CS 200, Spring 2014: Assignment 4
Hash Tables and finally Document Retrieval

Due Wednesday 11/19/14 by Noon

This assignment has two purposes: to practice hash table implementation and to handle multi-word queries. For the hash table part, as with the BST, you should be able to limit your modifications, as much as possible, to the WebPages class.

New Data Structure: HashTable

Hash tables can provide close to constant time access to data retrieved by key. So for this assignment, you will be implementing the term index using a hash table. Essentially, you should be re-doing what you did to replace the ArrayList implementation with the BST, but now you are replacing the BST with the hash table.

Some specifics:

· The hash table should be implemented with the *quadratic probing* as the method to deal with collisions. This makes it distinct from the standard Java implementation (see http://download.oracle.com/javase/1.5.0/docs/api/java/util/Hashtable.html for the API description).

· With quadratic probing, it can be difficult to tell if every possible position has been checked to find a value. So when searching for a key, in addition to the standard check for whether the next probed position has a value or RESERVED in it, you should count the number of probes and terminate if the number matches the size of the array (this is to prevent potential infinite loops).

· You should use the `hashCode()` method available in Java for String to determine the hash value for each term... with two caveats. hashCode is case sensitive; consequently, you need to convert your string to lower case before generating the hash code for it. hashCode can return negative values, so take the absolute value.

· Your hash table size will be read in as the first line in the input file. For grading it is critical that all students use the same array size and allow us to modify it as part of the test case. So the hash table constructor should take the size as an argument.

· By its nature, the hash table will mangle the ordering of the data. So be it, that's how we can tell your hash table is working properly.

Your hash table should implement the following interface:

```
public interface TermIndex {
     public void add(String filename, String newWord);
     public int size();
     public void delete(String word);
     public Term get(String word, Boolean printP);
}
```

Note: these methods have the same signatures as the ones for BST. So if you follow this interface, you should minimize the disruption to your existing code. In this case, the Boolean won't be used in get (it was there to print the depth in the BST).

You may have other methods in the HashTable class as well. The add method will need to expand the size of the array and re-hash all the entries when it approaches becoming full. Use a threshold of 80% full as the trigger for when to re-build the hash table. The code should calculate the next size using the following equation: `new_size = (2 * current_size) + 1`

You will also need an iterator to traverse the array for multiple reasons, e.g., printing, computing similarity (see below). The iterator should skip over array entries that do not contain data (i.e., null and "RESERVED" positions).

pruneStopWords Returns

Your code needs to be able to delete words from the hash table as well as search and add. So you need to add pruneStopWords back into the WebPages class. Your code will not need to find the most frequently appearing words. Now, its argument will be a String for the word that needs to be removed. The input format is being changed to include a set of words to be removed from the TermIndex.

The stop words will immediately follow the files in the input file. As with the files, each word will be on a separate line and will conclude with a flag: ***STOPs***. Call the method pruneStopWords for each of the words found in the input file.

Multi-Word Document Retrieval!

At this stage, we have most pieces in place to retrieve documents using multi-word queries. Your code will compute a pairwise similarity metric between the query and each document in the collection. Similarity is based on a measure from Information Retrieval literature called Cosine Similarity. Warning: the procedure for computing this is rather complicated. Follow the instructions carefully.

For those mathematically inclined, imagine that each document and the query are represented as vectors with each term as a dimension and the TFIDF value for the term/document pair as the value in that dimension. Similarity is the cosine between the query and each document vector. The one with the biggest value is considered to be the most similar. The equation is:

$$Sim(d,q) = \frac{\sum_{i=1}^{t} w_{i,d} \times w_{i,q}}{\sqrt{\sum_{i=1}^{t} w_{i,d}^2} \times \sqrt{\sum_{i=1}^{t} w_{i,q}^2}}$$

$$w_{i,d} = tf_{i,d} \times \ln(n/df_i)$$

$$w_{i,q} = .5 \times (1 + \ln(n/df_i))$$

d is a particular document (filename, webpage), *q* is the query, *t* is the number of unique terms found across all documents. The weights per term and document (*wi,d*) is what you computed in PA3 (*tf x idf)*. *n* is the number of documents read in. df_i is the number of documents that contain the particular term *i*. Use `Math.log` for ln.

For those not mathematically inclined, follow the directions below carefully! For those who are interested, look up "cosine similarity" and read any of the tutorials on the subject available on the Web.

A query is a set of words. Each query will be on a separate line in the input file; multiple words may be on the same line.

Supportive Data Structures

For each query, the Sim equation is computed for each document. Then the document with the highest value is chosen as the best page for the query. To support this computation, create arrays (in mathematics, these would be "vectors") for each of the components of the Sim equation:

- `docs`: supports a mapping between the positions in the component arrays and which documents are being referenced [**Hint**: keep it sorted for easy access]
- `common`: keeps the numerators
- `docSpecific`: keeps the first summation in the denominators

The size of these arrays will be equal to the number of documents that have been read in. So the first position in `docs` will hold the name of the document that corresponds to the values in the first position in `common` and `docSpecific`. Initialize all positions in the weight arrays to 0.

The second summation in the denominator is a scalar (single value) because there is only one query and so can be accumulated in a variable of type `double`. In the procedure below, this variable is called `queryWeights`.

Algorithm for Computing Cosine Similarity

You will need to add a new Method to WebPages: `bestPages` which determines which page is most similar to a query (i.e., the doc with the highest value of Sim(doc, query)) and also provides the cosine similarity computation for it.

My algorithm for the similarity comparison is:

1. Create arrays to hold the various weights and summed weights as specified above.
2. Traverse over the term index. For each term *i*:
    a. If the term is in the query, compute wi,q as above (meaning use the wiq equation and *square it*) and add it to `queryWeights` [Note: `queryWeights` is only used to compute the second term in the denominator]
    b. For each document *d* that contains term *i:*
        i. Compute the tfidf value (wi,d), square it and add it to the value in `docSpecific` in the position for doc *d* [**Hint**: you should be able to use `whichPages` from PA3 or something very much like it to do this part]
        ii. If the term is in both query and document *d*, mutiply wi,d with wi,q and add it to the value in `common` in the position for document *d*
3. For each document *d*:
    a. Compute sim(d,q) with: common[d] / (sqrt(docSpecific[d]))*(sqrt(queryWeights))
    b. Keep track of which document has the highest sim value
4. Return the document name and sim of the document that has the highest sim value. Note: in the case

of ties, return the document name that is towards the end alphabetically (i.e., assuming you are traversing the docs in alphabetical order, return the last one encountered.)

A few observations: Because similarity is relative, you really need at least three documents before documents might appear similar. If you have two identical documents only, then every position in their TFIDF vectors will be 0.

Hint: To simplify searching for terms, you should sort the words in the query and should store them together as an array or ArrayList.

Main Program: PA4

PA4 should work as before up to handling the queries. At that point, it should read queries one at a time and print the name of the document and its sim value as in the output example.

Examples

Because of the complexity of the computation, I have greatly simplified the example. Test4 references files: simple4a.txt, simple4b.txt and simple4c.txt. It should produce this output. For those who are excel savvy, this spreadsheet shows each of the calculations.

Other Hints

You need to carefully consider how to divide up the work and how to proceed. I recommend starting by making sure your TFIDF computation from PA3 is correct. You can implement and test the HashTable as a unit and then within PA3 before combining it with the enhancements here. Test the bestPages method on very simple cases first! Create some test cases before starting to code so that you think through what you need to program. Try creating your own examples using the spreadsheet as a template.