Graphs:

Set of vertices (or nodes)
denoted by V

Set of edges between pairs of vertices
denoted by E

Edges can be undirected so
E is a set of unordered pairs,
e.g., $E = \{\{1,2\},\{1,5\},\{3,4\},\{4,2\}\}$
it's symmetric: $\underbrace{\{i,j\}}_{\text{unordered pair}} \in E$ iff $\{j,i\} \in E$

OR Edges can be directed so
E is a set of ordered pairs
can denote by ordered set (i,j) or
can denote as $\overrightarrow{ij}$

The graph is denoted as $G=(V,E)$
and we use $n = |V| = \#$ of vertices
& $m = |E| = \#$ of edges.

Representing graphs:

1) Adjacency matrix A:

$n \times n$ matrix A where (i,j) entry is

$$A_{ij} = \begin{cases} 1 & \text{if edge } i \to j \\ 0 & \text{if no edge } i \to j \end{cases}$$

For undirected G, A is symmetric:
$$A_{ij} = A_{ji}$$

Takes $O(n^2)$ space (even if m is small)

Takes $O(1)$ time to check if there is an edge $i \to j$?

But takes $O(n)$ time to find all neighbors of a specific vertex.

2) Adjacency list:

Array of n linked lists.

Linked list i contains the neighbors of vertex i (in any order).

Uses $O(n+m)$ space.

Takes $O(\text{degree}(v))$ time to find all neighbors of v
& $O(n+m)$ time to find all edges of G.

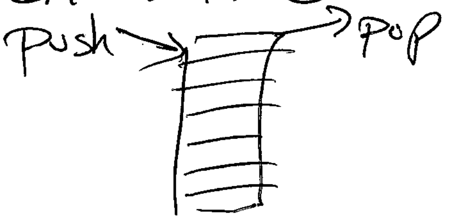
But $O(\text{degree}(v))$ time to check if edge $v \to w$.

Exploring graphs:

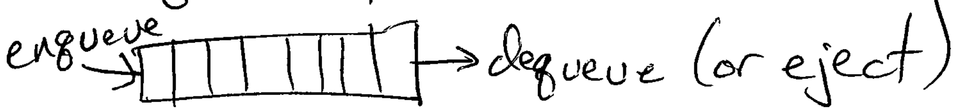
DFS = depth first search

BFS = breadth first search.

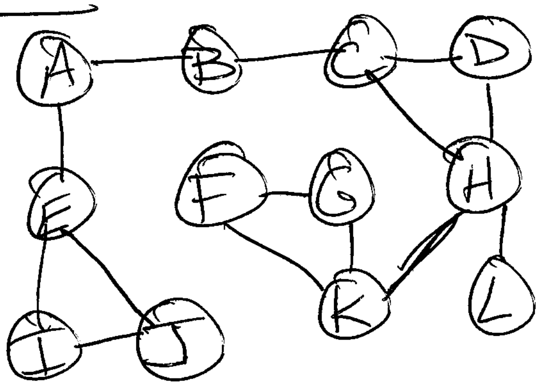
DFS uses a stack = LIFO = last-in first-out



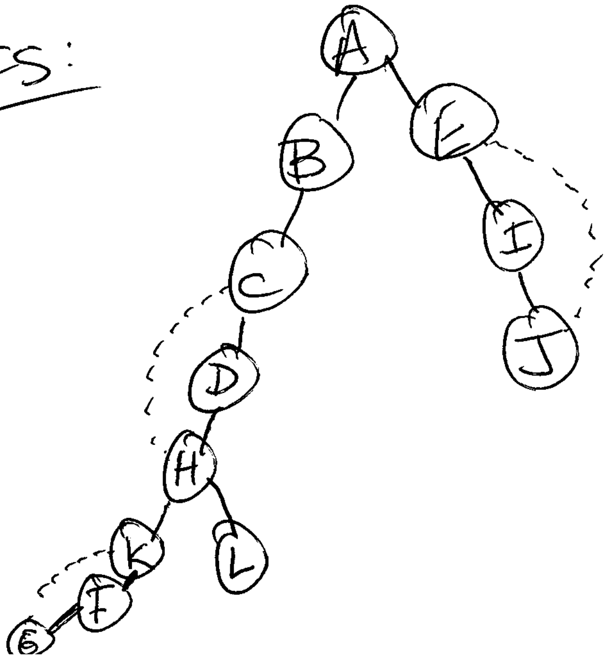
BFS uses a queue = FIFO = first-in first-out



Example:



DFS:



DFS: can implement a stack using recursion

Running DFS starting from a vertex v.

DFS(G):

for all $w \in V$, set visited(v) = FALSE
Explore(v)

Explore(w):

visited(w) = TRUE
for all $(w, z) \in E$:
if not visited(z) then Explore(z)

Explore(z): finds all vertices reachable from z

can use it to find connected components of undirected graphs.

DFS works on undirected & directed graphs.

assumes G is given in adj. list. form.

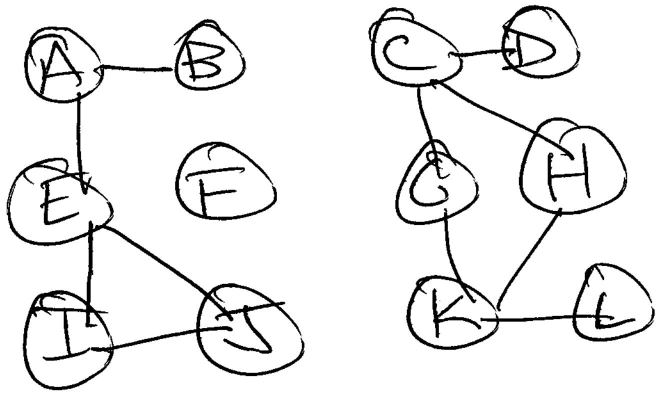
takes $O(n+m)$ time.

For undirected G:

vertices v & w are connected if path between them.

Connected component = maximal set of connected vertices.

Example:



3 components: {A, B, E, I, J}, {F}, {C, D, G, H, K, L}

Approach to find connected components:

1) Choose arbitrary start vertex z.
2) Run Explore(z) to find component containing z
3) Choose an unexplored z & repeat

DFS(G):

for all $v \in V$, visited(v) = FALSE
cc = 0
for all $v \in V$,
  if not visited(v) then:
    cc++
    Explore(v)

Explore(z):

visited(z) = TRUE
ccnum(z) = cc
for each $(z,w) \in E$:
  if not visited(w) then Explore(w)

Running time is $O(n+m)$, $n = |V|$, $m = |E|$

for G in adj. list representation.

For directed graphs to solve connectivity need more information from DFS.

Add a clock, keep track of preorder # and postorder #

pre(v) = time start exploring v

post(v) = time finish exploring all neighbors of v.

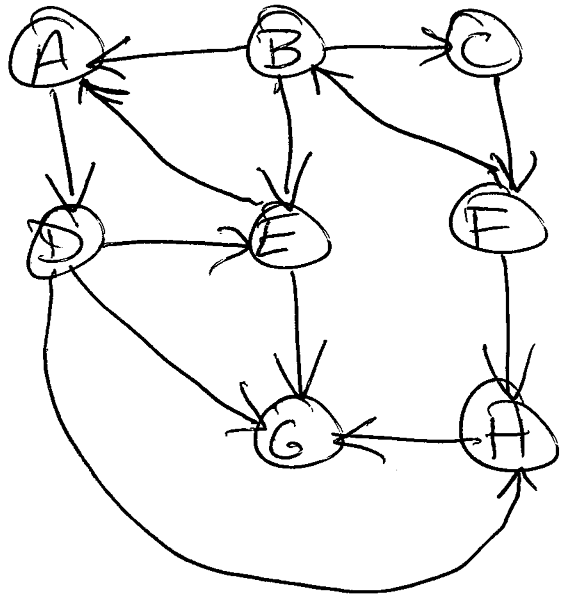
DFS(G):

```
for all v∈V, visited(v) = FALSE
clock = 1
for all v∈V,
    if not visited(v) then Explore(v)
```

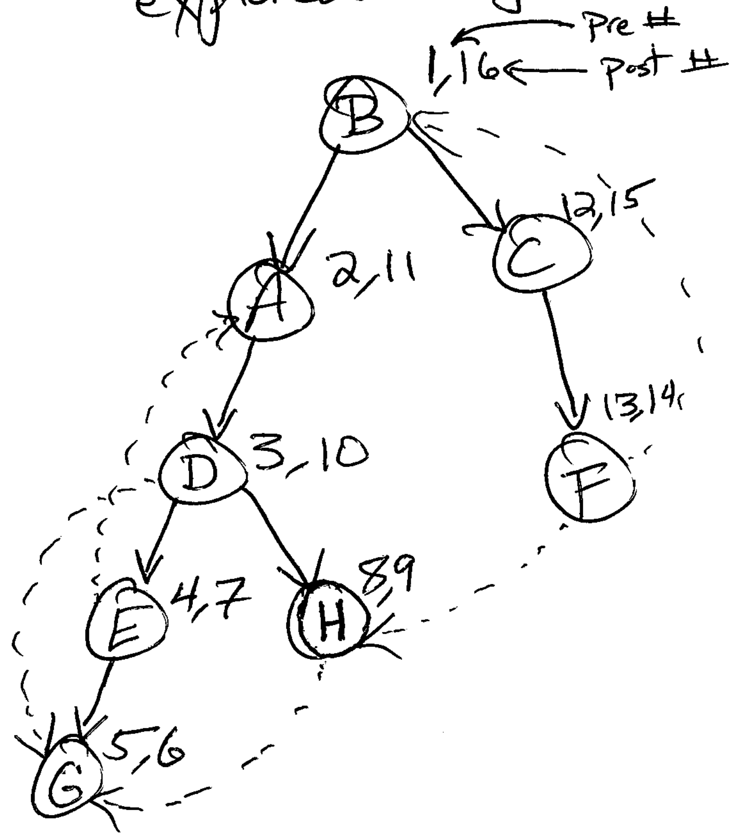
Explore(z):

```
visited(z) = TRUE
pre(z) = clock
clock++
for each (z,w)∈E:
    if not visited(w) then Explore(w)
post(z) = clock
clock++
```

Example:



Run DFS on above graph starting at B
explored edges = tree edges



Dashed edges = non-tree edges = unexplored edges

3 types of non-tree edges:

Forward: Example: $D \to G$ vertex $\to$ descendant

$$\underbrace{pre(D) < \underbrace{pre(G) < post(G)}_{G} < post(D)}_{D}$$

Back: $E \to A$, $F \to B$, vertex $\to$ ancestor

$$\underbrace{pre(B) < \underbrace{pre(F) < post(F)}_{F} < post(B)}_{B}$$

Cross: $F \to H$, $H \to G$

$$\underbrace{pre(H) < post(H)}_{H} < \underbrace{pre(F) < post(F)}_{F}$$

Tree edge: $B \to A$ vertex $\to$ child

$$\underbrace{pre(B) < \underbrace{pre(A) < post(A)}_{A} < post(B)}_{B}$$

(Same form as forward edge)

<u>Property</u>: Directed G has a cycle iff its DFS tree contains a back edge.

<u>Proof</u>:

($\Leftarrow$) Consider a back edge, say $e = w \to v$.
So w is a descendent of v in the DFS tree.
In the DFS tree there is a path P from v to w. Take this path P plus e this gives a cycle $C = P \cup e$.

($\Rightarrow$) Consider a cycle C in G.
Some vertex of C is visited first, say v.
Rest of C is reachable from v so is in v's subtree
One of these other vertices in $C - v$ has an edge to v, and this will be a back edge.

$\blacksquare$

So if no cycles then no back edges.

For edge $z \to w$,
if back edge, $post(z) < post(w)$
if cross, forward, or tree edge,
$post(w) < post(z)$.

So can detect if G has a cycle by checking post #'s to see if we ever find a back edge.

DAG = directed acyclic graph
no cycles
hence no back edges

Topologically sorting a DAG
= order the vertices so that all edges go left to right
(lower) (higher)

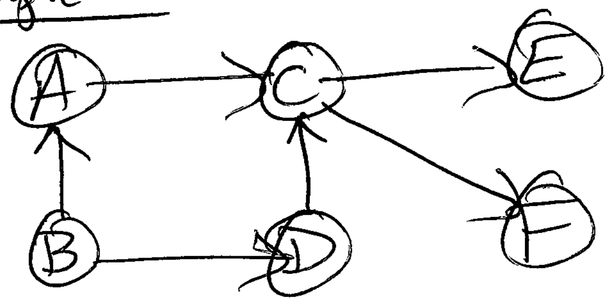
Easy to do:
Run DFS &
sort by decreasing post #

highest post # $\longrightarrow$ lowest post #

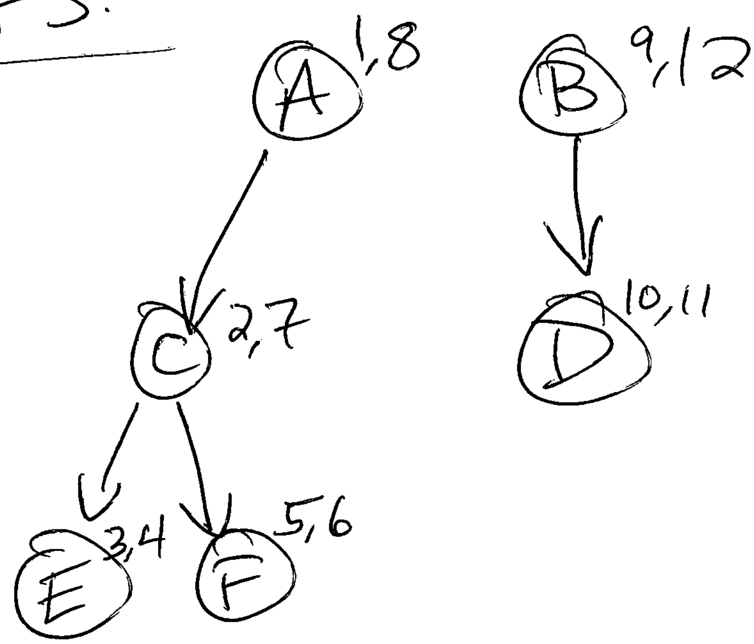
$z \longrightarrow w$

Since no back edges, for every edge $z \to w$
we have $post(w) < post(z)$

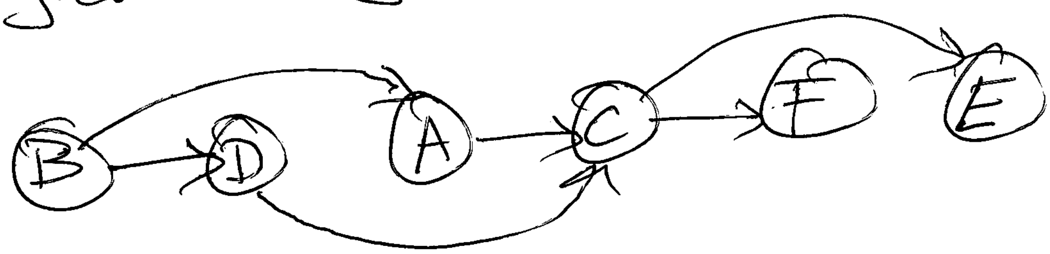
Example:



Run DFS:



Topological sorting:



(Note, there are other topological orderings)