CS2040 Data Structures and Algorithms
Lecture Note #11 – Part 2

Graphs
Part 2: Traversal Algorithms

Review – Binary Tree Traversal

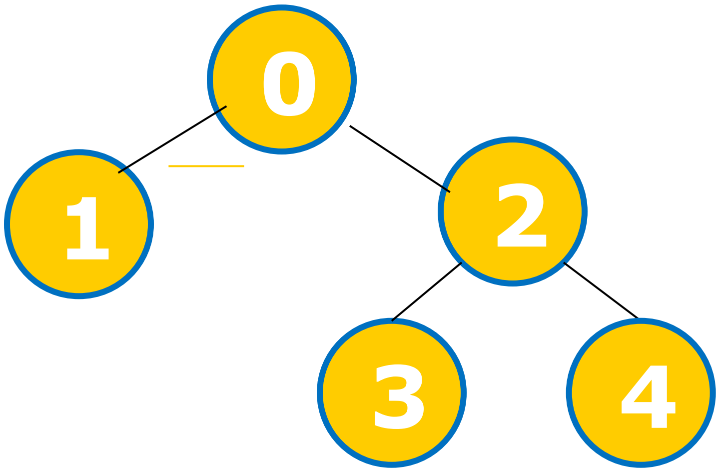
In a binary tree, there are three standard traversal:

- Preorder
- **Inorder**
- Postorder
- (Note: "level order" is just BFS which we will see next)

`pre(u)`
`  visit(u);`
`  pre(u->left);`
`  pre(u->right);`
	`in(u)`
	`  in(u->left);`
	`  visit(u);`
	`  in(u->right);`
		`post(u)`
		`  post(u->left);`
		`  post(u->right);`
		`  visit(u);`

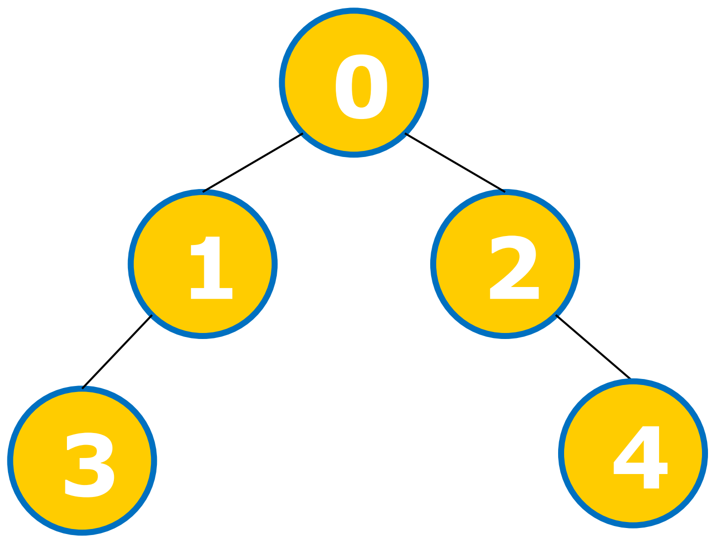
We start binary tree traversal from root:

- pre(root)/in(root)/post(root)
  - pre = 0, 1, 2, 3, 4
  - in = 1, 0, 3, 2, 4
  - post = 1, 3, 4, 2, 0



What is the PostOrder Traversal of this Binary Tree?

1. 0 1 2 3 4
2. 0 1 3 2 4
3. 3 4 1 2 0
4. 3 1 4 2 0



Traversing a Graph (1)

Two ingredients are needed for a **traversal:**

1. The start
2. The movement

Defining the start ("source")

- In tree, we *normally* start from root
  - Note: Not all tree are rooted though!
    - In that case, we have to select one vertex as the "source", see below
- In general graph, we do not have the notion of root
  - Instead, we start from a distinguished vertex
    - We call this vertex as the **"source" s**

Traversing a Graph (2)

Defining the movement:

- In (binary) tree, we only have (at most) two choices:
  - Go to the **left subtree** or to the **right subtree**
- In general graph, we can have more choices:
  - If **vertex u** and **vertex v** are adjacent/connected with edge (**u**, **v**);
    and we are now in **vertex u**; then we can also go to **vertex v** by traversing that edge (**u**, **v**)
- In (binary) tree, there is **no cycle**
- In general graph, we **may have (trivial/non trivial) cycles**
  - We need a way to avoid revisiting **u** → **v** → **w** → **u** → **v** … indefinitely

Traversing a Graph (2)

Solution: BFS and DFS

Idea: If a vertex **v** is reachable from **s**, then all neighbors

of **v** will also be reachable from **s**

(recursive definition)

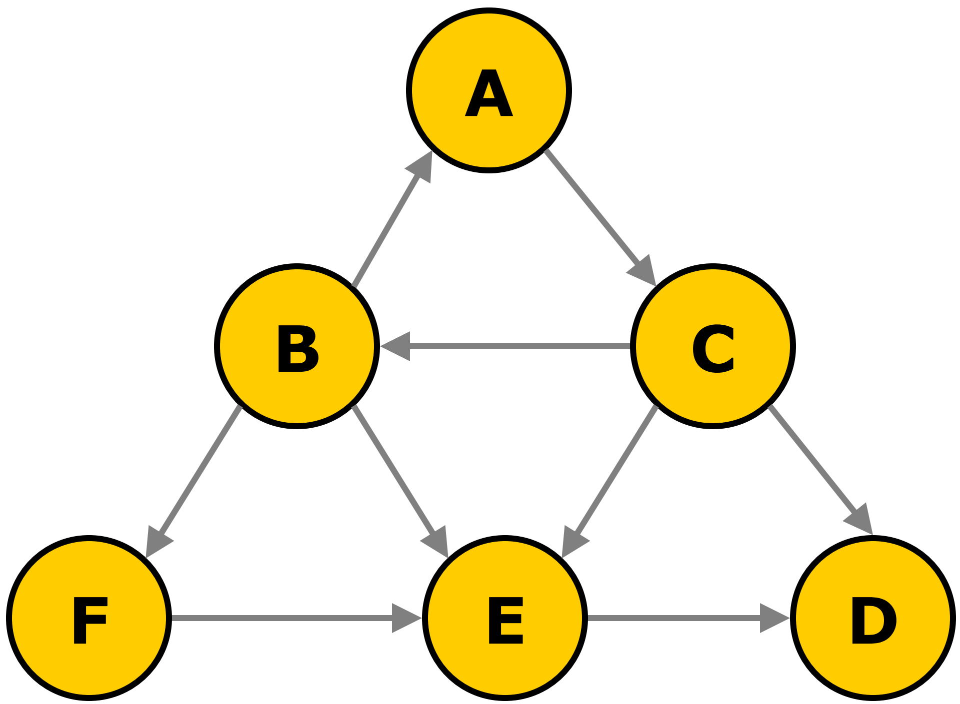
Breadth-First Search

Traversing a Graph

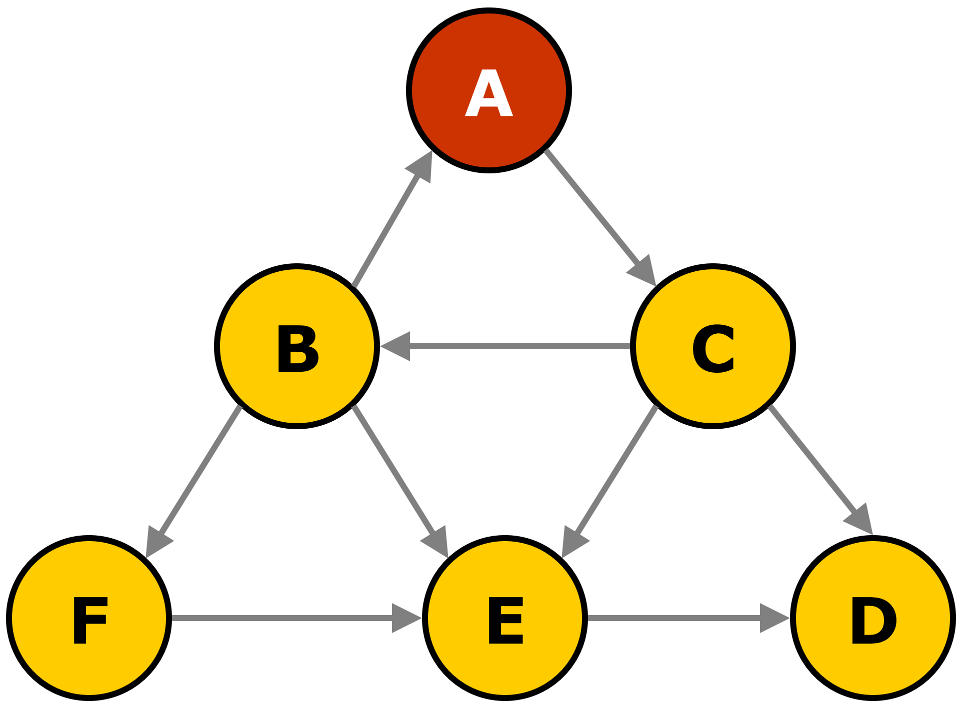
Breadth First Search (BFS) – Ideas

- Start from **s**
- BFS visits vertices of G in *breadth-first* manner (when viewed from source vertex s)
  - Q: How to maintain such order?
    - A: Use queue **Q**, initially, it contains only **s**
  - Q: How to differentiate visited vs unvisited vertices (to avoid cycle)?
    - A: 1D array/Vector **visited** of size V,
      visited[v] = 0 initially, and **visited[v] = 1** when **v** is visited
  - Q: How to memorize the path?
    - A: 1D array/Vector **p** of size V,
      p[v] denotes the **p**redecessor (or **p**arent) of **v**
- Edges used by BFS in the traversal will form a BFS “spanning” tree of G (tree that includes all

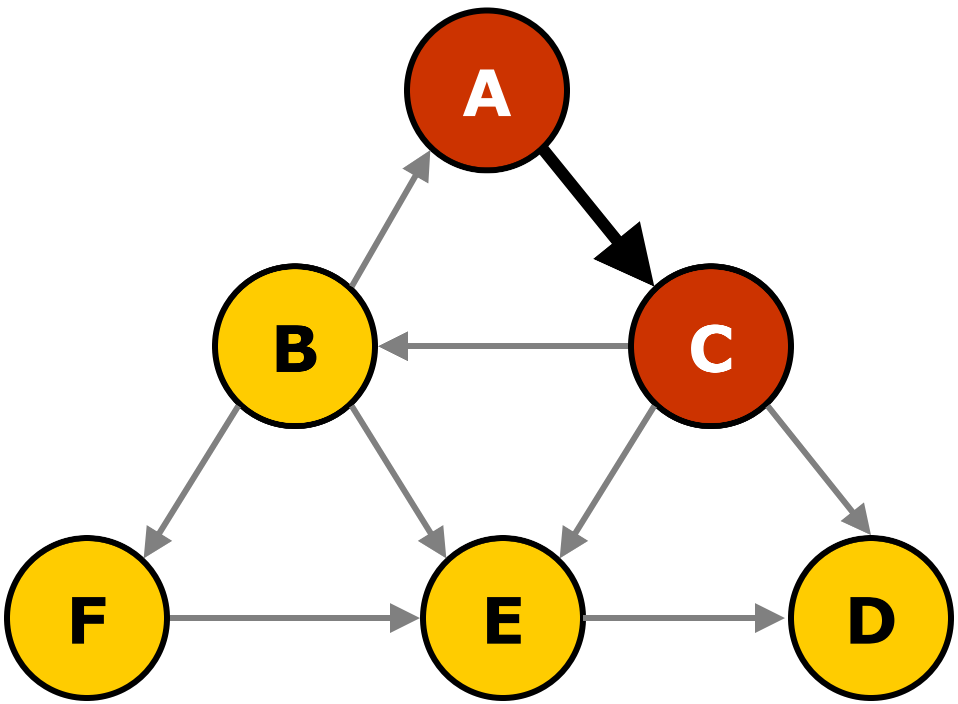
Breadth-first search



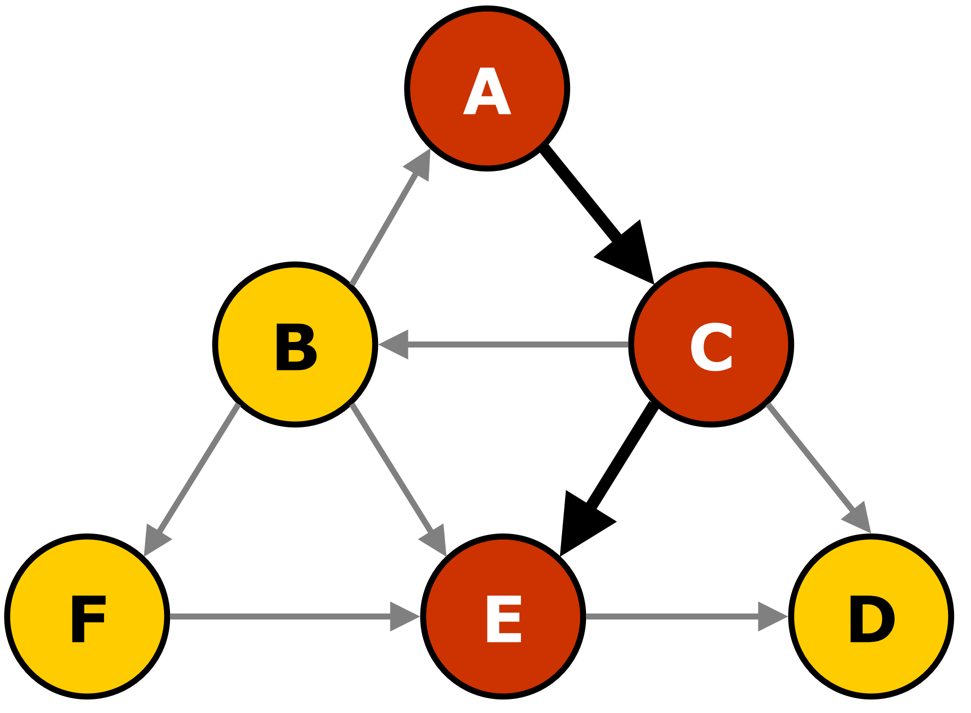
Breadth-first search



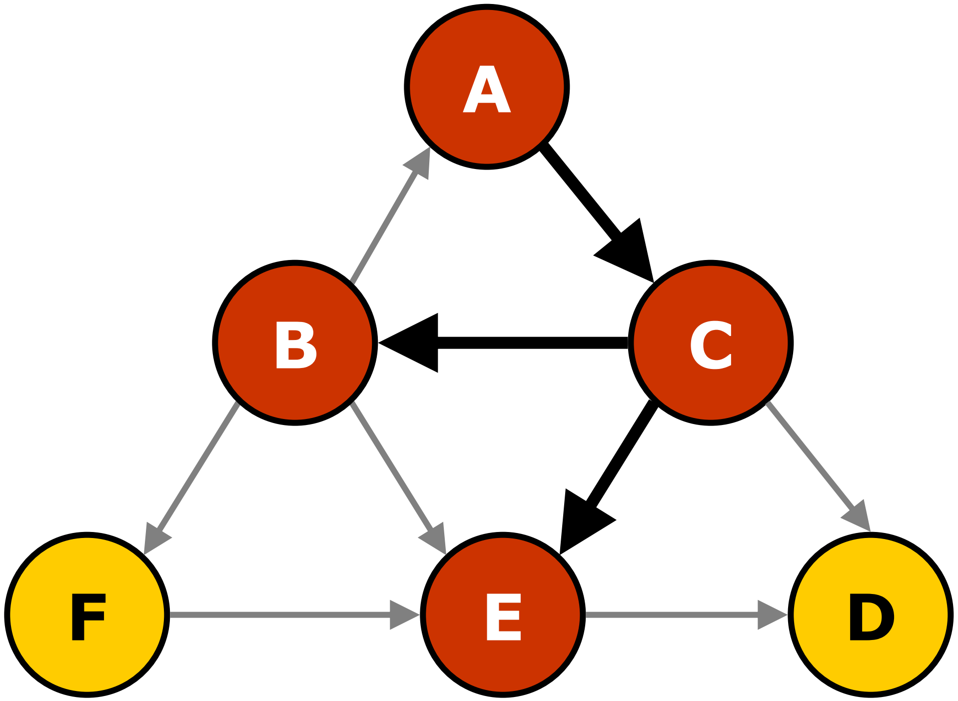
Breadth-first search



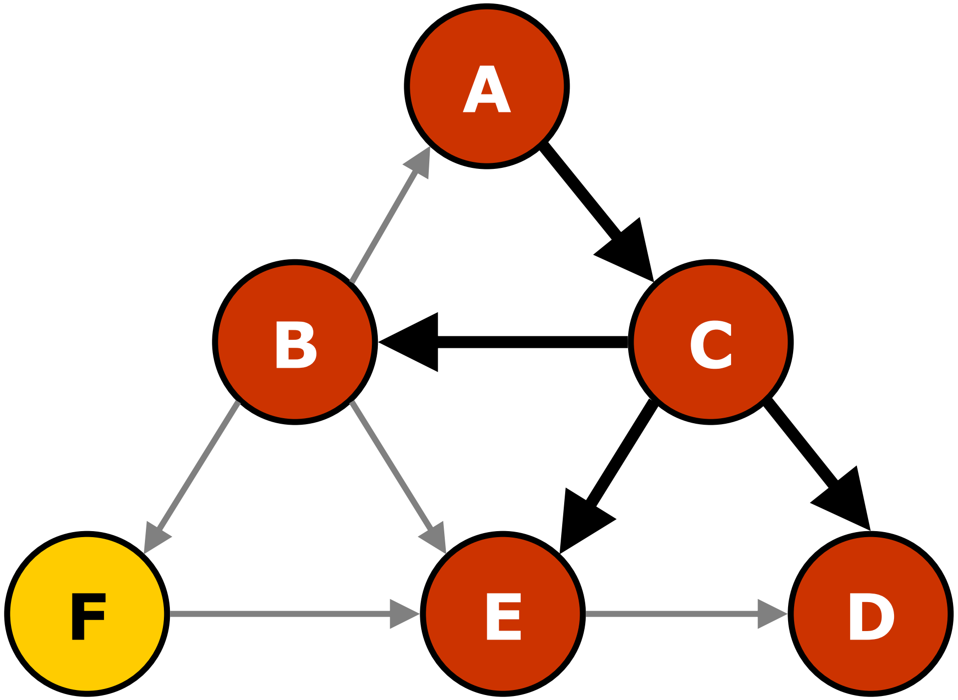
Breadth-first search



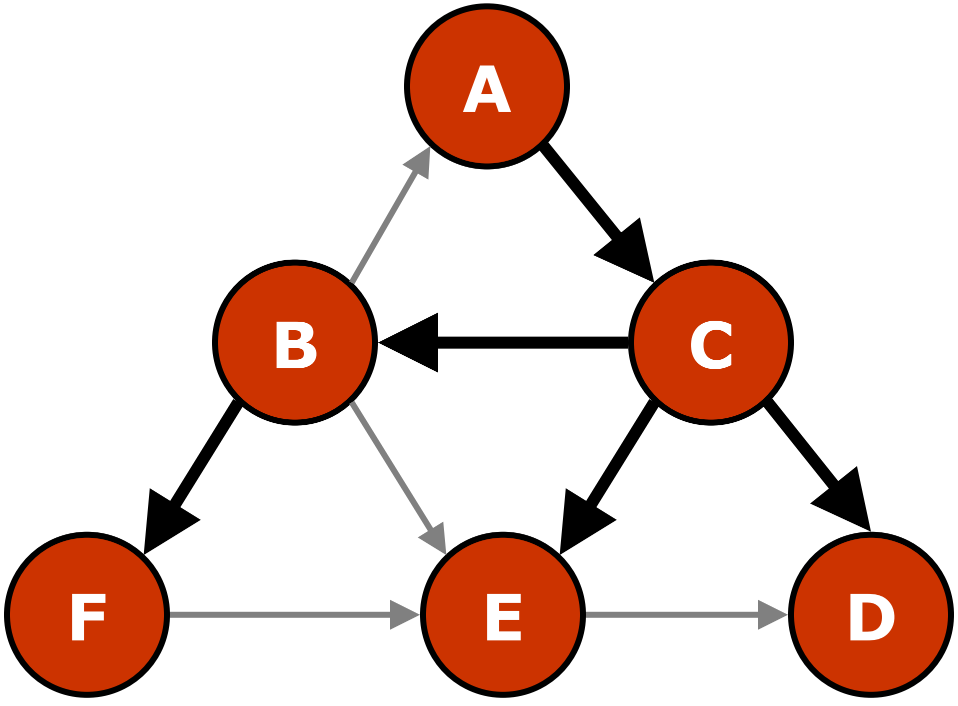
Breadth-first search



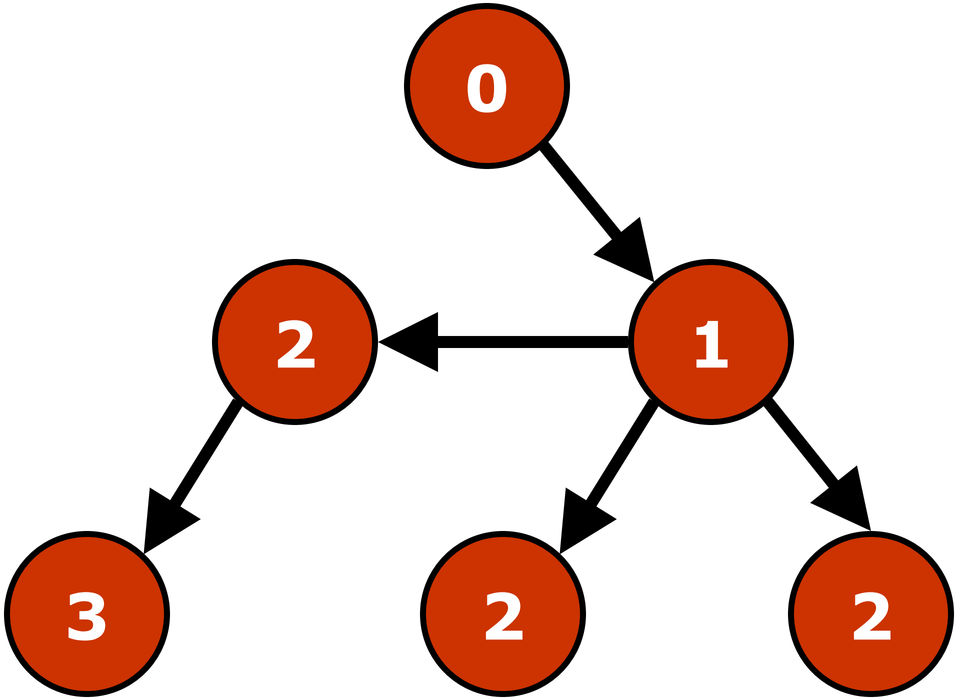
Breadth-first search



Breadth-first search

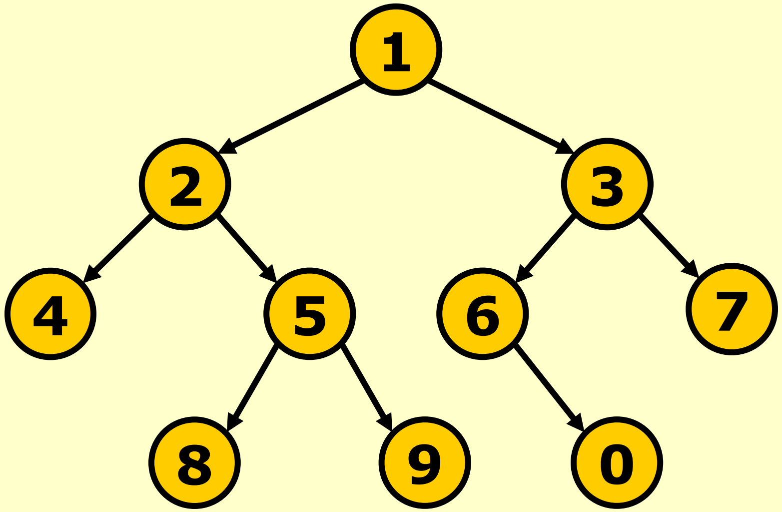


Breadth-first search



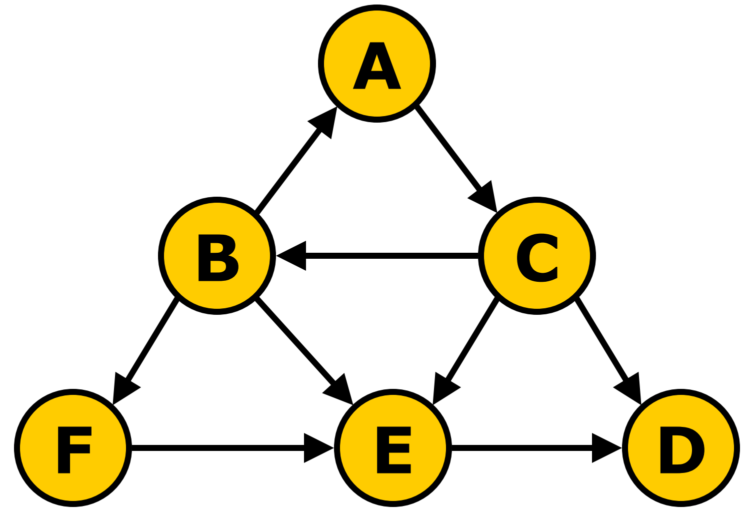
Level-Order on Tree

```
if T is empty return
Q = new Queue
Q.enq(T)
while Q is not empty
   curr = Q.deq()
   print curr.element
   if T.left is not empty
        Q.enq(curr.left)
   if curr.right is not empty
        Q.enq(curr.right)
```



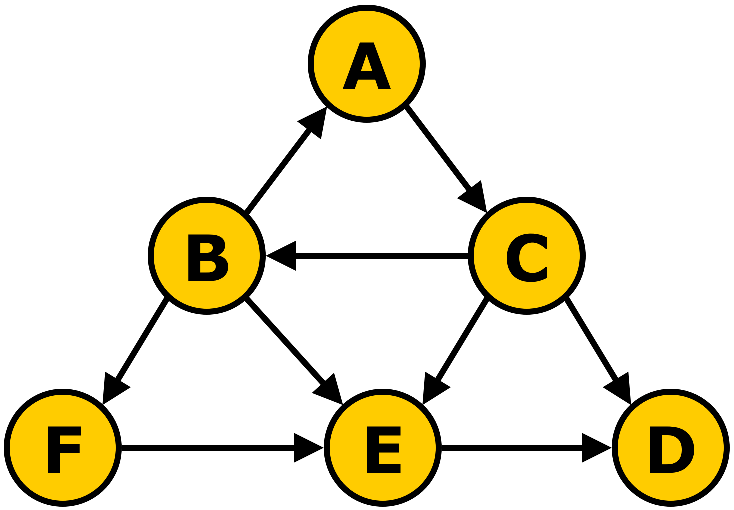
BFS(v)

```
Q = new Queue
Q.enq (v)
mark v as visited
while Q is not empty
   curr = Q.deq()
   print curr
   foreach w in adj(curr)
      if w is not visited
            Q.enq(w)
            mark w as visited
```



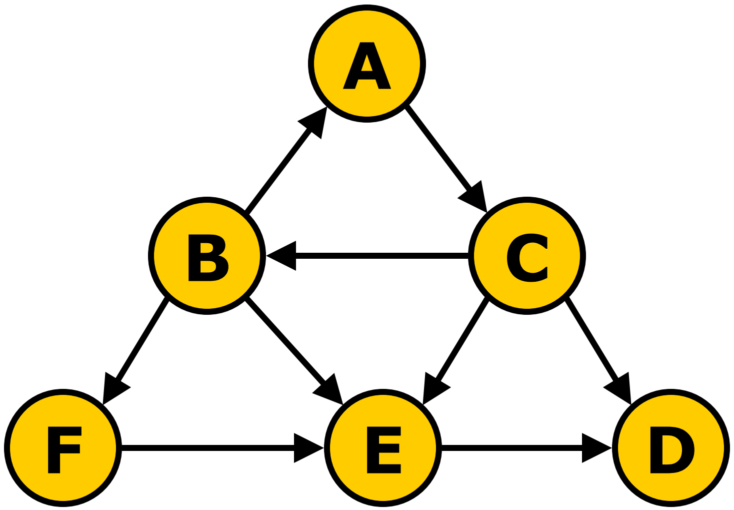
Building the BFS Tree

```
Q = new Queue
Q.enq (v)
mark v as visited
while Q is not empty
    curr = Q.deq()
    print curr
    foreach w in adj(curr)
        if w is not visited
            Q.enq(w)
            w.parent = curr
            mark w as visited
```



Calculating Level

```
Q = new Queue
Q.enq (v)
mark v as visited
v.level = 0
while Q is not empty
   curr = Q.deq()
   print curr
   foreach w in adj(curr)
      if w is not visited
         Q.enq(w)
         w.level = curr.level + 1
         mark w as visited
```



Search all vertices

Search(G)

foreach vertex *v*

mark *v* as unvisited

foreach vertex *v*

if *v* is not visited

BFS(*v*)

Running time

```
Q = new Queue
Q.enq (v)
mark v as visited
while Q is not empty
    curr = Q.deq()
    print curr
    foreach w in adj(curr)
        if w is not visited
            Q.enq(w)
            mark w as visited
```

Main Loop

$$O\left(\sum_{curr \in V} adj(curr) \right) = O(E)$$

Initialization

$$O(V)$$

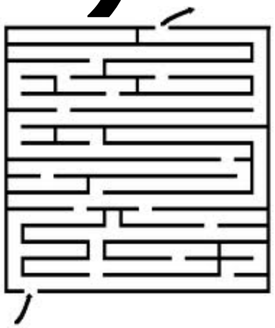
Total Running Time

$$O(V + E)$$

Depth-First Search

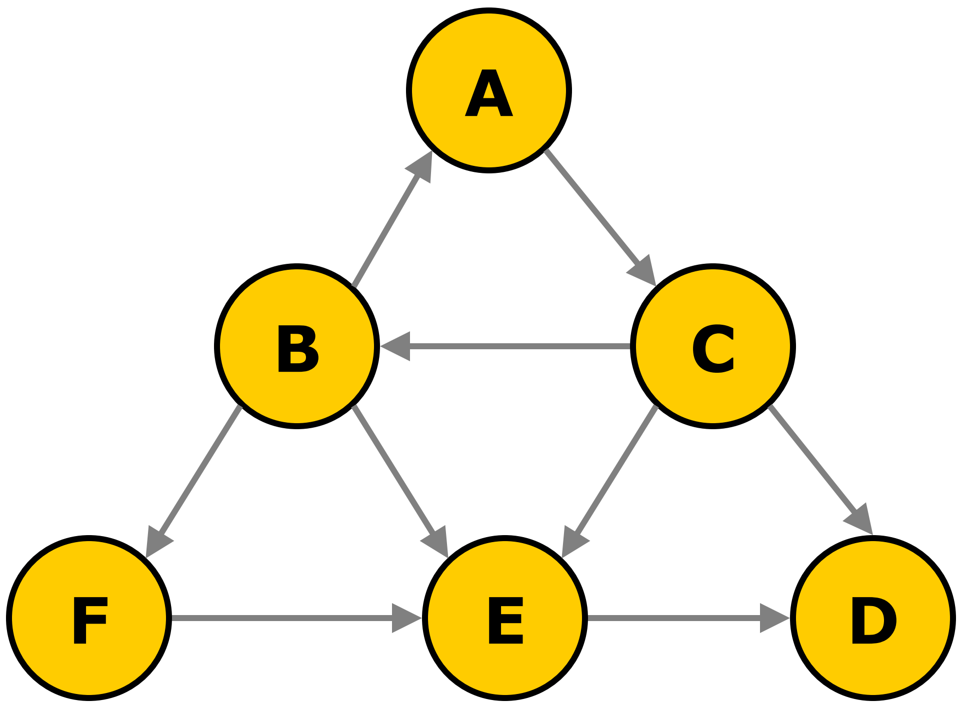
Traversing a Graph

Depth First Search (DFS) – Ideas

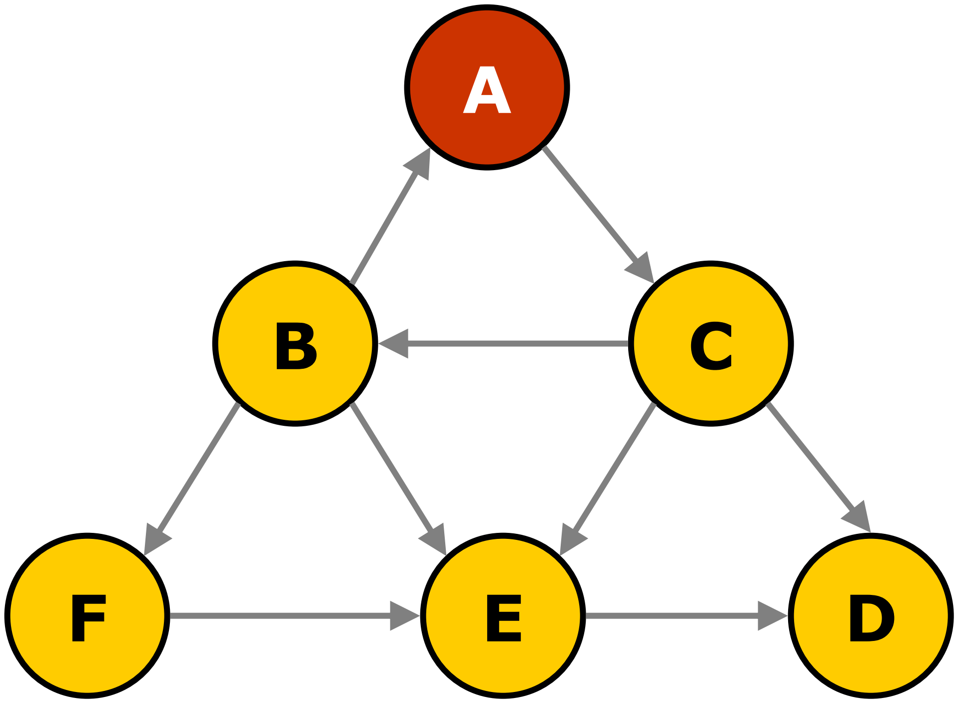


- Start from **s**
- DFS visits vertices of G in *depth-first* manner (when viewed from source vertex s)
  - Q: How to maintain such order?
    - A: Stack **S**, but we will simply use recursion (an implicit stack)
  - Q: How to differentiate visited vs unvisited vertices (to avoid cycle)?
    - A: 1D array/Vector **visited** of size V,
      visited[v] = 0 initially, and **visited[v] = 1** when **v** is visited
  - Q: How to memorize the path?
    - A: 1D array/Vector **p** of size V,
      p[v] denotes the **p**redecessor (or **p**arent) of **v**
- Edges used by DFS in the traversal will form a DFS "spanning" tree of G (tree that includes all vertices of G) stored in **p**

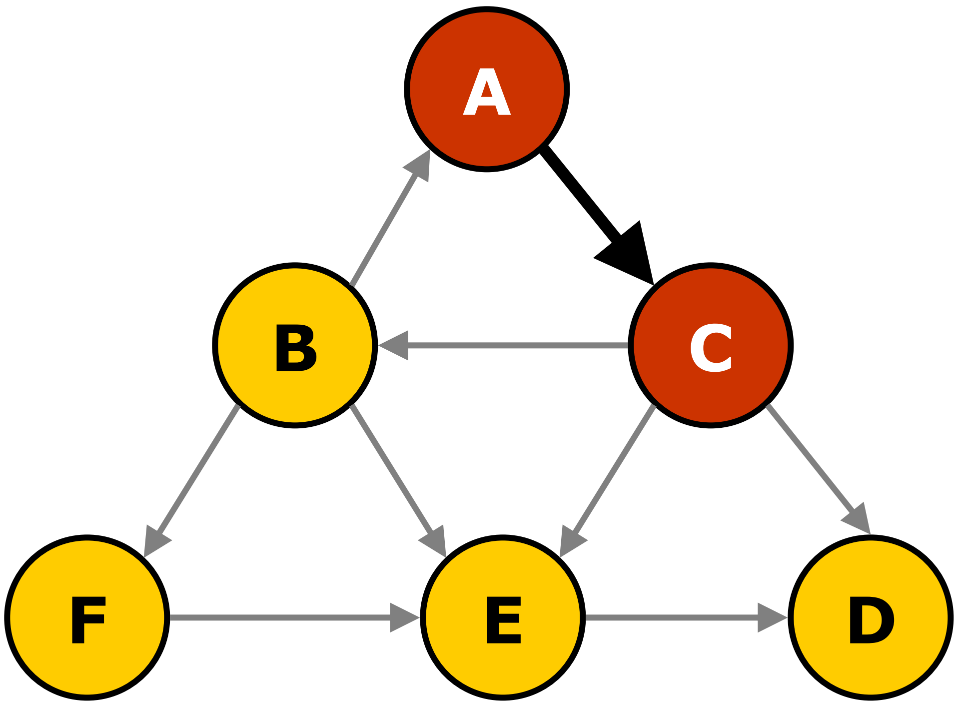
Depth-first search



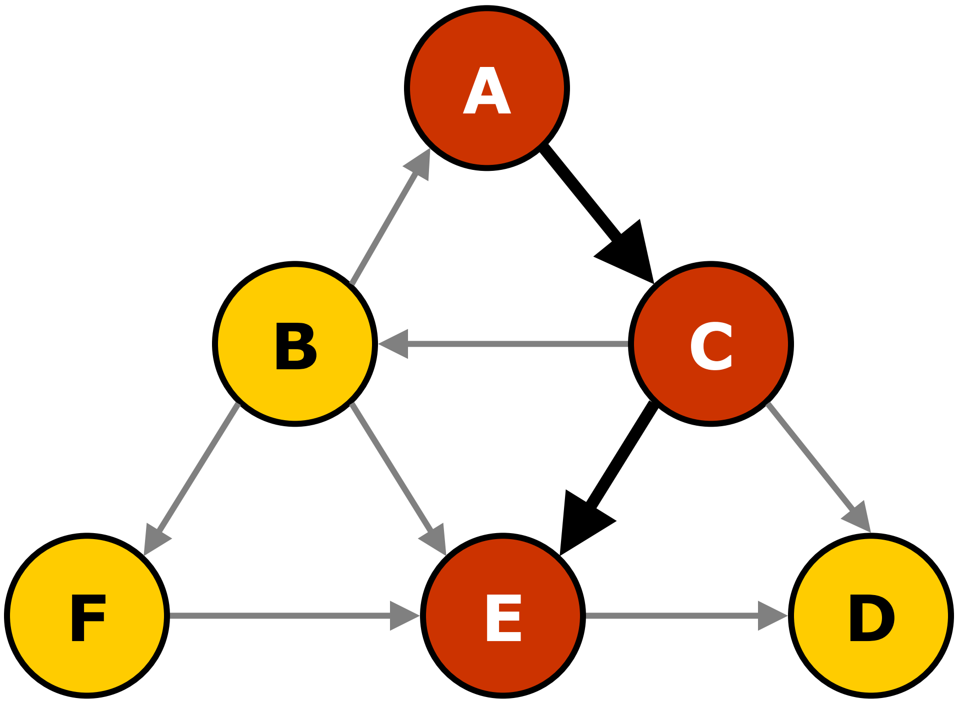
Depth-first search



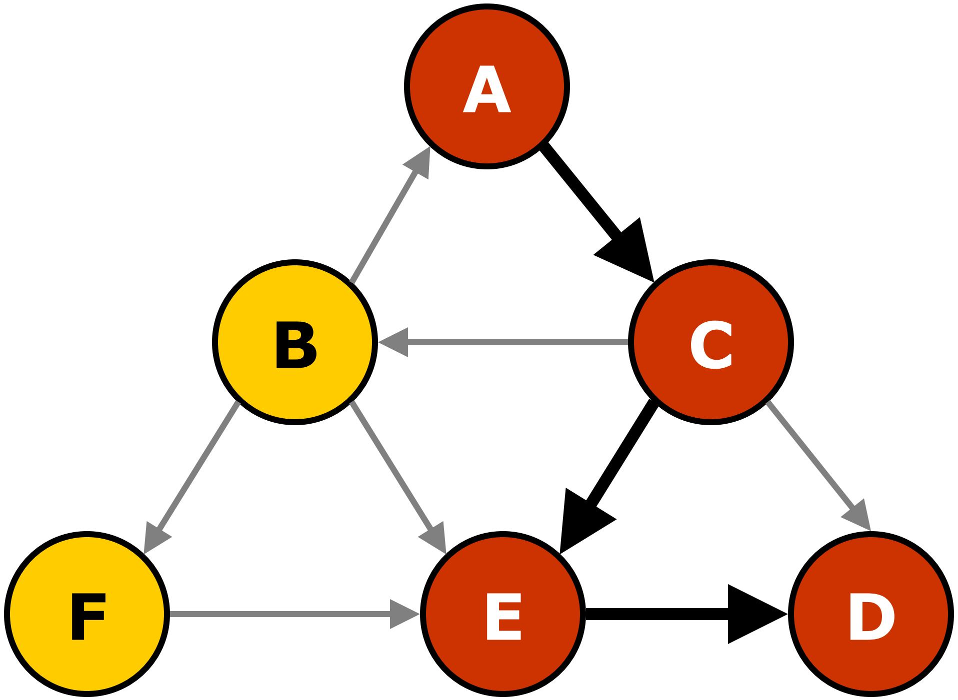
Depth-first search



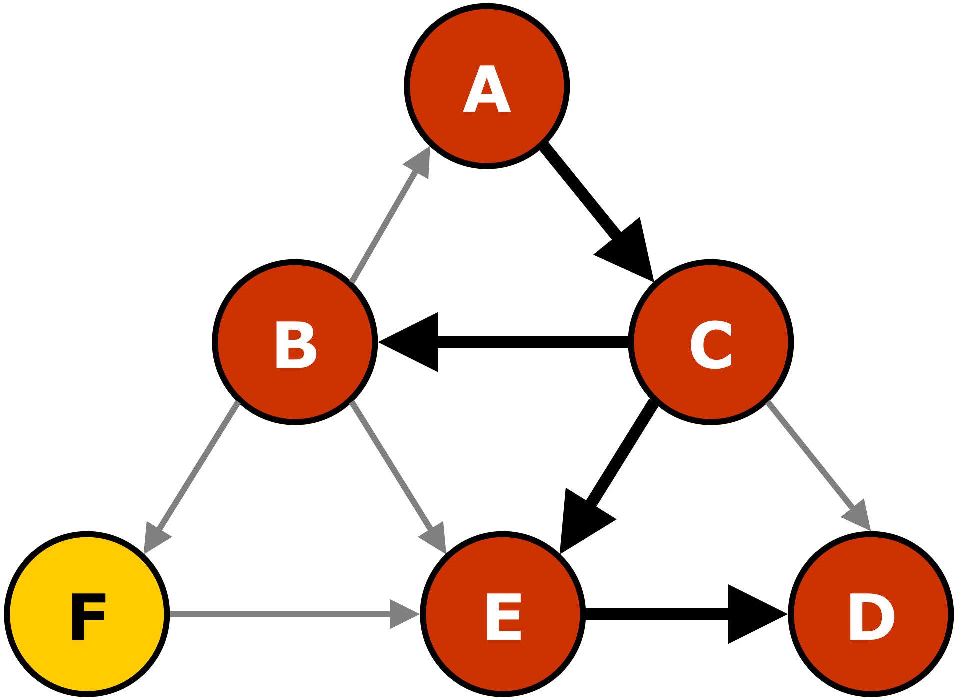
Depth-first search



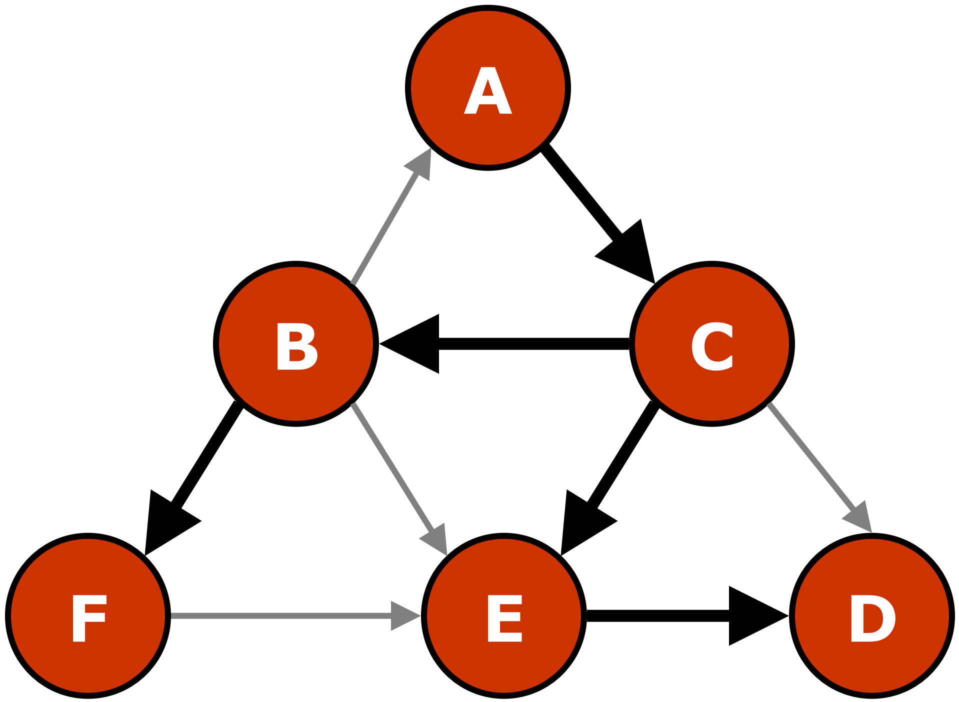
Depth-first search



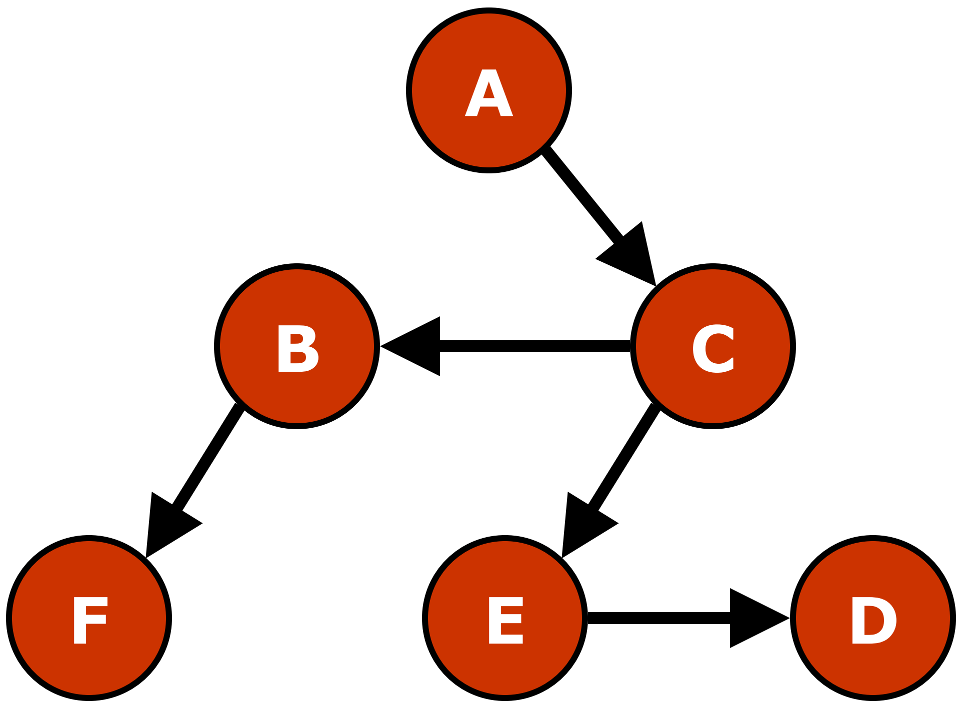
Depth-first search



Depth-first search

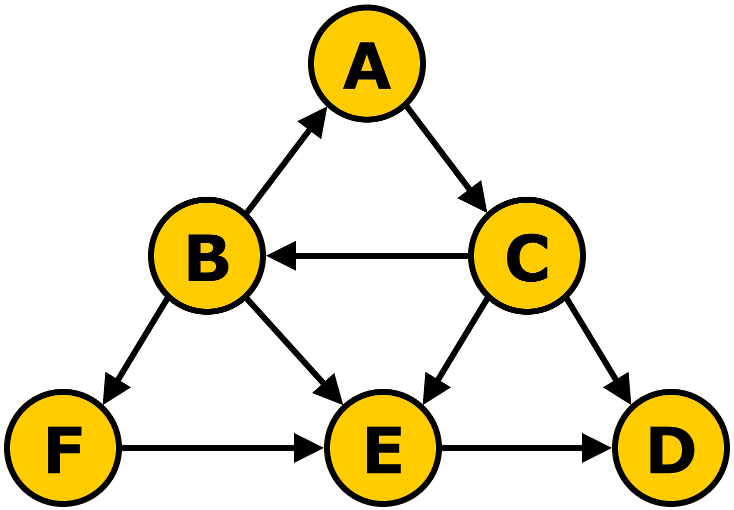


Depth-first search



DFS(v)

```
S = new Stack
S.push (v)
Print and mark v as visited
while S is not empty
    curr = S.top()
    if every vertex in adj(curr)
        is visited
            S.pop()
    else
            let w be an unvisited vertex in adj(curr)
            S.push(w)
            print and mark w as visited
```



Recursive version: DFS(v)

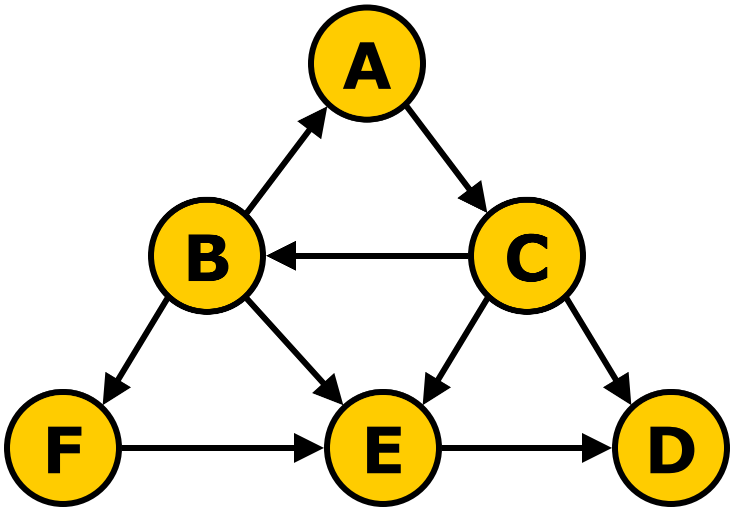
print v

marked v as visited

foreach w in adj(v)

  if w is not visited

    DFS(w)



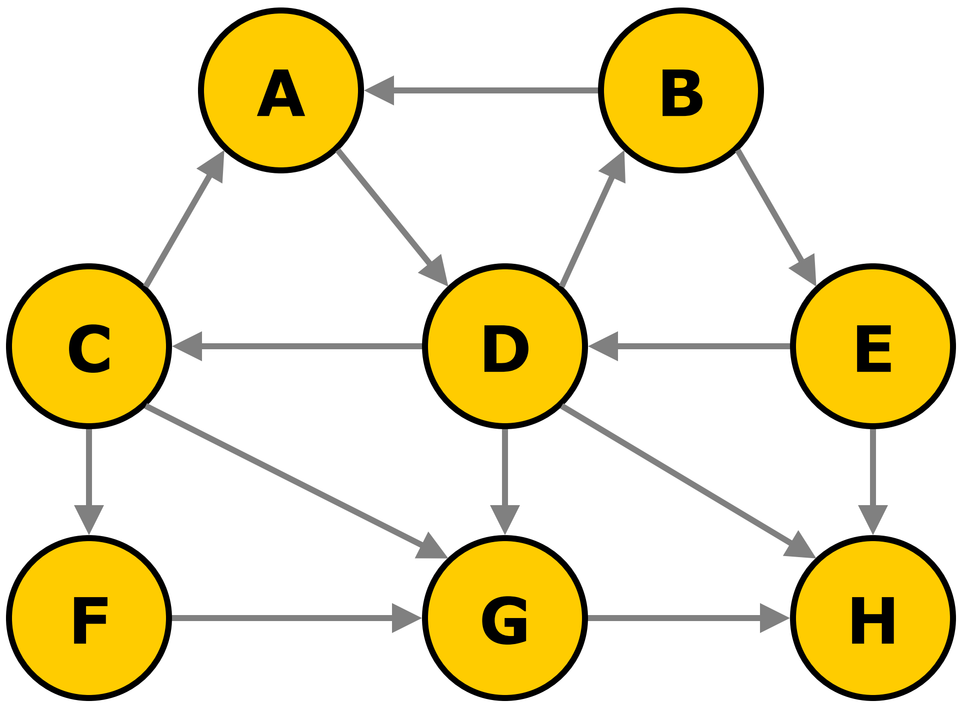
Search all vertices

```
Search(G)
  foreach vertex v
      mark v as unvisited
  foreach vertex v
      if v is not visited
          DFS(v)
```

Running time

- DFS: $\Theta(V + E)$

Two more times!

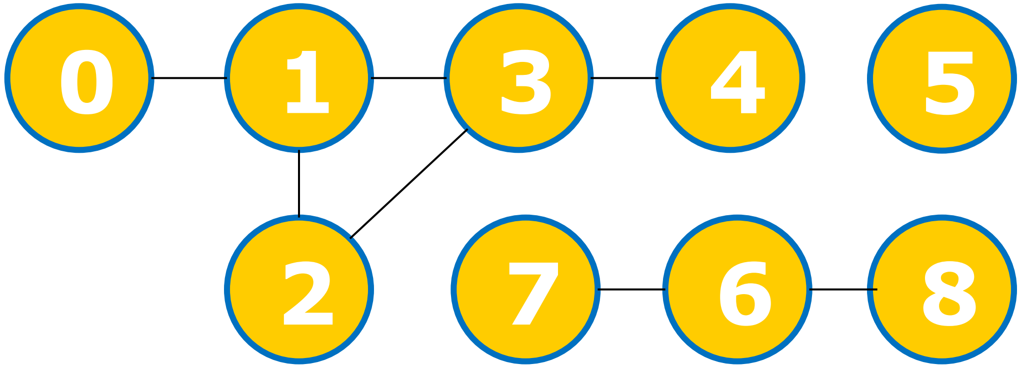


What can we do with BFS/DFS? (1)

Several tasks, let's see ***some of them***:

- Reachability test
  - Test whether vertex **v** is reachable from vertex **u**?
  - Start BFS/DFS from **s = u**
  - If **visited[v] = 1** after BFS/DFS terminates, then **v** is *reachable* from **u**; otherwise, **v** is *not reachable* from **u**

```
BFS(u) // DFSrec(u)
if visited[v] == 1
  Output "Yes"
else
  Output "No"
```



What can we do with BFS/DFS? (2)

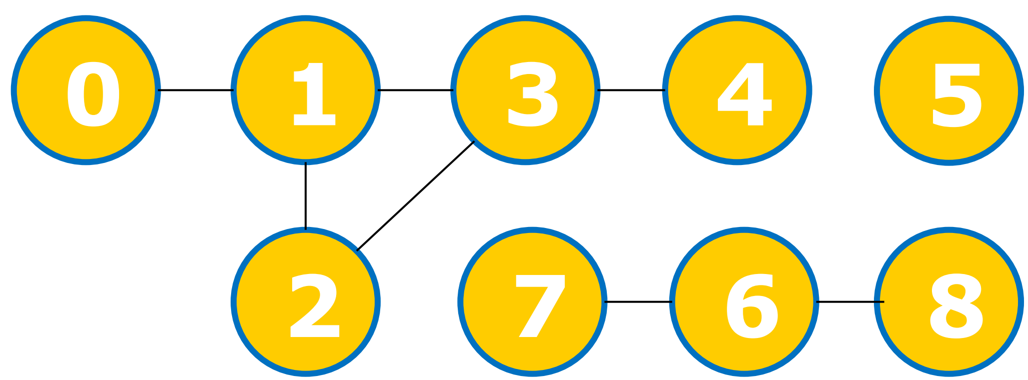
- Find Shortest Path between 2 vertices in an unweighted graph/graph where edges have same weight
  - When the graph is **unweighted*/edges have same weight**, shortest path between any 2 vertices **u**,**v** is finding the **least number of edges** traversed from u to v
  - The O(**V**+**E**) Breadth First Search (BFS) traversal algorithm precisely measures this
    - Run BFS from u as source
    - Construct shortest path from u to v from **p** after BFS finishes
    - Cost of shortest path from u to v is (number of edges in the path)×(edge weight for weighted edges)

* Can treat the edge weight as 1

What can we do with BFS/DFS? (3)

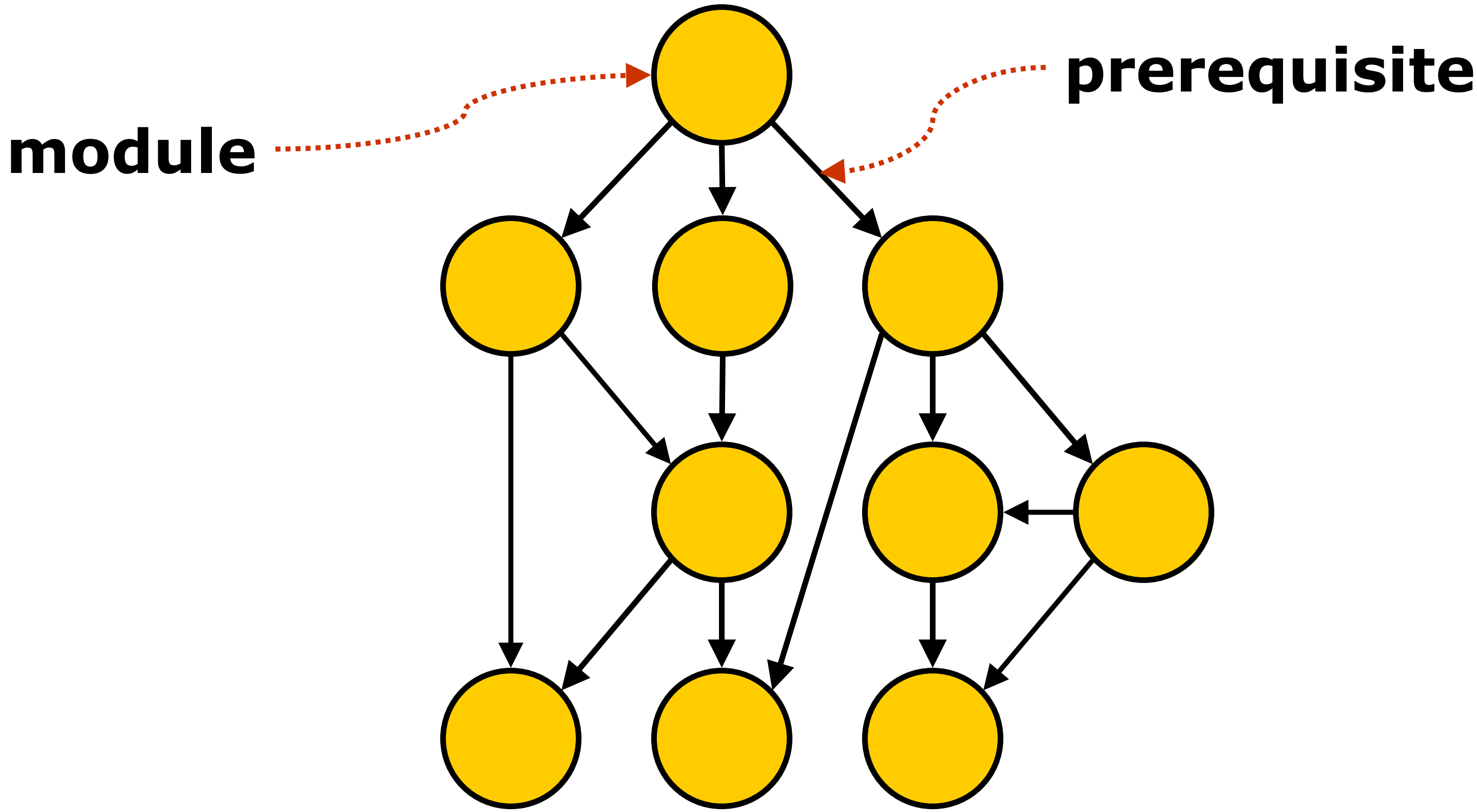
- Identifying component(s)
  - Component is sub graph containing 1 or more vertices in which any 2 vertices are connected to each other by at least one path, and is connected to no additional vertices
  - With BFS/DFS, we can identify components by labeling/counting them in graph G
  - Solution:

```
CC ← 0
for all v in V
  visited[v] ← 0
for all v in V // O(V)?
  if visited[v] == 0
    CC ← CC + 1
    DFSrec(v)//O(V+E)?
    // BFS from v
    // is also OK
```



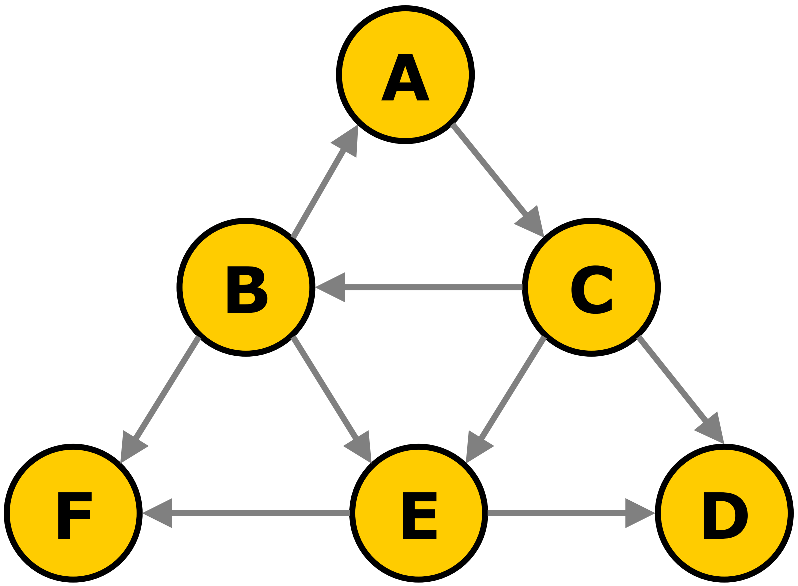
Topological Sort

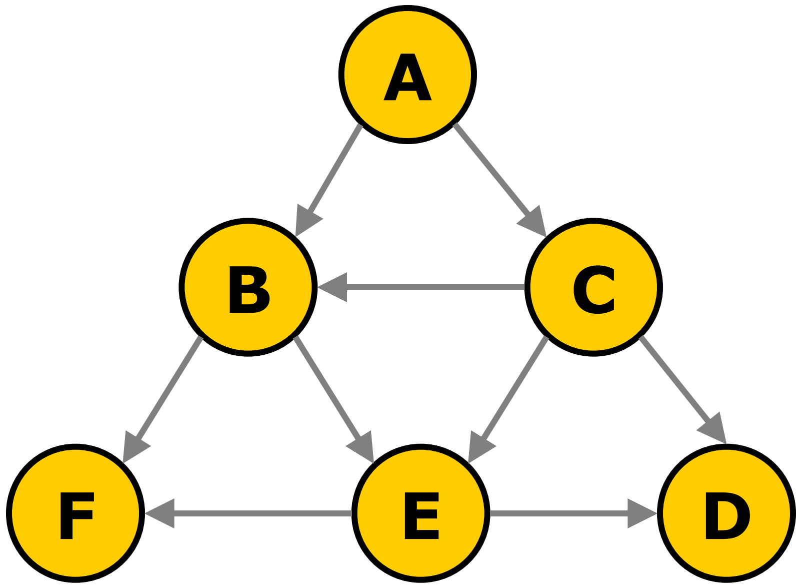
Module selection



Definition

- Directed Acyclic Graph (DAG): A directed graph with no cycle.





Definition

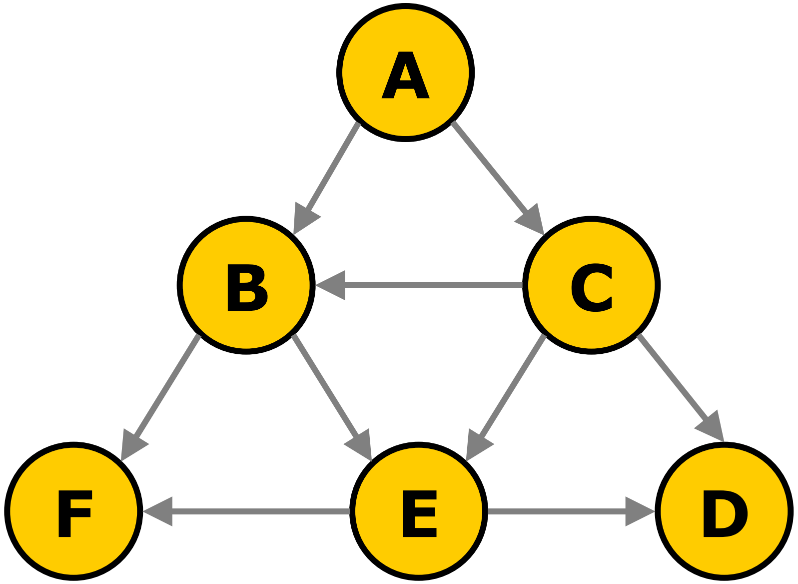
- in-degree of a vertex
  - number of incoming edges
- out-degree of a vertex
  - number of outgoing edges

Topological sort

- Goal: Order the vertices, such that if there is a path from u to v, u appears before v in the output.

Topological sort

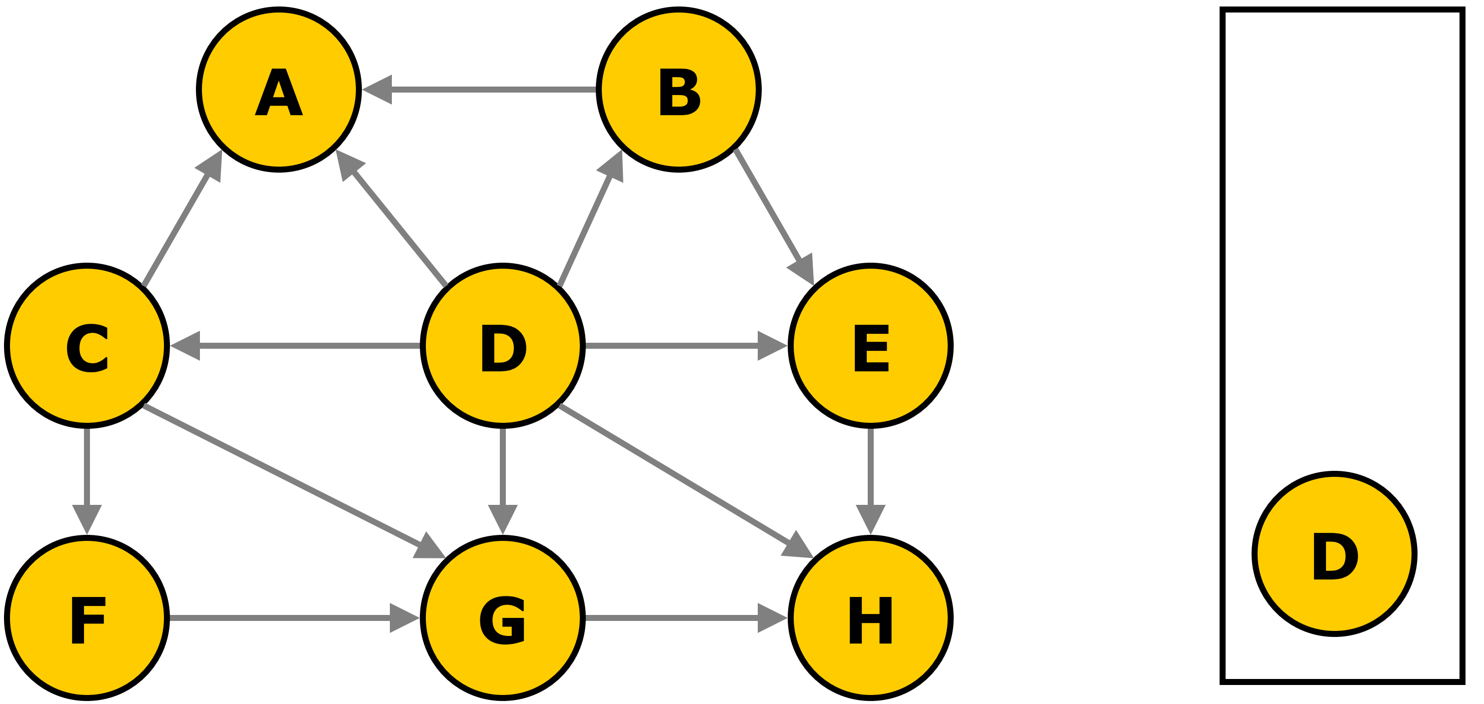
- ACBEFD
- ACBEDF
- ACDBEF



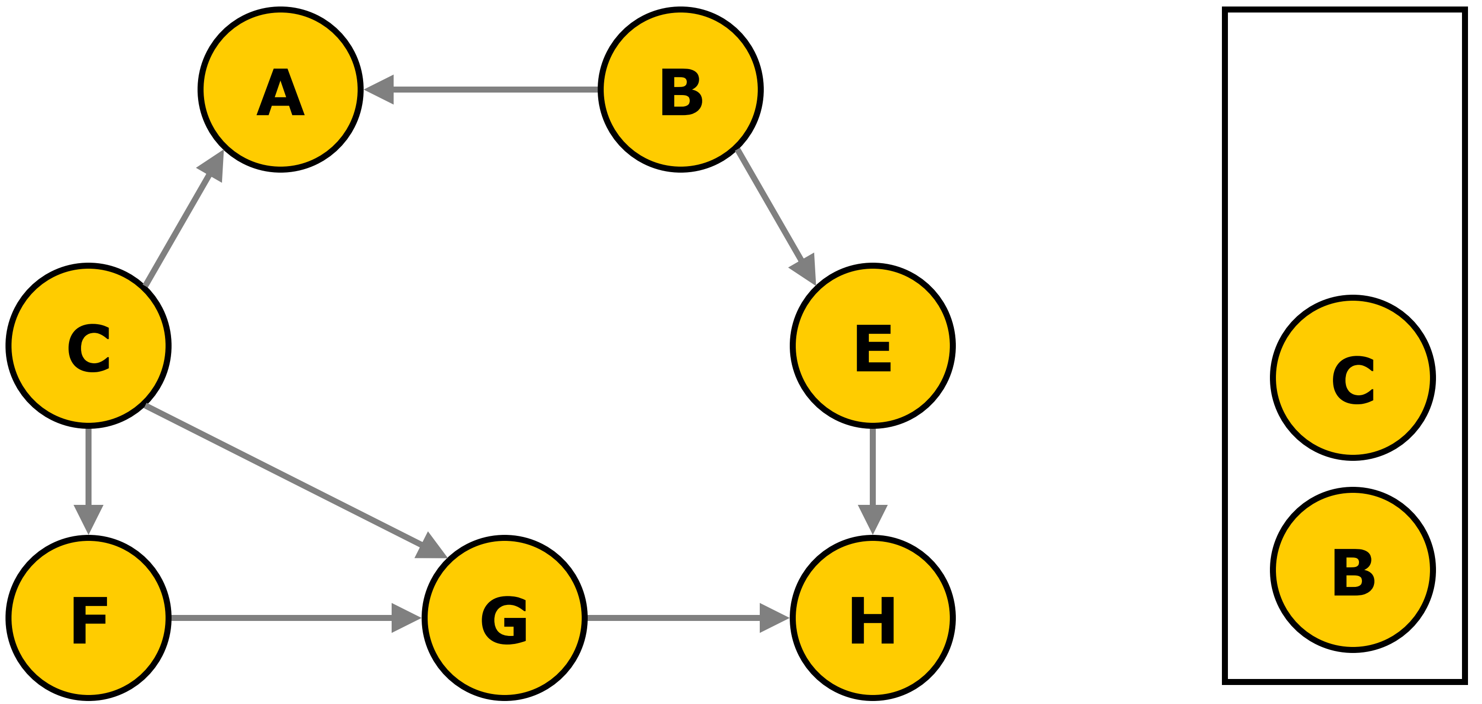
Pseudocode for Toposort

```
q = new Queue()
put all vertices with in-degree 0 into q
while q is not empty
   v = q.deq()
   print v
   remove v from G
   enqueue neighbours of v with in-degree 0
```

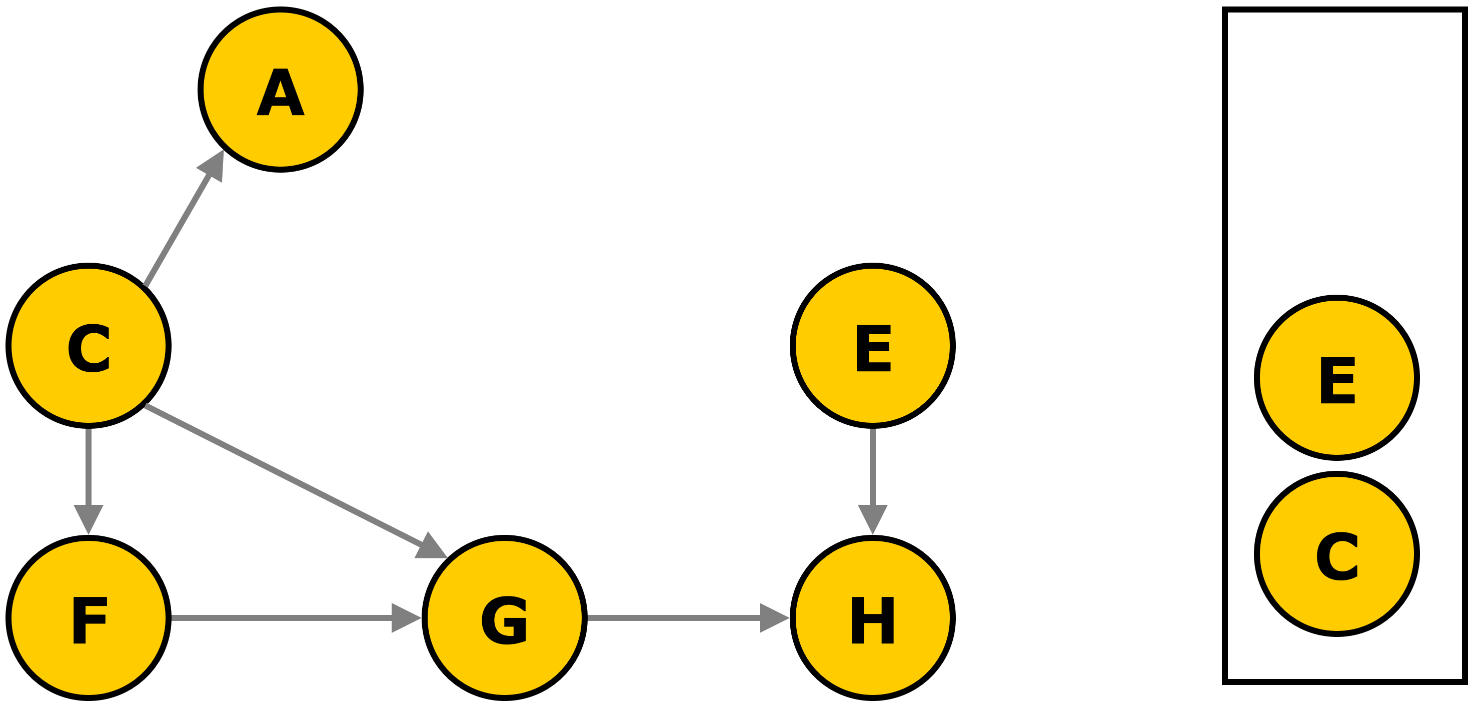
Example



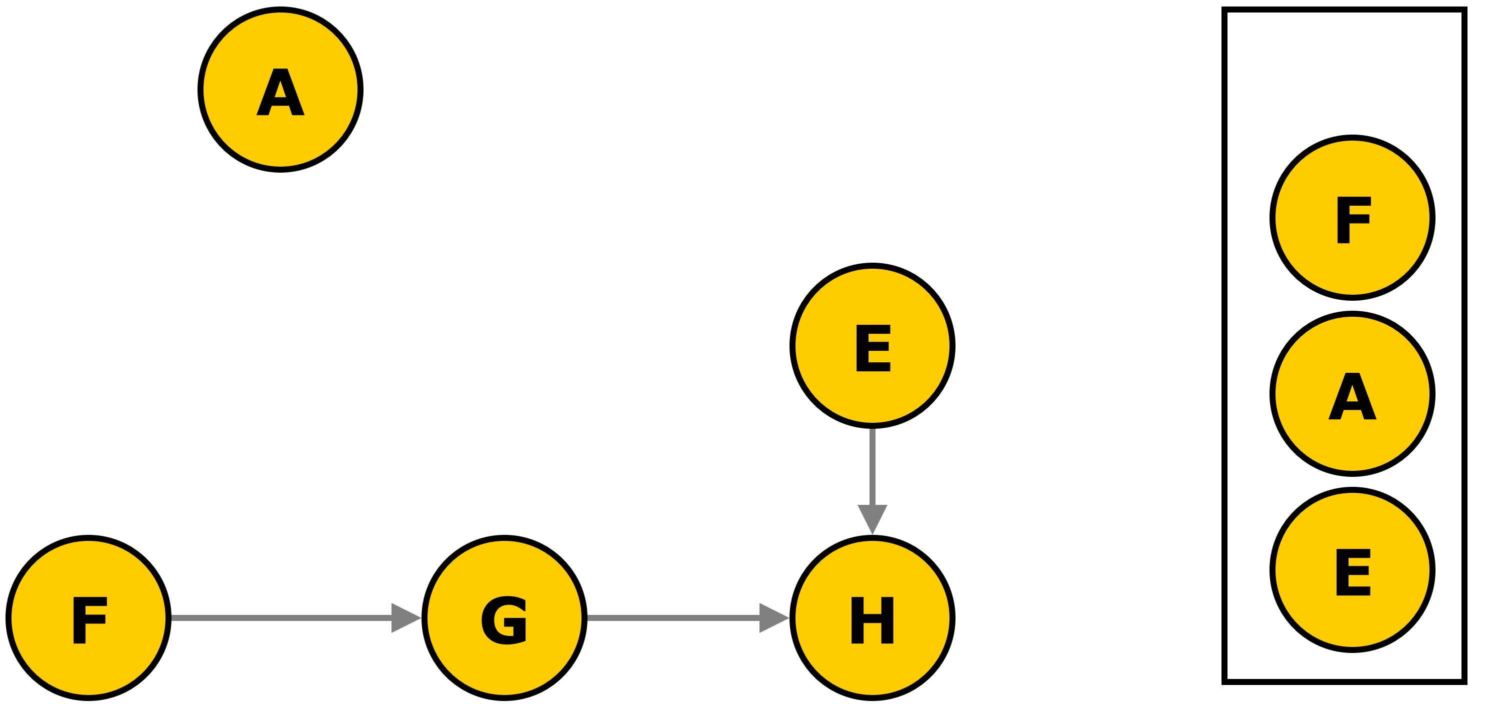
Output: D



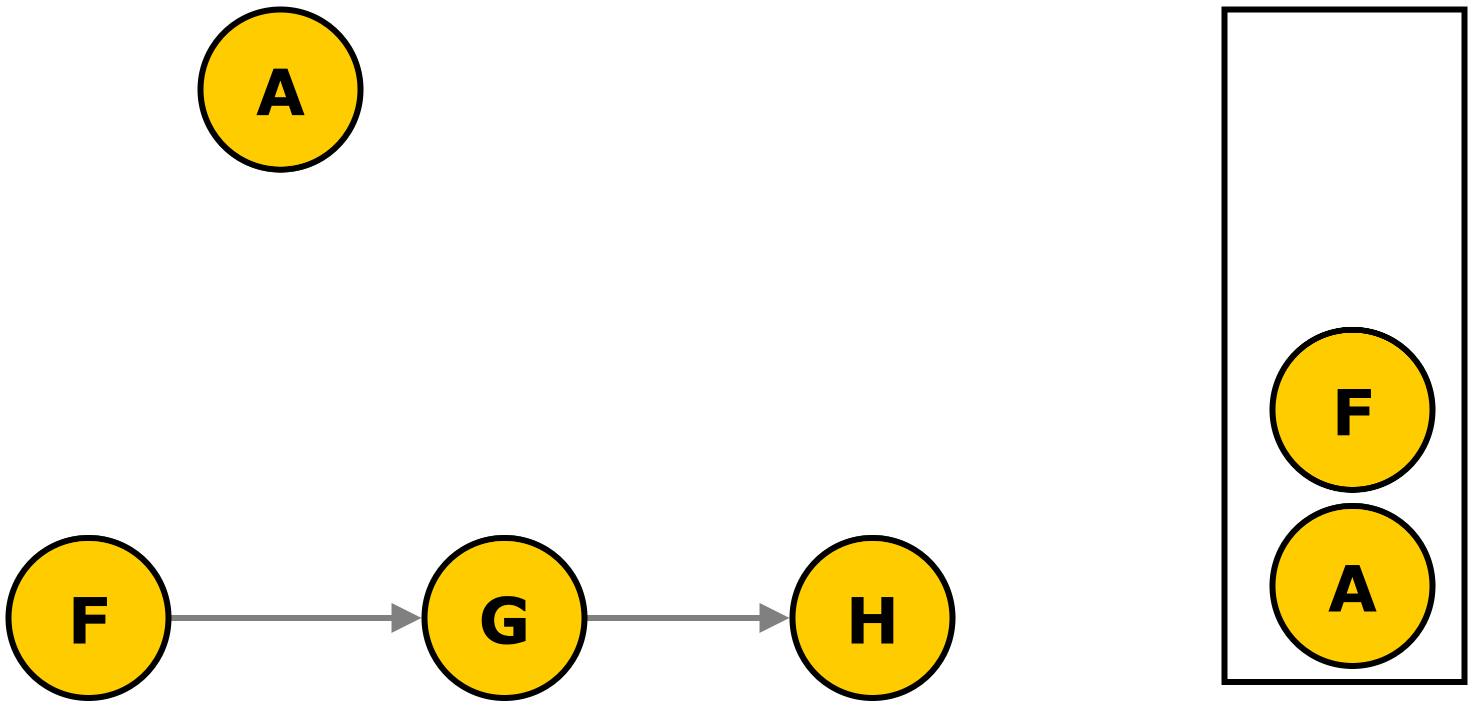
Output: DB



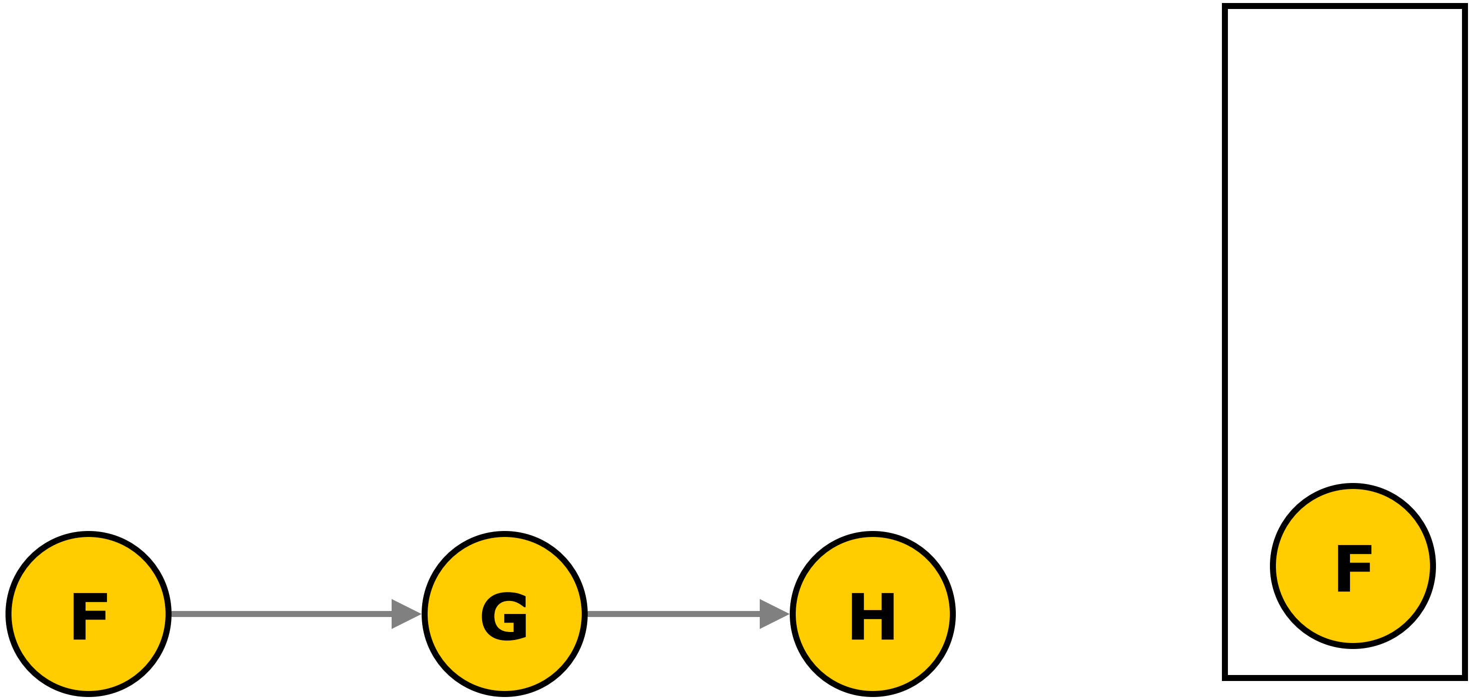
Output: DBC



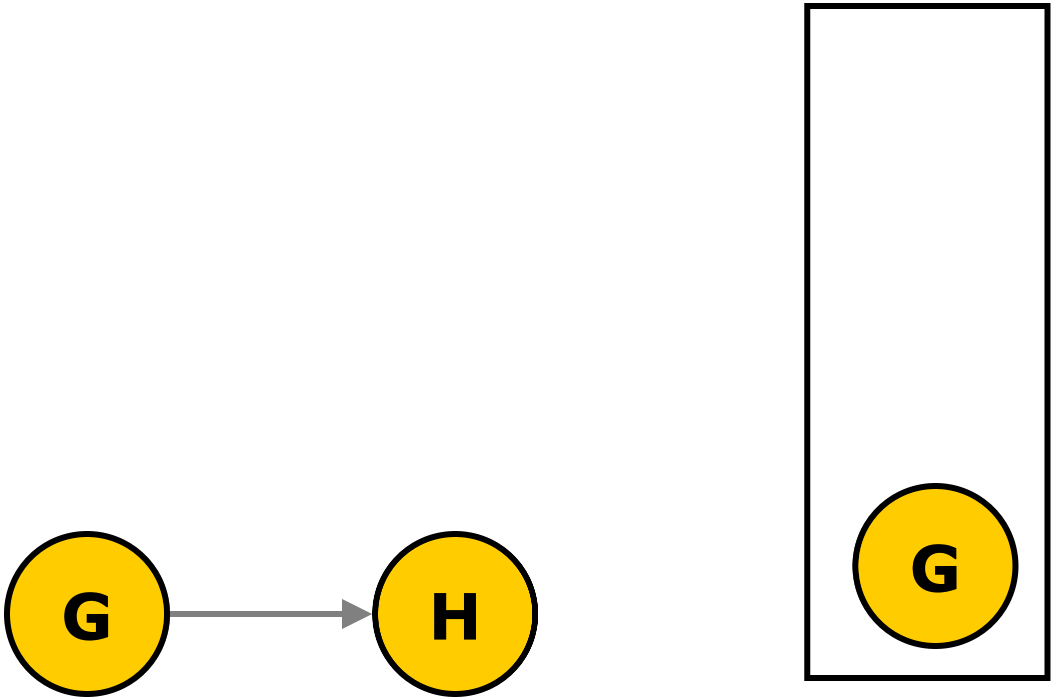
Output: DBCE



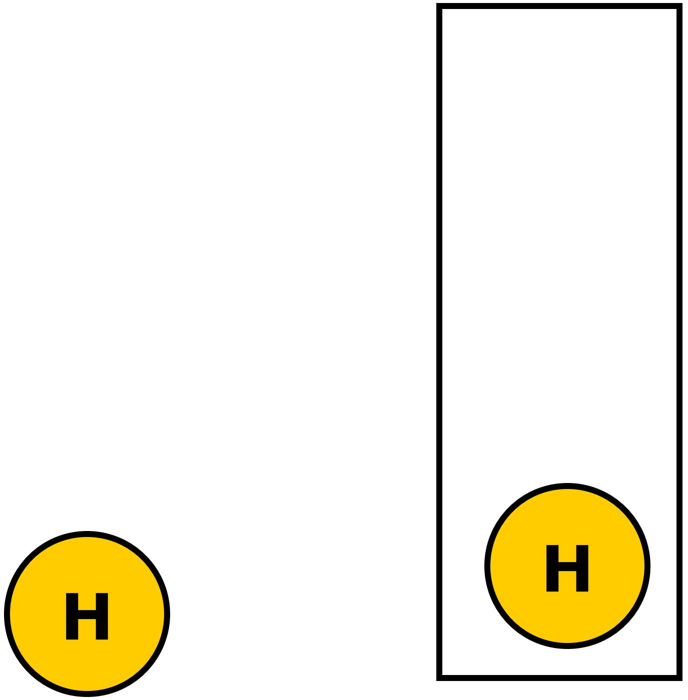
Output: DBCEA



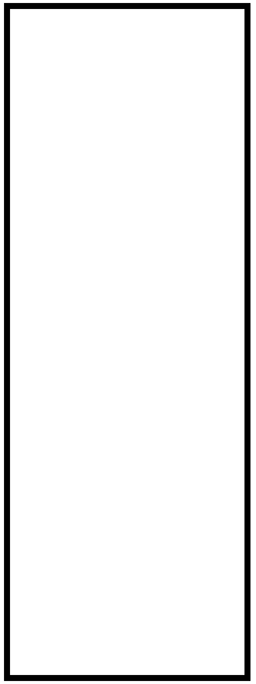
Output: DBCEAF



Output: DBCEAFG



Output: DBCEAFGH



Pseudocode for Toposort

```
q = new Queue()
put all vertices with in-degree 0 into q
while q is not empty
   v = q.deq()
   print v
   remove v from G
   enqueue neighbours of v with in-degree 0
```

Which DS to use?

What is the pre-process?

What is the complexity?