Programming Assignment 3

Little Search Engine

In this assignment you will implement a simple search engine for text documents using hash tables.

Worth 100 points = 10% of your course grade

Posted Fri, Nov 6

Due Fri, Nov 20, 11:00 PM

Late submission 1: By Sat, Nov 21, 11:00 PM - 10 point penalty

Late submission 2: By Sun, Nov 22, 11:00 PM - 20 point penalty

WARNING!! NO GRACE PERIOD FOR ANY SUBMISSION DEADLINE

- You will work on this assignment individually. Read the DCS Academic Integrity Policy for Programming Assignments - you are responsible for abiding by the policy. In particular, note that "**All Violations of the Academic Integrity Policy will be reported by the instructor to the appropriate Dean**".

- **IMPORTANT - READ THE FOLLOWING CAREFULLY!!!**

  We will only grade submissions in Autolab.

  Assignments emailed to the instructor/TAs/graders will be ignored--they will NOT be accepted for grading. Also, any assignment on your computer, or any other online space, will NOT be accepted for grading, even if it appears to be not modified after a submission date.

  If your program does not compile, you will not get any credit.

  Most compilation errors occur for two reasons:

  1. You are programming outside Eclipse, and you delete the "package" statement at the top of the file. If you do this, you are changing the program structure, and it will not compile when we test it.
  2. You make some last minute changes, and submit without compiling.

  To avoid these issues, (a) START EARLY, and give yourself plenty of time to work through the assignment, and (b) Make use of the pregrading run on Autolab 48 hours before the submission deadline, so you can catch errors earlier than the final submission.
  You can submit as many times as you like, only the latest submission will be picked up by Autolab whenever grading is done.

Summary

You will implement a little search engine to do two things: (a) gather and index keywords that appear in a set of plain text documents, and (b) search for user-input keywords against the index and return a list of matching documents in which these keywords occur.

Implementation

In the Autolab assignment page, under **Attachments**, you will see a `lse_project.zip` file, which is an Eclipse project file.

DO NOT unzip the project zip file. Instead, follow the instructions on the Eclipse page under the section "Importing a Zipped Project into Eclipse" to get the entire project into your Eclipse workspace.

Here are the contents of the project:

- A class, `lse.LittleSearchEngine`. This is where you will fill in your code, details follow.
- A supporting class, `lse.Occurrence`, which you will NOT change.
- Two sample text documents, `AliceCh1.txt`, and `WowCh1.txt`, directly under the project folder, for preliminary testing. Be sure to get other online text documents--or make your own--for more rigorous testing.
- A `noisewords.txt` file that contains a list of "noise" words, one per line. Noise words are commonplace words (such as "the") that must be ignored by the search engine. You will use this file (and this file ONLY) to filter out noise words from the documents you read, when gathering keywords.
- A `docs.txt` file that has a list of all documents (in this case `AliceCh1.txt` and `WowCh1.txt`) from which the search engine should extract keywords.

NOTE: You will need to write your own driver to test your implementation. This driver can take as inputs a file that contains the names of all the documents (such as `docs.txt`), as well as the `noisewords.txt` file. It can then set up a `LittleSearchEngine` object and call its methods as needed to test the implementation. The `docs.txt` and `noisewords.txt` filenames will be sent in as the arguments to the `makeIndex` method in `LittleSearchEngine`.

Following is the sequence of method calls that will be performed on a `LittleSearchEngine` object, to index and search keywords.

- `LittleSearchEngine()` - Already implemented.

  The constructor creates new (empty) `keywordsIndex` and `noiseWords` hash tables. The `keywordsIndex` hash table is the MASTER hash table, which indexes all keywords from all input documents. The `noiseWords` hash table stores all the noise words. Both of these are fields in the `LittleSearchEngine` class.

  Every key in the `keywordsIndex` hash table is a keyword. The associated value for a keyword is an array list of (document,frequency) pairs for the documents in which the keyword occurs, *arranged in descending order of frequencies*. A (document,frequency) pair is held in an `Occurrence` object. The `Occurrence` class is defined in the `LittleSearchEngine.java` file, at the top. In an `Occurrence` object, the `document` field is the name of the document, which is basically the file name, e.g. AliceCh1.txt.

- `void makeIndex(String docsFile, String noiseWordsFile)` - Already implemented.

  Indexes all the keywords in all the input documents. See the method documentation and body in the `LittleSearchEngine.java` file for details.

  If you want to index the given sample documents, the first parameter would be the file `docs.txt` and the second parameter would be the noise words file, `noisewords.txt`

  After this method finishes executing, the full index of all keywords found in all input documents will be in the `keywordsIndex` hash table.

  The `makeIndex` methods calls methods `loadKeywordsFromDocument` and `mergeKeywords`, both of which you need to implement.

    - `HashMap<String,Occurrence> loadKeywordsFromDocument(String docFile)` - You implement.

This method creates a hash table for all keywords in a single given document. See the method documentation for details.

This method MUST call the `getKeyword` method, which you need to implement.

- **String getKeyword(String word)** - You implement.

    Given an input word read from a document, it checks if the word is a keyword, and returns the keyword equivalent if it is.

    FIRST, see the method documentation in the code for details, including a specific short list of punctuations to consider for filtering out. THEN, look at the following illustrative examples of input word, and returned value.

Input Parameter	Returned value
distance.	distance (strip off period)
equi-distant	null (not all alphabetic characters)
Rabbit	rabbit (convert to lowercase)
Through	null (noise word)
we're	null (not all alphabetic characters)
World...	world (strip trailing periods)
World?!	world (strip trailing ? and !)
What,ever	null (not all alphabetic characters)

Observe that (as per the rules described in the method documentation), if there is more than one trailing punctuation (as in the "World..." and "World?!" examples above), the method strips all of them. Also, the last example makes it clear that punctuation appearing anywhere but at the end is not stripped, and the word is rejected.

Note that this is a much simplified filtering mechanism, and will reject certain words that might be accepted by a real-world engine. But the idea is to not unduly complicate this process, focusing instead on hash tables, which is the point of this assignment. So, just stick to the rules described here.

- **void mergeKeywords** - You implement.

Merges the keywords loaded from a single document (in method `loadKeywordsFromDocument`) into the global `keywordsIndex` hash table.

See the method documentation for details. This method MUST call the `insertLastOccurence` method, which you need to implement.

- **ArrayList<Integer> insertLastOccurrence(ArrayList<Occurrence> occs)** - You implement.

    See the method documentation for details. Note that this method uses binary search on frequency values to do the insertion. The return value is the sequence of mid points encountered during the search, using the regular (not lazy) binary search we covered in class. This return value is not used by the calling method-it is only going to be used for grading this method.

    For example, suppose the list had the following frequency values (including the last one, which is to be inserted):

    ```
    --------------------
    12  8   7   5   3   2   6
    --------------------
     0   1   2   3   4   5   6
    ```

    Then, the binary search (on the list *excluding* the last item) would encounter the following sequence of midpoint indexes:

<div align="center">2   4   3</div>

Note that if a subarray has an even number of items, then the midpoint is the last item in the first half.

After inserting 6, the input list would be updated to this:

```
--------------------
12  8  7  6  5  3  2
--------------------
 0  1  2  3  4  5  6
```

and the sequence 2 4 3 would be returned.

If the new item is a duplicate of something that already exists, it doesn't matter if the new item is placed before or after the existing item.

Note that the items are in DESCENDING order, so the binary search would have to be done accordingly.

- `ArrayList<String> top5search(String kw1, String kw2)` - You implement.

  This method computes the search result for the input "kw1 OR kw2", using the `keywordsIndex` hash table. The result is a list of names of documents (same as name of the text file for that document), limited to the top 5 in which either of the words "kw1" or "kw2" occurs, **arranged in descending order of frequencies.** See the method documentation in the code for additional details.

  As an example, suppose the search is for "deep or world", in the given test documents, `AliceCh1.txt` (call it A) and `WowCh1.txt` (call it W). The word "deep" occurs twice in A and once in W, and the word "world" occurs once in A and 7 times in W:

  ```
  deep:  (A,2), (W,1)
  world: (W,7), (A,1)
  ```

  The result of the search is:

  ```
  WowCh1.txt, AliceCh1.txt
  ```

  in that order.

  NOTE:

    - If there are no matches for either keyword, return null or empty list, either is fine.
    - If a document occurs in both keywords' match list, consider the one with the higher frequency - do NOT add frequencies.
    - Return AT MOST 5 non-duplicate entries. This means if there are more than 5 non-duplicate entries, then return the five top frequency entries, but if there are fewer than 5 non-duplicate entries, then return all of them.
    - If a document in the first match list (for the first keyword) has the same frequency as a document in the second match list (for the second keyword), and both are candidates for inclusion in the output (they are not the same document), then pick the document in the first list before the document in the second list.

You may NOT MAKE ANY CHANGES to the `LittleSearchEngine.java` file EXCEPT to (a) fill in the body of the required methods, or (b) add private helper methods.

You may NOT MAKE ANY CHANGES to the `Occurrence` class (you will only be submitting `LittleSearchEngine.java`). When we test your submission, we will use the exact same version of `Occurrence` that we shipped to you.

Grading

Method	Points
getKeyword	13
loadKeywordsFromDocument	20
insertLastOccurrence	12
mergeKeywords	25
top5search	30

When a method is graded, the correct versions of other methods will be used. Also, all data structures will be set to their correct (expected) states before a method is called.

You need not do any error checking in your program for bad inputs.

Submission

Submit your `LittleSearchEngine.java` file.