GRAPHS

DIRECTED GRAPHS

Ruben Acuña

Spring 2019

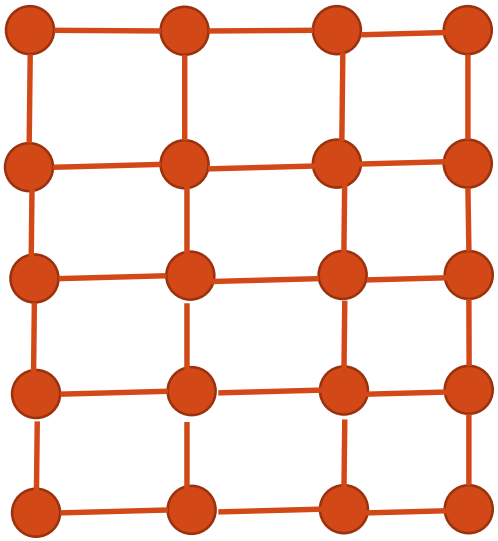
INTRODUCTION

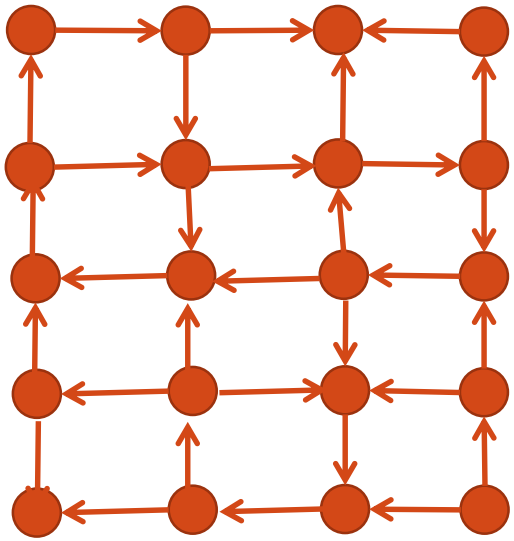
THE DIRECTED GRAPH CONCEPT

- So, what's the basic idea?
  - Mark edges to indicate their direction.

WHAT'S NEW?

- Our intuition. Which of these two graphs is connected?





TERMINOLOGY

- V is a Direct Predecessor of W if: there is a v to w edge.
- V is a Direct Successor of W if: there is a w to v edge.
- V is a Predecessor of W if: v is a direct predecessor of w or a predecessor of a direct predecessor of w.
- V is a Successor of W if: v is a direct successor of w or a successor of a direct successor of w.

DIRECTED GRAPH ADT

- What's changed since the undirected ADT?
- Will adjacency matrices and adjacency lists be sufficient to store the data?

public class	Digraph	
	Digraph(int V)	*create a V-vertex digraph with no edges*
	Digraph(In in)	*read a digraph from input stream* in
int	V()	*number of vertices*
int	E()	*number of edges*
void	addEdge(int v, int w)	*add edge v->w to this digraph*
Iterable<Integer>	adj(int v)	*vertices connected to* v *by edges pointing from* v
Digraph	reverse()	*reverse of this digraph*
String	toString()	*string representation*

CODE

Nothing surprising…

Again: what's changed?

```
private final int V;
    private int E;
    private LinkedList<Integer>[] adj;

    public Digraph(int V) {
        this.V = V;
        this.E = 0;
        adj = (LinkedList<Integer>[]) new LinkedList[V];
        for (int v = 0; v < V; v++)
            adj[v] = new LinkedList<>();
    }

    public int V() { return V; }

    public int E() {return E; }

    public void addEdge(int v, int w) {
        adj[v].add(w);
        E++;
    }

    public Iterable<Integer> adj(int v) {
        return adj[v];
    }

    public Digraph reverse() {
        Digraph R = new Digraph(V);
        for (int v = 0; v < V; v++)
            for (int w : adj(v))
                R.addEdge(w, v);
        return R;
    }
```

THE DIRECTED SEARCH PROBLEM

SEARCH

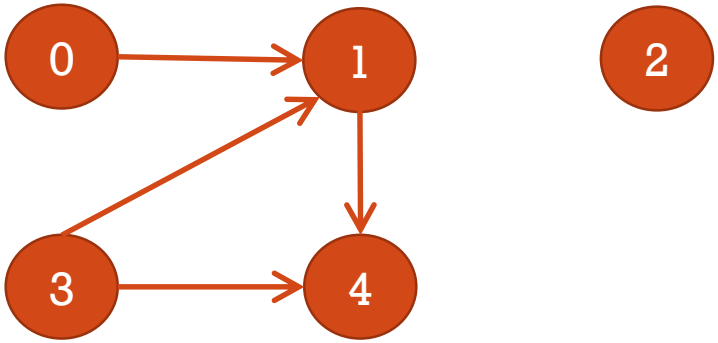
```
public class DirectedDFS
```

	`DirectedDFS(Digraph G, int s)`	*find vertices in G that are reachable from s*
	`DirectedDFS(Digraph G, Iterable<Integer> sources)`	*find vertices in G that are reachable from sources*
`boolean`	`marked(int v)`	*is v reachable?*

Let's be a little more specific defining our goals:

- *"Single-source reachability*: Given a digraph and source s, is there a directed path from s to v?"
- "*Multiple-source reachability*: Given a digraph and a set of source vertices, is there a directed path from any vertex in the set to v?"

Can we still use the idea of DFS?



search(G, 0)
marked(4)→T
marked(3)→ F
marked(2)→ F

search(G, {0, 2})
marked(4)→T
marked(3)→ F
marked(2)→ T

Quotes from http://algs4.cs.princeton.edu/42digraph/.

DIGRAPH DFS CODE

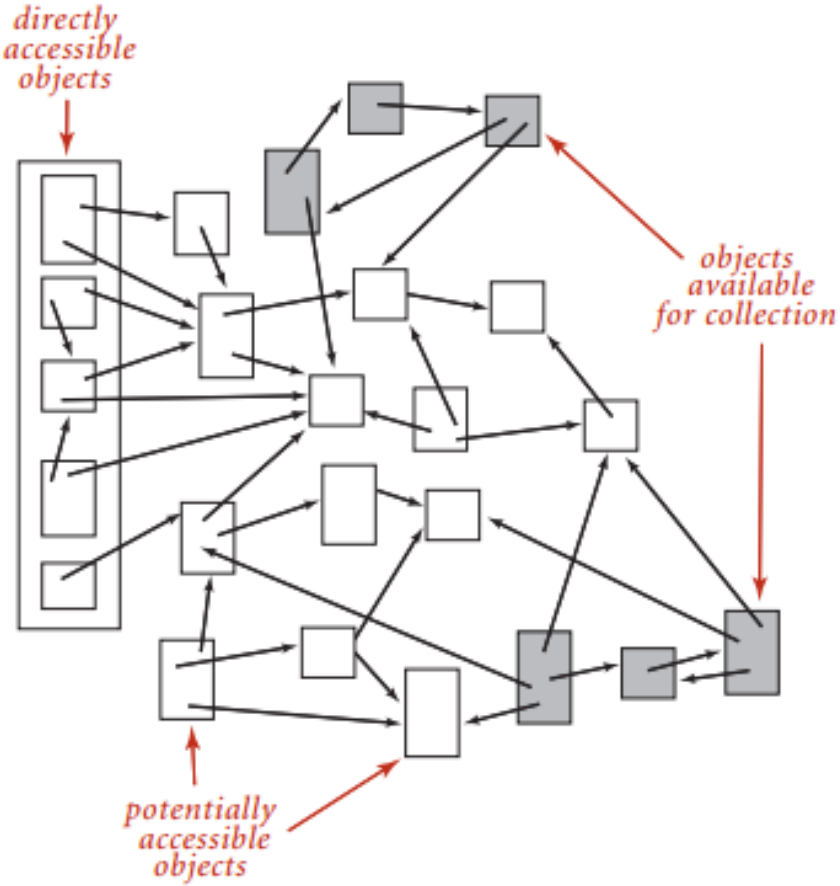
- Spoiler: the DFS method didn't change.
- Why?

```
private boolean[] marked;
public DirectedDFS(Digraph G, int s) {
    marked = new boolean[G.V()];
    dfs(G, s);
}

public DirectedDFS(Digraph G,
                   Iterable<Integer> srcs){
    marked = new boolean[G.V()];
    for (int s : srcs)
        if (!marked[s]) dfs(G, s);
}

private void dfs(Digraph G, int v) {
    marked[v] = true;
    for (int w : G.adj(v))
        if (!marked[w]) dfs(G, w);
}

public boolean marked(int v) {
    return marked[v];
}
```

MULTIPLE-SOURCE REACHABILITY

- Application: garbage collection.

- Idea: list out all the objects that are on the stack or statically allocated. Then, run a multiple-source search on those nodes in the context of a graph that shows which objects reference one another.



13

THE SCHEDULING PROBLEM

THE SCHEDULING PROBLEM

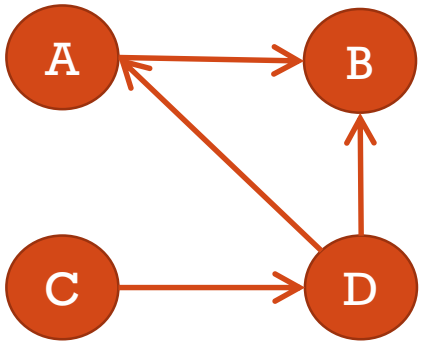
- Many problems can be expressed in terms of a dependency graph:

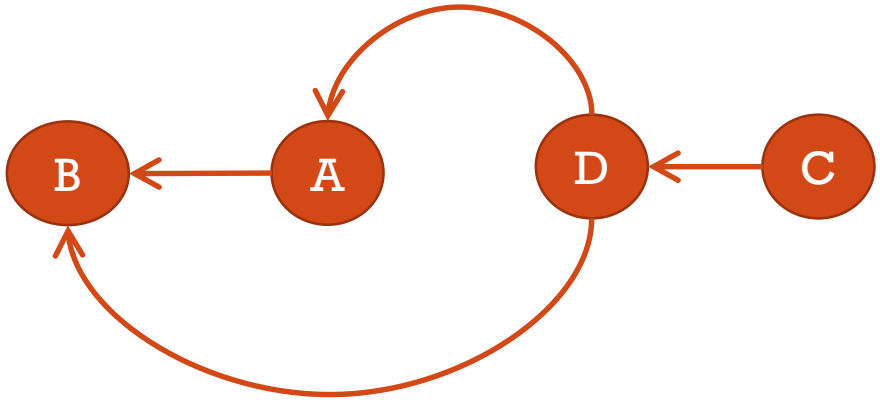
- May be read as A must happen before B, or A is a dependency of B.

- Let's say that A and B are tasks.
- Then, we can think about the scheduling problem: what order of vertices would I need to visit, in order to ensure I see all dependencies before each dependent?

TOPOLOGICAL SORTS

- To solve this problem, we must produce a *topological sort* of a graph. A *topological sort* is an ordered list of vertices in graph such that all dependencies are listed before their dependent.





- Do all directed graphs have a topological sort?

16

THE DIRECTED CYCLE PROBEM

DIRECTED CYCLES

The issue we have to worry about for finding the topological sort of a graph is if it has cycles. (Why?) We'll want to make sure we are working with a:

- Directed Acyclic Graph (DAG): a digraph without any cycles.

Our goal will be to implement a new algorithm, based on the undirected approach, to check if a graph contains a cycle.

public class	DirectedCycle	
	DirectedCycle(Digraph G)	*cycle-finding constructor*
boolean	hasCycle()	*does G have a directed cycle?*
Iterable<Integer>	cycle()	*vertices on a cycle (if one exists)*

FINDING CYCLES

- We've already seen how to check if a cycle exists in a undirected graph.
  - We basically run either DFS or BFS, and if any point, we encounter already marked node, we say there is a cycle.
- Could we do same thing for directed graphs?

FINDING CYCLES

- We've already seen how to check if a cycle exists in a undirected graph.
  - We basically run either DFS or BFS, and if any point, we encounter already marked node, we say there is a cycle.
- Could we do same thing for directed graphs?

DIRECTED CYCLE IMPLEMENTATION

```
private boolean[] marked;
private int[] edgeTo;
private Stack<Integer> cycle;
private boolean[] onStack;
public DirectedCycle(DiGraph G)
{
    onStack = new boolean[G.V()];
    edgeTo = new int[G.V()];
    marked = new boolean[G.V()];
    for (int v = 0; v < G.V(); v++)
        if (!marked[v]) dfs(G, v);
}
```

- Cycle is a stack that records the order of the nodes that leads to a cycle.
- onStack is flag for each node that tracks which nodes have been visited in the current recursive call.

```
private void dfs(DiGraph G, int v) {
    onStack[v] = true;
    marked[v] = true;
    for (int w : G.adj(v))
        if (this.hasCycle())
            return;
        else if (!marked[w]) {
            edgeTo[w] = v; dfs(G, w);
        }
        else if (onStack[w])
        {
            cycle = new Stack<Integer>();
            for (int x = v; x != w; x = edgeTo[x])
                cycle.push(x);
            cycle.push(w);
            cycle.push(v);
        }
    onStack[v] = false;
}
public boolean hasCycle() { return cycle != null; }
public Iterable<Integer> cycle() { return cycle; }
```

TOPOLOGICAL SORT

HOW DO WE GET TOPOLOGICAL SORTING?

- We already have it.
- Consider this idea: record the nodes as we visit them in the graph. This will give us a kind of structure.

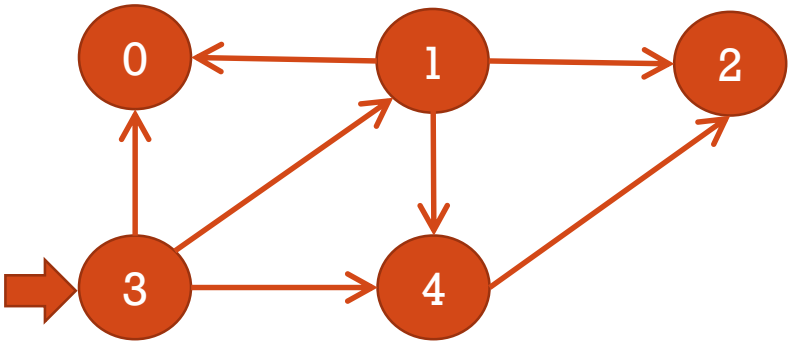
- Preorder: adding element to queue before the loop is started.
- Postorder: add element to queue after loop is done.
- Postorder reverse: add element to stack after loop is done.

DF ORDER CLASS

```
public class DepthFirstOrder {
    private boolean[] marked;
    private Queue<Integer> pre;
    private Stack<Integer> reversePost;

    public DepthFirstOrder(Digraph G) {
        pre = new LinkedList<>();
        reversePost = new Stack<>();
        marked = new boolean[G.V()];
        for (int v = 0; v < G.V(); v++)
            if (!marked[v])
                    dfs(G, v);
    }
    private void dfs(Digraph G, int v) {
        pre.add(v);
        marked[v] = true;
        for (int w : G.adj(v))
            if (!marked[w])
                dfs(G, w);
        reversePost.push(v);
    }

    public Iterable<Integer> reversePost() {
        return reversePost;
    }
}
```

POST ORDER EXAMPLE

DFS(3)
  DFS(0)
  DFS(1)
    DFS(2)
    DFS(4)

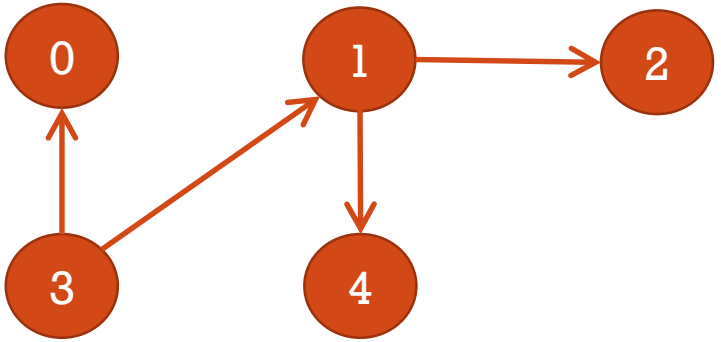


Preorder: 3, 0, 1, 2, 4

Postorder: 0, 2, 4, 1, 3

Reverse post: 3, 1, 4, 2, 0

Is this a topological sort?



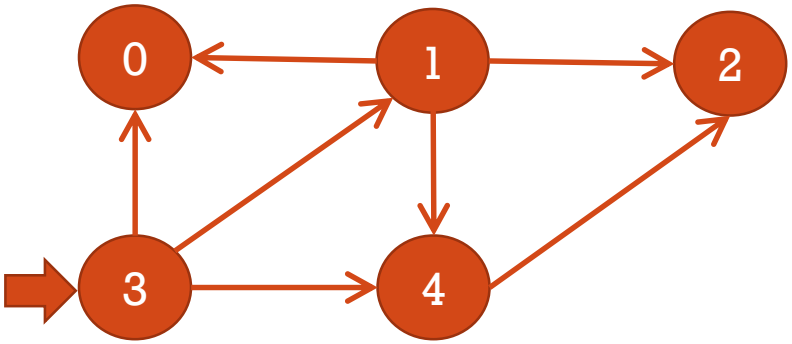
DFS root

BUT HOW?

- Since this is DFS and we are recording the vertices after the loop executes, any successors will already have been recorded.

- That is, before we push 3 on the stack, it already contains: 1, 4, 2, 0. These are all of it's successors. If the stack did not contain all of 3's successors, then 3 could not have been added yet.

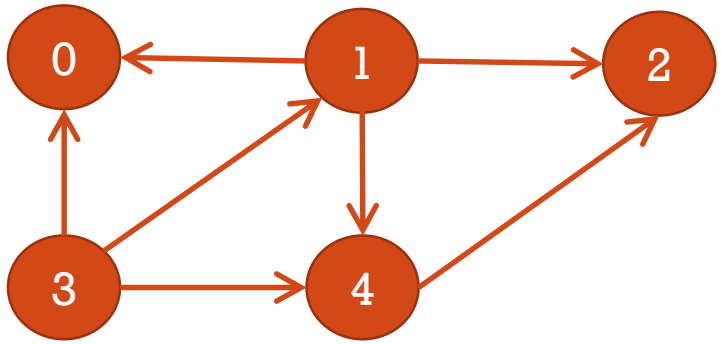
- This is recursive.




```
private Stack<Integer> reversePost;

private void dfs(Digraph G, int v) {
    marked[v] = true;
    for (int w : G.adj(v))
        if (!marked[w])
            dfs(G, w);
    reversePost.push(v);
}
```

NOT QUITE DONE…

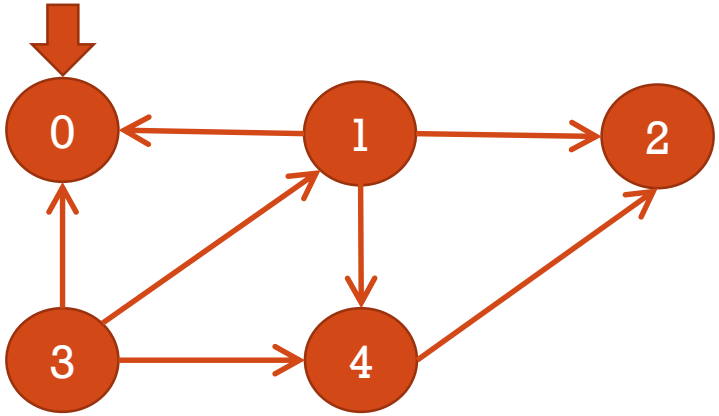
- Notice: we ran BFS(3). How did we know to pick that node? Could we have searched over the nodes looking for something of in-degree zero?
- Won't work. Consider if we have two nodes of in-degee zero, then we will need two searches to see them any anything they connect.
- For sure, we need multiple DFSs, but that means we have multiple results?!?!



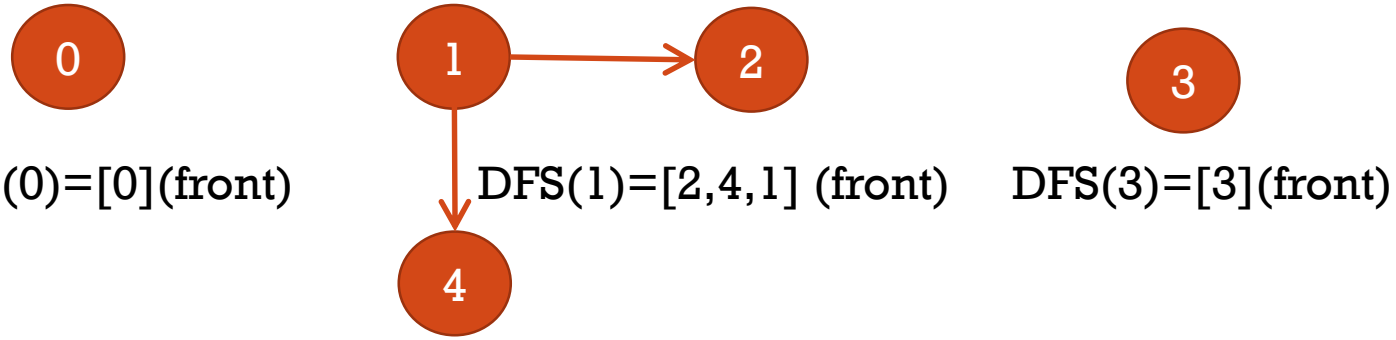
NOT QUITE DONE...

Consider this idea:

- Loop over every node, running DFS on that node if it hasn't been marked already.
- The result is a number of DFS trees with their own order.



Here: DFS(0), DFS(1), DFS(3)



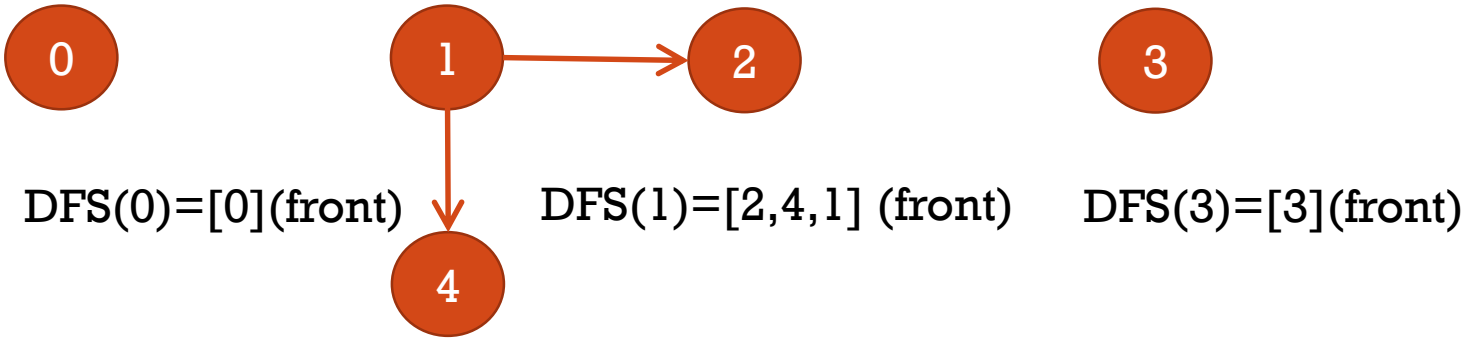
DFS(0)=[0](front)

DFS(1)=[2,4,1] (front)

DFS(3)=[3](front)

Observation #1: the result is a number of DFS trees, each with an associated topological sort for the nodes that were reached.

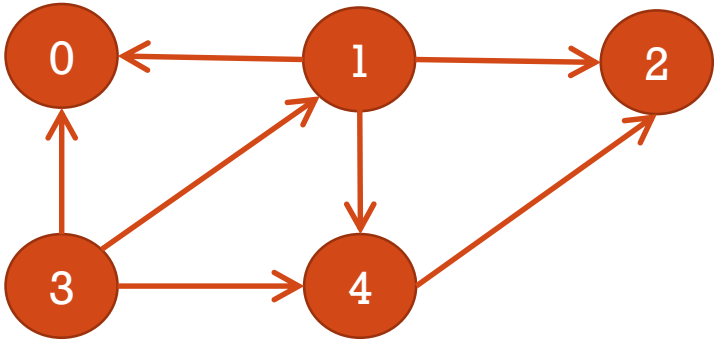
NOT QUITE DONE...



- Observation #2: since each tree includes all nodes that were reachable from the root, DFS(1) and DFS(3) cannot contain any nodes that are successors of DFS(0). Likewise, DFS(3) cannot contain any successors of DFS(1).
- Observation #3: DFS(1) may contain predecessors of DFS(0), DFS(3) may contain predecessors of DFS(0) or DFS(1).

Thus: if we process the topological sort for the last DFS tree, followed by the second to last DFS, and so on, the dependencies will be seen in order.

[0, 2, 4, 1, 3] (front)



THE TOPOLOGICAL CLASS

Straight Forward:

- Check if graph has a cycle, otherwise, run DFS on it and compute the nodes visited in reverse post order.

```
private Iterable<Integer> order;

public Topological(Digraph G) {
    DirectedCycle cyclefinder = new DirectedCycle(G);
    if (!cyclefinder.hasCycle()) {
        DepthFirstOrder dfs = new DepthFirstOrder(G);
        order = dfs.reversePost();
    }
}

public Iterable<Integer> order() {
    return order;
}

public boolean isDAG() {
    return order == null;
}
```