1 Lab Assignment 7

You are to continue working with counting votes by processing a precinct's worth of data and counting up votes. Unfortunately, this will be done the hard way; doing the count with a more sophisticated data structure will come in a future homework assignment.

I have given you initial code. This includes a revised `OneLine` class that is complete except for the `compareTo` method.

Note that I have replaced all the horrendous blank spaces with underscore characters so you can read the contest names and the candidate names with a `next`.

You will need to create a `compareTo` method that will sort FIRST BY CONTEST AND THEN BY CANDIDATE, alphabetically in both cases. That is, you should have all the "GOVERNOR" votes collected together, and inside all the GOVERNOR votes they should be sorted alphabetically by candidate name. Note that this will sort by first name, not last name. For governor, "Morgan" comes before "Nikki" which comes before "Vincent". You don't need to worry about sorting in any particular order. Just use the standard collating sequence for the strings of characters.

Start with your Lab 4 code, which lets you read in one line of vote data at a time and insert the new line into proper order in an `ArrayList`. This is the same as Lab 4 except that you are are reading data in a slightly different format. (I won't ever make you parse the actual ASCII text file that I parsed; that's tedious and annoying and requires lots of special cases for string matching, but doesn't really teach you anything.) Read in the data, and insert it into place.

Now, you need to rename your `ManagedList` class to call it `VoteCounter` (This can be done most easily by using the "refactor" option on the "file" tab in Eclipse; refactor will change the name and also change all the references to that name in the code.). Having changed the name, you need to add a method `countTheVotes` that will actually do the vote counting.

The first thing your `countTheVotes` method needs to do is run a loop on the `ArrayList` of `OneLine` data and simply count up the number of lines that have the `newBallot` flag set. This should give you a count of the number of actual ballots cast.

(This data resembles a lot of real world data. Although there might be 100 ballots cast by 100 different people, it may not be the case that the number of votes cast for all candidates, even for something like Governor,

totals up to 100, so you have to process the data under the assumption that nothing is going to add up perfectly.)

Having determined and printed out the number of total ballots cast, you should then count votes per contest per candidate. You have the `ArrayList` in sorted order, and now you need to do what is called "breakpoint processing". You need an `oldCandidate` and an `oldContest` variable. As you step through the `ArrayList` one item at a time, check the candidate and the contest against what is stored in `oldCandidate` and `oldContest`. Keep incrementing a count each time they are the same. When either of these changes, you have hit a breakpoint in the data and it is time to print out the "old" data.

There is nothing complicated about breakpoint processing, except for the fact that it is rife with opportunity to make off-by-one errors.

1. If you set your initial "old" string to an empty string, then the first time you read any data, the new string will not equal the old string, your code will want to print out a line, but the values will be zero because you haven't actually read any data yet.

2. Inside the loop, your code determines that you have reached a breakpoint in the line *after* the last line of the data you are to print out totals for.

3. Or you can do a lookahead to the next line of the data in the `ArrayList`, but then you have to be careful about the top end of your loop subscripting to make sure you don't try to read one past the end.

4. After you finish the loop, you will probably have read all the data but you will not have printed out the count for the last block of data read, because you print when the value changes and the value didn't change because you ran out of data instead. So you may need a second print statement after the loop.

So you need to be careful.

There are two moral lessons to be learned from this exercise.

One is that sorting can be slow. There are only 1289 ballots in this precinct (which is my own, so my own vote is in there somewhere), but it takes six minutes on my desktop to sort by insertion sort. There has to be a better way to sort, and there is, and we will cover that soon.

Second, in order to process this data with breakpoint processing, you need to read in all the data so you can sort it and then total on the breakpoints. That requires reading in all that stuff into memory. We will learn, and you will code in the homework assignment, a way to do this processing WITHOUT having to save the data in memory and WITHOUT having to read the data more than once. It's all a matter of using the right data structure for the task at hand.

I have given you the entire data file for precinct 378, and I have also given you input and output for the first 1000 lines of that file, since that's small enough to process as test data.