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 11**. 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, instance/global variables, or method signatures. For example, do not add **throws** to the method headers since they are not necessary.
4. Do not add additional public methods. You may create helper methods, but any helper method you create should be **private**.
5. Do not use anything that would trivialize the assignment. (e.g. don't import/use java.util.ArrayList for an ArrayList assignment. Ask if you are unsure.)
6. Always consider the efficiency of your code. Even if your method is *O(n)*, traversing the structure multiple times is considered inefficient unless that is absolutely required (and that case is very rare).
7. You must submit your source code - the .java files. Do not submit compiled code - the .class files.
8. Only the last submission will be graded. Make sure your last submission has all required files. Resubmitting voids prior submissions.

Collaboration Policy

Every student is expected to read, understand and abide by the Georgia Tech Academic Honor Code.

When working on homework assignments, you **may not** directly copy code from any source (other than your own past submissions). You are welcome to collaborate with peers and consult external resources, but you **must** personally write all of the code you submit. **You must list, at the top of each file in your submission, every student with whom you collaborated and every resource you consulted while completing the assignment**.

You may not directly share any files containing assignment code with other students or post your code publicly online. If you wish to store your code online in a personal private repository, you can use Github Enterprise to do this for free.

The only code you may share is JUnit test code on a pinned post on the official course Piazza. Use JUnits from other students at your own risk; **we do not endorse them**. See each assignment's PDF for more details. If you share JUnits, they **must** be shared on the site specified in the Piazza post, and not anywhere else (including a personal GitHub account).

Violators of the collaboration policy for this course will be turned into the Office of Student Integrity.

Style and Formatting

It is important that your code is not only functional, but written clearly and with good programming style. Your code will be checked against a style checker. The style checker is provided to you and it located on Canvas. A point is deducted for every style error that occurs. Please double check before you submit that your code is in the appropriate style so that you don't lose any unnecessary points!

Javadocs

Javadocs should be written for any private helper methods that you create. They should follow a style similar to the existing javadocs on the assignment. Any javadocs you write must be useful and describe the contract, parameters, and return value of the method. Random or useless javadocs added only to appease checkstyle will lose points.

Additionally, make sure you include your name, version, user ID, and GT ID in any file submitted to Gradescope.

Vulgar/Obscene Language

Any submission that contains profanity, vulgar, or obscene language will receive an automatic zero on the assignment. This policy applies to all aspects of your code, such as comments, variable names, and 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", "Oof - Bad things are happening", and "FAIL" are *not* good messages. Additionally, the name of an exception itself is not a good message.

In addition, you may not use try catch blocks to catch an excpetion unless you are catching an exception you have explicitly thrown yourself with the **throw new ExceptionName('Exception Message");** syntax

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 any of the following in your code at any time in CS 1332. If you are not sure whether you can use something, and it is not explicitly listed here, just ask. Debug print statements are fine, but should be either removed or commented out prior to submission. If print statements are left in, assignments are messy to grade, and checkstyle points will be deducted.

- package
- System.arraycopy()
- clone()
- assert()
- Arrays class
- Thread class
- Collections class
- Collection.toArray()
- Refleciton APIs

- Inner or nested classes
- Lambda Expressions
- Method References (using the :: operator to obtain a reference to a method)

JUnits

We have provided a **very basic** set of tests for your code. These tests do not guarantee the correctness of your code, nor do they 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.

Deliverables

You must submit all of the following file(s) to the corresponding assignment on Gradescope. Make sure all file(s) listed below are in each submission, as only the last submission is graded. Make sure the filename(s) matches the filename(s) below, and that *only* the following file(s) are present.

1. LinearProbingHashMap.java

HashMap

You are to code a `LinearProbingHashMap`, a key-value hash map with a linear probing collision resolution strategy. A `HashMap` maps unique keys to values and allows $O(1)$ average case lookup of a value when the key is known.

The table should **not** contain duplicate keys, but **can** contain duplicate values. In the event of trying to add a duplicate key, replace the value in the existing (key, value) pair with the new value and return the old value.

You should implement two constructors for this `HashMap`. As per the javadocs, you should use constructor chaining to implement the no-arg constructor.

Do not use magic numbers in your code. That is, use the provided `INITIAL_CAPACITY` in your code rather than hardcoding its values.

Hash and Compression Functions

You should **not** write your own hash functions for this assignment. Instead, use the `hashCode()` method that every `Object` has. For the compression function, mod by table length first, then take the absolute value (it must be done in this order to prevent overflow in certain cases). As a reminder, you should be using the `hashCode()` method on **only the keys** (and not the `LinearProbingMapEntry` object itself) since that is what is used to look up the values. After converting a key to an integer with a hash function, it must be compressed to fit in the array backing the `HashMap`.

Linear Probing

Your hash map must implement a linear probing collision policy. If the index corresponding to the hash value of the key is occupied, probe in linear increments. For example, if the hash value of your key is 7 with a backing array of capacity 9, and index 7 mod 9 in the array is occupied, check index $(7 + 1)$ mod 9, then $(7 + 2)$ mod 9, then $(7 + 3)$ mod 9, etc. until you hit a null spot in the array or after you have encountered `size` non-removed (key, value) pairs.

To maintain optimal efficiency, avoid doing more probes than necessary in any probing operation. For example, as soon as a null spot is found during a probe, you should not do any additional probes. If a removed map entry contains the key for which you are searching, you can avoid additional probes, since it is guaranteed that the key is not *currently* contained in the map. Once `size` non-removed entries have been seen, you should stop probing since this will always result in less than `table.length` iterations.

Adding Items

When adding a key/value pair to a `HashMap`, add the pair to the array in the correct position. Also remember that keys are unique in a hash map, so you must ensure that duplicate keys are not added. When searching for a spot to add, after ensuring no duplicates, you should add at the first encountered removed spot (if there are any). If no removed spots were encountered, add at the null spot that terminated your search.

Load Factor and Resizing

A `HashMap` will lose efficiency if it becomes too full. To combat this, if adding to the table would cause the load factor (LF) to **exceed** (greater than, not greater than or equal to) the `MAX_LOAD_FACTOR` constant provided in the `.java` file, the table should be resized to have a capacity of $2n+1$, where n is the current capacity before adding the parameterized element. See the javadocs for specific instructions on how to resize.

Removing Items

Since linear probing is an open addressing scheme, you should not set removed entries to null. Instead, you need to implement a "soft remove" using the removed flag in `LinearProbingMapEntry.java`. All the flag does is keeps track of what entries have been removed, but you need to implement the logic for what to do with the removed entries. Though the objects may still be in memory, as far as the user is concerned, the data has been removed. Do **not** null out the key and value of removed entries.

Grading

Here is the grading breakdown for the assignment. There are various deductions not listed that are incurred when breaking the rules listed in the PDF and in other various circumstances.

- LinearProbingHashMap − 90 pts
- Checkstyle − 10 pts
- Total: 100 pts

Provided

The following file(s) have been provided to you. There are several, but we've noted the ones to edit.

These are the classes to be implemented. Feel free to add private helper methods but **do not add any new public methods, inner/nested classes, instance variables, or static variables**.

1. `LinearProbingHashMap.java`

These are classes that represent entries in your hashing structure. **Do not alter these files.**

1. `LinearProbingMapEntry.java`

These are test classes that contain a set of tests covering the basic operations on their respective data structures. 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.**

1. `LinearProbingHashMapStudentTest.java`