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.
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 be very conscious of efficiency. Even if your method is to be $O(n)$, traversing the structure multiple times is considered inefficient 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. Only the last submission will be graded. Make sure your last submission has **all** required files. Resubmitting will void all previous submissions.
9. 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.

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 is

located on Canvas. It can be found under Files, along with instructions on how to use it. A point is deducted for every style error that occurs. If there is a discrepancy between what you wrote in accordance with good style and the style checker, then address your concerns with the Head TA.

Javadocs

Write javadocs for any helper methods you create in a style similar to the existing javadocs. 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.

Vulgar/Obscene Language

Any submission that contains profanity, vulgar, or obscene language will receive an automatic zero on the assignment. This policy applies not only to comments/javadocs, but also things like variable names.

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:
Bad: `throw new IndexOutOfBoundsException(``Index is out of bounds.'');`
Good: `throw new IllegalArgumentException(``Cannot insert null data into data structure.'');`
In addition, you may not use try catch blocks to catch an exception unless you catching an exception you have explicitly thrown yourself with the `throw new ExceptionName(``Exception Message'');` syntax (replacing `ExceptionName` and `Exception Message` with the actual exception name and message respectively).

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.

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

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

If you're not sure on whether you can use something, and it's not mentioned here or anywhere else in the homework files, just ask.

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

JUnits

We have provided a **very basic** set of tests for your code. These tests do not guarantee the correctness of your code (by any measure), 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. 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 Canvas under Files, to help you run JUnits on the command line or in IntelliJ.

HashMap

You are to code an `HashMap`, a key-value hash map with an external chaining 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.

Capacity

The starting capacity of the `HashMap` using the default constructor should be the constant `INITIAL_CAPACITY` defined in `HashMap.java`. Reference the constant as-is. Do **not** simply copy the value of the constant. Do **not** change the constant. Do **not** regrow the backing array when removing elements.

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 when to resize. There is a method called `resizeBackingTable` that you should use for resizing.

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 `MapEntry` object itself) since that's 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`.

External Chaining

Your hash map must implement an external chaining collision policy. That is, in the event of a collision, colliding entries are stored as a chain of `MapEntry` objects at that index. As such, if you need to search for a key, you'll need to traverse the entire chain at the hashed index to look for it. See `MapEntry.java` to see what is stored and what methods are available for use; do **not** use Java's `LinkedList` to handle the chaining functionality.

Adding

When adding a key/value pair to a hash map, add the pair to the front of the chain in the correct position. Also remember that keys are unique in a hash map, so you must ensure that duplicate keys are not added. Each index of the table should point to an `MapEntry` that represents the head of a linked list. That linked list contains all elements that collide at that index.

Removing

When removing a key/value pair from a hash map using external chaining, you can safely remove the item unlike in open addressing techniques such as linear probing where you must use a DEL marker. Removing from a chain is very similar to removing from a Singly-Linked List, treating the first table entry as the head, so refer to your notes and homework assignments from earlier in the course if you

need a refresher. As usual, if the entry you are removing is the only entry at that index, you should make sure to null out that spot rather than leaving it there.

Grading

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

Methods:	
constructor	2pts
put	18pts
remove	10pts
get	10pts
containsKey	10pts
keySet	5pts
values	5pts
resizeBackingTable	10pts
clear	5pts
Other:	
Checkstyle	10pts
Efficiency	15pts
Total:	100pts

Provided

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

1. `HashMap.java`

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

2. `MapEntry.java`

   This class stores a key-value pair and the next reference for your hash map. **Do not alter this file.**

3. `HashMapStudentTest.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) to Gradescope. Make sure all file(s) listed below are in each submission, as only the last submission will be graded. Make sure the filename(s) matches the filename(s) below, and that *only* the following file(s) are present. Do **NOT** submit `MapEntry.java`; if you do, your homework will not compile on Gradescope. If you resubmit, be sure only one copy of each file is present in the submission. If there are multiple files, do not zip up the files before submitting; submit them all as separate files.

Once submitted, double check that it has uploaded properly on Gradescope. To do this, download your uploaded file(s) to a new folder, copy over the support file(s), recompile, and run. It is your sole responsibility to re-test your submission and discover editing oddities, upload issues, etc.

1. `HashMap.java`