Laboratories from

Java Structures

Data Structures in Java for the Principled Programmer

Second Edition

Duane A. Bailey

Williams College

Laboratory: The Day of the Week Calculator

Objective. To (re)establish ties with Java: to write a program that reminds us of the particulars of numeric calculations and array manipulation in Java.

Discussion. In this lab we learn to compute the day of the week for any date between January 1, 1900, and December 31, 2099.[1] During this period of time, the only calendar adjustment is a leap-year correction every 4 years. (Years divisible by 100 are normally not leap years, but years divisible by 400 always are.) Knowing this, the method essentially computes the number of days since the beginning of the twentieth century in modulo 7 arithmetic. The computed remainder tells us the day of the week, where 0 is Saturday.

An essential feature of this algorithm involves remembering a short table of monthly adjustments. Each entry in the table corresponds to a month, where January is month 1 and December is month 12.

Month	1	2	3	4	5	6	7	8	9	10	11	12
Adjustment	1	4	4	0	2	5	0	3	6	1	4	6

If the year is divisible by 4 (it's a leap year) and the date is January or February, you must subtract 1 from the adjustment.

Remembering this table is equivalent to remembering how many days are in each month. Notice that 144 is 12^2, 025 is 5^2, 036 is 6^2, and 146 is a bit more than 12^2. Given this, the algorithm is fairly simple:

1. Write down the date numerically. The date consists of a month between 1 and 12, a day of the month between 1 and 31, and the number of years since 1900. Grace Hopper, computer language pioneer, was born December 9, 1906. That would be represented as year 6. Jana the Giraffe, of the National Zoo, was born on January 18, 2001. That year would be represented as year 101.

2. Compute the sum of the following quantities:

   - the month adjustment from the given table (e.g. 6 for Admiral Hopper)
   - the day of the month
   - the year
   - the whole number of times 4 divides the year (e.g. 25 for Jana the Giraffe)

[1] This particular technique is due to John Conway, of Princeton University. Professor Conway answers 10 day of the week problems before gaining access to his computer. His record is at the time of this writing well under 15 seconds for 10 correctly answered questions. See "Scientist at Work: John H. Conway; At Home in the Elusive World of Mathematics," *The New York Times*, October 12, 1993.

3. Compute the remainder of the sum of step 2, when divided by 7. The remainder gives the day of the week, where Saturday is 0, Sunday is 1, etc. Notice that we can compute the remainders *before* we compute the sum. You may also have to compute the remainder after the sum as well, but if you're doing this in your head, this considerably simplifies the arithmetic.

What day of the week was Tiger Woods born?

1. Tiger's birth date is 12-30-75.
2. Remembering that $18 \times 4 = 72$, we write the sum as follows:

$$6 + 30 + 75 + 18$$

   which is equivalent to the following sum, modulo 7:

$$6 + 2 + 5 + 4 = 17 \equiv 3 \bmod 7$$

3. He was born on day 3, a Tuesday.

Now you practice: Which of Grace and Jana was born on a Thursday? (The other was born on a Sunday.)

Procedure. Write a Java program that performs Conway's day of the week challenge:

1. Develop an object that can hold a date.
2. Write a method to compute a random date between 1900 and 2099. How will you limit the range of days potentially generated for any particular month?
3. Write a method of your date class to compute the day of the week associated with a date. Be careful: the table given in the discussion has January as month 1, but Java would prefer it to be month 0! Don't forget to handle the birthday of Jimmy Dorsey (famous jazzman), February 29, 1904.

Jimmy was a Monday's child.

4. Your `main` method should repeatedly (1) print a random date, (2) read a predicted day of the week (as an integer/remainder), and (3) check the correctness of the guess. The program should stop when 10 dates have been guessed correctly and print the elapsed time. (You may wish to set this threshold lower while you're testing the program.)

Helpful Hints. You may find the following Java useful:

1. Random integers may be selected using the `java.util.Random` class:

```
Random r = new Random();
int month = (Math.abs(r.nextInt()) % 12) + 1;
```

You will need to `import java.util.Random;` at the top of your program to make use of this class. Be aware that you need to only construct one random number generator per program run. Also, the random number generator potentially returns negative numbers. If `Math.abs` is not used, these values generate negative remainders.

2. You can find out how many thousandths of seconds have elapsed since the 1960s, by calling the Java method, `System.currentTimeMillis()`. It returns a value of type `long`. We can use this to measure the duration of an experiment, with code similar to the following:

In 2001, 1 trillion millis since the '60s. Dig that!

```
long start = System.currentTimeMillis();
//
// place experiment to be timed here
//
long duration = System.currentTimeMillis()-start;
System.out.println("time: "+(duration/1000.0)+" seconds.");
```

   The granularity of this timer isn't any better than a thousandth of a second. Still, we're probably not in Conway's league yet.

After you finish your program, you will find you can quickly learn to answer 10 of these day of the week challenges in less than a minute.

Thought Questions. Consider the following questions as you complete the lab:

1. True or not: In Java is it true that `(a % 7) == (a - a/7*7)` for `a >= 0`?
2. It's rough to start a week on Saturday. What adjustments would be necessary to have a remainder of 0 associated with Sunday? (This might allow a mnemonic of Nun-day, One-day, Twos-day, Wednesday, Fours-day, Fives-day, Saturday.)
3. Why do you *subtract* 1 in a leap year if the date falls before March?
4. It might be useful to compute the portion of any calculation associated with this year, modulo 7. Remembering that value will allow you to optimize your most frequent date calculations. What is the remainder associated with this year?

For years divisible by 28: think zero!

Notes:

Laboratory: Using Javadoc Commenting

Objective. To learn how to generate formal documentation for your programs.

Discussion. The `javadoc` program[2] allows the programmer to write comments in a manner that allows the generation web-based documentation. Programmers generating classes to be used by others are particularly encouraged to consider using `javadoc`-based documentation. Such comments are portable, web-accessible, and they are directly extracted from the code.

In this lab, we will write documentation for an extended version of the `Ratio` class we first met in Chapter **??**.

Comments used by `javadoc` are delimited by a `/** */` pair. Note that there are two asterisks in the start of the comment. Within the comment are a number of keywords, identified by a leading "at-sign" (`@`). These keywords identify the purpose of different comments you right. For example, the text following an `@author` comment identifies the programmer who originally authored the code. These comments, called `javadoc` comments, appear *before* the objects they document. For example, the first few lines of the `Assert` class are:

```
package structure;
/**
 * A library of assertion testing and debugging procedures.
 * <p>
 * This class of static methods provides basic assertion testing
 * facilities.  An assertion is a condition that is expected to
 * be true at a certain point in the code.  Each of the
 * assertion-based routines in this class perform a verification
 * of the condition and do nothing (aside from testing side-effects)
 * if the condition holds.  If the condition fails, however, the
 * assertion throws an exception and prints the associated message,
 * that describes the condition that failed.  Basic support is
 * provided for testing general conditions, and pre- and
 * post-conditions. There is also a facility for throwing a
 * failed condition for code that should not be executed.
 * <p>
 * Features similar to assertion testing are incorporated
 * in the Java 2 language beginning in SDK 1.4.
 * @author duane a. bailey
 */
public class Assert
{
   . . .
}
```

[2] `javadoc` is a feature of command-line driven Java environments. Graphical environments likely provide `javadoc`-like functionality, but pre- and postcondition support may not be available.

For each class you should provide any class-wide documentation, including id@author and `@version`-tagged comments.

Within the class definition, there should be a `javadoc` comment for each instance variable and method. Typically, `javadoc` comments for instance variables are short comments that describe the role of the variable in supporting the class state:

```
/**
 * Size of the structure.
 */
int size;
```

Comments for methods should include a description of the method's purpose. A comment should describe the purpose of each parameter (`@param`), as well as the form of the value returned (`@return`) for function-like methods. Programmers should also provide pre- and postconditions using the `@pre` and `@post` keywords.[3]

```
/**
 *
 * This method computes the square root of a double value.
 * @param x The value whose root is to be computed.
 * @return The square root of x.
 * @pre x >= 0
 * @post computes the square root of x
 */
```

To complete this lab, you are to

1. Download a copy of the

2. Ratio.java source from the Java Structures web site. This version of the `Ratio` package does not include full comments.

3. Review the code and make sure that you understand the purpose of each of the methods.

4. At the top of the `Ratio.java` file, place a `javadoc` comment that describes the class. The comment should describe the features of the class and an example of how the class might be used. Make sure that you include an `@author` comment (use your name).

[3] In this book, where there are constraints on space, the pre- and postconditions are provided in non-javadoc comments. Code available on the web, however, is uniformly commented using the `javadoc` comments. `javadoc` and be upgraded to recognize pre- and postconditions; details are available from the *Java Structures* web site.

5. Run the documentation generation facility for your particular environment. For Sun's Java environment, the `javadoc` command takes a parameter that describes the location of the source code that is to be documented:

   ```
   javadoc prog.java
   ```

   The result is an `index.html` file in the current directory that contains links to all of the necessary documentation. View the documentation to make sure your description is formatted correctly.

6. Before each instance variable write a short `javadoc` comment. The comment should begin with `/**` and end with `*/`. Generate and view the documentation and note the change in the documentation.

7. Directly before each method write a `javadoc` comment that includes, at a minimum, one or two sentences that describe the method, a `@param` comment for each parameter in the method, a `@return` comment describing the value returned, and a `@pre` and `@post` comment describing the conditions.

   Generate and view the documentation and note the change in the documentation. If the documentation facility does not appear to recognize the `@pre` and `@post` keywords, the appropriate `javadoc` doclet software has not been installed correctly. More information on installation of the `javadoc` software can be found at the Java Structures web site.

Notes:

Laboratory: The Silver Dollar Game

Objective. To implement a simple game using `Vector`s or arrays.

Discussion. *The Silver Dollar Game* is played between two players. An arbitrarily long strip of paper is marked off into squares:

The game begins by placing silver dollars in a few of the squares. Each square holds at most one coin. Interesting games begin with some pairs of coins separated by one or more empty squares.

The goal is to move all the n coins to the leftmost n squares of the paper. This is accomplished by players alternately moving a single coin, constrained by the following rules:

1. Coins move only to the left.
2. No coin may pass another.
3. No square may hold more than one coin.

The last person to move is the winner.

Procedure. Write a program to facilitate playing the Silver Dollar Game. When the game starts, the computer has set up a random strip with 3 or more coins. Two players are then alternately presented with the current game state and are allowed to enter moves. If the coins are labeled 0 through $n - 1$ from left to right, a move could be specified by a coin number and the number of squares to move the coin to the left. If the move is illegal, the player is repeatedly prompted to enter a revised move. Between turns the computer checks the board state to determine if the game has been won.

Here is one way to approach the problem:

1. Decide on an internal representation of the strip of coins. Does your representation store *all* the information necessary to play the game? Does your representation store *more* information than is necessary? Is it easy to test for a legal move? Is it easy to test for a win?
2. Develop a new class, `CoinStrip`, that keeps track of the state of the playing strip. There should be a constructor, which generates a random board. Another method, `toString`, returns a string representation of the coin strip. What other operations seem to be necessary? How are moves performed? How are rules enforced? How is a win detected?

3. Implement an application whose `main` method controls the play of a single game.

Thought Questions. Consider the following questions as you complete the lab:

Hint: When flipped, the Belgian Euro is heads 149 times out of 250.

1. How might one pick game sizes so that, say, one has a 50 percent chance of a game with three coins, a 25 percent chance of a game with four coins, a $12\frac{1}{2}$ percent chance of a game with five coins, and so on? Would your technique bias your choice of underlying data structure?
2. How might one generate games that are not immediate wins? Suppose you wanted to be guaranteed a game with the possibility of n moves?
3. Suppose the computer could occasionally provide good hints. What opportunities appear easy to recognize?
4. How might you write a method, `computerPlay`, where the computer plays to win?
5. A similar game, called *Welter's Game* (after C. P. Welter, who analyzed the game), allows the coins to pass each other. Would this modification of the rules change your implementation significantly?

Notes:

Laboratory: How Fast Is Java?

Objective. To develop an appreciation for the speed of basic Java operations including assignment of value to variables, arrays, and `Vectors`.

Discussion. How long does it take to add two integers in Java? How long does it take to assign a value to an entry in an array? The answers to these questions depend heavily on the type of environment a programmer is using and yet play an important role in evaluating the trade-offs in performance between different implementations of data structures.

If we are interested in estimating the time associated with an operation, it is difficult to measure it accurately with clocks available on most modern machines. If an operation takes 100 ns (*nanoseconds*, or billionths of a second), 10,000 of these operations can be performed within a single millisecond clock tick. It is unlikely that we would see a change in the millisecond clock while the operation is being performed.

One approach is to measure, say, the time it takes to perform a million of these operations, and divide that portion of the time associated with the operation by a million. The result *can* be a very accurate measurement of the time it takes to perform the operation. Several important things must be kept in mind:

- Different runs of the experiment can generate different times. This variation is unlikely to be due to significant differences in the speed of the operation, but instead to various interruptions that regularly occur when a program is running. Instead of computing the *average* of the running times, it is best to compute the *minimum* of the experiment's elapsed times. It's unlikely that this is much of an underestimate!

- Never perform input or output while you are timing an experiment. These operations are very expensive and variable. When reading or writing, make sure these operations appear before or after the experiment being timed.

- On modern systems there are many things happening concurrently with your program. Clocks tick forward, printer queues manage printers, network cards are accepting viruses. If you can keep your total experiment time below, say, a tenth of a second, it is likely that you will eliminate many of these distractions.

- The process of repeating an operation takes time. One of our tasks will be to measure the time it takes to execute an empty `for` loop. The loop, of course, is not really empty: it performs a test at the top of the loop and an increment at the bottom. Failing to account for the overhead of a `for` loop makes it impossible to measure any operation that is significantly faster.

- Good compilers can recognize certain operations that can be performed more efficiently in a different way. For example, traditional computers can assign a value of 0 much faster than the assignment of a value of 42. If an experiment yields an unexpectedly short operation time, change the Java to obscure any easy optimizations that may be performed. Don't forget to subtract the overhead of these obscuring operations!

Keeping a mindful eye on your experimental data will allow you to effectively measure very, very short events accurate to nanoseconds. In one nanosecond, light travels 11.80 inches!

Procedure. The ultimate goal of this experiment is a formally written lab report presenting your results. Carefully design your experiment, and be prepared to defend your approach. The data you collect here is experimental, and necessarily involves error. To reduce the errors described above, perform multiple runs of each experiment, and carefully document your findings. Your report should include results from the following experiments:

1. A description of the machine you are using. Make sure you use this machine for all of your experiments.

2. Write a short program to measure the time that elapses, say, when an empty `for` loop counts to one million. Print out the elapsed time, as well as the per-iteration elapsed time. Adjust the number of loops so that the total elapsed time falls between, say, one-hundredth and one-tenth of a second.

   Recall that we can measure times accurate to one-thousandth of a second using `System.currentTimeMillis()`:

   ```
   int i, loops;
   double speed;
   loops = 10000000;
   long start,stop,duration;

   start = System.currentTimeMillis();
   for (i = 0; i < loops; i++)
   {
       // code to be timed goes here
   }
   stop = System.currentTimeMillis();

   duration = stop-start;
   System.out.println("# Elapsed time: "+duration+"ms");
   System.out.println("# Mean time: "+
                      (((double)duration)/loops*NANOSPERMILLI)+
                      "nanoseconds");
   ```

3. Measure the time it takes to do a single integer assignment (e.g., `i=42;`). Do not forget to subtract the time associated with executing the `for` loop.

4. Measure the time it takes to assign an integer to an array entry. Make sure that the array has been allocated *before* starting the timing loop.

5. Measure the time it takes to assign a `String` reference to an array.

6. Measure the length of time it takes to assign a `String` to a `Vector`. (Note that it is not possible to directly assign an `int` to a `Vector` class.)

7. Copy one `Vector` to another, manually, using `set`. Carefully watch the elapsed time and do not include the time it takes to construct the two `Vectors`! Measure the time it takes to perform the copy for `Vectors` of different lengths. Does this appear to grow linearly?

Formally present your results in a write-up of your experiments.

Thought Questions. Consider the following questions as you complete the lab:

1. Your Java compiler and environment may have several switches that affect the performance of your program. For example, some environments allow the use of *just-in-time* (jit) compilers, that compile frequently used pieces of code (like your timing loop) into machine-specific instructions that are likely to execute faster. How does this affect your experiment?

2. How might you automatically guarantee that your total experiment time lasts between, say, 10 and 100 milliseconds?

3. It is, of course, possible for a timer to *underestimate* the running time of an instruction. For example, if you time a single assignment, it is certainly possible to get an elapsed time of 0—an impossibility. To what extent would a timing underestimate affect your results?

Notes:

Laboratory: Sorting with Comparators

Objective. To gain experience with Java's `java.util.Comparator` interface.

Discussion. In Chapter **??** we have seen a number of different sorting techniques. Each of the techniques demonstrated was based on the fixed, natural ordering of values found in an array. In this lab we will modify the `Vector` class so that it provides a method, `sort`, that can be used—with the help of a `Comparator`—to order the elements of the `Vector` in any of a number of different ways.

Procedure. Develop an extension of `structure.Vector`, called `MyVector`, that includes a new method, `sort`.

Here are some steps toward implementing this new class:

1. Create a new class, `MyVector`, which is declared to be an extension of the `structure.Vector` class. You should write a default constructor for this class that simply calls `super();`. This will force the `structure.Vector` constructor to be called. This, in turn, will initialize the protected fields of the `Vector` class.

2. Construct a new `Vector` method called `sort`. It should have the following declaration:

   ```
   public void sort(Comparator c)
   // pre: c is a valid comparator
   // post: sorts this vector in order determined by c
   ```

   This method uses a `Comparator` type object to actually perform a sort of the values in `MyVector`. You may use any sort that you like.

3. Write an application that reads in a data file with several fields, and, depending on the `Comparator` used, sorts and prints the data in different orders.

Thought Questions. Consider the following questions as you complete the lab:

1. Suppose we write the following `Comparator`:

   ```
   import structure.*;
   import java.util.Iterator;
   import java.util.Comparator;
   public class RevComparator implements Comparator
   {
       protected Comparator base;

       public RevComparator(Comparator baseCompare)
       {
   ```

```
        base = baseCompare;
    }

    public int compare(Object a, Object b)
    {
        return -base.compare(a,b);
    }
}
```

What happens when we construct:

```
MyVector v = new MyVector();
ReadStream r = new ReadStream();

for (r.skipWhite(); !r.eof(); r.skipWhite())
{
    v.add(new Integer(r.readInt()));
}

Comparator c = new RevComparator(new IntegerComparator());
v.sort(c);
```

2. In our examples, here, a new `Comparator` is necessary for each sorting order. How might it be possible to add state information (`protected` data) to the `Comparator` to allow it to sort in a variety of different ways? One might imagine, for example, a method called `ascending` that sets the `Comparator` to sort into increasing order. The `descending` method would set the `Comparator` to sort in reverse order.

Notes:

Laboratory: The Two-Towers Problem

Objective. To investigate a difficult problem using `Iterator`s.

Discussion. Suppose that we are given n uniquely sized cubic blocks and that each block has a face area between 1 and n. Build two towers by stacking these blocks. How close can we get the heights of the two towers? The following two towers built by stacking 15 blocks, for example, differ in height by only 129 millions of an inch (each unit is one-tenth of an inch):

Still, this stacking is only the *second-best* solution! To find the best stacking, we could consider all the possible configurations.

We *do* know one thing: the total height of the two towers is computed by summing the heights of all the blocks:

$$h = \sum_{i=1}^{n} \sqrt{i}$$

If we consider all the *subsets* of the n blocks, we can think of the subset as the set of blocks that make up, say, the left tower. We need only keep track of that subset that comes closest to $h/2$ without exceeding it.

In this lab, we will represent a set of n distinct objects by a `Vector`, and we will construct an `Iterator` that returns each of the 2^n subsets.

Procedure. The trick to understanding how to generate a subset of n values from a `Vector` is to first consider how to generate a subset of indices of elements from 0 to $n-1$. Once this simpler problem is solved, we can use the indices to help us build a `Vector` (or subset) of values identified by the indices.

There are exactly 2^n subsets of values 0 to $n-1$. We can see this by imagining that a coin is tossed n times—once for each value—and the value is added to the subset if the coin flip shows a head. Since there are $2 \times 2 \times \cdots \times 2 = 2^n$ different sequences of coin tosses, there are 2^n different sets.

We can also think of the coin tosses as determining the place values for n different digits in a binary number. The 2^n different sequences generate binary

numbers in the range 0 through $2^n - 1$. Given this, we can see a line of attack: count from 0 to $2^n - 1$ and use the binary digits (*bits*) of the number to determine which of the original values of the `Vector` are to be included in a subset.

Computer scientists work with binary numbers frequently, so there are a number of useful things to remember:

- An `int` type is represented by 32 bits. A `long` is represented by 64 bits. For maximum flexibility, it would be useful to use `long` integers to represent sets of up to 64 elements.
- The *arithmetic shift* operator (`<<`) can be used to quickly compute powers of 2. The value 2^i can be computed by shifting a unit bit (1) i places to the left. In Java we write this `1<<i`. This works only for nonnegative, integral powers.
- The *bitwise and* of two integers can be used to determine the value of a single bit in a number's binary representation. To retrieve bit `i` of an integer `m` we need only compute `m & (1<<i)`.

Armed with this information, the process of generating subsets is fairly straightforward. One line of attack is the following:

1. Construct a new extension to the `AbstractIterator` class. (By extending the `AbstractIterator` we support both the `Iterator` and `Enumeration` interfaces.) This new class should have a constructor that takes a `Vector` as its sole argument. Subsets of this `Vector` will be returned as the `Iterator` progresses.
2. Internally, a `long` value is used to represent the current subset. This value increases from 0 (the empty set) to $2^n - 1$ (the entire set of values) as the `Iterator` progresses. Write a `reset` method that resets the subset counter to 0.
3. Write a `hasNext` method that returns `true` if the current value is a reasonable representation of a subset.
4. Write a `get` method that returns a new `Vector` of values that are part of the current subset. If bit `i` of the current counter is 1, element `i` of the `Vector` is included in the resulting subset `Vector`.
5. Write a `next` method. Remember it returns the current subset *before* incrementing the counter.
6. For an `Iterator` you would normally have to write a `remove` method. If you extend the `AbstractIterator` class, this method is provided and will do nothing (this is reasonable).

You can now test your new `SubsetIterator` by having it print all the subsets of a `Vector` of values. Remember to keep the `Vector` small. If the original values are all distinct, the subsets should all have different values.

To solve the two-towers problem, write a `main` method that inserts the values $\sqrt{1}$, $\sqrt{2}$,...,$\sqrt{n}$ as `Double` objects into a `Vector`. A `SubsetIterator` is then used to construct 2^n subsets of these values. The values of each subset are summed, and the sum that comes closest to, but does not exceed, the value $h/2$ is remembered. After all the subsets have been considered, print the best solution.

Thought Questions. Consider the following questions as you complete the lab:

1. What is the best solution to the 15-block problem shown above?

2. This method of exhaustively checking the subsets of blocks will not work for very large problems. Consider, for example, the problem with 50 blocks: there are 2^{50} different subsets. One approach is to repeatedly pick and evaluate random subsets of blocks (stop the computation after 1 second of elapsed time, printing the best subset found). How would you implement `randomSubset`, a new `SubsetIterator` method that returns a random subset?

Notes:

Laboratory: Lists with Dummy Nodes

Objective. To gain experience implementing `List`-like objects.

Discussion. Anyone attempting to understand the workings of a doubly linked list understands that it is potentially difficult to keep track of the references. One of the problems with writing code associated with linked structures is that there are frequently *boundary cases*. These are special cases that must be handled carefully because the "common" path through the code makes an assumption that does not hold in the special case.

Take, for example, the `addFirst` method for `DoublyLinkedLists`:

```
public void addFirst(Object value)
// pre: value is not null
// post: adds element to head of list
{
    // construct a new element, making it head
    head = new DoublyLinkedListElement(value, head, null);
    // fix tail, if necessary
    if (tail == null) tail = head;
    count++;
}
```

The presence of the `if` statement suggests that sometimes the code must reassign the value of the `tail` reference. Indeed, if the list is empty, the first element must give an initial non-`null` value to `tail`. Keeping track of the various special cases associated with a structure can be very time consuming and error-prone.

One way that the complexity of the code can be reduced is to introduce *dummy nodes*. Usually, there is one dummy node associated with each external reference associated with the structure. In the `DoublyLinkedList`, for example, we have two references (`head` and `tail`); both will refer to a dedicated dummy node:

These nodes appear to the code to be normal elements of the list. In fact, they do not hold any useful data. They are completely hidden by the abstraction of the data structure. They are *transparent*.

Because most of the boundary cases are associated with maintaining the correct values of external references and because these external references are now "hidden" behind their respective dummy nodes, most of the method code is simplified. This comes at some cost: the dummy nodes take a small amount of space, and they must be explicitly stepped over if we work at either end of the list. On the other hand, the total amount of code to be written is likely to be reduced, and the running time of many methods decreases if the special condition testing would have been expensive.

Procedure. In this lab we will extend the `DoublyLinkedList`, building a new class, `LinkedList`, that makes use of two dummy nodes: one at the head of the list, and one at the end.

You should begin taking a copy of the `LinkedList.java` starter file. This file simply declares `LinkedList` to be an extension of the `structure` package's `DoublyLinkedList` class. The code associated with each of the existing methods is similar to the code from `DoublyLinkedList`. You should replace that code with working code that makes use of two dummy nodes:

`LinkedList`

1. First, recall that the three-parameter constructor for `DoublyLinkedList-Element`s takes a value and two references—the nodes that are to be `next` and `previous` to this new node. That constructor will also update the `next` and `previous` nodes to point to the newly constructed node. You may find it useful to use the one parameter constructor, which builds a node with `null next` and `previous` references.

2. Replace the constructor for the `LinkedList`. Instead of constructing `head` and `tail` references that are `null`, you should construct two dummy nodes; one node is referred to by `head` and the other by `tail`. These dummy nodes should point to each other in the natural way. Because these dummy nodes replace the `null` references of the `DoublyLinkedList` class, we will not see any need for `null` values in the rest of the code. Amen.

3. Check and make necessary modifications to `size`, `isEmpty`, and `clear`.

4. Now, construct two important `protected` methods. The method `insert-After` takes a `value` and a reference to a node, `previous`. It inserts a new node with the value `value` that directly follows `previous`. It should be declared `protected` because we are not interested in making it a formal feature of the class. The other method, `remove`, is given a reference to a node. It should unlink the node from the linked list and return the value stored in the node. You should, of course, assume that the node removed is not one of the dummy nodes. These methods should be simple with no `if` statements.

5. Using `insertAfter` and `remove`, replace the code for `addFirst`, `addLast`, `getFirst`, `getLast`, `removeFirst`, and `removeLast`. These methods should be very simple (perhaps one line each), with no `if` statements.

6. Next, replace the code for the indexed versions of methods `add`, `remove`, `get`, and `set`. Each of these should make use of methods you have already written. They should work without any special `if` statements.

7. Finally, replace the versions of methods `indexOf`, `lastIndexOf`, and `contains` (which can be written using `indexOf`), and the `remove` method that takes an object. Each of these searches for the location of a value in the list and then performs an action. You will find that each of these methods is simplified, making no reference to the `null` reference.

Thought Questions. Consider the following questions as you complete the lab:

1. The three-parameter constructor for `DoublyLinkedListElement`s makes use of two `if` statements. Suppose that you replace the calls to this constructor with the one parameter constructor and manually use `setNext` and `setPrevious` to set the appropriate references. The `if` statements disappear. Why?

2. The `contains` method can be written making use of the `indexOf` method, but not the other way around. Why?

3. Notice that we could have replaced the method `insertAfter` with a similar method, `insertBefore`. This method inserts a new value *before* the indicated node. Some changes would have to be made to your code. There does not appear, however, to be a choice between versions of `remove`. Why is this the case? (Hint: Do you ever pass a dummy node to `remove`?)

4. Even though we don't need to have the special cases in, for example, the indexed version of `add`, it is desirable to handle one or more cases in a special way. What are the cases, and why is it desirable?

5. Which file is bigger: your final result source or the original?

Notes:

Laboratory: A Stack-based Language

Objective. To implement a PostScript-based calculator.

Discussion. In this lab we will investigate a small portion of a stack-based language called PostScript. You will probably recognize that PostScript is a file format often used with printers. In fact, the file you send to your printer is a program that instructs your printer to draw the appropriate output. PostScript is stack-based: integral to the language is an operand stack. Each operation that is executed pops its operands from the stack and pushes on a result. There are other notable examples of stack-based languages, including `forth`, a language commonly used by astronomers to program telescopes. If you have an older Hewlett-Packard calculator, it likely uses a stack-based input mechanism to perform calculations.

We will implement a few of the math operators available in PostScript.

To see how PostScript works, you can run a PostScript simulator. (A good simulator for PostScript is the freely available `ghostscript` utility. It is available from `www.gnu.org`.) If you have a simulator handy, you might try the following example inputs. (To exit a PostScript simulator, type `quit`.)

1. The following program computes $1 + 1$:

   ```
   1 1 add pstack
   ```

   Every item you type in is a *token*. Tokens include numbers, booleans, or symbols. Here, we've typed in two numeric tokens, followed by two symbolic tokens. Each number is pushed on the internal stack of operands. When the `add` token is encountered, it causes PostScript to pop off two values and add them together. The result is pushed back on the stack. (Other mathematical operations include `sub`, `mul`, and `div`.) The `pstack` command causes the entire stack to be printed to the console.

2. Provided the stack contains at least one value, the `pop` operator can be used to remove it. Thus, the following computes 2 and prints nothing:

   ```
   1 1 add pop pstack
   ```

3. The following "program" computes $1 + 3 * 4$:

   ```
   1 3 4 mul add pstack
   ```

   The result computed here, 13, is different than what is computed by the following program:

   ```
   1 3 add 4 mul pstack
   ```

   In the latter case the addition is performed first, computing 16.

4. Some operations simply move values about. You can duplicate values—the following squares the number 10.1:

   ```
   10.1 dup mul pstack pop
   ```

   The `exch` operator to exchange two values, computing $1 - 3$:

   ```
   3 1 exch sub pstack pop
   ```

5. Comparison operations compute logical values:

   ```
   1 2 eq pstack pop
   ```

   tests for equality of 1 and 2, and leaves `false` on the stack. The program

   ```
   1 1 eq pstack pop
   ```

   yields a value of `true`.

6. Symbols are defined using the `def` operation. To define a symbolic value we specify a "quoted" symbol (preceded by a slash) and the value, all followed by the operator `def`:

   ```
   /pi 3.141592653 def
   ```

   Once we define a symbol, we can use it in computations:

   ```
   /radius 1.6 def
   pi radius dup mul mul pstack pop
   ```

   computes and prints the area of a circle with radius 1.6. After the pop, the stack is empty.

Procedure. Write a program that simulates the behavior of this small subset of PostScript. To help you accomplish this, we've created three classes that you will find useful:

- `Token`. An immutable (constant) object that contains a double, boolean, or symbol. Different constructors allow you to construct different `Token` values. The class also provides methods to determine the type and value of a token.

`Token`

- `Reader`. A class that allows you to read `Tokens` from an input stream. The typical use of a reader is as follows:

`Reader`

```
Reader r = new Reader();
Token t;
while (r.hasNext())
{
    t = (Token)r.next();
    if (t.isSymbol() && // only if symbol:
        t.getSymbol().equals("quit")) break;
    // process token
}
```

This is actually our first use of an `Iterator`. It always returns an `Object` of type `Token`.

- `SymbolTable`. An object that allows you to keep track of `String`–`Token` associations. Here is an example of how to save and recall the value of π:

```
SymbolTable table = new SymbolTable();
// sometime later:
table.add("pi",new Token(3.141592653));                    SymbolTable
// sometime even later:
if (table.contains("pi"))
{
    Token token = table.get("pi");
    System.out.println(token.getNumber());
}
```

You should familiarize yourself with these classes before you launch into writing your interpreter.

To complete your project, you should implement the PostScript commands `pstack`, `add`, `sub`, `mul`, `div`, `dup`, `exch`, `eq`, `ne`, `def`, `pop`, `quit`. Also implement the nonstandard PostScript command `ptable` that prints the symbol table.

Thought Questions. Consider the following questions as you complete the lab:

1. If we are performing an `eq` operation, is it necessary to assume that the values on the top of the stack are, say, numbers?

2. The `pstack` operation should print the contents of the operand stack without destroying it. What is the most elegant way of doing this? (There are many choices.)

3. PostScript also has a notion of a *procedure*. A procedure is a series of `Token`s surrounded by braces (e.g. `{ 2 add }`). The `Token` class reads procedures and stores the procedure's `Token`s in a `List`. The `Reader` class has a constructor that takes a `List` as a parameter and returns a `Reader` that iteratively returns `Token`s from its list. Can you augment your PostScript interpreter to handle the definition of functions like `area`, below?

```
/pi 3.141592653 def
/area { dup mul pi mul } def
1.6 area
9 area pstack
quit
```

Such a PostScript program defines a new procedure called `area` that computes πr^2 where r is the value found on the top of the stack when the procedure is called. The result of running this code would be

```
254.469004893
8.042477191680002
```

4. How might you implement the `if` operator? The `if` operator takes a boolean and a token (usually a procedure) and executes the token if the boolean is true. This would allow the definition of the absolute value function (given a less than operator, `lt`):

   ```
   /abs { 0 lt { -1 mul } if } def
   3 abs
   -3 abs
   eq pstack
   ```

   The result is `true`.

5. What does the following do?

   ```
   /count { dup 1 ne { dup 1 sub count } if } def
   10 count pstack
   ```

Notes:

Laboratory: The Web Crawler

Objective. To crawl over web pages in a breadth-first manner.

Discussion. Web crawling devices are a fact of life. These programs automatically venture out on the Web and internalize documents. Their actions are based on the links between pages. The data structures involved in keeping track of the progress of an avid web crawler are quite complex: imagine, for example, how difficult it must be for such a device to keep from chasing loops of references between pages.

In this lab we will build a web crawler that determines the distance from one page to another. If page A references page B, the distance from A to B is 1. Notice that page B may not have any links on it, so the distance from B to A may not be defined.

Here is an approach to determining the distance from page A to arbitrary page B:

- Start a list of pages to consider. This list has two columns: the page, and its distance from A. We can, for example, put page A on this list, and assign it a distance of zero. If we ever see page B on this list, the problem is solved: just print out the distance associated with B.
- Remove the first page on our list: call it page X with distance d from page A. If X has the same URL as B, B must be distance d from A. Otherwise, consider any link off of page X: either it points to a page we've already seen on our list (it has a distance d or less), or it is a new page. If it's a new page we haven't encountered, add it to the end of our list and associate with it a distance $d+1$—it's a link farther from A than page X. We consider all the links off of page X before considering a new page from our list.

If the list is a FIFO, this process is a *breadth-first* traversal, and the distance associated with B is the shortest distance possible. We can essentially think of the Web as a large maze that we're exploring.

Procedure. Necessary for this lab is a class `HTML`. It defines a reference to a textual (HTML) web page. The constructor takes a URL that identifies which page you're interested in:

```
HTML page = new HTML("http://www.yahoo.com");
```

Only pages that appear to have valid HTML code can actually be inspected (other types of pages will appear empty). Once the reference is made to the page, you can get its content with the method `content`:

`HTML`

```
System.out.println(page.content());
```

Two methods allow you to get the URL's associated with each link on a page: `hasNext` and `nextURL`. The `hasNext` method returns `true` if there are more links you have not yet considered. The `nextURL` method returns a URL (a `String`) that is pointed to by this page. Here's a typical loop that prints all the links associated with a page:

```
int i = 0;
while (page.hasNext())
{
    System.out.println(i+": "+page.nextURL());
    i++;
}
```

For the sake of speed, the HTML method downloads 10K of information. For simple pages, this covers at least the first visible page on the screen, and it might be argued that the most important links to other pages probably appear within this short start (many crawlers, for example, limit their investigations to the first part of a document). You can change the size of the document considered in the constructor:

```
HTML page = new HTML("http://www.yahoo.com",20*1024);
```

You should probably keep this within a reasonable range, to limit the total impact on memory.

Write a program that will tell us the maximum number of links (found on the first page of a document) that are necessary to get from your home page to any other page within your personal web. You can identify these pages because they all begin with the same prefix. We might think of this as a crude estimate of the "depth" or "diameter" of a site.

Thought Questions. Consider the following questions as you complete the lab:

1. How do your results change when you change the buffer size for the page to 2K? 50K? Under what conditions would a large buffer change cause the diameter of a web to decrease? Under what conditions would this change cause the diameter of a web to increase?

Notes:

Laboratory: Computing the "Best Of"

Objective. To efficiently select the largest k values of n.

Discussion. One method to select the largest k values in a sequence of n is to sort the n values and to look only at the first k. (In Chapter **??**, we will learn of another technique: insert each of the n values into a max-heap and extract the first k values.) Such techniques have two important drawbacks:

- The data structure that keeps track of the values must be able to hold $n >> k$ values. This may not be possible if, for example, there are more data than may be held easily in memory.
- The process requires $O(n \log n)$ time. It should be possible to accomplish this in $O(n)$ time.

One way to reduce these overheads is to keep track of, at all times, the best k values found. As the n values are passed through the structure, they are only remembered if they are potentially one of the largest k values.

Procedure. In this lab we will implement a `BestOf OrderedStructure`. The constructor for your `BestOf` structure should take a value k, which is an upper bound on the number of values that will be remembered. The default constructor should remember the top 10.

An `add` method takes an `Object` and adds the element (if reasonable) in the correct location. The `get(i)` method should return the ith largest value encountered so far. The `size` method should return the number of values currently stored in the structure. This value should be between 0 and k, inclusive. The `iterator` method should return an `Iterator` over all the values. The `clear` method should remove all values from the structure.

Here are the steps that are necessary to construct and test this data structure:

1. Consider the underlying structure carefully. Because the main considerations of this structure are size and speed, it would be most efficient to implement this using a fixed-size array. We will assume that here.
2. Implement the `add` method. This method should take the value and, like a pass of insertion sort, it should find the correct location for the new value. If the array is full and the value is no greater than any of the values, nothing changes. Otherwise, the value is inserted in the correct location, possibly dropping a smaller value.
3. Implement the `get(i)`, `size`, and `clear` methods.
4. Implement the `iterator` method. A special `AbstractIterator` need not be constructed; instead, values can be added to a linear structure and the result of *that* structure's `iterator` method is returned.

To test your structure, you can generate a sequence of integers between 0 and $n-1$. By the end, only values $n-k \ldots n-1$ should be remembered.

Thought Questions. Consider the following questions as you complete the lab:

1. What is the (big-O) complexity of one call to the `add` method? What is the complexity of making n calls to `add`?

2. What are the advantages and disadvantages of keeping the array sorted at all times? Would it be more efficient to, say, only keep the smallest value in the first slot of the array?

3. Suppose that $f(n)$ is defined to be $n/2$ if n is even, and $3n+1$ if n is odd. It is known that for small values of n (less than 10^{40}) the sequence of values generated by repeated application of f starting at n eventually reaches 1. Consider all the sequences generated from $n < 10,000$. What are the maximum values encountered?

4. The `BestOf` structure can be made more general by providing a third constructor that takes k and a `Comparator`. The comparisons in the `BestOf` class can now be recast as calls to the `compare` method of a `Comparator`. When a `Comparator` is not provided, an instance of the `structure` package's `NaturalComparator` is used, instead. Such an implementation allows the `BestOf` class to order non-`Comparable` values, and `Comparable` values in alternative orders.

Notes:

Laboratory: Playing Gardner's Hex-a-Pawn

Objective. To use trees to develop a game-playing strategy.

Discussion. In this lab we will write a simulator for the game Hex-a-Pawn. This game was developed in the early sixties by Martin Gardner. Three white and three black pawns are placed on a 3×3 chessboard. On alternate moves they may be either moved forward one square, or they may capture an opponent on the diagonal. The game ends when a pawn is promoted to the opposite rank, or if a player loses all his pieces, or if no legal move is possible.

In his article in the March 1962 *Scientific American*, Gardner discussed a method for teaching a computer to play this simple game using a relatively small number of training matches. The process involved keeping track of the different states of the board and the potential for success (a win) from each board state. When a move lead directly to a loss, the computer forgot the move, thereby causing it to avoid that particular loss in the future. This *pruning* of moves could, of course, cause an intermediate state to lead indirectly to a loss, in which case the computer would be forced to prune out an intermediate move.

Gardner's original "computer" was constructed from matchboxes that contained colored beads. Each bead corresponded to a potential move, and the pruning involved disposing of the last bead played. In a modern system, we can use nodes of a tree stored in a computer to maintain the necessary information about each board state. The degree of each node is determined by the number of possible moves.

Procedure. During the course of this project you are to

1. Construct a tree of Hex-a-Pawn board positions. Each node of the tree is called a `GameTree`. The structure of the class is of your own design, but it is likely to be similar to the `BinaryTree` implementation.

2. Construct three classes of `Player`s that play the game of Hex-a-Pawn. These three classes may interact in pairs to play a series of games.

Available for your use are three Javafiles:

`HexBoard` This class describes the state of a board. The default board is the 3×3 starting position. You can ask a board to print itself out (`toString`) or to return the `HexMove`s (`moves`) that are possible from this position. You can also ask a `HexBoard` if the current position is a win for a particular color—`HexBoard.WHITE` or `HexBoard.BLACK`. A static utility method, `opponent`, takes a color and returns the opposite color. The `main` method of this class demonstrates how `HexBoard`s are manipulated.

`HexBoard`

`HexMove` This class describes a valid move. The components of the `Vector` returned from the `HexBoard.moves` contains objects of type `HexMove`. Given a `HexBoard` and a `HexMove` one can construct the resulting `HexBoard` using a `HexBoard` constructor.

`HexMove`

`Player` When one is interested in constructing players that play Hex-a-Pawn, the `Player` interface describes the form of the `play` method that must be provided. The `play` method takes a `GameTree` node and an opposing `Player`. It checks for a loss, plays the game according to the `GameTree`, and then turns control over to the opposing player.

`Player` Read these class files carefully. You should not expect to modify them.

There are many approaches to experimenting with Hex-a-Pawn. One series of experiments might be the following:

1. Compile `HexBoard.java` and run it as a program. Play a few games against the computer. You may wish to modify the size of the board. Very little is known about the games larger than 3×3.

2. Implement a `GameTree` class. This class should have a constructor that, given a `HexBoard` and a color (a `char`, `HexBoard.WHITE` or `HexBoard.BLACK`), generates the tree of all boards reachable from the specified board position during normal game play. Alternate levels of the tree represent boards that are considered by alternate players. Leaves are winning positions for the player at hand. The references to other `GameTree` nodes are suggested by the individual moves returned from the `moves` method. A complete game tree for 3×3 boards has 370 nodes.

3. Implement the first of three players. It should be called `HumanPlayer`. If it hasn't already lost (i.e., if the opponent hasn't won), this player prints the board, presents the moves, and allows a human (through a `ReadStream`) to select a move. The play is then handed off to the opponent.

4. The second player, `RandPlayer`, should play randomly. Make sure you check for a loss before attempting a move.

5. The third player, called `CompPlayer`, should attempt to have the `CompPlayer` object modify the game tree to remove losing moves.

Clearly, `Player`s may be made to play against each other in any combination.

Thought Questions. Consider the following questions as you complete the lab:

1. How many board positions are there for the 3×4 board? Can you determine how many moves there are for a 3×5 board?

2. If you implement the learning machine, pit two machines against each other. Gardner called the computer to move first H.I.M., and the machine to move second H.E.R. Will H.I.M. or H.E.R. ultimately win more frequently? Explain your reasoning in a short write-up. What happens for larger boards?

3. In Gardner's original description of the game, each matchbox represented a board state *and its reflection*. What modifications to `HexBoard` and `HexMove` would be necessary to support this collapsing of the game tree?

Laboratory: Simulating Business

Objective. To determine if it is better to have single or multiple service lines.

Discussion. When we are waiting in a fast food line, or we are queued up at a bank, there are usually two different methods of managing customers:

1. Have a single line for people waiting for service. Every customer waits in a single line. When a teller becomes free, the customer at the head of the queue moves to the teller. If there are multiple tellers free, one is picked randomly.

2. Have multiple lines—one for each teller. When customers come in the door they attempt to pick the line that has the shortest wait. This usually involves standing in the line with the fewest customers. If there are multiple choices, the appropriate line is selected randomly.

It is not clear which of these two methods of queuing customers is most efficient. In the single-queue technique, tellers appear to be constantly busy and no customer is served before any customer that arrives later. In the multiple-queue technique, however, customers can take the responsibility of evaluating the queues themselves.

Note, by the way, that some industries (airlines, for example) have a mixture of both of these situations. First class customers enter in one line, while coach customers enter in another.

Procedure. In this lab, you are to construct a simulation of these two service mechanisms. For each simulation you should generate a sequence of customers that arrive at random intervals. These customers demand a small range of services, determined by a randomly selected service time. The simulation is driven by an event queue, whose elements are ranked by the event time. The type of event might be a customer arrival, a teller freeing up, etc.

For the single line simulation, have the customers all line up in a single queue. When a customer is needed, a single customer (if any) is removed from the customer queue, and the teller is scheduled to be free at a time that is determined by the service time. You must figure out how to deal with tellers that are idle—how do they wait until a customer arrives?

For the multiple line simulation, the customers line up at their arrival time, in one of the shortest teller queues. When a teller is free, it selects the next customer from its dedicated queue (if any). A single event queue is used to drive the simulation.

To compare the possibilities of these two simulations, it is useful to run the same random customers through both types of queues. Think carefully about how this might be accomplished.

Thought Questions. Consider the following questions as you complete the lab:

1. Run several simulations of both types of queues. Which queue strategy seems to process all the customers fastest?
2. Is their a difference between the average wait time for customers between the two techniques?
3. Suppose you simulated the ability to jump between lines in a multiple line simulation. When a line has two or more customers than another line, customers move from the end one line to another until the lines are fairly even. You see this behavior frequently at grocery stores. Does this change the type of underlying structure you use to keep customers in line?
4. Suppose lines were dedicated to serving customers of varying lengths of service times. Would this improve the average wait time for a customer?

Notes:

Laboratory: Improving the BinarySearchTree

Objective. To understand that improve an implementation.

Discussion. As we have seen in the implementation of the `BinarySearchTree` class, the insertion of values is relative to the root of the tree. One of the situations that must be handled carefully is the case where more than one node can have the same key. If equal keys are allowed in the binary search tree, then we must be careful to have them inserted on one side of the root. This behavior increases the complexity of the code, and when there are many duplicate keys, it is possible that the tree's depth can be increased considerably.

Procedure. An alternative approach is to have all the nodes with similar keys stored in the same location. When the tree is constructed in this manner, then there is no need to worry about keeping similar keys together—they're *always together*.

In this lab, we will implement a `BinaryMultiTree`—a `BinarySearchTree`-like structure that stores a multiset (a set of values with potential duplicates). We are not so concerned with the set features, but we are demanding that different values are kept in sorted order in the structure. In particular, the traversal of the `BinaryMultiTree` should return the values in order.

In this implementation, a `BinaryTree` is used to keep track of a `List` of values that are equal when compared with the `compare` method of the `ordering Comparator`. From the perspective of the structure, there is no distinguishing the members of the list. Externally, the interface to the `BinaryMultiTree` is exactly the same as the `BinarySearchTree`, but the various methods work with values stored in `List`s, as opposed to working with the values directly. For example, when we look at a value stored in a node, we find a `List`. A `getFirst` of this `List` class picks out an example that is suitable, for example, for comparison purposes.

Here are some things to think about during your implementation:

1. The `size` method does not return the number of nodes; it returns the number of values stored in all the nodes. The bookkeeping is much the same as it was before, but `size` is an upper bound on the actual size of the search tree.

2. The `add` method compares values to the heads of lists found at each node along the way. A new node is created if the value is not found in the tree; the value is inserted in a newly created `List` in the `BinaryTreeNode`. When an equal key is found, the search for a location stops, and the value is added to the `List`. A carefully considered `locate` method will help considerably here.

3. The `contains` method is quite simple: it returns `true` if the `getFirst` of any of the `List`s produces a similar value.

4. The `get` method returns one of the matching values, if found. It should probably be the same value that would be returned if a `remove` were executed in the same situation.

5. The `iterator` method returns an `Iterator` that traverses all the values of the `BinarySearchTree`. When a list of equal values is encountered, they are all considered before a larger value is returned.

When you are finished, test your code by storing a large list of names of people, ordered only by last name (you will note that this is a common technique used by stores that keep accounts: "Smith?" "Yes!" "Are you Paul or John?"). You should be able to roughly sort the names by inserting them into a `BinaryMultiTree` and then iterating across its elements.

Thought Questions. Consider the following questions as you complete the lab:

1. Recall: What is the problem with having equal keys stored on either side of an equal-valued root?

2. Does it matter what type of `List` is used? What kinds of operations are to be efficient in this `List`?

3. What is the essential difference between implementing the tree as described and, say, just directly storing linked lists of equivalent nodes in the `BinarySearchTree`?

4. An improved version of this structure might use a `Comparator` for primary and secondary keys. The primary comparison is used to identify the correct location for the value in the `BinaryMultiTree`, and the secondary key could be used to order the keys that appear equal using the primary key. Those values that are equal using the primary key are kept within an `OrderedStructure` that keeps track of its elements using the secondary key `Comparator`.

Notes:

Laboratory: The Soundex Name Lookup System

Objective. To use a `Map` structure to keep track of similar sounding names.

Discussion. The United States National Archives is responsible for keeping track of the census records that, according to the Constitution, must be gathered every 10 years. After a significant amount of time has passed (70 or more years), the census records are made public. Such records are of considerable historical interest and a great many researchers spend time looking for lost ancestors among these records.

To help researchers find individuals, the censuses are often indexed using a phonetic system called *Soundex*. This system takes a name and produces a short string called the Soundex key. The rules for producing the Soundex key of a name are precisely:

1. The entire name is translated into a series of digit characters:

Digit	Letter of name
`'1'`	b, p, f, v
`'2'`	c, s, k, g, j, q, x, z
`'3'`	d, t
`'4'`	l
`'5'`	m, n
`'6'`	r
`'7'`	*all other letters*

   For example, `O'Niell` would be translated into the string `757744`.

2. All double digits are reduced to single digits. Thus, `757744` would become `7574`.

3. The first digit is replaced with the first letter of the original name, in uppercase. Thus, `7574` would become `O574`.

4. All 7's are removed. Thus, `O574` becomes `O54`.

5. The string is truncated to four characters. If the resulting string is shorter than four characters, it is packed with enough `'0'` characters to bring the length to four. The result for `O'Niell` would be `O540`. Notice that, for the most part, the nonzero characters represent the significant sounded letters of the beginning of the name.

Other names translate to Soundex keys as follows:

Bailey	becomes	B400
Ballie	becomes	B400
Knuth	becomes	K530
Scharstein	becomes	S623
Lee	becomes	L000

Procedure. You are to write a system that takes a list of names (the UNIX spelling dictionary is a good place to find names) and generates an ordered map whose entries are indexed by the Soundex key. The values are the actual names that generated the key. The input to the program is a series of names, and the output is the Soundex key associated with the name, along with all the names that have that same Soundex key, in alphabetical order.

Pick a data structure that provides performance: the response to the query should be nearly instantaneous, even with a map of several thousand names.

Thought Questions. Consider the following questions as you complete the lab:

1. What is the Soundex system attempting to do when it encodes many letters into one digit? For example, why are d and t both encoded as `'3'`?
2. Why does the Soundex system ignore any more than the first four sounds in a name?

Notes:

Laboratory: Converting Between Units

Objective. To perform the transitive closure of a graph.

Discussion. An interesting utility available on UNIX systems is the `units` program. With this program you can convert between one unit and another. For example, if you were converting between feet and yards, you might have the following interaction with the `units` program:

```
You have: yard
You want: inch
Multiply by 36.0
```

The program performs its calculations based on a database of values which has entries that appear as follows:

```
1 yard 3 foot
1 foot 12 inch
1 yard 39.37 inch
```

Notice that there is no direct conversion between yards and inches.

In this lab you are to write a program that computes the relations between units. When the program starts, it reads in a database of values that describe the ratios between units. Each unit becomes a node of a graph, and the conversions are directed edges between related units. Note that the edges of the graph must be directed because the factor that converts inches to yards is the reciprocal of the factor that converts yards to inches.

In order to deduce conversions between distantly related units, it will be necessary for you to construct the closure of the graph. The labels associated with adjacent edges are multiplied together to label the direct edge.

Once your program reads the unit conversion database, the program should prompt for the source units and the destination units. It should then print out the conversion factor used to multiply the source units by to get the destination units. If the units do not appear to be related, you should print out a message that indicates that fact. Prompting continues until the user enters a blank line for the source units.

Thought Questions. Consider the following questions as you complete the lab:

1. There are two approaches to this problem: (1) construct the closure of the graph, or (2) perform a search from the source unit node to the destination unit node. What are the trade-offs to the two approaches?

2. What does it mean if the graph is composed of several disconnected components?

Notes: