List of Slides

Topic 11

Recursive Data Structures: Directed Graphs

Section 1

Directed Graphs

Directed Graphs

- Another recursive data structure related to trees is **directed graph**s.
- These are mostly used for dependency modelling.
- For example the task dependencies in the mini-project were modelled using a program called **DependEdit**, which uses a directed graph.

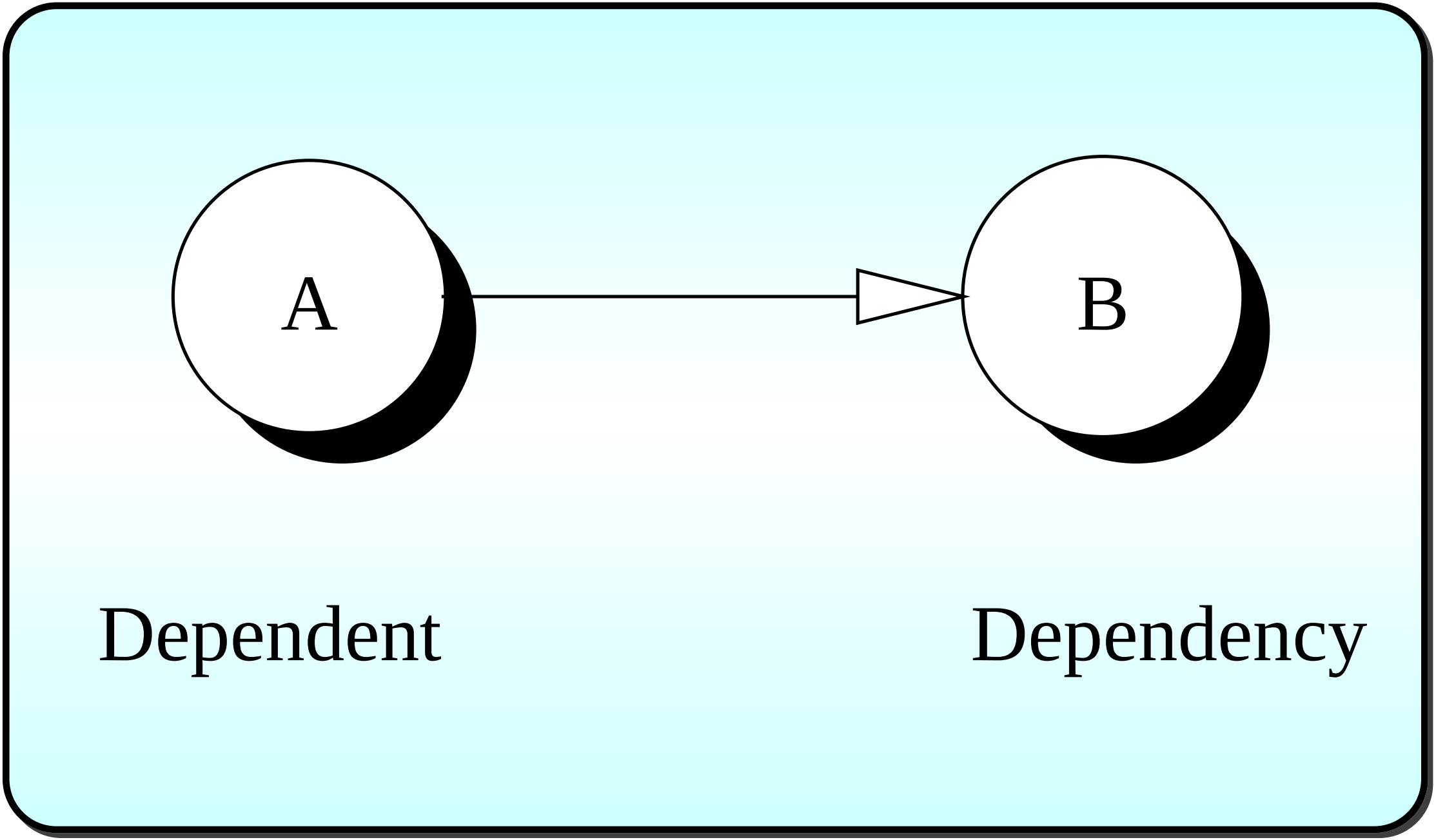
What are Directed Graphs?

- A **directed graph** consists of **graph node**s linked together by **graph arc**s.
- Arcs are *directed* – they connect *from* one node *to* one other. (In non-directed graphs the arcs just join two nodes, with no order of the nodes.)
- When used for dependency modelling, the nodes are entities from the real world scenario being modelled, and the arcs are dependencies between entities.
- E.g. nodes might be tasks in the mini-project, and arcs the dependencies between tasks: division depends on multiplication.
- We shall not look at graphs where the arcs have no direction, and so for economy we shall refer to directed graphs as just **graph**s.

Terminology: Dependent versus dependency

- It is important to get the terminology right early on.
- Say node *A* depends on *B*. E.g. division depends on multiplication.
- *A* is a **dependent** of *B*. E.g. division is a dependent of multiplication.
- *B* is a **dependency** of *A*. E.g. multiplication is a dependency of division.
- In general *A* can have many dependencies, including *B*.
- In general *B* can have many dependents, including *A*.

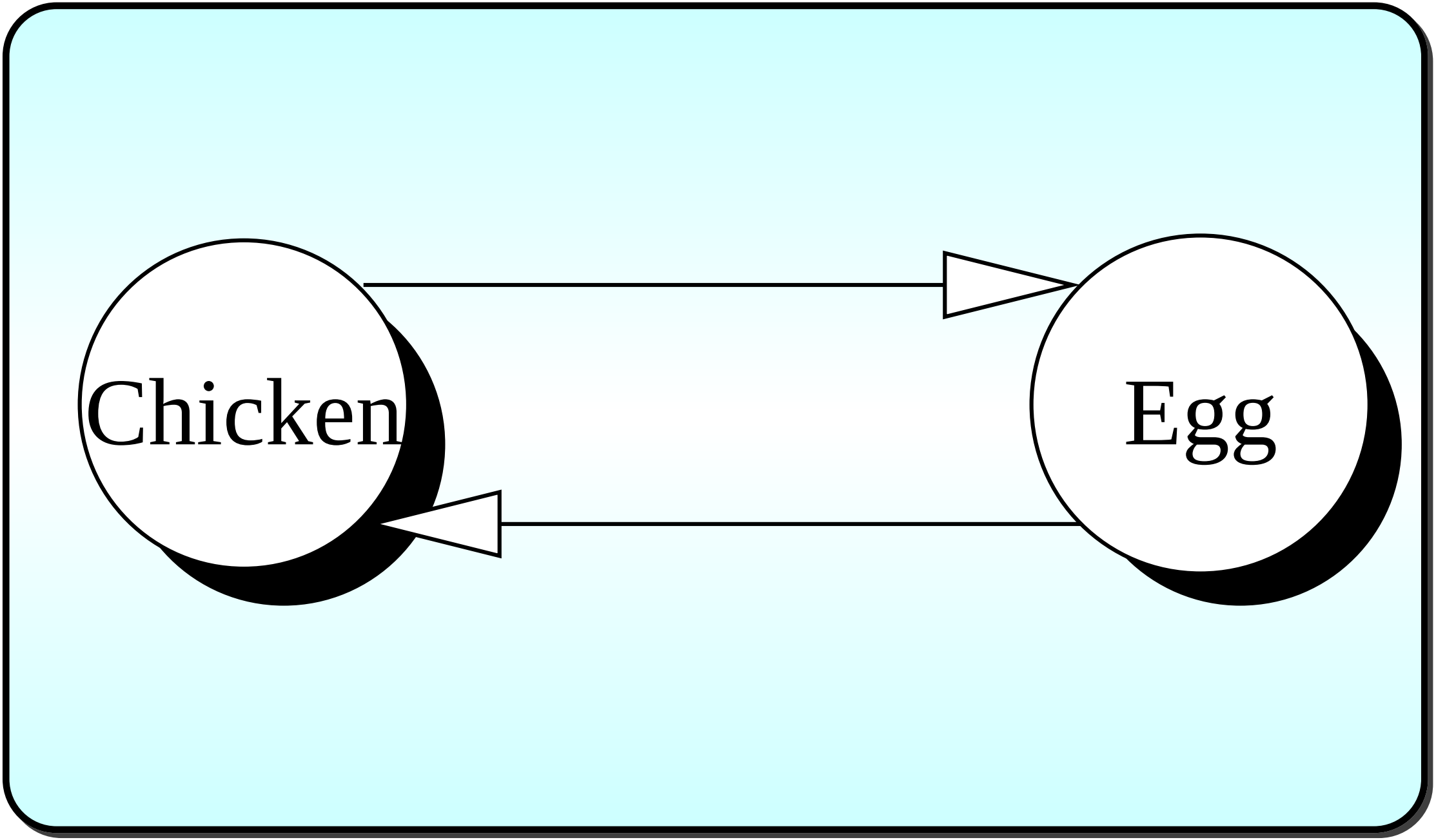
Terminology: Dependent versus dependency



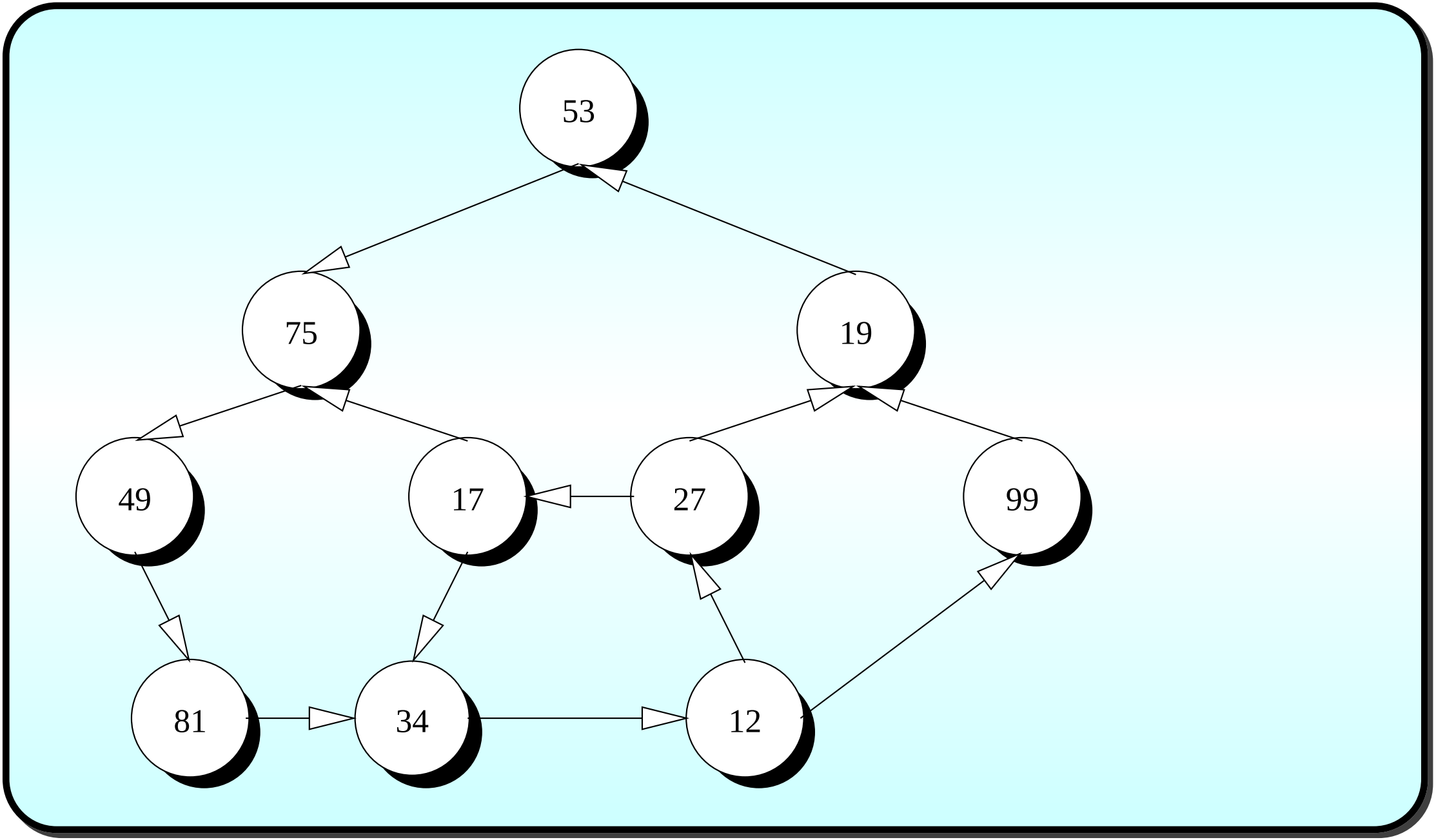
Cyclic graphs

- Graphs can be **cyclic graph**s.
- This means it is possible to follow arcs from some node and arrive back at that same node.
- Let us not permit a node to be directly dependent on itself. That is, every arc must connect from one node to a different node. (Some people permit self dependency in their graphs – we shall not.)
- In dependency modelling terms, if a graph is cyclic, then some entity is **mutually dependent** on one or more other entities.

Cyclic graph: chicken and egg



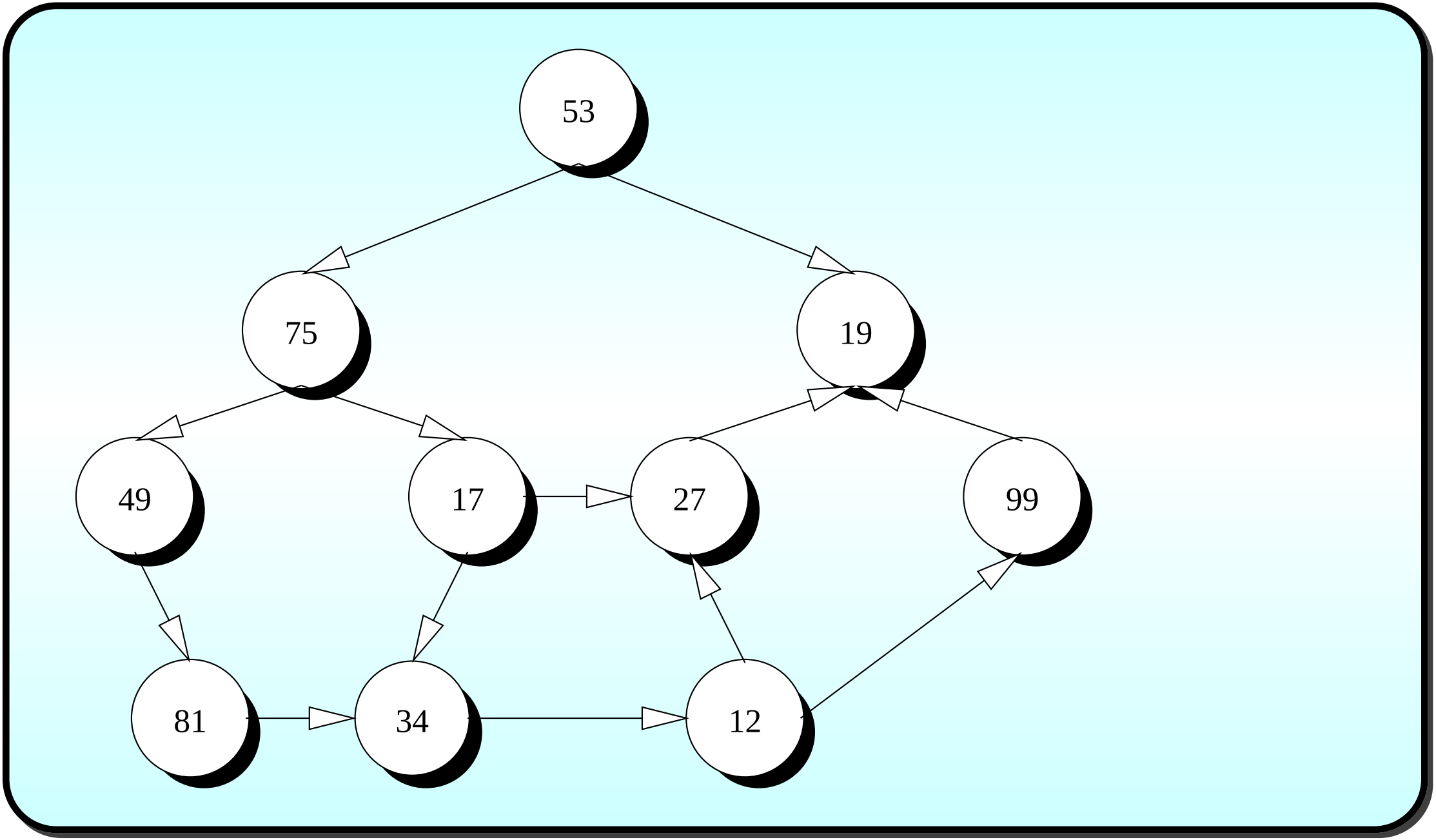
Cyclic graph



Acyclic graphs

- An **acyclic graph** has no cycles.
- There are no infinite paths.
- There may still be many paths from any node to any other node.
- But there are no (non-empty) routes from any node to itself.

Acyclic graph

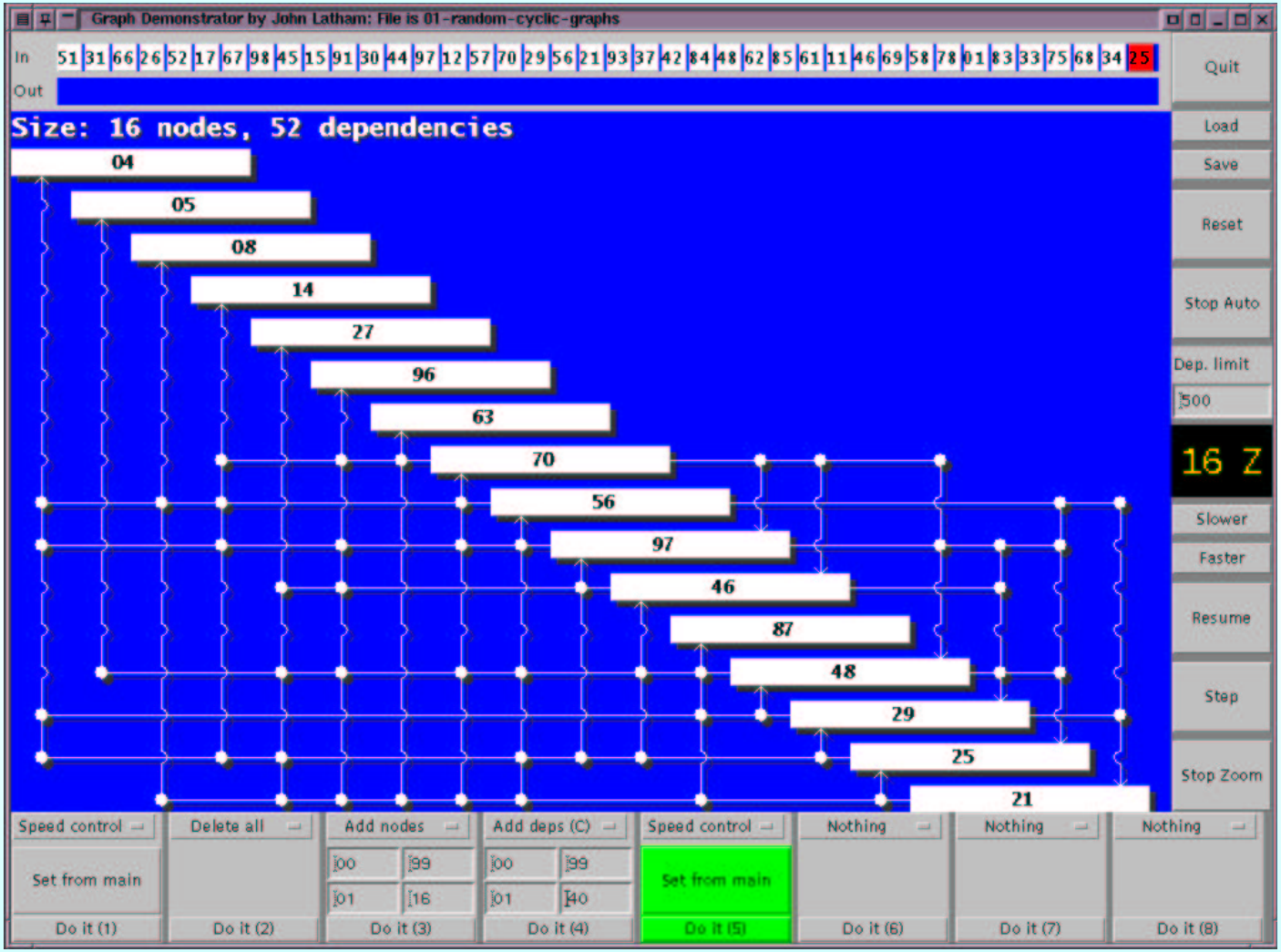


Trees are graphs

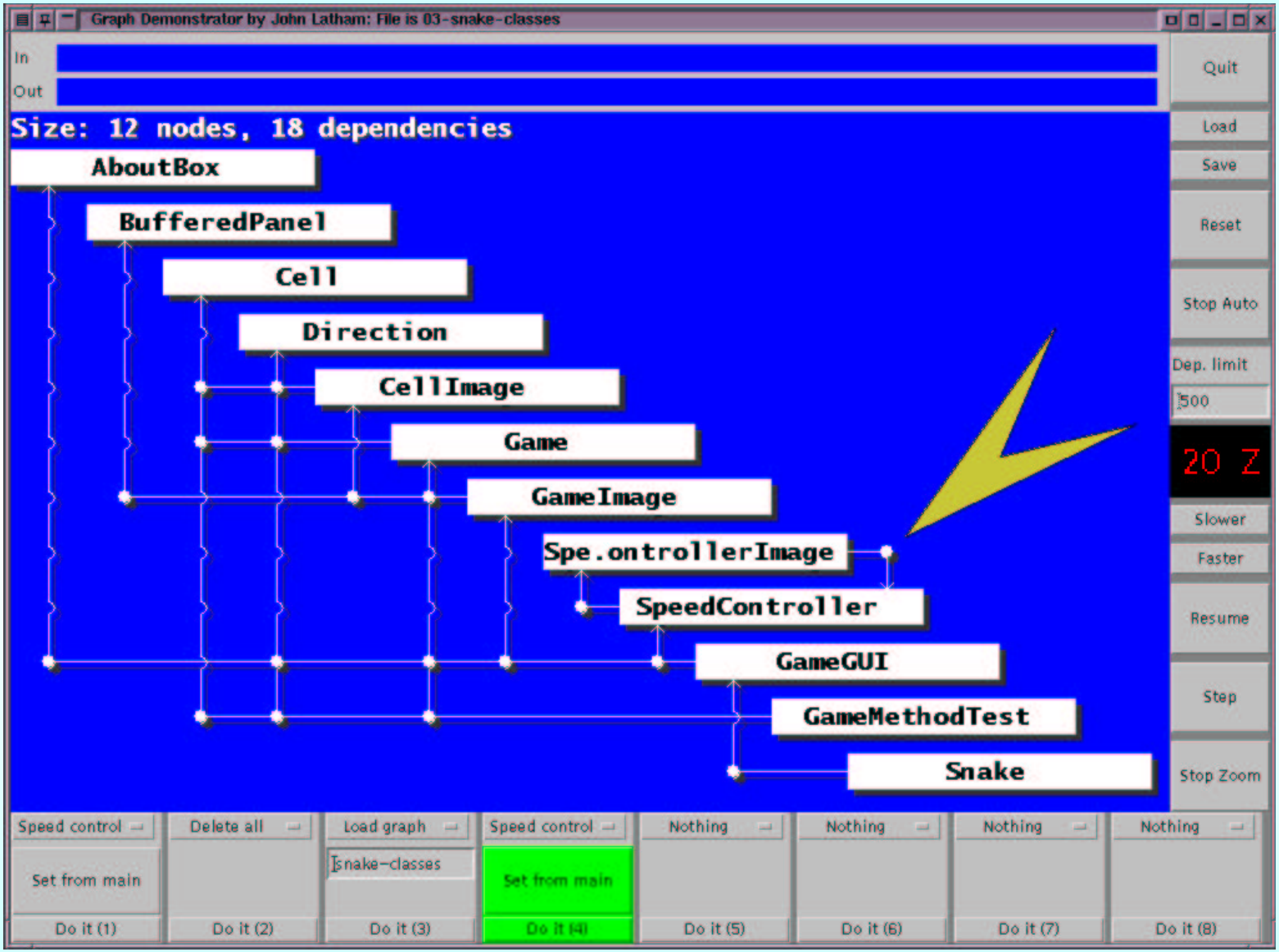
- A **tree** is a special kind of **acyclic graph**.
- Each node, except the root, has exactly one dependency – called the parent. The **root** has none.
- Nodes can have many dependents – called children. In **binary tree**s the **non-empty node**s have two dependents, which are the roots of the **left child** and **right child** sub-trees.
- There is at most one route from any node to any other node, because each node has only one dependency. (There is no route from a node to most others.)
- Trees are always drawn with dependencies going upwards so there is no need to show the arrow heads.
- But they are still graphs.

Graph demonstrator

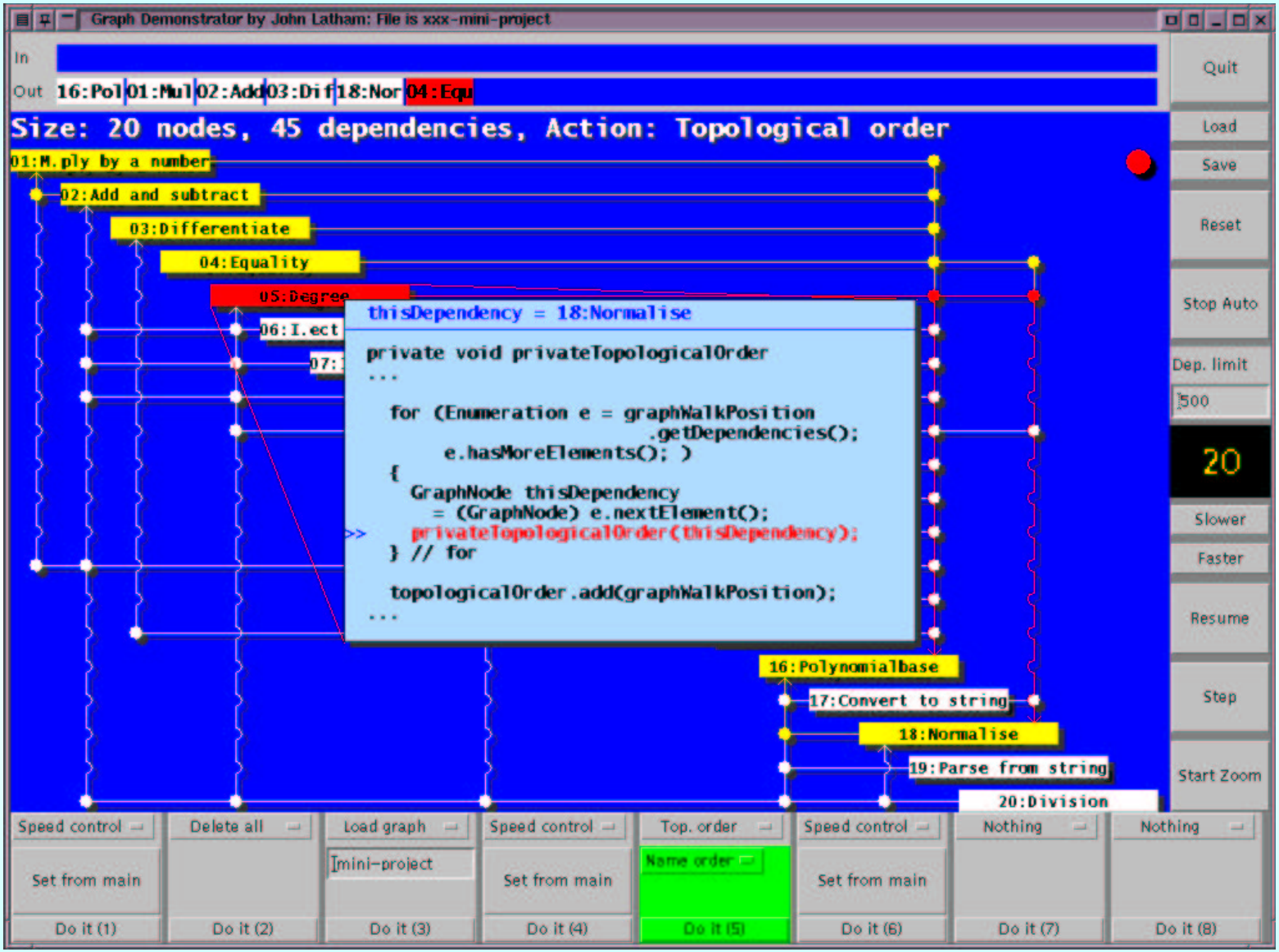
I have built a graph demonstrator which we will use to explore graphs.



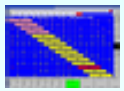
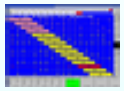
Graph demonstrator



Graph demonstrator



Example graphs

- Let us look at some randomly created **cyclic graph**s and **acyclic graph**s.
- Graph demo:  random-cyclic-graphs
  - Note the way graphs are drawn here: the diagonal line of nodes allows us to easily connect any node to any other. We permit arcs destined for the same node to join together for economy of display. Arcs on the left of the diagonal are going upwards, whereas arcs on the right are going downwards.
- Graph demo:  random-acyclic-graphs
  - Note that the acyclic graphs have no arcs on the right of the diagonal line of nodes. The way the graph demonstrator (usually) draws the graphs enables us to easily spot any cycles.

Applications of graphs: Java classes

- When we develop Java programs, we can easily end up with a large number of classes – indeed we usually start by identifying these classes in advance.
- A useful notion is simple ‘textual dependency’ of the classes in a Java program: i.e. one class makes *some* reference to another.
- The textual dependencies of the classes in a directory can be found using a simple shell script.

Shell script to find class dependencies

```
#!/bin/sh
for i in ‘ls *.java | sed "s,.java,,g"‘
do
  echo $i
  for j in ‘ls *.java | sed "s,.java,,g"‘
  do
    if test "$i" != "$j" \
       &&  cat $i.java | sed ’s,"[^"]*",,g’ \
           | egrep \(^\|[^A-Za-z0-9]\)$j\(\$\|[^A-Za-z0-9]\) \
               > /dev/null
    then
      echo $j
    fi
  done
  echo
done
```

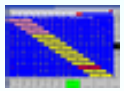
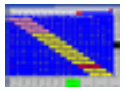
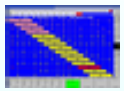
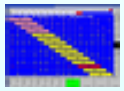
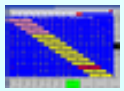
Output for Snake program (as 3 columns)

```
AboutBox
BufferedPanel
CellImage
Cell
Direction
Cell
Direction
```

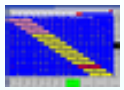
```
GameGUI
AboutBox
Direction
GameImage
Game
SpeedController
GameImage
BufferedPanel
CellImage
Game
Game
Cell
Direction
```

```
GameMethodTest
Cell
Direction
Game
Snake
GameGUI
SpeedControllerImage
SpeedController
SpeedController
SpeedControllerImage
```

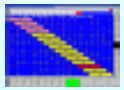
Applications of graphs: Java classes

- We can run the shell script and store the results in a file. Quite deliberately, the output format matches the files used by **DependEdit** and **GraphDemo**, so the graphs can be displayed by those programs.
- Graph demo:  snake-classes
- Graph demo:  lottery-classes
- Graph demo:  graph-demo-classes
- Graph demo:  tree-demo-classes
- Graph demo:  depend-edit-classes

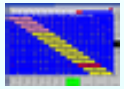
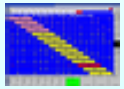
Applications of graphs: Web pages

- A similar script can find the references in a set of html files.
- For example, the web pages for `AnotherLevelUp`.
  - Graph demo:  ALU-html

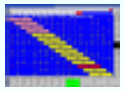
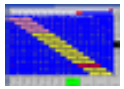
Topological order

- A **topological order** of a graph is a list of the nodes, such that each node is listed after any it depends upon.
- Clearly it is only possible to have a topological order for an *acyclic* graph.
- We can attempt to find a topological order for any graph, of course, but any cycles will stop us achieving one.
- The graph demonstrator (usually) attempts to display the nodes in a topological order – that is why there are no arcs on the right hand side for acyclic graphs: the dependency lines only need to point upwards.
- Graph demo:  random-acyclic-graphs

Applications of graphs: task order planning

- A **topological order** is particularly useful when we wish to plan an order of tasks which have inter dependencies: e.g. we cannot put hot water on our tea bag until we have boiled the kettle.
- If these dependencies are not cyclic, then we can build a graph where the nodes are the tasks, express the dependencies, and find a topological order.
- We can then perform the tasks in that order.
- Graph demo:  house
- Graph demo:  mini-project

Topological order is not uniq

- Most **acyclic graphs** have more than one **topological order**.
- For example, if *C* depends on *A* and *C* also depends on *B*, then both *A* and *B* must appear before *C*, but *A* and *B* could come in either order if there is no dependency between them.
- Graph demo:  house-orders
- Graph demo:  family-orders
  - A curious use of graphs is to model a family tree. These are acyclic (of course!) but they are not really trees in the strict sense.
  - Children depend on both their parents, and people can have children with different people as co-parent, so parents are separate nodes.
  - As we have only one kind of arc in our graphs, we (arbitrarily?) decide that a husband / male partner depends on his wife / female partner.

Section 2

Implementation of Directed Graphs

Implementation of directed graphs

- We shall study a way of implementing graphs.
- We shall study the simplest algorithm for finding a topological order.
- We shall talk about maintaining transitive dependencies.
- We shall talk about more complicated topological order algorithms.

Graphs are a recursive data structure

- Graphs are a recursive data structure.
- Here is one definition, leading to an implementation.
  - A graph node comprises a name (or a piece of data),
  - a possibly empty set of graph nodes, which it depends upon,
  - and a possibly empty set of graph nodes, which depend on it.
- Note that we do not really need to store both dependents and dependencies: one would do to maintain the graph structure. For efficiency we store them both and we must keep them consistent.

GraphNode code

```
import java.util.*;

public class GraphNode
{
  // The name of this node.
  private String name;

  // The nodes this node depends upon.
  private ArrayList dependencies = new ArrayList();

  // The nodes which depend on this node.
  private ArrayList dependents = new ArrayList();
```

GraphNode constructor

```
// Create a graph node.
public GraphNode(String requiredName)
{
  name = requiredName;
} // GraphNode
```

getName()

```
public String getName()
{
  return name;
} // getName
```

addDependency()

```
// Record that this node depends on the given one.
public boolean addDependency(GraphNode dependency)
{
  if (this == dependency)
    return false; // Self dependency not permitted.
  else
  {
    // Add dependency to dependencies if not already present.
    if (dependencies.contains(dependency))
      return false;
    dependencies.add(dependency);
    dependency.dependents.add(this);
    return true;
  } // if
} // addDependency
```

deleteDependency()

```
// Record that this node does not depend on the given one.
public boolean deleteDependency(GraphNode dependency)
{
  // Remove dependency from dependencies.
  boolean result = dependencies.remove(dependency);
  // Remove this from dependents of dependency.
  dependency.dependents.remove(this);
  return result;
} // deleteDependency
```

deleteAllDependencies()

```
// Remove all the dependencies of this node.
public void deleteAllDependencies()
{
  for (int i = 0; i < dependencies.size(); i++)
  {
    GraphNode dependency
      = (GraphNode) dependencies.get(i);
    dependency.dependents.remove(this);
  } // for
  dependencies.clear();
} // deleteAllDependencies
```

dependsOn()

```
// Does this node depend on the given other one?
public boolean dependsOn(GraphNode other)
{
  return dependencies.contains(other);
} // dependsOn
```

dependencyCount()

```
// Get the number of dependencies.
public int dependencyCount()
{
  return dependencies.size();
} // dependencyCount
```

dependentCount()

```
// Get the number of dependents.
public int dependentCount()
{
  return dependents.size();
} // dependentCount
```

getDependencies()

```
// Get the dependencies, in a default order.
public Iterator getDependencies()
{
  return dependencies.iterator();
} // getDependencies
```

getDependents()

```
// Get the dependents, in a default order.
public Iterator getDependents()
{
  return dependents.iterator();
} // getDependents
```

Finding a topological order

- We are given a graph node.
- First we ensure all its dependencies are in the resulting topological order, by recursively considering each of them.
- Then we add the given graph node to the topological order.
- Overall then we have added the given graph node and all its dependencies to the topological order, if they were not already in it.
- So if we start with an empty topological order, we end up with the topological order of all nodes which are accessible from the given starting one.
- If we terminate . . . .

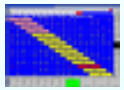
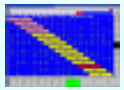
Dealing with cycles

- How do we deal with cycles?
- Cycles would lead to infinite recursion as we attempt to process the dependencies of a node before we add it to the result, and cycles mean some nodes are their own indirect dependency.
- We detect cycles by keeping a set of all the graph nodes we are currently considering, and do nothing if we arrive at one which is already in that set.
- So, if our graph is cyclic we will finish with an attempted topological order, and if it is acyclic, we will get a topological order.
- The topological order will contain all the nodes in the graph, if they are accessible from the node we started with . . . .

Which graph node do we start at?

- How do we make sure all the graph nodes are processed and so end up in the topological order?
- Depending on where we start we might not visit every node.
- Indeed, our graph might not even be a **connected graph**: it might actually consist of a collection of separate graphs.
- One simple way of making all our graph nodes be part of a single graph is to have one extra node, called the **end node**. We make sure this node is dependent upon all the other nodes in the graph.
- For example, the dependency graph of making a house can be thought of as having one one extra node, perhaps called “Finished!”. Every task which is part of the project has this end node as a dependent.
- To find the topological order of all the nodes in a graph, we simply start with the end node.

Example topological order

- The graph demonstrator maintains an end node for its graphs, but does not normally show it because it is not interesting.
- One time when we can see it is during the computing of a topological order.
- Also, graphs are usually displayed in a topological order. However, during the computing of a topological order they are shown in alphabetical order by name.
- Graph demo:  house-top-order
- Graph demo:  mini-project-top-order

topologicalOrder()

```
// A simple topologicalOrder method.
public Iterator topologicalOrder()
{
  topologicalOrder = new ArrayList();
  nodesBeingConsidered = new HashSet();
  privateTopologicalOrder(this);
  return topologicalOrder.iterator();
} // topologicalOrder


private ArrayList topologicalOrder;
private HashSet nodesBeingConsidered;
```

privateTopologicalOrder()

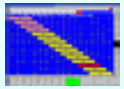
```
// Recursive part of the simple topologicalOrder method.
private void privateTopologicalOrder
                (GraphNode graphWalkPosition)
{
  // Check for cycles & multiple paths.
  if (nodesBeingConsidered.contains(graphWalkPosition))
    return;
  // Check if already added this node.
  else if (topologicalOrder.contains(graphWalkPosition))
    return;

  nodesBeingConsidered.add(graphWalkPosition);
```

privateTopologicalOrder()

```
    for (Iterator i = graphWalkPosition.getDependencies();
         i.hasNext(); )
    {
      GraphNode thisDependency = (GraphNode) i.next();
      privateTopologicalOrder(thisDependency);
    } // for

    topologicalOrder.add(graphWalkPosition);
    nodesBeingConsidered.remove(graphWalkPosition);
  } // privateTopologicalOrder
```

- Graph demo:  mini-project-top-order
  - See the code. Please note that to clarify things, the demonstrator ensures the dependencies of a node are considered in alphabetic order.

End of `GraphNode`

```
} // class GraphNode
```

- That concludes our short detailed coverage of the implementation of graphs.
- Remaining issues for less detailed discussion include
  - maintaining transitive dependencies
  - and more complicated topological order algorithms.

Transitive dependencies

- It can be useful to know if a graph node **transitively depend**s upon another.
- If *A* transitively depends on *C* it means there is another, *B*, such that *A* depends on *B* and *B* depends on *C*, or *B* transitively depends on *C*.
- This could be computed on demand as needed, but it is not quick to do so.
- The approach taken here (but not shown in detail) is to have as instance variables a third `ArrayList` called `transitiveDependencies` and an associated `boolean` variable called `transitiveDependenciesAreUpToDate`.
- Every time the dependencies of a node are changed, we set `transitiveDependenciesAreUpToDate` to false, and also do so (recursively) for all nodes which are dependents of that node.

Transitive dependencies

- Whenever we need to look up a transitive dependency, if `transitiveDependenciesAreUpToDate` is false we build the `transitiveDependencies` first and set that variable to true.
- To build `transitiveDependencies` we add every dependency of every dependency of this node. We then do the same for every transitive dependency of every dependency of this node.
- Finding the transitive dependencies of every dependency of this node recursively causes them to be computed, if necessary.
- Sounds complicated? It is!

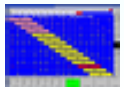
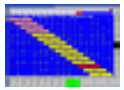
More complicated topological orders

- We have said that topological orders are (generally) not unique.
- The resulting order depends on the order in which the dependencies of each node are considered: essentially the sooner a node is considered, the sooner it will appear in the topological order.
- The simple alogorithm we saw in detail uses whatever default order is obtained by iteration from the `ArrayList` of dependencies, which will be the order the dependencies were added. The graph demonstrator actually considers the dependencies in alphabetical order for clarity.

More complicated topological orders

- Other possible orders include
  - random order – the dependencies of a node are jumbled into a random order before they are processed.
  - ascending dependency – the dependencies of a node are processed from the least total dependency to the highest. The total dependency is the sum of the number of dependencies and the number of transitive dependencies.
  - descending dependency – the dependencies of a node are processed from the most total dependency to the lowest.
- We can also make our topological order **eager for dependents**. This means that when we add a graph node to the topological order, we immediately consider all the nodes which are dependents of it. This can lead to **clustering** of groups of closely dependent nodes.

Example topological orders (again)

- Graph demo:  house-orders
- Graph demo:  family-orders