CS61B Lab #9

In this lab, we mostly ask you to play around with some data structures we've been looking at in lecture, and answer a few questions (add them to the file lab9.txt).

1. Testing Project 2

We've provided a Python script, test-jump61, on the instructional servers for testing your program. Typing 'test-jump61' (with no arguments) will print documentation of the program. Normally, one supplies one or more input files. Basically, each input file has one of the forms

```
   # Any number of # comment lines     # Any number of # comment lines
   COMMAND1                            COMMAND1
   COMMAND2                            None
   ===#1===                            ===#1===
   INPUT1                              INPUT1
   ===#2===                            ===#2===
   INPUT2
```

where
  * COMMAND1 and COMMAND2 are Unix shell commands to run instances of jump61. "None" means that only one program runs.
  * INPUT1 and INPUT2 are inputs that are sent to these programs, plus some special commands that indicate what kind of output test-jump61 should be expecting.

For each input file FOO.in, test-jump61 will also take the outputs from the two programs, filter out everything except board dumps (delimited by === and ===), and compare those outputs with the corresponding FOO.std (if it exists).

For two examples of the right-hand form (COMMAND2 is None, running just your program) see tests/*/*.in and tests/*/*.std in your project 2 skeleton.

1.1 Some simple scripts

Try your hand at writing scripts that test the staff program (invoked with the command 'staff-jump61' on the instructional servers). Specifically, write

A. A test (test1.in and test1.std) that the 'set' command works. This test should use 'dump' to check that a certain setup is as expected.
B. A test (test2.in, test2.std) that several moves of a fully manual game work properly. Again, use 'dump' to check the result.

1.2 More complex tests.
----

test-jump61 recognizes a number of special commands that cause it to wait and check output from the program(s) you are testing. You can find the complete list and descriptions by running 'test-jump61' with no arguments. Here they are:

```
    %ib    Wait for a blue move ('Blue moves <R> <C>.') from the
           other program and insert it here.
    %ir    Wait for a red move from the other program and insert
           it here.
    %mb    Wait for this program to print ('Blue moves <R> <C>.') send
           <R> <C> to the other program.
    %mr    Wait for this program to print a red move, and send it to
           the other program.
    %br    Wait for a sequence of alternating 'Blue moves...' and
           'Red moves...." from this program (i.e., where both players
           are automated), starting with blue, up to an announcement of
           a winner.
    %rb    As for %br, but starting with a red move.
    %ib... As for %ib, and then wait for a red move from this
           program, send it to the other program, and then repeat
           these steps up to an announcement of a winner.
    %ir... As for %ir, and then wait for a blue move from this
           program, send it to the other program, and then repeat
           these steps up to an announcement of a winner.
    %e     Indicates a point where test-jump61 expects an announcement
           of a winner (for either side). You can use this command
           after %ib..., %rb, etc. to look at the winner, or in a
           situation where you manually enter a complete game.
    %eb    As for %e, but expects to see "Blue wins."
    %er    As for %e, but expects to see "Red wins."
    %ae    As for %e, but also prints winner.
```

Using these commands, write

C. A test (test3.in) that the staff program can play a small game with both sides automated and come up with a winner.
D. A test (test4.in) that one instance of the staff program can play a small game against another instance. The first instance of the staff program will have an automated red player and a manual blue player, and the second will have a manual red play and an automated blue player. The first program will send red moves to the second, and the second will send blue moves to the first.

1.3 WARNING

If you include these tests in your proj2 submission (and you should), be sure to substitute your own program execution for staff-jump61!

2. Hashing

Be sure you've read Chapter 7 of _Data Structures (Into Java)_ and looked at the notes from Lecture #26.

The Java library allows any kind of reference object to be stored in a hash table. To make this work, it makes use of the following two methods defined in java.lang.Object

```
  public native int hashCode();
     // "Native" means "implemented in some other language".

  public boolean equals(Object obj) {
      return (this == obj);
  }
```

Since all Objects have at least these default implementations, all objects may be stored in and retrieved from hash tables.

When defining a new type, you may freely override both methods. However, they are related---when you override one, you must should generally override the other, or uses of hash tables will break, as we'll see. You might want to find the documentation of the Java library class java.lang.Object and look up the comments on these two methods.

2.1 Goodness of hashing functions

The file HashTesting.java contains various routines and classes for testing and timing hash tables. Compile this file in your directory. The file contains a wrapper class String1, which simply contains a String and returns via the toString method. The .equals method on this class compares the Strings in the two comparands. The hashCode method is an adaptation of that used in the real String class. It gives you the ability to "tweak" the algorithm by choosing to have the hashCode method look only at some of the characters. Take a look at class String1, and especially at its hashCode method.

The test

```
  java HashTesting test1 $MASTERDIR/lib/words 1
```

will read a list of about 100000 words from $MASTERDIR/lib/words (a small dictionary taken from a recent Ubuntu distribution), store them in a hash table (the Java library class HashSet), and then check that each is in the set. It times these last two steps and reports the time.

The argument 1 here causes it to use the same hashing function for the strings as Java normally does for java.lang.String. If you run

```
  java HashTesting test1 $MASTERDIR/lib/words 2
```

the hash function looks only at every other character, presumably making it faster to compute.

Try this command with various values of the second parameter. Explain why the timings change as they do in lab9.txt, question #1.

2.2 The effect of data

The command

```
  java HashTesting test2 N
```

for N an integer, will time the storage and retrieval of N**2 four-letter words in a hash table. These words all have the form xxyy , where the character codes for x and y vary from 1 to N (most of these "words" won’t be readable). The command

```
  java HashTesting test3 N
```

does the same thing, but the "words" have the form xXyY, where x and y vary as before, and

```
    Y=2**16-31y-1, X=2**16-31x-1.
```

Run these two commands with various values of N (start at 20). Explain in as much detail as you can the reasons for the relative timing behavior of these tests (that is, why test3 takes longer than test2), putting your answer in lab9.txt, question #2.

2.3 A faulty class

The class FoldedString is another wrapper class for Strings whose .equals and .compareTo methods ignore case (e.g., they treat "foo", "Foo", and "FOO" as equivalent). The command

```
  java HashTesting test4 the quick brown fox
```

will treat the trailing command-line arguments ("the", "quick", etc.) as FoldedStrings, and insert them into a HashSet and (for comparison) a TreeSet, which uses a balanced binary search tree to store its data. The program will then check that it can find the all-upper-case version of each of the words in these sets (since they are supposed to compare equal).

Try running test4 as shown. HashSet fails to find the words entered into it, when they are upper-cased, while the TreeSet seems to work just fine. Explain why in lab9.txt, question #3.

The problem is in the FoldedString class. Change it so that java HashTesting test4... works. Try to do so in a reasonable way: make sure that large HashSets full of FoldedStrings will continue to work well.

3. LinkedLists

The type java.util.LinkedList allows one to create ListIterators on a list, which provide the .previous() as well the .next() methods. The provision of .previous() suggests that LinkedList is, in fact, a doubly linked list (indeed, the documentation says so.) Fill in the program ListTesting.java to provide a demonstration that LinkedList is (or is not) indeed doubly linked. That is, your program should perform some kind of measurement that evidences double linking. You might find useful bits of code in HashTesting. Explain your demonstration in lab9.txt.

4. What to turn in

Submit your test*.in and test*.std files, your fixed version of FoldedString.java, your filled-in ListTesting.java, and lab9.txt as assignment lab9.