

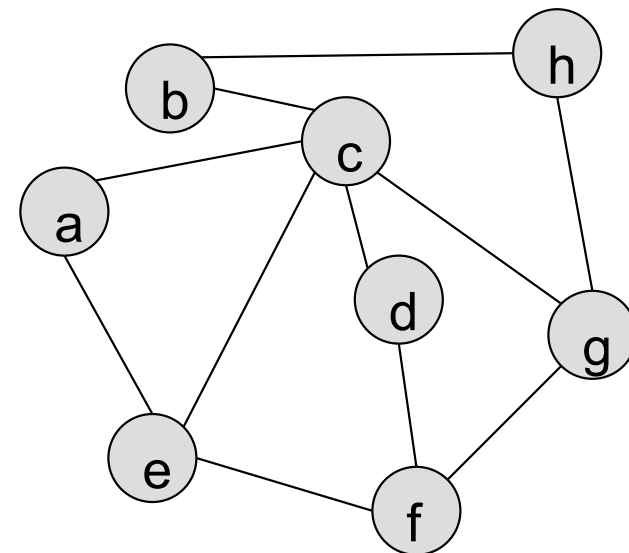
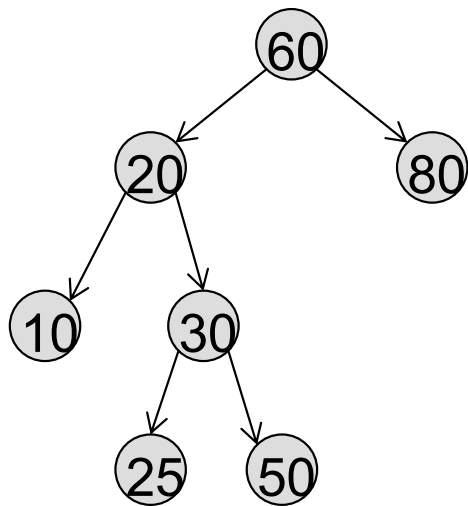
# CSCI 104

# Recursive Graph & Tree Traversals Algorithms

CSCI 104 Teaching Team  
Reviewed for Spring 2025

# Traversal Algorithms

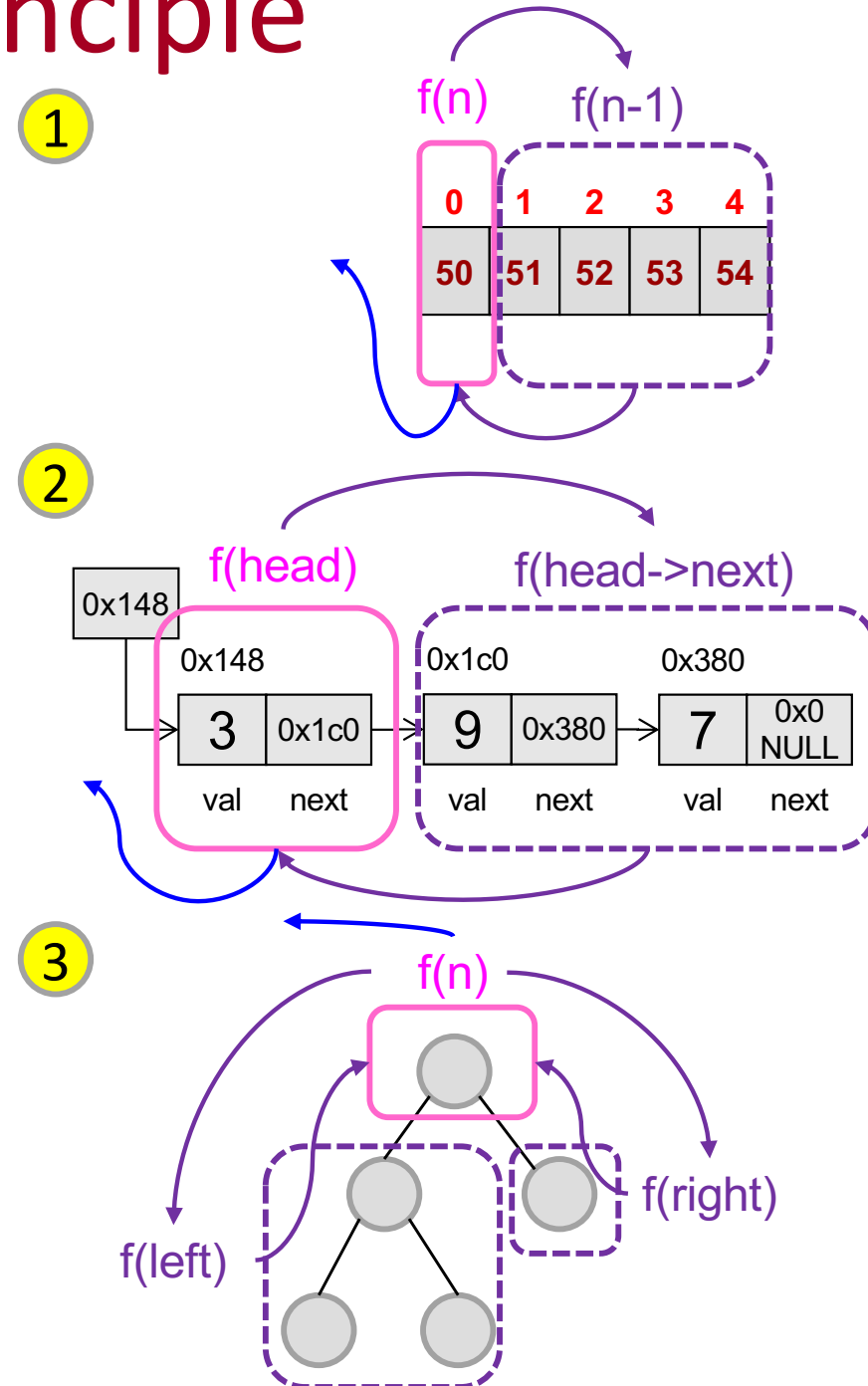
- Traversals should visit (and potentially apply some operation or processing to) each node once



# RECURSIVE TREE TRAVERSALS

# Guiding Recursive Principle

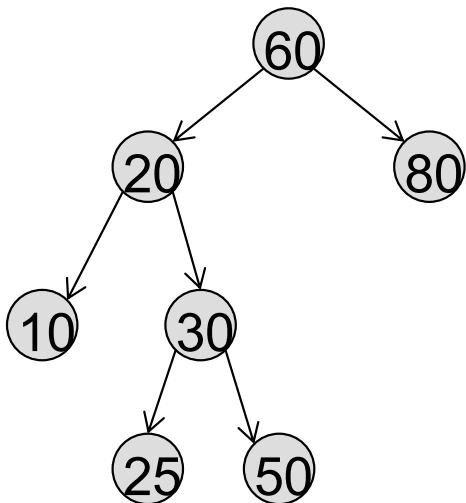
- A useful principle when trying to develop recursive solutions is that the recursive code **should handle only 1 element**, which might be:
  1. An element in an array
  2. A node a linked list
  3. A node in a tree
  4. One choice in a sequence of choices
- Then use recursion to handle the remaining elements
- And finally **combine the solution(s)** to the recursive call(s) with the **one element being handled**



# Recursive Tree Traversals

- A traversal iterates over all nodes of the tree
  - Usually using a depth-first, recursive approach
- Three general traversal orderings
  - Pre-order [Process root then visit subtrees]
  - In-order [Visit left subtree, process root, visit right subtree]
  - Post-order [Visit left subtree, visit right subtree, process root]

```
// Node definition
struct TNode
{
    int val;
    TNode *left, *right;
};
```



```
Preorder(TNode* t)
{
    if t == NULL return
    process(t) // print val.
    Preorder(t->left)
    Preorder(t->right)
}
```

60 20 10 30 25 50 80

```
Inorder(TNode* t)
{
    if t == NULL return
    Inorder(t->left)
    process(t) // print val.
    Inorder(t->right)
}
```

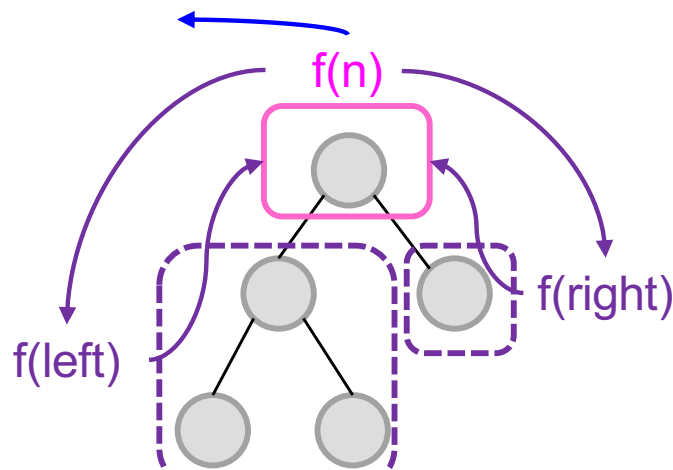
10 20 25 30 50 60 80

```
Postorder(TNode* t)
{
    if t == NULL return
    Postorder(t->left)
    Postorder(t->right)
    process(t) // print val.
}
```

10 25 50 30 20 80 60

# Example 1: Count Nodes

- Write a recursive function to **count how many nodes** are in the binary tree
  - Only process 1 node at a time
  - Determine pre-, in-, or post-order based on whose answers you need to compute the result for your node
  - For in- or post-order traversals, determine how to use/combine results from recursion on children



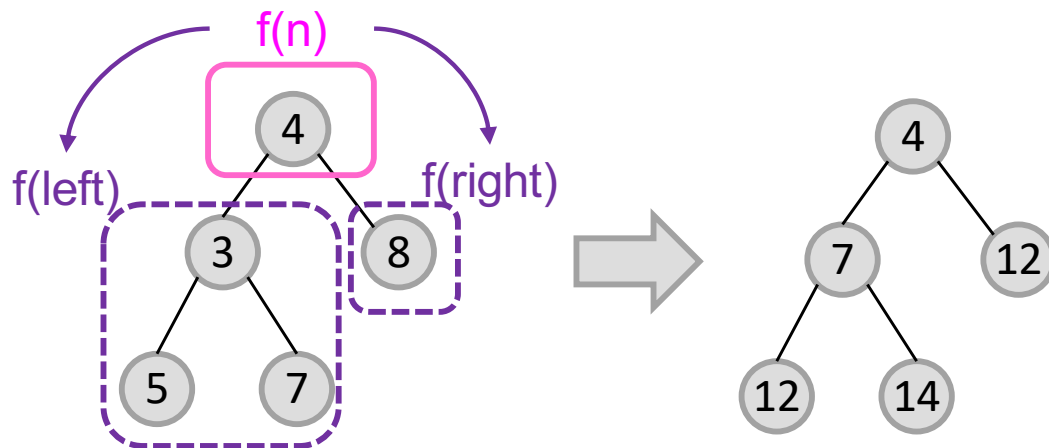
```
// Node definition
struct Tnode {
    int val;
    TNode *left, *right;
};

int count(TNode* root)
{
    if( root == NULL ) _____;
    else {

    } }
}
```

# Example 2: Prefix Sums

- Write a recursive function to **have each node store the sum of the values on the path from the root to each node.**
  - Only process 1 node at a time
  - Determine pre-, in-, or post-order based on whose answers you need to compute the result for your node



```
void prefixH(TNode* root, int psum)

void prefix(TNode* root)
{
    prefixH(root, 0);
}

void prefixH(TNode* root, int psum)
{
    if( root == NULL ) _____;
    else {

    }
}
```

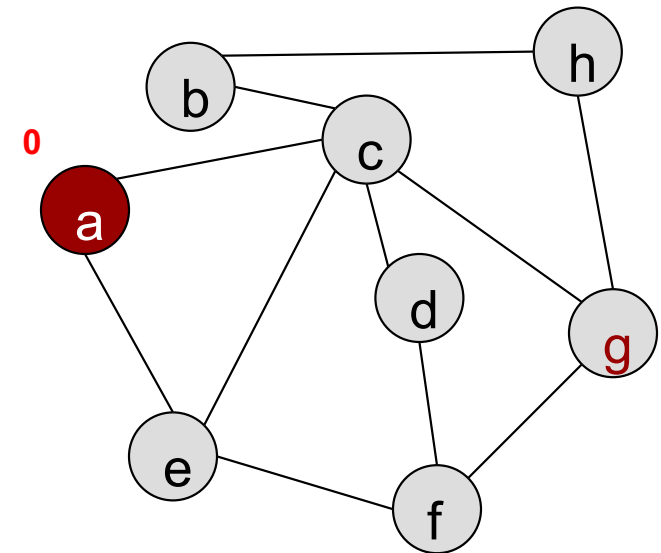
# GENERAL GRAPH TRAVERSALS



# BREADTH-FIRST SEARCH

# Breadth-First Search

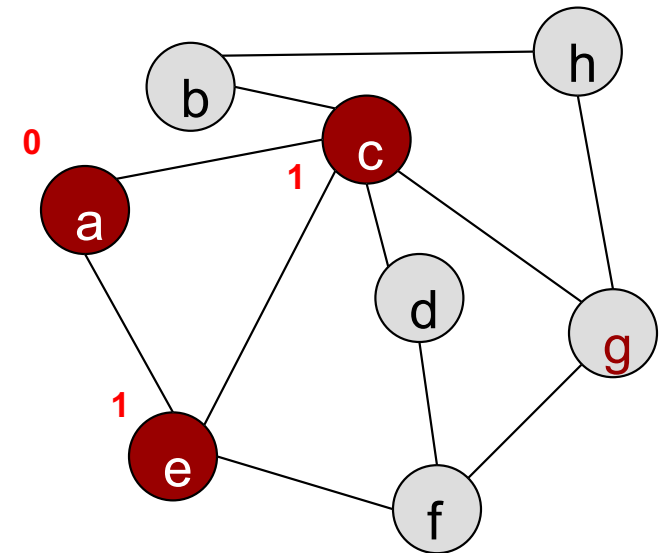
- Given a graph with vertices,  $V$ , and edges,  $E$ , and a starting vertex that we'll call  $u$
- BFS starts at  $u$  ('a' in the diagram to the left) and fans-out along the edges to nearest neighbors, then to their neighbors and so on
- Goal: Find shortest paths (a.k.a. minimum number of hops or depth) from the start vertex to every other vertex



Depth 0: a

# Breadth-First Search

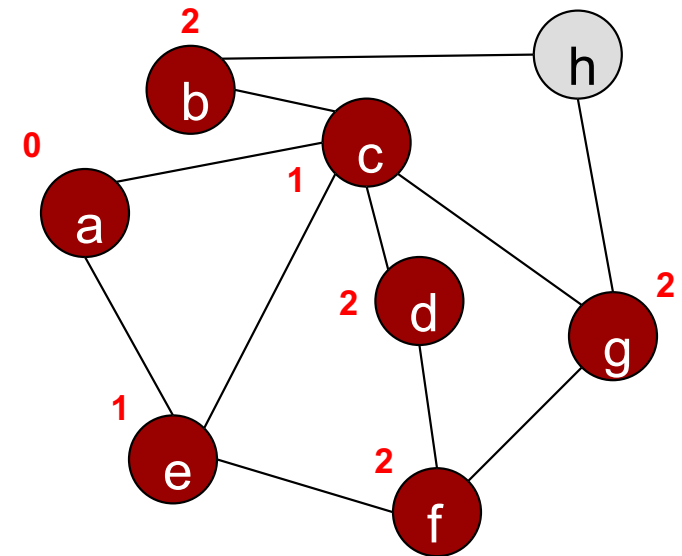
- Given a graph with vertices,  $V$ , and edges,  $E$ , and a starting vertex,  $u$
- BFS starts at  $u$  ('a' in the diagram to the left) and fans-out along the edges to nearest neighbors, then to their neighbors and so on
- Goal: Find shortest paths (a.k.a. minimum number of hops or depth) from the start vertex to every other vertex



Depth 0: a  
Depth 1: c,e

# Breadth-First Search

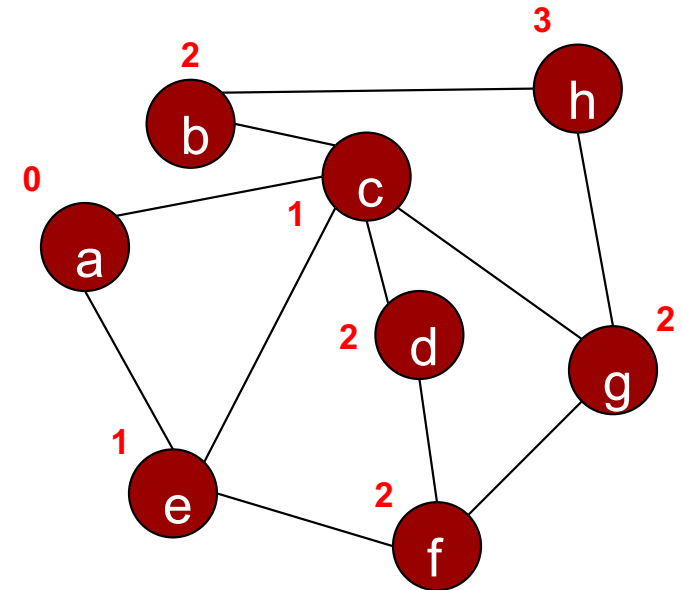
- Given a graph with vertices,  $V$ , and edges,  $E$ , and a starting vertex,  $u$
- BFS starts at  $u$  ('a' in the diagram to the left) and fans-out along the edges to nearest neighbors, then to their neighbors and so on
- Goal: Find shortest paths (a.k.a. minimum number of hops or depth) from the start vertex to every other vertex



**Depth 0: a**  
**Depth 1: c,e**  
**Depth 2: b,d,f,g**

# Breadth-First Search

- Given a graph with vertices,  $V$ , and edges,  $E$ , and a starting vertex,  $u$
- BFS starts at  $u$  ('a' in the diagram to the left) and fans-out along the edges to nearest neighbors, then to their neighbors and so on
- Goal: Find shortest paths (a.k.a. minimum number of hops or depth) from the start vertex to every other vertex



**Depth 0: a**  
**Depth 1: c,e**  
**Depth 2: b,d,f,g**  
**Depth 3: h**

# Developing the Algorithm

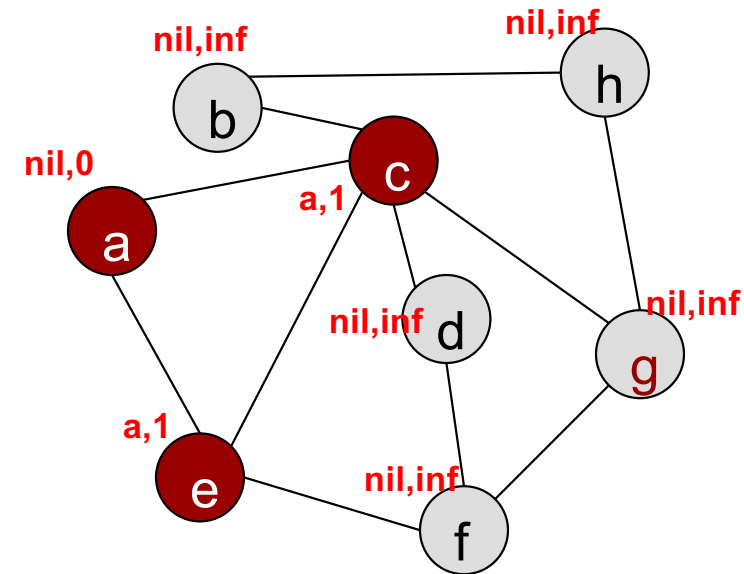
- Exploring all vertices in the order they are found implies we will explore vertices in First-In/First-Out order which implies use of a **Queue**
  - Important: BFS implies use of a **queue**
  - Put newly found vertices in the back and pull out a vertex from the front to explore next
- We don't want to put a vertex in the queue more than once...
  - "mark" a vertex the first time we encounter it (only allowing unmarked vertices to be put in the queue)
  - We can "mark" a vertex by **adding them to a set** OR by simply **setting some data member that indicates we've seen this vertex before**
- May also keep a "predecessor" structure or value per vertex that indicates which prior vertex found this vertex
  - Allows us to find a shortest-path back to the start vertex (i.e. retrace our steps)

# Breadth-First Search

## Algorithm:

```

BFS(G,u)
1 for each vertex v
2   pred[v] = nil, d[v] = inf.
3 Q = new Queue
4 Q.enqueue(u), d[u]=0
5 while Q is not empty
6   v = Q.front(); Q.dequeue()
7   foreach neighbor, w, of v:
8     if pred[w] == nil // w not found
9       Q.enqueue(w)
10      pred[w] = v, d[w] = d[v] + 1
    
```



v = a

Q:

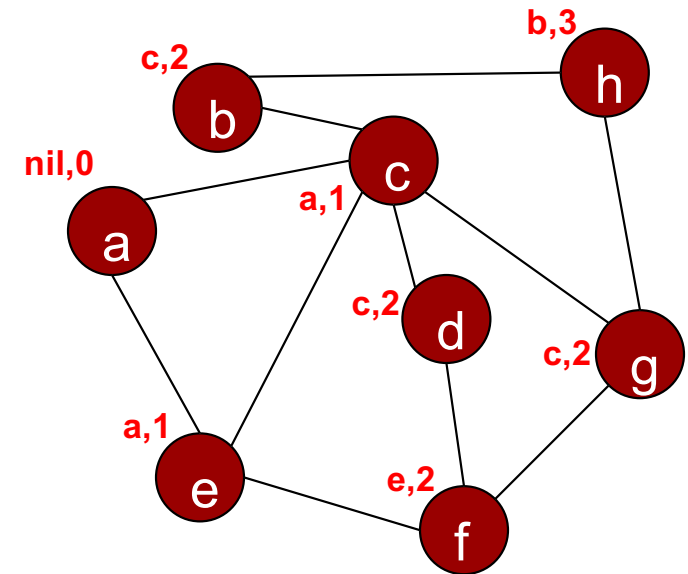


# Breadth-First Search

## Algorithm:

```

BFS(G,u)
1 for each vertex v
2   pred[v] = nil, d[v] = inf.
3 Q = new Queue
4 Q.enqueue(u), d[u]=0
5 while Q is not empty
6   v = Q.front(); Q.dequeue()
7   foreach neighbor, w, of v:
8     if pred[w] == nil // w not found
9       Q.enqueue(w)
10      pred[w] = v, d[w] = d[v] + 1
    
```



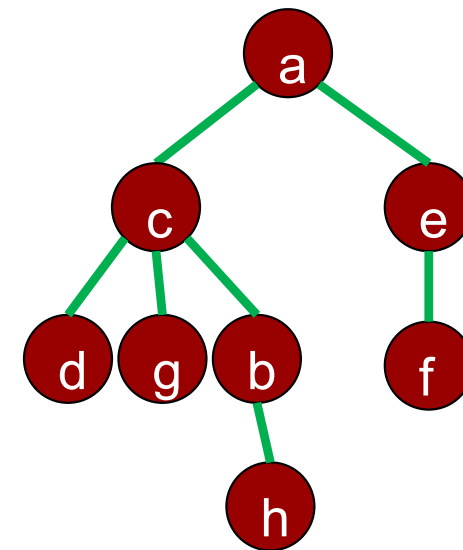
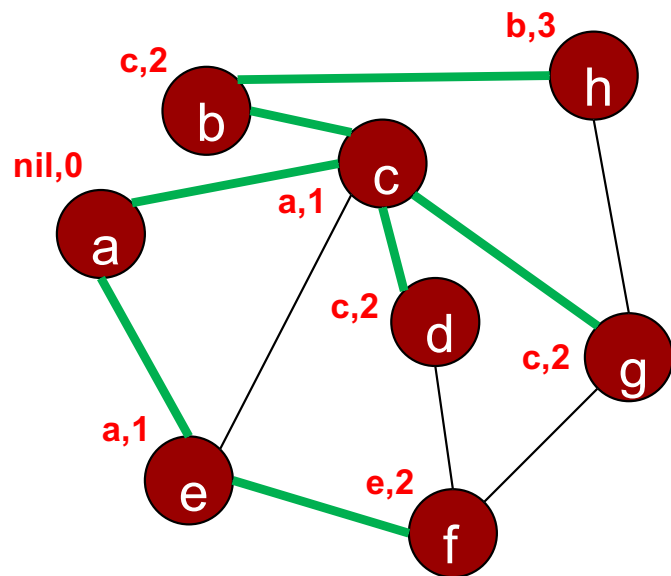
v = h

Q:



# Breadth-First Search Trees

- BFS visits each node once and will induce a tree subgraph (as will DFS) from the original graph
  - BFS is tree of shortest paths from the source to all other vertices (in connected component)



Original graph, G

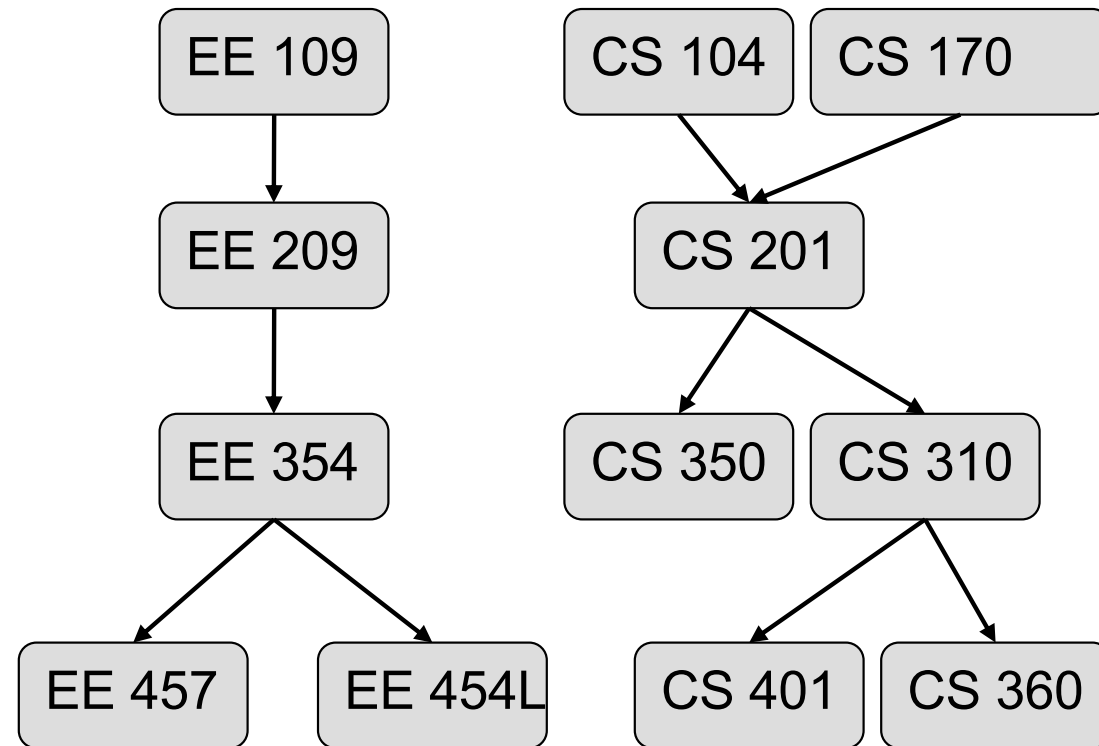
BFS Induced Tree

Topological Search

# DEPTH FIRST SEARCH MOTIVATING EXAMPLE

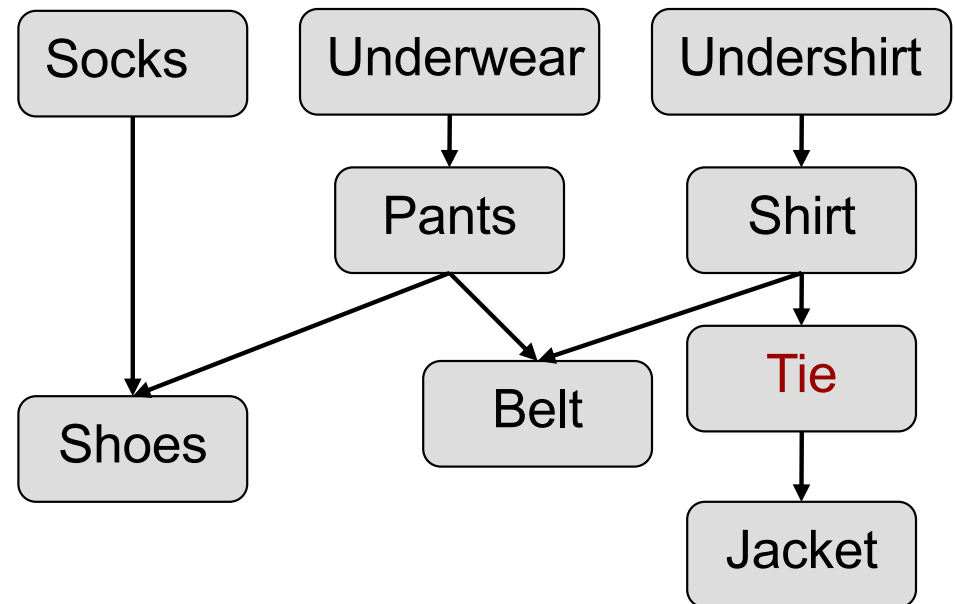
# DFS Application: Topological Sort

- Breadth-first search doesn't solve all our problems.
- Given a graph of dependencies (tasks, prerequisites, etc.) **topological** sort creates a consistent ordering of tasks (vertices) where no dependencies are violated
- Many possible valid topological orderings exist
  - EE 109, EE 209, EE 354, EE 454, EE 457, CS104, PHYS 152, CS 201,...
  - CS 104, EE 109, CS 170, EE 209,...



# Topological Sort

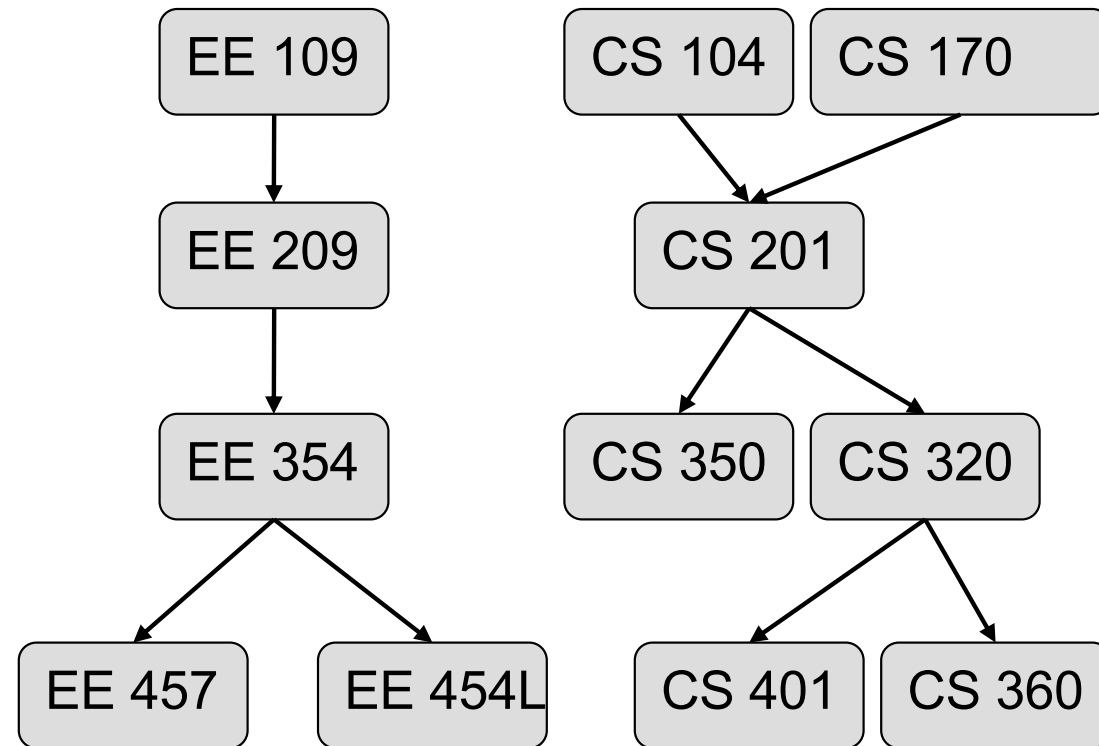
- Another example
  - Getting dressed
- More Examples:
  - Project management scheduling
  - Build order in a Makefile or other compile project
  - Cooking using a recipe
  - Instruction execution on an out-of-order pipelined CPU
  - Production of output values in a simulation of a combinational gate network



<http://www.personal.kent.edu/~rmuhamma/Algorithms/MyAlgorithms/GraphAlgor/topoSort.htm>

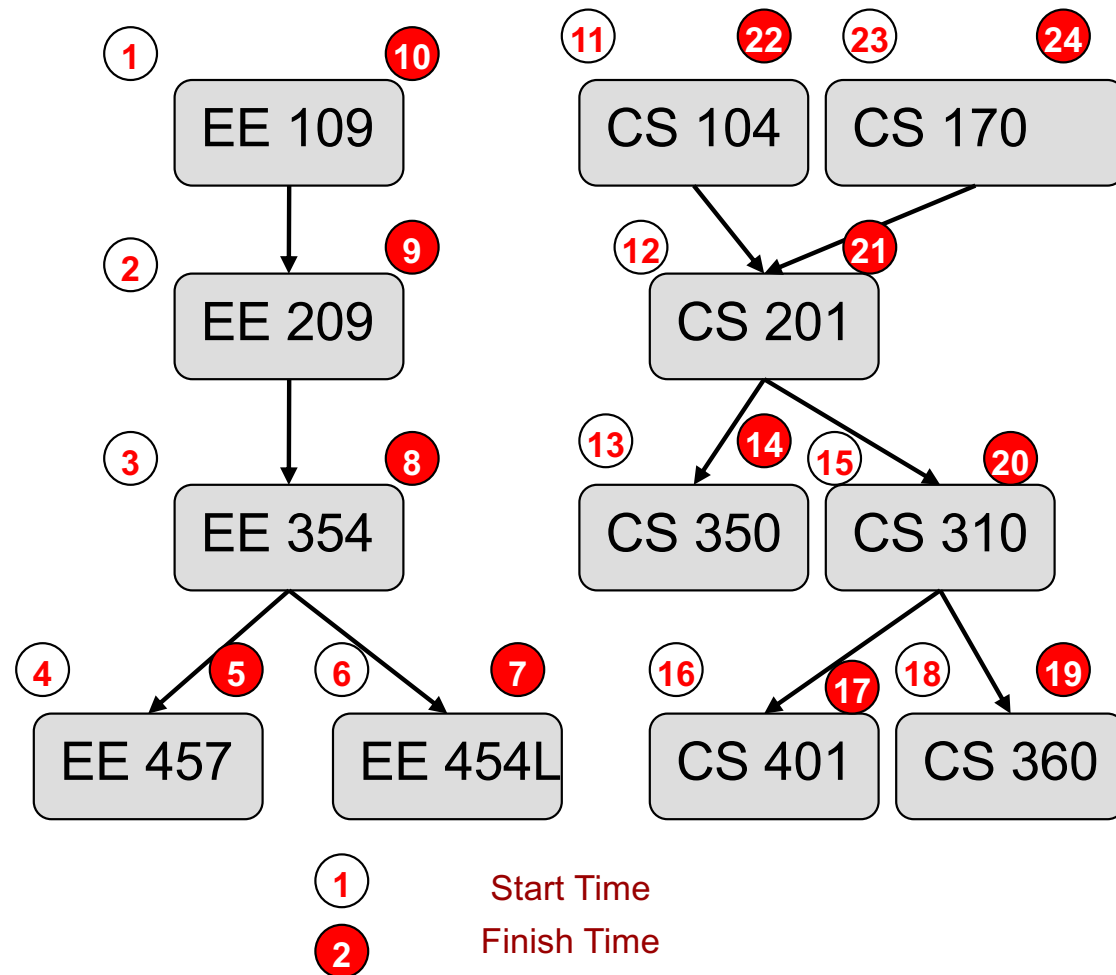
# Topological Sort

- Does breadth-first search work?
  - No. What if we started at CS 170...
  - We'd go to CS 201L before CS 104
- All parent nodes need to be completed before any child node
- BFS only guarantees *some* parent has completed before child
- Turns out a Depth-First Search will be part of our solution



# Depth First Search

- Explores ALL children before completing a parent
  - Note: BFS completes a parent before ANY children
- For DFS let us assign:
  - A start time when the node is first found
  - A finish time when a node is completed
- If we look at our nodes in reverse order of finish time (i.e. last one to finish back to first one to finish) we arrive at a...
  - Topological ordering!!!



Reverse Finish Time Order

CS 170, CS 104, CS 201, CS 310, CS 360, CS 401, CS 350,  
 EE 109, EE 209L, EE 354, EE 454L, EE 457

# DFS Algorithm

- DFS visits and completes all children before completing (and going on to a sibling)
- Process:
  - Visit a node
  - Mark as visited (started)
  - For each visited neighbor, visit it and perform DFS on all of their children
  - Only then, mark as finished
- Let's trace recursive DFS!!
- If cycles in the graph, ensure we don't get caught visiting neighbors endlessly
  - Use some status (textbooks use "colors" but really just some integer)
  - White = unvisited,
  - Gray = visited but not finished
  - Black = finished

## DFS-All (G)

```
1 for each vertex u
2   u.color = WHITE
3 finish_list = empty_list
4 for each vertex u do
5   if u.color == WHITE then
6     DFS-Visit (G, u, finish_list)
7 return finish_list
```

## DFS-Visit (G, u, finish\_list)

```
1   u.color = GRAY
2   for each vertex v in Adj(u) do
3     if v.color = WHITE then
4       DFS-Visit (G, v)
5   u.color = BLACK
6   finish_list.append(u)
```

Source: "Introduction to Algorithms",  
Cormen, Leiserson, Rivest

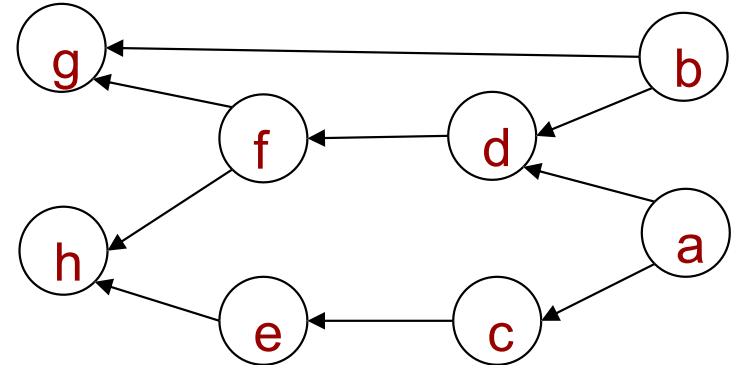
# Depth First-Search

## DFS-All (G)

```
1 for each vertex u
2   u.color = WHITE
3 finish_list = empty_list
4 for each vertex u do
5   if u.color == WHITE then
6     DFS-Visit (G, u, finish_list)
7 return finish_list
```

## DFS-Visit (G, u, l)

```
1   u.color = GRAY
2   for each vertex v in Adj(u) do
3     if v.color = WHITE then
4       DFS-Visit (G, v)
5   u.color = BLACK
6   l.append(u)
```





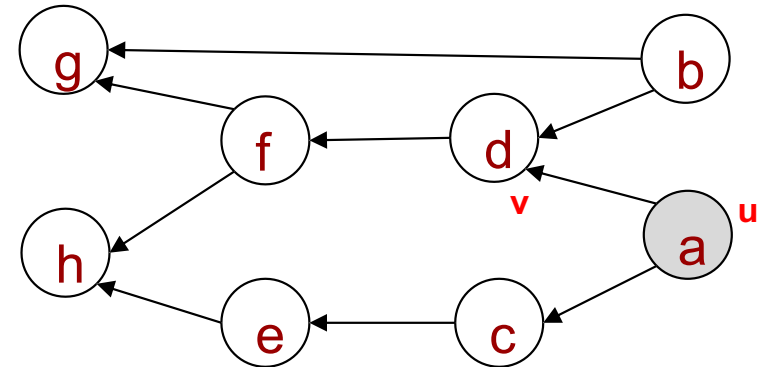
# Depth First-Search

## DFS-All (G)

- 1 for each vertex  $u$
- 2  $u.color = WHITE$
- 3  $finish\_list = empty\_list$
- 4 for each vertex  $u$  do
- 5   if  $u.color == WHITE$  then
- 6     DFS-Visit (G,  $u$ ,  $finish\_list$ )
- 7 return  $finish\_list$

## DFS-Visit (G, $u$ , $l$ )

- 1  $u.color = GRAY$
- 2 for each vertex  $v$  in  $Adj(u)$  do
- 3   if  $v.color = WHITE$  then
- 4     DFS-Visit (G,  $v$ )
- 5  $u.color = BLACK$
- 6  $l.append(u)$



**DFS-Visit(G,a,l):**

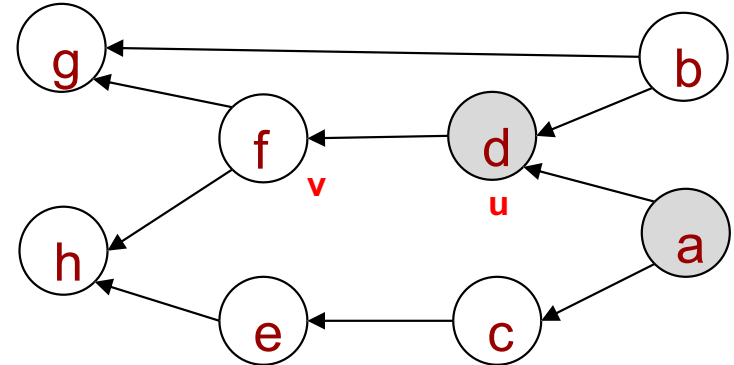
# Depth First-Search

## DFS-All (G)

- 1 for each vertex  $u$
- 2  $u.color = WHITE$
- 3  $finish\_list = empty\_list$
- 4 for each vertex  $u$  do
- 5 if  $u.color == WHITE$  then
- 6     DFS-Visit (G,  $u$ ,  $finish\_list$ )
- 7 return  $finish\_list$

## DFS-Visit (G, $u$ , $l$ )

- 1  $u.color = GRAY$
- 2 for each vertex  $v$  in  $Adj(u)$  do
- 3 if  $v.color = WHITE$  then
- 4     DFS-Visit (G,  $v$ )
- 5  $u.color = BLACK$
- 6  $l.append(u)$



**DFS-Visit(G,d,l):**

**DFS-Visit(G,a,l):**

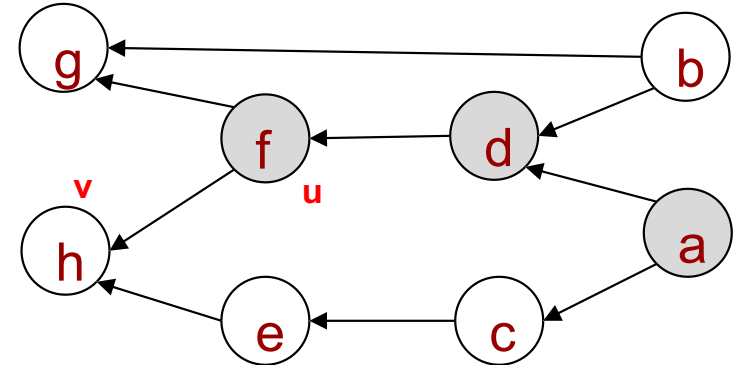
# Depth First-Search

## DFS-All (G)

- 1 for each vertex u
- 2   u.color = WHITE
- 3 finish\_list = empty\_list
- 4 for each vertex u do
- 5   if u.color == WHITE then
- 6     DFS-Visit (G, u, finish\_list)
- 7 return finish\_list

## DFS-Visit (G, u, l)

- 1   u.color = GRAY
- 2   for each vertex v in Adj(u) do
- 3     if v.color = WHITE then
- 4       DFS-Visit (G, v)
- 5   u.color = BLACK
- 6   l.append(u)



**DFS-Visit(G,f,l):**

**DFS-Visit(G,d,l):**

**DFS-Visit(G,a,l):**

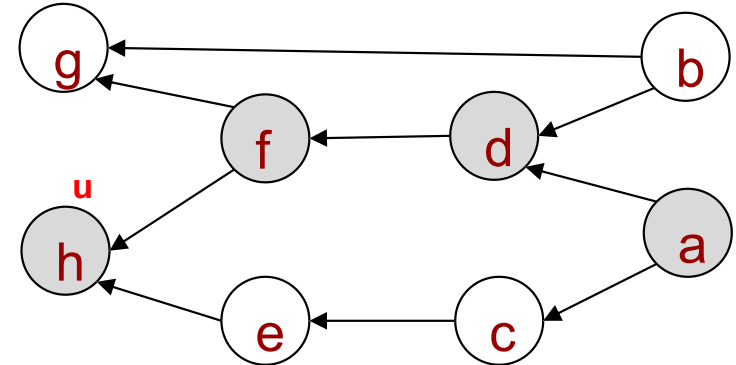
# Depth First-Search

## DFS-All (G)

- 1 for each vertex u
- 2   u.color = WHITE
- 3 finish\_list = empty\_list
- 4 for each vertex u do
- 5   if u.color == WHITE then
- 6     DFS-Visit (G, u, finish\_list)
- 7 return finish\_list

## DFS-Visit (G, u, l)

- 1   u.color = GRAY
- 2   for each vertex v in Adj(u) do
- 3     if v.color = WHITE then
- 4       DFS-Visit (G, v)
- 5   u.color = BLACK
- 6   l.append(u)



**DFS-Visit(G,h,l):**

**DFS-Visit(G,f,l):**

**DFS-Visit(G,d,l):**

**DFS-Visit(G,a,l):**

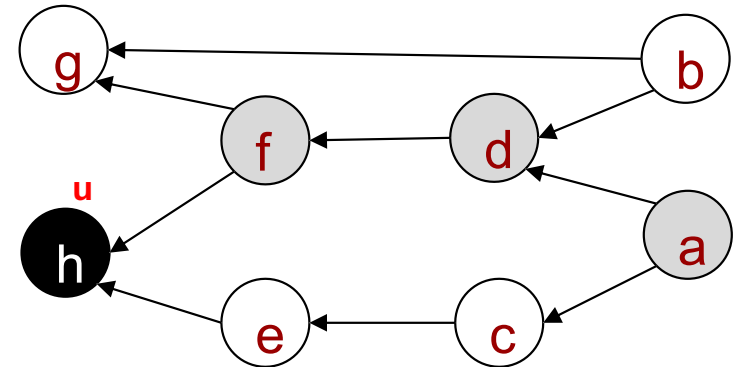
# Depth First-Search

## DFS-All (G)

- 1 for each vertex u
- 2   u.color = WHITE
- 3 finish\_list = empty\_list
- 4 for each vertex u do
- 5   if u.color == WHITE then
- 6     DFS-Visit (G, u, finish\_list)
- 7 return finish\_list

## DFS-Visit (G, u, l)

- 1   u.color = GRAY
- 2 for each vertex v in Adj(u) do
- 3   if v.color = WHITE then
- 4     DFS-Visit (G, v)
- 5   u.color = BLACK
- 6   l.append(u)



**Finish\_list:**



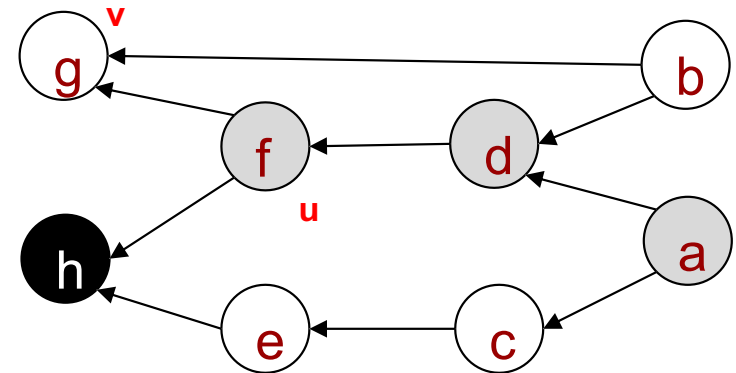
# Depth First-Search

## DFS-All (G)

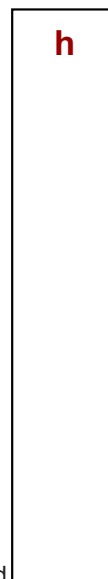
- 1 for each vertex  $u$
- 2  $u.color = WHITE$
- 3  $finish\_list = empty\_list$
- 4 for each vertex  $u$  do
- 5 if  $u.color == WHITE$  then
- 6     DFS-Visit (G,  $u$ ,  $finish\_list$ )
- 7 return  $finish\_list$

## DFS-Visit (G, $u$ , $l$ )

- 1  $u.color = GRAY$
- 2 for each vertex  $v$  in  $Adj(u)$  do
- 3 if  $v.color = WHITE$  then
- 4     DFS-Visit (G,  $v$ )
- 5  $u.color = BLACK$
- 6  $l.append(u)$



**Finish\_list:**



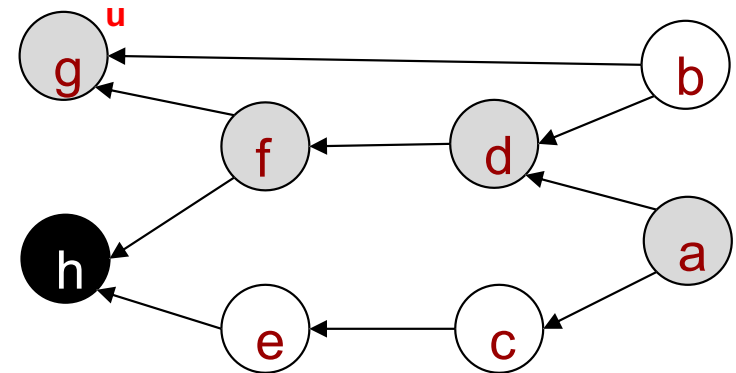
# Depth First-Search

## DFS-All (G)

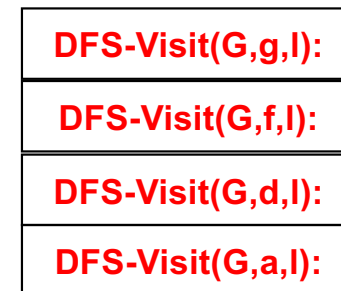
- 1 for each vertex u
- 2 u.color = WHITE
- 3 finish\_list = empty\_list
- 4 for each vertex u do
- 5 if u.color == WHITE then
- 6 DFS-Visit (G, u, finish\_list)
- 7 return finish\_list

## DFS-Visit (G, u,l)

- 1 u.color = GRAY
- 2 for each vertex v in Adj(u) do
- 3 if v.color = WHITE then
- 4 DFS-Visit (G, v)
- 5 u.color = BLACK
- 6 l.append(u)



Finish\_list:



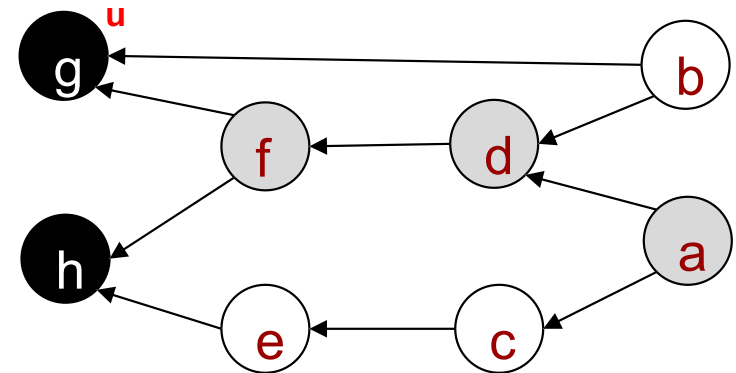
# Depth First-Search

## DFS-All (G)

- 1 for each vertex u
- 2   u.color = WHITE
- 3 finish\_list = empty\_list
- 4 for each vertex u do
- 5   if u.color == WHITE then
- 6     DFS-Visit (G, u, finish\_list)
- 7 return finish\_list

## DFS-Visit (G, u, l)

- 1   u.color = GRAY
- 2 for each vertex v in Adj(u) do
- 3   if v.color = WHITE then
- 4     DFS-Visit (G, v)
- 5   u.color = BLACK
- 6   l.append(u)



**Finish\_list:**

h, g
---------

<b>DFS-Visit(G,g,l):</b>
<b>DFS-Visit(G,f,l):</b>
<b>DFS-Visit(G,d,l):</b>
<b>DFS-Visit(G,a,l):</b>



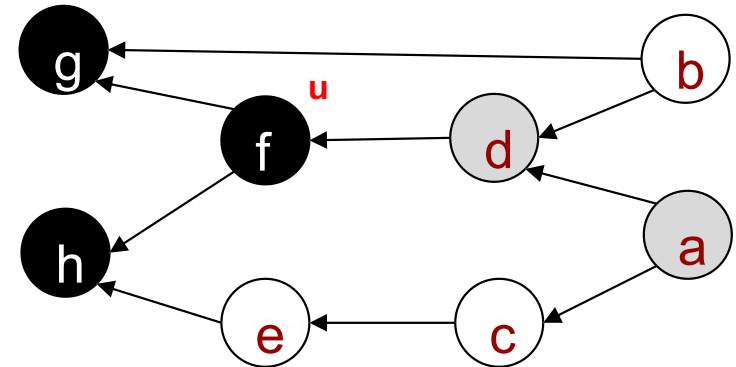
# Depth First-Search

## DFS-All (G)

- 1 for each vertex u
- 2 u.color = WHITE
- 3 finish\_list = empty\_list
- 4 for each vertex u do
- 5 if u.color == WHITE then
- 6 DFS-Visit (G, u, finish\_list)
- 7 return finish\_list

## DFS-Visit (G, u, l)

- 1 u.color = GRAY
- 2 for each vertex v in Adj(u) do
- 3 if v.color = WHITE then
- 4 DFS-Visit (G, v)
- 5 u.color = BLACK
- 6 l.append(u)



**Finish\_list:**

h,  
 g,  
 f

DFS-Visit(G,f,l):

DFS-Visit(G,d,l):

DFS-Visit(G,a,l):

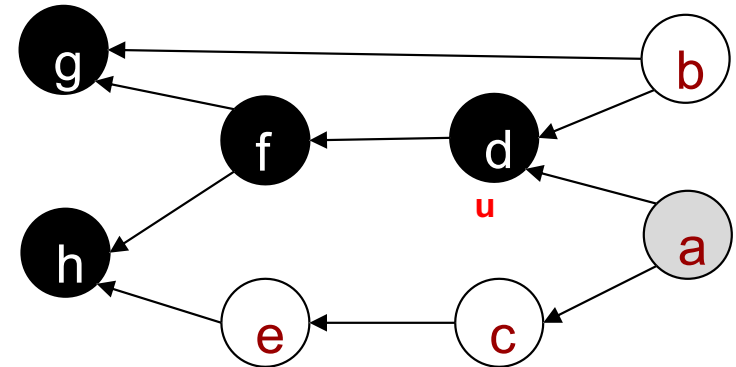
# Depth First-Search

## DFS-All (G)

- 1 for each vertex u
- 2   u.color = WHITE
- 3 finish\_list = empty\_list
- 4 for each vertex u do
- 5   if u.color == WHITE then
- 6     DFS-Visit (G, u, finish\_list)
- 7 return finish\_list

## DFS-Visit (G, u, l)

- 1   u.color = GRAY
- 2 for each vertex v in Adj(u) do
- 3   if v.color = WHITE then
- 4     DFS-Visit (G, v)
- 5   u.color = BLACK
- 6   l.append(u)



**Finish\_list:**

h,  
g,  
f,  
d

**DFS-Visit(G,d,l):**

**DFS-Visit(G,a,l):**

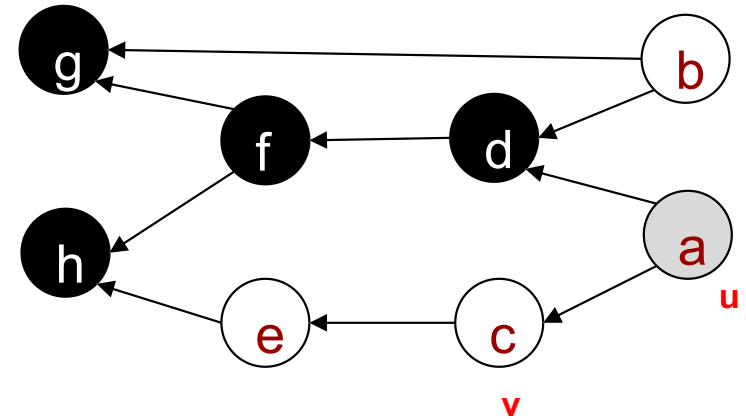
# Depth First-Search

## DFS-All (G)

- 1 for each vertex  $u$
- 2  $u.color = WHITE$
- 3  $finish\_list = empty\_list$
- 4 for each vertex  $u$  do
- 5 if  $u.color == WHITE$  then
- 6     DFS-Visit (G,  $u$ ,  $finish\_list$ )
- 7 return  $finish\_list$

## DFS-Visit (G, $u$ , $l$ )

- 1  $u.color = GRAY$
- 2 for each vertex  $v$  in  $Adj(u)$  do
- 3 if  $v.color = WHITE$  then
- 4     DFS-Visit (G,  $v$ )
- 5  $u.color = BLACK$
- 6  $l.append(u)$



**Finish\_list:**

h,  
g,  
f,  
d

**DFS-Visit(G,a,l):**

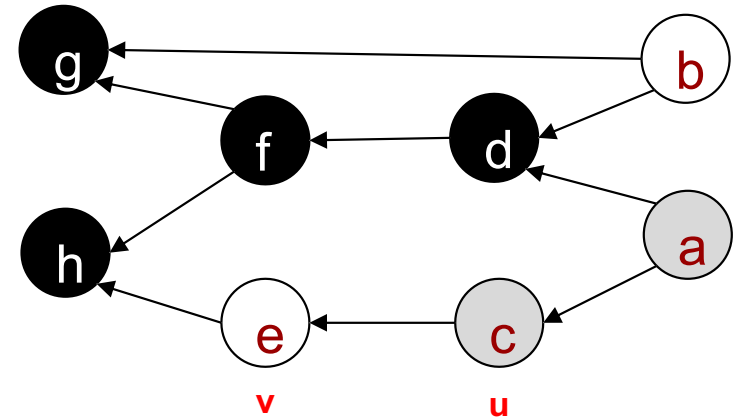
# Depth First-Search

```

DFS-All (G)
1 for each vertex u
2   u.color = WHITE
3 finish_list = empty_list
4 for each vertex u do
5   if u.color == WHITE then
6     DFS-Visit (G, u, finish_list)
7 return finish_list
    
```

```

DFS-Visit (G, u, l)
1 u.color = GRAY
2 for each vertex v in Adj(u) do
3   if v.color = WHITE then
4     DFS-Visit (G, v)
5 u.color = BLACK
6 l.append(u)
    
```



**Finish\_list:**

- h,
- g,
- f,
- d

```

DFS-Visit(G,c,l):
DFS-Visit(G,a,l):
    
```

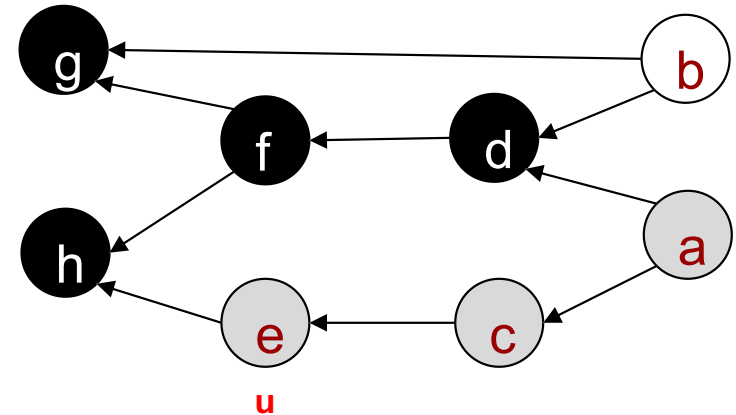
# Depth First-Search

## DFS-All (G)

- 1 for each vertex u
- 2   u.color = WHITE
- 3 finish\_list = empty\_list
- 4 for each vertex u do
- 5   if u.color == WHITE then
- 6     DFS-Visit (G, u, finish\_list)
- 7 return finish\_list

## DFS-Visit (G, u, l)

- 1   u.color = GRAY
- 2 for each vertex v in Adj(u) do
- 3   if v.color = WHITE then
- 4     DFS-Visit (G, v)
- 5   u.color = BLACK
- 6   l.append(u)



**Finish\_list:**

h,  
g,  
f,  
d

**DFS-Visit(G,e,l):**

**DFS-Visit(G,c,l):**

**DFS-Visit(G,a,l):**

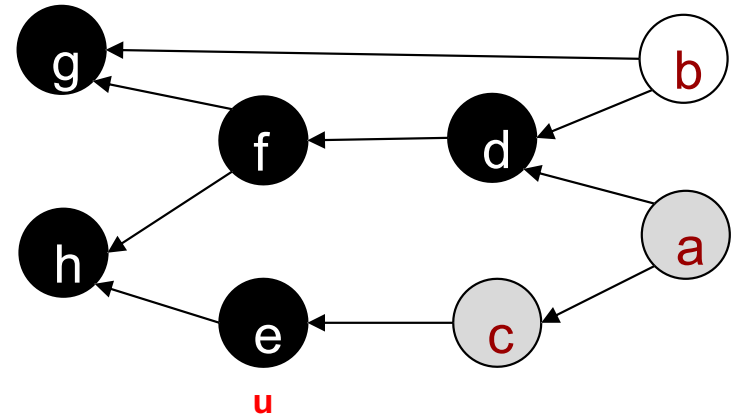
# Depth First-Search

**DFS-All (G)**

- 1 for each vertex  $u$
- 2  $u.color = WHITE$
- 3  $finish\_list = empty\_list$
- 4 for each vertex  $u$  do
- 5   if  $u.color == WHITE$  then
- 6     DFS-Visit (G,  $u$ ,  $finish\_list$ )
- 7 return  $finish\_list$

**DFS-Visit (G,  $u$ ,  $l$ )**

- 1  $u.color = GRAY$
- 2 for each vertex  $v$  in  $Adj(u)$  do
- 3   if  $v.color = WHITE$  then
- 4     DFS-Visit (G,  $v$ )
- 5  $u.color = BLACK$
- 6  $l.append(u)$



**Finish\_list:**

h,  
g,  
f,  
d,  
e

**DFS-Visit(G,e,l):**

---

**DFS-Visit(G,c,l):**

---

**DFS-Visit(G,a,l):**

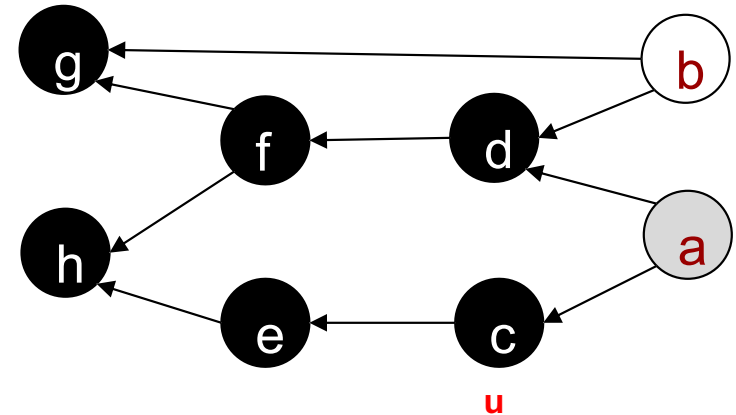
# Depth First-Search

## DFS-All (G)

- 1 for each vertex  $u$
- 2  $u.color = WHITE$
- 3  $finish\_list = empty\_list$
- 4 for each vertex  $u$  do
- 5 if  $u.color == WHITE$  then
- 6     DFS-Visit (G,  $u$ ,  $finish\_list$ )
- 7 return  $finish\_list$

## DFS-Visit (G, $u$ , $l$ )

- 1  $u.color = GRAY$
- 2 for each vertex  $v$  in  $Adj(u)$  do
- 3 if  $v.color = WHITE$  then
- 4     DFS-Visit (G,  $v$ )
- 5  $u.color = BLACK$
- 6  $l.append(u)$



**Finish\_list:**

h,  
g,  
f,  
d,  
e,  
c

**DFS-Visit(G,c,l):**

**DFS-Visit(G,a,l):**

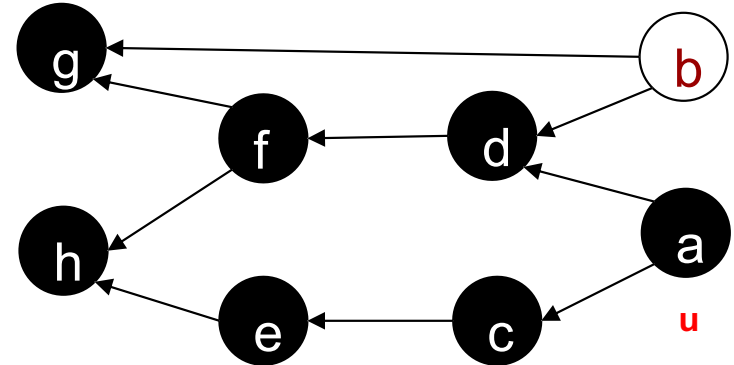
# Depth First-Search

## DFS-All (G)

- 1 for each vertex  $u$
- 2  $u.color = WHITE$
- 3  $finish\_list = empty\_list$
- 4 for each vertex  $u$  do
- 5   if  $u.color == WHITE$  then
- 6     DFS-Visit (G,  $u$ ,  $finish\_list$ )
- 7 return  $finish\_list$

## DFS-Visit (G, $u$ , $l$ )

- 1  $u.color = GRAY$
- 2 for each vertex  $v$  in  $Adj(u)$  do
- 3   if  $v.color = WHITE$  then
- 4     DFS-Visit (G,  $v$ )
- 5  $u.color = BLACK$
- 6  $l.append(u)$



**Finish\_list:**

h,  
g,  
f,  
d,  
e,  
c,  
a

**DFS-Visit(G,a,l):**

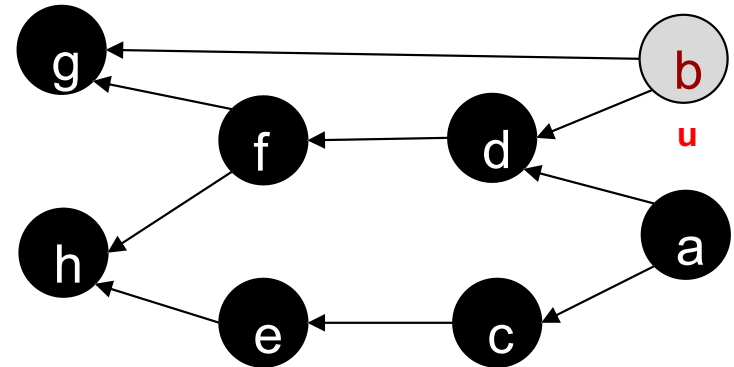


# Depth First-Search

## DFS-All (G)

- 1 for each vertex u
- 2 u.color = WHITE
- 3 finish\_list = empty\_list
- 4 for each vertex u do
- 5 if u.color == WHITE then
- 6 DFS-Visit (G, u, finish\_list)
- 7 return finish\_list

May iterate through many complete vertices before finding b to launch a new search from



## DFS-Visit (G, u, l)

- 1 u.color = GRAY
- 2 for each vertex v in Adj(u) do
- 3 if v.color = WHITE then
- 4 DFS-Visit (G, v)
- 5 u.color = BLACK
- 6 l.append(u)

Finish\_list:

h,  
g,  
f,  
d,  
e,  
c,  
a

DFS-Visit(G,b,l):

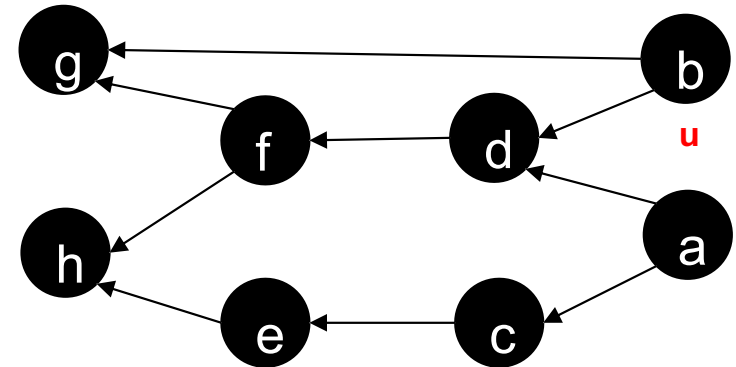
# Depth First-Search

## DFS-All (G)

- 1 for each vertex u
- 2   u.color = WHITE
- 3 finish\_list = empty\_list
- 4 for each vertex u do
- 5   if u.color == WHITE then
- 6     DFS-Visit (G, u, finish\_list)
- 7 return finish\_list

## DFS-Visit (G, u, l)

- 1   u.color = GRAY
- 2 for each vertex v in Adj(u) do
- 3   if v.color = WHITE then
- 4     DFS-Visit (G, v, l)
- 5   u.color = BLACK
- 6   l.append(u)



**Finish\_list:**

h,  
g,  
f,  
d,  
e,  
c,  
a,  
b

**DFS-Visit(G,b,l):**

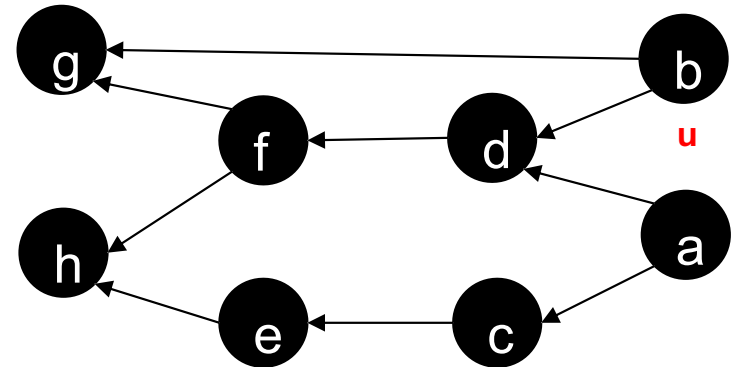
# Depth First-Search

## DFS-All (G)

- 1 for each vertex u
- 2   u.color = WHITE
- 3 finish\_list = empty\_list
- 4 for each vertex u do
- 5   if u.color == WHITE then
- 6     DFS-Visit (G, u, finish\_list)
- 7 return finish\_list

## DFS-Visit (G, u, l)

- 1   u.color = GRAY
- 2 for each vertex v in Adj(u) do
- 3   if v.color = WHITE then
- 4     DFS-Visit (G, v)
- 5   u.color = BLACK
- 6   l.append(u)



**Finish\_list:**

h,  
 g,  
 f,  
 d,  
 e,  
 c,  
 a,  
 b

With Cycles in the graph

**ANOTHER EXAMPLE (IF TIME)**

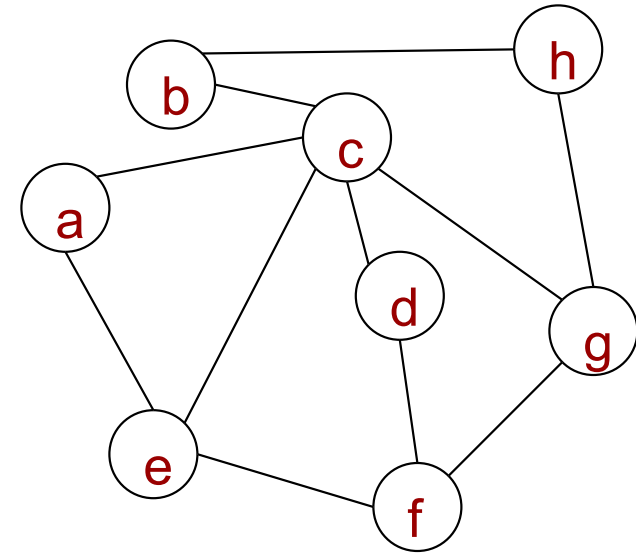
# Depth First-Search

## Toposort(G)

```
1 for each vertex u
2   u.color = WHITE
3 finish_list = empty_list
4 for each vertex u do
5   if u.color == WHITE then
6     DFS-Visit (G, u, finish_list)
```

## DFS-Visit (G, u)

```
1   u.color = GRAY
2   for each vertex v in Adj(u) do
3     if v.color = WHITE then
4       DFS-Visit (G, v)
5   u.color = BLACK
6   finish_list.append(u)
```



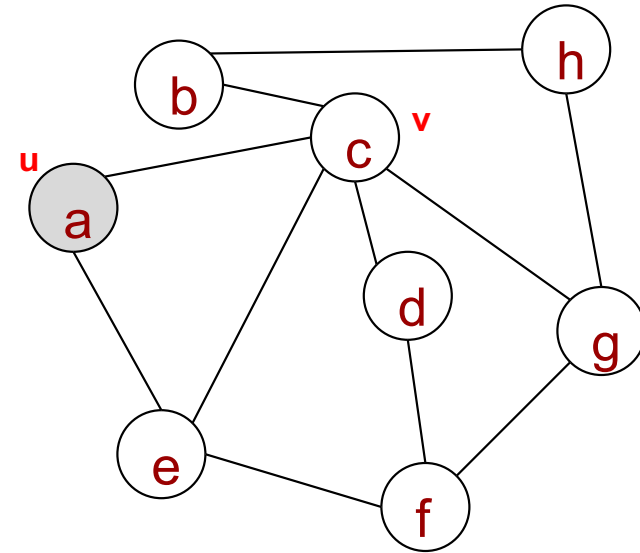
# Depth First-Search

## Toposort(G)

- 1 for each vertex  $u$
- 2  $u.color = WHITE$
- 3  $finish\_list = empty\_list$
- 4 for each vertex  $u$  do
- 5   if  $u.color == WHITE$  then
- 6     DFS-Visit (G,  $u$ ,  $finish\_list$ )

## DFS-Visit (G, $u$ , $l$ )

- 1  $u.color = GRAY$
- 2 for each vertex  $v$  in  $Adj(u)$  do
- 3   if  $v.color = WHITE$  then
- 4     DFS-Visit (G,  $v$ ,  $l$ )
- 5  $u.color = BLACK$
- 6  $finish\_list.append(u)$



**DFS-Visit(G,a):**

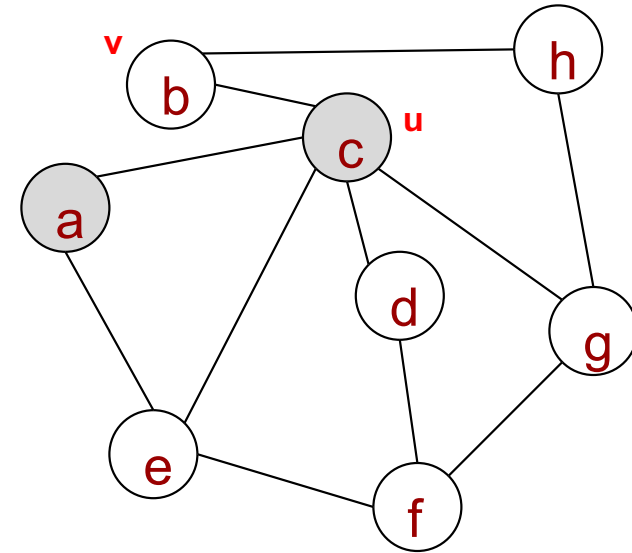
# Depth First-Search

## Toposort(G)

- 1 for each vertex  $u$
- 2  $u.color = WHITE$
- 3  $finish\_list = empty\_list$
- 4 for each vertex  $u$  do
- 5 if  $u.color == WHITE$  then
- 6  $DFS-Visit(G, u, finish\_list)$

## DFS-Visit (G, u)

- 1  $u.color = GRAY$
- 2 for each vertex  $v$  in  $Adj(u)$  do
- 3 if  $v.color = WHITE$  then
- 4  $DFS-Visit(G, v)$
- 5  $u.color = BLACK$
- 6  $finish\_list.append(u)$



**DFS-Visit(G,c):**

**DFS-Visit(G,a):**

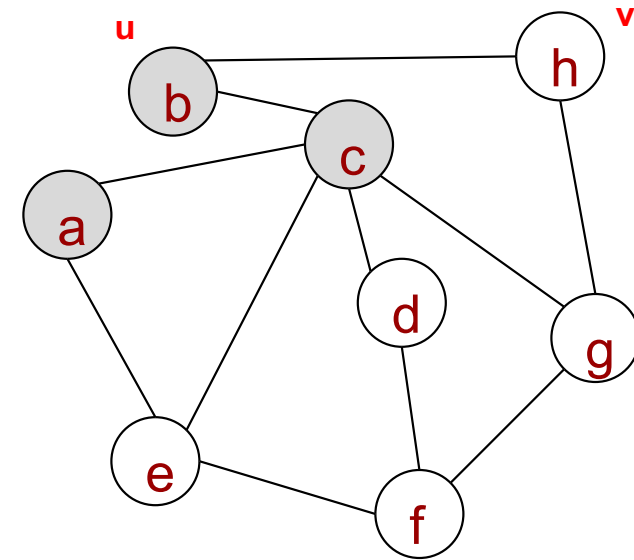
# Depth First-Search

## Toposort(G)

- 1 for each vertex  $u$
- 2  $u.color = WHITE$
- 3  $finish\_list = empty\_list$
- 4 for each vertex  $u$  do
- 5   if  $u.color == WHITE$  then
- 6     DFS-Visit (G,  $u$ ,  $finish\_list$ )

## DFS-Visit (G, $u$ )

- 1  $u.color = GRAY$
- 2 for each vertex  $v$  in  $Adj(u)$  do
- 3   if  $v.color = WHITE$  then
- 4     DFS-Visit (G,  $v$ )
- 5  $u.color = BLACK$
- 6  $finish\_list.append(u)$



DFS-Visit(G,b):

DFS-Visit(G,c):

DFS-Visit(G,a):



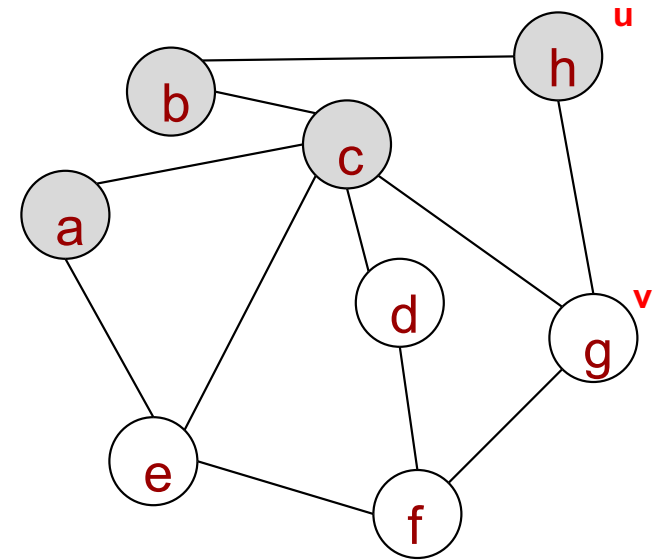
# Depth First-Search

## Toposort(G)

- 1 for each vertex  $u$
- 2  $u.color = WHITE$
- 3  $finish\_list = empty\_list$
- 4 for each vertex  $u$  do
- 5     if  $u.color == WHITE$  then
- 6         DFS-Visit (G,  $u$ ,  $finish\_list$ )

## DFS-Visit (G, $u$ )

- 1  $u.color = GRAY$
- 2 for each vertex  $v$  in  $Adj(u)$  do
- 3     if  $v.color = WHITE$  then
- 4         DFS-Visit (G,  $v$ )
- 5  $u.color = BLACK$
- 6  $finish\_list.append(u)$



**DFS-Visit(G,h):**

**DFS-Visit(G,b):**

**DFS-Visit(G,c):**

**DFS-Visit(G,a):**

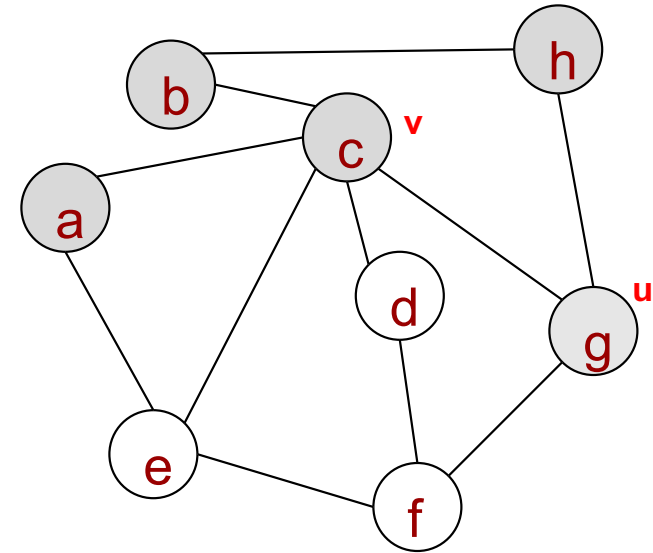
# Depth First-Search

## Toposort(G)

- 1 for each vertex  $u$
- 2  $u.color = WHITE$
- 3  $finish\_list = empty\_list$
- 4 for each vertex  $u$  do
- 5   if  $u.color == WHITE$  then
- 6     DFS-Visit (G,  $u$ ,  $finish\_list$ )

## DFS-Visit (G, $u$ )

- 1  $u.color = GRAY$
- 2 for each vertex  $v$  in  $Adj(u)$  do
- 3   if  $v.color = WHITE$  then
- 4     DFS-Visit (G,  $v$ )
- 5  $u.color = BLACK$
- 6  $finish\_list.append(u)$



**DFS-Visit(G,g):**

**DFS-Visit(G,h):**

**DFS-Visit(G,b):**

**DFS-Visit(G,c):**

**DFS-Visit(G,a):**

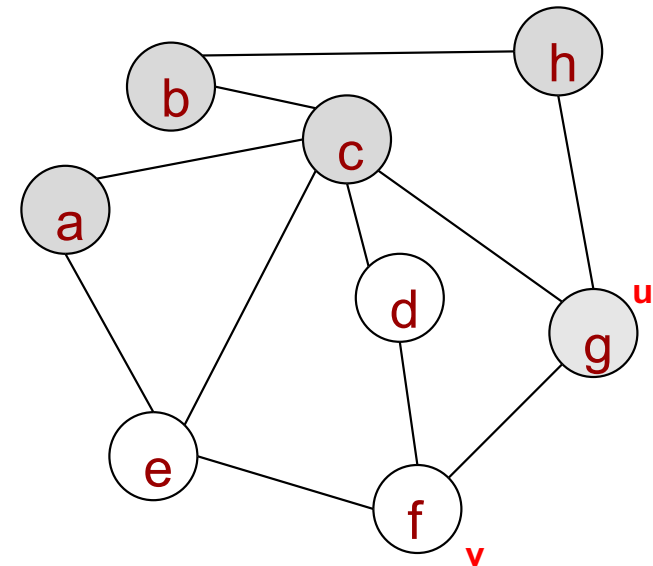
# Depth First-Search

## Toposort(G)

- 1 for each vertex  $u$
- 2  $u.color = WHITE$
- 3  $finish\_list = empty\_list$
- 4 for each vertex  $u$  do
- 5     if  $u.color == WHITE$  then
- 6         DFS-Visit (G,  $u$ ,  $finish\_list$ )

## DFS-Visit (G, $u$ )

- 1  $u.color = GRAY$
- 2 for each vertex  $v$  in  $Adj(u)$  do
- 3     if  $v.color = WHITE$  then
- 4         DFS-Visit (G,  $v$ )
- 5  $u.color = BLACK$
- 6  $finish\_list.append(u)$



**DFS-Visit(G,g):**

**DFS-Visit(G,h):**

**DFS-Visit(G,b):**

**DFS-Visit(G,c):**

**DFS-Visit(G,a):**

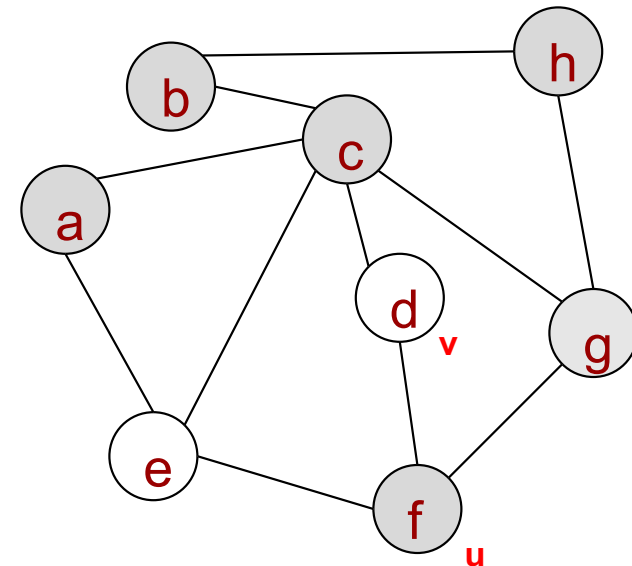
# Depth First-Search

## Toposort(G)

- 1 for each vertex u
- 2   u.color = WHITE
- 3 finish\_list = empty\_list
- 4 for each vertex u do
- 5   if u.color == WHITE then
- 6     DFS-Visit (G, u, finish\_list)

## DFS-Visit (G, u)

- 1   u.color = GRAY
- 2   for each vertex v in Adj(u) do
- 3     if v.color = WHITE then
- 4       DFS-Visit (G, v)
- 5   u.color = BLACK
- 6   finish\_list.append(u)



**DFS-Visit(G,f):**

**DFS-Visit(G,g):**

**DFS-Visit(G,h):**

**DFS-Visit(G,b):**

**DFS-Visit(G,c):**

**DFS-Visit(G,a):**

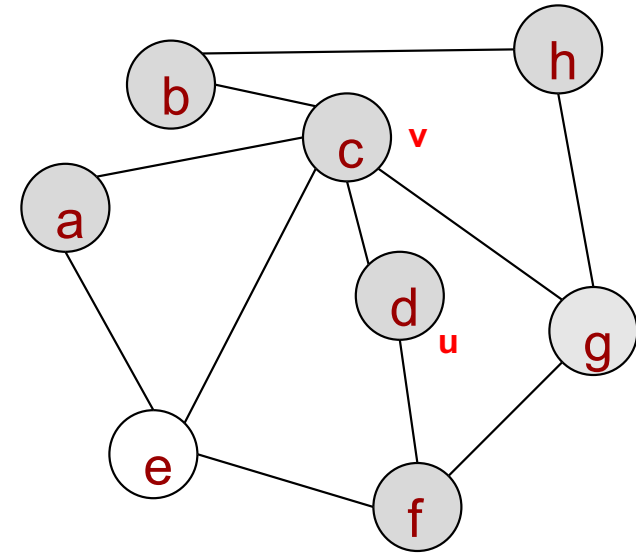
# Depth First-Search

## Toposort(G)

- 1 for each vertex  $u$
- 2  $u.color = WHITE$
- 3  $finish\_list = empty\_list$
- 4 for each vertex  $u$  do
- 5     if  $u.color == WHITE$  then
- 6         DFS-Visit (G,  $u$ ,  $finish\_list$ )

## DFS-Visit (G, $u$ )

- 1  $u.color = GRAY$
- 2 for each vertex  $v$  in  $Adj(u)$  do
- 3     if  $v.color = WHITE$  then
- 4         DFS-Visit (G,  $v$ )
- 5  $u.color = BLACK$
- 6  $finish\_list.append(u)$



DFS-Visit(G,d):
DFS-Visit(G,f):
DFS-Visit(G,g):
DFS-Visit(G,h):
DFS-Visit(G,b):
DFS-Visit(G,c):
DFS-Visit(G,a):

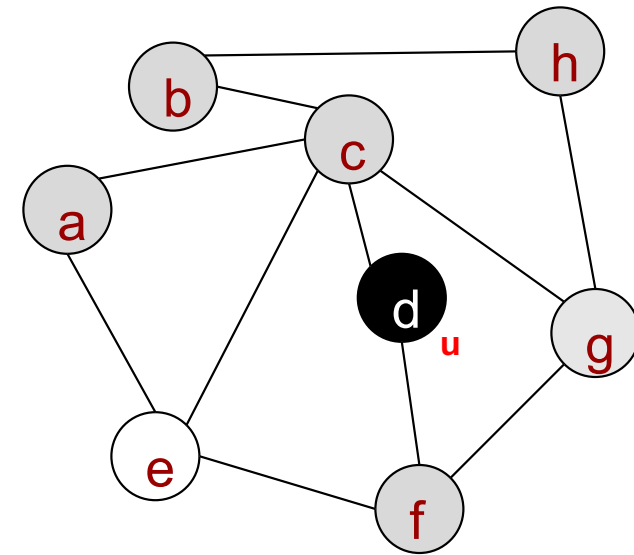
# Depth First-Search

## Toposort(G)

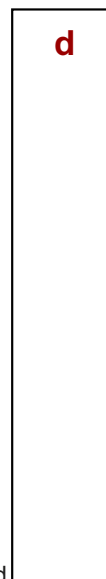
- 1 for each vertex  $u$
- 2  $u.color = WHITE$
- 3  $finish\_list = empty\_list$
- 4 for each vertex  $u$  do
- 5   if  $u.color == WHITE$  then
- 6     DFS-Visit (G,  $u$ ,  $finish\_list$ )

## DFS-Visit (G, $u$ )

- 1  $u.color = GRAY$
- 2 for each vertex  $v$  in  $Adj(u)$  do
- 3   if  $v.color = WHITE$  then
- 4     DFS-Visit (G,  $v$ )
- 5  $u.color = BLACK$
- 6  $finish\_list.append(u)$



**DFSQ:**



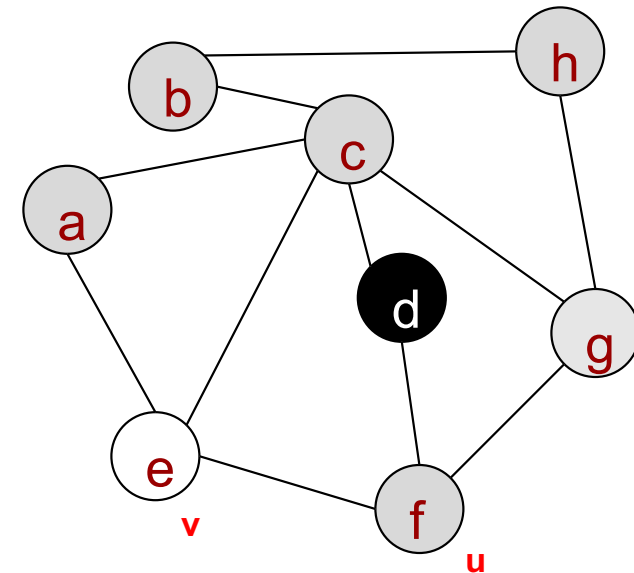
# Depth First-Search

## Toposort(G)

- 1 for each vertex  $u$
- 2  $u.color = WHITE$
- 3  $finish\_list = empty\_list$
- 4 for each vertex  $u$  do
- 5     if  $u.color == WHITE$  then
- 6         DFS-Visit (G,  $u$ ,  $finish\_list$ )

## DFS-Visit (G, $u$ )

- 1  $u.color = GRAY$
- 2 for each vertex  $v$  in  $Adj(u)$  do
- 3     if  $v.color = WHITE$  then
- 4         DFS-Visit (G,  $v$ )
- 5  $u.color = BLACK$
- 6  $finish\_list.append(u)$



DFSQ:



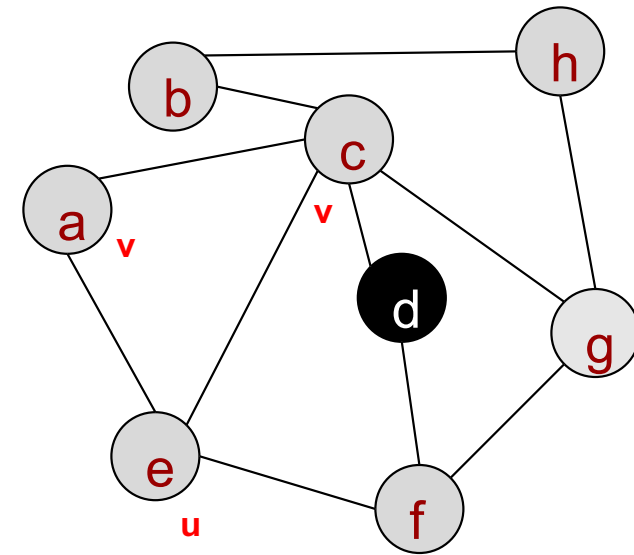
# Depth First-Search

## Toposort(G)

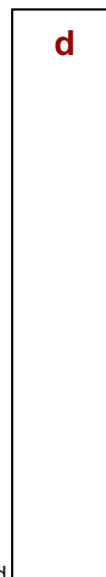
- 1 for each vertex u
- 2   u.color = WHITE
- 3 finish\_list = empty\_list
- 4 for each vertex u do
- 5   if u.color == WHITE then
- 6     DFS-Visit (G, u, finish\_list)

## DFS-Visit (G, u)

- 1   u.color = GRAY
- 2   for each vertex v in Adj(u) do
- 3     if v.color = WHITE then
- 4       DFS-Visit (G, v)
- 5   u.color = BLACK
- 6   finish\_list.append(u)



**DFSQ:**









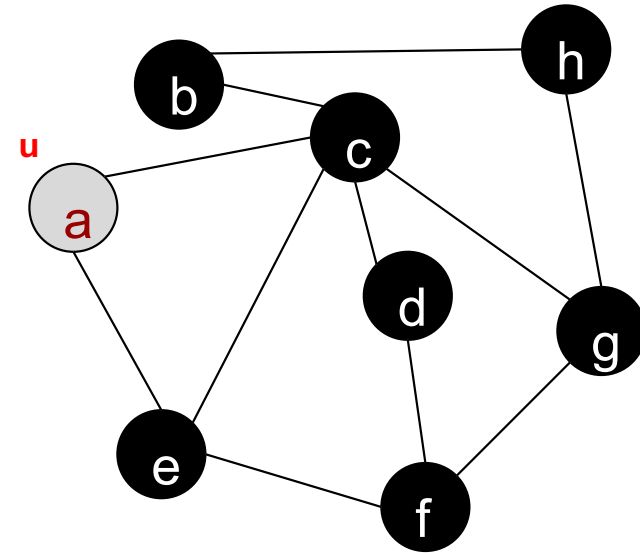
# Depth First-Search

## Toposort(G)

- 1 for each vertex  $u$
- 2  $u.color = WHITE$
- 3  $finish\_list = empty\_list$
- 4 for each vertex  $u$  do
- 5   if  $u.color == WHITE$  then
- 6     DFS-Visit (G,  $u$ ,  $finish\_list$ )

## DFS-Visit (G, $u$ )

- 1  $u.color = GRAY$
- 2 for each vertex  $v$  in  $Adj(u)$  do
- 3   if  $v.color = WHITE$  then
- 4     DFS-Visit (G,  $v$ )
- 5  $u.color = BLACK$
- 6  $finish\_list.append(u)$



DFSQ:

d  
e  
f  
g  
h  
b  
c

DFS-Visit(G,a):

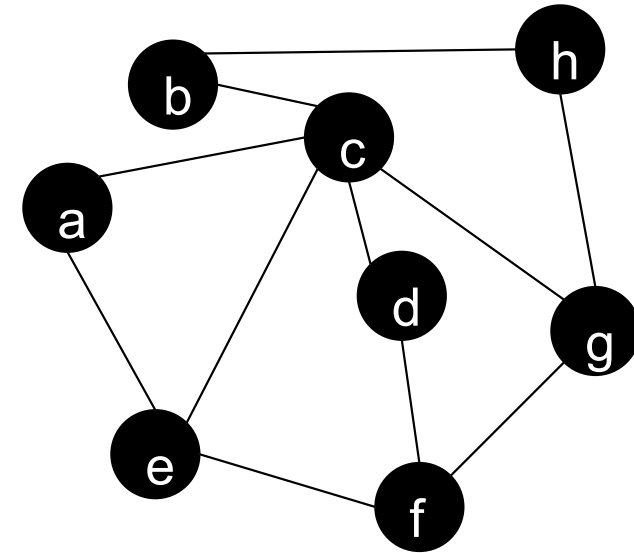
# Depth First-Search

## Toposort(G)

- 1 for each vertex u
- 2   u.color = WHITE
- 3 finish\_list = empty\_list
- 4 for each vertex u do
- 5   if u.color == WHITE then
- 6     DFS-Visit (G, u, finish\_list)

## DFS-Visit (G, u)

- 1   u.color = GRAY
- 2   for each vertex v in Adj(u) do
- 3     if v.color = WHITE then
- 4       DFS-Visit (G, v)
- 5   u.color = BLACK
- 6   finish\_list.append(u)



**DFSQ:**

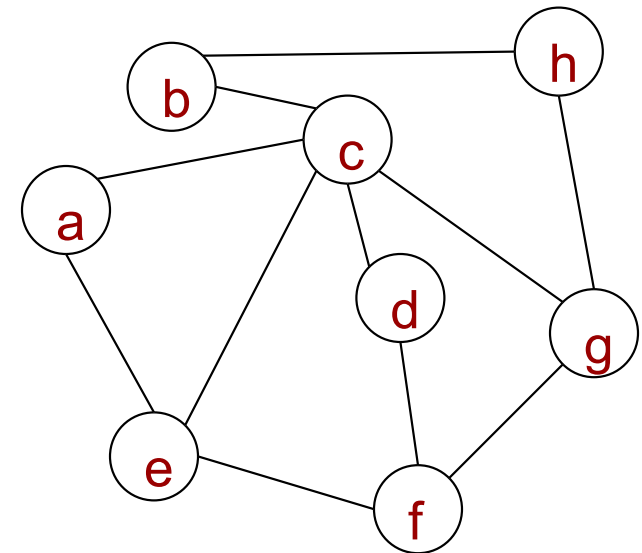
d  
e  
f  
g  
h  
b  
c  
a

# ITERATIVE VERSION

# Depth First-Search

DFS (G,s)

```
1 for each vertex u
2   u.color = WHITE
3 st = new Stack
4 st.push_back(s)
5 while st not empty
6   u = st.back()
7   if u.color == WHITE then
8     u.color = GRAY
9     foreach vertex v in Adj(u) do
10      if v.color == WHITE
11        st.push_back(v)
12  else if u.color != WHITE
13    u.color = BLACK
14  st.pop_back()
```



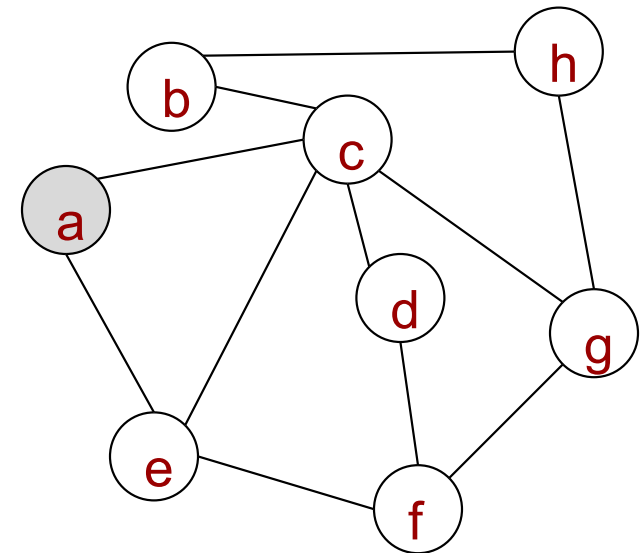
st:

a

# Depth First-Search

DFS (G,s)

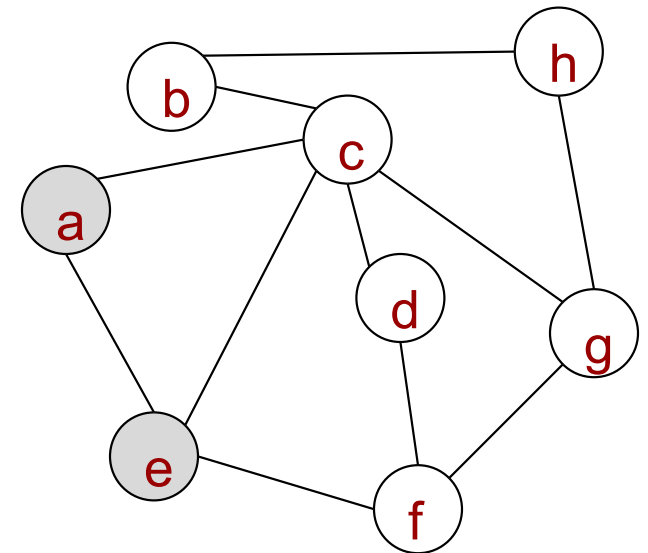
- 1 for each vertex u
- 2   u.color = WHITE
- 3 st = new Stack
- 4 st.push\_back(s)
- 5 while st not empty
- 6   u = st.back()
- 7   if u.color == WHITE then
- 8     u.color = GRAY
- 9     foreach vertex v in Adj(u) do
- 10      if v.color == WHITE
- 11        st.push\_back(v)
- 12    else if u.color != WHITE
- 13     u.color = BLACK
- 14    st.pop\_back()



# Depth First-Search

DFS (G,s)

- 1 for each vertex u
- 2   u.color = WHITE
- 3 st = new Stack
- 4 st.push\_back(s)
- 5 while st not empty
- 6   u = st.back()
- 7   if u.color == WHITE then
- 8     u.color = GRAY
- 9     foreach vertex v in Adj(u) do
- 10      if v.color == WHITE
- 11        st.push\_back(v)
- 12    else if u.color != WHITE
- 13     u.color = BLACK
- 14     st.pop\_back()

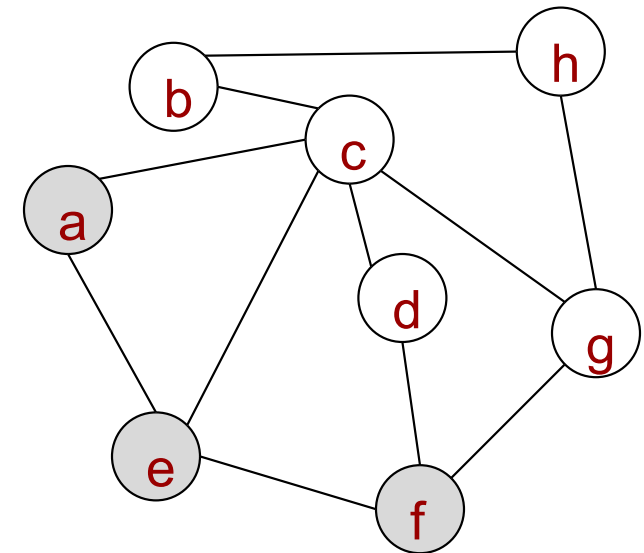




# Depth First-Search

DFS (G,s)

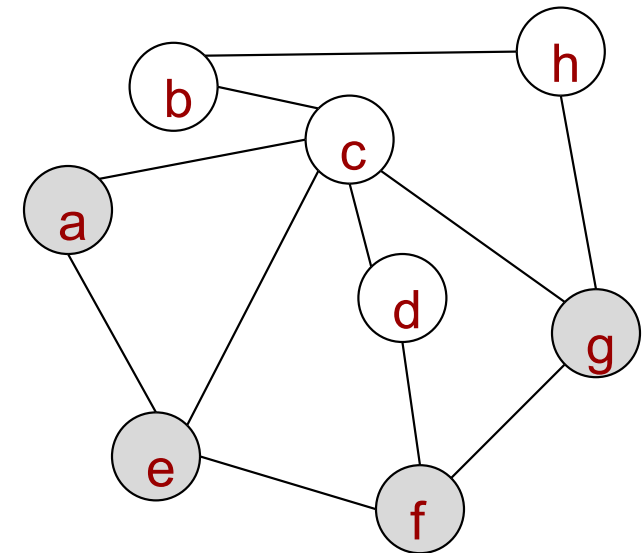
- 1 for each vertex u
- 2   u.color = WHITE
- 3 st = new Stack
- 4 st.push\_back(s)
- 5 while st not empty
- 6   u = st.back()
- 7   if u.color == WHITE then
- 8     u.color = GRAY
- 9     foreach vertex v in Adj(u) do
- 10      if v.color == WHITE
- 11        st.push\_back(v)
- 12    else if u.color != WHITE
- 13     u.color = BLACK
- 14    st.pop\_back()



# Depth First-Search

DFS (G,s)

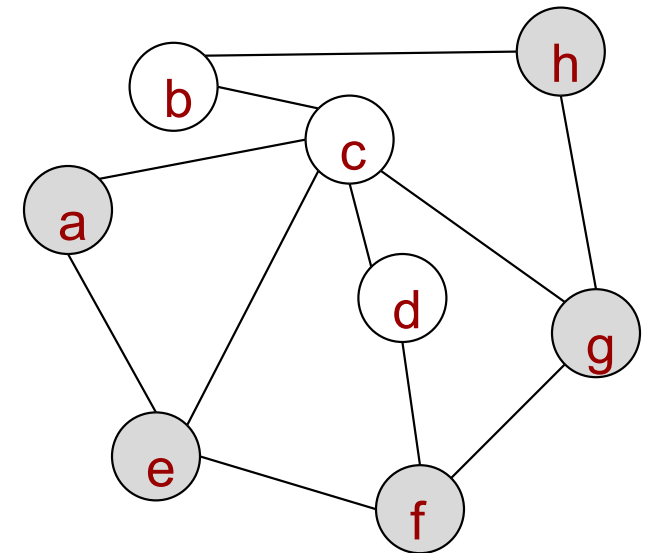
- 1 for each vertex u
- 2   u.color = WHITE
- 3 st = new Stack
- 4 st.push\_back(s)
- 5 while st not empty
- 6   u = st.back()
- 7   if u.color == WHITE then
- 8     u.color = GRAY
- 9     foreach vertex v in Adj(u) do
- 10      if v.color == WHITE
- 11        st.push\_back(v)
- 12    else if u.color != WHITE
- 13      u.color = BLACK
- 14      st.pop\_back()



# Depth First-Search

DFS (G,s)

```
1  for each vertex u
2    u.color = WHITE
3  st = new Stack
4  st.push_back(s)
5  while st not empty
6    u = st.back()
7    if u.color == WHITE then
8      u.color = GRAY
9      foreach vertex v in Adj(u) do
10         if v.color == WHITE
11           st.push_back(v)
12     else if u.color != WHITE
13       u.color = BLACK
14     st.pop_back()
```



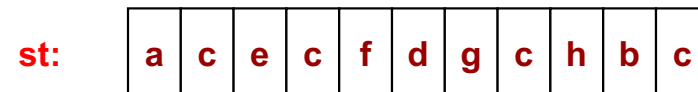
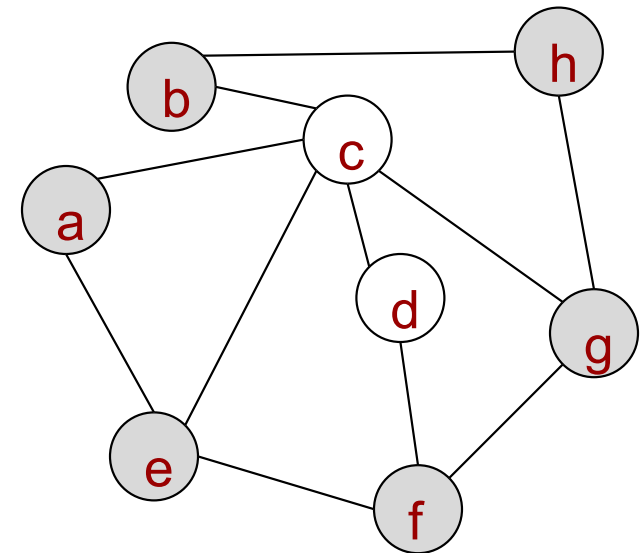
st:



# Depth First-Search

DFS (G,s)

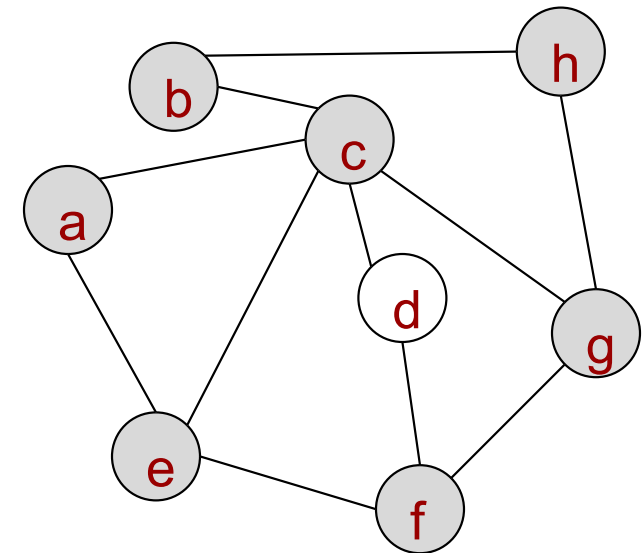
- 1 for each vertex u
- 2   u.color = WHITE
- 3 st = new Stack
- 4 st.push\_back(s)
- 5 while st not empty
- 6   u = st.back()
- 7   if u.color == WHITE then
- 8     u.color = GRAY
- 9     foreach vertex v in Adj(u) do
- 10      if v.color == WHITE
- 11        st.push\_back(v)
- 12    else if u.color != WHITE
- 13     u.color = BLACK
- 14     st.pop\_back()



# Depth First-Search

DFS (G,s)

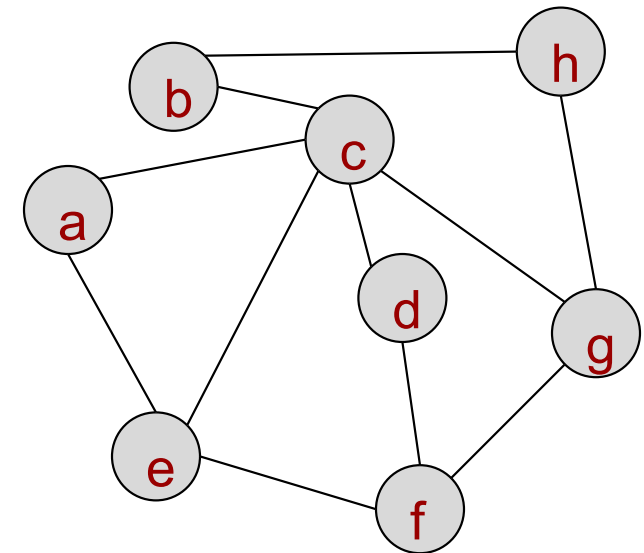
- 1 for each vertex u
- 2   u.color = WHITE
- 3 st = new Stack
- 4 st.push\_back(s)
- 5 while st not empty
- 6   u = st.back()
- 7   if u.color == WHITE then
- 8     u.color = GRAY
- 9     foreach vertex v in Adj(u) do
- 10      if v.color == WHITE
- 11        st.push\_back(v)
- 12    else if u.color != WHITE
- 13     u.color = BLACK
- 14     st.pop\_back()



# Depth First-Search

DFS (G,s)

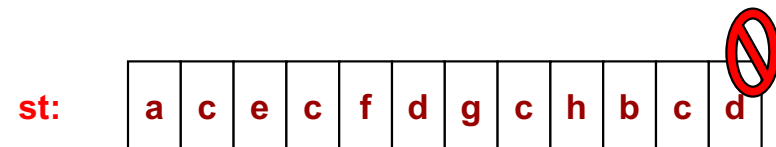
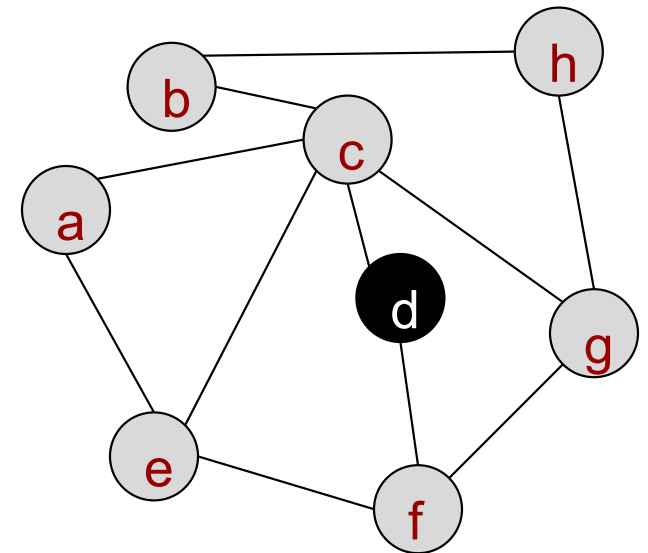
- 1 for each vertex u
- 2   u.color = WHITE
- 3 st = new Stack
- 4 st.push\_back(s)
- 5 while st not empty
- 6   u = st.back()
- 7   if u.color == WHITE then
- 8     u.color = GRAY
- 9     foreach vertex v in Adj(u) do
- 10      if v.color == WHITE
- 11        st.push\_back(v)
- 12    else if u.color != WHITE
- 13     u.color = BLACK
- 14     st.pop\_back()



# Depth First-Search

DFS (G,s)

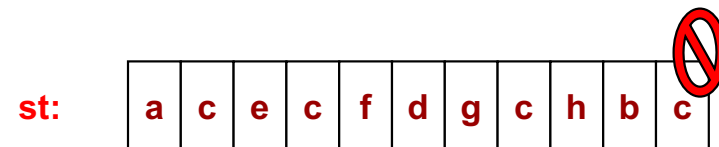
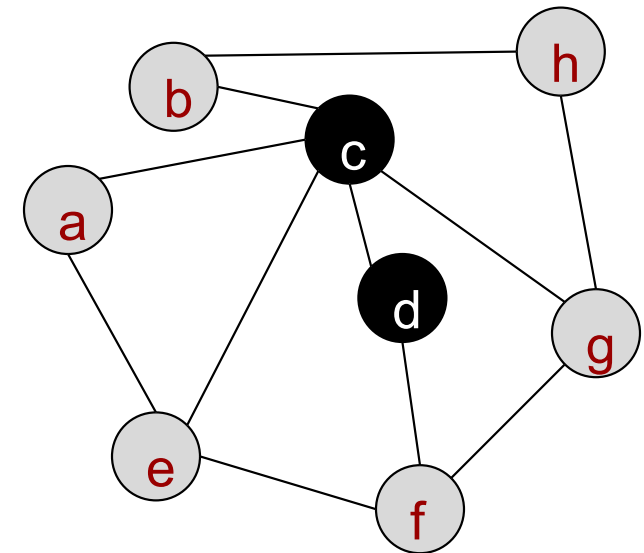
- 1 for each vertex u
- 2   u.color = WHITE
- 3 st = new Stack
- 4 st.push\_back(s)
- 5 while st not empty
- 6   u = st.back()
- 7   if u.color == WHITE then
- 8     u.color = GRAY
- 9     foreach vertex v in Adj(u) do
- 10      if v.color == WHITE
- 11        st.push\_back(v)
- 12    else if u.color != WHITE
- 13      u.color = BLACK
- 14      st.pop\_back()



# Depth First-Search

DFS (G,s)

- 1 for each vertex u
- 2   u.color = WHITE
- 3 st = new Stack
- 4 st.push\_back(s)
- 5 while st not empty
- 6   u = st.back()
- 7   if u.color == WHITE then
- 8     u.color = GRAY
- 9     foreach vertex v in Adj(u) do
- 10      if v.color == WHITE
- 11        st.push\_back(v)
- 12    else if u.color != WHITE
- 13     u.color = BLACK
- 14     st.pop\_back()

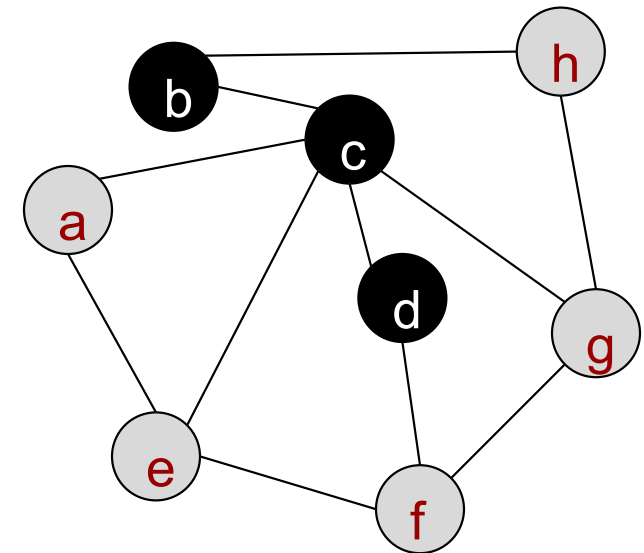




# Depth First-Search

DFS (G,s)

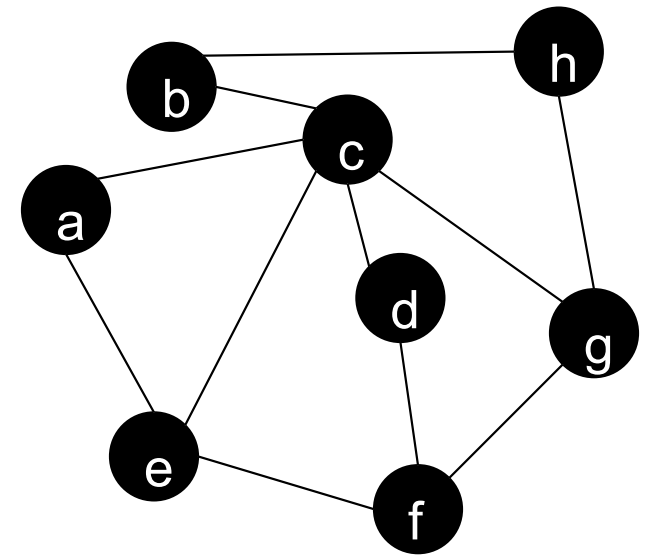
- 1 for each vertex u
- 2   u.color = WHITE
- 3 st = new Stack
- 4 st.push\_back(s)
- 5 while st not empty
- 6   u = st.back()
- 7   if u.color == WHITE then
- 8     u.color = GRAY
- 9     foreach vertex v in Adj(u) do
- 10      if v.color == WHITE
- 11        st.push\_back(v)
- 12    else if u.color != WHITE
- 13     u.color = BLACK
- 14     st.pop\_back()



# Depth First-Search

DFS (G,s)

```
1  for each vertex u
2    u.color = WHITE
3  st = new Stack
4  st.push_back(s)
5  while st not empty
6    u = st.back()
7    if u.color == WHITE then
8      u.color = GRAY
9      foreach vertex v in Adj(u) do
10         if v.color == WHITE
11           st.push_back(v)
12     else if u.color != WHITE
13       u.color = BLACK
14     st.pop_back()
```



# BFS vs. DFS Algorithm

- BFS and DFS are more similar than you think
  - Do we use a FIFO/Queue (BFS) or LIFO/Stack (DFS) to store vertices as we find them

## BFS-Visit (G, start\_node)

```
1 for each vertex u
2   u.color = WHITE
3   u.pred = nil
4   bfsq = new Queue
5   bfsq.push_back(start_node)
6   while bfsq not empty
7     u = bfsq.pop_front()
8     if u.color == WHITE
9       u.color = GRAY
10    foreach vertex v in Adj(u) do
11      bfsq.push_back(v)
```

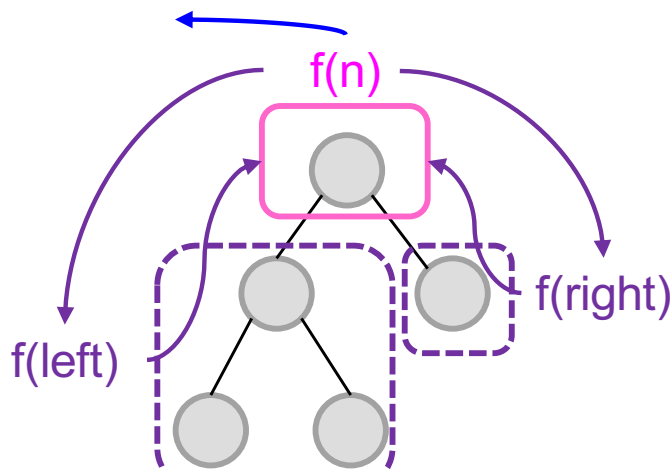
## DFS-Visit (G, start\_node)

```
1 for each vertex u
2   u.color = WHITE
3   u.pred = nil
4   st = new Stack
5   st.push_back(start_node)
6   while st not empty
7     u = st.top(); st.pop()
8     if u.color == WHITE
9       u.color = GRAY
10    foreach vertex v in Adj(u) do
11      st.push_back(v)
```

# SOLUTIONS

# Example 1: Count Nodes

- Write a recursive function to **count how many nodes** are in the binary tree
  - Only process 1 node at a time
  - Determine pre-, in-, or post-order based on whose answers you need to compute the result for your node
  - For in- or post-order traversals, determine how to use/combine results from recursion on children

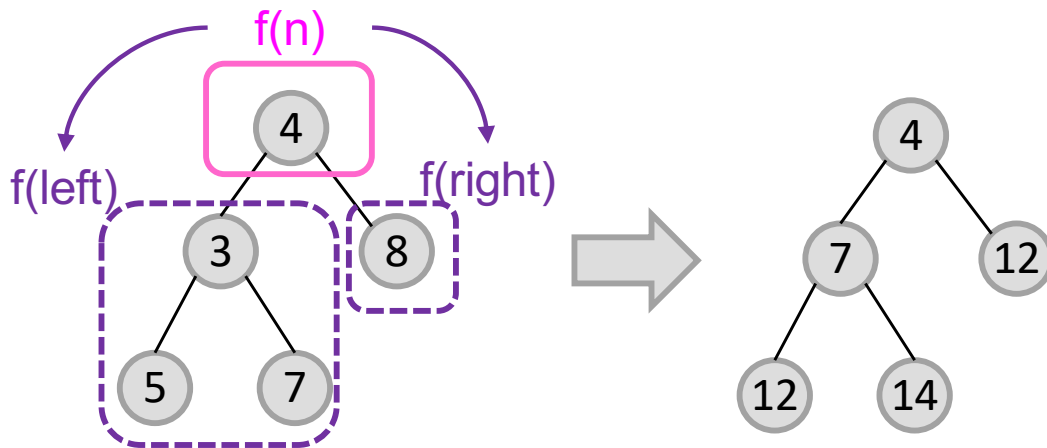


```
// Node definition
struct Tnode {
    int val;
    TNode *left, *right;
};

int count(TNode* root)
{
    if( root == NULL ) return 0;
    else {
        return 1 + count(root->left) +
                count(root->right);
    }
}
```

# Example 2: Prefix Sums

- Write a recursive function to **have each node store the sum of the values on the path from the root to each node.**
  - Only process 1 node at a time
  - Determine pre-, in-, or post-order based on whose answers you need to compute the result for your node



```

void prefixH(TNode* root, int psum)

void prefix(TNode* root)
{
    prefixH(root, 0);
}

void prefixH(TNode* root, int psum)
{
    if( root == NULL ) return;
    else {
        root->val += psum;
        prefixH(root->left, root->val);
        prefixH(root->right, root->val);
    }
}
    
```