Homework 6: Web Search – Ranking Search Results

Due **Wednesday December 14 1:30pm (early afternoon)** via InstructAssist.

Note the due date: You can have until 1pm (early afternoon) on Wednesday, but we will not have many office hours Wednesday. We're trying to strike a balance between having homework and the final due on different days, while still leaving us time to run the autograding check before you all leave for winter break on Thursday. There is no reason you can't submit the assignment as usual on Tuesday if you wish.

In Homework 5, you wrote a basic search engine that caches the results of previous queries. The `runQuery` method returns a list of `Webpage` objects, but in no particular order.

In a real search engine, search reesults get ordered by several criteria. In hwk6, we add two criteria for ranking webpages, and modify `runQuery` to return the list of pages sorted according to these criteria. We also practice exceptions, in the context of one of the ranking criteria.

You should start with your homework 5 code, extending it with the material for homework 6. You may continue to fix errors from hwk 5 as you work on hwk 6.

Note: This assignment looks long, but that's more because we are trying to break this down carefully so you can work step by step. Many of these steps are fairly straightforward to implement. The code for this week is not harder than last week. There are just several pieces to fit together.

1 Ranking Criteria

On this assignment, two criteria will affect a page's position in a results order:

- Whether many other pages link to this page (a measure of how valuable others think this page is) – this is the essence of the PageRank algorithm that Google uses (though we are implementing only a simplified version of it).
- Whether the page is *sponsored* by an organization that pays money for its pages to be ranked higher.

High level, your job on this assignment will be to add data structures that track sponsors, to compute the rank of pages according to these two criteria, and to sort the results of `runQuery` accordingly. You will also provide tests showing that all of this works.

Breaking this computation down into smaller methods goes a long way to getting this working. If you write small methods that you can test (either with JUnit tests or informally), it will be much easier to confirm that your methods are indeed working as expected.

To help us in grading, please put any new methods that you add for this assignment at the bottom of the corresponding class. For example, we should be able to find all new `SearchEngine` methods for hwk6 at the bottom of your `SearchEngine` class.

2 Starter File Additions and Modifications

The starter files for this week contain one new file and two updated ones (neither of which you should have edited for last week anyway).

- `ISearchEngine.java` has an additional method called `updateSponsor` (needed for Step 3 and Step 5 – you may comment out the entire line or the `throws` statement until you get to these steps).
- `SimpleMarkDownReader.java` has been modified to (a) leave a space between each word in the body, and (b) to have no duplicates in the list of `referencedURLs`.
- `AdminWindow.java` is new, and provides a keyboard interface for interacting with one of the new methods for this week. This will initially yield a compilation error due to uncaught exceptions. To avoid that, comment out the call to `updateSponsor` until you are ready to start working with the exceptions.

2.1 Step 1: Add a `rank` field to Webpages

Add a field of type `Double` to the webpage class that will store the "rank" of a page. Later steps will tell you how to compute the rank for a specific page.

Added 12/10, 4pm: Name the field `rank`.

Added 12/10, 4pm: To enable testing, leave this new rank field `public`. By agreeing on this, your tests and ours can access the rank of a webpage directly when writing tests. If you write a getter, our solution might not know about it, which would make our tests fail.

Added 12/10, 4pm: Similarly, leave the `url` field public and use that rather than any getters you might have written when writing test cases (if you need/want to check the URL).

Added 12/10, 4pm: Do NOT edit the `Webpage` constructor to also take the rank field, as this will break other parts of the starter code. Instead, set the rank field to a default of 0 within the `Webpage` constructor.

2.2 Step 2: Sort pages by their ranks

Add a method to your `SearchEngine` class that takes a `LinkedList` of webpages and returns a list of those same pages sorted in decreasing order of the "rank" field that you added to webpages in step 1. If your data structures are such that you want sort to take a different type of input, that's fine. It must, however, still return a `LinkedList` of webpages.

You do not need to implement sorting manually for this. Java provides a `sort` method that depends on `compareTo` for the class you are sorting over. To use it, you need to:

1. include `import java.util.Collections;`
2. have `Webpage` implement the interface `Comparable<Webpage>`

3. Assuming `myPages` is your original list, use `Collections.sort(myPages)` to do the sorting. This changes the order of the pages within the original list. It does NOT return a new list.

Note: Even though we haven't told you how to compute actual ranks for pages yet, you can finish these two steps and test that sorting is working on handmade webpage objects before you proceed.

2.3 Step 3: Add Data Structures for Sponsors

Organizations (such as WPI) want to be able to pay money to have their pages appear higher in query result lists. We need to add data structures to manage this, and to update the webpage ranks based on sponsorship.

Specifically, we need a data structure that stores (a) the name of the sponsor (say "WPI") and (b) a `Double` indicating the rate the sponsor will pay per page returned in a query result. Normal rate values are no more than `.1` (for ten cents per page). A sponsor has only one rate that applies across all of their pages.

1. Create a data structure for storing sponsor rate information. The choice is up to you.

2. For additional points, encapsulate this data structure in its own class (this is an informal version of a Core vs Full assignment – you can still get a good grade on the assignment for skipping this, but it will cost a few program design points.)

3. In the `SearchEngine` class, add a method `updateSponsor` that takes a sponsor name and a new rate (a `double`) and sets the sponsor to have the given rate. The method either adds the sponsor to the data structure (if the sponsor is new) or updates the existing rate (if the sponsor already has a rate).

   Note: you can check whether a key is already in a HashMap by using the `containsKey` method on hashmaps. You can also use `get` and check whether the result is null, but if that seems to result in `NullPointerExceptions` (as Kathi had writing her solution to this), know that `containsKey` is an alternative.

   This method will need to be in the `ISearchEngine` interface. If you did not add this as part of our starter file modifications, add it now.

[Added 12/10, 4pm]: Let's all agree to compare sponsor names and urls by downcasing (use `toLowerCase`) on both sponsors and urls. Hopefully this will make it easier for more tests to pass.

2.4 Step 4: Use Exceptions to Report Invalid or Lowered Rates

Three issues can arise as we update rates for sponsors: they could be invalid (negative numbers), implausible (larger than 10 cents per page), or lower than the sponsor's current rate (which the search engine company wants to confirm is intended and not a typo). We are going to introduce exceptions to handle these.

Specifically, create the following two kinds of exceptions:

- `InvalidRateException` which `updateSponsor` throws if the entered rate is lower than 0 or greater than .1 (rates of 0 are valid). The exception class should store a single double for the invalid rate.

- `LowerRateException` which `updateSponsor` throws if the sponsor already exists and the new rate is lower than the old rate.

  [Added 12/10, 4pm]: The exception class should store the old and new rates as fields.

Note that while `updateSponsor` throws both of these exceptions (as indicated by a `throws` statement on the header), the actually `throw` that creates the exception may be in another one of your methods (depending on which classes you made for your data). Practice good encapsulation here – the exception should originate from whichever method is actually about to store the updated rate in a field. If that isn't `updateSponsor`, then `updateSponsor` will simply pass the exception along.

As this point, you don't yet have anything that catches these exceptions (because nothing other than tests would be calling `updateSponsor` just yet.

2.4.1 Catching Exceptions

The starter files have a class `AdminWindow`, which provides a keyboard interface to calling `updateSponsor`. Add `try/catch` blocks to the `adminScreen` method to handle these two exceptions. In both cases, you can just print a warning to the user and restart the `adminScreen`.

If you want a bit more of a challenge (and a more realistic scenario), handle the `LowerRateException` by reporting that this is a lower rate, asking the human user to confirm this was intentional and then making the change if the human user approves (then start the `adminScreen` again). **This version is not required**, but you may find it informative and good practice with exceptions.

2.4.2 Testing (With) Exceptions

Add tests for your exceptions. When you test exceptions, you can do a simple test that the right kind of exception gets thrown, or a more complicated test that also checks the fields within the thrown exception. We'll just do the simpler version here. Here is an example:

```
// Testing an update that is too high and should throw an exception
@Test(expected = InvalidRateException.class)
public void invalidRateTest1() throws LowerRateException,
                                      InvalidRateException {
  s.updateSponsor("WPI", 2.0);
}
```

In the line with the `@Test` annotation, you indicate which class of exception you expected. The test method needs a `throws` annotation with any exception that *could* be thrown when running the code in the test method body. JUnit checks that the first exception thrown in the test body is of the `expected` class.

If you know or want to teach yourself the more advanced style of exception testing, feel free to use it. This simple form will suffice for grading (and either form will work with autograding).

[Added 12/10, 4pm] What if you want to test a method that *can* throw an exception, but won't in this particular case? The test method still needs the `throws` annotation, but, you leave off the `expected` annotation from the previous example:

```
  // test update sponsor with no exception expected
  @Test
  public void validRateTest1() throws LowerRateException,
InvalidRateException {
    s.updateSponsor("WPI", 0.05);
  }

}
```

2.5 Step 5: Modify Page Ranking based on Sponsorship

Now, we want query results to be ordered based on sponsorship.

1. Create a method called `getSponsoredRate` that takes a webpage URL string (like `"aboutWPI.md"`) and returns a `double` indicating how much sponsors are paying for this page. We will assume that each page has at most one sponsor.

   A sponsor pays for a page if the sponsor name is a substring of the url. So if `"WPI"` is a sponsor paying 5 cents per page, then

   ```
   getSponsoredRate("aboutWPI.md")
   ```

   should return `0.05`. Remember that you can use `contains` on strings to check whether one string contains another.

   If none of the known sponsor names are a substring of the given URL, the methods should just return 0.

   You get to decide what to call this method and which class this method belongs in. We will check your choice towards program design/encapuslation points.

2. Create a method that iterates over all of your pages and adds the sponsoredRate for that page to the page's current rank. You decide what to call this method and which class this method belongs in.

Stop and check whether updating the sponsoredRates and then sorting the results gives you the order you expect. You don't have to write formal JUnit tests for this, but you should make sure this part is working at this point.

2.5.1 Use SponsoredRates in `runQuery`

Now, update `runQuery` (from the `SearchEngine` class) to sort results based on sponsorship. Specifically, after you compute the list of `WebPages` that match a query:

- Reset the ranks of all the pages to 0 (so old queries don't impact new ones).
- Run the method that iterates over the pages to update their rank based on sponsorship
- Sort the pages based on ranking, and return the sorted list as the result of `runQuery`.

2.6 Step 6: Consider Page Links in Page Ranks

Google is famous for its PageRank algorithm, which prioritizes search results based on links to pages. Roughly, if many other sites reference a webpage, PageRank takes that as a signal that a page is more reliable, and puts it higher in the search results. We are going to add a simple version of PageRank to this assignment (the full version involves multiple iterations over pages, as the linked website discusses).

In our simple version, each website has one "credit" of ranking that it shares equally among the pages that it links to. Let's do this via an example. Imagine that we had three pages:

```
WPIhomepage.md: links to aboutWPI.md and WPIProjects.md
aboutWPI.md links to WPIProjects.md
WPIProjects.md doesn't link to anything
```

`WPIhomepage.md` would divide its 1 credit between the two pages it links to (giving each .5 credits). `aboutWPI.md` would give its entire 1 credit to `WPIProjects.md`. `WPIProjects.md` wouldn't give credit to any other page, since it has no outgoing links.

So at the end, `WPIProjects` would receive `1.5` ranking points based on links, `aboutWPI` would receive `.5` credits, and `WPIhomepage` wouldn't receive any credits.

Clarification Added 12/12, 9pm: The amount of credit should be based on the number of (non-self) links in a page, not the number of links to pages that have already been visited. So if Page1 links to Page2 and Page3, but Page2 has never been visited (and hence doesn't have a `Webpage` object), Page3 still gets only .5 credits from Page1. This means you can safely ignore the http-based link if you still have it in one of your `PageFile` examples from the Starter Code.

2.6.1 Writing Code to Distribute Ranking Credits

Write a method (you decide name and location) that iterates over all pages (not just those returned by a query) and distributes the link credits as in the example above. Specifically, for each page:

- Count how many references there are to other pages **excluding** self links (a page shouldn't be able to bump up its ranking by linking to itself).
- Add `1 / this-link-count` to the rating for each page that this one links to (again, excluding self-links). You may well already have a method on `Webpage` that adds to the `rank` field (from when you handled page sponsors). You can reuse that here.

2.7 Include Link Credits in `runQuery`

Extend `runQuery` to also distribute ranking credits before sorting the pages. (This should amount to adding a single call to the method you just wrote.)

3 Common/Anticipated Questions

- **How often do we recompute the rank per page?**

Recompute the rank on every call to `runQuery`. A real system would optimize this, and only recompute the ranks when needed. But this is, after all, only a 2000-level CS class. In each call to `runQuery`, just reset all the webpage rank fields to 0 and recompute the ranks based on sponsors and links.

- **Do the cached queries need to be stored in ranking order?**

  No, they do not. The list of pages that you "remember" for a query can be stored in any order. Again, this is something a real system might do to optimize performance, but it has many corner cases that would make this assignment too hard. Just re-sort the list of pages every time in `runQuery`, even if the query is already in the cache.

- **What if one page has multiple links to the same page? Does the linkedTo page get twice the credit?**

  No. The referencedURLs list in each `Webpage` object will not have any duplicates (this is a change we are making to the Markdown Reader for this assignment). You only have to filter out the self-reference (if any). You don't have to worry about duplicates.

4 Testing

For this assignment, we are focusing on the order of your queries in the output of `runQuery`. Simply add tests (to what you had for hwk5) that check for the order of queries returned by `runQuery`. Our broken implementations will make errors that affect either the ranks assigned to pages (sponsorship or links) or the sorting of pages based on ranks.

We do also want to see tests for the exceptions from `updateSponsor`. Basically, show us that you can identify the range of scenarios that need to be tested regarding the exceptions.

Please group all of your exceptions tests together, clearly marked with a comment in your `Examples` class. This will help us in grading.

5 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).

6 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 about sponsors. The Program Design theme will look closely at encapsulation and use of access modifiers. 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 15% deduction across all themes.