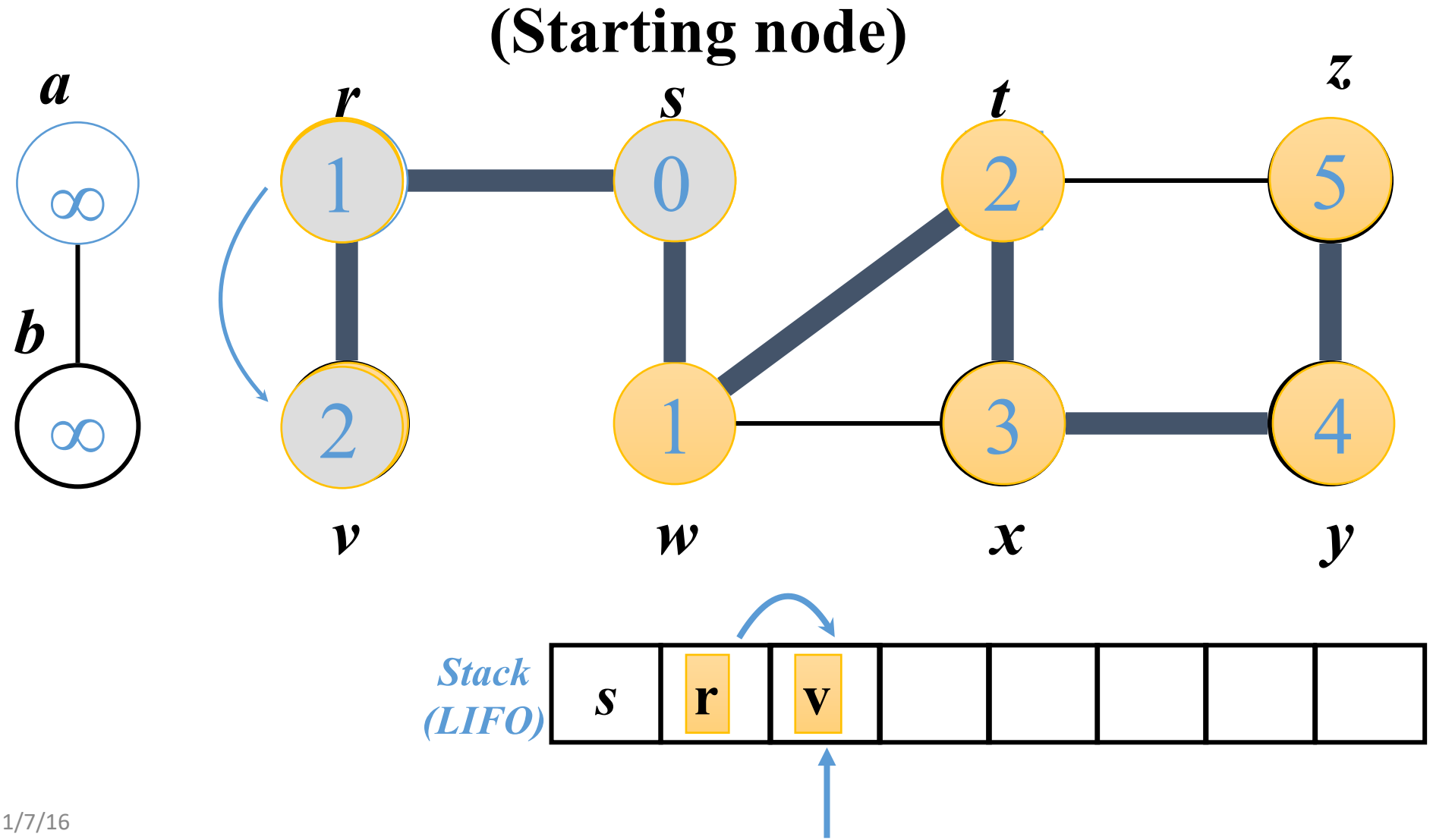
Depth-First Search: Example



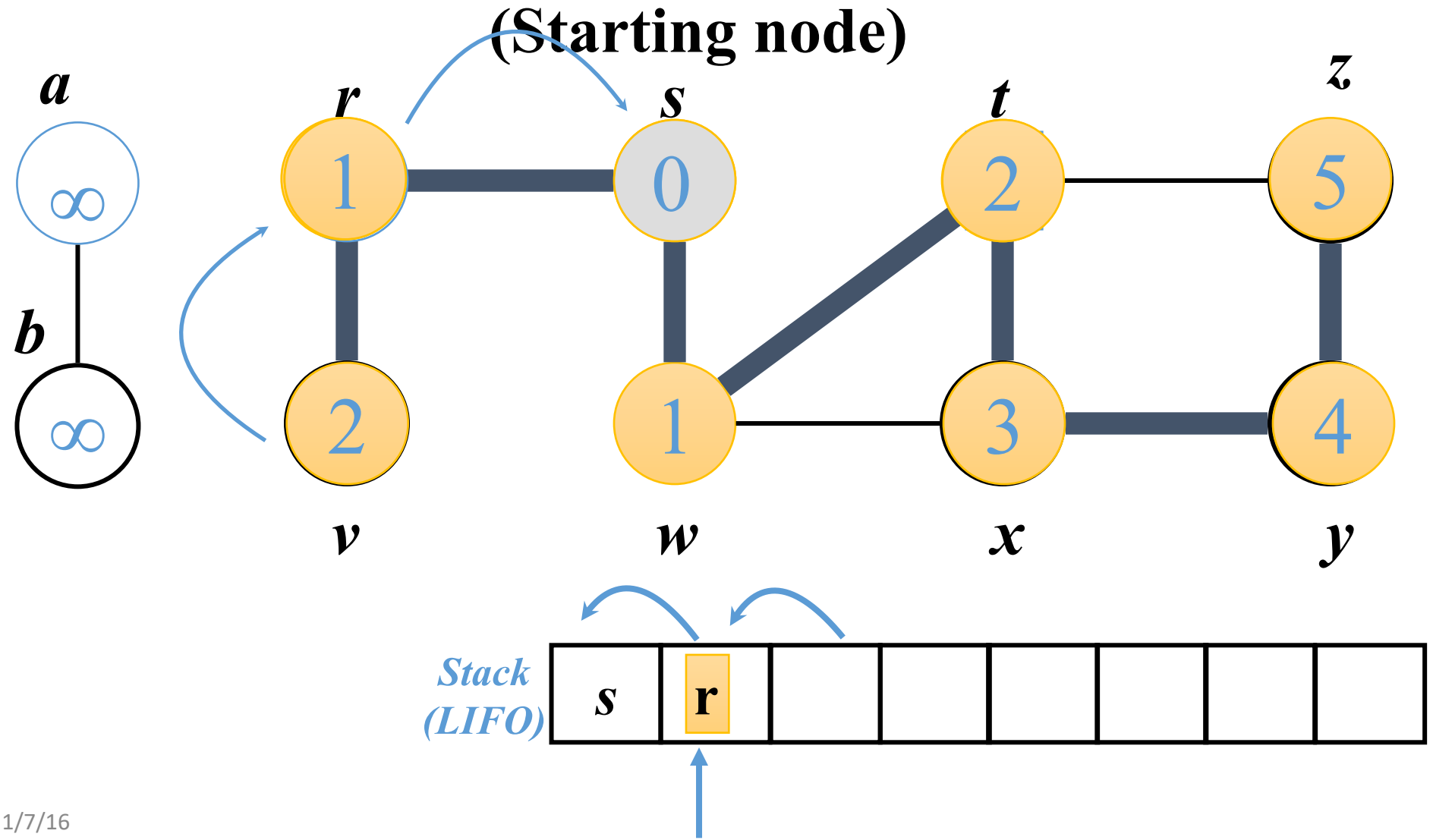
Depth-First Search: Example



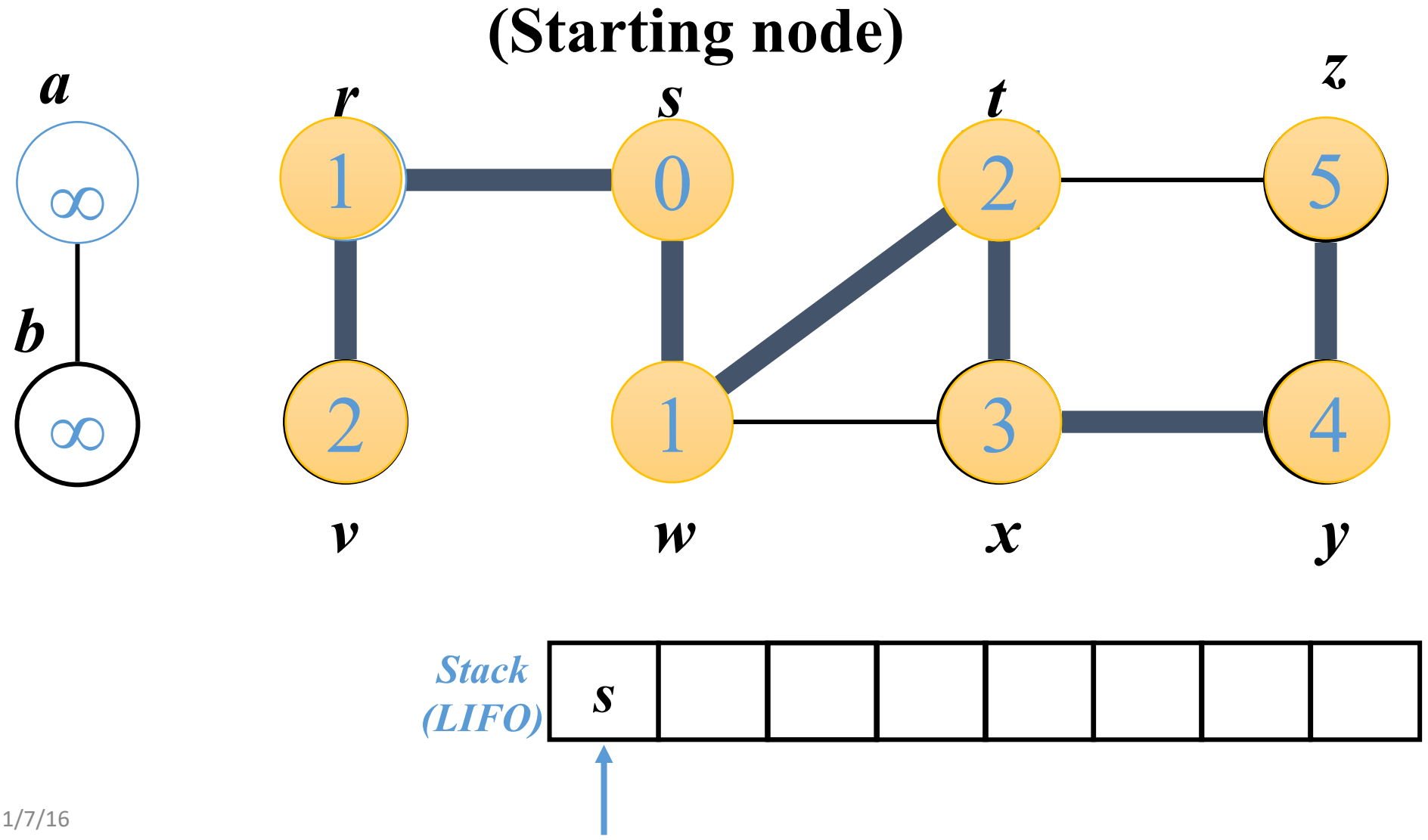
Depth-First Search: Example



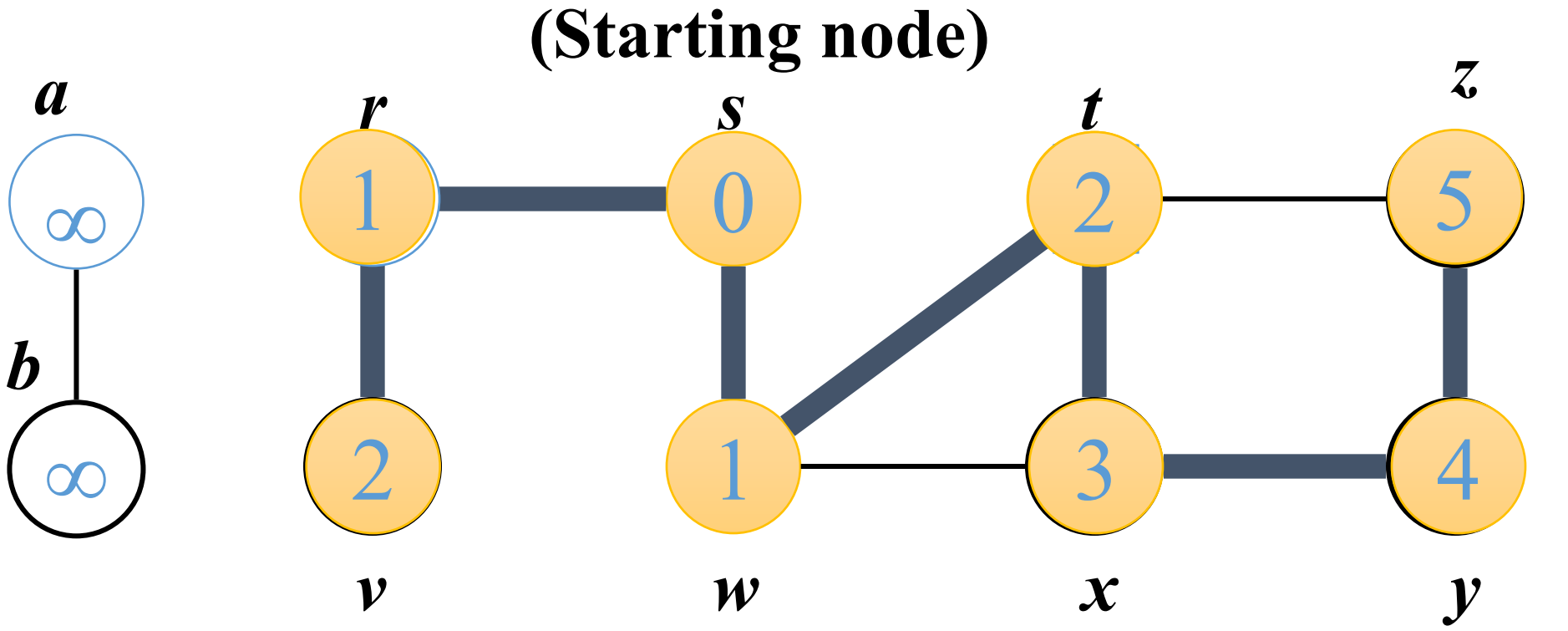
Depth-First Search: Example



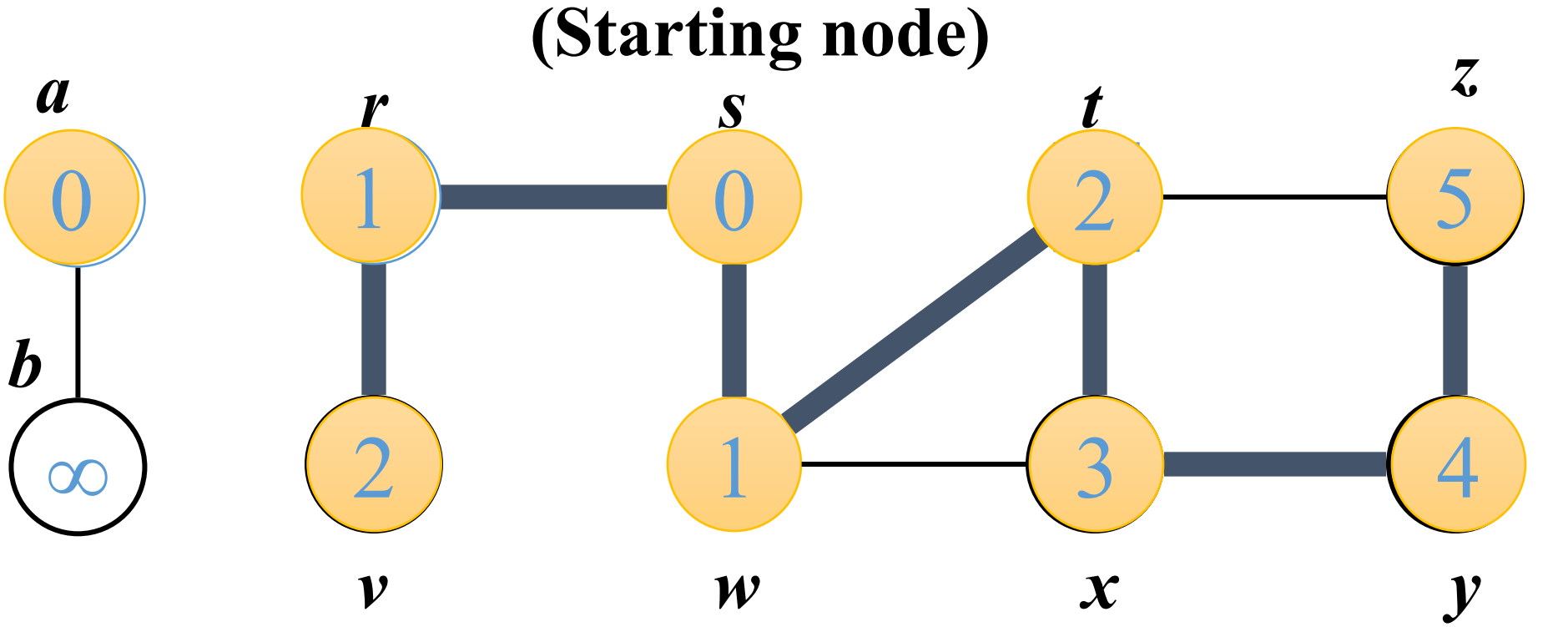
Depth-First Search: Example

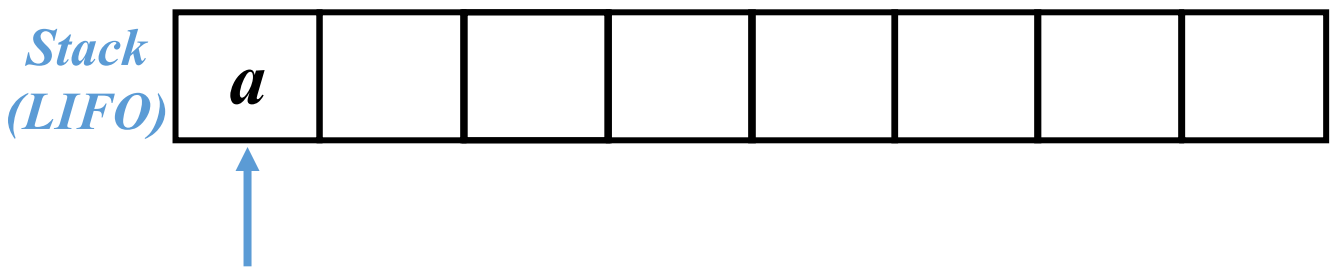


Depth-First Search: Example

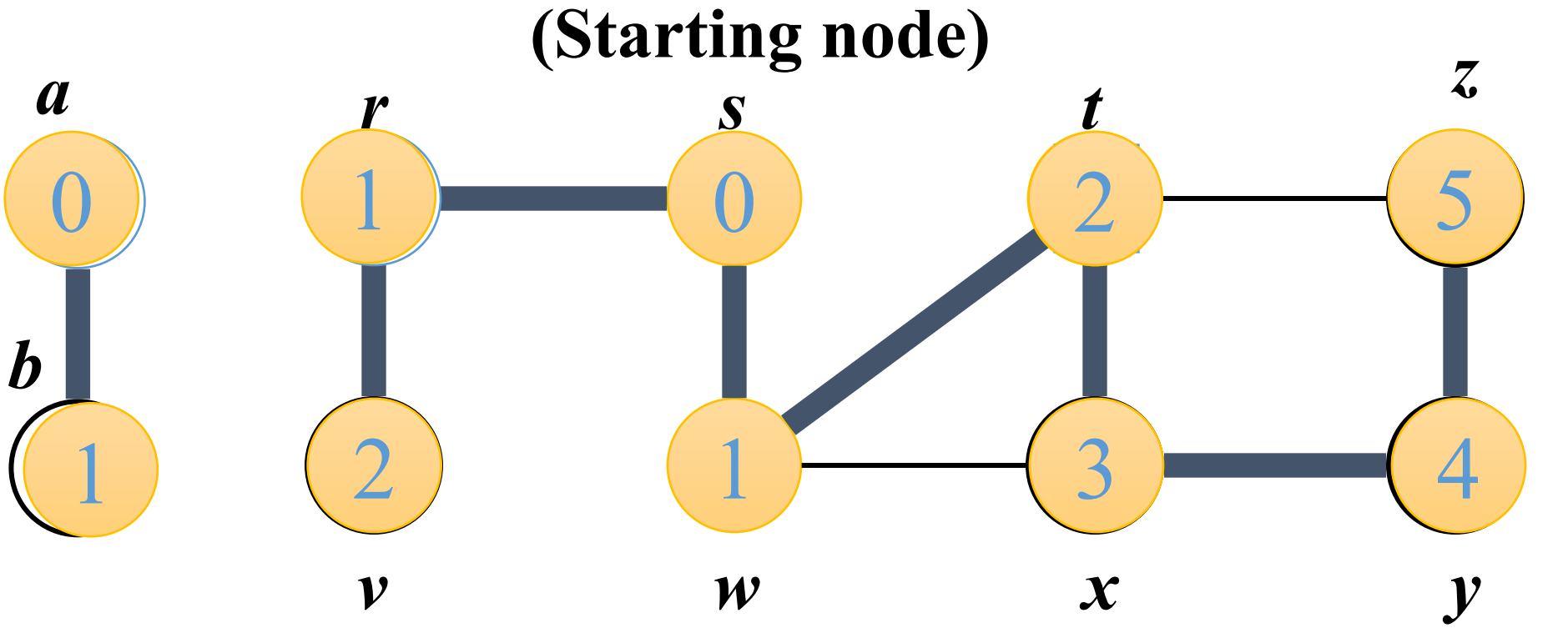


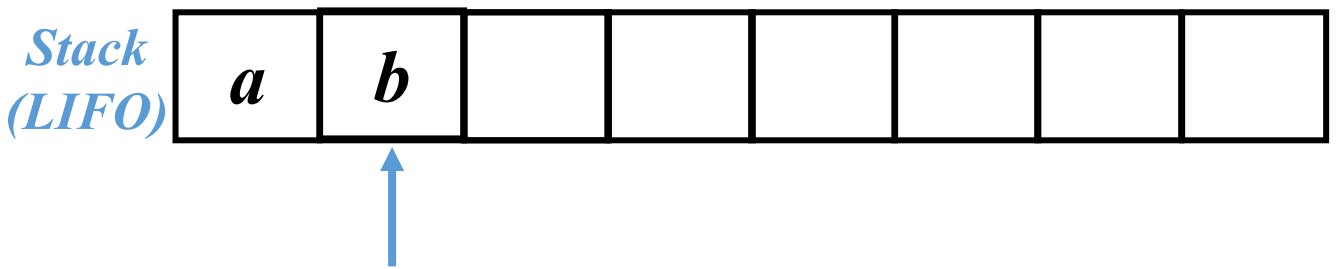
Depth-First Search: Example



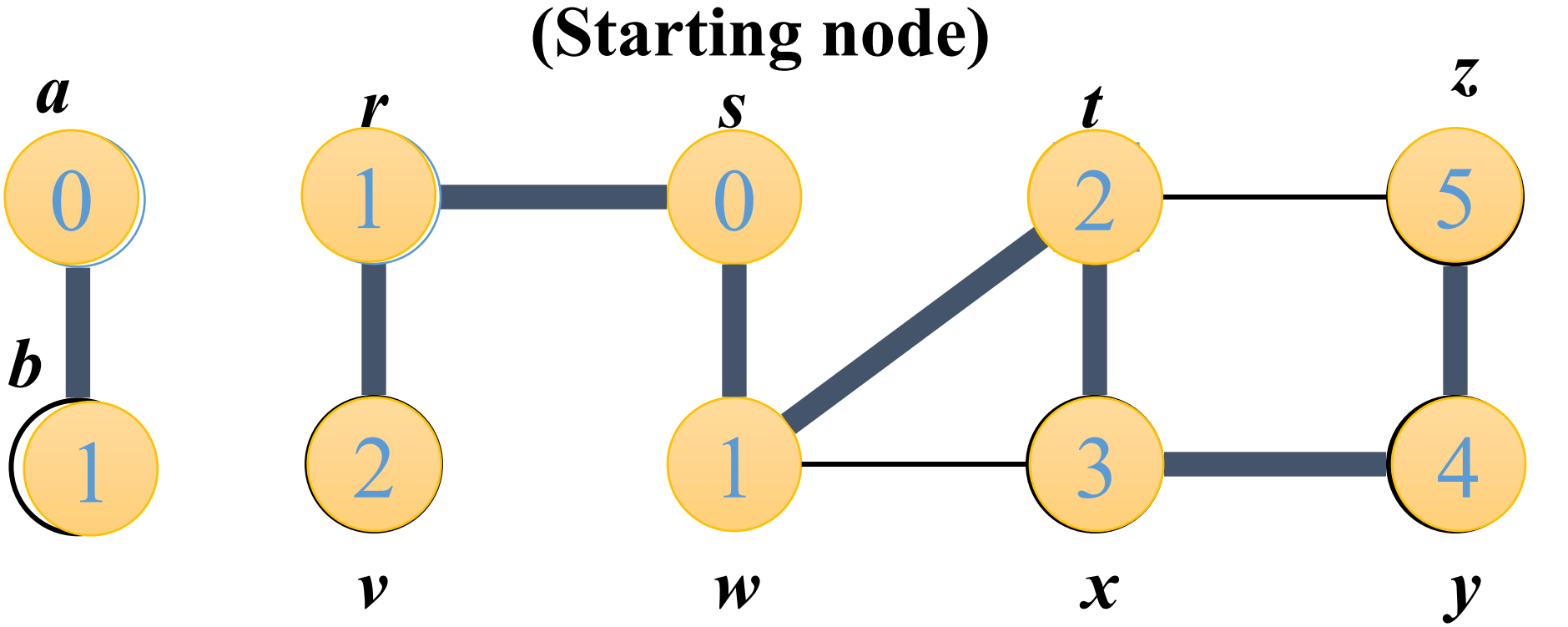


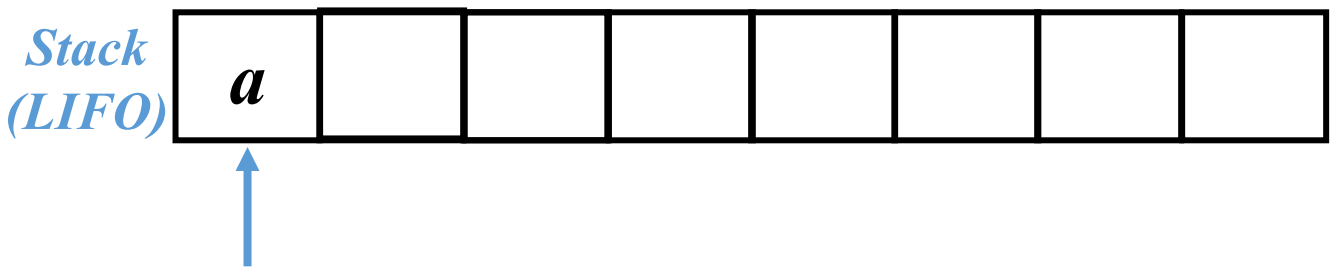
Depth-First Search: Example



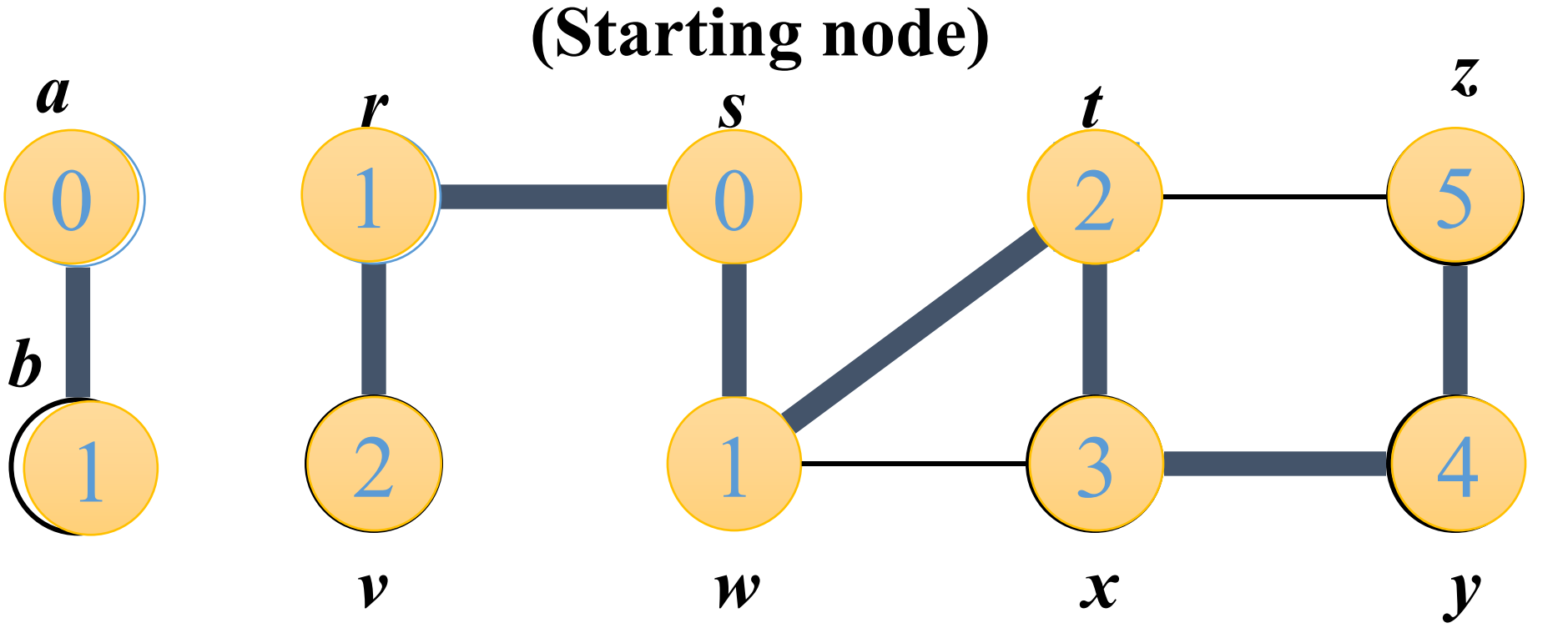


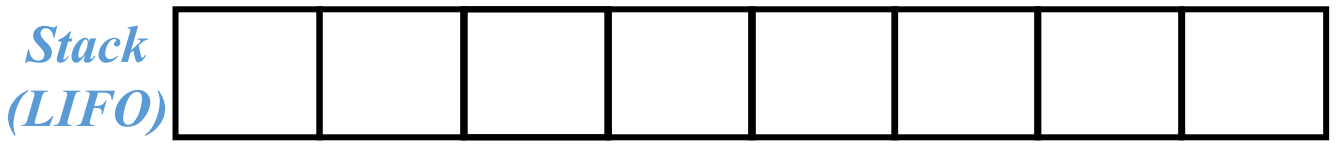
Depth-First Search: Example





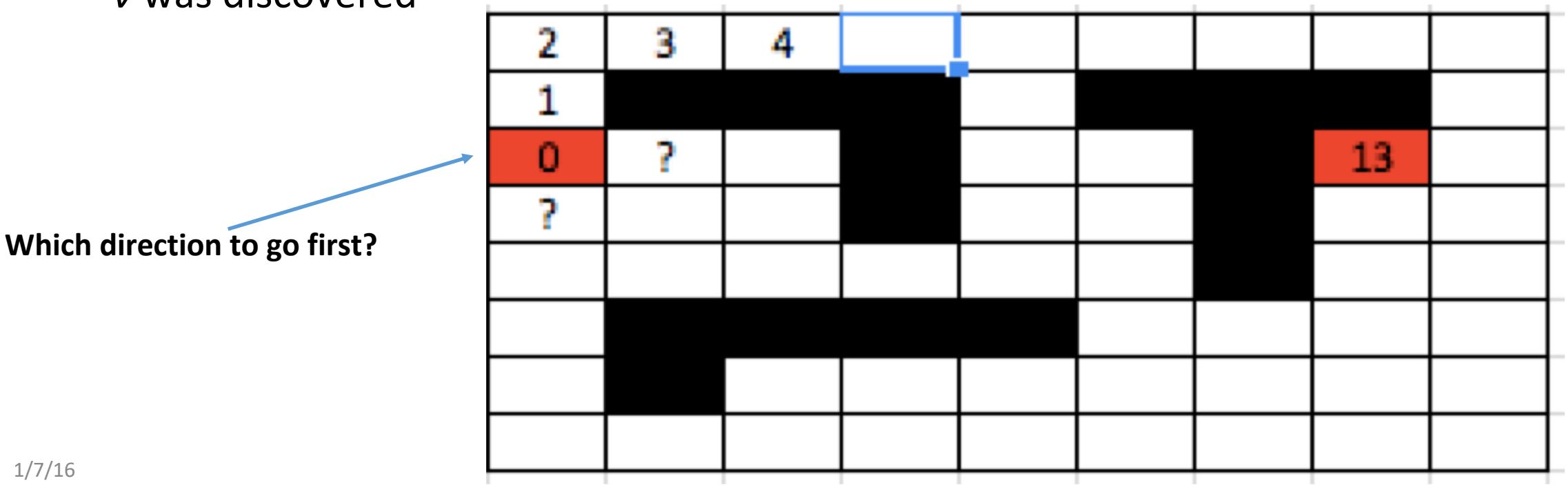
Depth-First Search: Example





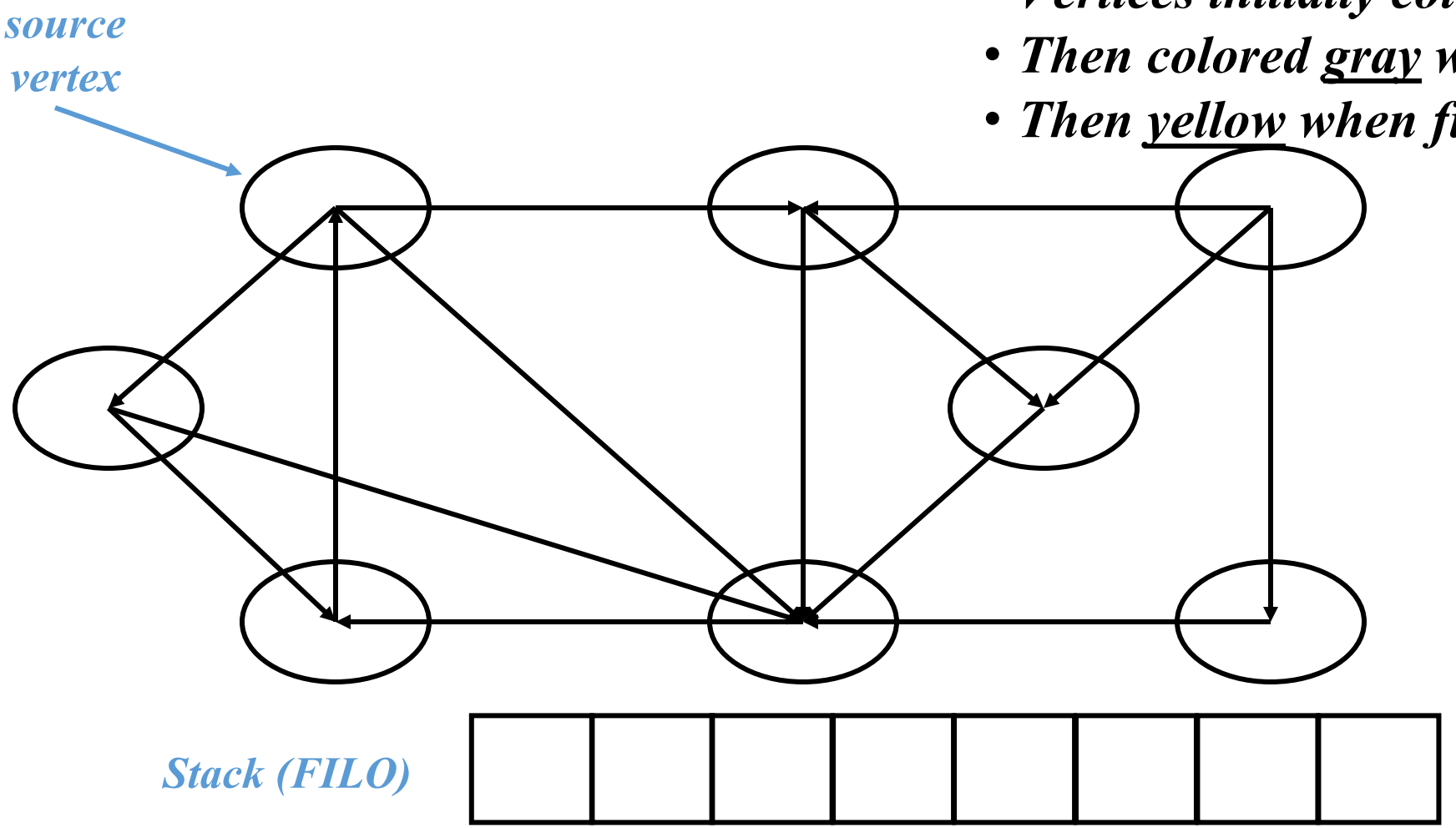
Depth-First Search

- *Depth-first search* is another strategy for exploring a graph
  - Explore "**deeper**" in the graph whenever possible
  - Edges are explored out of the most recently discovered vertex *v* that still has unexplored edges
  - When all of *v*'s edges have been explored, backtrack to the vertex from which *v* was discovered

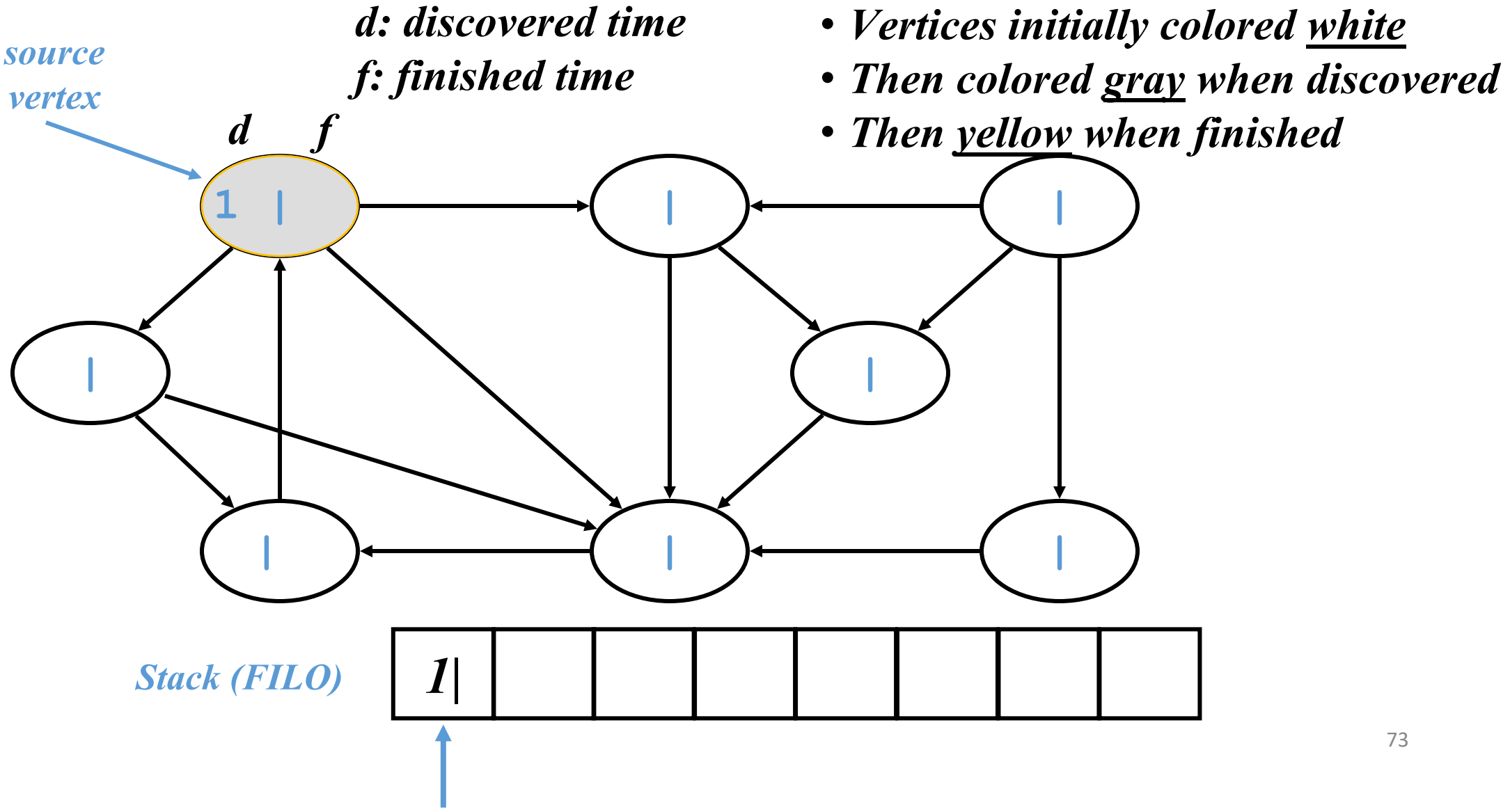


DFS Example

- ***Vertices initially colored white***
- ***Then colored gray when discovered***
- ***Then yellow when finished***

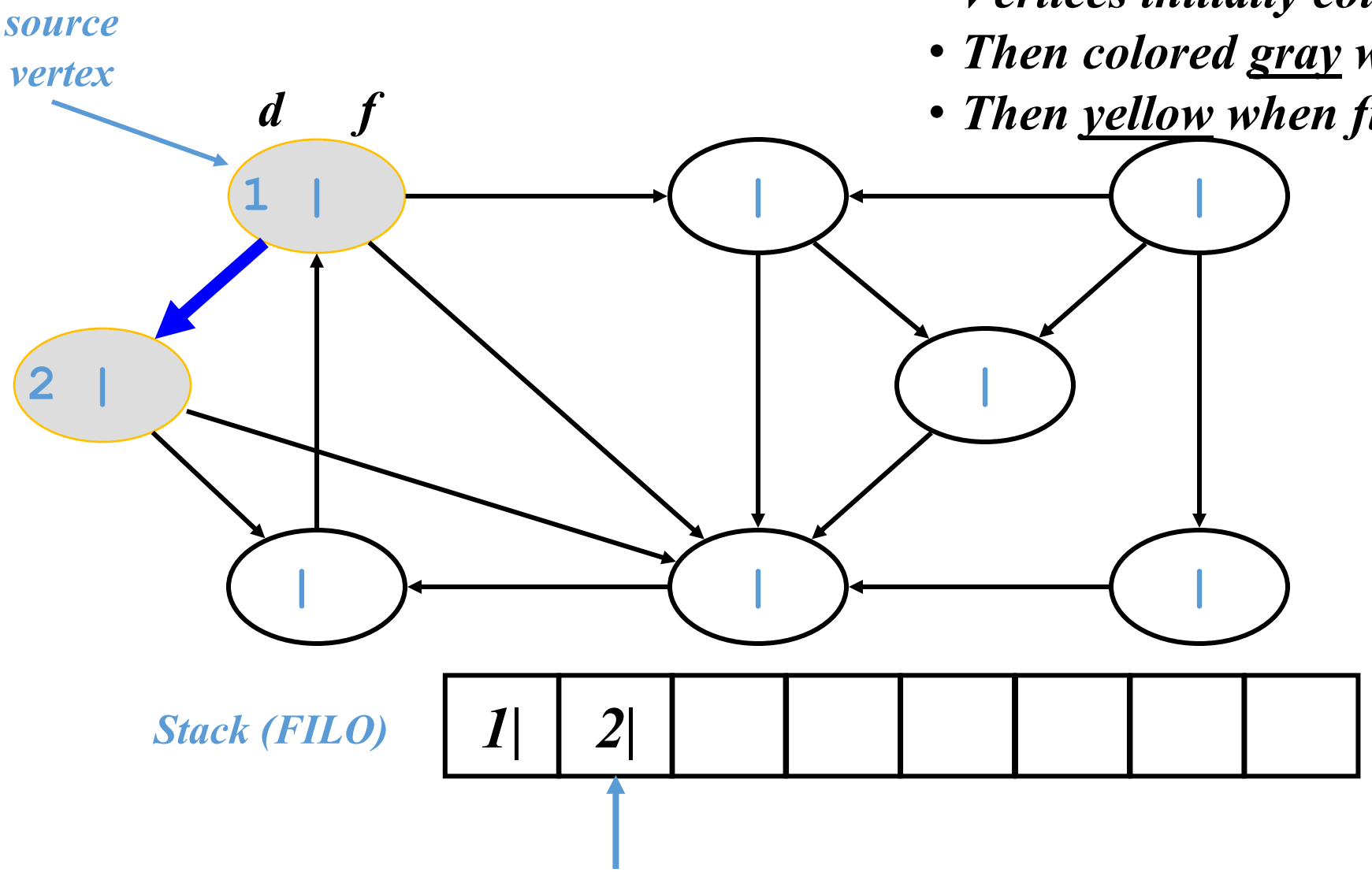


DFS Example


d: discovered time
f: finished time
• Vertices initially colored white
• Then colored gray when discovered
• Then yellow when finished
source vertex
d
f
1
Stack (FILO)
1

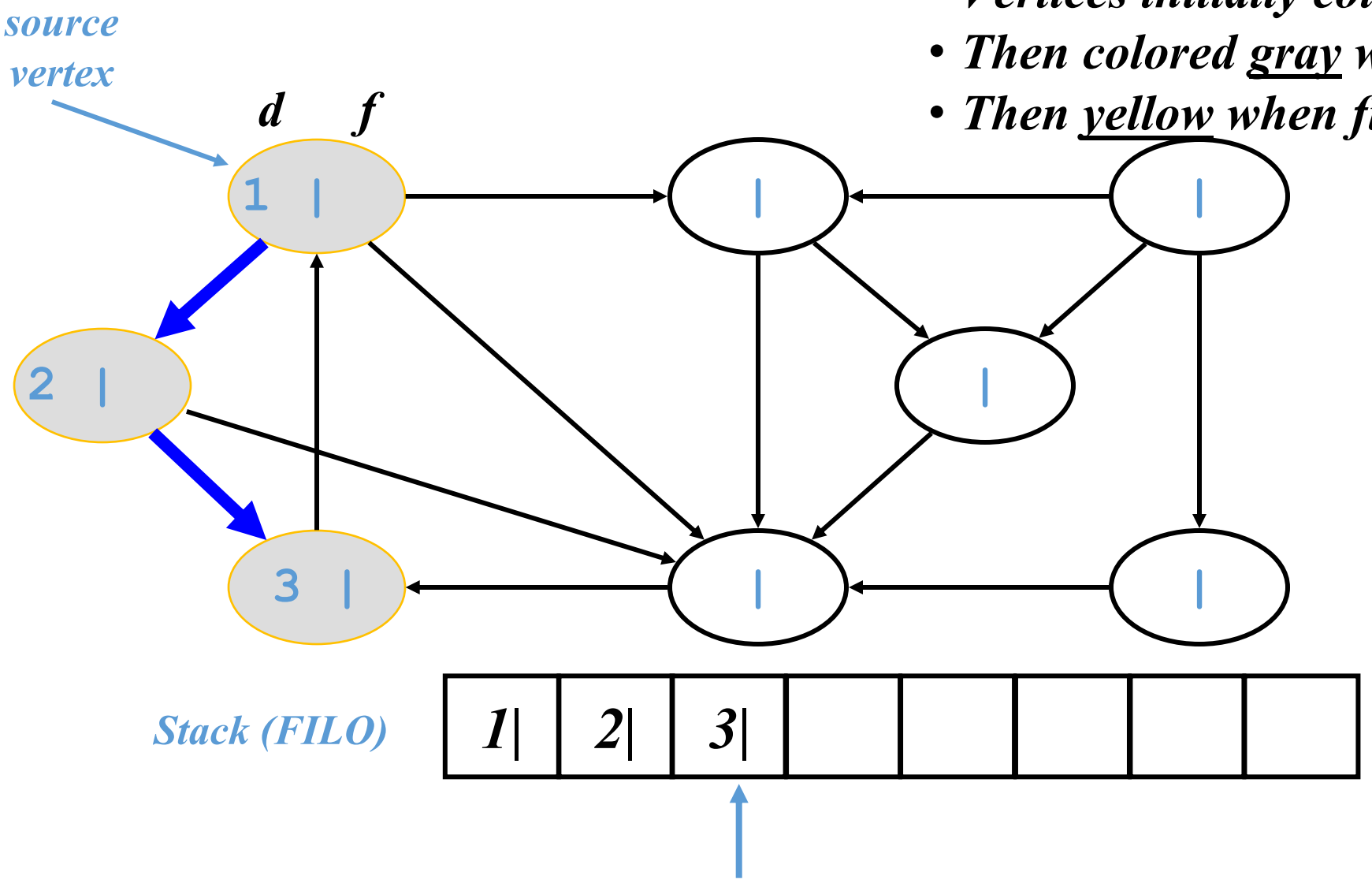
DFS Example

- ***Vertices initially colored white***
- ***Then colored gray when discovered***
- ***Then yellow when finished***



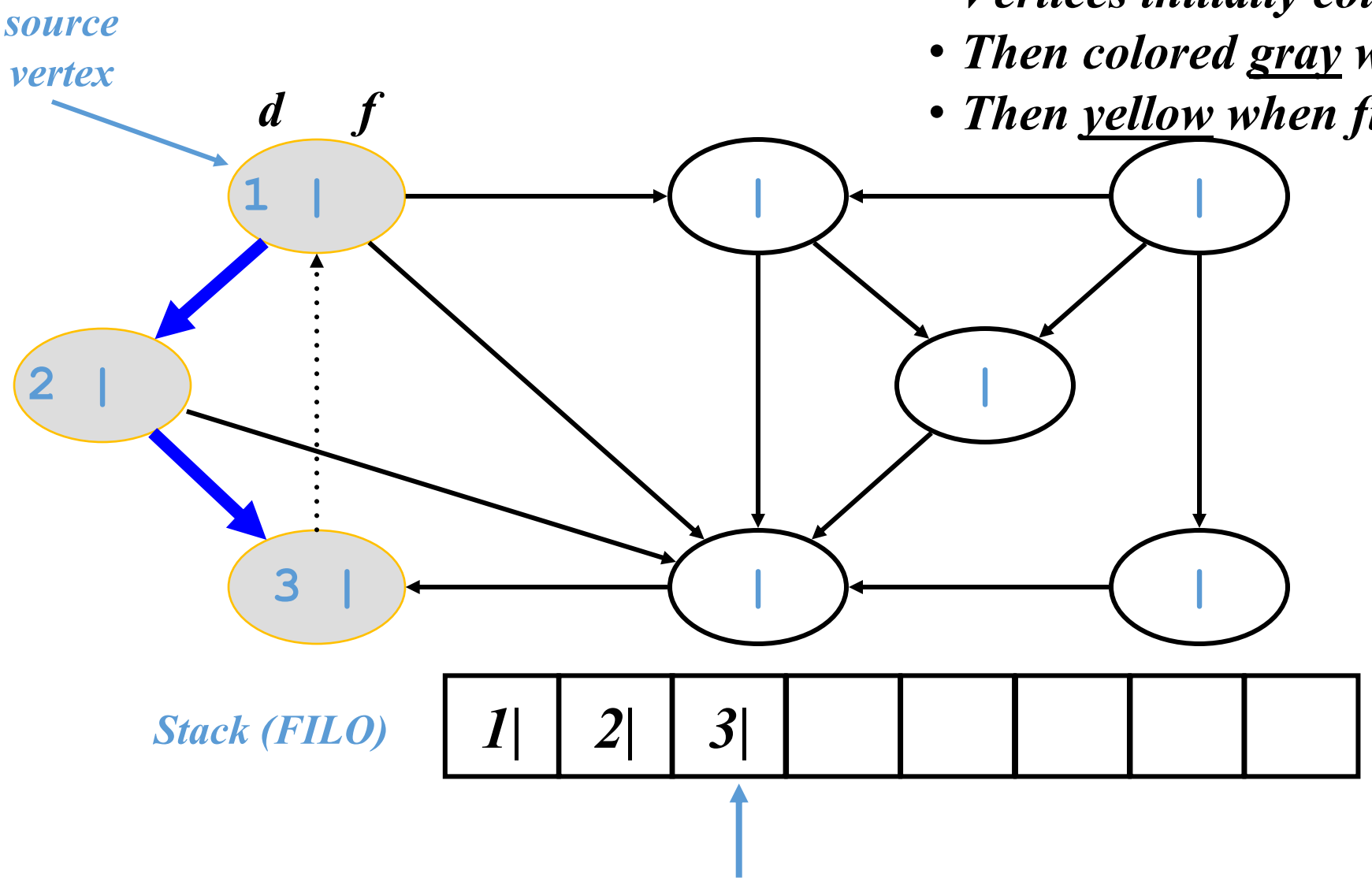
DFS Example

- *Vertices initially colored white*
- *Then colored gray when discovered*
- *Then yellow when finished*



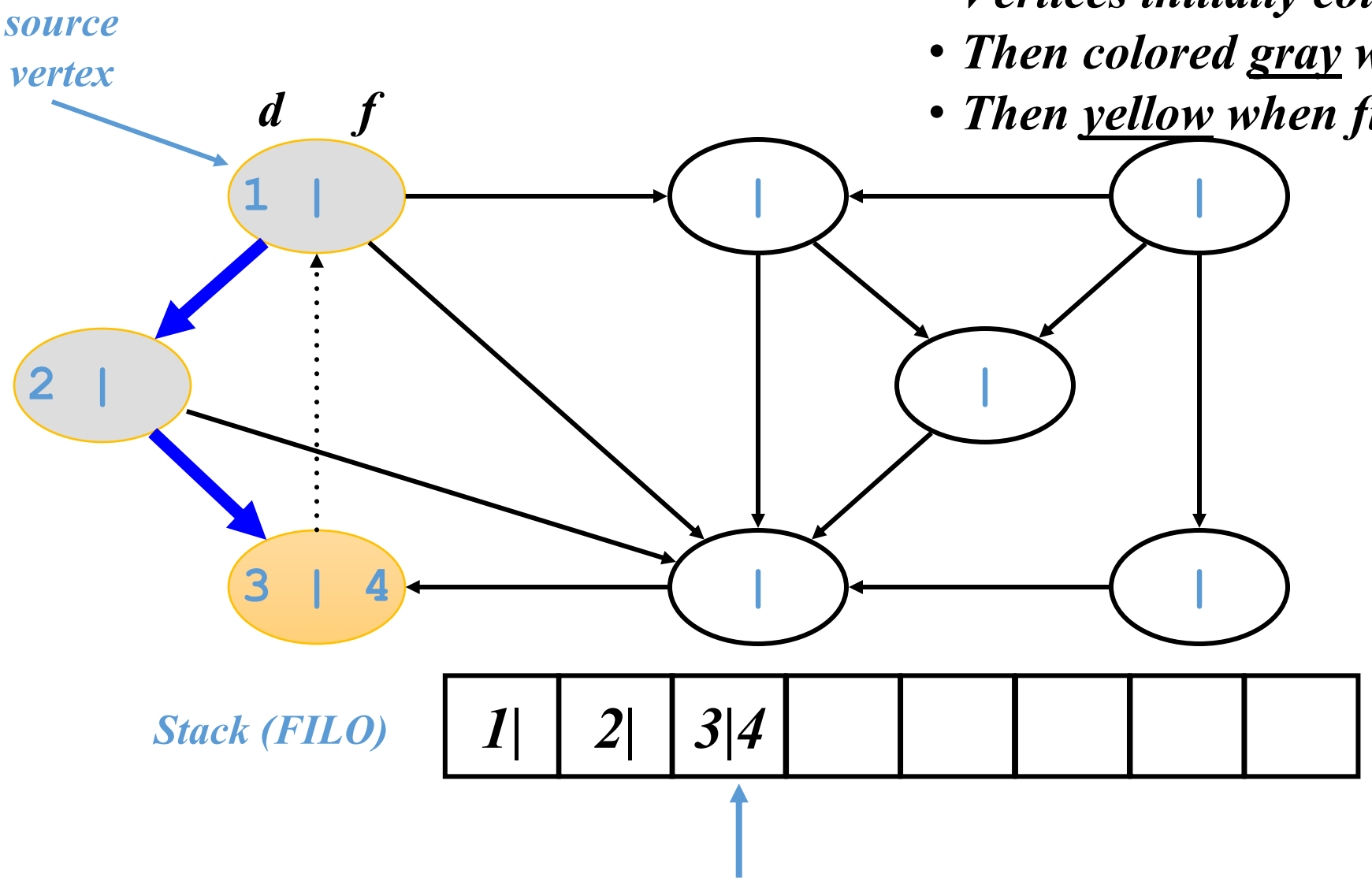
DFS Example

- *Vertices initially colored white*
- *Then colored gray when discovered*
- *Then yellow when finished*



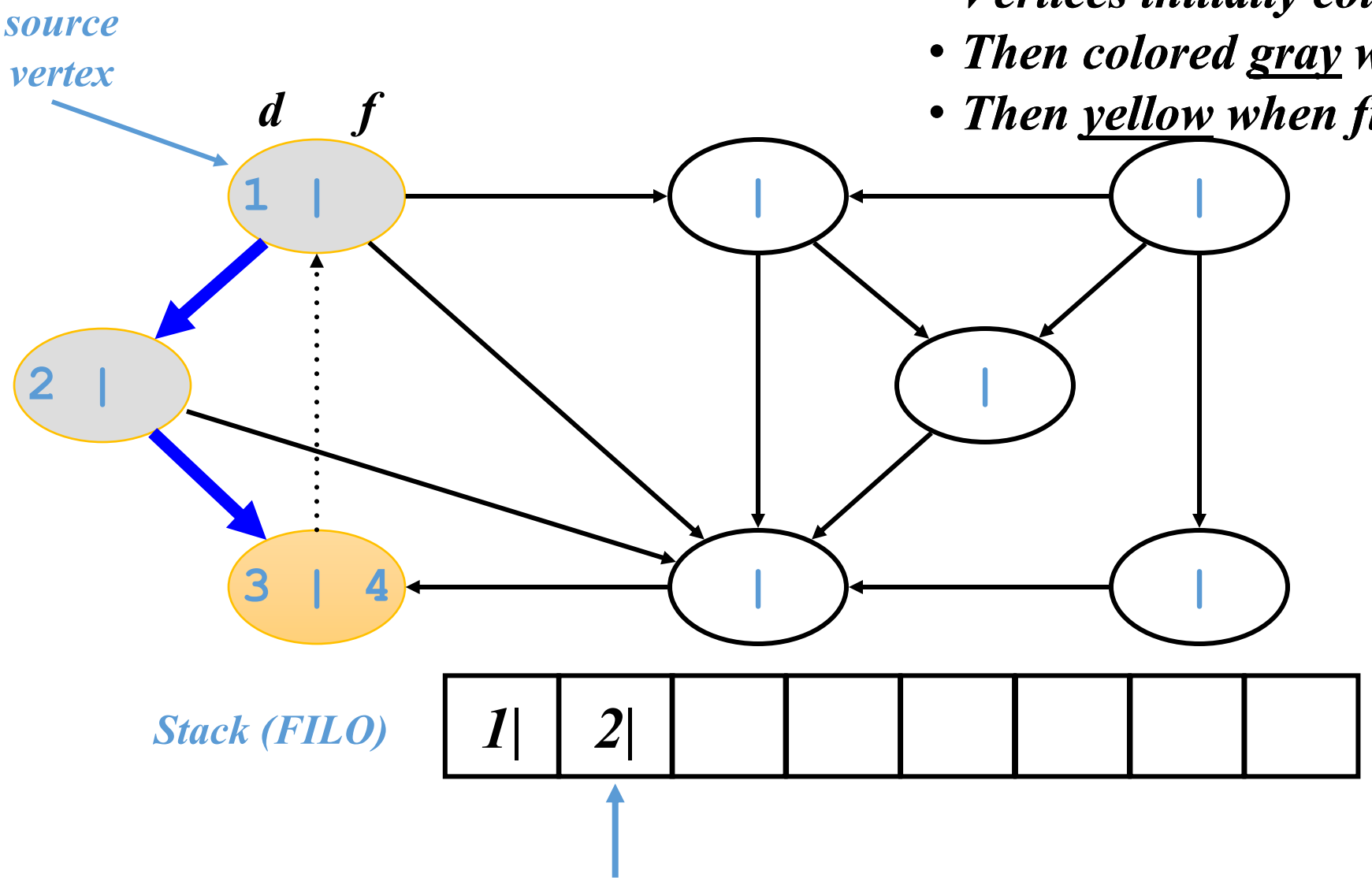
DFS Example

- *Vertices initially colored white*
- *Then colored gray when discovered*
- *Then yellow when finished*



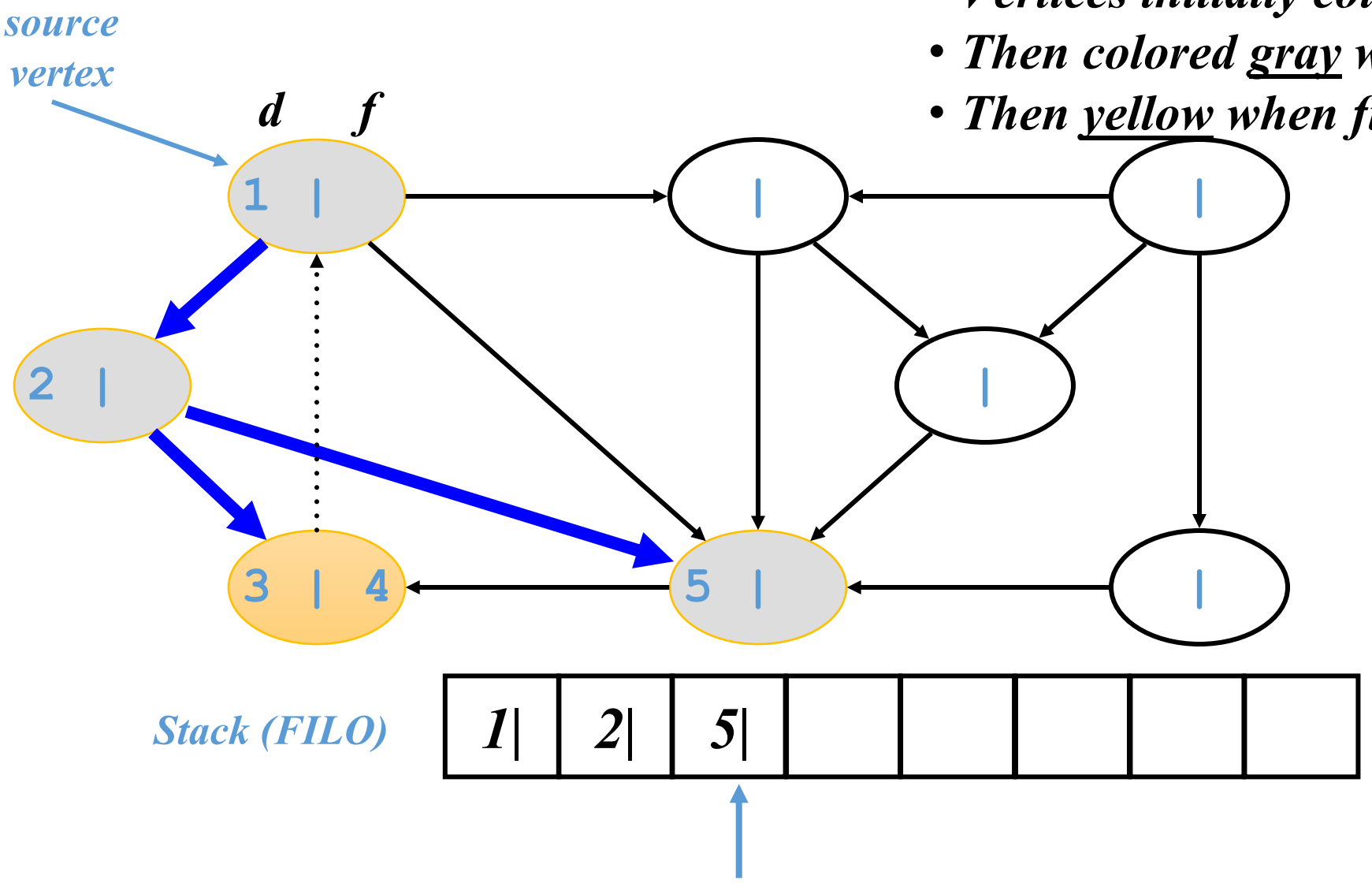
DFS Example

- *Vertices initially colored white*
- *Then colored gray when discovered*
- *Then yellow when finished*



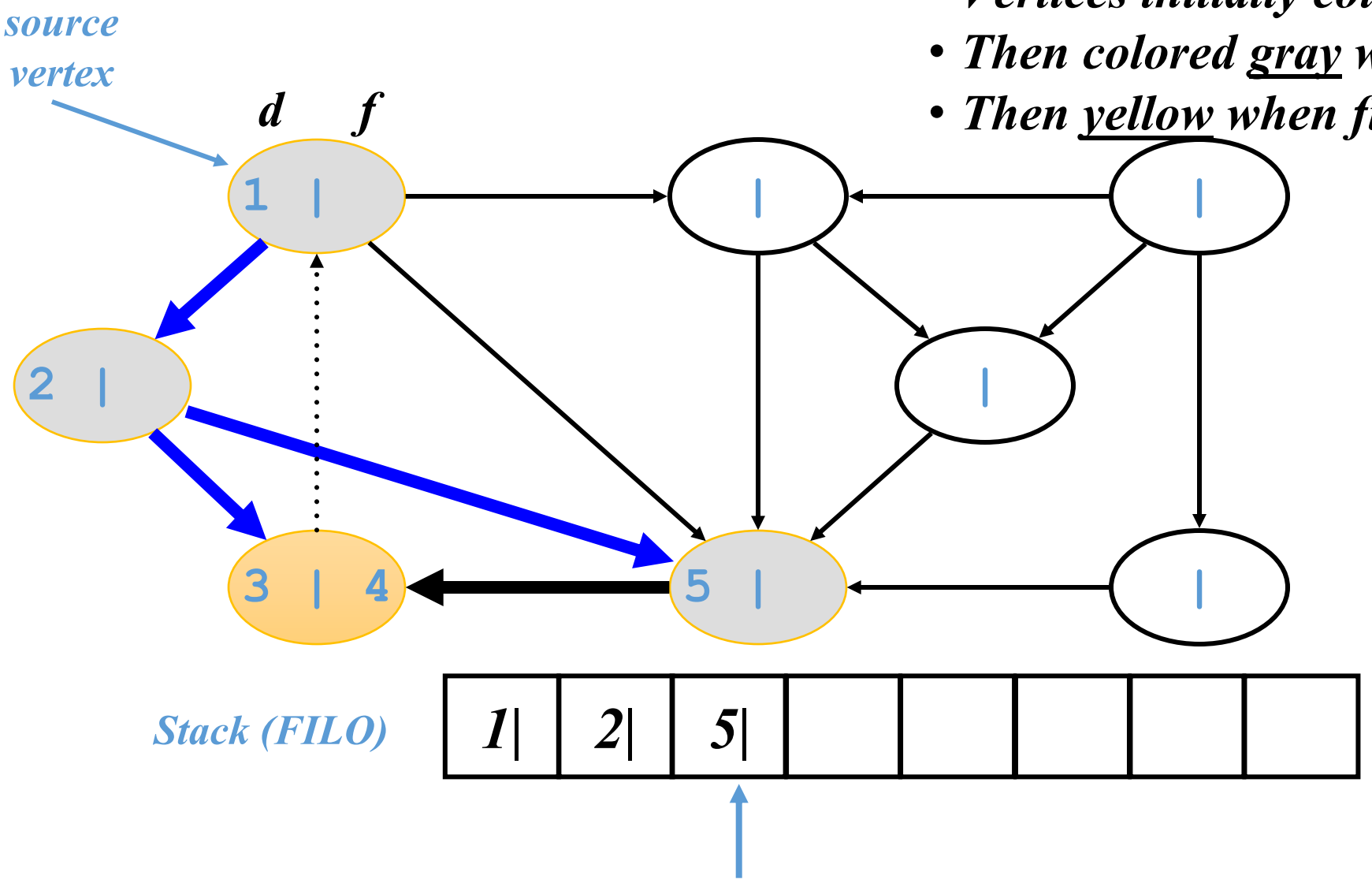
DFS Example

- *Vertices initially colored white*
- *Then colored gray when discovered*
- *Then yellow when finished*



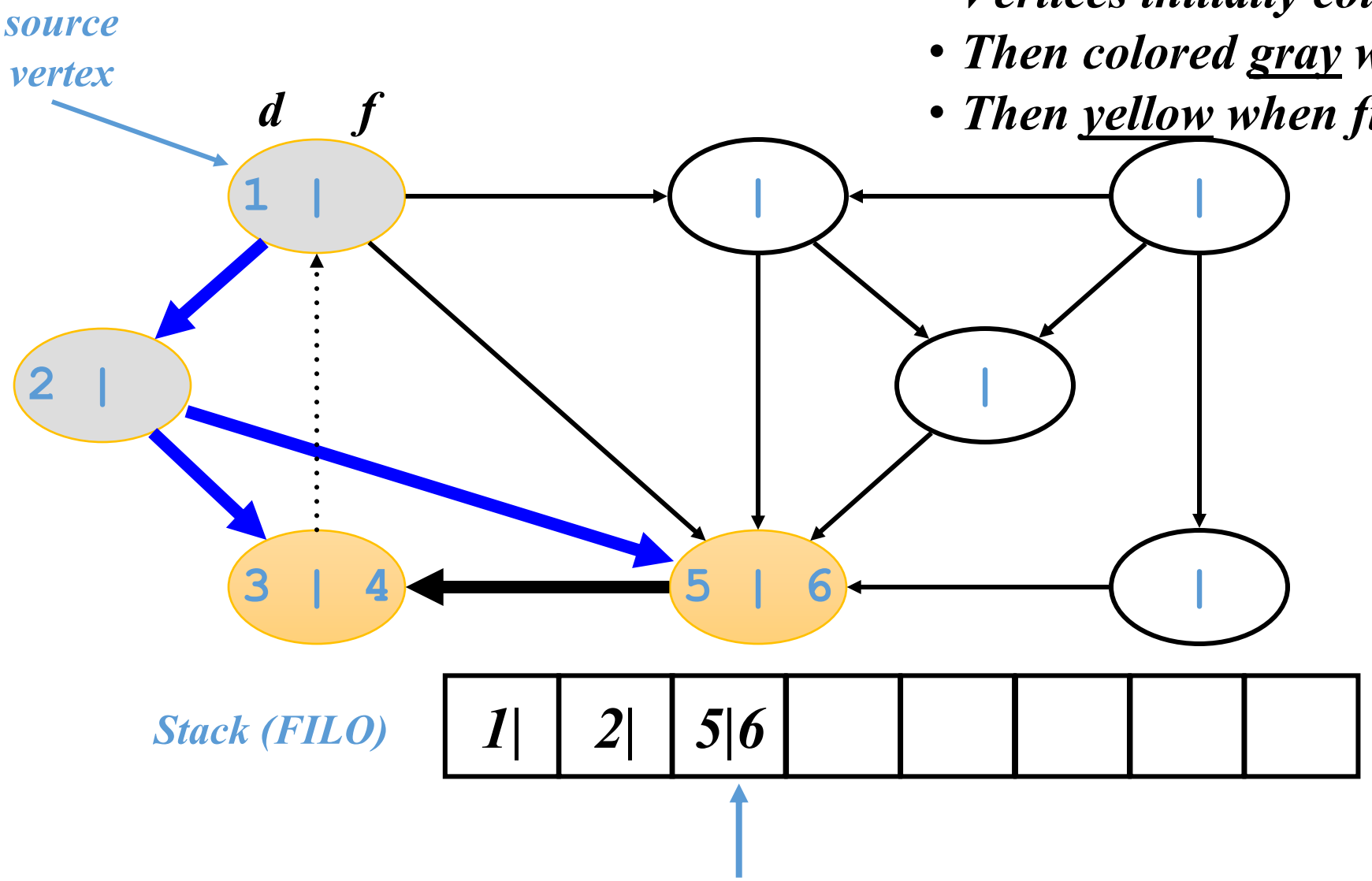
DFS Example

- *Vertices initially colored white*
- *Then colored gray when discovered*
- *Then yellow when finished*



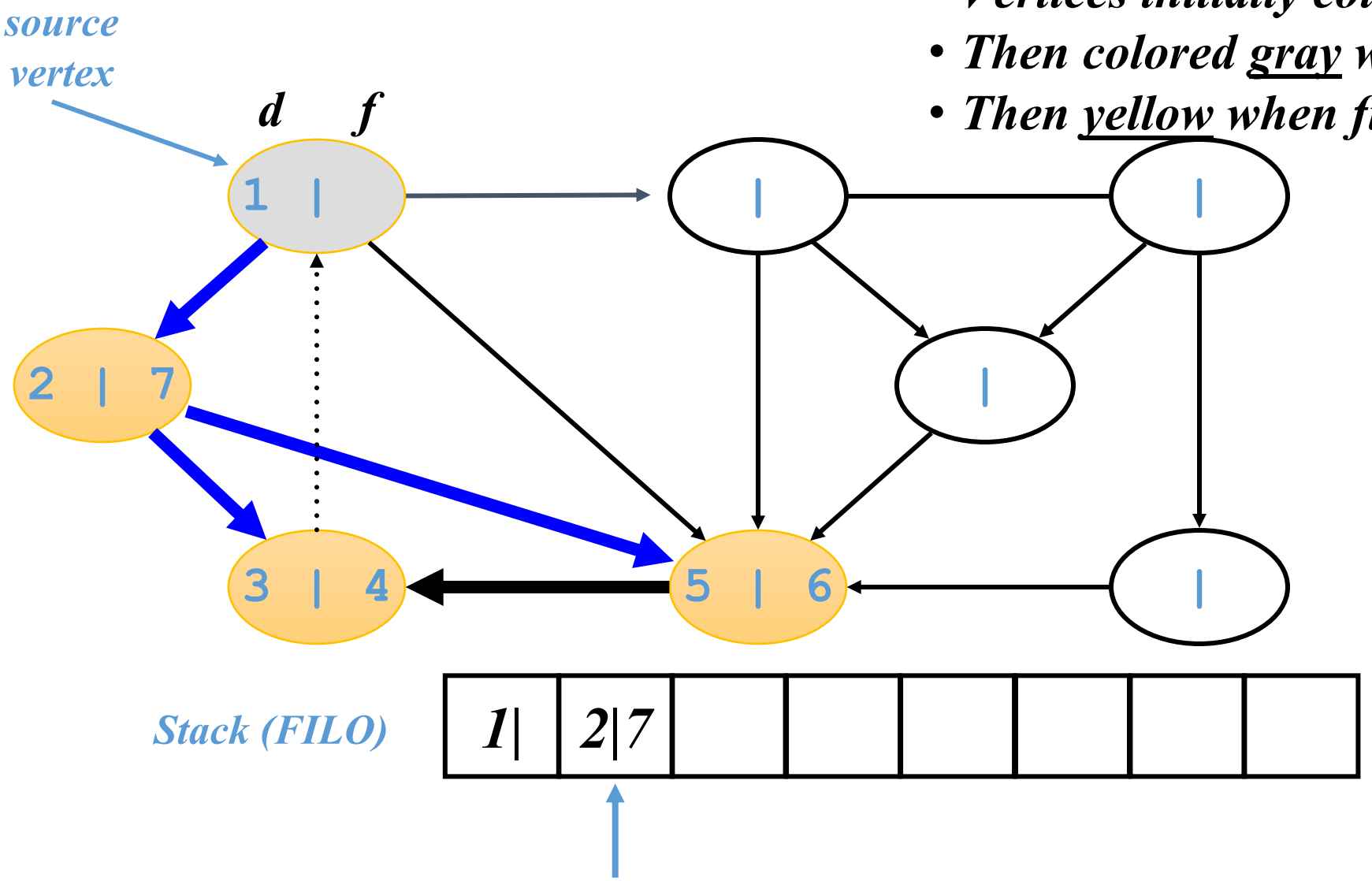
DFS Example

- *Vertices initially colored white*
- *Then colored gray when discovered*
- *Then yellow when finished*



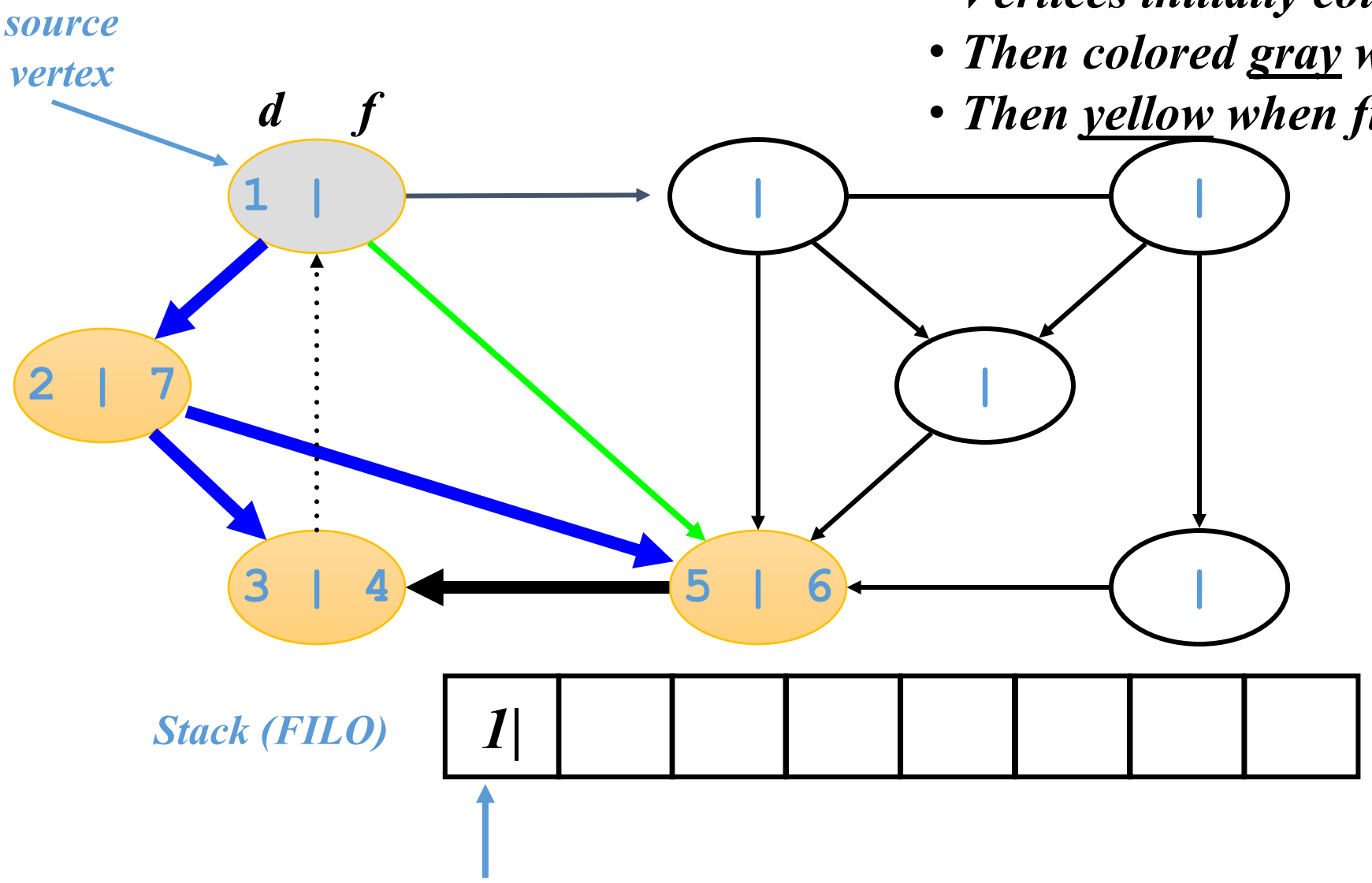
DFS Example

- *Vertices initially colored white*
- *Then colored gray when discovered*
- *Then yellow when finished*



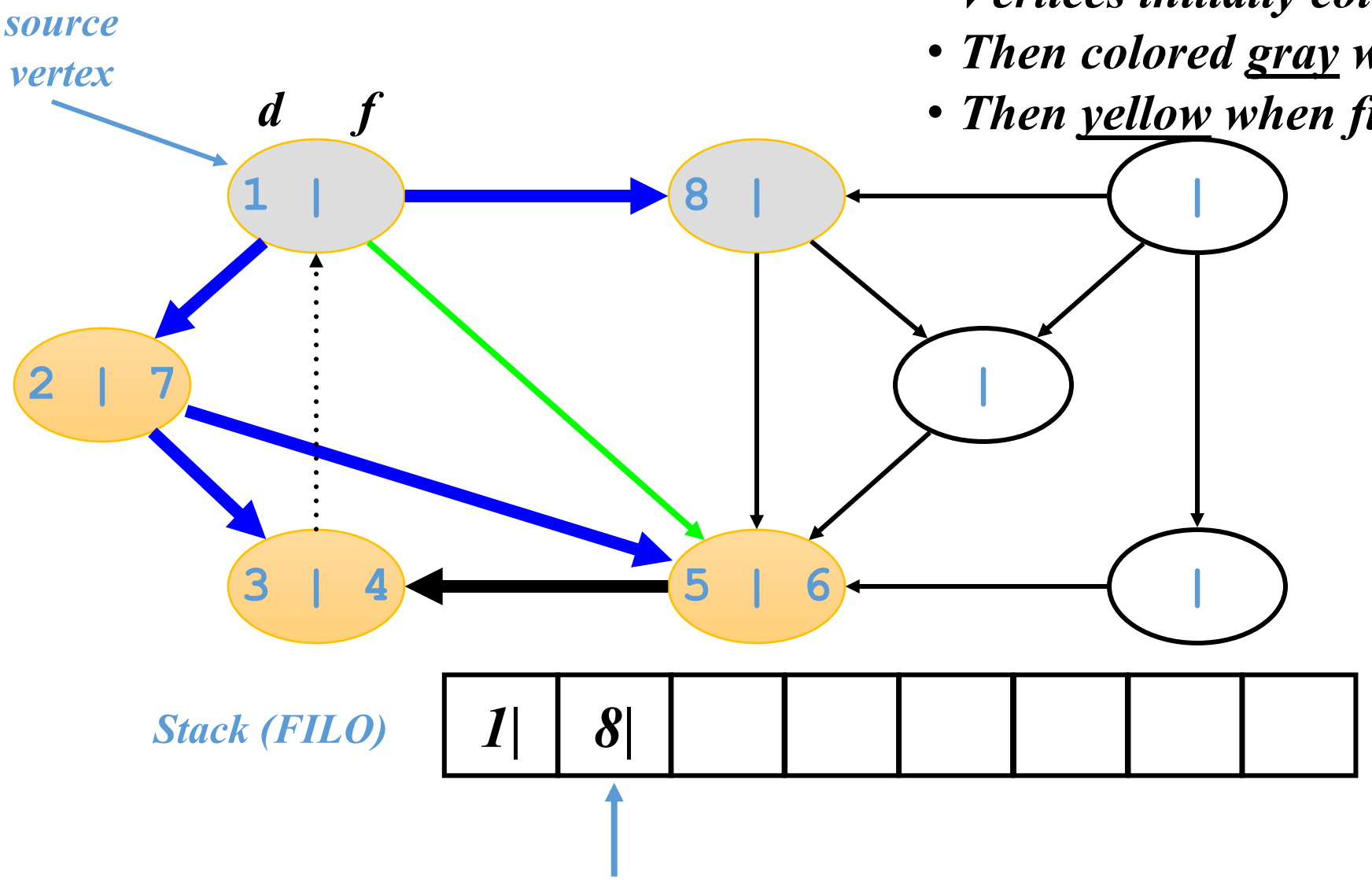
DFS Example

- *Vertices initially colored white*
- *Then colored gray when discovered*
- *Then yellow when finished*



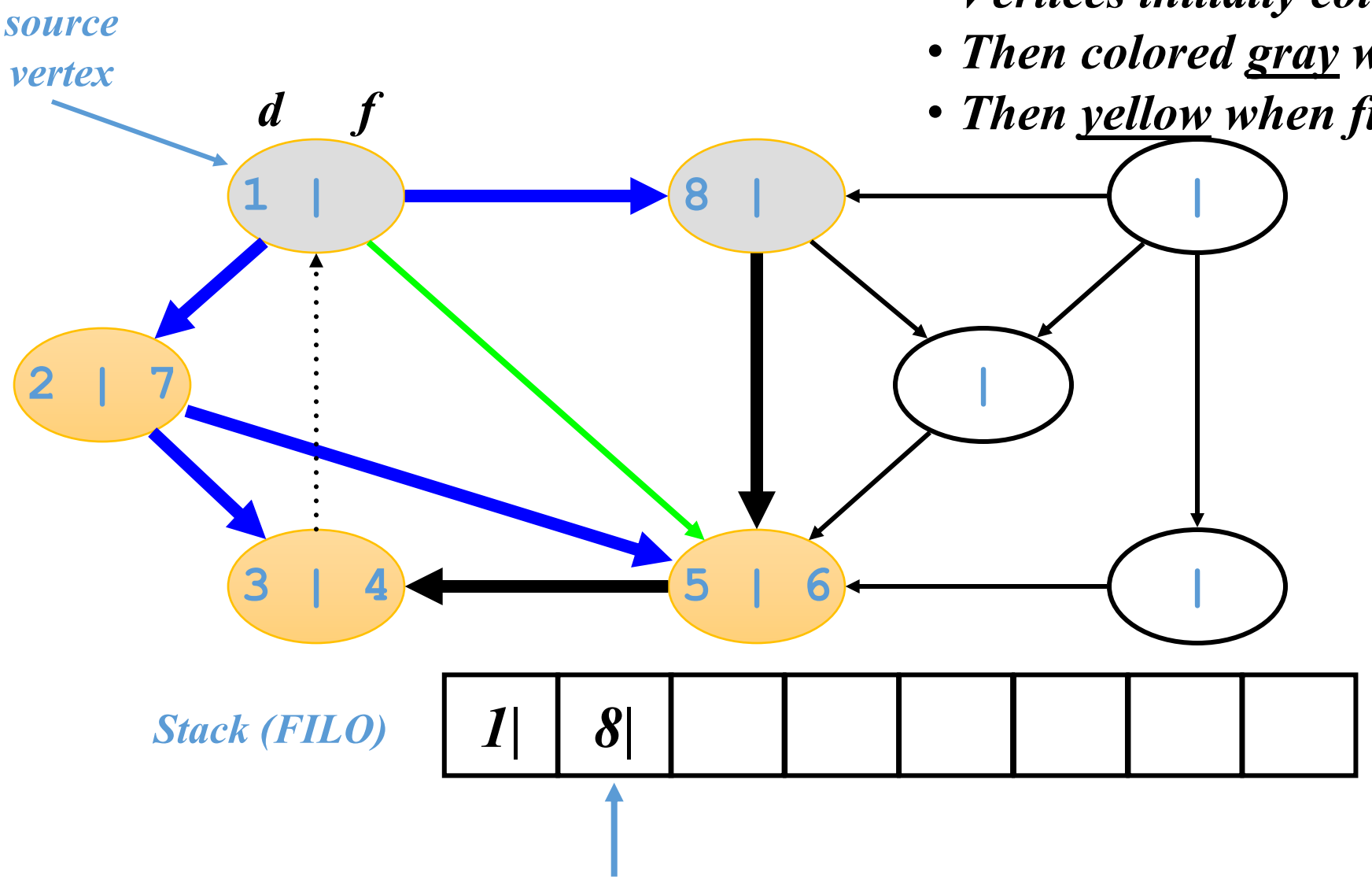
DFS Example

- *Vertices initially colored white*
- *Then colored gray when discovered*
- *Then yellow when finished*



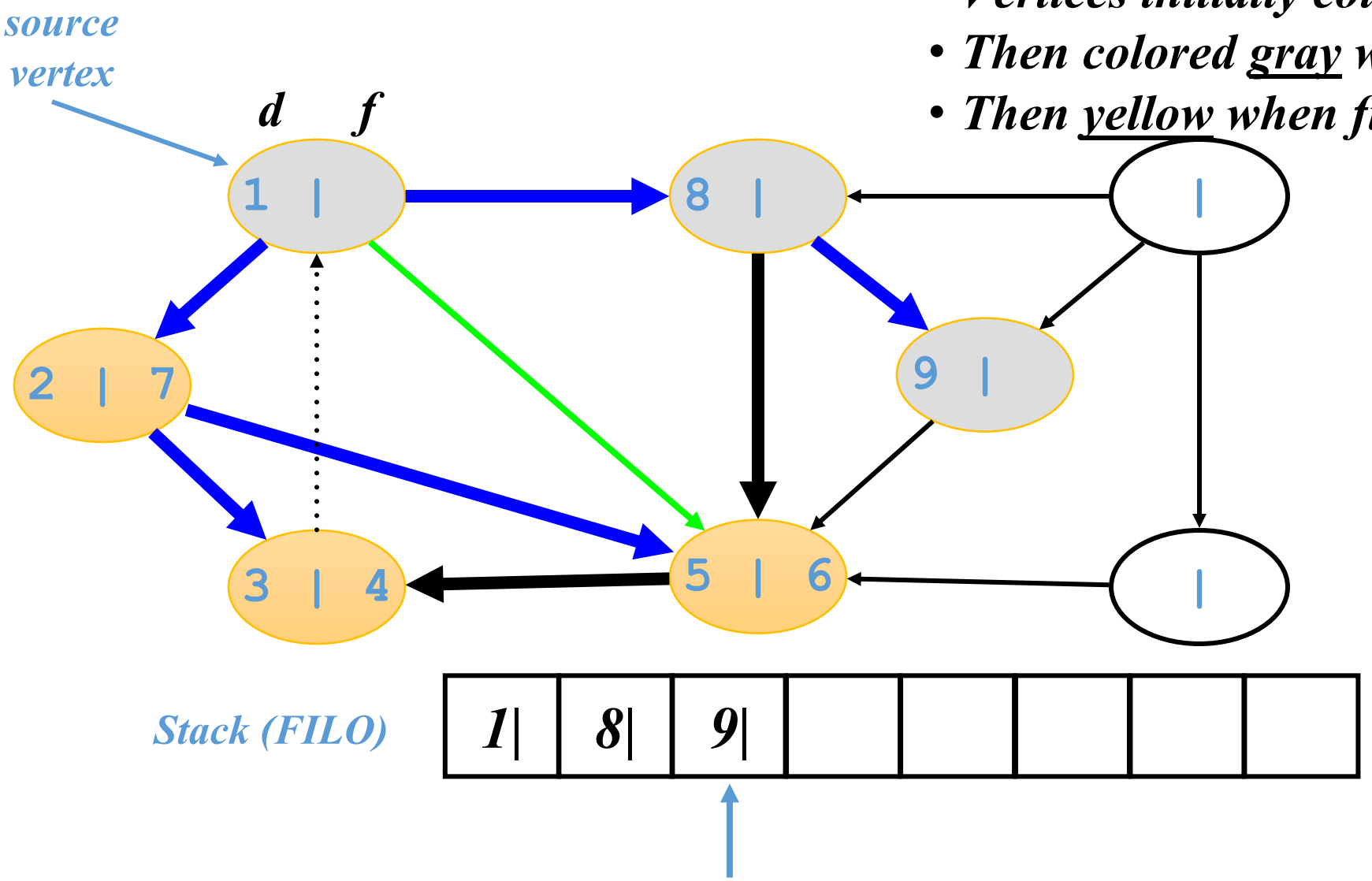
DFS Example

- *Vertices initially colored white*
- *Then colored gray when discovered*
- *Then yellow when finished*



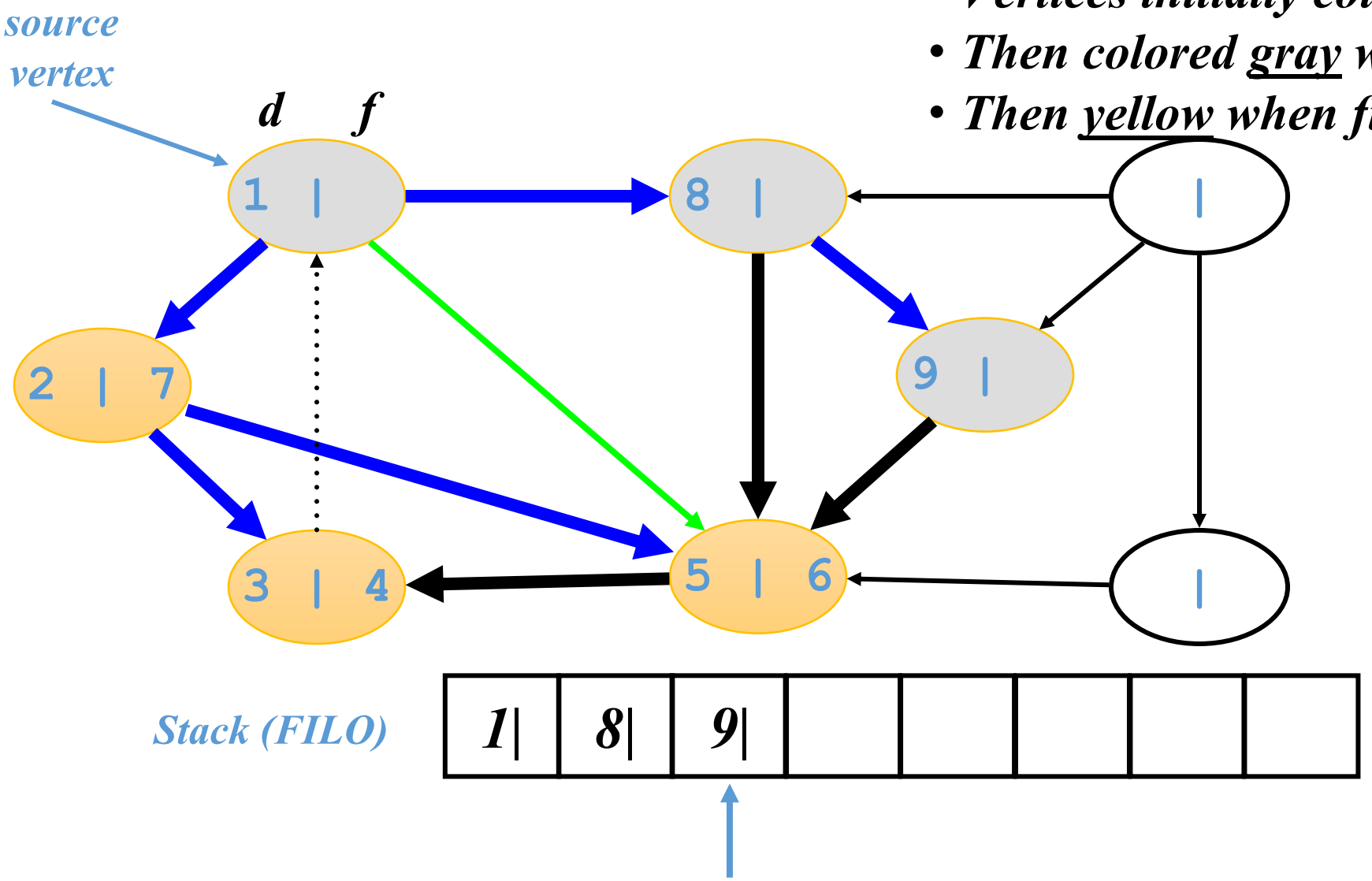
DFS Example

- *Vertices initially colored white*
- *Then colored gray when discovered*
- *Then yellow when finished*



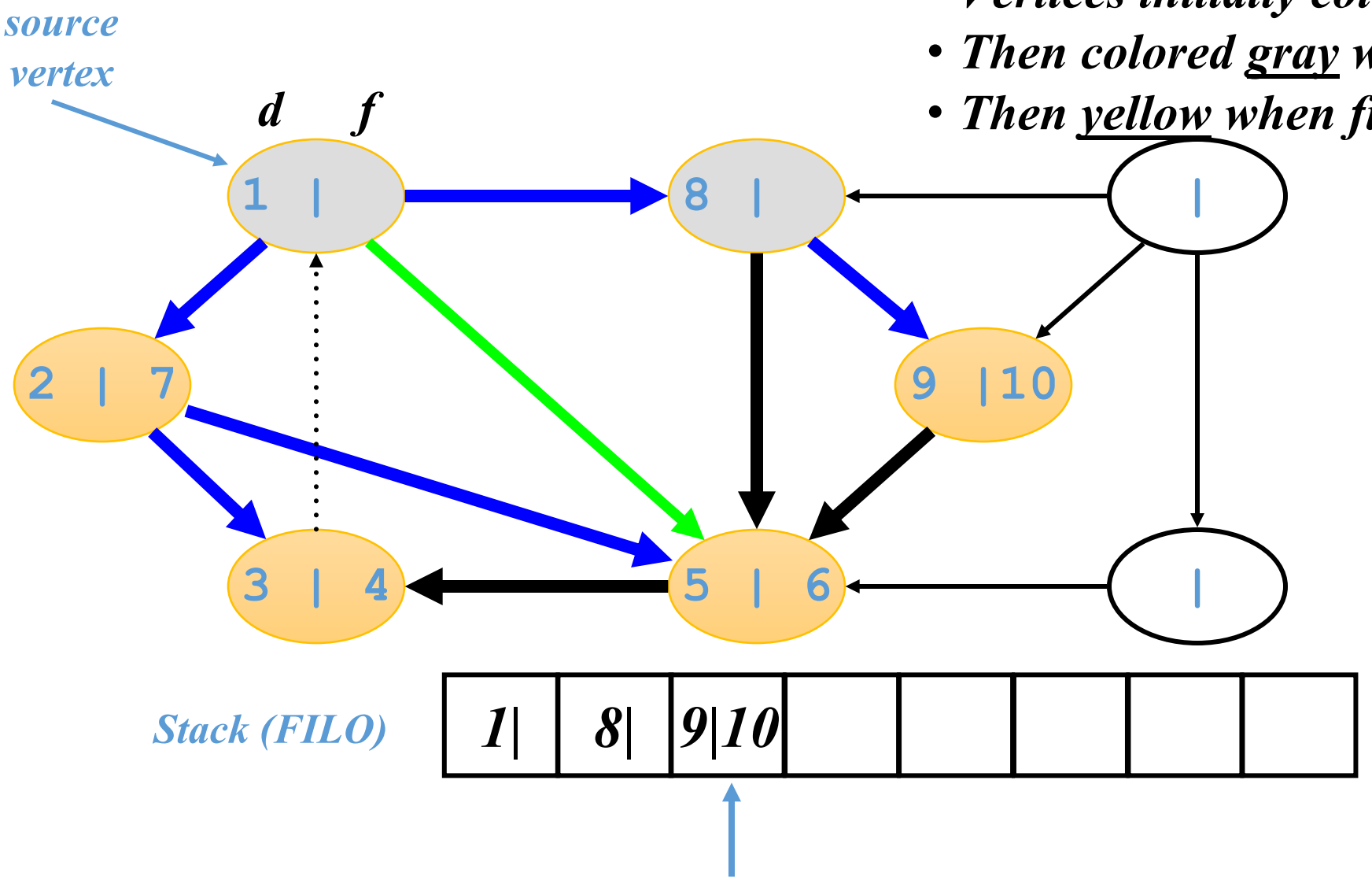
DFS Example

- *Vertices initially colored white*
- *Then colored gray when discovered*
- *Then yellow when finished*



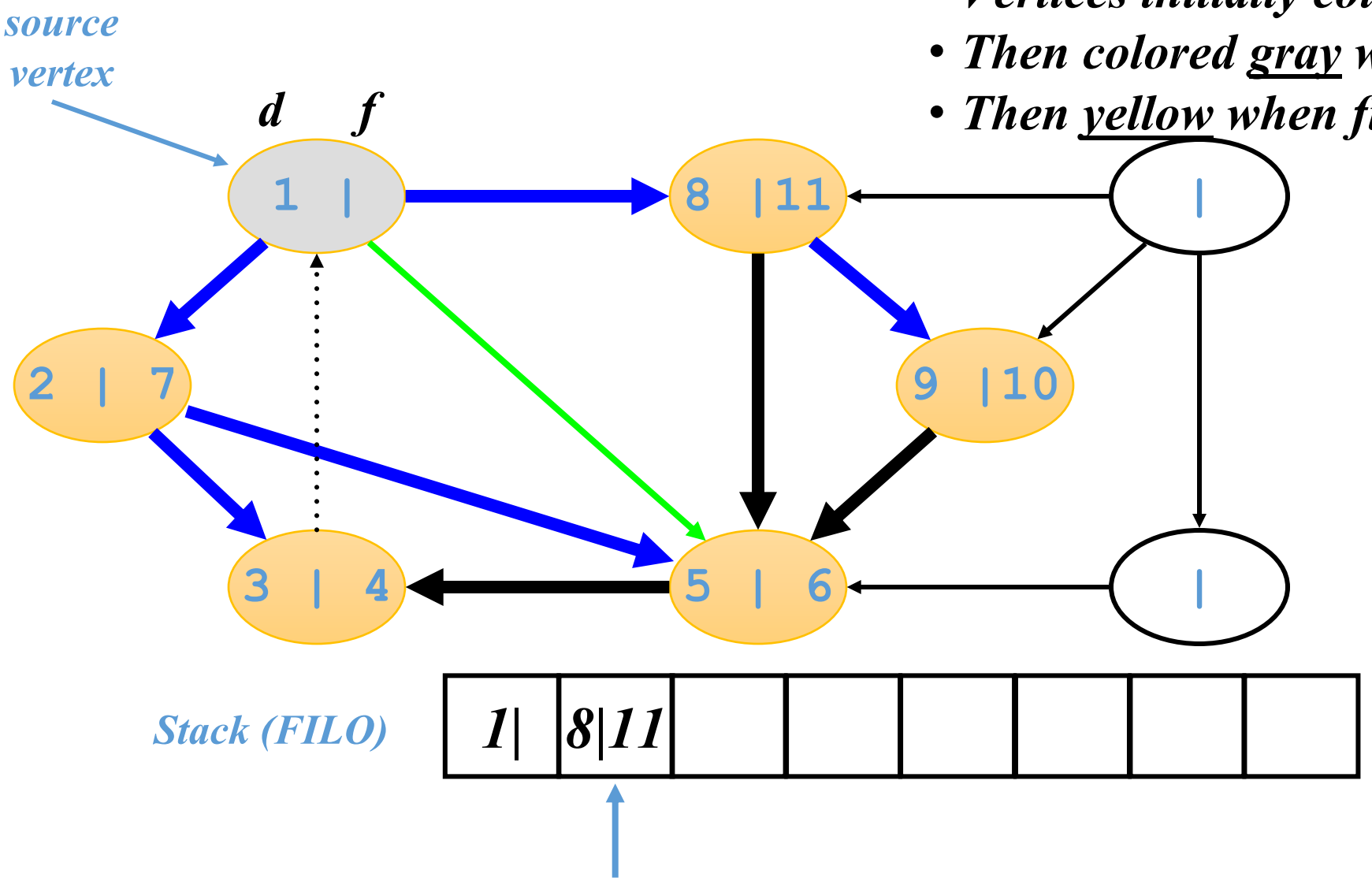
DFS Example

- *Vertices initially colored white*
- *Then colored gray when discovered*
- *Then yellow when finished*



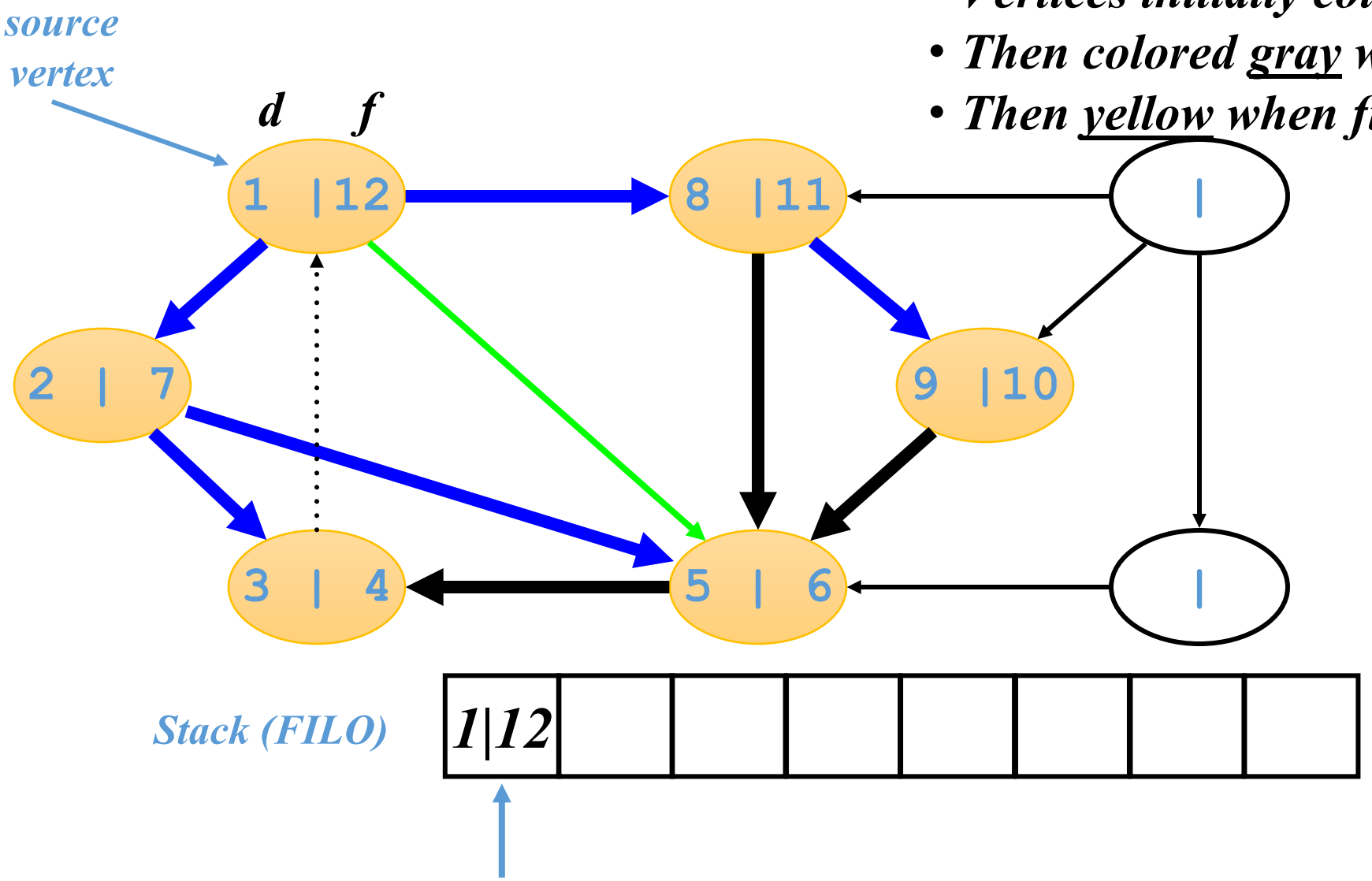
DFS Example

- *Vertices initially colored white*
- *Then colored gray when discovered*
- *Then yellow when finished*



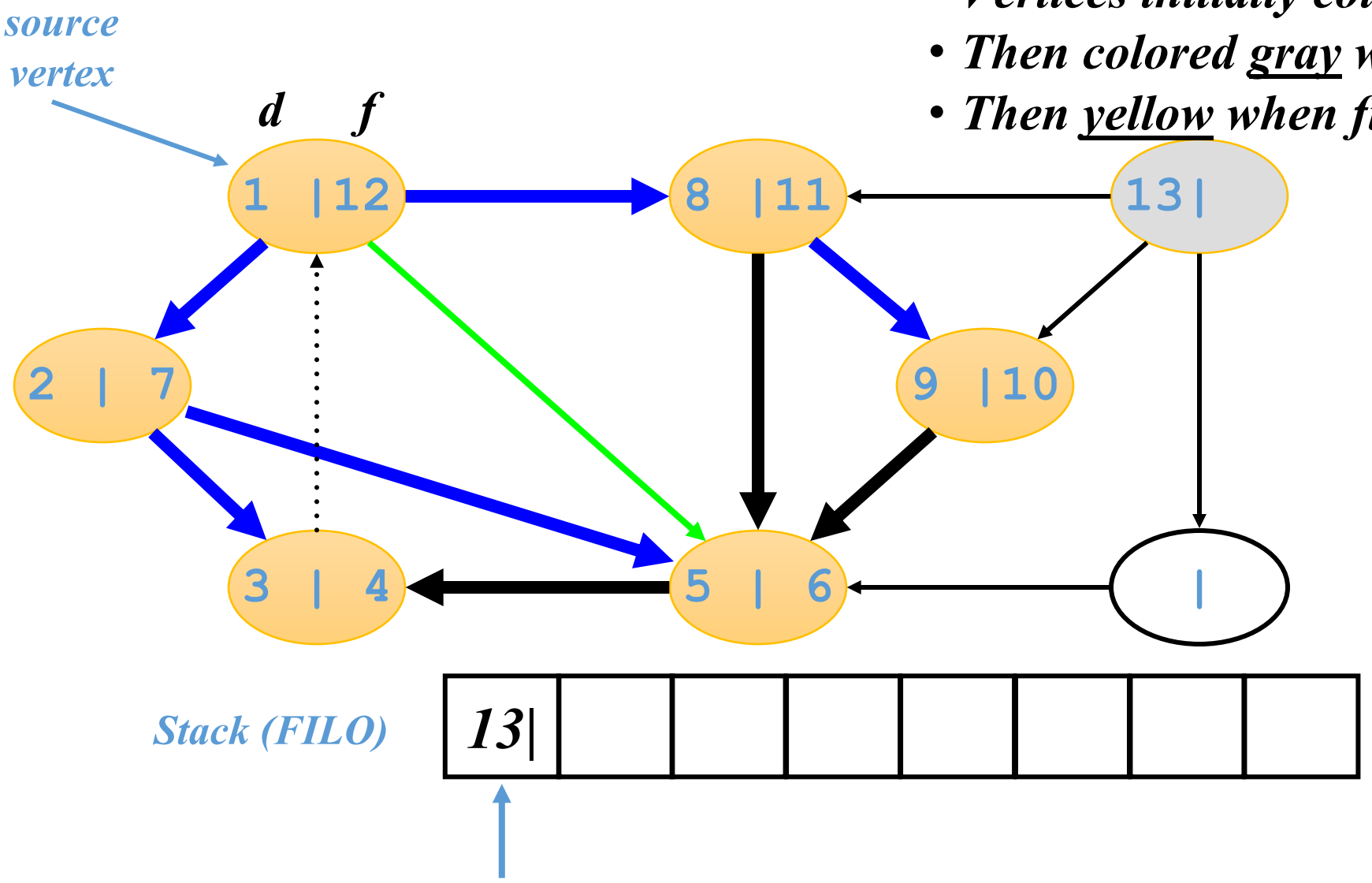
DFS Example

- *Vertices initially colored white*
- *Then colored gray when discovered*
- *Then yellow when finished*



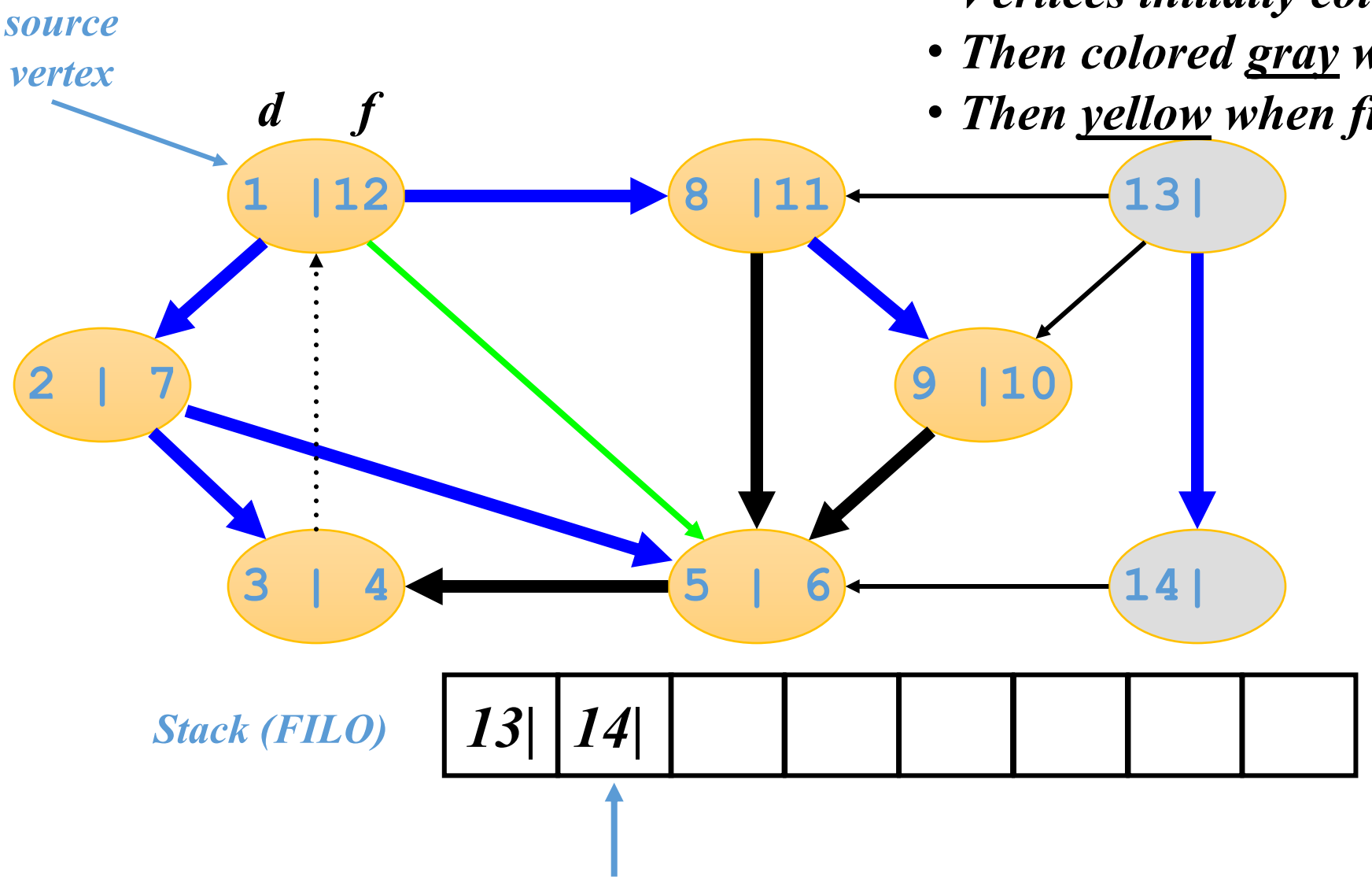
DFS Example

- *Vertices initially colored white*
- *Then colored gray when discovered*
- *Then yellow when finished*



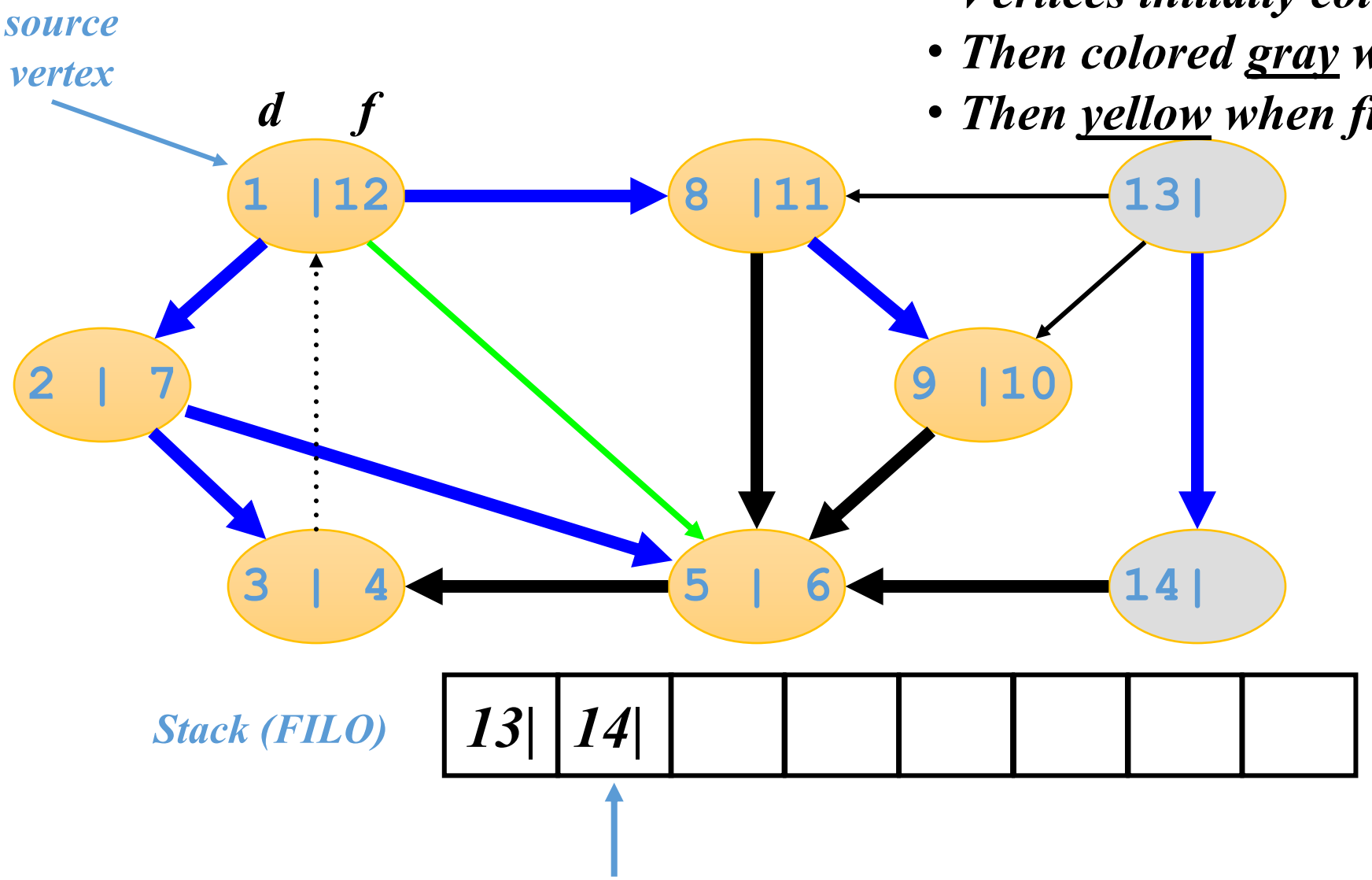
DFS Example

- *Vertices initially colored white*
- *Then colored gray when discovered*
- *Then yellow when finished*



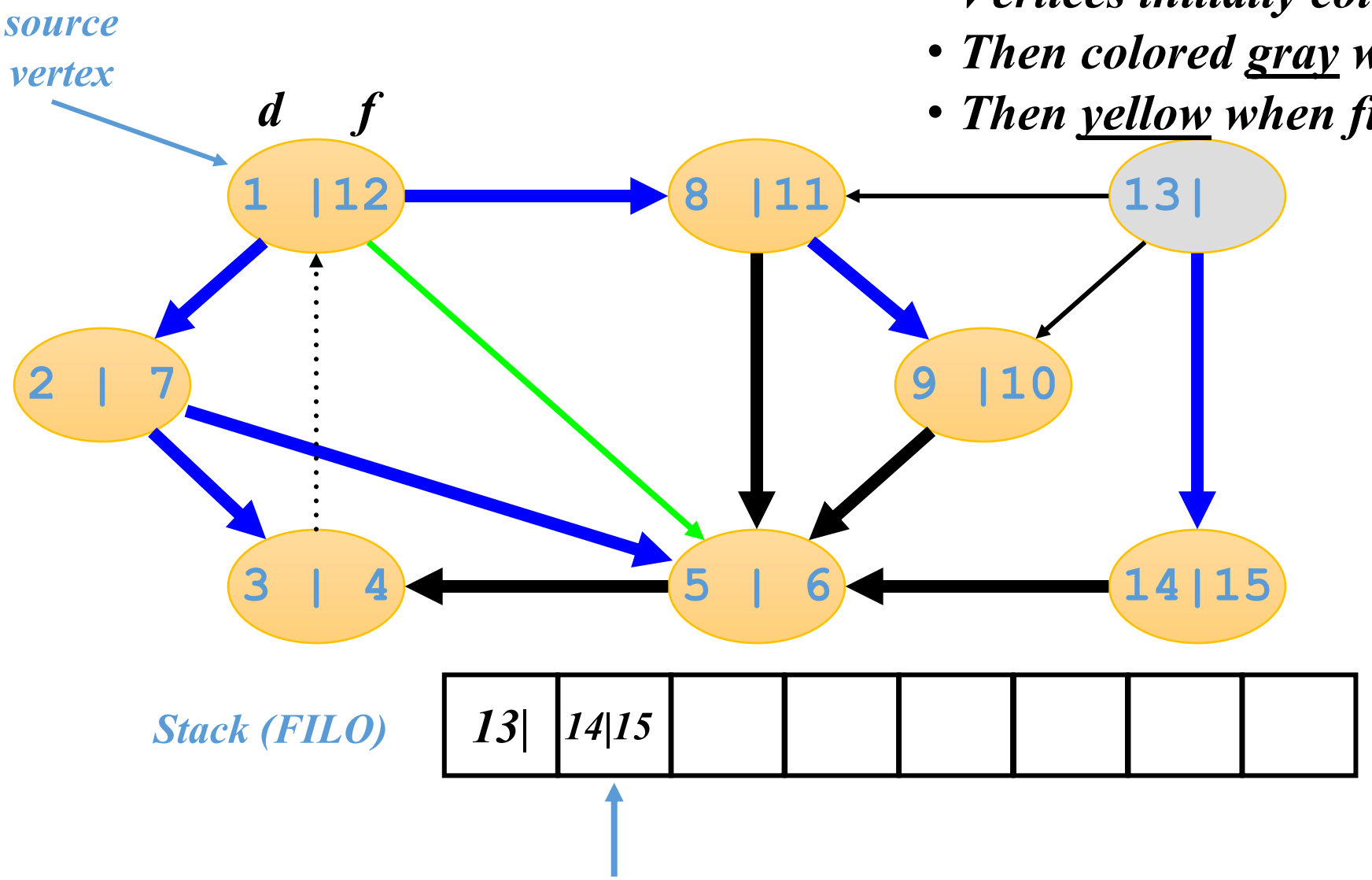
DFS Example

- *Vertices initially colored white*
- *Then colored gray when discovered*
- *Then yellow when finished*



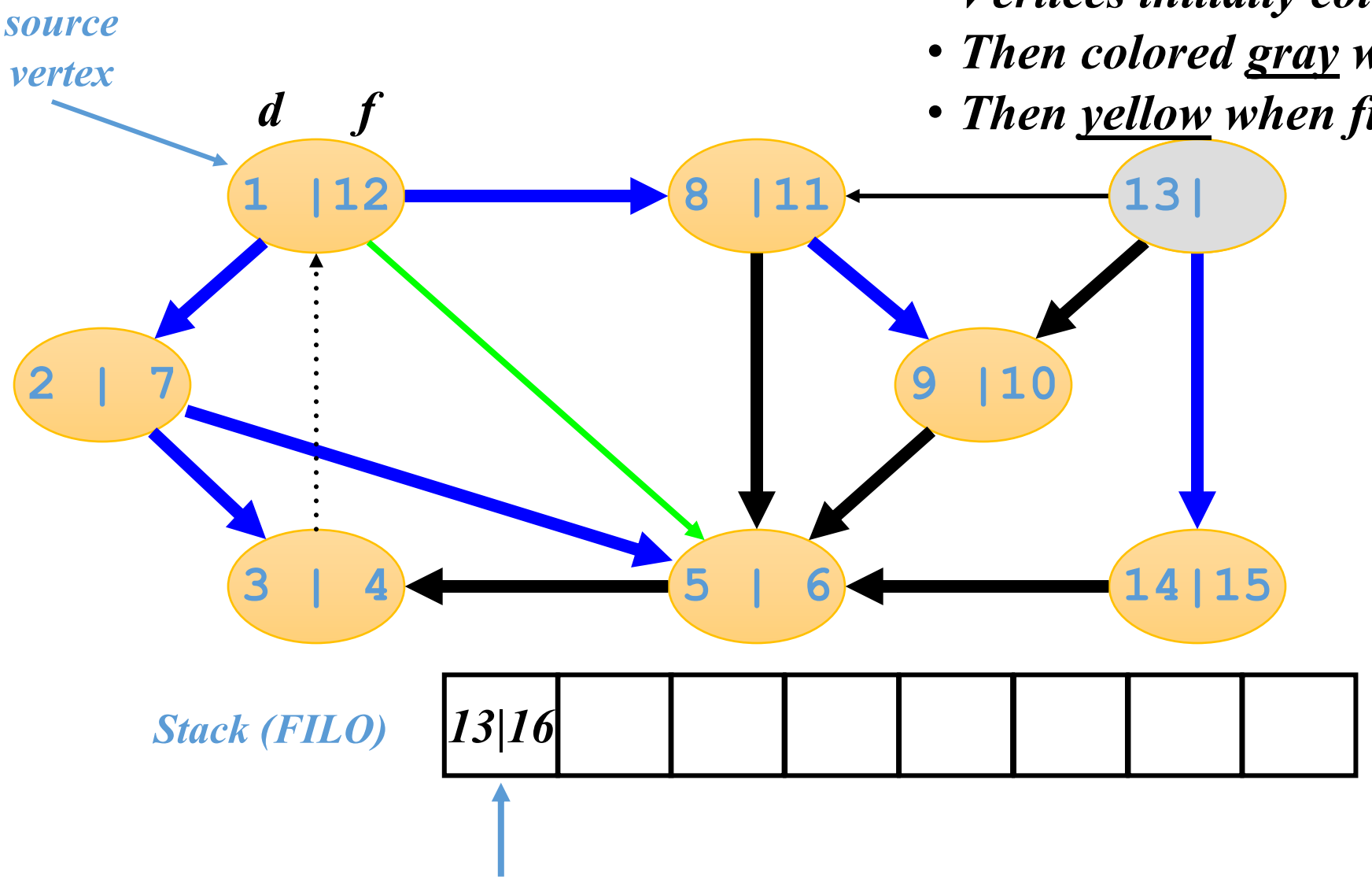
DFS Example

- *Vertices initially colored white*
- *Then colored gray when discovered*
- *Then yellow when finished*



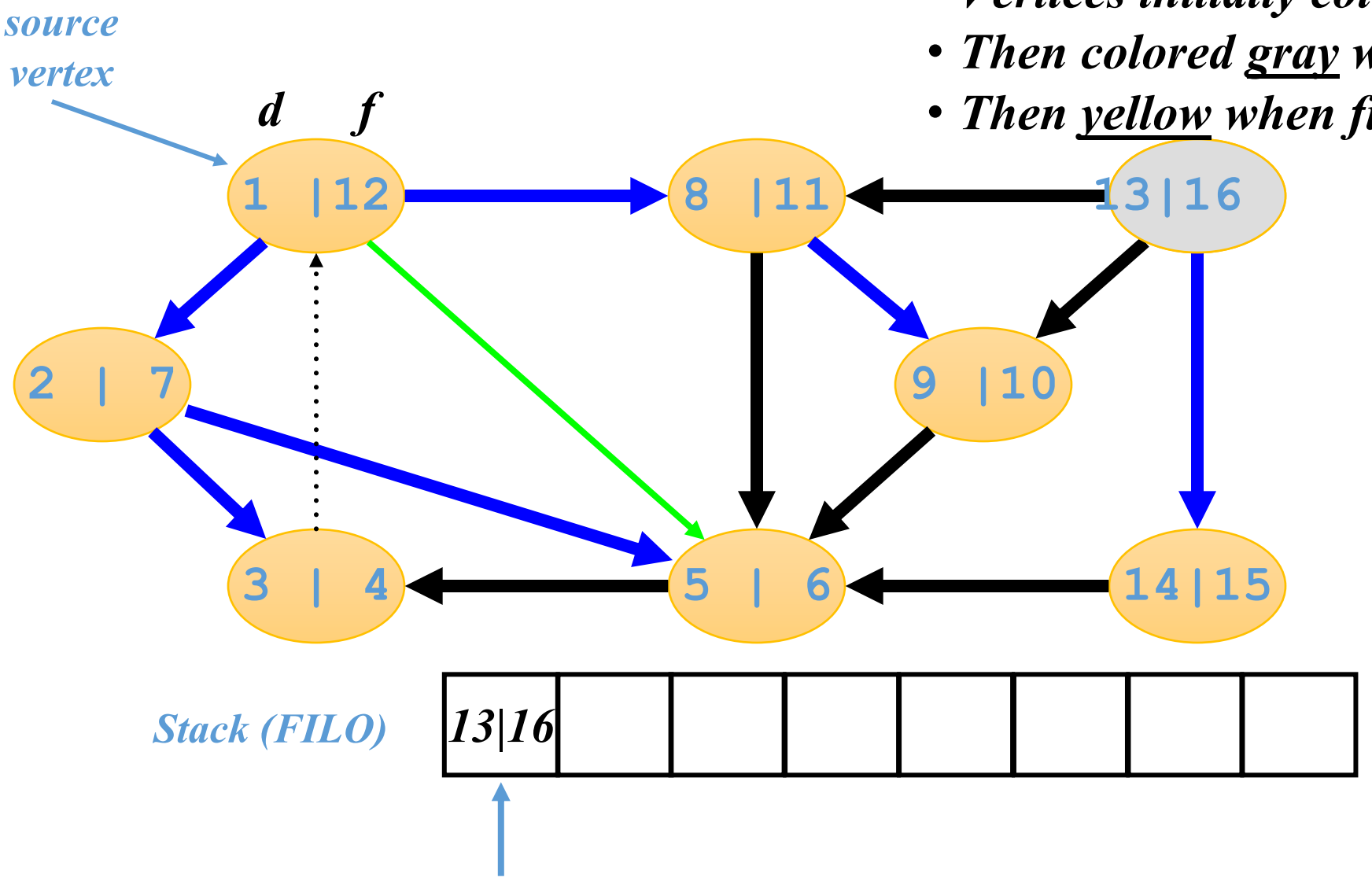
DFS Example

- *Vertices initially colored white*
- *Then colored gray when discovered*
- *Then yellow when finished*



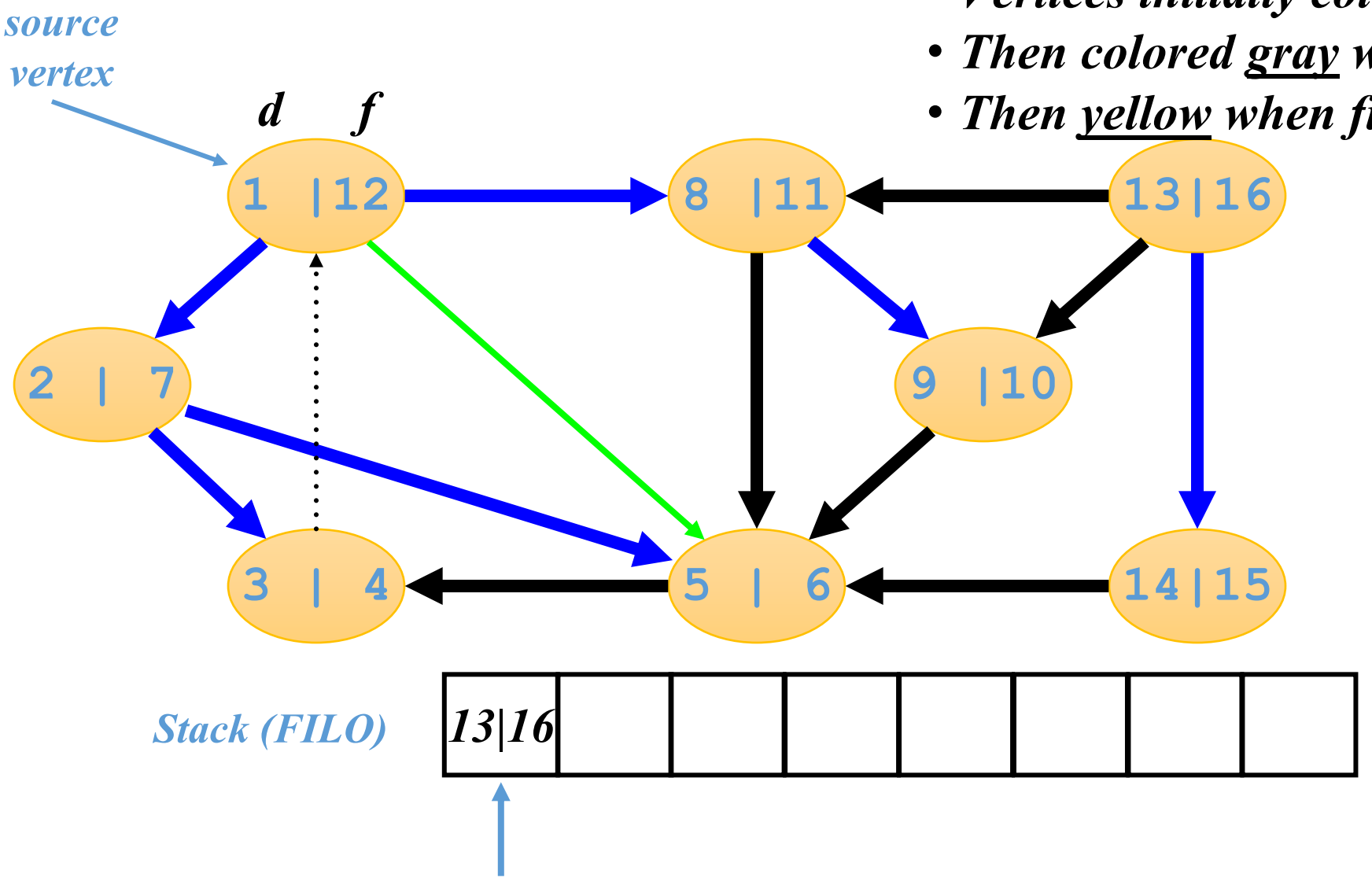
DFS Example

- *Vertices initially colored white*
- *Then colored gray when discovered*
- *Then yellow when finished*



DFS Example

- *Vertices initially colored white*
- *Then colored gray when discovered*
- *Then yellow when finished*

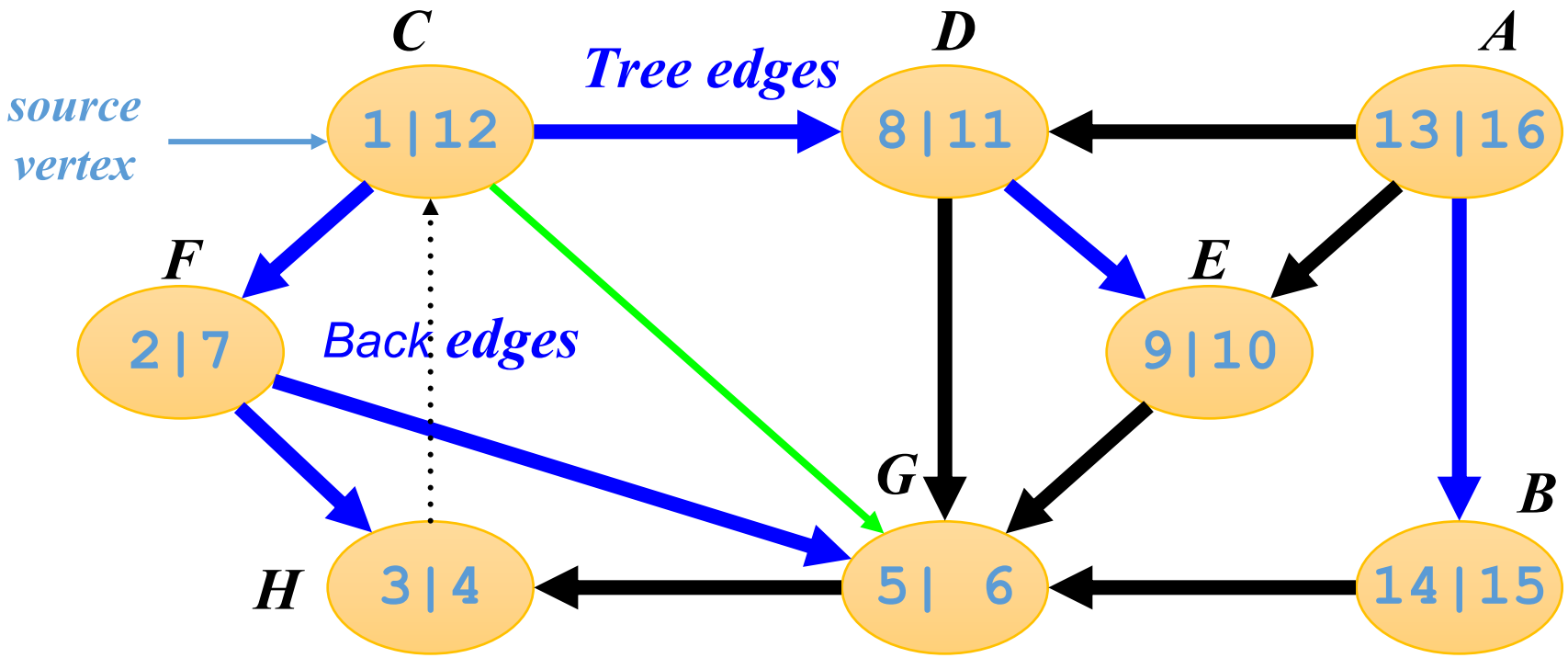


DFS: 4 Kinds of edges

A graph G is acyclic iff a DFS of G yields no back edges

Topological ordering = the inverse order of the 2nd numbers

$$A>B>C>D>E>F>G>H$$

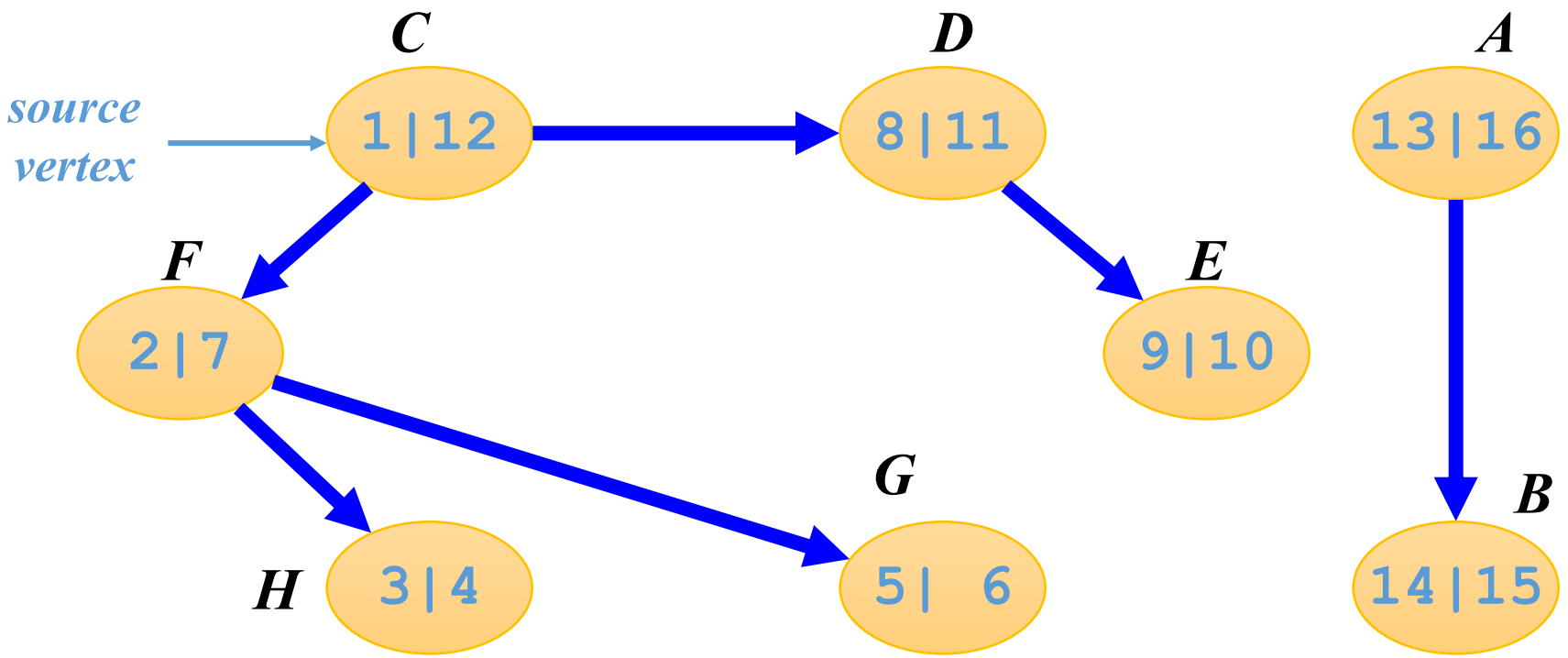


DFS: 4 Kinds of edges

A graph G is acyclic iff a DFS of G yields no back edges

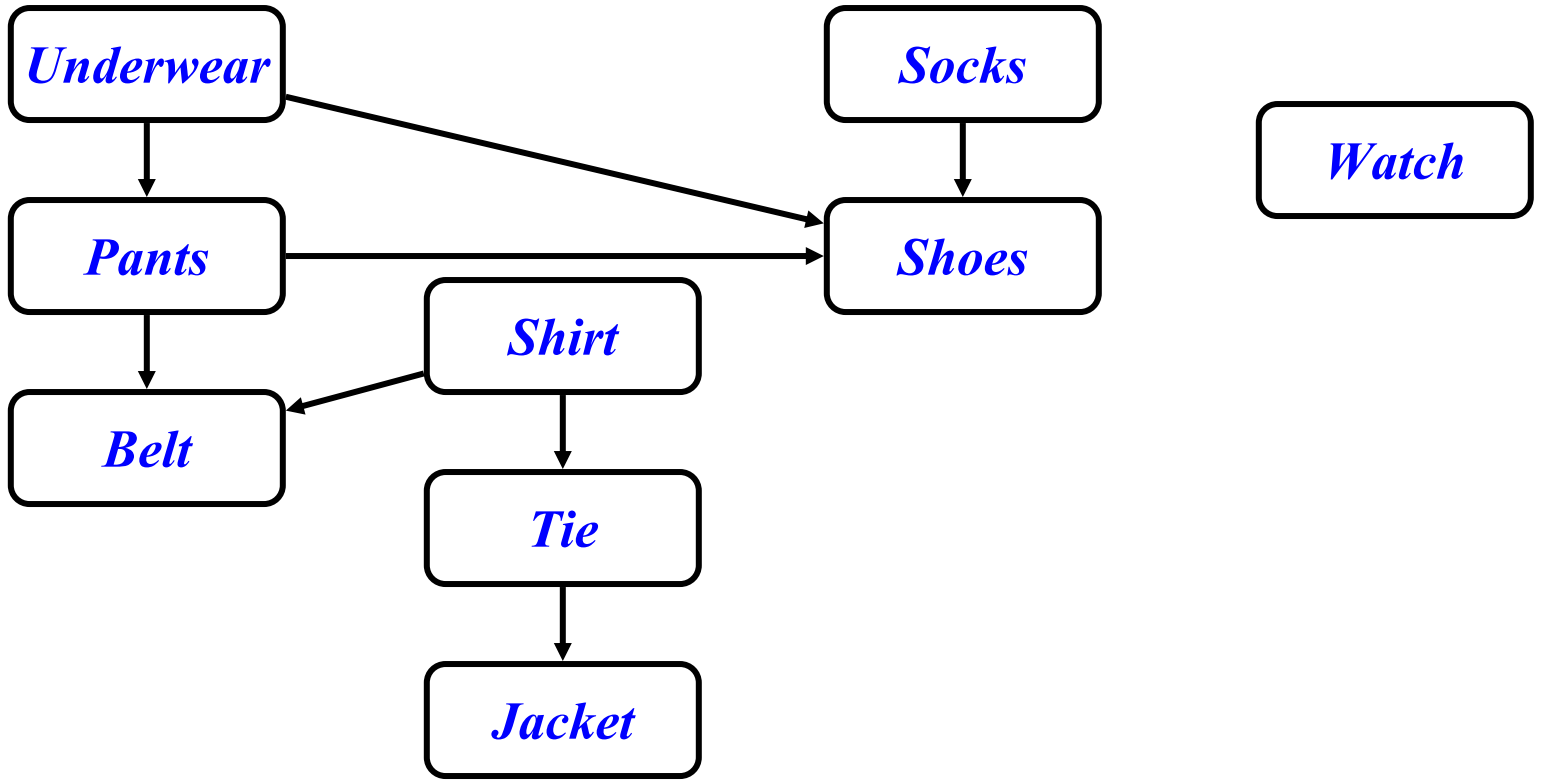
Topological ordering = the inverse order of the 2nd numbers

$$A>B>C>D>E>F>G>H$$

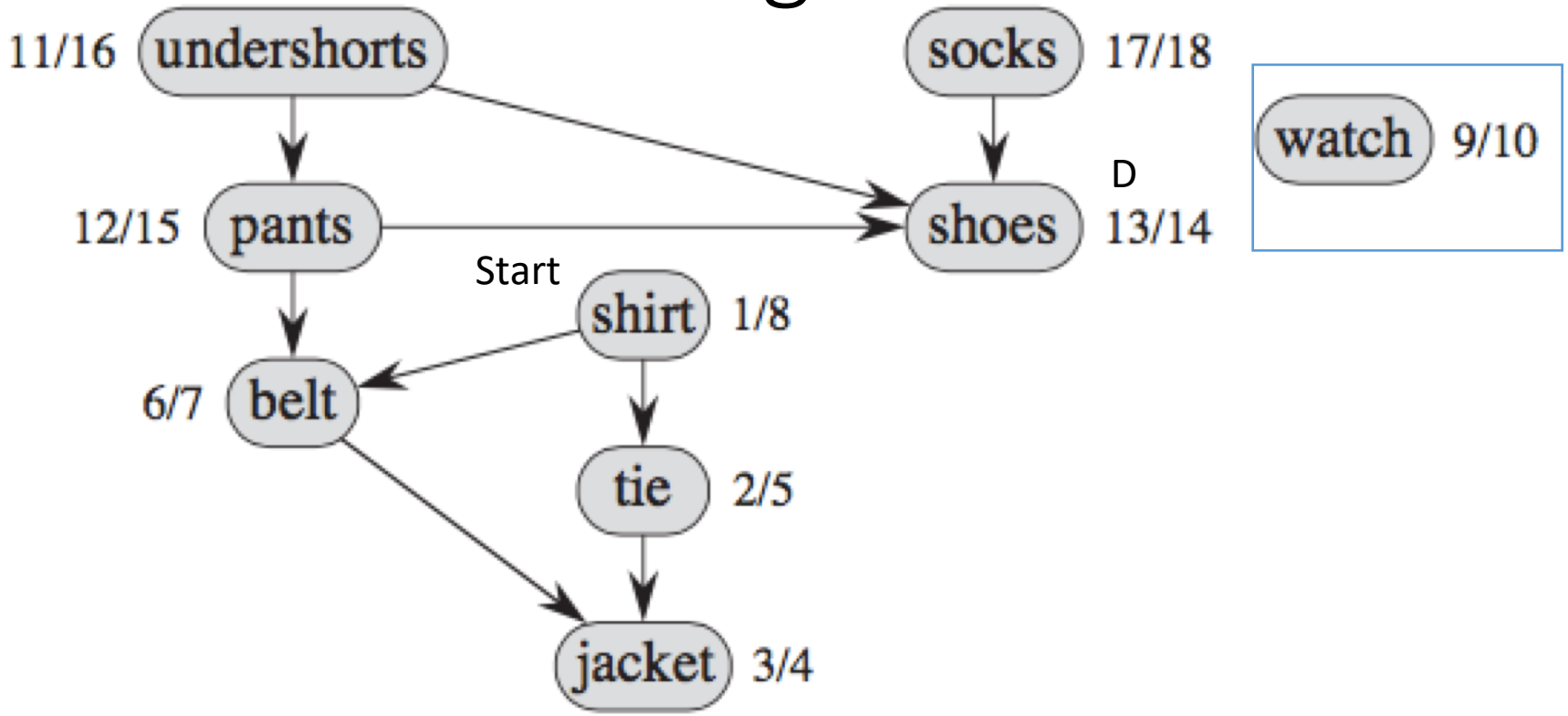


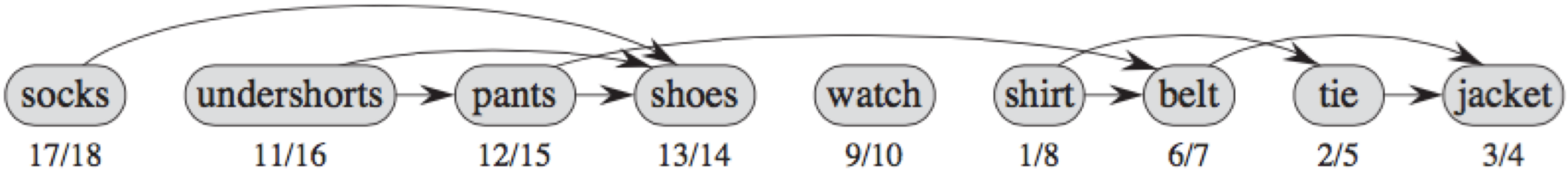
Topological Sort

- *Topological sort* of a DAG:
  - Linear ordering of all vertices in graph G such that vertex *u* comes before vertex *v* if edge (*u*, *v*) $\in$ G



Getting Dressed





Topological ordering = the inverse order of the 2nd numbers

Topological Sort Algorithm

```
Topological-Sort()
{  // condition: the graph is a DAG
   Run DFS
   When a vertex is finished, output it
   Vertices are output in reverse topological order
}
```

- Time: O(V+E)
- Correctness: Want to prove that

  $(u,v) \in G \Rightarrow u \rightarrow f > v \rightarrow f$

LeetCode 210. Course Schedule II

- Total n courses to take, labeled from 0 to n - 1.
- Courses have prerequisites:
  - [0,1] means course 1 is the prerequisite of course 0
- Compute the ordering of courses to satisfy all the prerequisite pairs.
  - There may be multiple or no answer.
- For example: 4, [[1,0],[2,0],[3,1],[3,2]]
  - Total 4 courses to take.
  - courses 1 and 2 are prerequisites of course 3.
  - Course 0 is the prerequisite of course 1 and 2.
  - Answers: [0,1,2,3] or [0,2,1,3].

Topological Sort Runtime

Runtime:

- $O(|V|)$ to build heap + $O(|E|)$ DECREASE-KEY ops

$\Rightarrow$ $O(|V| + |E| \log |V|)$ with a binary heap

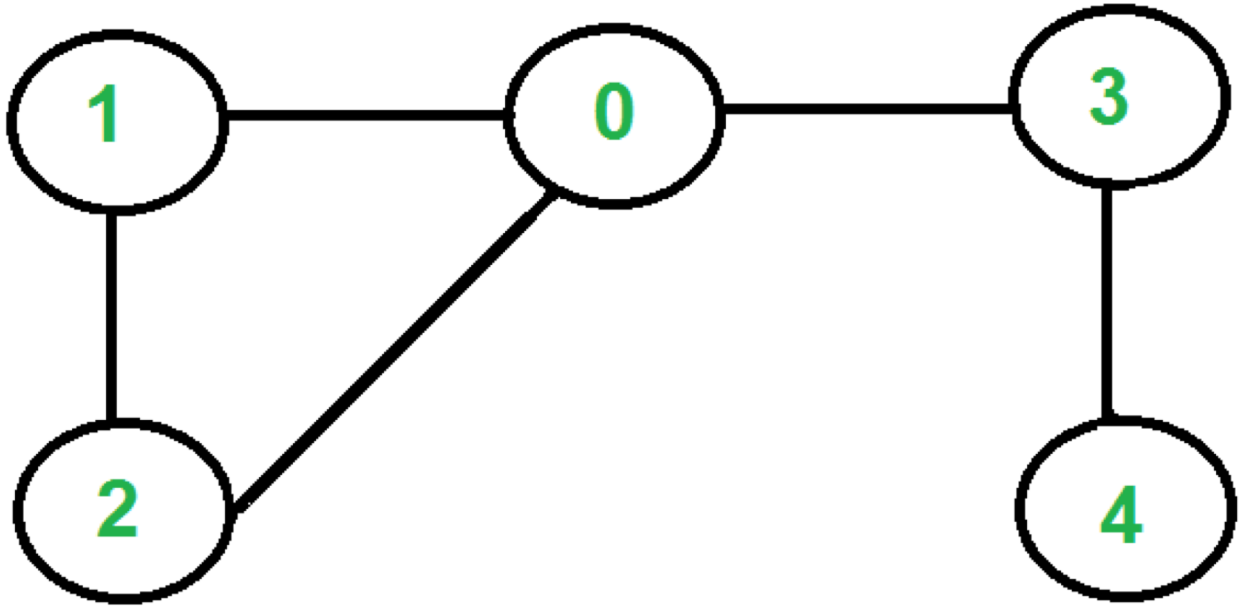
$\Rightarrow$ $O(|V|^2)$ with an array

Compare to DFS:

$\Rightarrow$ $O(|V|+|E|)$

DFS Application:
Eulerian Path & Fleury's Algorithm

http://www.geeksforgeeks.org/eulerian-path-and-circuit/

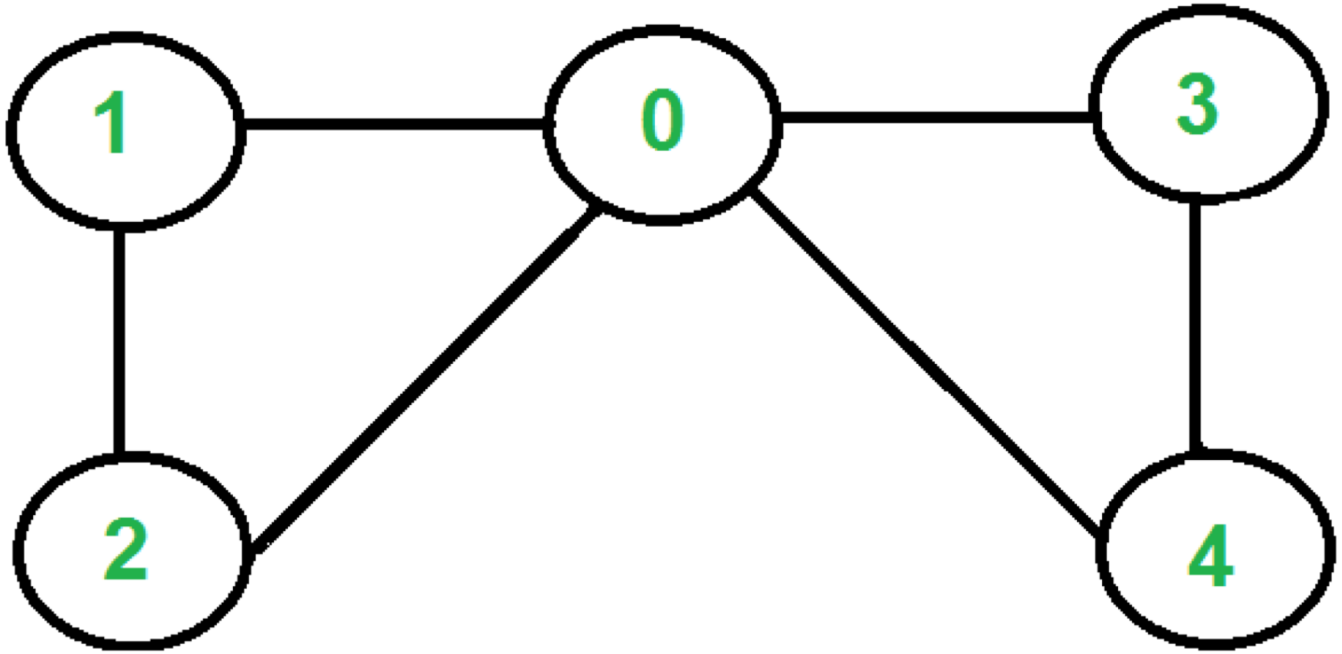


Can we trace the graph without lifting pencil from the paper and without tracing any of the edges more than once?

How many nodes have odd-number degree?

DFS Application:
Eulerian Path & Fleury's Algorithm

http://www.geeksforgeeks.org/eulerian-path-and-circuit/

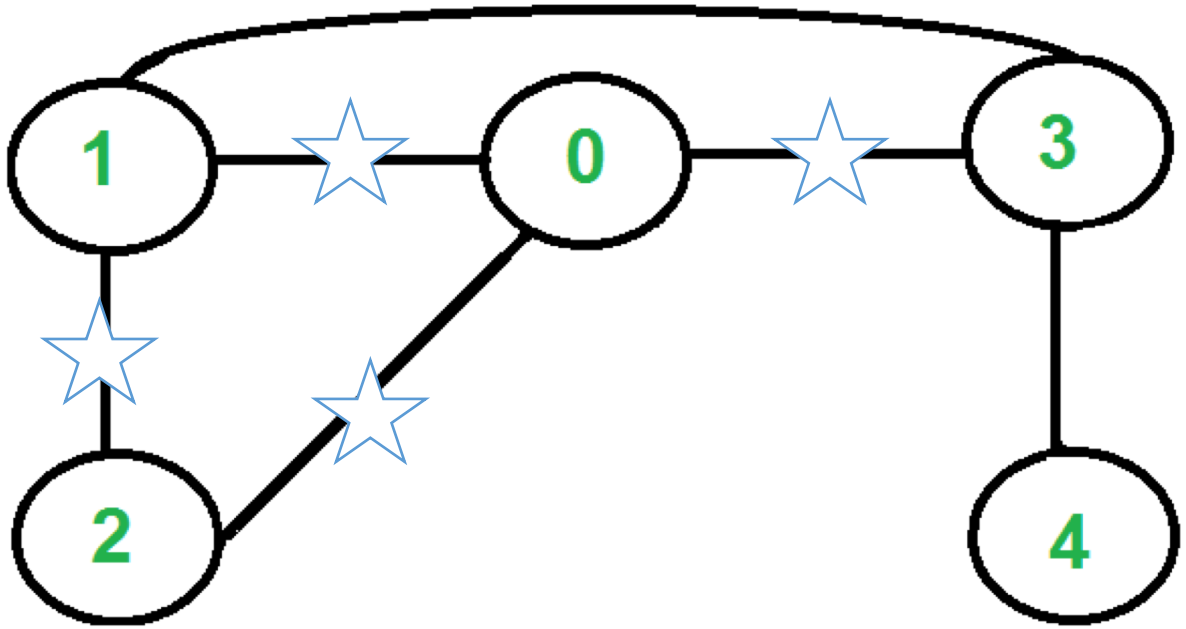


Can we trace the graph without lifting pencil from the paper and without tracing any of the edges more than once?

How many nodes have odd-number degree?

DFS Application:
Eulerian Path & Fleury's Algorithm

http://www.geeksforgeeks.org/eulerian-path-and-circuit/



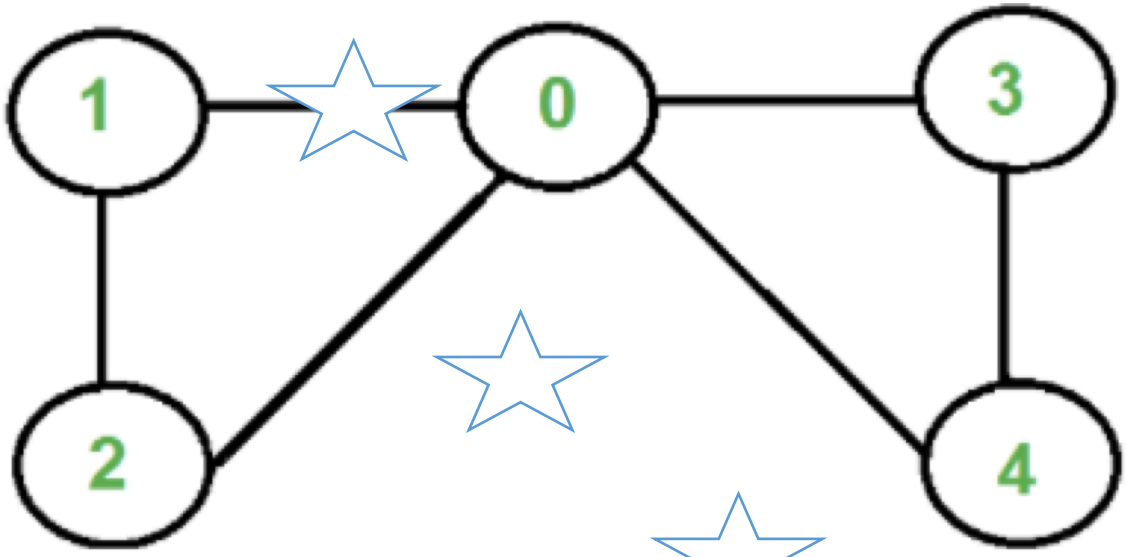
Can we trace the graph without lifting pencil from the paper and without tracing any of the edges more than once?

How many nodes have odd-number degree?

DFS Application:
Eulerian Path & Fleury's Algorithm

http://www.geeksforgeeks.org/fleurys-algorithm-for-printing-eulerian-path/

- Make sure the graph has either 0 or 2 odd vertices.
- If there are 0 odd vertices, start anywhere. If there are 2 odd vertices, start at one of them.
- Follow edges one at a time.
  - **Do not burn the bridge**: ff you have a choice between a bridge and a non-bridge, always choose the non-bridge
- Stop when you run out of edges.

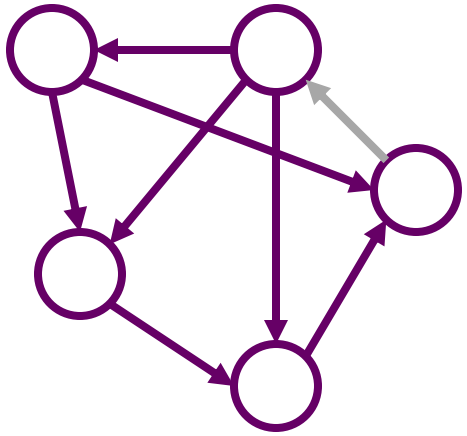


The graph has Eulerian Cycles, for example "2 1 0 3 4 0 2"
Note that all vertices have even degree

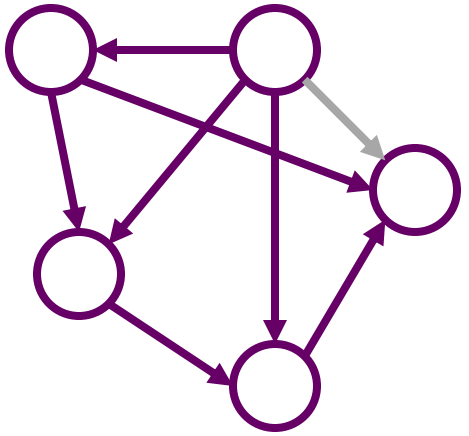
Strongly Connected Components

- Every pair of vertices are reachable from each other
- Graph G is strongly connected if, for every u and v in V, there is some path from u to v and some path from v to u.

Strongly
Connected

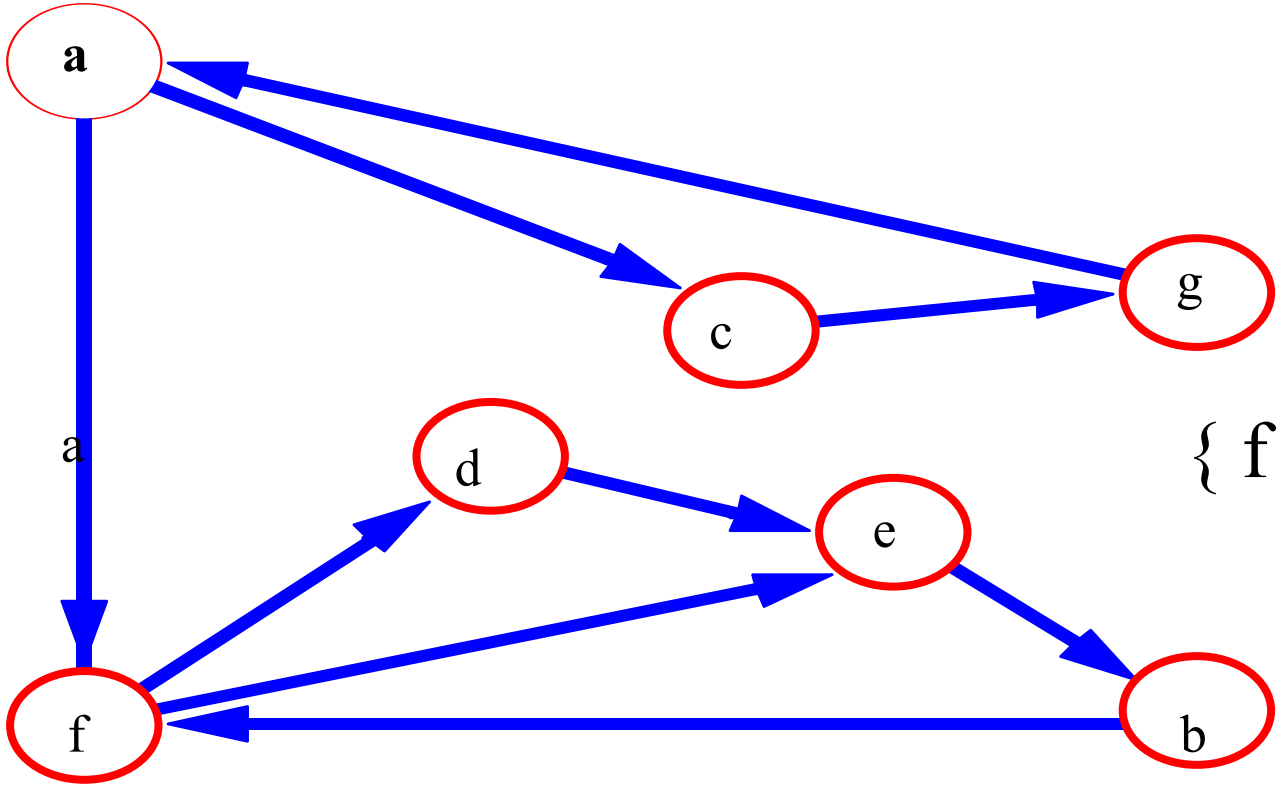


Not Strongly
Connected



Finding Strongly Connected Components

- Input: A directed graph G = (V,E)
- Output: a partition of V into disjoint sets so that each set defines a strongly connected component (SCC) of G



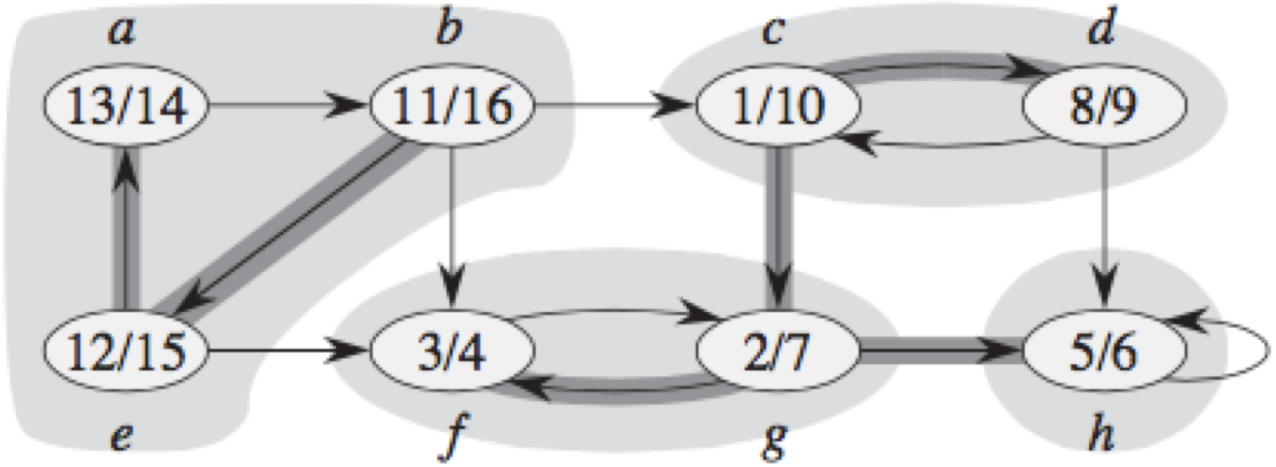
{ a , c , g }

{ f , d , e , b }

Strongly-Connected-Components(G)

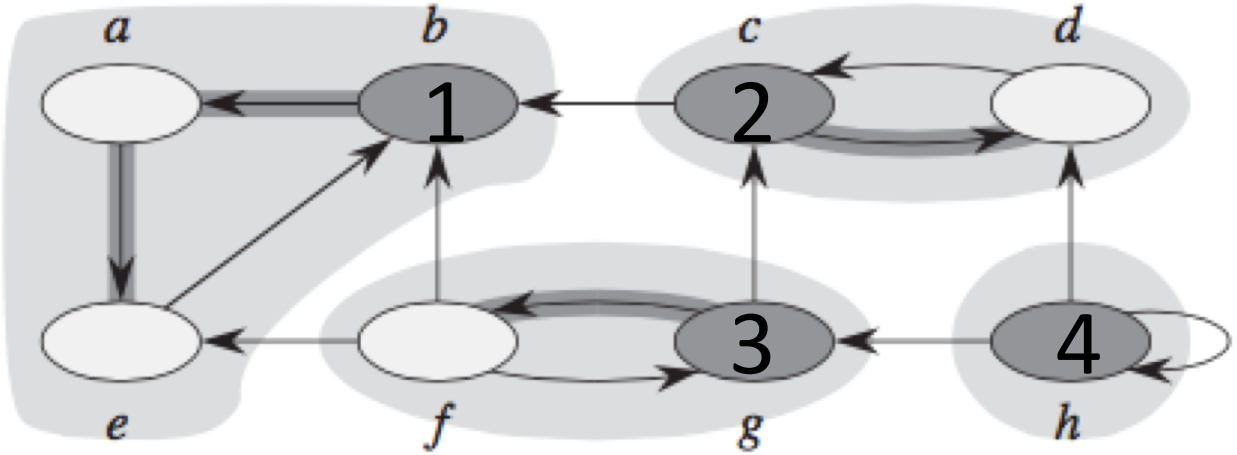
1. Call DFS(G) to compute finishing times f[u] for each vertex u.
   - Cost: O(E+V)
2. Compute G^T (transpose of G, all edges are revered.)
   - Cost: O(E+V)
3. Call DFS(G^T), but in the main loop of DFS, consider the vertices in order of decreasing f[u]
   - Cost: O(E+V)
4. Each tree output at step 3 is a SCC of G

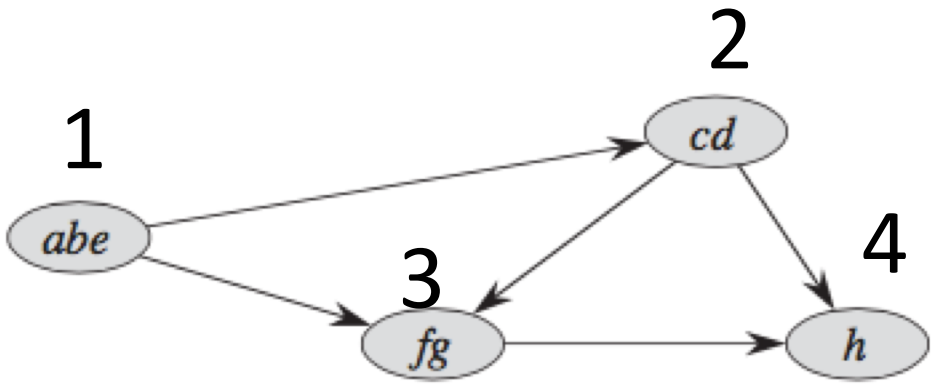
1. Call DFS(G) to compute finishing times f[u] for each vertex u. Cost: O(E+V)



DFS(G^T)

2. Compute GT (transpose of G, all edges are revered.) :Cost: O(E+V)





3. Call DFS(GT), but in the main loop of DFS, consider the vertices in order of decreasing f[u]
   Cost: O(E+V)

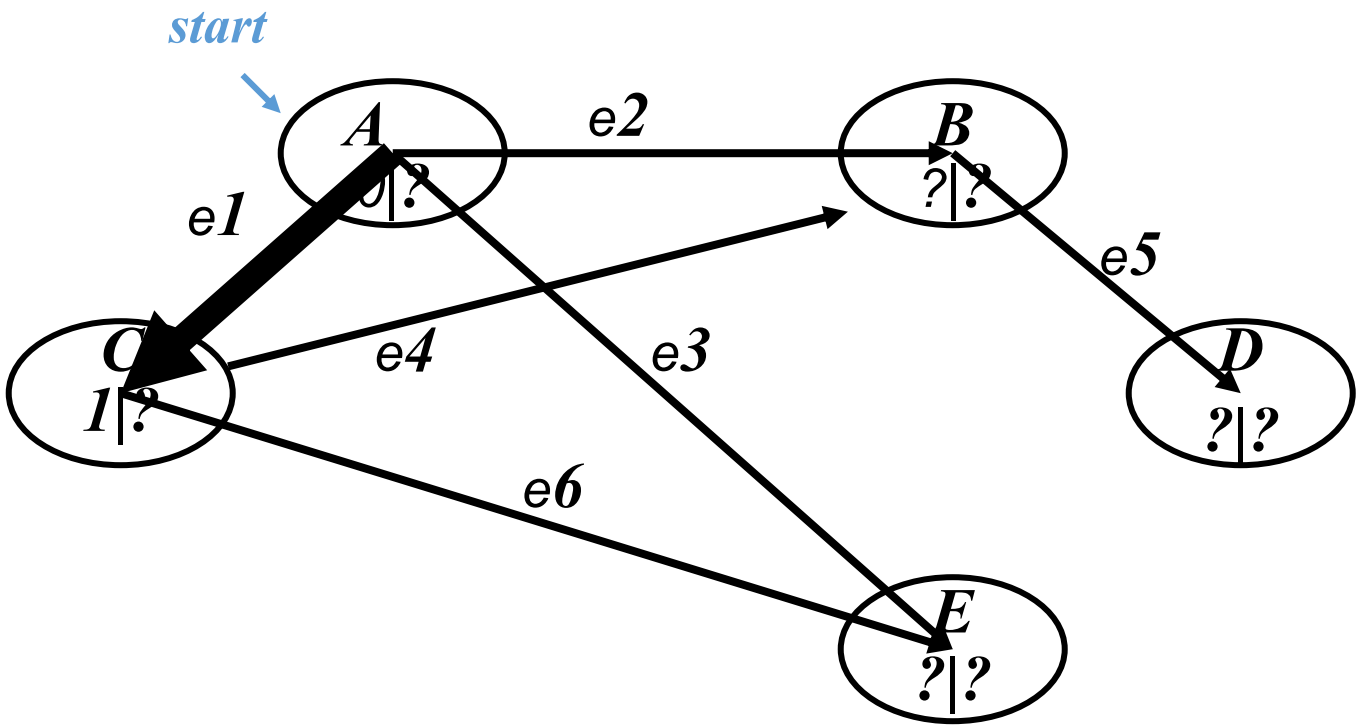
4. Each tree output at step 3 is a SCC of G

Programming Exercise

- Do the programming exercise at http://www.geeksforgeeks.org/strongly-connected-components/

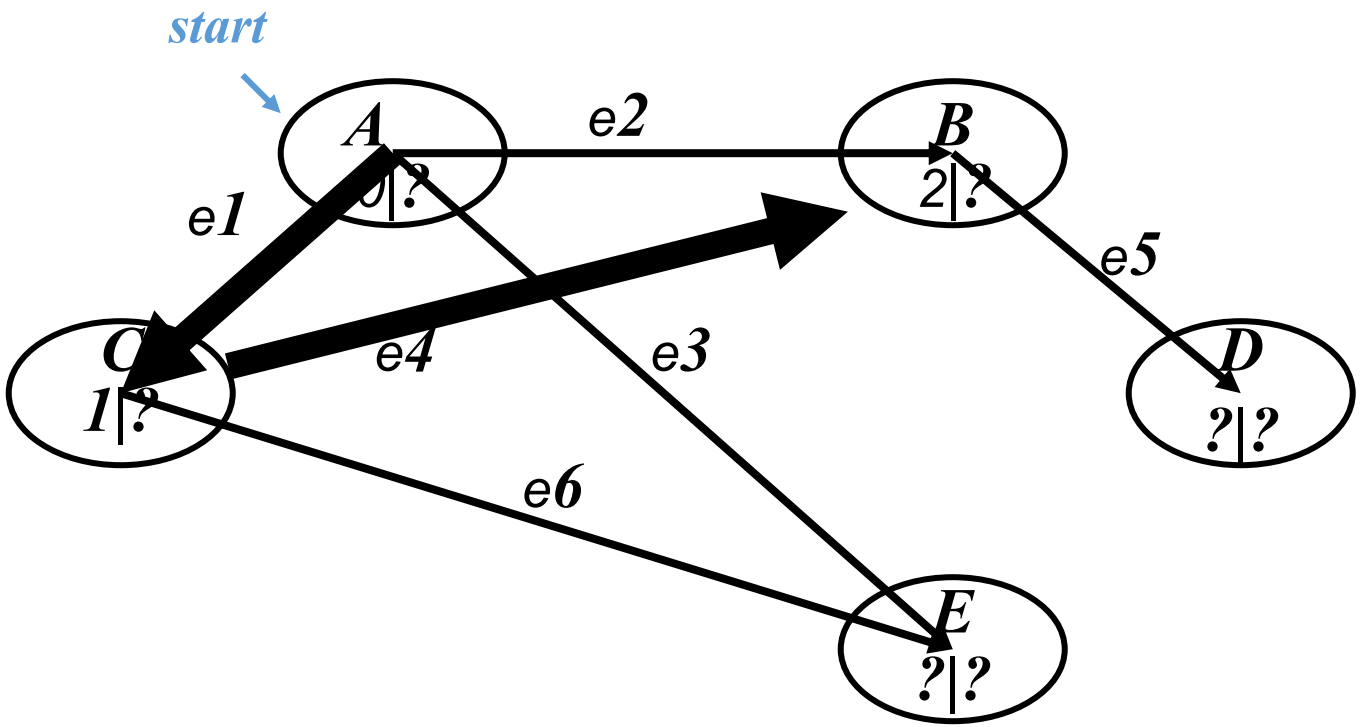
Example

Question: Consider the example below. What are the discovered and finished times for each node, and the topological order? (Note that the answers can be not unique. To make the answer unique, we assume that lower-numbered edges are explored first.)



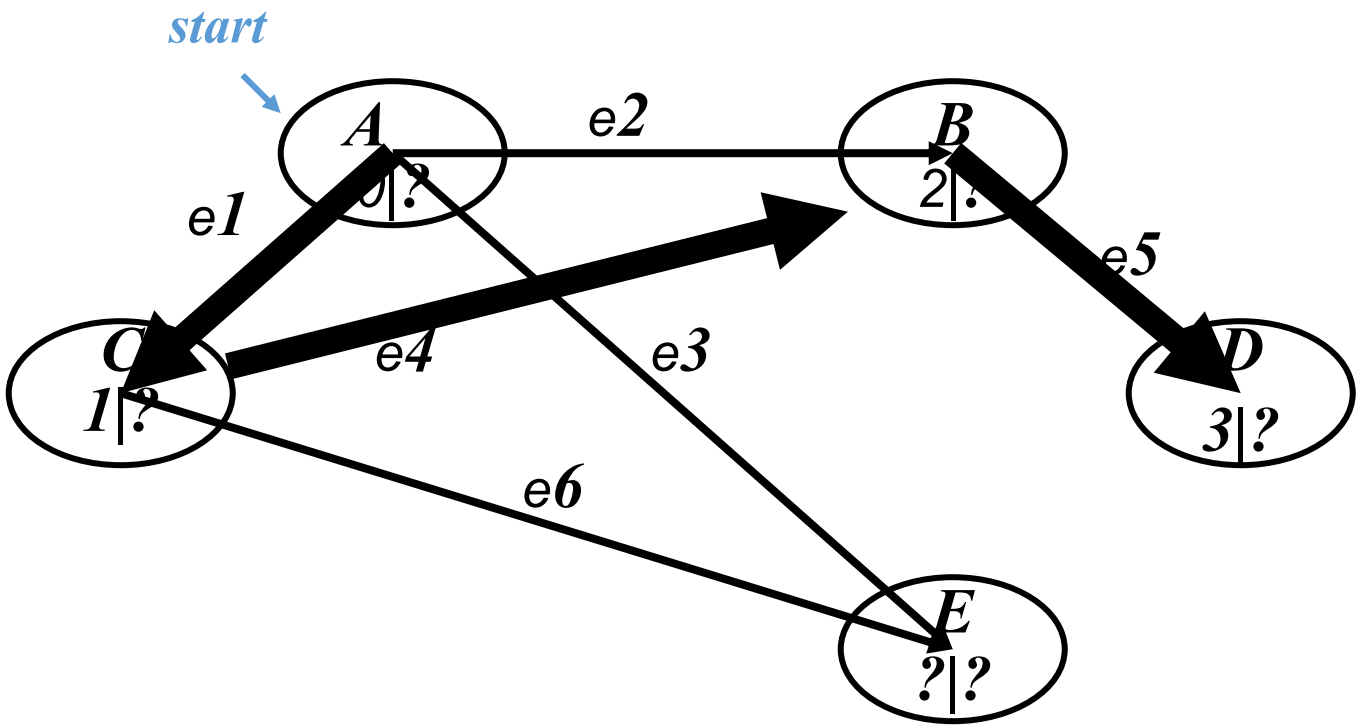
Example

Question: Consider the example below. What are the discovered and finished times for each node, and the topological order? (Note that the answers can be not unique. To make the answer unique, we assume that lower-numbered edges are explored first.)



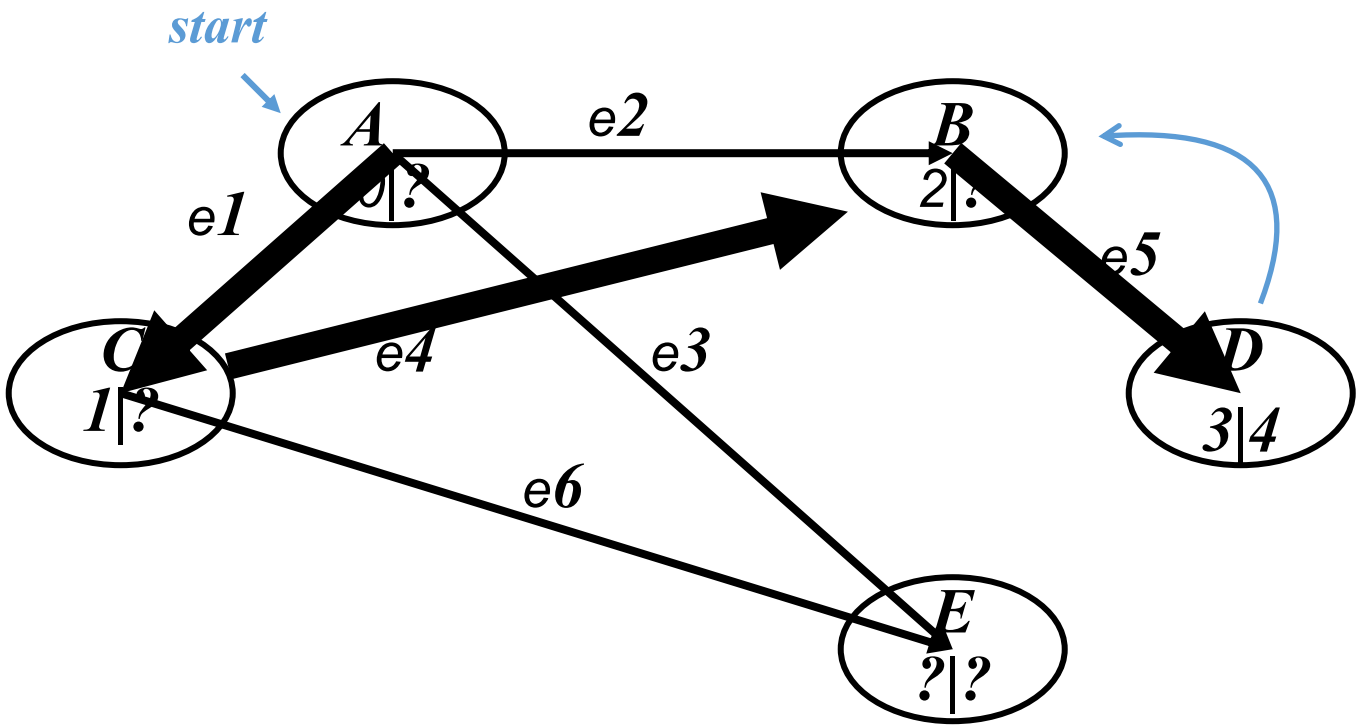
Example

Question: Consider the example below. What are the discovered and finished times for each node, and the topological order? (Note that the answers can be not unique. To make the answer unique, we assume that lower-numbered edges are explored first.)



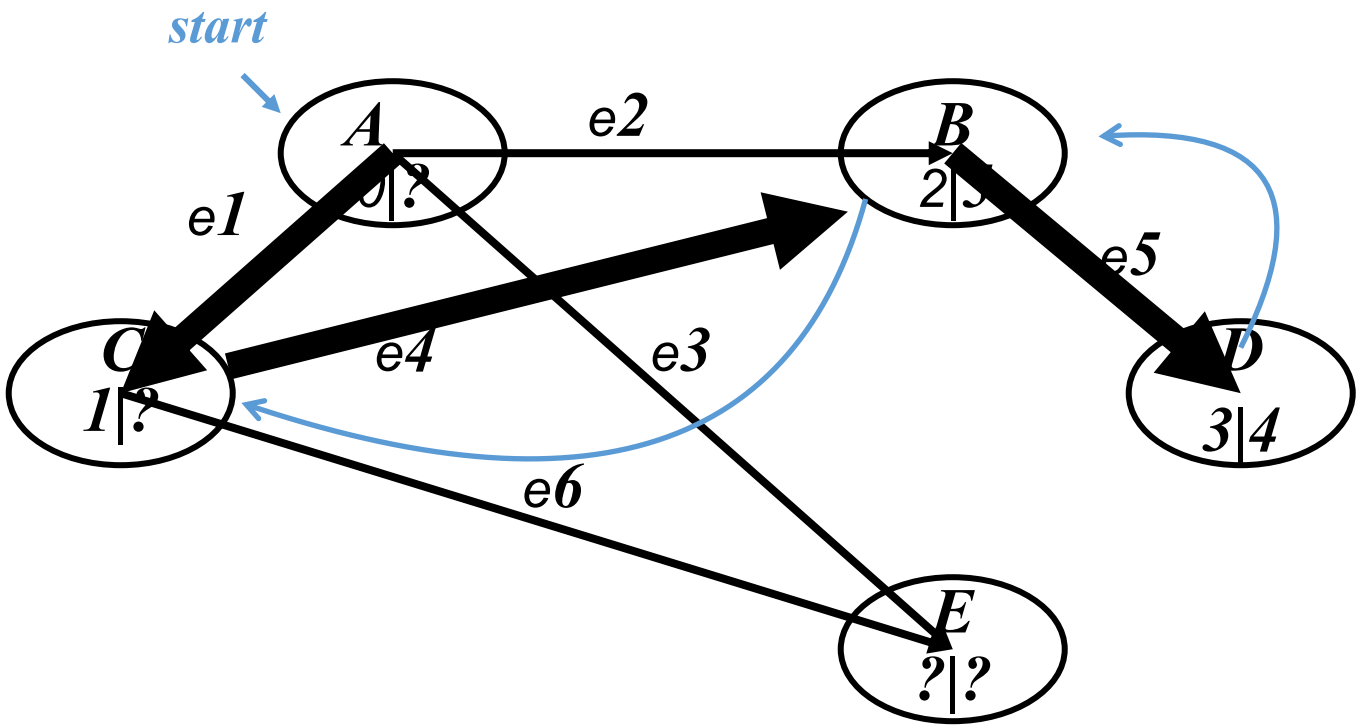
Example

Question: Consider the example below. What are the discovered and finished times for each node, and the topological order? (Note that the answers can be not unique. To make the answer unique, we assume that lower-numbered edges are explored first.)



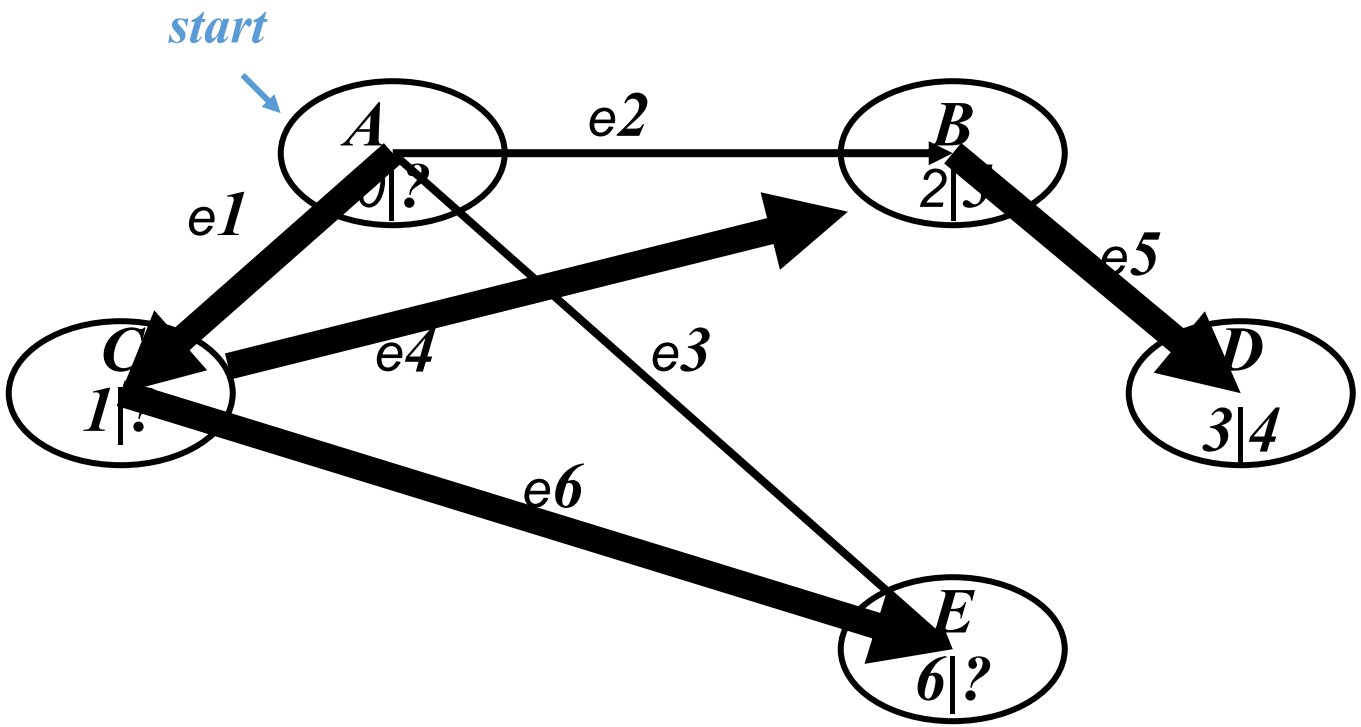
Example

Question: Consider the example below. What are the discovered and finished times for each node, and the topological order? (Note that the answers can be not unique. To make the answer unique, we assume that lower-numbered edges are explored first.)



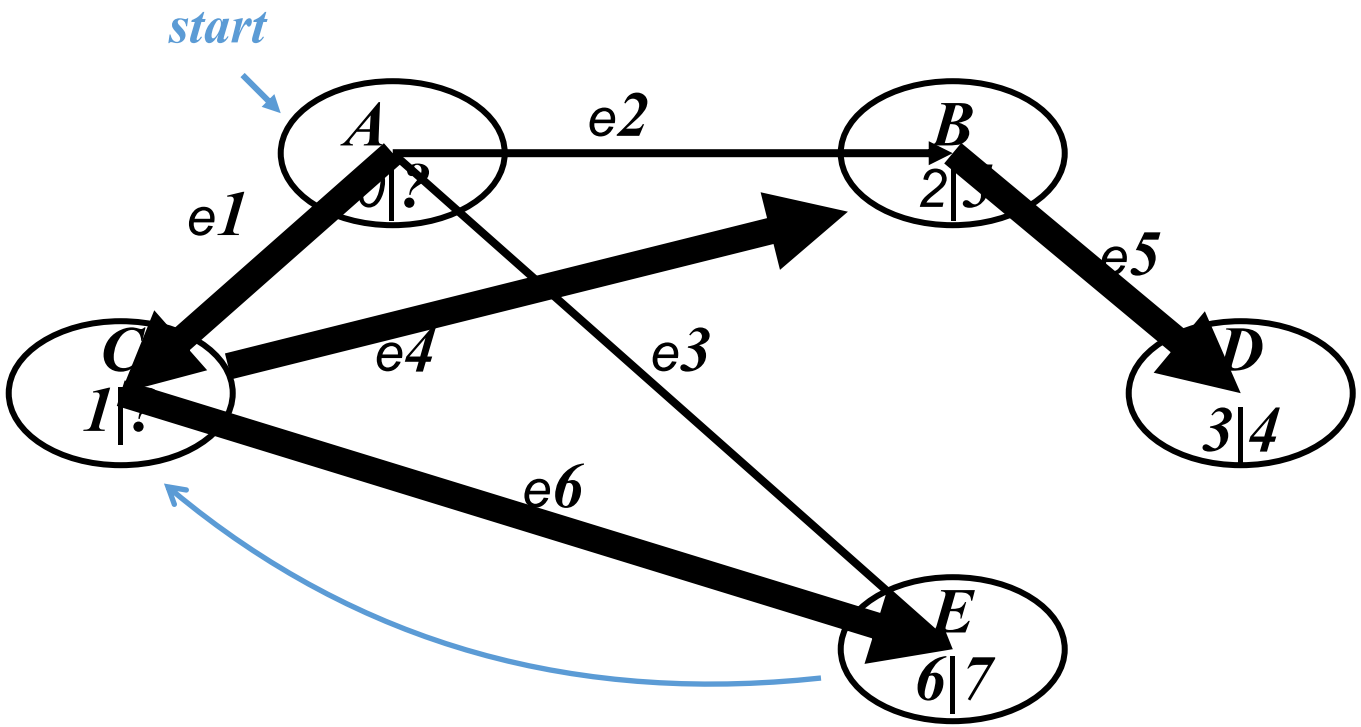
Example

Question: Consider the example below. What are the discovered and finished times for each node, and the topological order? (Note that the answers can be not unique. To make the answer unique, we assume that lower-numbered edges are explored first.)



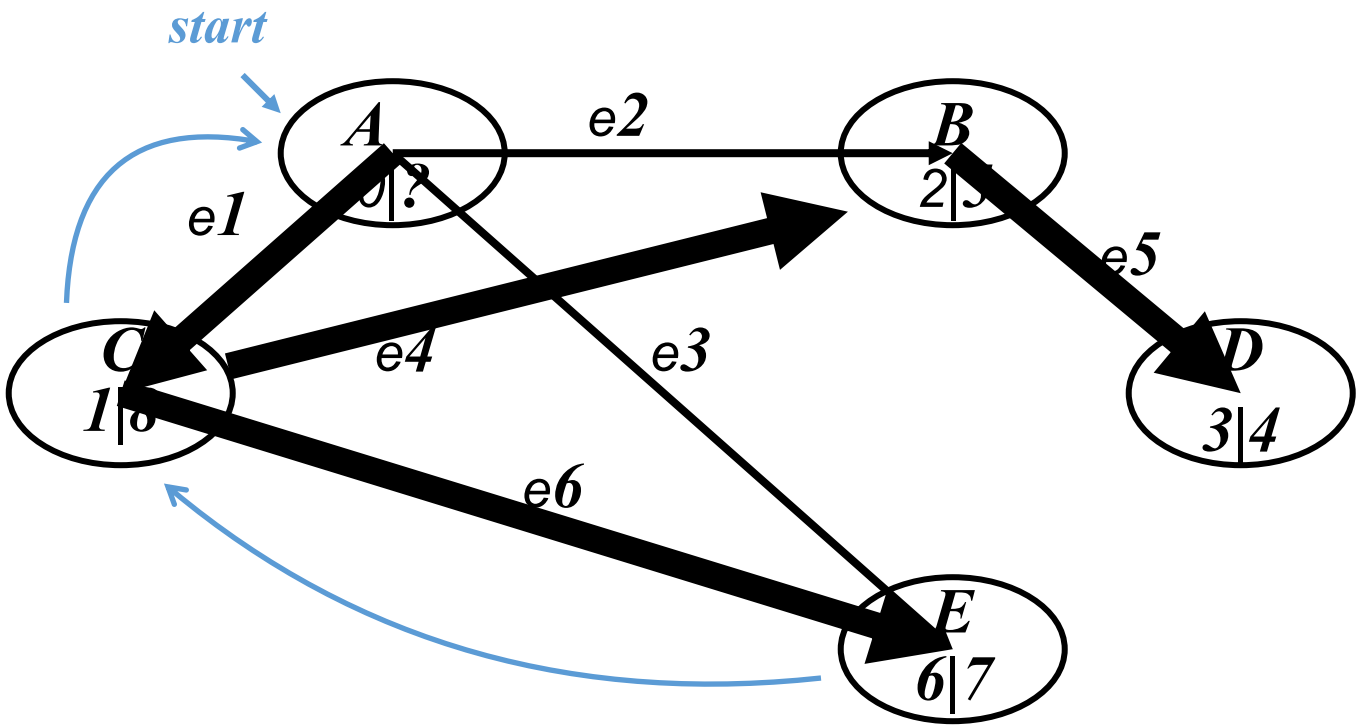
Example

Question: Consider the example below. What are the discovered and finished times for each node, and the topological order? (Note that the answers can be not unique. To make the answer unique, we assume that lower-numbered edges are explored first.)



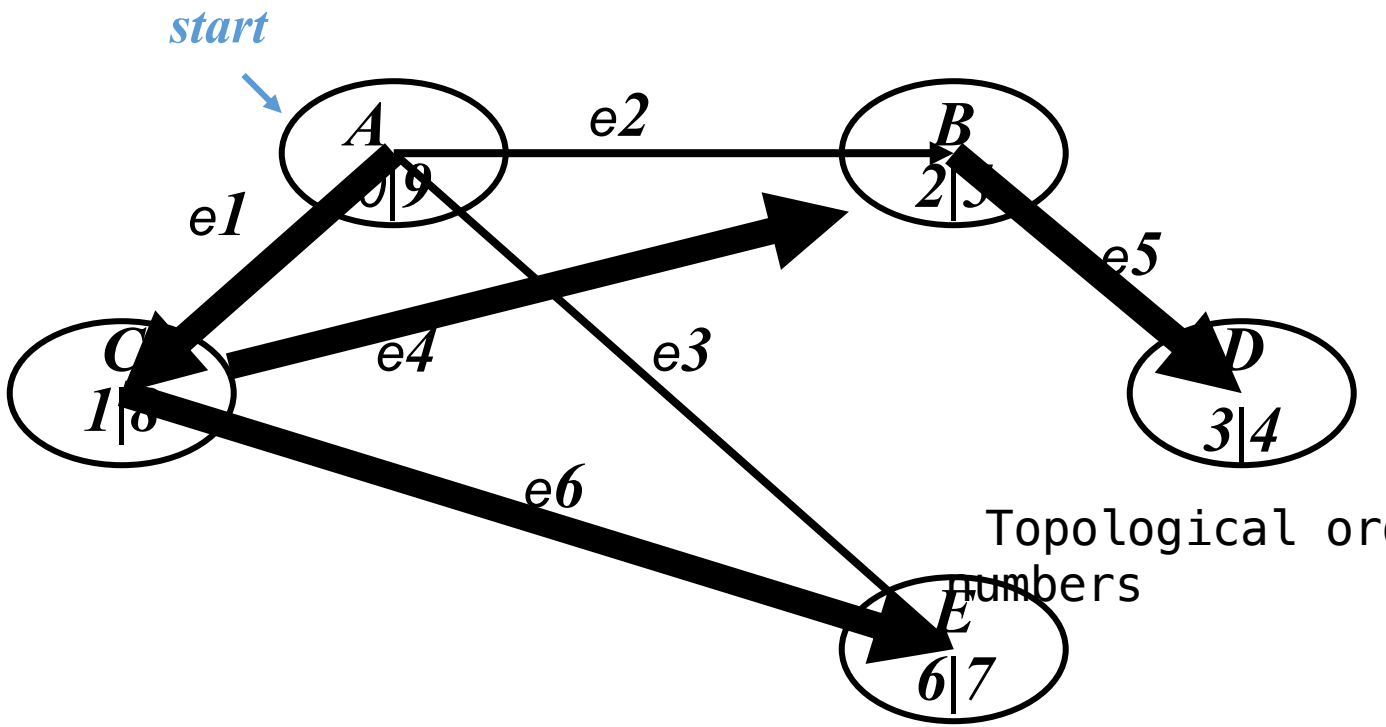
Example

Question: Consider the example below. What are the discovered and finished times for each node, and the topological order? (Note that the answers can be not unique. To make the answer unique, we assume that lower-numbered edges are explored first.)



Example

Question: Consider the example below. What are the discovered and finished times for each node, and the topological order? (Note that the answers can be not unique. To make the answer unique, we assume that lower-numbered edges are explored first.)



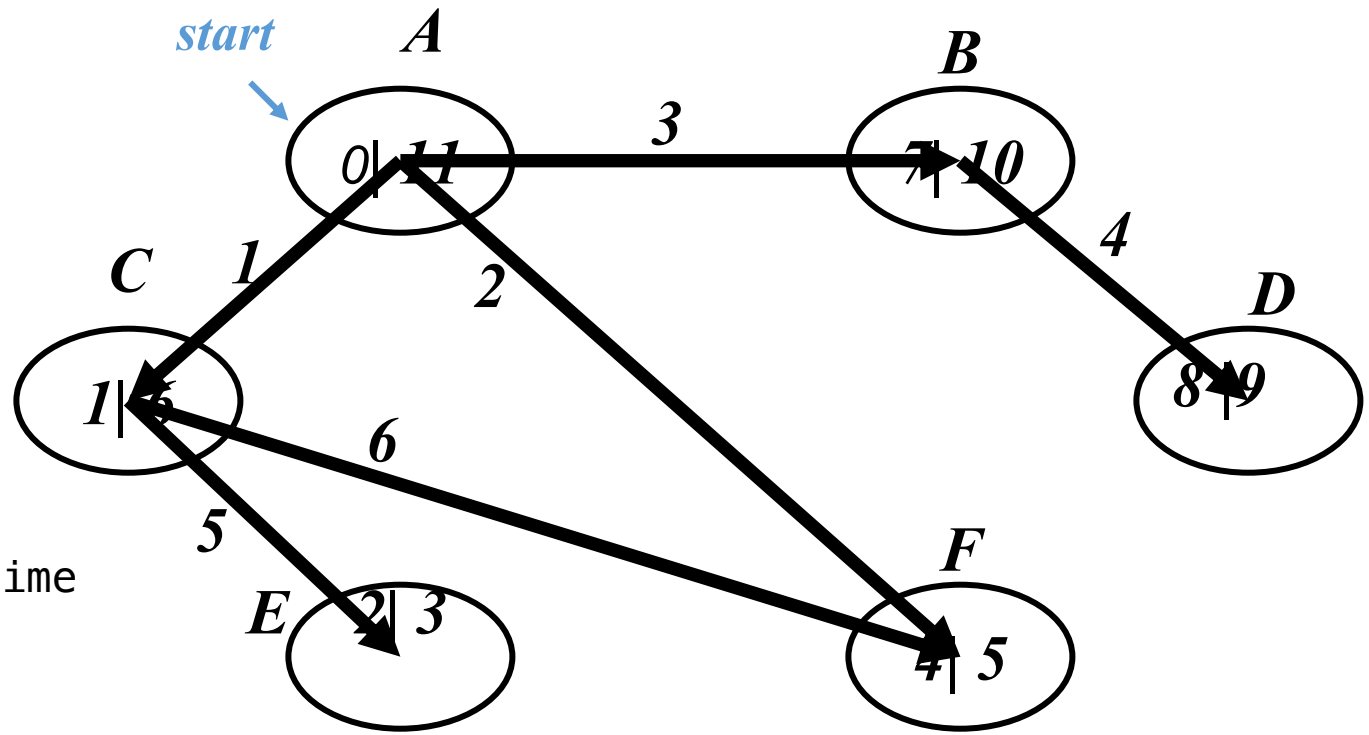
1) Node A? Answer(1pt): 0 | 9
2) Node B? Answer(1pt): 2 | 5
3) Node C? Answer(1pt): 1 | 8
4) Node D? Answer(1pt): 3 | 4
5) Node E? Answer(1pt): 6 | 7
6) What is the topological order?
Answer(1pt):

Topological ordering = the inverse order of the 2nd numbers

Example 2

Consider the example on the right.
What are the the discovered and finished time for each node, and the topological order?

Note that the answers can be not unique.
To make the answer unique, we assume that lower-numbered edges are explored first.



What are the the discovered and finished time for each node?

Node A (1pt) ___0___ | ____11_____
Node B (1pt) ___7___ | ____10_____
Node C (1pt) ___1___ | _____6____
Node D (1pt) ___8___ | _____9____
Node E (1pt) ___2___ | _____3____
Node F (1pt) ___4___ | _____5____

What is the topological oder? ABDCFE