Homework 8: Hashing

Due by 11:59PM ET on Monday, November 23, 2020

1 Required Problem (50 points)

Setup and Logistics

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

2 files to submit: `HashMap.java` , `HashMapTest.java` *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-$O(1)$ $O(1)$ 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.

Part 1: `HashMap` (50 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.

Lazy Iterators

We use iterators in general because it allows for lazy evaluation. We don't have to figure out what is next in a sequence until we actually call next and do the work, which may save us time in certain applications if we aren't iterating through an entire collection. You will have to create an inner class that implements the iterator class and have it run over your HashMap.

Your iterator must be lazy—that is, you will not receive any credit if you pre-compute everything to iterate over and run through that. You can safely assume that we will not be modifying the HashMap during iteration.

Here is an example of a lazy iterator for a LinkedList. The one you will be implementing is more involved, but this should give you the general structure.

```
private class LinkedListIterator implements Iterator<E> {
    private LinkedListNode<E> currNode;

    public LinkedListIterator(LinkedListNode<E> head) {
        currNode = head;
    }

    @Override
    public boolean hasNext() {
        return currNode != null;
    }

    @Override
    public E next() {
        if (hasNext()) {
            E currVal = currNode.get();
            currNode = currNode.next();
            return currVal;
        } else {
            throw new NoSuchElementException();
        }
    }
}
```

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 $O(n)O(n)$ to $O(\log n)O(\log n)$ because insert, search, and delete in balanced binary search trees take $O(\log n)O(\log n)$ time. However, it also adds increased overhead in terms of space and does not change the expected $O(1)O(1)$ 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 $O(\log n)$)$O(\log n)$). 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.

Style & Tests

The above parts together are worth a total of 50 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`).