《数据结构与算法》课程组

重庆大学计算机学院

Data Structures & Algorithms

15 ELEMENTARY GRAPH ALGORITHMS

outlines

15.1 Basic notions of graphs

15.2 Standard graph-traversal algorithms

15.3 Topological sorting

15.1 Basic notions of graphs

Definitions

- *Graph* $G = (V, E)$
  - V = set of vertices
  - E = set of edges $\subseteq (V \times V)$
- $|E| = O(|V|^2)$

Types of Graphs

- Undirected: edge $(u, v) = (v, u)$;
  for all v, $(v, v) \notin E$ (No self-loop)

- Directed: (u, v) is edge from u to v, denoted as $u \to v$. Self loops are allowed.

- Weighted: each edge has an associated weight, given by a weight function

$$w : E \to \mathbf{R}.$$

Definitions –continue-

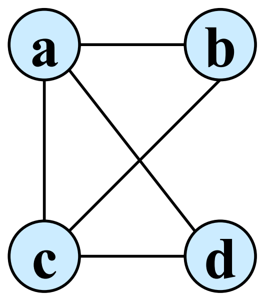
- If $(u, v) \in E$, then vertex v is adjacent to vertex u.

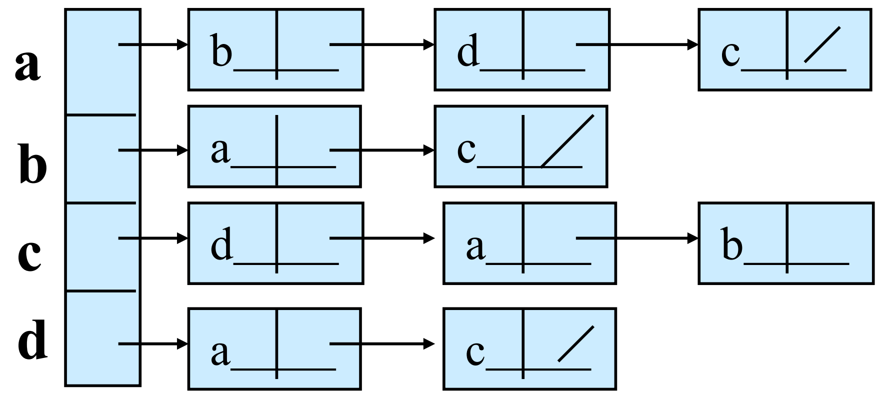
- Adjacency relationship is:
  - Symmetric if G is undirected.
  - Not necessarily so if G is directed.

- If G is connected:
  - There is a path between every pair of vertices.
  - $|E| \geq |V| - 1$.
  - Furthermore, if $|E| = |V| - 1$, then G is a tree.

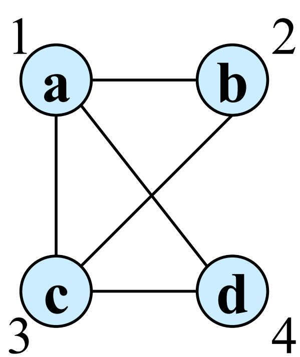
Representation of Graphs

- Two standard ways.
  - Adjacency Lists.





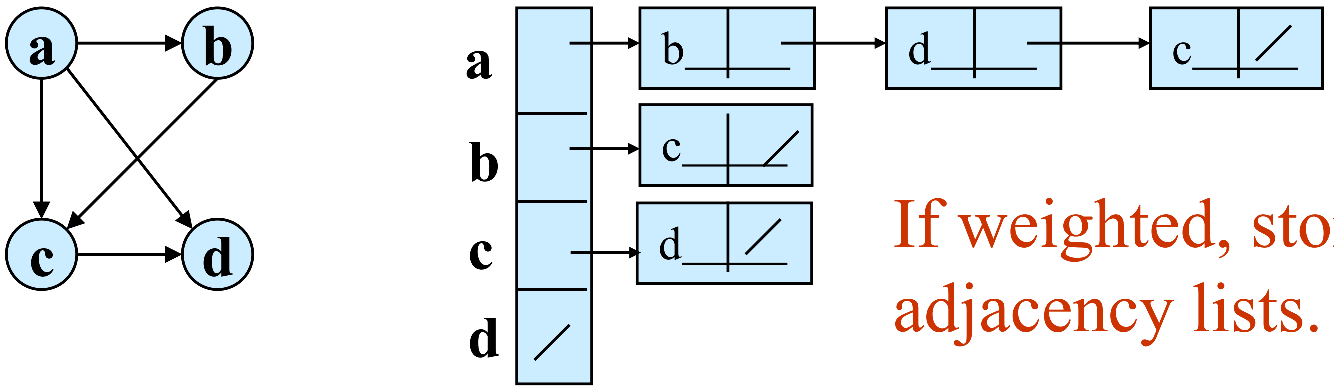
  - Adjacency Matrix.



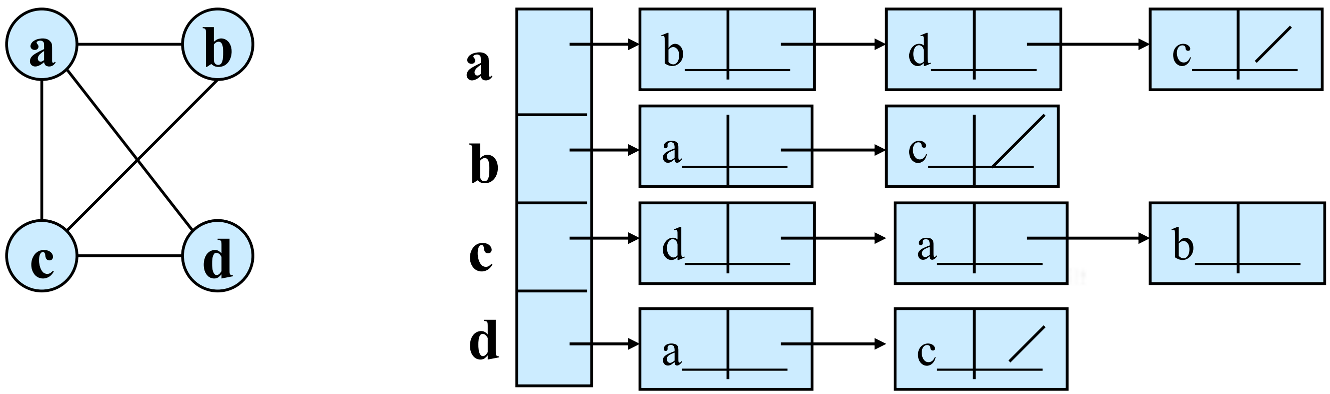
	1	2	3	4
1	0	1	1	1
2	1	0	1	0
3	1	1	0	1
4	1	0	1	0

Adjacency Lists

- Consists of an array *Adj* of $|V|$ lists.
- One list per vertex.
- For $u \in V$, *Adj*[*u*] = {all vertices adjacent to *u*}.



If weighted, store weights also in adjacency lists.



Storage Requirement

- **For directed graphs:**
  - **Sum of lengths of all adj. lists is**

    $$\sum_{v \in V} \text{out-degree}(v) = |E|$$

    Number of edges leaving v
  - **Total storage:** $\Theta(|V|+|E|)$

- **For undirected graphs:**
  - **Sum of lengths of all adj. lists is**

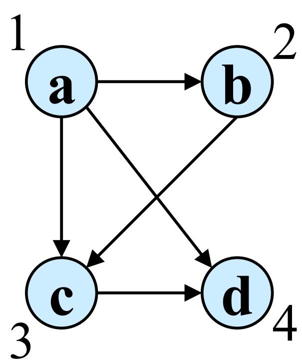
    $$\sum_{v \in V} \text{degree}(v) = 2|E|$$

    Number of edges incident on v.
  - **Total storage:** $\Theta(|V|+|E|)$

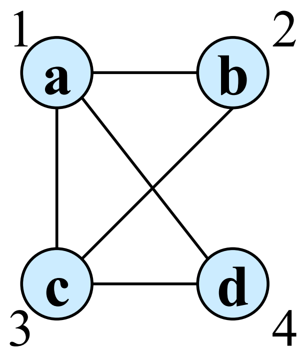
Adjacency Matrix

- $|V| \times |V|$ matrix A.
- Number vertices from 1 to $|V|$
- 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



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

$A = A^{\mathrm{T}}$ for undirected graphs.

Space and Time

- **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 for weighted graph.**

15.2 Graph-Traversal Algorithms

Standard Algorithms

- **Searching a graph**:
  - Systematically follow the edges of a graph to visit the vertices of the graph

- **discovering the structure of a graph.**

- **Standard graph-searching algorithms.**
  - **Breadth-first Search (BFS).**
  - **Depth-first Search (DFS).**

Breadth-First Search

- **Input:**
  - Graph $G = (V, E)$, either directed or undirected,
  - *source vertex* $s \in V$.

- **Output:** for all $v \in V$
  - $d[v]$ = length of shortest path from s to v ($d[v] = \infty$ if v is not reachable from s).
  - $\pi[v] = u$ if (u, v) is last edge on shortest path $s \rightsquigarrow v$.
    - u is v's predecessor.
  - breadth-first tree = a tree with root s that contains all reachable vertices.

Definitions on BSF

- **Path between vertices u and v:**

  vertices (v_1, v_2, …, v_k) such that

  u=v_1 and v =v_k,

  (v_i,v_{i+1}) $\in E$, for all $1 \leq i \leq k$-1.

- **Length of the path: Number of edges in the path.**

- **Path is simple if no vertex is repeated.**

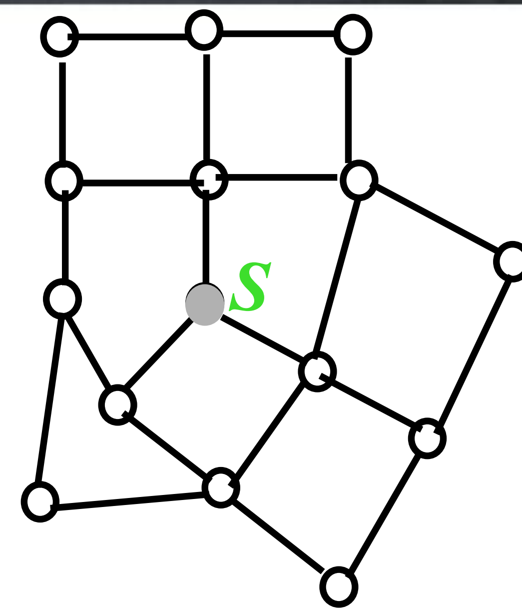
Principle of Breadth-First Search

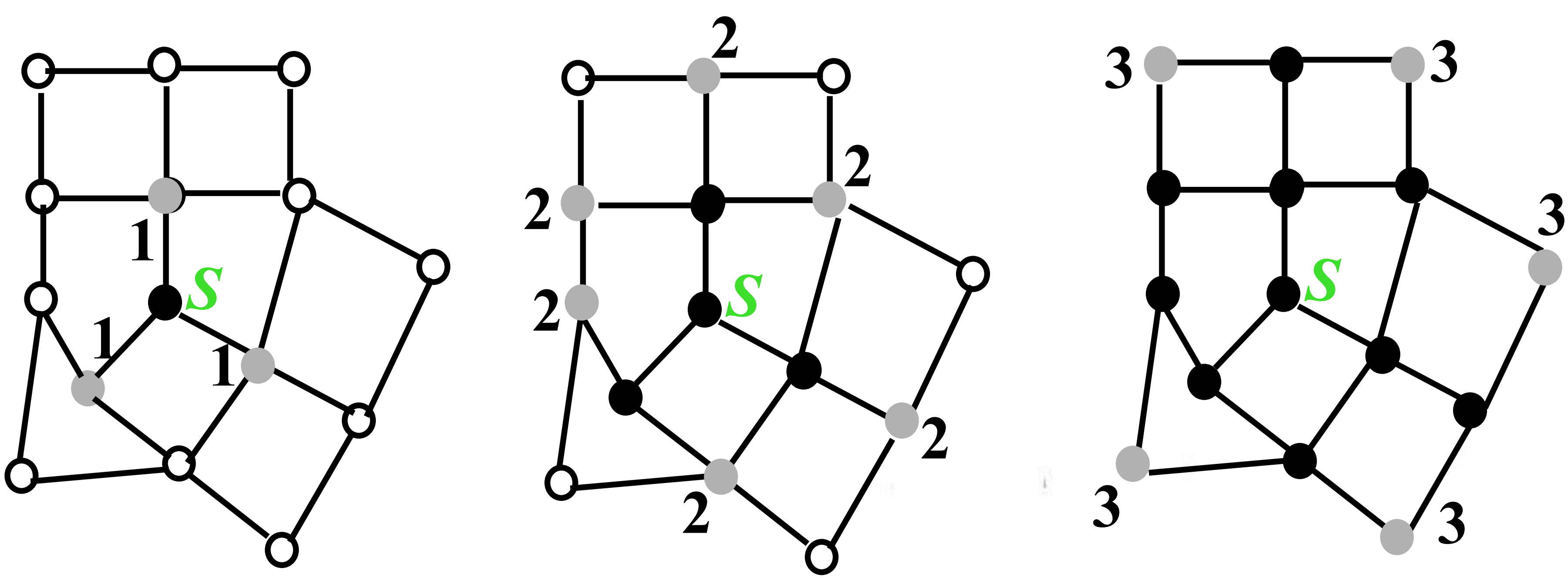
- **Expands the frontier between discovered and undiscovered vertices uniformly across the breadth of the frontier.**
  - **A vertex is "discovered" the first time it is encountered during the search.**
  - **A vertex is "finished" if all vertices adjacent to it have been discovered.**

BFS for Shortest Paths

Colors the vertices to keep track of progress.

- Undiscovered
- Discovered
- Finished





BFS(G,s)

```
for each vertex u in V[G] – {s}
        do color[u] ← white
           d[u] ← ∝
           π[u] ← nil
color[s] ← gray
d[s] ← 0
π[s] ← nil
Q ← Φ
enqueue(Q,s)
while Q ≠ Φ
        do u ← dequeue(Q)
                for each v in Adj[u]
                        do if color[v] = white
                                then color[v] ← gray
                                     d[v] ← d[u] + 1
                                     π[v] ← u
                                     enqueue(Q,v)
                color[u] ← black
```

white: undiscovered
gray: discovered
black: finished

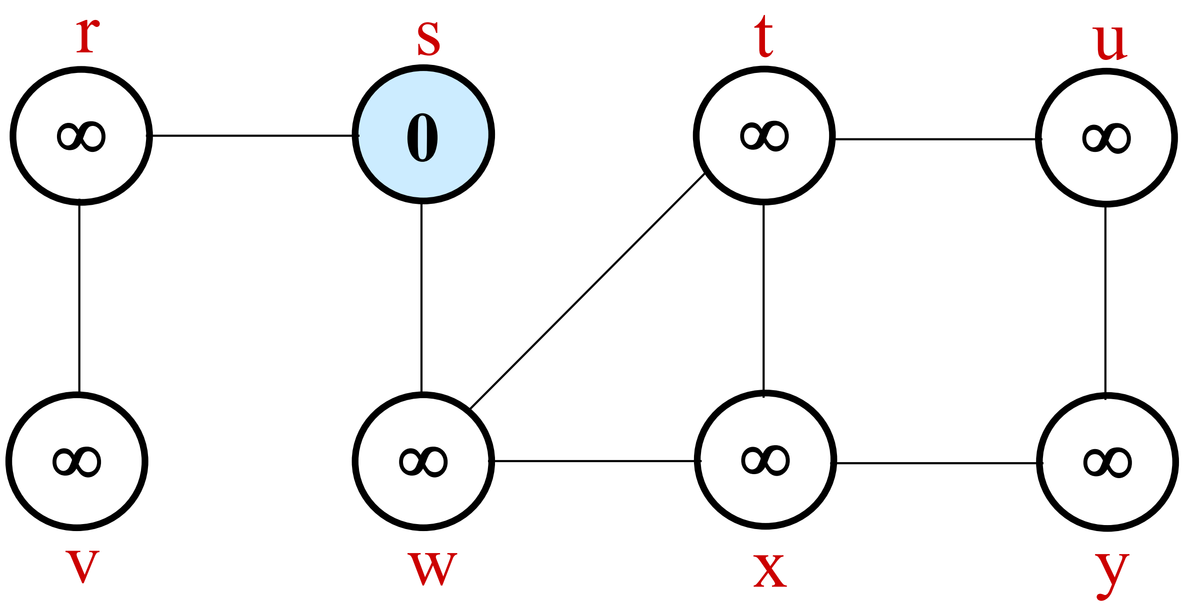
Q: a queue of discovered vertices

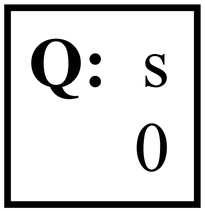
color[*v*]: color of v

d[*v*]: distance from s to v

π[*u*]: predecessor of v

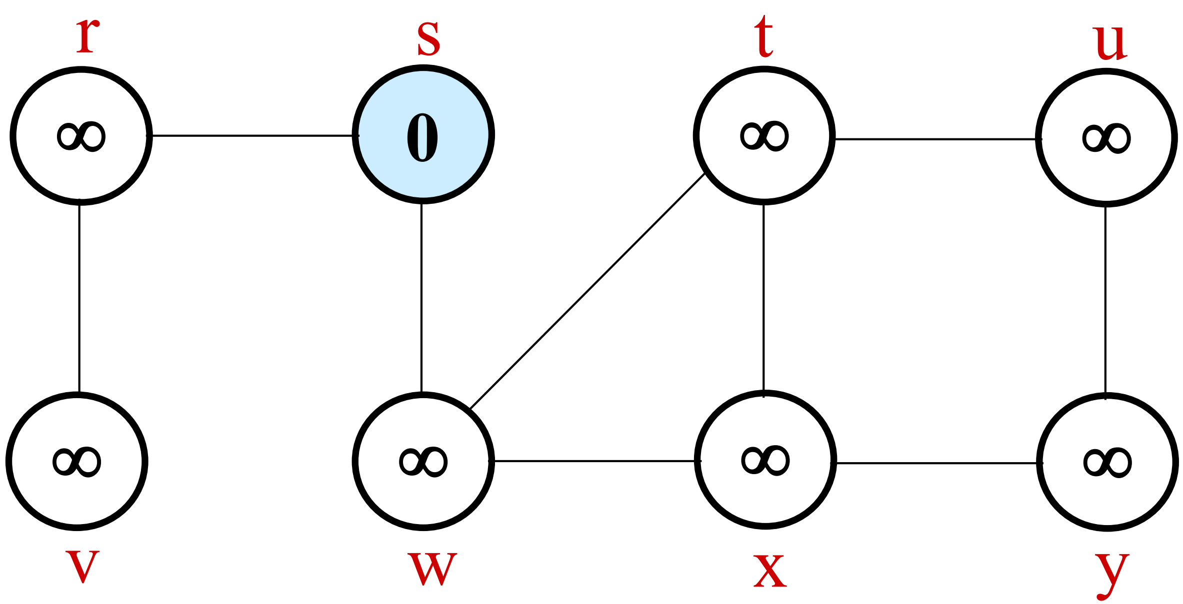
Example (BFS)





frontier

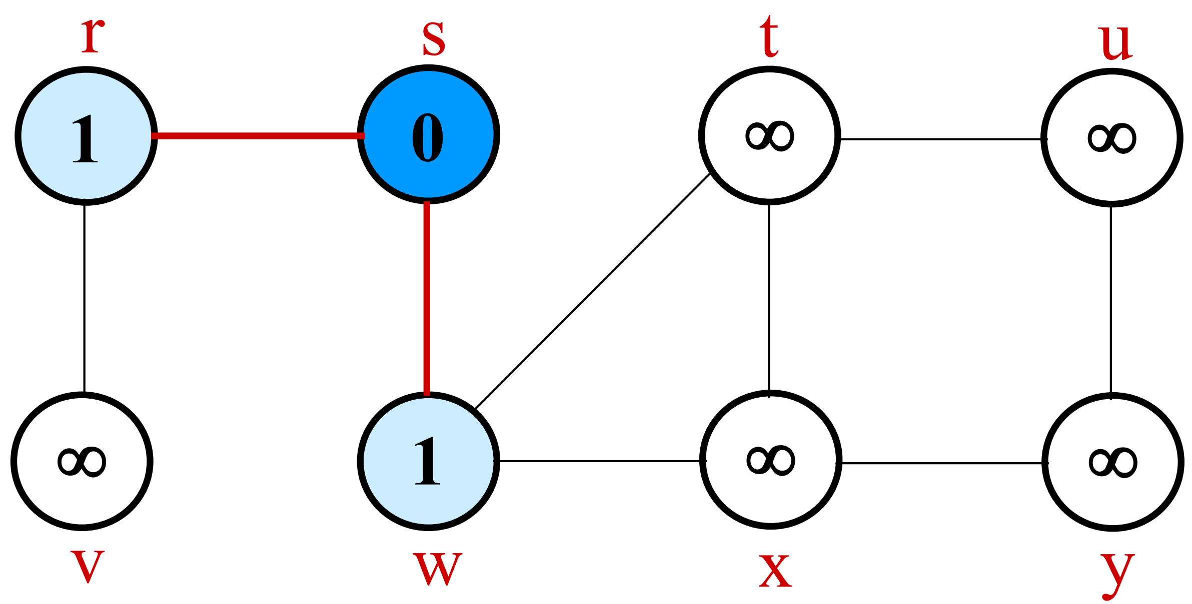
Example (BFS)



Q: s
0

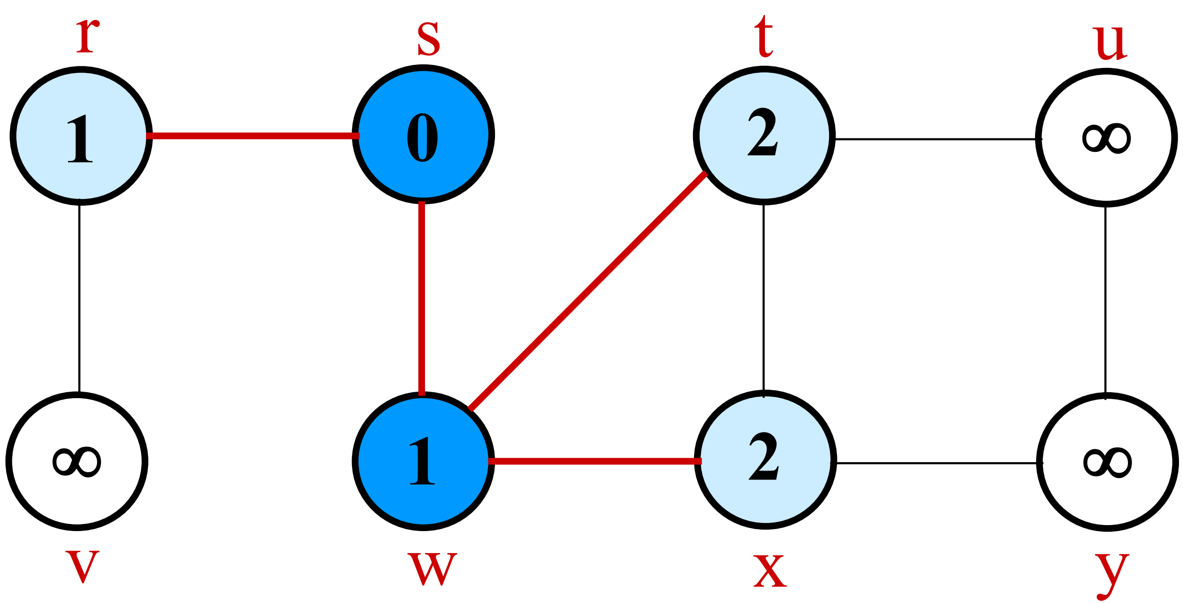
frontier

Example (BFS)



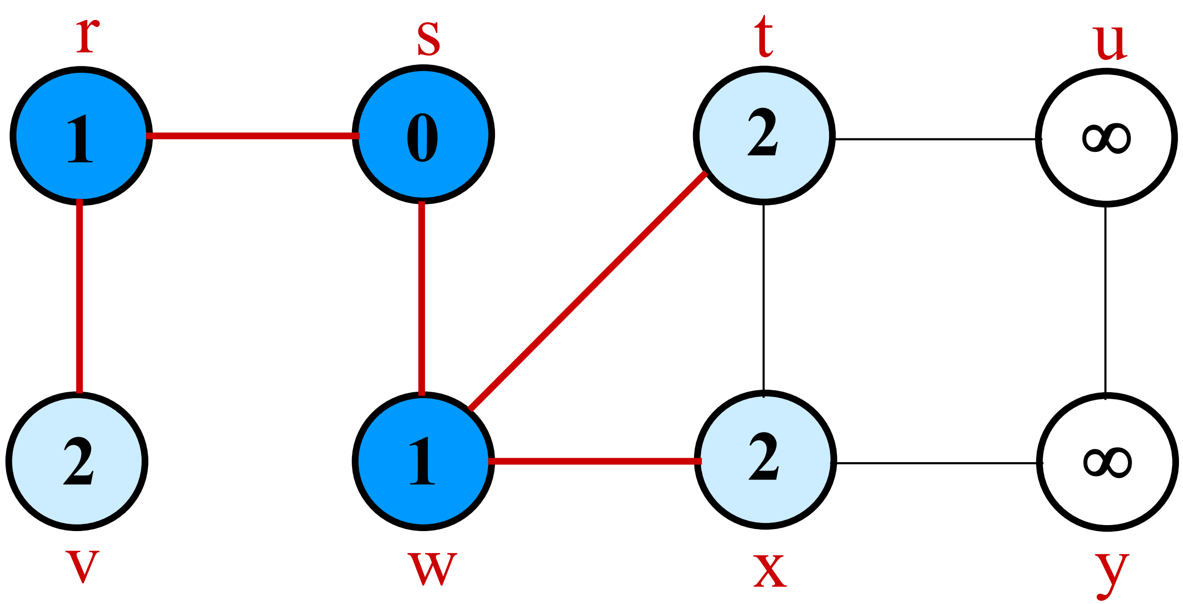
Q: w r
1 1

Example (BFS)



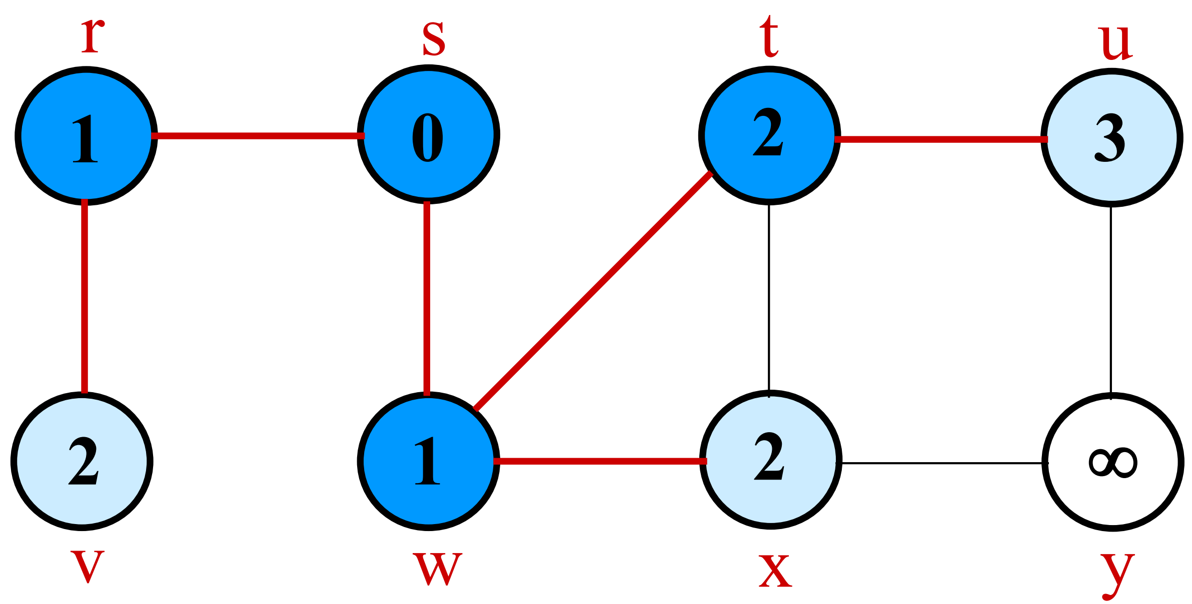
Q:	r	t	x
	1	2	2

Example (BFS)



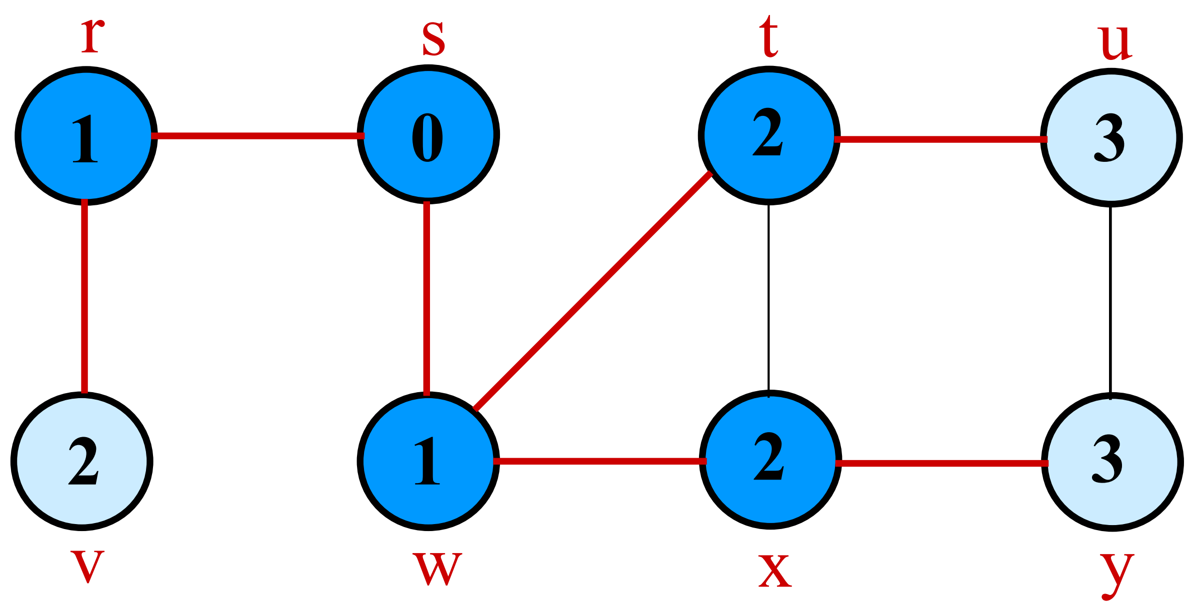
Q:	t	x	v
	2	2	2

Example (BFS)



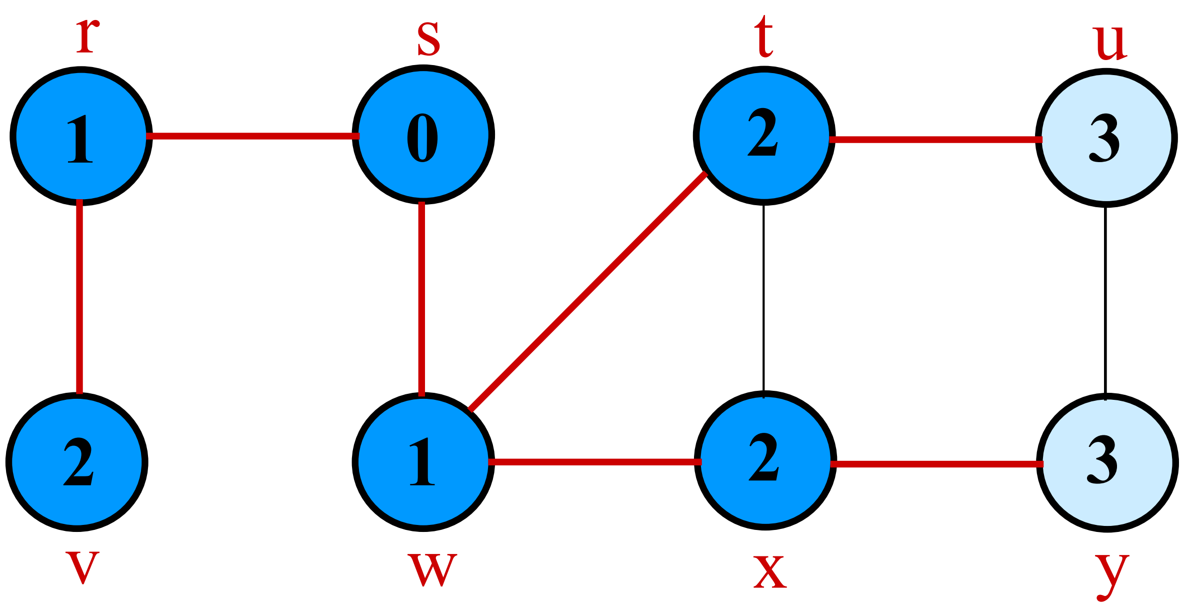
Q:	x	v	u
	2	2	3

Example (BFS)



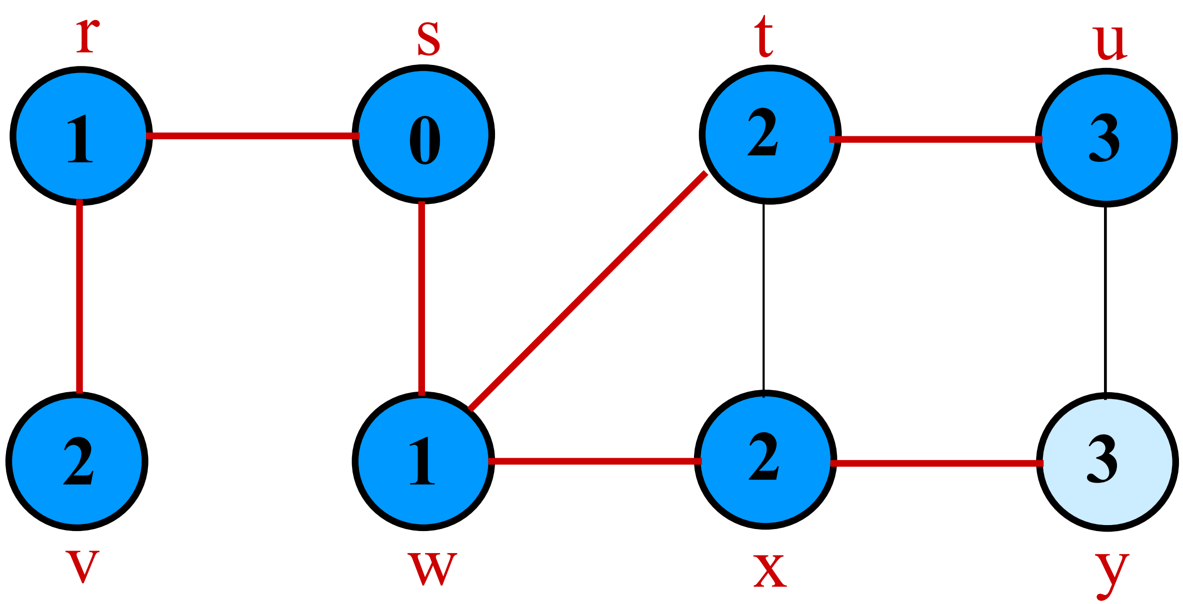
Q:	v	u	y
	2	3	3

Example (BFS)



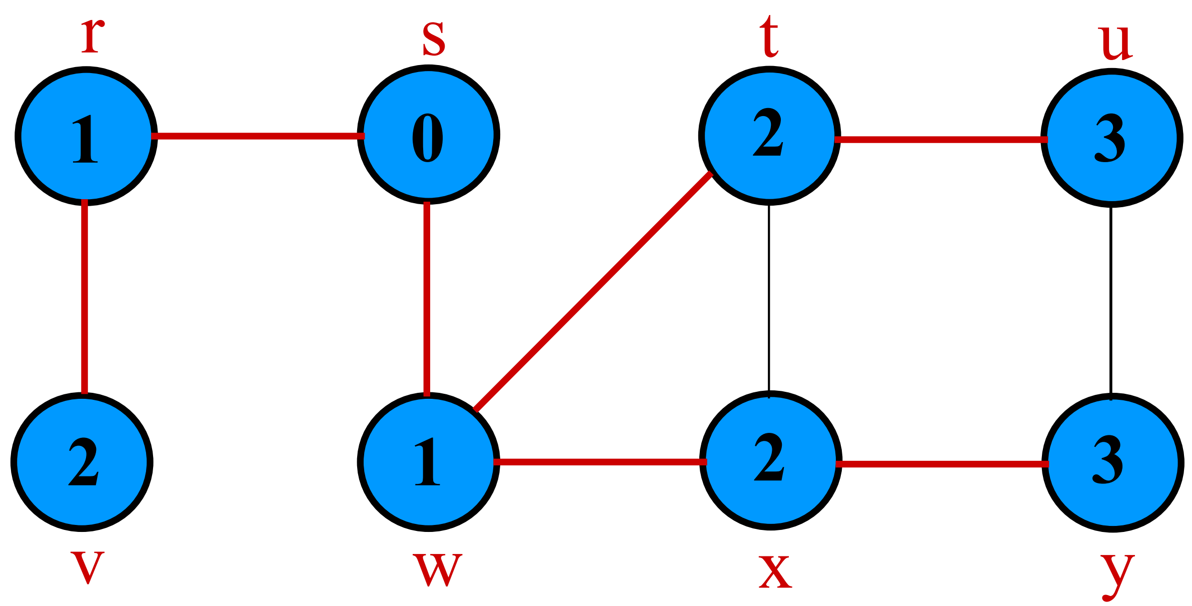
Q:	u	y
	3	3

Example (BFS)



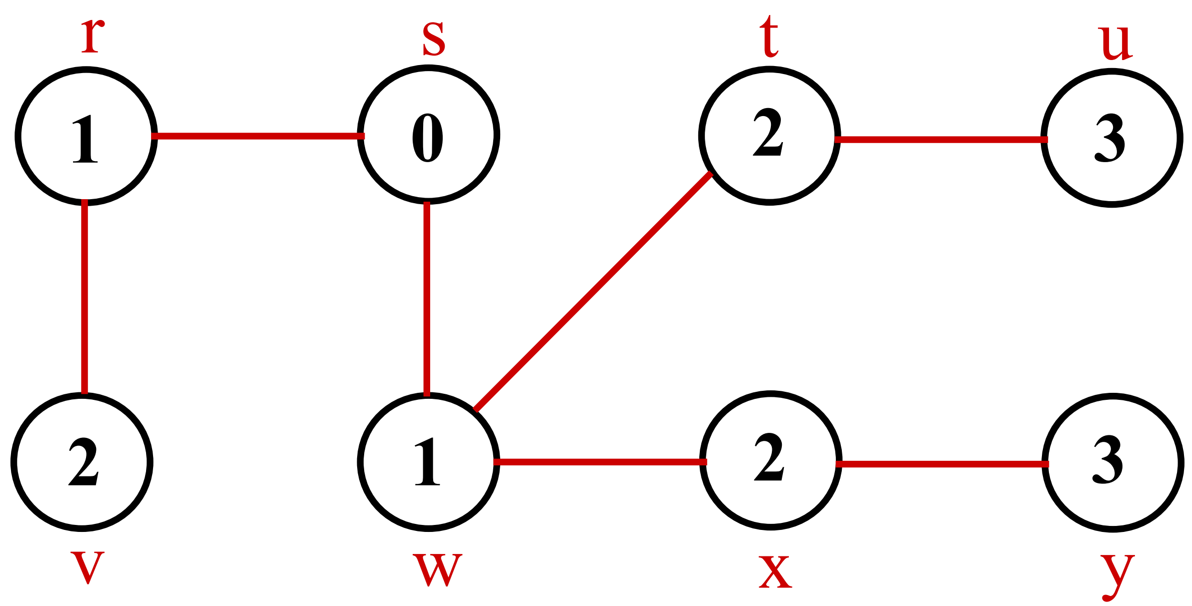
Q: y
3

Example (BFS)



Q: $\emptyset$

Example (BFS)



BF Tree

Breadth-First Tree

- **Predecessor sub-graph** of $G = (V, E)$ with source s is
  $G_\pi = (V_\pi, E_\pi)$ where
  - $V_\pi = \{v \in V : \pi[v] \neq \text{NIL}\} + \{s\}$
  - $E_\pi = \{(\pi[v], v) \in E : v \in V_\pi - \{s\}\}$
- G_π is a **breadth-first tree** if:
  - V_π consists of the vertices reachable from s
  - for all $v \in V_\pi$, there is a unique simple path from s to v in G_π
  - the path is also a shortest path from s to v in G.
- The edges in E_π are called **tree edges**.
  $|E_\pi| = |V_\pi| - 1$.

Analysis of BFS

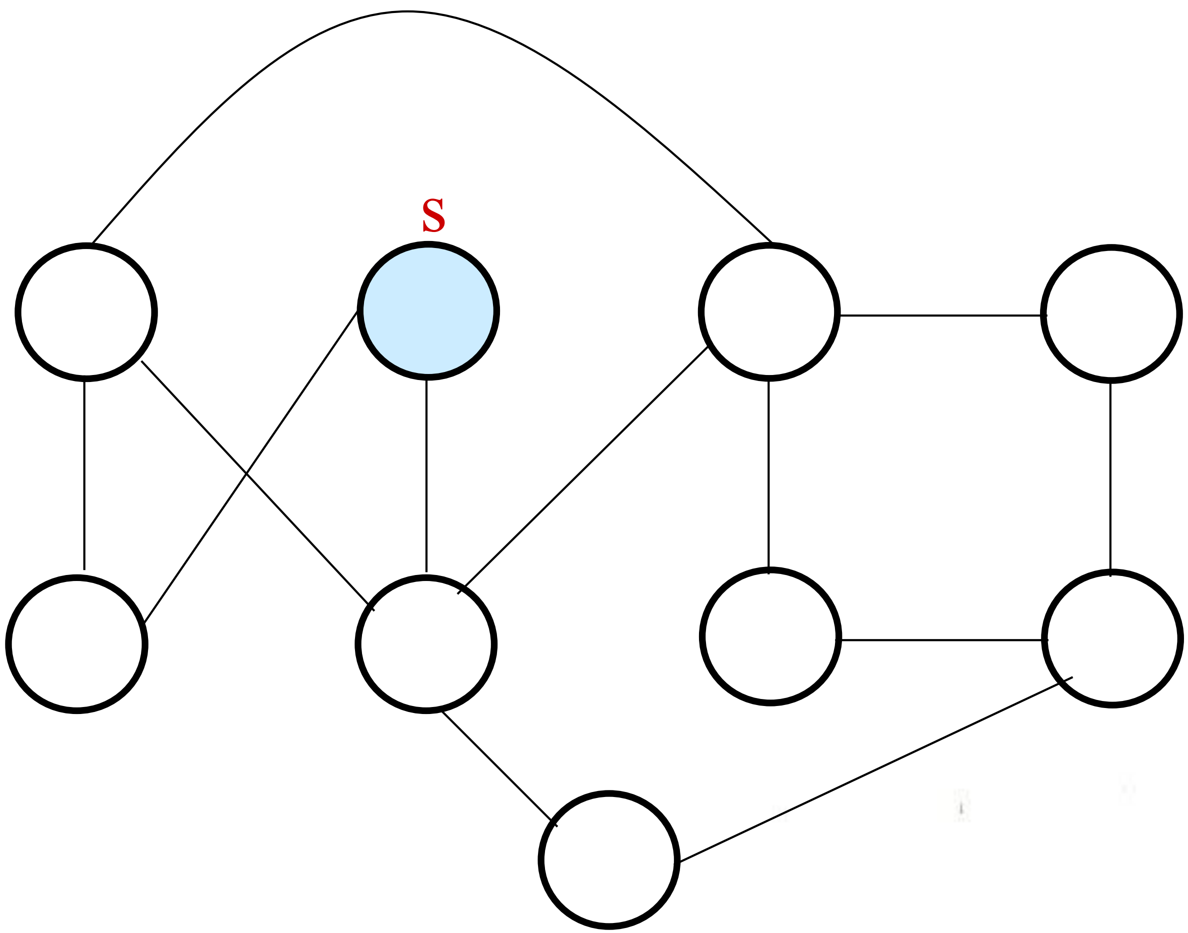
- **Initialization takes $O(|V|)$.**

- **Traversal Loop**
  - **Each vertex is enqueued and dequeued at most once, so the total time for queuing is $O(|V|)$.**
  - **The adjacency list of each vertex is scanned at most once.**
  - **The sum of lengths of all adjacency lists is $\Theta(|E|)$.**

- **Total running time of BFS is $O(|V|+|E|)$**
- **Correctness of BFS (see Dijkstra later)**

Short Test in Class

Compute the shortest distances of each vertex from the source vertex s, and give the BSF tree of the graph below.



Depth-First Search (DFS)

- **Explore edges out of the most recently discovered vertex *v*.**
- **When all edges of *v* have been explored, backtrack to its *predecessor* to explore other edges**
- **“Search as deep as possible first.”**
- **Continue until all vertices reachable from the original source are discovered.**

Depth-First Search

- **Input:** $G = (V, E)$, directed or undirected. No source vertex given!

- **Output:**
  - 2 time stamps on each vertex.
    - $d[v]$ = *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.

Program

DFS(G)

1. for each vertex $u \in V[G]$
2. do $color[u] \leftarrow$ white
3. $\pi[u] \leftarrow$ NIL
4. $time \leftarrow 0$
5. for each vertex $u \in V[G]$
6. do if $color[u] =$ white
7. then DFS-Visit(u)

DFS-Visit(u)

1. $color[u] \leftarrow$ GRAY

∇ u has been discovered

1. $time \leftarrow time + 1$

2. $d[u] \leftarrow time$

3. **for** each $v \in Adj[u]$

4. **do if** $color[v] =$ WHITE

5. **then** $\pi[v] \leftarrow u$

6. DFS-Visit(v)

7. $color[u] \leftarrow$ BLACK

∇ Blacken u; it is finished.

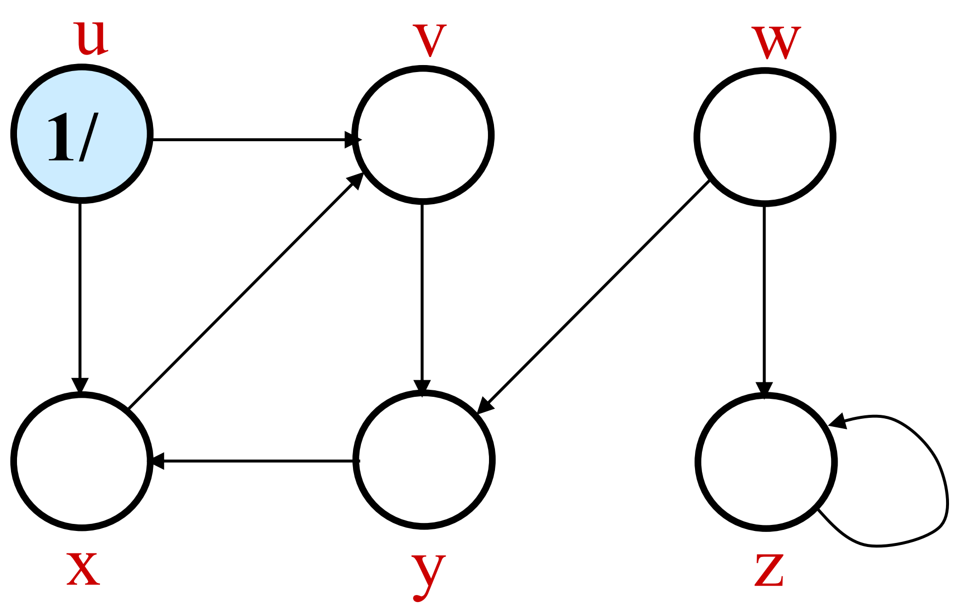
1. $f[u] \leftarrow time \leftarrow time + 1$

Uses a global timestamp *time*.

DFS: Kinds of edges

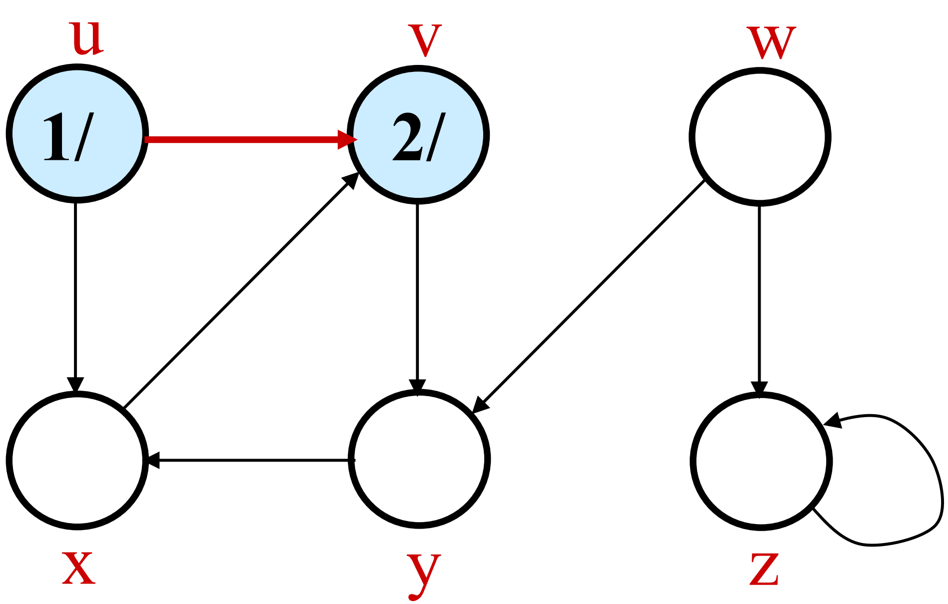
- **DFS introduces an important distinction among edges in the original graph:**
  - ***Tree edge*: encounter new (white) vertex**
  - ***Back edge*: from descendent to ancestor**
  - ***Forward edge*: from ancestor to descendent**
  - ***Cross edge*: between a tree or subtrees**

Example (DFS)



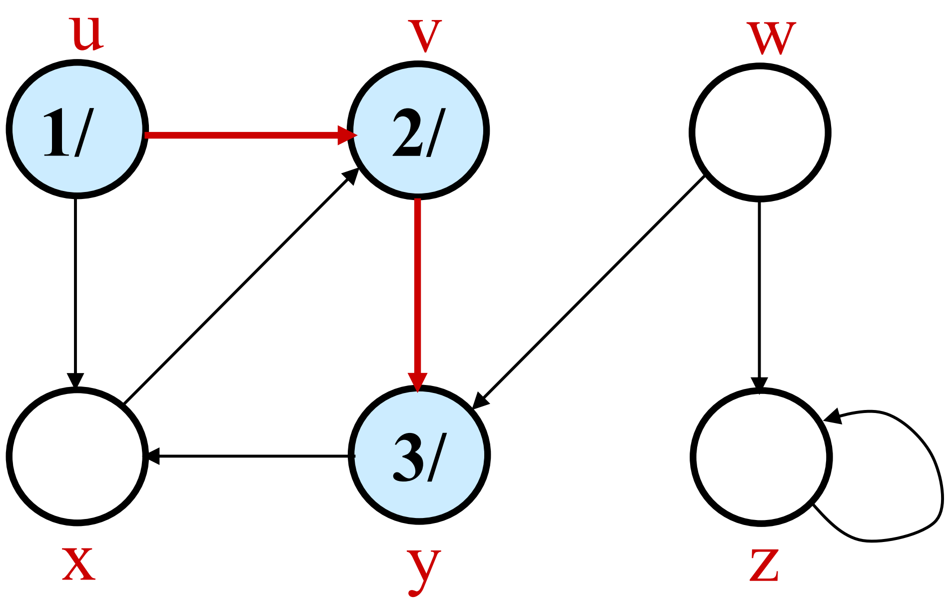
Tree edge: encounter new (white) vertex
Back edge: from descendent to ancestor
Forward edge: from ancestor to descendent
Cross edge: between a tree or subtrees

Example (DFS)



Tree edge: encounter new (white) vertex
Back edge: from descendent to ancestor
Forward edge: from ancestor to descendent
Cross edge: between a tree or subtrees

Example (DFS)



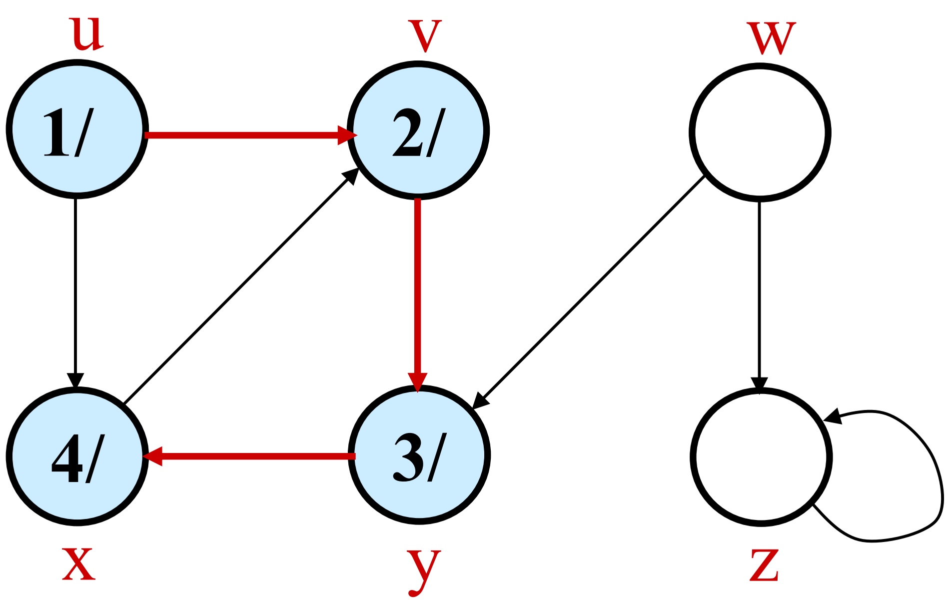
Tree edge: encounter new (white) vertex

Back edge: from descendent to ancestor

Forward edge: from ancestor to descendent

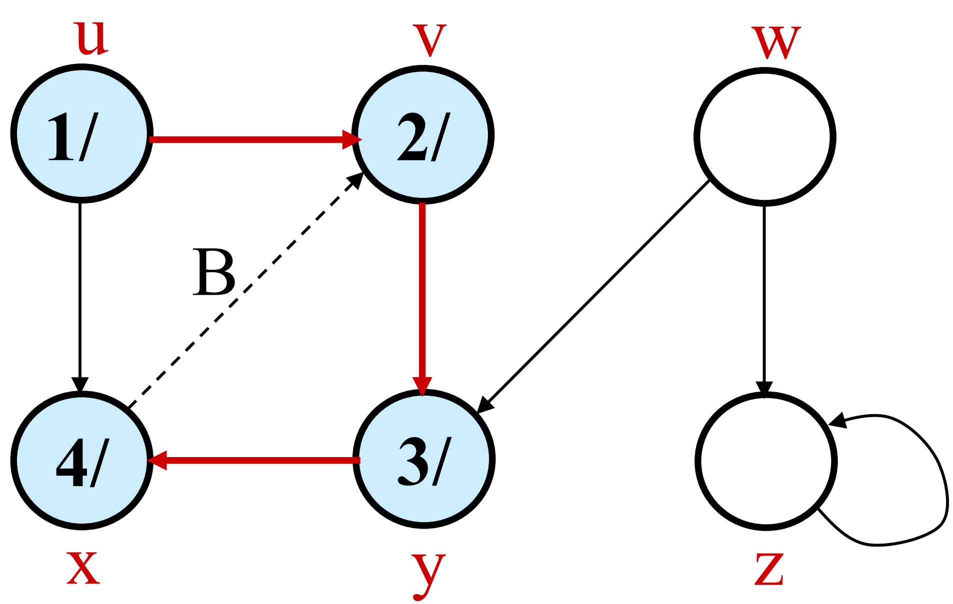
Cross edge: between a tree or subtrees

Example (DFS)



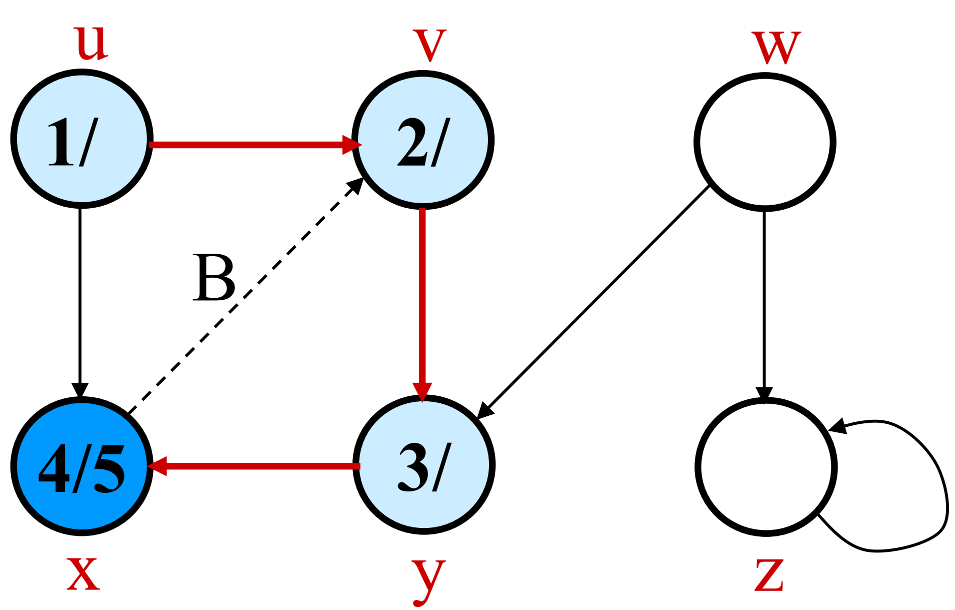
Tree edge: encounter new (white) vertex
Back edge: from descendent to ancestor
Forward edge: from ancestor to descendent
Cross edge: between a tree or subtrees

Example (DFS)



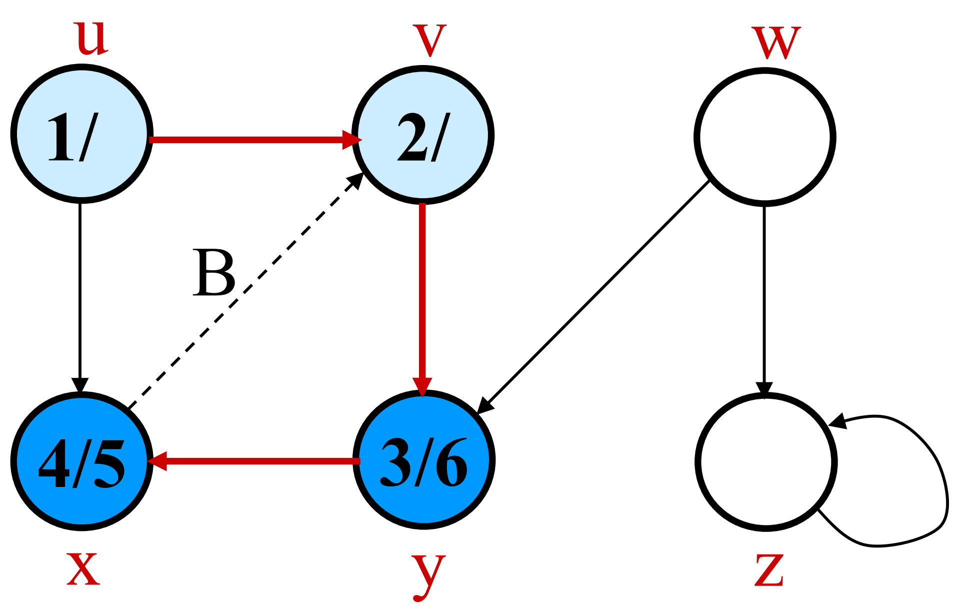
Tree edge: encounter new (white) vertex
Back edge: from descendent to ancestor
Forward edge: from ancestor to descendent
Cross edge: between a tree or subtrees

Example (DFS)



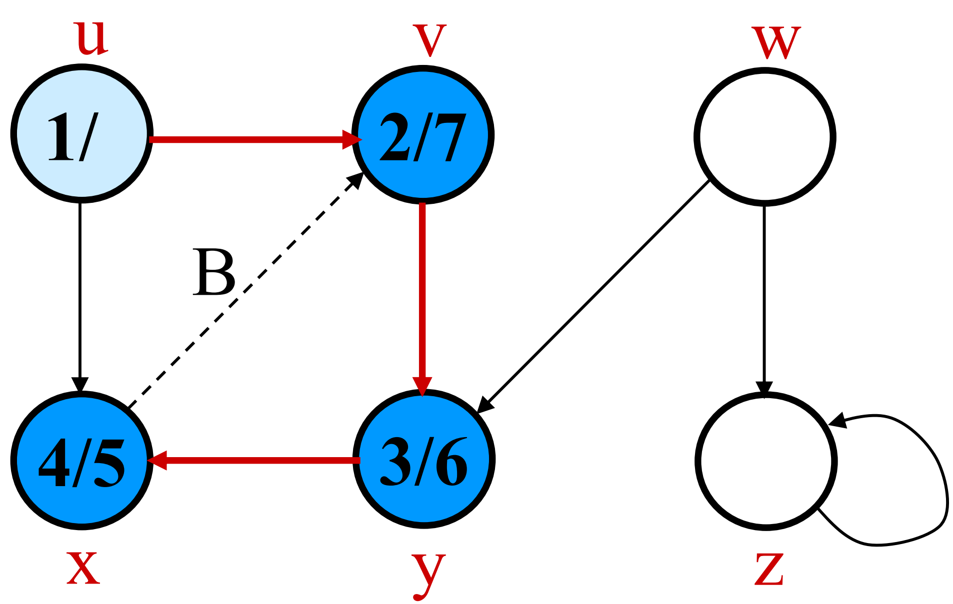
Tree edge: encounter new (white) vertex
Back edge: from descendent to ancestor
Forward edge: from ancestor to descendent
Cross edge: between a tree or subtrees

Example (DFS)



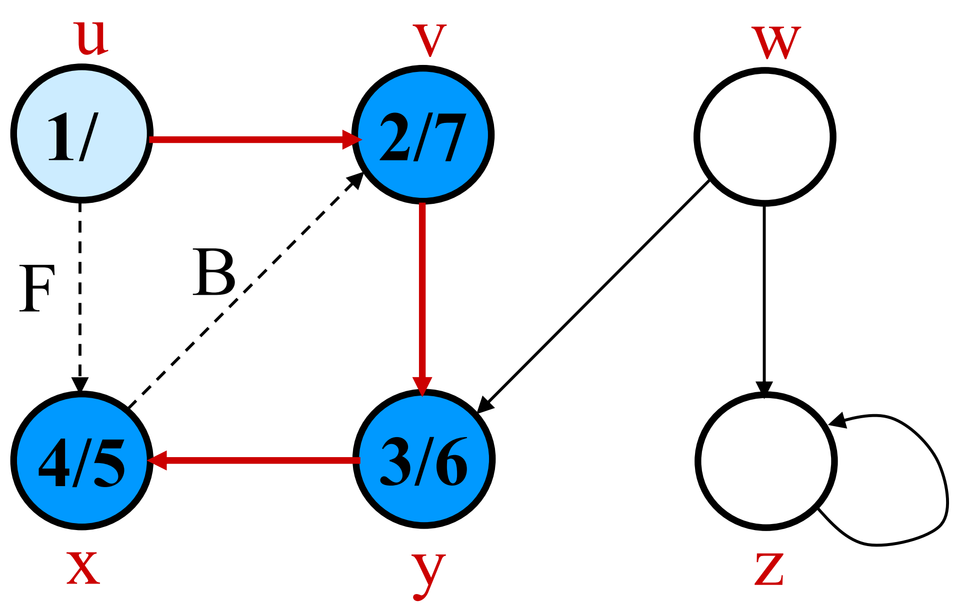
Tree edge: encounter new (white) vertex
Back edge: from descendent to ancestor
Forward edge: from ancestor to descendent
Cross edge: between a tree or subtrees

Example (DFS)



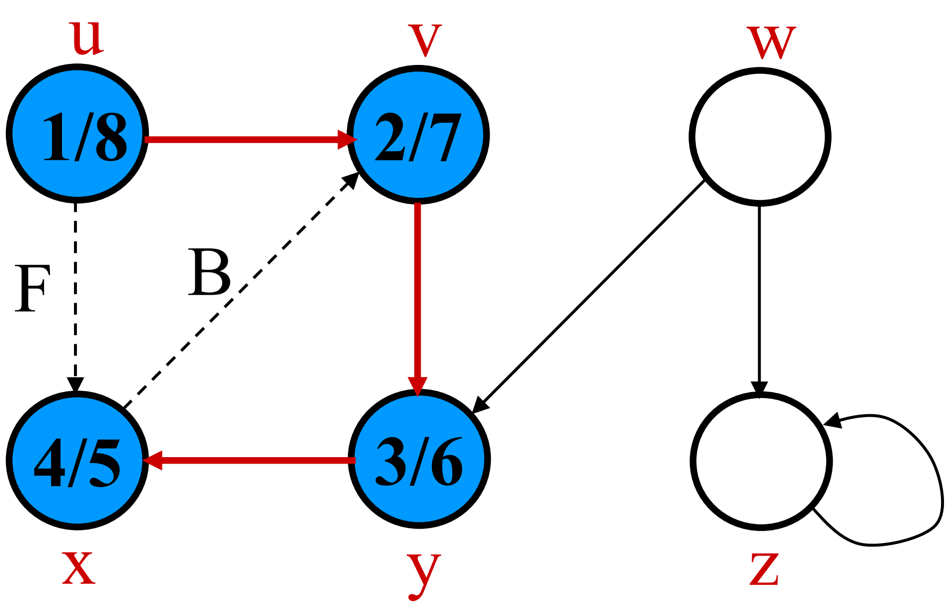
Tree edge: encounter new (white) vertex
Back edge: from descendent to ancestor
Forward edge: from ancestor to descendent
Cross edge: between a tree or subtrees

Example (DFS)



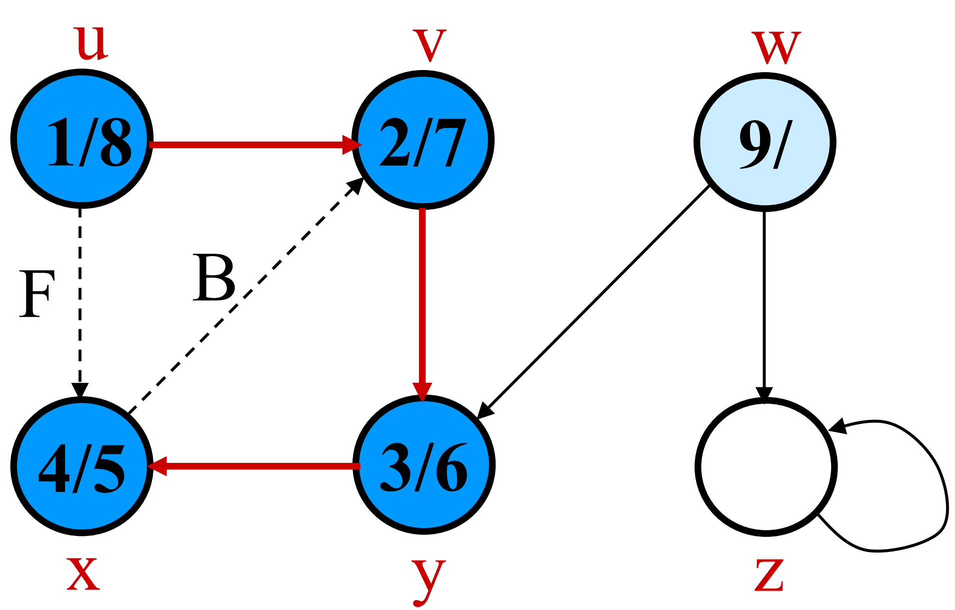
Tree edge: encounter new (white) vertex
Back edge: from descendent to ancestor
Forward edge: from ancestor to descendent
Cross edge: between a tree or subtrees

Example (DFS)



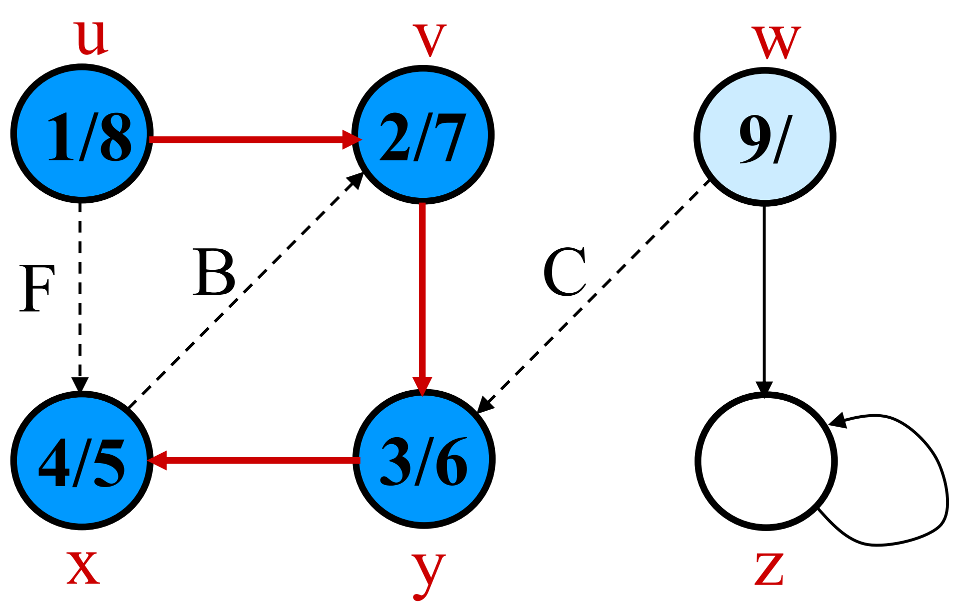
Tree edge: encounter new (white) vertex
Back edge: from descendent to ancestor
Forward edge: from ancestor to descendent
Cross edge: between a tree or subtrees

Example (DFS)



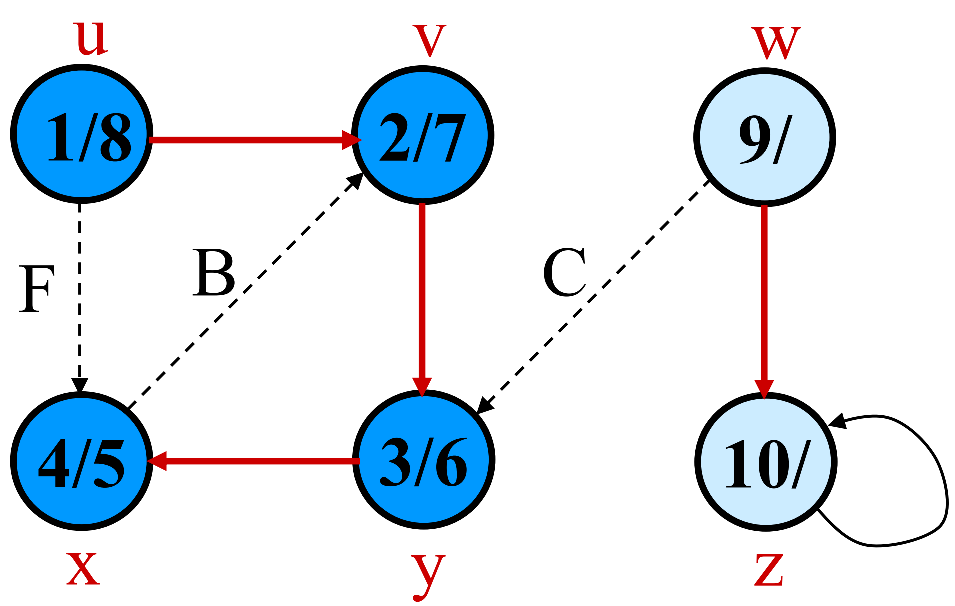
Tree edge: encounter new (white) vertex
Back edge: from descendent to ancestor
Forward edge: from ancestor to descendent
Cross edge: between a tree or subtrees

Example (DFS)



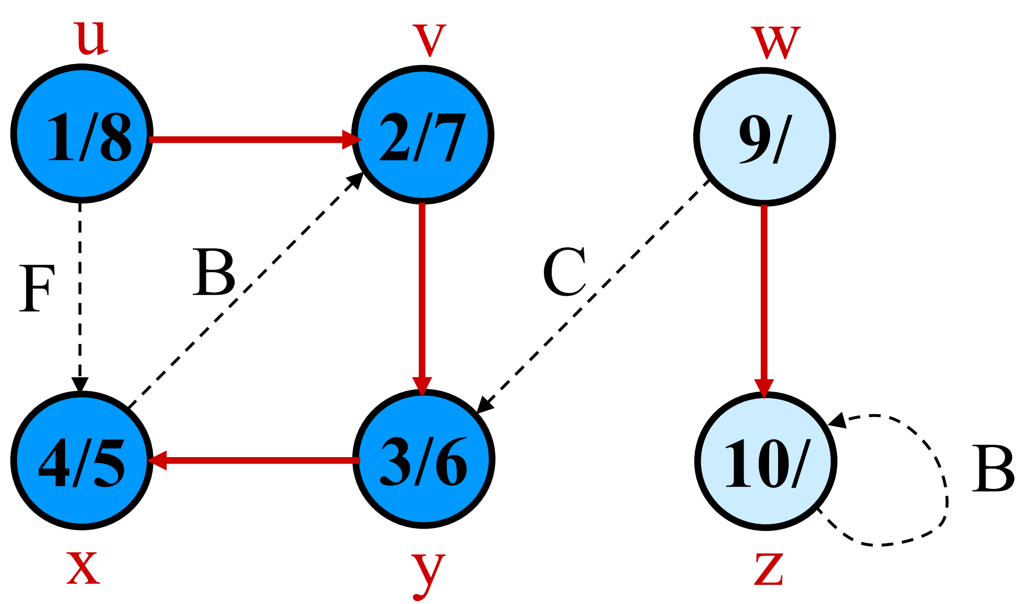
Tree edge: encounter new (white) vertex
Back edge: from descendent to ancestor
Forward edge: from ancestor to descendent
Cross edge: between a tree or subtrees

Example (DFS)



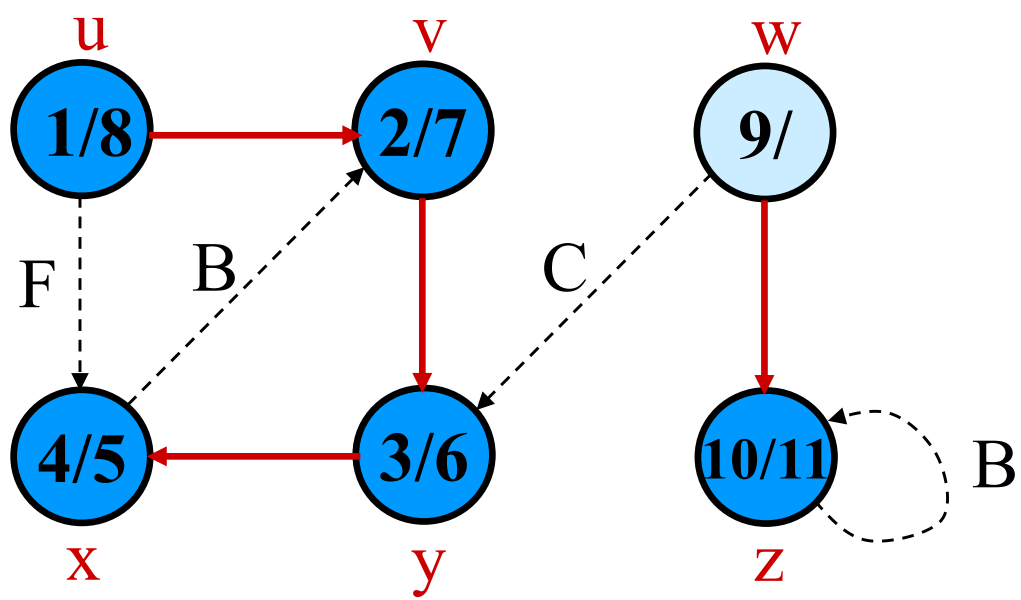
Tree edge: encounter new (white) vertex
Back edge: from descendent to ancestor
Forward edge: from ancestor to descendent
Cross edge: between a tree or subtrees

Example (DFS)



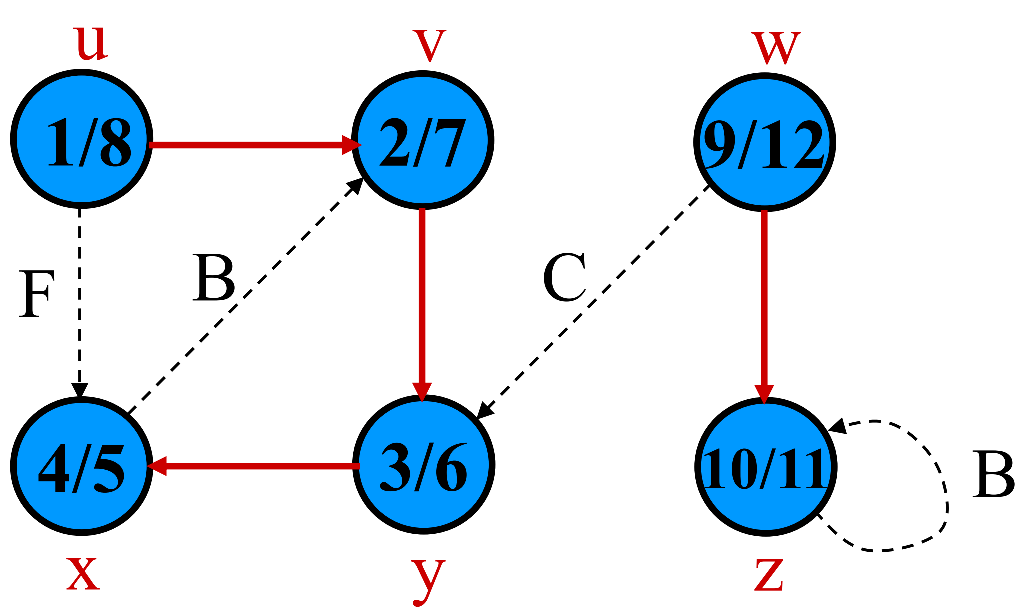
Tree edge: encounter new (white) vertex
Back edge: from descendent to ancestor
Forward edge: from ancestor to descendent
Cross edge: between a tree or subtrees

Example (DFS)



Tree edge: encounter new (white) vertex
Back edge: from descendent to ancestor
Forward edge: from ancestor to descendent
Cross edge: between a tree or subtrees

Example (DFS)



Tree edge: encounter new (white) vertex
Back edge: from descendent to ancestor
Forward edge: from ancestor to descendent
Cross edge: between a tree or subtrees

Classification of Edges

- **Tree edge:** in the **depth-first forest**, by exploring (*u*, *v*).
- **Back edge:** (*u*, *v*), where *u* is a descendant of *v* (in the depth-first tree). (include self-loop)
- **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.

Identification of Edges

- **Edge type for edge (*u*, *v*) can be identified when it is first explored by DFS.**

- **Identification is based on the color of *v*.**
  - **White – tree edge.**
  - **Gray – back edge.**
  - **Black – forward or cross edge.**

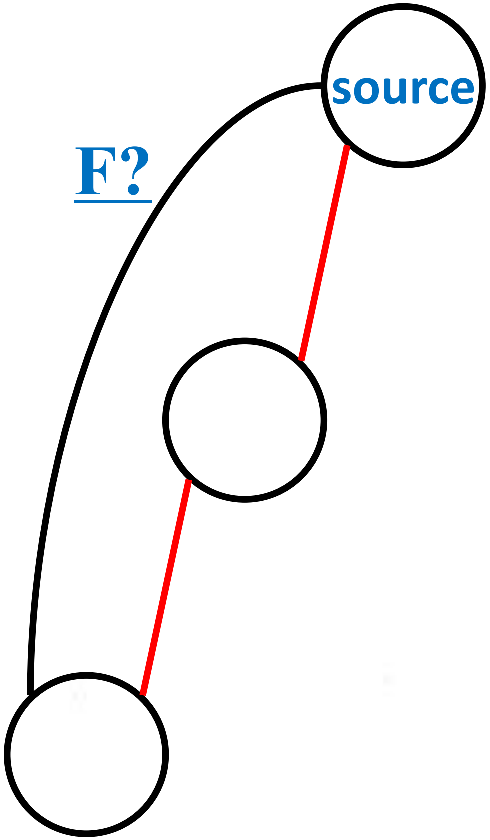
Identification of Edges

Theorem:
In DFS of an undirected graph, we get only tree and back edges. No forward or cross edges.

Proof by contradiction:

Assume there's a forward edge

But F? edge must actually be a back edge (*why?*)



Identification of Edges

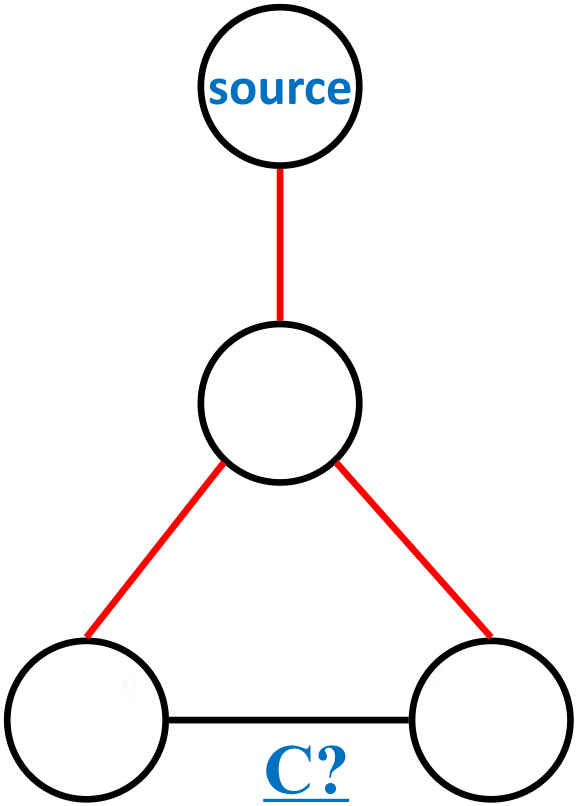
Theorem:
In DFS of an undirected graph, we get only tree and back edges. No forward or cross edges.

Proof by contradiction:

Assume there's a cross edge

But C? edge cannot be cross!

So in fact the picture is wrong…both lower tree edges cannot in fact be tree edges



Depth-First Trees

- **Predecessor subgraph is slightly different from that of BFS.**

- **The predecessor subgraph of DFS**

$$G_\pi = (V, E_\pi)$$

$$E_\pi = \{(\pi[v], v) : v \in V \text{ and } \pi[v] \neq \text{NIL}\}.$$

  - **G_π forms a *depth-first forest* composed of several *depth-first trees*. E_π consists of *tree edges*.**

Definition:
Forest: An acyclic graph G that may be disconnected.

Analysis of DFS

<u>DFS(G)</u>

for each vertex $u \in V[G]$
do $color[u] \leftarrow$ white
$\pi[u] \leftarrow$ NIL
$time \leftarrow 0$
for each vertex $u \in V[G]$
do if $color[u] =$ white
then DFS-Visit(u)

<u>DFS-Visit(u)</u>

$color[u] \leftarrow$ GRAY
∇ u has been discovered
$time \leftarrow time + 1$
$d[u] \leftarrow time$
for each $v \in Adj[u]$
do if $color[v] =$ WHITE
then $\pi[v] \leftarrow u$
DFS-Visit(v)
$color[u] \leftarrow$ BLACK
∇ Blacken u; it is finished.
$f[u] \leftarrow time \leftarrow time + 1$

Uses a global timestamp ***time***.

Analysis of DFS

- **Loops on lines 1-2 & 5-7 take $\Theta(|V|)$ time, excluding time to execute DFS-Visit.**
- **DFS-Visit is called once for each white vertex $v \in V$ when it's painted gray the first time. Lines 3-6 of DFS-Visit is executed |Adj[v]| times. The total cost of executing DFS-Visit is**

$$\sum_{v \in V} |\mathrm{Adj}[v]| = \Theta(|E|)$$

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

Parenthesis Theorem

Theorem 22.7

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

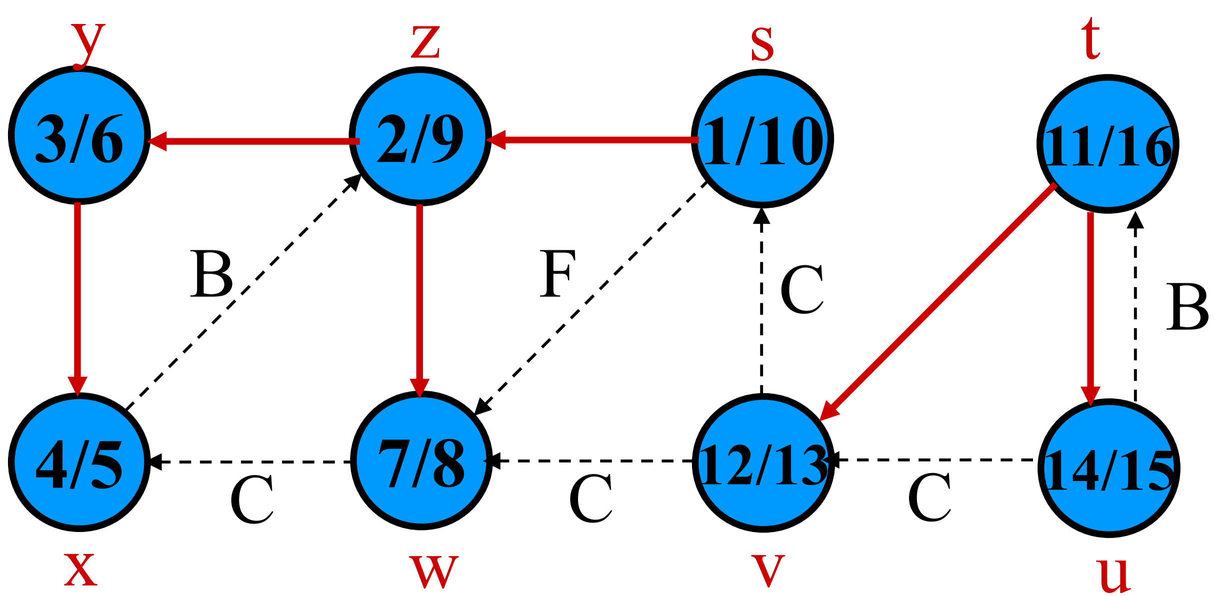
1. **$d[u] < f[u] < d[v] < f[v]$ or $d[v] < f[v] < d[u] < f[u]$ and neither u nor v is a descendant of the other.**
2. **$d[u] < d[v] < f[v] < f[u]$ and v is a descendant of u.**
3. **$d[v] < d[u] < f[u] < f[v]$ and u is a descendant of v.**

- So $d[u] < d[v] < f[u] < f[v]$ *cannot* happen.
- Like parentheses:
  - OK: ( ) [ ] ([ ]) [( )]
  - Not OK: ([ )] [( ])

Corollary

v is a proper descendant of u if and only if $d[u] < d[v] < f[v] < f[u]$.

Example (Parenthesis Theorem)



(s (z (y (x x) y) (w w) z) s) (t (v v) (u u) t)

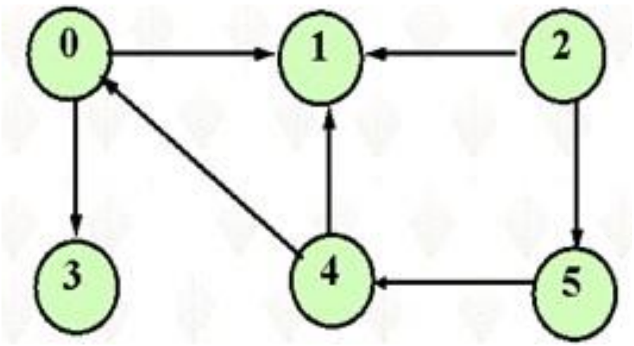
15.3 Topological Sorting

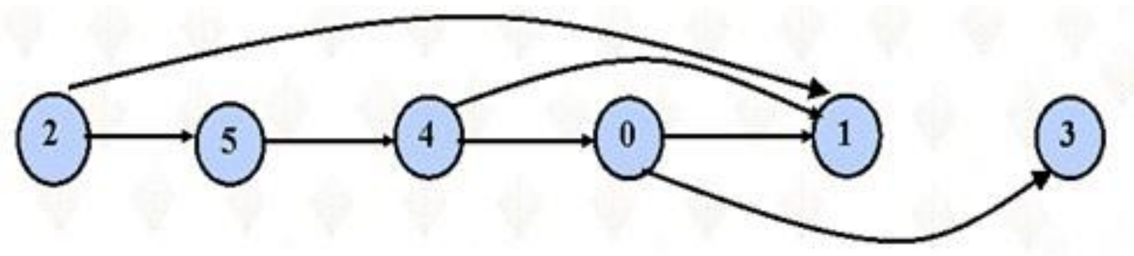
Directed Acyclic Graph

- **DAG – Directed graph with no cycles.**
- **Good for modeling processes and structures that have a partial order:**
  - $a > b$ **and** $b > c \Rightarrow a > c$. **(transitive closure)**
  - **But may have** a **and** b **such that neither** $a > b$ **nor** $b > a$.
- **Can always make a total order (either** $a > b$ **or** $b > a$ **for all** $a \neq b$**) from a partial order.**

Topological Ordering

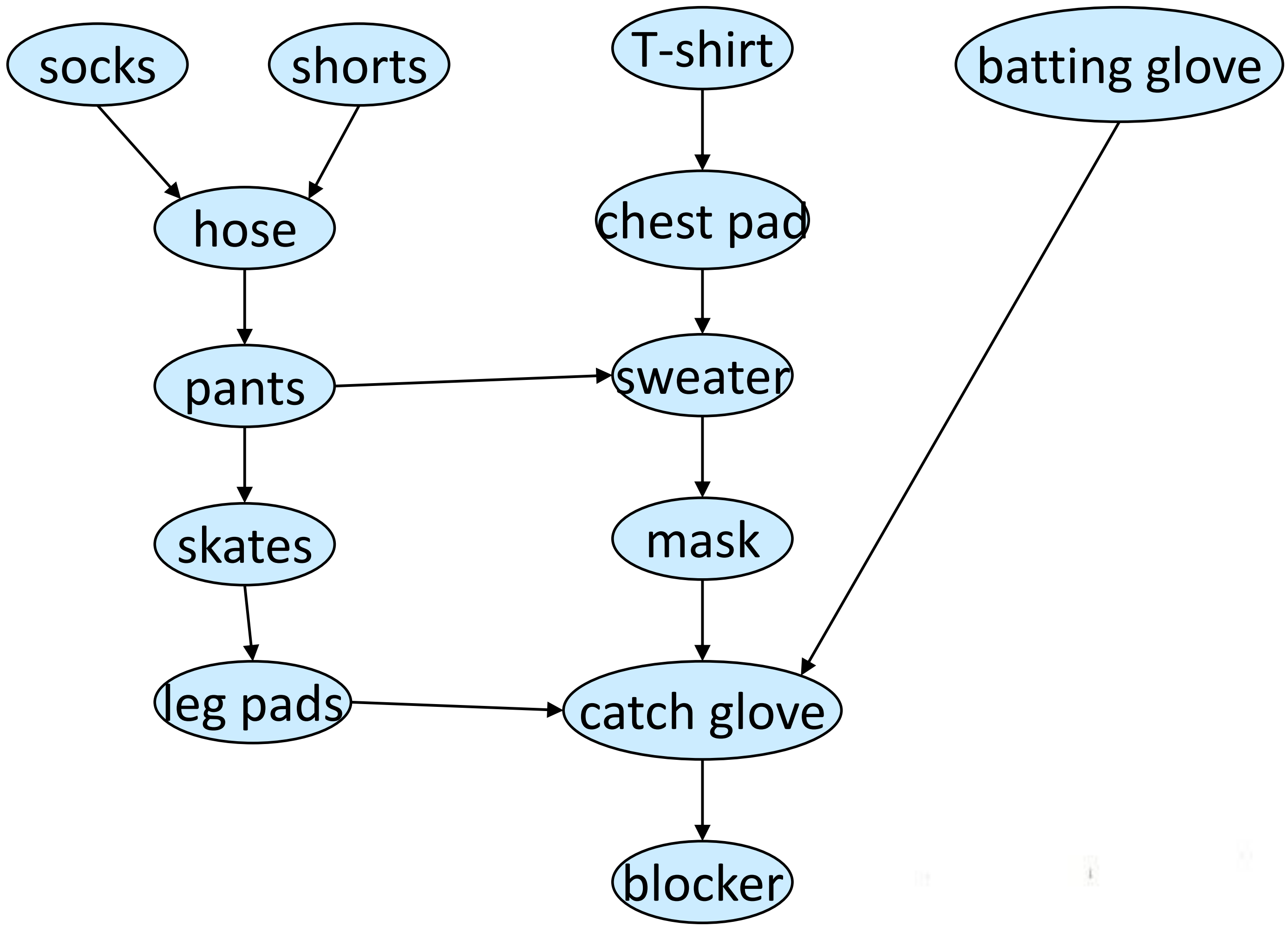
- Suppose that G is a directed graph which contains no directed cycles.
- Then a topological ordering of the vertices in G is a sequential listing of the vertices such that for any pair of vertices, v and w in G, if <v,w> is an edge in G then v precedes w in the sequential listing.



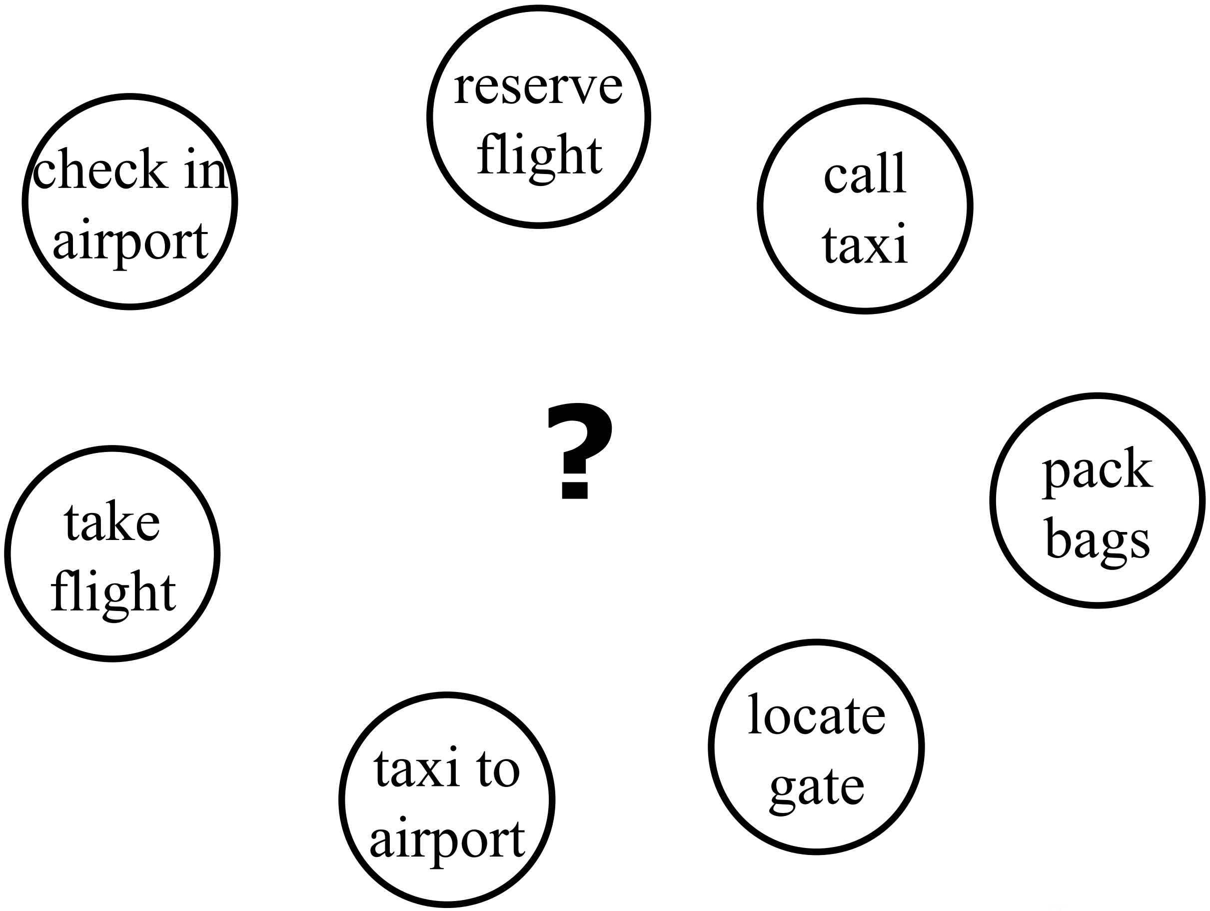


Example

DAG of dependencies for putting on goalie equipment.

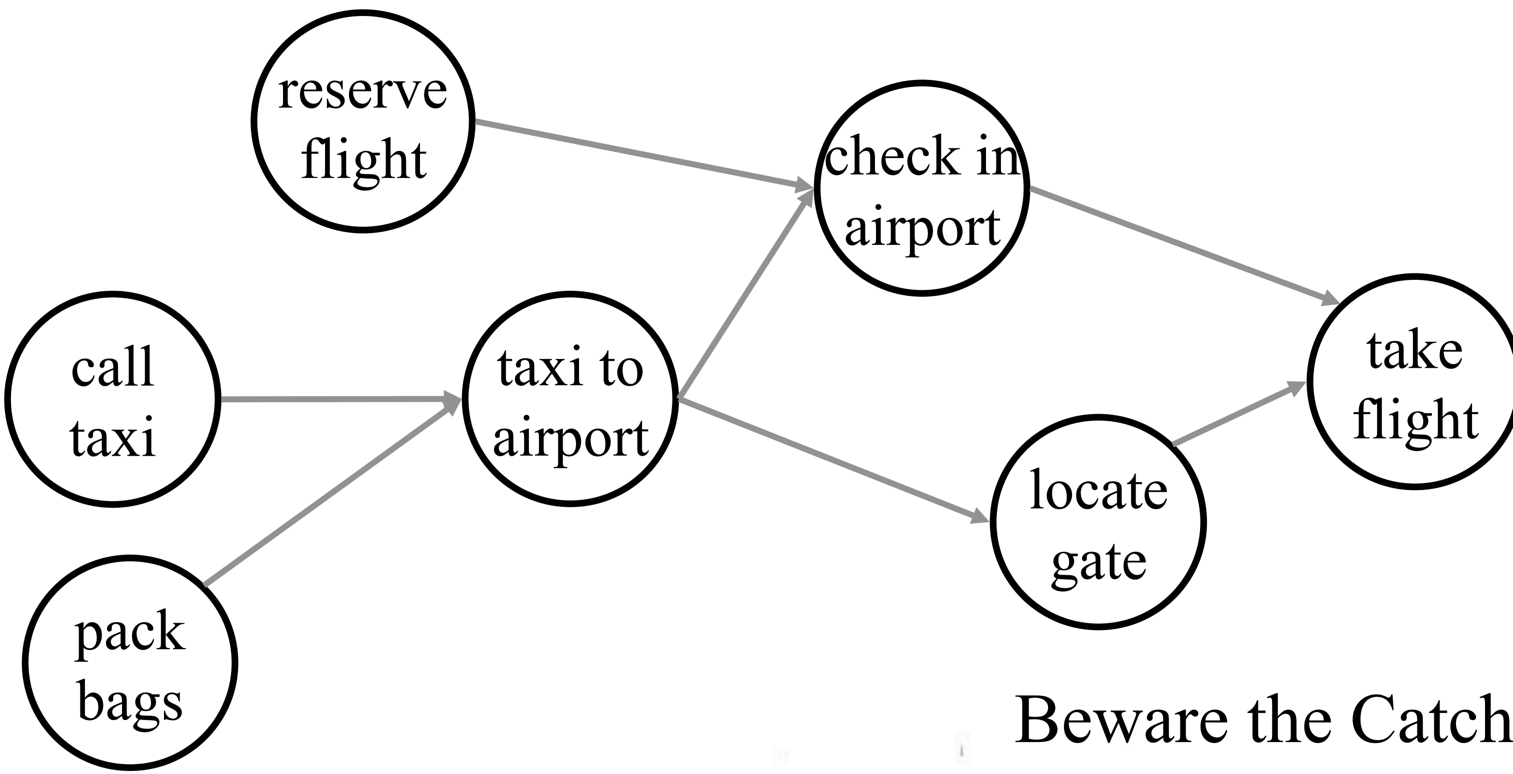


Partial Order: Planning a Trip with GF!



Partial Order: Planning a Trip with GF!

- **Given a graph, `G = (V, E)`, output all the vertices in `V` such that no vertex is output before any other vertex with an edge to it.**



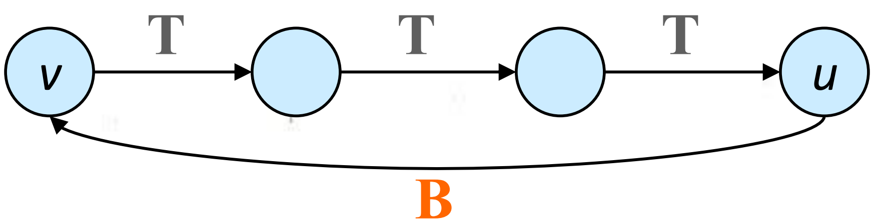
Beware the Catch-22!

Characterizing a DAG

Lemma 22.11 A directed graph *G* is acyclic iff a DFS of G yields no back edges.

Proof:

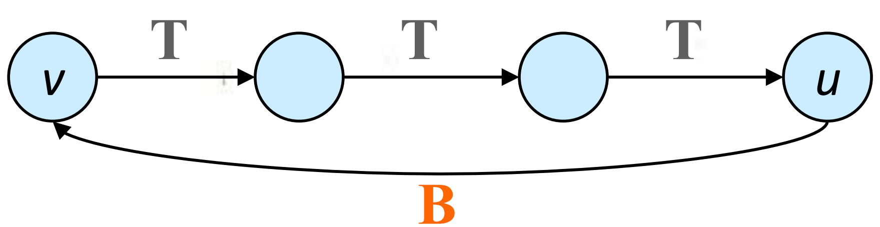
- **⇒: Show that back edge ⇒ cycle.**
  - **Suppose there is a back edge <*u, v*>. Then *v* is ancestor of *u* in depth-first forest.**
  - **Therefore, there is a path *v* $\rightsquigarrow$ *u*, so *v* $\rightsquigarrow$ *u* $\rightsquigarrow$ *v* is a cycle.**



Characterizing a DAG

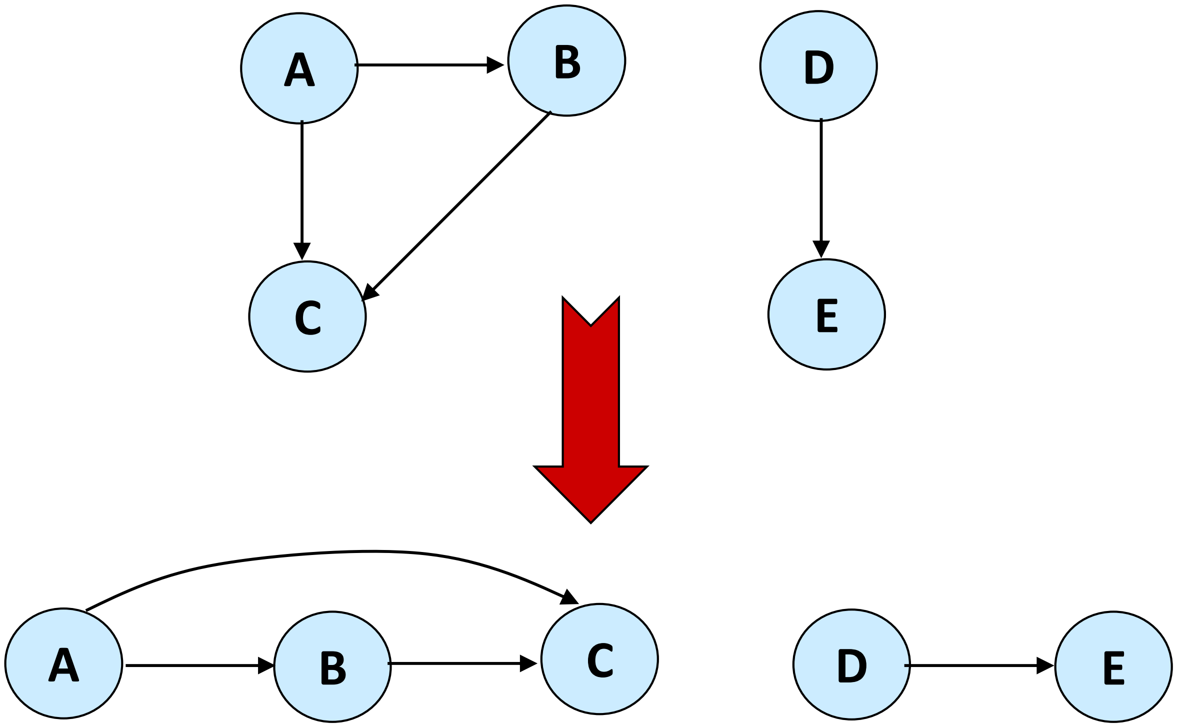
Proof (Contd.):

- $\Leftarrow$: **Show that a cycle implies a back edge.**
  - c : cycle in G, v : first vertex discovered in c, $<u, v>$: v's preceding edge in c.
  - At time $d[v]$, vertices of c form a white path $v \rightsquigarrow u$. Why?
  - By white-path theorem, u is a descendent of v in depth-first forest.
  - Therefore, $<u, v>$ is a back edge.



Topological Sort

Want to “sort” a directed acyclic graph (DAG).



Think of original DAG as a **partial order**.

Want a **total order** that extends this partial order.

Topo-Sort Take One

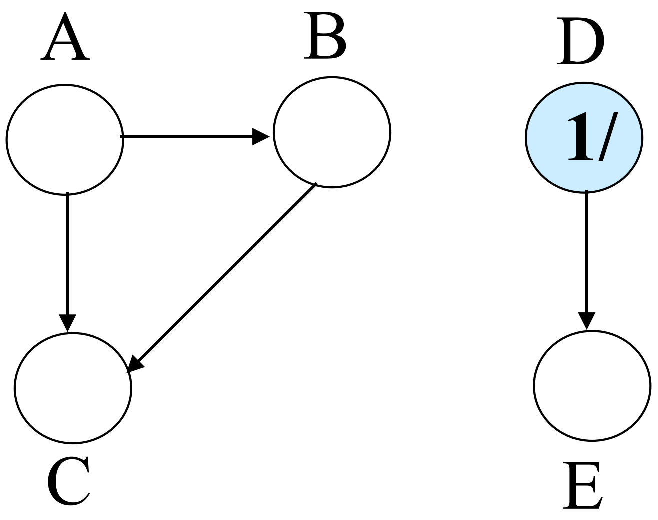
- **Performed on a DAG.**
- **Linear ordering of the vertices of G such that if $\langle u, v \rangle \in E$, then u appears somewhere before v.**

Topological-Sort (G)

1. Call DFS(G) to compute $f[v]$ for all $v \in V$
2. As each vertex is finished, insert it onto the front of a linked list
3. **Return** the linked list of vertices

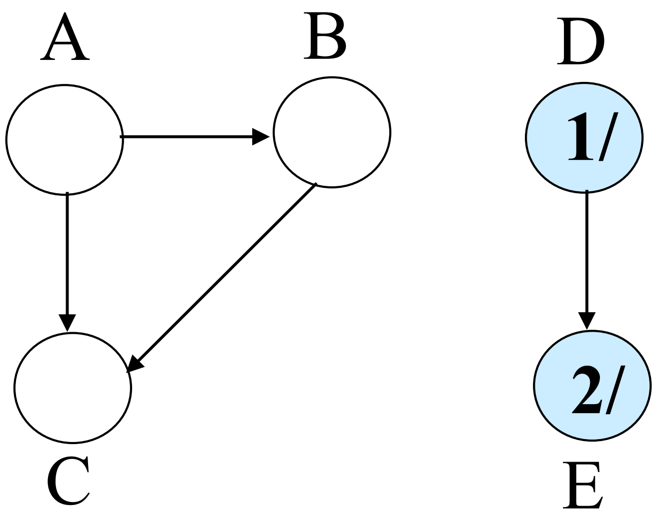
Time: $\Theta(V + E)$.

Example



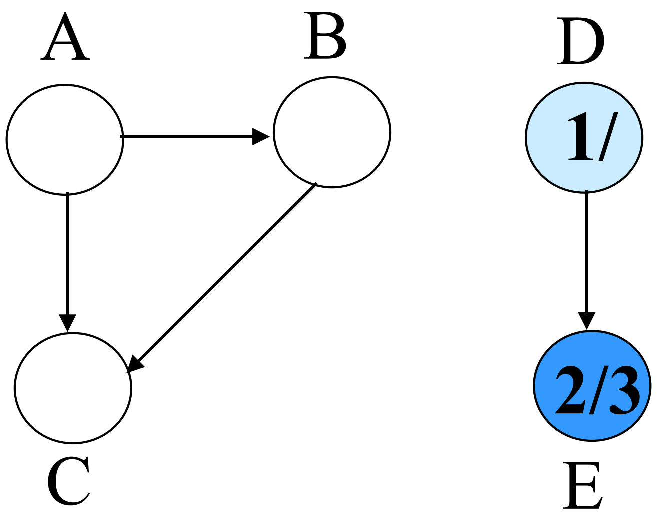
Linked List:

Example

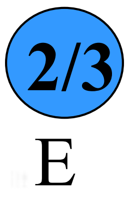


Linked List:

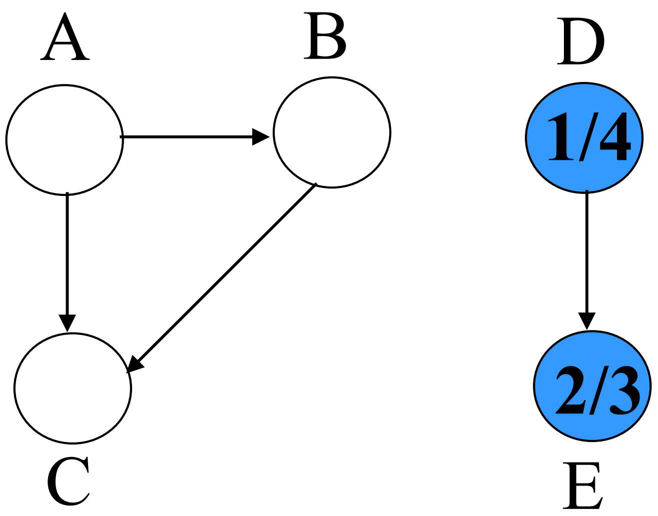
Example



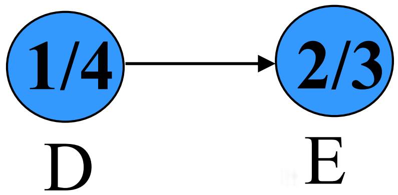
Linked List:



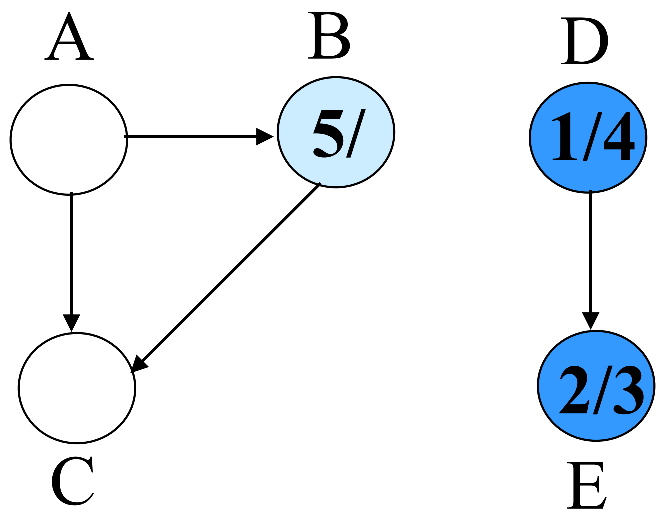
Example



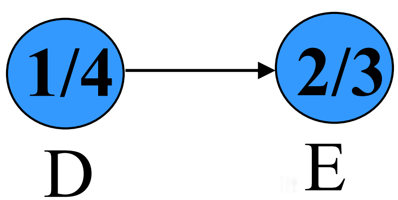
Linked List:



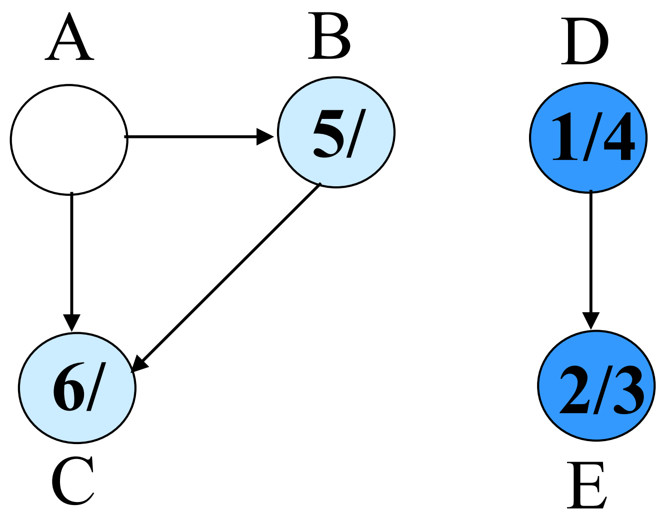
Example



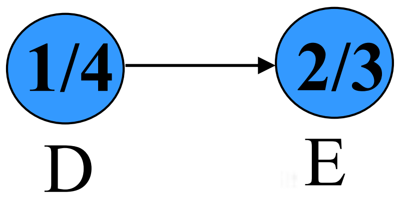
Linked List:



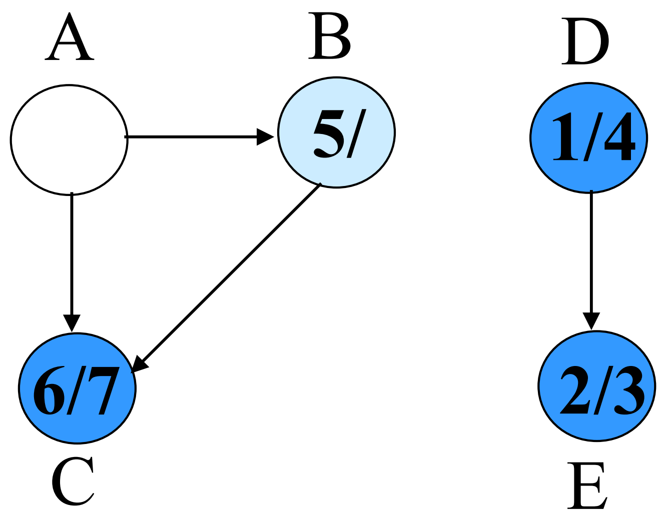
Example



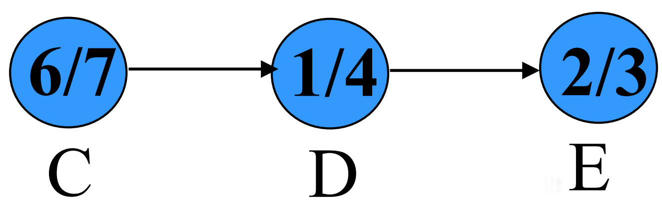
Linked List:



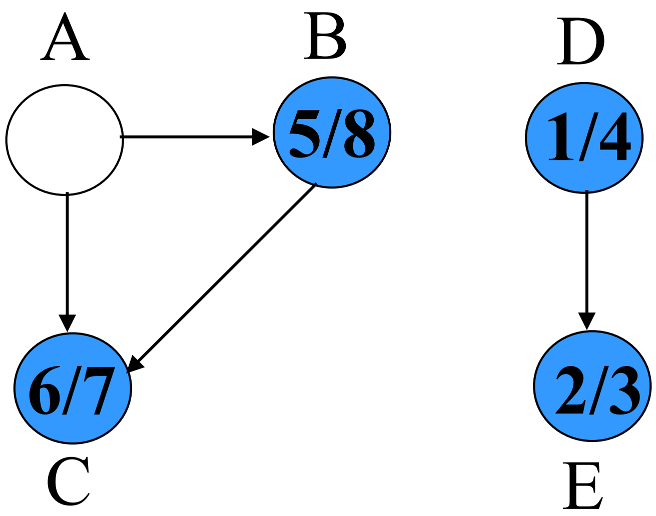
Example



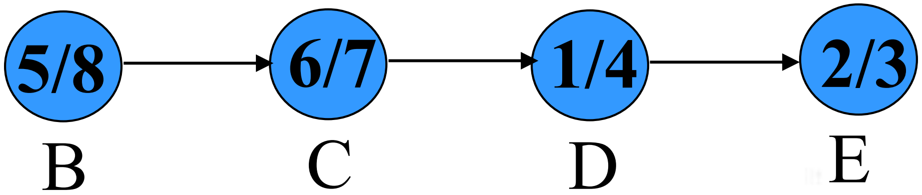
Linked List:



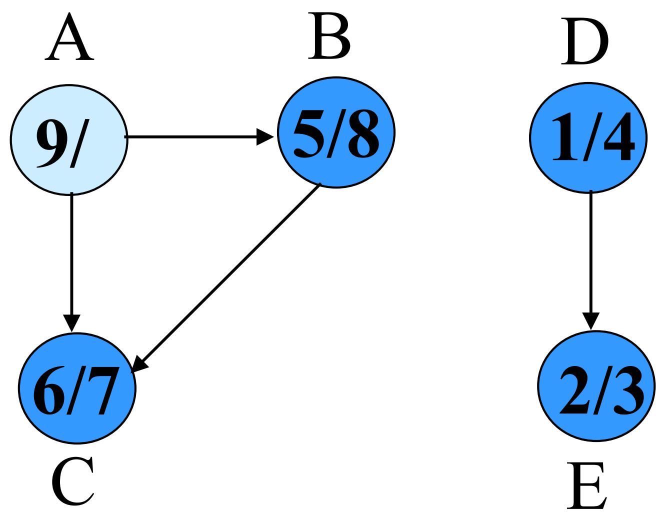
Example



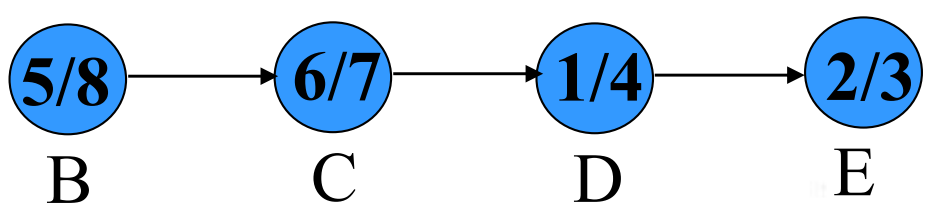
Linked List:



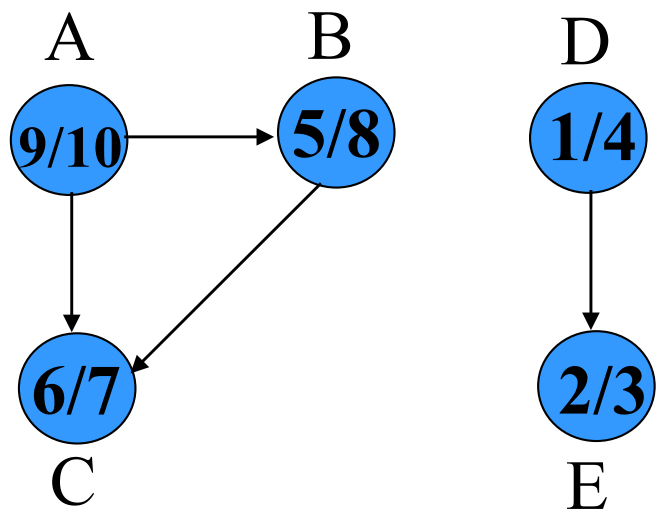
Example



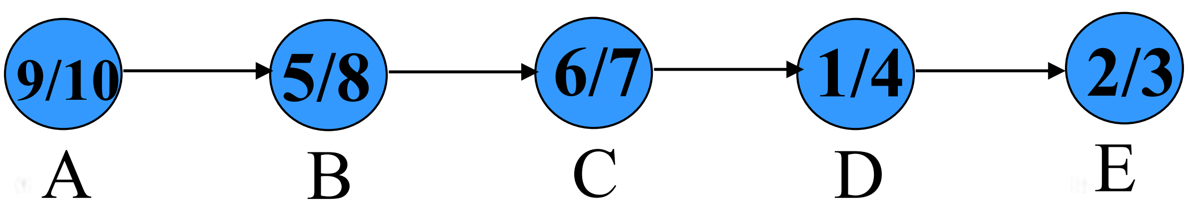
Linked List:



Example



Linked List:



Correctness Proof

- Show if $\langle u, v\rangle \in E$, then $f[v] < f[u]$.
- When we explore $\langle u, v\rangle$, what are the colors of u and v?
  - u is gray.
  - Is v gray, too?
    - *No*, because then v would be ancestor of u.
    - $\Rightarrow \langle u, v\rangle$ is a back edge.
    - $\Rightarrow$ contradiction of Lemma 22.11 (dag has no back edges).
  - Is v white?
    - Then becomes descendant of u.
    - By parenthesis theorem, $d[u] < d[v] < \underline{f[v] < f[u]}$.
  - Is v black?
    - Then v is already finished.
    - Since we're exploring $\langle u, v\rangle$, we have not yet finished u.
    - Therefore, $f[v] < f[u]$.

```
void topsort(Graph* G) {   // Topological sort: recursive
  int i;
  for (i=0; i<G->n(); i++) // Initialize Mark array
    G->setMark(i, UNVISITED);
  for (i=0; i<G->n(); i++) // Process all vertices
    if (G->getMark(i) == UNVISITED)
      tophelp(G, i);       // Call recursive helper function
}

void tophelp(Graph* G, int v) { // Process vertex v
  G->setMark(v, VISITED);
  for (int w=G->first(v); w<G->n(); w = G->next(v,w))
    if (G->getMark(w) == UNVISITED)
      tophelp(G, w);
  printout(v);                  // PostVisit for Vertex v
}
```

Topo-Sort Take Two

- **Label each vertex's *in-degree* (# of inbound edges)**

- **While there are vertices remaining**
  - **Pick a vertex with in-degree of zero and output it**
  - **Reduce the in-degree of all vertices adjacent to it**
  - **Remove it from the list of vertices**

Runtime?

```
// Topological sort: Queue
void topsort(Graph* G, Queue<int>* Q) {
  int Count[G->n()];
  int v, w;
  for (v=0; v<G->n(); v++) Count[v] = 0; // Initialize
  for (v=0; v<G->n(); v++)   // Process every edge
    for (w=G->first(v); w<G->n(); w = G->next(v,w))
       Count[w]++;             // Add to v2's prereq count
  for (v=0; v<G->n(); v++)   // Initialize queue
    if (Count[v] == 0)         // Vertex has no prerequisites
      Q->enqueue(v);
  while (Q->length() != 0) { // Process the vertices
    v = Q->dequeue();
    printout(v);               // PreVisit for "v"
    for (w=G->first(v); w<G->n(); w = G->next(v,w)) {
      Count[w]--;              // One less prerequisite
      if (Count[w] == 0)       // This vertex is now free
        Q->enqueue(w);
    }
  }
}
```

数据结构与算法课程组

重庆大学计算机学院

End of Section.