You (Auto)Complete me!

Due 5 November

Assignment 8
CSC 062: Fall, 2017

Prefix and Autocompletion

This week you will be exploring a phenomenon found on many web search pages. Bring up your favorite web browser and start typing a phrase in the search bar. As you type, suggested completions will appear. When you find a completion you like for the prefix you have typed, you can just select it and a page will be brought up with the results corresponding to the phrase you selected. This phenomenon appears on other search sites as well. If you go to netflix.com, click on the search icon, and then start typing, it will suggest a variety of completions that correspond to movies in its database. Eclipse will helpfully suggest completions as you are typing in Java code. Even you cell phone will suggest completions when you are typing in a text message.

If you try this several times, you will find that this kind of search has some interesting properties. First, only a limited number of completions are displayed. That should make sense to you. Suppose you have typed only a few letters. If there were no limitations on completions displayed the system would end up displaying hundreds of thousands of possible completions. Given that only a limited number of completions are displayed, how do you decide which ones to display? [*]

Typically the application depends on some historical data on how often certain completions are chosen, and each completion is associated with a number representing its frequency or likelihood of being chosen. (We did something like this earlier in the semester in Assignment 2, when you gathered information on the occurrence of triplets of words occurring in a text and then generated new text.)

For this lab we will assume that you are given a file with keys and associated weights. You will then read in the given information and suggest completions when the user types in a prefix. The original file will be sorted by the keys. When a query is given, you will find all the keys that start with the query, sort the matching terms by weight, and then report out the top entries.

Notes

1. The starter files are in the `/common/cs/cs062/assignments/assignment08` directory. Familiarize yourself with these files *before* starting to work on your lab. Every class you write should either have an associated JUnit test file or have a `main` method that can be used to test that all the methods work correctly. Part of your grade on this assignment will be based on how complete your tests are.

2. Write a class `Term` to represent a pair of a `key` (represented as a `String`) and a `weight` (represented as a `long` – the frequencies can be large!). The objects generated should be immutable (no changes allowed to the `key` or `weight` fields).

The class should implement the interface `Comparable<Term>`, and thus it must include the `compareTo` method. It should also override the `toString` method so you can see the key and weight of the term.

We also ask you to implement two static methods that return `Comparators`.[†] The first method, `byReverseWeightOrder` should return a comparator that has a compare method that ignores the key field, but compares the weights in reverse order by size. That is, if used in a sort, terms with higher weight would occur before those elements with smaller weight.

The second static method, `byPrefixOrder(int r)`, returns a comparator with a compare method that only considers the first r characters of the key field, and represents the usual lexicographic order. Thus if r is 3, the term with key "hello" would come before "hopper", but "hello" and "help" would be considered equal (because their first three characters are the same.

You may build the comparators in the static methods using anonymous classes or inner classes, but you will find it simpler if you use Java lambda expressions.

[*]This is a difficult question for all searches. For an early (and influential) answer, see the paper "The Anatomy of a Large-Scale Hypertextual Web Search Engine" `http://ilpubs.stanford.edu:8090/361/` by Brin and Page.

[†]These can be static because they don't depend on the instance variables of the `Term`. They return `Comparator`s that can be used to compare any two `Terms`.

Test the methods in this class thoroughly (using JUnit or a `main` method) before proceeding to the other classes. We suggest you build a small `ArrayList` of `Term`s and then sort them in several different ways using the comparators returned by the static methods. Feel free to use the built-in static `sort` method in `Collections`. See the Java documentation for details.

3. Next you should implement class `BinarySearchForAll`. This is an unusual class in that it has no instance variables and only provides two static public methods to find elements in a list. The first finds the index of the first element in a list that equals (according to the comparator) the key. The second finds the index of the last equaling (according to the comparator) the key. This will allow us to quickly grab all the terms that match a prefix in the `Autocomplete` class.

   Notice the phrase "according to the comparator"! We will be using this with the `Term` comparator `byPrefixOrder(r)` and we will want to get a match if the key we are searching for is a prefix of the element in the list. The bottom line is that you should not use the `equals` method to check for a match. Instead see if `compare` returns a 0.[‡]

```
/**
 * Returns the index of the first element in aList that equals key
 *
 * @param aList
 *           Ordered (via comparator) list of items to be searched
 * @param key
 *           item searching for
 * @param comparator
 *           Object with compare method corresponding to order on aList
 * @return Index of first item in aList matching key or -1 if not in aList
 **/
public static <Key> int firstIndexOf(List<Key> aList, Key key,
                      Comparator<Key> comparator);

/**
 * Returns the index of the last element in aList that equals key
 *
 * @param aList
 *           Ordered (via comparator) list of items to be searched
 * @param key
 *           item searching for
 * @param comparator
 *           Object with compare method corresponding to order on aList
 * @return Location of last item of aList matching key or -1 if no such key.
 **/
public static <Key> int lastIndexOf(List<Key> a, Key key,
                     Comparator<Key> comparator);
```

4. `Autocomplete`: This class will use `Term` and `BinarySearchForAll` to find all of the terms that match a given prefix and to return them in a list held in descending order by weight. The constructor of the class should take in a `List<Term>` and sort it according to the keys of the terms. The class has only a single method:

[‡]The problem that we have is that the comparator may not be consistent with the `equals` method. If we knew what comparator we would be using for comparing terms then we could override `equals`, but in this case we will use different comparators at different times.

```
/**
 * @param prefix
 *           string to be matched
 * @return List of all matching terms in descending order by weight
 */
List<Term> allMatches(String prefix);
```

Don't forget to the list returned should be sorted in descending order by weight!

We have provided an interface `AutocompleteInterface` that your class should implement.

5. Finally you should write a `static main` method in a class `AutocompleteMain` that takes two run-time parameters. The first is an `int` that determines how many matching items should be printed in response to a query, while the second is the name of the file holding the weights and keys. The file's first line will be an `int` specifying the number of lines of data in the file. All subsequent lines will consist of a `long` value representing the weight followed by one or more tab symbols and then a `String` representing the key. You can use a `Scanner` to read in the input. After reading in the `long` value, we suggest you read in the rest of the line and then use the `trim` method in `String` to throw away any white space preceding or following the key. You can then package these into a value of type `Term`. The `Terms` read in should be held in an `ArrayList`.

   Once the file is loaded, the program should prompt the user to type in a prefix and then print the top group of matching items (both keys and weights). It should then prompt the user to enter another prefix. Be aware that the number of matching items may be smaller than the number specified to be returned. In that case, just print out all the matching items.

   To test your program we have provided two files, "cities.txt" and "wiktionary.txt". They are both very large so don't print them out. For testing feel free to select a small amount of data from these files.

   Here is some sample output if the command line parameters are 5 and "cities.txt".

```
Enter a new prefix: hel
There are 10 matches.
The 5 largest are:
24371200 help
23547400 held
4048780 helped
3461920 helen
3177030 hell

Enter a new prefix: pom
There are 2 matches.
The matching items are:
873729 pomp
403557 pompous

Enter a new prefix:
```

6. **Performance** The method `Collections.sort` is a modified merge sort that is guaranteed to be `O(n log n)` in the worst case. Your methods should be efficient. In particular, assuming comparisons between terms takes constant time, your binary search methods should be `O(log n)`. The `allMatches` method in `Autocomplete` should take time `O(log n + m log m)` if the big list has n elements and there are m matching terms.

   Of course your program should not crash on any reasonable input (though if you try to read a non-existent file, you can print an error message and terminate).

7. Answer the following thought questions in the comments at the top of class `AutocompleteMain`. While the answers won't affect your grade on this project, you will find it useful to contemplate them. Please

post your answers to the second part of the last question in Piazza so that everyone can enjoy them. *But don't reveal the name of the film!*

(a) We had you use an `ArrayList` to hold the data in this program. Are there any reasons that it might be better or worse to use an array rather than `ArrayList`.

(b) Rather than defining the (rather odd) new class `BinarySearchForAll` that consisted only of static methods, why didn't we just subclass `ArrayList` and add the new methods to the subclass (leaving off the `ArrayList` parameters, of course)? Hint: What parts of your program would have broken?

(c) What movie does the name of this lab riff off of? (Hint: It was named the tenth best film in the sports genre by the American Film Institute.) Post another well known (preferably corny) line from this movie on Piazza for our entertainment. If all the good quotes are gone by the time you post, you may supply a corny quote from some other movie that the stars were in (together or apart).

Turn in

As usual, turn in your entire project using the submit script by Sunday at midnight. BE SURE THAT YOU USE EXACTLY THE NAMES SPECIFIED IN THE WRITE-UP!!

If you would like some extra credit, create a GUI version of the program that pops up a window, allows the user to select the number of matches to be returned and the file to be used, and then uses a JComboBox to get input from the user and display the matches.

Grading

You will be graded based on the following criteria:

Criterion	points
Term class & comparables	3
Search operations	3
allMatches	2
Load files & user interaction	3
Thoroughness of test code	2
Quality of code	4
Appropriate comments (including JavaDoc)	3

Acknowledgement: This assignment is based on a similar exercise developed by Matthew Drabick and Kevin Wayne at Princeton University.