Homework 5: Web Search – Saving/Caching Query Results

Due **Tuesday December 6 11:00pm** via InstructAssist. Submit to either *Hwk5-Core* or *Hwk5-Full* as appropriate.

In a lab earlier this term, you organized URLs into a hierarchy of categories, as a way to help people find information online. Although early web-search systems were designed and maintained this way, the rapid increase in websites made manual organization impossible. In addition, there isn't a clean category hierarchy for all pages: pages can be about all sorts of topics at once, with too many nuances for hierarchies to make sense.

Today, search engines don't try to organize the pages. Instead, they store the results of previous queries to speed up search: if the same query comes up a second time, the results can be returned immediately, rather than having to search the web (this is why web search is so fast, despite all the pages out there). The technical term for saving the results of previous computations is *caching*.

This week, you will implement the core of a simple web search engine that caches results of previous queries. We'll deal with only one user (who might do the same search multiple times), and only use simple keyboard/console input/output (rather than a web browser). These should, however, suffice to let you understand the core of how a modern search engine organizes its information.

This assignment will also help you practice encapsulation, and the general separation of a system into different classes for the user interface (a.k.a. *view*), underlying data (a.k.a. *model*), and core operations (a.k.a. *controller*). The banking system from this week's lectures give you an example of this separation.

Homework 6 will build on homework 5. You should work with the same partner (if any) for homeworks 5 and 6.

If you are planning to go on to 3000- or 4000-level CS classes, this would be a good time to switch over to Eclipse or IntelliJ if you've been using DrJava all term.

This assignment can be done at two levels (Full or Core), depending on which grade you are aiming for in the course. See Assignment Options Based on Desired Course Grade for details. If you work with a partner, you must both be doing the same level on this assignment.

1 Overview and Starter Files

The starter files provide a very rudimentary "search engine" (in `SearchEngine.java`): it presents a login screen that gives you the choice of visiting a "page" (a file, not a normal URL) or running a query. At a high level, your tasks on this assignment are to:

- Create and manage data structures for caching/using the results of previous web queries,
- modify the search engine methods to use your data structures,
- while using exceptions to report visits to non-existing "pages",

- and producing clean, encapsulated code.

1.1 What is a Query?

In our web-search system, a query will just be a simple string like "WPI" or "what is pizza". A webpage *matches* a query if either its title or its body contains the query (as a substring, regardless of capitalization). **Queries should not include punctuation** (like question marks) – our system isn't sophisticated enough to handle punctuation properly. As we describe in Step 1 below, we will strip meaningless words (like "what is") from queries before we run/store them (mimicing what real search engines do).

1.2 How Should Queries Work?

The `screen` method in the initial `SearchEngine.java` file shows that our engine/browser provides main two operations: finding pages that match a query, and visiting a webpage. In the starter code, a user can select these operations, but the code doesn't do anything meaningful (you need to implement these operations). This section explains what each operation *should* do.

1. When the user runs a query (the `runQuery` method), the search engine should check whether that query has been run before. If it has, the engine simply returns the saved/cached results.

2. If the query hasn't been run before, the engine should check all pages that it knows about, gathering up those that match the query. The engine will return a list of those pages, saving the results in case the same query is made again later.

3. When the user visits a webpage (the `visitPage` method), the engine should check whether that page has been visited before. If not, the engine will check each saved query against the new page contents, and add the new page to the saved results for any query that matches the new page. The history of visited pages gets reset every time you start up the search engine (i.e., compile or run the program). **We are NOT trying to remember visit history across runs of the code**.

   Clarification added 12/4, 8:20pm: If a page hasn't been visited before, `visitPage` should add the page to the collection of pages that it knows about. You may create additional data structures to track known pages, or extract that information from other data structures you already have. That choice is up to you. There is no points penalty for creating additional data structures, as long as your choices are reasonable for the information you are trying to store.

1.3 What are "Pages"?

To enable you to write your own test cases, we are using a simple file format to simulate web pages. The web pages live in a directory called `PageFiles`. The syntax for files is a very limited subset of a language called Markdown (which is now a standard tool for lightweight document formatting). The page title should appear on the first line, which starts with the # symbol. All the following lines are the body of the page. Links to other "pages" (which we will largely ignore until hwk6) are written in the form

```
  [text](filename)
```

The file `PageFiles/aboutWPI.md` shows a full example.

The starter files include a class `SimpleMarkdownReader.java` that reads in one of these files and returns a `Webpage` object. You do not need to understand how this class works. The `addSite` method in the initial `SearchEngine.java` file uses this class to read files into webpages.

The `SearchEngine` constructor takes as input a list of initial "pages" (files in a directory within the starter code), and calls `addSite` on each page/filename.

2 Detailed Problems

Warning: Do this assignment in stages, making sure that your program runs properly after each stage. If you try to make all of the changes at once, you'll spend far longer than needed on this assignment. Build up to your solution step by step. You can get much more credit for fully working early steps than for having most of the steps only partly working.

If you want to test your program design skills, spend a little time thinking about how you might do this before reading how we break this assignment into steps below.

As you work on these problems, you may modify any of the methods in the starter file, as long as your code continues to implement the `ISearchEngine` interface provided in the starter code. You may also add fields and/or methods to the initial `Webpage` class.

2.1 Step 1: Store Previous Search Results (Core and Full)

Create a data structure that stores the webpages that match individual queries. The choice of data structure is up to you (but you will be graded on whether your choice is reasonable).

Eliminating "junk" words from queries: In practice, people often put meaningless "extra" words in queries, such as "what is" in "what is pizza". The search engine should strip those "filler" words out of the queries before finding or saving webpages that match the query. The starter code includes a method called `stripFillers` that will remove a pre-defined set of filler words from your queries (please don't edit this list, as our reference solution/autograder assumes the one in the starter code).

Our "junk works" processor does not handle punctuation. Simply avoid punctuation (like ? at the end of queries) when writing queries (we will do the same in testing your code).

Making queries case-insensitive: The two queries "pizza" and "Pizza" should return the same results, but by default Java String operations pay attention to case. To avoid this, you should convert all queries to lowercase before storing and matching them. The built-in `String` method `toLowerCase` will do this for you. You just need to figure out where to call it.

2.2 Step 2: Run Queries (Core and Full)

Edit the `runQuery` method in `SearchEngine.java` (in the starter file it simply returns null) to provide the behavior described in items 1 and 2 in How Should Queries Work?.

2.3 Step 3: Visit Pages (Core and Full)

Edit the `visitPage` method in `SearchEngine.java` (in the starter file it simply returns null) to provide the behavior described in item 3 in How Should Queries Work?.

2.4 Step 4: Include Appropriate Access Modifiers (Core and Full)

Put an appropriate access modifier (`public`, `private`, or `protected`) on each field and method in your submitted files.

[Update 12/1, 2:50pm] We strongly recommend that you get the code working through step 4, write your tests, *save a copy of your code* and then go on to step 5 (if you are doing full). Neither step 5 nor step 6 affect your tests.

2.5 Step 5: Encapsulate the Data Structures (Full Only)

Ideally, the search engine methods should not know the details of which data structures you chose for storing queries. Reorganize your classes (creating whatever additional classes and/or interfaces you want) to hide details of the query-cache data structure from the `SearchEngine` class.

Do NOT change the inputs to the `SearchEngine` constructor (or you will break autograding/testing). Similarly, make sure your code continues to satisfy the `ISearchEngine` interface.

[UPDATE 12/1, 2:15pm] You only need to encapsulate the data structure for storing queries. You may (but do not need to) also encapsulate any other data structures that you create. (Mainly, we are looking to see you do one example of this as evidence that you understand it, and we're standardizing on the query data structure.)

2.6 Step 6: Separate the User Interface from the Search Engine and Data Structures (Full Only)

The starter file put the entire web browser (engine and user-interface) into a single class. Step 5 isolated the data structures. For this step, separate out the user-interface (the part that communicates with the user, such as the `screen` method and what it depends on). Put the interface into a class called `BrowserWindow`. The constructor inputs for this class are up to you (we will auto-test your `SearchEngine` class, not your `BrowserWindow` class).

3 Testing (Full and Core)

Throughout the steps, you should be developing a good set of tests for the `SearchEngine` class. You do not need to test the user-interface (`screen` method), but you should test the `ISearchEngine` methods relative to your data structures.

Separate your tests that against the `ISearchEngine` interface from those against your own code. The autograder will "pick up" all tests in classes that have `Example` or `Test` as part of the classname. If tests in such classes reference a specific data structure within your code, your tests won't compile against our reference solution (which may use different data structures).

You can have tests against your own methods (a good idea) as long as you put them in a separate class that doesn't have `Example` or `Test` as part of the classname.

You may add new `.md` files to `PageFiles` as part of your tests (including ones that we autograde). You will submit your `PageFiles` directory as part of your solution.

We will grade your tests as usual: running your tests against our correct solution and several broken ones. The broken tests will violate the behavior of the `runQuery` and `visitPages` methods as described in How Should Queries Work?.

4 What to Turn In

Submit a `zip` file containing all of your classes (code and tests), as well as your `PageFiles` directory (at the same level as your `.java` classes). You do not need to submit our original `SearchEngine.java` file – we expect that you will have replaced it with your own.

5 Grading Expectations

This assignment will earn points on all four course themes. The Data Structures theme will look at whether you made good choices for how to organize the data for the search engine. The Program Design theme will look closely at encapsulation (for those doing the Full level), as well as your use of access modifiers (full and core levels). Java Programming and Testing will be graded largely by the autograder, as usual.

Solutions that fail to compile against the autograder (after the resubmit window if appropriate) will take a 10% deduction across all themes.

6 Assignment Options Based on Desired Course Grade

This assignment can be done at two "levels", which differ in the program design criteria that you try to meet (which in turn affects the course grade you can earn).

- The *full* level includes both data-structure encapsulation and separating interface from system code (as we did this week for the banking system).
- The *core* level lets you focus on basic Java programming and testing, skipping encapsulation and leaving the interface in with the rest of the code. **Doing the core option will cap your overall course grade at B.**

The core option is designed for those who are struggling in the course and are more concerned about passing the course than with getting a high grade. If you are in this group, we'd rather see you master the basics than do a poorly-quality job on more material.

You cannot get an A in the course without attempting the full level. This does not mean that you need a particular score on the assignment to get an A, but we need to see you making a good faith attempt at encapsulation and interface organization as part of earning an A.

The subsection headings for the individual steps clearly mark which ones are required for each of full and core.