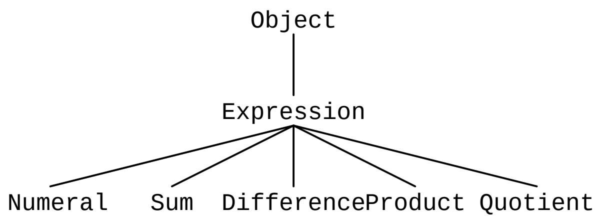
Solutions to Exercises in Chapter 14

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

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

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

(c) The class hierarchy for Program 14.12 is:



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

14.5 An implementation of ordered trees is outlined in Program S14.1.

```
public class LinkedOrderedTree implements Tree {

   // Each LinkedOrderedTree object is an ordered tree whose
   // elements are arbitrary objects.

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

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

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

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

   …

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

   public void makeRoot (Object elem) {
   // Make this tree consist of just a root node containing element elem.
      root = new LinkedOrderedTree.Node(elem);
   }
```

Program S14.1 Outline implementation of ordered trees using linked data structures

(continued on next page).

```
   public Tree.Node addChild (Tree.Node node,
                 Object elem) {
   // Add a new node containing element elem as the last child of node in
   // this tree, and return the new node. The new node has no children of its
   // own.
      LinkedOrderedTree.Node parent =
           (LinkedOrderedTree.Node)node;
      LinkedOrderedTree.Node newChild =
           new LinkedOrderedTree.Node(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.Node node) {
   // Remove node from this tree, together with all its descendants.
      if (node == root) {
         root = null;
         return;
      }
      LinkedOrderedTree.Node parent = node.parent;
      if (node == parent.firstChild) {
         parent.firstChild = node.nextSib;
         if (parent.firstChild == null)
            parent.lastChild = null;
      } else {
         LinkedOrderedTree.Node prevSib =
              parent.firstChild;
         while (prevSib.nextSib != node)
            prevSib = prevSib.nextSib;
         prevSib.nextSib = node.nextSib;
         if (prevSib.nextSib == null)
            parent.lastChild = prevSib;
      }
   }

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

   …

   //////////// Inner class definition for tree nodes ////////////

   private static class Node implements Tree.Node {

      // Each LinkedOrderedTree.Node object is a node of an
      // ordered tree, and contains a single element.

      // This tree node 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 Object element;
      private LinkedOrderedTree.Node firstChild,
           lastChild, parent, nextSib;

      …

   }
}
```

Program S14.1 Outline implementation of ordered trees using linked data structures *(continued)*.

14.6 The following methods visit, in pre-order, all of the nodes in a given tree:

```
static void preOrderTraverse (Tree tree) {
  if (tree.root() != null)
    preOrderTraverseSubtree(tree, tree.root());
}

static void preOrderTraverseSubtree (Tree tree,
            Tree.Node parent) {
  …  // Visit parent.
  Iterator children = tree.children(parent);
  while (children.hasNext()) {
    Tree.Node child =
        (Tree.Node)children.next();
    preOrderTraverseSubtree(tree, child);
  }
}
```

14.8 Methods to visit, in *post*-order, all of the nodes in a given tree would be similar to the methods of Exercise 14.6, except that the code to visit `parent` must *follow* the while-loop that traverses the children.

14.10 To visit the nodes of *tree* in depth order:

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

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

```
static void depthOrderTraverse (Tree tree) {
  LinkedList nodeQueue = new LinkedList();
  nodeQueue.addLast(tree.root());
  while (! nodeQueue.isEmpty()) {
    Tree.Node node =
        (Tree.Node)nodeQueue.removeFirst();
    …  // Visit node.
    Iterator children = tree.children(node);
    while (children.hasNext()) {
      Tree.Node child =
          (Tree.Node)children.next();
      nodeQueue.addLast(child);
    }
  }
}
```

14.11 An implementation of unordered trees using arrays is outlined in Program S14.2.

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

```
public class ArrayUnorderedTree implements Tree {

   // Each ArrayUnorderedTree object is an unordered tree whose
   // elements are arbitrary objects.

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

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

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

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

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

   public Tree.Node parent (Tree.Node node) {
   // Return the parent of node in this tree, or null if node is the root node.
      return node.parent;
   }

   public int childCount (Tree.Node node) {
   // Return the number of children of node in this tree.
      ArrayUnorderedTree.Node parent =
            (ArrayUnorderedTree.Node)node;
      return parent.childCount;
   }

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

   public void makeRoot (Object elem) {
   // Make this tree consist of just a root node containing element elem.
      root = new ArrayUnorderedTree.Node(elem);
   }

   public Tree.Node addChild (Tree.Node node,
                  Object elem) {
   // Add a new node containing element elem as a child of node in this
   // tree, and return the new node. The new node has no children of its own.
      ArrayUnorderedTree.Node parent =
            (ArrayUnorderedTree.Node)node;
      ArrayUnorderedTree.Node newChild =
            new ArrayUnorderedTree.Node(elem);
      newChild.parent = parent;
      if (parent.childCount == parent.children.length)
         parent.expand();
      parent.children[parent.childCount++] = newChild;
      return newChild;
   }
```

Program S14.2 Outline implementation of unordered trees using arrays
(continued on next page).

```
public void remove (Tree.Node node) {
// Remove node from this tree, together with all its descendants.
   if (node == root) {
      root = null;
      return;
   }
   ArrayUnorderedTree.Node parent = node.parent;
   parent.childCount--;
   int i = 0;
   while (parent.children[i] != node)  i++;
   while (i < parent.childCount) {
      parent.children[i] = parent.children[i+1];
      i++;
   }
}

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

…

//////////// Inner class definition for tree nodes ////////////

private static class Node implements Tree.Node {

   // Each ArrayUnorderedTree.Node object is a node of an
   // unordered tree, and contains a single element.

   // This tree node 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 Object element;
   private ArrayUnorderedTree.Node parent;
   private ArrayUnorderedTree.Node[] children;
   private int childCount;

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

   …

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

}
}
```

Program S14.2 Outline implementation of unordered trees using arrays *(continued)*.

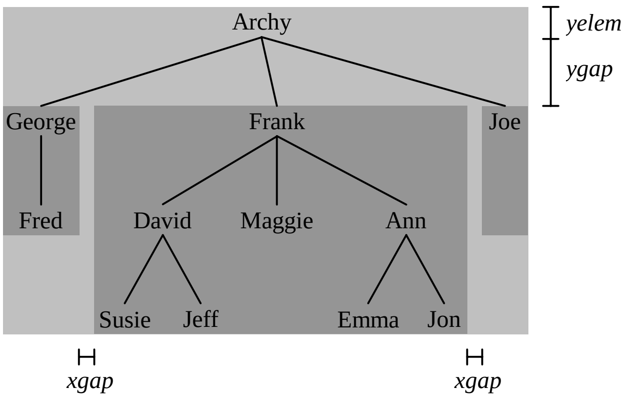
14.14 In the linked (or array) implementation of an unordered tree, the explicit reference to a node's parent could be removed, but the `parent` operation must then search the tree to find the node's parent. This search can be done by a pre-order traversal, terminating when the parent is found:

```
public Tree.Node parent (Tree.Node node) {
// Return the parent of node in this tree, or null if node is the root
// node.
  if (root == node)
    return null;
  else
    return findParent(node, root);
}

private Tree.Node findParent (
              Tree.Node node,
              Tree.Node ancestor) {
// Return the parent of node in this tree, assuming that ancestor
// is a parent or grandparent or … of node.
  Iterator children = tree.children(ancestor);
  while (children.hasNext()) {
    Tree.Node child =
        (Tree.Node)children.next();
    if (child == node)  return ancestor;
    Tree.Node parent =
        findParent(node, 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.

14.20 In the drawing of a tree, let each subtree's *bounding rectangle* be the smallest rectangle that encloses all that subtree's nodes. The following example shows a family tree and some of the bounding rectangles:



Here is one simple idea for drawing a tree. Consider a node *N* and its subtrees. Place the subtrees' bounding rectangles side by side, with their tops aligned, leaving a small gap (*xgap* above) between neighboring rectangles. Draw node *N*'s element centered above these rectangles, leaving a small gap (*ygap* above). Then the bounding rectangle for the tree whose top node is *N* is the smallest rectangle that encloses node *N*'s element and all the subtrees' rectangles.

To draw *tree*:

1. Draw the subtree whose topmost node is *tree*'s root, with the top left of its bounding rectangle at (0, 0).
2. Terminate.

To draw the subtree whose topmost node is *N*, with the top left of its bounding rectangle at (*x*, *y*):

1. Let *c* be the number of children of node *N*.
2. If *c* = 0:
   2.1. Set *width* to the width of node *N*'s element when drawn.
   3.4. Set *xtop* to *x*+*width*/2.
   2.2. Draw node *N*'s element centered at (*xtop*, *y*).
3. If *c* > 0:
   3.1. Set *xleft* to *x*, and set *ychild* to *y*+*yelem*+*ygap*.
   3.2. For *i* = 1, …, *c*, repeat:
      3.2.1. Draw the subtree whose topmost node is the *i*th child of *N*, with the top left of its bounding rectangle at (*xleft*, *ychild*), and let its width be *w*.
      3.2.2. Set *xchild*[*i*] to *xleft*+*w*/2.
      3.2.3. Increment *xleft* by *w*+*xgap*.
   3.3. Set *width* to *xleft*–*xgap*–*x*.
   3.4. Set *xtop* to *x*+*width*/2.
   3.5. Draw node *N*'s element centered at (*xtop*, *y*).
   3.6. For *i* = 1, …, *c*, repeat:
      3.6.1. Draw a straight line from (*xtop*, *y*+*yelem*) to (*xchild*[*i*], *ychild*).
4. Terminate with answer *width*.