Important

There are general homework guidelines you must always follow. If you fail to follow any of the following guidelines you risk receiving a **0** for the entire assignment.

1. All submitted code must compile under **JDK 8**. This includes unused code, so don't submit extra files that don't compile. Any compile errors will result in a 0.
2. Do not include any package declarations in your classes.
3. Do not change any existing class headers, constructors, or method signatures.
4. Do not add additional public methods.
5. Do not use anything that would trivialize the assignment. (e.g. don't import/use `java.util.LinkedList` for a Linked List assignment. Ask if you are unsure.)
6. Always be very conscious of efficiency. Even if your method is to be $O(n)$, traversing the structure multiple times is considered non-efficient unless that is absolutely required (and that case is extremely rare).
7. You must submit your source code, the `.java` files, not the compiled `.class` files.
8. After you submit your files redownload them and run them to make sure they are what you intended to submit. You are responsible if you submit the wrong files.

Hash Maps

In this homework, you will implement a key-value hash map with a linear probing collision policy. A hash map maps keys to values and allows $O(1)$ average case lookup of a value when the key is known. This hash map must be backed by an array of initial size 9, and must be resized to have a size of $2n + 1$ when the table exceeds (greater than, not greater than or equal to) a load factor of 0.67. These values are provided as constants in the interface, and the constants should be used within your code.

The table **should not add duplicate keys**, but duplicate values are acceptable. In the event of a duplicate key, replace the existing value with the new value.

Neiither keys nor values may be null.

Hash Functions

You should not write your own hash functions for this assignment; use the `hashCode()` method (every `Object` has one). If this is a negative value, use the absolute value of the hash code (and *then* mod by the table length).

Linear Probing

Your hash map must implement a linear probing collision policy. If the hash value of the key is occupied, probe in linear increments. For example, if the hash value of your key is 3 with a backing array of size 9, and index 3 in the array is occupied, check index $3 + 1 \mod 9$, then $3 + 2 \mod 9$, then $3 + 3 \mod 9$, etc.

Adding Items

When adding a key/value pair to a hash map, add the pair to the next available slot, starting at the computed index. Remember that available slots include both `null` and items marked as removed. Also remember that keys are unique in a hash map, so you must ensure that duplicate keys are not added.

For example, in an array of length 9, if a key is supposed to index 7 (based on its hash code), but index 7 is occupied by a different index, indices 8 and 0 have other removed keys, and index 1 is `null`, then add the pair into index 8; do `not` add it into index 1. Similarly, if a key/value pair needs to go into index 0, but indices 0-4 is occupied, indices 5-7 have removed entries, and index 8 is `null`, add the pair into index 5, **not** into index 8.

Removing Items

When removing items, only set the `removed` attribute to true; do **not** set the key and value to `null`. When resizing the backing array, do not copy over removed items.

A note on JUnits

We have provided a **very basic** set of tests for your code, in `HashMapStudentTests.java`. These tests do not guarantee the correctness of your code (by any measure), nor does it guarantee you any grade. You may additionally post your own set of tests for others to use on the Georgia Tech GitHub as a gist. Do **NOT** post your tests on the public GitHub. There will be a link to the Georgia Tech GitHub as well as a list of JUnits other students have posted on the class Piazza.

If you need help on running JUnits, there is a guide, available on T-Square under Resources, to help you run JUnits on the command line or in IntelliJ.

Style and Formatting

It is important that your code is not only functional but is also written clearly and with good style. We will be checking your code against a style checker that we are providing. It is located in T-Square, under Resources, along with instructions on how to use it. We will take off a point for every style error that occurs. If you feel like what you wrote is in accordance with good style but still sets off the style checker please email Jonathan Jemson (jonathanjemson@gatech.edu) with the subject header of "CheckStyle XML".

Javadocs

Javadoc any helper methods you create in a style similar to the existing Javadocs. If a method is overridden or implemented from a superclass or an interface, you may use `@Override` instead of writing Javadocs.

Exceptions

When throwing exceptions, you must include a message by passing in a String as a parameter. **The message must be useful and tell the user what went wrong**. "Error", "BAD THING HAPPENED", and "fail" are not good messages. The name of the exception itself is not a good message.

For example:

```
throw new PDFReadException("Did not read PDF, will lose points.");

throw new IllegalArgumentException("Cannot insert null data into data structure.");
```

Generics

If available, use the generic type of the class; do **not** use the raw type of the class. For example, use `new LinkedList<Integer>()` instead of `new LinkedList()`. Using the raw type of the class will result in a penalty.

Forbidden Statements

You may not use these in your code at any time in CS 1332.

- `break` may only be used in switch-case statements
- `continue`
- `package`
- `System.arraycopy()`
- `clone()`
- `assert()`
- `Arrays` class
- `Array` class
- `Collections` class
- `Collection.toArray()`
- Reflection APIs
- Inner, nested, or private classes

Debug print statements are fine, but nothing should be printed when we run them. We expect clean runs - printing to the console when we're grading will result in a penalty. If you use these, we will take off points.

Provided

The following file(s) have been provided to you. There are several, but you will only edit one of them.

1. `HashMapInterface.java`

   This is the interface you will implement in `HashMap`. All instructions for what the methods should do are in the javadocs. **Do not alter this file.**

2. `HashMap.java`

   This is the class in which you will implement `HashMapInterface`. Feel free to add private helper methods but **do not add any new public methods, inner/nested classes, instance variables, or static variables**.

3. `MapEntry.java`

   This class stores a key-value pair and a `removed` attribute for your hash map. **Do not alter this file.**

4. `HashMapStudentTests.java`

   This is the test class that contains a set of tests covering the basic operations on the `HashMap` class. It is not intended to be exhaustive and does not guarantee any type of grade. **Write your own tests to ensure you cover all edge cases.**

Deliverables

You must submit **all** of the following file(s). Please make sure the filename matches the filename(s) below, and that *only* the following file(s) are present. T-Square does **not** delete files from old uploads; you must do this manually. Failure to do so may result in a penalty.

After submitting, be sure you receive the confirmation email from T-Square, and then download your uploaded files to a new folder, copy over the interfaces, recompile, and run. It is your responsibility to re-test your submission and discover editing oddities, upload issues, etc.

1. `HashMap.java`