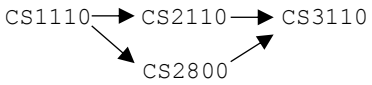
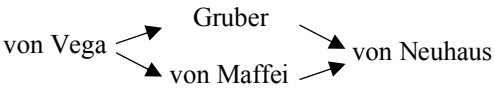
DAGS and Topological Sort

A graph is *acyclic* if it does not contain a cycle. A *d*irected *a*cyclic *g*raph is called a *DAG*.



To the right is an example of a DAG. The nodes are four of the courses that can be taken to satisfy a computer science major at Cornell. A directed edge from course `c1` to course `c2` is drawn if `c1` is a prerequisite for `c2`. In the full graph for the CS major, there better not be a cycle! One can use the graph to find the longest prerequisite chain —it shouldn't be more than 8 courses, and preferably less, so students can graduate in eight semesters.

DAGs can be used to represent scheduling problems. The tasks to be performed are the nodes of a graph, and a directed edge is drawn to indicate that one task (the source of the edge) must be done before another task (the sink of the edge). Laying out jobs to be done on a factory floor and figuring out the order in which to put clothes on (socks before shoes!) are examples.



Website https://www.genealogy.math.ndsu.nodak.edu contains the genealogy tree of mathematicians and computer scientists, with over 230,675 entries as of July 1918. Draw arrows from each PhD to the PhD's advisors (a PhD has one or two advisors). You might think that the result is a tree, but often it is a DAG. Here is an example from David Gries's PhD genealogy: von Vega advised Gruber and von Maffei, and those two together advised von Neuhaus. To see Gries' genealogy, type "Gries" or "DAG" into the JavaHyperText search field.

Family trees are often DAGS and not trees. But Mark Twain once wrote about how a person could be his own grandfather, which meant there was a cycle! Based on Twain's writing, Latham and Jaffe wrote a song "I'm My Own GrandPa". Google it.

Topological sort

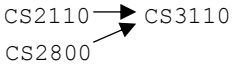
Suppose we want to order the nodes of a DAG so that the source of each directed edge precedes its sink. With the prerequisite DAG shown above, there are the two possible orderings:

`CS1110, CS2110, CS2800, CS3110`
`CS1110, CS2800, CS2110, CS3110`

Ordering the nodes of a DAG in this fashion is a called *topological sort*. We develop an algorithm for performing a topological sort, giving first an abstract algorithm based on a neat idea and then figuring out how to implement it efficiently.

The basic idea

A DAG must have a node with indegree 0 —if not, there would be a cycle and it would not be a DAG. So, choose a node with indegree 0, make it the first node in the ordering, and delete it and all edges leaving it from the graph.



For the prerequisite graph shown above, `CS1110` has indegree 0. Therefore, it comes first, and the graph is changed to the one on the right.

Repeat the process: Choose a node with indegree 0 —there are two, `CS2110` and `CS2800`, and either one can be chosen—; make it the next node in the ordering; and delete it and all edges leaving it from the graph. Continue this process.

Thus, we have this little algorithm:

```
while (the graph has a node) {
   Choose a node w (say) with indegree 0;
   Make w the next node in the ordering;
   Delete w and all edges leaving it from the graph
}
```

Note that if at some point the graph contains a node but no node has indegree 0, then there is a cycle. Thus, this little algorithm can be modified to determine whether a digraph is indeed a DAG.

Writing an efficient Java program for topological sort

The algorithm written above is not realistic. The graph can't be changed. Instead, we have to maintain information about how the graph *would be* changed during the above algorithm.

To the right, we give the spec and header of the method we will write. Besides class `Graph`, there will be classes `Node` and `Edge`. We'll use obvious methods of these classes and explain everything carefully. The method starts by saving the number of nodes and the nodes themselves in local variables. We now develop the rest of the method.

```
/** Topological sort; Return a list of nodes in DAG g such
 *  that for each edge, its source precedes its sink in the list.
 *  Throw an IllegalArgumentException if g is not a DAG. */
public static List<Node> topologicalSort(Graph g)
   int n= g.getNodesSize();       // number of nodes in g
   HashSet<Node> nodes= g.getNodes(); // nodes of g
```

We need to maintain the list of indegrees of the nodes. They indegrees will change, as they do in the abstract algorithm above. It would be nice if class `Node` had a field for this purpose, but it doesn't. So, we need a data structure for it. The most efficient is class `HashMap<Node, Integer>` —give an object of the class a node and it tells you its indegree. Also, we will want a list of nodes of indegree 0 —we make it a stack, but it could be list or set.

Thus, we use two local variables, `s` and `indegrees`. We initialize `s` to contain nodes of indegree 0 and `indegrees` to contain all other nodes.

```
Stack<Node> s= new Stack<>();
HashMap<Node, Integer> indegrees= new HashMap<>();

// Store in s a list of all nodes of indegree 0 and in
// indegrees all nodes with indegree > 0, with those indegrees
for (Node node : nodes) {
   int d= node.indegree();    // retrieve the indegree of node
   if (d == 0) s.add(node);
   else indegrees.put(node, d);
}
```

We will use a `List res` to contain the list of nodes in topological order. We are about to write the loop, each iteration of which places a node into `res`. In order to be able to understand this loop, we have to first give its invariant, defining each data structure carefully. Study it reading further! It is important to note that graph `GP` is the graph mentioned in the earlier abstract algorithm, and it is found by deleting all nodes in list `res`.

```
/* invariant: Consider the graph GP containing:
 * (1) the nodes of g that are not in res and
 * (2) the edges of g that have node in GP as sources.
 * Then
 * 1. For each node w in res, if some edge e in g has w as
 *    the sink, e's source precedes w in res.
 * 2. s contains exactly the nodes in GP with indegree 0.
 * 3. For each node m in GP of indegree > 0,
       (m, indegree of m in GP) is in indegrees. */
```

Now, it's a simple task to write the main loop of the method. Each iteration of the loop places another node in list `res`. Method `node.directedNeighbors` returns all neighbors of `node`.

It's fairly easy to check the correctness of this loop using the four loop questions for loops.

Finally, remember that the original graph had `n` nodes. If `n` nodes have not been placed in `res` at the end, then graph `GP` is not empty and has no node of indegree 0, so there must be a cycle. In this case, the exception is thrown.

You can verify that this method takes expected time O(`n + e`) where `n` is the number of nodes and `e` is the number of edges in `g`.

```
List<Node> res= new ArrayList<>();
// invariant: see above
while (s.size() > 0) {
     Node node= s.pop();
     result.add(node);
     List<Node> nn= node.directedNeighbors();
     for (Node sink : nn) {
        int d= indegrees.get(sink)-1;
        if (d == 0) s.push(sink);
        else indegrees.put(sink, d);
     }
}
if (result.size() < n)
     throw new IllegalArgumentException("g not a DAG");
```