In this lab, you will time different programs to show the relationship between predicted run time and observed running time (how long the program actually takes to run).

Experimenting with data structures

First, get the given code from the course homepage (http://courses.knox.edu/cs142, under “Assignments”). The given code is packaged as a .zip file, which you should download and import into Eclipse. (Remember to do so as an Existing Project rather than an archive.)

This will give you a project called “lab4-timing”. So far, all this program does is create an array of 25,000 words in an array, print a message, and then quit because we haven’t told it to do anything else. Not very satisfying. To make the program actually do something, we will expand the method timeArrayList to run some tests on the data in our Timing object and print its results to the console. Write a first version of this method, which should create a local ArrayList<String> and use a loop to add each element from data to the list. (Note that a list is like Bag, but it keeps track of the order in which elements are added; we’ll talk about it later in the term. ArrayList is the Java library’s built-in array-based list. To add to it, use the add method, which just takes the value to be added; this value is added to the end of the list.) Once you’ve made the program add all the elements, compile and run the resulting program. It still won’t appear to do anything, because we haven’t measured anything yet or printed our results.

Normally, to measure how long something takes, you look at the clock before it started, then look again afterwards, and subtract to find the result. That’s what we’ll do here, except that the clock we look at is the computer’s internal clock, and we will measure time in milliseconds (thousandths of a second). A call to System.currentTimeMillis() returns the value of the system clock as a long (essentially, a large int). Store its value before and after the test run, and subtract these values to get the elapsed time.

Go ahead and add two calls to System.currentTimeMillis()—one before you add everything, one after—and store their results in local variables. At the end of the timeArrayList method, add a print statement to report the results of the test. Compile and run this.

If you’re still running it on 25,000 words of input, the number it reports will be very small, probably just 0 or 1. That’s because computers are so fast at such a simple task that measuring in milliseconds is too coarse. Since that’s the best we can do, we pull another trick out of the toolbox: do the task repeatedly so that the combined time is large enough to measure. To do this, edit timeArrayList again and put the creation of the new ArrayList<String> and the adding of the objects into an outer loop so it repeats four hundred times. (Crucially, the time measurements must still be outside this loop! You’ll also want to declare the variable outside the loop and just make the list creation (the part with new) inside it.) Compile and run again, and the number you get will still be small, but at least larger than 1. (You can also try running it on a larger number of words; we will come back to that in a moment.)

The rest of the code you will write for this lab will be similar to this, but using different operations and data structures. First, after you’ve added everything into the ArrayList four hundred times, keep the last version of that list and measure how long it takes to get a value in the middle of the list with a call to

`list.get(this.data.length / 2)`

The get method allows the program to access elements in the list; it takes an index and returns the value at that index. This operation is even faster than adding to the list, so wrap it in a loop that executes it 40,000 times. You’ll now have three time measurements: before you start, after you add everything, and after you

---

1 Based on a lab by Don Blaheta.
finish getting. You should report the difference between the first two (as the time it takes to add everything 400 times) and the difference between the last two (as the time it takes to get the middle element 40,000 times).

The last test on ArrayList will be to try removing the first element in the list with

```java
list.remove(0);
```

Do this one twenty thousand times.

Once all that is done, write a method `timeLinkedList` that does all the same things except now with a LinkedList<String> instead of an ArrayList. (LinkedList is the library’s built-in implementation of list using linked memory. It supports the same methods as ArrayList.) Be sure to use cut and paste so you don’t have to type all that code again.

Now we will do something similar with `timeTreeSet` testing TreeSet<String>. (This is an implementation of the Set ADT that we talked about.) The adding part should be identical to what you were doing with the lists (i.e. use `add`). Since sets don’t keep objects in a particular order, though, it doesn’t make sense to get “the middle element” from a set. So instead, we’ll test how long it takes to run

```java
set.contains(this.data[i % data.length]);
```

where i is the loop variable you are using to repeat this operation. As with the `contains` method we wrote for lists, this method checks whether its argument is in the set. For reasons that will eventually become clear, you want to run this trial twenty million times to get good solid data about it. (Note that you should not include `remove` for sets.)

Finally, write a method `timeHashSet` that is like `timeTreeSet` except for using a HashSet<String>. This is a different implementation of the Set ADT.

Now that you’ve written all these tests, it is time to run them. We want to compare the performance of the different data structures as the input grows. For easy comparison and discussion tomorrow, here are the sizes I want you to run:

<table>
<thead>
<tr>
<th>size</th>
<th>ArrayList</th>
<th>LinkedList</th>
<th>TreeSet</th>
<th>HashSet</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>add</td>
<td>get</td>
<td>remove</td>
<td>add</td>
</tr>
<tr>
<td>25,000</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>50,000</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100,000</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>200,000</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>400,000</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>800,000</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Write it all down and remember to bring it with you to class tomorrow!