Solutions to Exercises 13

13A The class hierarchy of a Java program, reflecting the subclass relationship between classes, can be represented by a tree.

(a) The root vertex of the class hierarchy tree corresponds to the `Object` class.

(b) The class hierarchy is a tree because Java enforces single inheritance, i.e., each class (except `Object`) has exactly one superclass.

(d) If Java interfaces are included, the 'hierarchy' is no longer a tree because a class may implement any number of interfaces.

13B Here is an outline of an implementation of ordered trees:

```
public class LinkedOrderedTree<E> implements Tree<E> {

    // Each LinkedOrderedTree object is an ordered tree whose
    // elements are of type E.

    // This tree is represented by a reference to its root vertex (root), which is
    // null if the tree is empty. Each vertex contains links to its first and last
    // children, to its parent, and to its next sibling.
    private MyVertex root;

    //////////// Constructor ////////////

    public LinkedOrderedTree () {
    // Construct a tree, initially empty.
        root = null;
    }

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

    …

    //////////// Transformers ////////////

    public void makeRoot (E elem) {
    // Make this tree consist of just a root vertex containing element elem.
        root = new MyVertex(elem);
    }

    public Tree.Vertex addChild (Tree.Vertex v,
                    E elem) {
    // Add a new vertex containing element elem as the last child of v in
    // this tree, and return the new vertex. The new vertex has no children of its
    // own.
        MyVertex parent = (MyVertex)v;
        MyVertex newChild = new MyVertex(elem);
        newChild.parent = parent;
        if (parent.firstChild == null)
            parent.firstChild = newChild;
        else
            parent.lastChild.nextSib = newChild;
        parent.lastChild = newChild;
        return newChild;
    }

    public void remove (Tree.Vertex v) {
    // Remove v from this tree, together with all its descendants.
        if (v == root) {
            root = null;
            return;
        }
        MyVertex parent = v.parent;
        if (v == parent.firstChild) {
            parent.firstChild = v.nextSib;
            if (parent.firstChild == null)
                parent.lastChild = null;
        } else {
            MyVertex prevSib = parent.firstChild;
            while (prevSib.nextSib != v)
                prevSib = prevSib.nextSib;
            prevSib.nextSib = v.nextSib;
            if (prevSib.nextSib == null)
                parent.lastChild = prevSib;
        }
```

```
    }

    //////////// Iterator ////////////

    …

    //////////// Inner class ////////////

    private static class MyVertex
                  implements Tree.Vertex {

        // Each MyVertex object is a vertex of an ordered tree,
        // and contains a single element.

        // This vertex consists of an element (element), a link to its first
        // and last children (firstChild, lastChild) a link to its parent
        // (parent), and a link to its next sibling (nextSib).
        private E element;
        private MyVertex firstChild, lastChild,
             parent, nextSib;

        …

    }
}
```

13C The following methods visit, in pre-order, all of the vertices in a given tree:

```
static void traversePreorder (Tree<E> tree) {
  if (tree.root() != null)
    traverseSubtreePreorder(tree, tree.root());
}

static void traverseSubtreePreorder (
              Tree<E> tree,
              Tree<E>.Vertex top) {
  ...  // Visit top.
  Iterator<Tree<E>.Vertex> children =
      tree.children(top);
  while (children.hasNext()) {
    Tree<E>.Vertex child = children.next();
    traverseSubtreePreorder(tree, child);
  }
}
```

13E To visit the vertices of *tree* in depth order:

1. Make *vertex-queue* contain only the root vertex of *tree*.
2. While *vertex-queue* is non-empty, repeat:
   2.1. Remove the front element of *vertex-queue* into *v*.
   2.2. Visit *v*.
   2.3. Add all the children of *v* to the rear of *vertex-queue*.
3. Terminate.

Implementation (using the `java.util.LinkedList` representation of a queue):

```
static void traverseDepthOrder (Tree<E> tree) {
  Queue<Tree<E>.Vertex> vertexQueue =
      new LinkedList<Tree<E>.Vertex>();
  vertexQueue.addLast(tree.root());
  while (! vertexQueue.isEmpty()) {
    Tree<E>.Vertex v =
        vertexQueue.removeFirst();
    ...  // Visit v.
    Iterator<Tree<E>.Vertex> children =
        tree.children(v);
    while (children.hasNext()) {
      Tree.Vertex child = children.next();
      nodeQueue.addLast(child);
    }
  }
}
```

13F Here is an outline of an implementation of unordered trees using arrays:

```
public class ArrayUnorderedTree<E>
    implements Tree<E> {

  // Each ArrayUnorderedTree<E> object is an unordered tree whose
  // elements are of type E.

  // This tree is represented by a reference to its root vertex (root), which is
  // null if the tree is empty. Each tree vertex contains an array of children.
  private MyVertex root;

  //////////// Constructor ////////////

  public ArrayUnorderedTree () {
  // Construct a tree, initially empty.
    root = null;
  }

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

  public Tree.Vertex root () {
  // Return the root vertex of this tree, or null if this tree is empty.
    return root;
  }

  public Tree.Vertex parent (Tree<E>.Vertex v) {
  // Return the parent of v in this tree, or null if v is the root vertex.
    return v.parent;
  }

  public int childCount (Tree<E>.Vertex v) {
  // Return the number of children of v in this tree.
    MyVertex parent = (MyVertex)v;
    return parent.childCount;
  }

  //////////// Transformers ////////////

  public void makeRoot (E elem) {
  // Make this tree consist of just a root vertex containing element elem.
    root = new MyVertex(elem);
  }

  public Tree.Vertex addChild (Tree<E>.Vertex v,
              E elem) {
  // Add a new vertex containing element elem as a child of v in this
  // tree, and return the new vertex. The new vertex has no children of its
  // own.
    MyVertex parent = (MyVertex)v;
    MyVertex newChild = new MyVertex(elem);
    newChild.parent = parent;
    if (parent.childCount == parent.children.length)
      parent.expand();
    parent.children[parent.childCount++] = newChild;
    return newChild;
  }

  public void remove (Tree<E>.Vertex v) {
  // Remove v from this tree, together with all its descendants.
    if (v == root) {
      root = null;
      return;
    }
    MyVertex parent = v.parent;
    parent.childCount--;
```

```
      int i = 0;
      while (parent.children[i] != v)  i++;
      while (i < parent.childCount) {
         parent.children[i] = parent.children[i+1];
         i++;
      }
   }

   //////////// Iterator ////////////

   …

   //////////// Inner class ////////////

   private static class MyVertex
                  implements Tree<E>.Vertex {

      // Each MyVertex object is a vertex of an unordered tree,
      // and contains a single element.

      // This tree vertex consists of an element (element), a link to its
      // parent (parent), an array of links to its children (children), and
      // the number of children (childCount).
      private E element;
      private MyVertex parent;
      private MyVertex[] children;
      private int childCount;

      private MyVertex (E elem) {
      // Construct a tree vertex, containing element elem, that has no parent
      // and no children.
         this.element = elem;
         this.parent = null;
         this.children = new MyVertex[4];
         this.childCount = 0;
      }

      …

      public void expand () {
      // Increase the length of this vertex's array of links to children.
         …
      }

   }
}
```

The `addChild` operations has time complexity $O(1)$. If c is the maximum number of children per vertex, the `remove` operation has time complexity $O(c)$.

13G In the linked (or array) implementation of an unordered tree, the explicit reference to a vertex's parent could be removed, but the `parent` operation must then search the tree to find the vertex's parent. This search can be done by a preorder traversal, terminating when the parent is found:

```
public Tree<E>.Vertex parent (Tree<E>.Vertex v) {
// Return the parent of v in this tree, or null if v is the root
// vertex.
   if (root == v)
      return null;
   else
      return findParent(v, root);
}

private Tree<E>.Vertex findParent (
               Tree<E>.Vertex v,
               Tree<E>.Vertex ancestor) {
// Return the parent of v in this tree, assuming that ancestor
// is a parent or grandparent or … of v.
   Iterator<Tree<E>.Vertex> children =
        children(ancestor);
   while (children.hasNext()) {
      Tree<E>.Vertex child = children.next();
      if (child == v)  return ancestor;
      Tree<E>.Vertex parent =
           findParent(v, child);
      if (parent != null)  return parent;
   }
   return null;
}
```

The `parent` operation now has time complexity $O(n)$, as does any other operation that must call the `parent` operation.

Solutions to Exercises 14

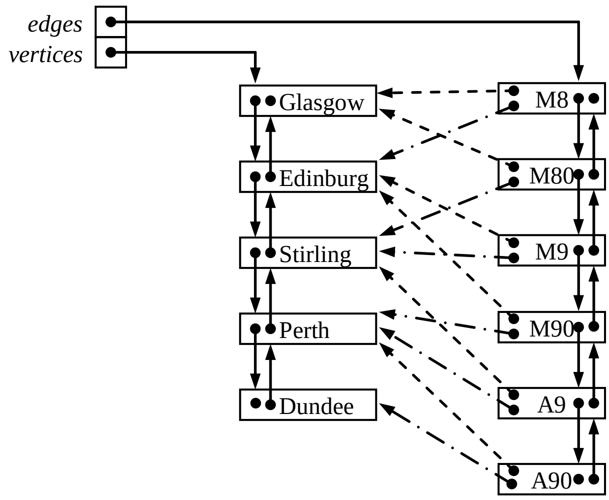
14A Paths between Glasgow and Perth:

«Glasgow, Stirling, Perth»

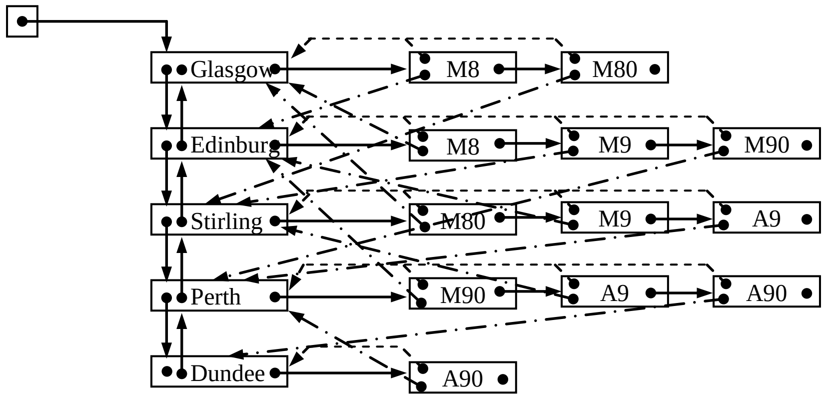
« Glasgow, Edinburgh, Perth »

« Glasgow, Edinburgh, Stirling, Perth »

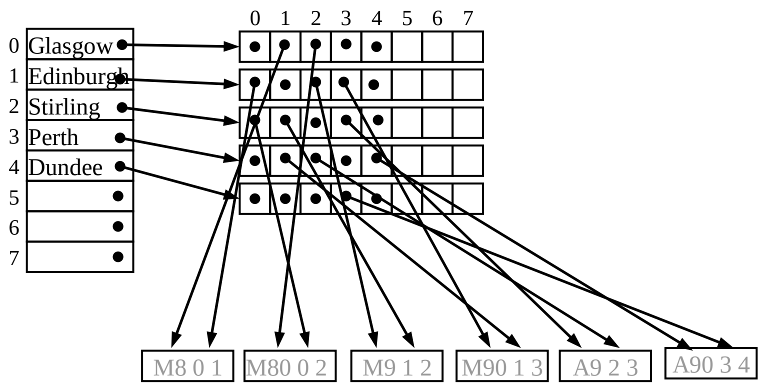
14B Edge-set representation of the Scottish road network:



Adjacency-set representation of the Scottish road network (noting that edge nodes are duplicated):



Adjacency-matrix representation of the Scottish road network (with m = 8) (noting that two matrix cells point to each edge node):



(Here, for the sake of clarity, the edge objects are shown as containing vertex position numbers. In actual fact they contain links to the corresponding vertex objects.)

14C In the edge-set representation of graphs, we can represent the vertex set by a SLL (rather than a DLL). The `removeEdge` operation then has to use the SLL deletion algorithm, which is $O(n_e)$, where n_e is the number of edges. The following table summarizes the algorithms and their time complexities:

Operation	Algorithm	Time complexity
`containsEdge`	linear search of edge-set DLL	$O(n_e)$
`addVertex`	insertion at front of vertex-set DLL	$O(1)$
`addEdge`	insertion at front of edge-set SLL	$O(1)$
`removeVertex`	deletion in vertex-set DLL, plus multiple deletions in edge-set SLL	$O(n_e)$
`removeEdge`	deletion in edge-set SLL	$O(n_e)$

Alternatively we can represent the vertex set by a hash-table with elements as keys (rather than by a DLL). The `addVertex` and `removeVertex` operations then use (more or less) the standard hash-table insertion and deletion algorithms. The following table summarizes the algorithms and their time complexities, where n is the number of vertices:

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

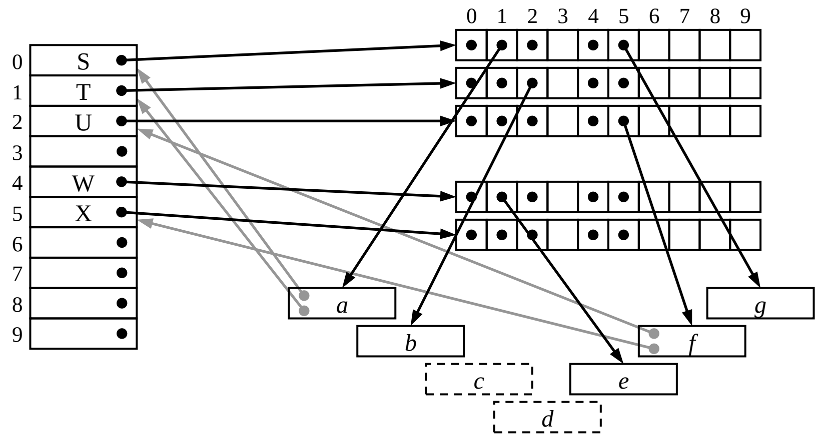
14D In the adjacency-set representation of graphs, we can represent the adjacency sets by DLLs (rather than SLLs). The `removeEdge` operation then uses DLL deletion, which is faster. The following table summarizes the algorithms and their time complexities, where n_e is the number of edges and d is the maximum degree of any vertex:

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

Alternatively we can provide each vertex 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 `removeVertex` 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. The following table summarizes the algorithms and their time complexities:

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

14E Starting from the adjacency-matrix representation of a directed graph in the course notes, the effect of removing vertex V is shown below. The matrix is no longer compact: position numbers 0, 1, 2, 4, and 5 are used, but not position number 3.



To keep the matrix compact, the implementation would have to be modified as follows. Whenever the vertex with position number p is removed, decrement the position numbers of vertices $p+1 \ldots n$ (e.g., vertices W and X above). Shift these vertices up by one row, and shift the corresponding columns in the matrix left by one column. Also adjust every edge whose source's and/or destination's position number has changed.

The effect on time complexities of the graph operations is shown in the table below. The `addVertex` operation is now trivial and $O(1)$, but the `removeVertex` operation now entails shifting of both rows and columns in the matrix.

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

14F Depth-first and breadth-first graph *search* algorithms:

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

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

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

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

14G Implementations of the graph traversal algorithms are shown below. These implementations use sets to record which vertices have been marked during the traversal.

```
static void traverseDepthFirst (Digraph<E,A> g,
              Graph<E,A>.Vertex start) {
  Stack<Graph<E,A>.Vertex> vertexStack =
      new Stack<Graph<E,A>.Vertex>();
  vertexStack.addLast(start);
  Set<Graph<E,A>.Vertex> marked =
      new HashSet<Graph<E,A>.Vertex>();
  marked.add(start);
  while (! vertexStack.empty()) {
    Graph<E,A>.Vertex v = vertexStack.pop();
    …  // Visit vertex v.
    Iterator<Graph<E,A>.Vertex> successors =
        g.successors(v);
    while (successors.hasNext()) {
      Graph<E,A>.Vertex w = successors.next();
      if (! marked.contains(w)) {
        vertexStack.push(w);
        marked.add(w);
      }
    }
  }
}

static void traverseBreadthFirst (Digraph<E,A> g,
              Graph<E,A>.Vertex start) {
  Queue<Graph<E,A>.Vertex> vertexQueue =
      new LinkedList<Graph<E,A>.Vertex>();
  vertexQueue.addLast(start);
  Set<Graph<E,A>.Vertex> marked =
      new HashSet<Graph<E,A>.Vertex>();
  marked.add(start);
  while (! vertexQueue.isEmpty()) {
    Graph<E,A>.Vertex v = vertexQueue.removeFirst();
    …  // Visit vertex v.
    Iterator<Graph<E,A>.Vertex> successors =
        g.successors(v);
    while (successors.hasNext()) {
      Graph<E,A>.Vertex w = successors.next();
      if (! marked.contains(w)) {
        vertexQueue.addLast(w);
        marked.add(w);
      }
    }
  }
}
```

14H The following algorithm determines whether there is a path between two given vertices 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.)

To determine whether directed graph *g* contains a path from vertex *start* to vertex *finish*:

1. Make *vertex-queue* contain only vertex *start*, and mark *start* as reached.
2. While *vertex-queue* is not empty, repeat:
   2.1. Remove the front element of *vertex-queue* into *v*.
   2.2. If *v* = *finish*:
      2.2.1. Terminate with answer true.
   2.3. For each unreached successor *w* of vertex *v*, repeat:
      2.3.1. Add *w* to *vertex-queue*, and mark *w* as reached.
3. Terminate with answer false.

Here is a possible implementation:

```
static boolean containsPath (Digraph<E,A> g,
            Graph<E,A>.Vertex start,
            Graph<E,A>.Vertex finish) {
  Stack<Graph<E,A>.Vertex> vertexStack =
      new Stack<Graph<E,A>.Vertex>();
  vertexStack.push(start);
  Set<Graph<E,A>.Vertex> marked =
      new HashSet<Graph<E,A>.Vertex>();
  marked.add(start);
  while (! vertexStack.empty()) {
    Graph<E,A>.Vertex v = vertexStack.pop();
    if (v == finish)  return true;
    Iterator<Graph<E,A>.Vertex> successors =
        g.successors(v);
    while (successors.hasNext()) {
      Graph<E,A>.Vertex w = successors.next();
      if (! marked.contains(w)) {
        vertexStack.push(w);
        marked.add(w);
      }
    }
  }
  return false;
}
```

14J Shortest path from Ed(inburgh) to Du(ndee):

Du	Ed	Gl	Pe	St	places
none,∞	none,0	none,∞	none,∞	none,∞	{Du,Ed,Gl,Pe,St}
none,∞	none,0	Ed,70	Ed,100	Ed,50	{Du,Gl,Pe,St}
none,∞	none,0	Ed,70	St,90	Ed,50	{Du,Gl,Pe}
none,∞	none,0	Ed,70	St,90	Ed,50	{Du,Pe}
Pe,160	none,0	Ed,70	St,90	Ed,50	{Du}

Shortest path is «Ed,St,Pe,Du».