Solutions to Exercises in Chapter 15

15.2 In the motorway network of Figure 15.1, many paths connect any given pair of cities. Below are just two possible paths for each pair of cities.

(a) Perth to Exeter:

«Perth, Edinburgh, Glasgow, Carlisle, Manchester, Birmingham, Bristol, Exeter»	7 edges	870 km
«Perth, Edinburgh, Glasgow, Carlisle, Manchester, Leeds, Rugby, London, Bristol, Exeter»	9 edges	1180 km
…		

(b) Dover to Leeds:

«Dover, London, Rugby, Leeds»	3 edges	440 km
«Dover, London, Birmingham, Manchester, Leeds»	4 edges	510 km
…		

(c) Liverpool to Cambridge:

«Liverpool, Manchester, Birmingham, London, Cambridge»	4 edges	480 km
«Liverpool, Manchester, Leeds, Rugby, London, Cambridge»	5 edges	550 km
…		

15.4 The `qualified` method of Example 15.1 will unnecessarily visit the same node (and its successors) more than once, if that node contains a course that is a prerequisite of more than one other course.

To avoid this inefficiency, ‘mark’ nodes as suggested in Section 15.7 (page 417). In the following, the auxiliary method `qual` has an extra parameter that is a set of marked nodes. On the initial call, that set is empty.

15.5 Edge-set, adjacency-set, and adjacency-matrix representations of the directed graph of Figure 15.3 are shown in Figures S15.2, S15.3, and S15.4, respectively.

15.6 In the edge-set representation of graphs, we can represent the node set by an SLL (rather than a DLL). The `removeEdge` operation then has to use the SLL deletion algorithm, which is $O(e)$, where e is the number of edges. See Table S15.5.

Alternatively we can represent the node set by a hash table with elements as keys (rather than by a DLL). The `addNode` and `removeNode` operations then use (more or less) the standard hash-table insertion and deletion algorithms. See Table S15.6.

```
public static boolean qualified (
              Graph.Node courseNode,
              Set coursesPassed,
              Digraph curriculum) {
  return qual(courseNode, coursesPassed,
      new TreeSet(), curriculum);
}

private static boolean qual (
              Graph.Node courseNode,
              Set coursesPassed,
              Set nodesMarked,
              Digraph curriculum) {
  nodesMarked.add(courseNode);
  Iterator prereqs =
      curriculum.successors(courseNode);
  while (prereqs.hasNext()) {
    Graph.Node prereqNode =
        (Graph.Node) prereqs.next();
    if (! coursesPassed.contains(
          prereqNode.getElement()))
      return false;
    if (! nodesMarked.contains(prereqNode))
        && ! qual(prereqNode, coursesPassed,
              nodesMarked, curriculum))
      return false;
  }
  return true;
}
```

Program S15.1 Method to test whether a student is qualified to take a given course.

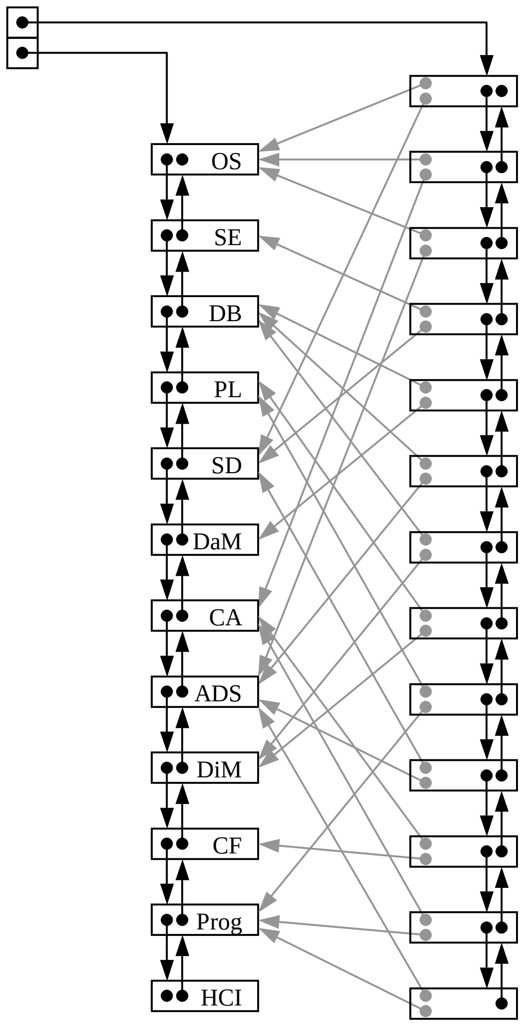

Figure S15.2 Edge-set representation of the directed graph of Figure 15.3.

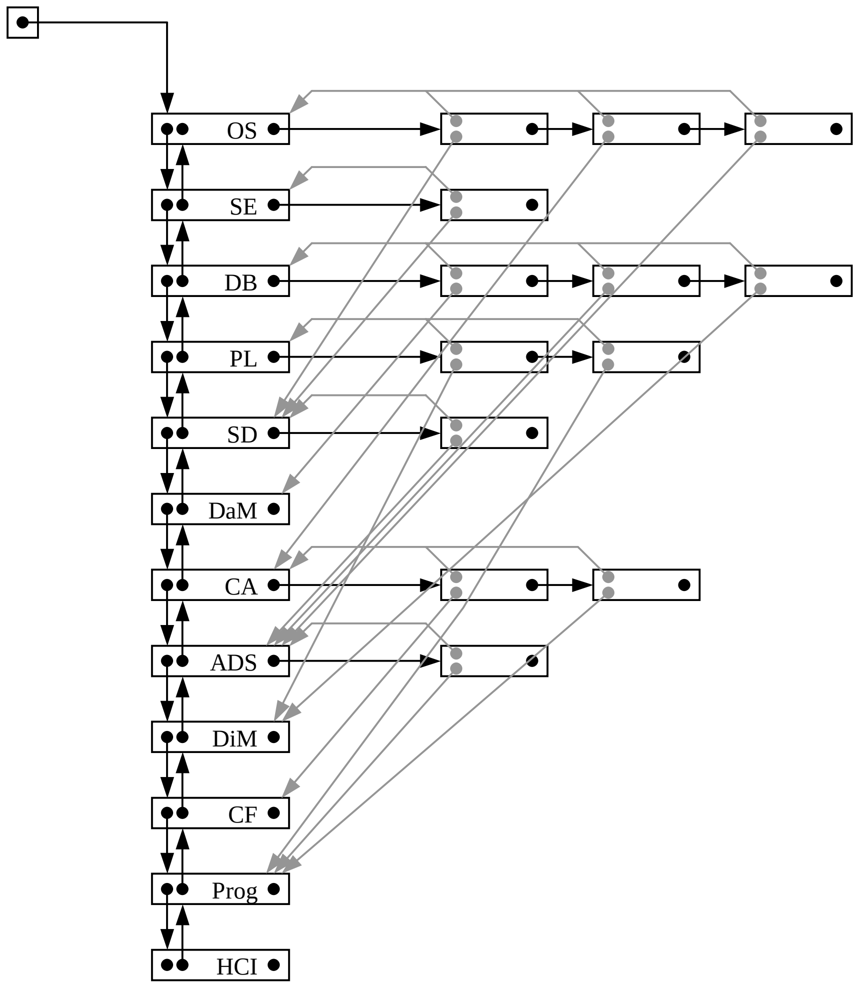

Figure S15.3 Adjacency-set representation of the directed graph of Figure 15.3.

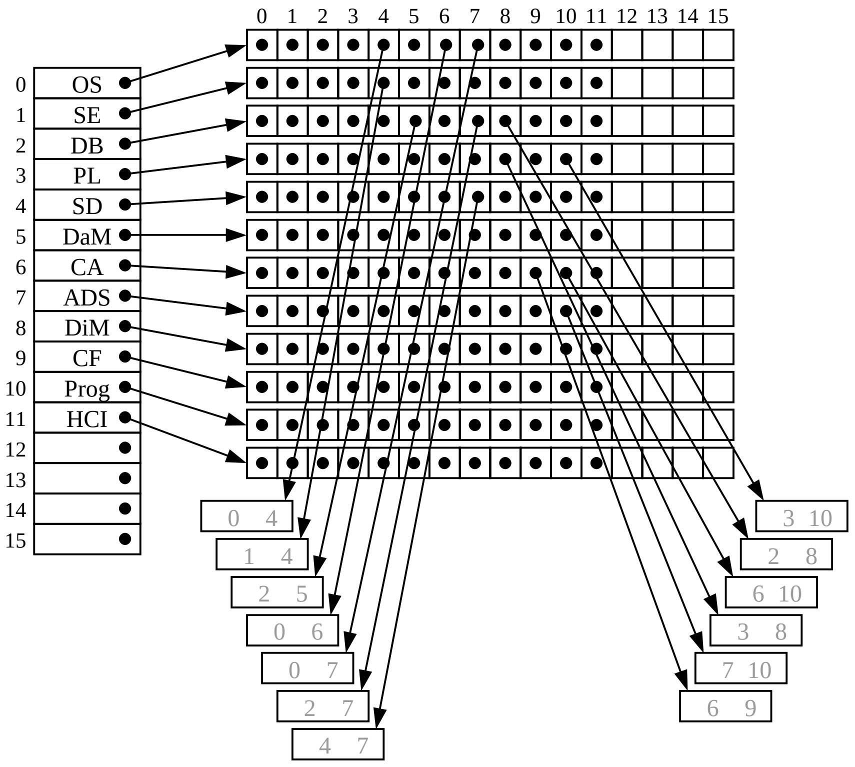

Figure S15.4 Adjacency-matrix representation of the directed graph of Figure 15.3 (with *m* = 16).
(Here, for the sake of clarity, the edge objects are shown as containing node position numbers. In actual fact they contain links to the corresponding node objects.)

Table S15.5 Edge-set representation of graphs with an SLL for the node set (where *e* is the number of edges).

Operation	Algorithm	Time complexity
`containsEdge`	linear search of edge-set DLL	$O(e)$
`addNode`	insertion at front of node-set DLL	$O(1)$
`addEdge`	insertion at front of edge-set SLL	$O(1)$
`removeNode`	deletion in node-set DLL, plus multiple deletions in edge-set SLL	$O(e)$
`removeEdge`	deletion in edge-set SLL	$O(e)$

Table S15.6 Edge-set representation of graphs with a hash table for the edge set (where *e* is the number of edges and *n* is the number of nodes).

Operation	Algorithm	Time complexity	
`containsEdge`	linear search of edge-set DLL	$O(e)$	
`addNode`	insertion in node-set hash table	$O(1)$ $O(n)$	best worst
`addEdge`	insertion at front of edge-set DLL	$O(1)$	
`removeNode`	deletion in node-set hash table, plus multiple deletions in edge-set DLL	$O(e)$ $O(n+e)$	best worst
`removeEdge`	deletion in edge-set DLL	$O(1)$	

15.7 In the adjacency-set representation of graphs, we can represent the adjacency sets by DLLs (rather than SLLs). The `removeEdge` operation then uses DLL deletion, which is faster. See Table S15.7.

Alternatively we can provide each node with an adjacency set for its in-edges (as well as one for its out-edges). However, we must continue to ensure that each edge is represented by a single `Edge` object. So we superimpose the in-edge SLLs on the out-edge SLLs, with each `Edge` object containing a link to the next in-edge as well as a link to the next out-edge. The `removeNode` operation must delete all in-edges and out-edges, which is tricky because each in-edge must also be deleted from the out-edge SLL that contains it, and vice versa. See Table S15.8.

Table S15.7 Adjacency-set representation of graphs with a DLL for each adjacency set (where *e* is the number of edges and *d* is the maximum degree of each node).

Operation	Algorithm	Time complexity
`containsEdge`	linear search of adjacency-set DLL	$O(d)$
`addNode`	insertion at front of node-set DLL	$O(1)$
`addEdge`	insertion at front of adjacency-set DLL	$O(1)$
`removeNode`	deletion in node-set DLL, plus traversal of all adjacency-set DLLs to find and delete connecting edges	$O(e)$
`removeEdge`	deletion in adjacency-set DLL	$O(1)$

Table S15.8 Adjacency-set representation of graphs with adjacency sets for both in-edges and out-edges (where *e* is the number of edges and *d* is the maximum degree of each node).

Operation	Algorithm	Time complexity
`containsEdge`	linear search of out-edges (or in-edges) SLL	$O(d)$
`addNode`	insertion at front of node-set DLL	$O(1)$
`addEdge`	insertion at front of in-edges and out-edges SLLs	$O(1)$
`removeNode`	deletion in node-set DLL, plus deletion of all in-edges and out-edges	$O(e)$
`removeEdge`	deletion in adjacency-set SLL	$O(d)$

15.8 Starting from the adjacency-matrix representation of a directed graph in Figure 15.16, the effect of removing node V is shown in Figure S15.9. The matrix is no longer compact: position numbers 0, 1, 2, 4, and 5 are used, but not position number 3.

An implementation modified to ensure that `removeNode` keeps the matrix compact is shown in Program S15.10. Whenever the node with position number *p* is removed, this implementation decrements the position numbers of all nodes with position numbers greater than *p*. The effect on time complexities of the graph operations is shown in Table S15.11. The `addNode` operation is now trivial and *O*(1), but the `removeNode` operation now entails shifting of both rows and columns in the matrix.

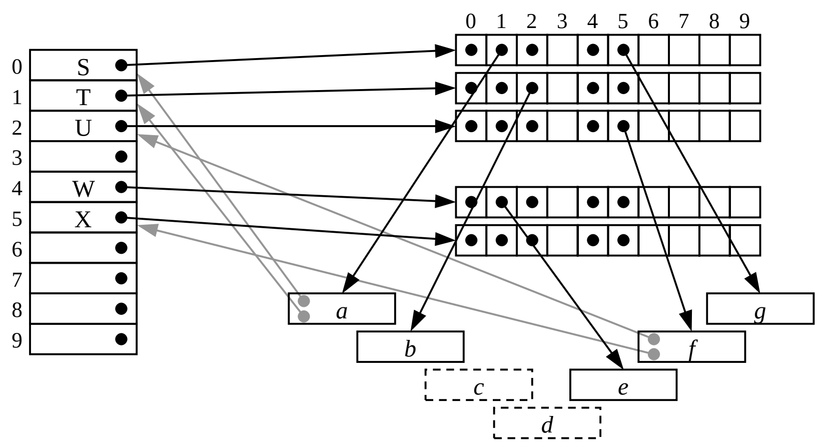

Figure S15.9 Effect of removing node V in the adjacency-matrix representation of the directed graph of Figure 15.10(a) ($m = 10$).

```
public class AMDigraph implements Digraph {

   // Each AMDigraph object is a directed graph whose elements and
   // edge attributes are arbitrary objects.

   // This directed graph is represented by an adjacency matrix as follows.
   // nodes is an array of AMDigraph.Node objects, each of which
   // contains an element, the node's position number, and a link to an array of
   // AMDigraph.Edge objects. The latter array represents the node's
   // possible out-edges (with null indicating the absence of a particular out-
   // edge). Each AMDigraph.Edge object contains an attribute and is
   // linked to the edge's source and destination nodes. size contains the
   // graph's size.

   private AMDigraph.Node[] nodes;
   private int size;

   public AMDigraph (int maxsize) {
   // Construct a directed graph, initially empty, whose size will be bounded
   // by maxsize.
      nodes = new AMDigraph.Node[maxsize];
      size = 0;
   }

   //////////// Accessors ////////////

   public boolean containsEdge (Graph.Node node0,
                  Graph.Node node1) {
   // Return true if and only if there is an edge connecting node0 to
   // node1 in this graph.
      AMDigraph.Node source = (AMDigraph.Node)node0,
           dest = (AMDigraph.Node)node1;
      int p = source.position, q = dest.position;
      return (nodes[p].outEdges[q] != null);
   }

   …
```

Program S15.10 Outline of adjacency-matrix representation of directed graphs, modified to keep the matrix compact *(continued on next page)*.

```
//////////// Transformers ////////////

public void clear () {
// Make this graph empty.
   for (int p = 0; p < size)  nodes[p] = null;
   size = 0;
}

public Graph.Node addNode (Object elem);
// Add to this graph a new node containing element elem, but with no
// connecting edges, and return the new node.
   int p = size++;
   nodes[p] = new AMDigraph.Node(elem, p);
}

public void removeNode (Graph.Node node) {
// Remove node from this graph, together with all its connecting edges.
   AMDigraph.Node nodeP = (AMDigraph.Node)node;
   int p = nodeP.position;
   size--;
   for (int q = p; q < size; q++) {
      AMDigraph.Node nodeQ = nodes[q+1];
      nodeQ.position--;
      nodes[q] = nodeQ;
   }
   for (int q = 0; q < size; q++) {
      AMDigraph.Node nodeQ = nodes[q];
      if (nodeQ.outEdges[p] != null)
         nodeQ.outDegree--;
      for (int r = p; r < size; r++)
         nodeQ.outEdges[r] =
               nodeQ.outEdges[r+1];
   }
}

…

//////////// Iterators ////////////

…

//////////// Inner class for graph nodes ////////////

private static class Node implements Graph.Node {

   // Each AMDigraph.Node object is a directed graph node, and
   // contains a single element.

   private Object element;
   private int outDegree;
   private AMDigraph.Edge[] outEdges;
   private int position;

   public Node (Object elem, int pos) {
      this.element = elem;  this.position = pos;
   }

   …   // For constructor and methods, see the companion Web site.

}

//////////// Inner class for graph edges ////////////

…

}
```

Program S15.10 Outline of adjacency-matrix representation of directed graphs, modified to keep the matrix compact *(continued)*.

Table S15.11 Adjacency-matrix representation of graphs, modified to keep the matrix compact: summary of algorithms.

Operation	Algorithm	Time complexity
`containsEdge`	matrix indexing	$O(1)$
`addNode`	trivial	$O(1)$
`addEdge`	matrix indexing	$O(1)$
`removeNode`	deleting a matrix row and column	$O(m^2)$
`removeEdge`	matrix indexing	$O(1)$

15.9 Depth-first and breadth-first graph *search* algorithms are shown as Algorithms S15.12 and S15.13.

```
To find which (if any) node of directed graph g contains an element equal to
target-elem, searching in depth-first order and starting at node start:

1. Make node-stack contain only node start, and mark start as reached.
2. While node-stack is not empty, repeat:
   2.1. Remove the top element of node-stack into v.
   2.2. If node v's element is equal to target-elem:
        2.2.1. Terminate with answer v.
   2.3. For each unreached successor w of node v, repeat:
        2.3.1. Add node w to node-stack, and mark w as reached.
3. Terminate with answer none.
```

Algorithm S15.12 Depth-first search algorithm for a directed graph.

```
To find which (if any) node of directed graph g contains an element equal to
target-elem, searching in breadth-first order and starting at node start:

1. Make node-queue contain only node start, and mark start as reached.
2. While node-queue is not empty, repeat:
   2.1. Remove the front element of node-queue into v.
   2.2. If node v's element is equal to target-elem:
        2.2.1. Terminate with answer v.
   2.3. For each unreached successor w of node v, repeat:
        2.3.1. Add node w to node-queue, and mark w as reached.
3. Terminate with answer none.
```

Algorithm S15.13 Breadth-first search algorithm for a directed graph.

15.10 Implementations of the graph traversal algorithms are shown as Programs S15.14 and S15.15. These implementations use the `java.util.LinkedList` representation of stacks and queues. They also use sets to record which nodes have been marked during the traversal.

```
static void traverseDepthFirst (Digraph g,
              Graph.Node start) {
  LinkedList nodeStack = new LinkedList();
  nodeStack.addLast(start);
  Set markedNodes = new HashSet();
  markedNodes.add(start);
  while (! nodeStack.isEmpty()) {
    Graph.Node v =
         (Graph.Node)nodeStack.removeLast();
    …  // Visit node v.
    Iterator successors = g.successors(v);
    while (successors.hasNext()) {
      Graph.Node w = (Graph.Node)successors.next();
      if (! markedNodes.contains(w)) {
        nodeStack.addLast(w);
        markedNodes.add(w);
      }
    }
  }
}
```

Program S15.14 Implementation of the depth-first graph traversal algorithm.

```
static void traverseBreadthFirst (Digraph g,
              Graph.Node start) {
  LinkedList nodeQueue = new LinkedList();
  nodeQueue.addLast(start);
  Set markedNodes = new HashSet();
  markedNodes.add(start);
  while (! nodeQueue.isEmpty()) {
    Graph.Node v =
         (Graph.Node)nodeQueue.removeFirst();
    …  // Visit node v.
    Iterator successors = g.successors(v);
    while (successors.hasNext()) {
      Graph.Node w = (Graph.Node)successors.next();
      if (! markedNodes.contains(w)) {
        nodeQueue.addLast(w);
        markedNodes.add(w);
      }
    }
  }
}
```

Program S15.15 Implementation of the breadth-first graph traversal algorithm.

15.12 Algorithm S15.16 determines whether there is a path between two given nodes in a directed graph, using a variant of the breadth-first graph search algorithm. (A variant of the depth-first graph search algorithm would also be suitable.) Program S15.17 shows an implementation.

```
To determine whether directed graph g contains a path from node start to node
finish:
1. Make node-queue contain only node start, and mark start as reached.
2. While node-queue is not empty, repeat:
   2.1. Remove the front element of node-queue into v.
   2.2. If v = finish:
        2.2.1. Terminate with answer true.
   2.3. For each unreached successor w of node v, repeat:
        2.3.1. Add node w to node-queue, and mark w as reached.
3. Terminate with answer false.
```

Algorithm S15.16 Path search algorithm for a directed graph.

```
static boolean containsPath (Digraph g,
            Graph.Node start, Graph.Node finish) {
  LinkedList nodeStack = new LinkedList();
  nodeStack.addLast(start);
  Set markedNodes = new HashSet();
  markedNodes.add(start);
  while (! nodeStack.isEmpty()) {
    Graph.Node v =
        (Graph.Node)nodeStack.removeLast();
    if (v == finish)  return true;
    Iterator successors = g.successors(v);
    while (successors.hasNext()) {
      Graph.Node w = (Graph.Node)successors.next();
      if (! markedNodes.contains(w)) {
        nodeStack.addLast(w);
        markedNodes.add(w);
      }
    }
  }
  return false;
}
```

Program S15.17 Implementation of the path search algorithm for a directed graph.

15.13 Program S15.18 computes the distance along the shortest path between a given node and every other node in a graph, where the distance is the number of edges along the path. The results are recorded in a map.

```
static Map findShortestPaths (Graph g,
                Graph.Node start) {
   LinkedList nodeQueue = new LinkedList();
   Set markedNodes = new HashSet();
   markedNodes.add(start);
   Map distMap = new HashMap();
   distMap.put(start, new Integer(0));
   while (! nodeQueue.isEmpty()) {
      Graph.Node v =
            (Graph.Node)nodeQueue.removeFirst();
      markedNodes.add(v);
      int distV = distance(distMap, v);
      Iterator neighbors = g.neighbors(v);
      while (neighbors.hasNext()) {
         Graph.Node w = (Graph.Node)neighbors.next();
         if (! markedNodes.contains(w)) {
            int d = distV + 1;
            int distW = distance(distMap, w);
            if (d < distW)  distMap.put(w, d);
            nodeQueue.addLast(w);
         }
      }
   }
   return distMap;
}

static int distance (Map distMap, Graph.Node v) {
   Integer distValue = (Integer)distMap.get(v);
   if (distValue == null)
      return INFINITY;
   else
      return distValue.intValue();
}

static final int INFINITY = 1000000000;
```

Program S15.18 Implementation of the shortest-path algorithm for a undirected graph.

15.14 Algorithm S15.19 computes the distance along the shortest path between a given node and every other node in a graph, where the distance is the sum of the (positive-integer) edge attributes of edges along the path. The implementation is shown as Program S15.20.

To find the shortest path in graph *g* from node *start* to every other node:

```
1. Make node-queue contain only node start.
2. Set $dist_{start}$ to 0, and set $dist_v$ for all other nodes v to infinity.
3. While node-queue is not empty, repeat:
   3.1. Remove the front element of node-queue into v, and mark node v as
        reached.
   3.2. For each edge e connecting node v to an unreached neighbor w, repeat:
        3.2.1. Let d be $dist_v$ + edge attribute of e.
        3.2.2. If $d < dist_w$, set $dist_w$ to d.
        3.2.3. Add node w to node-queue.
4. Terminate.
```

Algorithm S15.19 Shortest-path algorithm for an undirected graph with positive-integer edge attributes.

```java
static Map findShortestPaths (Graph g,
            Graph.Node start) {
  LinkedList nodeQueue = new LinkedList();
  Set markedNodes = new HashSet();
  markedNodes.add(start);
  Map distMap = new HashMap();
  distMap.put(start, new Integer(0));
  while (! nodeQueue.isEmpty()) {
    Graph.Node v =
        (Graph.Node)nodeQueue.removeFirst();
    markedNodes.add(v);
    int distV = distance(distMap, v);
    Iterator edges = g.connectingEdges(v);
    while (edges.hasNext()) {
      Graph.Edge e = (Graph.Edge)edges.next();
      Graph.Node vw = e.getNodes();
      Graph.Node w = (vw[0] == v ? vw[1] : vw[0]);
      if (! markedNodes.contains(w)) {
        int d = distV +
            ((Integer)e.getAttribute()).getValue();
        int distW = distance(distMap, w);
        if (d < distW)  distMap.put(w, d);
        nodeQueue.addLast(w);
      }
    }
  }
  return distMap;
}
```

Program S15.20 Implementation of the shortest-path algorithm for a undirected graph with positive-integer edge attributes (changes from Program S15.18 italicized).

15.16 If the directed graph *g* is cyclic, the topological sort algorithm will produce an incomplete list of nodes. In particular, a node *v* that participates in a cycle will never be added to the list because in_v will never decrease to zero.

To make the topological sort algorithm deal with a cyclic graph, simply insert a new step 5 as shown in Algorithm S15.21.

To make the algorithm work without *node-queue*, first observe that *node-list* contains nodes v for which $in_v = 0$ and whose out-edges have been processed, whereas *node-queue* contains nodes v for which $in_v = 0$ but whose out-edges have not yet been processed; all nodes in *node-queue* will eventually be removed and added to *node-list* in the same order. As an alternative, we can make *node-list* contain *all* nodes v for which $in_v = 0$, on the understanding that only the first p nodes in node-list have had their out-edges processed. This is the basis of Algorithm S15.21.

```
To find a topological ordering of directed acyclic graph g:
1. Make node-list empty.
2. Set p to 0.
3. For each node v of g, repeat:
   3.1. Set $in_v$ to the in-degree of node v.
   3.2. If $in_v = 0$, add node v to node-list.
4. While p < length of node-list, repeat:
   4.1. Let v be the element of node-list with index p.
   4.2. Increment p.
   4.3. For each successor w of node v, repeat:
        4.3.1. Decrement $in_w$.
        4.3.2. If $in_w = 0$, add node w to node-list.
5. If length of node-list < size of graph g:
   5.1. Terminate with a warning that g is cyclic.
6. Terminate with answer node-list.
```

Algorithm S15.21 Topological sort algorithm for a directed acyclic graph (with a warning if the graph is cyclic).

15.17 An implementation of the topological sort algorithm (Algorithm 15.25) in Java is shown as Program S15.22.

```
static List topologicalSort (Digraph g) {
  LinkedList nodeList = new LinkedList();
  LinkedList nodeQueue = new LinkedList();
  Map inMap = new HashMap();
  Iterator nodes = g.nodes();
  while (nodes.hasNext()) {
    Graph.Node v = (Graph.Node)nodes.next();
    int inV = g.degree(v) - g.outDegree(v);
    inMap.put(v, new Integer(inV));
    if (inV == 0)  nodeQueue.add(v);
  }
  while (! nodeQueue.isEmpty()) {
    Graph.Node v =
        (Graph.Node)nodeQueue.removeFirst();
    nodeList.add(v);
    Iterator successors = g.successors(v);
    while (successors.hasNext()) {
      Graph.Node w = (Graph.Node)successors.next();
      int inW = ((Integer)inMap.get(v)).getValue();
      inW--;
      inMap.put(w, new Integer(inW));
      if (inW == 0)  nodeQueue.add(w);
    }
  }
  return nodeList;
}
```

Program S15.22 Implementation of the topological sort algorithm.

15.18 Program S15.23 shows an implementation of a depth-first iterator for directed graphs. It is expressed entirely in terms of the other directed-graph operations, and so could be added to `ESDigraph`, `ASDigraph`, or `AMDigraph`.

The `DepthFirstIterator` constructor is the same as Program S15.14, except that it adds nodes to a queue, `track`, rather than visiting them. The `next` operation simply removes a node from the front of the queue.

```
public Iterator depthFirstIterator (Graph.Node node) {
// Return an iterator that will visit all nodes of this graph that are
// reachable from node, in a depth-first traversal.
  return new DepthFirstIterator(node);
}

//////////// Inner class for depth-first iterators ////////////

private class DepthFirstIterator implements Iterator {

  // A DepthFirstIterator object is an iterator that will visit, in
  // depth-first order, all the nodes reachable from a given node in a graph.

  // This iterator is represented by a queue of nodes still to be visited,
  // track.
  private LinkedList track;

  private DepthFirstIterator (Graph.Node start) {
    track = new LinkedStack();
    LinkedList nodeStack = new LinkedList();
    nodeStack.addLast(start);
    Set markedNodes = new HashSet();
    markedNodes.add(start);
    while (! nodeStack.isEmpty()) {
      Graph.Node v =
          (Graph.Node)nodeStack.removeLast();
      track.addLast(v);  // Remember to visit node v.
      Iterator successors = g.successors(v);
      while (successors.hasNext()) {
        Graph.Node w =
            (Graph.Node)successors.next();
        if (! markedNodes.contains(w)) {
          nodeStack.addLast(w);
          markedNodes.add(w);
        }
      }
    }
  }

  public boolean hasNext () {
    return (! track.isEmpty());
  }

  public Object next () {
    if (track.isEmpty())
      throw new NoSuchElementException();
    Graph.Node node =
        (Graph.Node)track.removeFirst();
    return node.getElement();
  }

  …  // The remove method is omitted here.

}
```

Program S15.23 Depth-first iterator for graphs (as an inner class).

15.19 Program S15.24 shows an implementation of a breadth-first iterator for directed graphs. It is expressed entirely in terms of the other directed-graph operations, and so could be added to `ESDigraph`, `ASDigraph`, or `AMDigraph`.

The `BreadthFirstIterator` constructor is the same as Program S15.15, except that it adds nodes to a queue, `track`, rather than visiting them. The `next` operation simply removes a node from the front of the queue.

```
public Iterator breadthFirstIterator
            (Graph.Node node) {
// Return an iterator that will visit all nodes of this graph that are
// reachable from node, in a breadth-first traversal.
  return new BreadthFirstIterator(node);
}

//////////// Inner class for breadth-first iterators ////////////

private class BreadthFirstIterator
            implements Iterator {

  // A BreadthFirstIterator object is an iterator that will visit, in
  // breadth-first order, all the nodes reachable from a given node in a graph.

  // This iterator is represented by a queue of nodes still to be visited,
  // track.
  private LinkedList track;

  private BreadthFirstIterator (Graph.Node start) {
    track = new LinkedStack();
    LinkedList nodeQueue = new LinkedList();
    nodeQueue.addLast(start);
    Set markedNodes = new HashSet();
    markedNodes.add(start);
    while (! nodeQueue.isEmpty()) {
      Graph.Node v =
          (Graph.Node)nodeQueue.removeFirst();
      track.addLast(v);  // Remember to visit node v.
      Iterator successors = g.successors(v);
      while (successors.hasNext()) {
        Graph.Node w =
            (Graph.Node)successors.next();
        if (! markedNodes.contains(w)) {
          nodeQueue.addLast(w);
          markedNodes.add(w);
        }
      }
    }
  }

  public boolean hasNext () {
    return (! track.isEmpty());
  }

  public Object next () {
    if (track.isEmpty())
      throw new NoSuchElementException();
    Graph.Node node =
        (Graph.Node)track.removeFirst();
    return node.getElement();
  }

  …  // The remove method is omitted here.

}
```

Program S15.24 Breadth-first iterator for graphs (as an inner class).