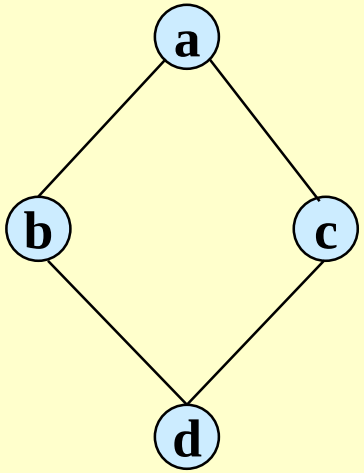
# Elementary Graph Algorithms

- Graph representation
- Graph traversal
  - Breadth-first search
  - Depth-first search
- Parenthesis theorem

# Graphs

- *Graph* $G = (V, E)$
  - » $V$ = set of vertices
  - » $E$ = set of edges $\subseteq (V \times V)$




$V = \{a, b, c, d\}$

$E = \{(a, b), (a, c), (b, d), (c, d)\}$

# Graphs

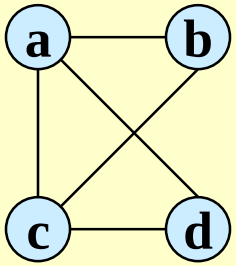
- Types of graphs
  - » Undirected: edge $(u, v) = (v, u)$; for all $v$, $(v, v) \notin E$ (No self loops.)
  - » Directed: $(u, v)$ is edge from $u$ to $v$, denoted as $u \rightarrow v$. Self loops are allowed.
  - » Weighted: each edge has an associated weight, given by a weight function $w : E \rightarrow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

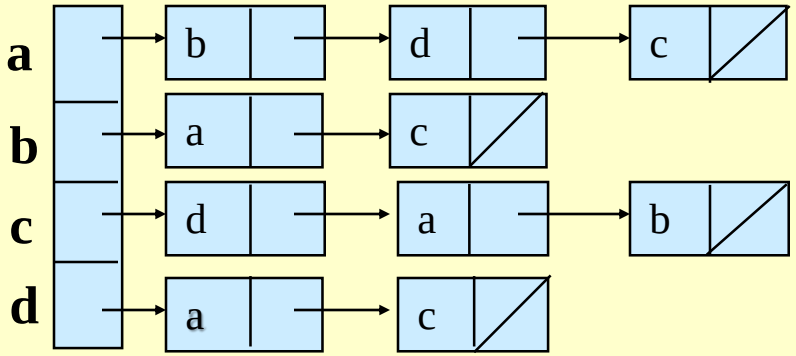
# Graphs

- 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 an undirected graph $G$ 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*.
  - If a directed graph $G$ is connected:
  - » Its undirected version is connected.
  - Other definitions in Appendix B (B.4 and B.5) as needed.

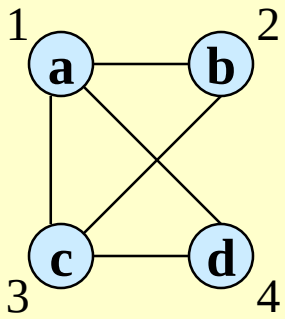
# Representation of Graphs

- Two standard ways.
  - » Adjacency Lists.



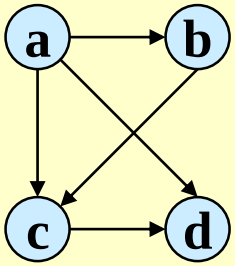

  - » Adjacency Matrix.

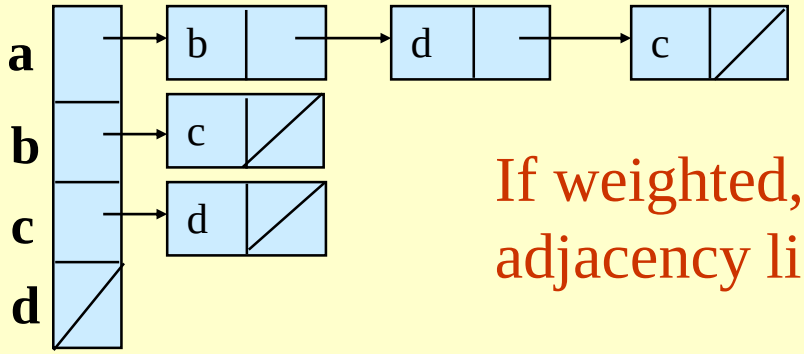



| | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | 0 | 1 | 1 | 1 |
| 2 | 1 | 0 | 1 | 0 |
| 3 | 1 | 1 | 0 | 1 |
| 4 | 1 | 0 | 1 | 0 |

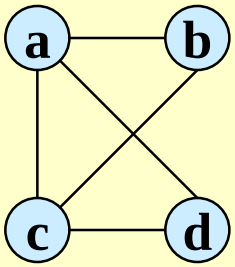
# Adjacency Lists

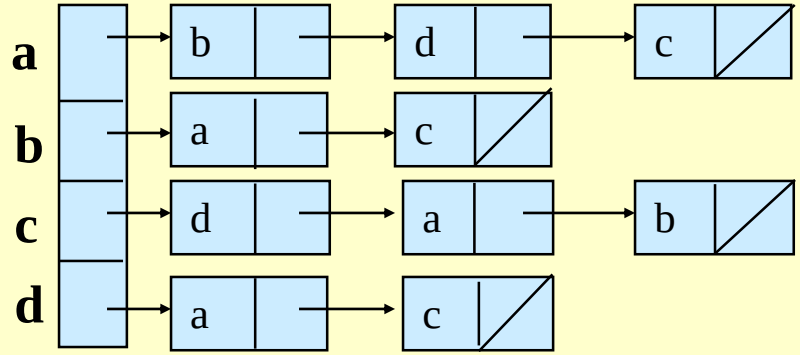
- Consists of an array *Adj* of $|V|$ lists.
- One list per vertex.
- For $u \in V$, *Adj*[*u*] consists of all vertices adjacent to *u*.




If weighted, store weights also in adjacency lists.



# Storage Requirement

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

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

No. of edges leaving $v$

  - » Total storage: $\Theta(|V| + |E|)$
- For undirected graphs:
  - » Sum of lengths of all adj. lists is

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

No. of edges incident on $v$. Edge $(u,v)$ is incident on vertices $u$ and $v$.

  - » Total storage: $\Theta(|V| + |E|)$

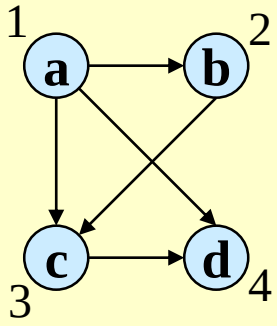
# Pros and Cons: adj list

- 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. $\Theta(\text{degree}(u))$ time.
    - $\Theta(|V|)$ in the worst case.

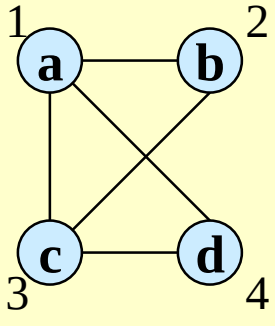
# Adjacency Matrix

- $|V| \times |V|$ 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 |




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

$A = A^T$ 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 instead of bits for weighted graph.

# Sparse Matrix

- Sparse matrices are typically stored in a format, or a representation, which avoids storing zero elements.
- CSR (Compressed Sparse Row) - store only non-zero values in a a one-dimentional data storage: data[ ].
- Two auxiliary data structures col_index[ ] and row_ptr[ ] to preserve the stucture of the original sparse matrix in the compressed representation.
- col_index[ ] gives the column index of every nonzero value in the original sparse matrix.
- Row_ptr[ ] indicates the starting nonzero location of every row in the compressed format.

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

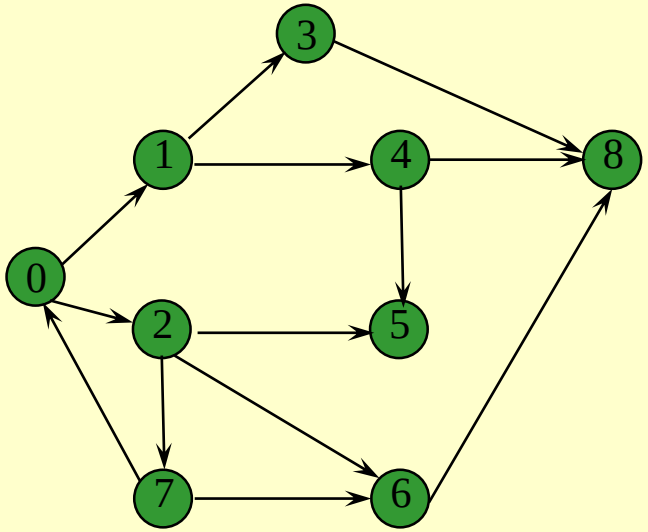
# Sparse Matrix

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

| | row0 | row2 | | | row3 | |
|---|---|---|---|---|---|---|
| Nonzero values data[ ] | {3 | 1 | 2 | 4 | 1 | 1 | 1} |
| Column indeces col_index[ ] | {0 | 2 | 1 | 2 | 3 | 0 | 3} |
| Row pointers row_ptr[ ] | {0 | 2 | 2 | 5 | 7} | | |

- In data[ ], value 3 and 1 came from column 0 and 2 in the original sparse matrix. The col_index[0] and col_index[1] elements are assigned to store the column indices for these two values. For another example, values 2, 4, and 1 came from column 1, 2, and 3 of row 2 in the original sparse matrix. Therefore, col_index[2], col_index[3], and col_index[4] store indices 1, 2, and 3.
- In row_ptr[ ], the values are the indices for the beginning locations of each row. For example, row_ptr[0] = 0 indicates the row 0 starts at location 0 of data[ ]. row_ptr[2] = 2 indicates the row 2 starts at location 2 of data[ ]. But we notice that row_ptr{1] is set to be 2, equal to row_ptr[2], showing all elements in row 1 in the original matrx are 0. Finally, row_ptr[4] stores the starting location of a non-existing ‘row 4’. (This choice is the convenience, as some algorithms need to use the starting location of the next row to delineate the end of the current row.)

# Sparse Graph




| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| 2 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 |
| 3 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 |
| 4 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 |
| 5 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| 6 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 7 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| 8 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Nonzero values data[ ] 1 1 1 1 1 1 1 1 1 1 1 1 1 1

destination[ ] 1 2 3 4 5 6 7 4 8 5 8 6 8 0 8

edges[ ] 0 2 4 7 9 11 12 13 15 16

# Graph-searching Algorithms

- Searching a graph:
  - » Systematically follow the edges of a graph to visit the vertices of the graph.
- Used to discover the structure of a graph.
- Standard graph-searching algorithms.
  - » Breadth-first Search (BFS).
  - » Depth-first Search (DFS).

# Breadth-first Search

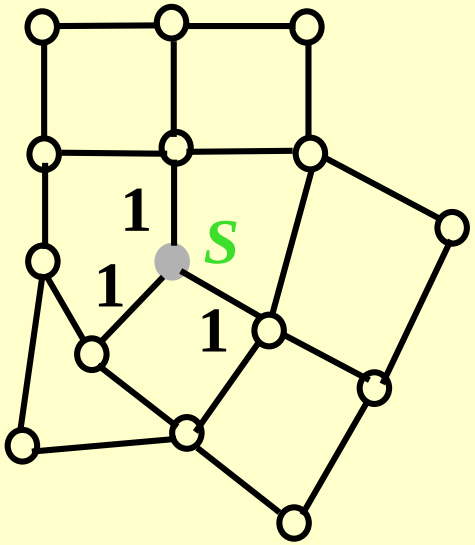
- **Input:** Graph $G = (V, E)$, either directed or undirected, and ***source vertex*** $s \in V$.
- **Output:**
  - » $d[v]$ = distance (smallest # of edges, or shortest path) from $s$ to $v$, for all $v \in V$. $d[v] = \infty$ if $v$ is not reachable from $s$.
  - » $\pi[v] = u$ such that $(u, v)$ is last edge on shortest path $s \rightsquigarrow v$.
    - $u$ is $v$'s predecessor.
  - » Builds breadth-first tree with root $s$ that contains all reachable vertices.

Definitions:
Path between vertices $u$ and $v$: Sequence of vertices $(v_1, v_2, \ldots, v_k)$ such that $u = v_1$ and $v = v_k$, and $(v_i, v_{i+1}) \in E$, for all $1 \le i \le k$-1.
Length of the path: Number of edges in the path.
Path is simple if no vertex is repeated.

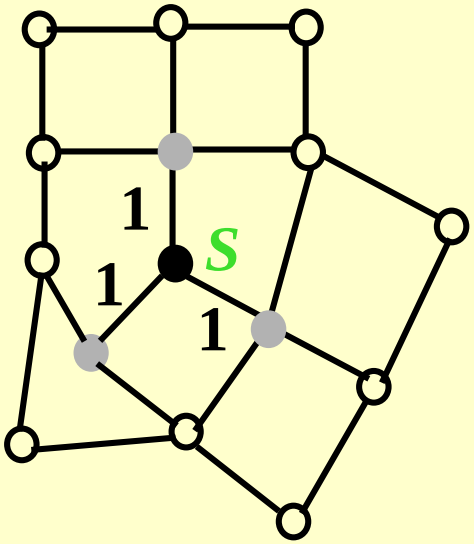
# Breadth-first Search

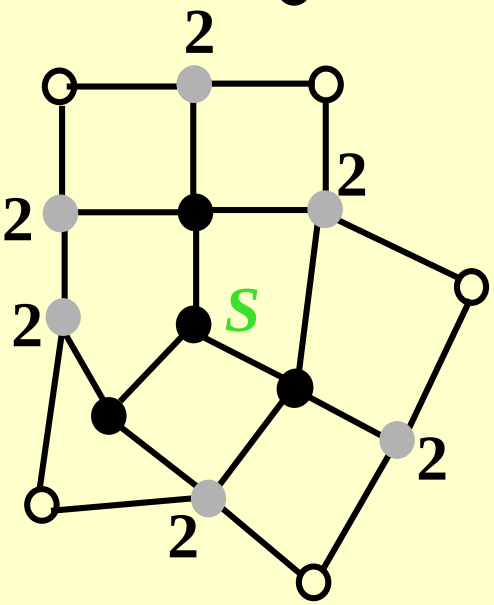
- Expands the frontier between discovered and undiscovered vertices uniformly across the breadth of the frontier.
  - » A vertex is "discovered" the first time it is encountered during the search.
  - » A vertex is "finished" if all vertices adjacent to it have been discovered.
- Colors the vertices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

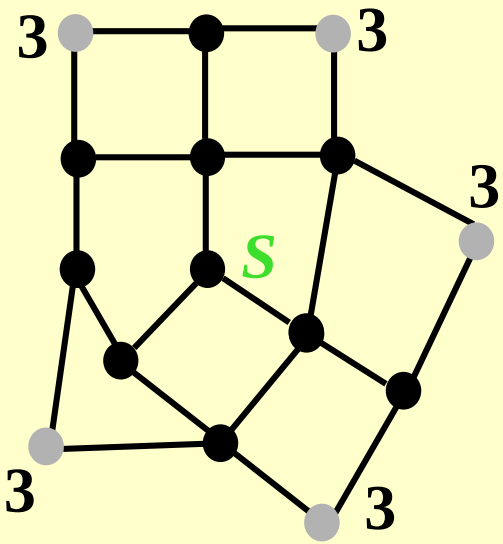
# BFS for Shortest Paths




- **Finished**
- **Discovered**
- **Undiscovered**



**BFS(G,s)**

```
1.  for each vertex u in V[G] – {s}
2.        do color[u] ← white
3.           d[u] ← ∝
4.           π[u] ← nil
5.  color[s] ← gray
6.  d[s] ← 0
7.   π[s] ← nil
8.  Q ← Φ
9.  enqueue(Q, s)
10. while Q ≠ Φ
11.       do u ← dequeue(Q)
12.       for each v in Adj[u] do
13.          if color[v] = white
14.          then color[v] ← gray
15.                d[v] ← d[u] + 1
16.                π[v] ← u
17.                enqueue(Q, v)
18.       color[u] ← black
```

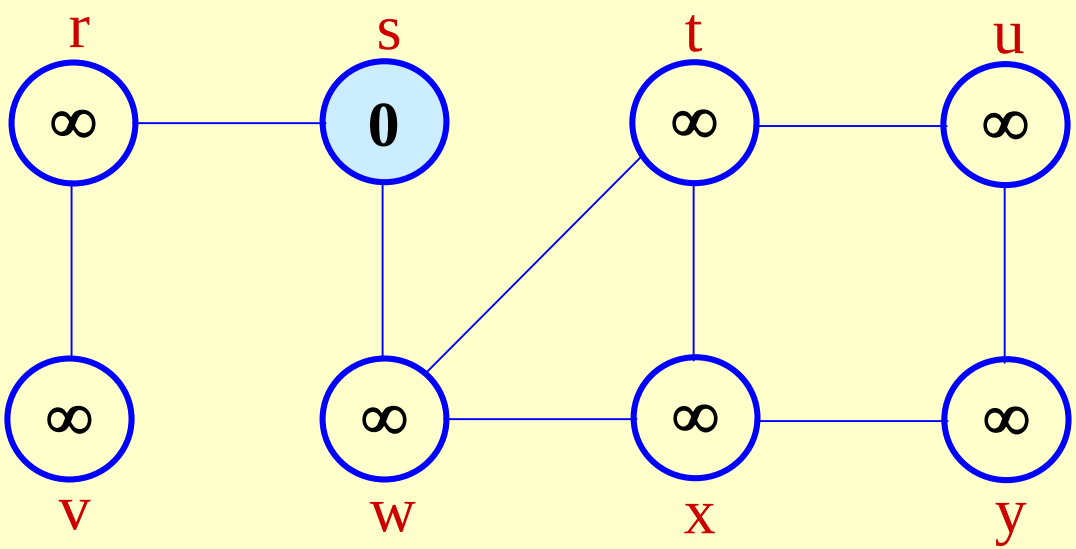
initialization

access source *s*

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)

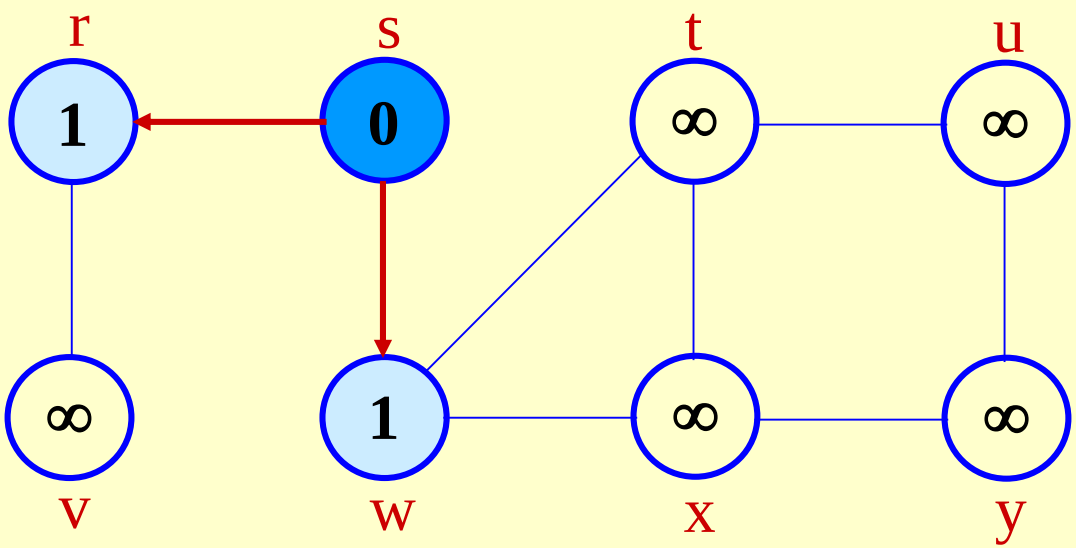



**Q:** s
0

**BFS(G,s)**

```
1.  for each vertex u in V[G] – {s}
2         do color[u] ← white
3            d[u] ← ∝
4            π[u] ← nil
5   color[s] ← gray
6   d[s] ← 0
7    π[s] ← nil
8   Q ← Φ
9   enqueue(Q, s)
10  while Q ≠ Φ
11        do u ← dequeue(Q)
12        for each v in Adj[u] do
13           if color[v] = white
14           then color[v] ← gray
15                d[v] ← d[u] + 1
16                π[v] ← u
17                enqueue(Q, v)
18        color[u] ← black
```

# Example (BFS)

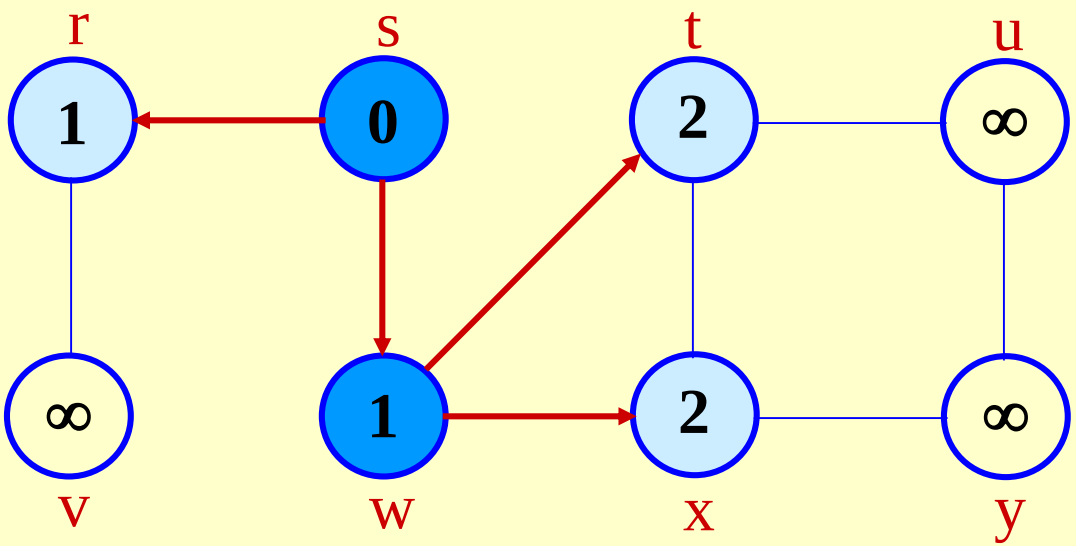



**Q:** w r
1 1

**BFS(G,s)**

```
1.  for each vertex u in V[G] – {s}
2       do color[u] ← white
3          d[u] ← ∞
4          π[u] ← nil
5   color[s] ← gray
6   d[s] ← 0
7    π[s] ← nil
8   Q ← Φ
9   enqueue(Q, s)
10  while Q ≠ Φ
11      do u ← dequeue(Q)
12      for each v in Adj[u] do
13          if color[v] = white
14          then color[v] ← gray
15               d[v] ← d[u] + 1
16               π[v] ← u
17               enqueue(Q, v)
18      color[u] ← black
```

# Example (BFS)

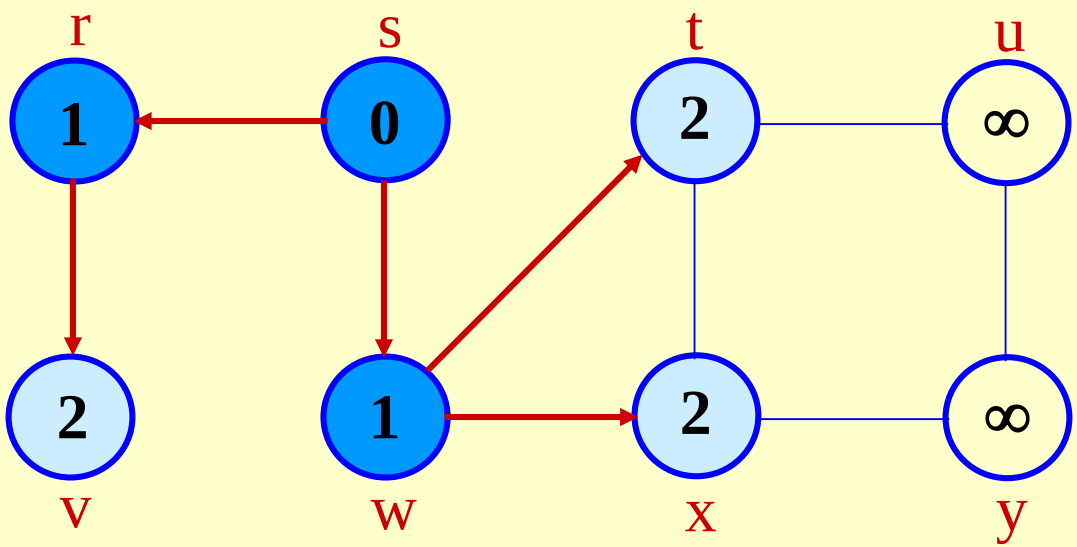



| **Q:** | r | t | x |
|---|---|---|---|
| | 1 | 2 | 2 |

**BFS(G,s)**

```
1.  for each vertex u in V[G] – {s}
2         do color[u] ← white
3            d[u] ← ∝
4            π[u] ← nil
5   color[s] ← gray
6   d[s] ← 0
7    π[s] ← nil
8   Q ← Φ
9   enqueue(Q, s)
10  while Q ≠ Φ
11        do u ← dequeue(Q)
12        for each v in Adj[u] do
13           if color[v] = white
14           then color[v] ← gray
15                d[v] ← d[u] + 1
16                π[v] ← u
17                enqueue(Q, v)
18        color[u] ← black
```

# Example (BFS)

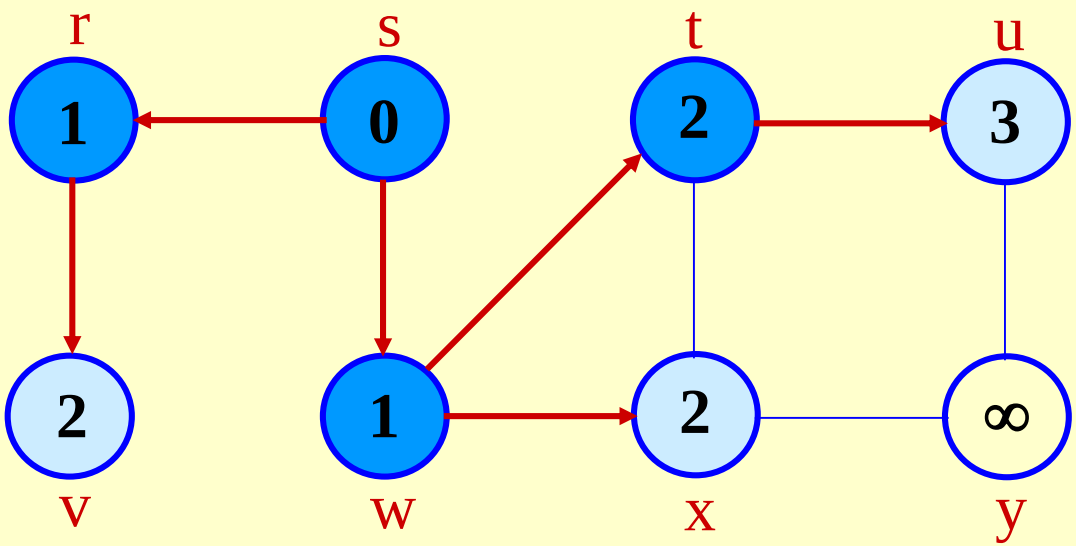



| **Q:** | t | x | v |
|---|---|---|---|
| | 2 | 2 | 2 |

**BFS(G,s)**

```
1.  for each vertex u in V[G] – {s}
2         do color[u] ← white
3            d[u] ← ∞
4            π[u] ← nil
5   color[s] ← gray
6   d[s] ← 0
7    π[s] ← nil
8   Q ← Φ
9   enqueue(Q, s)
10  while Q ≠ Φ
11        do u ← dequeue(Q)
12        for each v in Adj[u] do
13           if color[v] = white
14           then color[v] ← gray
15                d[v] ← d[u] + 1
16                π[v] ← u
17                enqueue(Q, v)
18        color[u] ← black
```

# Example (BFS)

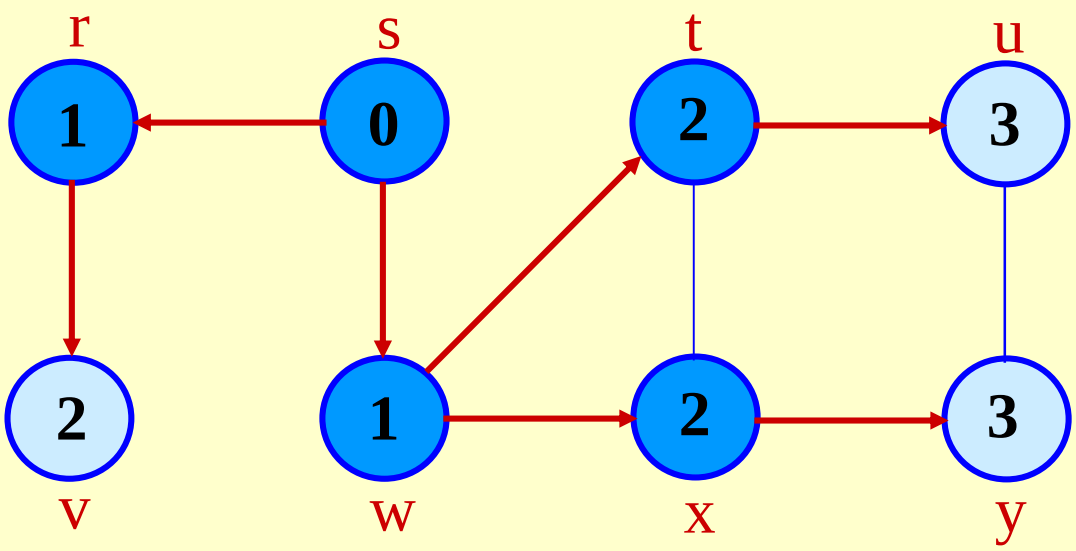



**Q:** x v u
2 2 3

**BFS(G,s)**

```
1.  for each vertex u in V[G] – {s}
2       do color[u] ← white
3          d[u] ← ∞
4          π[u] ← nil
5   color[s] ← gray
6   d[s] ← 0
7   π[s] ← nil
8   Q ← Φ
9   enqueue(Q, s)
10  while Q ≠ Φ
11      do u ← dequeue(Q)
12      for each v in Adj[u] do
13          if color[v] = white
14          then color[v] ← gray
15               d[v] ← d[u] + 1
16               π[v] ← u
17               enqueue(Q, v)
18      color[u] ← black
```

# Example (BFS)

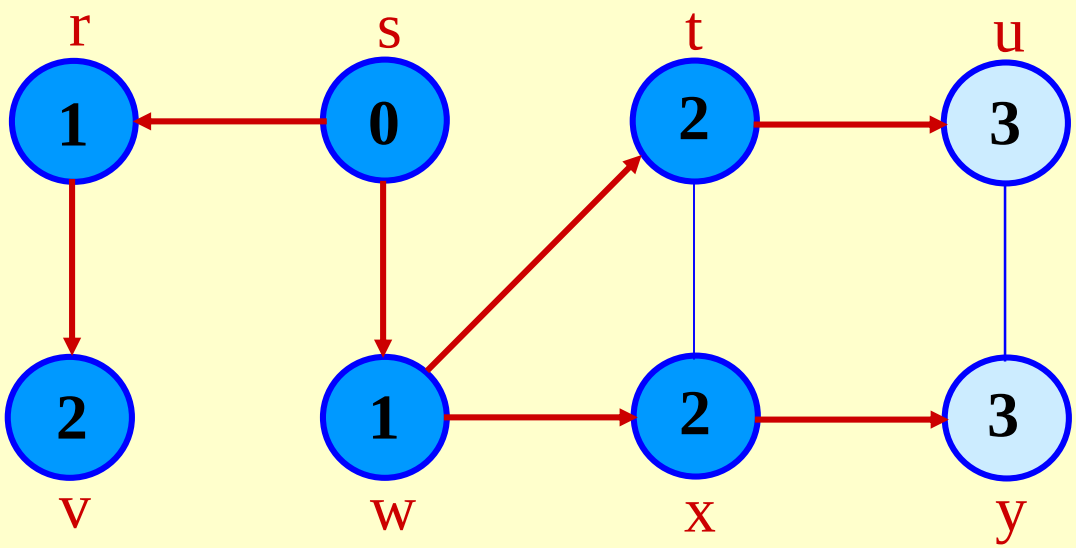



**Q:** v u y
2 3 3

**BFS(G,s)**

```
1.  for each vertex u in V[G] – {s}
2       do color[u] ← white
3          d[u] ← ∞
4          π[u] ← nil
5   color[s] ← gray
6   d[s] ← 0
7    π[s] ← nil
8   Q ← Φ
9   enqueue(Q, s)
10  while Q ≠ Φ
11      do u ← dequeue(Q)
12      for each v in Adj[u] do
13          if color[v] = white
14          then color[v] ← gray
15               d[v] ← d[u] + 1
16               π[v] ← u
17               enqueue(Q, v)
18      color[u] ← black
```

# Example (BFS)

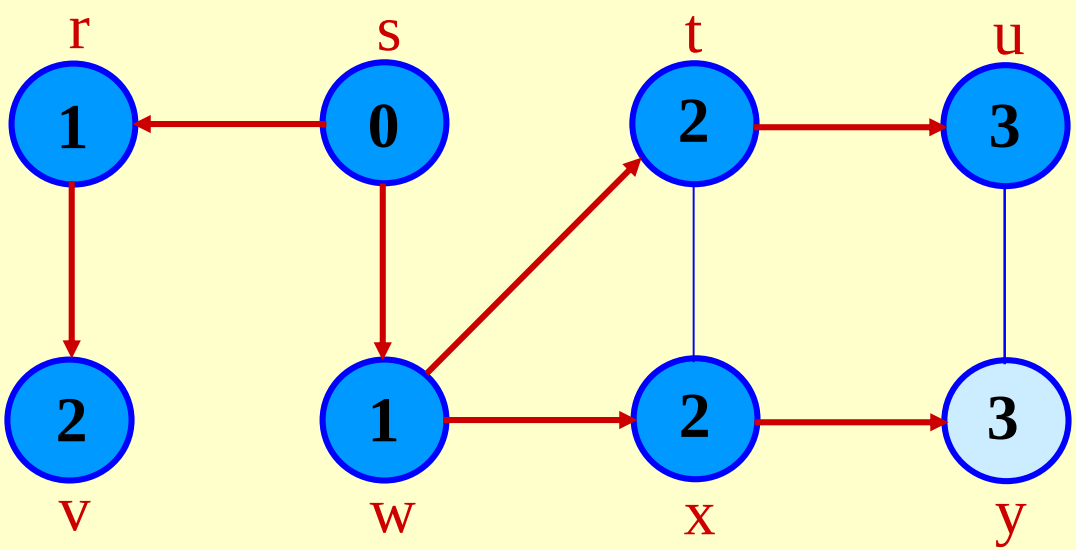



**Q:** u y
3 3

**BFS(G,s)**

```
1.  for each vertex u in V[G] – {s}
2       do color[u] ← white
3          d[u] ← ∝
4          π[u] ← nil
5   color[s] ← gray
6   d[s] ← 0
7    π[s] ← nil
8   Q ← Φ
9   enqueue(Q, s)
10  while Q ≠ Φ
11      do u ← dequeue(Q)
12      for each v in Adj[u] do
13          if color[v] = white
14          then color[v] ← gray
15               d[v] ← d[u] + 1
16               π[v] ← u
17               enqueue(Q, v)
18      color[u] ← black
```

# Example (BFS)

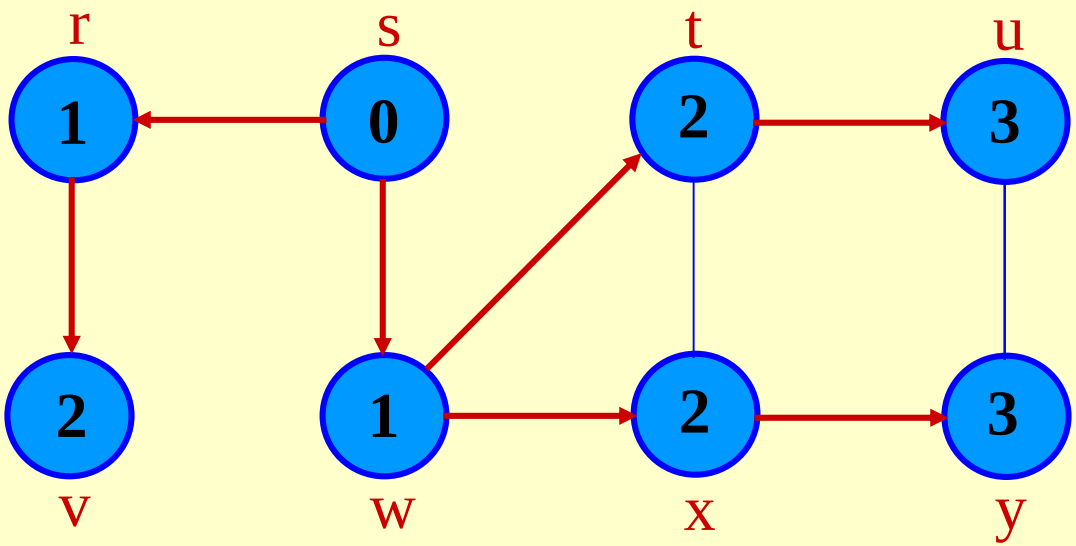



**Q:** y
3

**BFS(G,s)**

```
1.  for each vertex u in V[G] – {s}
2       do color[u] ← white
3          d[u] ← ∞
4          π[u] ← nil
5   color[s] ← gray
6   d[s] ← 0
7    π[s] ← nil
8   Q ← Φ
9   enqueue(Q, s)
10  while Q ≠ Φ
11      do u ← dequeue(Q)
12      for each v in Adj[u] do
13          if color[v] = white
14          then color[v] ← gray
15               d[v] ← d[u] + 1
16               π[v] ← u
17               enqueue(Q, v)
18      color[u] ← black
```

# Example (BFS)

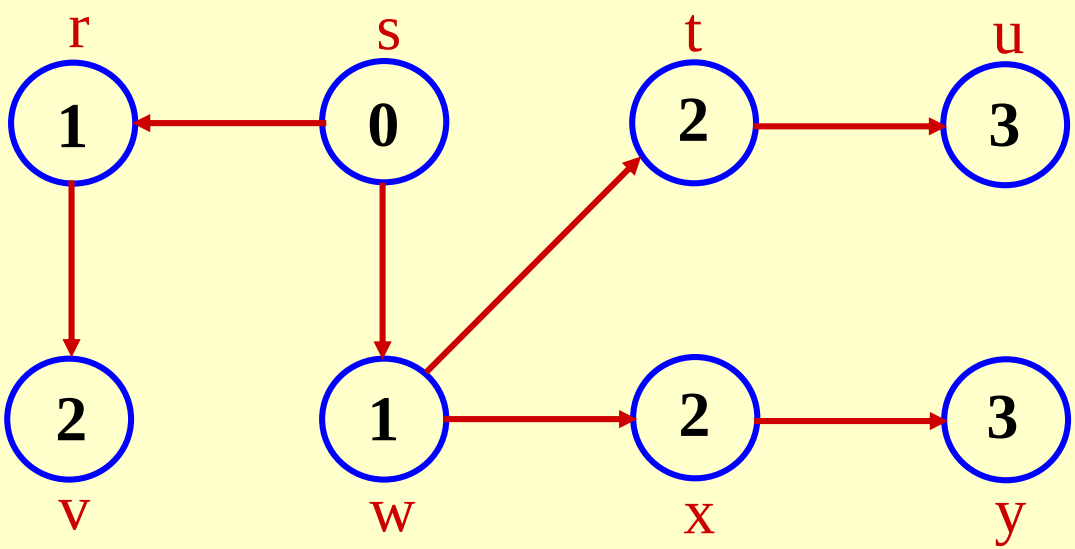



**Q:** ∅

**BFS(G,s)**

```
1.  for each vertex u in V[G] – {s}
2         do color[u] ← white
3            d[u] ← ∝
4            π[u] ← nil
5   color[s] ← gray
6   d[s] ← 0
7    π[s] ← nil
8   Q ← Φ
9   enqueue(Q, s)
10  while Q ≠ Φ
11        do u ← dequeue(Q)
12        for each v in Adj[u] do
13           if color[v] = white
14           then color[v] ← gray
15                d[v] ← d[u] + 1
16                π[v] ← u
17                enqueue(Q, v)
18        color[u] ← black
```

# Example (BFS)




**BF Tree**

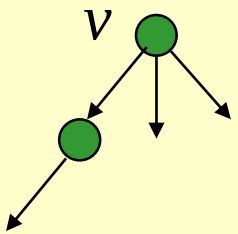
# Analysis of BFS

- Initialization takes $O(|V|)$.
- Traversal Loop
  - » 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 at most once. The sum of lengths of all adjacency lists is $\Theta(|E|)$.
- 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.
- Correctness Proof
  - » We omit for BFS and DFS.
  - » Will do for later algorithms.

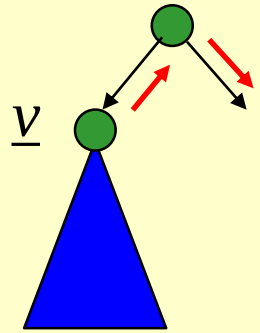
# Breadth-first Tree

- 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**. $|E_\pi| = |V_\pi| - 1$.

# Depth-first Search (DFS)




- 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.

# Depth-first Search

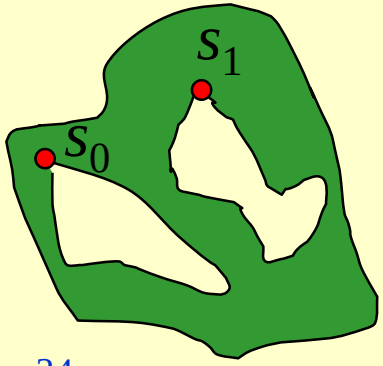
- **Input:** $G = (V, E)$, directed or undirected. No source vertex given!
- **Output:**
  - » 2 **timestamps** on each vertex. Integers between 1 and 2|V|.
    - $d[v]$ = ***discovery time*** ($v$ turns from white to gray)
    - $f[v]$ = ***finishing time*** ($v$ turns from gray to black)
  - » $\pi[v]$ : predecessor of $v = u$, such that $v$ was discovered during the scan of $u$'s adjacency list.
- Coloring scheme for vertices as BFS. A vertex is
  - » "undiscovered" (white) when it is not yet encountered.
  - » "discovered" (grey) the first time it is encountered during the search.
  - » "finished" (black) if it is a leaf node or all vertices adjacent to it have been finished.

# Pseudo-code

**DFS(*G*)**

```
1. for each vertex u ∈ V[G]
2.     do color[u] ← white
3.        π[u] ← NIL
4. time ← 0
5. for each vertex u ∈ V[G]
6.     do if color[u] = white
7.        then DFS-Visit(u)
```

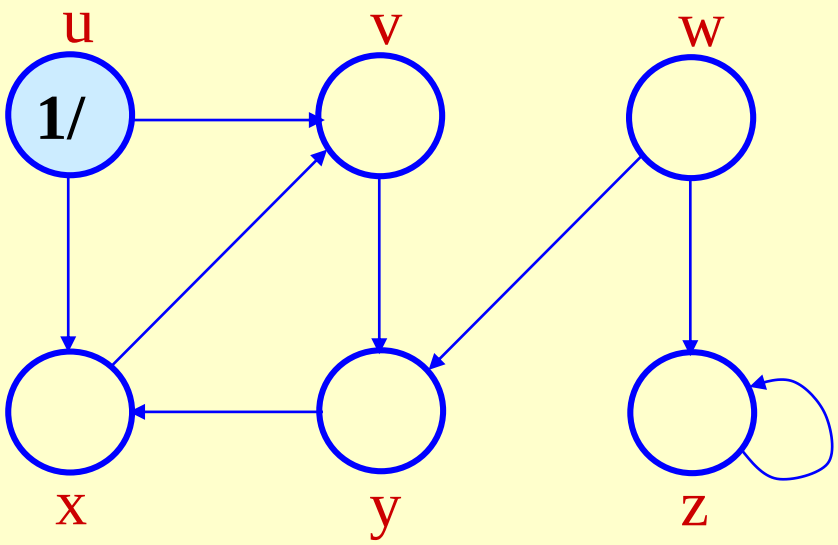
Uses a global timestamp ***time***.




**DFS-Visit(*u*)**

```
1.  color[u] ← GRAY  // White vertex u
                     has been discovered
2.  time ← time + 1
3.  d[u] ← time
4.  for each v ∈ Adj[u]
5.      do if color[v] = WHITE
6.         then π[v] ← u
7.              DFS-Visit(v)
8.  color[u] ← BLACK    // Blacken u;
                        it is finished.
9.  f[u] ← time ← time + 1
```

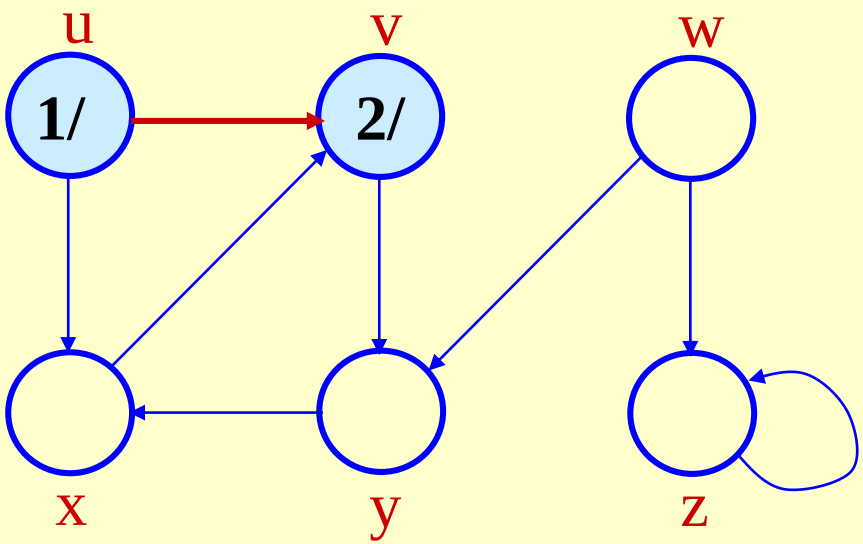
# Example (DFS)




**DFS-Visit(*u*)**

1. *color*[*u*] ← GRAY // White vertex *u* has been discovered
2. *time* ← *time* + 1
3. *d*[*u*] ← *time*
4. **for** each *v* ∈ *Adj*[*u*]
5. **do if** *color*[*v*] = WHITE
6. **then** π[*v*] ← *u*
7. DFS-Visit(*v*)
8. *color*[*u*] ← BLACK // Blacken *u*; it is finished.
9. *f*[*u*] ← *time* ← *time* + 1

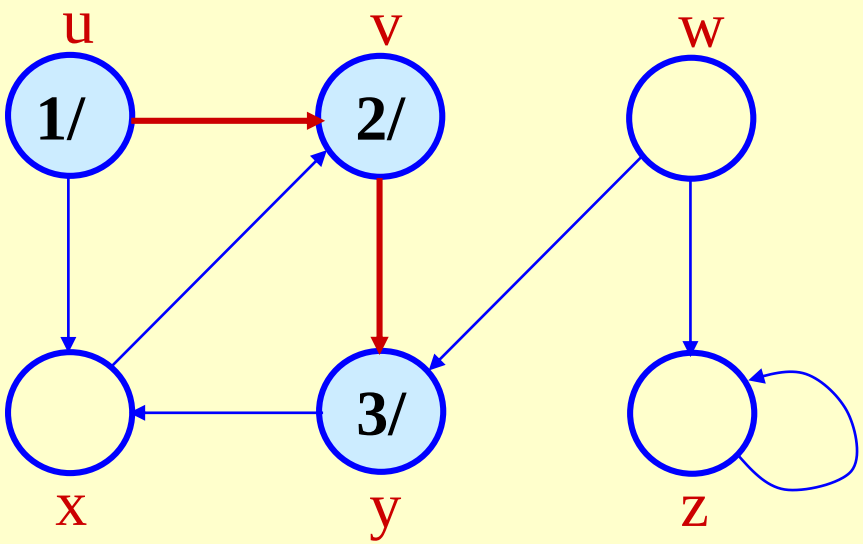
# Example (DFS)




**DFS-Visit(*u*)**

1. *color*[*u*] ← GRAY // White vertex *u* has been discovered
2. *time* ← *time* + 1
3. *d*[*u*] ← *time*
4. **for** each *v* ∈ *Adj*[*u*]
5. **do if** *color*[*v*] = WHITE
6. **then** π[*v*] ← *u*
7. DFS-Visit(*v*)
8. *color*[*u*] ← BLACK // Blacken *u*; it is finished.
9. *f*[*u*] ← *time* ← *time* + 1

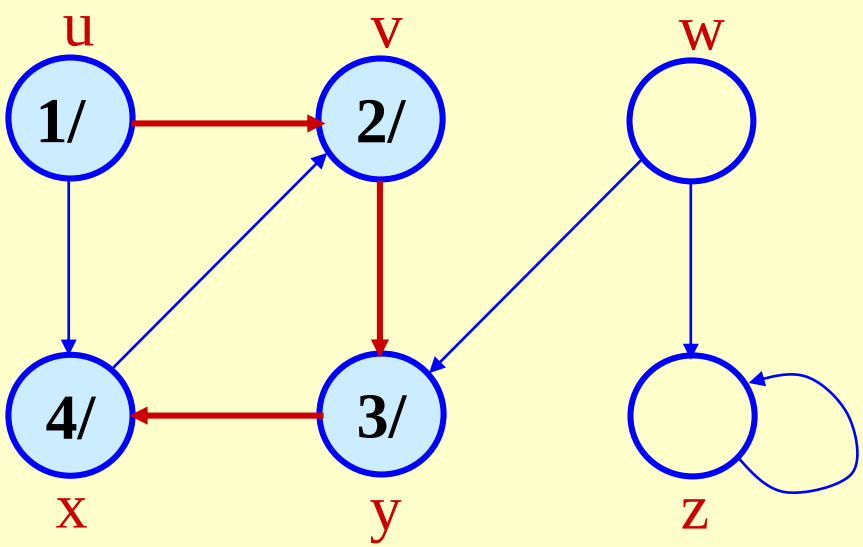
# Example (DFS)




**DFS-Visit(*u*)**

1. *color*[*u*] ← GRAY // White vertex *u* has been discovered
2. *time* ← *time* + 1
3. *d*[*u*] ← *time*
4. **for** each *v* ∈ *Adj*[*u*]
5. **do if** *color*[*v*] = WHITE
6. **then** π[*v*] ← *u*
7. DFS-Visit(*v*)
8. *color*[*u*] ← BLACK // Blacken *u*; it is finished.
9. *f*[*u*] ← *time* ← *time* + 1

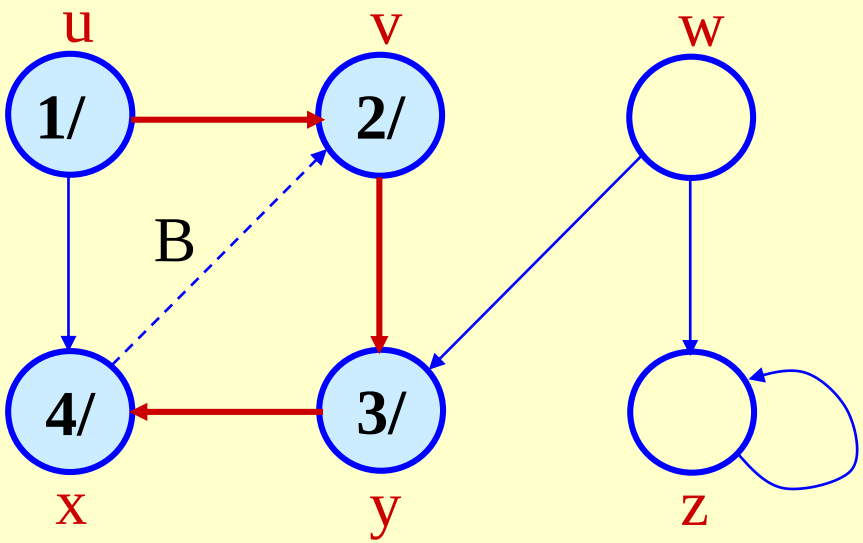
# Example (DFS)




**DFS-Visit(*u*)**

1. *color*[*u*] ← GRAY // White vertex *u* has been discovered
2. *time* ← *time* + 1
3. *d*[*u*] ← *time*
4. **for** each *v* ∈ *Adj*[*u*]
5. **do if** *color*[*v*] = WHITE
6. **then** π[*v*] ← *u*
7. DFS-Visit(*v*)
8. *color*[*u*] ← BLACK // Blacken *u*; it is finished.
9. *f*[*u*] ← *time* ← *time* + 1

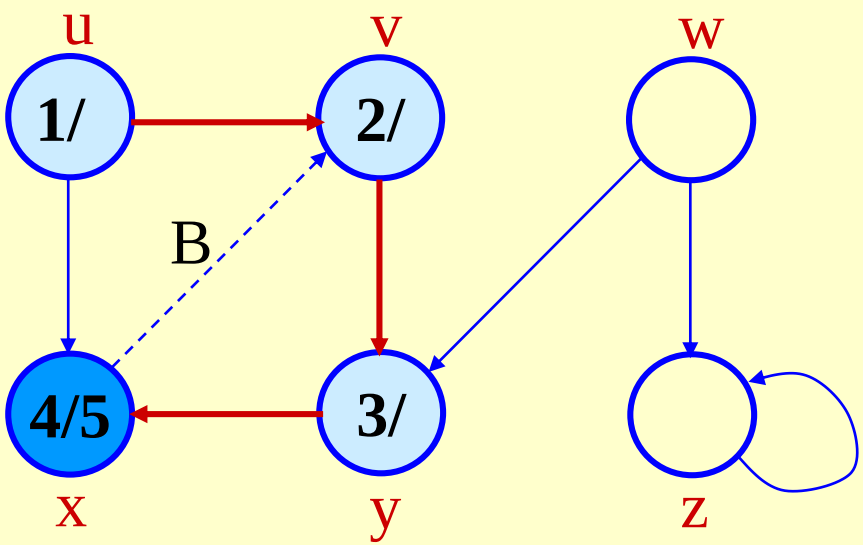
# Example (DFS)




**DFS-Visit(*u*)**

1. *color*[*u*] ← GRAY // White vertex *u* has been discovered
2. *time* ← *time* + 1
3. *d*[*u*] ← *time*
4. **for** each *v* ∈ *Adj*[*u*]
5. **do if** *color*[*v*] = WHITE
6. **then** π[*v*] ← *u*
7. DFS-Visit(*v*)
8. *color*[*u*] ← BLACK // Blacken *u*; it is finished.
9. *f*[*u*] ← *time* ← *time* + 1

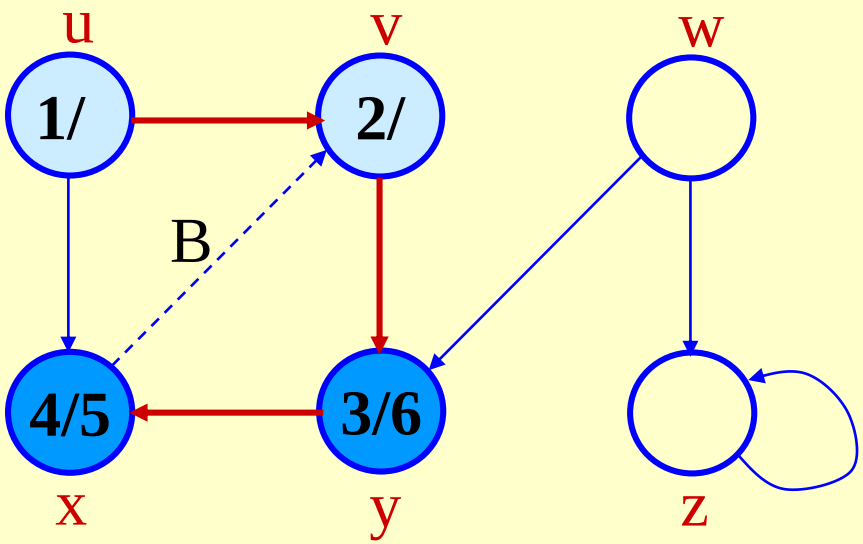
# Example (DFS)




**DFS-Visit(*u*)**

1. *color*[*u*] ← GRAY // White vertex *u* has been discovered
2. *time* ← *time* + 1
3. *d*[*u*] ← *time*
4. **for** each *v* ∈ *Adj*[*u*]
5. **do if** *color*[*v*] = WHITE
6. **then** π[*v*] ← *u*
7. DFS-Visit(*v*)
8. *color*[*u*] ← BLACK // Blacken *u*; it is finished.
9. *f*[*u*] ← *time* ← *time* + 1

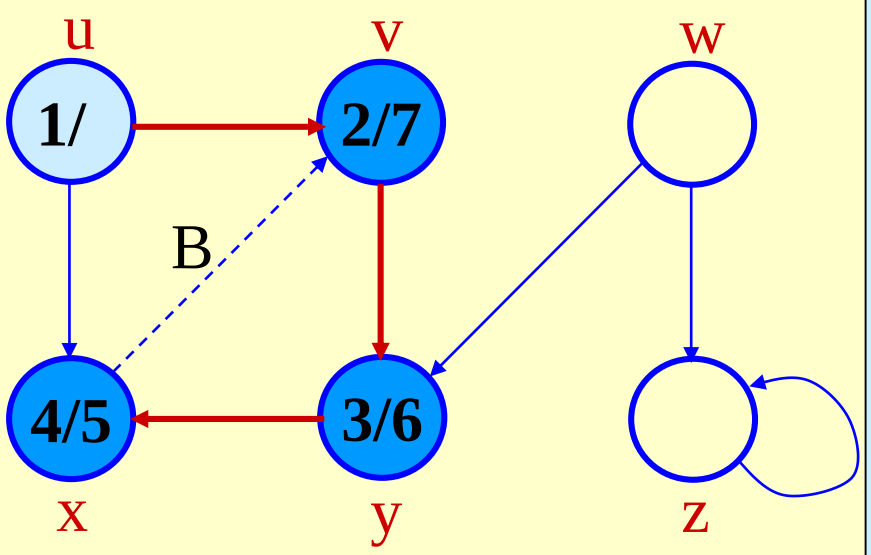
# Example (DFS)




**DFS-Visit(*u*)**

1. *color*[*u*] ← GRAY // White vertex *u* has been discovered
2. *time* ← *time* + 1
3. *d*[*u*] ← *time*
4. **for** each *v* ∈ *Adj*[*u*]
5. **do if** *color*[*v*] = WHITE
6. **then** π[*v*] ← *u*
7. DFS-Visit(*v*)
8. *color*[*u*] ← BLACK // Blacken *u*; it is finished.
9. *f*[*u*] ← *time* ← *time* + 1

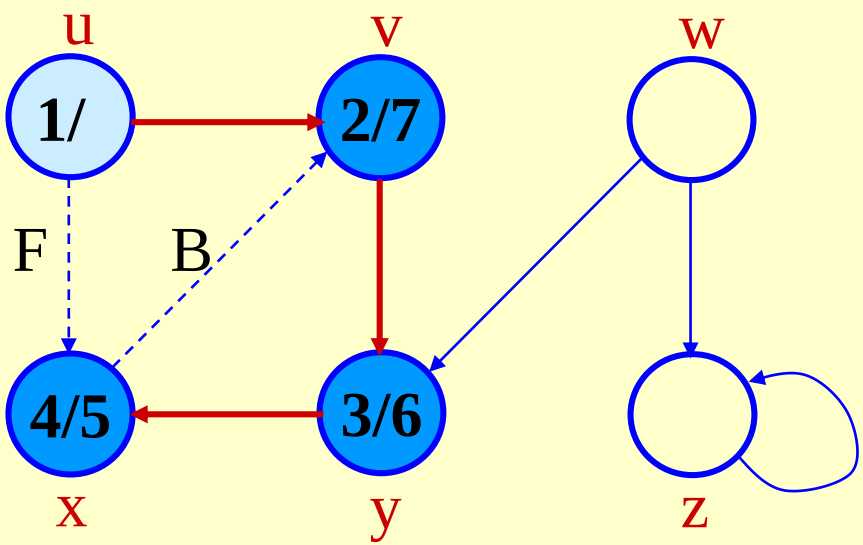
# Example (DFS)




**DFS-Visit(*u*)**

1. *color*[*u*] ← GRAY // White vertex *u* has been discovered
2. *time* ← *time* + 1
3. *d*[*u*] ← *time*
4. **for** each *v* ∈ *Adj*[*u*]
5. **do if** *color*[*v*] = WHITE
6. **then** π[*v*] ← *u*
7. DFS-Visit(*v*)
8. *color*[*u*] ← BLACK // Blacken *u*; it is finished.
9. *f*[*u*] ← *time* ← *time* + 1

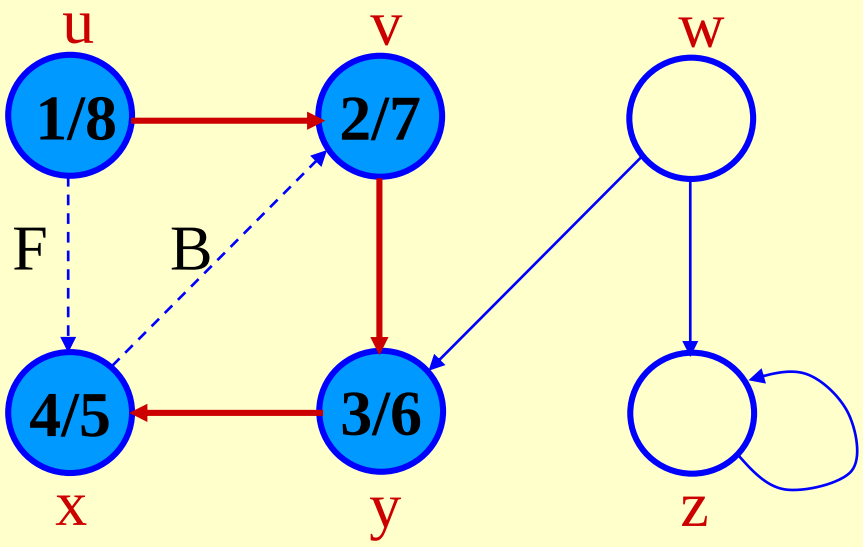
# Example (DFS)




**DFS-Visit(*u*)**

1. *color*[*u*] ← GRAY // White vertex *u* has been discovered
2. *time* ← *time* + 1
3. *d*[*u*] ← *time*
4. **for** each *v* ∈ *Adj*[*u*]
5. **do if** *color*[*v*] = WHITE
6. **then** π[*v*] ← *u*
7. DFS-Visit(*v*)
8. *color*[*u*] ← BLACK // Blacken *u*; it is finished.
9. *f*[*u*] ← *time* ← *time* + 1

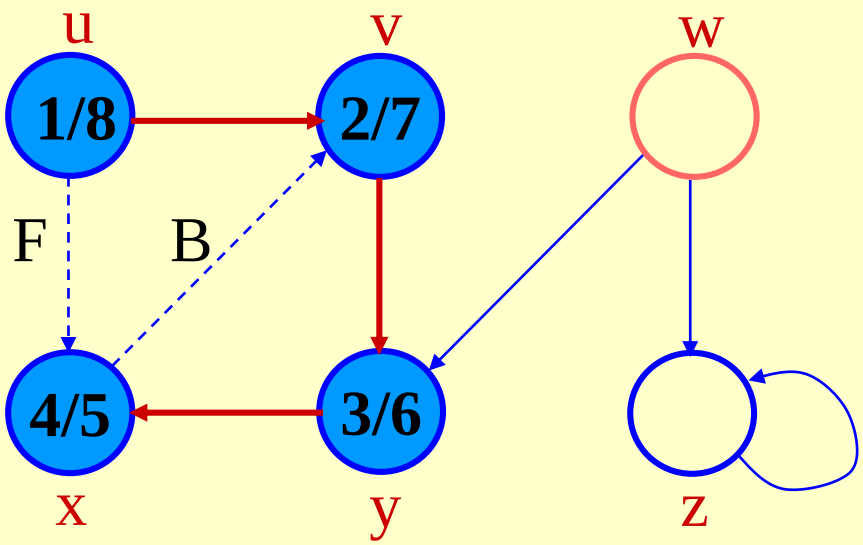
# Example (DFS)




**DFS-Visit(*u*)**

1. *color*[*u*] ← GRAY // White vertex *u* has been discovered
2. *time* ← *time* + 1
3. *d*[*u*] ← *time*
4. **for** each *v* ∈ *Adj*[*u*]
5. **do if** *color*[*v*] = WHITE
6. **then** π[*v*] ← *u*
7. DFS-Visit(*v*)
8. *color*[*u*] ← BLACK // Blacken *u*; it is finished.
9. *f*[*u*] ← *time* ← *time* + 1

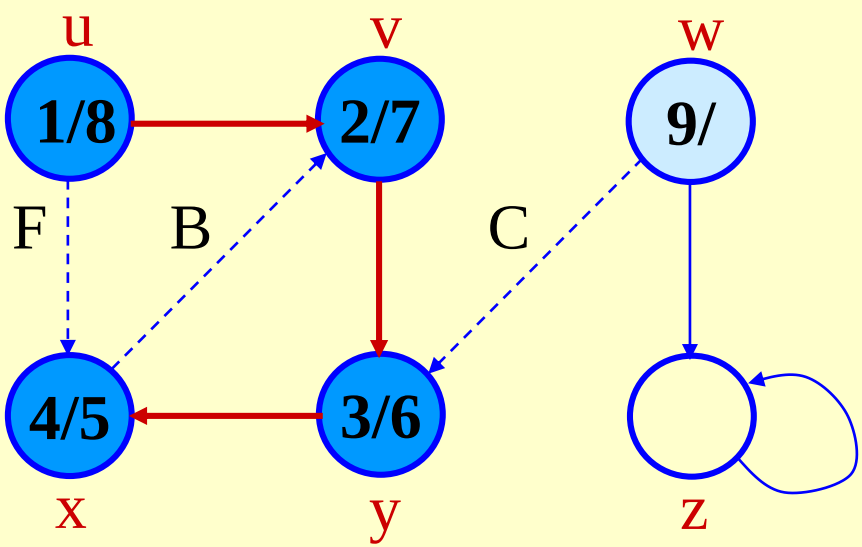
# Example (DFS)




**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($u$)

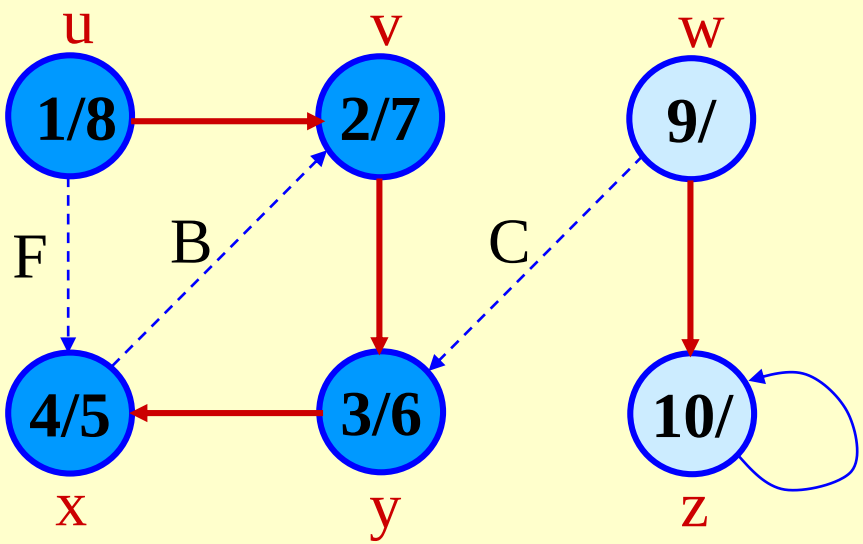
# Example (DFS)




**DFS-Visit(*u*)**

1. *color*[*u*] ← GRAY // White vertex *u* has been discovered
2. *time* ← *time* + 1
3. *d*[*u*] ← *time*
4. **for** each *v* ∈ *Adj*[*u*]
5. **do if** *color*[*v*] = WHITE
6. **then** π[*v*] ← *u*
7. DFS-Visit(*v*)
8. *color*[*u*] ← BLACK // Blacken *u*; it is finished.
9. *f*[*u*] ← *time* ← *time* + 1

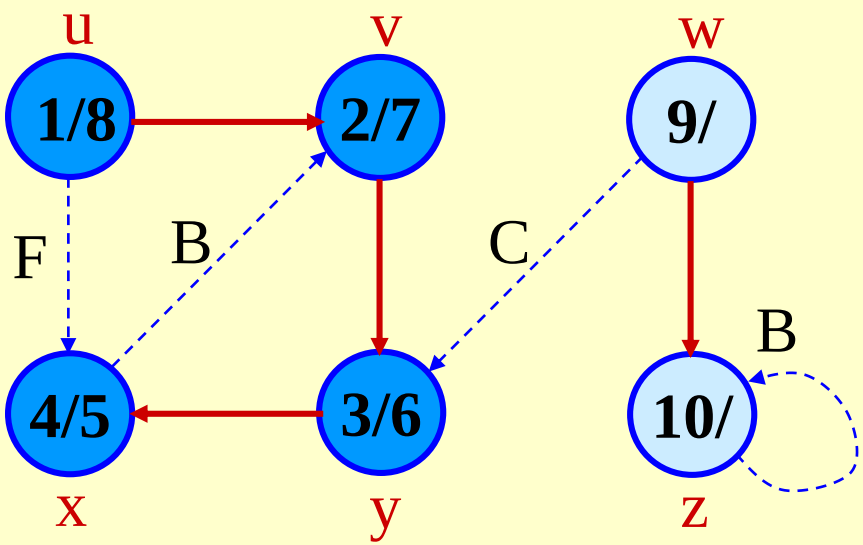
# Example (DFS)




**DFS-Visit(*u*)**

1. *color*[*u*] ← GRAY // White vertex *u* has been discovered
2. *time* ← *time* + 1
3. *d*[*u*] ← *time*
4. **for** each *v* ∈ *Adj*[*u*]
5. **do if** *color*[*v*] = WHITE
6. **then** π[*v*] ← *u*
7. DFS-Visit(*v*)
8. *color*[*u*] ← BLACK // Blacken *u*; it is finished.
9. *f*[*u*] ← *time* ← *time* + 1

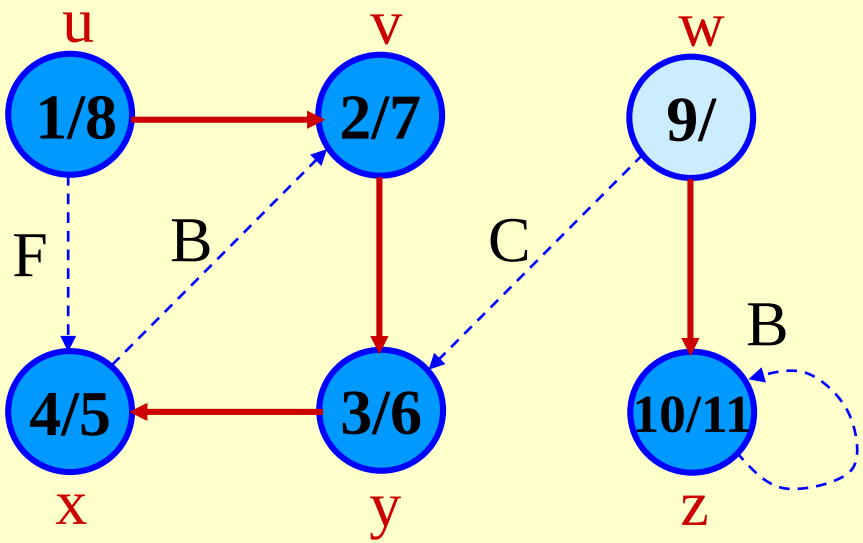
# Example (DFS)




**DFS-Visit(*u*)**

1. *color*[*u*] ← GRAY // White vertex *u* has been discovered
2. *time* ← *time* + 1
3. *d*[*u*] ← *time*
4. **for** each *v* ∈ *Adj*[*u*]
5. **do if** *color*[*v*] = WHITE
6. **then** π[*v*] ← *u*
7. DFS-Visit(*v*)
8. *color*[*u*] ← BLACK // Blacken *u*; it is finished.
9. *f*[*u*] ← *time* ← *time* + 1

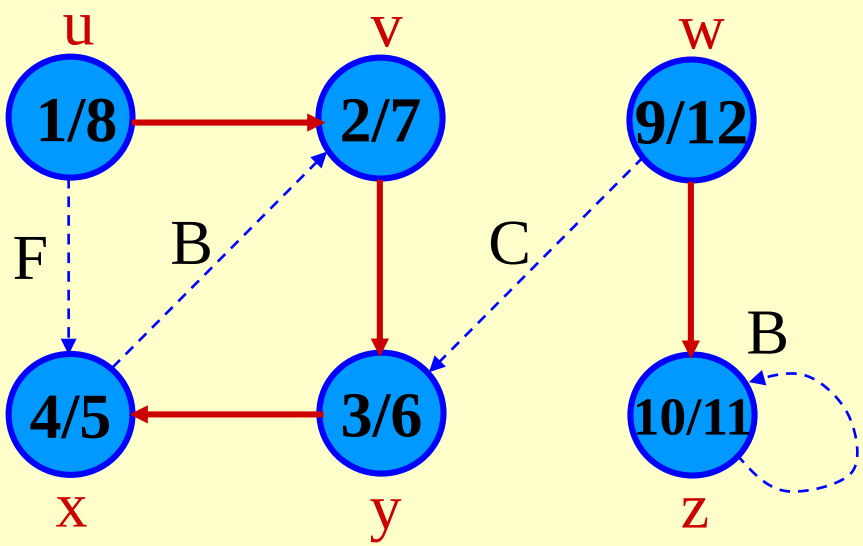
# Example (DFS)




**DFS-Visit(*u*)**

1. *color*[*u*] ← GRAY // White vertex *u* has been discovered
2. *time* ← *time* + 1
3. *d*[*u*] ← *time*
4. **for** each *v* ∈ *Adj*[*u*]
5. **do if** *color*[*v*] = WHITE
6. **then** π[*v*] ← *u*
7. DFS-Visit(*v*)
8. *color*[*u*] ← BLACK // Blacken *u*; it is finished.
9. *f*[*u*] ← *time* ← *time* + 1

# Example (DFS)




**DFS-Visit(*u*)**

1. *color*[*u*] ← GRAY // White vertex *u* has been discovered
2. *time* ← *time* + 1
3. *d*[*u*] ← *time*
4. **for** each *v* ∈ *Adj*[*u*]
5. **do if** *color*[*v*] = WHITE
6. **then** π[*v*] ← *u*
7. DFS-Visit(*v*)
8. *color*[*u*] ← BLACK // Blacken *u*; it is finished.
9. *f*[*u*] ← *time* ← *time* + 1

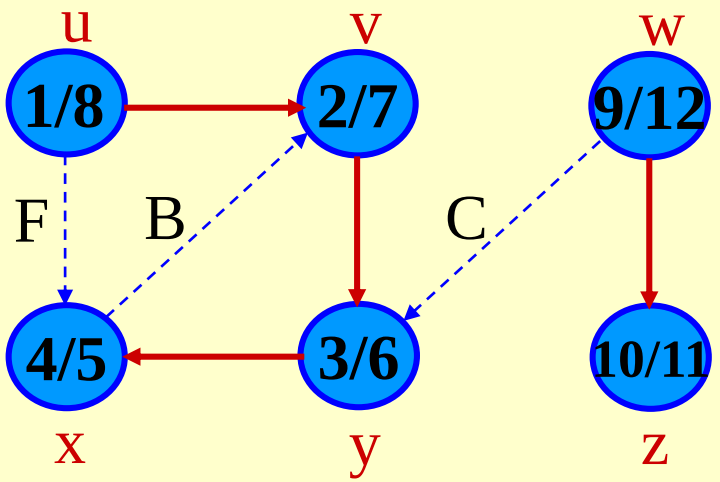
# Analysis of DFS

- 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 $|\text{Adj}[v]|$ times. The total cost of executing DFS-Visit is $\sum_{v \in V} |\text{Adj}[v]| = \Theta(|E|)$
- Total running time of DFS is $\Theta(|V| + |E|)$.

# Depth-First Trees

- Predecessor subgraph defined slightly different from that of BFS.
- The predecessor subgraph of DFS is $G_\pi = (V, E_\pi)$ where $E_\pi = \{(\pi[v], v) : v \in V \text{ and } \pi[v] \neq nil\}$.
  - » How does it differ from that of BFS?
  - » 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*.

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



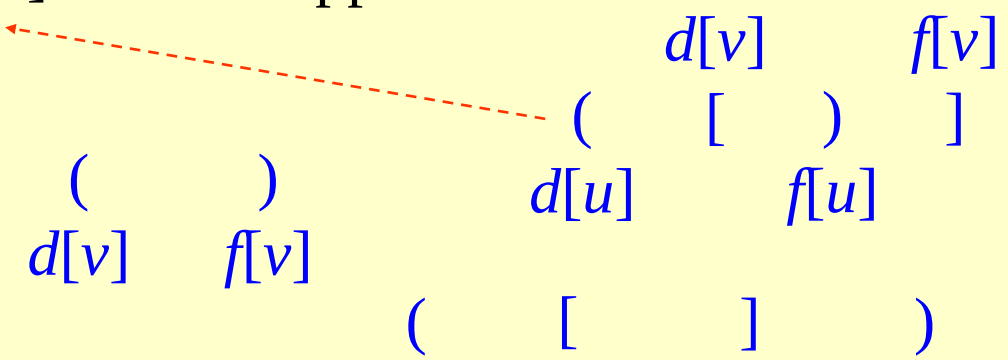
# Parenthesis Theorem

**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 in the *DF-tree*.
2. $d[u] < d[v] < f[v] < f[u]$ and $v$ is a descendant of $u$ in *DF-tree*.
3. $d[v] < d[u] < f[u] < f[v]$ and $u$ is a descendant of $v$ in *DF-tree*.

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

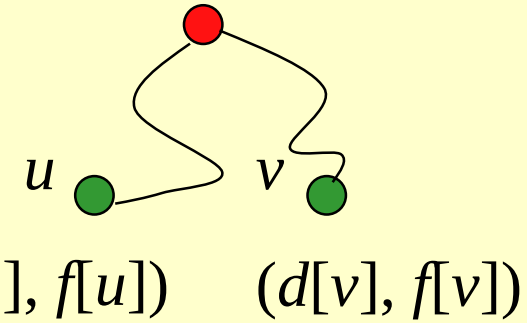


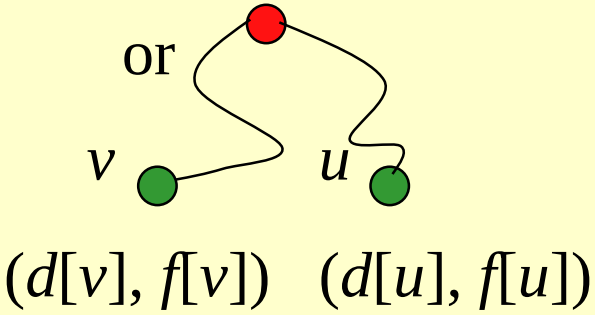

***Corollary***

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

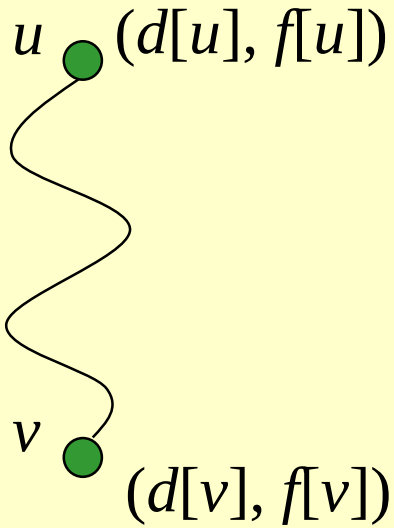
# Parenthesis Theorem

Case 1:

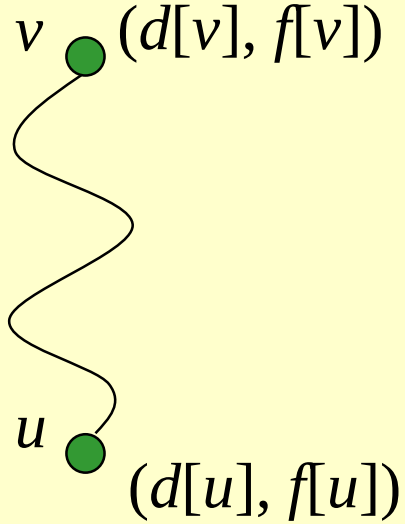


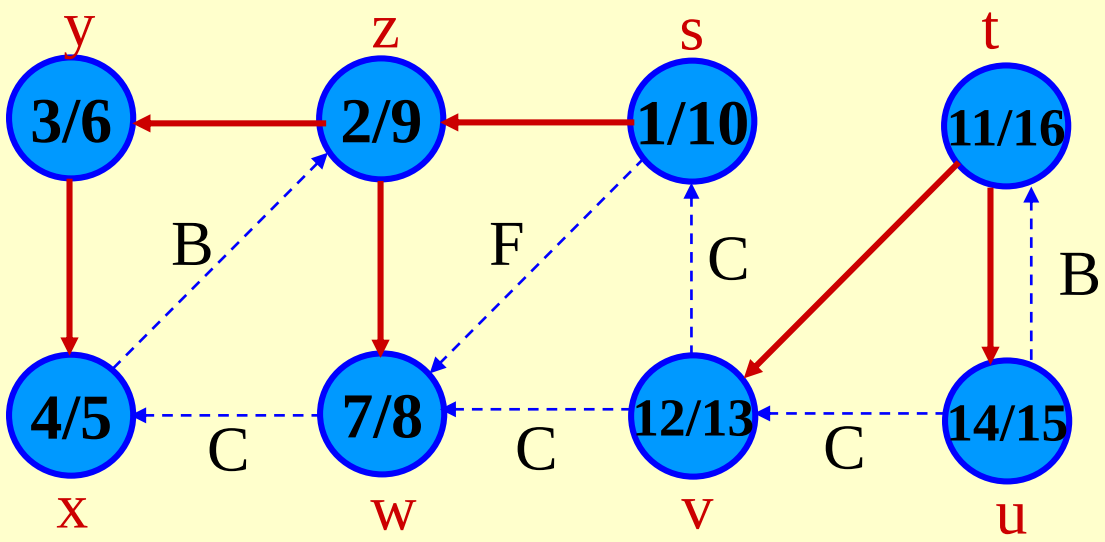

Case 2:




Case 3:



# Example (Parenthesis Theorem)




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

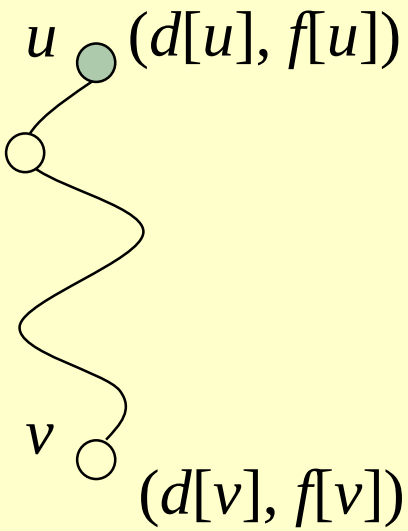
1<2<3<4<5<6<7<8<9<10 11<12<13<14<15<16

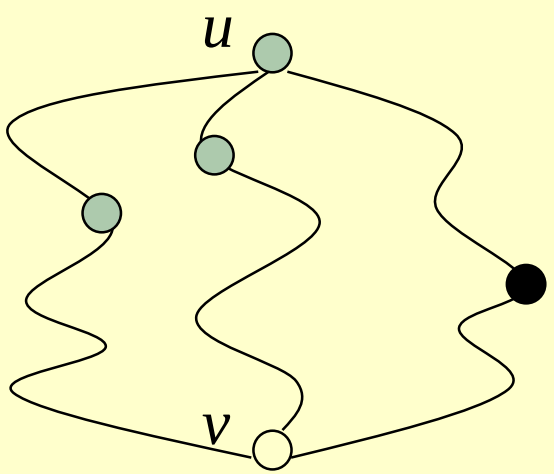
In general, if we use ‘(*v*’ to represent *d*[*v*], and ‘*v*)’ to represent *f*[*v*], the inequalities in the Parenthesis Theorem are just like parentheses in an arithmetical expression.

# White-path Theorem

**Theorem 22.9**

*v* is a descendant of *u* in *DF-tree* if and only if at time *d*[*u*], there is a path $u \rightsquigarrow v$ consisting of only white vertices. (Except for *u*, which was *just* colored gray.)



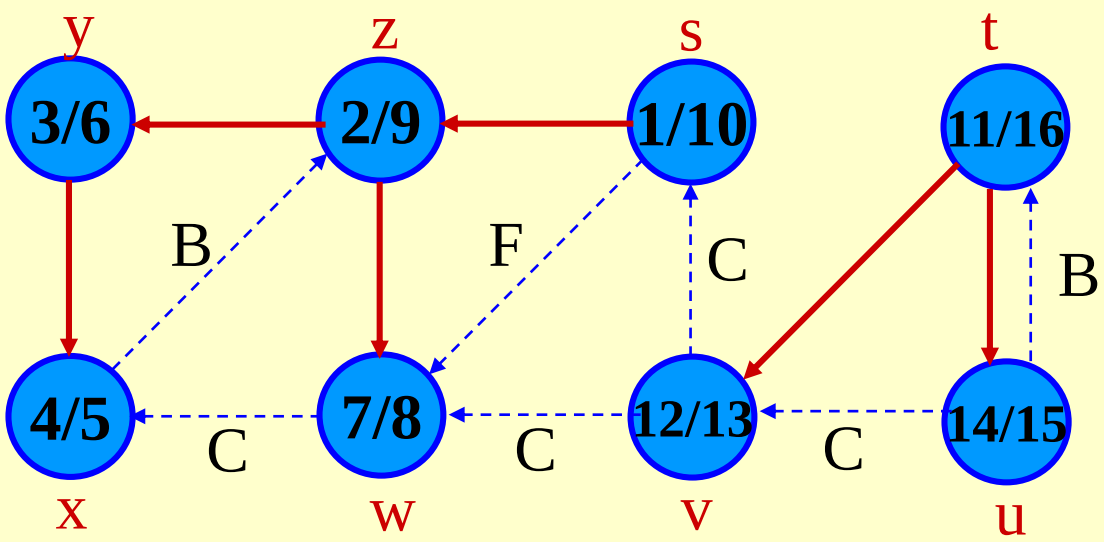



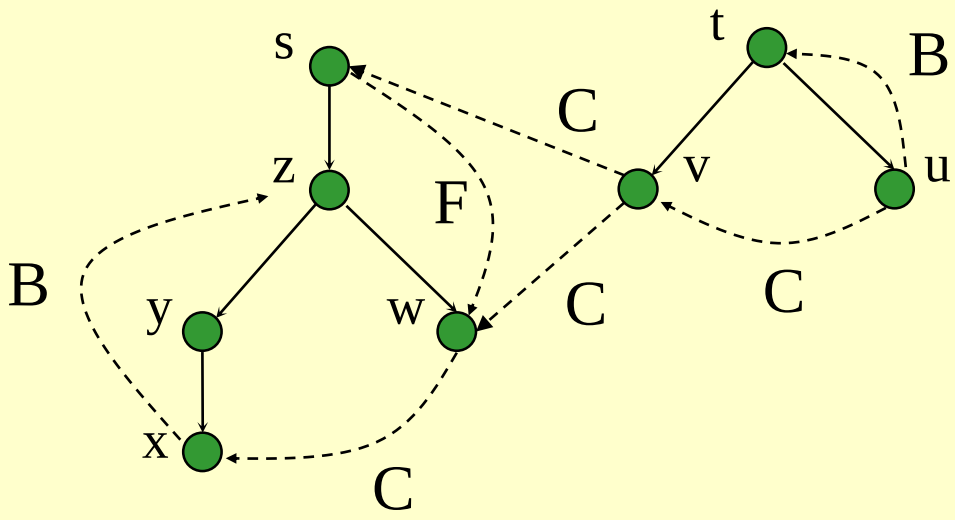
# Classification of Edges

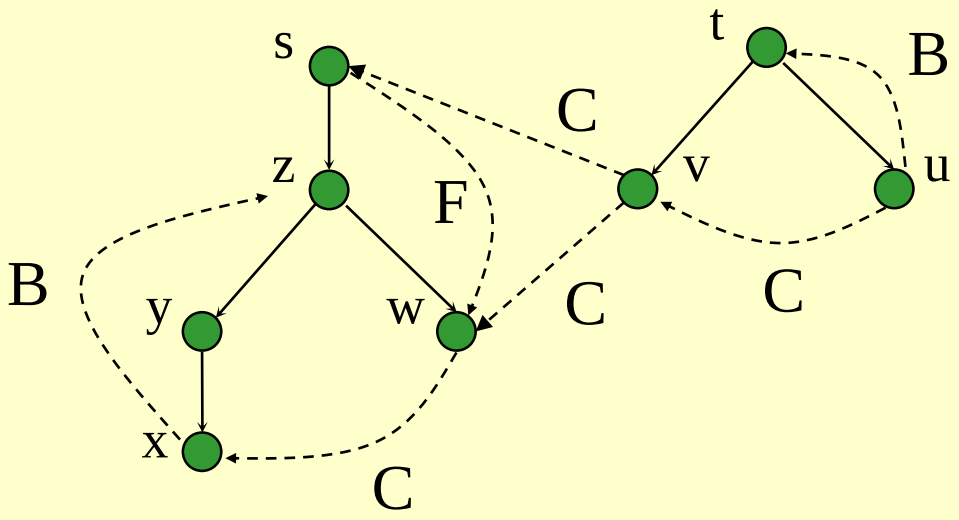
- **Tree edge:** in the depth-first forest. Found by exploring $(u, v)$.
- **Back edge:** $(u, v)$, where $u$ is a descendant of $v$ (in the depth-first tree).
- **Forward edge:** $(u, v)$, where $v$ is a descendant of $u$, but not a tree edge.
- **Cross edge:** any other edge $(u, v)$ such that $u$ is not a descendant of $v$ (in the depth-first tree) and *vice versa*.

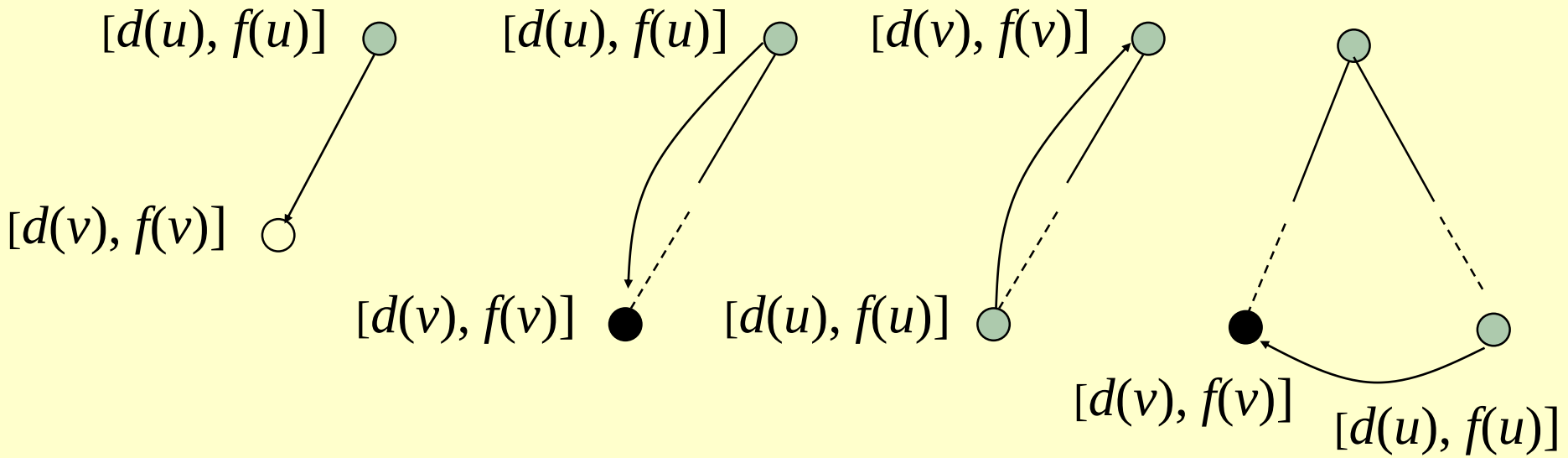
**Theorem:**
In DFS of an undirected graph, we get only tree and back edges. No forward or cross edges.

# Classification of Edges




s
t
B
C
z
v
u
F
B
y
w
C
C
x
C

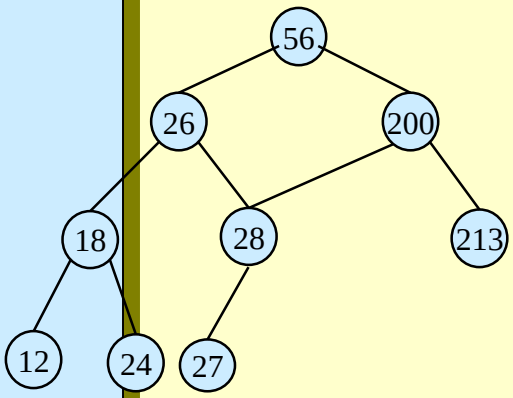

[d(u), f(u)]
[d(v), f(v)]
[d(u), f(u)]
[d(v), f(v)]
[d(v), f(v)]
[d(u), f(u)]
[d(v), f(v)]
[d(u), f(u)]

# DFS graph search using stack

Depth-first($x$)

1. **push**($S$, $x$)
2. **while** $S \neq$ empty **do**
3. $v$ := **pop**($S$)
4. print $key[x]$
5. let $v_1, \ldots, v_k$ be the children of $x$
6. **for** ($i = k$ to 1) **do**
7. **if** $v_i$ has not yet been accessed **then**
8. **push**($S$, $v_i$)

$S$ is a stack.




It is also called the preoreder search and top-down search.

# Bottom-up search of a directed graph

Bottom-up($x$)

**Bottom-up($x$)**

1.let $v_1, \ldots, v_k$ be the children of $x$

**2.for** ($i = k$ to 1) **do**

**3.** **if** $v_i$ has not yet accessed **then**

**4.** **Bottom-up**($v_i$)

**5.** Print($x$)