Change Log

3-8-2014 1:40.pm: Clarified the use of java.lang.Object's hashCode() method that is inherited. Change is under "Hash Table" heading in the first paragraph. The change is in red.

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 7**. This includes unused code, so don't submit extra files that don't compile. (Java is backwards compatabile so if it compiles under JDK 6 it *should* compile under JDK 7.)
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 when implementing an interface.
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). Traversing beyond your data and into unoccupied areas of the backing store is not-efficient and is not actually O(n). See more discussion of Big O and efficiency below.
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 Table

You will be coding a hash table for this homework. Hash tables are backed by an array of map entries, which you are given. You will be finding the proper index to store each map entry by using the built-in hashCode of the key. Use the hashCode() method inherited from java.lang.Object. (Since this is the ultimate ancestor of every class, every class inherits and has this hashCode() method.) If the hashCode() return value is negative, take the absolute value of that number. In any case, you will mod the possibly very large hashCode() value by the size of your table. Obviously hashCode() itself has no idea what range of indices you need.

Do not store null values in this hash table. If a user inputs null, throw an IllegalArgumentException. **IMPORTANT** Be sure to read the section on Big O, as your methods should fit within those requirements.

Collision Handling

This hash table will handle collisions with linear probing. This means that for entry removal you will need to set the removed flag on the entry to true instead of setting the location to null. On collision, find the next empty spot by stepping by 1 through the backing array. Make sure you understand the differences between occupied, removed, and null spots. (HINT: When searching for a key, you should stop probing ONLY once they hit a null spot).

Regrow

Regrow the table when adding the next element will cause the table to exceed maximum load factor. This means you will regrow the table BEFORE you add this element. Set the new size to twice the old size plus one. In other words, newSize = 2(oldSize) + 1. Remember, load factor is

$$loadFactor = \frac{occupied}{totalspots}$$

When copying over elements to the new backing array, add them in the order that they appear while traversing through the old table. Do NOT use Arrays.copyOf() when resizing.

Big O

Your methods must perform in the expected performance time. Note that these are worst case scenarios and often times your hash table should perform in O(1). For example, please please do NOT iterate through the entire array for search even though it has O(n) performance, that is not how a hash table works and points will be taken off.

Expected Big O

1. Add - O(n)
2. Remove - O(n)
3. Contains - O(n)
4. Get - O(n)

Disclaimer

We reserve the right to take off efficiency points for any method that is obviously wasteful even if the Big O is in the required range. Ex: A performance of O(3 * logn) when you can easily implement O(logn) is considered wasteful.

Style and Formatting

It is important that your code is functional and is also written clearly and with good style. Efficiency also always matters. Be sure you are using the updated style checker located in T-Square-Resources. (It was updated on February 5, 2014.) Be sure to run your code against this latest style checker. While it is backward compatible with the original style checker, some style rules have been made easier to pass, so it is in your interest to update to this Feb 5 style checker file.) If you feel like what you wrote is in accordance with good style but still sets off the style checker please email Kush Mansingh kmansingh3@gatech.edu with the subject header of "Style XML".

Javadocs

All public methods must have javadocs (or @Override's where appropriate). All private methods must be commented, but it is not necessarily that they be in javadoc style.

Provided

The following files have been provided to you:

1. `HashTableInterface.java`

2. `HashTable.java`
3. `MapEntry.java`
4. `CheckStyle.zip` - see T-Square Resources for the latest file from February 5, 2014

Deliverables

You must submit all of the following files. Please make sure the filename matches the filenames below.

1. `HashTable.java`

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. You may attach each file individually, or submit them in a zip archive.