Programming assignment #2: CSE 12 Spring 2013

Here's a description of the second programming assignment for CSE 12, Spring 2013. In this assignment, you will write a class that implements the Java Collections interface `List<E>`, using a singly-linked list, and collect timing data on its performance. The JUnit test program you wrote for the previous assignment will be useful here.

>>> Due Fri Apr 26 5:00 PM

>>> Required solution files: List12.java TimeList.java TimeList.pdf

This means that the files that contain your solution to the assignment must be turned in by the due deadline by using the bundleP2 script while logged into your cs12 account on ieng6. If your file is turned in 25 hours or more before the deadline, you will get 1 additional point added to your programming assignment score. The turnin procedure is described below in the section "Turning in your assignment".

It's a good idea to plan to do the assignment well before the deadlines; terminals and tutors can get very busy at the last minute.

In doing this assignment, it's your responsibility to understand the course rules for integrity of scholarship.

Getting started

First, familiarize yourself with the topics in section 2.3, chapter 3, and section 5.3 of the textbook, or corresponding lecture notes. This assignment uses material introduced and discussed there. You will also find online documentation of the `java.util.List<E>` interface useful; see http://java.sun.com/javase/7/docs/api/java/util/List.html.

Next, create a directory named "P2" under your cs12 home directory. Your solution to this assignment should be turned in from that directory by the due date and time. You will use the bundleP2 script to do that; see below.

List12.java

In this assignment, you will define a class named `List12` that implements the `java.util.List<E>` interface. This implementation must use a singly-linked list with a head pointer (i.e. a pointer, in the List object itself, to the first element of the list or to a dummy header node), but no tail pointer (i.e. no pointer in the List object itself to the last element of the list), and will not extend any other class. That is, the class definition will begin:

```
public class List12<E> implements java.util.List<E> {
```

The `List<E>` interface in the java.util package requires 25 different public instance methods. However, for this assignment you only need to implement these 9 methods:

```
public boolean add(E o);
public void add(int index, E element);
public boolean contains(Object o);
```

```
public E get(int index);
public Iterator<E> iterator();
public boolean remove(Object o);
public E remove(int index);
public E set(int index, E element);
public int size();
```

All methods of the `List<E>` interface will need to have definitions in your List12 class in order for it to compile, but they can, and should, be defined just to throw an `UnsupportedOperationException`.

You already have written a List12Tester program to test all of these methods. This is an example of "test-driven development". By writing the tests first, you have (hopefully) come to understand the required behavior of all these methods. Assuming your List12Tester is correct, use it for testing your List12 class!

TimeList.java

Your `List12<E>` class implements the `List<E>` interface using a singly-linked list. Another common way to implement that interface is by using an array. In fact, the `ArrayList<E>` class in the `java.util` package does that. Each of these approaches has some advantages for some purposes.

Suppose you started with an empty `List12<String>`, and added N data items to it, each of them at the front of the list:

```
List<String> theL = new List12<String>();
for(int i=0; i<N; i++) {
  theL.add(0,null);
}
```

Adding at the front of a linked list takes time independent of the length of the list, so building the list from empty up to size N that way should theoretically take O(N) time. But suppose you did the same with an `ArrayList<String>`:

```
List<String> theA = new ArrayList<String>();
for(int i=0; i<N; i++) {
  theA.add(0,null);
}
```

Adding at the front of an array takes time proportional to the length of the array (since each existing element needs to be moved "one space to the right", to make room for the new element), so building the list from empty to size N that way should theoretically take $O(N^2)$ time. What about building a list by adding elements at the back instead of the front:

```
theL = new List12<String>();
for(int i=0; i<N; i++) {
   theL.add(i,null);
 }

theA = new ArrayList<String>();
for(int i=0; i<N; i++) {
   theA.add(i,null);
```

```
}
```

Now the theoretical time costs are reversed; building the linked list takes $O(N^2)$ time, because with only a pointer to the first element of the list, the list must be traversed from the head each time an element is added at the end; while an array takes O(N) time because, as long as its capacity will not be exceeded, it can add an element at the end in time independent of the length of the array.

Write a program in a file TimeList.java that will actually measure the time it takes for these operations, for a range of N, to see how, and to what extent, the theoretical time costs are true in practice. Your TimeList program must take two command line arguments. The first is either `linked` or `array` and specifies whether a List12 singly-linked list implementation or an ArrayList will be used; the second is either `front` or `back` and specifies whether elements will be added to the front (the head or beginning) or the back (the tail or end) of the list. Then your program will measure the time taken to build the list, starting with an empty one, for N ranging from 2,000 to 50,000, in steps of 1000, and print results to standard output (i.e. System.out).

To measure the time it takes to do a computation, query the system clock before the computation starts, query it again after the computation is done, and take the difference. In Java, `System.nanoTime()` returns a `long` giving the current system time in nanoseconds. In addition, it can help to call `System.gc()` before starting to time a computation, to attempt to run the Java garbage collector before and not during the computation (if it runs during the computation, it will skew the results). So, you can do something like this:

```
System.gc();
long start = System.nanoTime();

/* computation... */

long end = System.nanoTime();
double seconds = (end - start) / 1.0e9;
```

Any one running of a computation is subject to variation due to various factors such as other processes that may be running on the machine. In order to get a more reliable measurement of the time taken by a computation, you will find it is necessary to average over several runs for each N. Smaller N will tend to have larger variation per run, so you should average over more runs when N is small than when N is large. One way to accomplish that is to average over, say, R = 100,000/N runs, for each N.

Each of the R runs for a particular N will give you one observation of the time required to build a certain kind of list with N elements in a certain way. But how good are these observations? In statistics, an often used measurement of the confidence in a set of observations is their empirical *standard deviation,* which is the square root of the empirical *variance*.
Here's a way to compute the standard deviation: You are doing R runs for a particular N. For each run *i*, measure the time taken t_i.

Now Adding up all those R times, and dividing by R, will give the average time for that N:

$$\hat{t} = \frac{1}{R} \sum_{i=1}^{R} t_i$$

But also sum the square of the times; then dividing that sum by R will give the average squared time for that N:

$$\hat{s} = \frac{1}{R} \sum_{i=1}^{R} t_i^2$$

Now take that average squared time, minus the square of the average time. That is the empirical variance for that N. Take the square root of the variance to get the standard deviation:

$$\sigma = \sqrt{\hat{s} - \hat{t}^2}$$

A small standard deviation means that the various observed times in the R runs were relatively close to each other, and so are probably reliable; a large standard deviation means the observations were spread out, which is usually caused by extraneous factors such as noise. (Wikipedia has an article on the topic which explains more about it.)

Your TimeList program should print its results with three numbers per line, separated by a tab character: the value of N, the average of the observed times in seconds to build the N-element list, and the standard deviation of the observations. So, for example, running

```
java TimeList linked front
```

might print to the screen data like

```
 2000	0.000023	0.000000
 3000	0.000032	0.000000
 4000	0.000044	0.000002
 5000	0.000053	0.000000
 6000	0.000063	0.000000
 7000	0.000073	0.000000
 8000	0.000084	0.000000
 9000	0.000095	0.000004
10000	0.000104	0.000000
11000	0.000113	0.000000
12000	0.000124	0.000000
13000	0.000135	0.000003
14000	0.000145	0.000004
15000	0.000157	0.000005
16000	0.000166	0.000004
17000	0.000179	0.000006
18000	0.000188	0.000006
19000	0.000197	0.000005
20000	0.000208	0.000006
21000	0.000396	0.000307
22000	0.000229	0.000005
23000	0.000237	0.000004
24000	0.000249	0.000007
25000	0.000258	0.000006
26000	0.000268	0.000005
27000	0.000278	0.000006
28000	0.000289	0.000006
29000	0.000299	0.000007
30000	0.000309	0.000005
```

```
 31000  0.000326        0.000003
 32000  0.000330        0.000006
 33000  0.000339        0.000005
 34000  0.000350        0.000005
 35000  0.000366        0.000000
 36000  0.000370        0.000005
 37000  0.000382        0.000006
 38000  0.000398        0.000003
 39000  0.000400        0.000005
 40000  0.000412        0.000007
 41000  0.000421        0.000006
 42000  0.000439        0.000003
 43000  0.000441        0.000006
 44000  0.000451        0.000005
 45000  0.000461        0.000005
 46000  0.000471        0.000005
 47000  0.000482        0.000006
 48000  0.000492        0.000006
 49000  0.000512        0.000016
 50000  0.000518        0.000000
```

To save the data to a file named, say `lf.txt` or `ab.txt`, you could redirect standard output:

```
java TimeList linked front > lf.txt
java TimeList array back > ab.txt
```

A good way to visualize the number of comparisons, as a function of N, is to use a data plotting program like `gnuplot`. To use `gnuplot`, start the program, and tell it to plot data with lines and errorbars:

```
$ gnuplot
Terminal type set to
'x11'
gnuplot> set style data yerrorlines
```

Now ask gnuplot to plot some of the data files you created, for example:

```
gnuplot> plot "lf.txt","ab.txt"
```

Look at the data and see if it agrees with the theoretical big-O time cost analysis. If there are places where the data disagrees, can you suggest a reason why it disagrees?

TimeList.pdf

In a file TimeList.pdf, include a page or two summarizing and analyzing the data your TimeList program generated. (You can create your document using any word processing program that you like, and then export it in .pdf format; LibreOffice Writer is available on the ieng6 machines and will work for this.) Does the data agree with the theoretical big-O time cost, or were too many important details abstracted away from in that analysis? If there are places where the data diverges from the theoretical, can you suggest a reason why it disagrees? What obstacles did you have to address in order to get good, low-variance data?

Include some plots of your data in your TimeList.pdf document. Both Microsoft Word and the Open Office word processing program (available on the lab machines) can import images in Enhanced Metafile format (.emf). To get gnuplot to save a plot in a file `plot1.emf`, first plot it to the screen as discussed above, then do:

```
gnuplot> set term emf
gnuplot> set output "plot1.emf"
gnuplot> replot
gnuplot> set term pop
```

Images included in your TimeList.pdf file will be viewed on a computer monitor, so they don't need to be high-resolution. The total size of your turned-in files is limited. If your TimeList.pdf file is too large to turn in, try using lower-resolution or smaller images for your plots.

Turning in your assignment

After you are done with the assignment, you can turn it in. This process is essentially the same as for your previous assignment: when logged into ieng6 or a lab workstation, cd to the directory containing your solution file and run the command

```
bundleP2
```

and carefully follow the dialog. If it is before the due deadline and all required assignment files are present, this will turn in those required files. (There is no way to turn in any additional files.) You can run bundleP2 more than once if you want (and it is a good idea to try bundleP2 well before the deadline, even if your solution files aren't complete just to test it); only the last one will count.

To find out the current "official" time, run the command "date" on ieng6. After you have turned in your files, your assignment will be graded, and your grade emailed to your cs12 account. (Grading may take up to 10 days after the due deadline; please be patient until then.)

Grading

There are 15 possible points on this assignment, not including early-turnin points. We will test your solution as follows:

- We will copy your turned-in files into an otherwise empty directory.
- We will then compile your solution with the command:

  ```
  javac List12.java TimeList.java
  ```

  (If your solution files do not compile, you will recieve 0 points for this assignment.)
- We will test your `List12` class to make sure that it implements correctly the `List12<E>` methods required by this assignment.
- We will test your `TimeList` class to make sure it meets the requirements of this assignment.
- We will read your `TimeList.pdf` document and check that it shows that you understand the process of measuring algorithm performance and relating the measurements to theoretical analysis.
- We will run your `TimeList` program and check that its output is consistent with the data shown in your `TimeList.pdf` writeup.

We will also consider your coding style, including commenting.

If you have any questions about what constitutes good commenting or coding style, see a tutor in the lab. At least you should include a javadoc comment for each public variable, method, and class. The javadoc comment on a public class should use the `@author` tag to indicate the name of the author of the code (that is, you).

Good luck!