Homework 6 - Hash Table
Due June 24, 2013 11:55 PM

For this assignment, you will be writing a Linear Probing Hash Table. A Hash Table is a very efficient lookup table. When you want to find the definition of a **word** in a dictionary, you need to go to the section of the dictionary that starts with the first letter of your desired word and **keep flipping pages** until you find the **definition** of the word you are looking for.
In the example above the word that you are looking for is analogous to **Key** in a Hash Table, the definition that you find is analogous to **Value** in a Hash Table, and the act of flipping pages after you find the desired section until you actually find the desired word is called **Probing**.

Hash Tables store **(Key, Value) pairs**, where the **Key** is always *unique*. These pairs are **stored in an array**. As you can see in the LinearProbingHashtable.java file, there is a private array called "hashTable". This array is the data structure that holds the (Key, Value) pairs.

put(key, value):
The put function is similar to the add() function in some of the data structures we've already learned. Basically, put will add a (**Key,Value)** pair to your Hash Table.
Below is the "hashTable" array mentioned above. Initially we don't have anything in it, and lets say we made it with initialSize=5.

0	
1	
2	
3	
4	

Now let's say we call **put(2, "Hello there children!")**. Your **hash function** will look at the **Key** (in this case the Integer 2) and generate a unique number, called a **hash code**. For this homework, you will use Java's pre built hash function(.hashCode()). Once you have the hash code, you will take the **modulus** of the hash code with the length of the array, and attempt to store the number in the resulting array index. This is known as **compressing** the hash code. So, for this example, we will use a simple hash function which returns the number itself as the hash code. For example, the hashcode for the Integer 123, will be 123. For the number 2, the hash function will return 2. We then mod this by the length of the array. So, 2%5 gives us the index where we will add the entry **(2, "Hello there children!")** . We look at position 2 in our array and if nothing is there we insert the pair at index 2. If index 2 was not empty, we run into a problem

known as **collision**. Collision simply means that the position at which you wanted to insert into is already taken up by another entry.

0	
1	
2	(2, “Hello there children!”)
3	
4	

For practice, lets add (9, “OMG they killed Kenny!!!, you something something!!!”) to our Hash Table. Here, the index where we insert is = 9%5, so we will add this entry at index 4.

0	
1	
2	(2, “Hello there children!”)
3	
4	(9, “OMG They killed Kenny!!!, you something something!!!”)

Now let’s say we want to add (7, “Alcohol is bad, mkay?!”). Here the index = 7%5. So, the pair (7, “Alcohol is bad, mkay?!”) should go into position 2 in the array, but that spot is already full. In this case you start **Probing**. We will probe *Linearly*. **Linear Probing** is a strategy to resolve **collisions** like this. Linear Probing is simply the process of finding the next available position to add the current entry since the location it maps to is taken up by another entry. Also, I encourage you to look up **Double Hashing** and **Quadratic Probing** as the alternatives of Linear Probing.

Since 7%5 is index 2, which is already taken up by an entry, we will go through the hash table in a linear manner and find the next open position. we see that index 3 is open, we will insert our entry there.

0	
1	
2	(2, “Hello there children!”)
3	(7, “Alcohol is bad, mkay?!”)
4	(9, “OMG they killed Kenny!!!, you something something!!!”)

Now let's add (8, "How do I reach these kidzzzzz?"). This will go at position 8%5, which is 3, but index 3 is already taken! We have to increment to index 4, but the 4 is already taken too, so we increment the index again to get 5, but since 5 is out of range of acceptable indices we mod it again with the array's length to get an index in the acceptable range.
Therefore our new index is 0. Because index zero is empty we can put the pair in it.

WAIT a second! The insertion of (8, "How do I reach these kidzzzzz?") almost took us O(n). Isn't the time complexity of adding to a Hash Table O(1)? The reason for this was that the array is almost full. In the case that the array is almost full we need to **"Rehash" *all* the elements**! Rehashing is the act of increasing the size of the array (for simplicity we will double the length of the array) and calling **put** on all the existing pairs. Since the array is larger, when we mod the hash code, we are more likely to not 'collide' into another entry. Also we should have some kind of threshold to realize that an array is almost full. We call this threshold **"Load Factor"**. For this homework, the default load factor should be 0.75 unless the user uses the constructor LinearProbingHashtable(10, 0.50) which in this case the load factor is explicitly specified by the user to be 0.50.

You must always make sure that the **Current Load Factor is *strictly less* than the Load Factor** passed into the Hash Table OR the default load factor of the the Hash Table.
Just to be clear: Current load factor is defined by Size/lengthOfArray. Here, size is the number of entries in the array. Do not confuse this with length of the array, these are two separate things. Throughout this guide I will use **loadFactor** to talk about the *threshold* of the Hash Table, and I will use size/array.length to show the *current load factor*.
Upon adding something to your Hashtable, first you have to check if the given pair is a valid pair to add (valid inputs are explained in the put() method's javadoc) then see if adding an element will violate the (size/lengthOfArray)<LoadFactor inequality, if the inequality doesn't hold you can **first rehash** and **then add** the new pair or **first add** and then **rehash**. It's your choice to do which one first! they effectively have the same time complexity. In this guide I will show you "first rehash" and then "add". Maybe it's even easier to do "add first" then "rehash" due to less edge cases that you have to take care about.
Note: if you do rehash first and then add, you should also call contains to make sure that the key is not already in the table. If the key is already in the table you won't rehash, you will just swap the new value with the old value and also you won't increase the size because you have just swapped the values.

In the case above size = 3 and there is no element in the Hash Table with key=8, so now the projected size after adding the pair will be 4. Since the array's length is 5, the inequality ⅘<0.75 doesn't hold and we have to rehash, and then add the new pair.

1) First rehash. Notice that after rehashing the pair with key=7 hashed to index 7 and the pair with key=9 hashed to index 9.

0	

1	
2	(2, “Hello there children!”)
3	
4	
5	
6	
7	(7, “Alcohol is bad, mkay?!”)
8	
9	(9, “OMG they killed Kenny!!!, you something something!!!”)

2) Then add the new pair (8, “How do I reach these kidzzzzz?”)

0	
1	
2	(2, “Hello there children!”)
3	
4	
5	
6	
7	(7, “Alcohol is bad, mkay?!”)
8	(8, “How do I reach these kidzzzzz?”)
9	(9, “OMG they killed Kenny!!!, you something something!!!”)

Let’s add one more element, I promise it will be the last pair we add! Let’s add (17, “I love Southpark, just in case you haven’t realized yet!”).

0	(17, “I love Southpark, just in case you haven’t realized yet!”)
1	
2	(2, “Hello there children!”)
3	

4	
5	
6	
7	(7, “Alcohol is bad, mkay?!”)
8	(8, “How do I reach these kidzzzzz?”)
9	(9, “OMG they killed Kenny!!!, you something something!!!”)

Note: one last *important* note. If a pair, let’s say (k1, v1), already exists in the hashtable and the user calls put(k1, v2), you need to put the value v2 instead of v1. In simple terms, update the value if the key is already present in the table.

Remove:

Let’s remove the pair with key=8. You need to again find the index of the element by doing index=key%array.length which in this case gives you hashCode=8. Then you look at index 8 to see if the pair at that location has key=8, if it has, remove that pair.

0	(17, “I love Southpark, just in case you haven’t realized yet!”)
1	
2	(2, “Hello there children!”)
3	
4	
5	
6	
7	(7, “Alcohol is bad, mkay?!”)
8	
9	(9, “OMG they killed Kenny!!!, you something something!!!”)

Now let’s say we want to remove the pair with key 12. As always we find the hashCode like the following index=12%10, which results in index=2.

There is an element at index 2 but its key is not equal to 12. Therefore we use linear probing to keep looking for an element with key=12. As a result index gets incremented, index = 3, but since that index doesn’t hold any pair **(because it’s null)** you will declare that remove was not successful.

Now let's try to remove the pair with key 17.
index = 17%10, therefore index = 7. There is an element at index 7 but its key is not 17, in this case we increment the index. Now index =8. When we look at index 8 we see that we don't have any pair at that position. If we were to follow the same logic as the previous time when we tried to remove key=12 we should declare that we didn't find the pair with key=17 in the Hash Table. But in reality we have an element with the key 17 in our Hash Table. The reason that 17 is at index zero is because after hashing, we found out that index 7 was full, AND at the time of insertion index 8 and 9 were full so we had to insert the pair with key=17 at the next available index which was index 0.

The problem occurred when we removed the pair with key=8. If instead of replacing that element with null we only marked it as **Available** for insertion, we would have effectively deleted it and at the same time we would have prevented the difficulties we faced when we wanted to remove the pair with key=17.

Now let's go back to the step that we removed key=8 and instead of deleting it, let's mark its availability as **True**, and mark the availability of the elements that have not been removed as **False**.

0	False -- (17, "I love Southpark, just in case you haven't realized yet!")
1	
2	False -- (2, "Hello there children!")
3	
4	
5	
6	
7	False -- (7, "Alcohol is bad, mkay?!")
8	True -- (8, "How do I reach these kidzzzzz?")
9	False -- (9, "OMG they killed Kenny!!!, you something something!!!")

Now let's attempt to delete the pair with the key=17 again.
index=17%10. So we need to look at index 7. Index 7 is full but the key is not equal to 17. Move on to the next position. NOW index 8 doesn't hold null but it holds a "removed pair". Since the availability is marked as True we won't even check its key against the goal key, we simply increment the index. Now index=9. The Availability of location 9 is False, meaning that there is an element there which has not been deleted, therefore we check its key against the goal, but since the keys don't match we increment the index and since index=10 and and therefore out of bounds we mod the index with the length of the array, which results in index=0. Now we look at

index 0 in the array and we find the element with the key=17. In this step we set the availability of that spot to True indicating that this element has been removed!

0	True -- (17, “I love Southpark, just in case you haven’t noticed yet!”)
1	
2	False -- (2, “Hello there children!”)
3	
4	
5	
6	
7	False -- (7, “Alcohol is bad, mkay?!”)
8	True -- (8, “How do I reach these kidzzzzz?”)
9	False -- (9, “OMG they killed Kenny!!!, you something something!!!”)

Note: When you rehash DO NOT copy the deleted pairs over (a.k.a the pairs with True availability) or Mint Berry Crunch will be after you!!!

Things to watch out for :

1. If you rehash entries which are “available” , you’re gonna have a bad time.
2. If you do not set your hashTable variable to the new array, when you resize, you’re gonna have a bad time.
3. If you perform O(n) operations, i.e iterate through the whole array and not use the hash function, you are gonna have a bad time.
4. If you import anything trivializes the assignment, you’re gonna have a VERY bad time.

So when do you have a good time?

1. When you code this homework and get a 100 :-D

Method explanations:

Please read the javadocs for an explanation about each method.

Instance Variables:

Feel free to use more variables than suggested below, but you should only need these suggested number of variables.

You only should need 3 instance variables for LinearProbingHashtable:

1) array of entries
2) load factor (double)
3) size (int)

You only should need 3 instance variables for TableEntry:

1) key
2) value
3)available

Deliverables:

1) LinearProbingHashtable.java
2) TableEntry.java
3) Any other files necessary to run your program (if any)