Important

There are a few guidelines you must follow in this homework. If you fail to follow any of the following guidelines you will receive a **0** for the entire assignment.

1. All submitted code must compile under **JDK 7**. This includes unused code, 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. Don't include any package declarations in your classes.
3. Don't change any *existing* class headers, constructors, or method signatures. (It is fine to add extra methods and classes)
4. Don't import anything that would trivialize the assignment. (e.g. don't import `java.util.LinkedList` for a Linked List assignment. Ask if you are unsure.)
5. You must submit your source code, the `.java` files, not the compiled `.class` files.

After you submit your files redownload them and run them to make sure they are what you intended to submit. We are not responsible if you submit the wrong files.

Description

For this assignment you will be implementing a HashTable. A HashTable is a simple data structure that has the functionality of a dictionary. In a HashTable you will be storing key value pairs. With a given key, you can manipulate the corresponding data. For example, say you have a set of names, paired with corresponding phone numbers. This is a perfect case where you would want to use a HashTable.

The general idea is that there is an underlying array. Each spot in the array will hold data and its corresponding key. In the homework, you will have an array of Entry objects, each entry holds a key and a value. So given a key, how do I find the data that corresponds to it? To find data in the table, you hash the key (turn it into a number). This number can be used as an index for your array. With this specific index, you can now access the Entry object at that location in the array and retrieve the data out of it.

However, you might already see a problem. What if we turn our key into a number that is outside of the indexing range of our array? The simple solution is to use a compression function. The compression function just mods the generated index number by the length of the underlying array. This gives an index within the range of the array. Good, now we have a valid index in the array, however, there is yet another problem. Imagine if our hash function is to add up the ascci values of characters in a string.

$$AB = A + B = 65 + 66 = 131 \qquad BA = B + A = 66 + 65 = 131$$

Derp...two different keys with the same hash. This means that they will land on the same index after the compression as well. two keys landing on the same index is called a collision. Now that we are moding our hashed keys, it is possible that two different numbers will wrap around to the same index as well.

There are many different schemes for handling collisions, but the one that we will be focusing on in this homework is linear probing. This means that, if we hash and mod to an index in our array that already has an Entry in it, we will look at the next index until we find a spot that we can place it (Ex. I take in the key "cat" and generate an index of 3. I look in my array at index 3...fiddle sticks...there is already an Entry object at index 3. No problem, maybe 4 is open, and if it is not, then I can look at 5 and so on). Things seem simple at this point.

At this point we haven't really discussed what happens when I take something out of the table. It would seem obvious that when you add you want to check and see if your index is null. If is null then you are free to insert your new Entry object. If not, we want to probe upwards. The next problem occurs when we are looking for something in the hashtable. Say we are given a key and want the corresponding data out of the hashTable. We hash and mod the key to get the index. With the index, check the array at that spot and see if there is something there. If there is an entry object at the index, we need to make sure it is the one we are looking for (The one we are actually looking for could have been probed upwards due to a collision). So how do we know how far to probe upwards in search of our data? We will look until we hit a null. Here we know that it couldn't have possibly probed past a null slot in the array.

Here arises another question about how to remove from the hashTable. If we simply set an index to null when we want to remove the entry at that location, it might make someone looking for a piece of data stop prematurely. So how can we tell someone in search that this removed spot does not have anything in it, but used to, and possibly could have caused adding data to probe over it. The simple solution is to place an available flag in the Entry object. Now we can distinguish. If a slot is null then nothing has ever been there. If an index has an Entry object with the available flag set to false, it has valid data that should not be bothered. If the index has an Entry object with the available flag set to true, we know that it once had living data, but that data is no longer important. This means that all we have to do on remove is find the Entry object with the correct key and set the available flag to true. On a search for data, we can still obey the law of only stopping at a null. You now know all of the slots occupied by an object that could have potentially sent the data you are in search of down stream. Lastly, we need to realize that we can fill in these Entry objects that have the available flag set to true. The data there is no longer valid, and if we come across an available Entry when attempting to add, we can simply stick in our new key value pair and set the available flag to false.

In light of this, we may want to regrow our underlying array, in the same way we regrow in an array list. Unlike an array list, we do not want to wait until the table is full but instead resize once a certain proportion of the indices have Entrys in them. We call this proportion the load factor. Once the load factor is exceeded, we will regrow. Note that for this homework you will want to check for regrow immediately after adding an item (Ex. an underlying array of size 5 and max load factor of 1/2 should never, at any point, have 3 indices full).

When you first create your array, do not instantiate all of the indices with Entry objects. Your array should start out with null in all of the indices, gradually creating Entry objects and inserting them as the user populates the structure.

Methods

put

This method inserts a value into the structure based on a key

1. hash the key
2. compress the hash
3. check if the index is null (stick a new node in it)
4. check to see if the Entry object there is available (stick in the key value pair)
5. if the spot is full, probe up until you find an available entry or a null index (if you reach the end loop around to the front)

remove

Thie method removes a value from the structure

1. hash the key
2. compress the hash
3. check if the index is null (return null)
4. if the entry object is not available and has the key you are looking for set the available flag to true and return the value
5. linear probe until you see null

The remaining methods are similar to the two described. You do not need to handle null keys in this assignment. You may import HashSet for completing the keySet and entrySet methods, but don't use it for values (use an array list or linked list)

Deliverables

You must submit all of the following files.

1. `HashTable.java`

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