2 Required Problems (75 points), 1 Optional Extra Credit Problem (10 points), Quantitative Analysis (15 points), and Style and Tests (10 points)

DO NOT modify methods, method headers, or variables that were already provided to you.
DO NOT add public or protected methods.
DO introduce helper methods that are package-private (i.e., do not have a privacy modifier) as you deem necessary.

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 Map interface. 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 to familiarize yourself with the API.

Tree? Trie?

A prefix trie 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 na¨ ıve trie, 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 in which each node that is an only child is merged with its parent. Unfortunately, you will not be implementing PATRICIA tries for this assignment.

Part One: HashMap (35 points)

Files to submit: HashMap.java, HashMapTest.java

Recall the method of chaining discussed in lecture 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:

1. get(K key)
2. containsKey(K key)
3. put(K key, V value)
4. resize(int newCapacity)
5. remove(K key)
6. containsValue(V value)
7. clear()
8. keySet()

Each method stub contains further instructions. It is critical that you read both the Javadoc comments and the implementation comments for each method. They contain necessary information on both the external and internal behavior of these methods.

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.

The trickiest part of this implementation will be managing pointers when resizing. It might help to draw out some examples before you begin coding this part! Additionally, you should make sure you are correctly handling null keys and null values. 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.

**TrieMap (40 points)**

Files to submit: TrieMap.java, TrieMapTest.java

This should be a straightforward “standard” trie implementation, as described in the lecture notes. 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:

1. put(CharSequence key, V value)
2. get(CharSequence key)
3. containsKey(CharSequence key)
4. containsValue(V value)
5. remove(CharSequence key)
6. 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. 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.

**Extra Credit: Lazy iterator (10 points)**

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 points. 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, maximum 5 points.

**Part Three: Quantitative Analysis (15 points)**

Files to submit: questions.pdf

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 (not in the src folder) to avoid `FileNotFoundException`s. 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. First, make sure the provided `jamm.jar` file is in the root of your project (not in the src folder), then 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. Next, right-click on `TestHarness.java` and go to Run As →Run Configurations. Under the “arguments” tab, add the string `-javaagent:jamm.jar` to the VM arguments field, then Apply and Run.

Now, run `TestHarness.java`. It will report the CPU time and memory usage of your implementation!

**Note:** this will take a few minutes to run on your computer. We recommend that you take a brief walk outdoors in the meantime, or sword-fight a friend.

Please answer the following questions in a LaTeX’d document called questions.pdf:

1. Please copy and paste the output of `TestHarness.java` as an answer to this question, and place it inside of a `verbatim` environment.

2. For `dictionary.txt`, which implementation 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. How much memory, both absolutely and relatively, does this lazy initialization save us? (Give actual numbers.) Would you say this lazy initialization is a worthwhile optimization?

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 (10 points)

The above parts together are worth a total of 90 points. The remaining 10 points are awarded for code style, documentation and sensible tests. Style is worth 5 of the 10 points, and you will be graded according to the 121 style guide.

On this assignment in particular, focus on the behaviors that are specific to one implementation or the other. Especially consider edge cases, exceptions, and null keys and values. As always, use multiple methods instead of cramming a bunch of asserts into a single test method, and be sure to demonstrate that you’ve considered “bad” inputs, exceptions, etc. Your test cases will be manually graded by the TAs, and are worth 5 of the 10 points. You will have to thoroughly test your code to get full points! This includes testing any additional helper methods you have written. 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.

Mocking `hashCode()`

When writing unit tests for your `HashMap`, you may find it useful to force collisions between `Objects` 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:

```java
@Test
public void testCollision() {
    Object obj1 = new Object() {
        @Override
        int hashCode() { return 5; }
    };
    Object obj2 = new Object() {
        @Override
        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:

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

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

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

Either of these approaches works for testing your chaining.