Graphs

Graph Terminology

Searching Graphs

Menu

- Graph Terminology
- Graph Modeling
- Searching
  - Breadth First Search
  - Depth First Search

Graphs :: Terminology

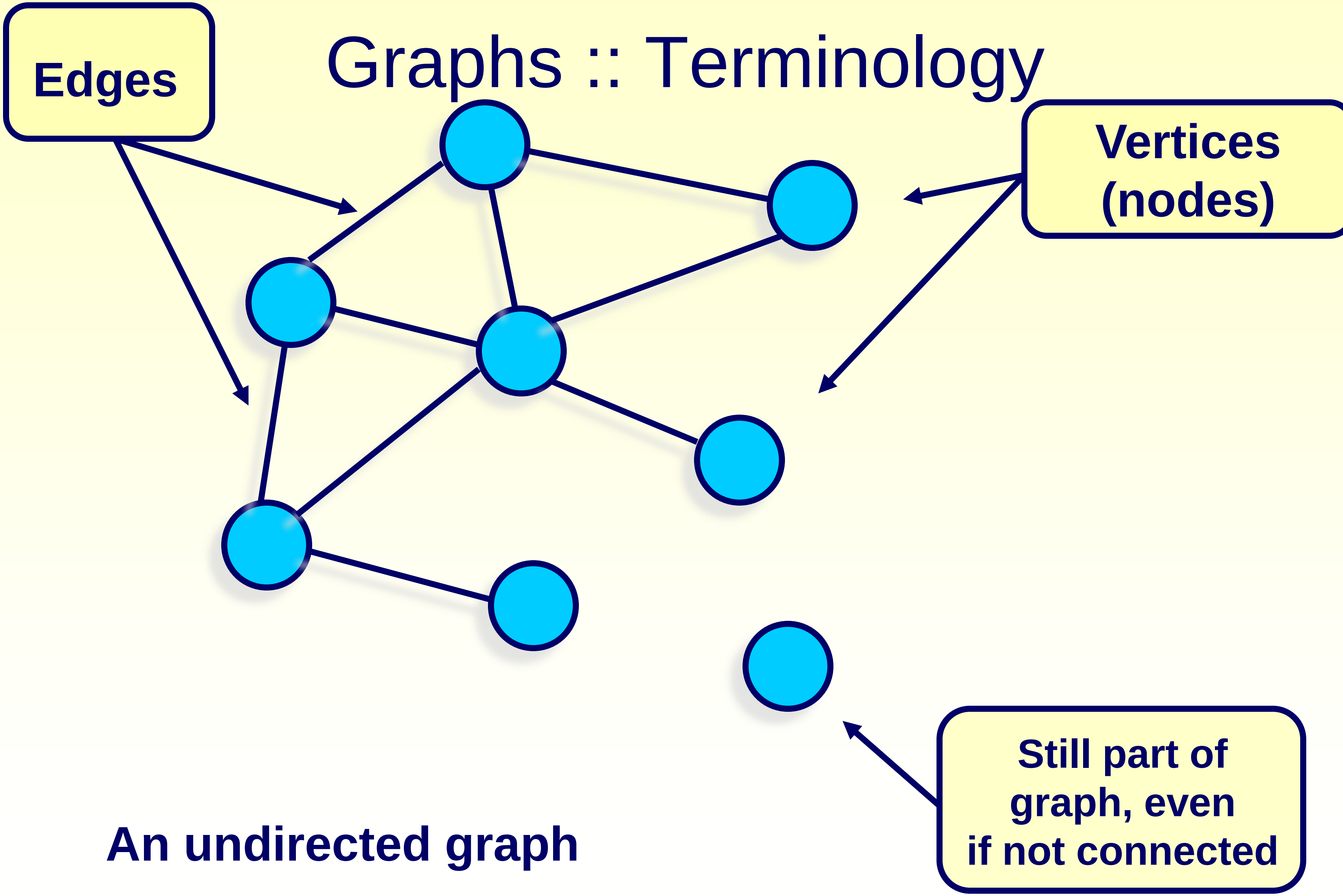
A *Graph* is a set of *vertices* (nodes) and a set of unordered *edges* (linked between these nodes).

The *order* of a graph is the number of vertices and the *size* is the edge count. The *degree* of a vertex is the number of edges incident to the vertex. (In-degree + out-degree = degree of a digraph vertex.) If all degrees are the same, then the graph is said to have that degree.

A *path* is a set of edges connecting two nodes.

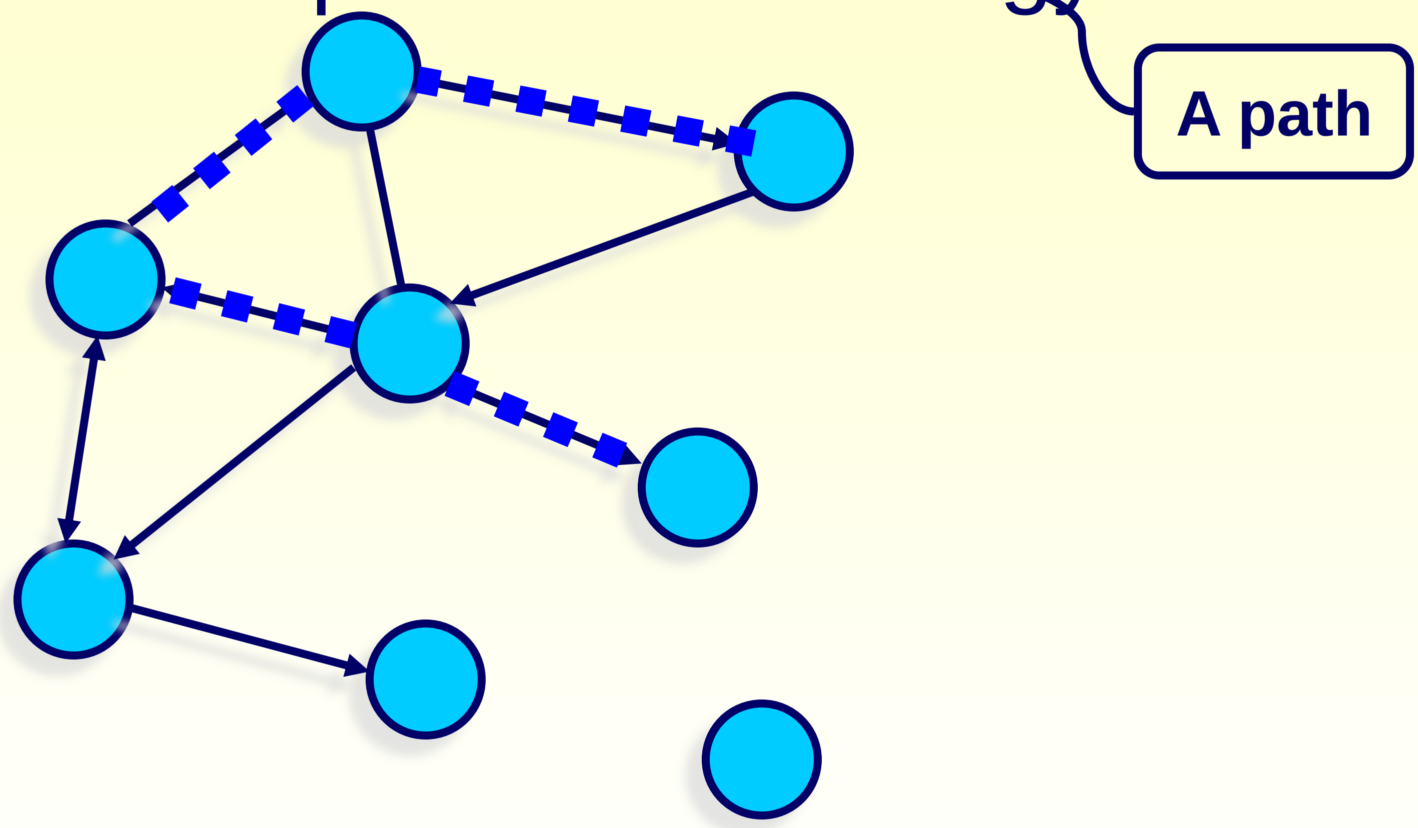
A *digraph* or *directed* graph has edges (arcs) that flow in only one direction.   In an *undirected* graph, edges flow in either direction.

Graphs :: Terminology



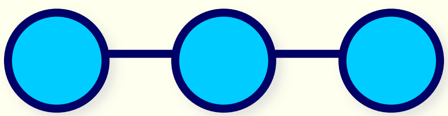
An undirected graph

Graphs :: Terminology



A directed graph

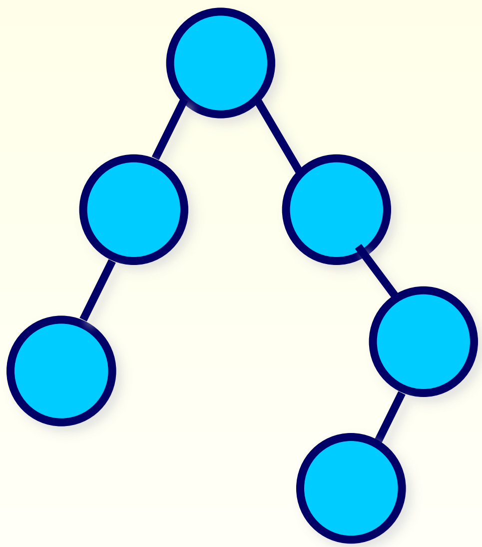
Graphs :: Contrasted to Simple Data Structures



Linked list

One 'next' node

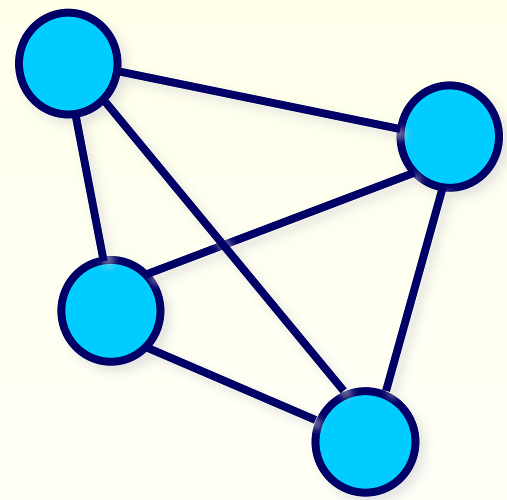
No cycles



Binary tree

Two children (for binary tree)

No cycles



Graph

Cycles allowed

Numerous adjacencies per node

Review:

~Graphs :: Terminology

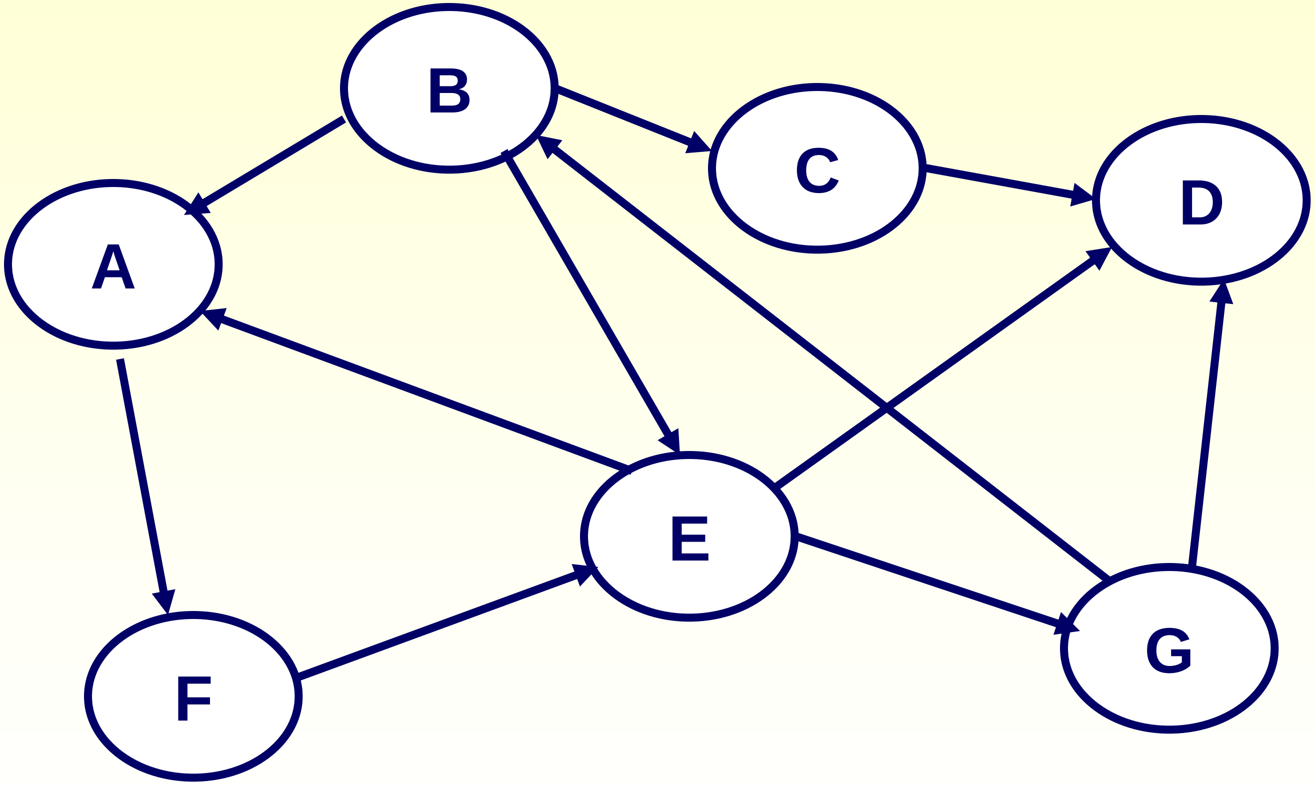
A *Graph* is a set of *vertices* (nodes) and a set of unordered *edges* (linked between these nodes).

The *order* of a graph is the number of vertices and the *size* is the edge count.

A *path* is a set of edges connecting two nodes.

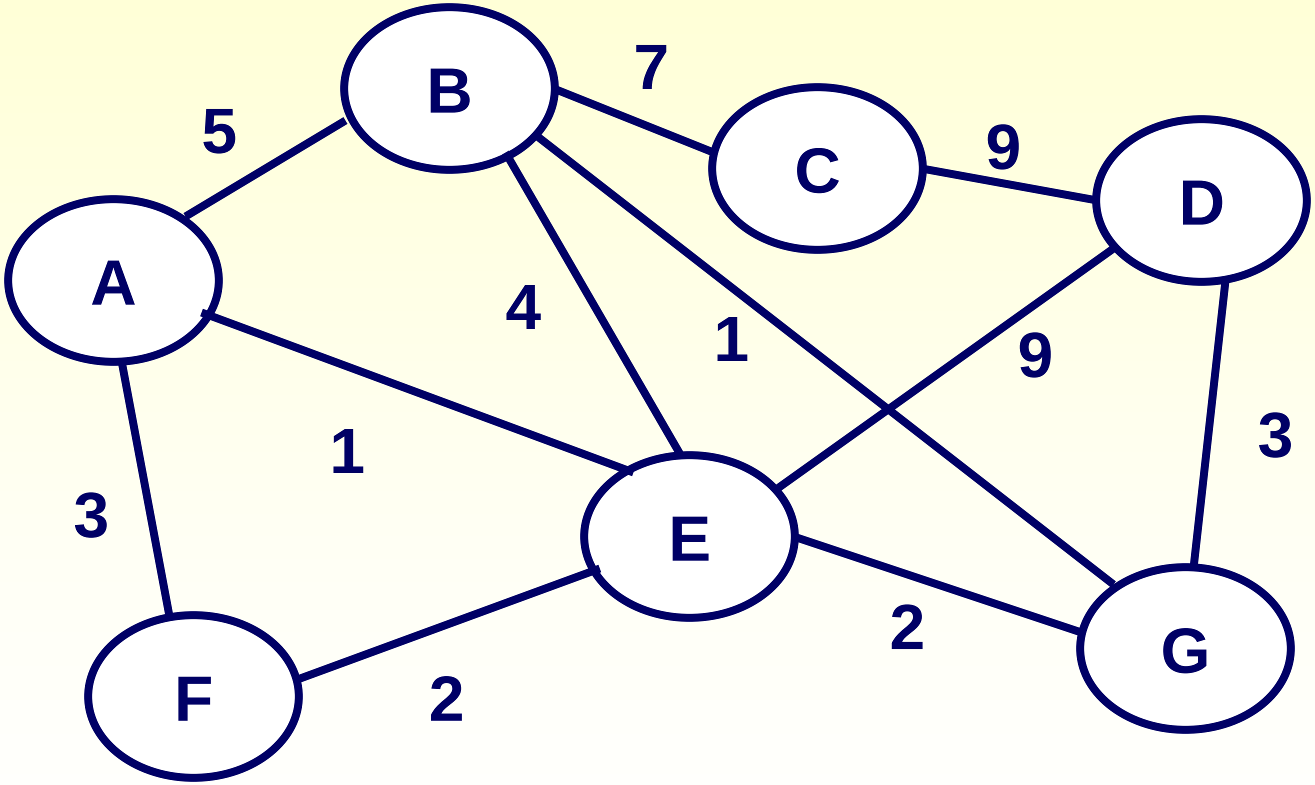
A *digraph* or *directed* graph has edges (arcs) that flow in only one direction. In an *undirected* graph, edges flow in either direction.

Directed Graphs



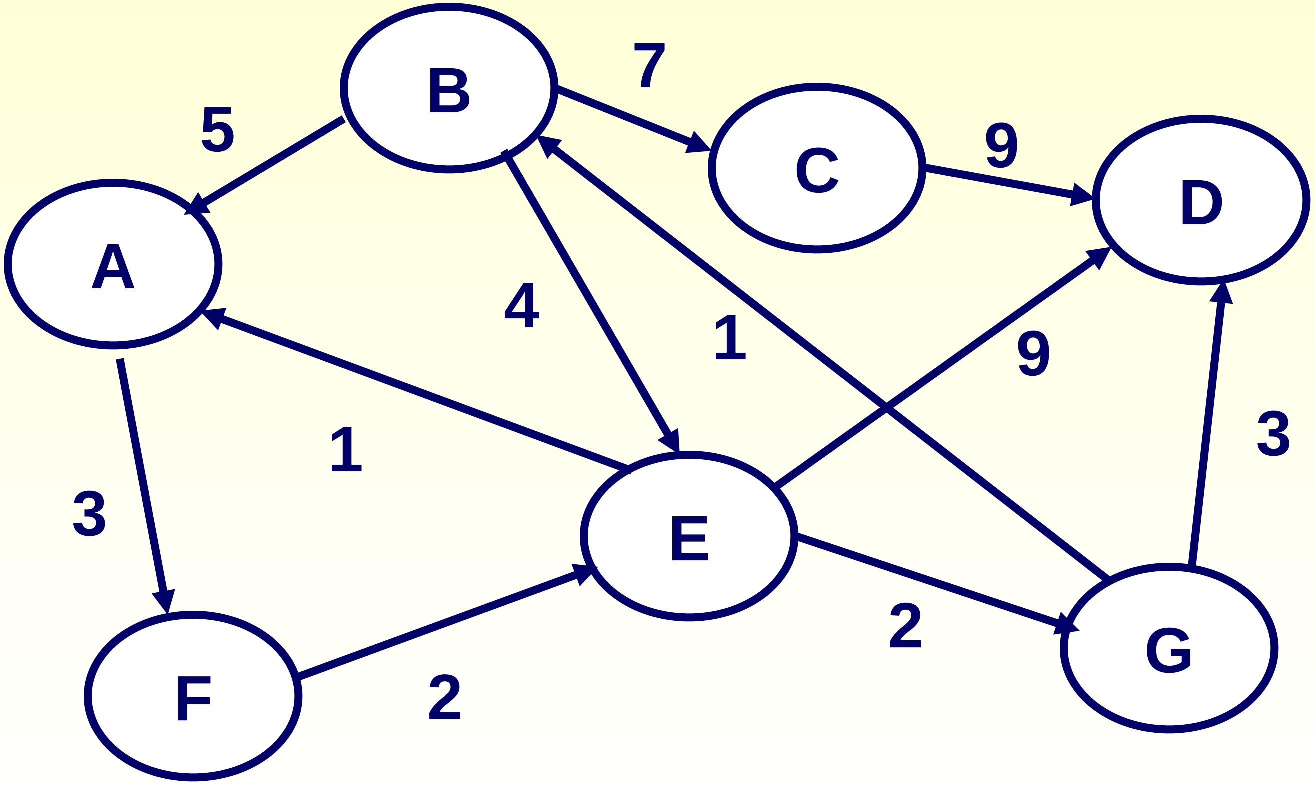
Directed edges only allow movement in one direction.

Weighted Edges



Edge weights represent cost.

Weighted Directed Graphs



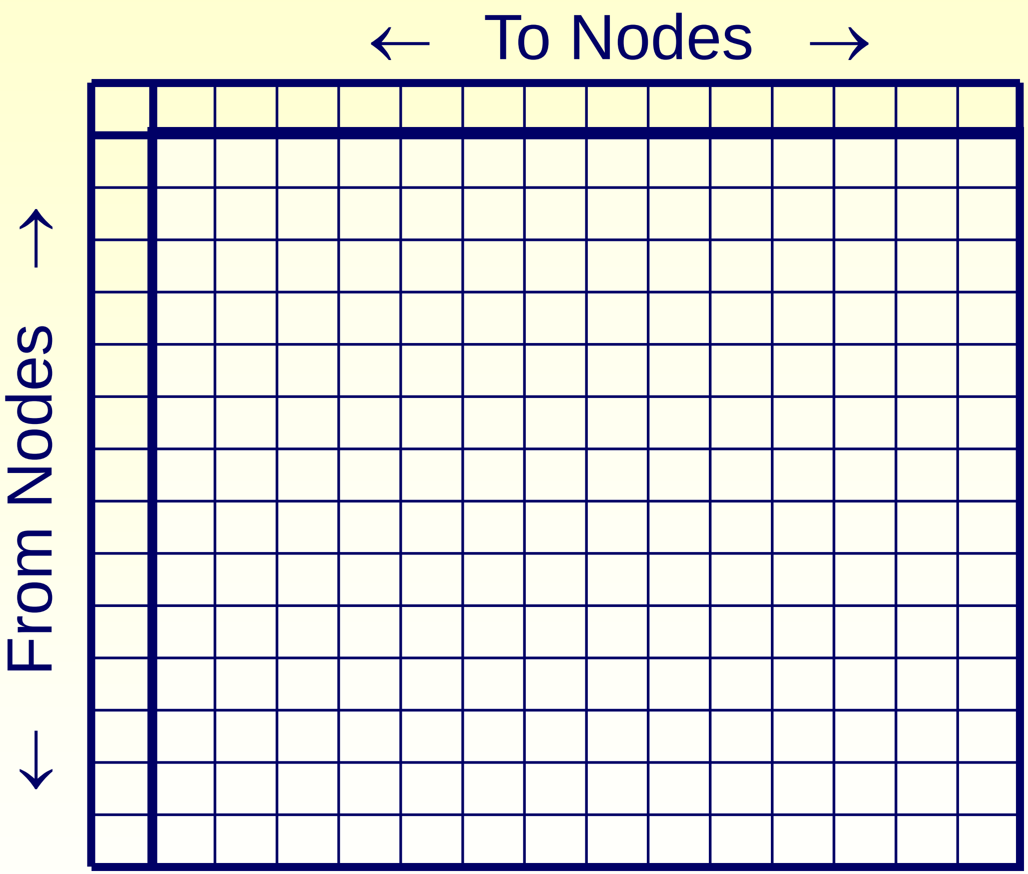
Directed edges only allow movement in one direction.

Representing Graphs

- How do we represent a graph that has any number of children or connections?
  - Adjacency matrices
  - Nodes held in some structure (adjacency list)
    - Each node has list of children
  - Links held in some kind of structure
    - Each link points to two nodes

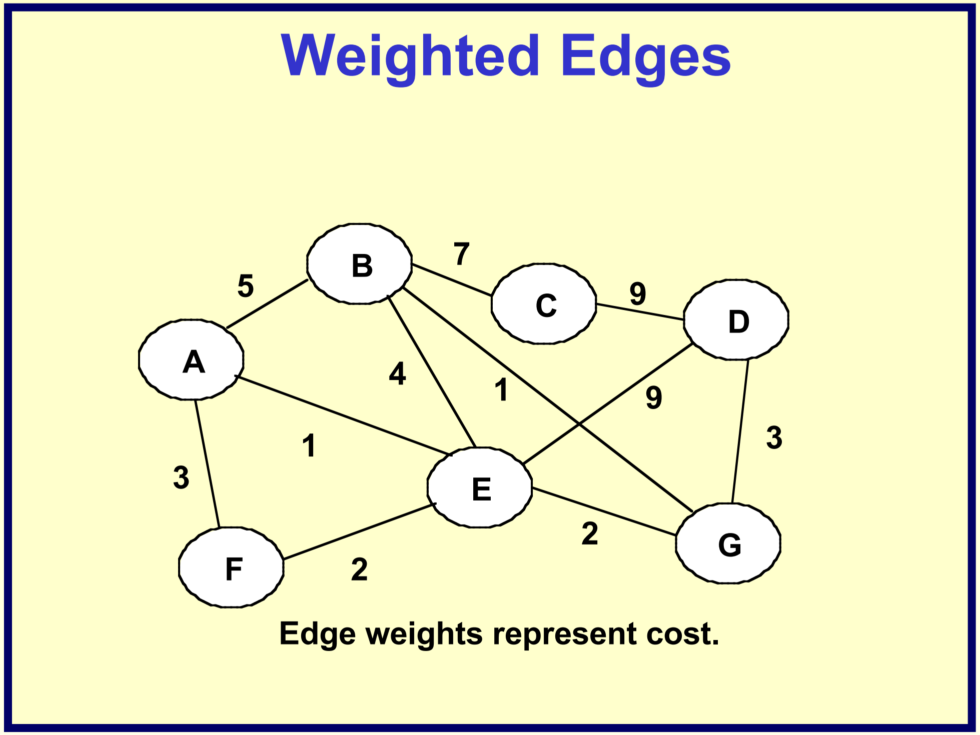
- Which way is best?
  - Depends!

Adjacency Matrix



- **Initially empty**
- **Each edge adds an entry**
- **Undirected graph can**
  - **Put in 2 entries per edge**
  - **Use just upper or lower diagonal**
- **Directed graph uses entire matrix**
- **Unweighted graph inserts '1'**
- **Weighted graph inserts the weight**

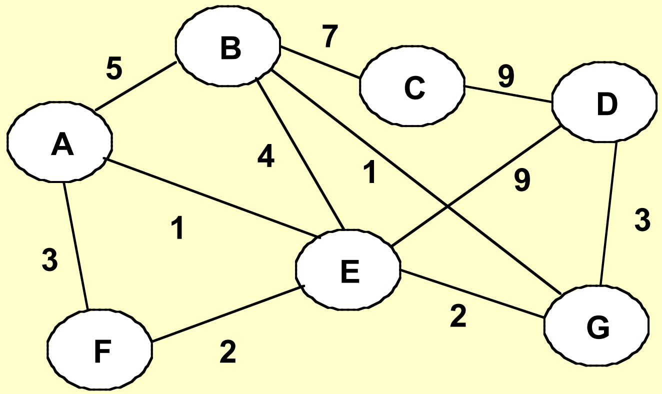
- **Size is $O(N^2)$**
- **Memory is usually sparsely utilized**



Undirected

	A	B	C	D	E	F	G
A	-	5	.	.	1	3	.
B	5	-	7	.	4	.	1
C	.	7	-	9	.	.	.
D	.	.	9	-	9	.	3
E	1	4	.	9	-	2	2
F	3	.	.	.	2	-	.
G	.	1	.	3	2	.	-

Weighted Edges

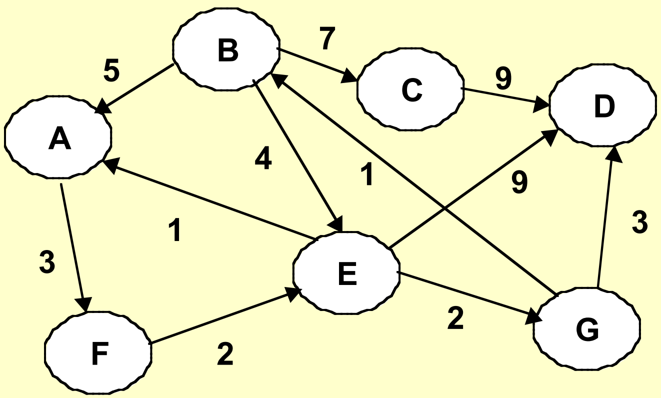


Edge weights represent cost.

Undirected

	A	B	C	D	E	F	G
A	-	5	.	.	1	3	.
B		-	7	.	4	.	1
C			-	9	.	.	.
D				-	9	.	3
E					-	2	2
F						-	.
G							-

Weighted Directed Graphs



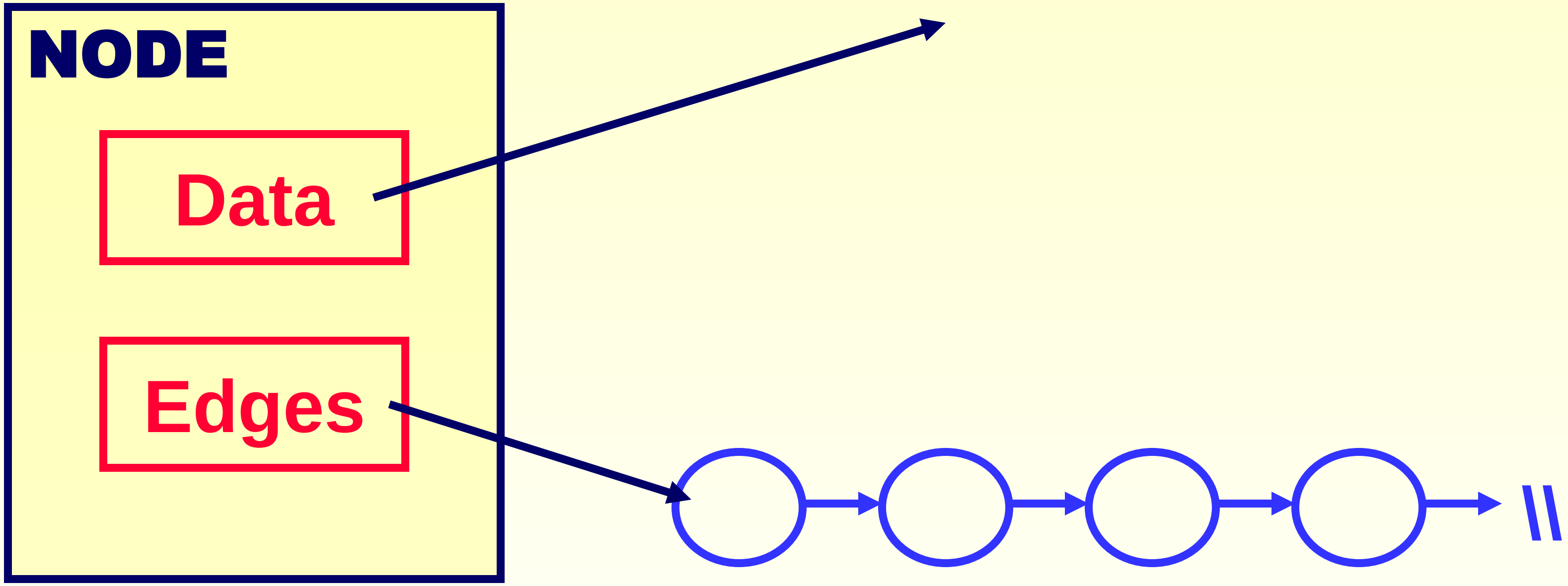
Directed edges only allow movement in one direction.

Directed

FROM \ TO	A	B	C	D	E	F	G
A	–	.	.	.	.	3	.
B	5	–	7	.	4	.	.
C	.	.	–	9	.	.	.
D	.	.	.	–	.	.	.
E	1	.	.	9	–	.	2
F	.	.	.	.	2	–	.
G	.	1	.	3	.	.	–

Implementation with Linked Lists

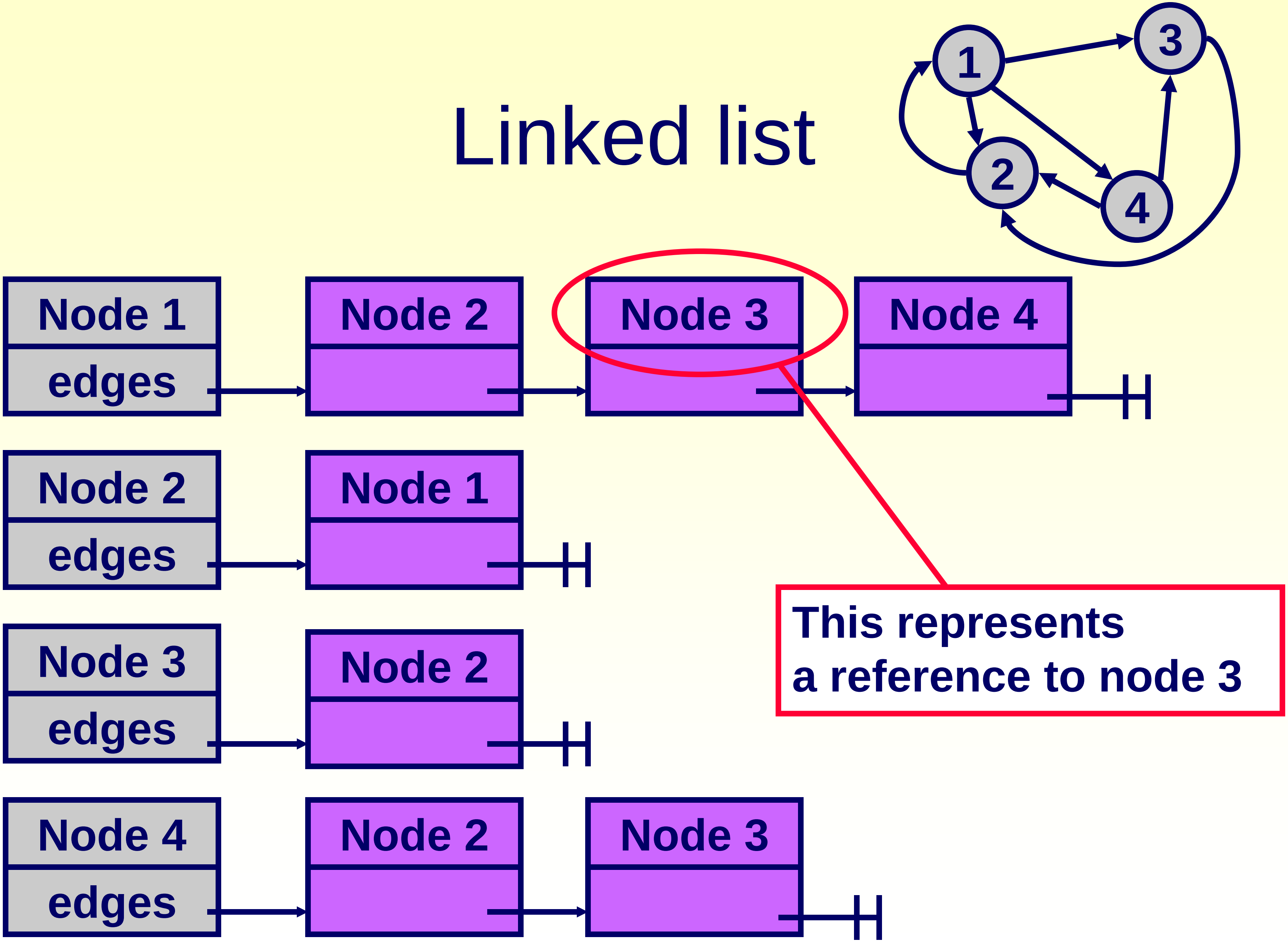
[ArrayList might be more appropriate]



Edges

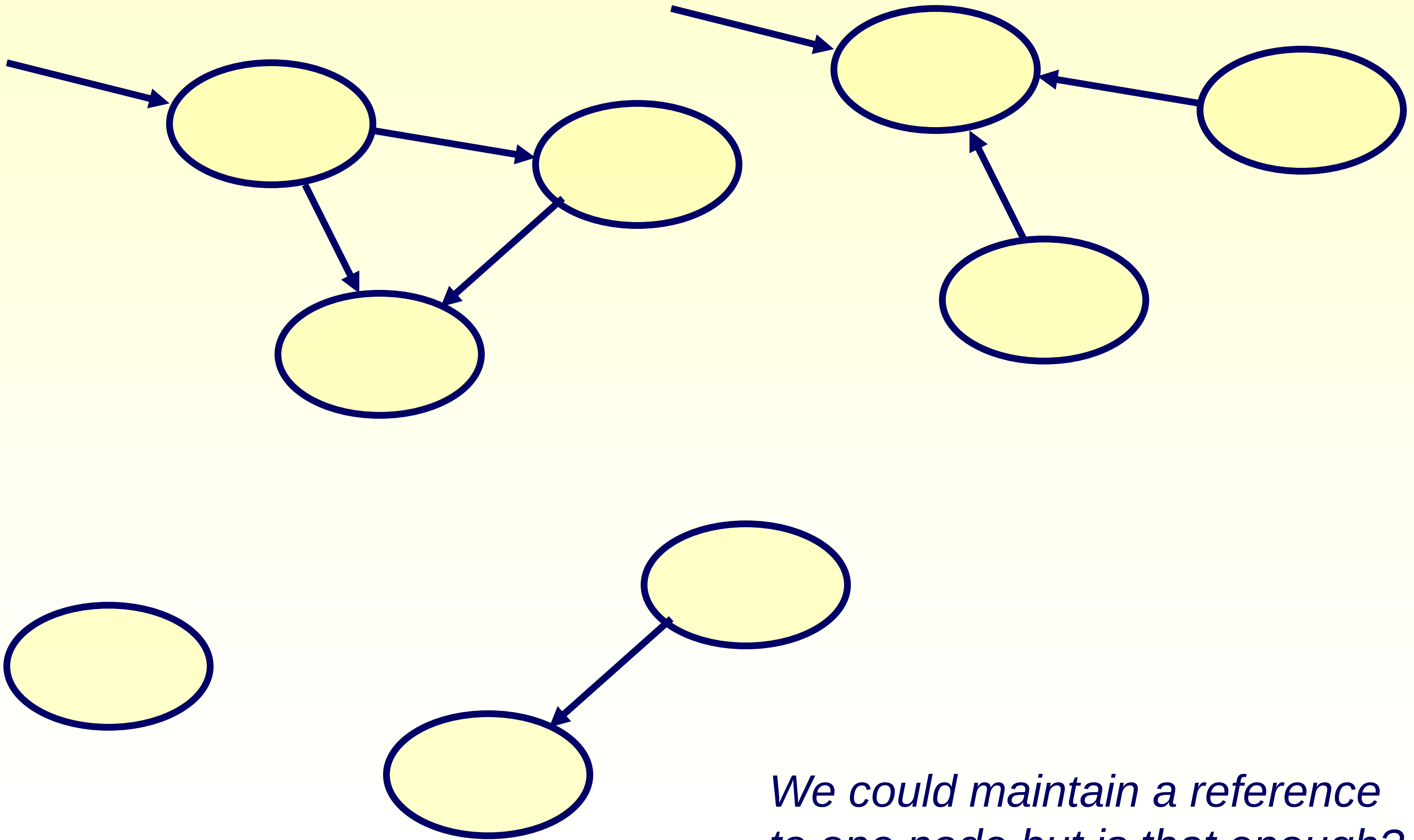
- **With references to other nodes**
- **Possibly with weights**

Linked list



1
2
3
4
Node 1
edges
Node 2
Node 3
Node 4
Node 2
edges
Node 1
Node 3
edges
Node 2
Node 4
edges
Node 2
Node 3
This represents
a reference to node 3

But, where are the Nodes?

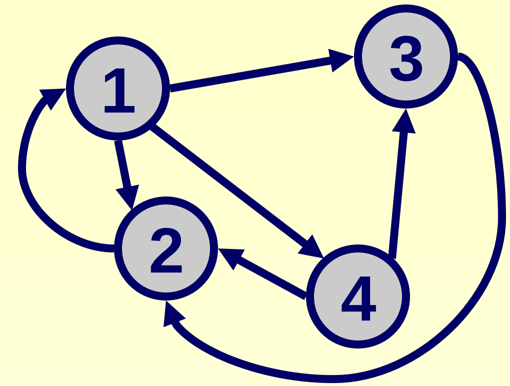


We could maintain a reference to one node but is that enough?

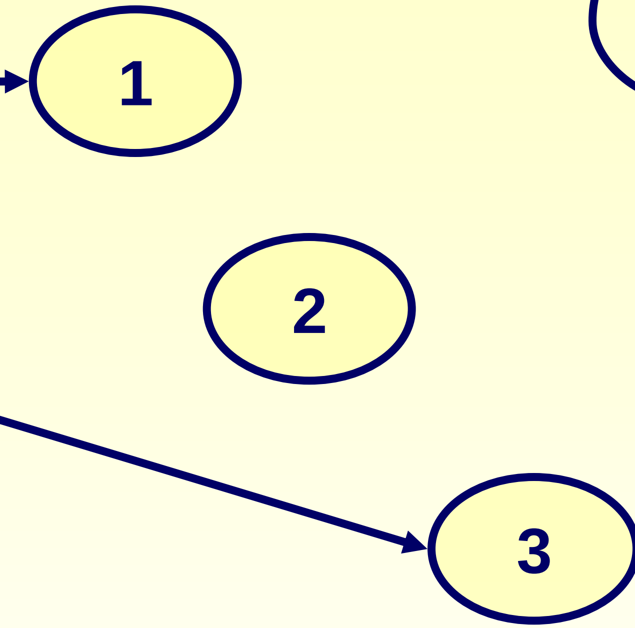
In addition to...

- ...Information about edges connecting nodes
- Might also maintain structure holding nodes?
  - List (Vector)
  - Tree
  - Array
  - Hash Table

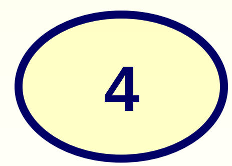
All reference arrows not shown



from	to
1	3
4	3
1	4
1	2
2	1
4	2
3	2



Store the links in a structure



Let the nodes float

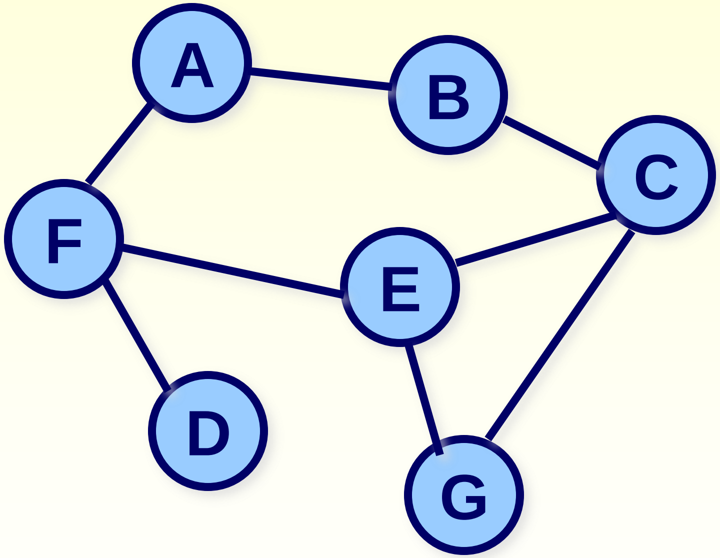
Problems?

Graph Traversal

Graphs :: Searching

Let's perform an inductive analysis of a search, and figure out how it works. We can then model this in code.

Given this graph:



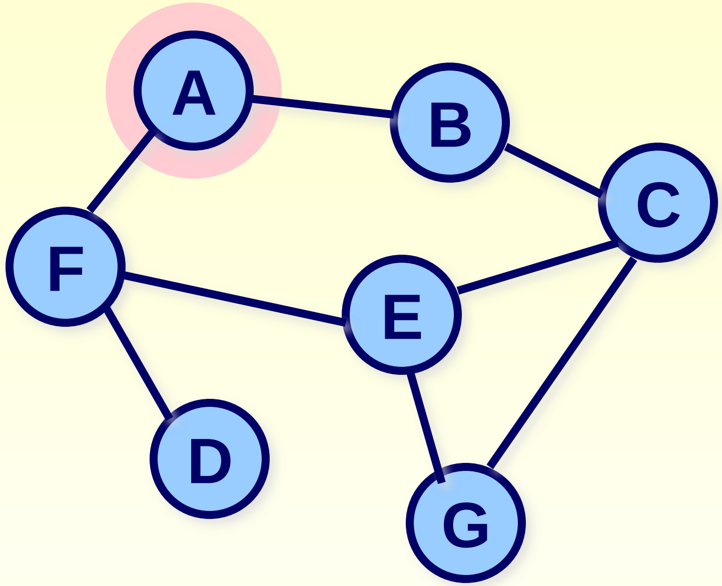
Let's see if there exists a path from A to G.

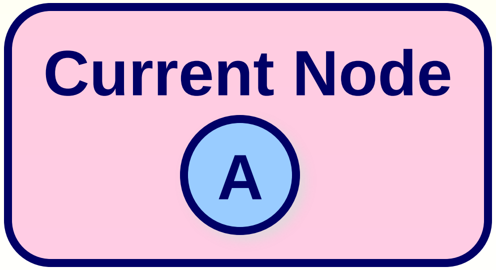
(Of course there's a path.
We can *see* that. But how can a *computer* determine this?)

Graphs :: Searching

We will perform a BFS.

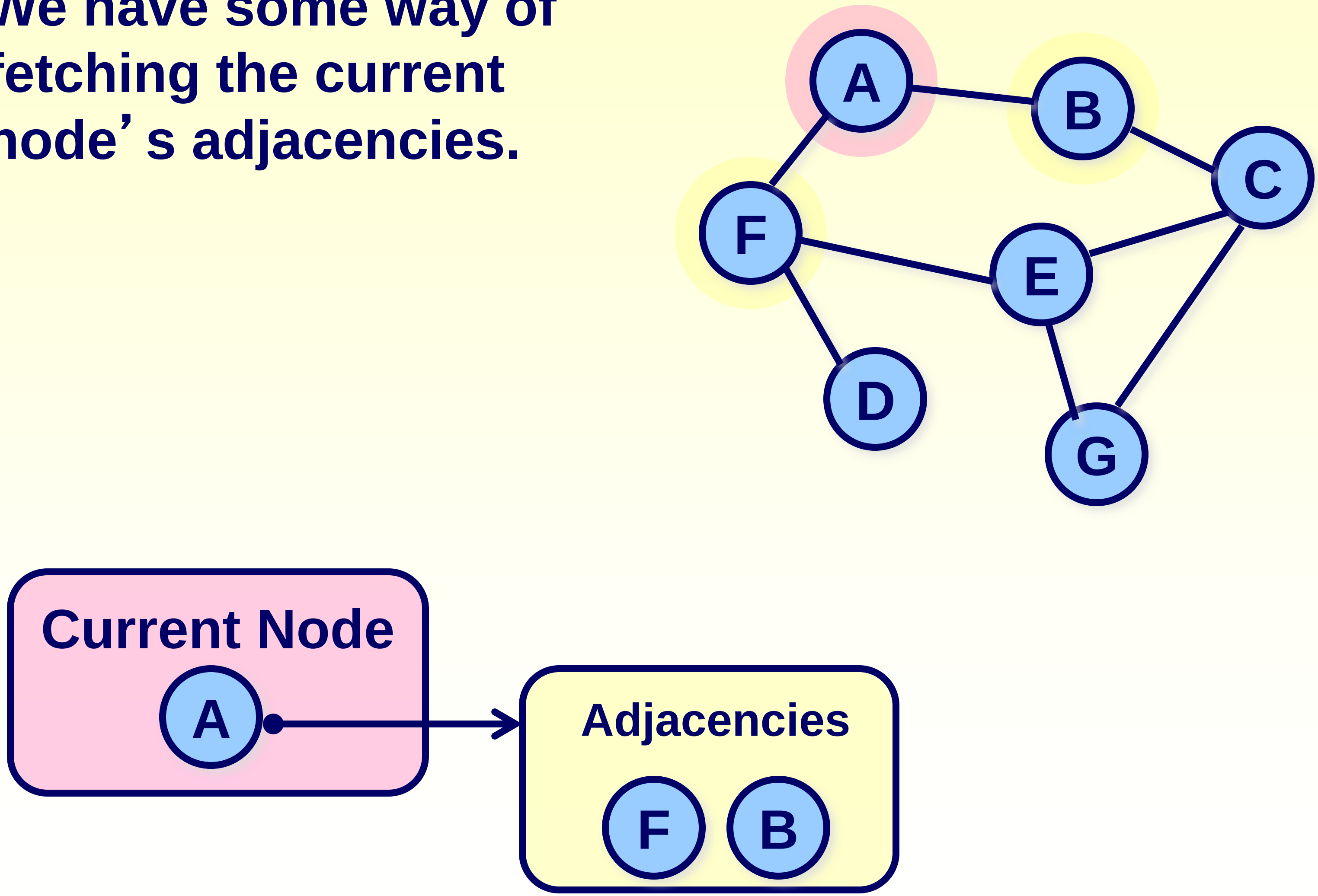
We are first given our start node, A, which we can designate as the “current” node we are visiting.





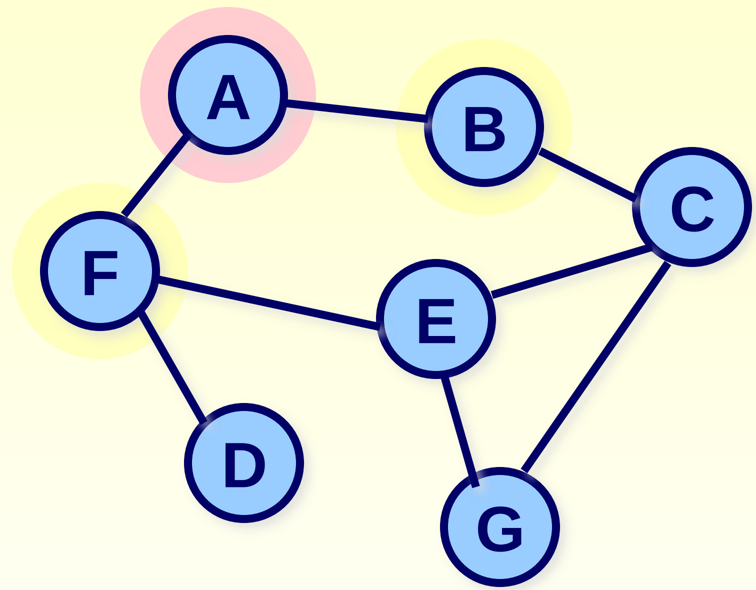
Graphs :: Searching (BFS)

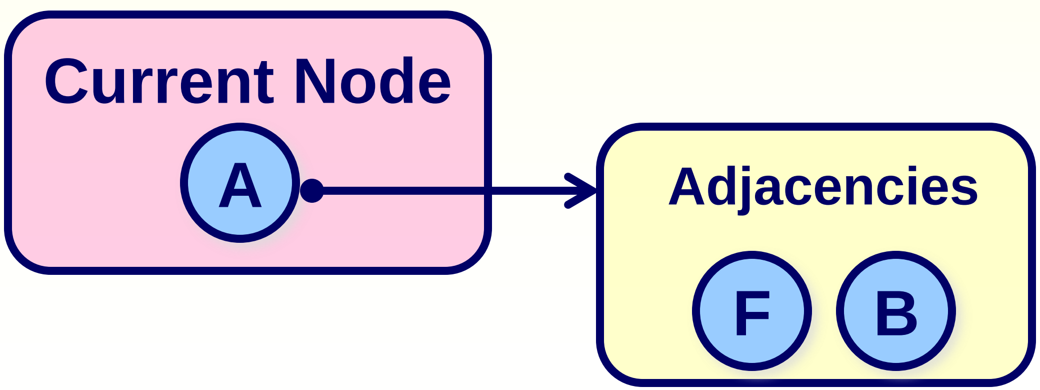
We have some way of fetching the current node's adjacencies.



Graphs :: Searching (BFS)

In a linked list or tree, we had a set number of “links” or children, so exploring them all was easy--just write a line of code to visit each child or adjacent node.

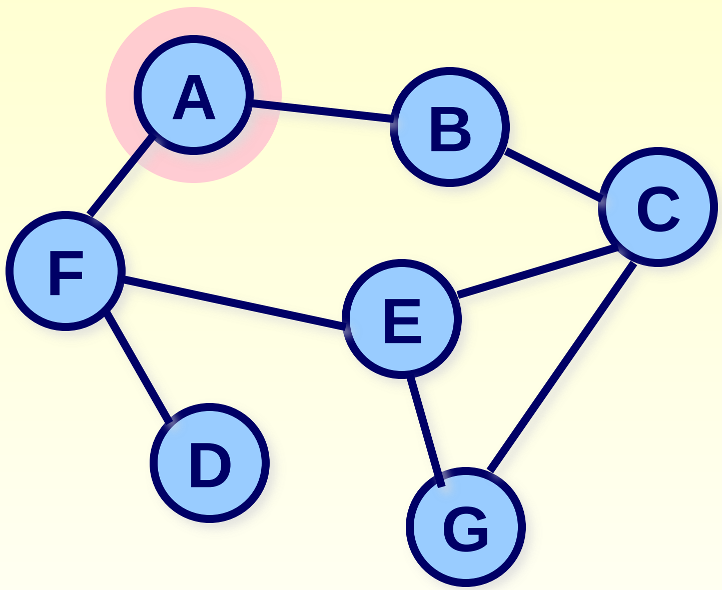


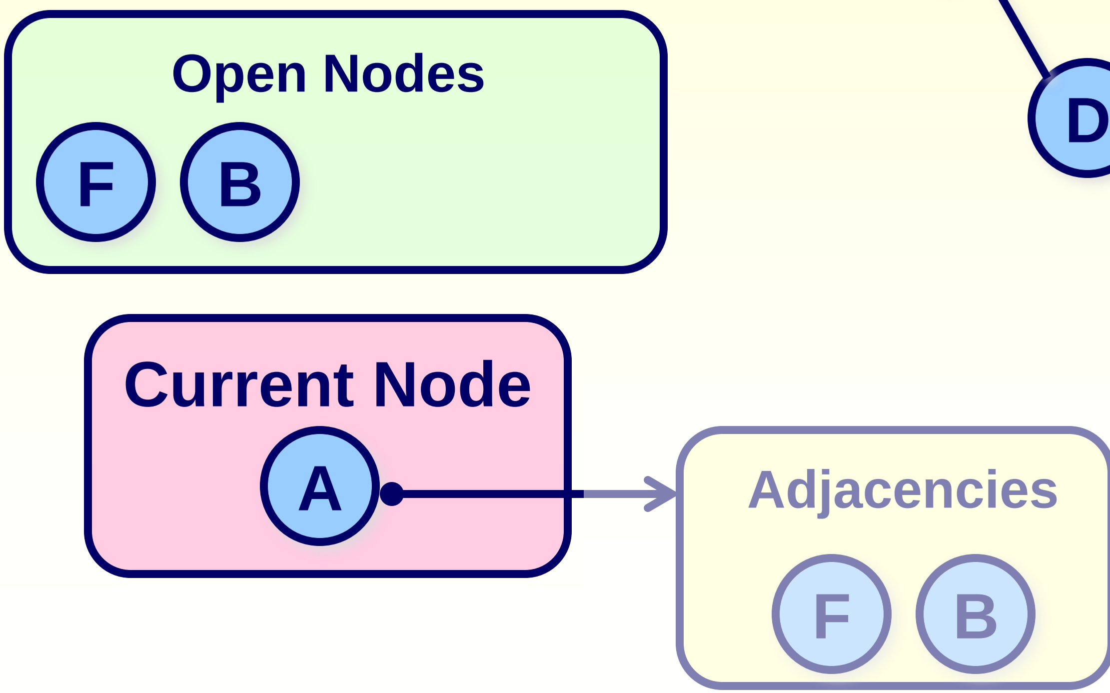


But in a graph, each node has a variable number of nodes. We need a set or list to manage the nodes we discover, but have not explored

Graphs :: Searching (BFS)

So we use a queue, since this is BFS. A DFS would have used a stack instead.

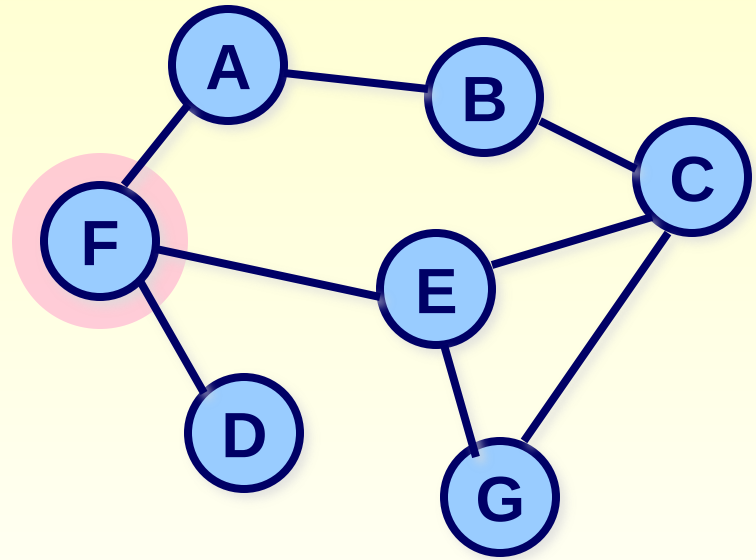


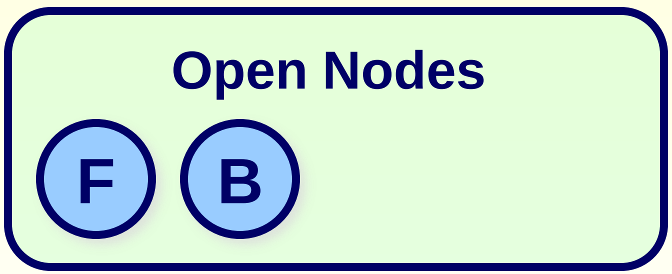


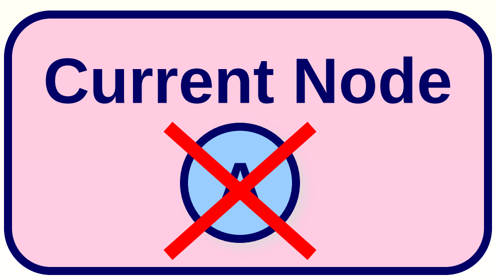
We place the current node's adjacencies in this list of open, unexplored nodes

Graphs :: Searching (BFS)

At this point, are done with the current node, and are ready to move on to the first node in the open list.

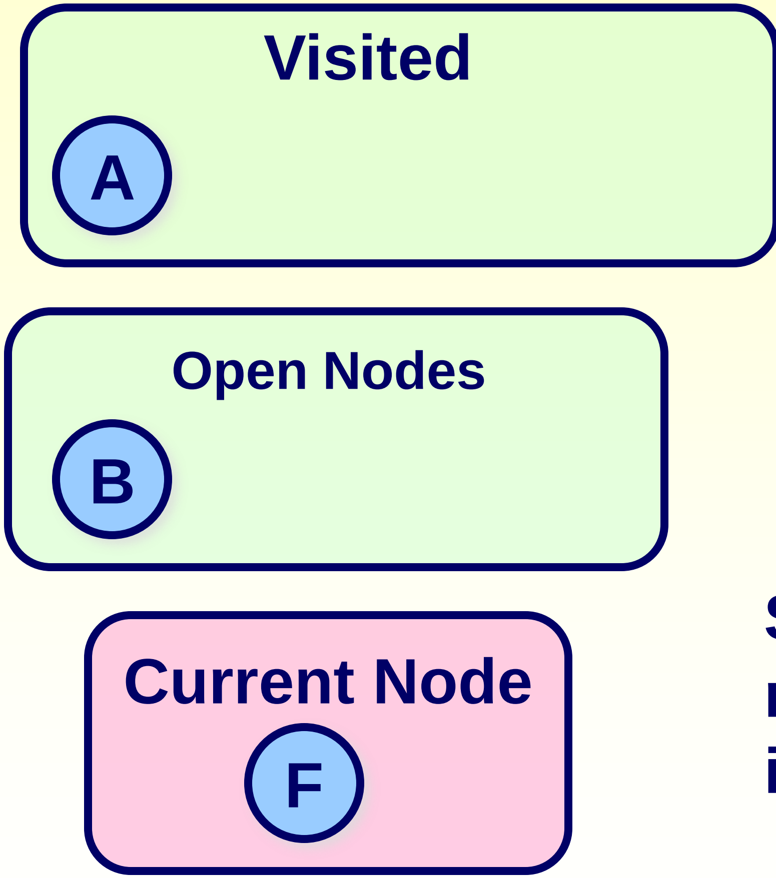


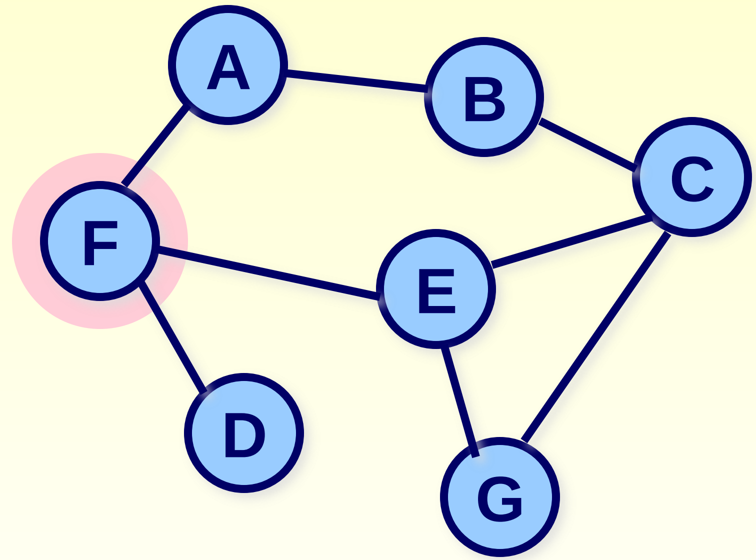




But wait! **Maybe we should keep a list of nodes we've already visited, so we don't return to them again. *Why would we visit them again?***

Graphs :: Searching (BFS)

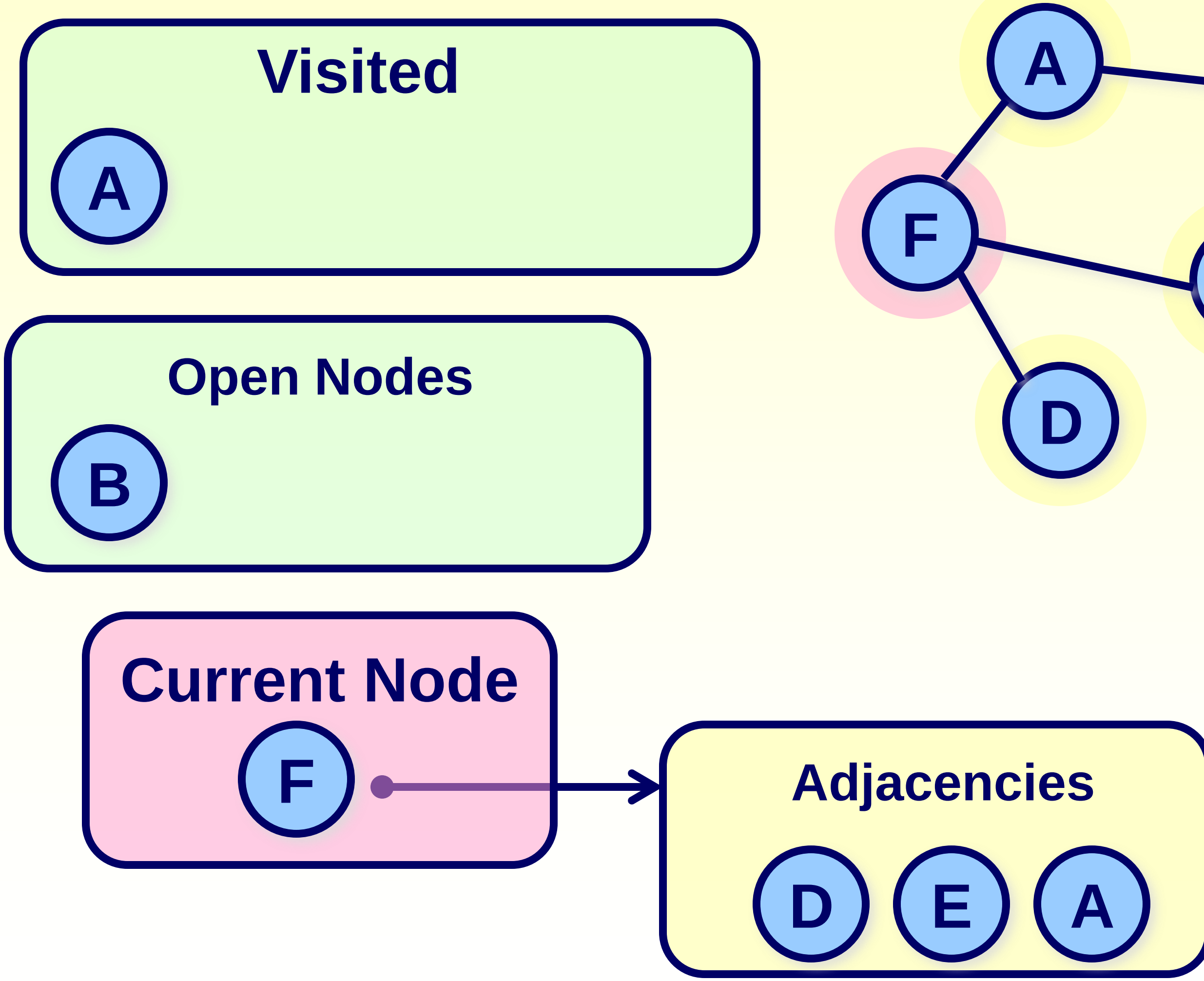


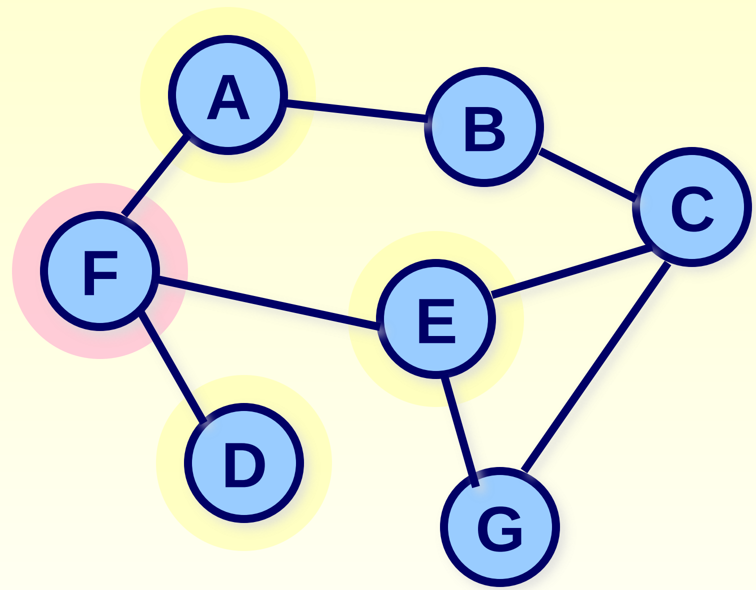


So we make a list to hold the nodes we' ve visited, and insert A into this list.

Our current node is now F. The node F is not the goal...

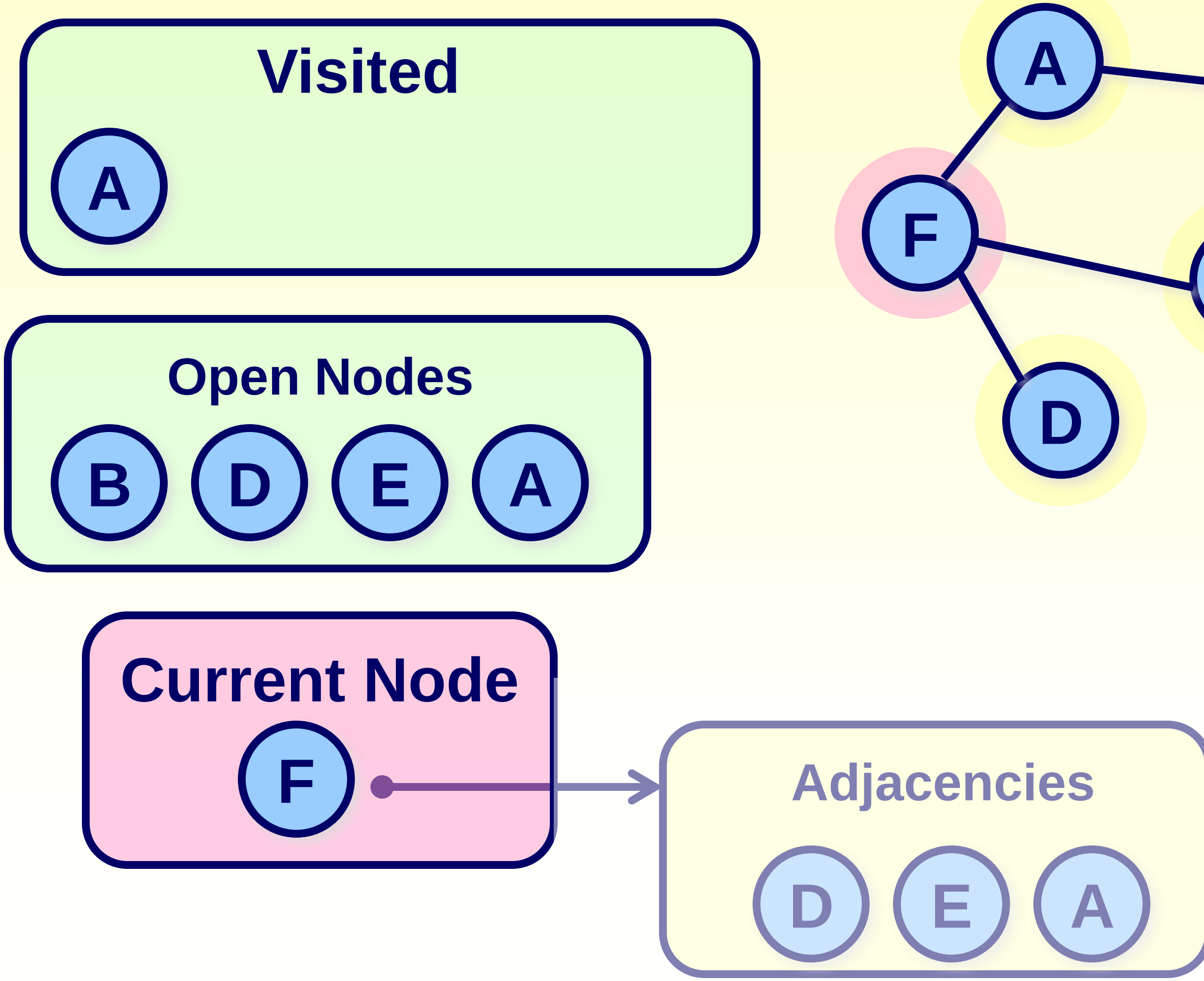
Graphs :: Searching (BFS)

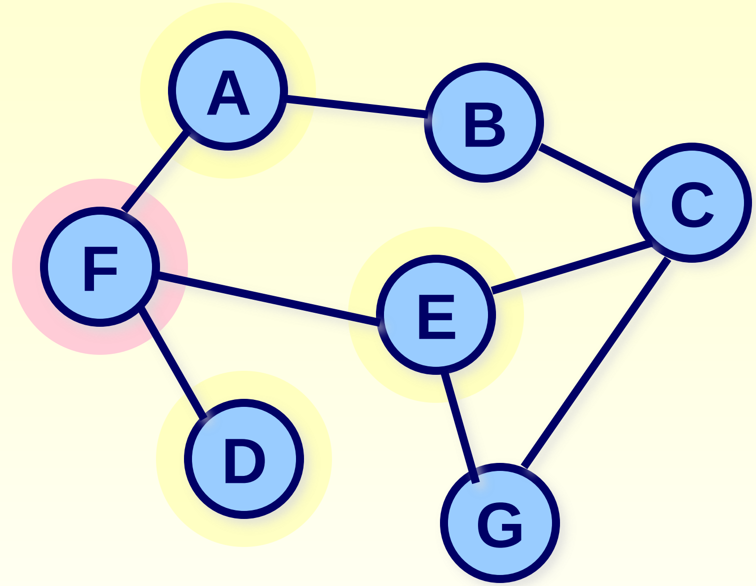




... so, we fetch the adjacencies to our new current node.

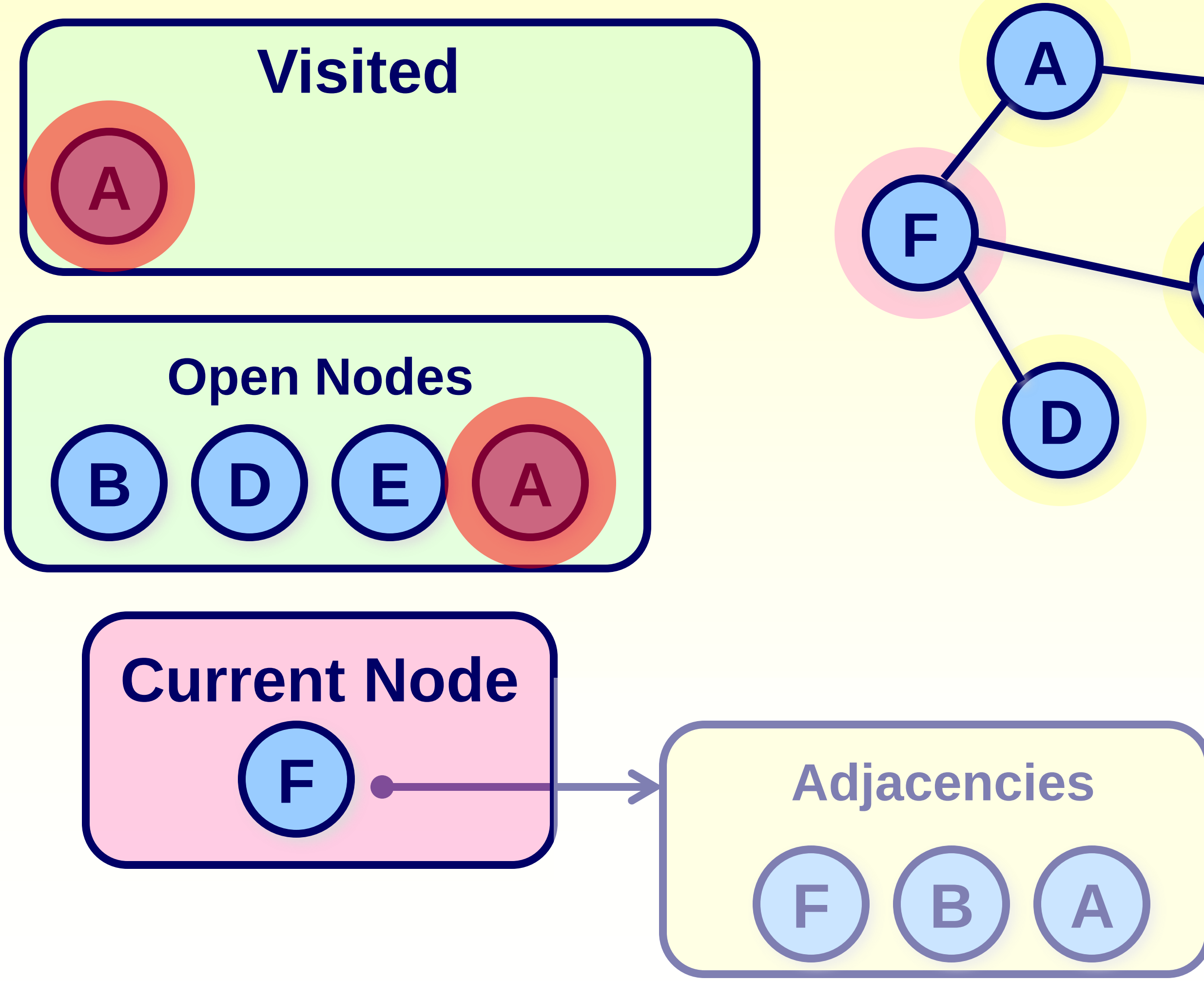
Graphs :: Searching (BFS)

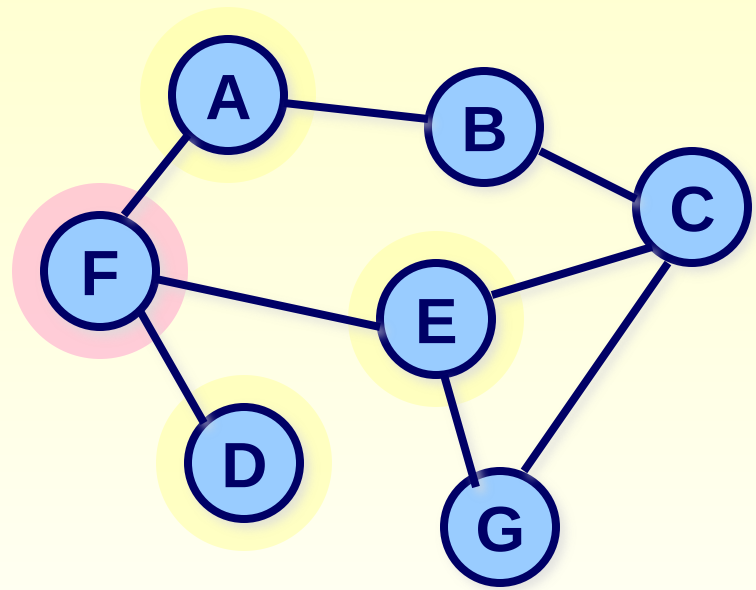




We prepare to copy the new adjacencies to our queue of open nodes . . .

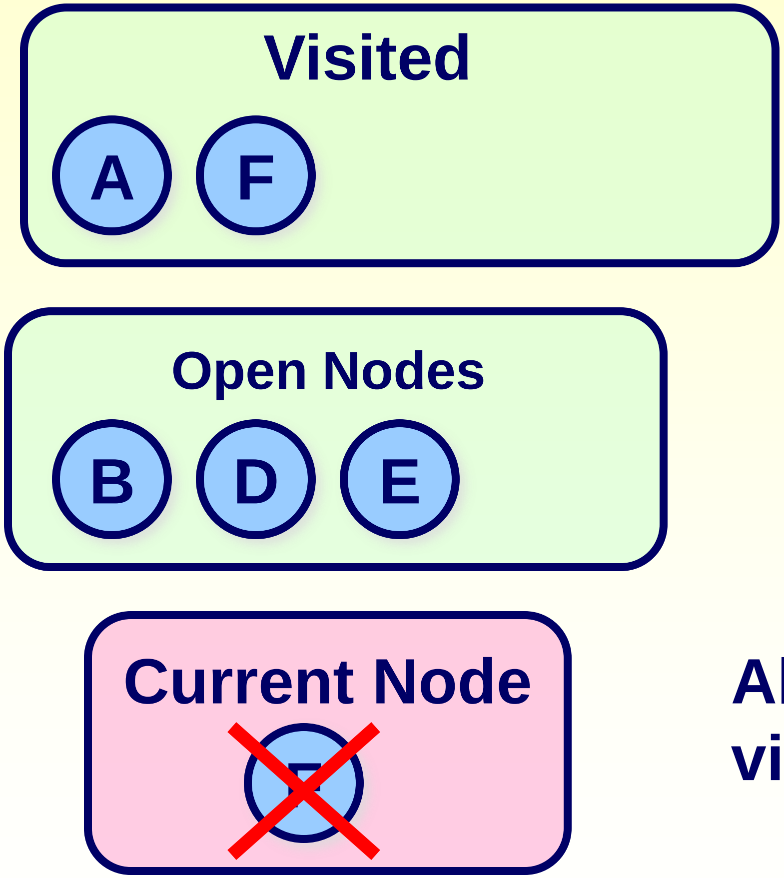
Graphs :: Searching (BFS)

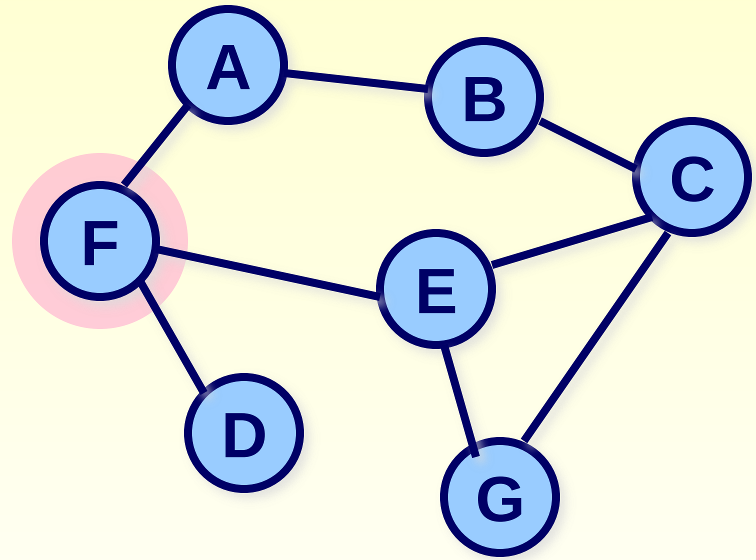




BUT WAIT! **We already visited node A. Don't place it in the open queue.**

Graphs :: Searching (BFS)

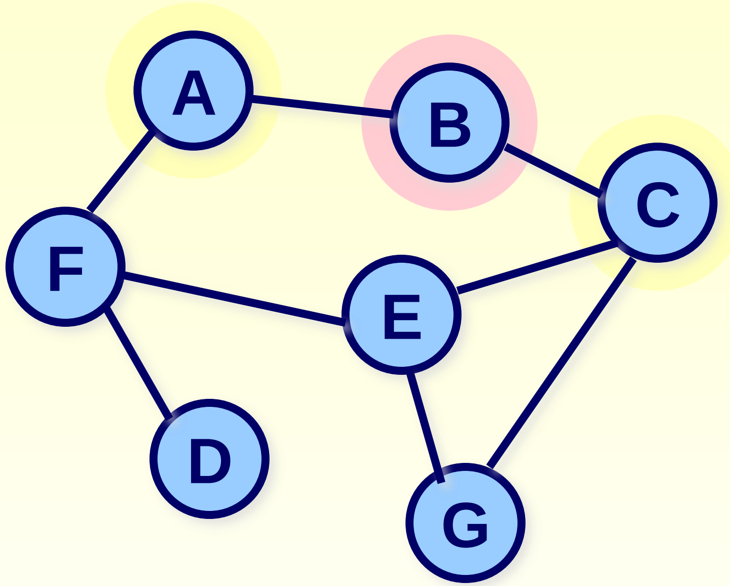


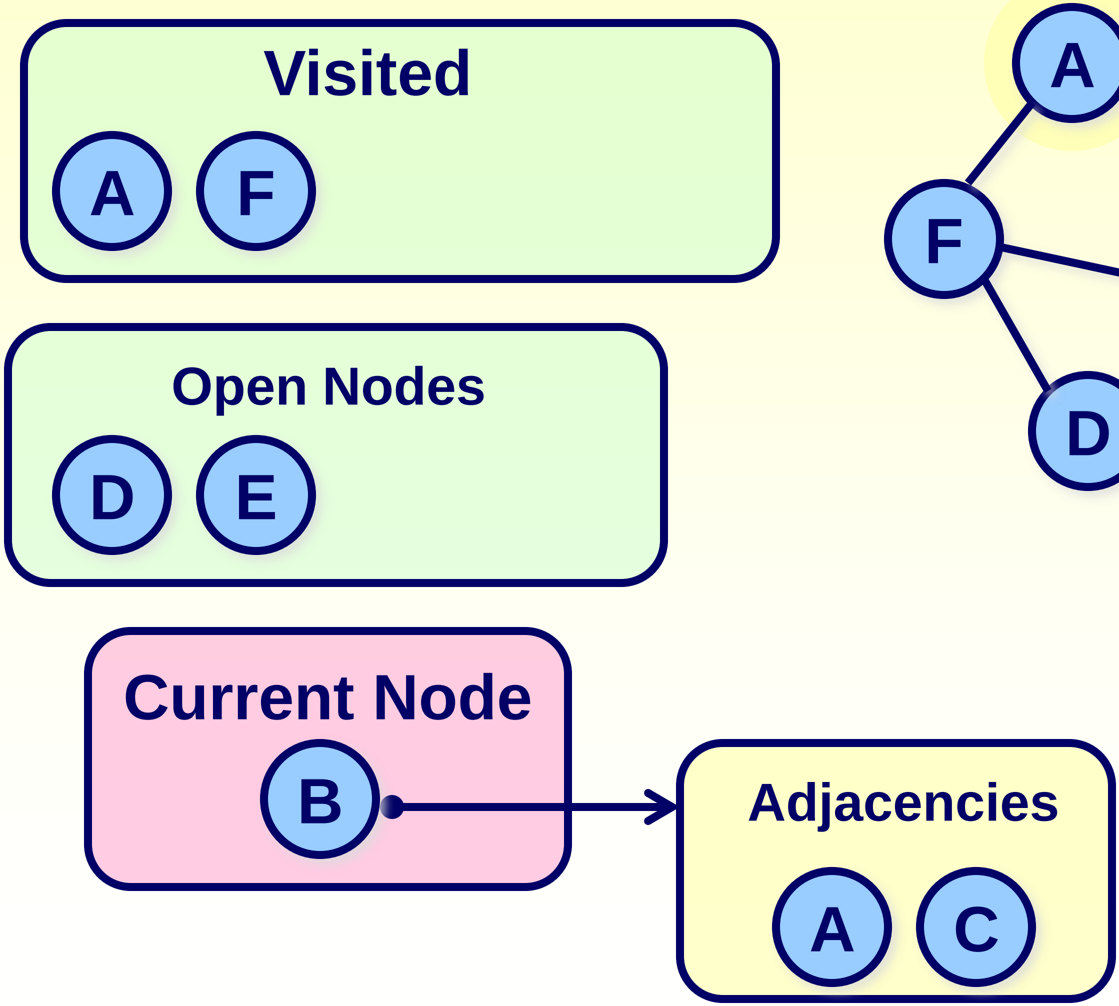


All done with F. Move it up to the visited list.

Let's check out B, the next node on our open queue.

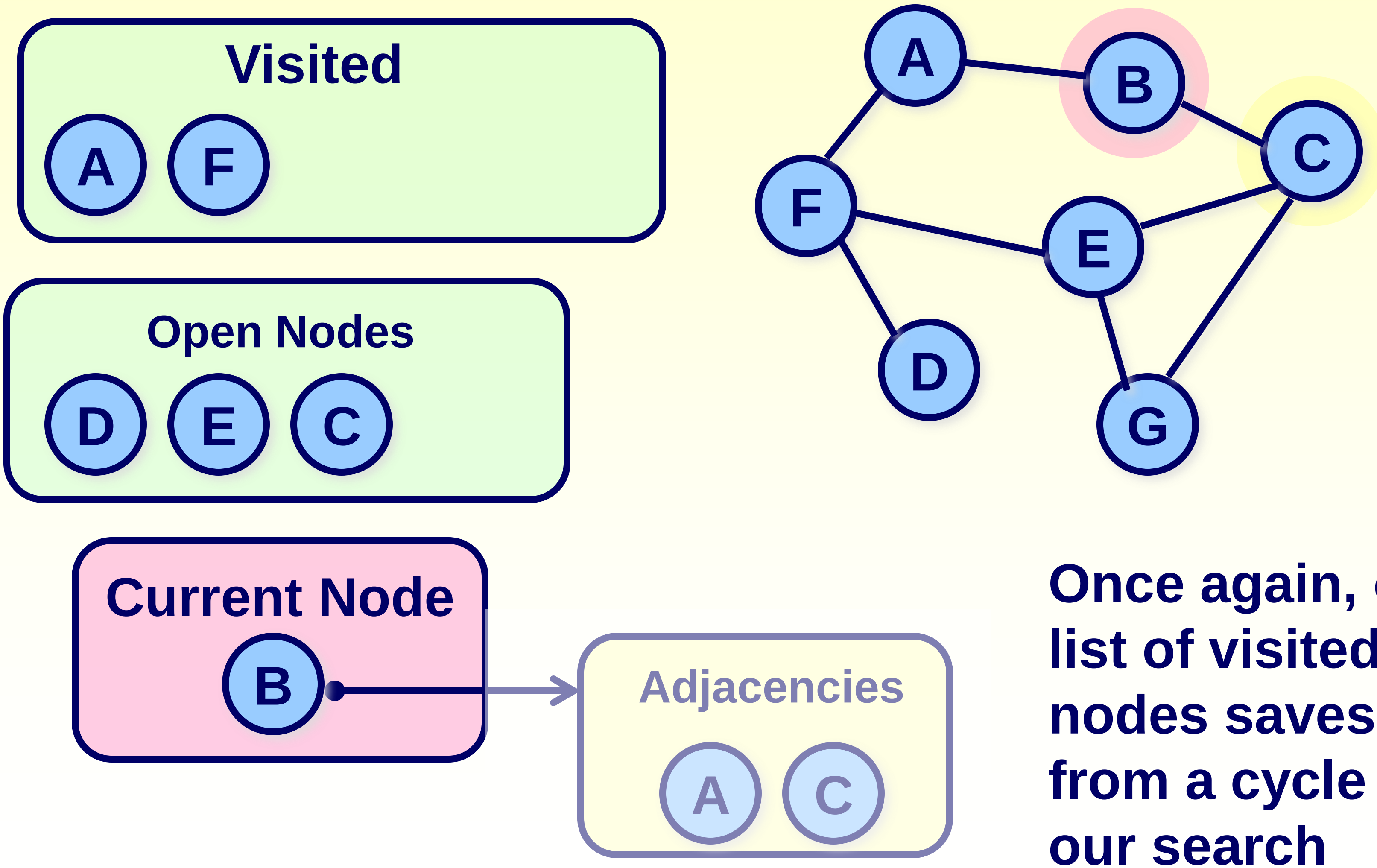
Graphs :: Searching (BFS)





What can we discover from B? We fetch its list of adjacent nodes.

Graphs :: Searching (BFS)



Once again, our list of visited nodes saves us from a cycle in our search

Graphs :: Searching (BFS)

Visited

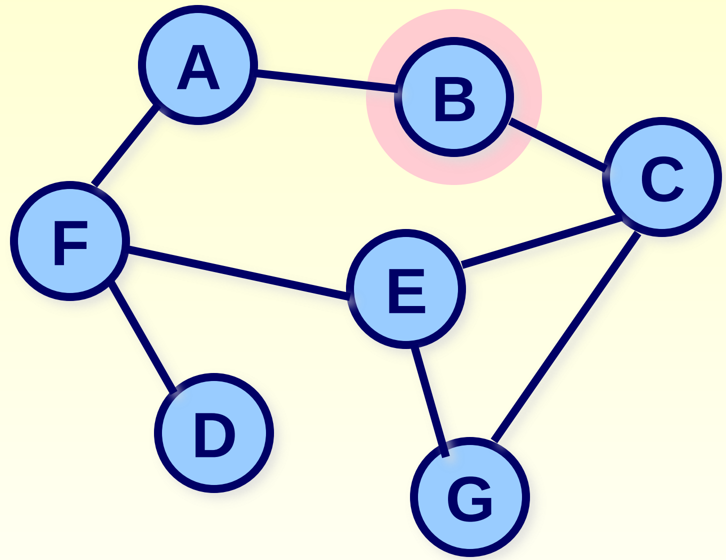
A F B

Open Nodes

D E C

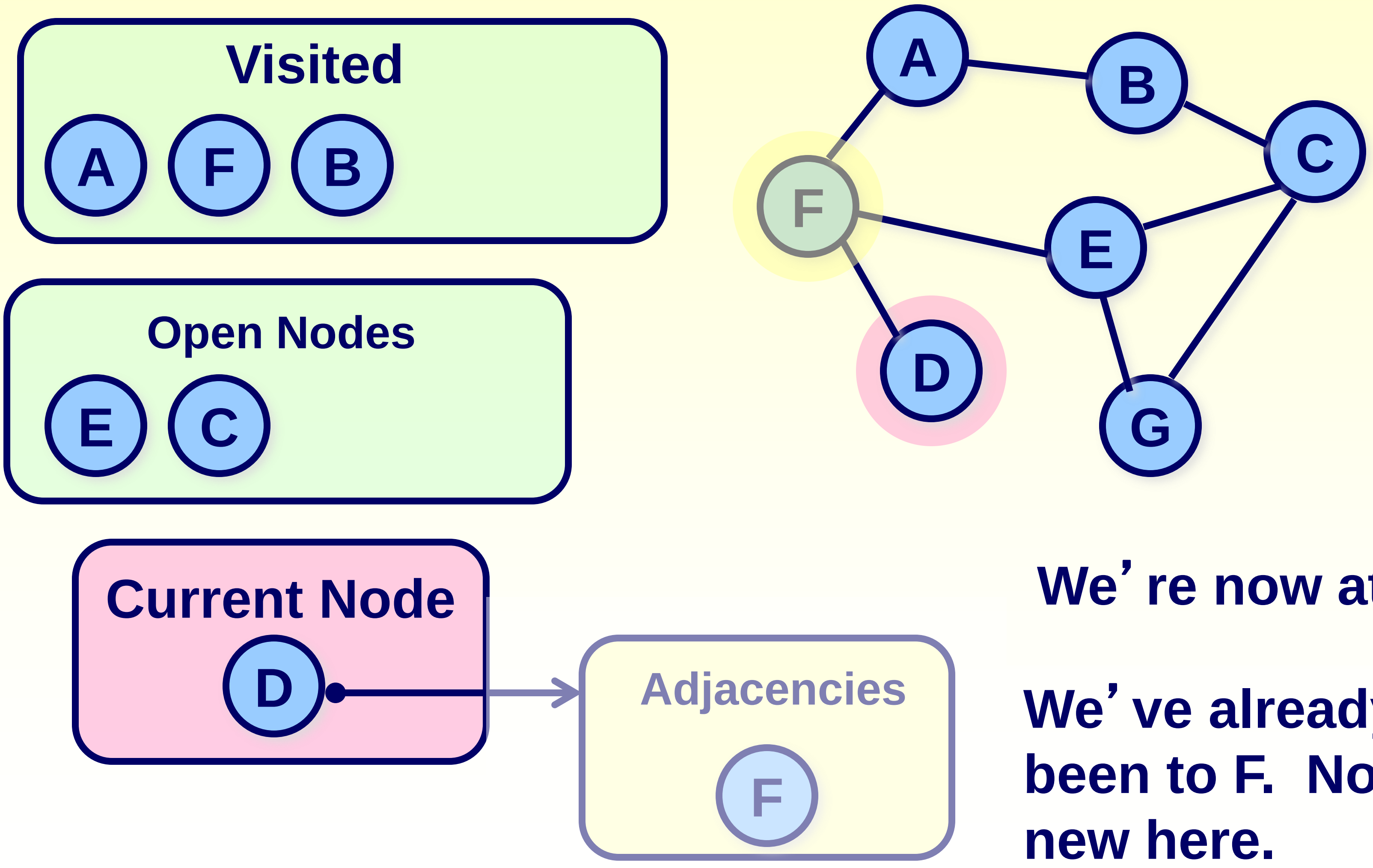
Current Node

~~B~~



We're done with B. Promote it to our visited list.

Graphs :: Searching (BFS)



We' re now at D.

We' ve already been to F. Nothing new here.

Graphs :: Searching (BFS)

Visited

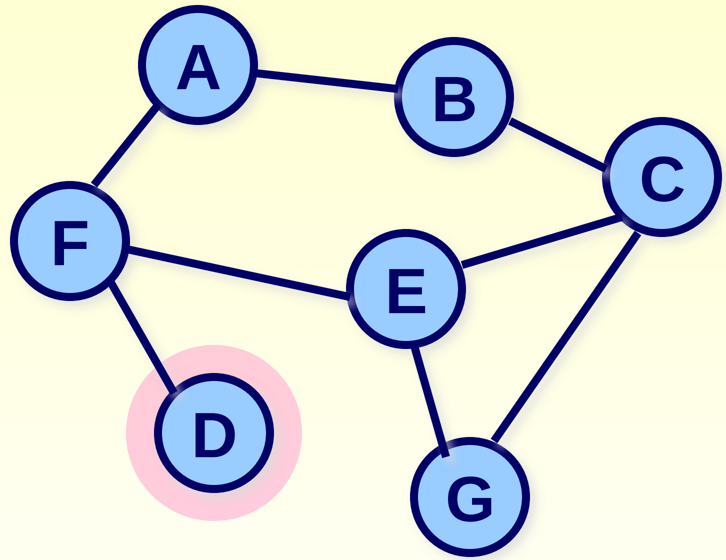
A F B D

Open Nodes

E C

Current Node

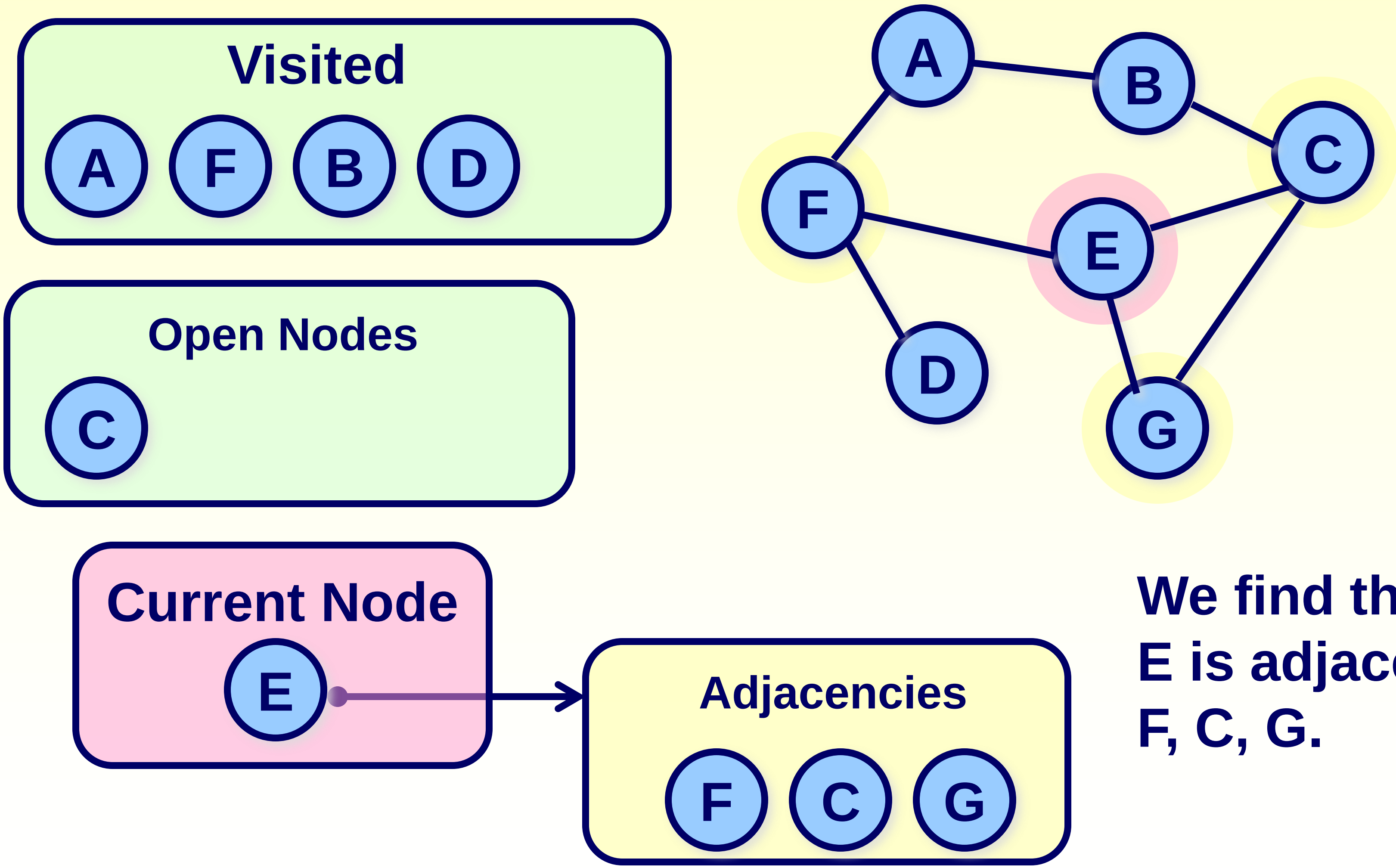
~~D~~



We' re done with D, so place a reference in our visited list.

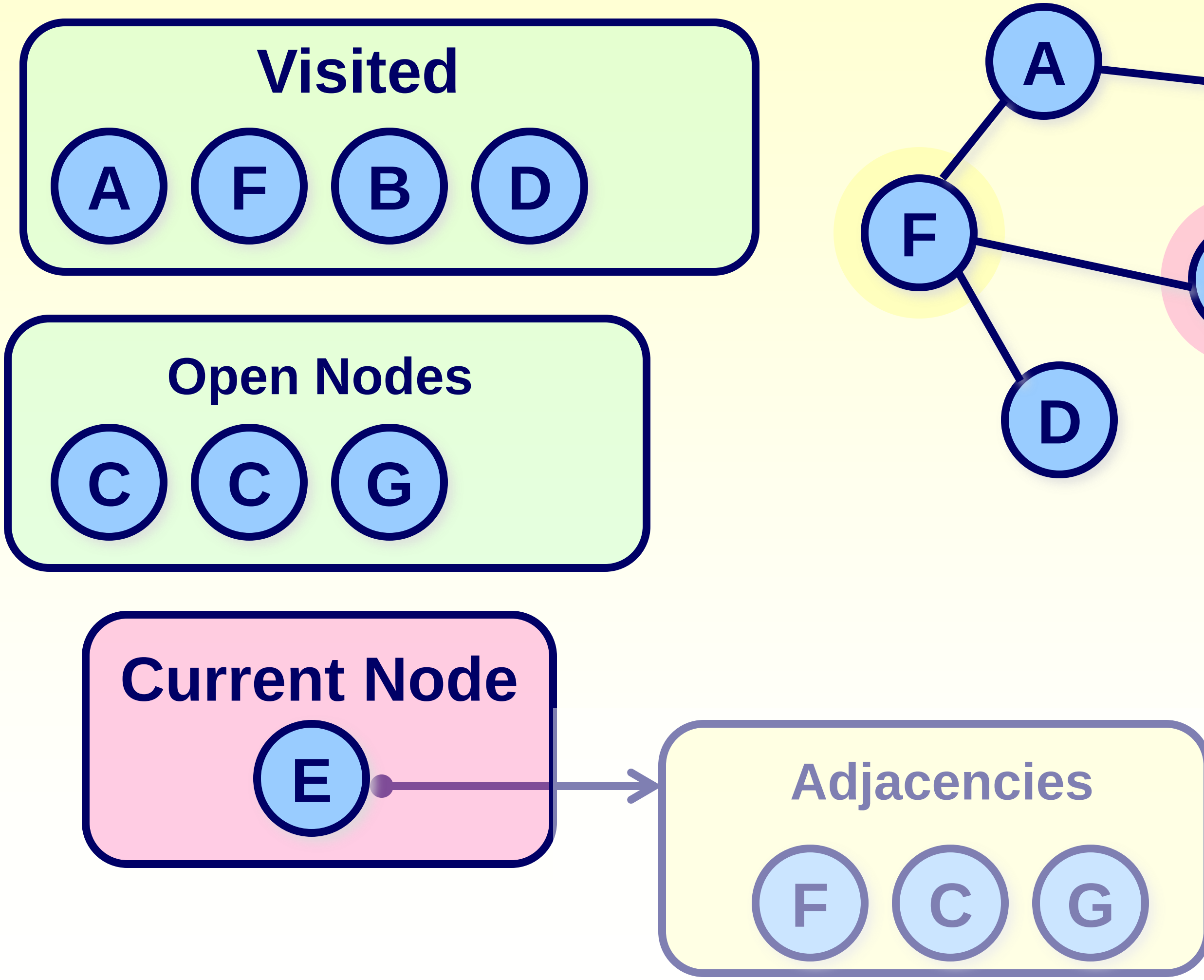
E is next up.

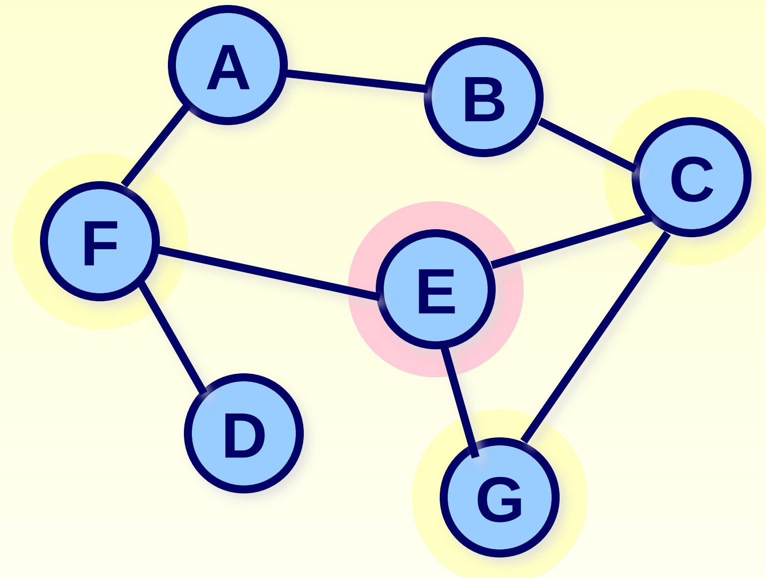
Graphs :: Searching (BFS)



We find that E is adjacent to F, C, G.

Graphs :: Searching (BFS)

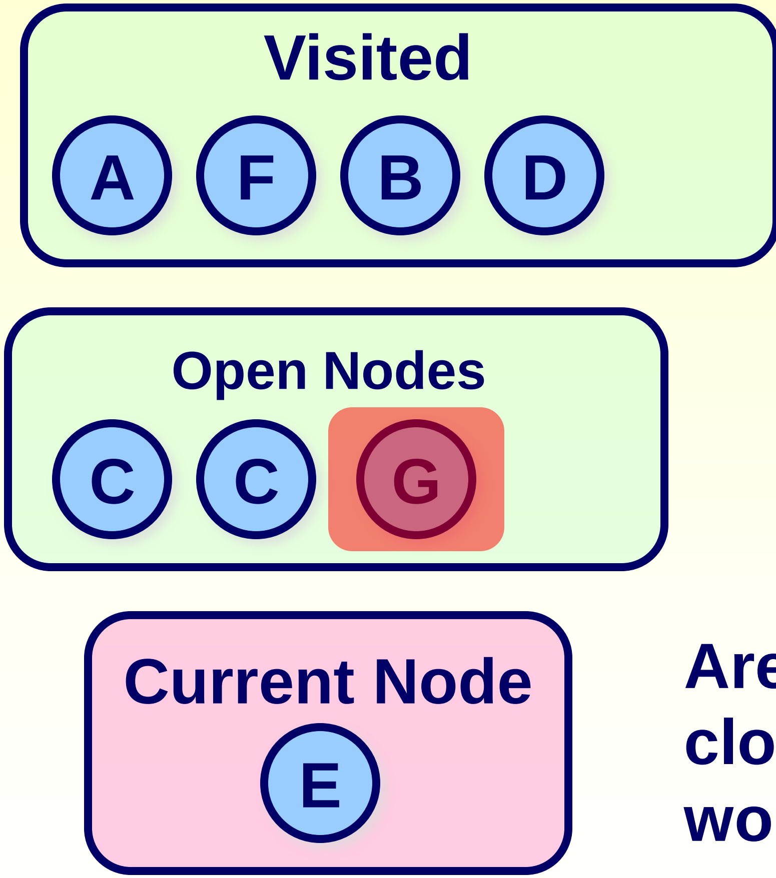


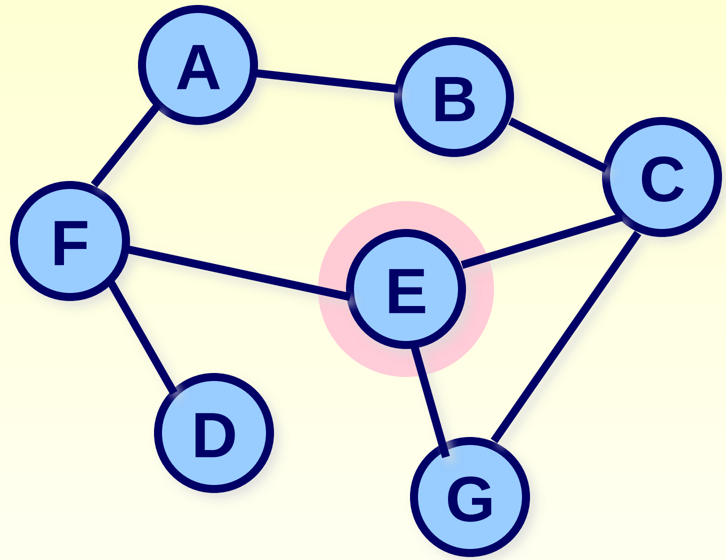


We copy these references to our open queue. The F node is omitted, since we've seen it already.

Graphs :: Searching (BFS)

Wait a minute. Can't we quit yet?

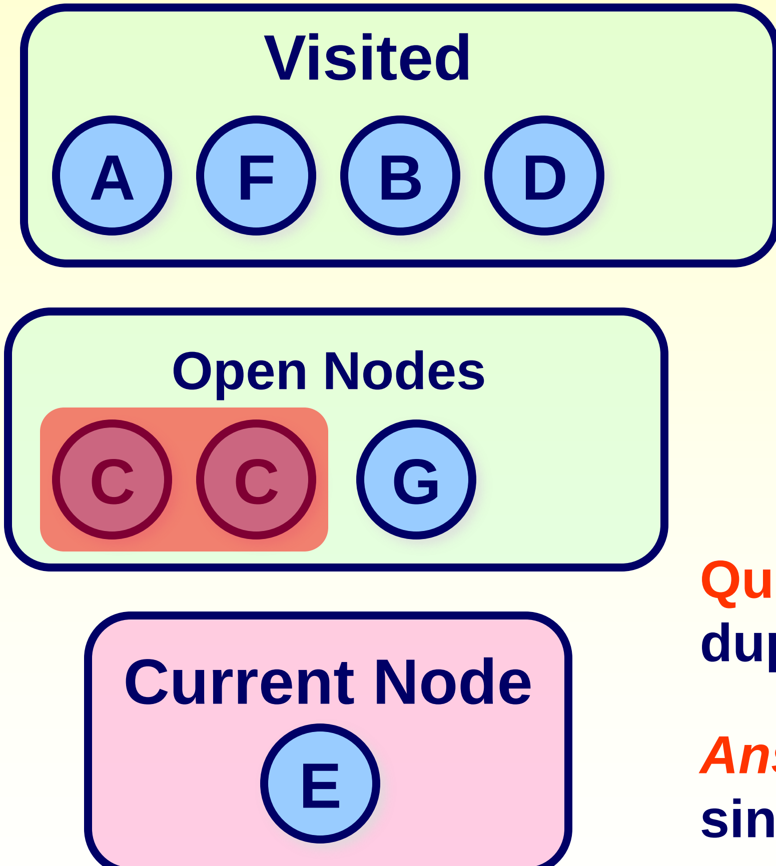


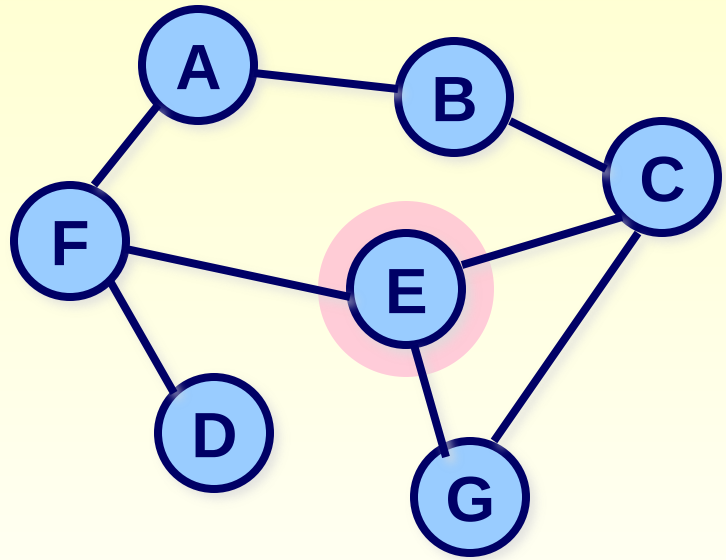


Aren't we done yet? No. We're close, but a simple, plain-vanilla BFS would not check to see if the newly enqueued nodes include the goal node. We'll find it soon enough.

Graphs :: Searching (BFS)

This is an important point: remember this!





Question: Don't we want to purge the duplicate C nodes?

Answer: **NO! Duplicates are harmless, since we check for cycles. Plus, these nodes were contributed by different nodes. As will be seen shortly, this is the key to returning a path.**

Graphs :: Searching (BFS)

Visited

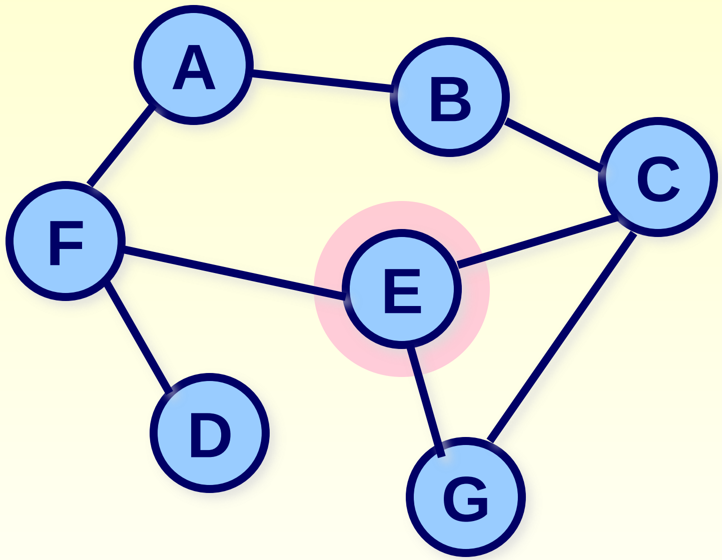
A F B D E

Open Nodes

C C G

Current Node

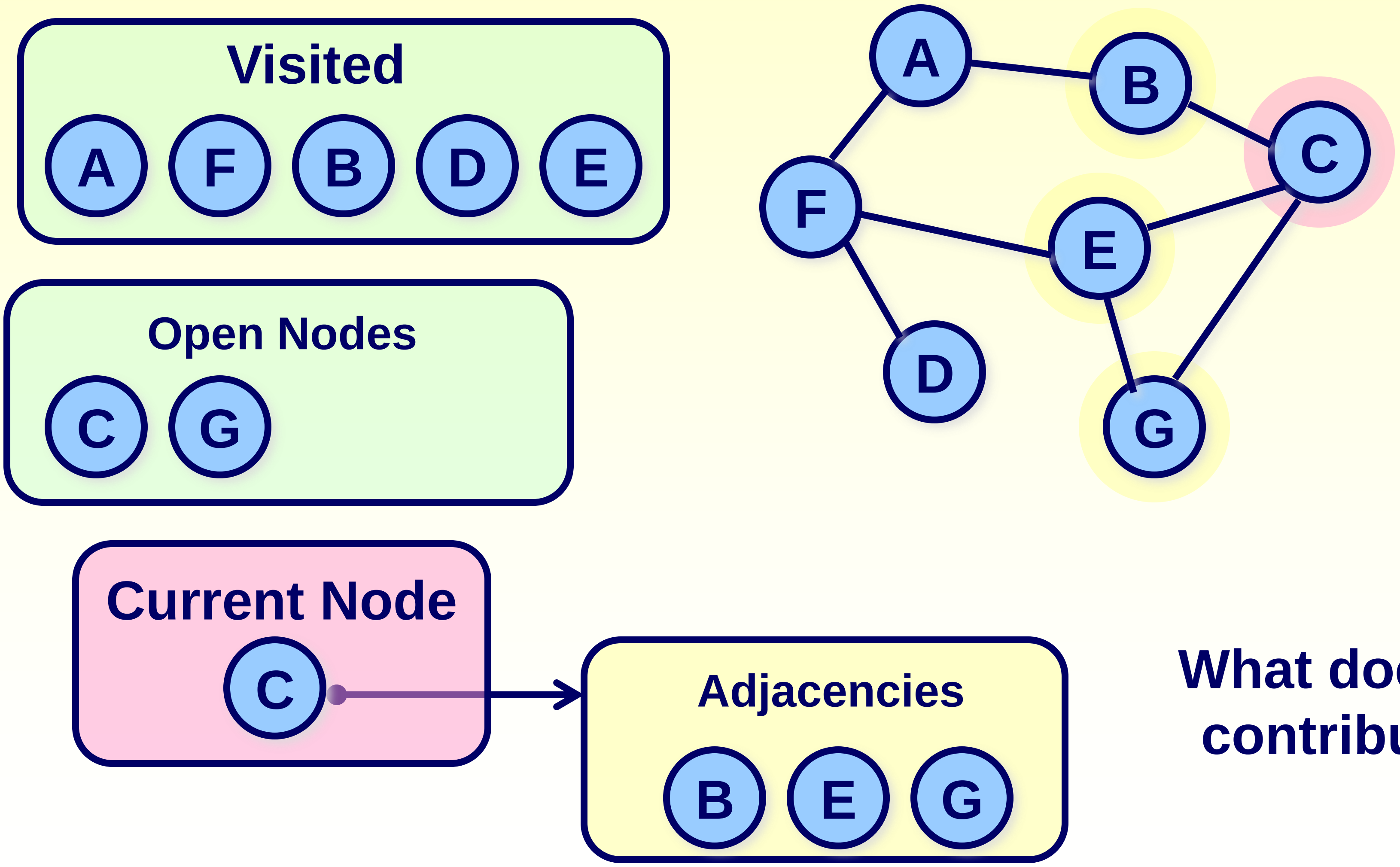
~~E~~



We’re done with E.

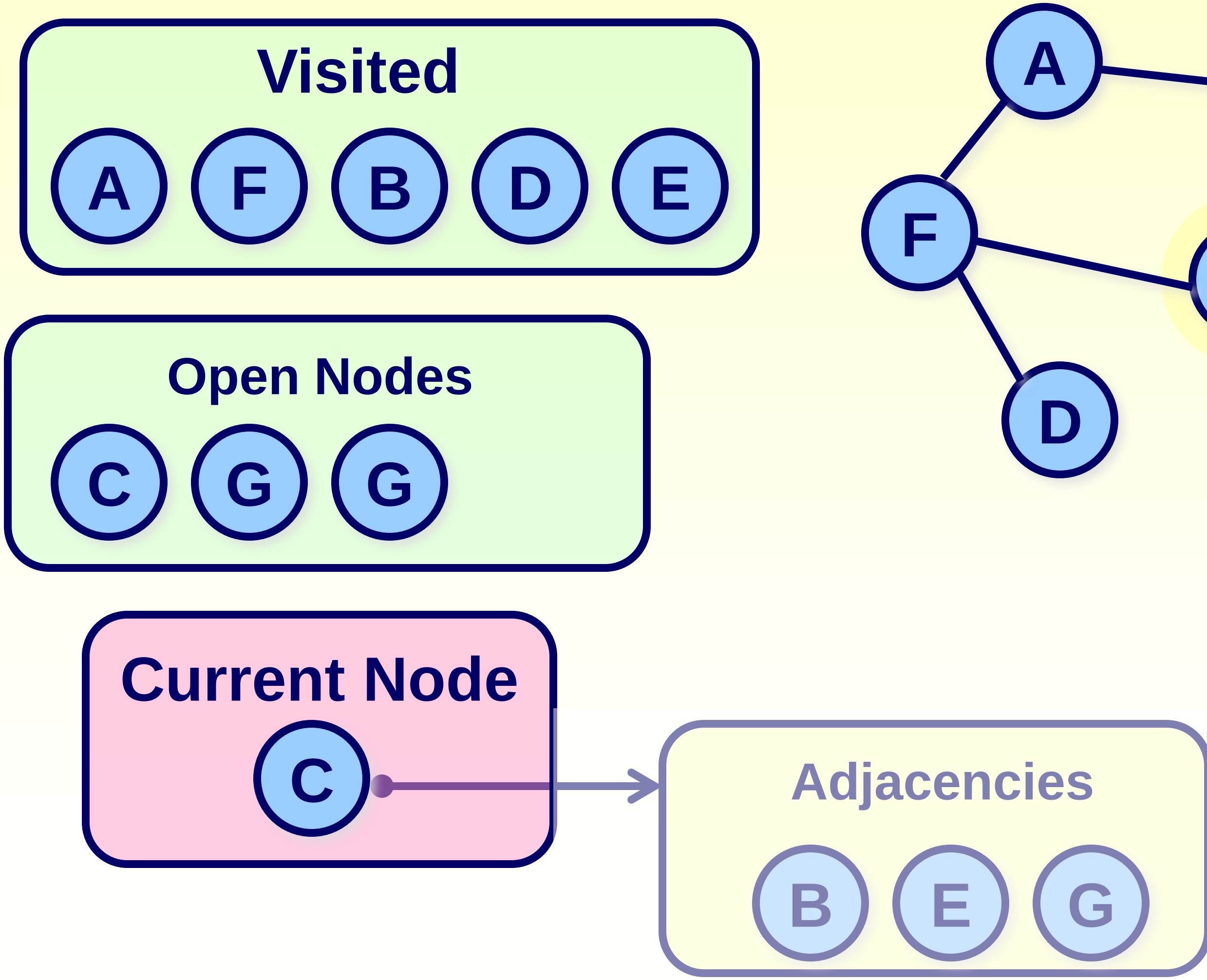
Promote it to the visited list.

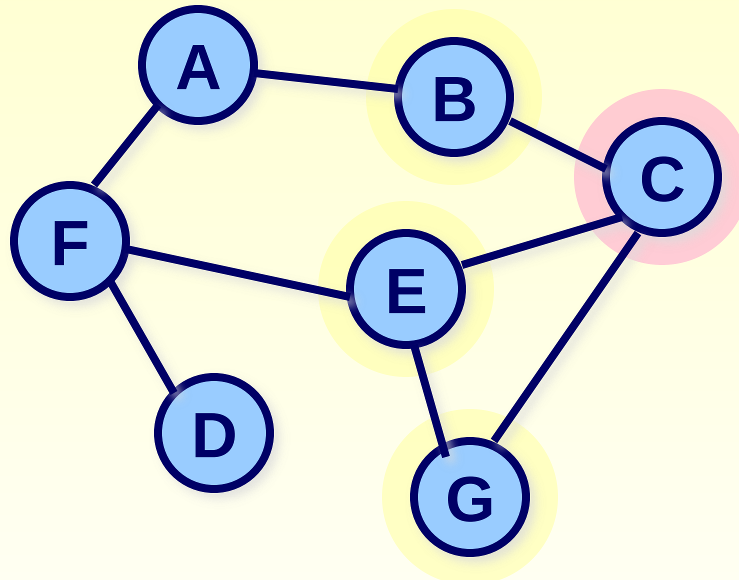
Graphs :: Searching (BFS)



What does C contribute?

Graphs :: Searching (BFS)

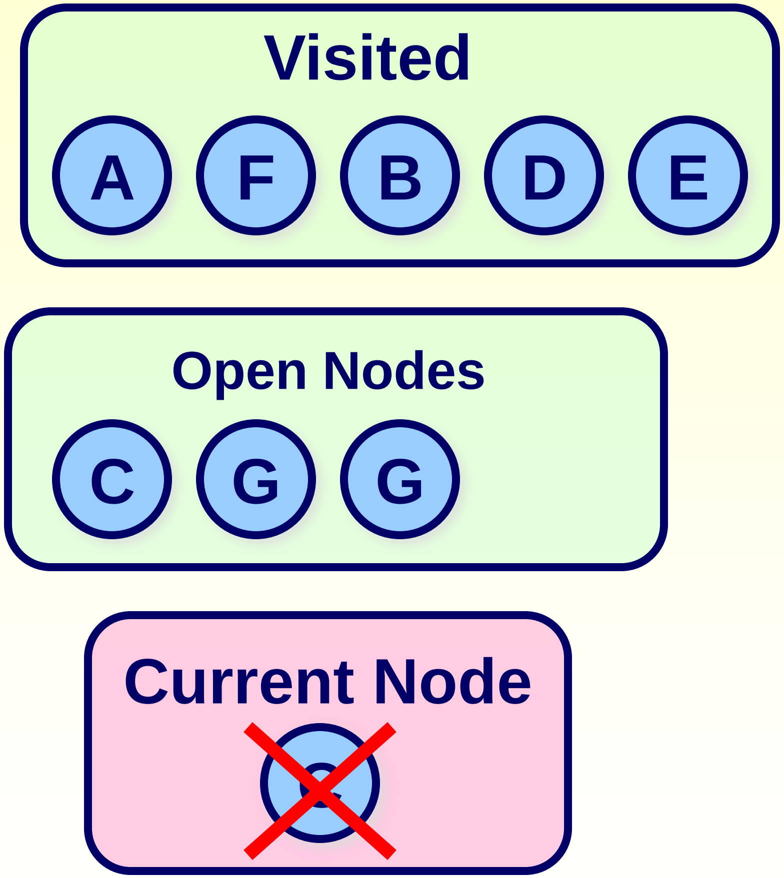


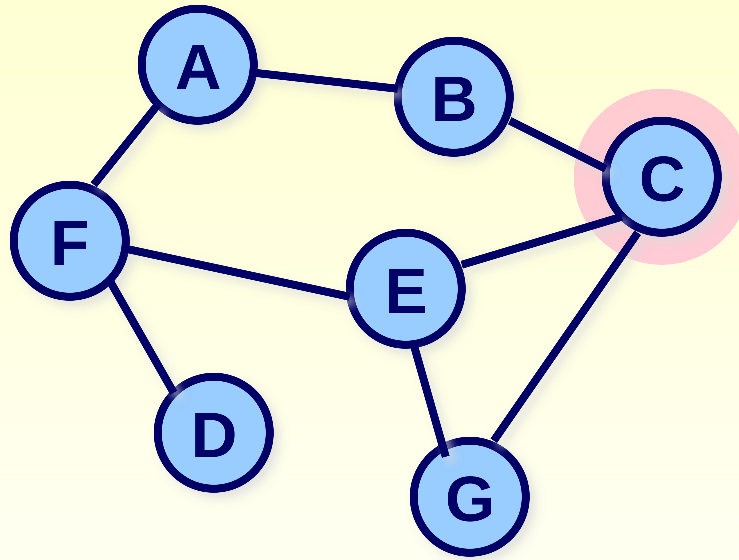


What does C contribute?

Another link to G, but from another path; the rest cycle

Graphs :: Searching (BFS)

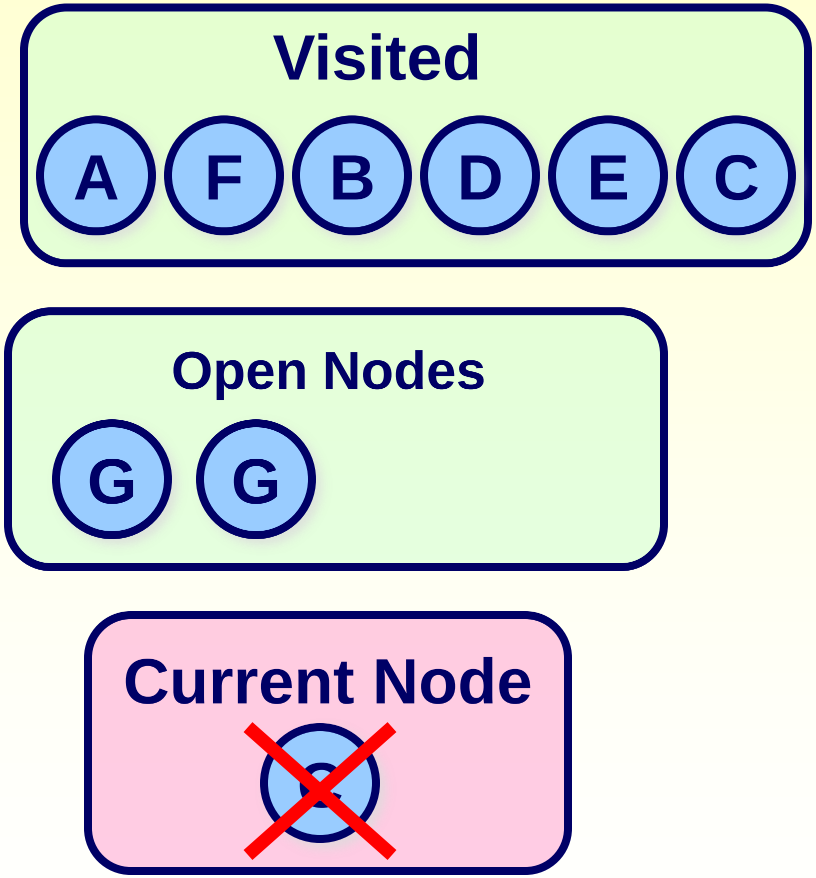


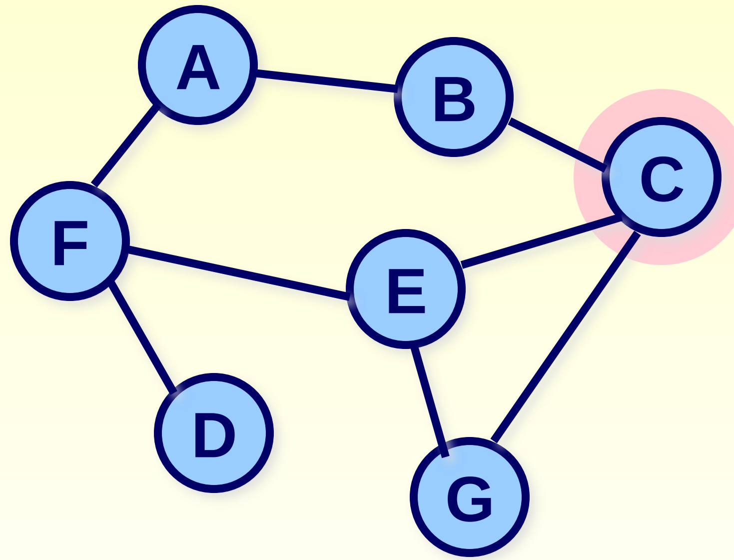


We’re done with C.

The next node up is C again.

Graphs :: Searching (BFS)

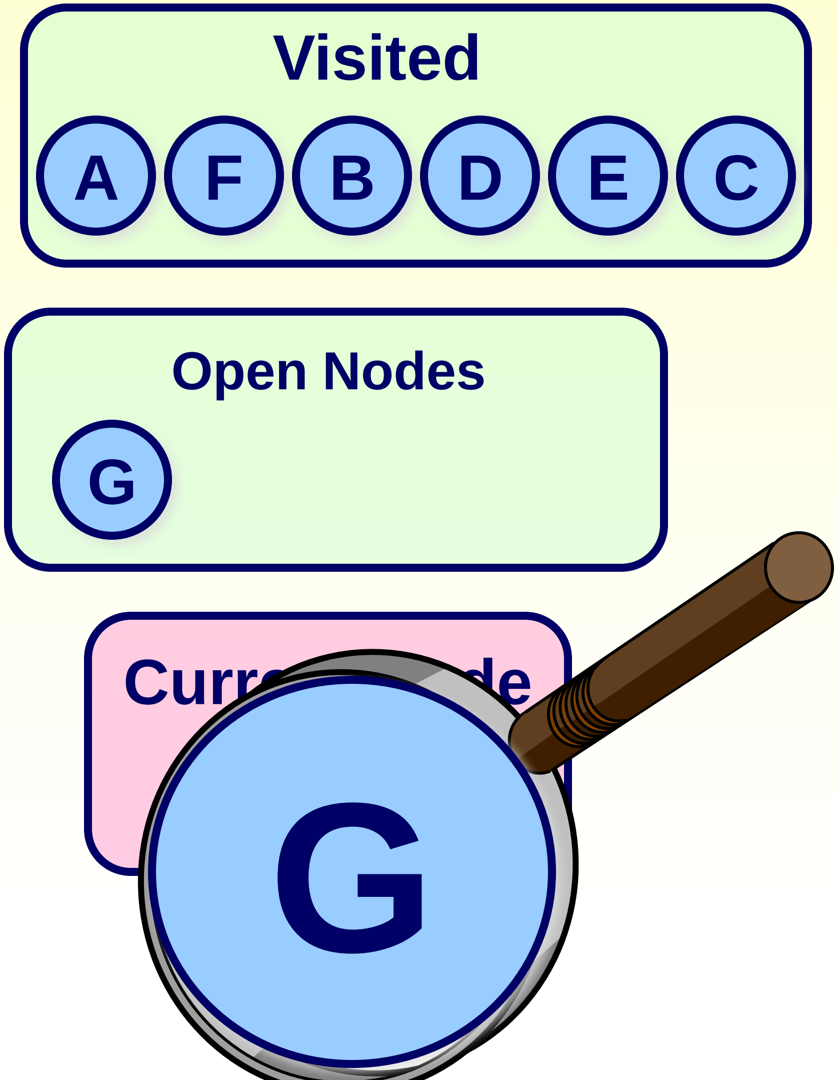


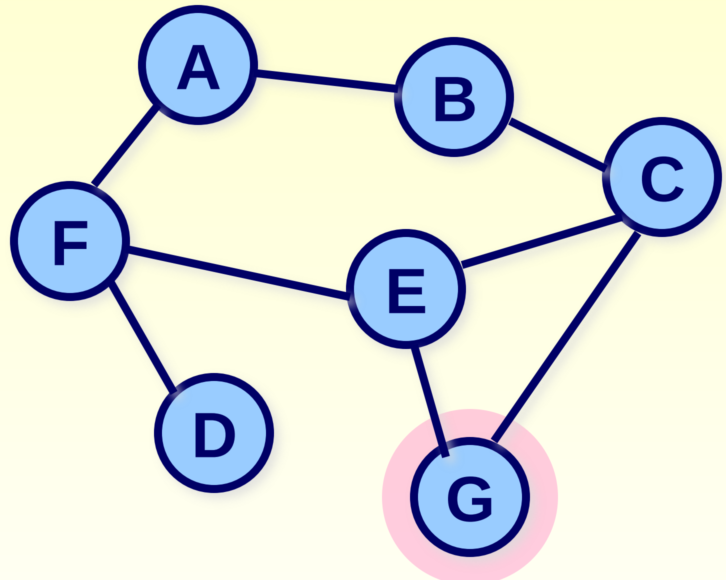


C has just been done, so there’s nothing new it can add . . .

The next node up is ...

Graphs :: Searching (BFS)



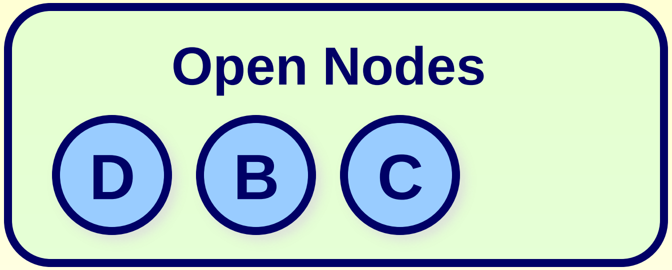


G, the goal node.

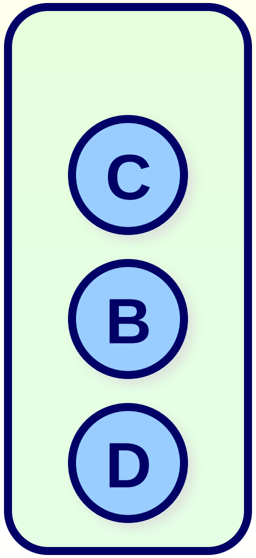
We' re done.

Observations

We used a queue for a BFS.



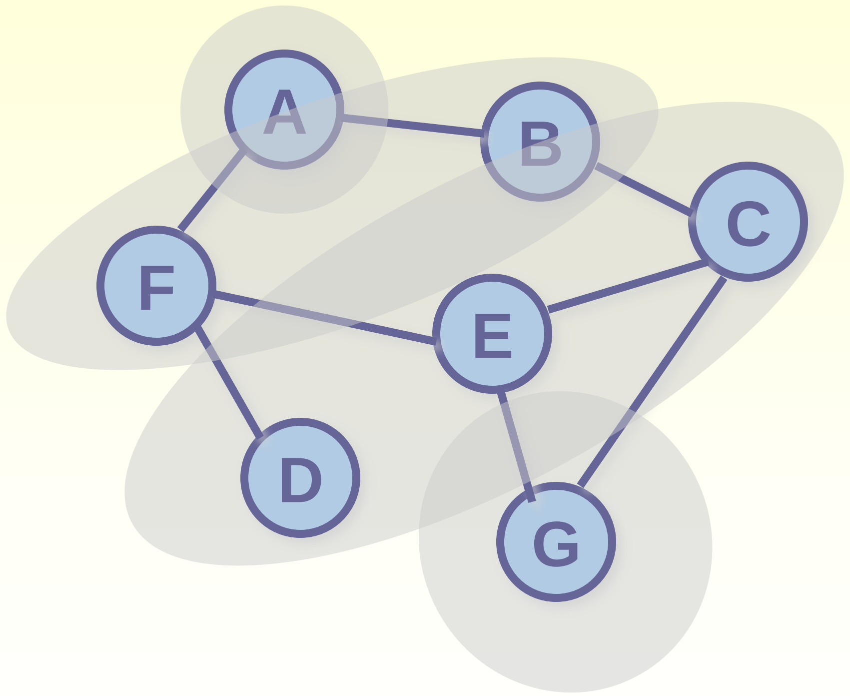
If we instead had used a stack, we would have performed a DFS.



This sometimes results in a different path, although both DFS and BFS are exhaustive searches. If there's a path, either will find it.

BFS: Step-by-Step

BFS, because it uses a queue, will examine all nodes one step away, then two steps away, then three, etc.



Breadth-First Search BFS

Problem: It is another systematic way of visiting the vertices of a graph G. Start from a vertex, step forward all vertices adjacent to it, then step forward all vertices adjacent to its sons,...

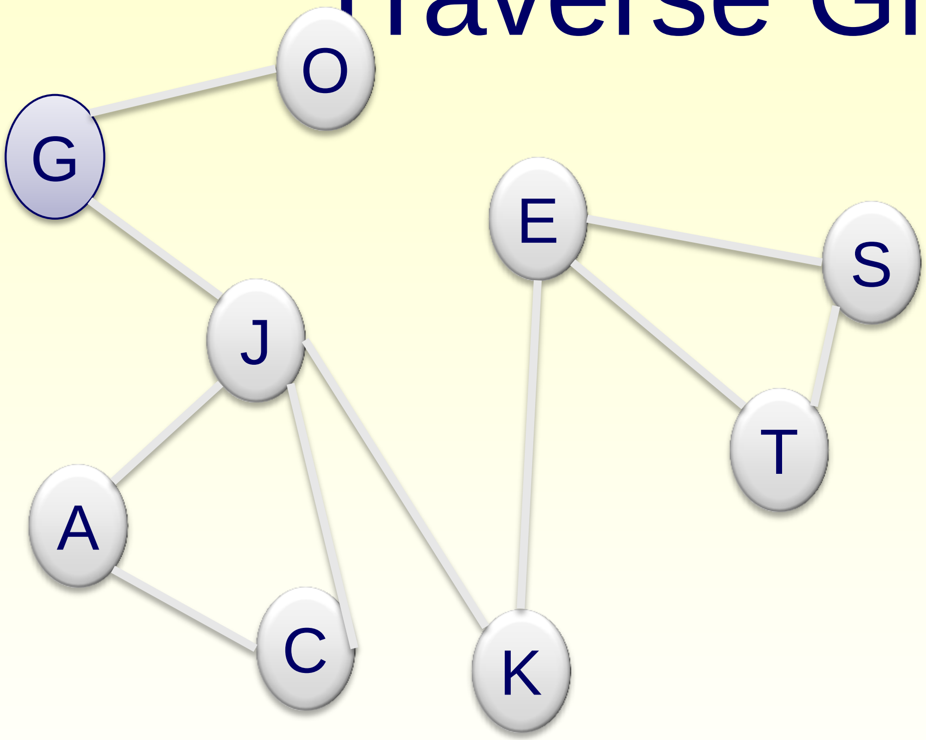
The Breadth-First Search algorithm is quite the same algorithm as the iterative DFS, you simply replace the stack with a queue

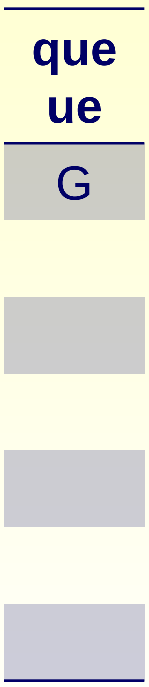
- BFS is classic method to find a path with the fewest nodes from source vertex v to target vertex u.

BFS Pseudo Code

```
bfs(s)
  initialize Q to be a queue with one element s
  while Q not empty
    take a node u from Q
    if explored[u]=false then
      set explored[u]= true
      for each edge (u,v) adjacent to u
        add v to Q
      end
    end
end
```

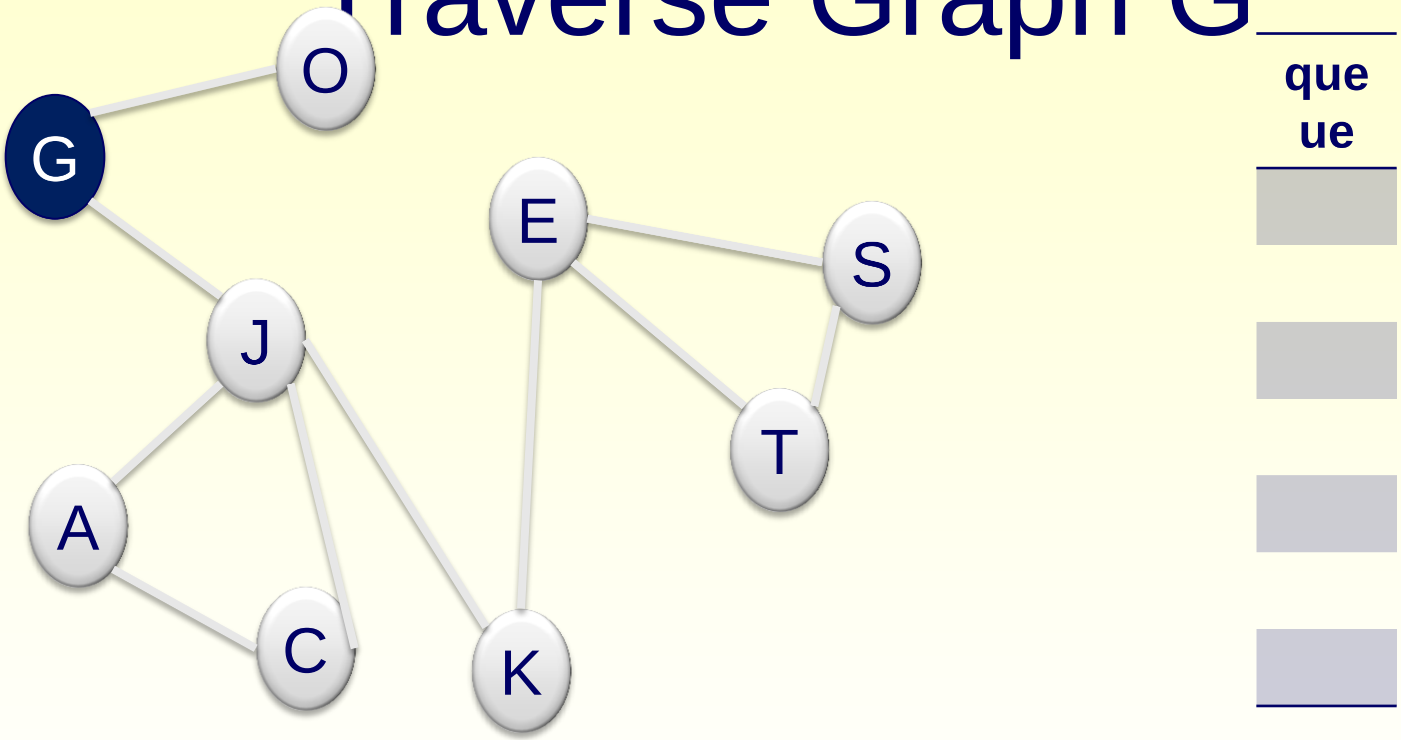
BFS Example
Traverse Graph G





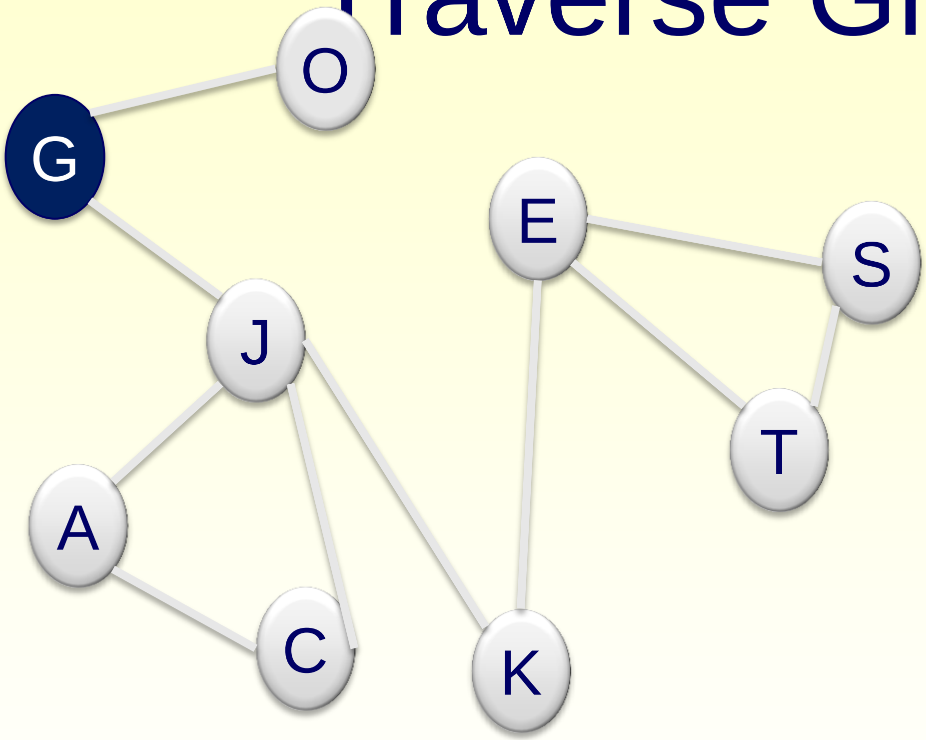
Output: G

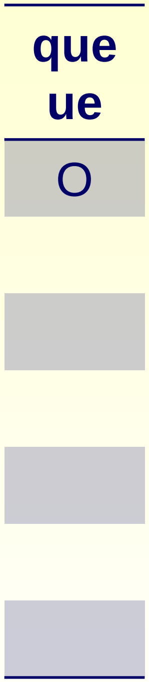
BFS Example
Traverse Graph G



Output: G

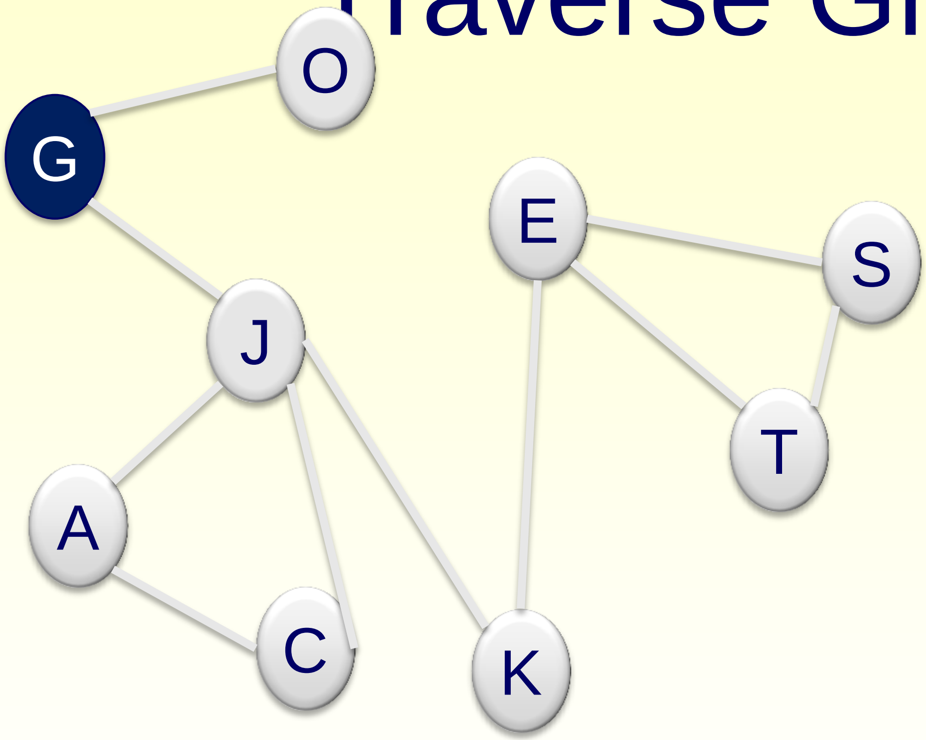
BFS Example
Traverse Graph G

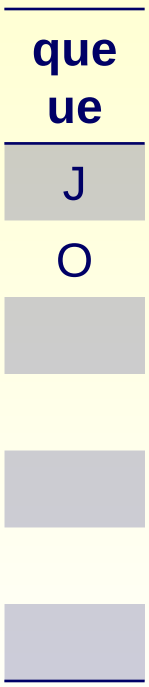




Output: G O

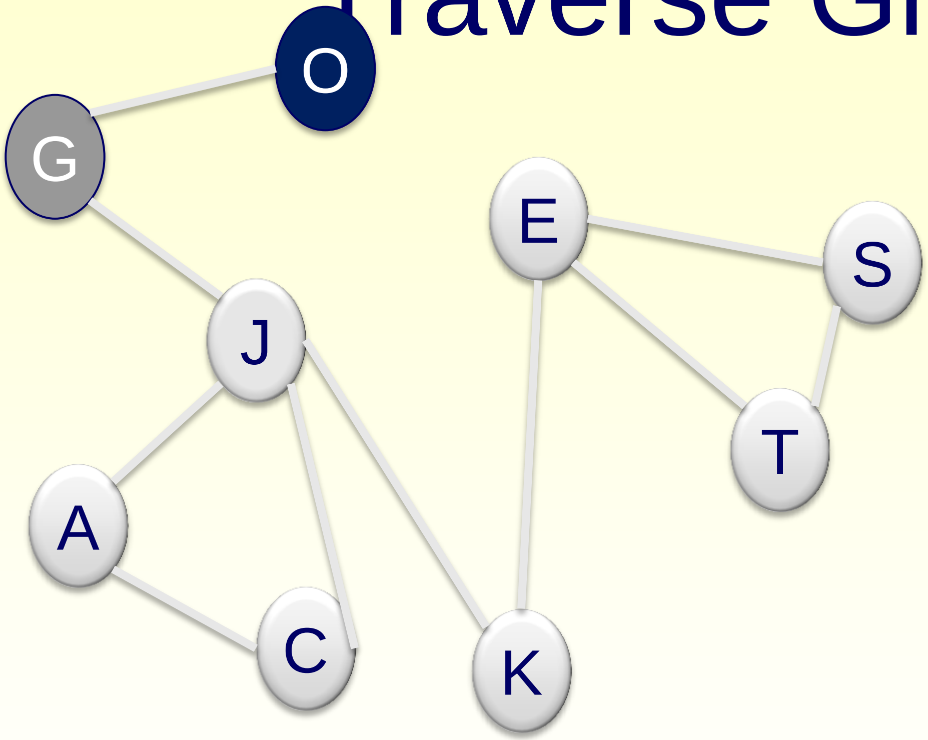
BFS Example Traverse Graph G

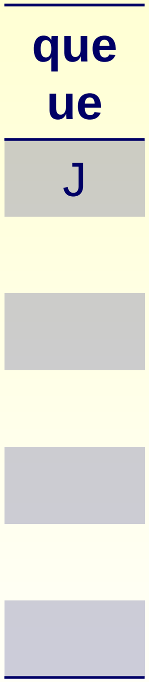




Output: G O J

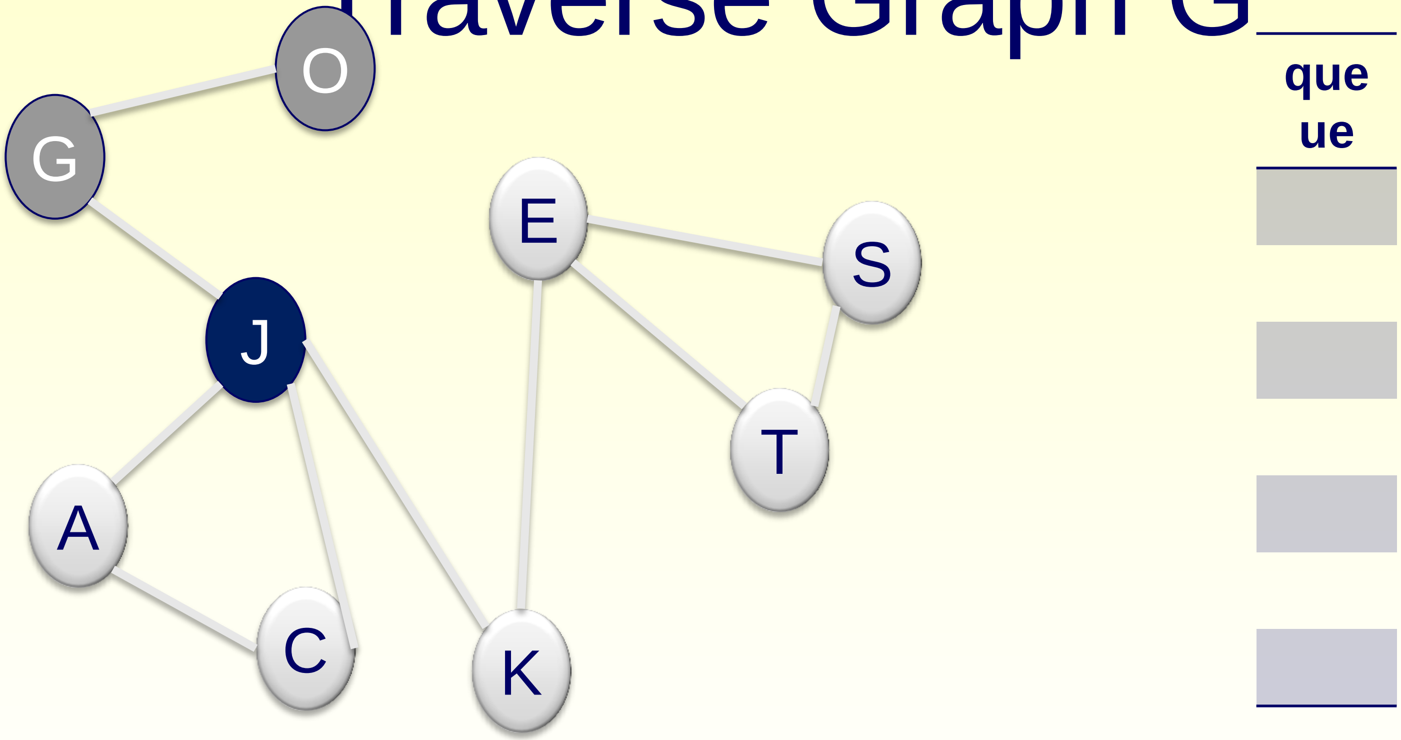
BFS Example
Traverse Graph G





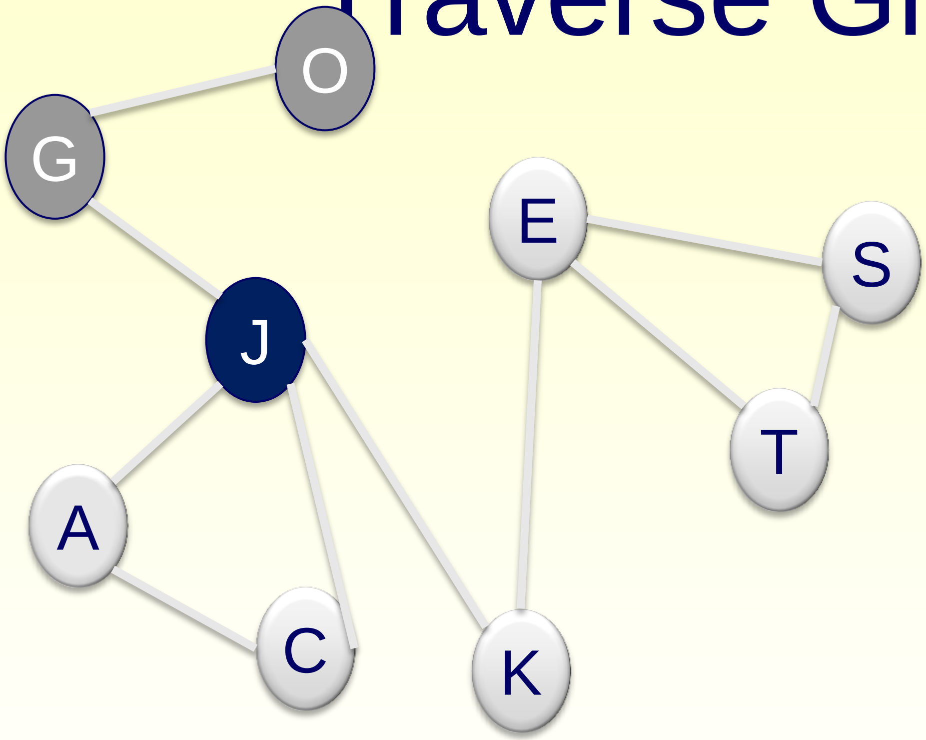
Output: G O J

BFS Example
Traverse Graph G



Output: G O J

BFS Example
Traverse Graph G

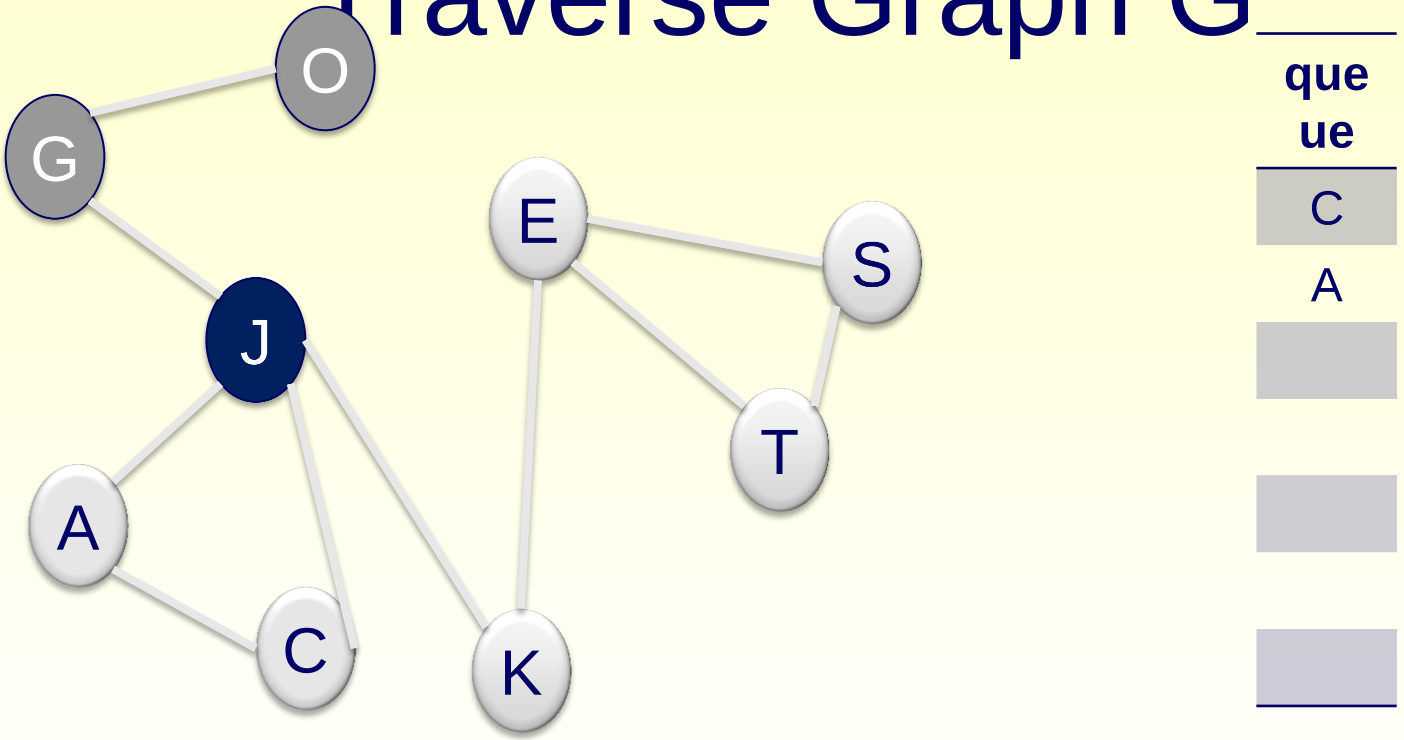


queue

A

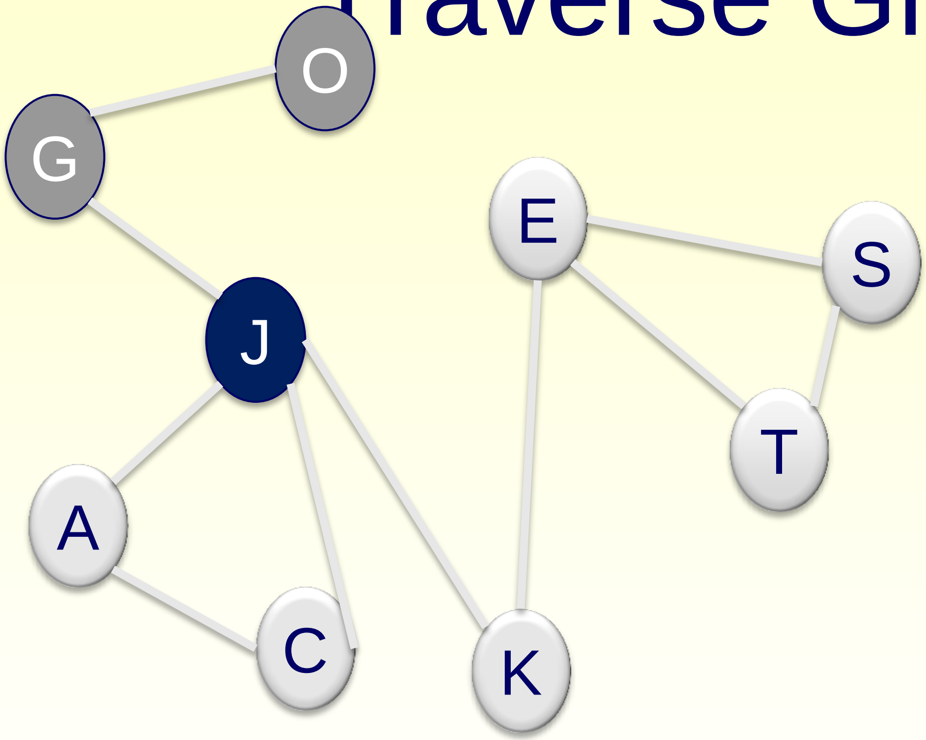
Output: G O J A

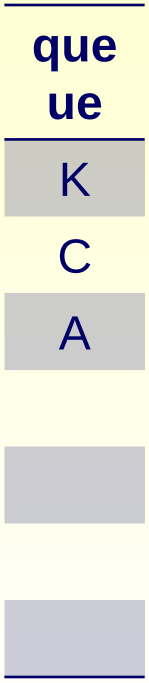
BFS Example
Traverse Graph G



Output: G O J A C

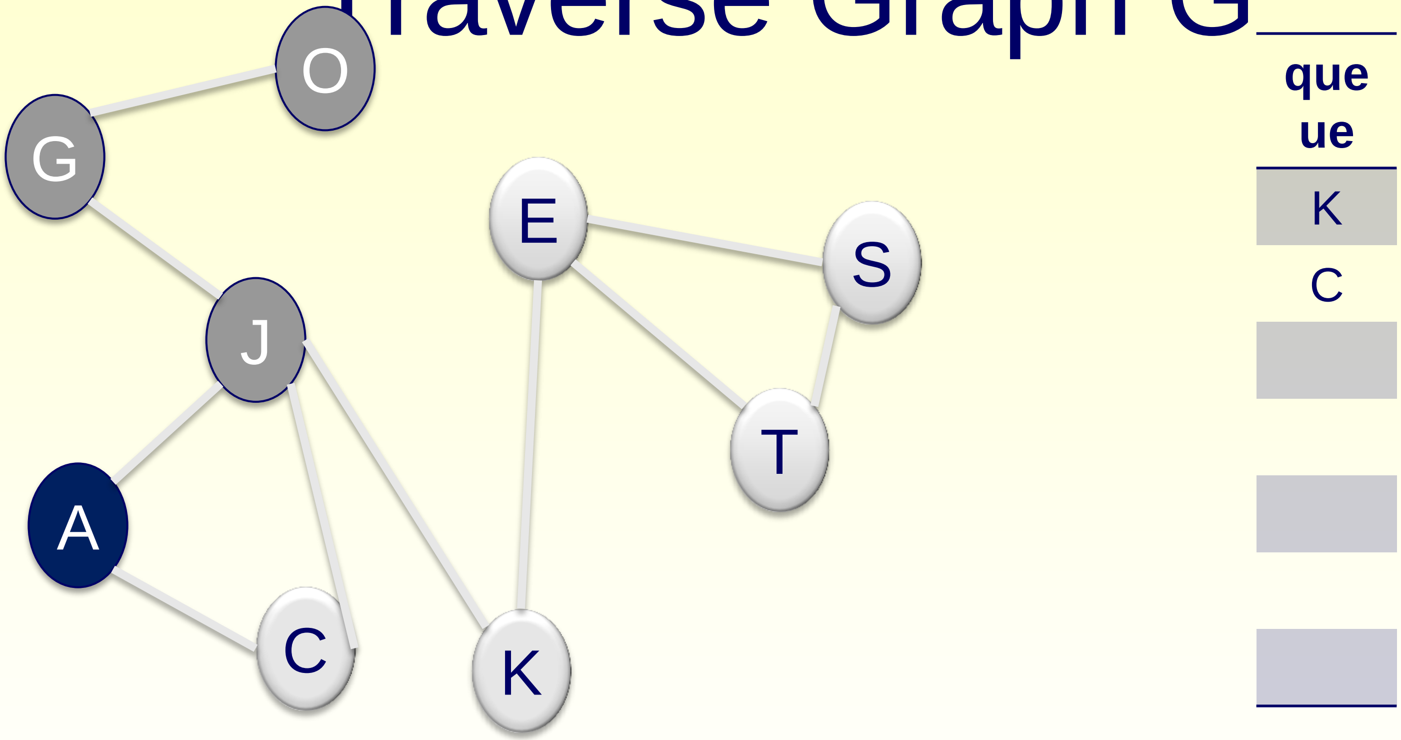
BFS Example Traverse Graph G





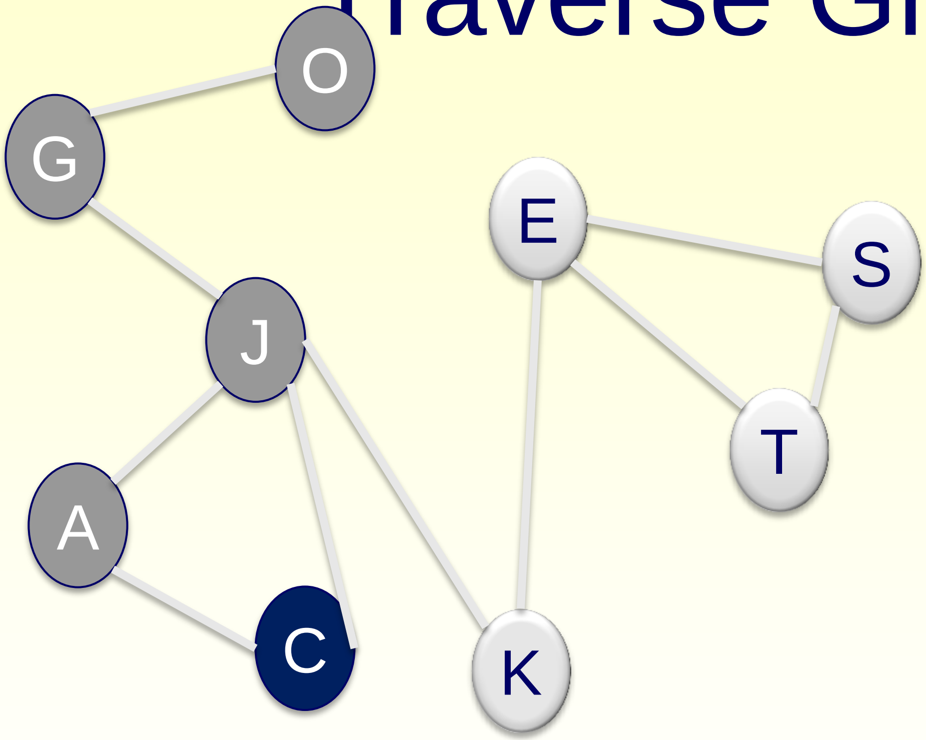
Output: G O J A C K

BFS Example
Traverse Graph G



Output: G O J A C K

BFS Example
Traverse Graph G

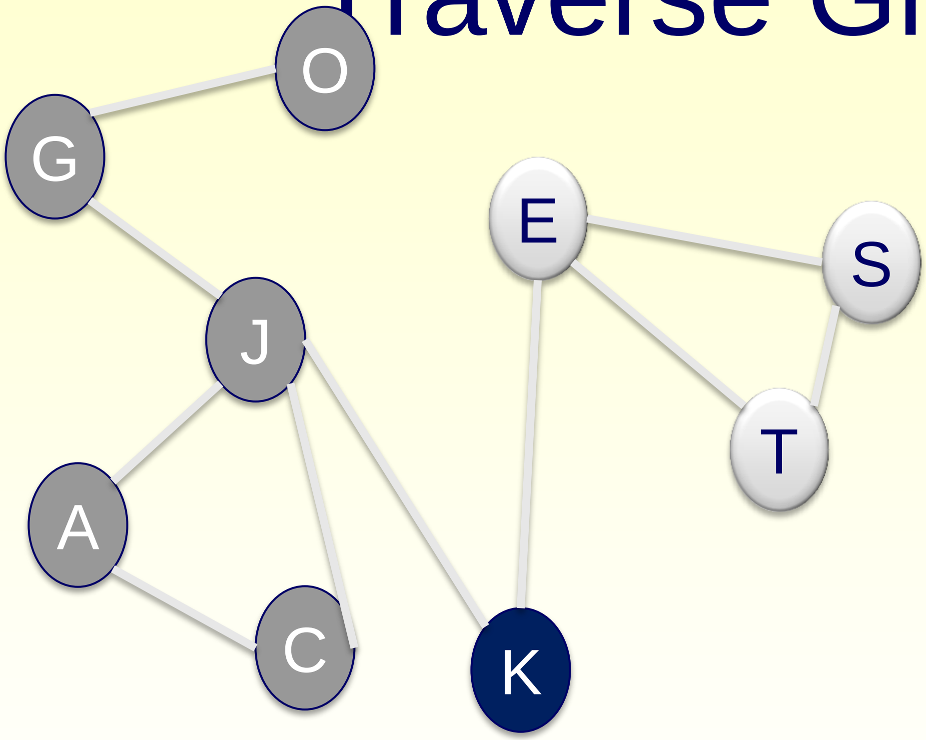


que
ue

K

Output: G O J A C K

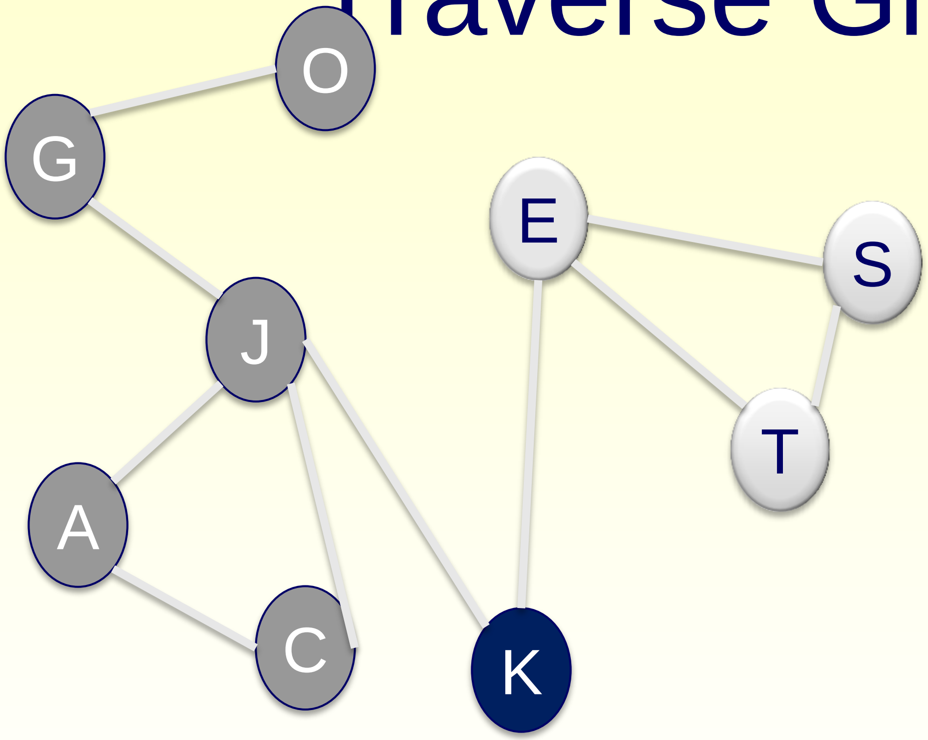
BFS Example
Traverse Graph G

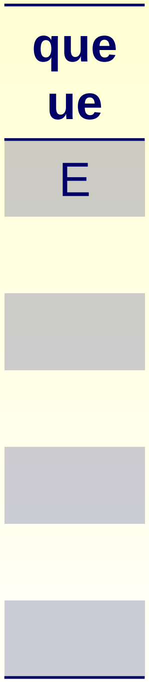


que ue

Output: G O J A C K

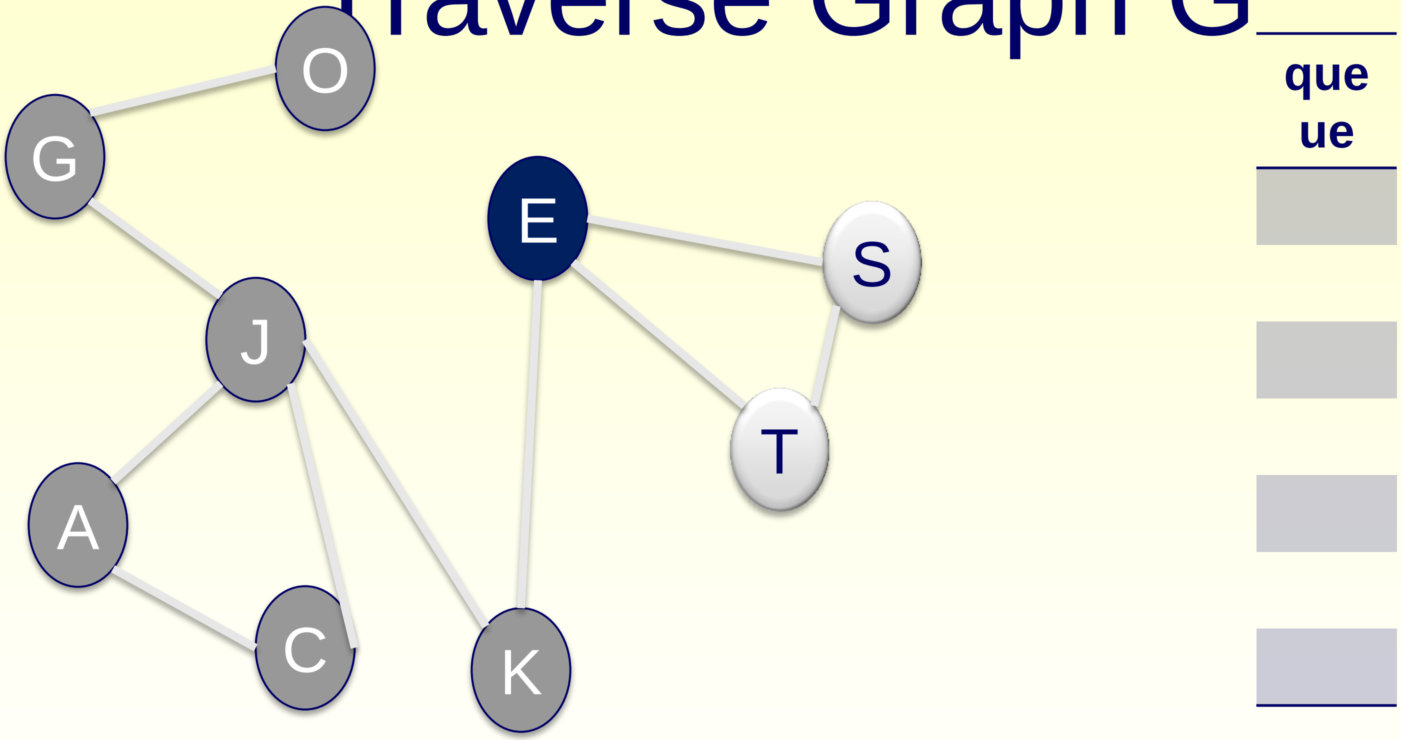
BFS Example
Traverse Graph G





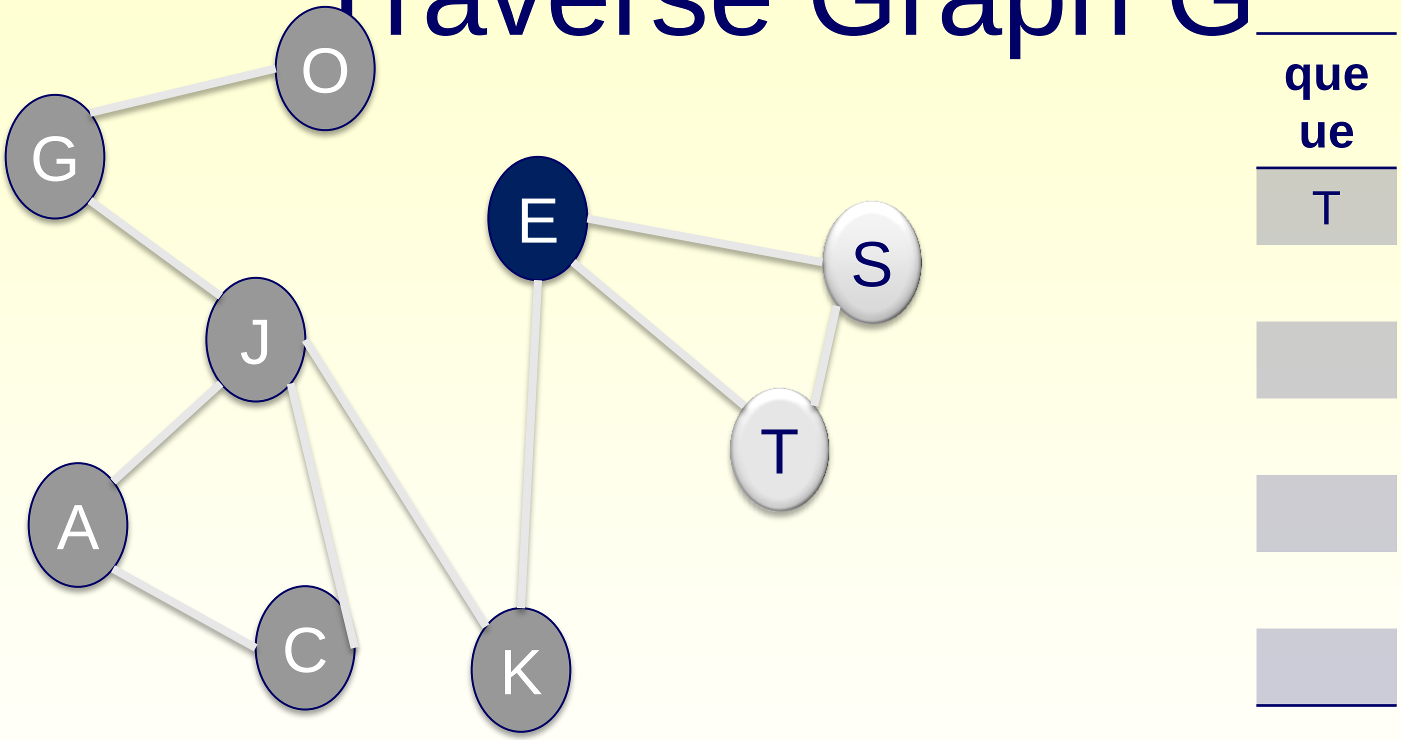
Output: G O J A C K E

BFS Example
Traverse Graph G



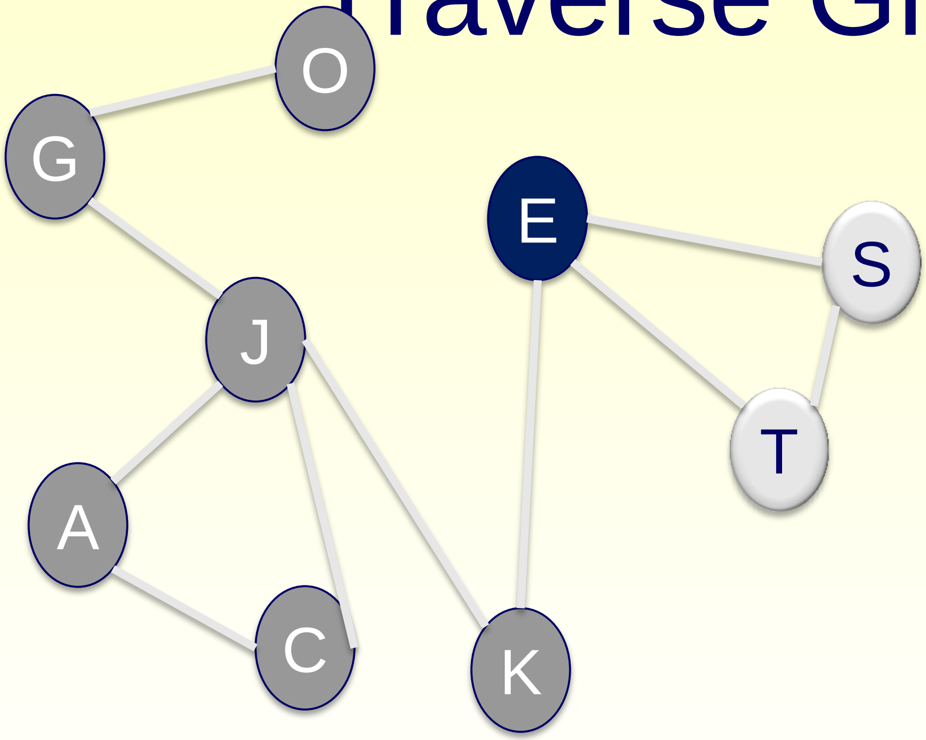
Output: G O J A C K E

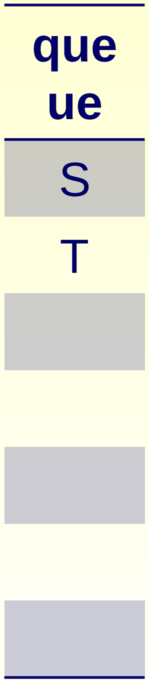
BFS Example
Traverse Graph G



Output: G O J A C K E T

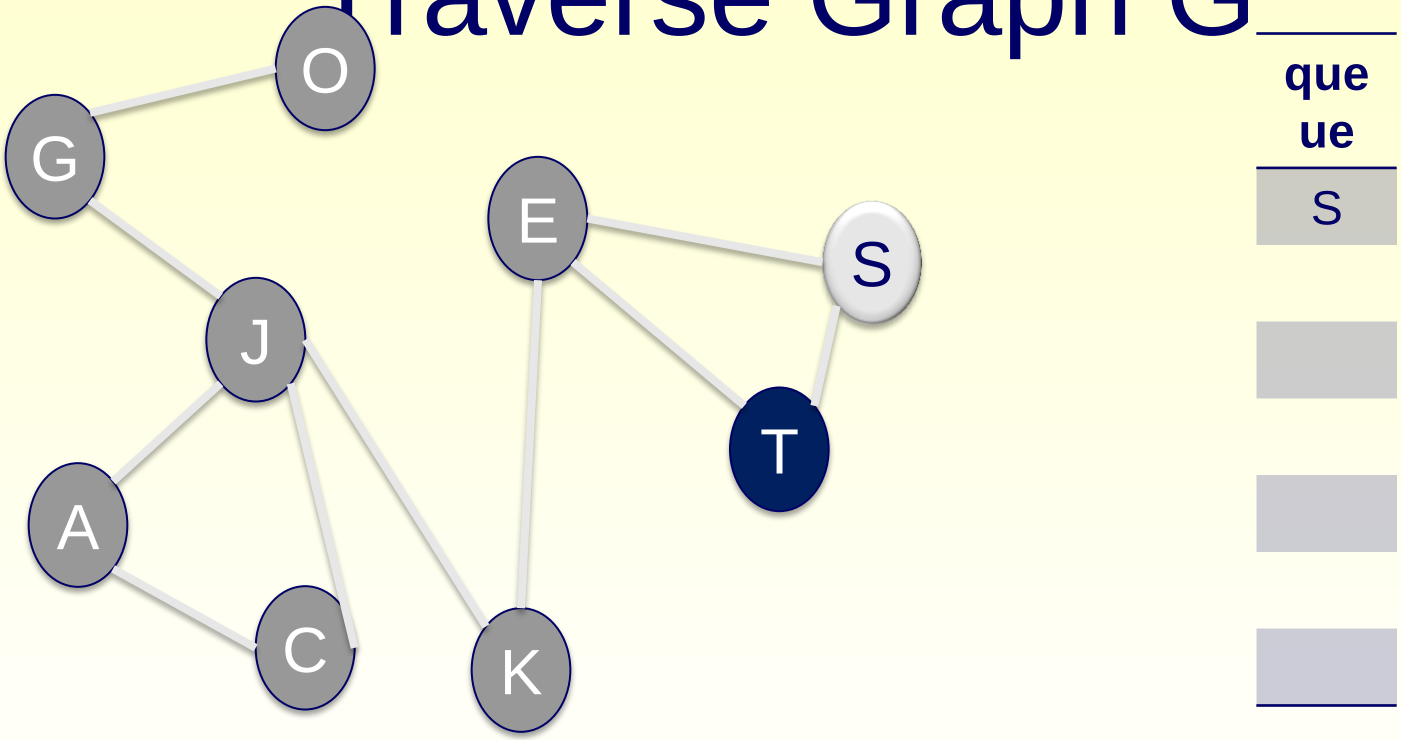
BFS Example
Traverse Graph G





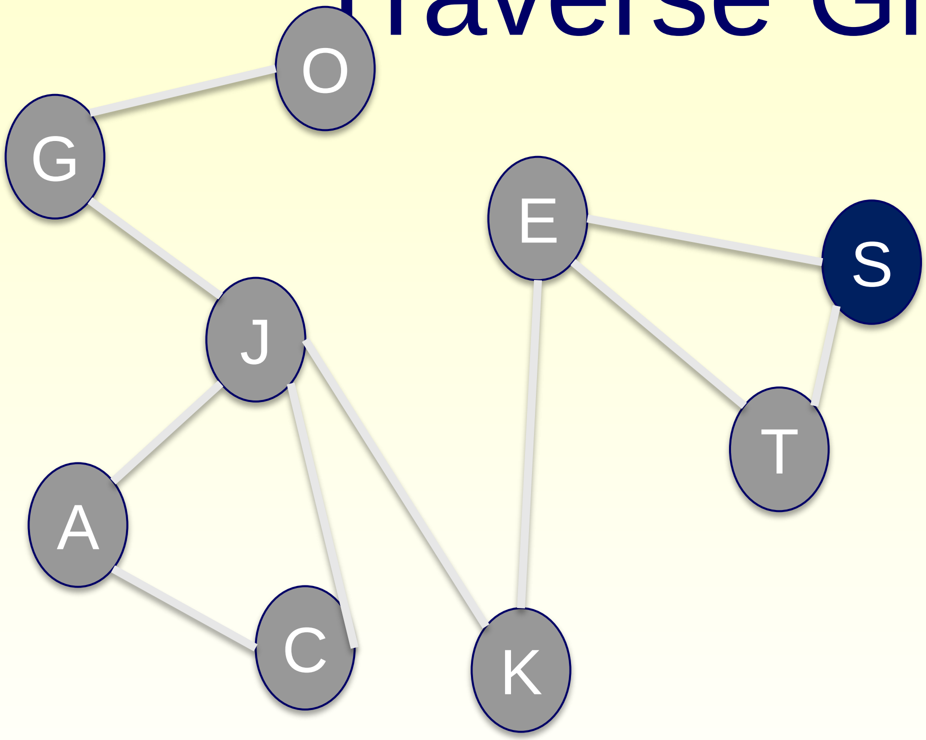
Output: G O J A C K E T S

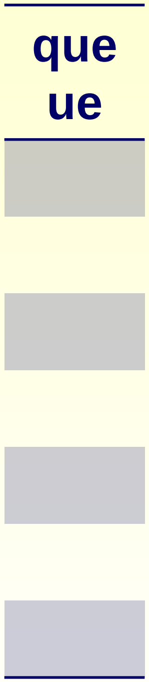
BFS Example
Traverse Graph G



Output: G O J A C K E T S

BFS Example
Traverse Graph G

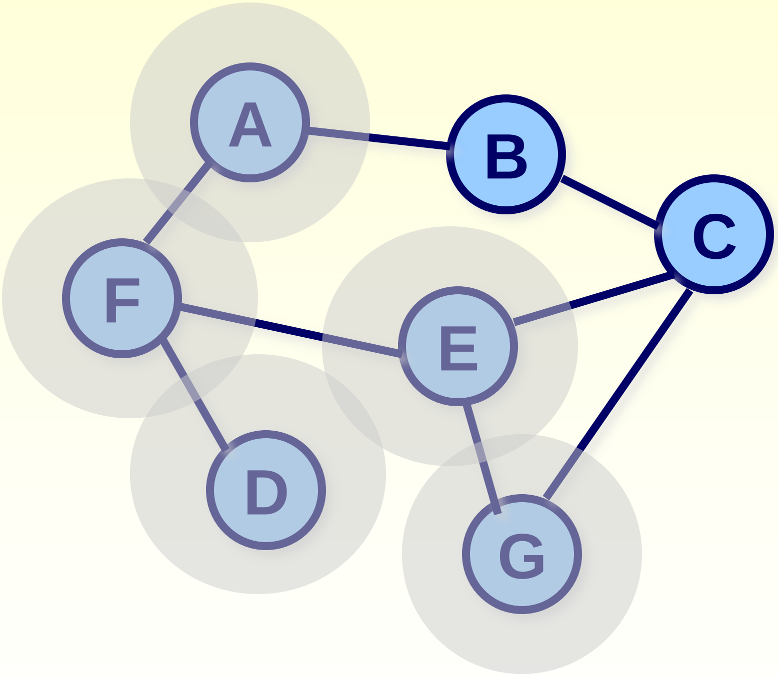




Output: G O J A C K E T S

Now DFS: Leap then Look

Because it uses a stack, when the DFS discovers a new node, it races down that branch . . .



Only if it hits a dead end will it back up and examine other adjacent nodes.

Depth-First Search DFS

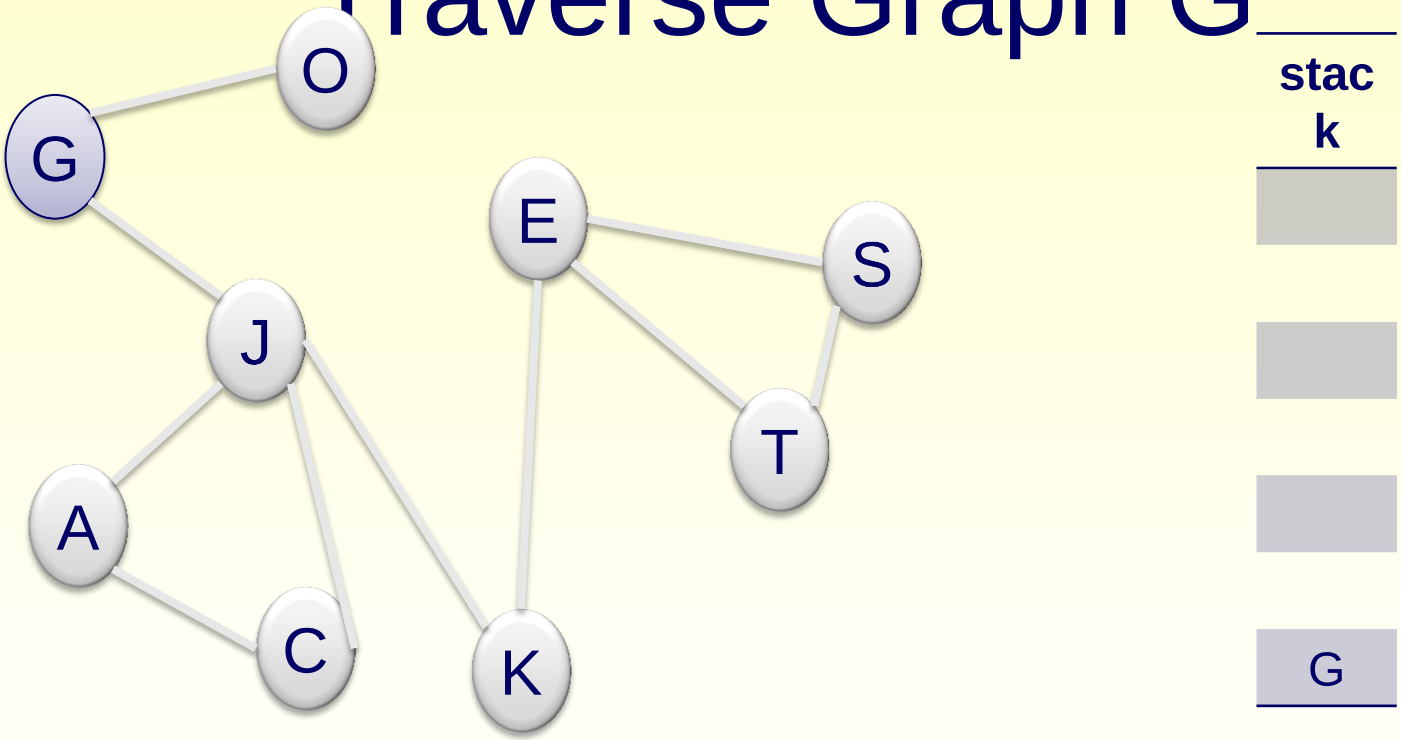
Problem Find a natural way to systematically visit every vertex and every edge of a graph :

- Start from one vertex
- Move forward all along <u>one</u> path (do not pass through a vertex already visited)
- When stuck, turn back until you can step forward to an unvisited vertex
- DFS finds some path from source vertex v to target vertex u.

DFS Pseudo Code

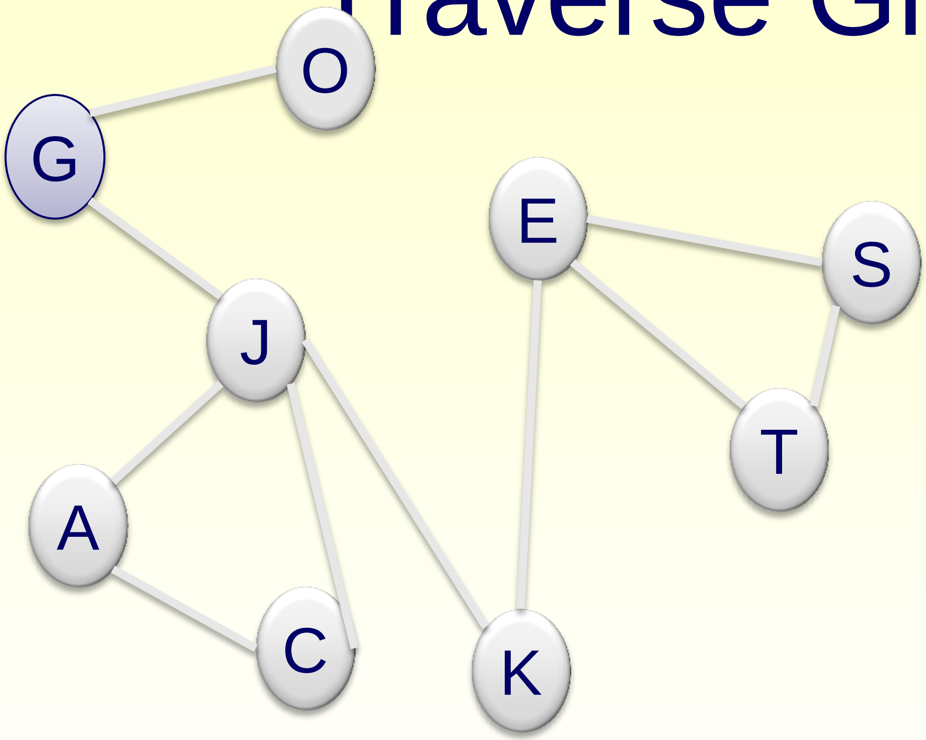
```
dfs(v)
  visit(v)
  for each neighbor w of v
    if w is unvisited
      dfs(w)
      add edge vw to tree T
    end
  end
end
```

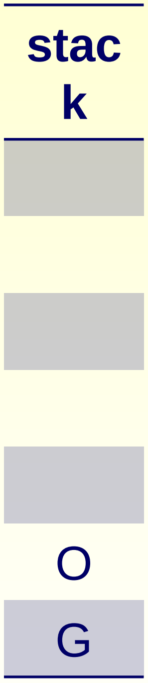
DFS Example
Traverse Graph G



Output: G

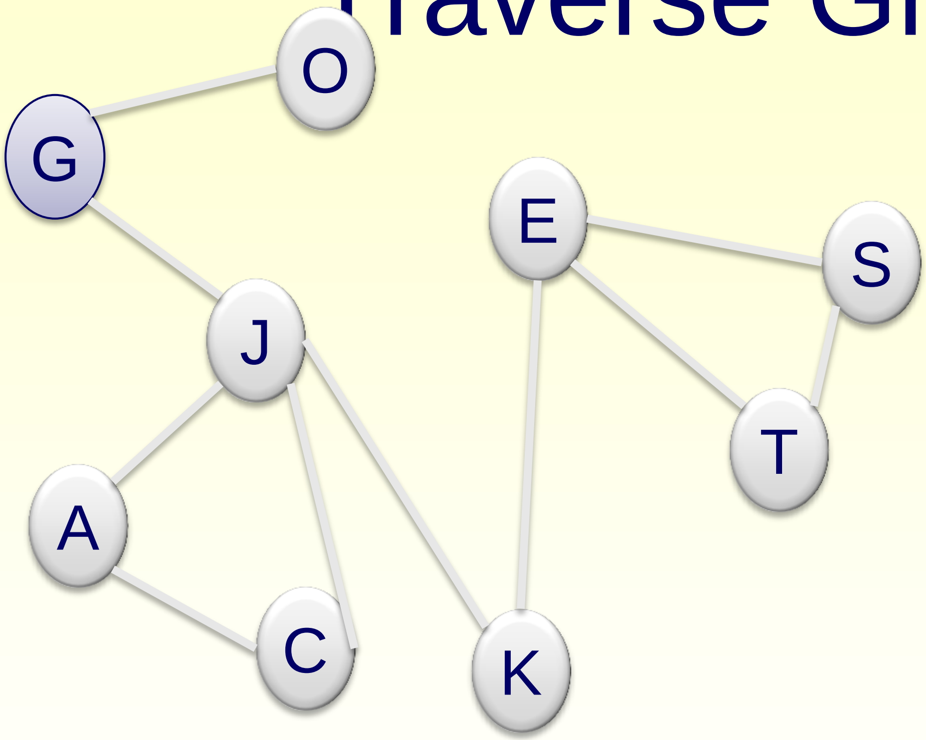
DFS Example
Traverse Graph G

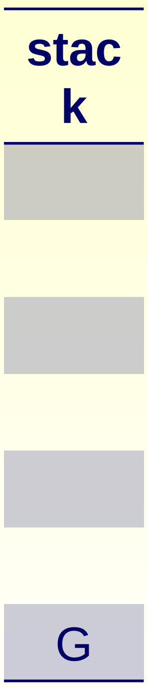




Output: G O

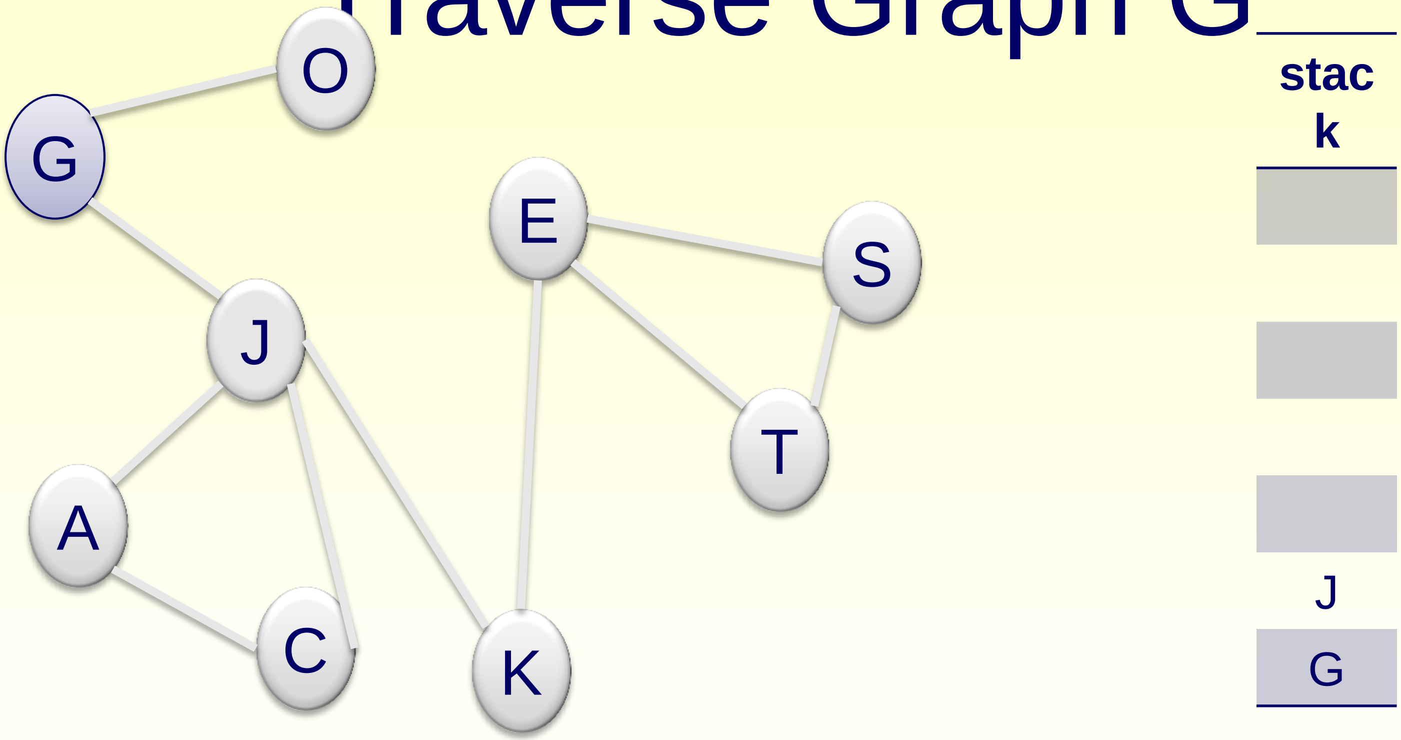
DFS Example
Traverse Graph G





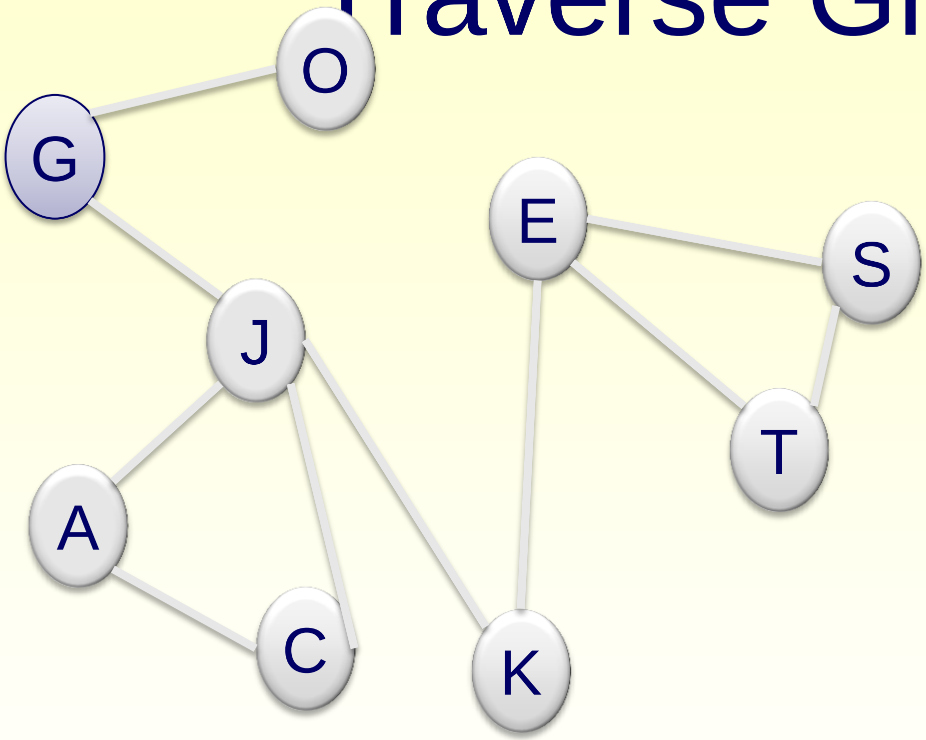
Output: G O

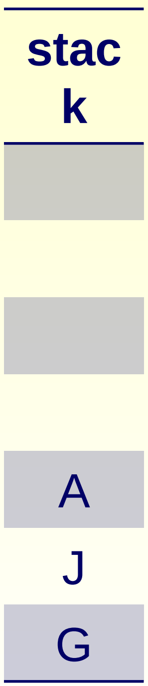
DFS Example
Traverse Graph G



Output: G O J

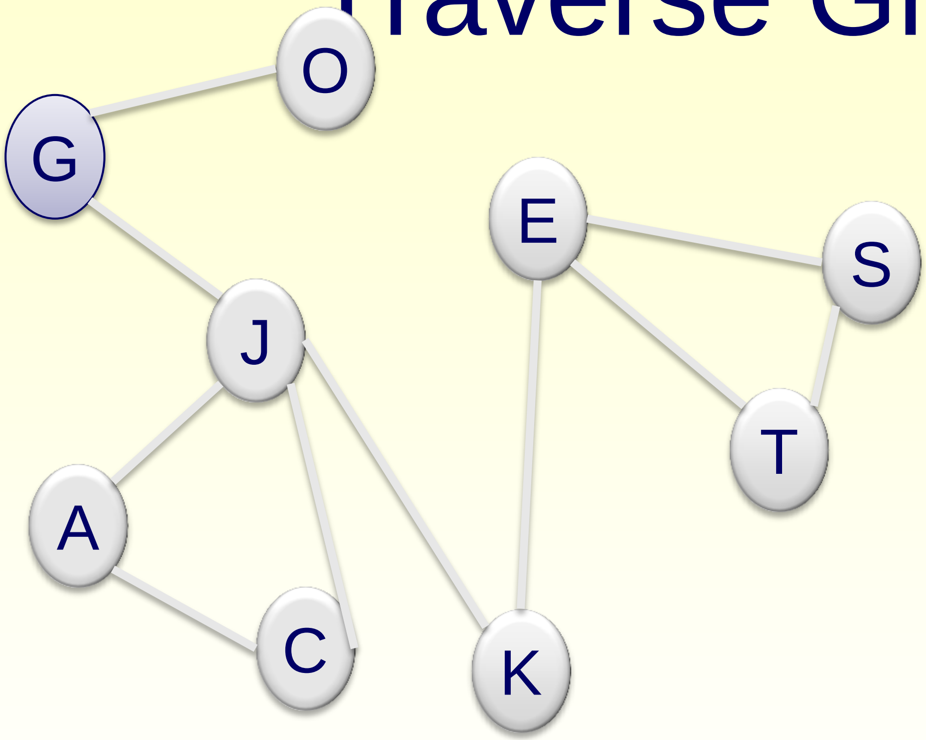
DFS Example
Traverse Graph G

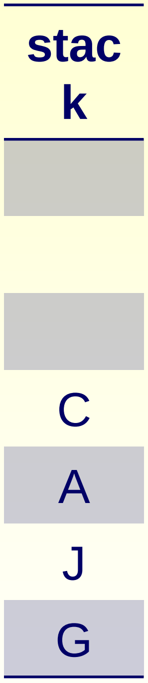




Output: G O J A

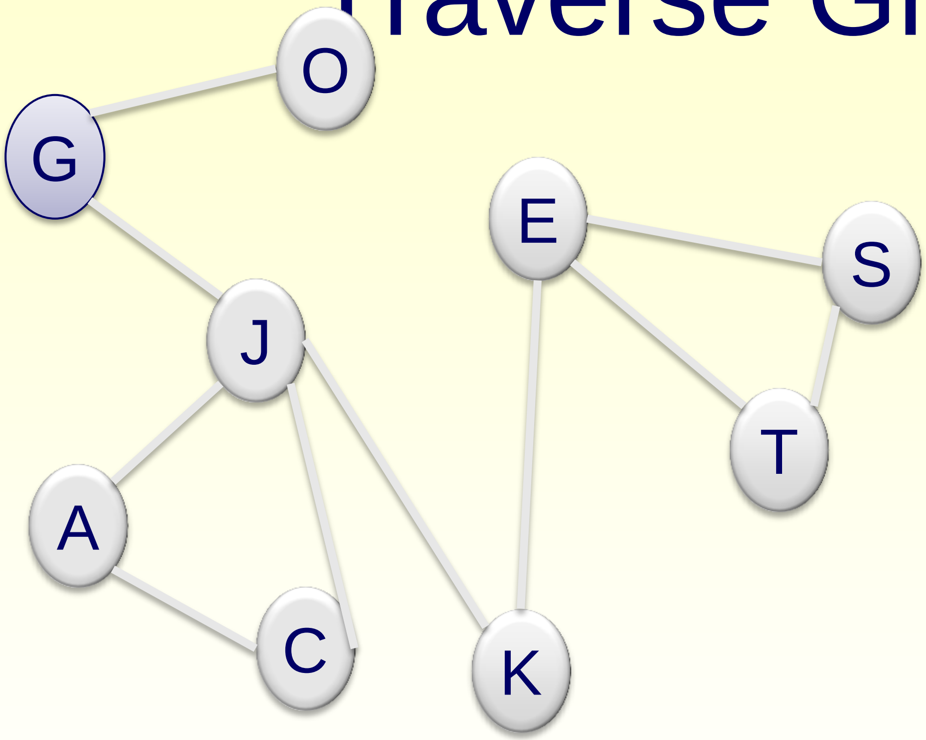
DFS Example
Traverse Graph G

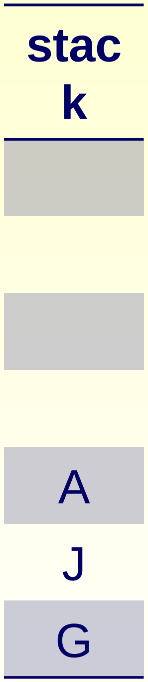




Output: G O J A C

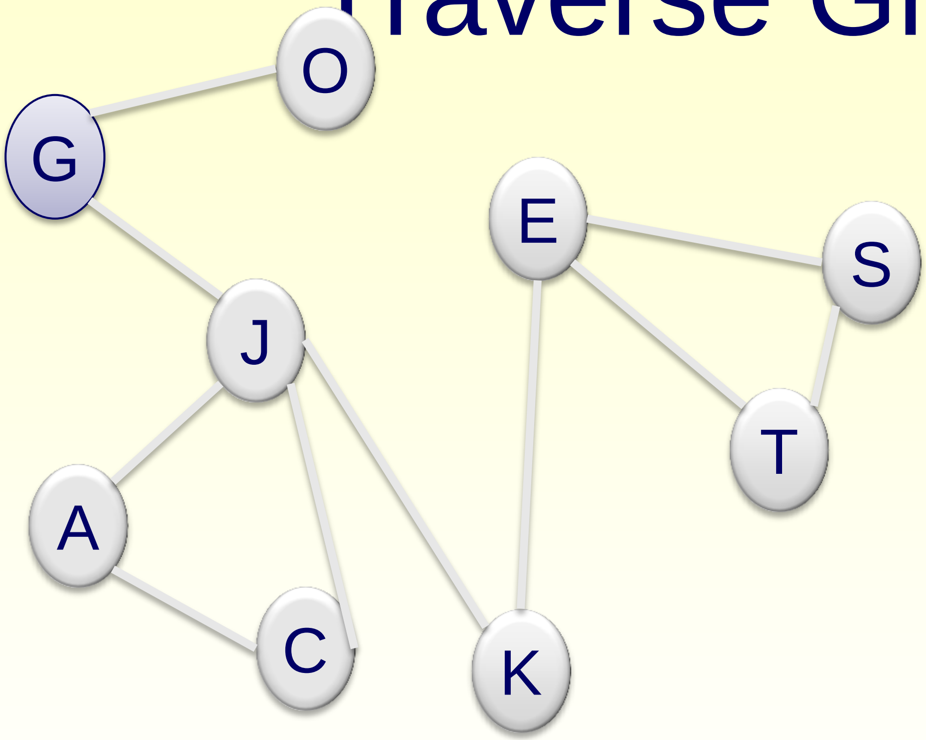
DFS Example
Traverse Graph G





Output: G O J A C

DFS Example
Traverse Graph G



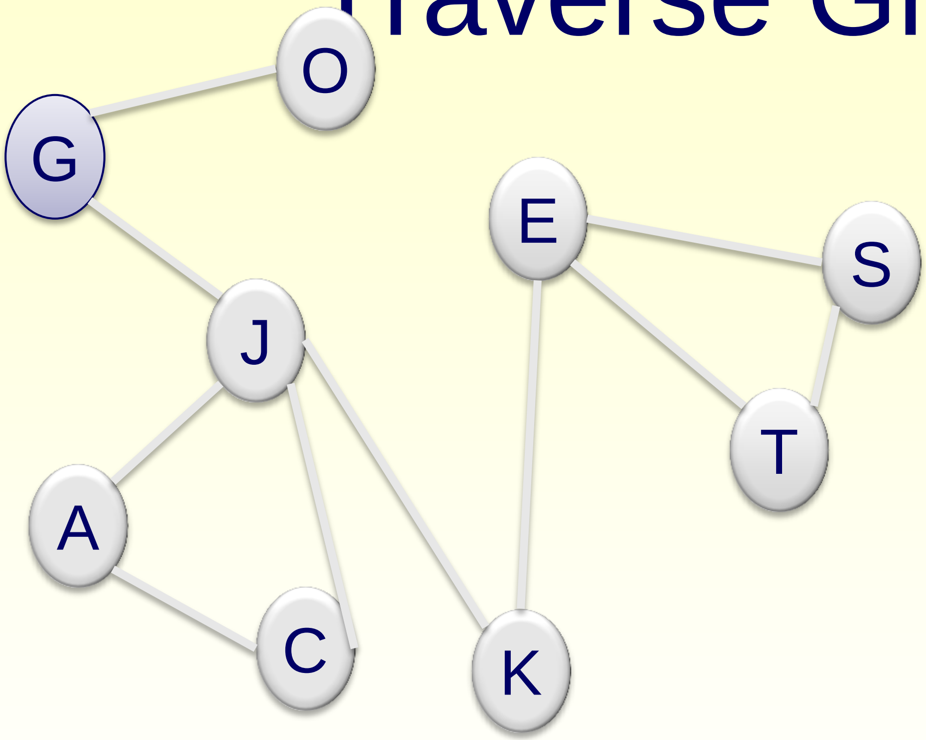
stack

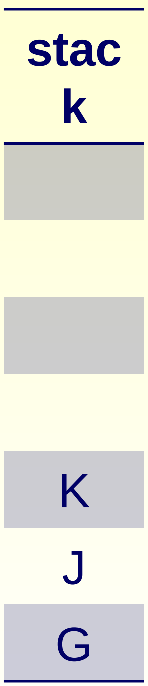
J

G

Output: G O J A C

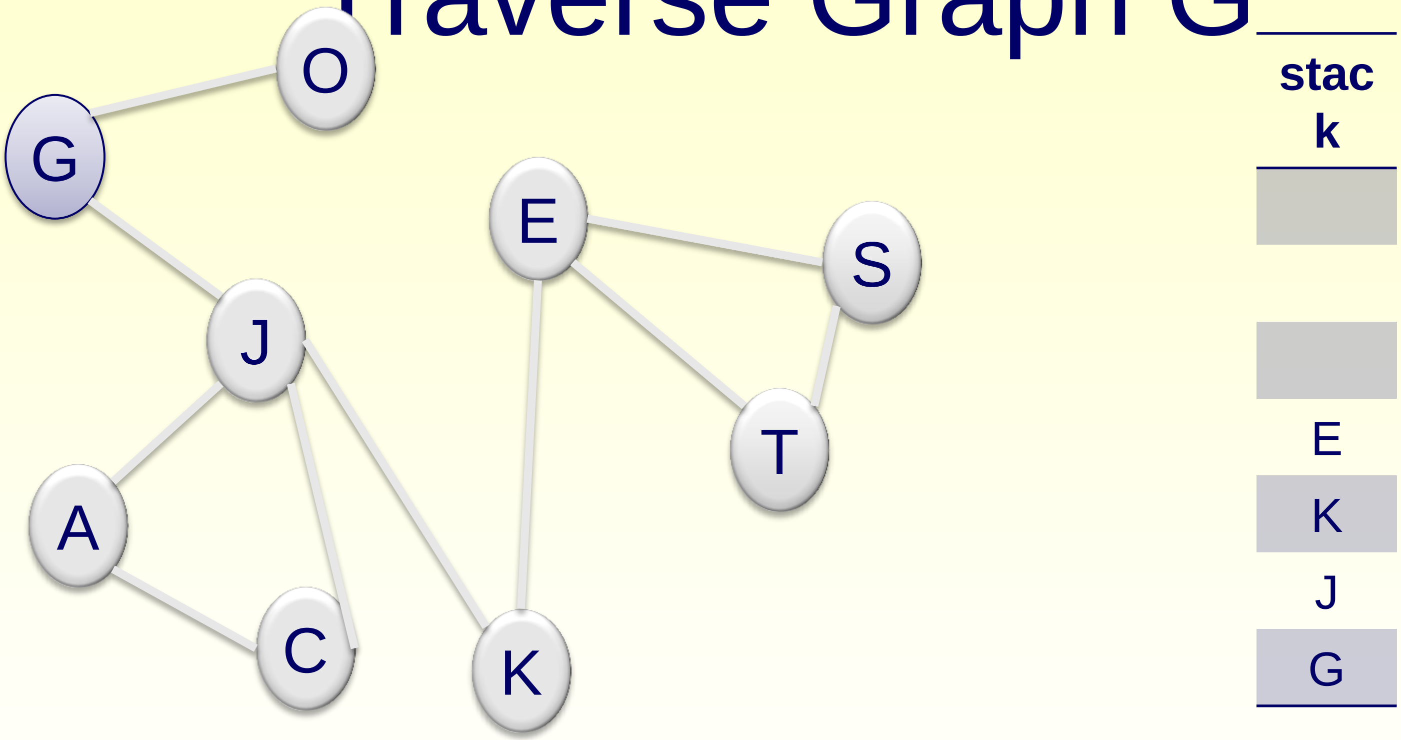
DFS Example
Traverse Graph G





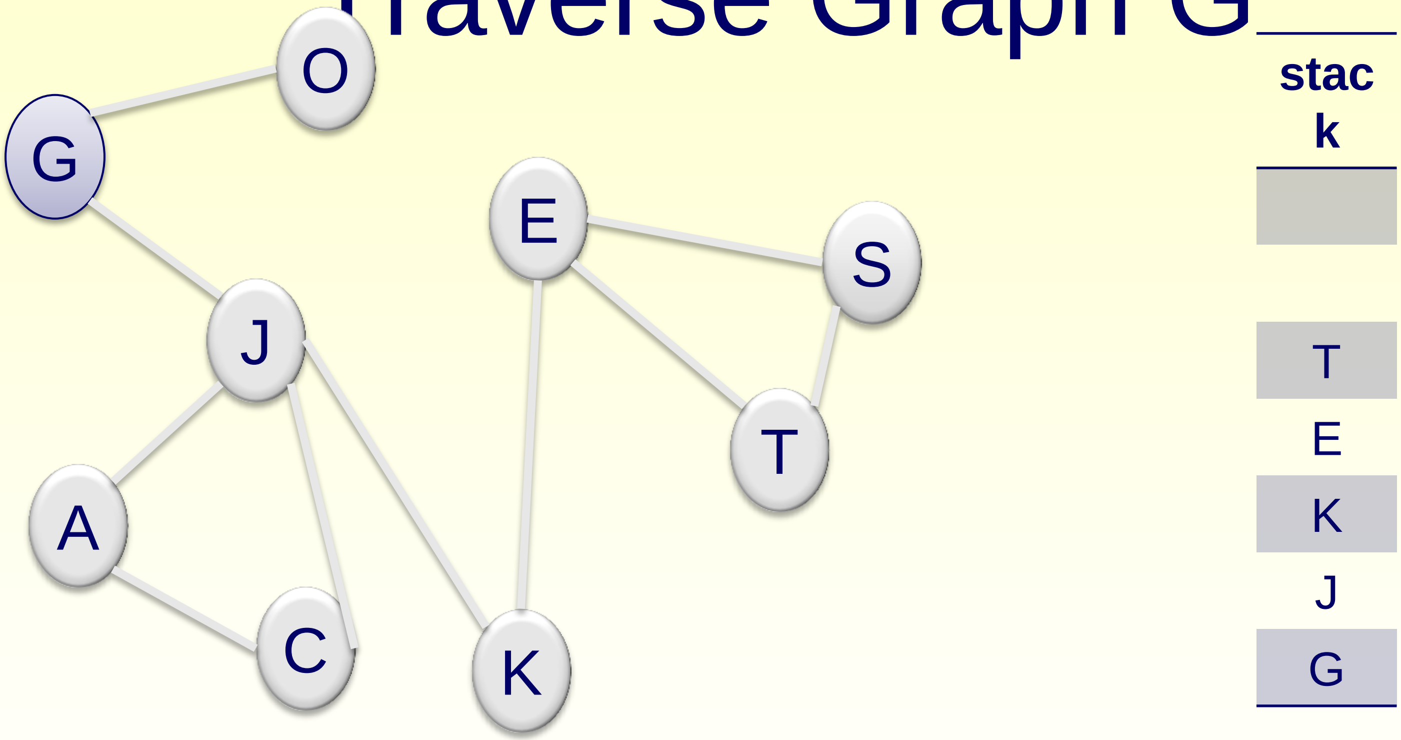
Output: G O J A C K

DFS Example
Traverse Graph G



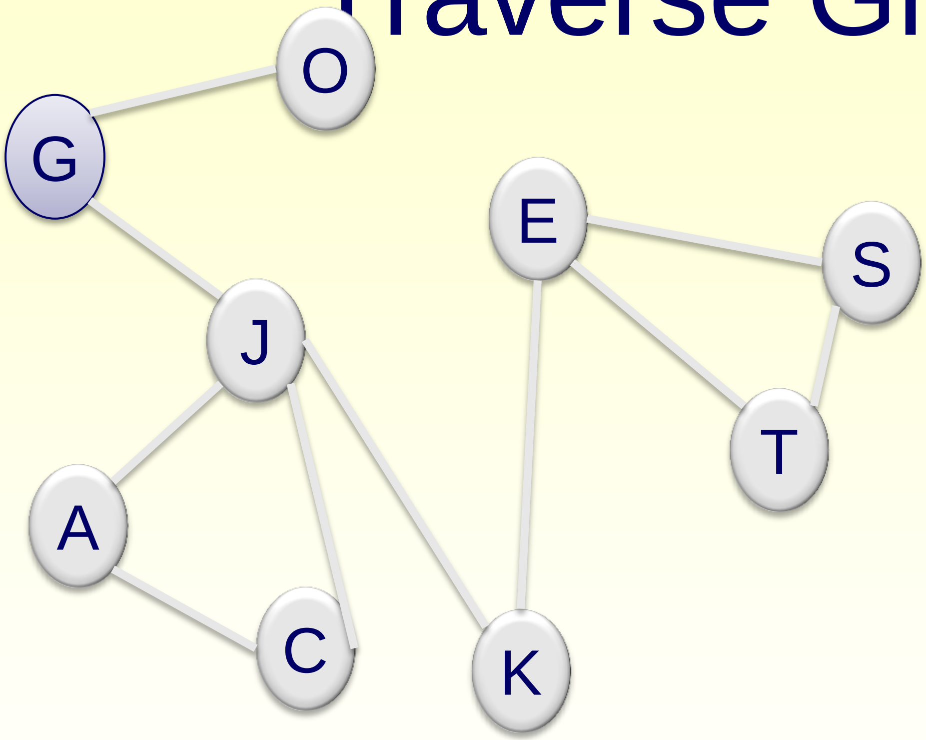
Output: G O J A C K E

DFS Example
Traverse Graph G



Output: G O J A C K E T

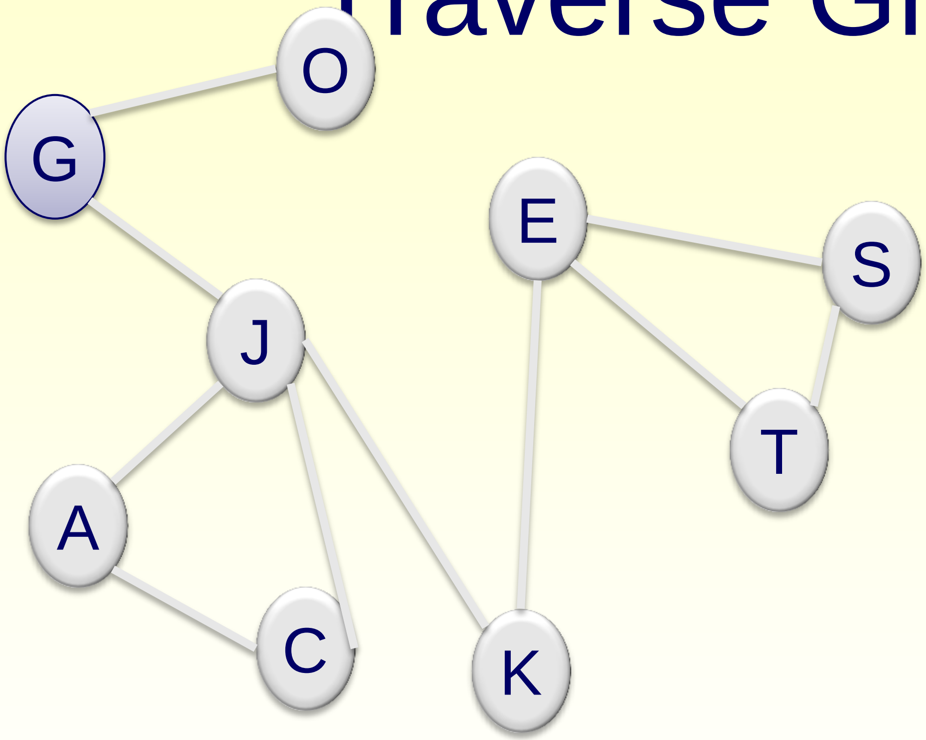
DFS Example
Traverse Graph G



stack
S
T
E
K
J
G

Output: G O J A C K E T S

DFS Example
Traverse Graph G

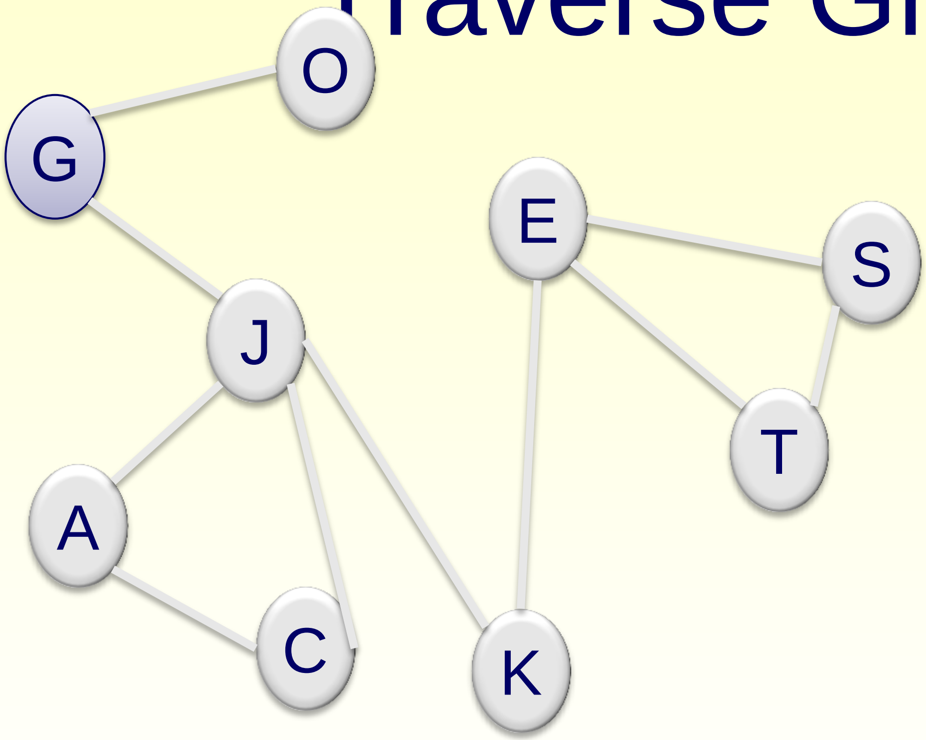


stack

T
E
K
J
G

Output: G O J A C K
E T S

DFS Example
Traverse Graph G

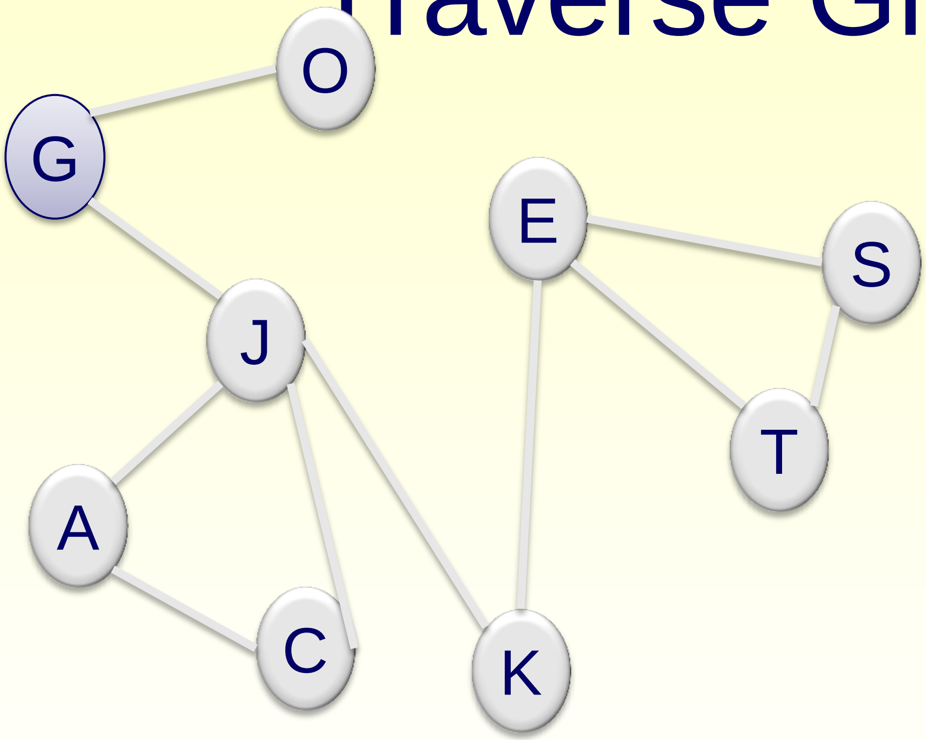


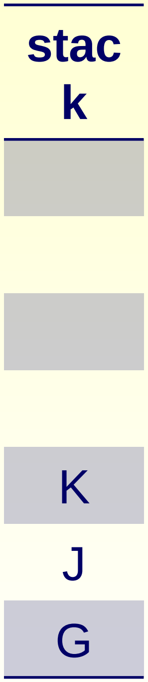
stack

E
K
J
G

Output: G O J A C K
E T S

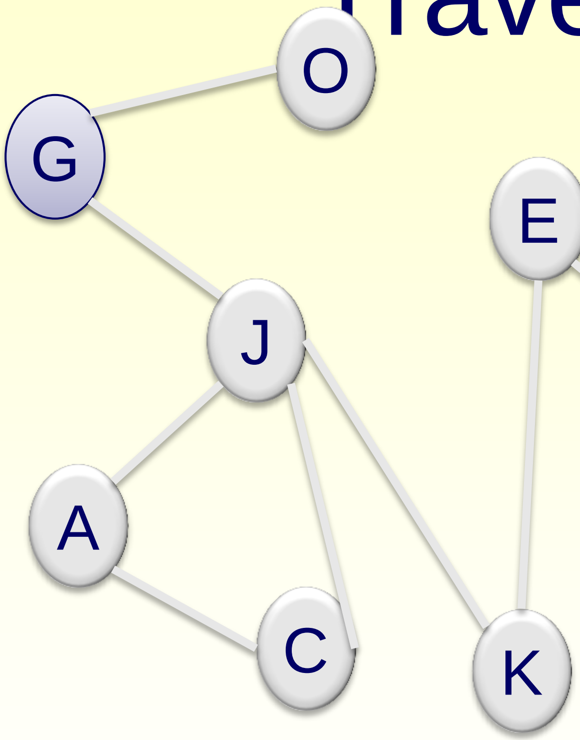
DFS Example
Traverse Graph G

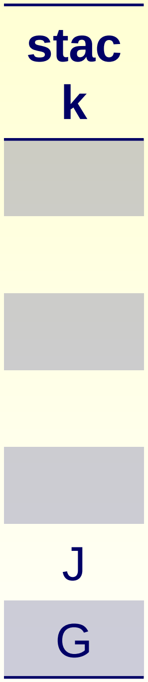




Output: G O J A C K E T S

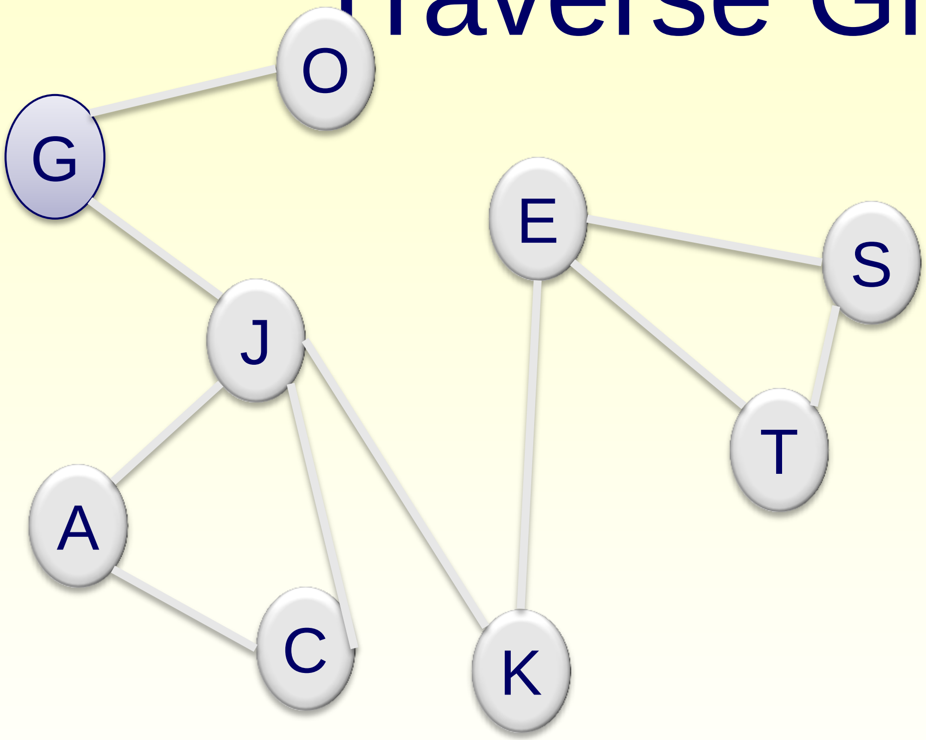
DFS Example
Traverse Graph G

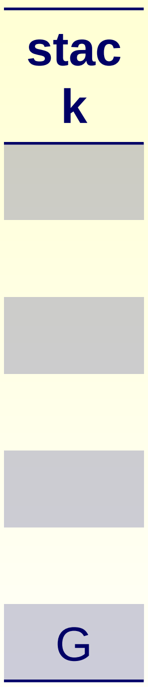




Output: G O J A C K
E T S

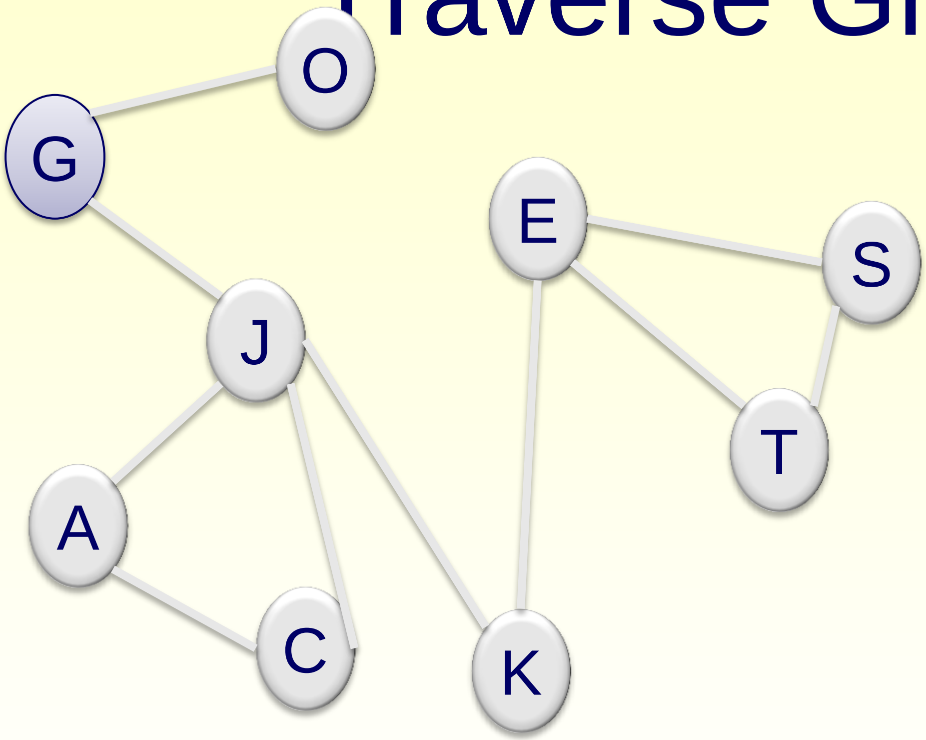
DFS Example
Traverse Graph G

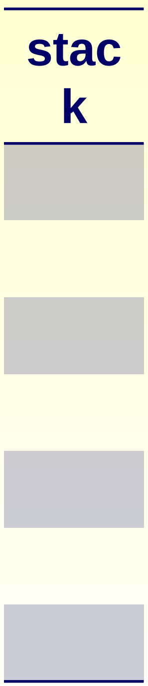




Output: G O J A C K
E T S

DFS Example
Traverse Graph G





Output: G O J A C K
E T S

Dijkstra’s Shortest Path

Problem Find the shortest path problem for weighted directional graphs:

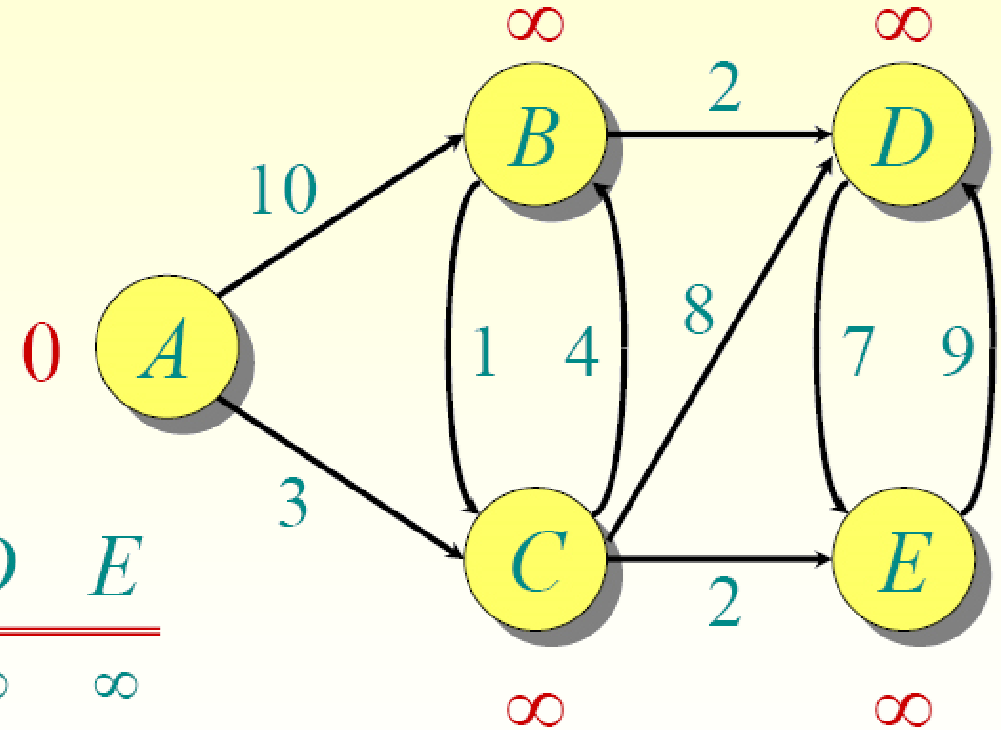
- Start from one vertex, choose stop vertex
- Move forward all along shortest path (do not pass through a vertex already visited) using BFS
- Keep track of the distance of the path as compared to other paths.
- If you do not reach stop vertex, the path is disregarded

Dijkstra's Pseudo Code

```
Dijk(g)
  initialize distance to source vertex, s, to zero
  for  all v ∈ V–{s}                    set all other vertices' distances to
infinity
     do  dist[v] ←∞
  create empty set, S, to hold the visited
  populate the queue Q initially with all vertices
  while Q is not empty
     do   select an element of Q with the min. distance), u
        add u to list of visited vertices)
        for all v ∈ neighbors[u]
           do  if   dist[v] > dist[u] + w(u, v)   (if new shortest path found)
           then      d[v] ←d[u] + w(u, v)       (set new value of shortest
path)
  return dist
```

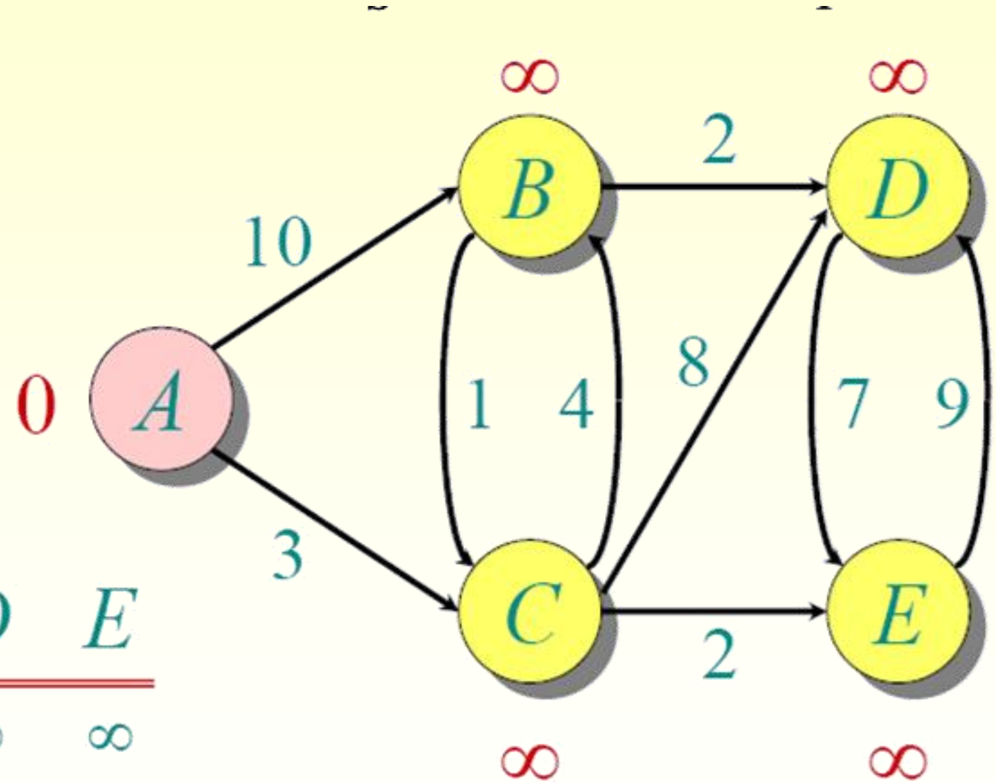
Dijkstra Animated Example

Initialize:

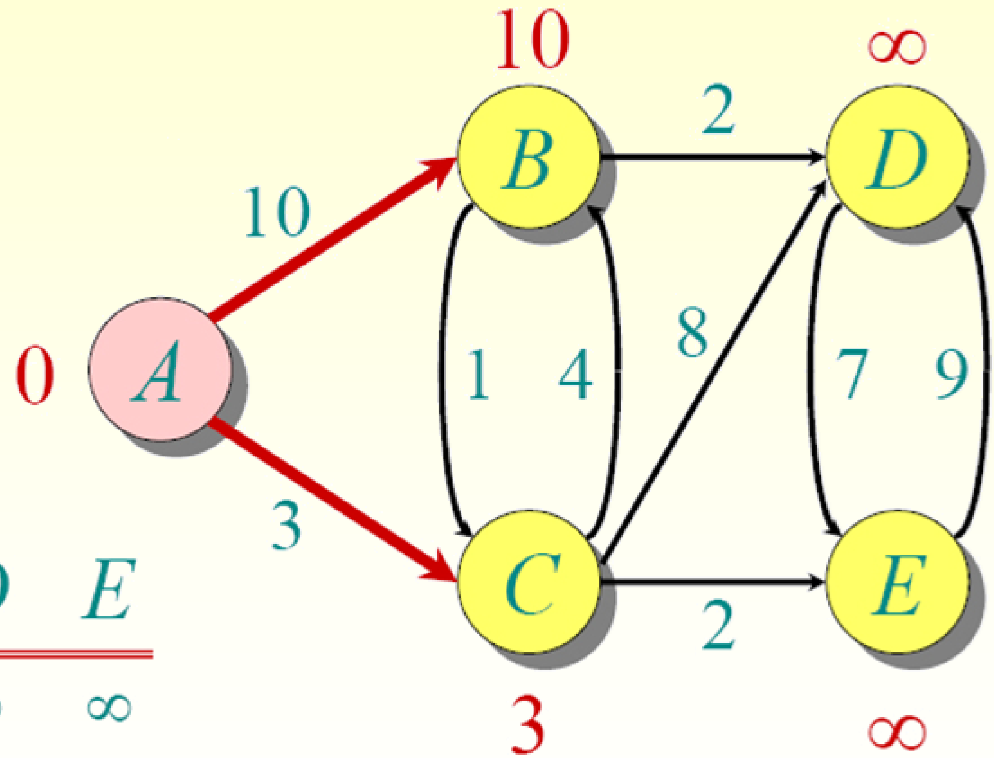


Q:	A	B	C	D	E
	0	∞	∞	∞	∞

S: {}

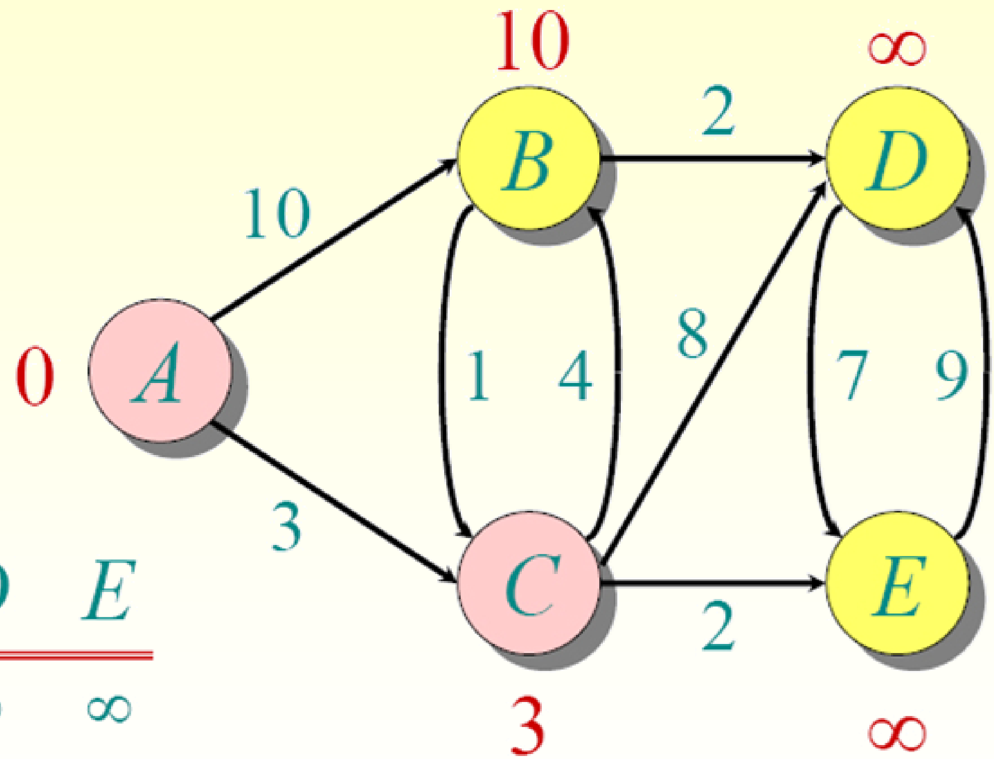


Q:	A	B	C	D	E
	0	∞	∞	∞	∞



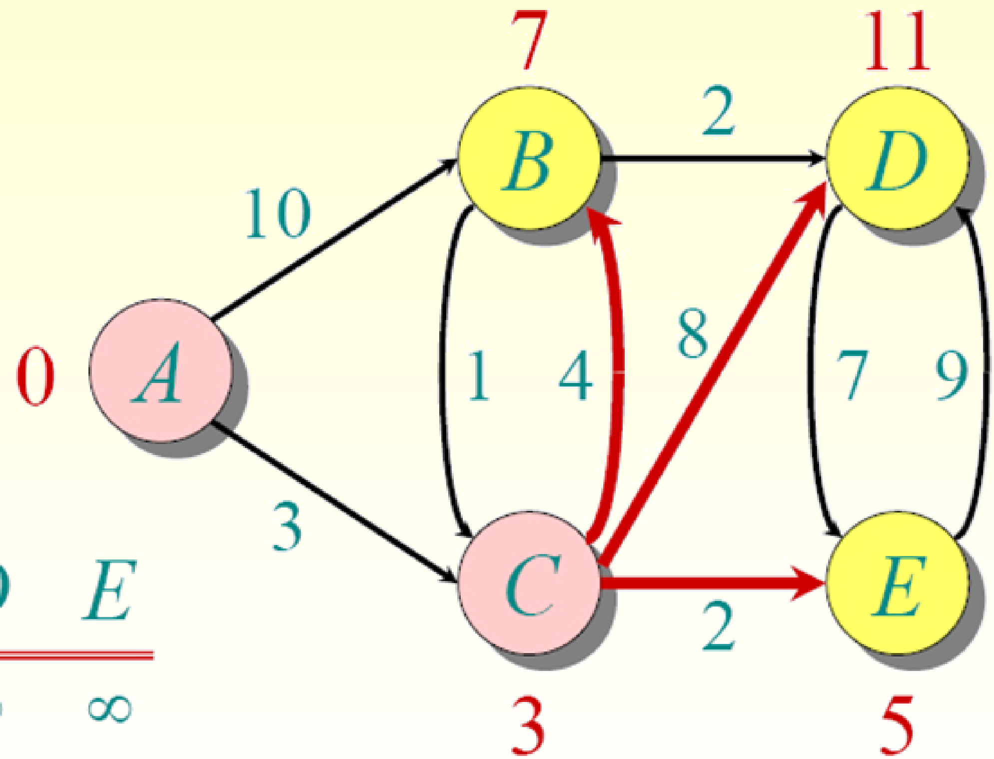
Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞

S: { A }



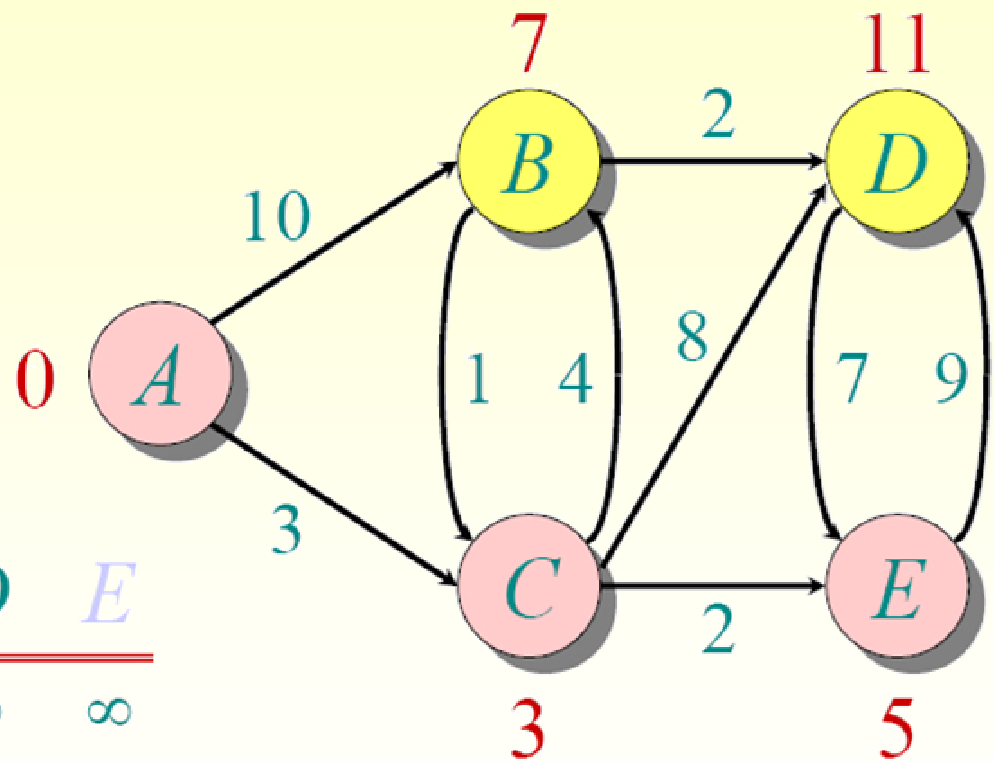
Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞

S: { A, C }



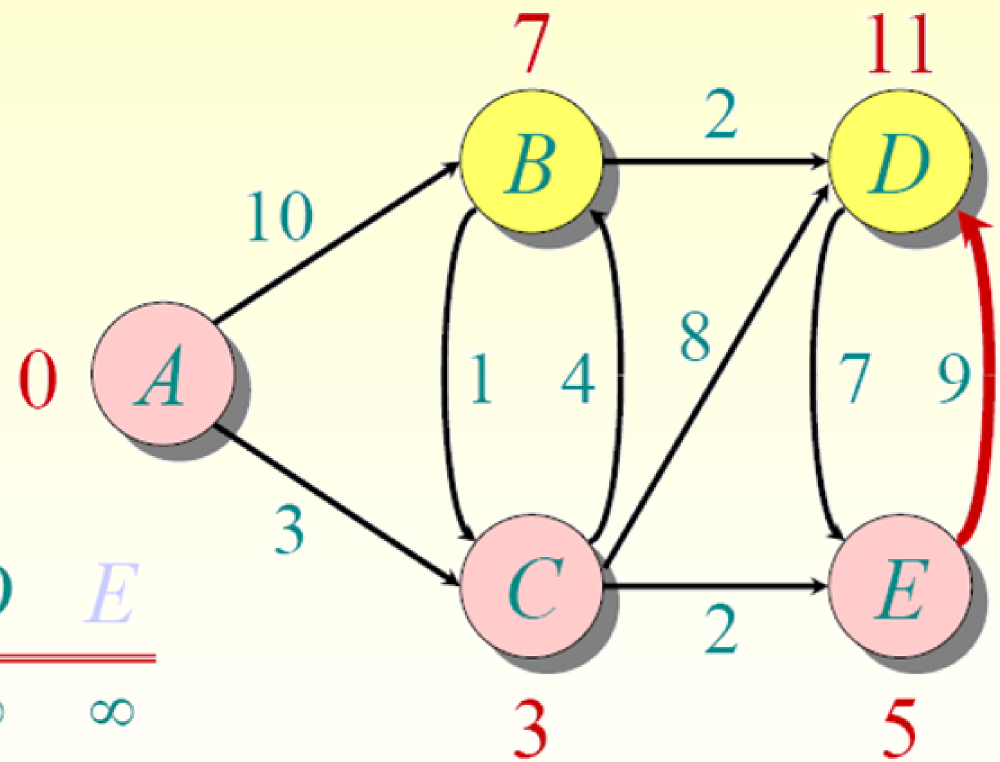
Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5

S: { *A*, *C* }



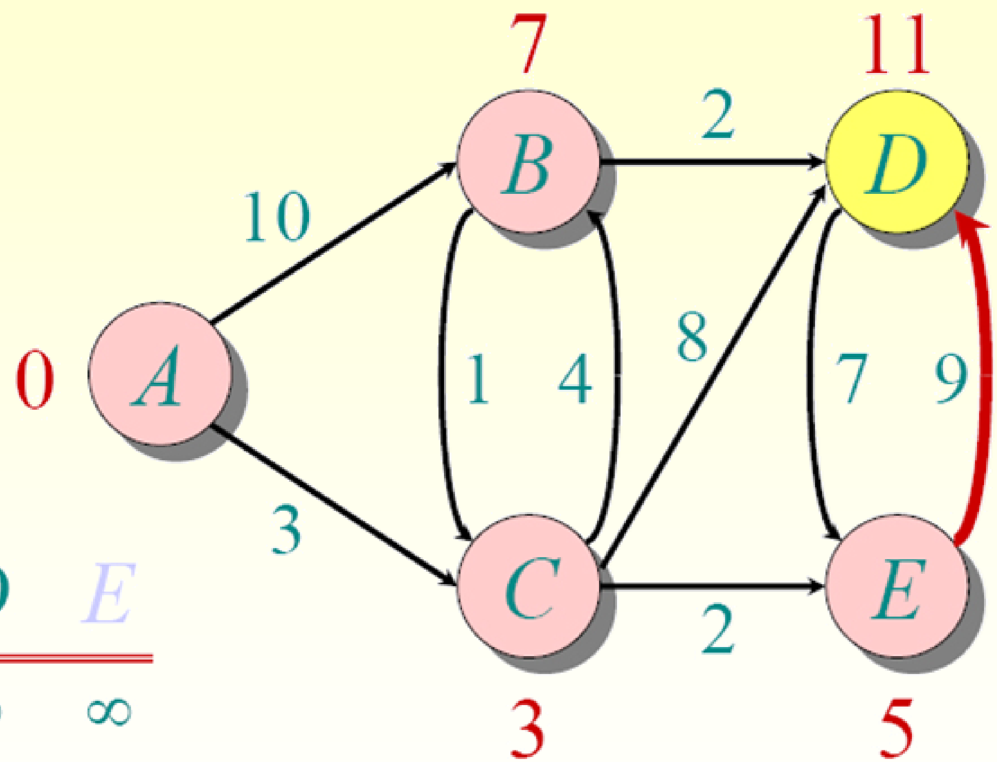
Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5

S: { A, C, E }



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	

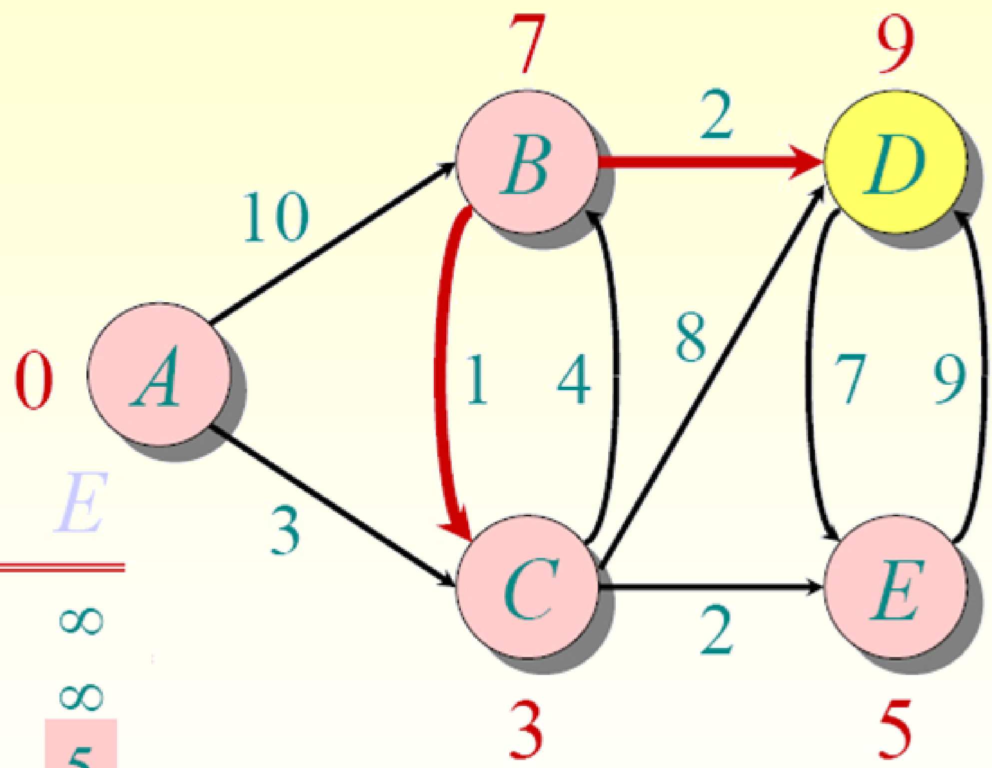
S: { A, C, E }



Q:	*A*	*B*	*C*	*D*	*E*
	$\boxed{0}$	∞	∞	∞	∞
		10	$\boxed{3}$	∞	∞
		7		11	$\boxed{5}$
		$\boxed{7}$		11	

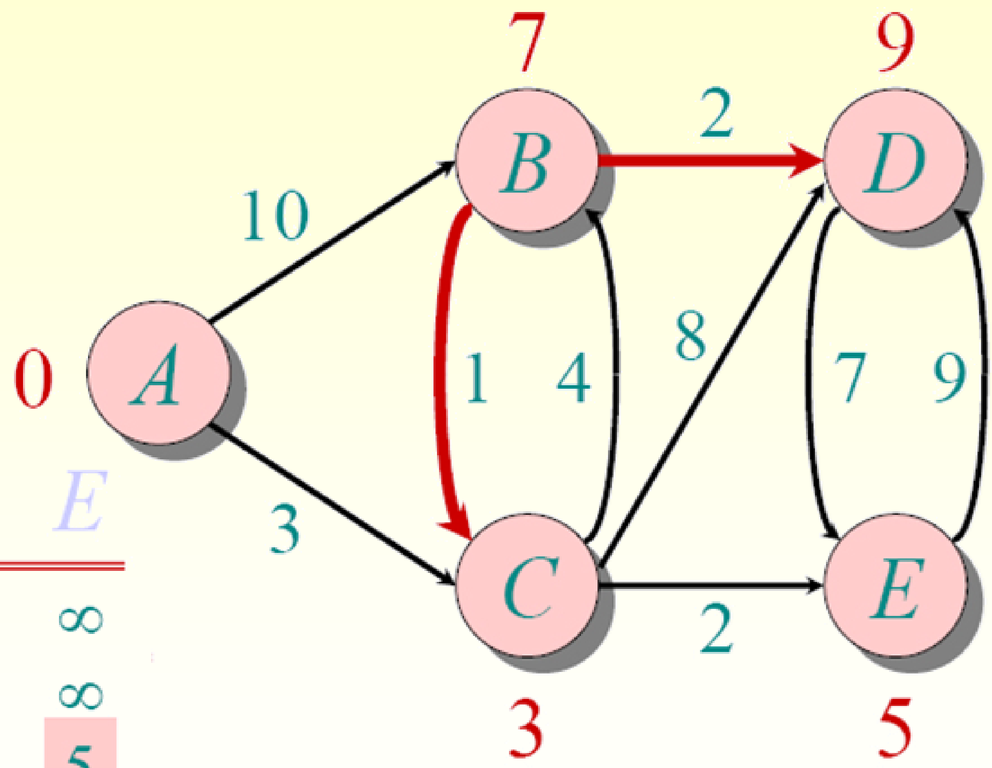
S: { *A*, *C*, *E*, *B* }

Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	
				9	



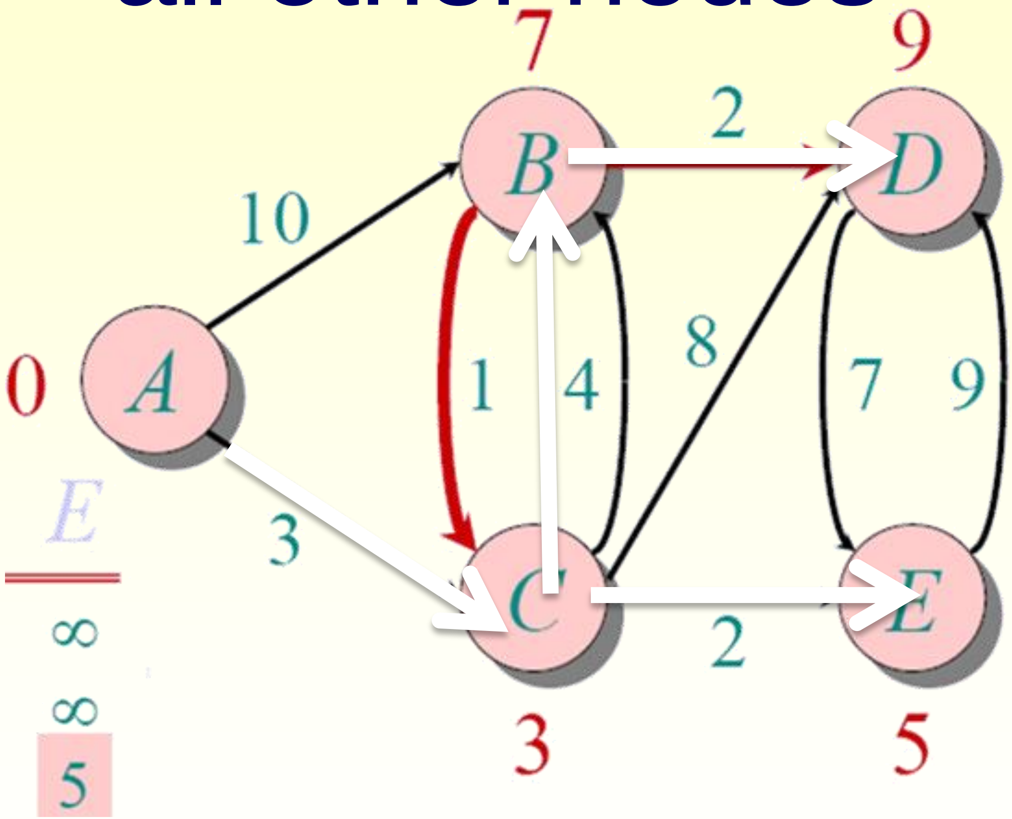
S: { A, C, E, B }

Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	
				9	



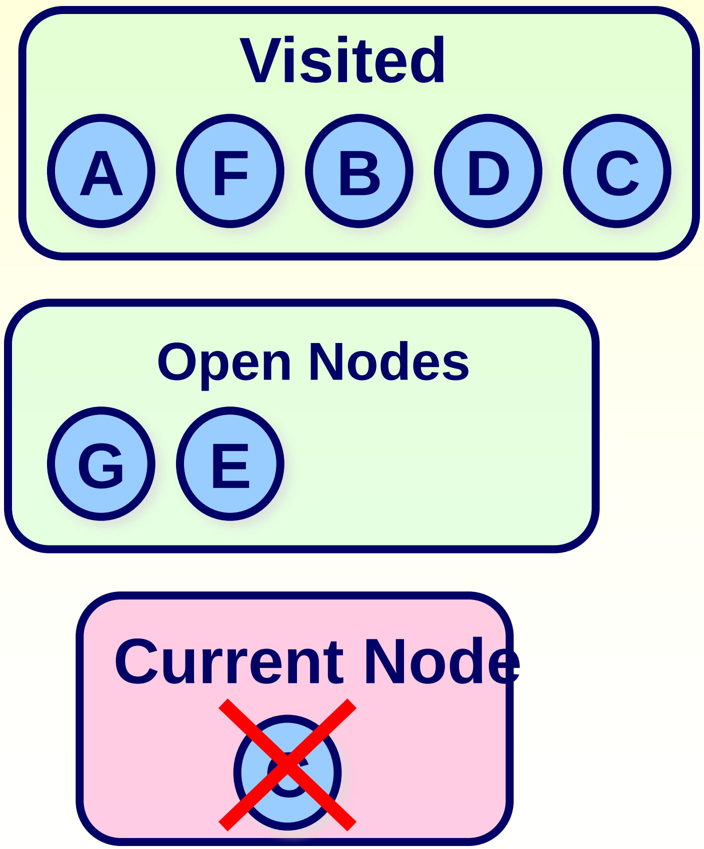
S: { A, C, E, B, D }

Shortest Path Tree from A to all other nodes



Searching Observations

We could code a method in Java. It's easy to see that some of our structures correspond to Java classes.




```
Vector visited =
           new Vector();
```

```
Vector openQueue =
           new Vector();
```

```
NodeType current;
```

However . . .

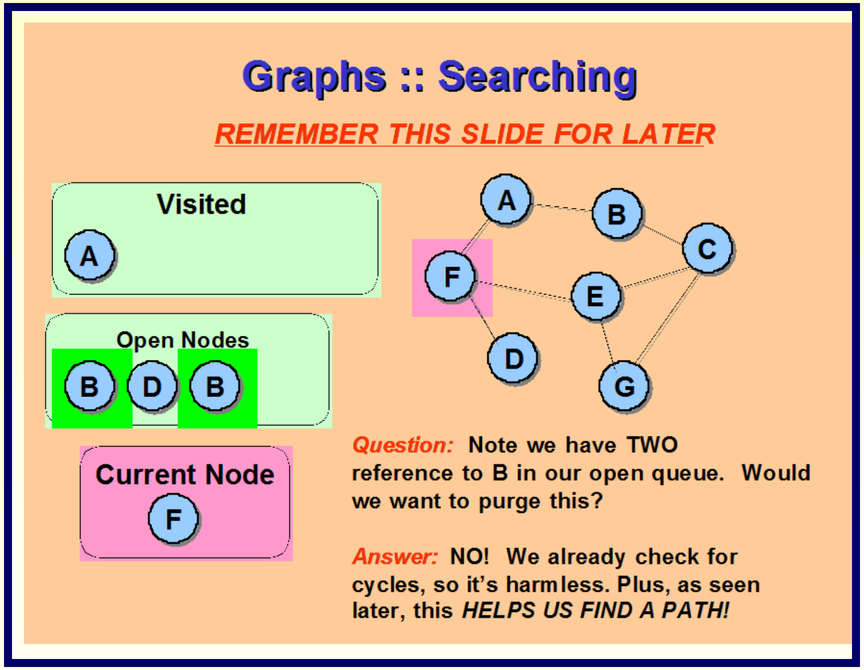
Our search only determined if there WAS a path from a node to a goal, not WHAT that path was.

There’s a way to save HOW to get to a node, in addition to determining WHETHER two nodes have a path.

Searching for a Path

Recall the point in our BFS where we found two identical references in our open node queue.

In fact, the references could be considered different, because they were contributed to the open queue by different nodes.



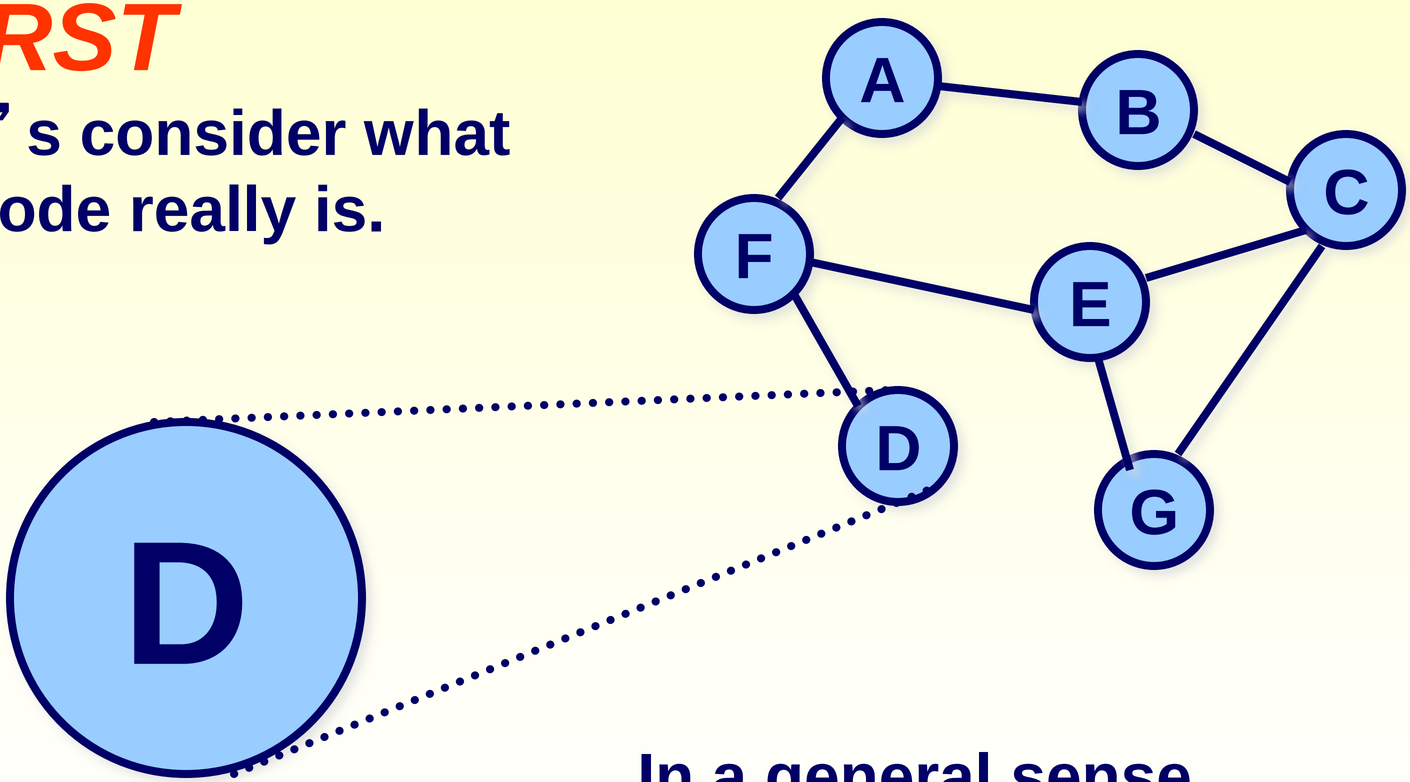
But we didn't save WHICH node contributed the reference to the open queue.

We need to keep track of where the nodes came from.

Thinking INSIDE the Box

FIRST

Let’s consider what a Node really is.



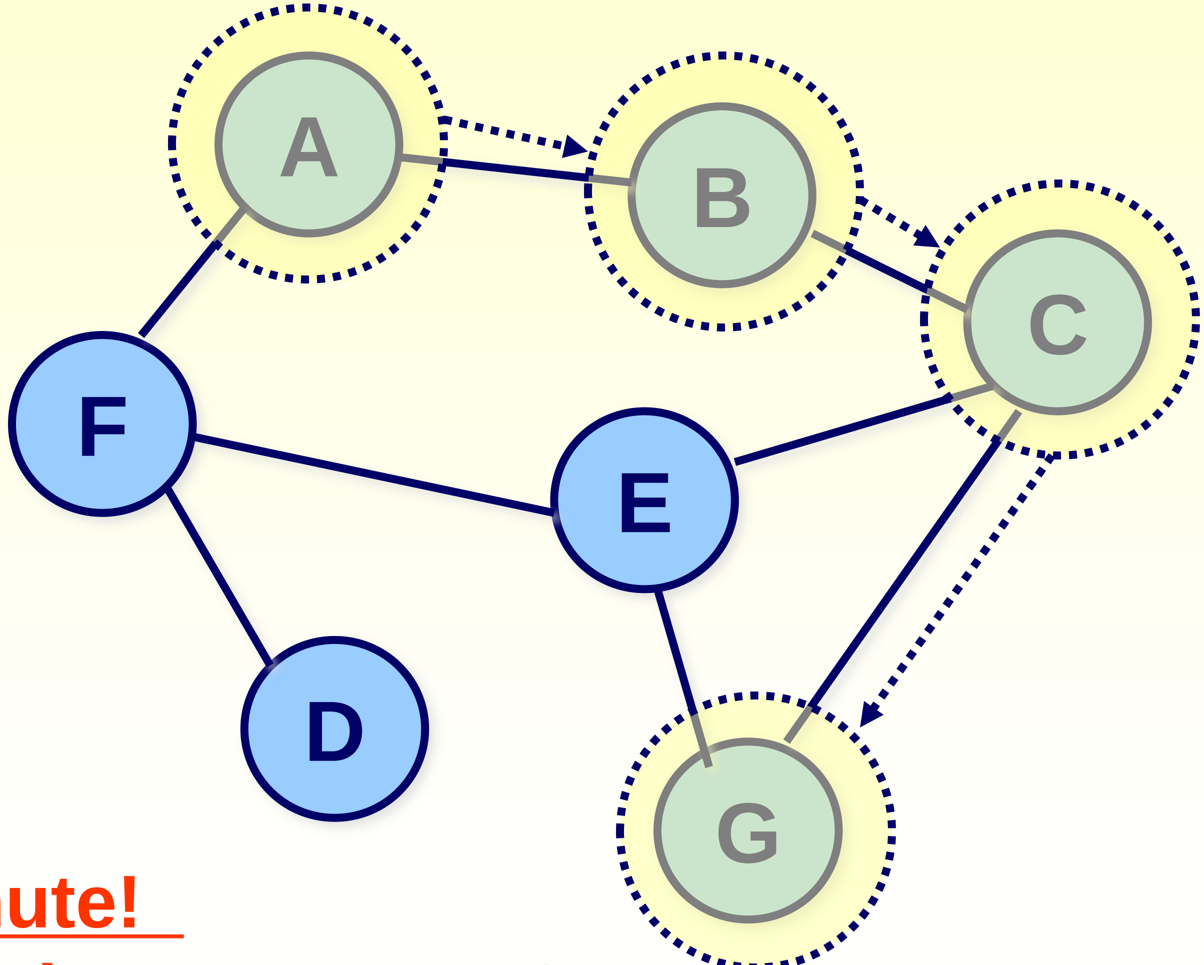
In a general sense,
a node is just a box.
We can place data in it,
like an Object or a Comparable.

Thinking INSIDE the Box

SECOND,

Let's consider what a "Path" really is.

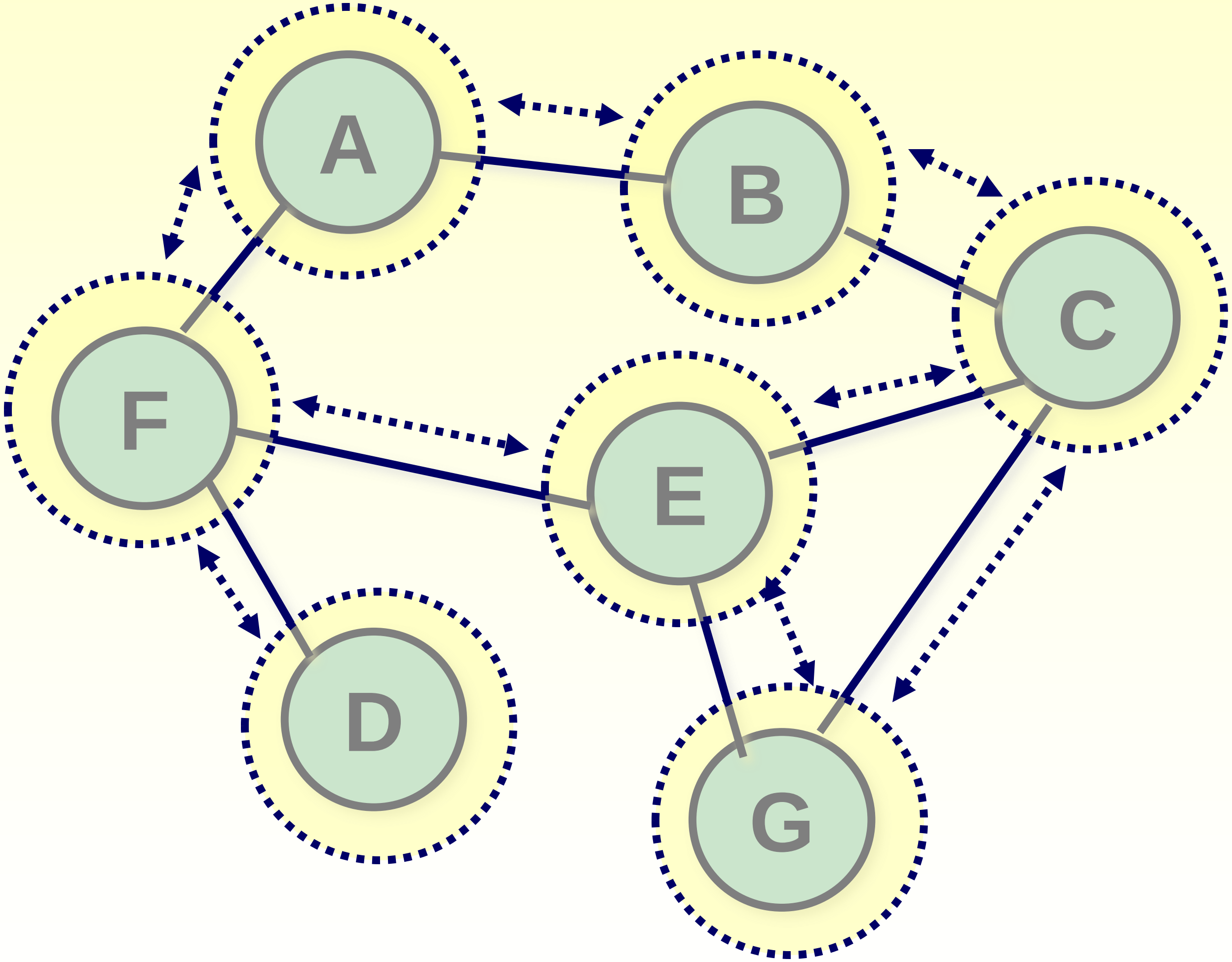
A path is a collection of nodes.



Hey, wait a minute!

A PATH IS (LIKE) A LINKED LIST!

Wow – things are coming together.

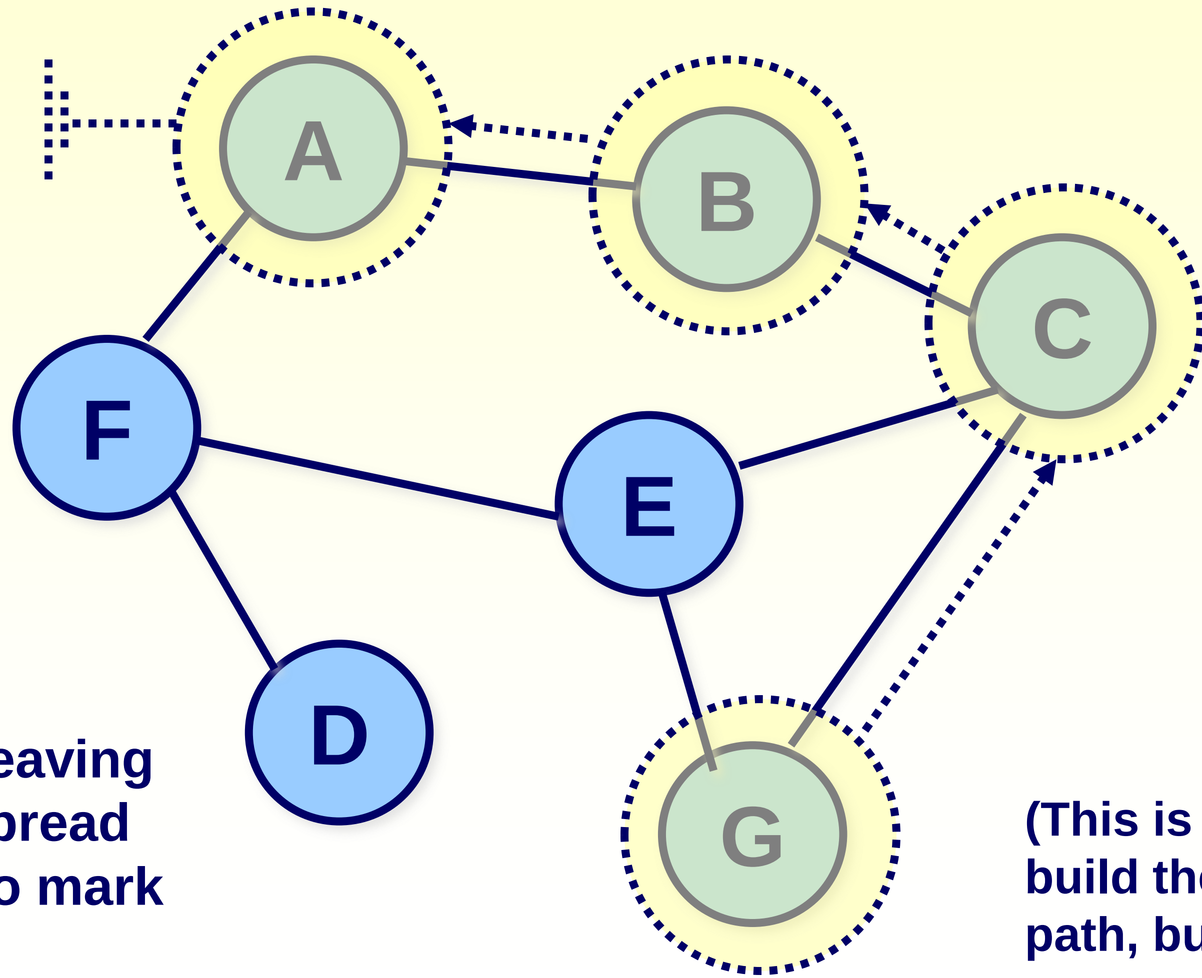


In a sense, we can think of a graph as a bunch of linked lists. Finding a path is just a matter of finding the right linked list that walks the graph.

Keep thinking:
"There is no Big O"
"There is no Big O"

So?

To form a path from start node to goal node, we must therefore make a "linked list" to record where we came from.

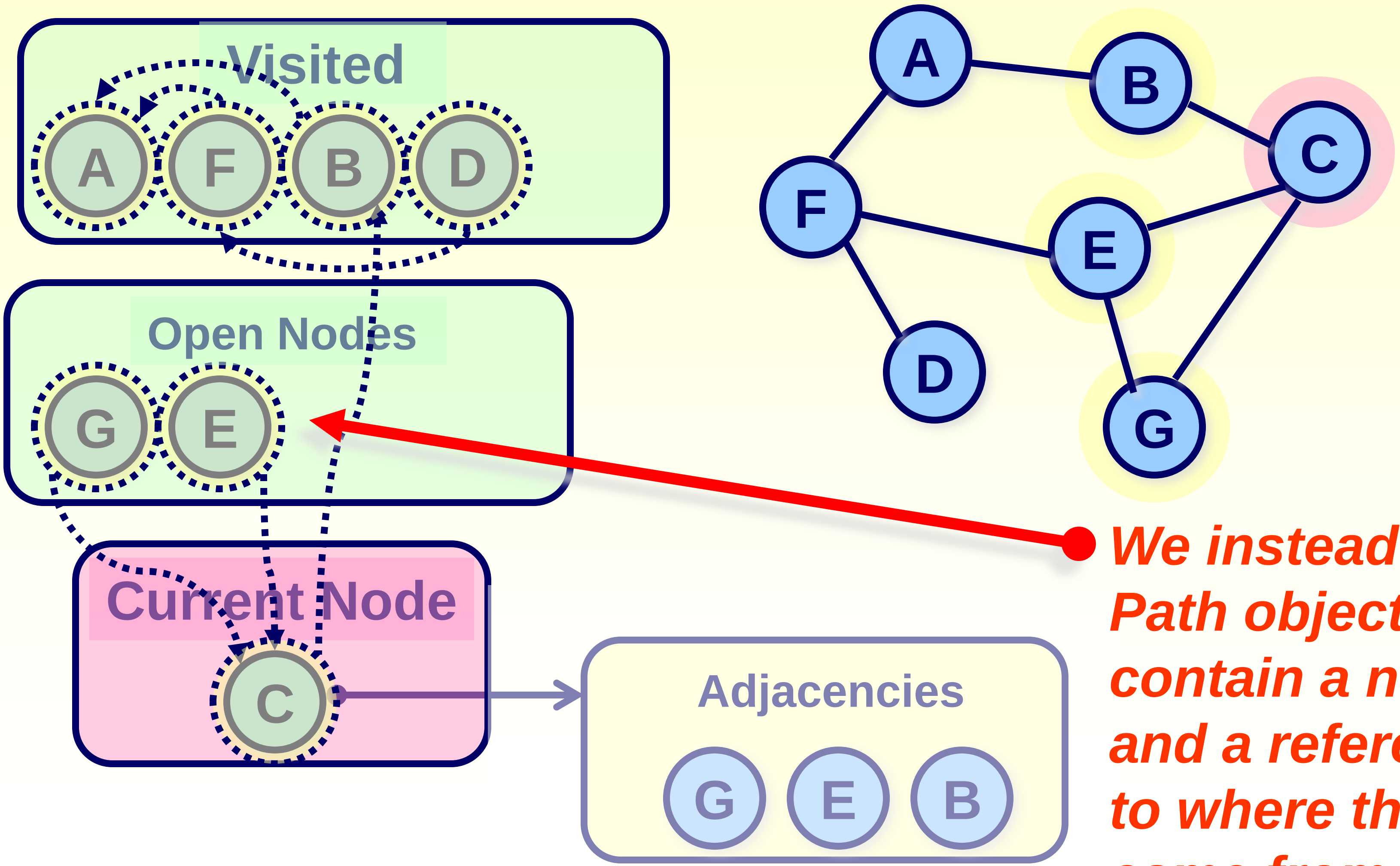


It's like leaving a trail of bread crumbs to mark our trail

(This is actually going to build the *backwards* path, but we can fix this.)

Thinking INSIDE the Box

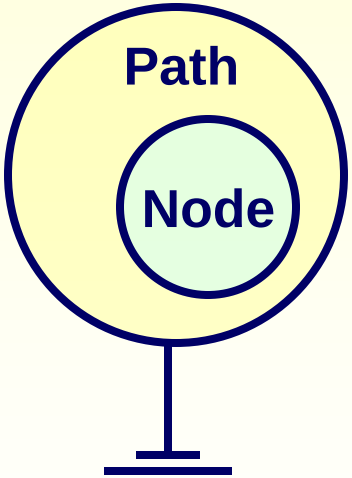
Thus, we don't add graph nodes to our open queue...



We instead add Path objects, that contain a node, and a reference to where they came from

```
public class Path {

    private Path previous;
    private Node node;

    public Path (Path previous, Node node) {
        setNode(node);
        setPrevious(previous);
    }

    public Node getNode(){return node;}
    public void setNode (Node n){
        this.node = n;
    }
    public Path getPrevious(){
        return previous;
    }
    public void setPrevious(Path p){
        this.previous = p;
    }

} // class Path
```

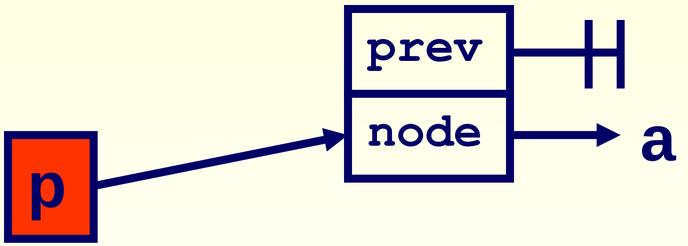
The term "Path" is misleading
It's actually a step or link node in the overall path.



This Path thing can be a little startling at first

- Imagine we have Nodes: a, b, c, d, e
- Here is the typical sequence as we traverse down these nodes in order

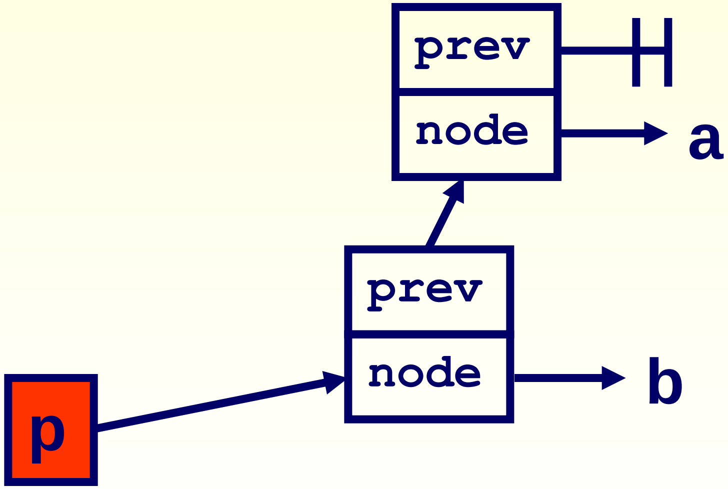
```
Path p = new Path(null, a);
```



This Path thing can be a little startling at first

- Imagine we have Nodes: a, b, c, d, e
- Here is the typical sequence as we traverse down these nodes in order

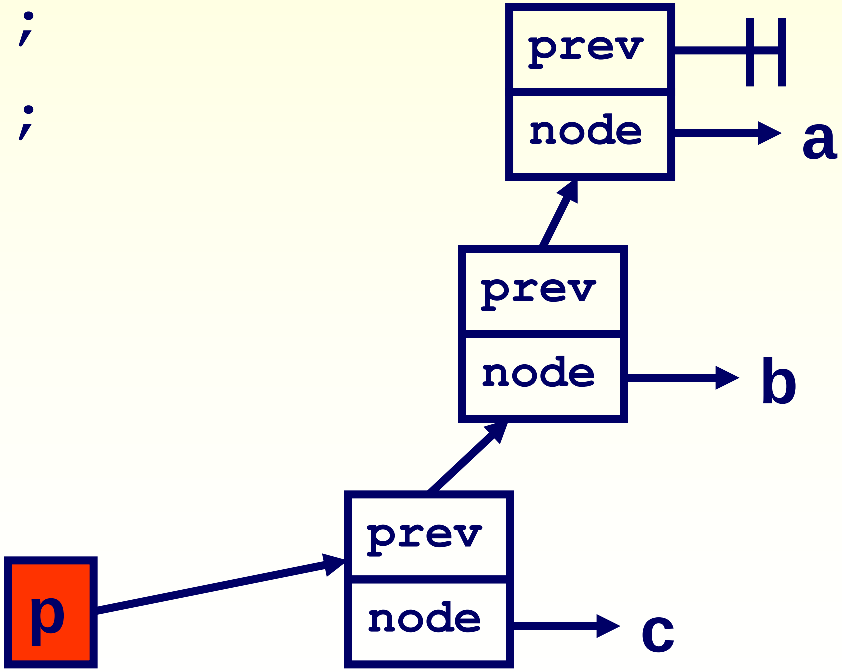
```
Path p = new Path(null, a);
p = new Path(p, b);
```



This Path thing can be a little startling at first

- Imagine we have Nodes: a, b, c, d, e
- Here is the typical sequence as we traverse down these nodes in order

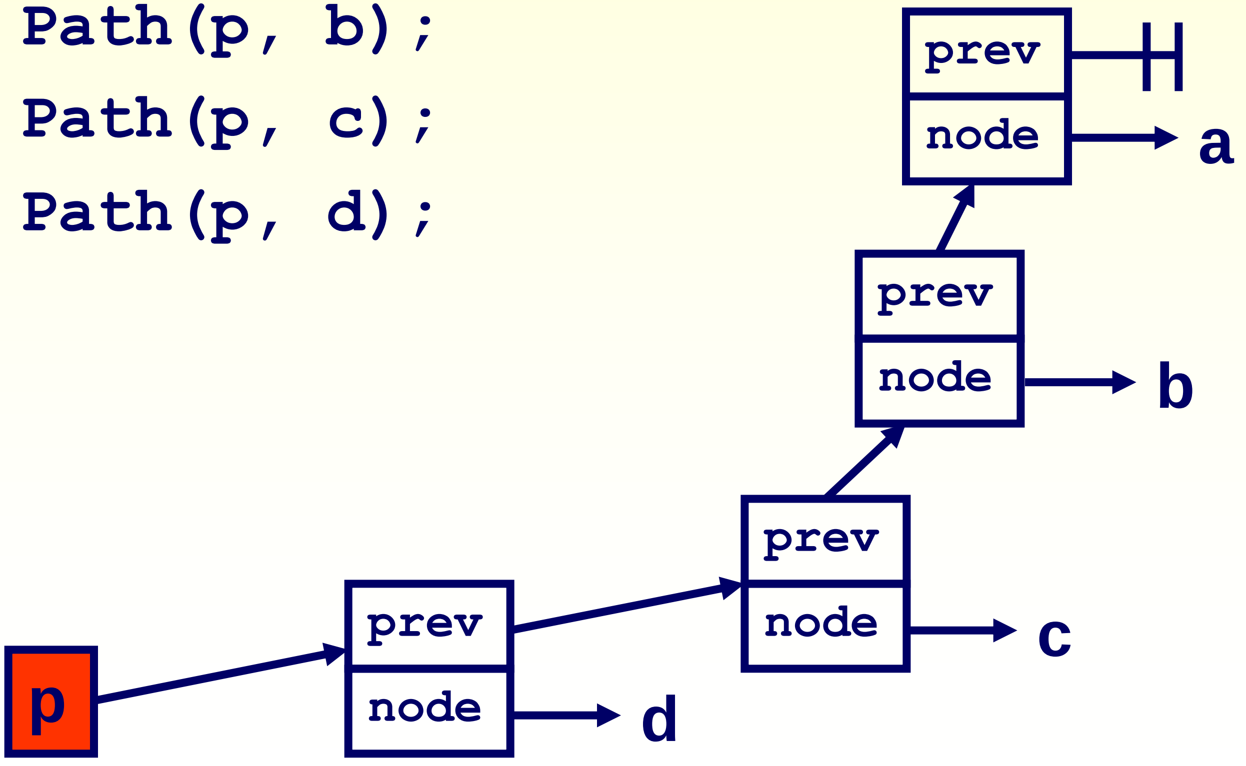
```
Path p = new Path(null, a);
p = new Path(p, b);
p = new Path(p, c);
```



This Path thing can be a little startling at first

- Imagine we have Nodes: a, b, c, d, e
- Here is the typical sequence as we traverse down these nodes in order

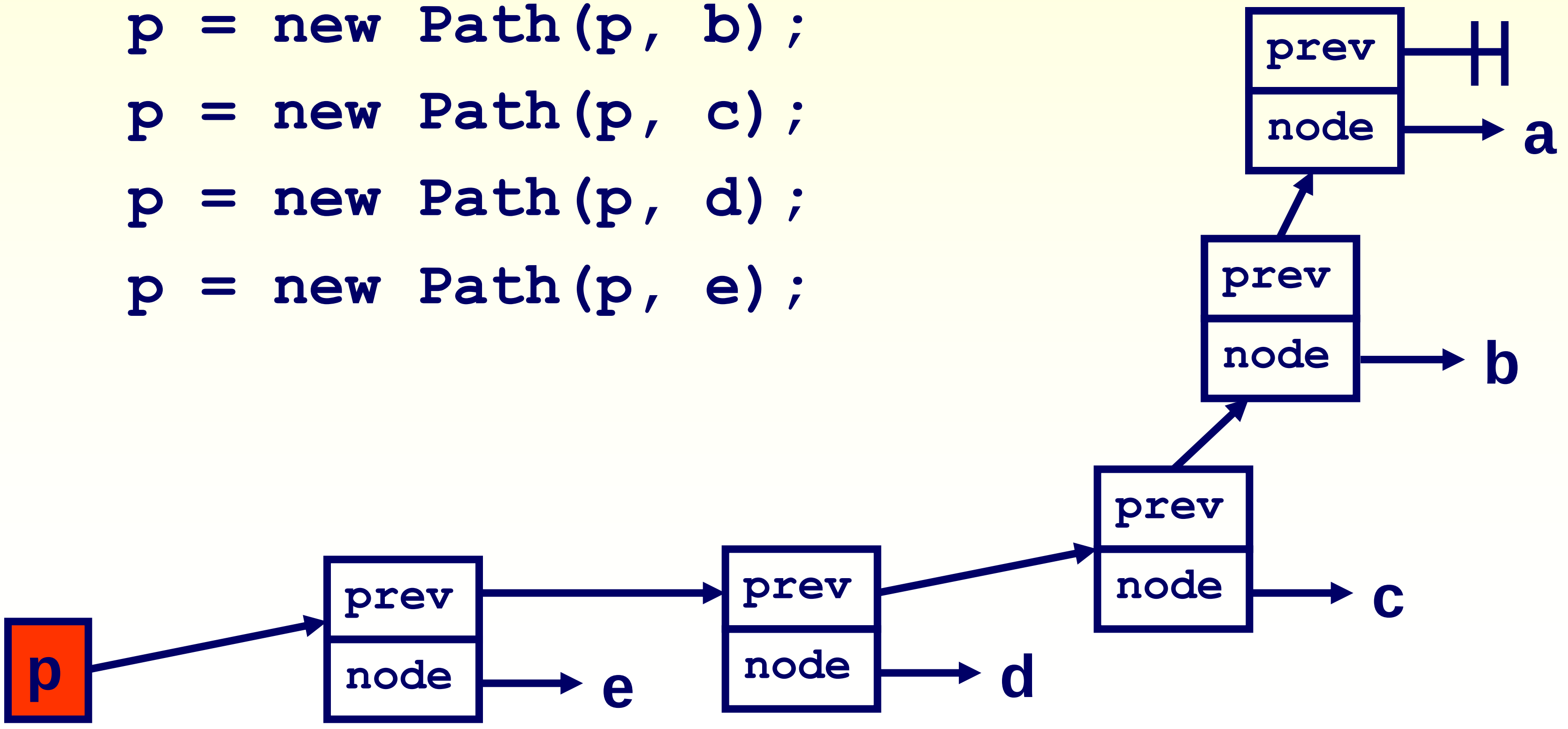
```
Path p = new Path(null, a);
p = new Path(p, b);
p = new Path(p, c);
p = new Path(p, d);
```



This Path thing can be a little startling at first

- Imagine we have Nodes: a, b, c, d, e
- Here is the typical sequence as we traverse down these nodes in order

```
Path p = new Path(null, a);
p = new Path(p, b);
p = new Path(p, c);
p = new Path(p, d);
p = new Path(p, e);
```

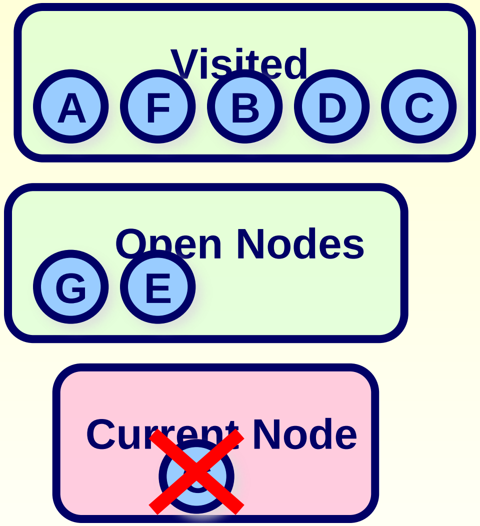


```
public Path findPathWithBFS( Node start, Node goal ){
    Vector vVisited = new Vector();
    Vector vOpen = new Vector();
    Path curr = new Path(null, start);
    Node n = null;
    vOpen.addElement(curr);
    while (vOpen.size() != 0) {
        curr = (Path) vOpen.elementAt(0);
        vOpen.removeElementAt(0);
        vVisited.addElement(curr);
        n = curr.getNode();
        if (n.equals(goal))
            return curr;  // Found path
        Vector vNext = getAdjacencies(n);
        for (int i =0; i < vNext.size(); i++)
        if (! vVisited.contains( vNext.elementAt(i)))
             vOpen.addElement
                (new Path (curr, vNext.elementAt(i)));
     }
     return null; // NO path found!
}
```

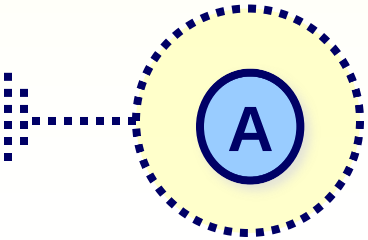
Note: This only returns the reversed path.

Analysis

```
public Path findPathWithBFS( Node start, Node goal ){
    Vector vVisited = new Vector();
    Vector vOpen = new Vector();
    Path curr = new Path(null, start);
    Node n = null;
    vOpen.addElement(curr);
```



These declare variables and structures, with one notable change. We don't use a Node as our "current" reference, but instead a Path that holds a node.



Our first Path has "null" as its previous, since it holds the start node.

In the loop!

Analysis

We start by removing the first Path from the adjacency list.

```
curr = (Path) vOpen.elementAt(0);
vOpen.removeElementAt(0);
vVisited.addElement(curr);
```

If the Path object contains the goal node, we’re done

```
n = curr.getNode();
if (n.equals(goal))
   return curr;  // Found path
```

Analysis

If the current Path is not the goal, we check each node adjacent to the current. If we've not visited it before, we add it to our open queue.

```
Vector vNext = getAdjacencies(n);
for (int i =0; i < vNext.size(); i++)
    if (! vVisited.contains( vNext.elementAt(i)))
        vOpen.addElement
          (new Path (curr, vNext.elementAt(i)));
```

If the 'while' loop exhausts the open queue, we have not found a Path to the goal. So, return null.

```
}// while

return null; // NO path found!
```

Study Guide

- Know the basic parts of a graph
- Trace the code for finding a path
  - BFS: Queue
  - DFS: Stack
- Understand why the different data structures yield different results

Summary – you should now …

- Graphs
- Searching
- Breadth-First Search (BFS)
- Depth-First Search (DFS)