TextAssociator

CSE373 Summer 2016
Due Tuesday, July 19th

Overview

Your task is to write a general tool that will allow you to maintain one-directional word associations. In order to do so you will be implementing a `TextAssociator`, which maintains word associations as a built-from-scratch `HashTable` of `WordInfo` objects. Your final implementation will be a self-resizing, non-generic `HashTable`, using separate chaining as a collision resolution.

Once you have completed your implementation you will test the functionality and behavior of your TextAssociator by writing client code that leverages your collection of word-associations.

You have been given skeleton code to assist you in the design and structure of this data structure, however many design choices will be left up to you.

To obtain the required files, in your eclipse workspace:
file >> import >> general >> Existing Projects into Workspace >> Select archive file >> hw2_TextAssociator.zip

Files given to you:

`TextAssociator.java`

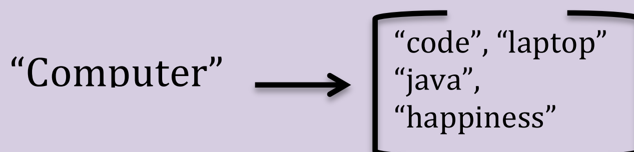
TextAssociator is Dictionary implemented using hashing in order to maintain associations between Strings. TextAssociator is mainly skeleton code for you to implement. You are to use an array of WordInfoSeparateChain objects to maintain your hashing structure of your TextAssociator. Certain methods in WordInfoSeparateChain also need to be implemented.

`ThesaurusClient.java`

ThesaurusClient uses your TextAssociator to maintain associations between words and their synonyms. Included in the provided files, you will find `simple_thesaurus.txt`, and `large_thesaurus.txt`. These two files specify the relationship between words and their synonyms. Each line has a list of comma-separated words. The first word in each line is the source word, and the remaining words on that line are the synonyms. You can specify which thesaurus file you want to use by updating the `THESAURUS_FILE` class constant.

`WordInfo.java`

WordInfo represents a relationship between a source String and a collection of Strings it should be associated with. This class is fully implemented for you.



1) Implement the method Stubs in WordInfoSeparateChain.java

Public Methods

`public boolean add(WordInfo wi) {`

Adds a new WordInfo to the WordInfoSeparateChain. If the WordInfo already exists, you should do nothing and return false. Otherwise, add it and return true.

`public boolean remove(WordInfo wi) {`

Remove the given WordInfo from the WordInfoSeparateChain. Return true if it is removed, false if it did not exist. This should remove the entire WordInfo object

2) Implement the method Stubs in TextAssociator.java

Public Methods

`public TextAssociator() {`

Constructor for a new TextAssociator. This should initializes all required fields. **Design Decision #1:** What should the starting capacity of your TextAssociator be?

`public boolean addNewWord(String word) {`

Adds a new `word` to the TextAssociator. If the `word` already exists, you should do nothing and return false. Otherwise, add it and return true. One case to keep in mind is if the appropriate index in the array index doesn't yet contain a WordInfoSeparateChain, then you will have to construct a new separate chain.

`public boolean addAssociation(String word, String association) {`

Associates the given *`word`* with the given `association`. If the given association already exists with the word, or if the word does not already exist in the TextAssociator, return false. Otherwise, add it and return true. Associations are one directional (i.e. add an association from `word->association`, not vice versa).

`public boolean remove(String word) {`

Remove the given word (and subsequently all of its associations) from the TextAssociator. Return true if it is removed, false if it did not exist. This should be removing an entire WordInfo object (not just an association).

`public Set<String> getAssociations(String word) {`

Return a set of all the associations from the given word, or null if the word does not exist.

Note: You may add other methods to these two classes as you see fit. Make sure the visibility of these methods (private, public, etc) makes sense.

Implementation Notes

-You will notice that the public interface to the client deals strictly with Strings, your WordInfoSeperateChain stores WordInfo objects behind the scenes. It will be your job to convert between Strings and WordInfo objects for varying method calls.

-Your TextAssociator must be able to store an arbitrary number of WordInfo object. Remember that because we are using separate chaining, in theory we would never have to resize our array. However, as discussed in class, having a large load factor can start to degrade your runtime for many operations. When the load factor for your TextAssociator reaches a certain threshold, you should resize your internal Array. **Design Decision #2:** At what load factor should you expand your internal capacity? Remember that you must recalculate the destination of each WordInfo object when you expand your array. **Design Decision #3:** What should the new size of your array be?

- Your WordInfoSeparateChain is a private inner class. Clients of this program should never interact directly with an Instance of this class, and should not know that it exists (neither in comments or public interface).

-There will be some redundant code between your public methods; it might help to make some private helper methods to clean up your code.

-Feel free to add additional public functionality that you think would be useful for a client (this is specifically applicable for part 3 of this assignment (see below)

Mutability: Be careful with methods such as WordInfoSeparateChain's getElements(), or WordInfo's getAssociations(), as they return references to their internal fields. This means a client (in this case, YOU) can directly modify the fields. While this may be helpful, you should be careful what operations you perform, so you don't introduce bugs into your code.

Hints

-Take a look at the prettyPrint() method in TextAssociator.java to help get some hints of how your internal structure should look

-Read the private inner class WordInfoSeparateChain and WordInfo.java very carefully and make sure you understand the methods being provided. They will assist in your implementation.

-This project does not require a lot of code! The sample solution for TextAssociator.java is less than 250 lines (including comments and starter-code). Make sure you fully understand how hashing and separate chaining work before you start trying to write code. Slides from class and the book are great resources.

2) ThesaurusClient

Once you think your implementation of TextAssociator is working, you can test it by running ThesaurusClient. A very simple test case to verify that your TextAssociator is working properly is to run this program with "simple_thesaurus.txt". This text file contains a handful of words that will be replaced with synonyms when you run the program. A simple test case is to input the following:

Input: `"my code is really good and I am very smart"`
Output: `"my code is absolutely marvelous and I am bona fide brainy"`

From this example you can see that "really" was replace with "absolutely" because our text file specified that "absolutely" was the only synonym for "really". You can also see that our simple_dictionary.txt did not specify any synonyms for the word "code", so it was left unreplaced in our output.

Once you are convinced that your TextAssociator is working properly (and properly resizing), run the program and input the following sentence exactly as follows (using large_thesaurus): "hello world it is fun to write code and have fun with data structures". Include the response from in your write-up. Keep in mind your sentence will be probably be nonsensical and will be different each time you run it. Include your favorite in the write-up ☺

Note: ThesaurusClient is not very robust and doesn't work very well with punctuation

3) Creating your own client code

Now that you have seen how one client could use your TextAssociator, your next task is to create your own client code that will use your TextAssociator in a different way. You will create a file named `MyClient.java` that has the following requirements:

1) Initializes and populates a TextAssociator object with at least 20 associations
2) Uses said associations to accomplish some goal (i.e. must be making calls to .getAssociations())
3) Outputs some text to System.out explaining what your client code is doing, etc.

You can use ThesaurusClient as a helpful example in writing your client code if you would like. Please write a very explanative class comment on you're `MyClient` and in your write-up explaining what your client does and how your TextAssociator was used in order to accomplish this goal.

If you are having trouble coming up with ideas, think about the following:

-spellchecker, contact list, auto-complete tool, etc.

4) Write up, please answer the following questions about your implementation.

Please keep your answers concise

1) For each of the Design Decisions mentioned above, please discuss possible options that you considered, what you ended up choosing, and why.

2) What hash function did you choose for your TextAssociator (i.e. did you did you use String's hashcode method, did you make your own)? Why was this hash function effective, are there alternative hash functions that you considered?

3) We chose to implement this TextAssociator with separate chaining, if you were instead going to use a different collision resolution scheme, what would you choose? How and where would your code change? Give several specific examples to illustrate your understanding.

4) How long did you spend on this assignment? What portion did you find most/least challenging?

Submission Information

You will submit `TextAssociator.java`, `MyClient.java`, and your discussion questions in `README.txt` (please make sure it's a .txt!) to the homework 3 dropbox (https://catalyst.uw.edu/collectit/dropbox/hzahn93/38544). You should not modify any files other than TextAssociator.java.