Homework 8: Hashing and Tries

Due by 11:59PM ET on Monday, April 20, 2020

2 Required Problems (80 points), Quantitative Analysis (10 points)

Setup and Logistics

We have put together several stub files — **click here to download them (/~cis121/current/hw/8-hashing-tries/hw8-stub-files.zip)**.

5 files to submit: `HashMap.java` , `HashMapTest.java` , `TrieMap.java` , `TrieMapTest.java` , `QuantitativeAnalysis.pdf`
Please do not submit any additional files.

Be sure to follow these guidelines:

- Ensure source files are not located in any package and that none of the files have a package declaration (e.g. `package src;`).
- When submitting, submit each file individually at the root level (not contained in a folder or zip files).
- Ensure that your own test files pass and do not run into infinite loops.
- You have unlimited submissions until the deadline on the autograder marked `[compilation]` , and **two free submissions** on the autograder marked `[full]` .
- For this assignment:
  - **Number of Free Submissions**: 2
  - **Deduction Per Additional Submission**: 10 points
  - **Max Code Coverage Deduction**: up to 10 points
  - **Max Style Score Deduction**: up to 5 points
  - **Manual TA Efficiency/Style Grading**: yes
- Do not wait until the last minute to try submitting. You are responsible for ensuring your submission goes through by the deadline.
- Before asking questions on Piazza, be sure to check and keep updated with the pinned clarifications post.
- **Gradescope issues?** Be sure to check out our **Gradescope help page (/~cis121/current/gradescope_help.html)** before posting to Piazza.

Motivation

`java.util.HashMap`: A Rite of Passage

We have used hash-based data structures throughout this class on various programming assignments, and we have proven certain properties about them in lecture and recitation. The time has now come to embark on a rite of passage that every budding computer scientist must take. It is time to implement a hash table.

But seriously, too many people go around blissfully using this magical all-operations-expected-$O(1)$ data structure unaware of how it works, how to get the most out of it, and when something else might be better. In this homework, to ensure you aren't that person, you will implement a production-grade hash-map which will conform to the Java 7 `Map` interface (see a section later below on the Java 8 changes). We provide a `BaseAbstractMap` which your implementation should extend from, which reduces the work for a few complex `Map` methods. Before you get started, take a look at the `java.util.Map` interface (https://docs.oracle.com/javase/8/docs/api/java/util/Map.html) to familiarize yourself with the API.

Tree? Trie?

A *prefix trie*[1] is an ordered tree data structure, which stores string keys by storing characters in nodes. The "prefix" part of the name comes from the fact that common prefixes between keys share the same path from the root in the trie. For a naive trie, there is a fair amount of overhead, since every character lives in its own node. One example of an optimization that addresses this object overhead issue is a PATRICIA trie (https://en.wikipedia.org/wiki/Radix_tree), in which each nodes are compressed into its parent when appropriate in order to optimize space and memory usage. Unfortunately, you will not be implementing PATRICIA tries for this assignment.

1: The term "trie" comes from re**trie**val, and is most commonly pronounced "try".

Part 0: Set up (0 points)

Important: Historically, many students have experienced issues setting up `jamm.jar` and `TestHarness.java`. Please ensure that you can get this up and running as soon as possible, since it's required for the quantitative analysis questions below. You **can**, however, do the programming part of this homework without this step.

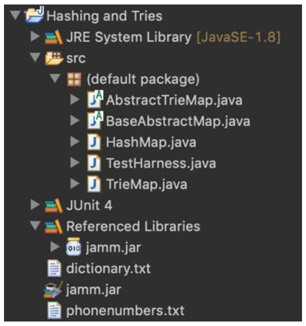
When you first import the project into your workspace, you may see some error messages associated with the `TestHarness.java` file. Here is a few setup you need to follow:

We've provided a test harness in `TestHarness.java` that records execution duration and memory usage of both map implementations. The harness tests two datasets, `dictionary.txt` (350,000 words) and `phonenumbers.txt` (8,000 phone numbers, all of the form (215)-898-xxxx, encoded as letters and thus starting with "cbfiji"). You might need to move these files to the root of your project (or wherever your working directory is, normally not in the `src` folder) to avoid `FileNotFoundExceptions`. Before you do run the harness, you should take a look at each dataset and form a hypothesis on the CPU times and memory usages of each implementation on each dataset; the results might surprise you!

There are a couple steps to set this up for Eclipse:

1. Acquire `jamm.jar` from the assignment's stub files.
2. Make sure the `jamm.jar` file is in the **root** (i.e., the folder you will click delete on if you want this whole project gone) of your project (not the `src` folder). Also, make sure the `dictionary.txt` and `phonenumbers.txt` files are at the root.
3. Right click on your project and go to Build Path → Configure Build Path → Libraries → Add JARs. Select `jamm.jar` from the file selector drop down and hit "Okay".

   If everything up to this point was done correctly, your directory should look like this:

   ```
   Hashing and Tries
     JRE System Library [JavaSE-1.8]
     src
       (default package)
         AbstractTrieMap.java
         BaseAbstractMap.java
         HashMap.java
         TestHarness.java
         TrieMap.java
     JUnit 4
     Referenced Libraries
       jamm.jar
     dictionary.txt
     jamm.jar
     phonenumbers.txt
   ```

4. Right-click on `TestHarness.java` and go to Run As → Run Configurations. In Java Application, under the "arguments" tab, add the string `–javaagent:jamm.jar` to the VM arguments field, then Apply and Run.
5. Once you've implemented the methods in `HashMap.java`, run `TestHarness.java`. It will report the CPU time and memory usage of your implementation! If you recieve an error, restart Eclipse and try again (you will use this part again in Part 3).

Note: If you are using OS X, you may receive the following error in the console:

```
Class JavaLaunchHelper is implemented in both /Library/Java/JavaVirtualMachines
/jdk1.8.0\_102.jdk/Contents/Home/bin/java and /Library/Java/JavaVirtualMachines/
jdk1.8.0\_102.jdk/Contents/Home/jre/lib/libinstrument.dylib. One of the two will be used. Which one is undefined.
```

This is fine, as the test harness will still run as long as everything else is set up properly.

Part 1: `HashMap` (40 points)

Files to submit: `HashMap.java`, `HashMapTest.java`

Recall the method of chaining discussed in lecture (https://www.cis.upenn.edu/~cis121/current/lectures/notes.pdf#page=139) and in recitation. A hash table first hashes an `Object`'s hash code into a bucket index appropriate for the length of the backing array. Each bucket is a linked list of map entries that can be added to, modified in place, or removed from. **Note that when you add an element to a bucket, you should add the element to the front of the bucket** (i.e., the head of the linked list).

We have provided a fair bit of skeleton code for you, namely the constructors, the table buckets, and the hashing methods. You need to worry about the following eight method stubs:

- `get(K key)`
- `containsKey(K key)`
- `put(K key, V value)`
- `resize(int newCapacity)`

- `remove(K key)`
- `containsValue(V value)`
- `clear()`
- `entryIterator()`

Each method stub contains further instructions. **It is critical that you read *both* the Javadoc specification (best done by using Eclipse to read the spec) *and* the implementation comments for each method as this will answer any potential questions you have about implementation details.** They contain necessary information on both the external and internal behavior of these methods, and hints on how to go about implementing them.

It is also critical that you understand the provided code and methods. You will want to pay special attention to the `threshold` and `loadFactor` variables, the `hash(int h, int length)` method, and the `Entry` inner class. Your solution will explicitly invoke these entities.

For `threshold` /resizing, you should resize when you realize that adding the *next* element would cause you to *reach* the threshold. For example if `threshold` is 3, you should resize when you're about to add the third element.

You should make sure you are correctly handling null keys and null values (for example, null keys should be hashed to index zero). Note that non-null keys can also be hashed to index zero. Be careful to explicitly handle and test those, and don't be afraid to repeat some code if you want to explicitly isolate the null cases.

Finally, `entryIterator()` can return the elements in any order, but must be lazy (you cannot just put everything into a list and return it).

IMPORTANT: Do not throw any `ClassCastException` s. Whenever the `HashMap` Javadocs say to throw a `ClassCastException` , you can safely ignore this.

Note: Java 8 HashMap Implementation

In Java 8, HashMaps actually are no longer implemented using simple linked lists for chaining. Instead, Java now uses balanced binary search trees as the buckets. This decision comes with certain trade-offs. Namely, it reduces the worst case access time for any element from $O(n)$ to $O(\log n)$ because insert, search, and delete in balanced binary search trees take $O(\log n)$ time. However, it also adds increased overhead in terms of space and does not change the expected $O(1)$ access time.

If you are interested in the specifics of the Java 8 implementation, it uses Red-Black trees (which are balanced binary search trees, meaning the tree is of height $O(\log n)$). These trees are only used when the HashMap passes certain thresholds: when there are more than 8 elements in one bucket and the number of slots in the HashTable is at least 64. For more information about these specifics, check out the discussion here (https://stackoverflow.com/questions/43911369/hashmap-java-8-implementation).

Mocking `hashCode()`

When writing unit tests for your `HashMap` , you may find it useful to force collisions between `Object` s that you are inserting in the map. Since your implementation will require the use of Java's `hashCode()` method, you will have to override this method when testing your chaining.

Suppose you want to force a collision between two objects in your unit test. You can do the following:

```
@Test
public void testCollision() {
    Object obj1 = new Object() {
        @Override
        public int hashCode() {
            return 5;
        }
    };

    Object obj2 = new Object() {
        @Override
        public int hashCode() {
            return 5;
        }
    };

    map.put(obj1, "foo");
    map.put(obj2, "bar");
    //...
}
```

An alternative, more clean approach would be to create a class where you can set the `hashCode` at construction:

```
static class MockHashObject {
    private final int hashCode;

    public MockHashObject(int hashCode) {
        this.hashCode = hashCode;
    }

    @Override
    public int hashCode() {
        return hashCode;
    }
}
```

Either of these approaches works for testing your chaining.

Part 2: `TrieMap` (40 points)

Files to submit: `TrieMap.java` , `TrieMapTest.java`

This should be a straightforward "standard" trie implementation, as described in the lecture notes (https://www.cis.upenn.edu/~cis121/current/lectures/notes.pdf#page=149). Your trie should support inserting strings which are prefixes of each other (ex: pen and penguin). We have provided a skeleton for you in the form of a nested class `Node` and a few helper methods, and we've left you to worry about the following method stubs:

- `put(CharSequence key, V value)`
- `get(CharSequence key)`
- `containsKey(CharSequence key)`
- `containsValue(Object value)`
- `remove(CharSequence key)`
- `clear()`

Each method stub contains further instructions. **It is critical that you read *both* the Javadoc specification (best done by using Eclipse to read the spec) *and* the implementation comments for each method as this will answer any potential questions you have about implementation details.** They contain necessary information on behavior of these methods, hints on how to go about implementing them, and important explanations of differences from the `HashMap` implementation.

The trickiest part of this implementation will be correct removal of keys. You might find it helpful to work out examples of each method on paper before going forward with your implementation.

Note that unlike in your HashMap implementation, your TrieMap should **not** support `null` keys or values. The empty string *is* still valid as a key, however.

If you feel the need to, you may modify the `Node` class given to you.

Kudos Points: Lazy Iterator

Files to submit: `TrieMap.java` .

Implement the `entryIterator()` method to return the entries in lexicographic order with respect to the keys. You must write this as a true lazy iterator—an implementation that simply dumps all the elements into a collection and retrieves an iterator from the collection will be awarded no credit. The iterator only stores enough state to do its job. Constraints:

- The running time must be linear in the number of elements in the trie.
- The space usage must be proportional to the height of the trie.

If this last constraint about space usage is too difficult, you can ignore that constraint for half credit. For more information, refer to the JavaDoc of the iterator.

This is for Kudos points only, and does **not** count for your grade. Therefore it is **completely optional**!

Part 3: Quantitative Analysis (10 points)

Please answer the following questions in a LaTeX typeset file called `QuantitativeAnalysis.pdf` :

Note: You must have followed the setup instructions for `TestHarness.java` and `jamm.jar` in Part 0 to be able to answer these questions.

Important: You must provide the actual program output for question 1, and actual numbers for question 5. If you do not, you will receive no credit for this section.

1. After finish your implementation of the HashMap and TrieMap, click run on `TestHarness.java` . Note: this will take a few minutes to run on your computer.

   Please copy and paste the output of `TestHarness.java` as an answer to this question, and place it inside of a verbatim environment (http://en.wikibooks.org/wiki/LaTeX/Paragraph_Formatting#Verbatim_text) in your LaTeX file.

2. For `dictionary.txt` , which implementation (between *your implementations* of `TrieMap` and `HashMap`) had a better running time? Which implementation had better space usage? What about for `phonenumbers.txt` ? Was this what you were expecting? Why or why not?

3. It was mentioned in lecture that tries are more space-efficient than hash tables due to the fact that they compress common prefixes. Did your implementation reflect this for `dictionary.txt` ? What about for `phonenumbers.txt` ? If so, why? If not, what could you potentially do to improve the memory consumption of your `TrieMap` ?

4. Based on your answer to 3, does Big-Oh notation tell us anything about the *actual* running time or space usage of an algorithm on a data set? Why or why not? What might an implication of this be for software development?

5. Add a call to `initChildren()` to the end of the `Node` constructor in `TrieMap.java` , then re-run the test harness, and consider the results for `TrieMap` and `dictionary.txt` . The difference here is that now we initialize the children array as soon as the node is constructed, instead of waiting until we actually add a child (the "lazy" way). How much memory, both absolutely and relatively, does the lazy initialization save us? (*Give actual numbers.*) Would you say the lazy initialization is a worthwhile optimization? Why or why not?

   Please remove this change once you're done answering Q5.

6. As mentioned above in the writeup, Java 8's implementation of `HashMap` s differs to what you implemented in this assignment. How do your results compare with Java's implementation of a `HashMap` in terms of both space and time? What do you attribute the differences to?

7. How do your results compare with Java's implementation of a `TreeMap` in terms of both space and time? What about the results between `TreeMap` and `java.util.HashMap` ? What do you attribute the large disparity in runtime to? Hopefully now you understand why `HashMap` s are so popular and much more widely in use :)

   NOTE: A `TreeMap` is **not** a `TrieMap` . These are very different. A `TreeMap` is implemented as a red-black balanced binary search tree.

Again, as a reminder: you must provide the actual program output for question 1, and actual numbers for question 5. If you do not, you will receive no credit for this section.

Style & Tests

The above parts together are worth a total of 90 points. Style and code coverage are graded on a subtractive basis with deductions as specified in the *Setup and Logistics* section of the writeup, and you will be graded according to the CIS 121 style guide (/~cis121/current/java_style_guide.html). Gradescope will give you your grade on this section immediately upon submission to either autograder.

IMPORTANT: Please **DO NOT** use any external files (.txt, etc.) to write your JUnit tests for this assignment. Our code coverage tool will fail, and you will not be able to submit your code due to failing test cases.

You will need to write comprehensive unit test cases for each of the classes, inner classes, and methods you implement (including helper methods!) in order to ensure correctness. Make sure you consider edge cases and exceptional cases in addition to typical use cases. Use multiple methods instead of cramming a bunch of asserts into a single test method. Your test cases will be auto-graded for code coverage. Be sure to read the testing guide (/~cis121/current/testing_guide.html) for some pointers on what level of testing we expect from you. Finally, due to the Code Coverage tools we use, **please be sure to use JUnit 4** as it is the only version supported.

Note: You will not be able to write JUnit test cases for any private methods. Instead, make them package-private by leaving off any privacy modifier (i.e., do not write `public` or `private`).

This assignment was developed by the CIS 121 Staff.
Last updated on Mon, Apr 13 at 05:48 PM.