Final Project

COMP 250 Winter 2020

posted: Wednesday, April 8, 2020
due: Thursday, April 30, 2020 at 23:59

General Instructions

- **Submission instructions**
  - For the final project there will not be any late days. If you are not able to submit the project by April 30th, then the project will **not** be graded. We will start running the grader on the project two days before the deadline. So, the earlier you submit more chances you will have to know if something major is wrong with the project.
  - Don't worry if you realize that you made a mistake after you submitted : you can submit multiple times. We encourage you to submit a first version a few days before the deadline (computer crashes do happen and myCourses may be overloaded during rush hours).
  - Please store all your files in a folder called "FinalProject", zip the folder and submit it to myCourses. Inside your zipped folder, there must be the following files.
    * `MyHashTable.java`
    * `Twitter.java`

    Do not submit any other files, especially .class files. Any deviation from these requirements may lead to lost marks
- The starter code for all the above classes is provided. You must **not** modify the `HashPair` and the `Tweet` class. Note that for this project, you are NOT allowed to import any other class besides the ones already imported for you. **Any failure to comply with these rules will give you an automatic 0.**
- We have included with these instruction a tester class, which is a mini version of the final tester used to evaluate your project. If your code fails those tests, it means that there is a mistake somewhere. Even if your code passes those tests, it may still contain some errors. We will test your code on a much more challenging set of examples. We therefore highly encourage you to modify the tester class and expand it. A stress tester (to test the efficiency of your code) will be released in a week or so.
- You will automatically get 0 if your code does not compile.
- Failure to comply with any of these rules will be penalized. If anything is unclear, it is up to you to clarify it by asking either directly a TA during office hours, or on the discussion board on Piazza.
- Feel free to change or delete the package statements in the starter code.

Your task

For this project you will write several classes to simulate searching through a Twitter-like data base. Make sure you implement all required methods according to the instructions given below. In addition to the required methods, you are free to add as many other **private** methods as you want. Note that `null` keys are not allowed in the hash table. Remember that in most of scenarios objects comparison does not use '=='. We highly suggest you read through the entire instructions before you start coding. You do not necessarily have to implement the methods in the order in which they are presented. Note also that this project is meant for you to practice first hand how to implement a hash table as well as learning when to use it and how to exploit its properties. For this project you want to focus on optimizing the time efficiency of your code. We do not care about space efficiency. Enjoy coding this last project!

[70 points] The class `MyHashTable` has the following fields:

- An `int` for the number of entries stored inside the table.
- An `int` for the number of buckets the table has (Note that this value is initialized by the constructor, but could change later on if the number of entries increases).
- A `final double` representing the maximum load factor for the hash table.
- An `ArrayList` of buckets used to store the entries to the table. Where each bucket is a LinkedList of HashPairs.

Implement the following `public` methods in the `MyHashTable` class:

- The constructor `MyHashTable()` which takes an `int` as input representing the initial capacity of the table.[1] Using the input, the constructor initializes all the fields.
- A `put()` method that takes a *key* and a *value* as input. The method adds a `HashPair` of the *key* and *value* to the hash table. If a `HashPair` with the *key* already exists in the hash table, then you should overwrite the old *value* associated with the *key* with the new one. This method should be $O(1)$ on average. If in this hash table there was a previous value associated to the given key, then the method overwrites it with the new value and returns the old one. If there was no value associated to the given key, then the method returns `null`. Remember that the load factor of the table should never be greater than the maximum load factor stored in the appropriate field.
- A `get()` method which takes a *key* as input and returns the *value* associated with it. If there is no such key in the hash table, then the method should return `null`. This method should run in $O(1)$ on average.
- A `remove()` method that takes a *key* as input and removes from the table the entry (i.e. the `HashPair`) associated to this *key*. The method should return the *value* associated to the *key*. If the *key* is not found, then the method returns `null`. This method should run in $O(1)$ on average.

[1] The capacity is the number of buckets in the hash table, and the initial capacity is simply the capacity at the time the hash table is created.

- A `rehash()` method which takes no input and modifies the table so that it contains double the number of buckets. This method should be $O(m)$ where m is the number of buckets in the table.[2]
- A `keys()` method which takes no input and returns an `ArrayList` containing all the keys in the table. The `keys` in the returned `ArrayList` may be in any order. This method should be $O(m)$ where m is the number of buckets in the table.
- A `values()` method which takes no input and returns an `ArrayList` containing all the ***unique*** values in the table. The returned `ArrayList` of *unique* `values` may be in any order. This method should be $O(m)$ where m is the number of buckets in the table.

Notice that inside this class you can also see two `static` methods. The method called `slowSort()` is already implemented. It takes as input an object of type `MyHashTable` where the values are `Comparable`. The method returns an `ArrayList` containing all the keys in the table, sorted in descending order based on the values they map to. The time complexity of `slowSort` is $O(n^2)$, where n is the number of pairs in the table. Your should implement:

- a method called `fastSort` which performs the same task as `slowSort` but with a time complexity of $O(n \cdot \log(n))$. It is up to you to decide which sorting algorithm you'd like to implement.

Finally, implement the following methods from the `private` nested class `MyHashIterator`:

- The constructor which should be $O(m)$, where m is the number of buckets in the table.
- A `hasNext()` method which should be $O(1)$ and returns `true` if the hash table has a next `HashPair`.
- A `next()` method which is also $O(1)$ and returns the next `HashPair`.

[30 points] Implement the following `public` methods inside the `Twitter` class. Note that you are allowed to add as many `private` methods and fields as you see fit.

- The constructor `Twitter()` which takes as input an `ArrayList` of `Tweet`s and an `ArrayList` of `String`s denoting the stop words. A stop word is a commonly used word that is ignored by search engines. We suggest you first look at what you need to implement in the rest of the class, before deciding how to implement the constructor. The time complexity of this method should be $O(n+m)$ where n is the number of tweets and m is the number of stop words.
- A method `addTweet()` which takes a `Tweet` as input and adds it to the Twitter. This method should be $O(1)$.
- A method `latestTweetByAuthor()` which takes a `String` as input and returns the latest `Tweet` the given author has posted. If the author has not posted any tweet, then the method returns `null`. This method should be $O(1)$.

[2] In the slides we mention that these methods run in $O(n+m)$ where n is the number of entries, and m is the number of buckets. Note that if you have a good hash function and a max load factor of 0.75, then this is equivalent to say that the method runs in $O(m)$.

- A method `tweetsByDate()` which takes a `String` as input representing a date in this format `YYYY-MM-DD`. The method returns an `ArrayList` of `Tweet`s containing all the tweets posted on that given date. If there are no tweets on the given date, then the method returns `null`. This method should be $O(1)$.
- A method `trendingTopics()` which takes no input and returns an `ArrayList` of `String`s containing all the words (which are not stop words!) that appear in all the tweets of this Twitter. The words should be ordered from most frequent to least frequent by counting in how many tweet messages the words appear. Note that if a word appears more than once in the same tweet, it should be counted only once. This method should be $O(n)$ where n is the number of tweets. Note that tweet messages have a limit on the number of words, so iterating through the words in a message increases the time complexity of the method by a constant. Here a couple of hints:
  - You can find in the template an helper method called `getWords` which you can use to extract an array list of words out of a message. Note that the method separate the words based on apostrophes and space characters. Characters that are not letters from the English alphabet are ignored. So, for instance "@pple!" becomes "pple" and "user521dg" becomes "userdg".
  - When comparing strings make sure to ignore the case of the characters. You want to be sure for instance not to miss stop words typed in either upper or lower case.

  If you test `trendingTopics` with an object of type `Twitter` created with the list of tweets from the `HashTableTester.java` and an empty list as stop words, then these are the top four trending words: "spirit" with 16 occurrence, "a" with 15 occurrences, "that" with 8 occurrences, and "time" with 7 occurrences.

Here is a table with the breakdown of the points for all the methods you have to implement in this project:

Method name	Correctness	Time complexity	Total
MyHashTable()	5	-	5
put()	3	7	10
get()	2	6	8
remove()	2	6	8
rehash()	4	6	10
keys()	2	3	5
values()	4	6	10
fastSort()	1	5	6
MyHashIterator class	5	3	8
Twitter()	5	-	5
addTweet()	1	4	5
latestTweetByAuthor()	1	4	5
tweetsByDate()	1	4	5
trendingTopics()	4	6	10
Total	40	60	100