CS583 Lecture 08

Jana Kosecka

Red-Black Trees
Graph Algorithms

Many slides here are based on E. Demaine , D. Luebke slides

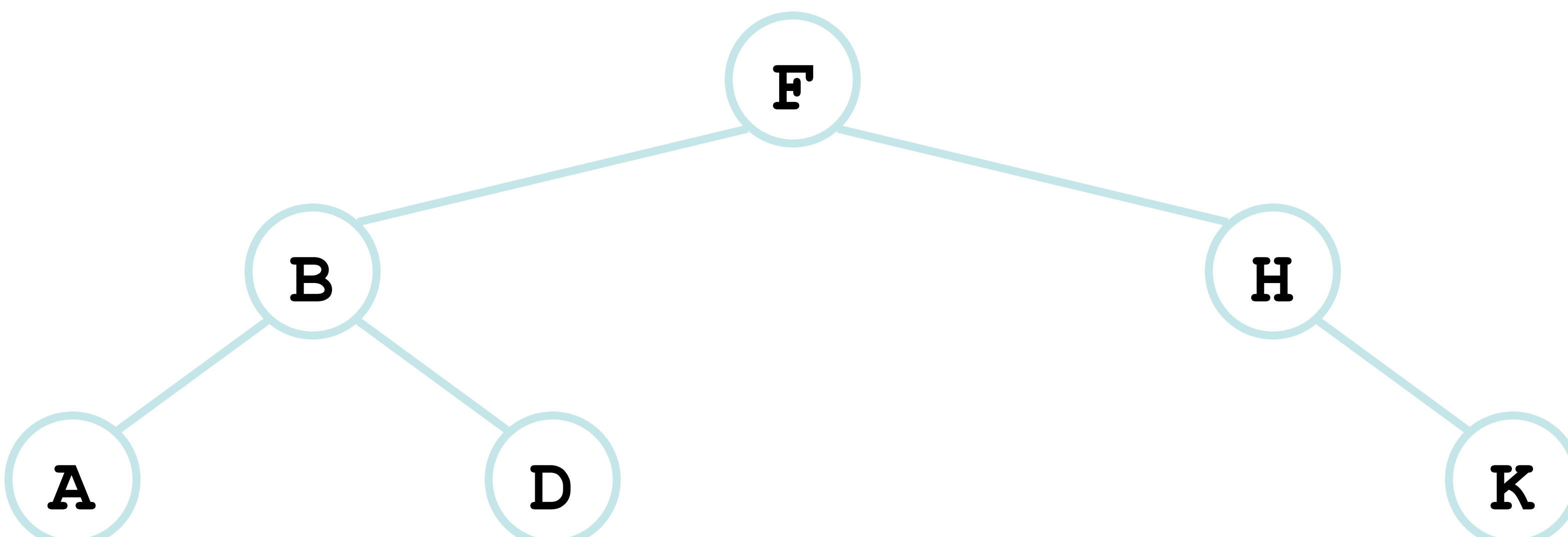
Review: Binary Search Trees

- *Binary Search Trees* (BSTs) are an important data structure for dynamic sets
- In addition to satellite data, eleements have:
  key: an identifying field inducing a total ordering
  left: pointer to a left child (may be NULL)
  right: pointer to a right child (may be NULL)
  p: pointer to a parent node (NULL for root)

Review: Binary Search Trees

- BST property:

  key[left(x)] ≤ key[x] ≤ key[right(x)]
- Example:



Review: Inorder Tree Walk

- An *inorder walk* prints the set in sorted order:

```
TreeWalk(x)
    TreeWalk(left[x]);
    print(x);
    TreeWalk(right[x]);
```

  Easy to show by induction on the BST property
  Preorder tree walk: print root, then left, then right
  Postorder tree walk: print left, then right, then root

Review: BST Search

```
TreeSearch(x, k)
    if (x = NULL  or  k = key[x])
        return x;
    if (k < key[x])
        return TreeSearch(left[x], k);
    else
        return TreeSearch(right[x], k);
```

Review: BST Search (Iterative)

```
IterativeTreeSearch(x, k)
    while (x != NULL  and  k != key[x])
        if (k < key[x])
            x = left[x];
        else
            x = right[x];
    return x;
```

Review: BST Insert

- Adds an element x to the tree so that the binary search tree property continues to hold
- The basic algorithm
  - Like the search procedure above
  - Insert x in place of NULL
  - Use a "trailing pointer" to keep track of where you came from (like inserting into singly linked list)
- Like search, takes time O(*h*), *h* = tree height

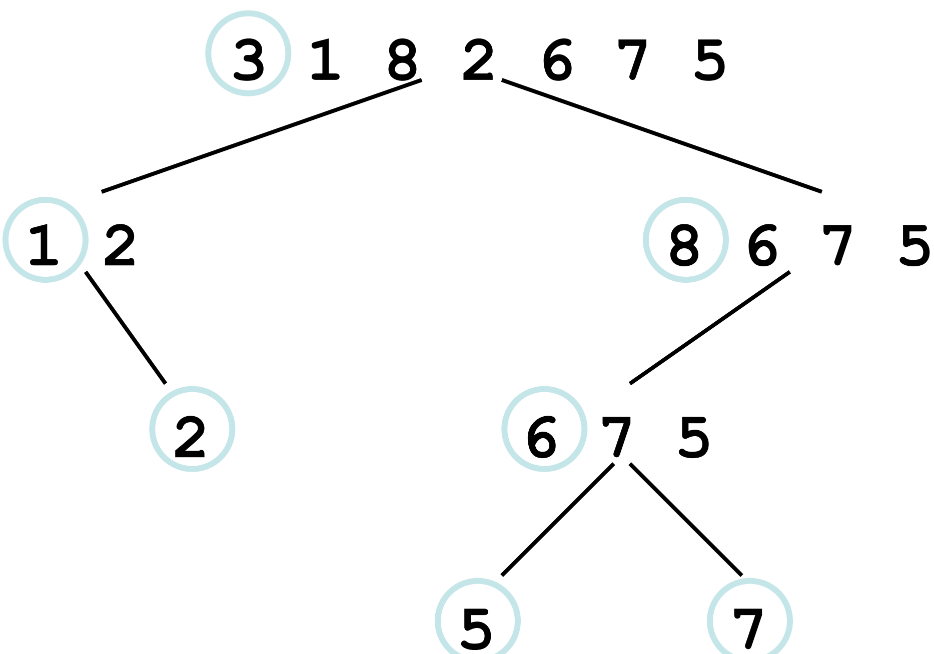
Review: Sorting With BSTs

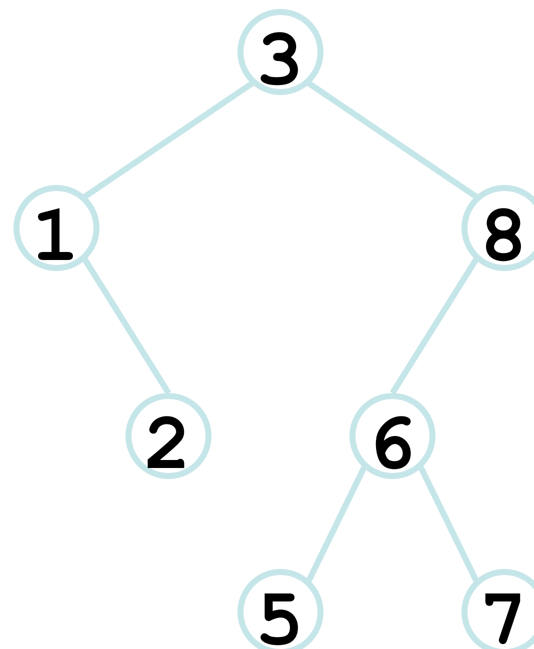
- Basic algorithm:
  Insert elements of unsorted array from 1..n
  Do an inorder tree walk to print in sorted order
- Running time:
  Best case: $\Omega(n \lg n)$ (it's a comparison sort)
  Worst case: $O(n^2)$
  Average case: $O(n \lg n)$ (it's a quicksort!)

Review: Sorting With BSTs

- Average case analysis
  It's a form of quicksort!

```
for i=1 to n
    TreeInsert(A[i]);
InorderTreeWalk(root);
```



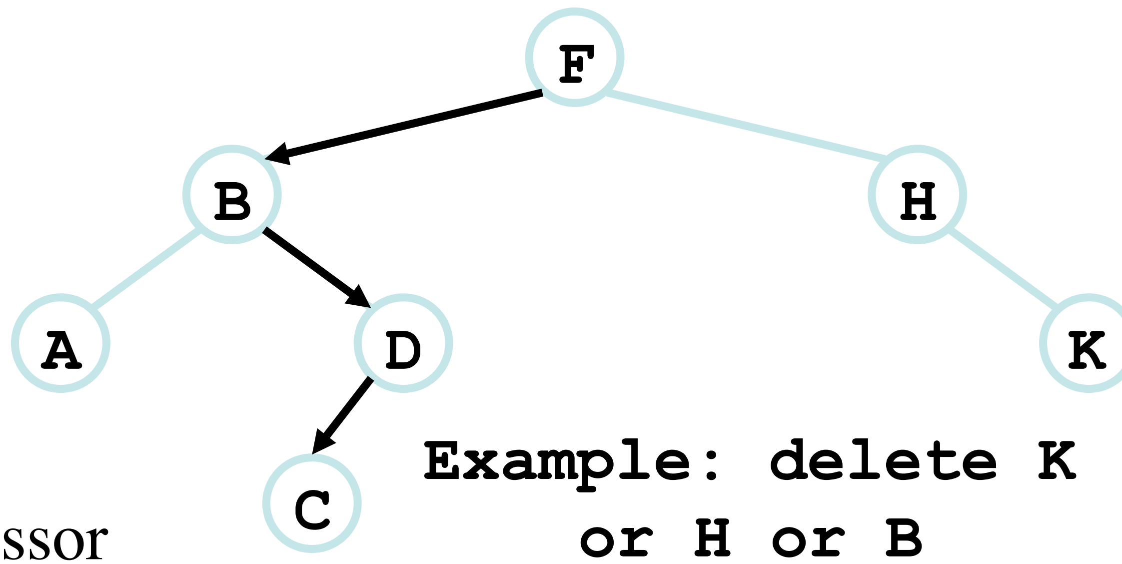


Review: More BST Operations

- Minimum:
  Find leftmost node in tree
- Successor:
  x has a right subtree: successor is minimum node in right subtree
  x has no right subtree: successor is first ancestor of x whose left child is also ancestor of x
    Intuition: As long as you move to the left up the tree, you're visiting smaller nodes.
- Predecessor: similar to successor

Review: More BST Operations

- Delete:
  - x has no children:
    - Remove x
  - x has one child:
    - Splice out x
  - x has two children:
    - Swap x with successor
    - Perform case 1 or 2 to delete it



Example: delete K or H or B

Red-Black Trees

- *Red-black trees*:
  Binary search trees augmented with node color
  Operations designed to guarantee that the height
  $h = O(\lg n)$
- First: describe the properties of red-black trees
- Then: prove that these guarantee $h = O(\lg n)$
- Finally: describe operations on red-black trees

Red-Black Properties

- The *red-black properties*:
  1. Every node is either red or black
  2. Every leaf (NULL pointer) is black
     Note: this means every "real" node has children
  3. If a node is red, both children are black
     Note: can't have 2 consecutive reds on a path
  4. Every path from node to descendent leaf contains the same number of black nodes
  5. The root is always black

Review: Red-Black Trees

- Put example on board and verify properties:
  1. Every node is either red or black
  2. Every leaf (NULL pointer) is black
  3. If a node is red, both children are black
  4. Every path from node to descendent leaf contains the same number of black nodes
  5. The root is always black
- *black-height:* # black nodes on path to leaf
  Label example with *h* and bh values

Review: Height of Red-Black Trees

- *What is the minimum black-height of a node with height h?*
- A: a height-h node has black-height $\geq h/2$
- Theorem: A red-black tree with n internal nodes has height $h \leq 2 \lg(n + 1)$

RB Trees: Proving Height Bound

- Thus at the root of the red-black tree:

$n \geq 2^{\mathrm{bh}(root)} - 1$ *(Why?)*

$n \geq 2^{h/2} - 1$ *(Why?)*

$\lg(n+1) \geq h/2$ *(Why?)*

$h \leq 2 \lg(n + 1)$ *(Why?)*

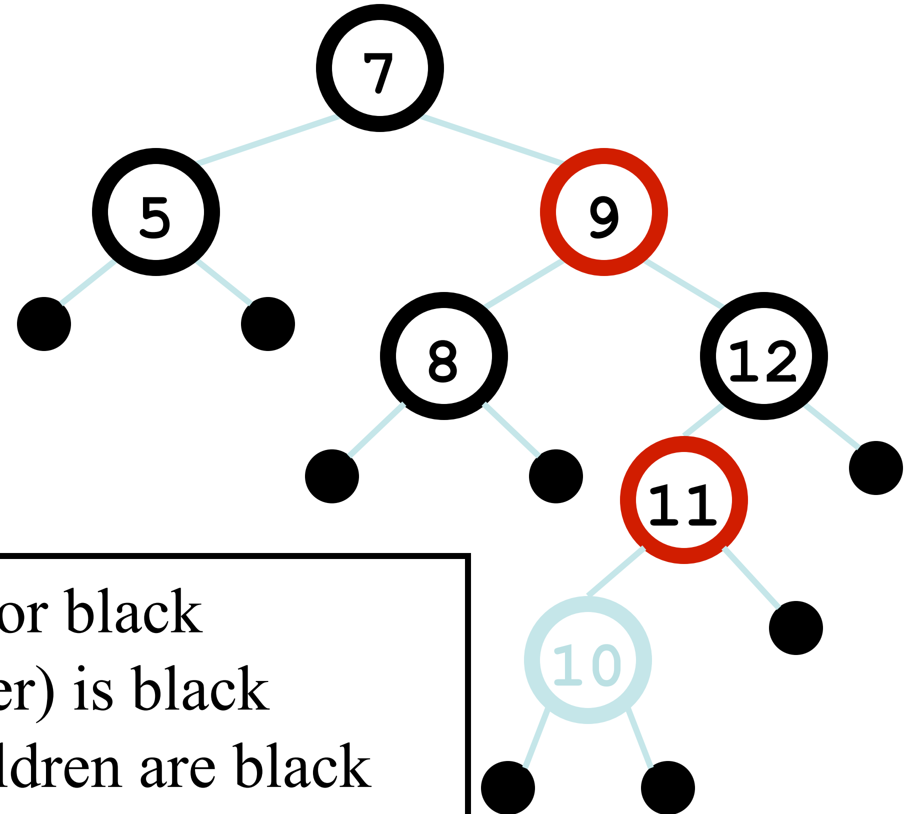
Thus $h = \mathrm{O}(\lg n)$

Red-Black Trees: The Problem With Insertion

- Insert 10

  Where does it go?

  What color?



1.	Every node is either red or black
2.	Every leaf (NULL pointer) is black
3.	If a node is red, both children are black
4.	Every path from node to descendent leaf contains the same number of black nodes
5.	The root is always black

Red-Black Trees: The Problem With Insertion

- Insert 10

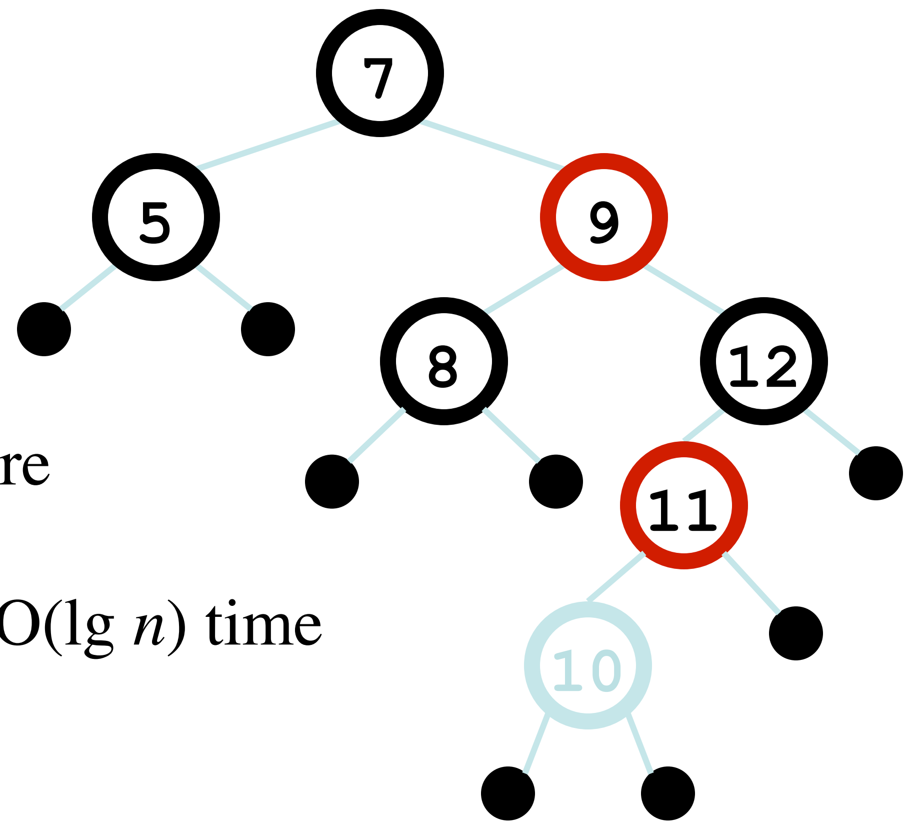
  Where does it go?

  What color?

  A: no color! Tree is too imbalanced

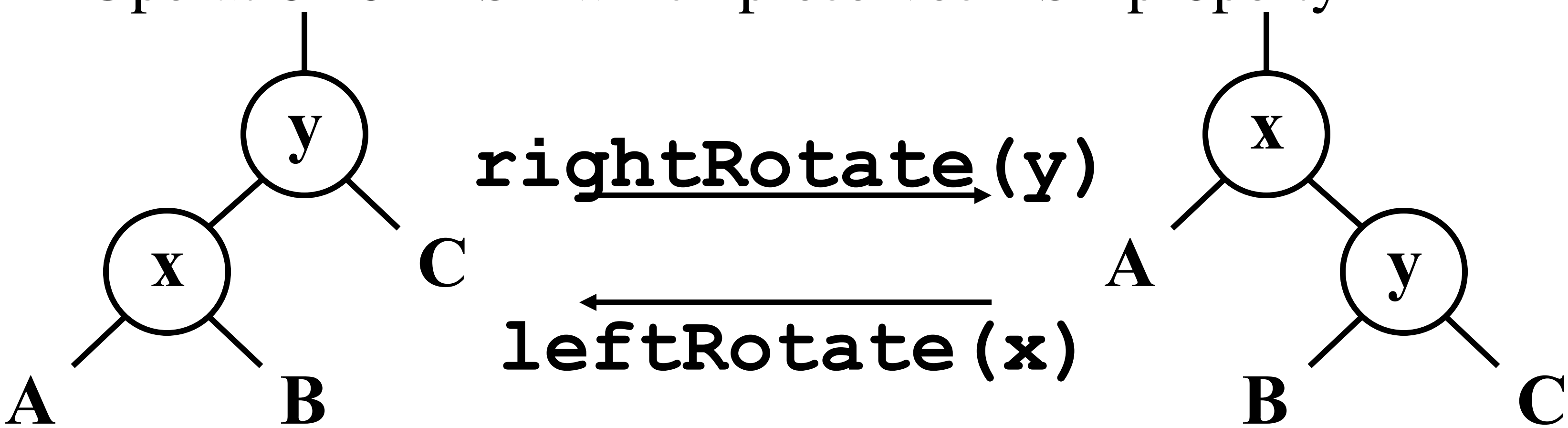
  Must change tree structure to allow recoloring

  Goal: restructure tree in O(lg *n*) time



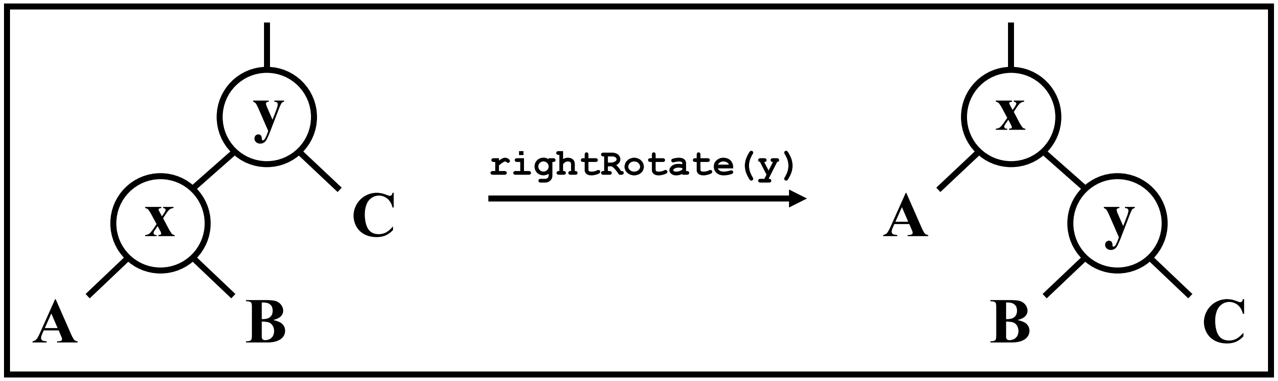
Review: RB Trees: Rotation

- Our basic operation for changing tree structure is called *rotation*:
- Operation on BST which preserves BST property



- *Does rotation preserve inorder key ordering?*
- *What would the code for* ***`rightRotate()`*** *actually do?*

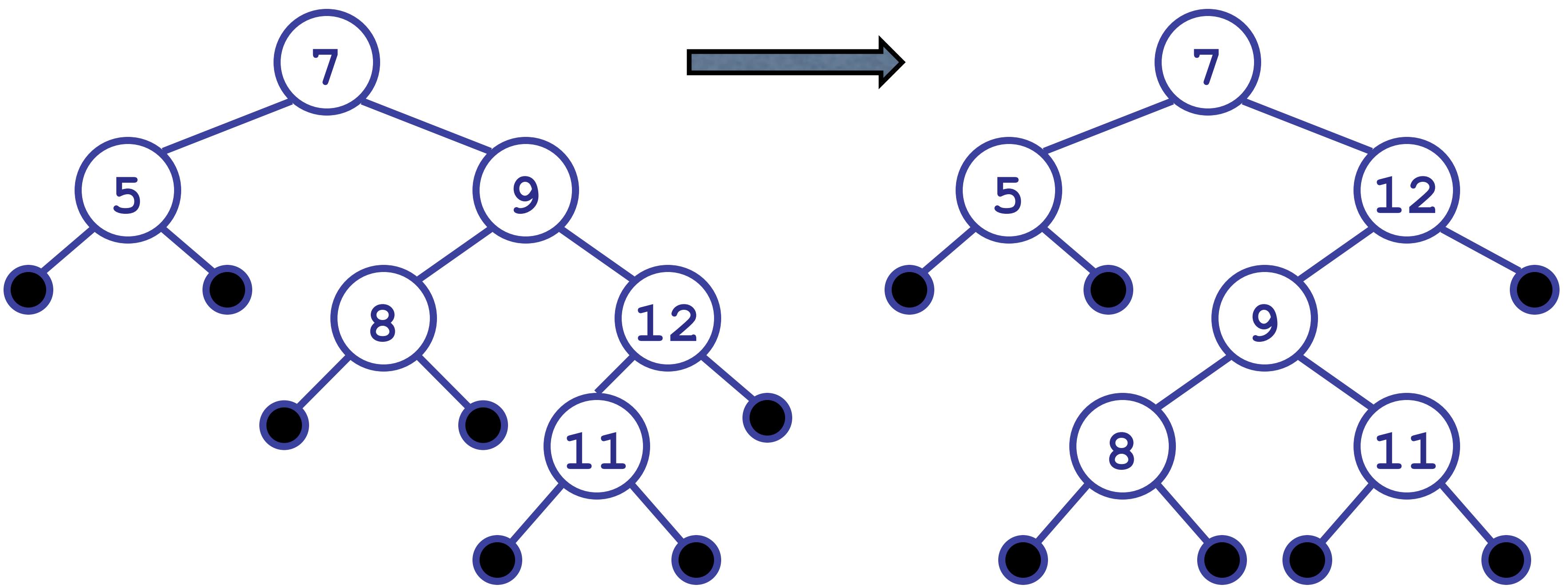
RB Trees: Rotation



- Answer: A lot of pointer manipulation
  - *x* keeps its left child
  - *y* keeps its right child
  - *x*'s right child becomes *y*'s left child
  - *x*'s and *y*'s parents change
- *What is the running time?*

Rotation Example

- Rotate left about 9:



Red-Black Trees: Insertion

- Insertion: the basic idea
- Insert x into tree, color x red
- Only r-b property 3 might be violated (if p[x] red)
- If so, move violation up tree until a place is found where it can be fixed
- Total time will be O(lg n)

```
rbInsert(x)
  treeInsert(x);
  x->color = RED;
  // Move violation of #3 up tree, maintaining #4 as invariant:
  while (x!=root && x->p->color == RED)
  if (x->p == x->p->p->left)
      y = x->p->p->right;
      if (y->color == RED)
          x->p->color = BLACK;
          y->color = BLACK;
          x->p->p->color = RED;
          x = x->p->p;                              Case 1
      else   // y->color == BLACK
          if (x == x->p->right)
              x = x->p;
              leftRotate(x);
          x->p->color = BLACK;
          x->p->p->color = RED;                     Case 2
          rightRotate(x->p->p);
  else    // x->p == x->p->p->right
      (same as above, but with                      Case 3
       "right" & "left" exchanged)
```

```
rbInsert(x)
  treeInsert(x);
  x->color = RED;
  // Move violation of #3 up tree, maintaining #4 as invariant:
  while (x!=root && x->p->color == RED)
  if (x->p == x->p->p->left)
      y = x->p->p->right;
      if (y->color == RED)
          x->p->color = BLACK;
          y->color = BLACK;
          x->p->p->color = RED;
          x = x->p->p;
      else   // y->color == BLACK
          if (x == x->p->right)
              x = x->p;
              leftRotate(x);
          x->p->color = BLACK;
          x->p->p->color = RED;
          rightRotate(x->p->p);
  else    // x->p == x->p->p->right
      (same as above, but with
       "right" & "left" exchanged)
```

Case 1:uncle is RED

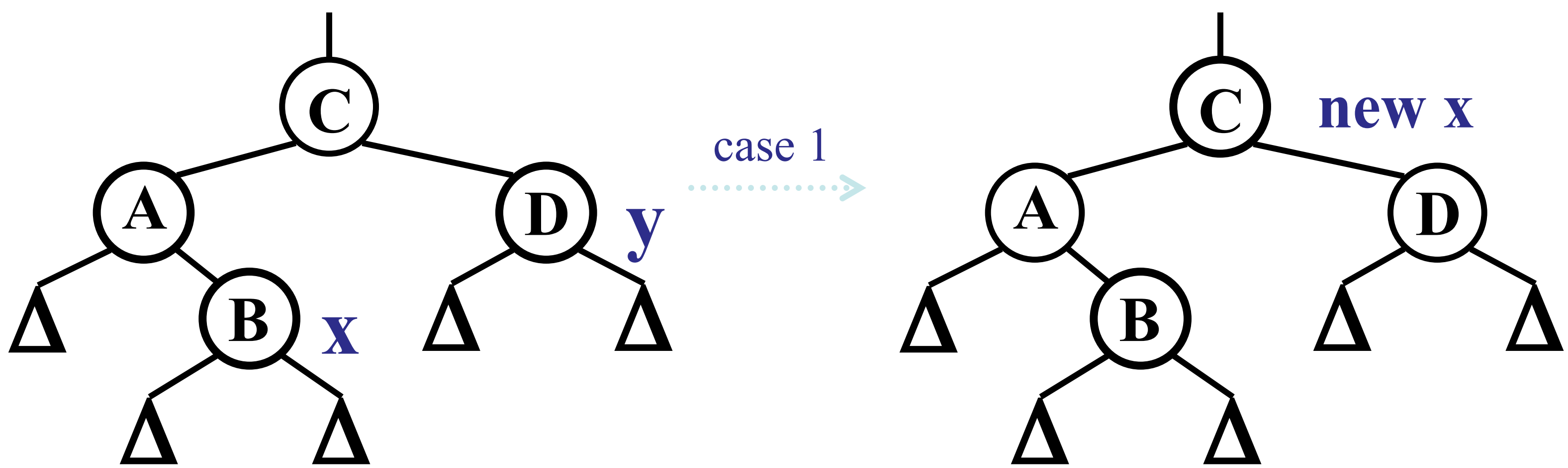
Case 2

Case 3

RB Insert: Case 1

```
if (y->color == RED)
    x->p->color = BLACK;
    y->color = BLACK;
    x->p->p->color = RED;
    x = x->p->p;
```

- Case 1: "uncle" is red
- In figures below, all Δ's are equal-black-height subtrees

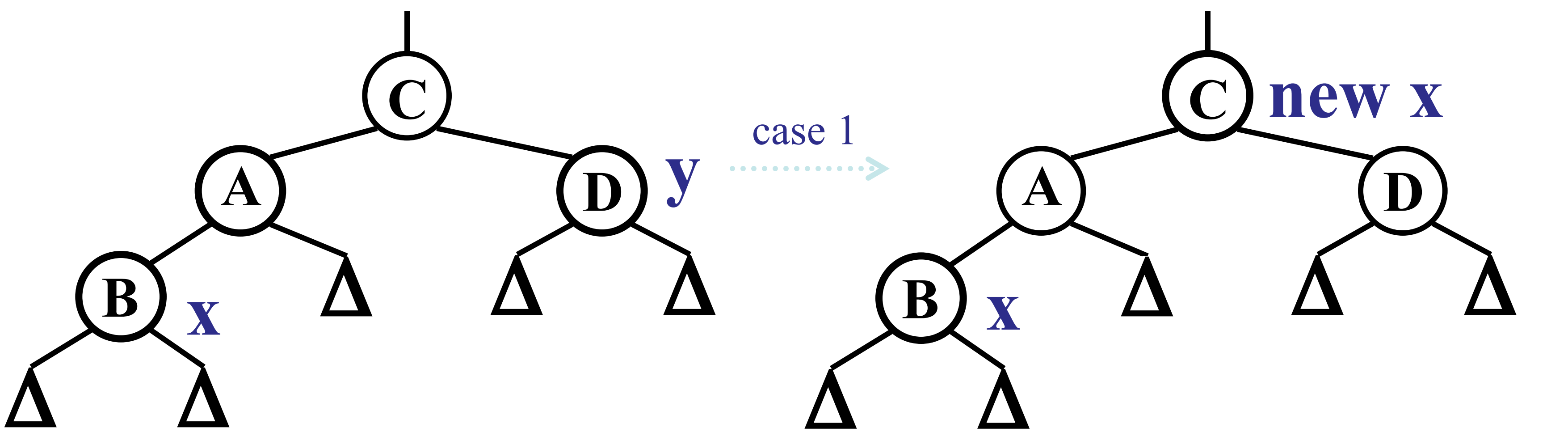


Change colors of some nodes, preserving #4:
all downward paths have equal b.h.
The while loop now continues with x's grandparent as the new x

RB Insert: Case 1

```
if (y->color == RED)
    x->p->color = BLACK;
    y->color = BLACK;
    x->p->p->color = RED;
    x = x->p->p;
```

- Case 1: "uncle" is red
- In figures below, all Δ's are equal-black-height subtrees

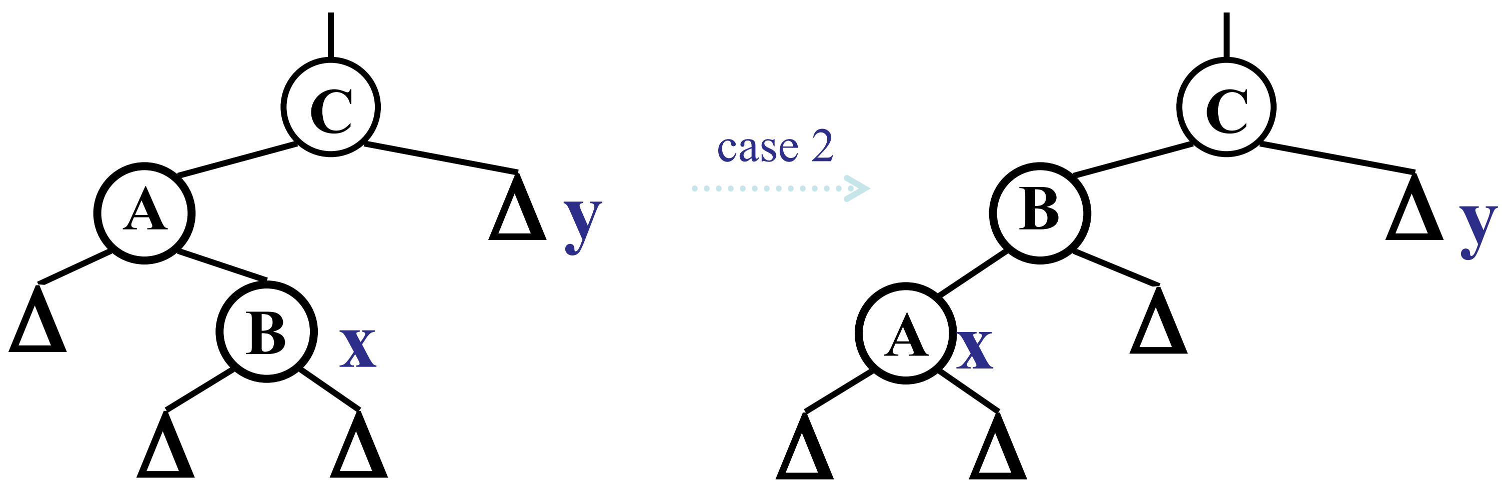


Same action whether x is a left or a right child

RB Insert: Case 2

```
if (x == x->p->right)
    x = x->p;
    leftRotate(x);
// continue with case 3 code
```

- Case 2:
  "Uncle" is black
  Node *x* is a right child
- Transform to case 3 via a left-rotation

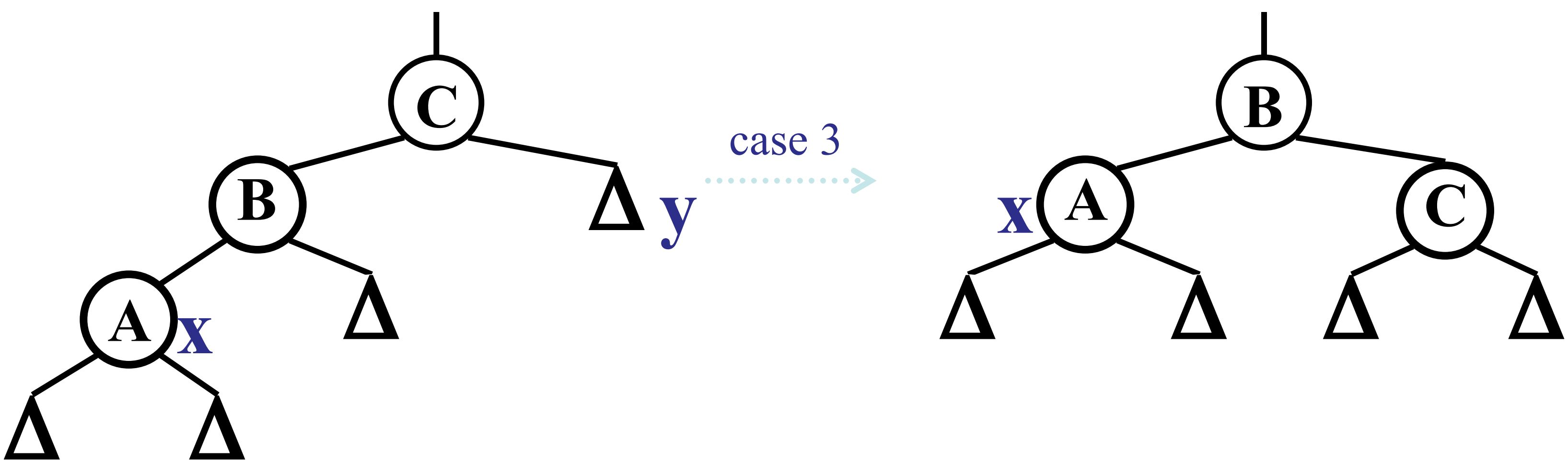


Transform case 2 into case 3 (x is left child) with a left rotation
This preserves property 4: all downward paths contain same number of black nodes

RB Insert: Case 3

```
x->p->color = BLACK;
x->p->p->color = RED;
rightRotate(x->p->p);
```

- Case 3:
  "Uncle" is black
  Node *x* is a left child
- Change colors; rotate right



Perform some color changes and do a right rotation
Again, preserves property 4: all downward paths contain same number of black nodes

RB Insert: Cases 4-6

- Cases 1-3 hold if x's parent is a left child
- If x's parent is a right child, cases 4-6 are symmetric (swap left for right)

Red-Black Trees: Deletion

- And you thought insertion was tricky…
- We will not cover RB delete in class

  You should read section 14.4 on your own

  Read for the overall picture, not the details

The End

- Coming up:
  Graph Algorithms

CS 583: Lecture 08

Jana Kosecka

Graph Algorithms

Graphs

- A graph $G = (V, E)$
  V = set of vertices
  E = set of edges = subset of $V \times V$
  Thus $|E| = O(|V|^2)$

Graph Variations

- 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 $u \rightarrow v$

Graph Variations

- More variations:

  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)

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

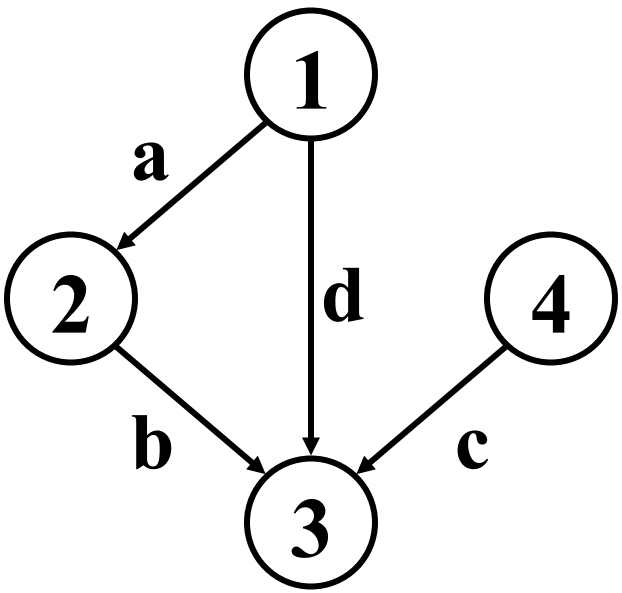
Representing Graphs

- Assume V = $\{1, 2, \ldots, n\}$
- An *adjacency matrix* represents the graph as a n x n matrix A:

$$
\begin{aligned}
A[i, j] \quad &= 1 \text{ if edge } (i, j) \in E \quad \text{(or weight of edge)} \\
&= 0 \text{ if edge } (i, j) \notin E
\end{aligned}
$$

Graphs: Adjacency Matrix

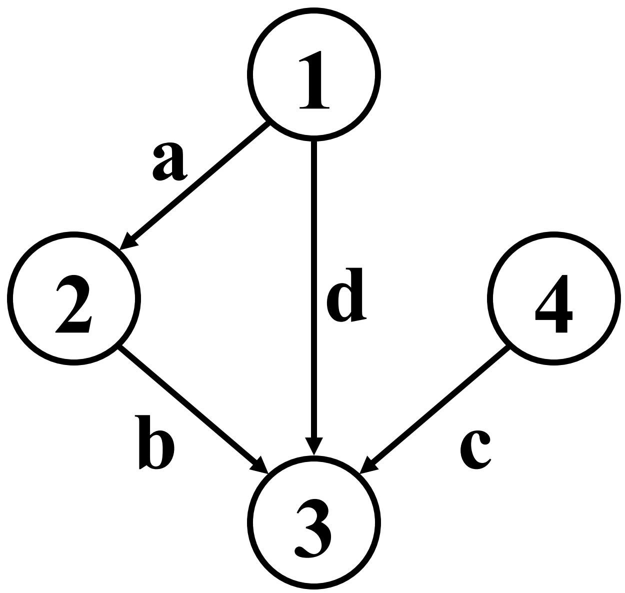
- Example:



A	1	2	3	4
1				
2				
3			??	
4				

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

Graphs: Adjacency Matrix

- *How much storage does the adjacency matrix require?*
- A: $O(V^2)$
- *What is the minimum amount of storage needed by an adjacency matrix representation of an undirected graph with 4 vertices?*
- A: 6 bits

  Undirected graph → matrix is symmetric

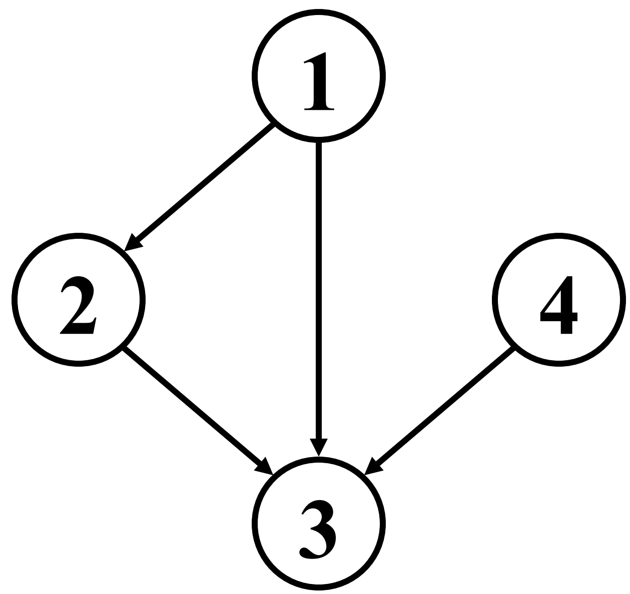
  No self-loops → don’t need diagonal

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 respresentation

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



Graphs: Adjacency List

- How much storage is required?

  The *degree* of a vertex v = # incident edges

    Directed graphs have in-degree, out-degree

  For directed graphs, # of items in adjacency lists is

  Σ out-degree(v) = |E|

    takes Θ(V + E) storage (*Why?*)

  For undirected graphs, # items in adj lists is

  Σ degree(v) = 2 |E| (*handshaking lemma*)

    also Θ(V + E) storage
- So: Adjacency lists take O(V+E) storage

Graph Searching

- Given: a graph $G = (V, E)$, directed or undirected
- Goal: methodically explore every vertex and every edge
- Ultimately: build a tree on the graph
  Pick a vertex as the root
  Choose certain edges to produce a tree
  Note: might also build a *forest* if graph is not connected

Breadth-First Search

- “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 tree over the graph
  Pick a *source vertex* to be the root
  Find (“discover”) its children, then their children, etc.

Breadth-First Search

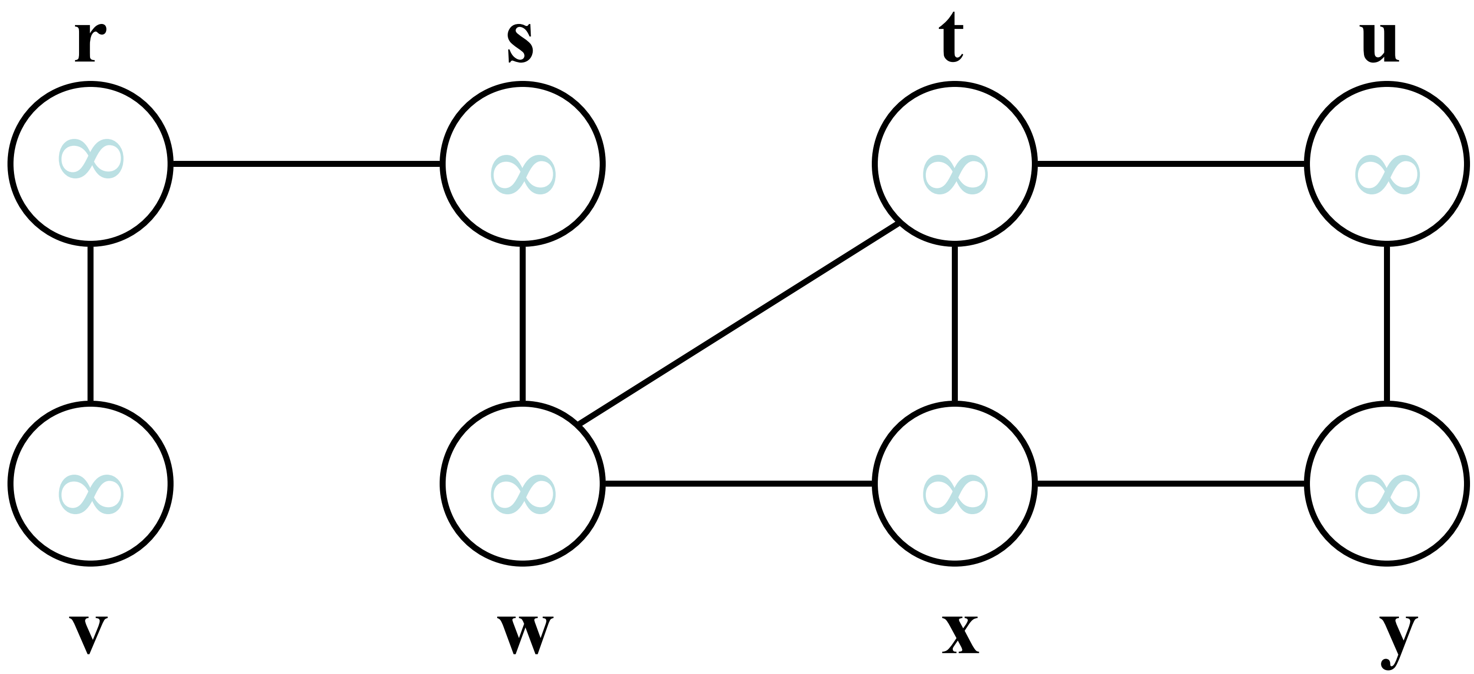
- Again will associate vertex “colors” to guide the algorithm
  - White vertices have not been discovered
    - All vertices start out white
  - Grey vertices are discovered but not fully explored
    - They may be adjacent to white vertices
  - Black vertices are discovered and fully explored
    - They are adjacent only to black and gray vertices
- Explore vertices by scanning adjacency list of grey vertices

Breadth-First Search

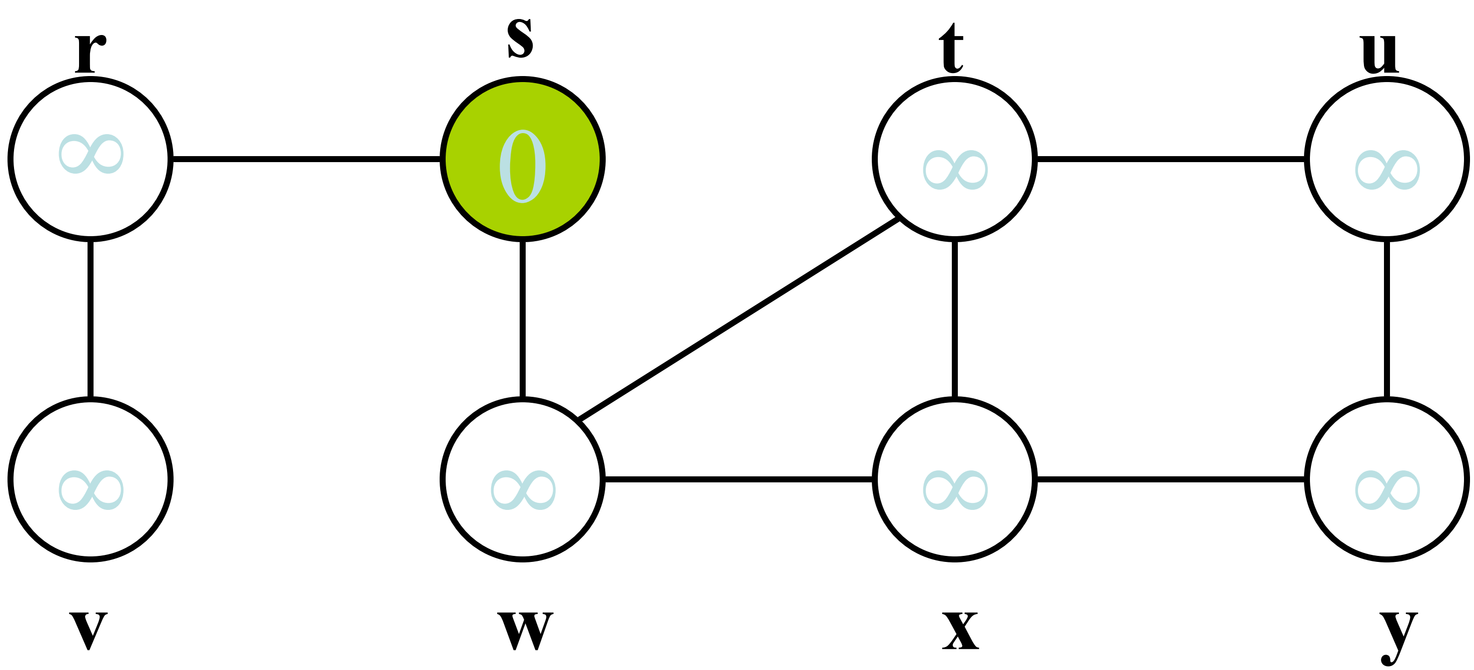
```
BFS(G, s) {
    initialize vertices;
    Q = {s};          // Q is a queue (duh); initialize
  to s
    while (Q not empty) {
        u = RemoveTop(Q);
        for each v ∈ u->adj {
            if (v->color == WHITE)
                v->color = GREY;
                v->d = u->d + 1;
             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

What does `v->d` represent?
What does `v->p` represent?

Breadth-First Search: Example

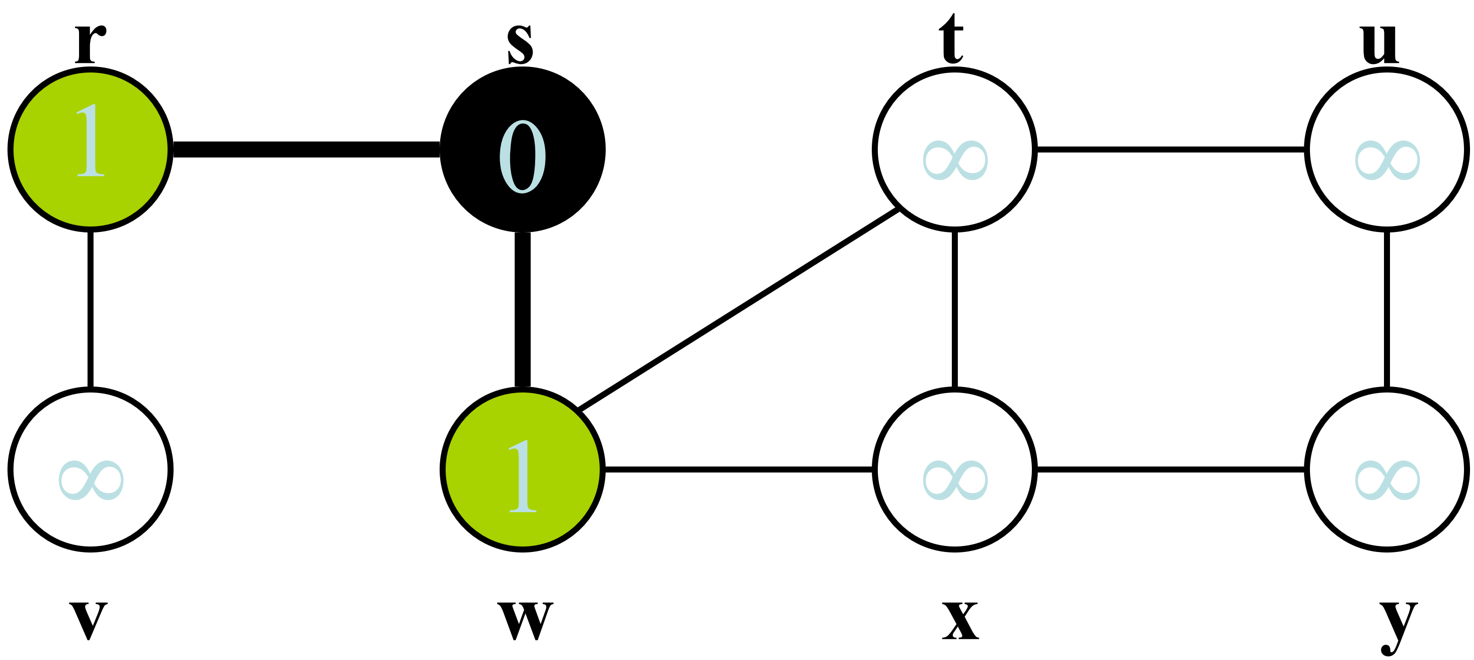


Breadth-First Search: Example



Q: s

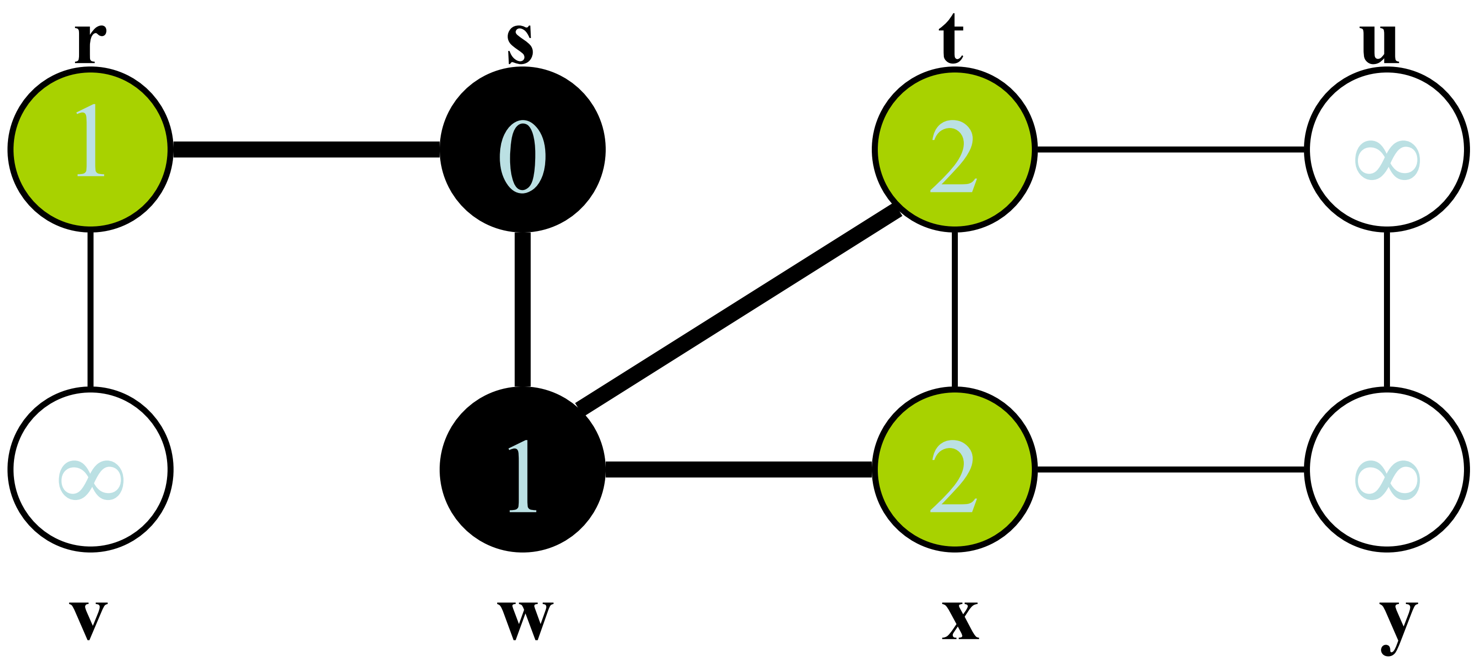
Breadth-First Search: Example



Q:

w	r

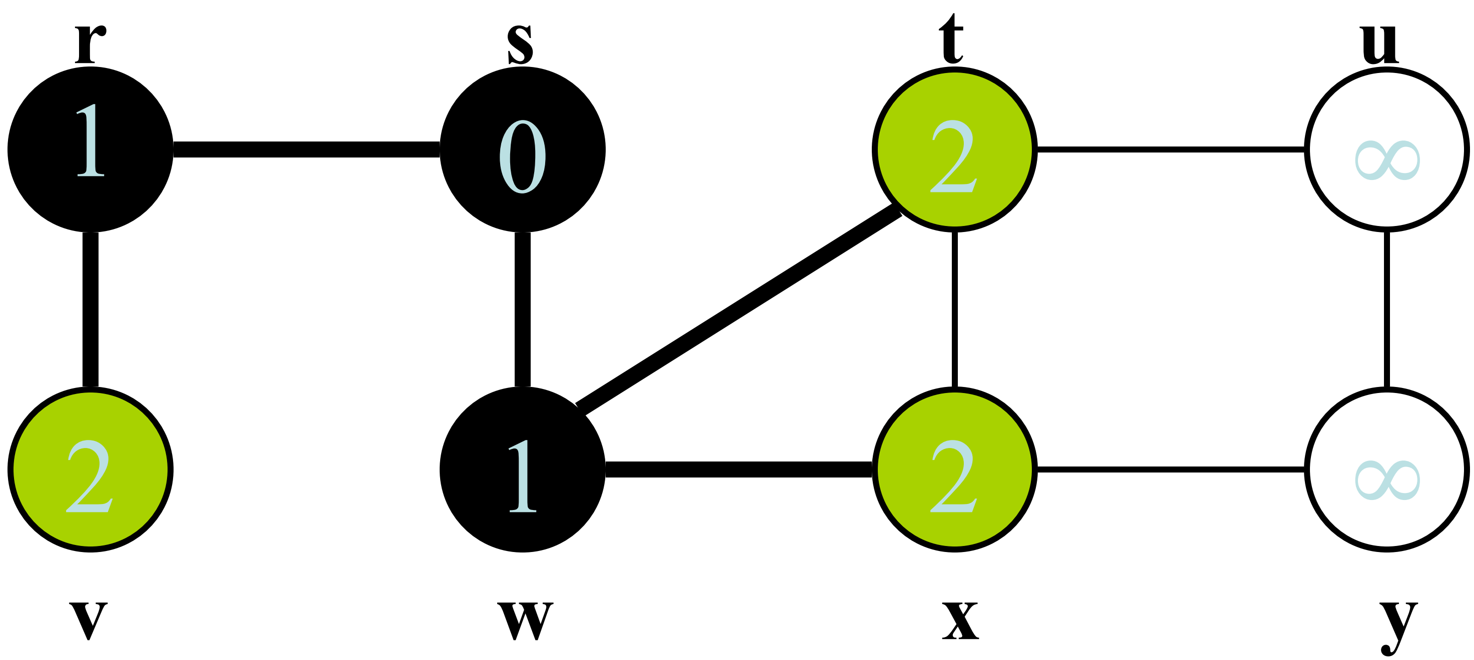
Breadth-First Search: Example



Q:

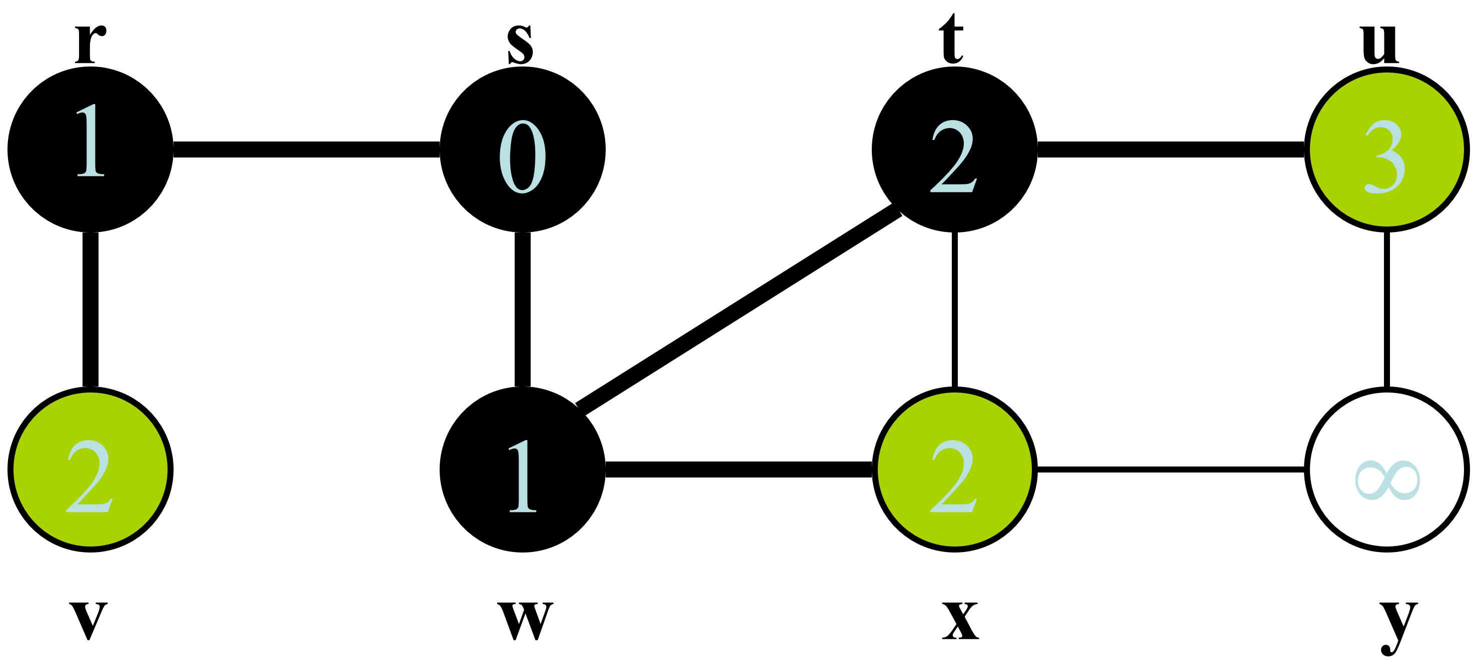
r	t	x

Breadth-First Search: Example



Q:	t	x	v

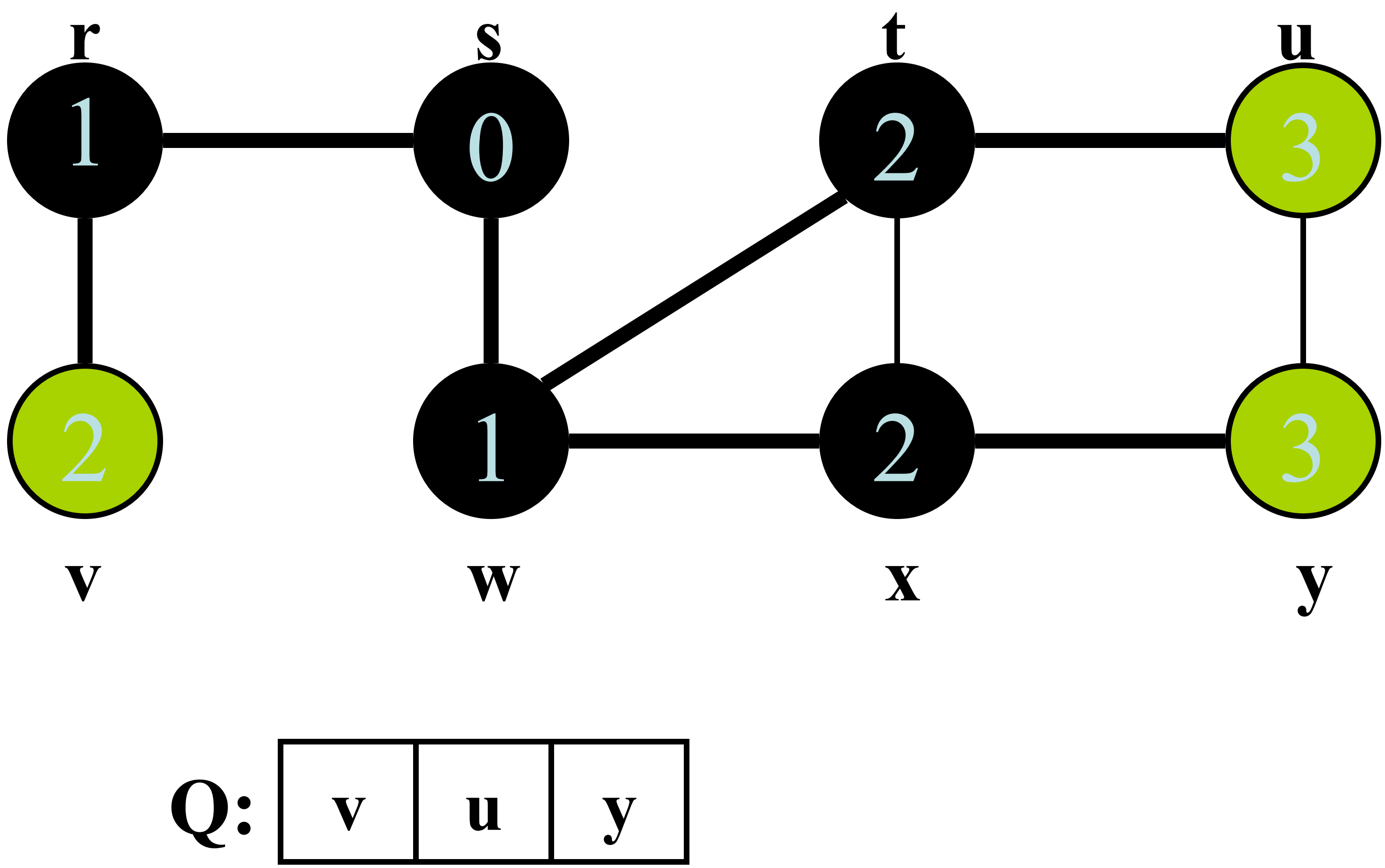
Breadth-First Search: Example



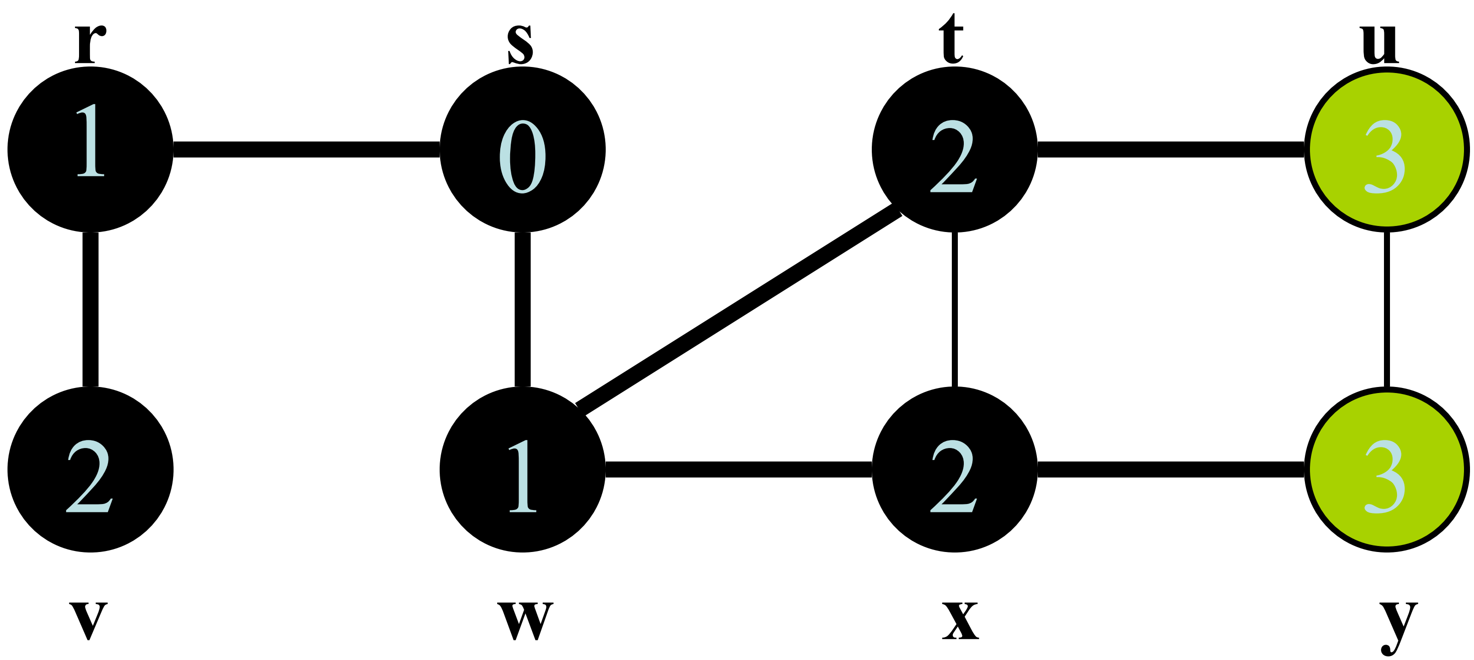
Q:

x	v	u

Breadth-First Search: Example

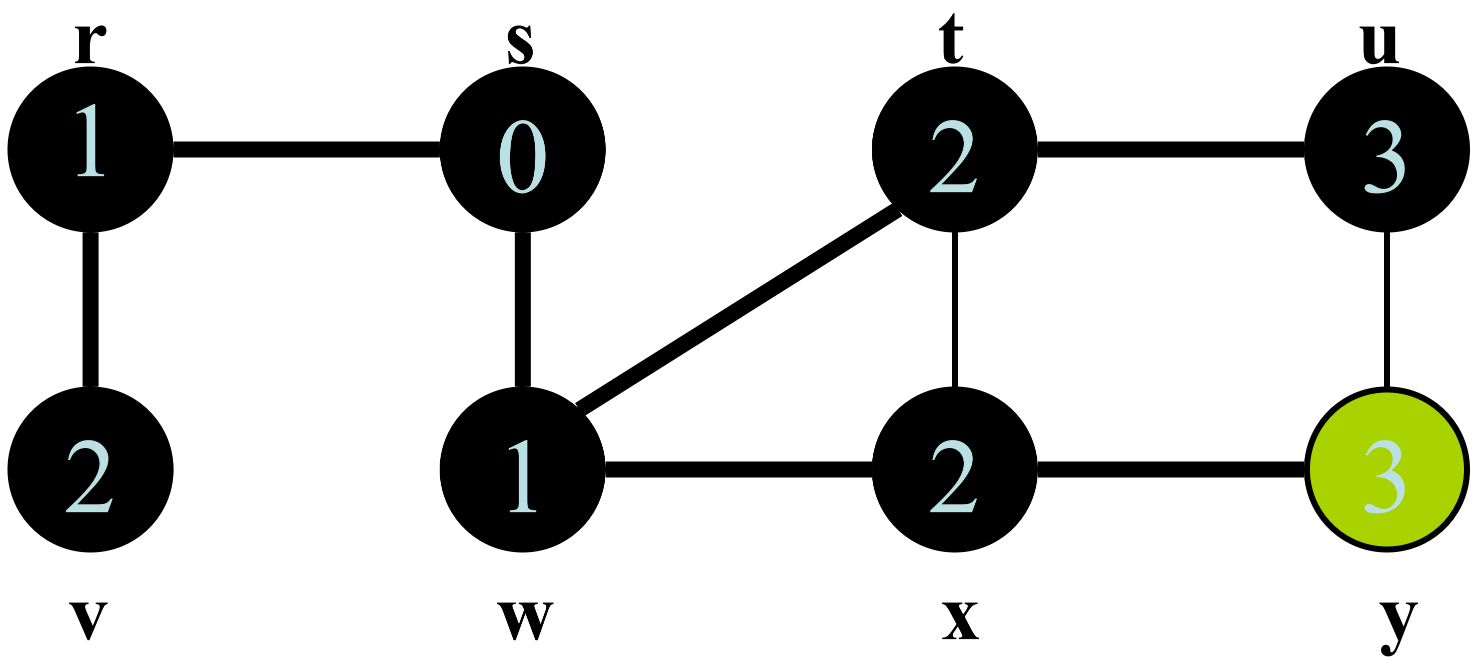


Breadth-First Search: Example



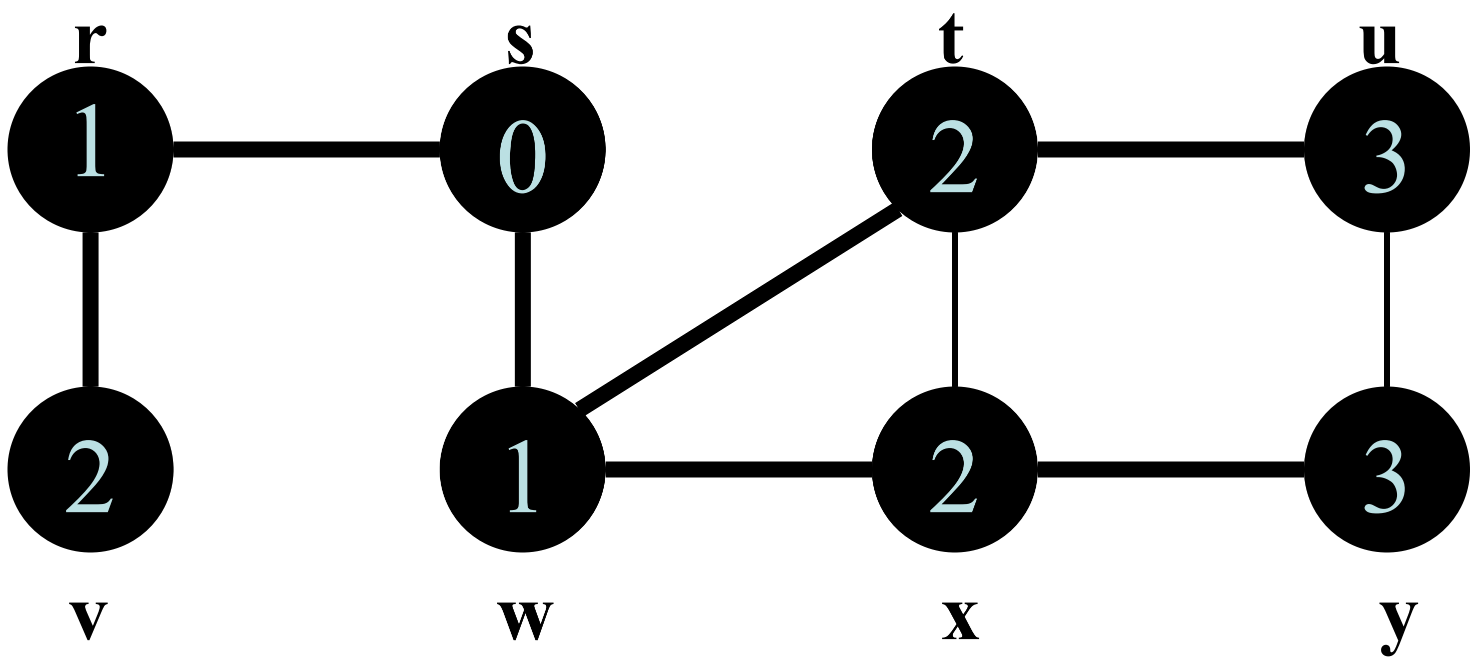
Q: | u | y |

Breadth-First Search: Example



Q: y

Breadth-First Search: Example



Q: Ø

BFS: The Code Again

```
BFS(G, s) {
    initialize vertices;
    Q = {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;
    }
}
```

Touch every vertex: O(V)

u = every vertex, but only once (Why?)

So v = every vertex that appears in some other vert's adjacency list

What will be the running time?

Total running time: O(V+E)

BFS: The Code Again

```
BFS(G, s) {
    initialize vertices;
    Q = {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 will be the storage cost in addition to storing the tree?

Total space used:

O(max(degree(v))) = O(E)

Breadth-First Search: Properties

- BFS calculates the *shortest-path distance* to the source node
- 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

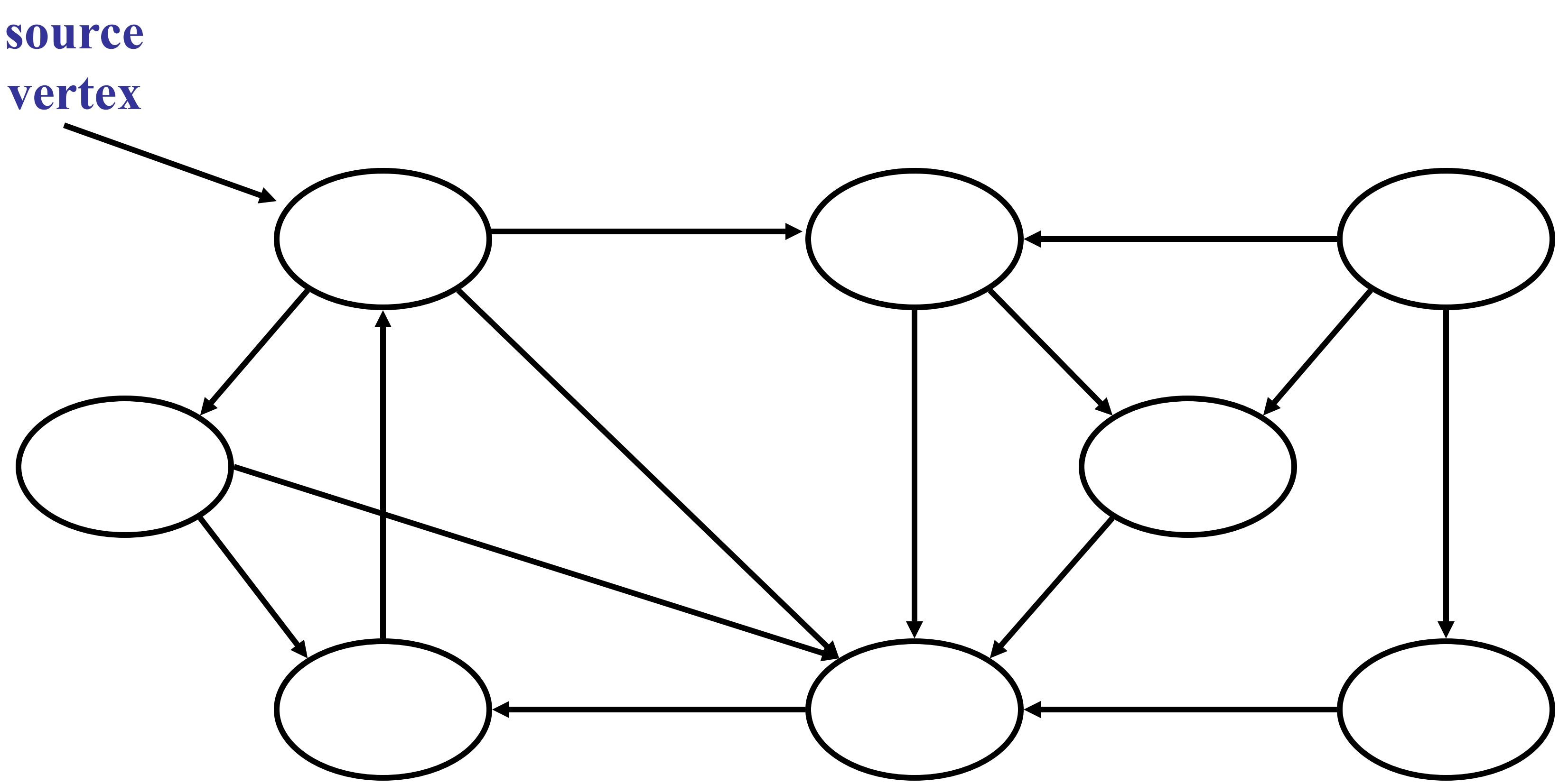
Depth-First Search

- *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

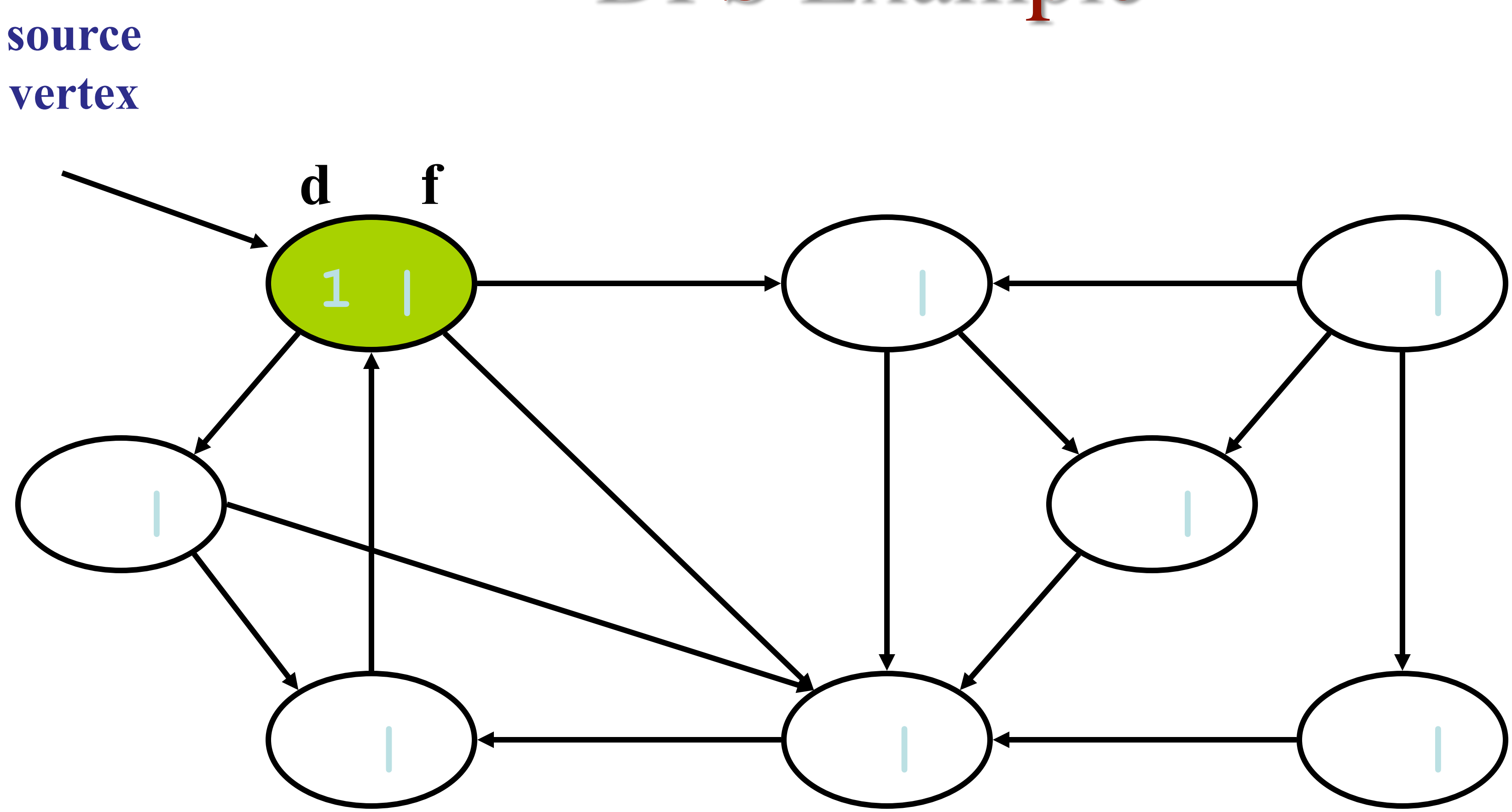
Depth-First Search

- Vertices initially colored white
- Then colored gray when discovered
- Then black when finished

DFS Example

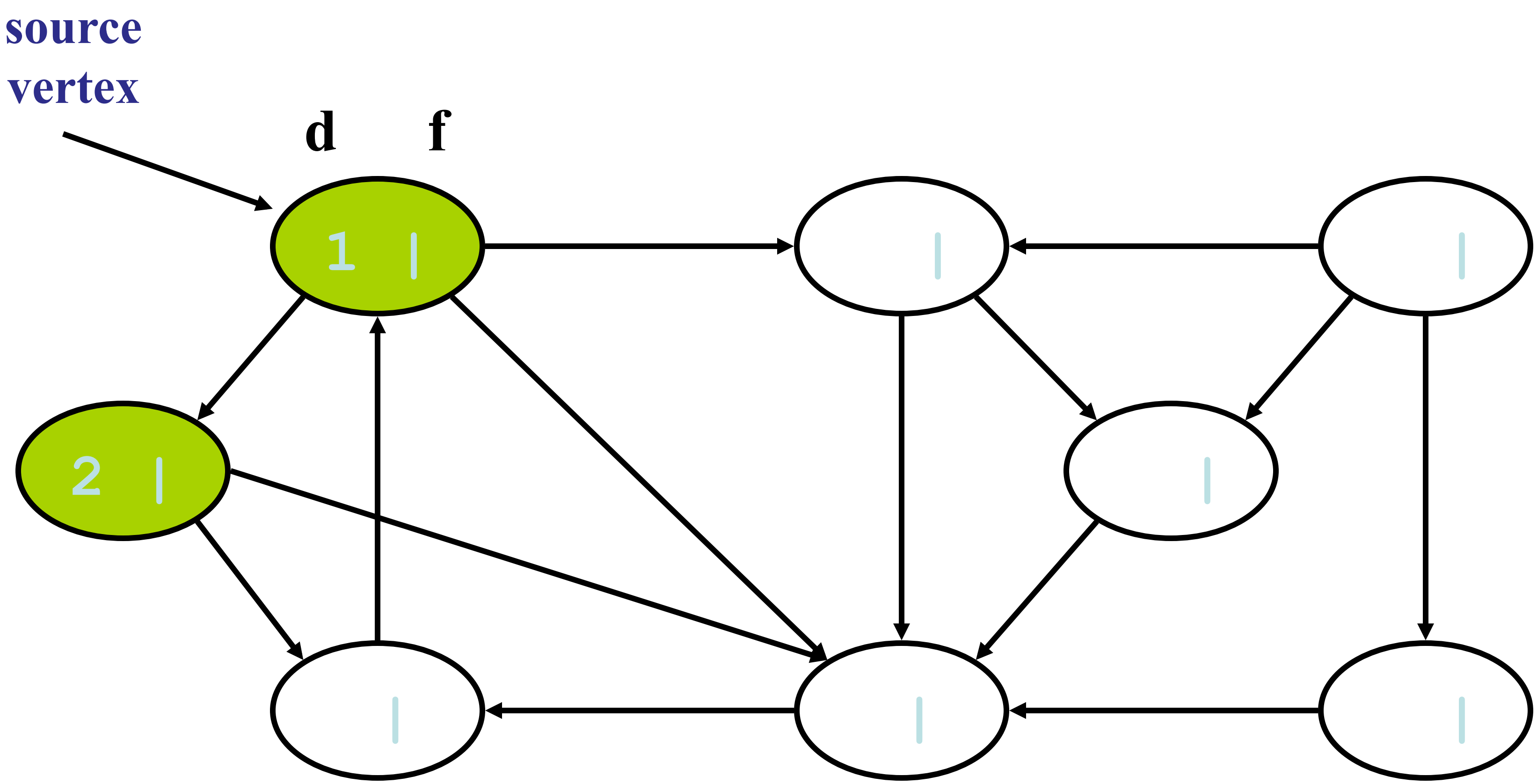


DFS Example

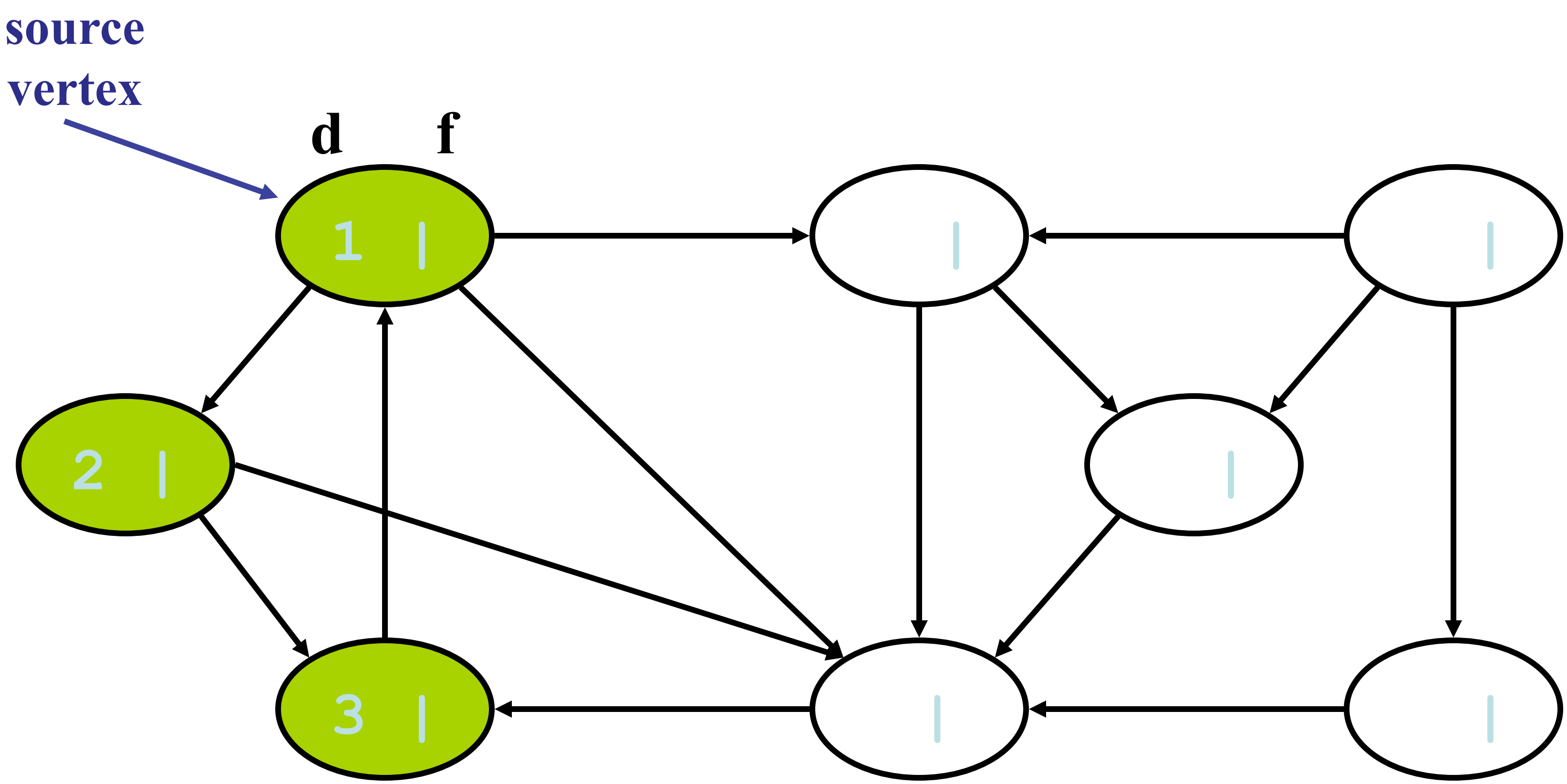


Green in figure -> gray in code

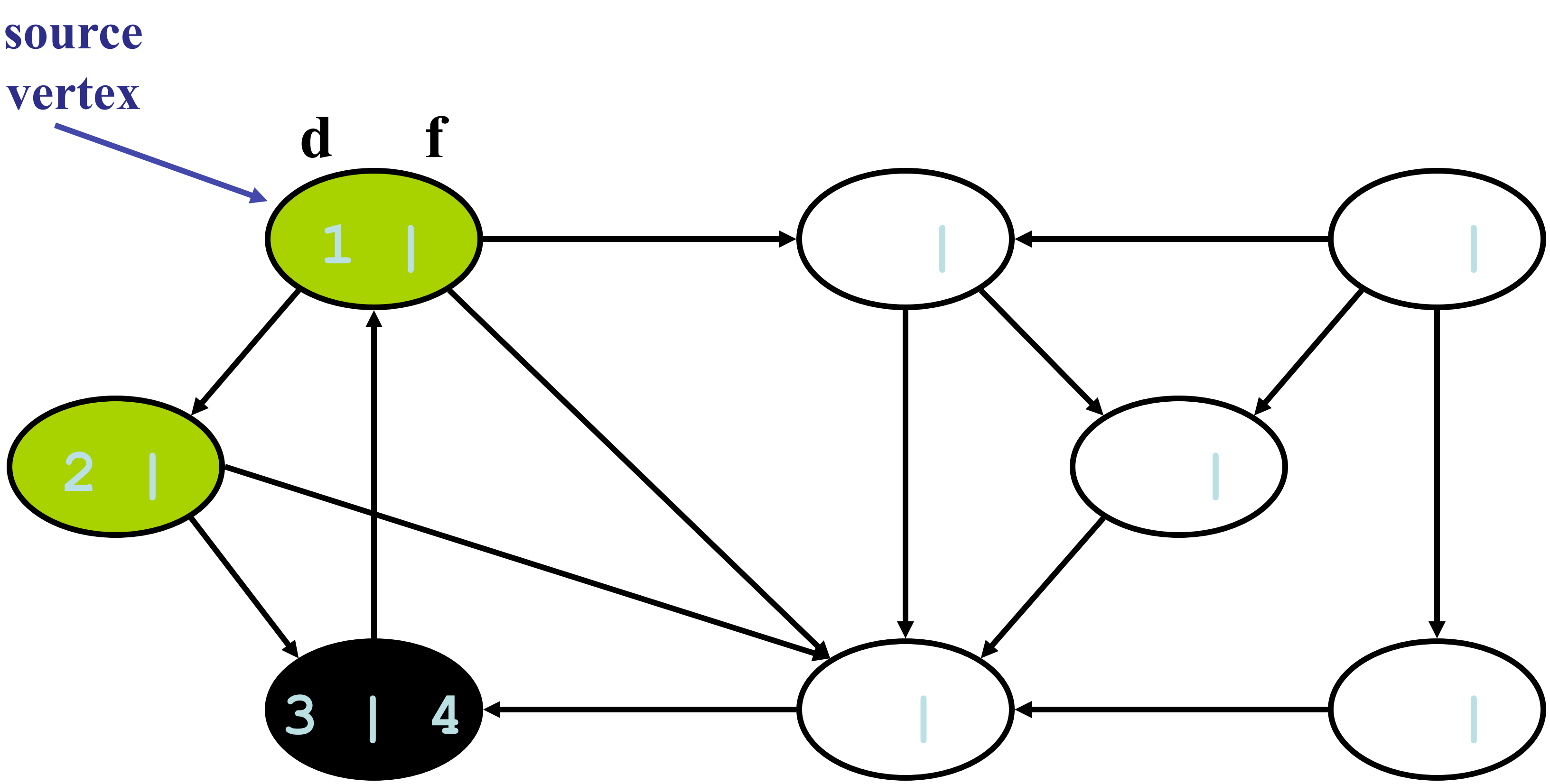
DFS Example



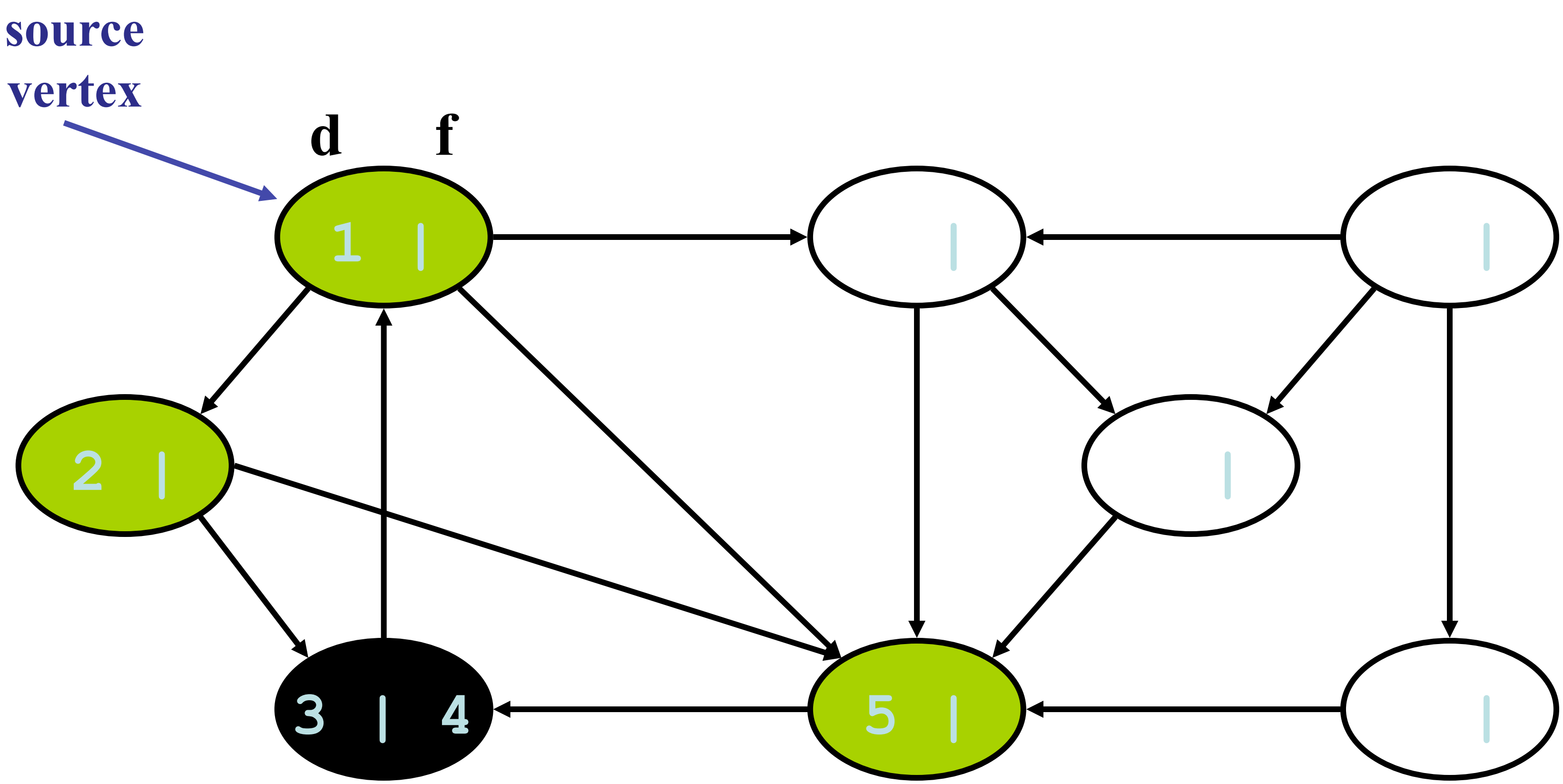
DFS Example



DFS Example



DFS Example



Depth-First Search: The Code

```
DFS(G)
{
    for each vertex u ∈ G->V
    {
        u->color = WHITE;
    }
    time = 0;
    for each vertex u ∈ G->V
    {
        if (u->color ==
  WHITE)
            DFS_Visit(u);
    }
}
```

```
DFS_Visit(u)
{
    u->color = GREY;
    time = time+1;
    u->d = time;
    for each v ∈ u->Adj[]
    {
        if (v->color ==
  WHITE)
            DFS_Visit(v);
    }
    u->color = BLACK;
    time = time+1;
    u->f = time;
}
```

Depth-First Search: The Code

```
DFS(G)
{
    for each vertex u ∈ G->V
    {
        u->color = WHITE;
    }
    time = 0;
    for each vertex u ∈ G->V
    {
        if (u->color ==
  WHITE)
            DFS_Visit(u);
    }
}
```

```
DFS_Visit(u)
{
    u->color = GREY;
    time = time+1;
    u->d = time;
    for each v ∈ u->Adj[]
    {
        if (v->color ==
  WHITE)
            DFS_Visit(v);
    }
    u->color = BLACK;
    time = time+1;
    u->f = time;
}
```

What does `u->d` represent?

Depth-First Search: The Code

```
DFS(G)
{
    for each vertex u ∈ G->V
    {
        u->color = WHITE;
    }
    time = 0;
    for each vertex u ∈ G->V
    {
        if (u->color ==
  WHITE)
            DFS_Visit(u);
    }
}
```

```
DFS_Visit(u)
{
    u->color = GREY;
    time = time+1;
    u->d = time;
    for each v ∈ u->Adj[]
    {
        if (v->color ==
  WHITE)
            DFS_Visit(v);
    }
    u->color = BLACK;
    time = time+1;
    u->f = time;
}
```

What does `u->f` represent?

Depth-First Search: The Code

```
DFS(G)
{
    for each vertex u ∈ G->V
    {
        u->color = WHITE;
    }
    time = 0;
    for each vertex u ∈ G->V
    {
        if (u->color ==
  WHITE)
            DFS_Visit(u);
    }
}
```

```
DFS_Visit(u)
{
    u->color = GREY;
    time = time+1;
    u->d = time;
    for each v ∈ u->Adj[]
    {
        if (v->color ==
  WHITE)
            DFS_Visit(v);
    }
    u->color = BLACK;
    time = time+1;
    u->f = time;
}
```

Will all vertices eventually be colored black?

Depth-First Search: The Code

```
DFS(G)
{
    for each vertex u ∈ G->V
    {
        u->color = WHITE;
    }
    time = 0;
    for each vertex u ∈ G->V
    {
        if (u->color ==
  WHITE)
            DFS_Visit(u);
    }
}
```

```
DFS_Visit(u)
{
    u->color = GREY;
    time = time+1;
    u->d = time;
    for each v ∈ u->Adj[]
     {
         if (v->color ==
   WHITE)
             DFS_Visit(v);
     }
    u->color = BLACK;
    time = time+1;
    u->f = time;
}
```

What will be the running time?

Depth-First Search: The Code

```
DFS(G)
{
    for each vertex u ∈ G->V
    {
        u->color = WHITE;
    }
    time = 0;
    for each vertex u ∈ G->V
    {
        if (u->color ==
  WHITE)
            DFS_Visit(u);
    }
}
```

```
DFS_Visit(u)
{
    u->color = GREY;
    time = time+1;
    u->d = time;
    for each v ∈ u->Adj[]
    {
        if (v->color ==
  WHITE)
            DFS_Visit(v);
    }
    u->color = BLACK;
    time = time+1;
    u->f = time;
}
```

Running time: $O(n^2)$ because call DFS_Visit on each vertex, and the loop over Adj[] can run as many as |V| times

Depth-First Search: The Code

```
DFS(G)
{
    for each vertex u ∈ G->V
    {
        u->color = WHITE;
    }
    time = 0;
    for each vertex u ∈ G->V
    {
        if (u->color ==
  WHITE)
            DFS_Visit(u);
    }
}
```

```
DFS_Visit(u)
{
    u->color = GREY;
    time = time+1;
    u->d = time;
    for each v ∈ u->Adj[]
    {
        if (v->color ==
  WHITE)
            DFS_Visit(v);
    }
    u->color = BLACK;
    time = time+1;
    u->f = time;
}
```

BUT, there is actually a tighter bound.
How many times will DFS_Visit() actually be called?

Depth-First Search: The Code

```
DFS(G)
{
    for each vertex u ∈ G->V
    {
        u->color = WHITE;
    }
    time = 0;
    for each vertex u ∈ G->V
    {
        if (u->color ==
  WHITE)
            DFS_Visit(u);
    }
}
```

```
DFS_Visit(u)
{
    u->color = GREY;
    time = time+1;
    u->d = time;
    for each v ∈ u->Adj[]
    {
        if (v->color ==
  WHITE)
            DFS_Visit(v);
    }
    u->color = BLACK;
    time = time+1;
    u->f = time;
}
```

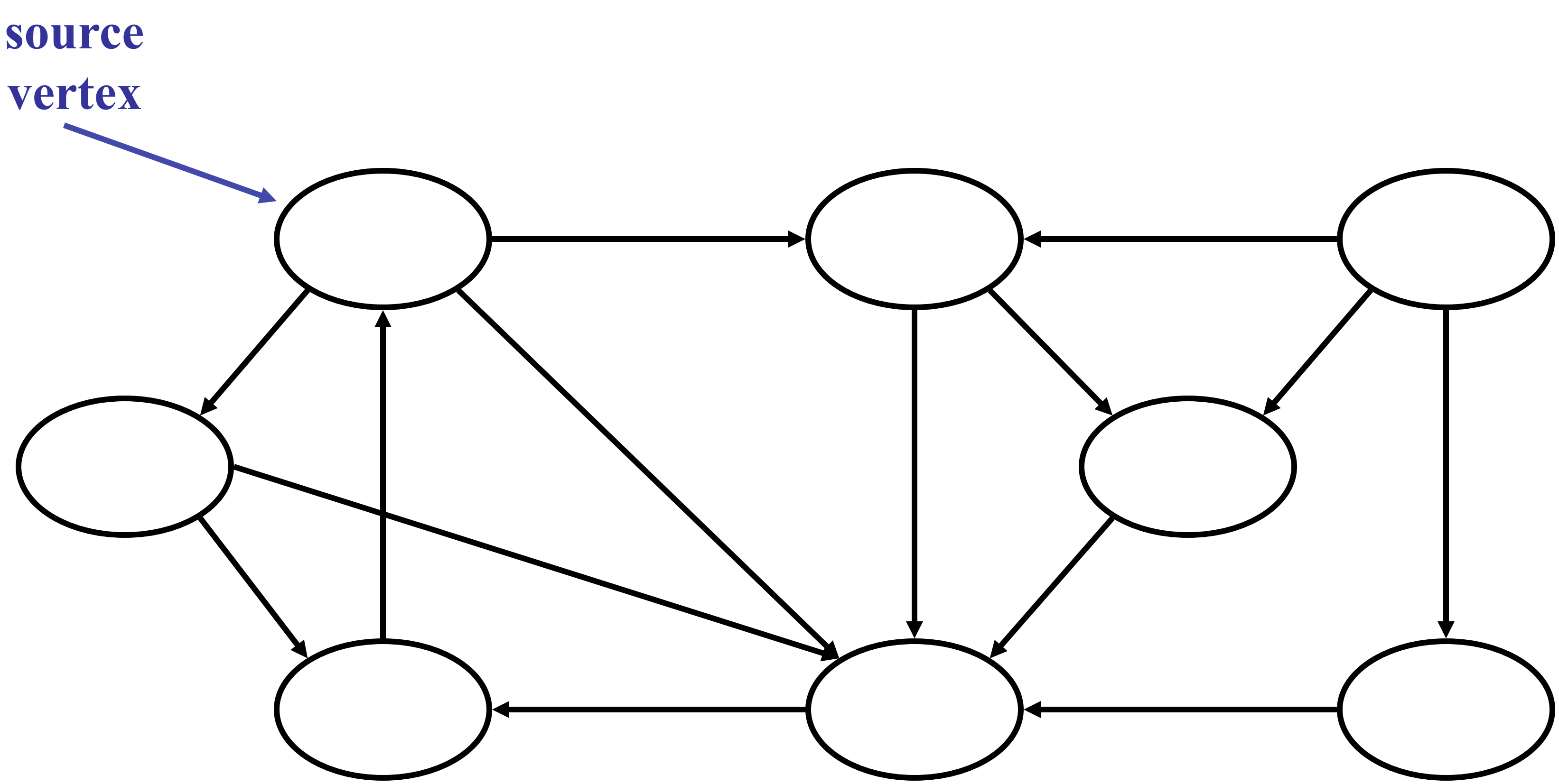
So, running time of DFS = O(V+E)

Depth-First Sort Analysis

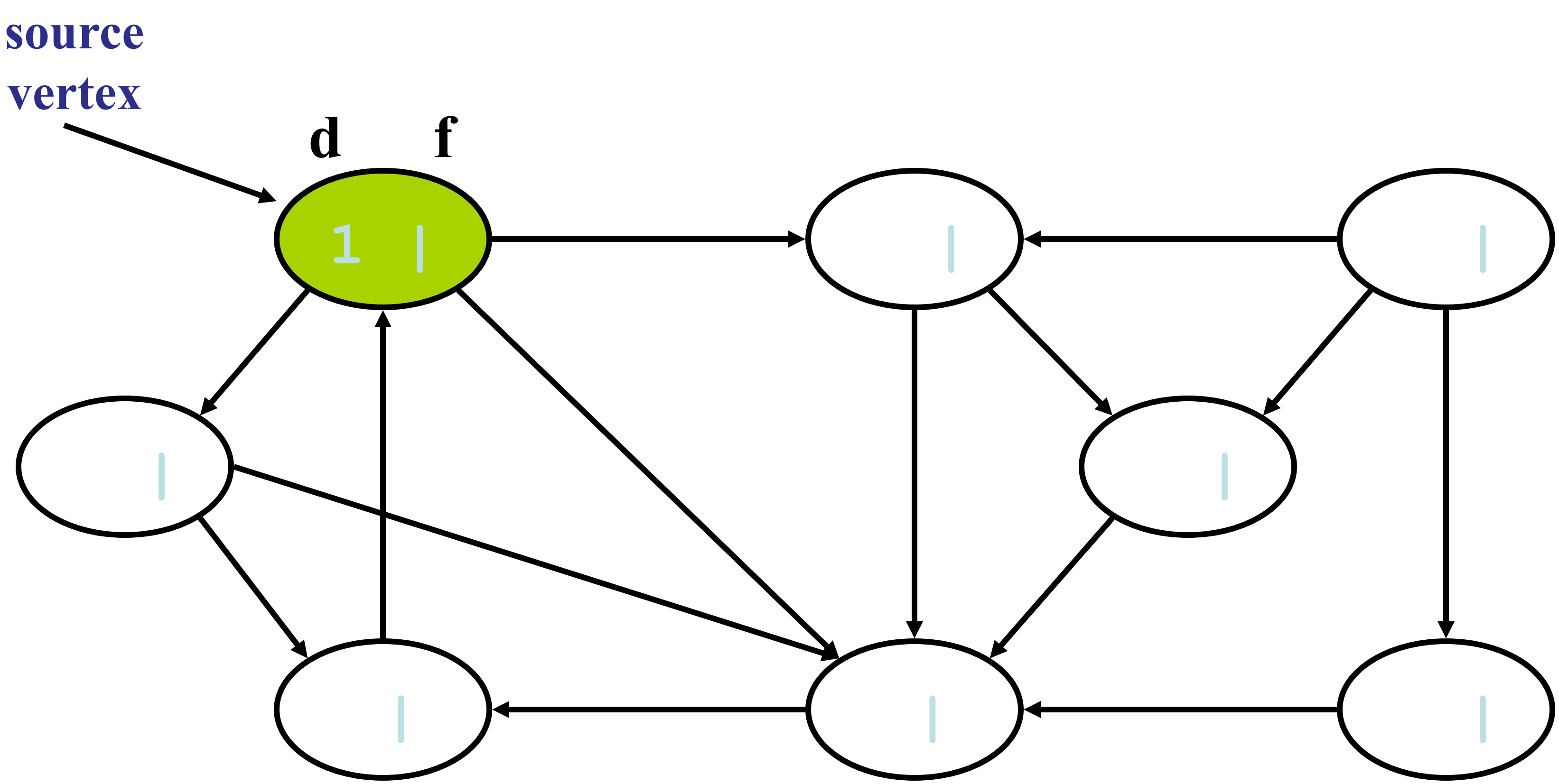
- This running time argument is an informal example of *amortized analysis*

- "Charge" the exploration of edge to the edge:
- Each loop in DFS_Visit can be attributed to an edge in the graph
- Runs once/edge if directed graph, twice if undirected
- Thus loop will run in O(E) time, algorithm O(V+E)
- Considered linear for graph, b/c adj list requires O(V+E) storage
- Important to be comfortable with this kind of reasoning and analysis

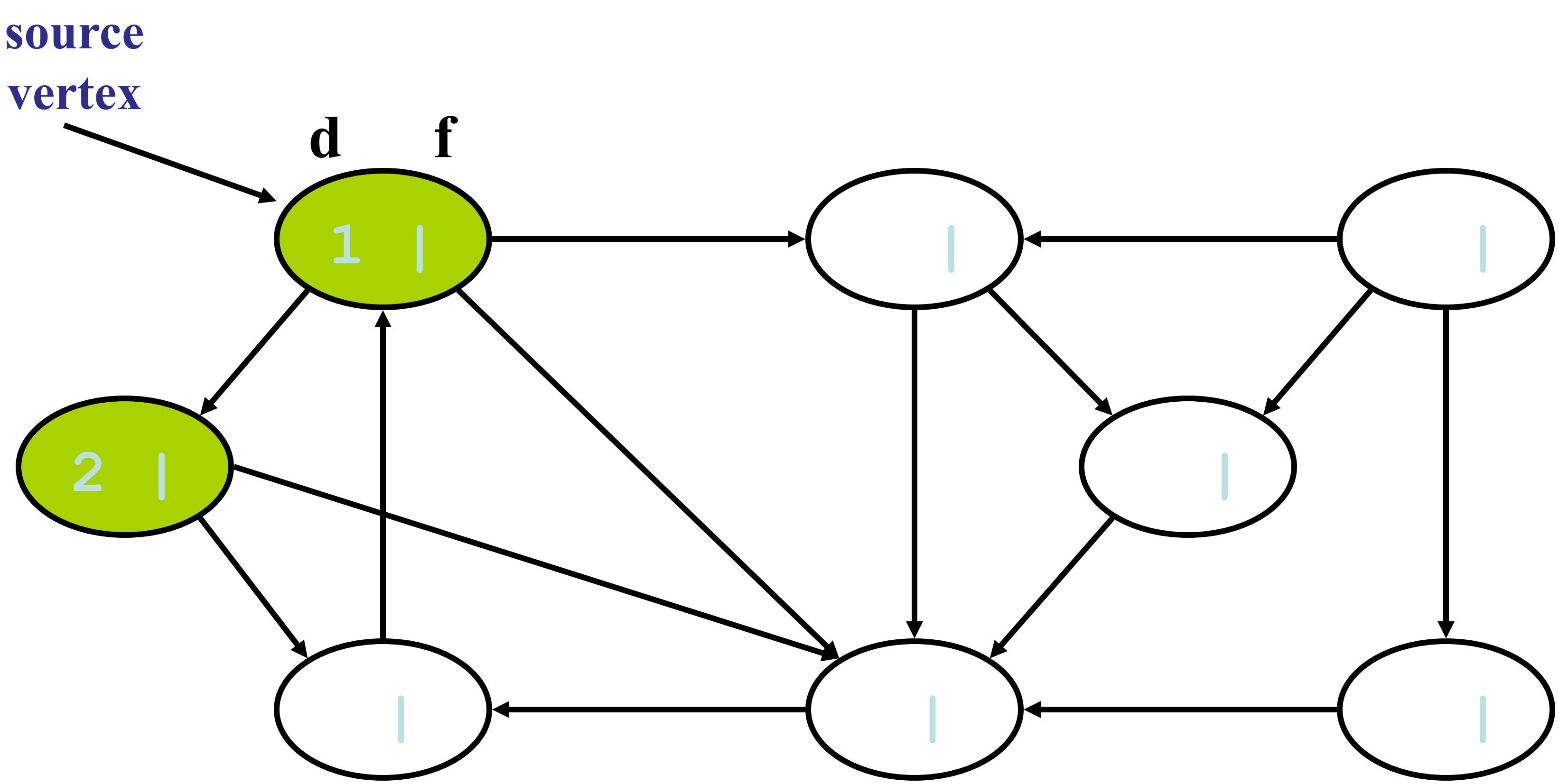
DFS Example



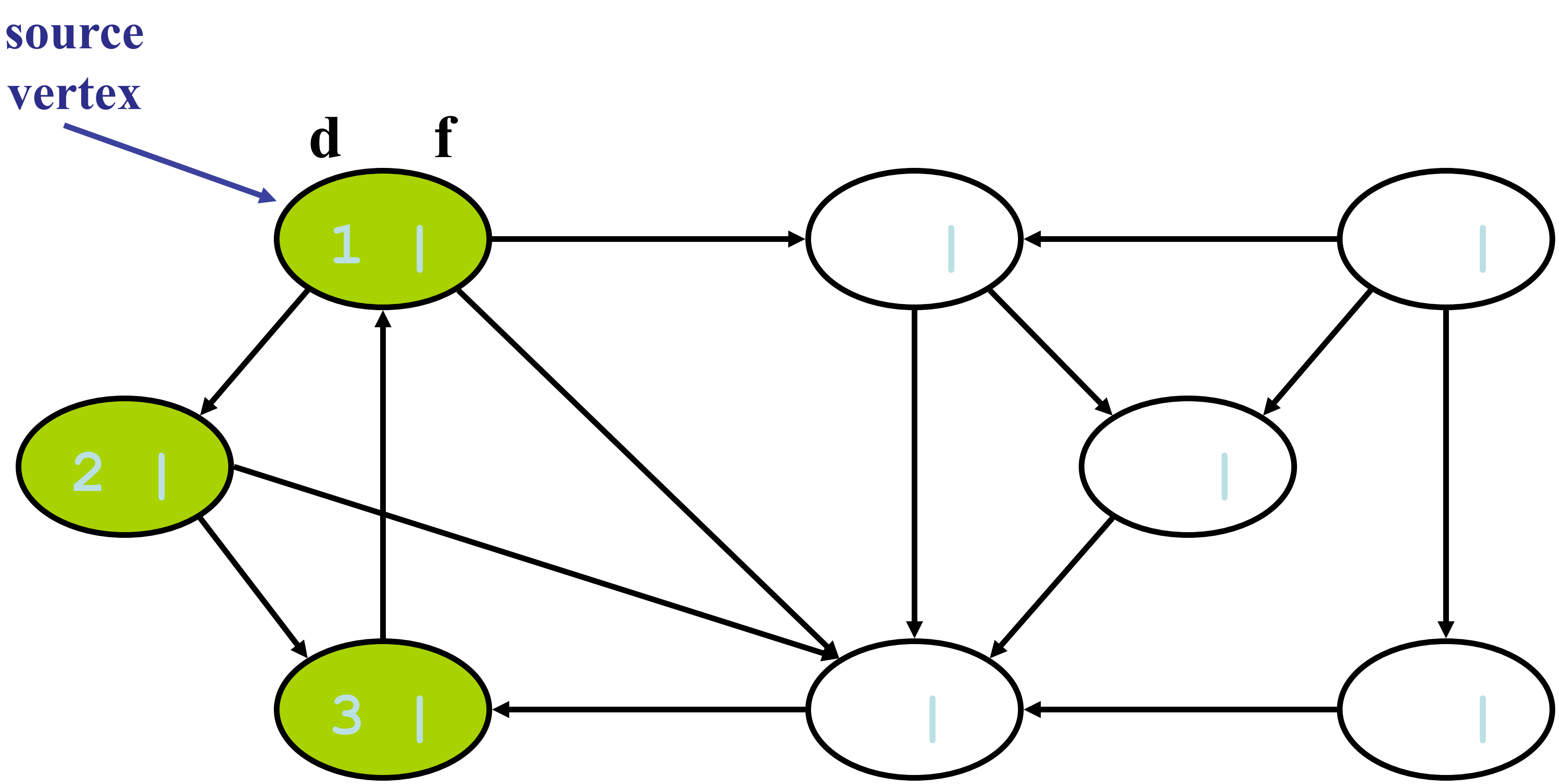
DFS Example



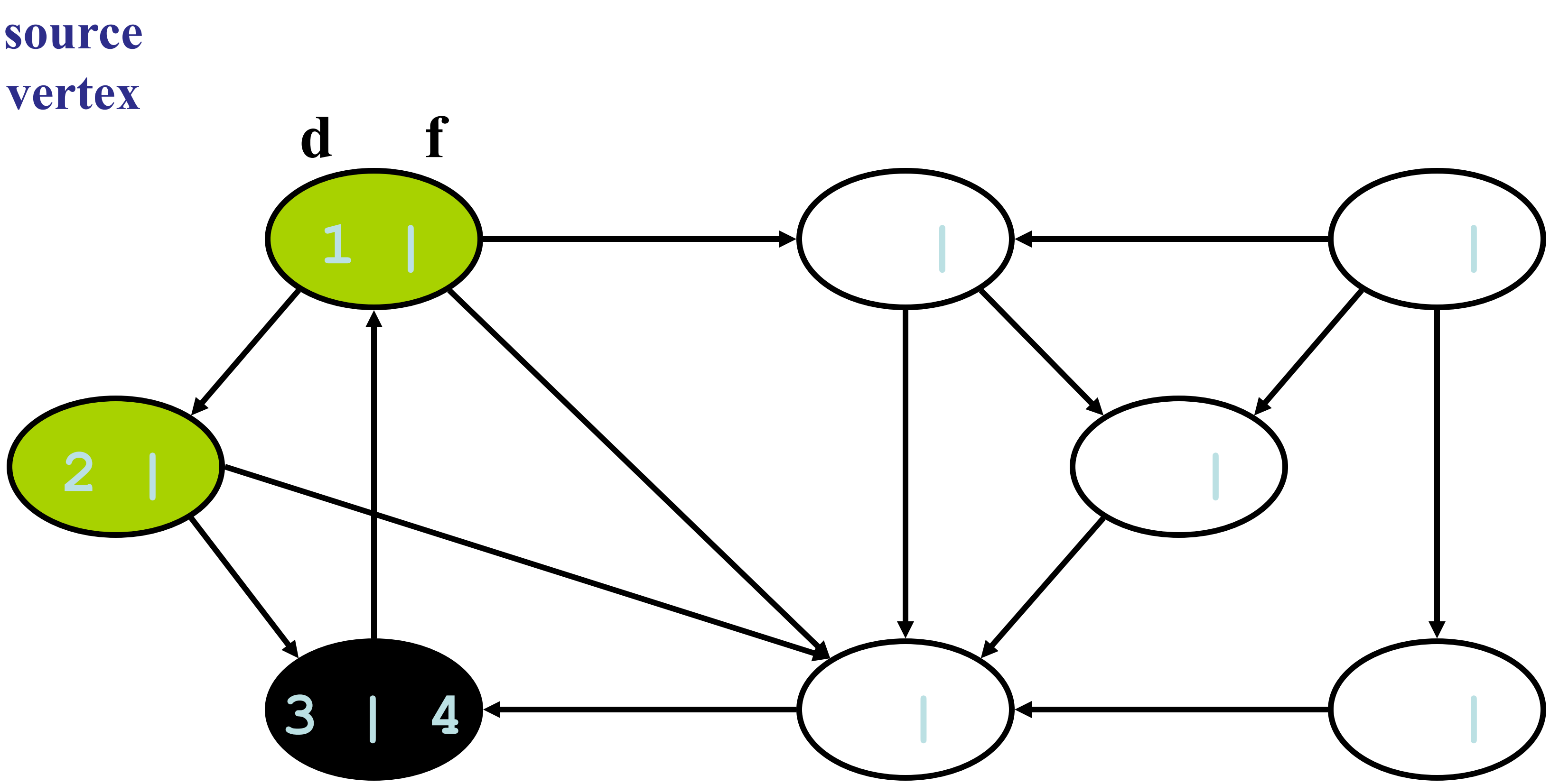
DFS Example



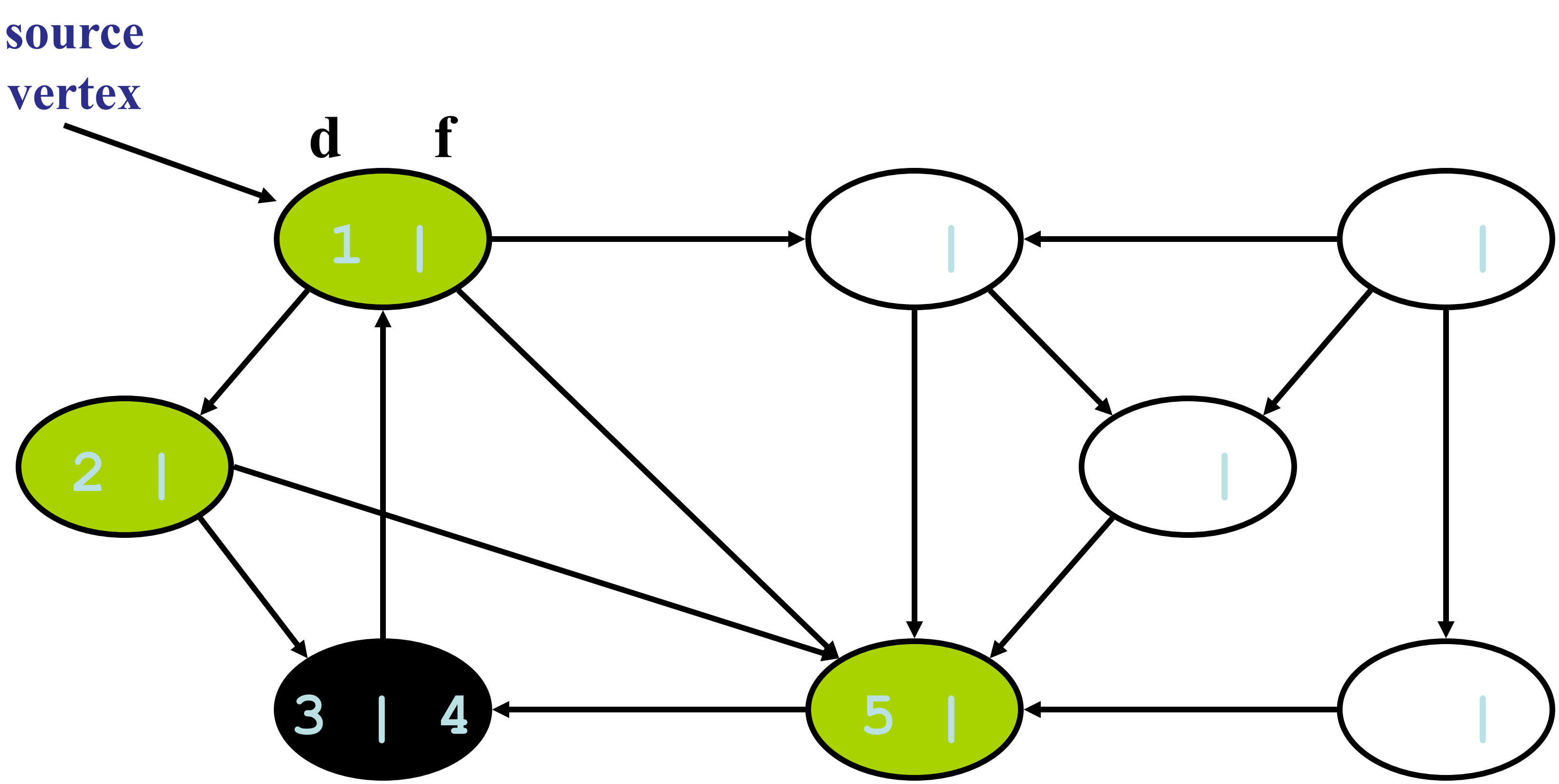
DFS Example



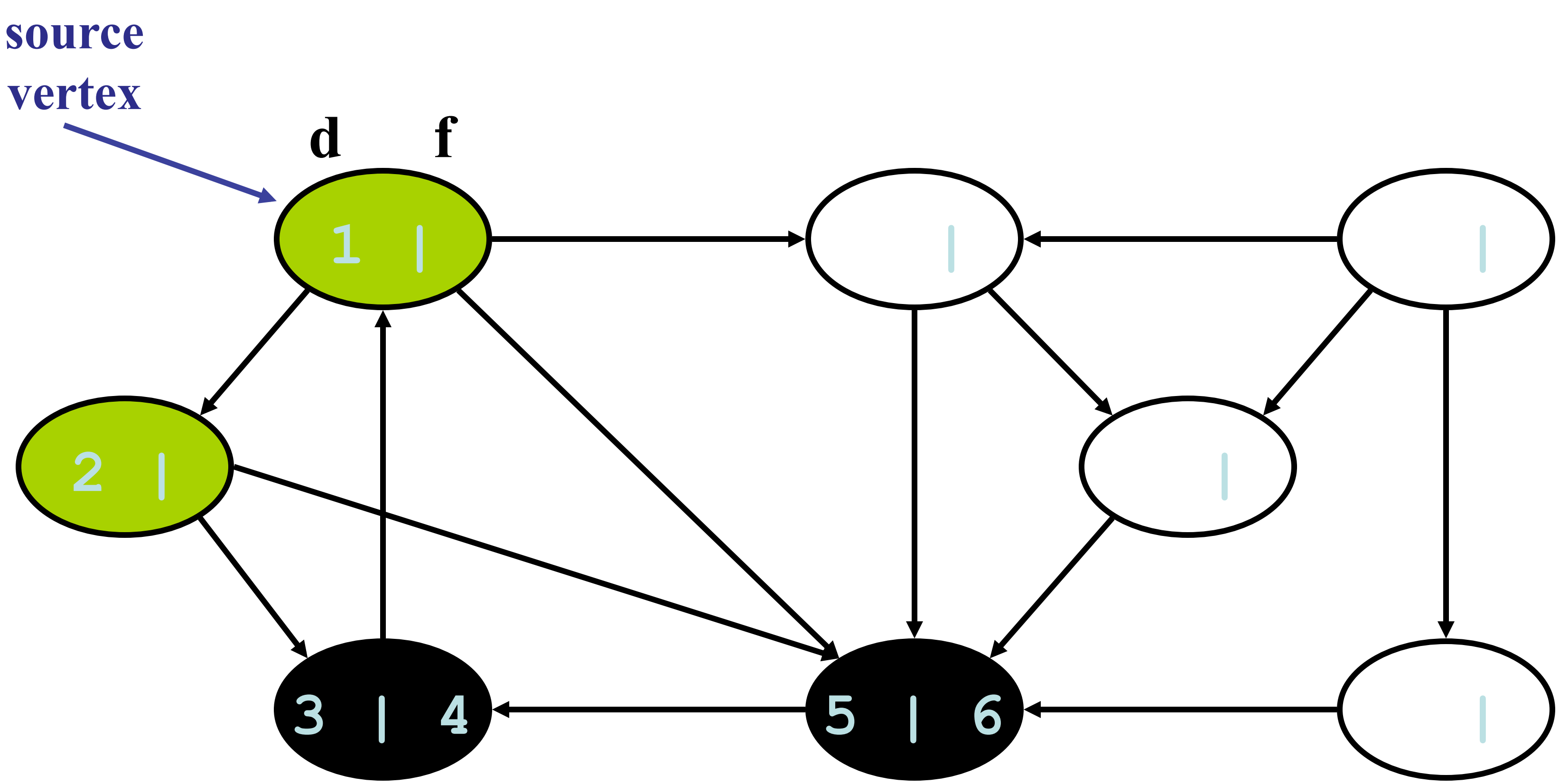
DFS Example



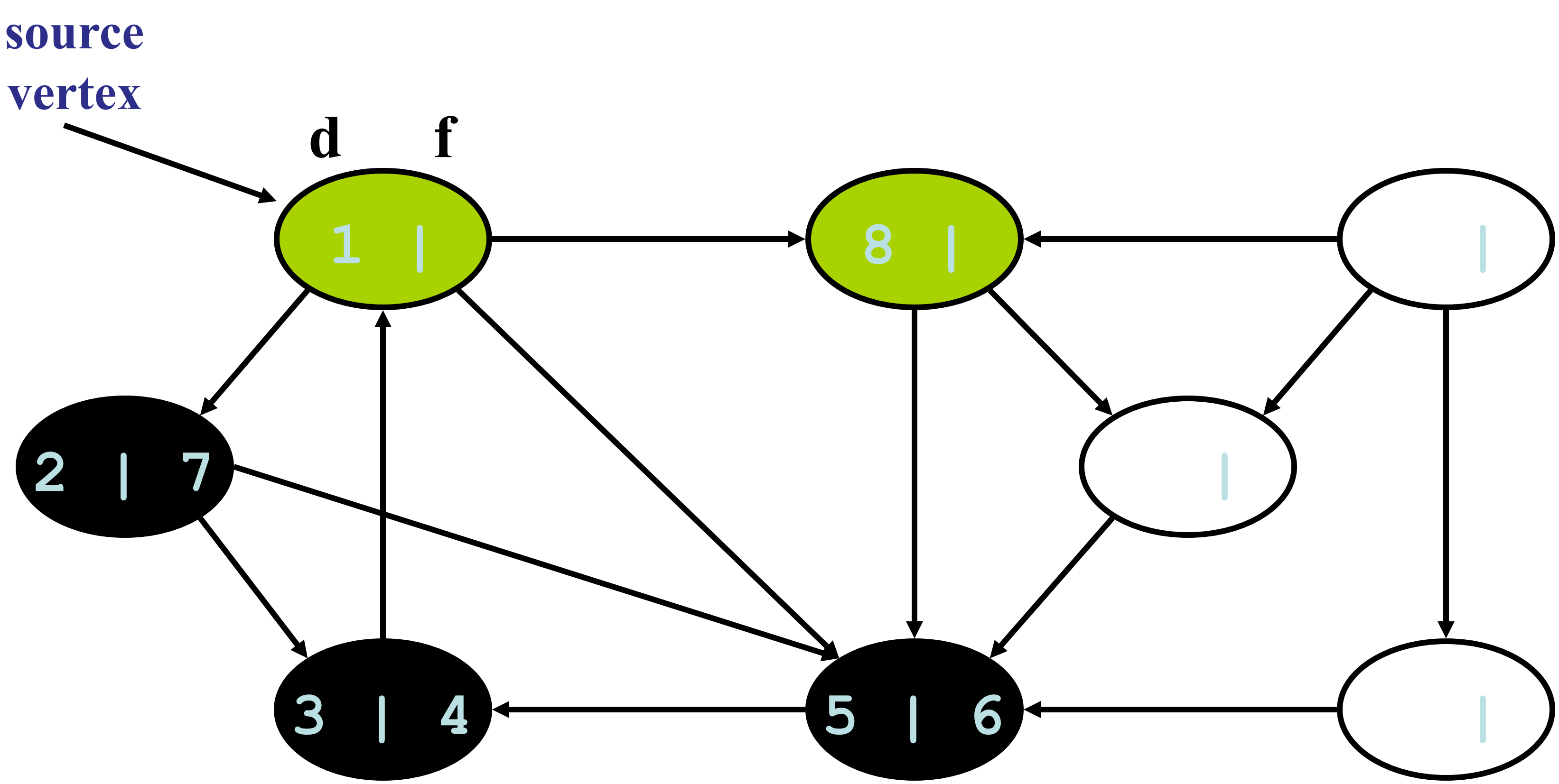
DFS Example



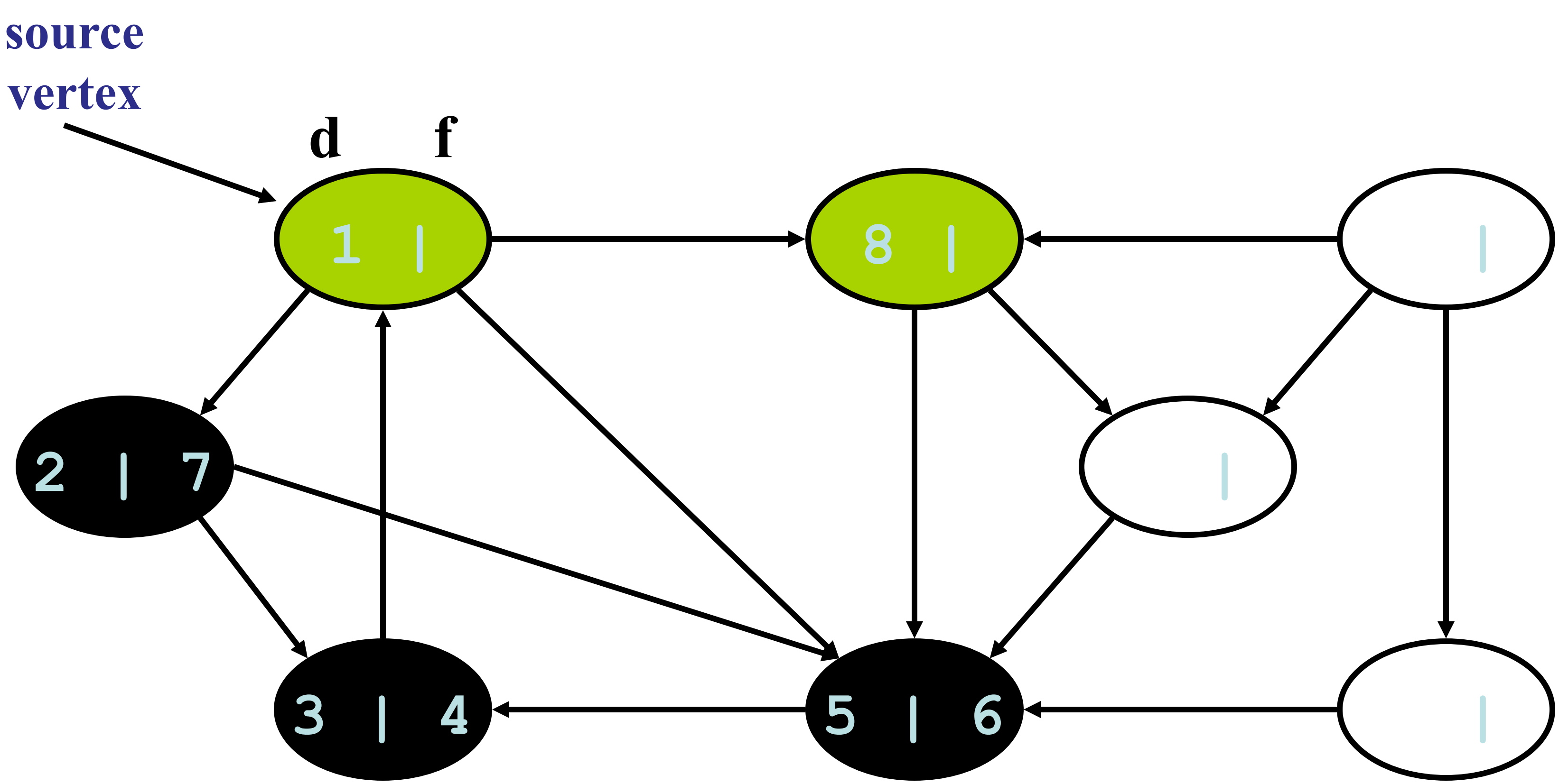
DFS Example



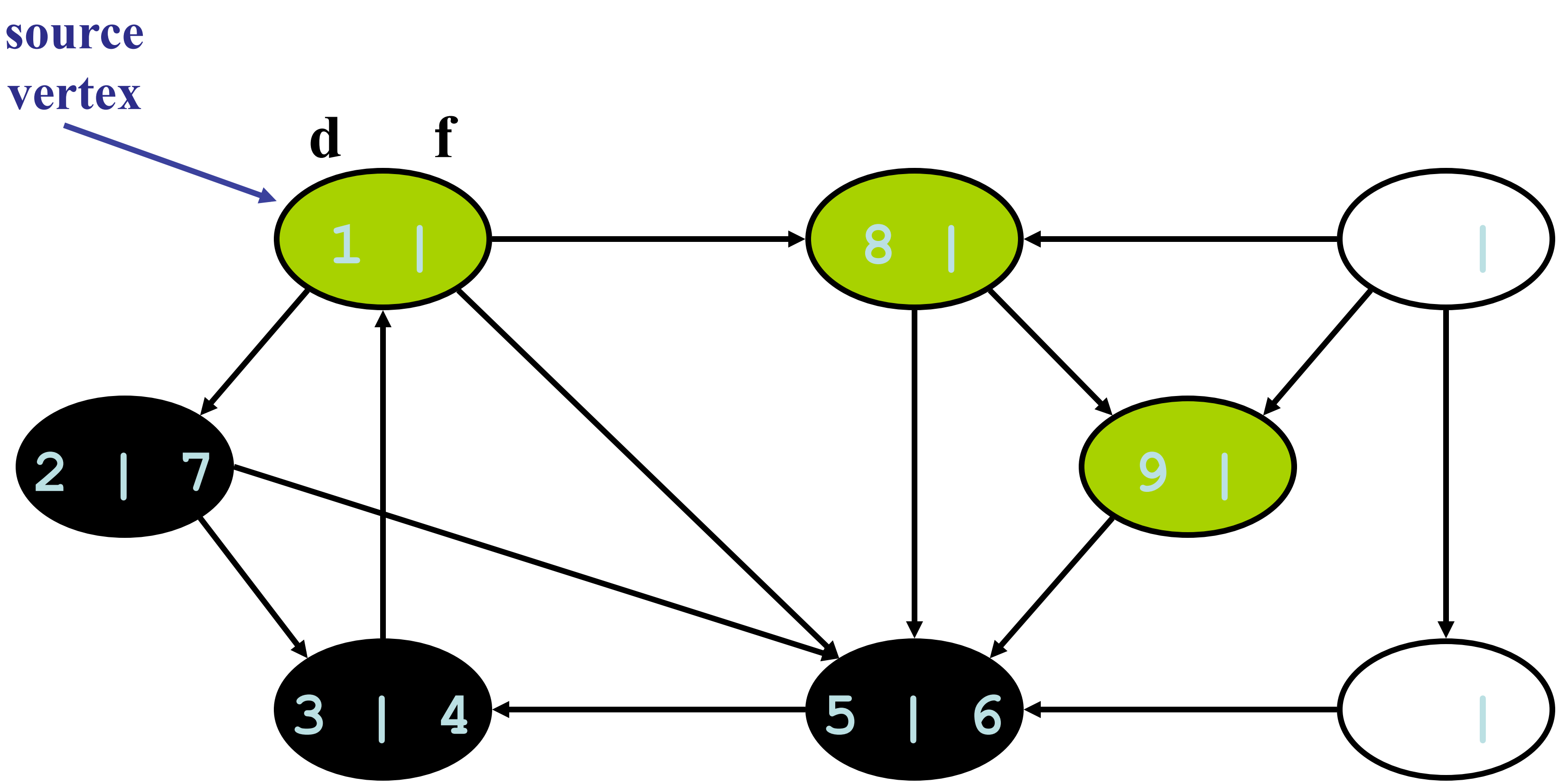
DFS Example



DFS Example

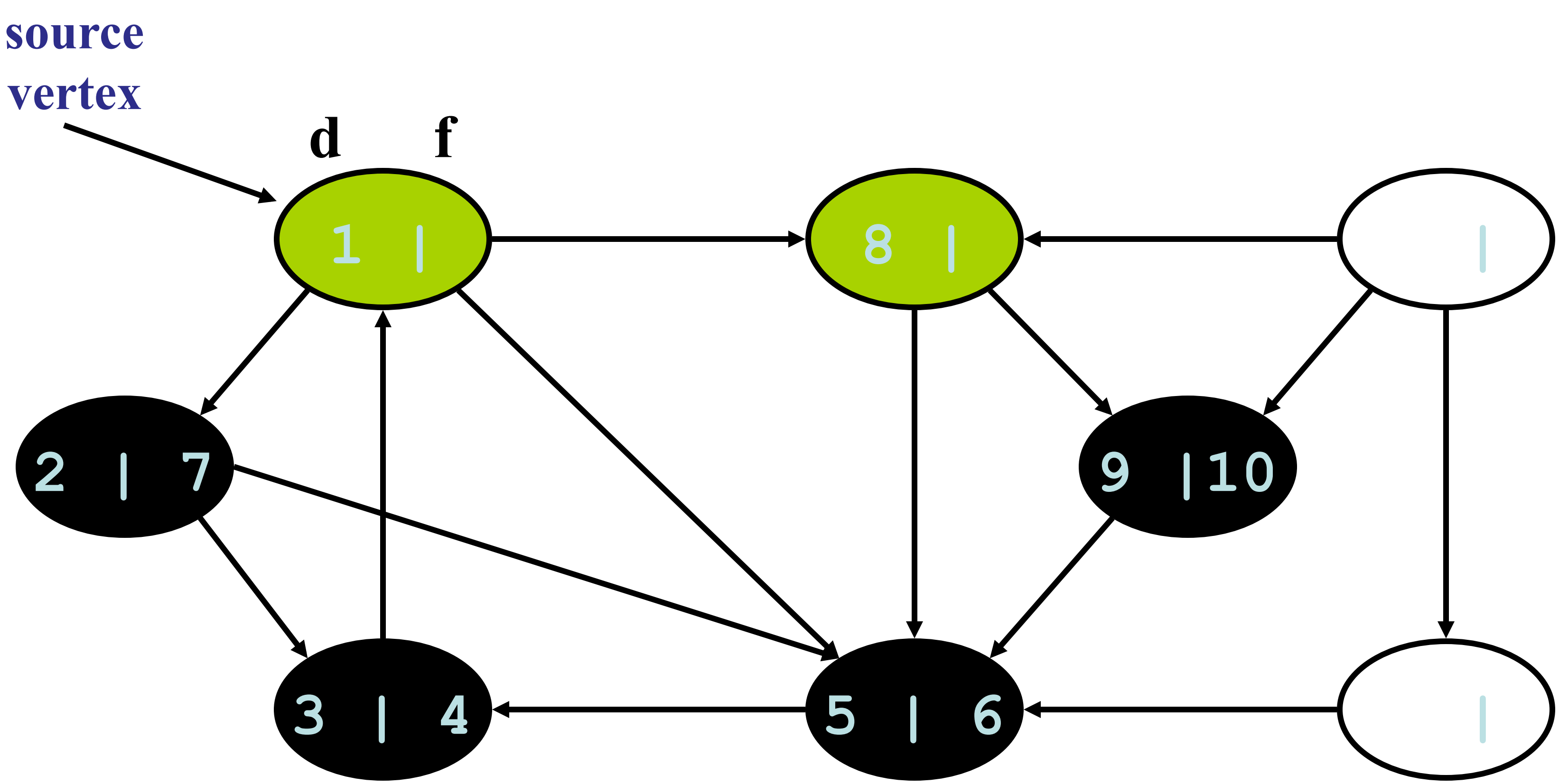


DFS Example

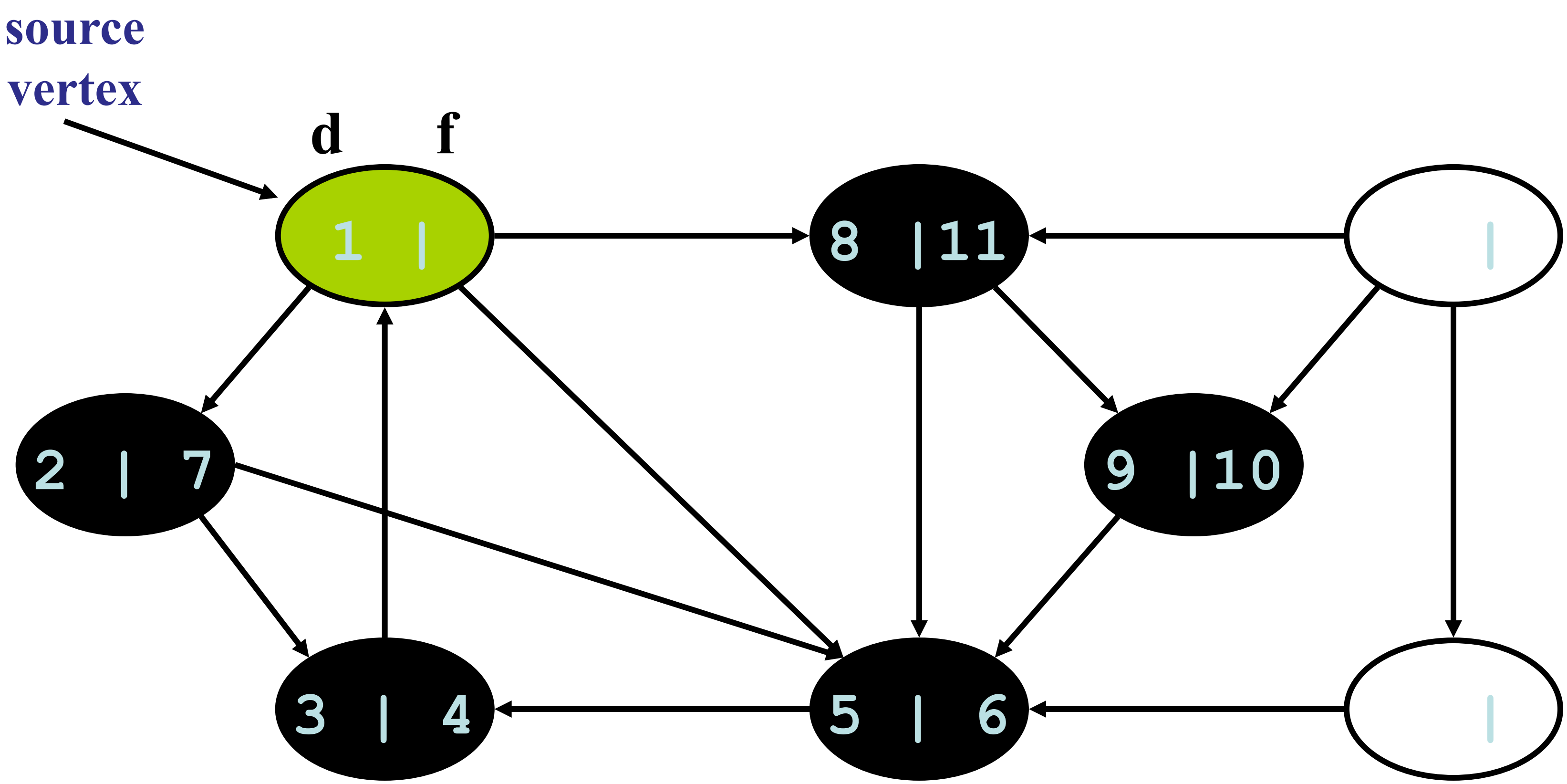


What is the structure of the green vertices?
What do they represent?

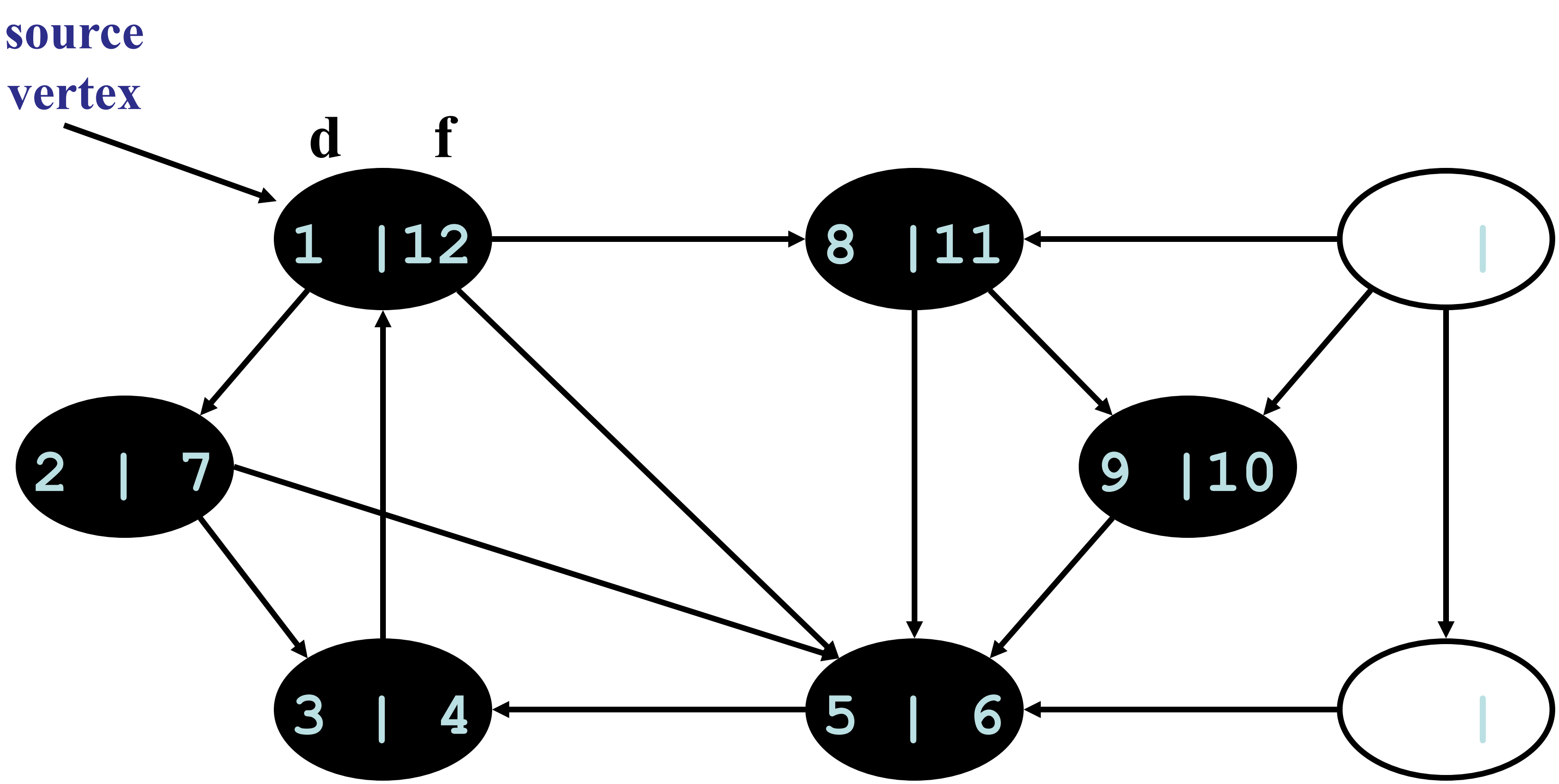
DFS Example



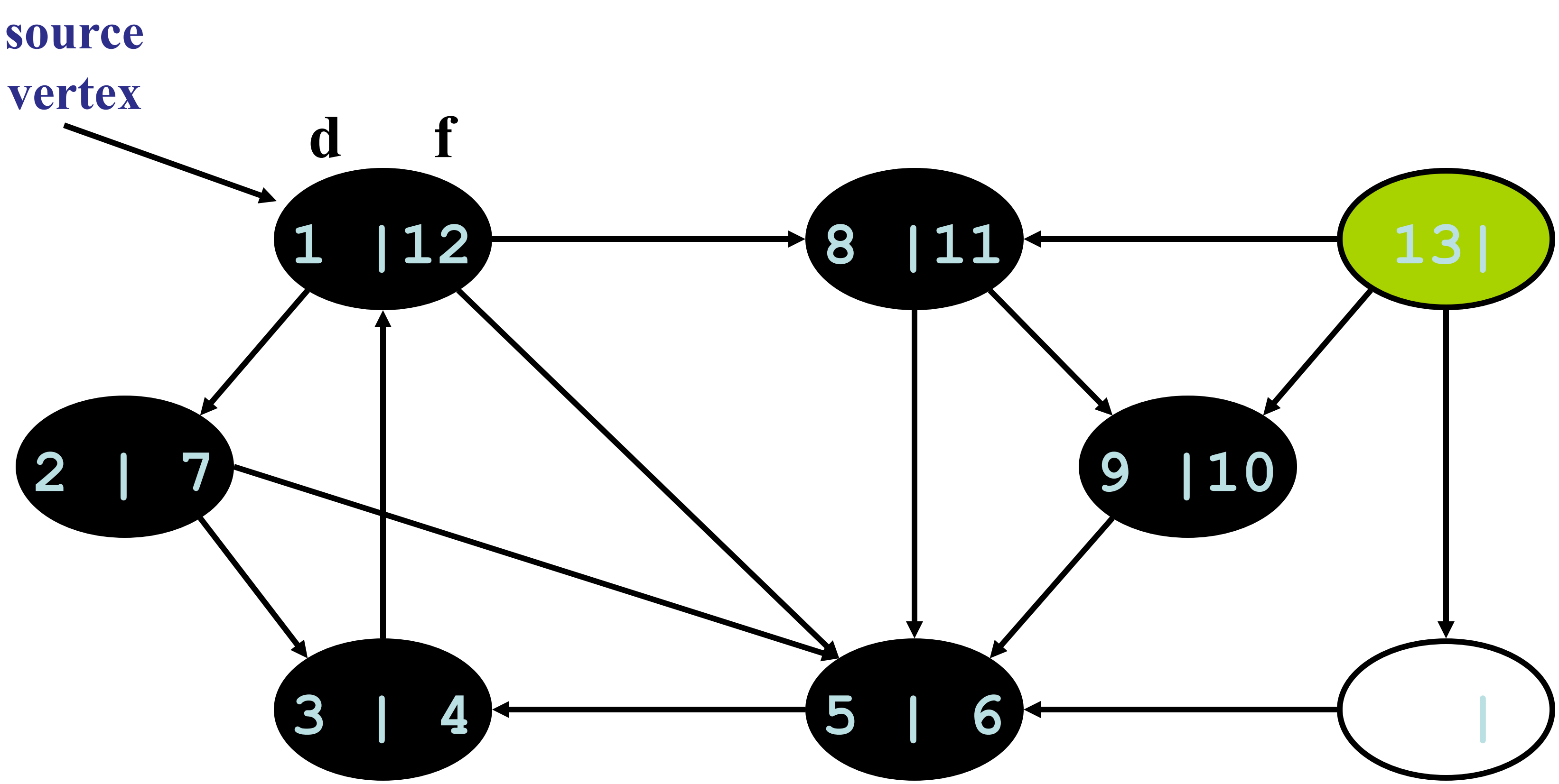
DFS Example



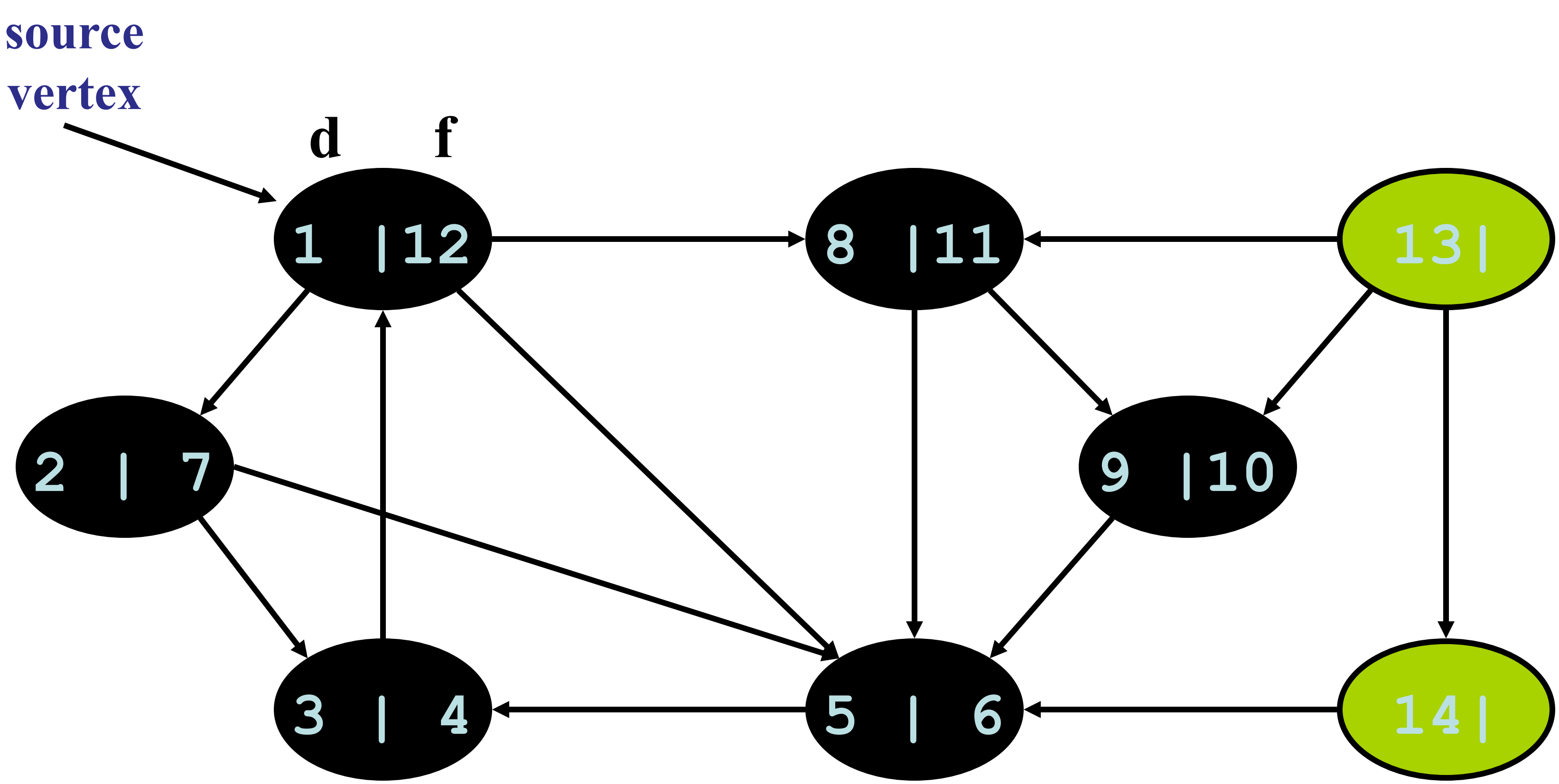
DFS Example



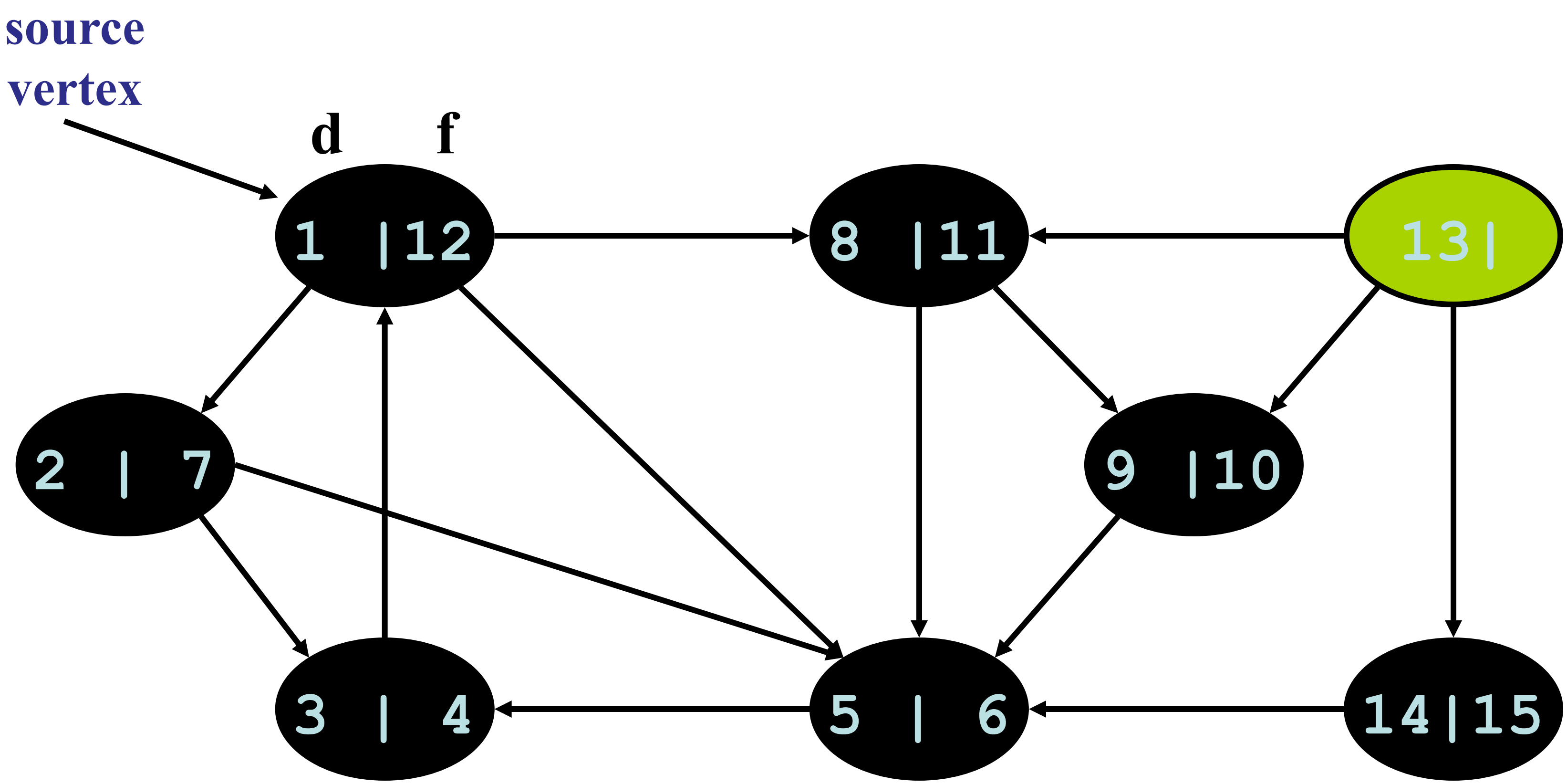
DFS Example



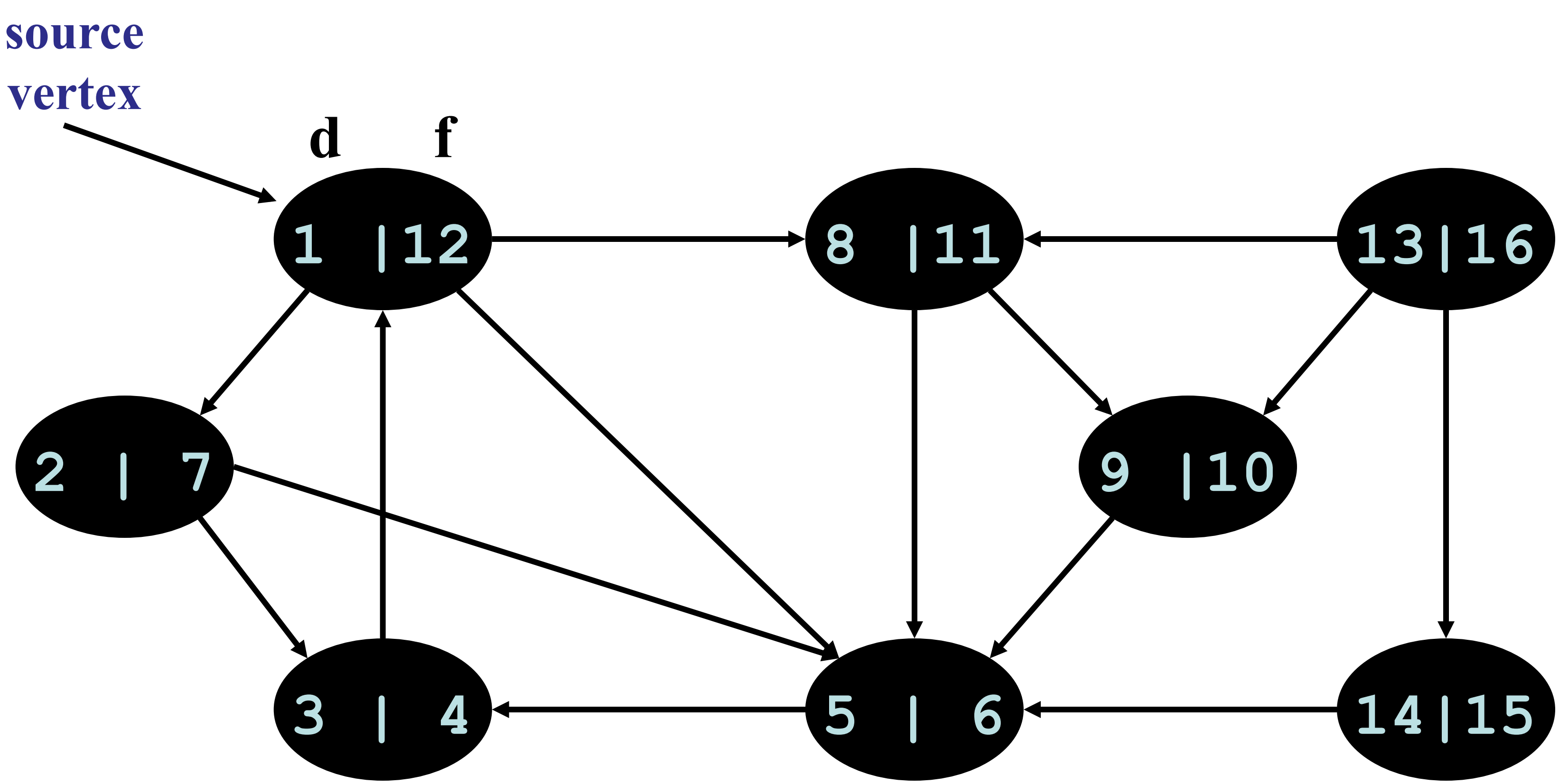
DFS Example



DFS Example



DFS Example



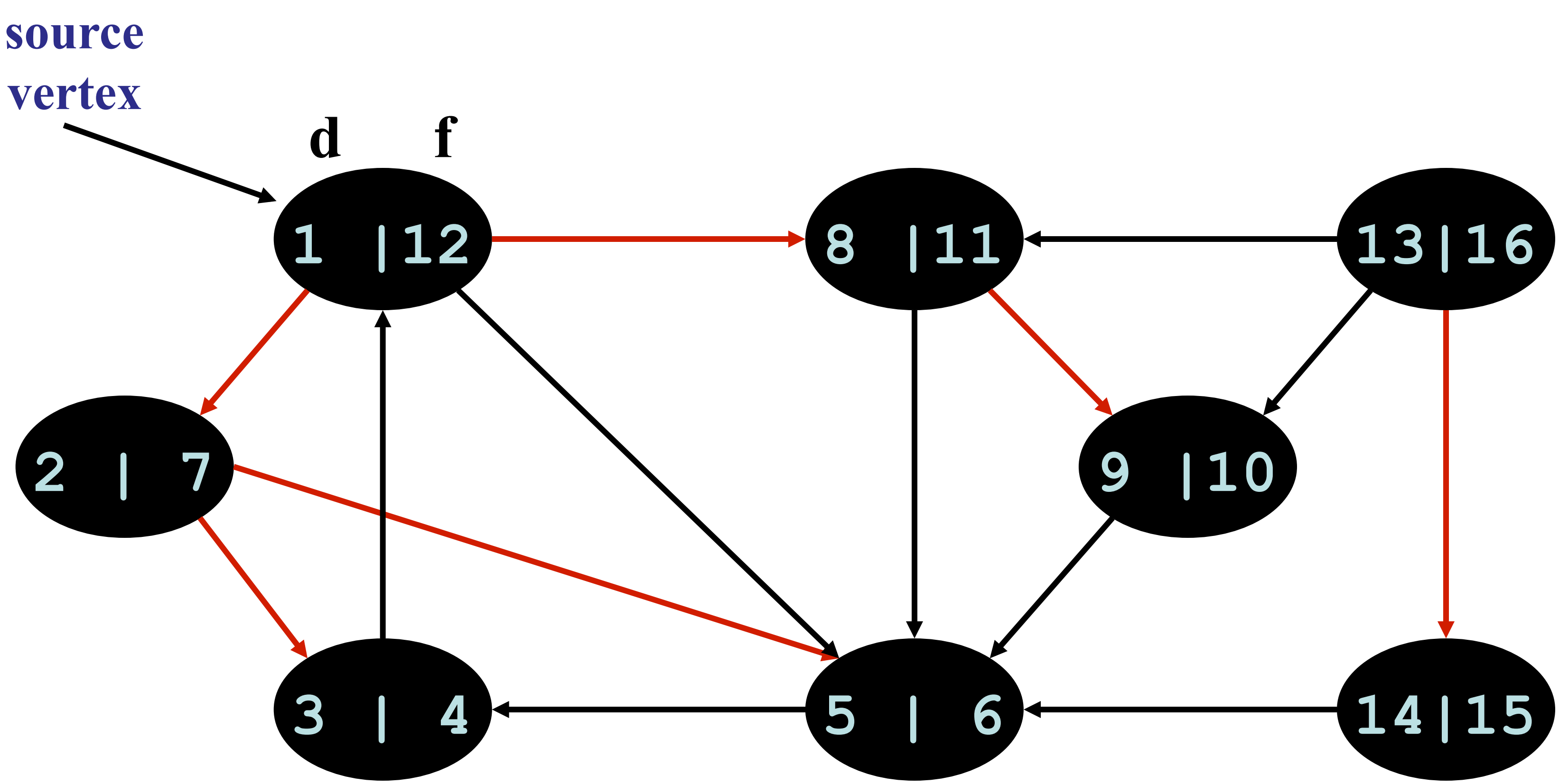
DFS: Kinds of edges

- DFS introduces an important distinction among edges in the original graph:

- *Tree edge*: encounter new (white) vertex
- The tree edges form a spanning forest

- *Can tree edges form cycles? Why or why not?*

DFS Example



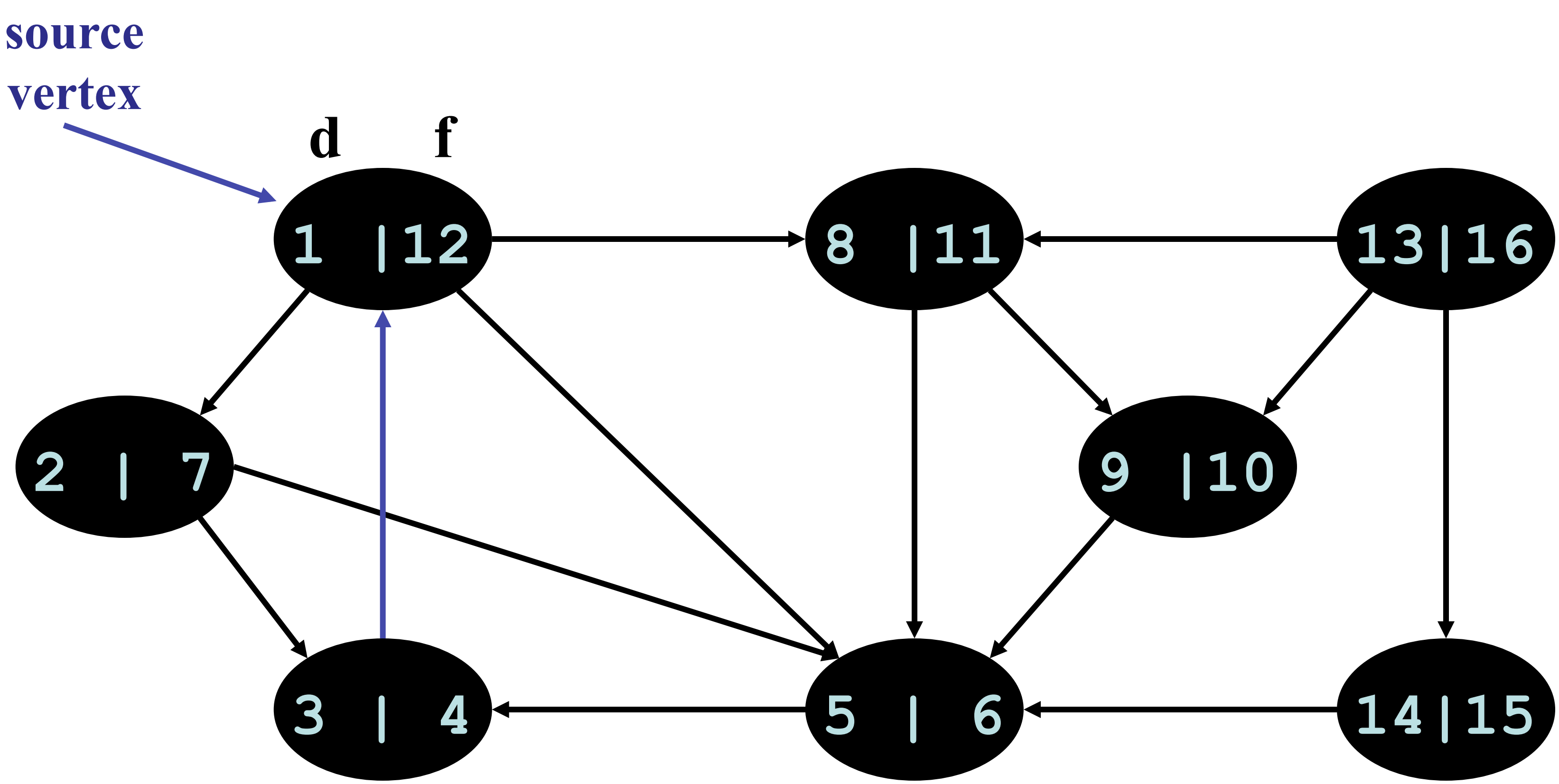
Tree edges

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
  Encounter a grey vertex (grey to grey)

DFS Example



Tree edges Back edges

DFS: Kinds of edges

- DFS introduces an important distinction among edges in the original graph:

  Tree edge: encounter new (white) vertex

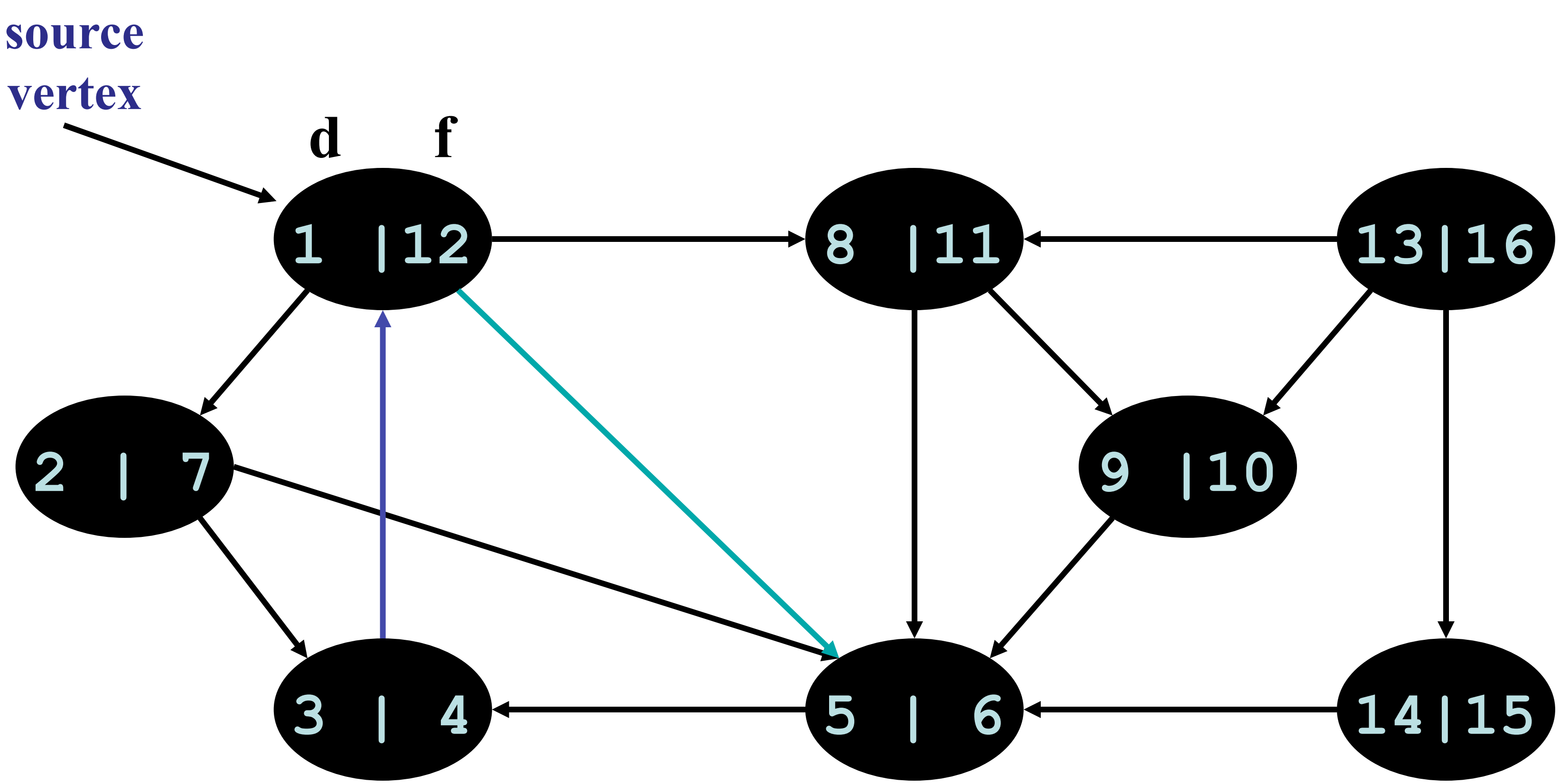
  Back edge: from descendent to ancestor

  Forward edge: from ancestor to descendent

  Not a tree edge, though

  From grey node to black node

DFS Example



Tree edges **Back edges** **Forward edges**

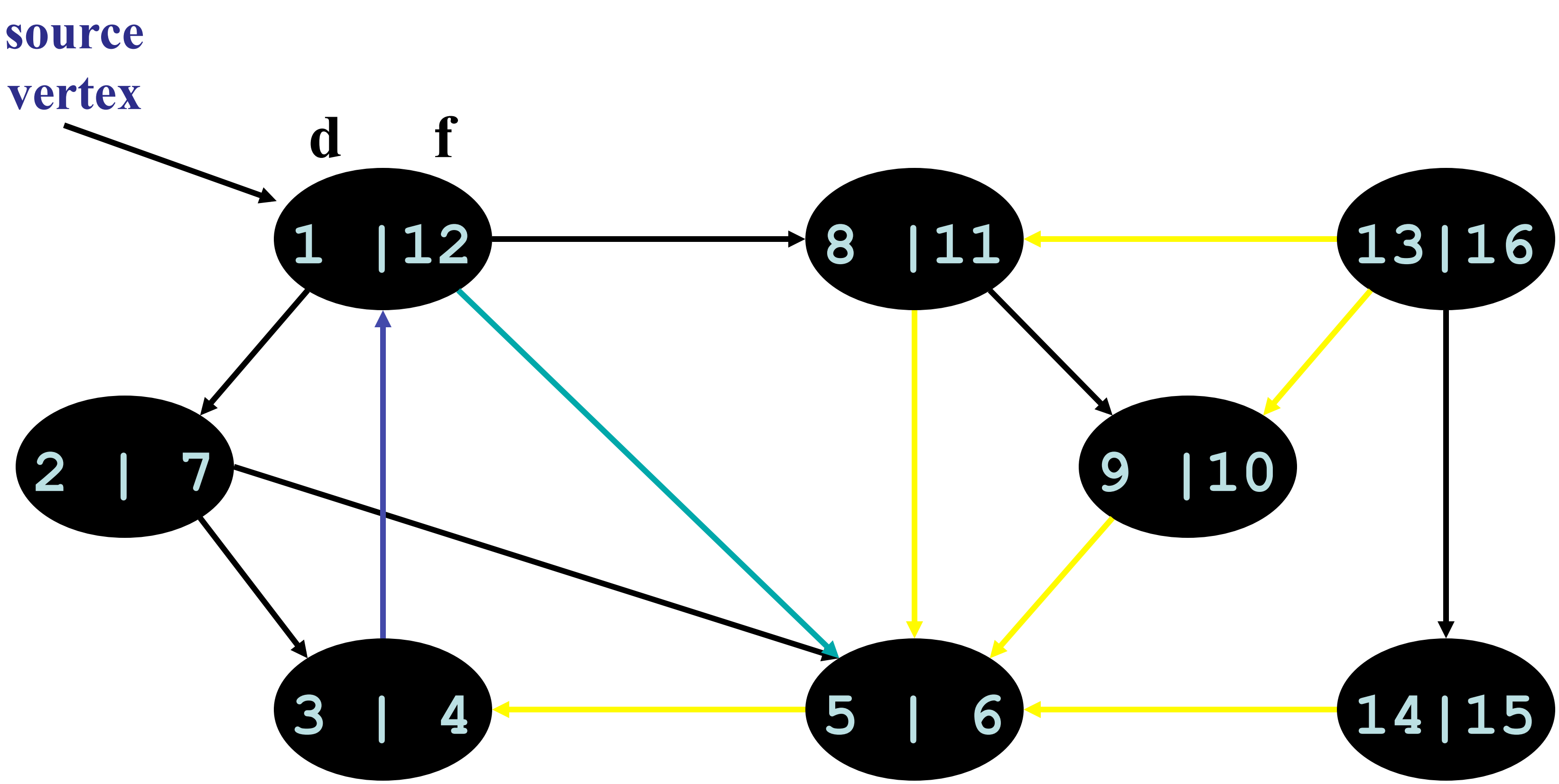
DFS: Kinds of edges

- 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

  From a grey node to a black node

DFS Example



Tree edges Back edges Forward edges Cross edges

DFS: Kinds of edges

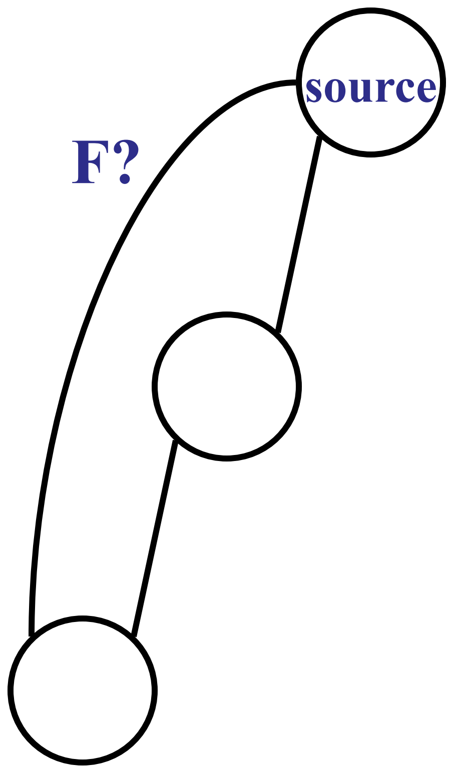
- DFS introduces an important distinction among edges in the original graph:
  Tree edge: encounter new (white) vertex
  Back edge: from descendent to ancestor
  Forward edge: from ancestor to descendent
  Cross edge: between a tree or subtrees
- Note: tree & back edges are important; most algorithms don't distinguish forward & cross

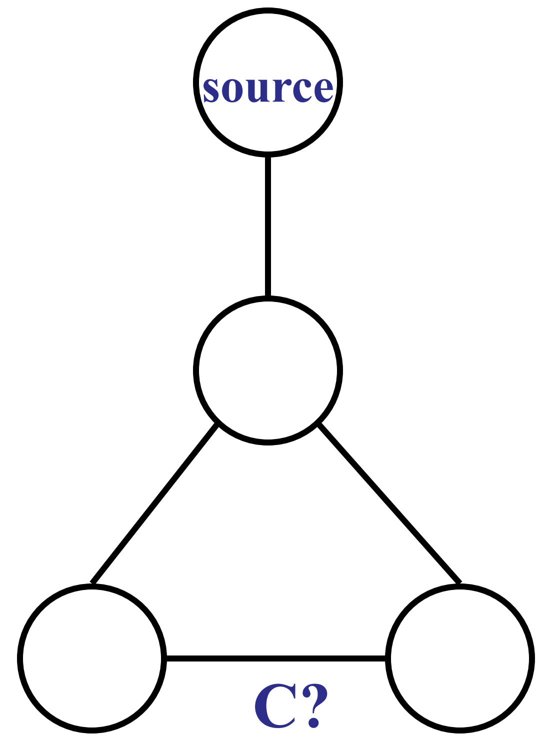
DFS: Kinds Of Edges

- Thm 23.9 (22.10 – in 3rd edition): If G is undirected, a DFS produces only tree and back edges
- Suppose you have $u.d < v.d$
- Then search discovered u before v, so first time v is discovered it is white – hence the edge *(u,v)* is a tree edge
- Otherwise the search already explored this edge in direction from *v to u*
- edge must actually be a back edge since
- *u* is still gray

DFS: Kinds Of Edges

- Thm 23.9: If G is undirected, a DFS produces only tree and back edges – cannot be a forward edge





DFS And Graph Cycles

- Thm: An undirected graph is *acyclic* iff a DFS yields no back edges
- If acyclic, no back edges (because a back edge implies a cycle
- If no back edges, acyclic

  No back edges implies only tree edges (*Why?*)

  Only tree edges implies we have a tree or a forest

  Which by definition is acyclic
- Thus, can run DFS to find whether a graph has a cycle

DFS And Cycles

- *How would you modify the code to detect cycles?*

```
DFS(G)
{
    for each vertex u ∈ G->V
    {
        u->color = WHITE;
    }
    time = 0;
    for each vertex u ∈ G->V
    {
        if (u->color == WHITE)
            DFS_Visit(u);
    }
}
```

```
DFS_Visit(u)
{
    u->color = GREY;
    time = time+1;
    u->d = time;
    for each v ∈ u->Adj[]
    {
        if (v->color == WHITE)
            DFS_Visit(v);
    }
    u->color = BLACK;
    time = time+1;
    u->f = time;
}
```

DFS And Cycles

- *What will be the running time ?*

```
DFS(G)
{
    for each vertex u ∈ G->V
    {
        u->color = WHITE;
    }
    time = 0;
    for each vertex u ∈ G->V
    {
        if (u->color == WHITE)
            DFS_Visit(u);
    }
}
```

```
DFS_Visit(u)
{
    u->color = GREY;
    time = time+1;
    u->d = time;
    for each v ∈ u->Adj[]
    {
        if (v->color == WHITE)
            DFS_Visit(v);
    }
    u->color = BLACK;
    time = time+1;
    u->f = time;
}
```

DFS And Cycles

- *What will be the running time?*
- A: O(V+E)
- We can actually determine if cycles exist in O(V) time:

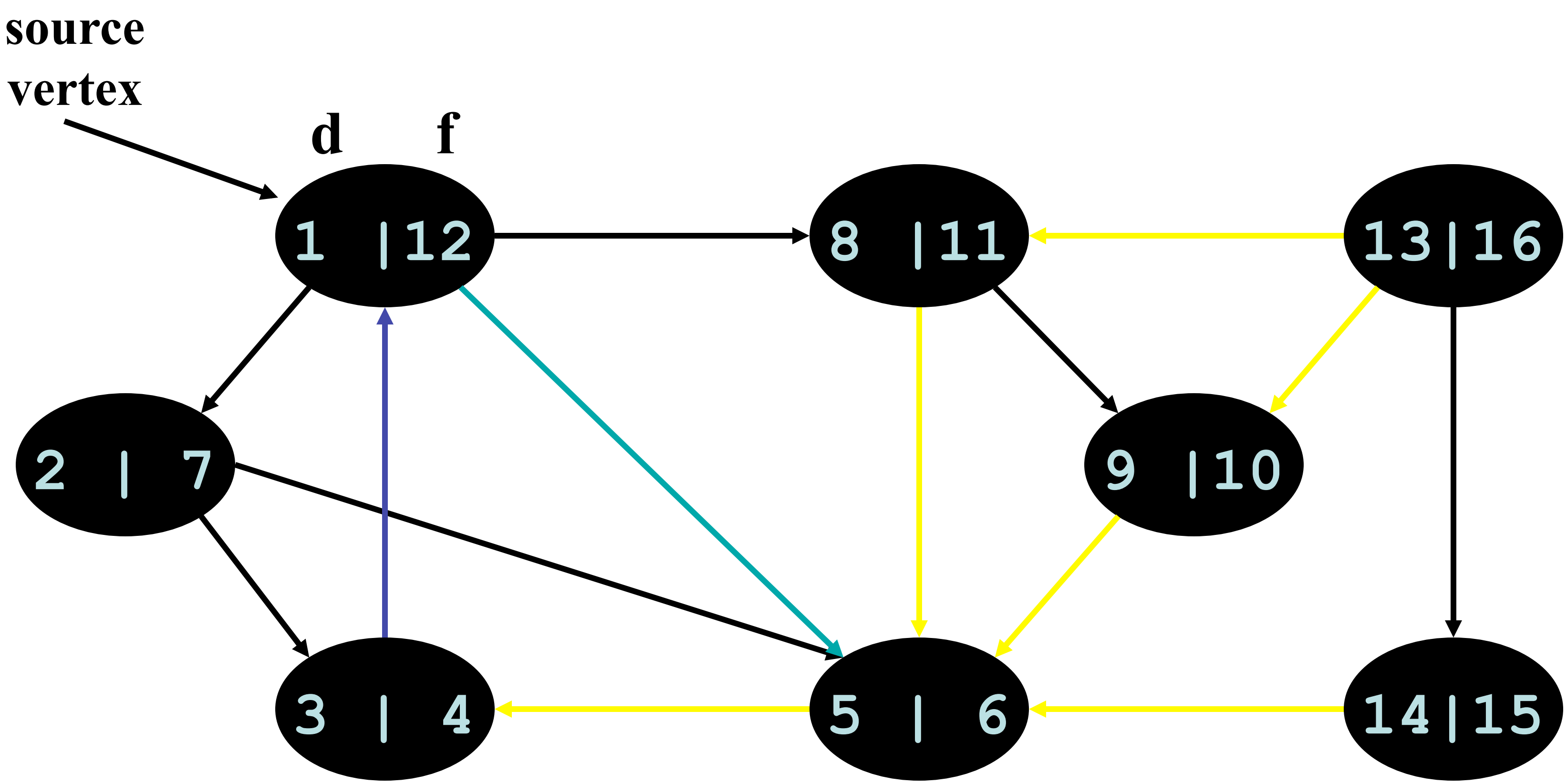
  In an undirected acyclic forest, $|E| \leq |V| - 1$

  So count the edges: if ever see $|V|$ distinct edges, must have seen a back edge along the way

Review: Kinds Of Edges

- Thm: If G is undirected, a DFS produces only tree and back edges
- Thm: An undirected graph is *acyclic* iff a DFS yields no back edges
- Thus, can run DFS to find cycles

Review: Kinds of Edges



Tree edges Back edges Forward edges Cross edges