# Basic Graph Algorithms

Gianmaria Silvello

Department of Information Engineering
University of Padua
silvello@dei.unipd.it
http://www.dei.unipd.it/~silvello/

# Outline

- Graph Representation
- Breadth-First Search
- Depth-First Search
- Directed Acyclic Graph

Reference: Chapter 22 of CLRS

Reference: Chapter 14 of Goodrich, Tamassia and Goldwasser

# Graphs

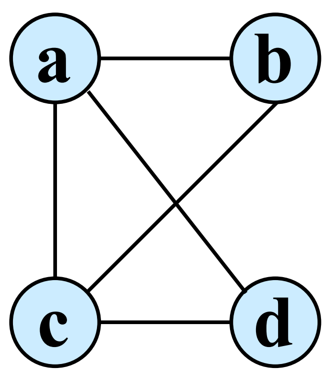
- Graph $G = (V, E)$
  - $V$ = set of vertices
  - $E$ = set of edges
- Types of graphs
  - *Undirected*: edge $(u, v) = (v, u)$; for all $v$, $(v, v) \notin E$ (No self loops)
  - *Directed*: $(u, v)$ is edge from $u$ to $v$, denoted as $u \rightarrow v$. Self loops are allowed
  - *Weighted*: each edge has an associated weight, given by a weight function $w : E \rightarrow \mathbf{R}$
  - *Dense*: $|E| \approx |V|^2$
  - *Sparse*: $|E| << |V|^2$
- $|E| = O(|V|^2)$

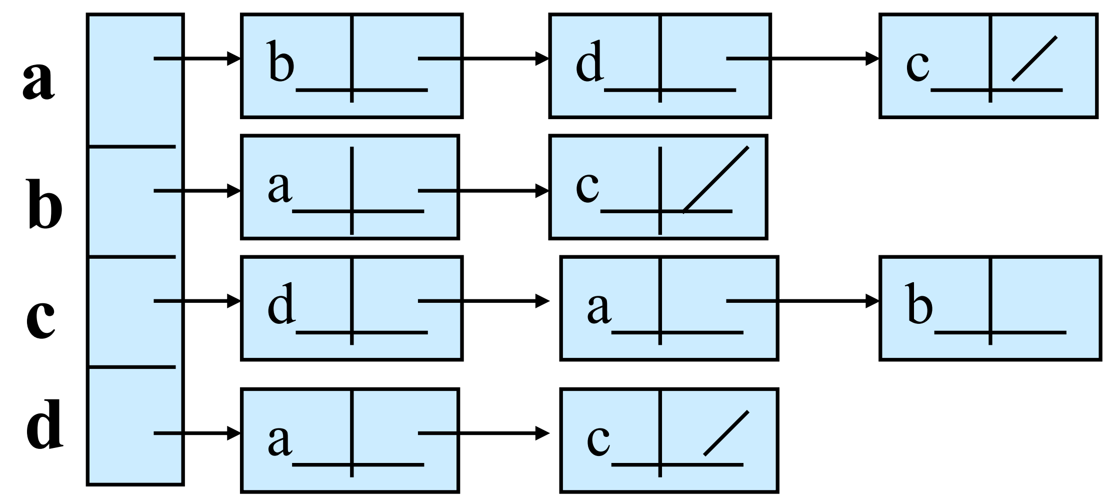
# Graphs

- If $(u, v) \in E$, then vertex $v$ is *adjacent* to vertex $u$.
- *Adjacency relationship* is:
  - Symmetric if G is undirected
  - Not necessarily so if G is directed
- If G is *connected*
  - There is a path between <u>every pair of vertices</u>
  - $|E| \geq |V| - 1$
  - If $|E| = |V| - 1$, then G is a tree

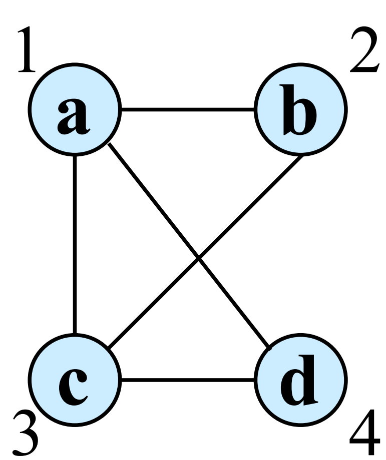
# Graph Representation

- Two standard ways
  - Adjacency Lists



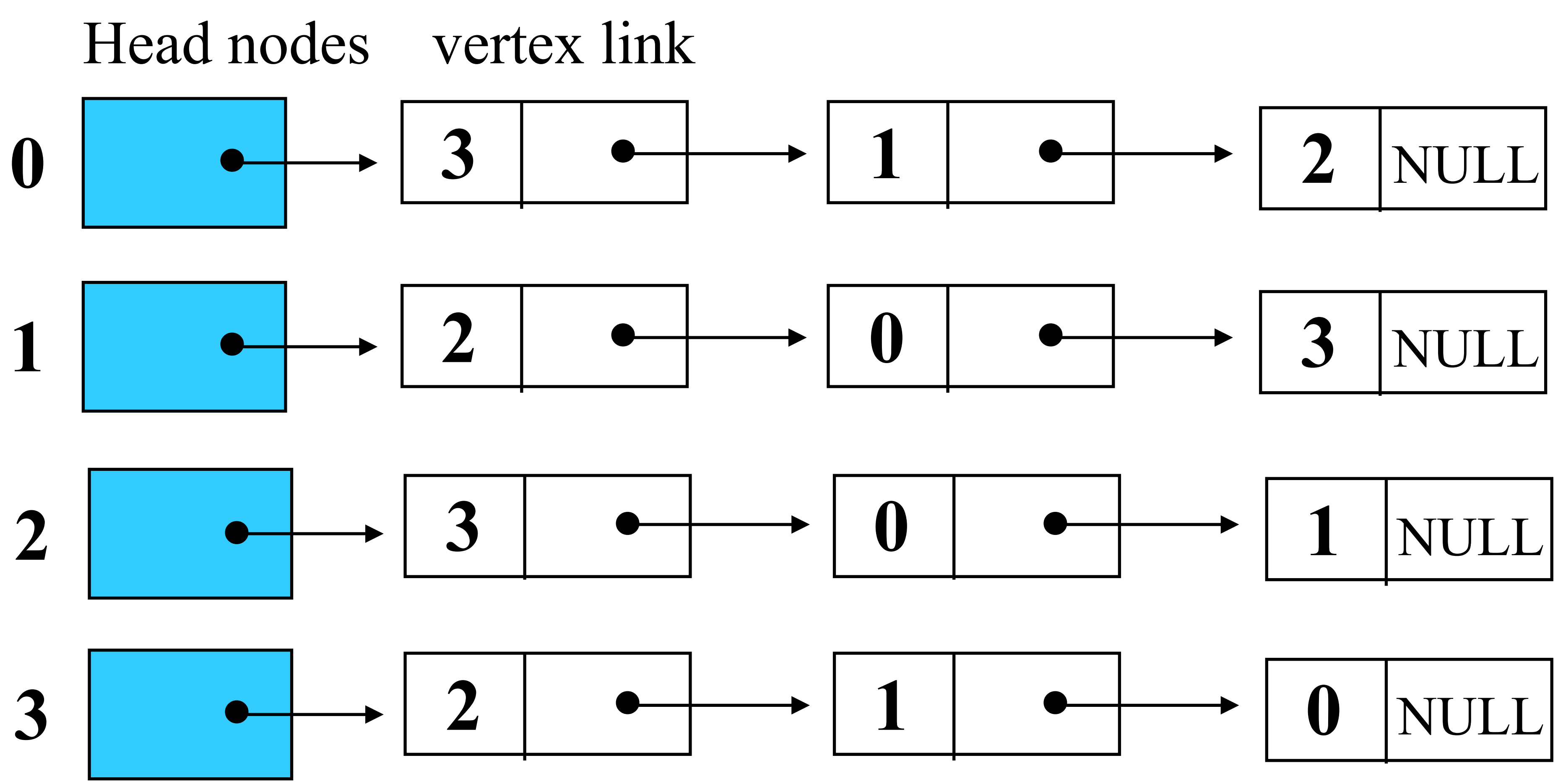

  - Adjacency Matrix

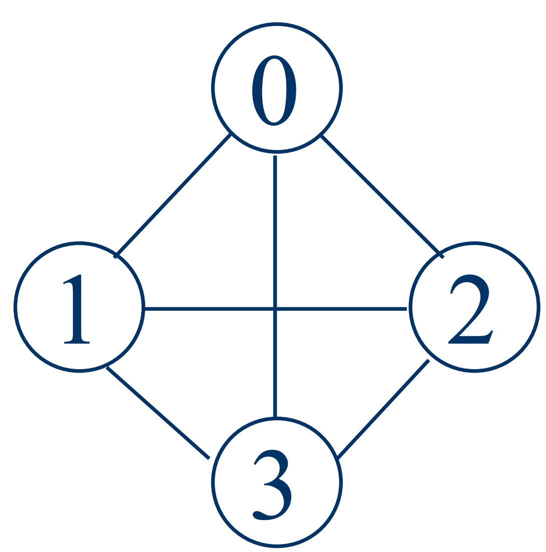



| | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | 0 | 1 | 1 | 1 |
| 2 | 1 | 0 | 1 | 0 |
| 3 | 1 | 1 | 0 | 1 |
| 4 | 1 | 0 | 1 | 0 |

# Adjacency List Representation

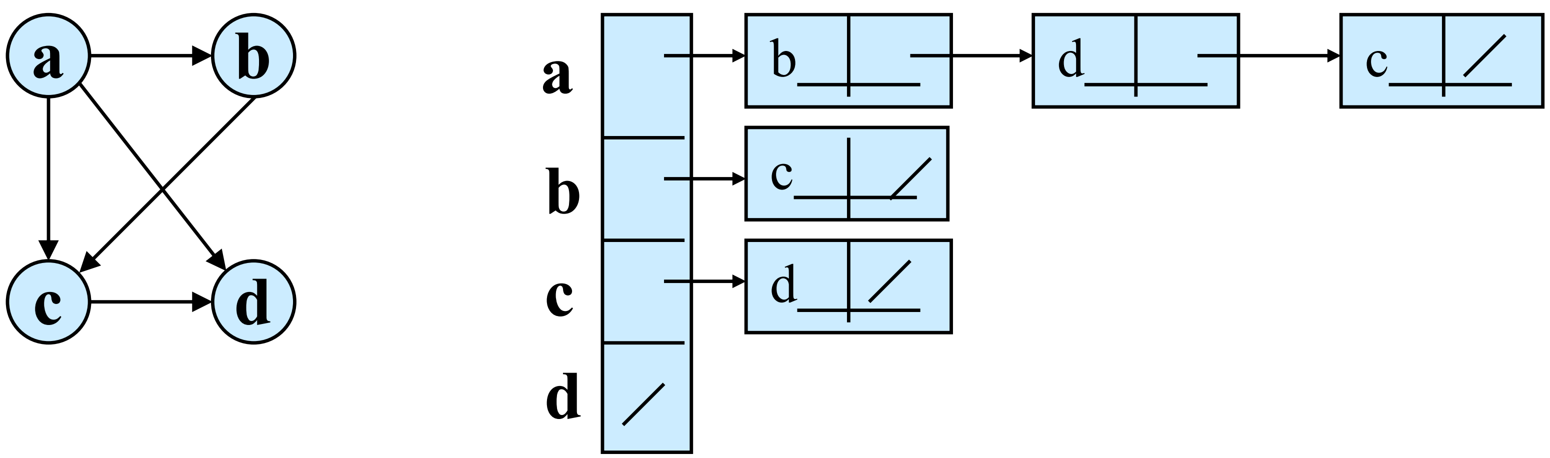
Order is of no significance.



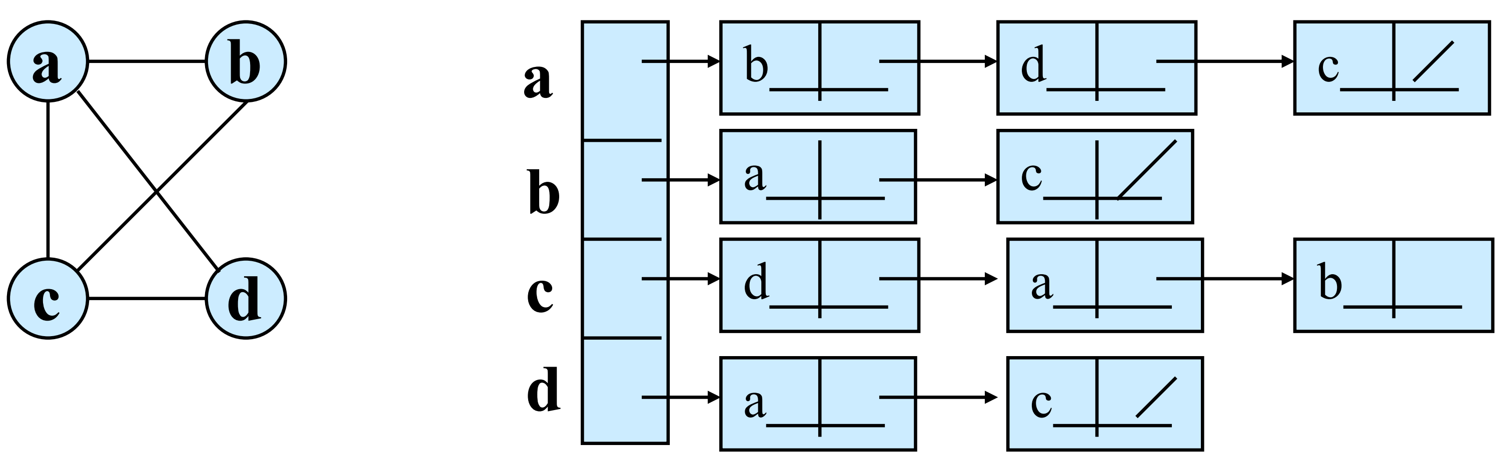



# Adjacency Lists

- Consists of an array *Adj* of |V| lists
- One list per vertex
- For u ∊ V, Adj[u] consists of all vertices adjacent to u



Directed Graph



Undirected Graph

# Adjacency Lists: Storage Requirements

- For directed graphs:
  - Sum of lengths of all adj. lists is $\sum_{v \in V}$ [out-degree(v)] = |E|
  - Total storage: $\Theta(|V|+|E|)$

out-degree(v) = number of edges outgoing from a node v

- For undirected graphs:
  - Sum of lengths of all adj. lists is $\sum_{v \in V}$ [degree(v)] = 2|E|
  - Total storage: $\Theta(|V|+|E|)$

degree(v) = number of edges incident on a node v

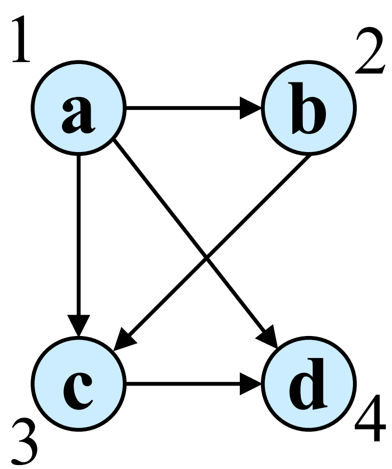
# Adjacency Lists: Pros and Cons

- Pros
    - Space-efficient, when a graph is sparse
    - Can be modified to support many graph variants

- Cons
    - Determining if an edge $(u,v) \in G$ is not efficient
    - Have to search in u's adjacency list. $\Theta$(degree(u))-time
    - $\Theta(V)$ in the worst case

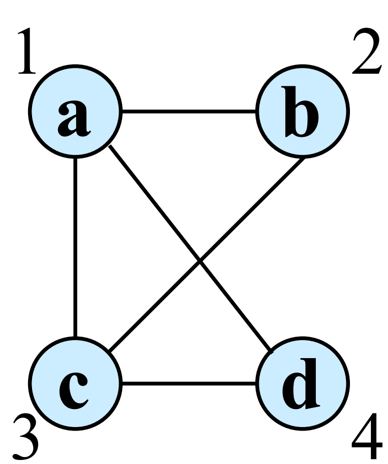
# Adjacency Matrix

- |V| x |V| matrix A
- Number vertices from 1 to |V| in some arbitrary manner
- A is then given by $A[i,j] = a_{ij} = \begin{cases} 1 & \text{if } (i,j) \in E \\ 0 & \text{otherwise} \end{cases}$




| | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | 0 | 1 | 1 | 1 |
| 2 | 0 | 0 | 1 | 0 |
| 3 | 0 | 0 | 0 | 1 |
| 4 | 0 | 0 | 0 | 0 |

Directed Graph




| | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | 0 | 1 | 1 | 1 |
| 2 | 1 | 0 | 1 | 0 |
| 3 | 1 | 1 | 0 | 1 |
| 4 | 1 | 0 | 1 | 0 |

Undirected Graph

# Space and Time Requirements

- Space: $\Theta(V^2)$
  - Not memory efficient for large graphs

- Time: to list all vertices adjacent to *u*: $\Theta(V)$.
- Time: to determine if $(u, v) \in E$: $\Theta(1)$.
- Can store weights instead of bits for weighted graph

# Graph: Nomenclature

- A *path* of length k from a vertex u to v in a graph G=(V,E) is a sequence p=<u, $u_1$, $u_2$, … , v> where (u,$u_1$), ($u_1$,$u_2$) and so on belongs to E; |p| = k
- In a graph G, we say that a node v is *reachable* from u, if there exists a path in G from u to v.
- A path is *simple* if all vertices in the path are distinct
- An undirected graph G is *connected* if there exists a path between every pair of vertices
- A graph G is *strongly connected* if every two vertices are reachable from each other

# Search a Graph

- Search a graph → Systematically follow the edges of a graph to visit the vertices of the graph → Graph traversal
- Used to discover the structure of a graph
- Basic graph-searching algorithms are
  - Breadth-first Search (**BFS**)
  - Depth-first Search (**DFS**)

# Breadth-First Search

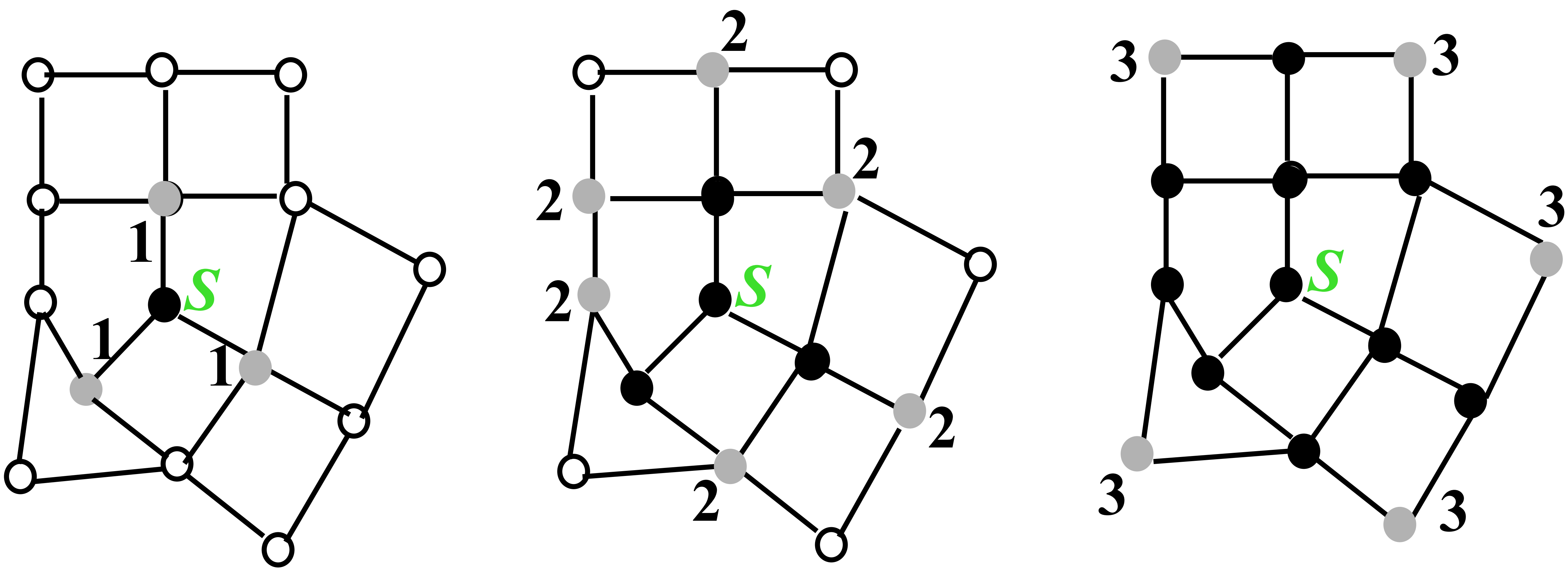
- Input:
  - Graph G = (V, E), either directed or undirected, and *source vertex* s ∈ V

- Output:
  - d[v] = distance (smallest # of edges, or shortest path) from *s* to *v*, for all $v \in V$. d[v] = ∞ if *v* is not reachable from *s*.
  - $\pi$[v] = *u* such that (*u*,*v*) is last edge on <u>shortest path</u> $s \leadsto v$
    - u is v's predecessor
  - Builds breadth-first tree with root *s* that contains all reachable vertices.

# Breadth-First Search

- Expands the frontier between discovered and undiscovered vertices uniformly across the breadth of the frontier
  - A vertex is "*discovered*" the first time it is encountered during the search
  - A vertex is "*finished*" if all vertices adjacent to it have been discovered
- It colours the vertices to keep track of progress
  - White → Undiscovered
  - Grey → Discovered but not finished
  - Black → Finished

# Breadth-First Search




● Finished ● Discovered ○ Undiscovered

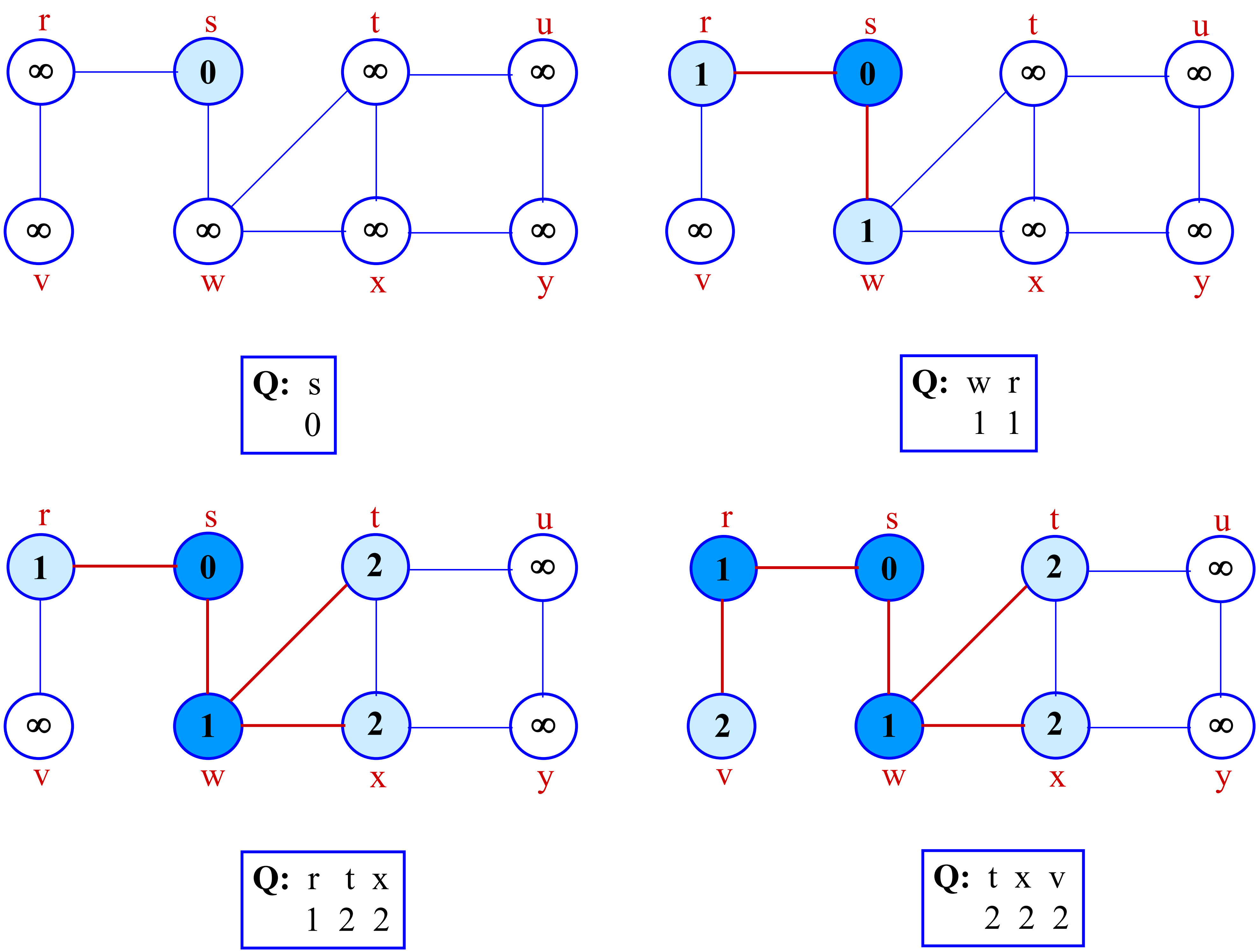
# BFS: Pseudo-code

```
BFS(G,s)
for each vertex u in V[G]-{s} do
  color[u] ← white
      d[u] ← ∞
      π[u] ← nil
color[s] ← grey
d[s] ← 0
π[s] ← nil
initialise the queue Q
Q.enqueue(s)
 while Q ≠ Ø do
   u ← Q.dequeue()
   for each v in Adj[u] do
      if color[v] == white then
         color[v] ← grey
             d[v] ← d[u] + 1
             π[v] ← u
             Q.enqueue(v)
   color[u] ← black
```

Adj: Adjacency List
Q: a queue of discovered vertices
color[v]: color of v
d[v]: distance from s to v
$\pi$[u]: predecessor of v

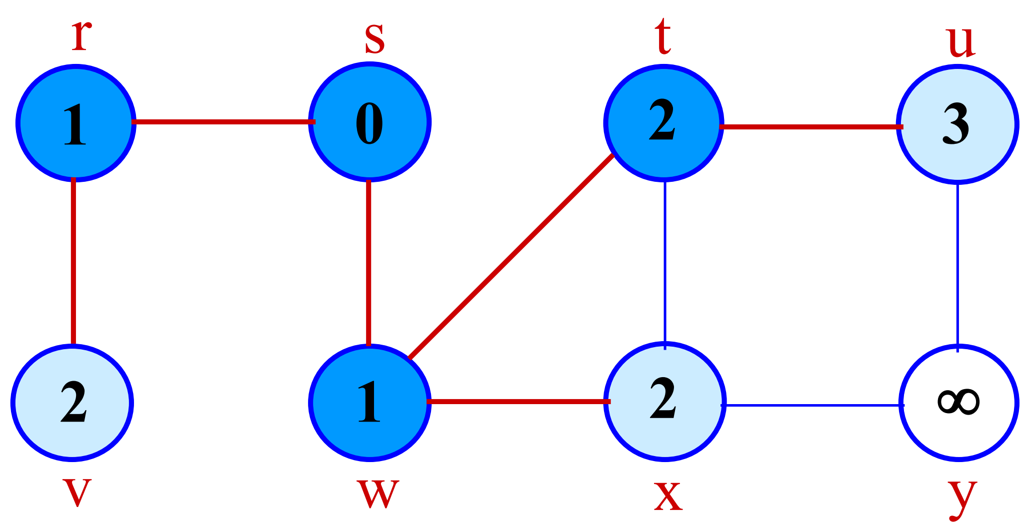
white: undiscovered
grey: discovered
black: finished

# BFS: Example

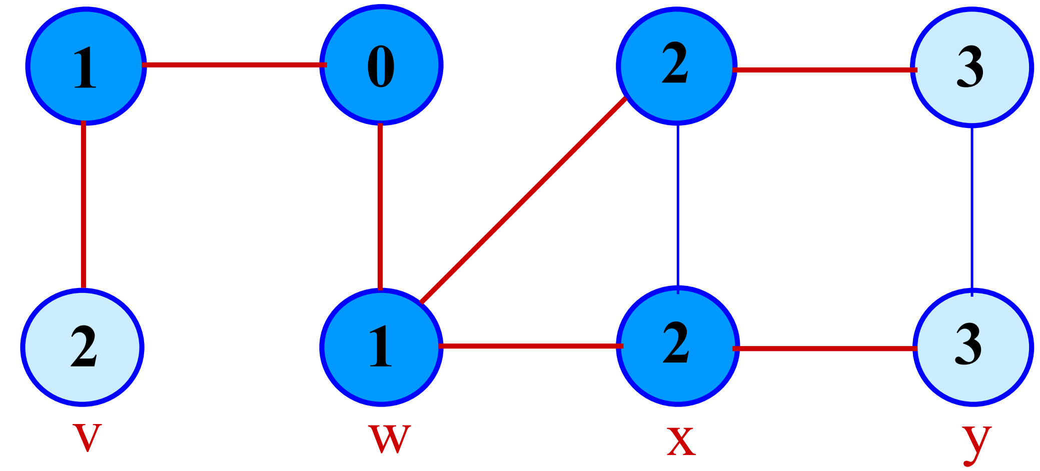


r s t u
∞ 0 ∞ ∞
∞ ∞ ∞ ∞
v w x y
Q: s
0
r s t u
1 0 ∞ ∞
∞ 1 ∞ ∞
v w x y
Q: w r
1 1
r s t u
1 0 2 ∞
∞ 1 2 ∞
v w x y
Q: r t x
1 2 2
r s t u
1 0 2 ∞
2 1 2 ∞
v w x y
Q: t x v
2 2 2

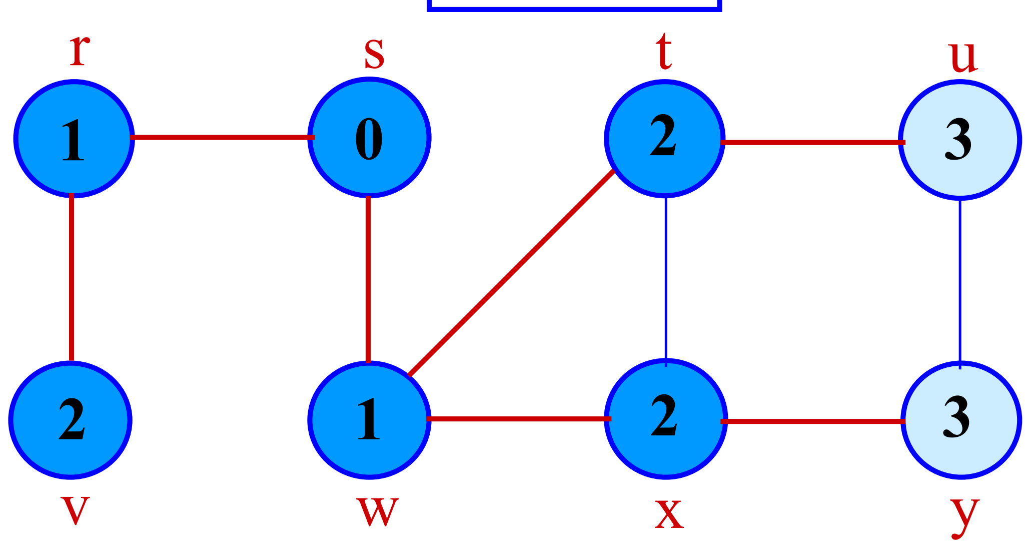
# BFS: Example



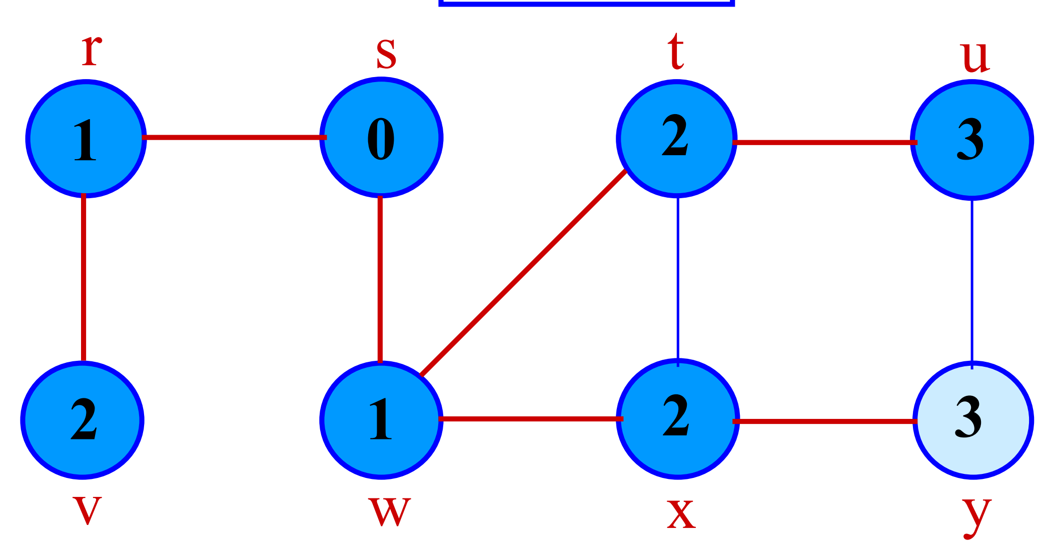

**Q:** x v u
2 2 3




**Q:** v u y
2 3 3




**Q:** u y
3 3

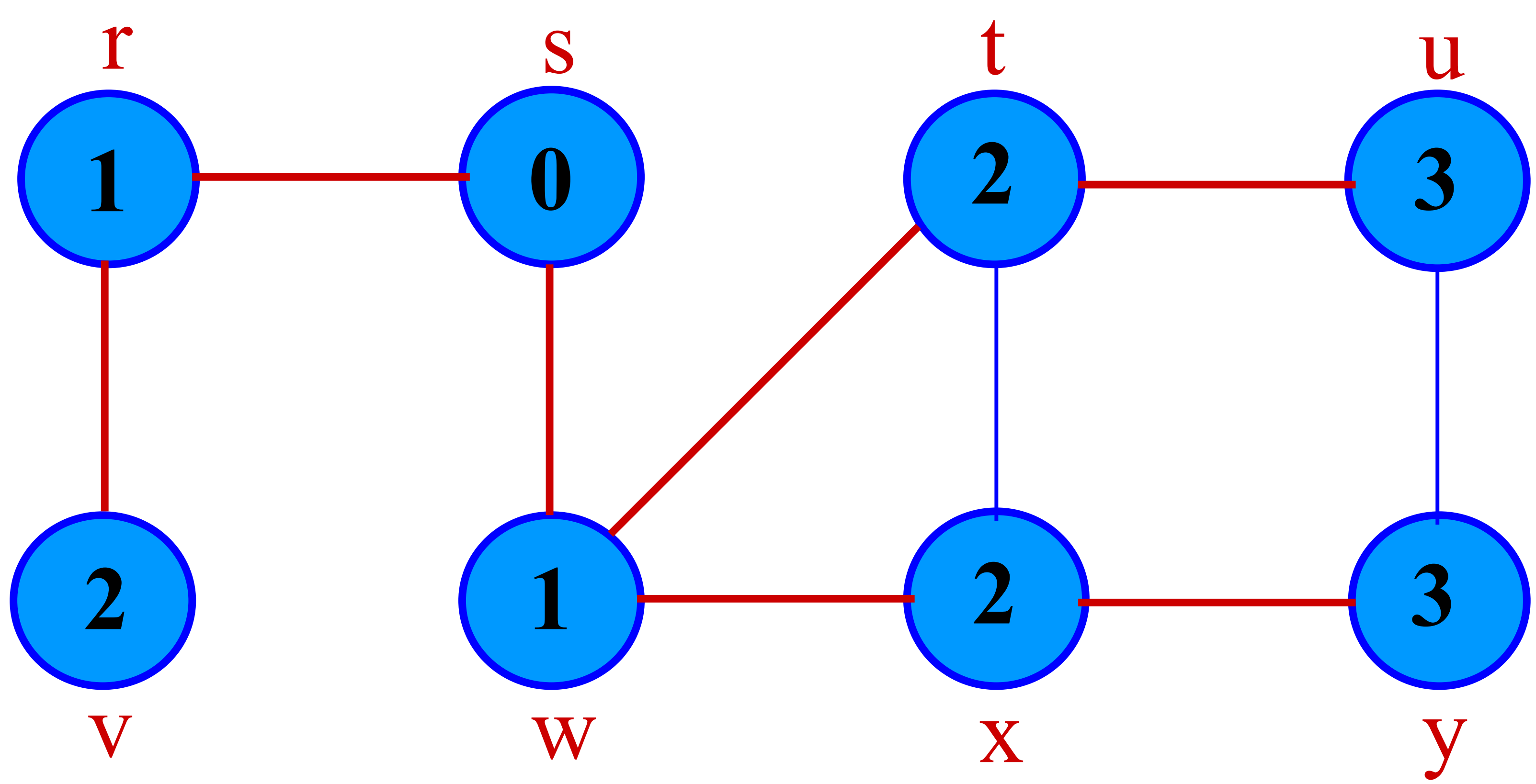



**Q:** y
3

# Breadth-First Tree

- For a graph $G = (V, E)$ with source $s$, the predecessor subgraph of $G$ is $G\pi = (V\pi, E\pi)$ where
  - $V\pi = \{v \in V : \pi[v] \neq NIL\} \cup \{s\}$
  - $E\pi = \{(\pi[v], v) \in E : v \in V\pi - \{s\}\}$
- The predecessor subgraph $G\pi$ is a breadth-first tree if:
  - $V\pi$ consists of the vertices reachable from $s$ and for all $v \in V\pi$, there is a unique simple path from $s$ to $v$ in $G\pi$ that is also a shortest path from $s$ to $v$ in $G$
  - The edges in $E\pi$ are called tree edges.
- $|E\pi| = |V\pi| - 1$

# BFS Tree




**Q:** $\varnothing$

# BFS: Analysis

```
BFS(G,s)
for each vertex u in V[G]-{s} do
    color[u] ← white
        d[u] ← ∞
        p[u] ← nil
color[s] ← gray
d[s] ← 0
p[s] ← nil
initialise the queue Q
Q.enqueue(s)
while Q ≠ Ø do
    u ← Q.dequeue()
    for each v in Adj[u] do
        if color[v] == white then
            color[v] ← gray
                d[v] ← d[u] + 1
                p[v] ← u
                Q.enqueue(v)
    color[u] ← black
```

Initialization takes O(V).

Traversal Loop (while)

After initialization, each vertex is enqueued and dequeued at most once, and each operation takes O(1). So, total time for queuing is O(V).

The adjacency list of each vertex is scanned (for each loop) at most once. The sum of lengths of all adjacency lists is O(E).

Summing up over all vertices

Total running time of BFS is O(V+E), linear in the size of the adjacency list representation of graph.

# Depth-First Search

- Explore edges out of the most recently discovered vertex v.
- When all edges of v have been explored, backtrack to explore other edges leaving the vertex from which v was discovered (its predecessor).
- “Search as deep as possible first.”
- Continue until all vertices reachable from the original source are discovered.
- If any undiscovered vertices remain, then one of them is chosen as a new source and search is repeated from that source.

# DFS

- Input: G = (V, E), directed or undirected. No source vertex given
- Output:
  - 2 timestamps on each vertex. Integers between 1 and $2|V|$.
  - $d[v]$ = discovery time (v turns from white to gray)
  - $f[v]$ = finishing time (v turns from gray to black)
  - $\pi[v]$ : predecessor of v = u, such that v was discovered during the scan of u's adjacency list.
- Uses the same colouring scheme for vertices as BFS

# DFS

```
DFS(G)
for each vertex u in V[G]do
     color[u] ← white
          π[u] ← nil
time ← 0
for each vertex u∈V[G] do
     if color[u]==white then
          DFS-Visit(u)
```

*time* is a global variable

```
DFS-Visit(u)
color[u]← grey
time ← time+1
d[u] ← time
for each v∈Adj[u] do
     if color[v]==white then
          π[v] ← u
          DFS-Visit(v)
color[u] ← black
f[u] ← time
time ← time + 1
```

Loops take $\Theta$(V) time

DFS-Visit is called once for each white vertex v∈V when it's painted grey the first time.

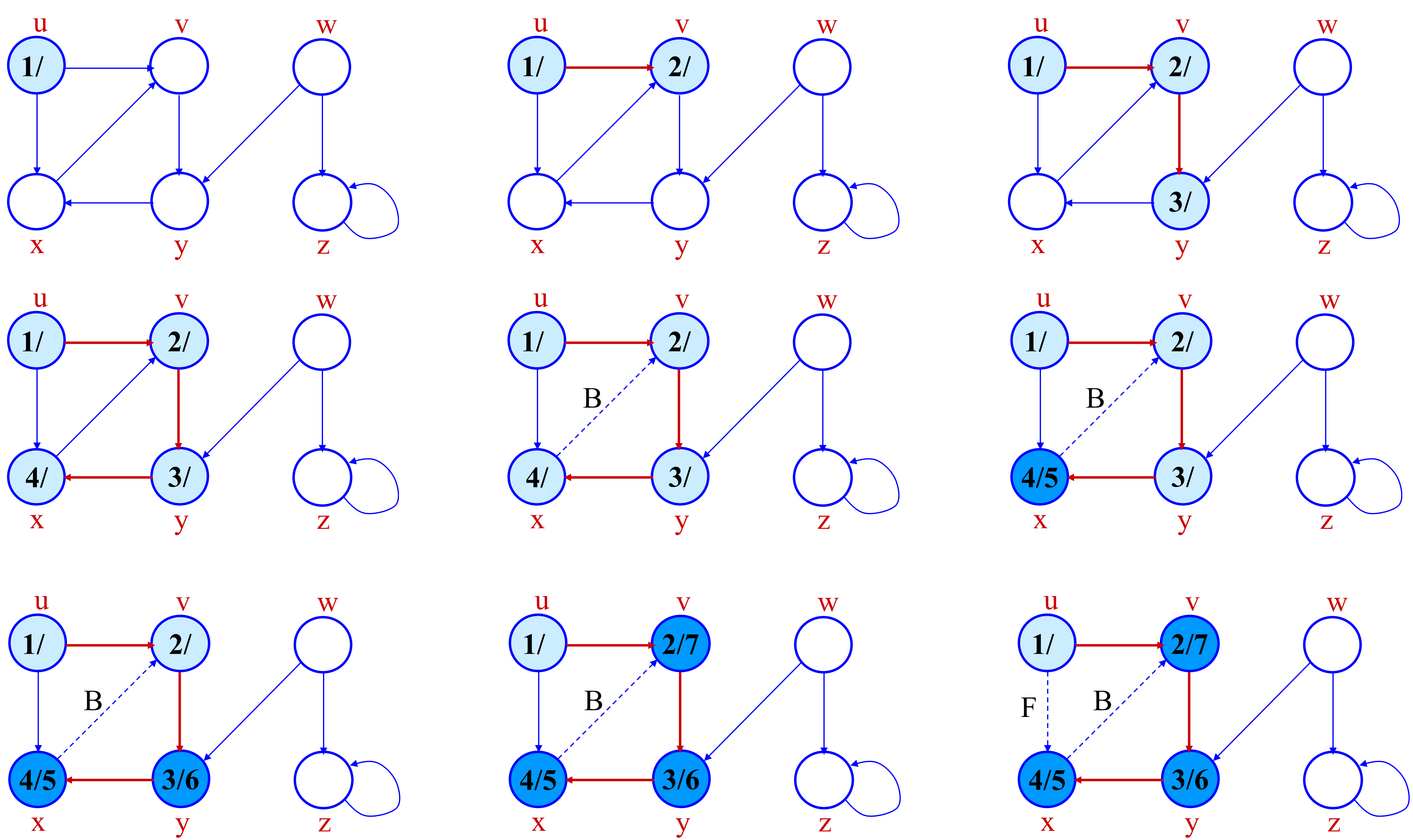
The total cost of executing DFS-Visit is $\sum_{v \in |V|} |Adj[v]| = \Theta(E)$

Total running time of DFS is $\Theta$(V+E).

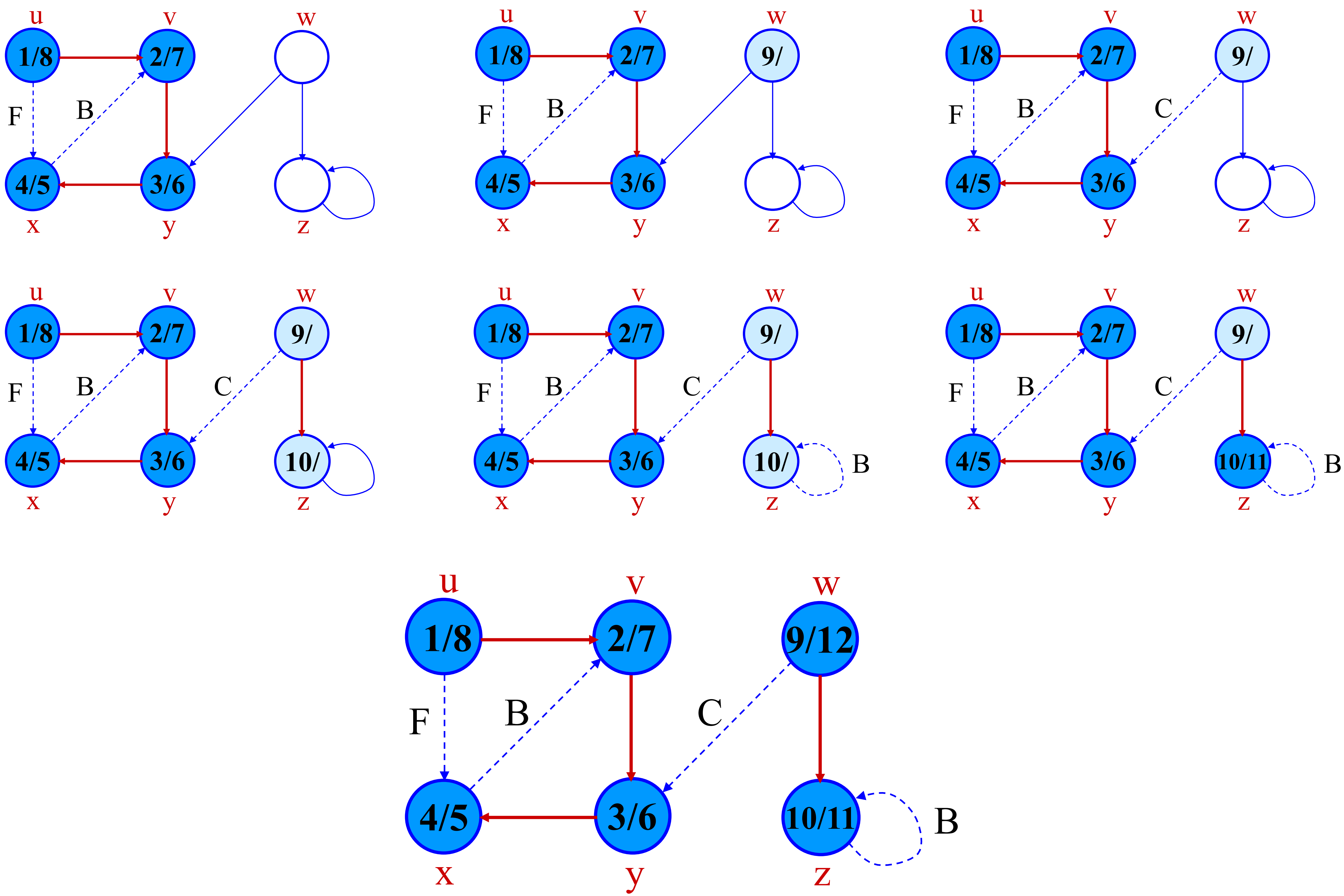
# Classification of edges

- *Tree edge*: in the depth-first forest.
  Found by exploring (u, v).
- *Back edge*: (u, v), where u is a descendant of v (in the depth-first tree).
- *Forward edge*: (u, v), where v is a descendant of u, but not a tree edge.
- *Cross edge*: any other edge. Can go between vertices in same depth-first tree or in different depth-first trees.

# DFS: Example


u v w
1/
x y z
u v w
1/ 2/
x y z
u v w
1/ 2/
3/
x y z
u v w
1/ 2/
4/ 3/
x y z
u v w
1/ 2/
B
4/ 3/
x y z
u v w
1/ 2/
B
4/5 3/
x y z
u v w
1/ 2/
B
4/5 3/6
x y z
u v w
1/ 2/7
B
4/5 3/6
x y z
u v w
1/ 2/7
F B
4/5 3/6
x y z

# DFS: Example


u v w
1/8 2/7
F B
4/5 3/6
x y z
u v w
1/8 2/7 9/
F B
4/5 3/6
x y z
u v w
1/8 2/7 9/
F B C
4/5 3/6
x y z
u v w
1/8 2/7 9/
F B C
4/5 3/6 10/
x y z
u v w
1/8 2/7 9/
F B C
4/5 3/6 10/ B
x y z
u v w
1/8 2/7 9/
F B C
4/5 3/6 10/11 B
x y z
u v w
1/8 2/7 9/12
F B C
4/5 3/6 10/11 B
x y z

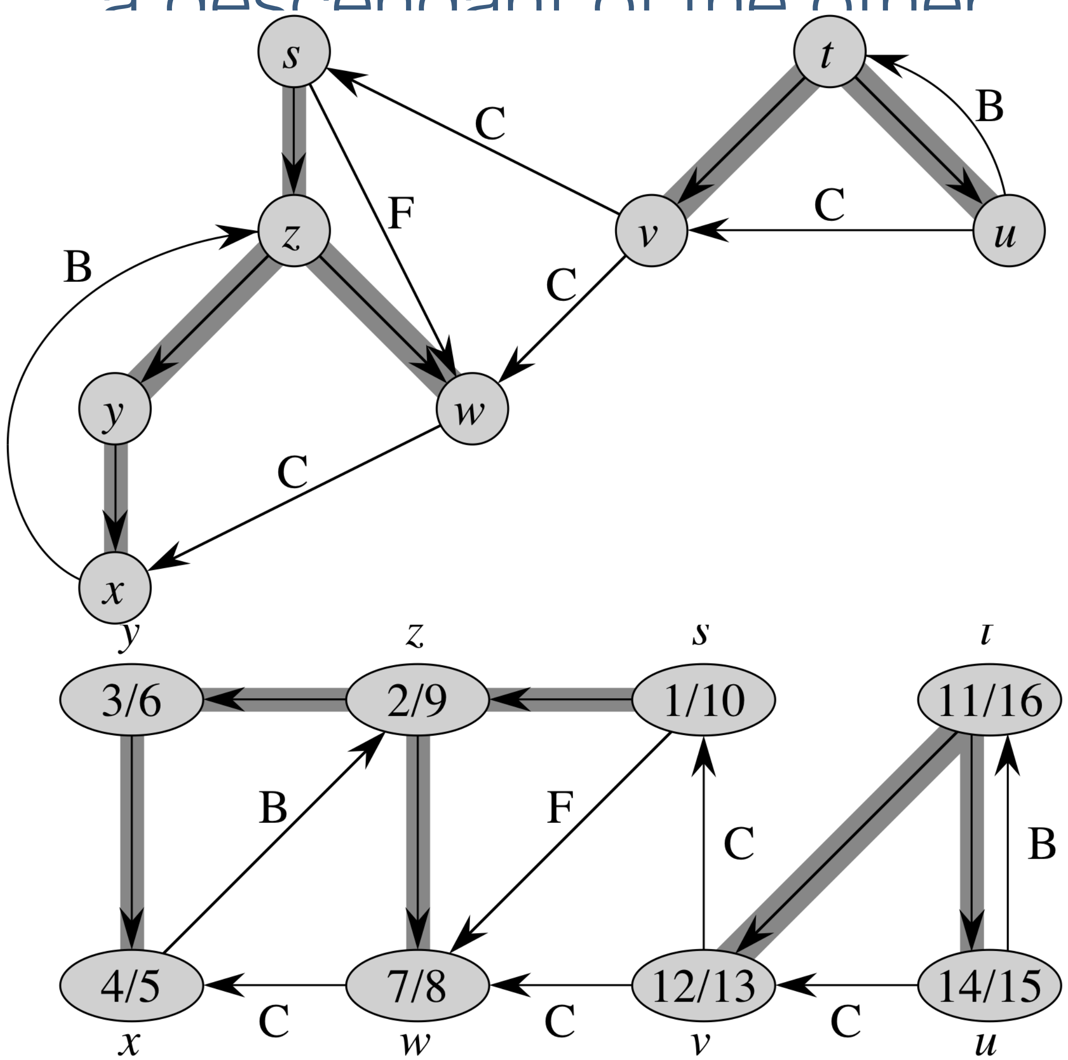
# DFS Trees

- Predecessor subgraph defined slightly different from that of BFS.
- The predecessor subgraph of DFS is $G\pi = (V\pi, E\pi)$ where $E\pi = \{(\pi[v], v) : v \in V \text{ and } \pi[v] \neq NIL\}$.
- The predecessor subgraph $G\pi$ forms a depth-first forest composed of several depth-first trees.
- The edges in $E\pi$ are called tree edges

# Parentheses Theorem

- For all u, v, exactly one of the following holds:

  1. if *d[u] < f[u] <d[v] < f[v]* or *d[v] < f[v] < d[u] < f [u]* then neither *u* nor *v* is a descendant of the other




*v* is a descendant of *u*

*u* is a descendant of *v*