CSCI 104

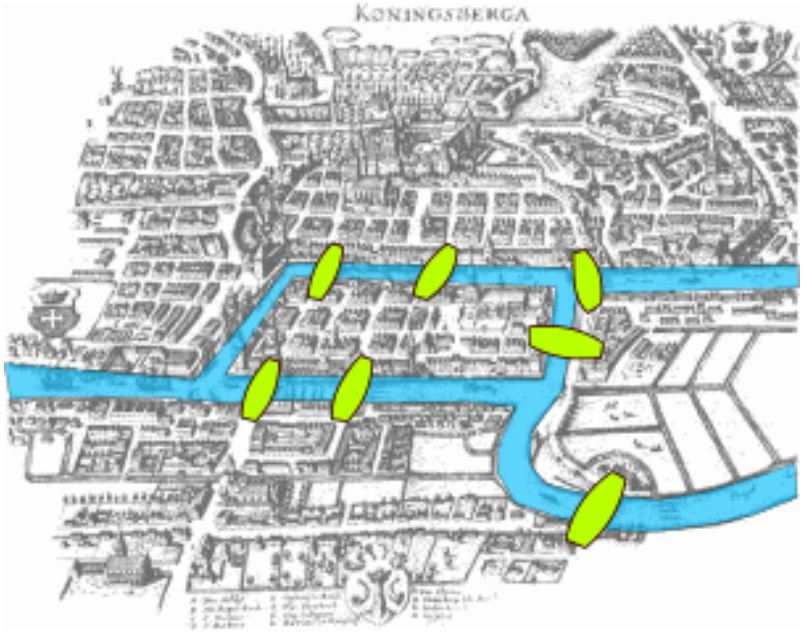
Rafael Ferreira da Silva

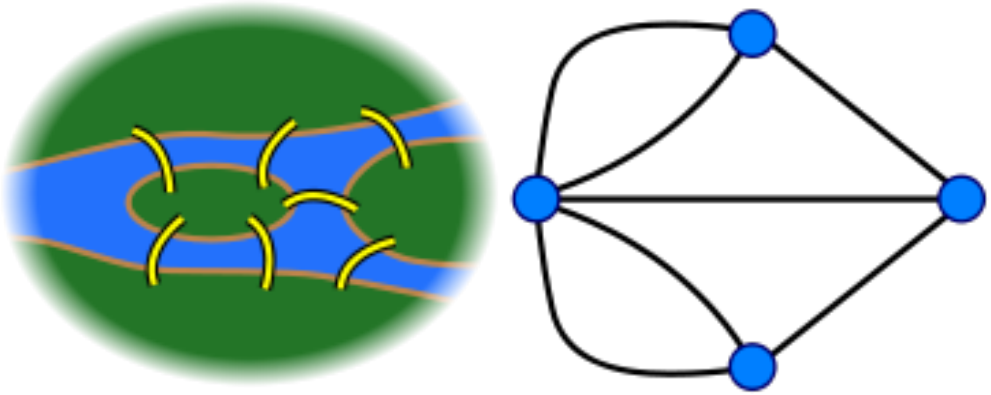
rafsilva@isi.edu

Slides adapted from: Mark Redekopp and David Kempe

Origin of Graph Theory

- In 1736, Euler solved the problem known as the Seven Bridges of Königsberg. The city of Königsberg, Prussia on the Pregel River, included two large islands connected to each other and the mainland by seven bridges.
- The problem is to decide whether it is possible to follow a path that crosses each bridge exactly once (and optionally: returns to the starting point)





http://en.wikipedia.org/wiki/Seven_Bridges_of_K%C3%B6nigsberg

Euler's Analysis

- Whenever you enter a non-terminal landmass by a bridge you must leave by another
  - Because its non-terminal you can't stay once you arrive
- Thus every non-terminal landmass must be touching an even number of bridges
  - So that you can enter on one bridge and leave on another
- However, all four of the land masses in the original problem are touched by an odd number of bridges (one is touched by 5 bridges, and each of the other three are touched by 3).

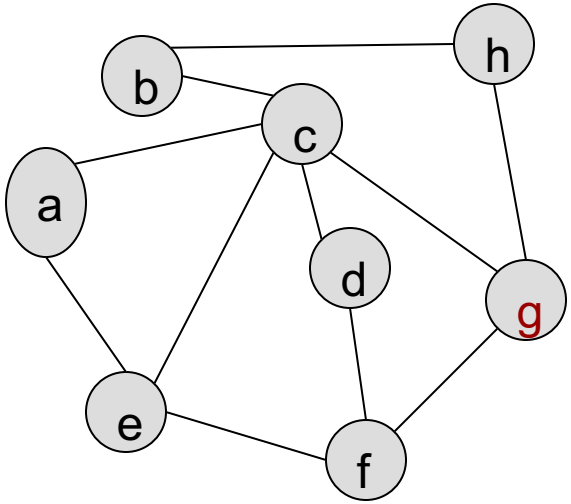
Explanation Using Graph Theory

- In "graph-speak", Euler showed that the possibility of a walk through a graph, traversing each edge exactly once, depends on the degrees of the nodes.
  - Euler walk = start/end at different vertices
  - Euler cycle = start/end at same vertex
  - The degree of a node is the number of edges touching it.
- Euler's argument shows that a necessary condition for the walk of the desired form is that the graph be connected and have exactly zero or two nodes of odd degree.
  - If there are 2 nodes of odd degree, we can form an Euler walk so that we will start at one of the odd-degree vertices and end at the other
- Since the graph corresponding to historical Königsberg has four nodes of odd degree, it cannot have an Eulerian path.

GRAPH REPRESENTATIONS

Graph Notation

- Graphs is a collection of vertices (or nodes) and edges that connect vertices
  - Let **V** be the set of vertices
  - Let **E** be the set of edges
  - Let **|V| or n** refer to the number of vertices
  - Let **|E| or m** refer to the number of edges



V
a
b
c
d
e
f
g
h

|V|=n=8

E
(a,c)
(a,e)
(b,h)
(b,c)
(c,e)
(c,d)
(c,g)
(d,f)
(e,f)
(f,g)
(g,h)

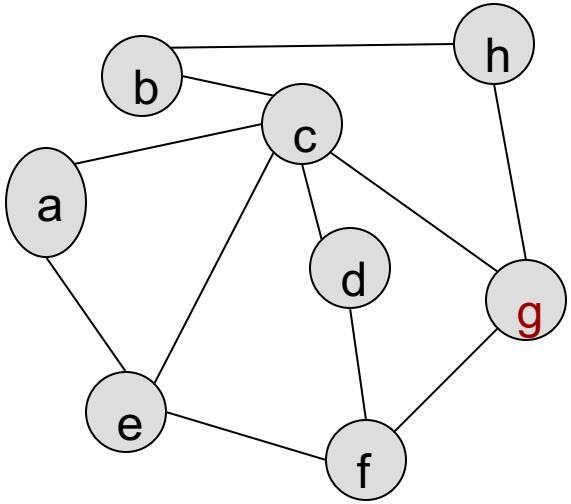
|E|=m=11

Graphs in the Real World

- Social networks
- Computer networks / Internet
- Path planning
- Interaction diagrams
- Bioinformatics

Basic Graph Representation

- Can simply store edges in list/array
  - Unsorted
  - Sorted



V
a
b
c
d
e
f
g
h

|V|=n=8

E
(a,c)
(a,e)
(b,h)
(b,c)
(c,e)
(c,d)
(c,g)
(d,f)
(e,f)
(f,g)
(g,h)

|E|=m=11

Graph ADT

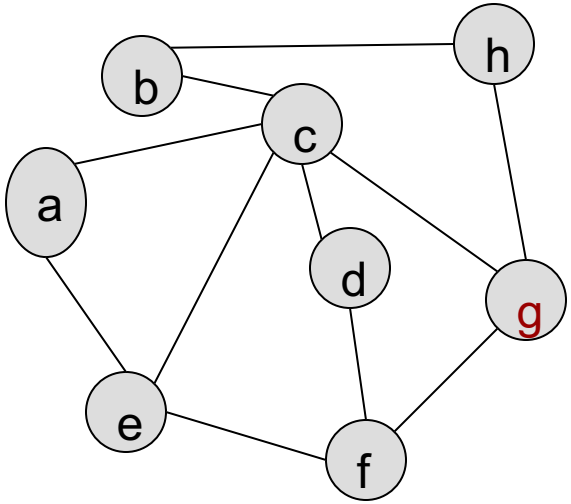
- What operations would you want to perform on a graph?
- addVertex() : Vertex
- addEdge(v1, v2)
- getAdjacencies(v1) : List<Vertices>
  - Returns any vertex with an edge from v1 to itself
- removeVertex(v)
- removeEdge(v1, v2)
- edgeExists(v1, v2) : bool

```
#include<iostream>
using namespace std;

template <typename V, typename E>
class Graph{



};
```

Perfect for templating the data associated with a vertex and edge as V and E

More Common Graph Representations

- Graphs are really just a list of lists
  - List of vertices each having their own list of adjacent vertices
- Alternatively, sometimes graphs are also represented with an adjacency matrix
  - Entry at (i,j) = 1 if there is an edge between vertex i and j, 0 otherwise



List of Vertices

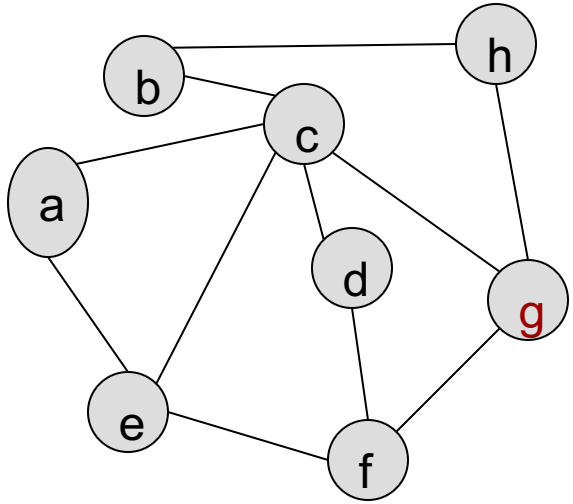
	Adjacency Lists
a	c,e
b	c,h
c	a,b,d,e,g
d	c,f
e	a,c,f
f	d,e,g
g	c,f,h
h	b,g

	a	b	c	d	e	f	g	h
a	0	0	1	0	1	0	0	0
b	0	0	1	0	0	0	0	1
c	1	1	0	1	1	0	1	0
d	0	0	1	0	0	1	0	0
e	1	0	1	0	0	1	0	0
f	0	0	0	1	1	0	1	0
g	0	0	1	0	0	1	0	1
h	0	1	0	0	0	0	1	0

Adjacency Matrix Representation

Graph Representations

- Let |V| = n = # of vertices and
  |E| = m = # of edges
- Adjacency List Representation
  - O(_____) memory storage
  - Existence of an edge requires searching adjacency list
- Adjacency Matrix Representation
  - O(_____) storage
  - Existence of an edge requires O(_____) lookup



List of Vertices

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

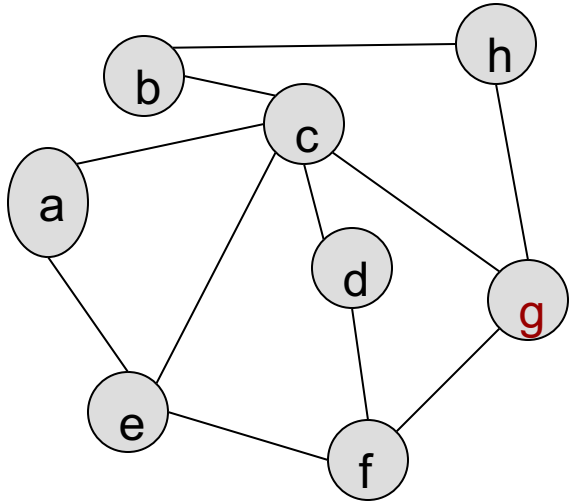
Adjacency Lists

	a	b	c	d	e	f	g	h
a	0	0	1	0	1	0	0	0
b	0	0	1	0	0	0	0	1
c	1	1	0	1	1	0	1	0
d	0	0	1	0	0	1	0	0
e	1	0	1	0	0	1	0	0
f	0	0	0	1	1	0	1	0
g	0	0	1	0	0	1	0	1
h	0	1	0	0	0	0	1	0

Adjacency Matrix Representation

Graph Representations

- Let |V| = n = # of vertices and |E| = m = # of edges
- Adjacency List Representation
  - O(|V| + |E|) memory storage
  - Existence of an edge requires searching adjacency list
  - Define **degree** to be the number of edges incident on a vertex (deg(a) = 2, deg(c) = 5, etc.
- Adjacency Matrix Representation
  - $O(|V|^2)$ storage
  - Existence of an edge requires O(1) lookup (e.g. matrix[i][j] == 1)



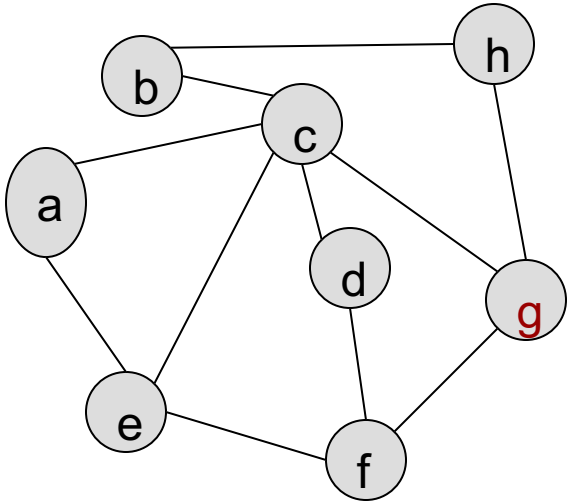
List of Vertices	Adjacency Lists
a	c,e
b	c,h
c	a,b,d,e,g
d	c,f
e	a,c,f
f	d,e,g
g	c,f,h
h	b,g

	a	b	c	d	e	f	g	h
a	0	0	1	0	1	0	0	0
b	0	0	1	0	0	0	0	1
c	1	1	0	1	1	0	1	0
d	0	0	1	0	0	1	0	0
e	1	0	1	0	0	1	0	0
f	0	0	0	1	1	0	1	0
g	0	0	1	0	0	1	0	1
h	0	1	0	0	0	0	1	0

Adjacency Matrix Representation

Graph Representations

- Can 'a' get to 'b' in two hops?
- Adjacency List
  - For each neighbor of a…
  - Search that neighbor's list for b
- Adjacency Matrix
  - Take the dot product of row a & column b



List of Vertices

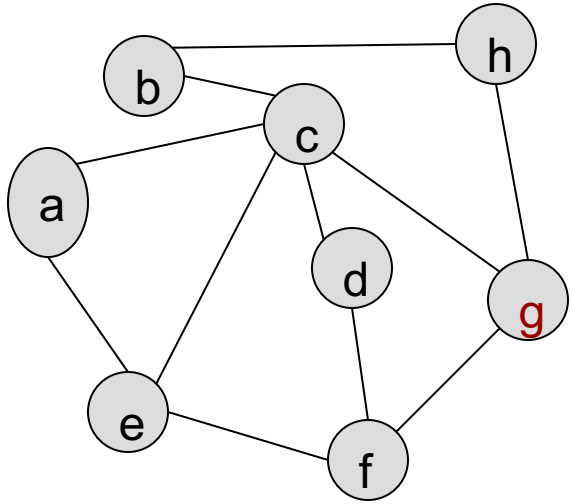
	Adjacency Lists
a	c,e
b	c,h
c	a,b,d,e,g
d	c,f
e	a,c,f
f	d,e,g
g	c,f,h
h	b,g

	a	b	c	d	e	f	g	h
a	0	0	1	0	1	0	0	0
b	0	0	1	0	0	0	0	1
c	1	1	0	1	1	0	1	0
d	0	0	1	0	0	1	0	0
e	1	0	1	0	0	1	0	0
f	0	0	0	1	1	0	1	0
g	0	0	1	0	0	1	0	1
h	0	1	0	0	0	0	1	0

Adjacency Matrix Representation

Graph Representations

- Can 'a' get to 'b' in two hops?
- Adjacency List
  - For each neighbor of a…
  - Search that neighbor's list for b
- Adjacency Matrix
  - Take the dot product of row a & column b



List of Vertices

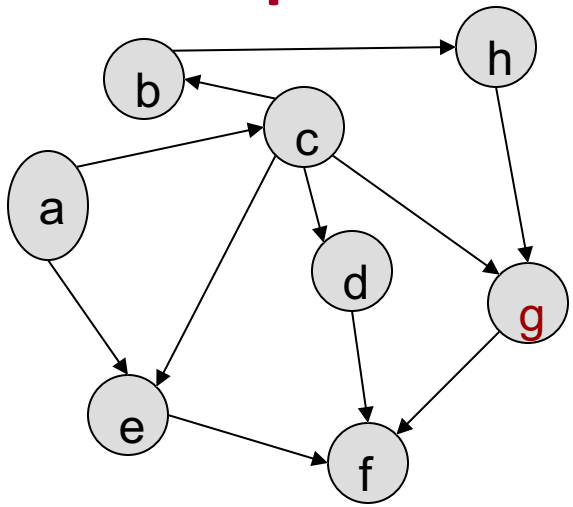
	Adjacency Lists
a	c,e
b	c,h
c	a,b,d,e,g
d	c,f
e	a,c,f
f	d,e,g
g	c,f,h
h	b,g

	a	b	c	d	e	f	g	h
a	0	0	1	0	1	0	0	0
b	0	0	1	0	0	0	0	1
c	1	1	0	1	1	0	1	0
d	0	0	1	0	0	1	0	0
e	1	0	1	0	0	1	0	0
f	0	0	0	1	1	0	1	0
g	0	0	1	0	0	1	0	1
h	0	1	0	0	0	0	1	0

Adjacency Matrix Representation

Directed vs. Undirected Graphs

- In the previous graphs, edges were **undirected** (meaning edges are 'bidirectional' or 'reflexive')
  - An edge (u,v) implies (v,u)
- In **directed** graphs, links are unidirectional
  - An edge (u,v) does not imply (v,u)
  - For Edge (u,v): the **source** is u, **target** is v
- For adjacency list form, you may need 2 lists per vertex for both predecessors and successors



List of Vertices	Outgoing Edges
a	c,e
b	h
c	b,d,e,g
d	f
e	f
f	
g	f
h	g

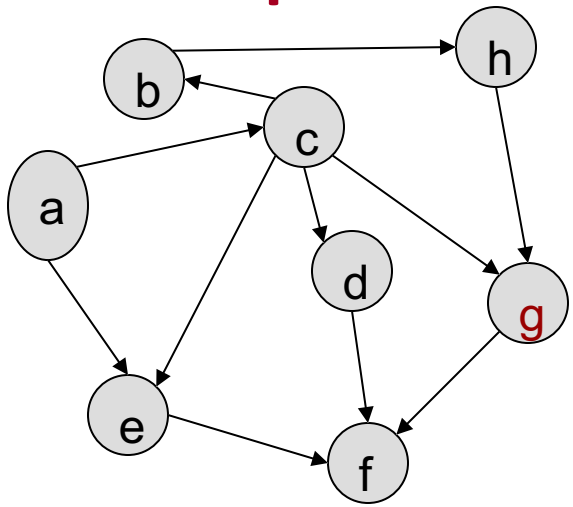
Target

Source	a	b	c	d	e	f	g	h
a	0	0	1	0	1	0	0	0
b	0	0	0	0	0	0	0	1
c	0	1	0	1	1	0	1	0
d	0	0	0	0	0	1	0	0
e	0	0	0	0	0	1	0	0
f	0	0	0	0	0	0	0	0
g	0	0	0	0	0	1	0	0
h	0	0	0	0	0	0	1	0

Adjacency Matrix Representation

Directed vs. Undirected Graphs

- In directed graph with edge (u,v) we define
  - Successor(u) = v
  - Predecessor(v) = u
- Using an adjacency list representation *may* warrant two lists predecessors and successors



List of Vertices	Succs	Preds
a	c,e	
b	h	c
c	b,d,e,g	a
d	f	c
e	f	a,c
f		d, e, g
g	f	c,h
h	g	b

Target

Source	a	b	c	d	e	f	g	h
a	0	0	1	0	1	0	0	0
b	0	0	0	0	0	0	0	1
c	0	1	0	1	1	0	1	0
d	0	0	0	0	0	1	0	0
e	0	0	0	0	0	1	0	0
f	0	0	0	0	0	0	0	0
g	0	0	0	0	0	1	0	0
h	0	0	0	0	0	0	1	0

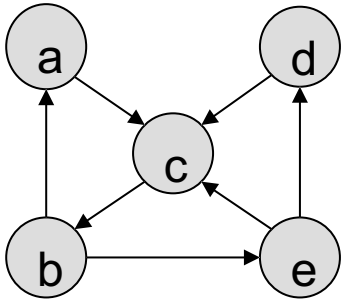
Adjacency Matrix Representation

Timeout: Real-world example

PAGERANK ALGORITHM

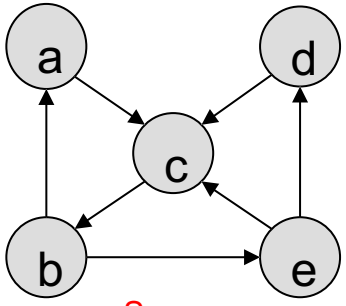
PageRank

- Consider the graph at the right
  - These could be webpages with links shown in the corresponding direction
  - These could be neighboring cities
- PageRank generally tries to answer the question:
  - **If we let a bunch of people randomly "walk" the graph, what is the probability that they end up at a certain location (page, city, etc.) in the "steady-state"**
- We could solve this problem through Monte-Carlo simulation (similar to CS 103 Coin-flipping game assignment)
  - Simulate a large number of random walkers and record where each one ends to build up an answer of the probabilities for each vertex
- But there are more efficient ways of doing it



PageRank

- Let us write out the adjacency matrix for this graph
- Now let us make a weighted version by normalizing based on the out-degree of each node
  - Ex. If you're at node B we have a 50-50 chance of going to A or E
- From this you could write a system of linear equations (i.e. what are the chances you end up at vertex I at the next time step, given you are at some vertex J now
  - pA = 0.5*pB
  - pB = pC
  - pC = pA + pD + 0.5*pE
  - pD = 0.5*pE
  - pE = 0.5*pB
  - We also know: pA + pB + pC + pD + pE = 1



Source

Target

	a	b	c	d	e
a	0	1	0	0	0
b	0	0	1	0	0
c	1	0	0	1	1
d	0	0	0	0	1
e	0	1	0	0	0

Adjacency Matrix

Source=j

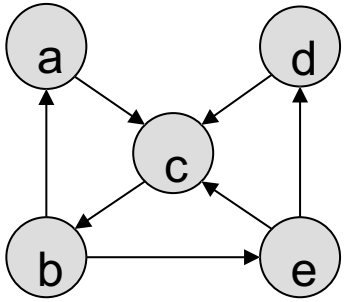
Target=i

	a	b	c	d	e
a	0	0.5	0	0	0
b	0	0	1	0	0
c	1	0	0	1	0.5
d	0	0	0	0	0.5
e	0	0.5	0	0	0

Weighted Adjacency Matrix
[Divide $(a_{i,j})$/degree(j)]

PageRank

- System of Linear Equations
  - pA = 0.5*pB
  - pB = pC
  - pC = pA + pD + 0.5*pE
  - pD = 0.5*pE
  - pE = 0.5*pB
  - We also know: pA + pB + pC + pD + pE = 1
- If you know something about linear algebra, you know we can write these equations in matrix form as a linear system
  - Ax = y



Source=j

	a	b	c	d	e
a	0	0.5	0	0	0
b	0	0	1	0	0
c	1	0	0	1	0.5
d	0	0	0	0	0.5
e	0	0.5	0	0	0

Target=i

Weighted Adjacency Matrix
[Divide by $(a_{i,j})$/degree(j)]

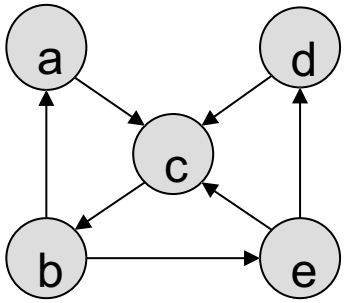
$$\begin{vmatrix} 0 & 0.5 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0.5 \\ 0 & 0 & 0 & 0 & 0.5 \\ 0 & 0.5 & 0 & 0 & 0 \end{vmatrix} * \begin{vmatrix} pA \\ pB \\ pC \\ pD \\ pE \end{vmatrix}$$

$$\begin{vmatrix} 0 & 0.5 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0.5 \\ 0 & 0 & 0 & 0 & 0.5 \\ 0 & 0.5 & 0 & 0 & 0 \end{vmatrix} * \begin{vmatrix} pA \\ pB \\ pC \\ pD \\ pE \end{vmatrix} = \begin{vmatrix} pA = 0.5PB \\ pB = pC \\ pC = pA+pD+0.5*pE \\ pD = 0.5*pE \\ pE = 0.5*pB \end{vmatrix}$$

PageRank

- But remember we want the steady state solution
  - The solution where the probabilities don't change from one step to the next
- So we want a solution to: **A**p = p
- We can:
  - Use a linear system solver (Gaussian elimination)
  - Or we can just seed the problem with some probabilities and then just iterate until the solution settles down

$$\begin{vmatrix} 0 & 0.5 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0.5 \\ 0 & 0 & 0 & 0 & 0.5 \\ 0 & 0.5 & 0 & 0 & 0 \end{vmatrix} * \begin{vmatrix} pA \\ pB \\ pC \\ pD \\ pE \end{vmatrix} = \begin{vmatrix} pA \\ pB \\ pC \\ pD \\ pE \end{vmatrix}$$



Source=j

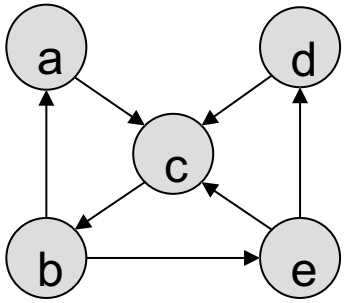
Target=i

	a	b	c	d	e
a	0	0.5	0	0	0
b	0	0	1	0	0
c	1	0	0	1	0.5
d	0	0	0	0	0.5
e	0	0.5	0	0	0

Weighted Adjacency Matrix
[Divide by $(a_{i,j})$/degree(j)]

Iterative PageRank

- But remember we want the steady state solution
  - The solution where the probabilities don't change from one step to the next
- So we want a solution to: **A**p = p
- We can:
  - Use a linear system solver (Gaussian elimination)
  - Or we can just seed the problem with some probabilities and then just iterate until the solution settles down



Step 0 Sol. Step 1 Sol.

$$\begin{vmatrix} 0 & 0.5 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0.5 \\ 0 & 0 & 0 & 0 & 0.5 \\ 0 & 0.5 & 0 & 0 & 0 \end{vmatrix} * \begin{vmatrix} .2 \\ .2 \\ .2 \\ .2 \\ .2 \end{vmatrix} = \begin{vmatrix} .1 \\ .2 \\ .5 \\ .1 \\ .1 \end{vmatrix}$$

Step 1 Sol. Step 2 Sol.

$$\begin{vmatrix} 0 & 0.5 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0.5 \\ 0 & 0 & 0 & 0 & 0.5 \\ 0 & 0.5 & 0 & 0 & 0 \end{vmatrix} * \begin{vmatrix} .1 \\ .2 \\ .5 \\ .1 \\ .1 \end{vmatrix} = \begin{vmatrix} .1 \\ .5 \\ .25 \\ .05 \\ .1 \end{vmatrix}$$

Step 29 Sol. Step 30 Sol.

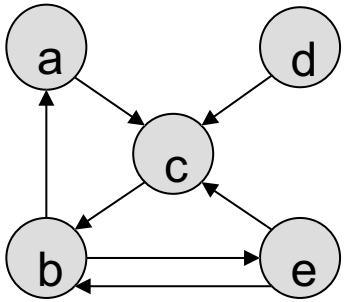
$$\begin{vmatrix} 0 & 0.5 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0.5 \\ 0 & 0 & 0 & 0 & 0.5 \\ 0 & 0.5 & 0 & 0 & 0 \end{vmatrix} * \begin{vmatrix} ? \\ ? \\ ? \\ ? \\ ? \end{vmatrix} = \begin{vmatrix} .1507 \\ .3078 \\ .3126 \\ .0783 \\ .1507 \end{vmatrix}$$

Actual PageRank Solution from solving linear system:

$$\begin{vmatrix} .1538 \\ .3077 \\ .3077 \\ .0769 \\ .1538 \end{vmatrix}$$

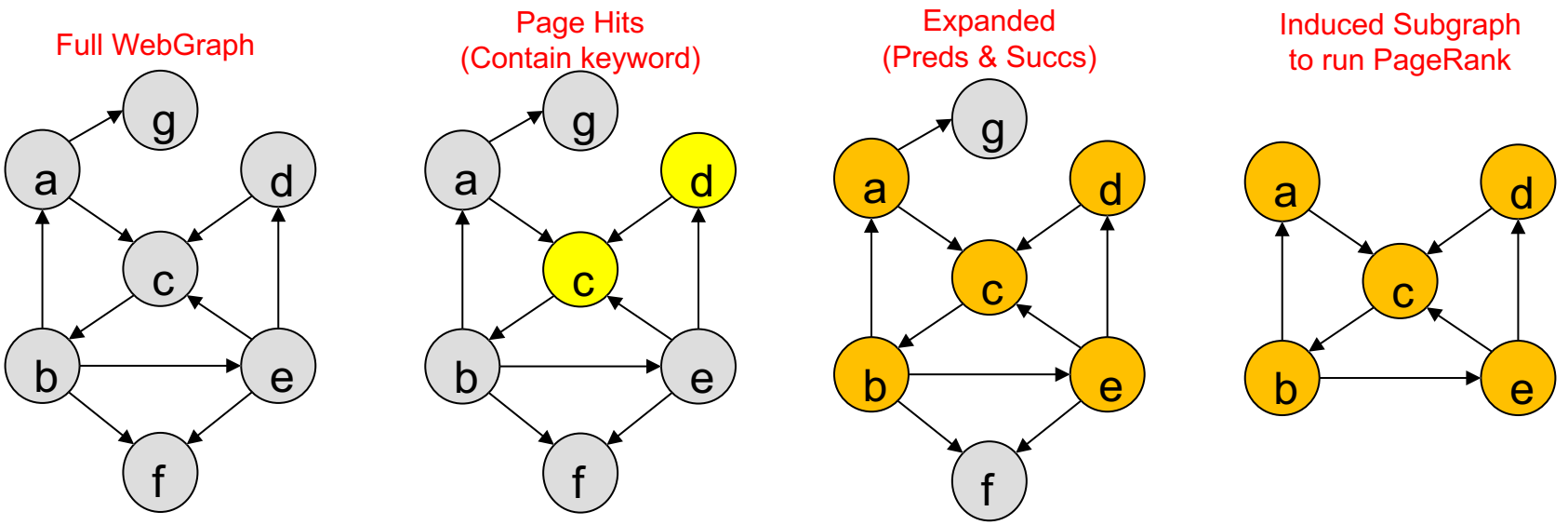
Additional Notes

- What if we change the graph and now D has no incoming links…what is its PageRank?
  - 0
- Most PR algorithms add a probability that someone just enters that URL (i.e. enters the graph at that node)
  - Usually define something called the damping factor, α (often chosen around 0.85)
  - Probability of randomly starting or jumping somewhere = 1-α
- So at each time step the next PR value for node i is given as:
  - $\Pr(i) = \frac{1-\alpha}{N} + \alpha * \sum_{j \in Pred(i)} \frac{\Pr(j)}{OutDeg(j)}$
  - N is the total number of vertices
  - Usually run 30 or so update steps
  - Start each Pr(i) = 1/N



In a Web Search Setting

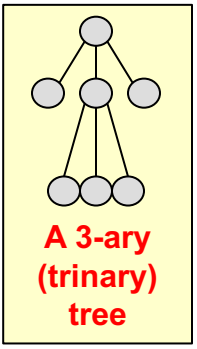
- Given some search keywords we could find the pages that have that matching keywords
- We often expand that set of pages by including all successors and predecessors of those pages
  - Include all pages that are within a radius of 1 of the pages that actually have the keyword
- Now consider that set of pages and the subgraph that it induces
- Run PageRank on that subgraph

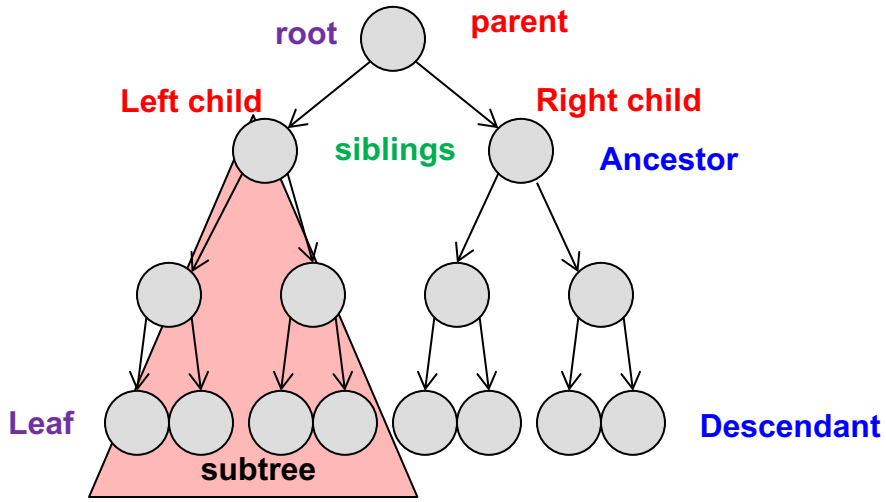


TREES

Tree Definitions – Part 1

- Definition: A connected, acyclic (no cycles) graph with:
  - A root node, r, that has 0 or more subtrees
  - Exactly one path between any two nodes
- In general:
  - Nodes have exactly one parent (except for the root which has none) and 0 or more children
- d-ary tree
  - Tree where each node has at most d children
  - Binary tree = d-ary Tree with n=2



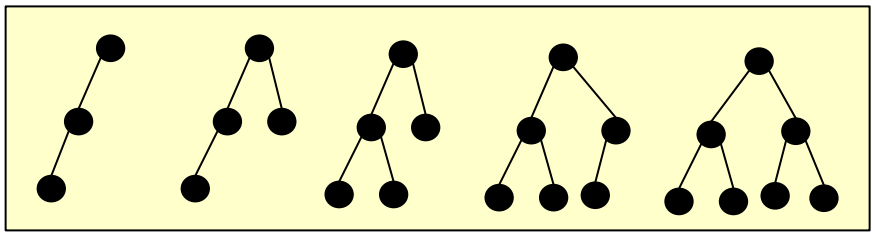


Terms:

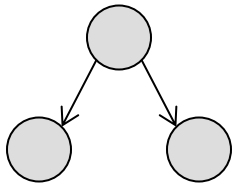
- **Parent(i)**: Node directly above node i
- **Child(i)**: Node directly below node i
- **Siblings**: Children of the same parent
- **Root:** Only node with no parent
- **Leaf:** Node with 0 children
- **Height:** Length of largest path from root to any leaf
- **Subtree(n)**: Tree rooted at node n
- **Ancestor(n)**: Any node on the path from n to the root
- **Descendant(n):** Any node in the subtree rooted at n

Tree Definitions – Part 2

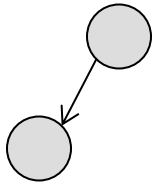
- Tree height: maximum # of nodes on a path from root to any leaf
- Full d-ary tree, T, where
  - Every vertex has 0 or d children and all leaf nodes are at the same level
  - If height h>1 and both subtrees are full binary trees of height, hnil
  - If height h==1, then it is full by definition
- Complete d-ary tree
  - Each level is filled left-to-right and a new level is not started until the previous one is complete
- Balanced d-ary tree
  - Tree where subtrees from any node differ in height by at most 1



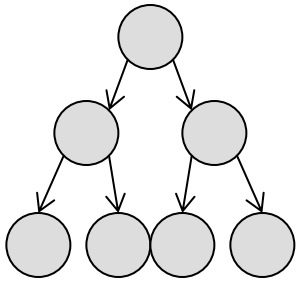
DAPS, 6th Ed. Figure 15-8



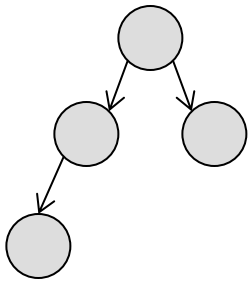
Full



Complete, but not full



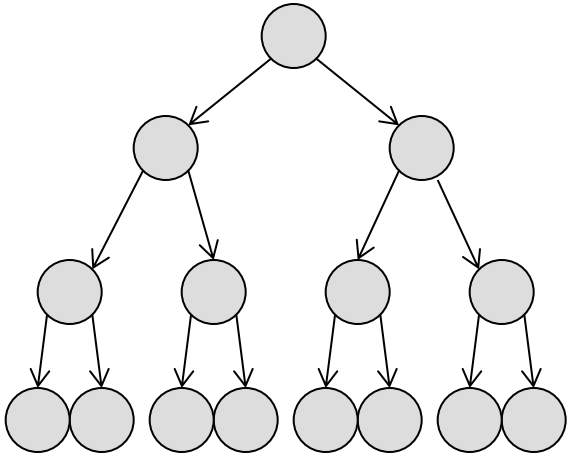
Full



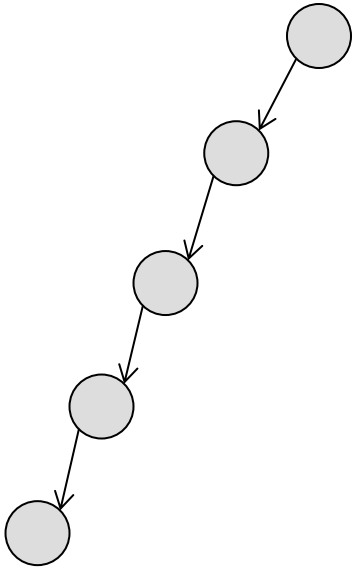
Complete

Tree Height

- A full binary tree of n nodes has height, $\lceil log_2(n+1) \rceil$
  - This implies the minimum height of any tree with n nodes is $\lceil log_2(n+1) \rceil$
- The maximum height of a tree with n nodes is, n



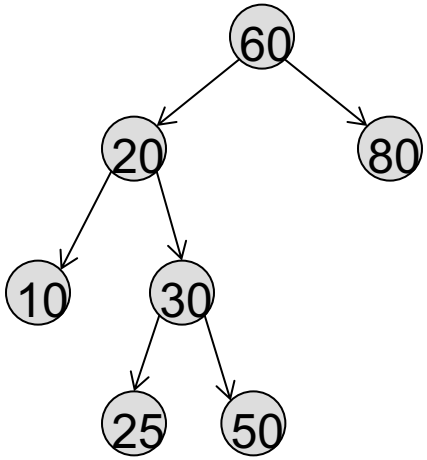
15 nodes => height $\log_2(16) = 4$



5 nodes => height = 5

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]




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

60 20 10 30 25 50 80

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

10 20 25 30 50 60 80

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

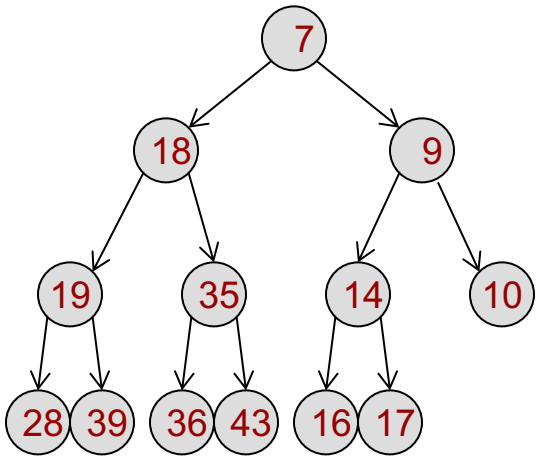
10 25 50 30 20 80 60

Array-based and Link-based

TREE IMPLEMENTATIONS

Array-Based Complete Binary Tree

- Binary tree that is complete (i.e. only the lowest-level contains empty locations and items added left to right) can be stored nicely in an array (let's say it starts at index 1 and index 0 is empty)
- Can you find the mathematical relation for finding the index of node i's parent, left, and right child?
  - Parent(i) = _____
  - Left_child(i) = _____
  - Right_child(i) = _____

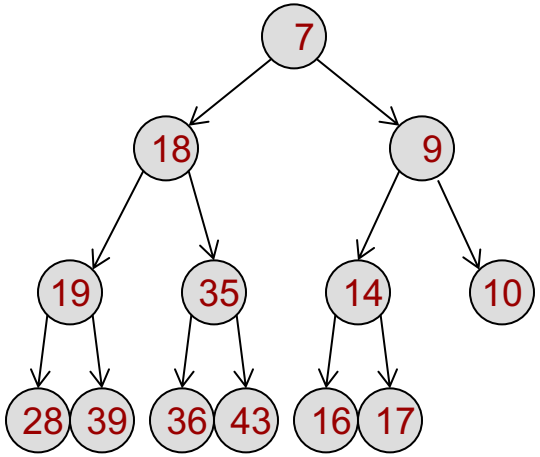


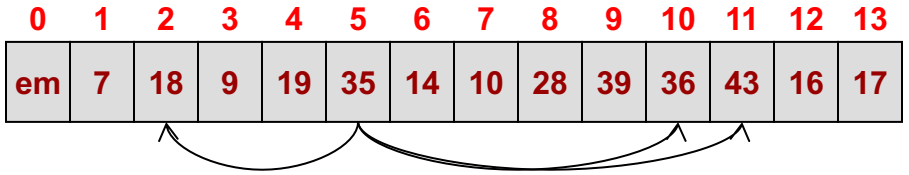
0	1	2	3	4	5	6	7	8	9	10	11	12	13
em	7	18	9	19	35	14	10	28	39	36	43	16	17

parent(5) = _____
Left_child(5) = _____
Right_child(5) = _____

Array-Based Complete Binary Tree

- Binary tree that is complete (i.e. only the lowest-level contains empty locations and items added left to right) can be stored nicely in an array (let's say it starts at index 1 and index 0 is empty)
- Can you find the mathematical relation for finding node i's parent, left, and right child?
  - Parent(i) = i/2
  - Left_child(i) = 2*i
  - Right_child(i) = 2*i + 1



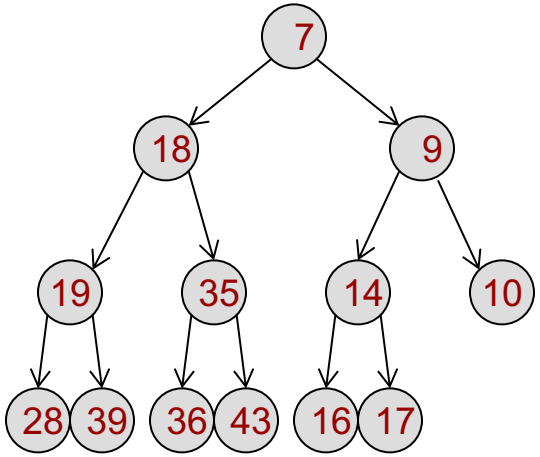


parent(5) = 5/2 = 2
Left_child(5) = 2*5 = 10
Right_child(5) = 2*5+1 = 11

Non-complete binary trees require much more bookeeping to store in arrays…usually link-based approaches are preferred

0-Based Indexing

- Binary tree that is complete (i.e. only the lowest-level contains empty locations and items added left to right) can be stored nicely in an array (let's say it starts at index 1 and index 0 is empty)
- Can you find the mathematical relation for finding the index of node i's parent, left, and right child?
  - Parent(i) = _____
  - Left_child(i) = _____
  - Right_child(i) = _____

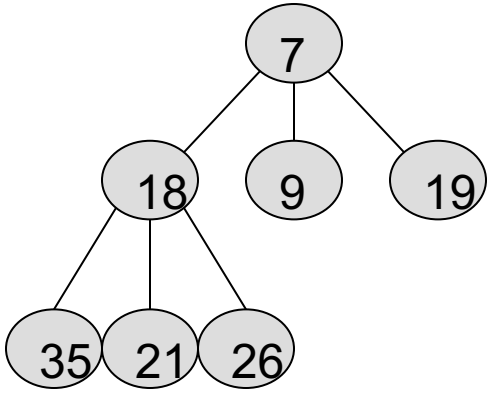


0	1	2	3	4	5	6	7	8	9	10	11	12
7	18	9	19	35	14	10	28	39	36	43	16	17

parent(5) = _____
Left_child(5) = _____
Right_child(5) = _____

D-ary Array-based Implementations

- Arrays can be used to store d-ary **complete** trees
  - Adjust the formulas derived for binary trees in previous slides in terms of **d**



A 3-ary (trinary) tree

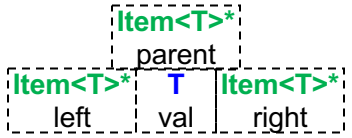
0	1	2	3	4	5	6
7	18	9	19	35	21	26

Link-Based Approaches

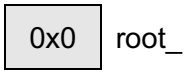
- Much like a linked list but now with two pointers per Item
- Use NULL pointers to indicate no child
- Dynamically allocate and free items when you add/remove them

```
#include<iostream>
using namespace std;

template <typename T>
struct BTItem {
  T val;
  BTItem<T>* left,right;
  BTItem<T>* parent;
};
// Bin. Search Tree
template <typename T>
class BinTree
{
 public:
 BinTree();
 ~BinTree();
 void add(const T& v);
 ...
 private:
 BTItem<T>* root_;
};
```

BTItem<T> blueprint:



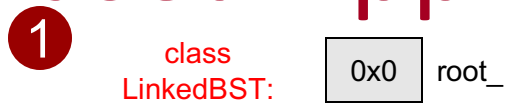
class LinkedBST:

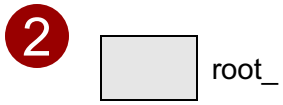


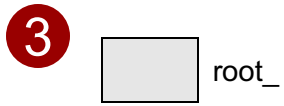
root_

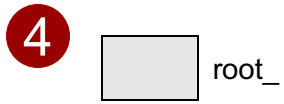
Link-Based Approaches

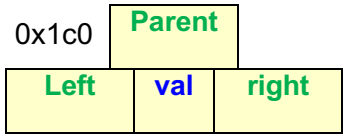
- Add(5)
- Add(6)
- Add(7)





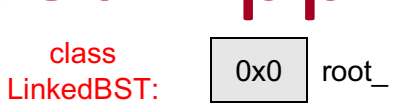


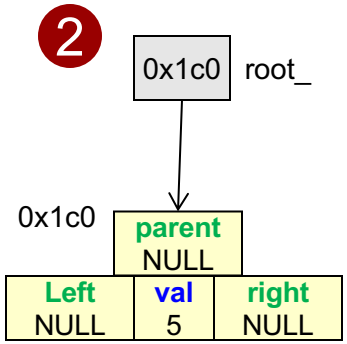


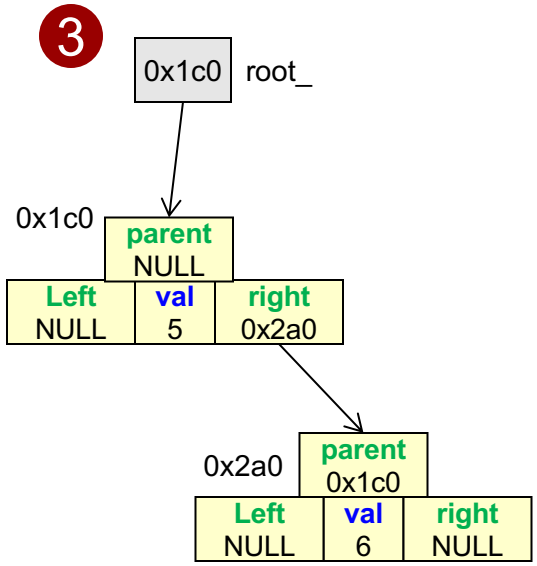


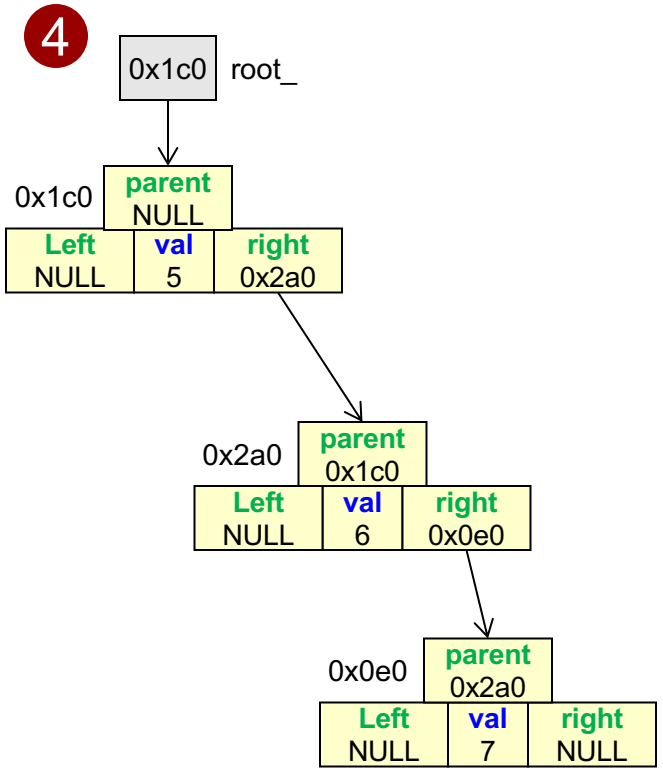
Link-Based Approaches

- Add(5)
- Add(6)
- Add(7)





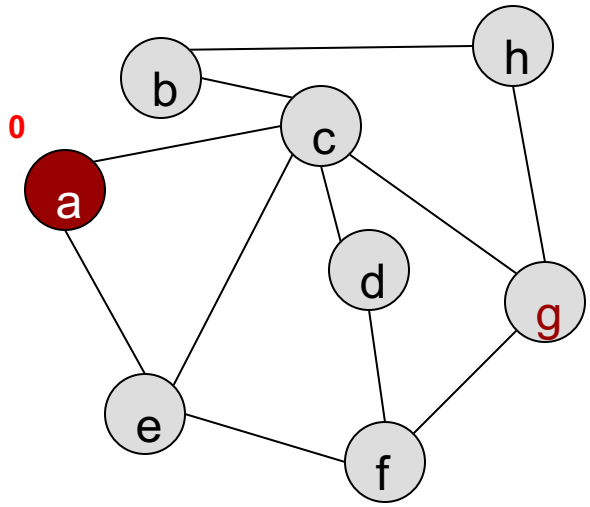




BREADTH-FIRST SEARCH

Breadth-First Search

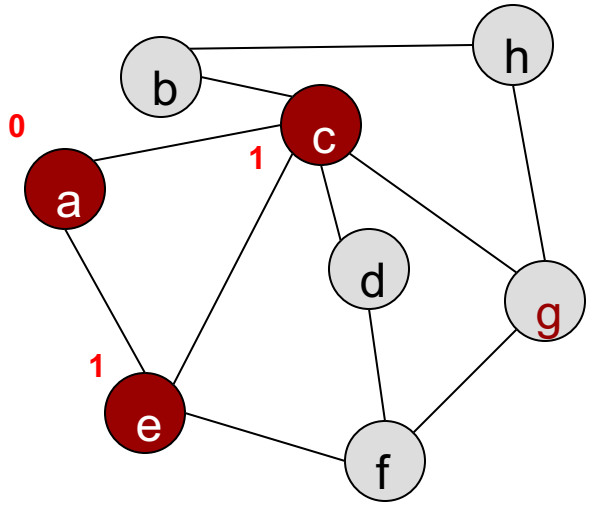
- Given a graph with vertices, V, and edges, E, and a starting vertex that we'll call u
- BFS starts at u (‘a’ in the diagram to the left) and fans-out along the edges to nearest neighbors, then to their neighbors and so on
- Goal: Find shortest paths (a.k.a. minimum number of hops or depth) from the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



Depth 0: a

Breadth-First Search

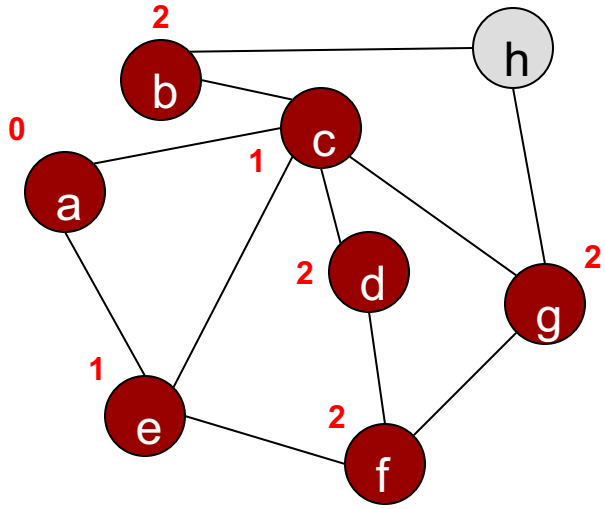
- Given a graph with vertices, V, and edges, E, and a starting vertex, u
- BFS starts at u (‘a’ in the diagram to the left) and fans-out along the edges to nearest neighbors, then to their neighbors and so on
- Goal: Find shortest paths (a.k.a. minimum number of hops or depth) from the start vertex to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



Depth 0: a
Depth 1: c,e

Breadth-First Search

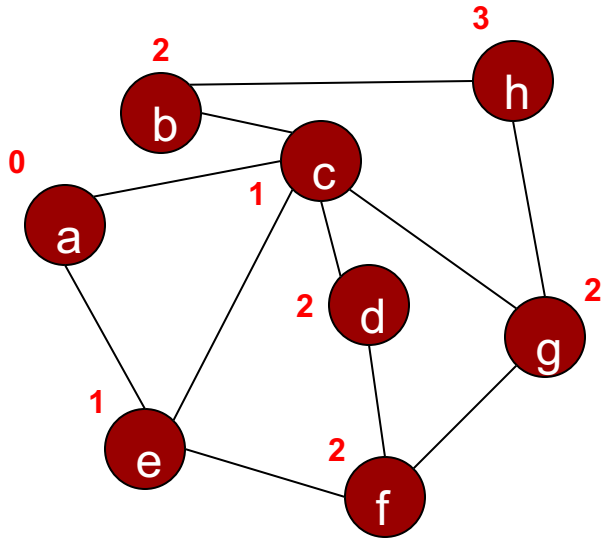
- Given a graph with vertices, V, and edges, E, and a starting vertex, u
- BFS starts at u (‘a’ in the diagram to the left) and fans-out along the edges to nearest neighbors, then to their neighbors and so on
- Goal: Find shortest paths (a.k.a. minimum number of hops or depth) from the start vertex to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



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

Breadth-First Search

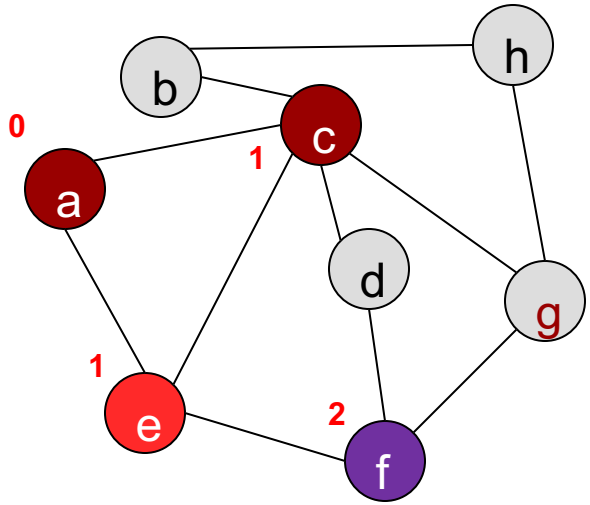
- Given a graph with vertices, V, and edges, E, and a starting vertex, u
- BFS starts at u (‘a’ in the diagram to the left) and fans-out along the edges to nearest neighbors, then to their neighbors and so on
- Goal: Find shortest paths (a.k.a. minimum number of hops or depth) from the start vertex to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



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

Developing the Algorithm

- Key idea: Must explore all nearer neighbors before exploring further-away neighbors
- From ‘a’ we find ‘e’ and ‘c’
  - Computer can only do one thing at a time so we have to pick either e or c to explore from
  - Let’s say we pick e…we will find f
  - Now what vertex should we explore (i.e. visit neighbors) next?
  - C!! (if we don’t we won’t find shortest paths…e.g. d)
  - Must explore all vertices at depth i before any vertices at depth i+1



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

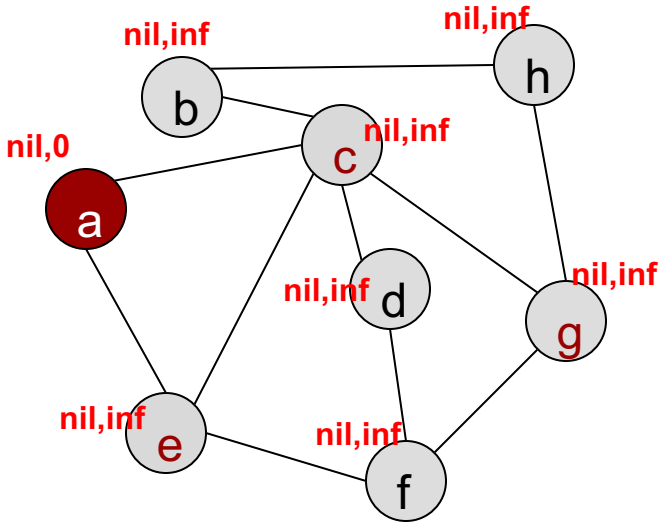
Developing the Algorithm

- Exploring all vertices in the order they are found implies we will explore all vertices at shallower depth before greater depth
  - Keep a first-in / first-out queue (FIFO) of neighbors found
- 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 allow unmarked vertices to be put in the queue
- May also keep a 'predecessor' structure that indicates how each vertex got discovered (i.e. which vertex caused this one to be found)
  - Allows us to find a shortest-path back to the start vertex

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] < 0 // w not found
9           Q.enqueue(w)
10          pred[w] = v,  d[w] = d[v] + 1
```



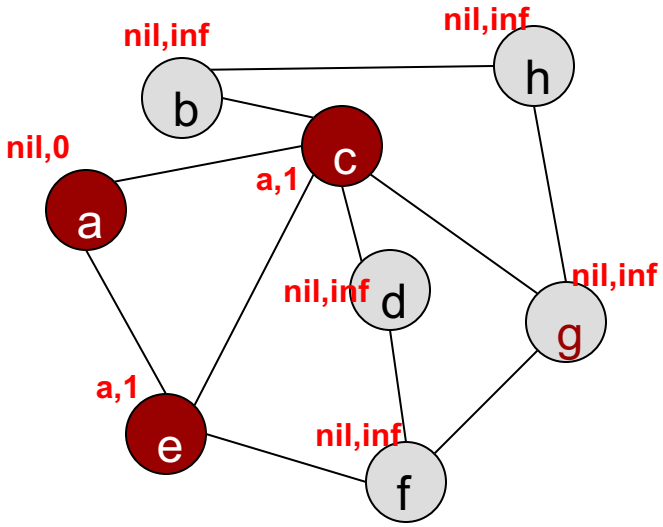
Q:

a

Breadth-First Search

Algorithm:

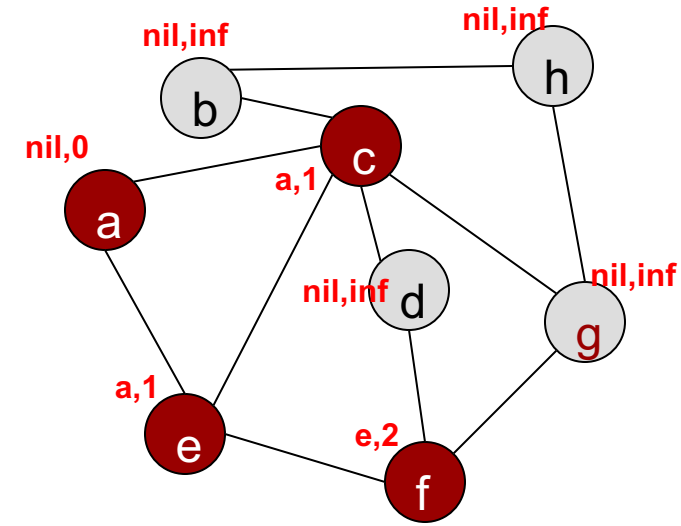
```
BFS(G,u)
1  for each vertex v
2      pred[v] = nil, d[v] = inf.
3  Q = new Queue
4  Q.enqueue(u),  d[u]=0
5  while Q is not empty
6     v = Q.front(); Q.dequeue()
7     foreach neighbor, w, of v:
8        if pred[w] < 0 // w not found
9           Q.enqueue(w)
10          pred[w] = v,  d[w] = d[v]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



Breadth-First Search

Algorithm:

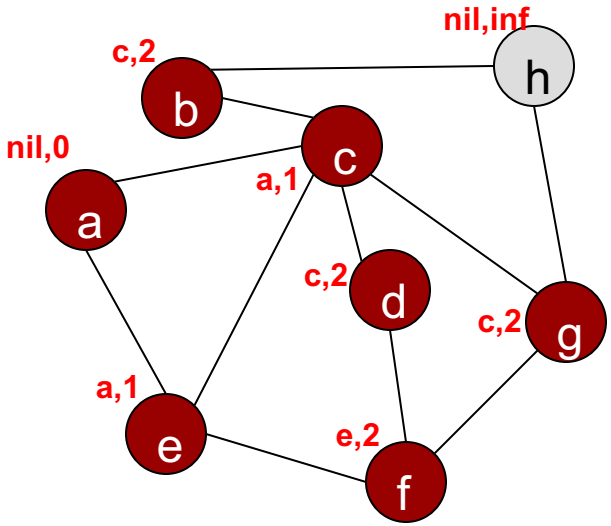
```
BFS(G,u)
1  for each vertex v
2      pred[v] = nil, d[v] = inf.
3  Q = new Queue
4  Q.enqueue(u),  d[u]=0
5  while Q is not empty
6     v = Q.front(); Q.dequeue()
7     foreach neighbor, w, of v:
8        if pred[w] < 0 // w not found
9           Q.enqueue(w)
10          pred[w] = v,  d[w] = d[v]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



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] < 0 // w not found
9           Q.enqueue(w)
10          pred[w] = v,  d[w] = d[v] + 1
```



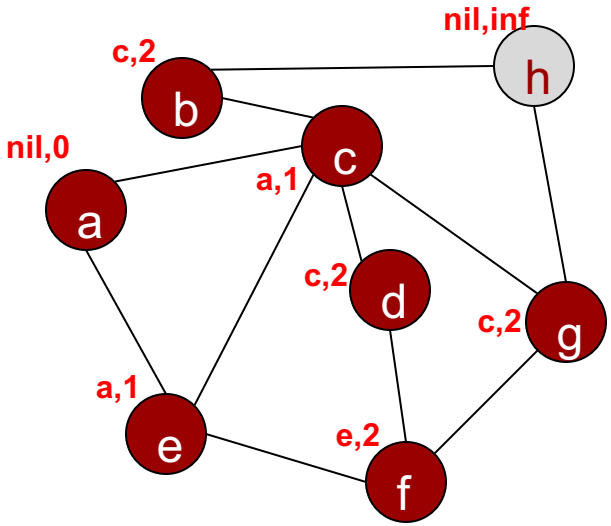
v = c

Q: f b d g

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] < 0 // w not found
9           Q.enqueue(w)
10          pred[w] = v,  d[w] = d[v] + 1
```



v = f

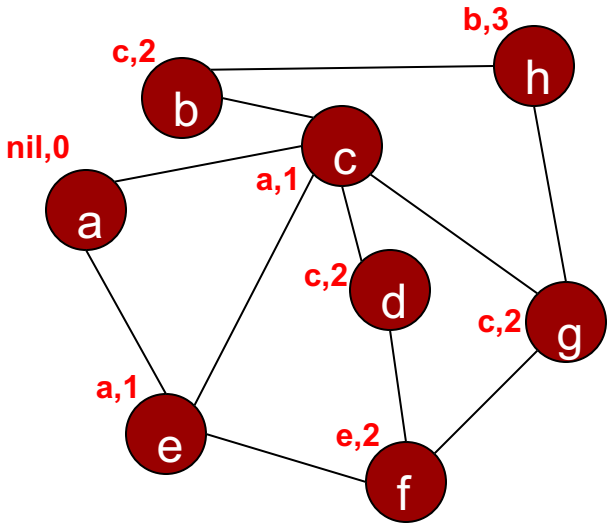
Q:

b	d	g

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] < 0 // w not found
9           Q.enqueue(w)
10          pred[w] = v,  d[w] = d[v] + 1
```



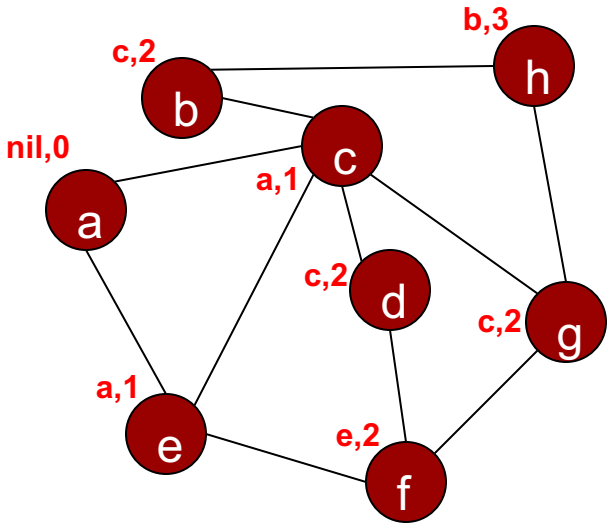
v = b

Q: d g h

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] < 0 // w not found
9           Q.enqueue(w)
10          pred[w] = v,  d[w] = d[v] + 1
```



v = d

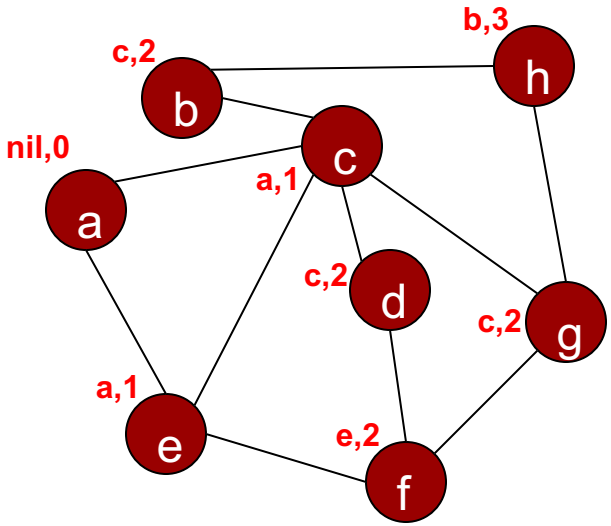
Q:

g	h

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] < 0 // w not found
9           Q.enqueue(w)
10          pred[w] = v,  d[w] = d[v] + 1
```



v = g

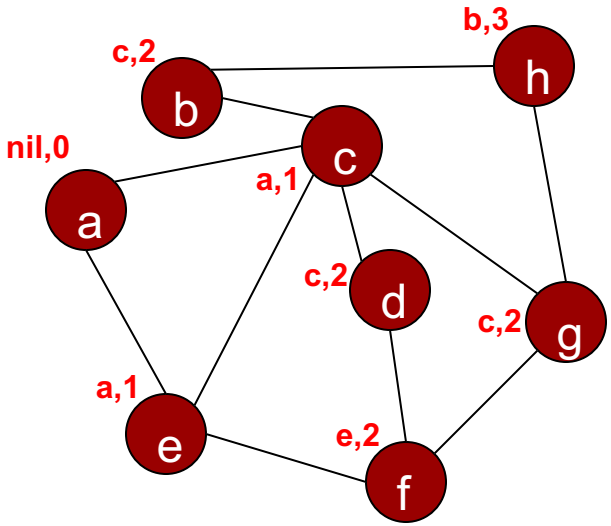
Q:

h

Breadth-First Search

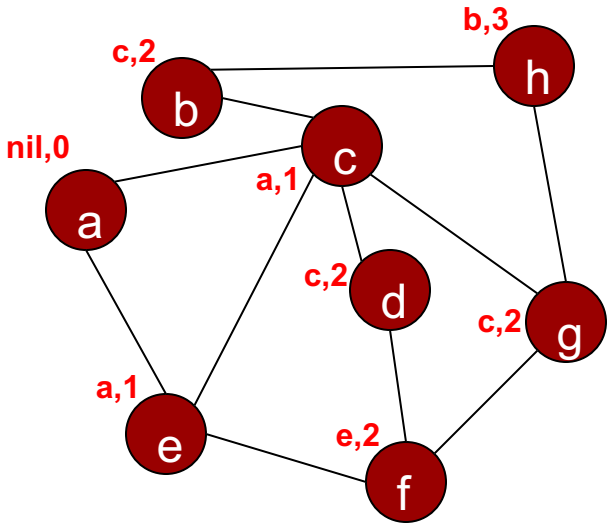
Algorithm:

```
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] < 0 // w not found
9           Q.enqueue(w)
10          pred[w] = v,  d[w] = d[v] + 1
```



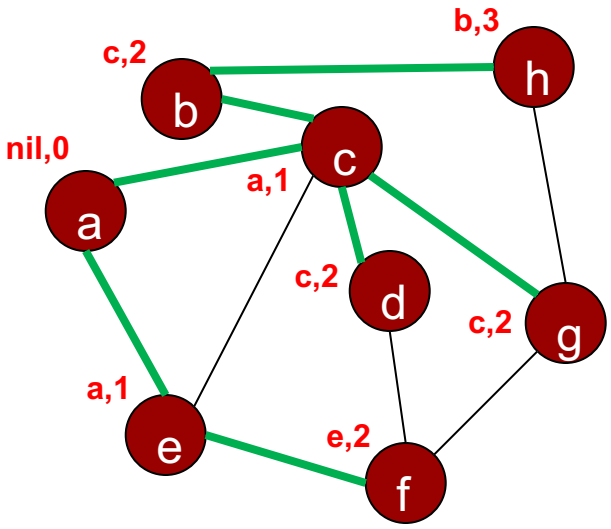
Breadth-First Search

- Shortest paths can be found by walking predecessor value from any node backward
- Example:
  - Shortest path from a to h
  - Start at h
  - Pred[h] = b (so walk back to b)
  - Pred[b] = c (so walk back to c)
  - Pred[c] = a (so walk back to a)
  - Pred[a] = nil … no predecessor, Done!!

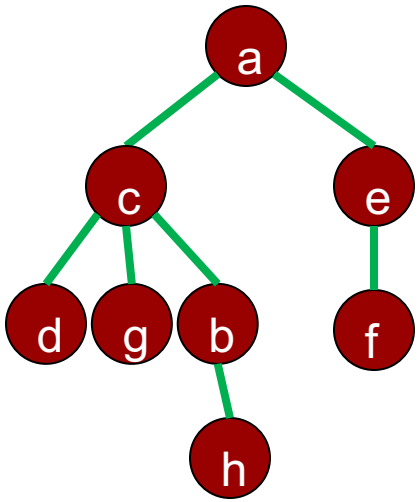


Breadth-First Search Trees

- BFS (and later DFS) will induce a tree subgraph (i.e. acyclic, one parent each) from the original graph
  - Really BFS finds a subset of edges that form the shortest paths from the source to all other vertices and this subset forms a tree



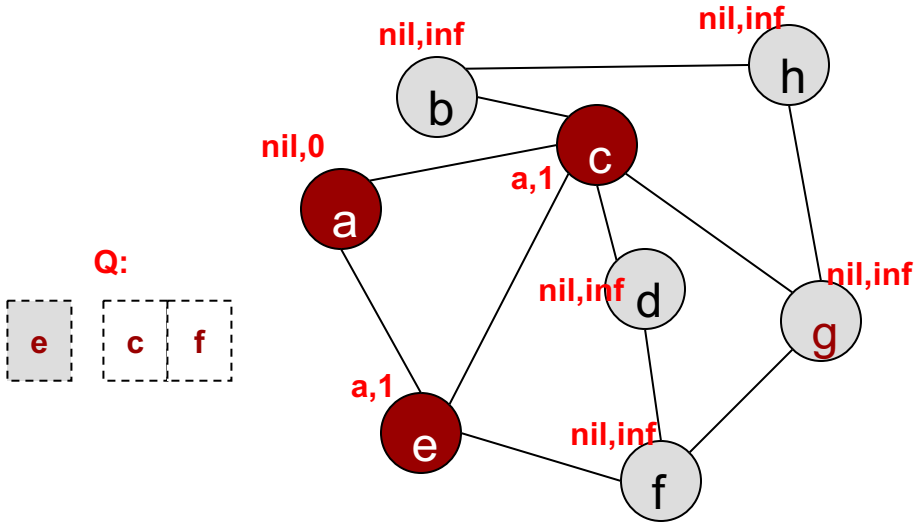
Original graph, G



BFS Induced Tree

Correctness

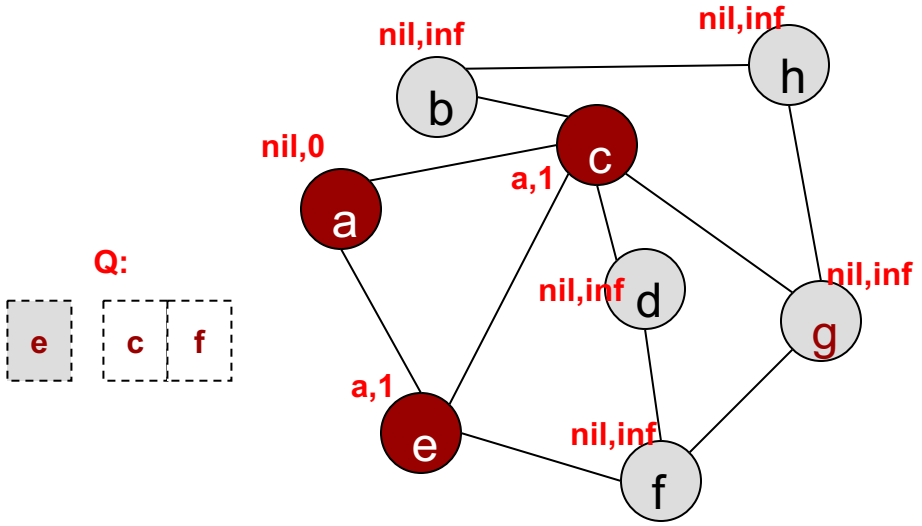
- Define
  - dist(s,v) = correct shortest distance
  - d[v] = BFS computed distance
  - p[v] = predecessor of v
- Loop invariant
  - What can we say about the nodes in the queue, their d[v] values, relationship between d[v] and dist[v], etc.?




```
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] < 0 // w not found
9           Q.enqueue(w)
10          pred[w] = v,  d[w] = d[v] + 1
```

Correctness

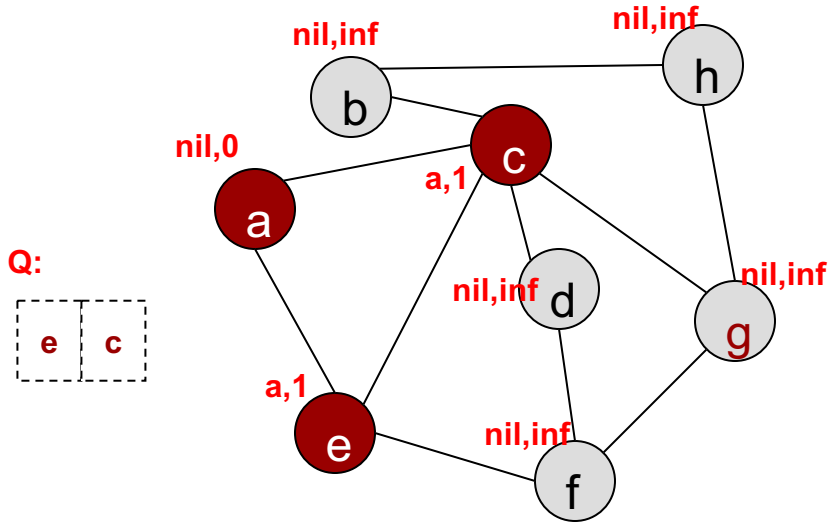
- Define
  - dist(s,v) = correct shortest distance
  - d[v] = BFS computed distance
  - p[v] = predecessor of v
- Loop invariant
  - All vertices with p[v] != nil (i.e. already in the queue or popped from queue) have d[v] = dist(s,v)
  - The distance of the nodes in the queue are sorted
    - If Q = {v_1, v_2, …, v_r} then d[v_1] <= d[v_2] <= … <= d[v_r]
  - The nodes in the queue are from 2 adjacent layers/levels
    - i.e. d[v_k] <= d[v_1] + 1
    - Suppose there is a node from a 3rd level (d[v_1] + 2), it must have been found by some, vi, where d[v_i] = d[v_1]+1




```
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] < 0 // w not found
9           Q.enqueue(w)
10          pred[w] = v,  d[w] = d[v] + 1
```

Breadth-First Search

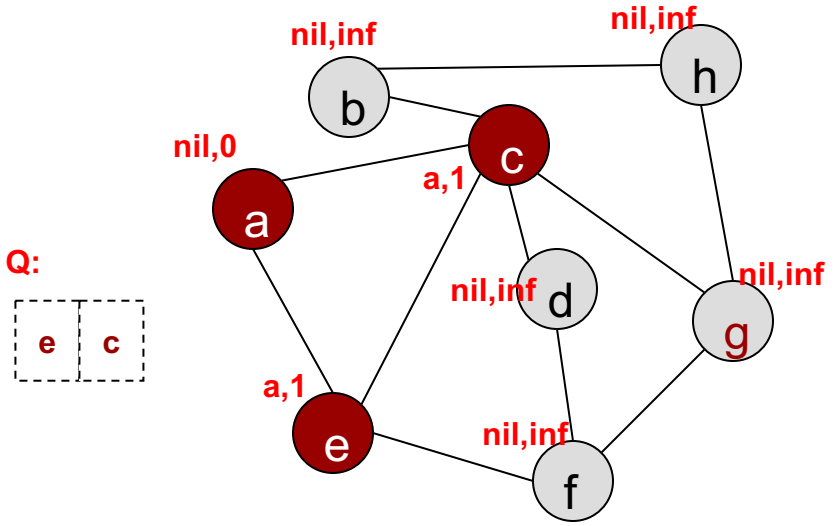
- Analyze the run time of BFS for a graph with n vertices and m edges
  - Find T(n,m)
- How many times does loop on line 5 iterate?
- How many times loop on line 7 iterate?




```
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] < 0 // w not found
9           Q.enqueue(w)
10          pred[w] = v,  d[w] = d[v] + 1
```

Breadth-First Search

- Analyze the run time of BFS for a graph with n vertices and m edges
  - Find T(n)
- How many times does loop on line 5 iterate?
  - N times (one iteration per vertex)
- How many times loop on line 7 iterate?
  - For each vertex, v, the loop executes deg(v) times
  - $= \sum_{v \in V} \theta[1 + \deg(v)]$
  - $= \theta(\sum_v 1) + \theta(\sum_v \deg(v))$
  - = Θ (n) + Θ (m)
- Total = Θ(n+m)

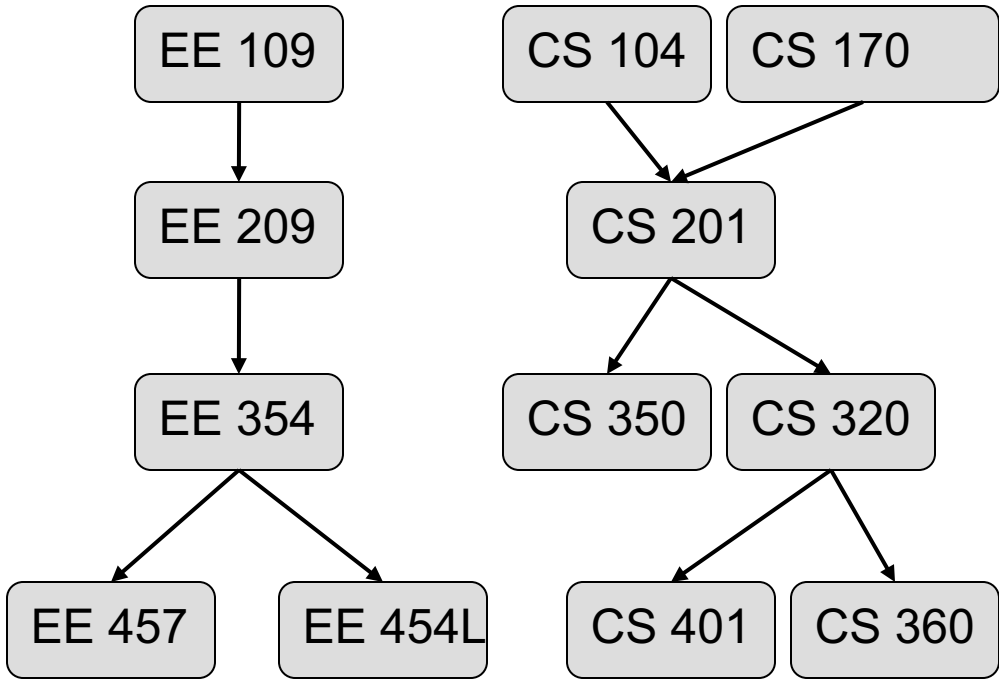



```
BFS(G,u)
1  for each vertex v
2      pred[v] = nil, d[v] = inf.
3  Q = new Queue
4  Q.enqueue(u),  d[u]=0
5  while Q is not empty
6     v = Q.front(); Q.dequeue()
7     foreach neighbor, w, of v:
8        if pred[w] < 0 // w not found
9           Q.enqueue(w)
10          pred[w] = v,  d[w] = d[v]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

DEPTH FIRST SEARCH

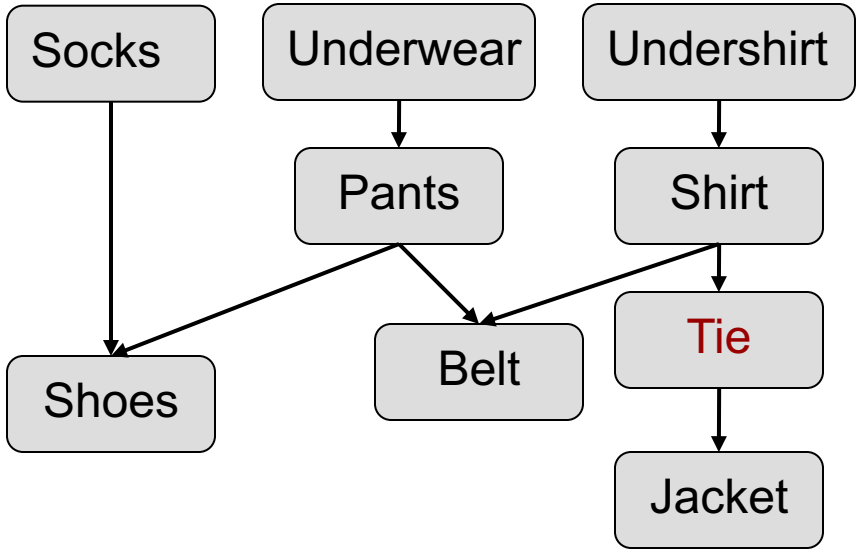
DFS Application: Topological Sort

- Breadth-first search doesn't solve all our problems.
- Given a graph of dependencies (tasks, prerequisities, 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

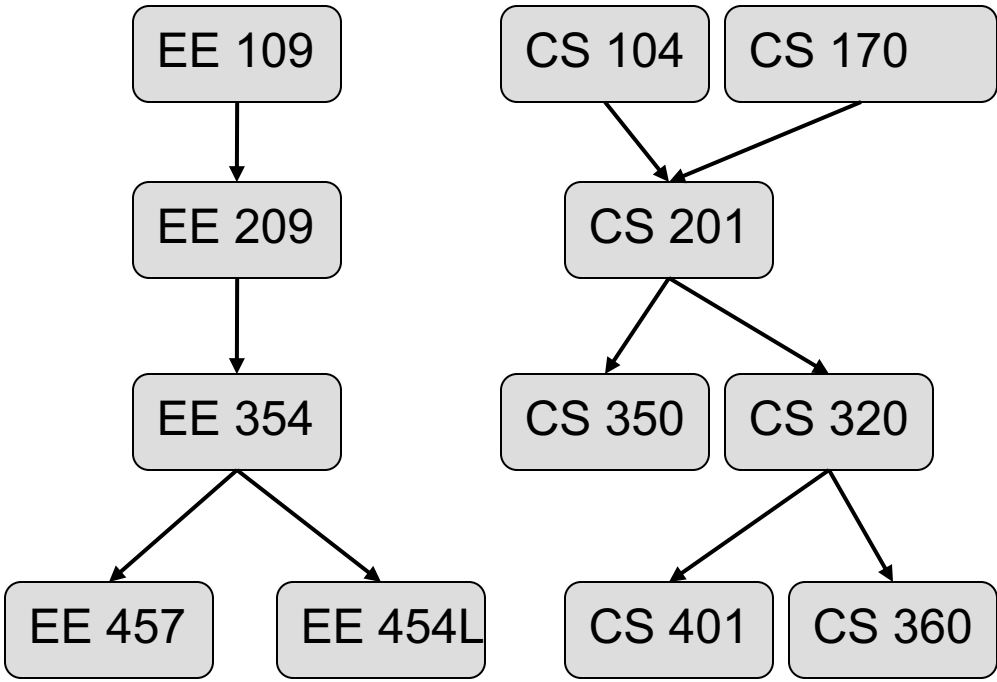
- Another example
  - Getting dressed
- More Examples:
  - Project management scheduling
  - Build order in a Makefile or other compile project
  - Cooking using a recipe
  - Instruction execution on an out-of-order pipelined CPU
  - Production of output values in a simulation of a combinational gate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



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

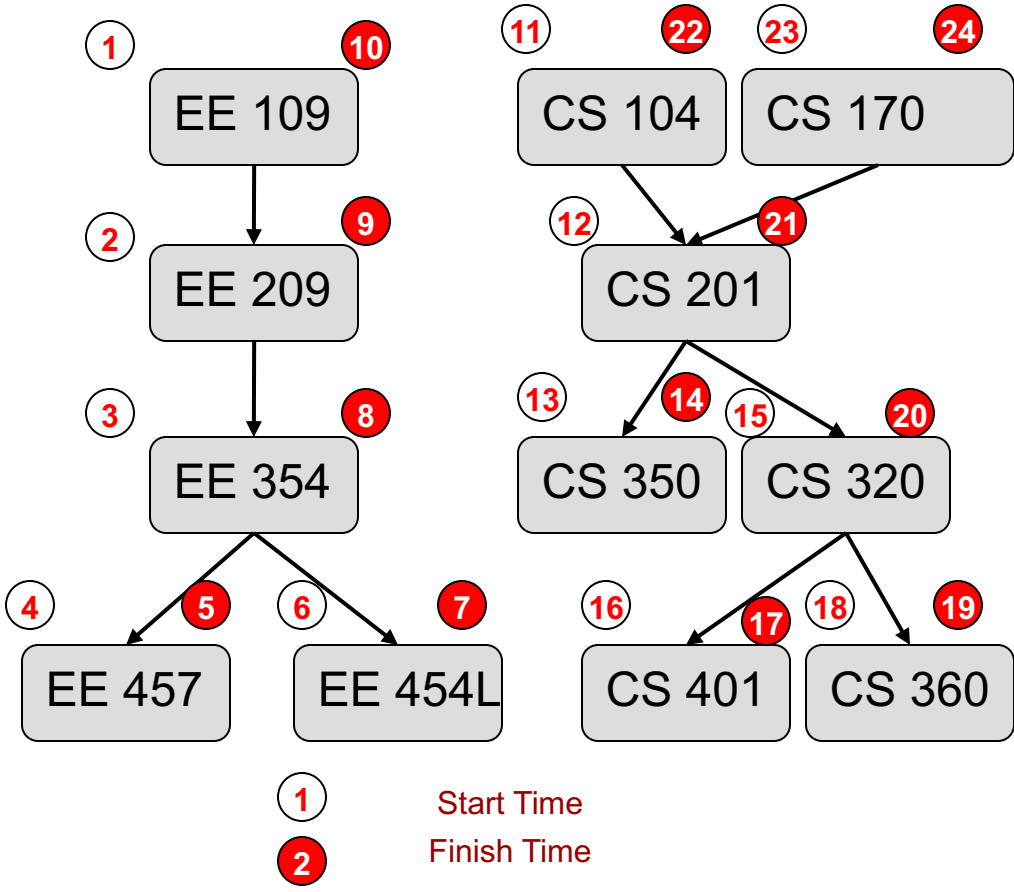
Topological Sort

- 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 320, CS 360, CS 477, CS 350, EE 109, EE 209L, EE 354, EE 454L, EE 457

DFS Algorithm

- 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
- DFS is recursive!!
- If cycles in the graph, ensure we don't get caught visiting neighbors endlessly
  - Color them as we go
  - White = unvisited,
  - Gray = visited but not finished
  - Black = finished

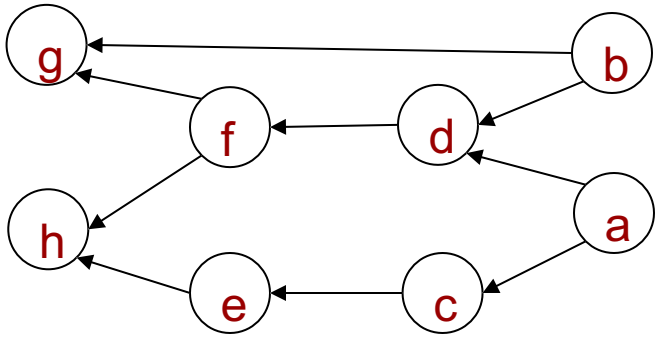
```
DFS-All (G)
1  for each vertex u
2    u.color = WHITE
3  finish_list = empty_list
4  for each vertex u do
5     if u.color == WHITE then
6        DFS-Visit (G, u, finish_list)
7  return finish_list
```

```
DFS-Visit (G, u)
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

```
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
```

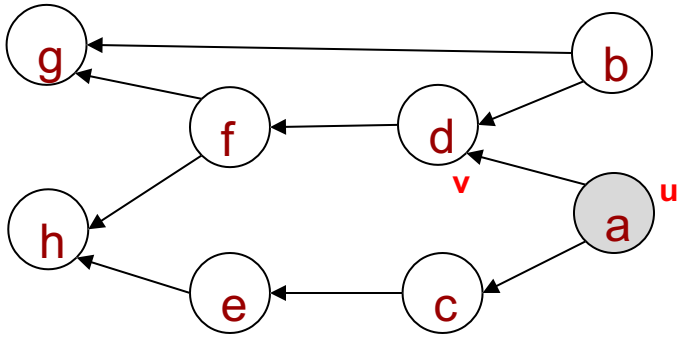
```
DFS-Visit (G, u)
1    u.color = GRAY
2    for each vertex v in Adj(u) do
3       if v.color = WHITE then
4            DFS-Visit (G, v)
5    u.color = BLACK
6    finish_list.append(u)
```



Depth First-Search

```
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)
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)
```

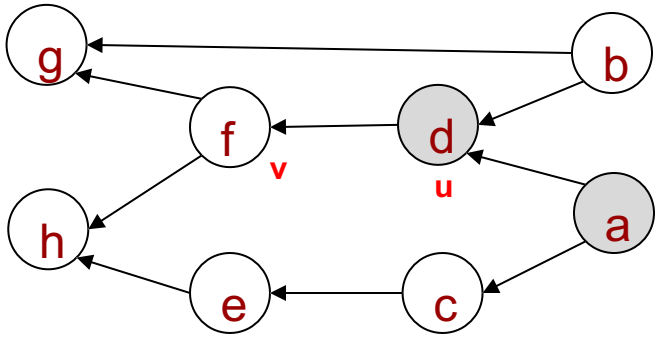


DFS-Visit(G,a):

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)
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)
```

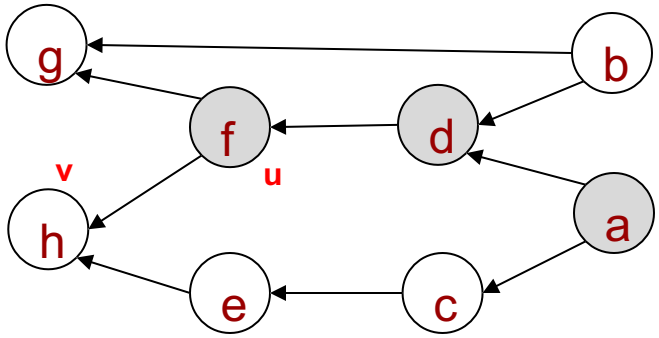


DFS-Visit(G,d):
DFS-Visit(G,a):

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)
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)
```

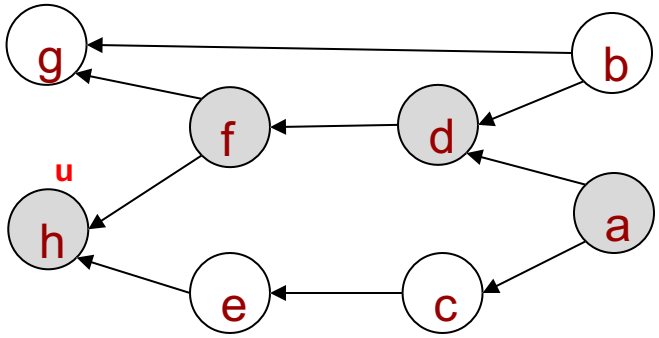


DFS-Visit(G,f):
DFS-Visit(G,d):
DFS-Visit(G,a):

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)
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)
```

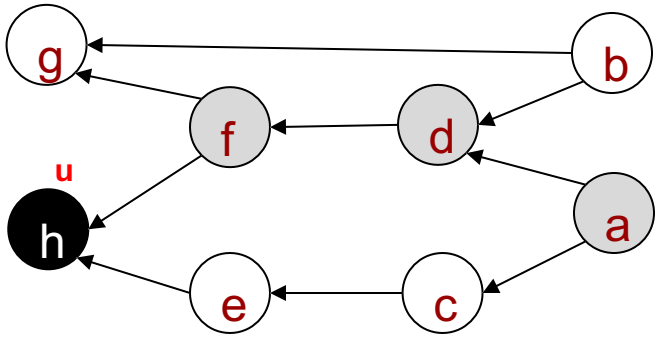


DFS-Visit(G,h):
DFS-Visit(G,f):
DFS-Visit(G,d):
DFS-Visit(G,a):

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)
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)
```



Finish_list:

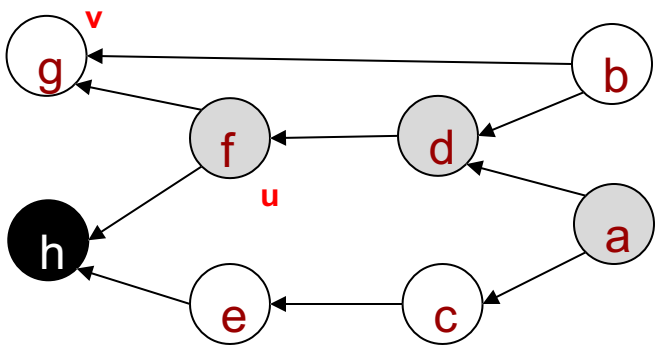
h

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

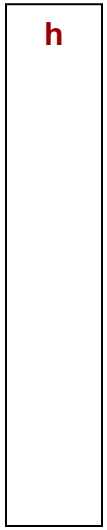
Depth First-Search

```
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)
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)
```



Finish_list:



DFS-Visit(G,f):

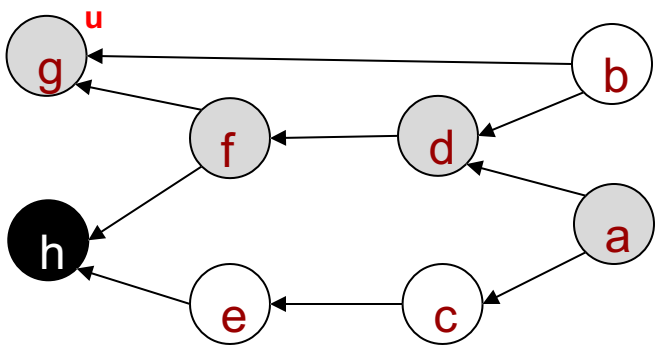
DFS-Visit(G,d):

DFS-Visit(G,a):

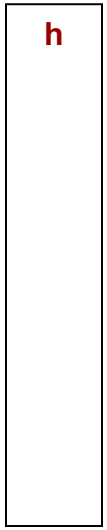
Depth First-Search

```
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)
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)
```



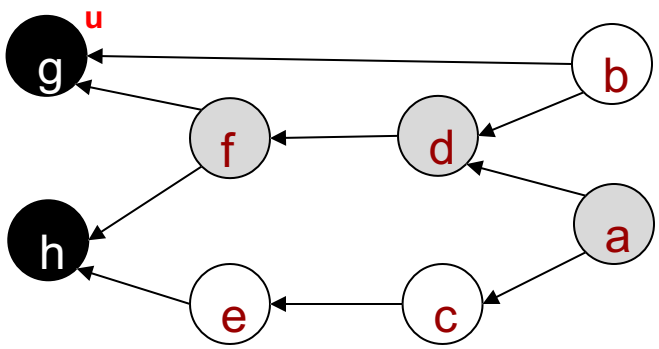
Finish_list:



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)
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)
```



Finish_list:

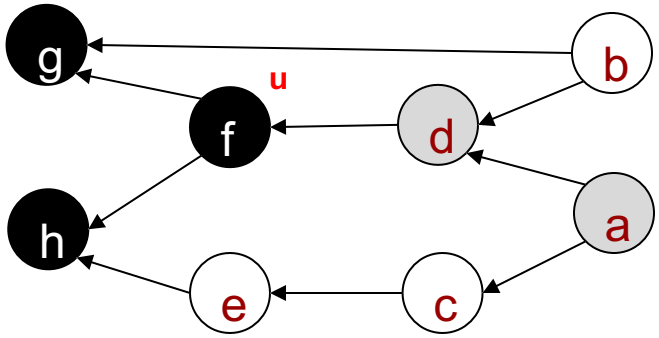
h, g

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

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)
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)
```



Finish_list:

h,
g,
f

DFS-Visit(G,f):

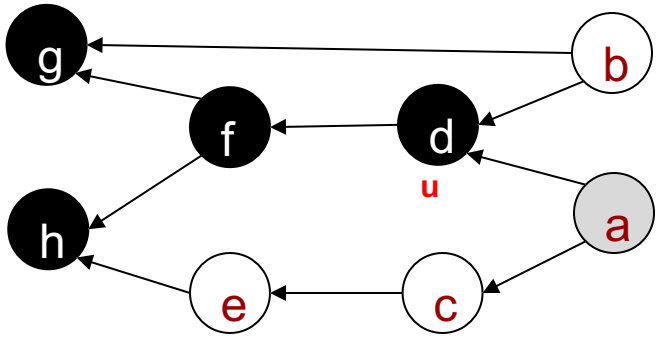
DFS-Visit(G,d):

DFS-Visit(G,a):

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)
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)
```



Finish_list:

h,
g,
f,
d

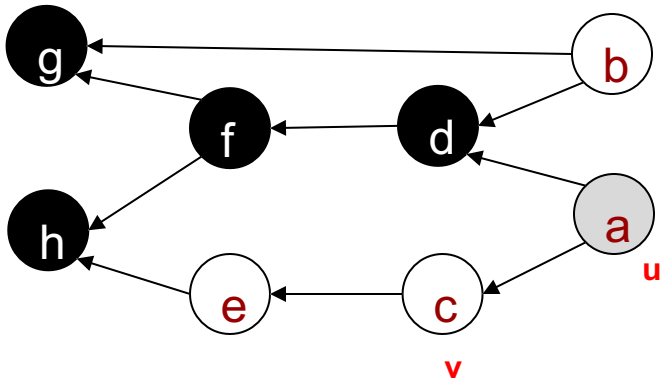
DFS-Visit(G,d):

DFS-Visit(G,a):

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)
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)
```



Finish_list:

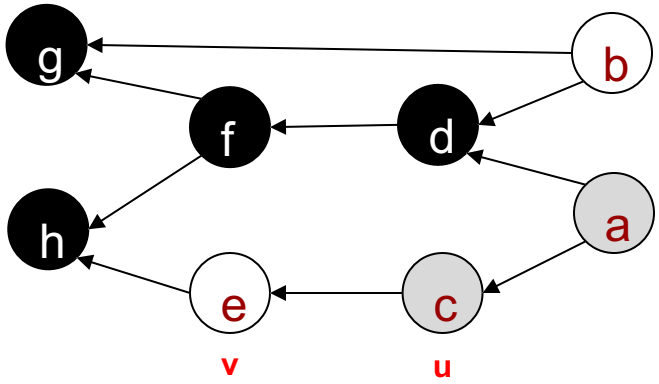
h,
g,
f,
d

DFS-Visit(G,a):

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)
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)
```



Finish_list:

h,
g,
f,
d

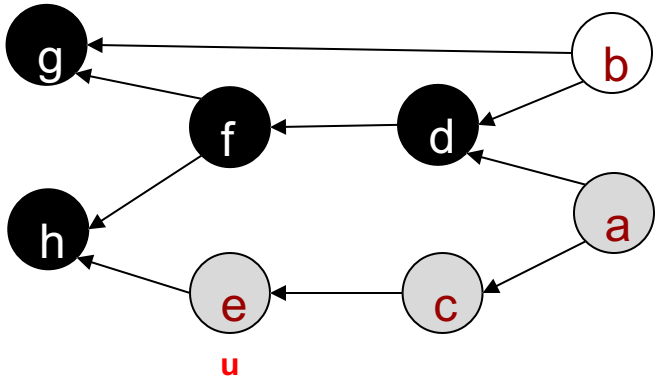
DFS-Visit(G,c):

DFS-Visit(G,a):

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)
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)
```



Finish_list:

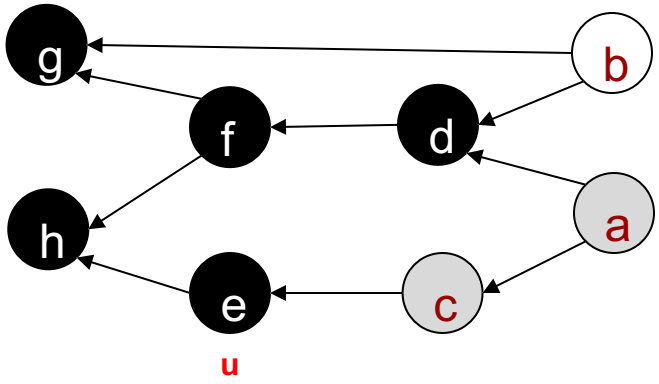
h,
g,
f,
d

DFS-Visit(G,e):
DFS-Visit(G,c):
DFS-Visit(G,a):

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)
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)
```



Finish_list:

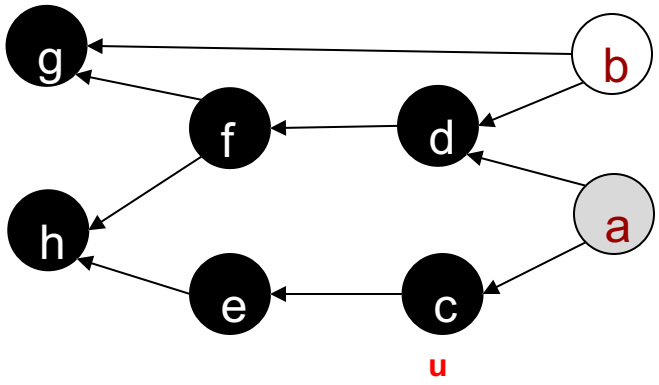
h,
g,
f,
d.
e

DFS-Visit(G,e):
DFS-Visit(G,c):
DFS-Visit(G,a):

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)
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)
```



Finish_list:

h,
g,
f,
d.
e,
c

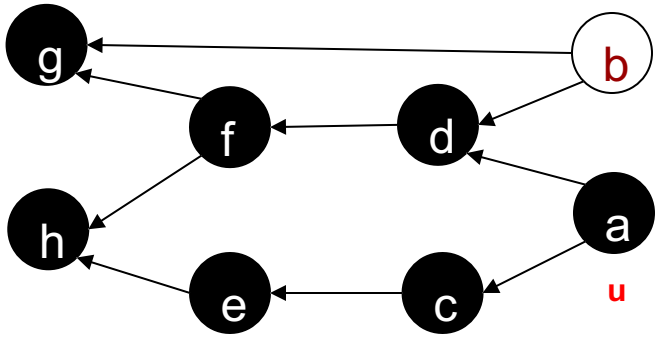
DFS-Visit(G,c):

DFS-Visit(G,a):

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)
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)
```



Finish_list:

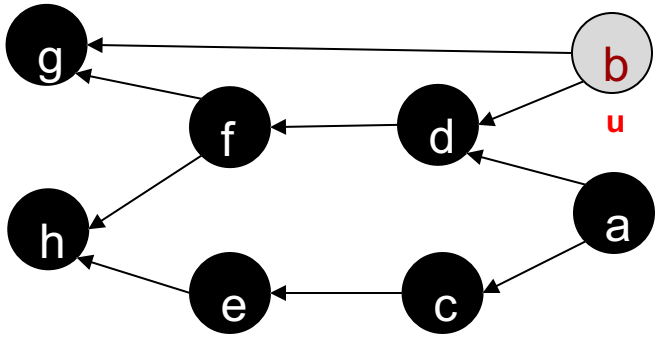
h,
g,
f,
d.
e,
c,
a

DFS-Visit(G,a):

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)
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)
```

Finish_list:

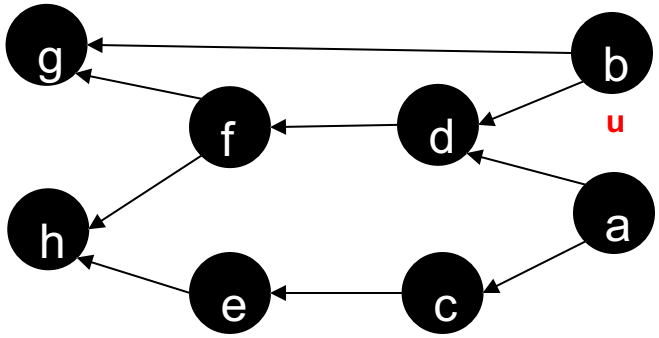
h,
g,
f,
d.
e,
c,
a

DFS-Visit(G,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)
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)
```



Finish_list:

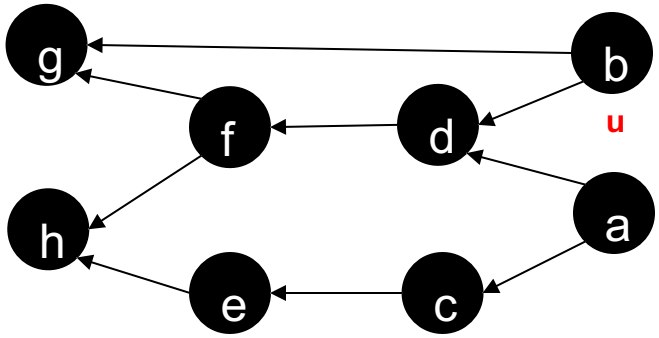
h,
g,
f,
d.
e,
c,
a,
b

DFS-Visit(G,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)
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)
```



Finish_list:

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

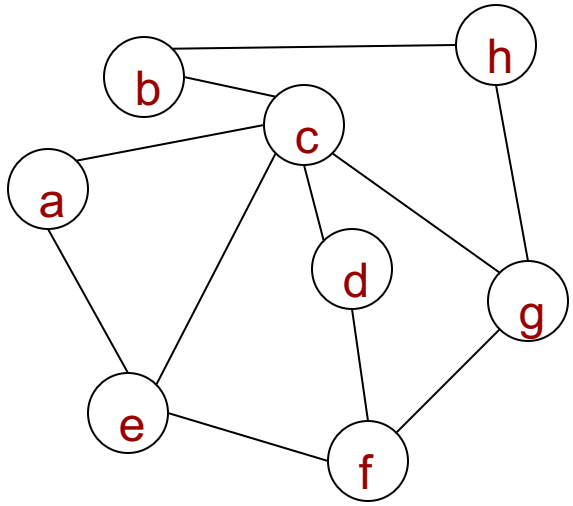
With Cycles in the graph

ANOTHER EXAMPLE

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)
```

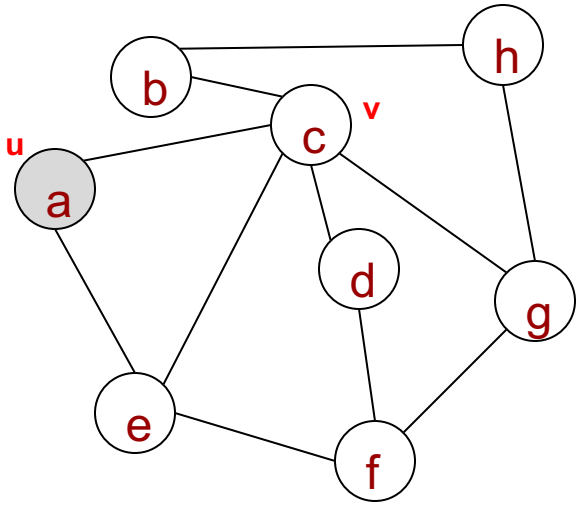
```
DFS-Visit (G, u)
1    u.color = GRAY
2    for each vertex v in Adj(u) do
3      if v.color = WHITE then
4          DFS-Visit (G, v)
5    u.color = BLACK
6    finish_list.append(u)
```



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)
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)
```

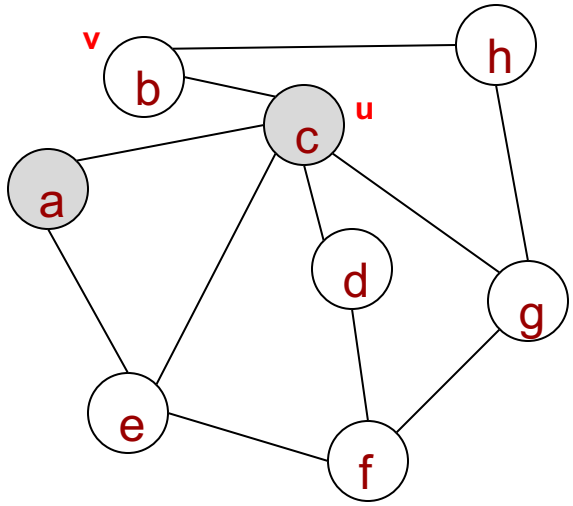


DFS-Visit(G,a):

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)
```

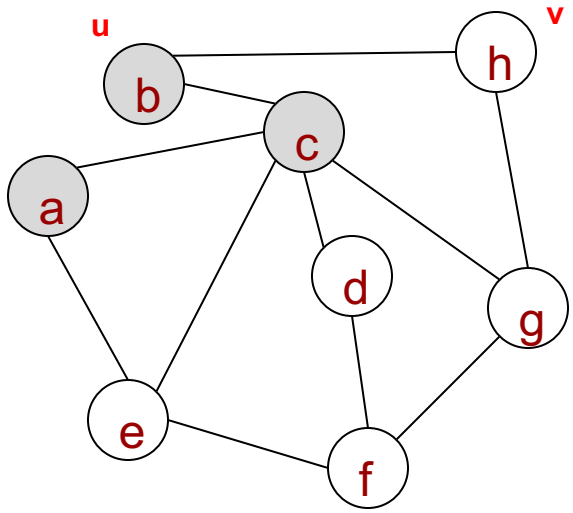


DFS-Visit(G,c):
DFS-Visit(G,a):

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)
```

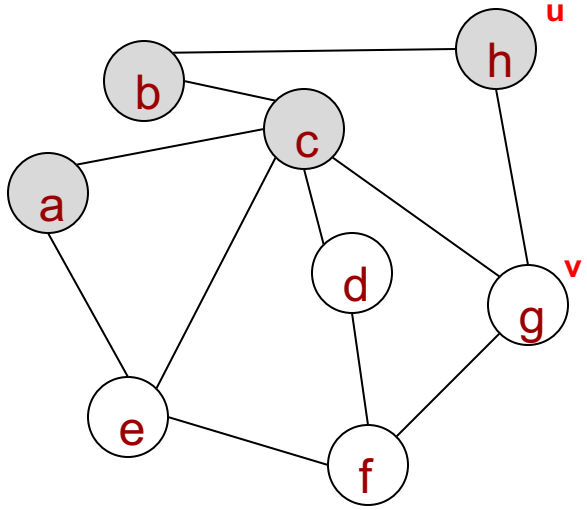


DFS-Visit(G,b):
DFS-Visit(G,c):
DFS-Visit(G,a):

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)
```

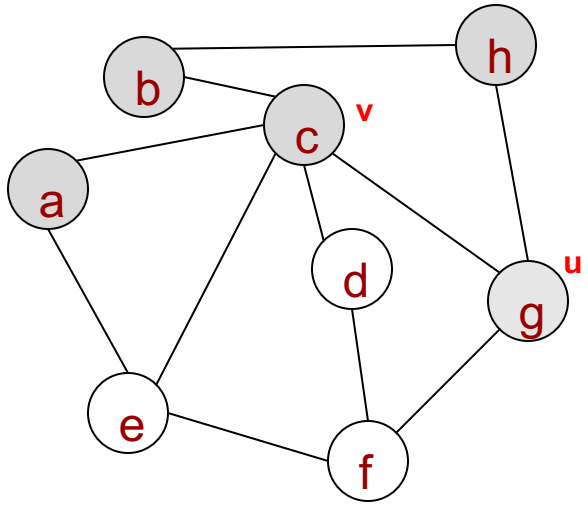


DFS-Visit(G,h):
DFS-Visit(G,b):
DFS-Visit(G,c):
DFS-Visit(G,a):

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)
```

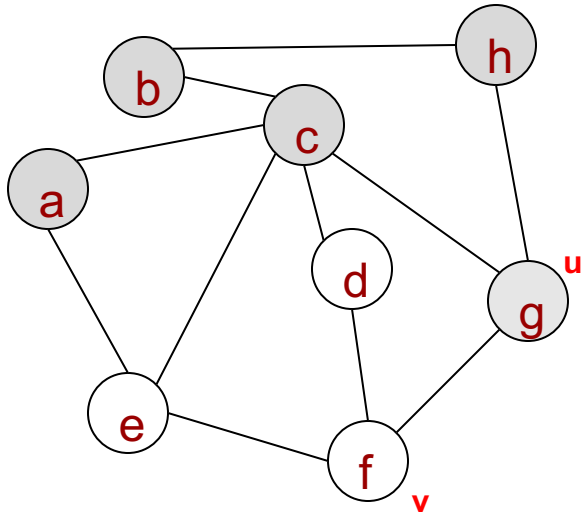


DFS-Visit(G,g):
DFS-Visit(G,h):
DFS-Visit(G,b):
DFS-Visit(G,c):
DFS-Visit(G,a):

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)
```

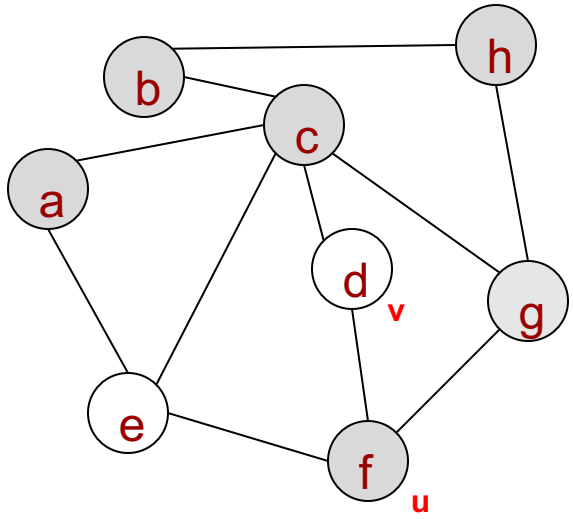


DFS-Visit(G,g):
DFS-Visit(G,h):
DFS-Visit(G,b):
DFS-Visit(G,c):
DFS-Visit(G,a):

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)
```

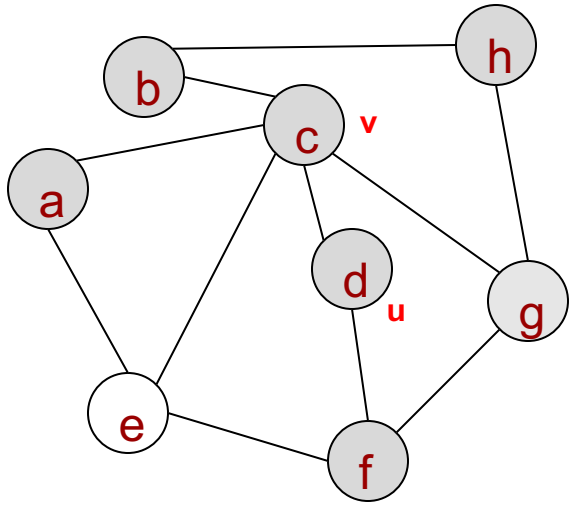


DFS-Visit(G,f):
DFS-Visit(G,g):
DFS-Visit(G,h):
DFS-Visit(G,b):
DFS-Visit(G,c):
DFS-Visit(G,a):

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)
```

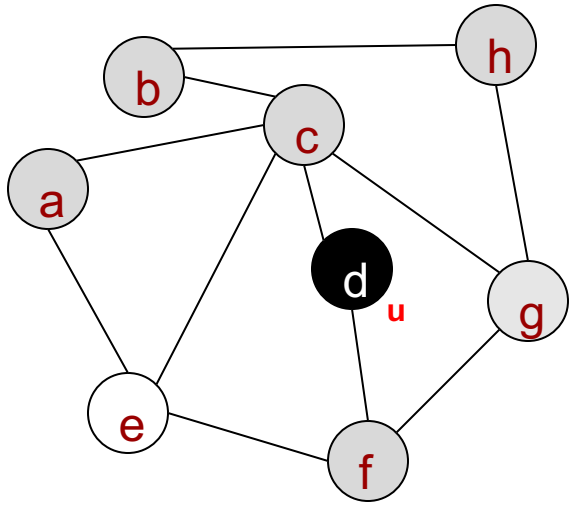


DFS-Visit(G,d):
DFS-Visit(G,f):
DFS-Visit(G,g):
DFS-Visit(G,h):
DFS-Visit(G,b):
DFS-Visit(G,c):
DFS-Visit(G,a):

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:

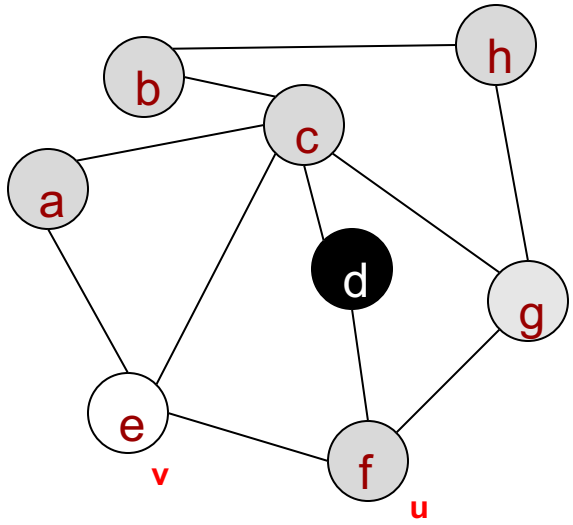
d

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):

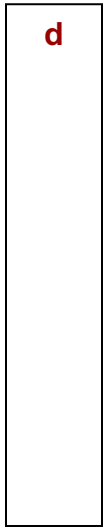
Depth First-Search

```
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)
```



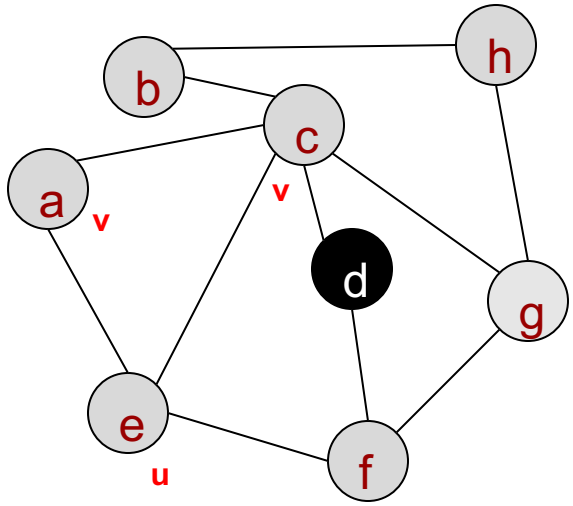
DFSQ:



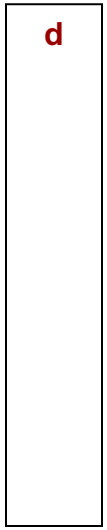
Depth First-Search

```
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)
```



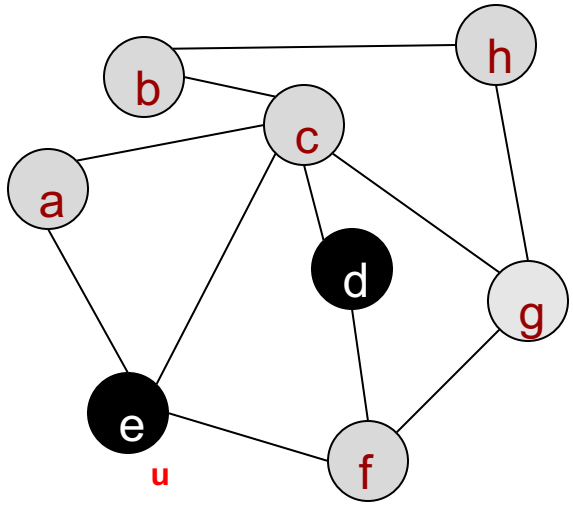
DFSQ:



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:

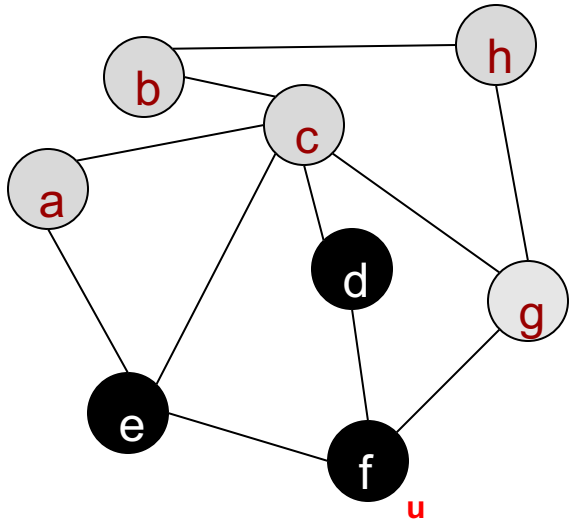
d
e

DFS-Visit(G,e):
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:

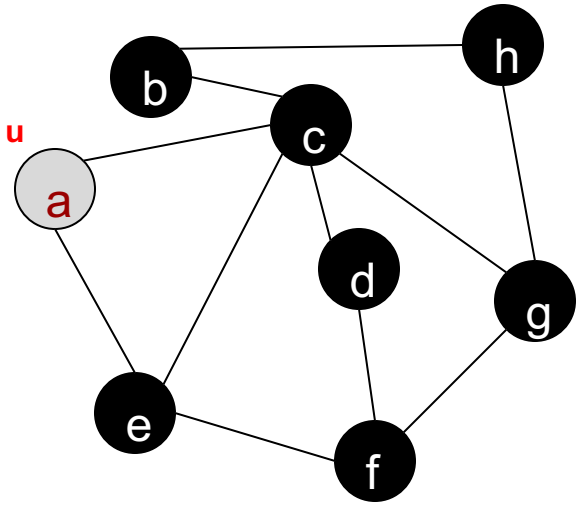
d
e
f

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:

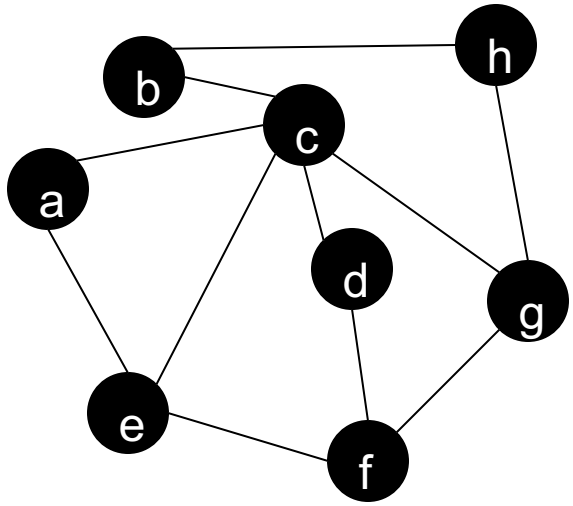
d
e
f
g
h
b
c

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)
```



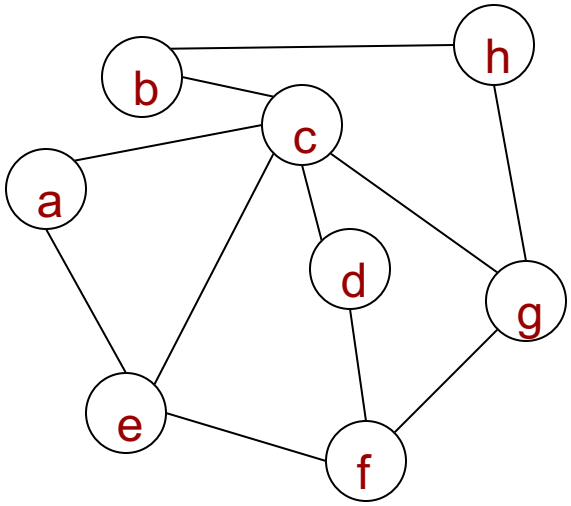
DFSQ:

d
e
f
g
h
b
c
a

ITERATIVE VERSION

Depth First-Search

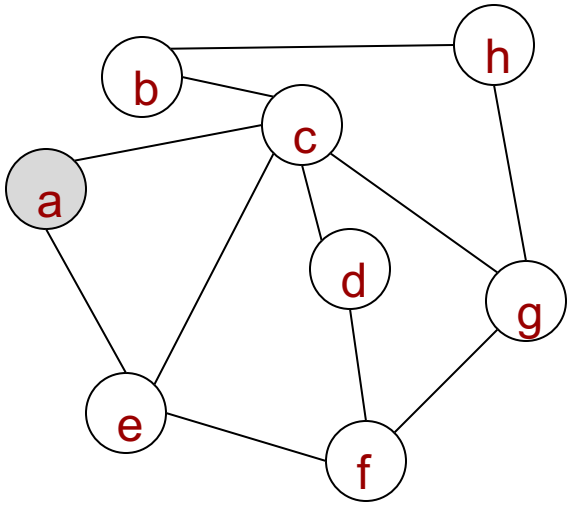
```
DFS (G,s)
1   for each vertex u
2     u.color = WHITE
3   st = new Stack
4   st.push_back(s)
5   while st not empty
6     u = st.back()
7     if u.color == WHITE then
8       u.color = GRAY
9       foreach vertex v in Adj(u) do
10        if v.color == WHITE
11          st.push_back(v)
12    else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



st: a

Depth First-Search

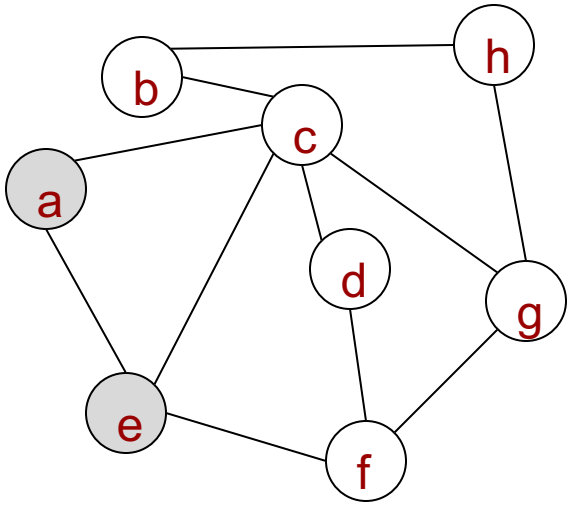
```
DFS (G,s)
1   for each vertex u
2     u.color = WHITE
3   st = new Stack
4   st.push_back(s)
5   while st not empty
6     u = st.back()
7     if u.color == WHITE then
8       u.color = GRAY
9       foreach vertex v in Adj(u) do
10        if v.color == WHITE
11          st.push_back(v)
12  else if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



st: | a | c | e |

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()
```

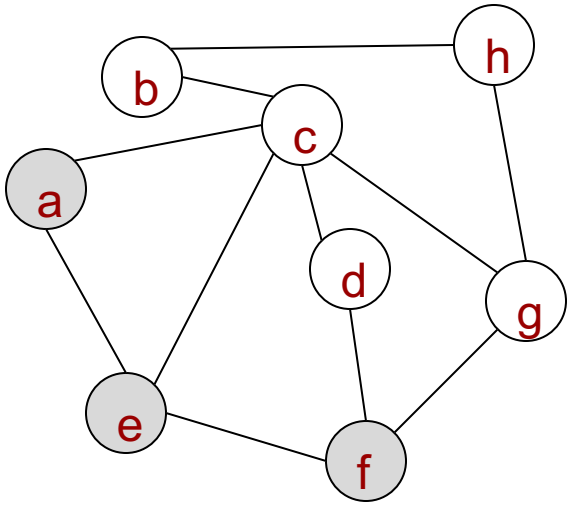


st:

a	c	e	c	f

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()
```

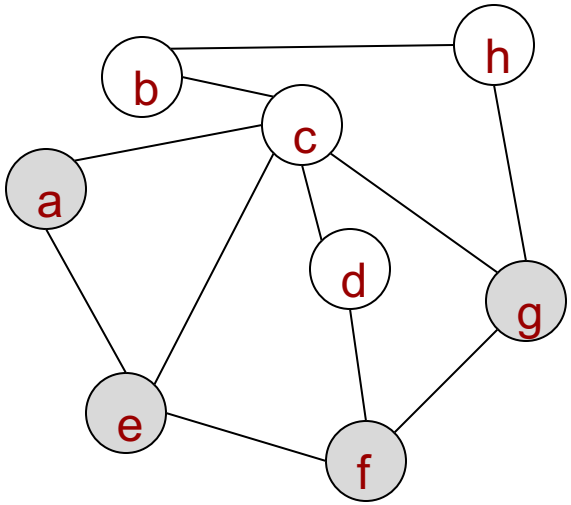


st:

a	c	e	c	f	d	g

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()
```

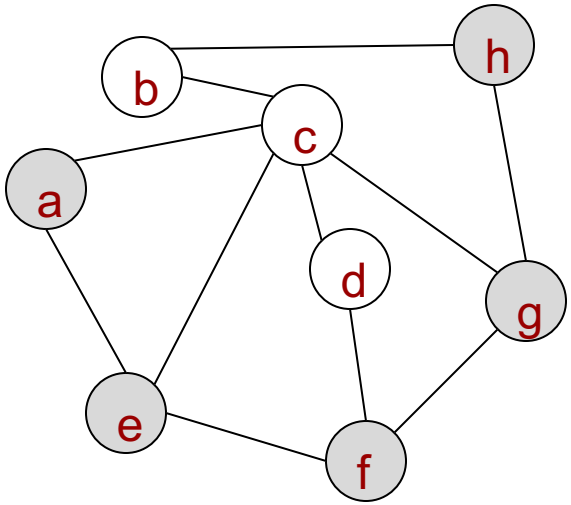


st:

a	c	e	c	f	d	g	c	h

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()
```

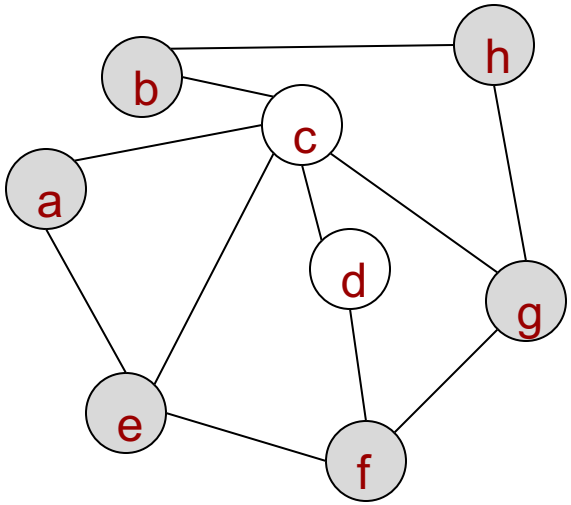


st:

a	c	e	c	f	d	g	c	h	b

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()
```

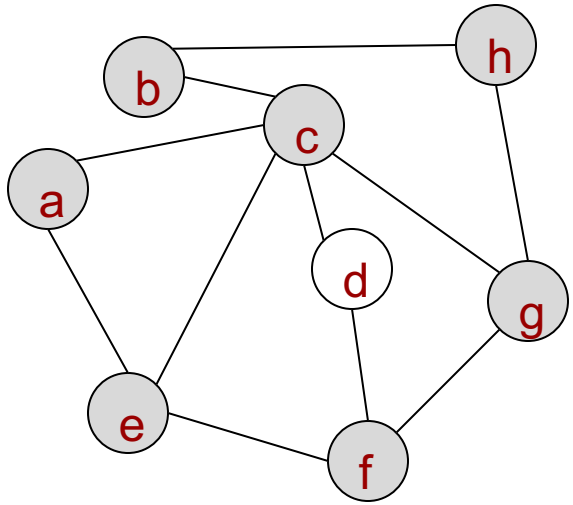


st:

a	c	e	c	f	d	g	c	h	b	c

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()
```

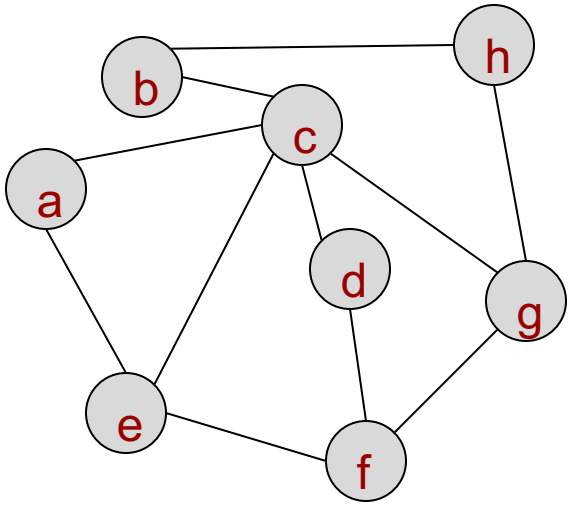


st:

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

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()
```

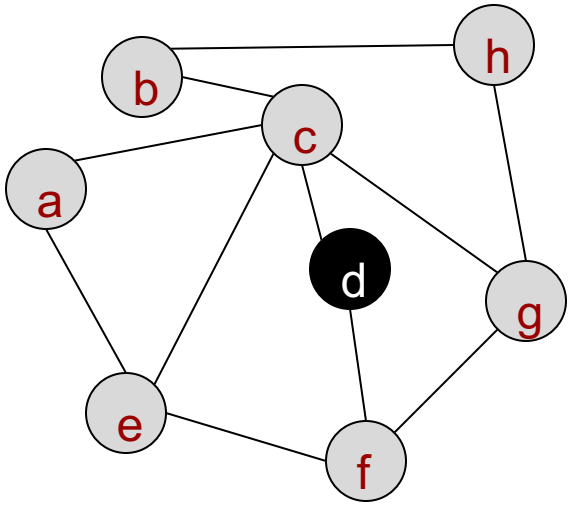


st:

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

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()
```

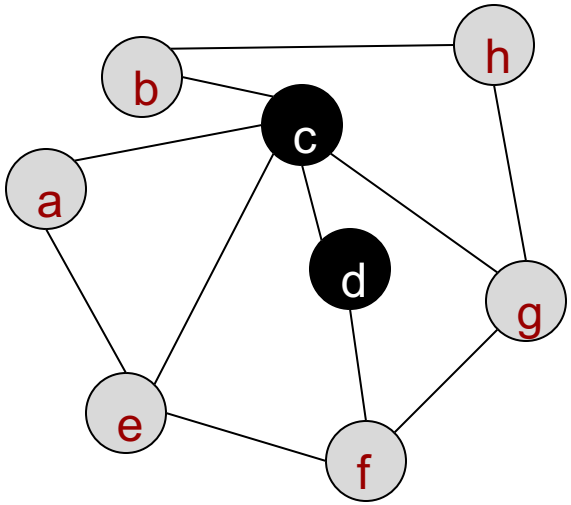


st:

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

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()
```

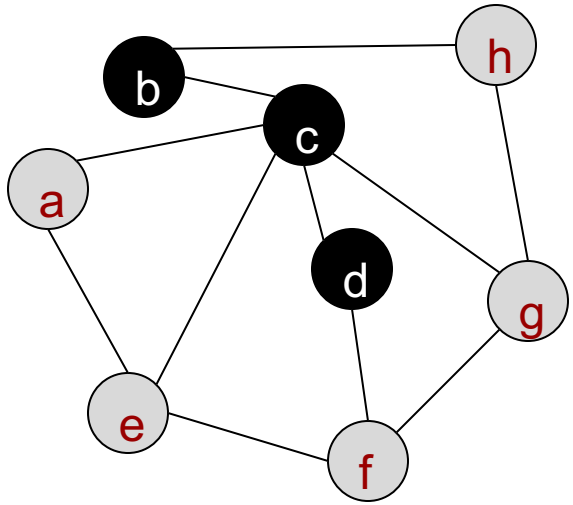


st:

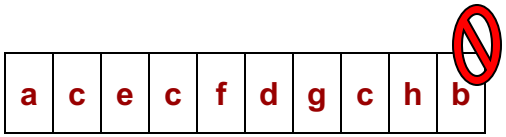
a	c	e	c	f	d	g	c	h	b	c

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()
```

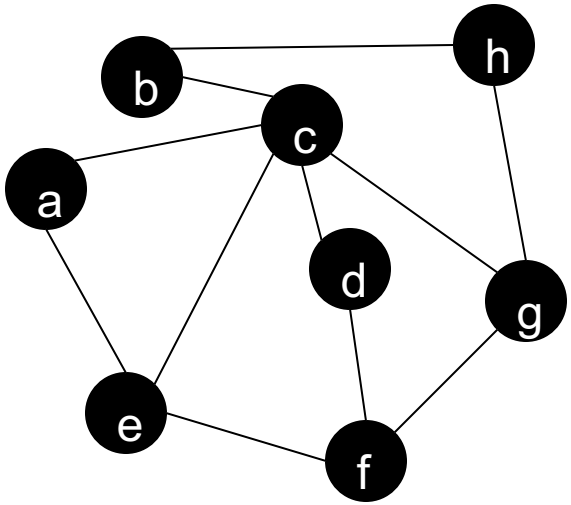


st:



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()
```



st: a

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.pop_back()
8       if u.color == WHITE
9          u.color = GRAY
10         foreach vertex v in Adj(u) do
11            st.push_back(v)
```