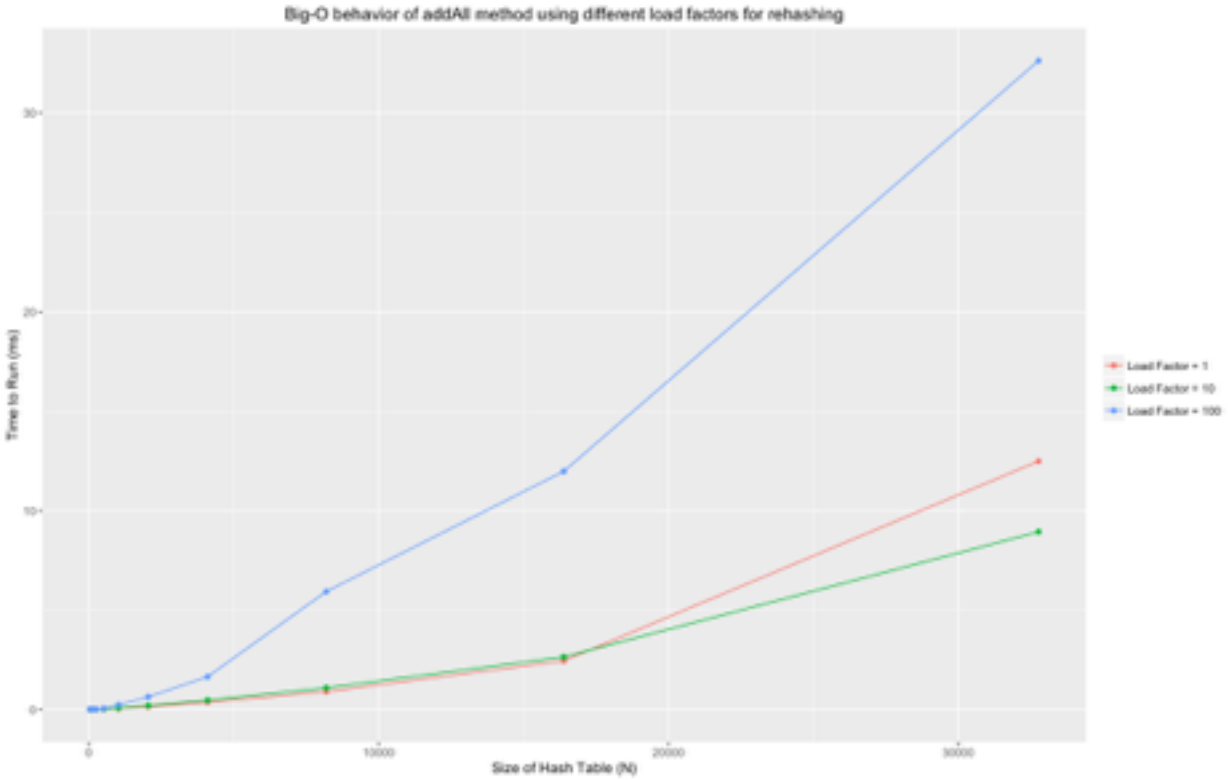
Kira Parker u1073760

1. What does the load factor mean for each of the two collision-resolving strategies (quadratic probing and separate chaining) and for what value of Î» does each strategy have good performance?

For quadratic probing, the load factor is the fraction of the hash table that is full. Quadratic probing has good performance is the load factor is less than 1/2. For separate chaining, the load factor is the average length of the linked lists. To determine the load values for which separate chaining has good performance, I ran an experiment where I timed how long it took to add N random strings of length 1-100 (N between 2^5 and 2^15, incrementing by a factor of 2) to a ChainingHashTable of initial size 11 for three different thresholds for rehashing: load factor = 1, 10, and 100. As can be seen by the graph below, the ChainingHashTable had best performance when the threshold for rehashing was when the load factor was 10 (green line).



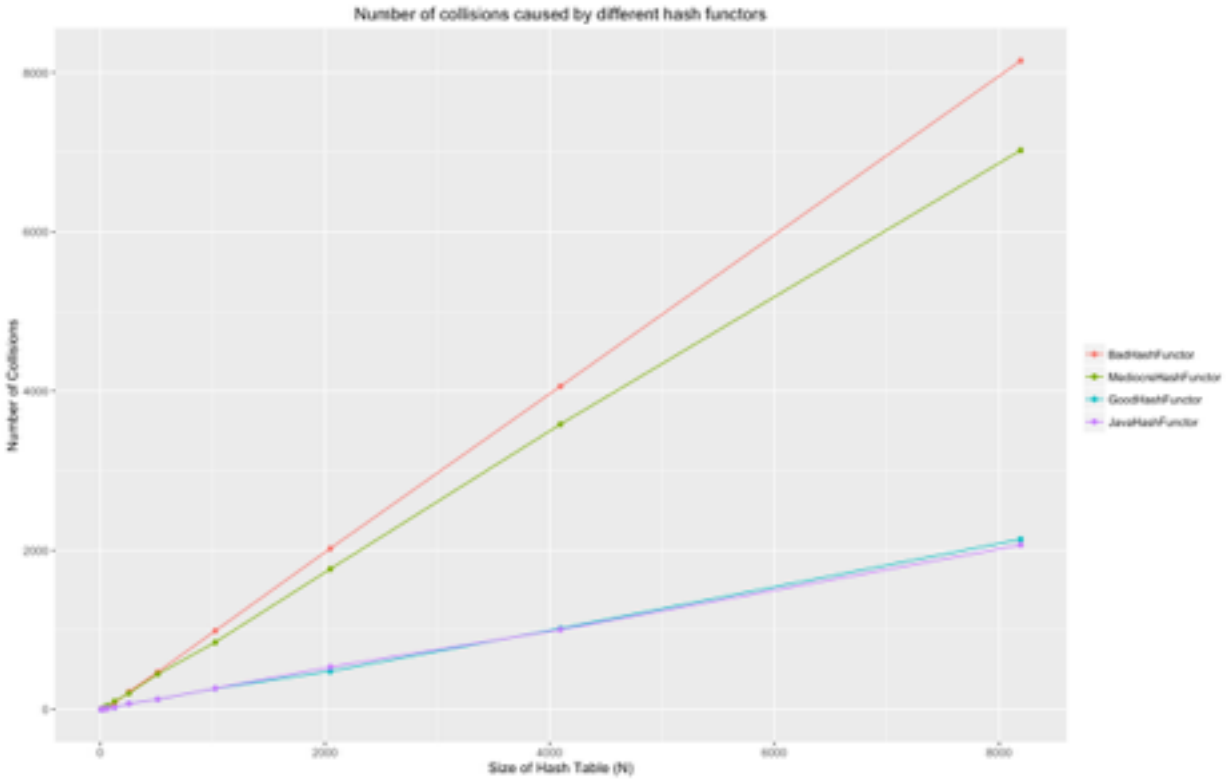
2. Give and explain the hashing function you used for BadHashFunctor. Be sure to discuss why you expected it to perform badly (i.e., result in many collisions).

For the BadHashFunctor, I simply returned the length of the string. This would be an ok HashFunctor if the strings you were going to be using had drastically different lengths, but for how I was testing by added random strings of length 1-100, this resulted in a lot of collisions and did not evenly distribute the strings across the storage array in the hash table. No string could get a hash value over 100, so for large N almost the entire array was either empty (in the case of the ChainingHashTable) or not being directly mapped into (in the case of the QuadProbeHashTable).

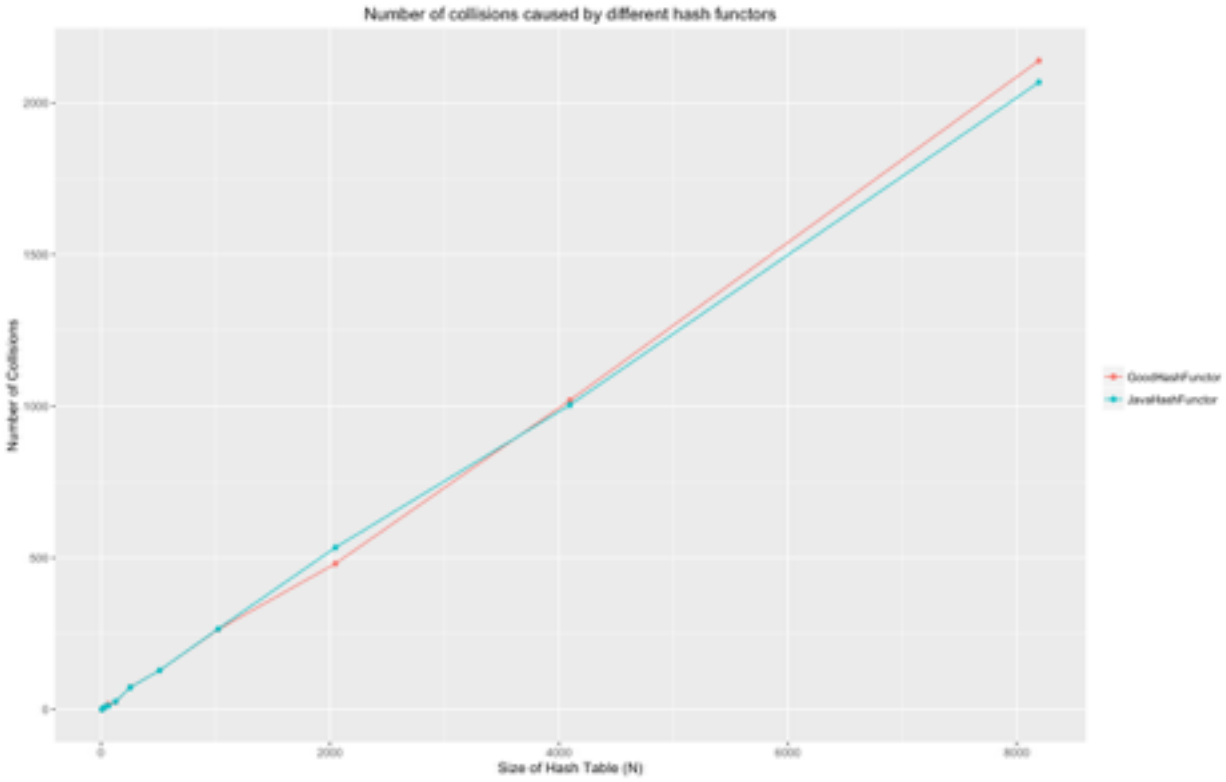
3. Give and explain the hashing function you used for MediocreHashFunctor. Be sure to discuss why you expected it to perform moderately (i.e., result in some collisions).
For the MediocreHashFunctor, I multiplied all the integer values of the characters in the string. I originally thought that this would be a GoodHashFunctor, but upon reflection I realized that by multiplying all the numbers together, the result is too likely to be divisible by the size of the storage array, and several collisions will result. There were also a few issues with overflow.

4. Give and explain the hashing function you used for GoodHashFunctor. Be sure to discuss why you expected it to perform well (i.e., result in few or no collisions).
For GoodHashFunctor I added the integer values of all the characters in the string and returned the absolute value (in case of wrap-around due to exceeding the integer limit). I thought that this would result in few collisions because having two identical sums is unlikely, and they issue of divisibility that occurred in the MediocreHashFunctor wouldn't occur when adding up the values. If the input strings are all extremely short and the storage array is very large, however, this method will not work well because it will not produce a wide enough range of hash values to fill the array.

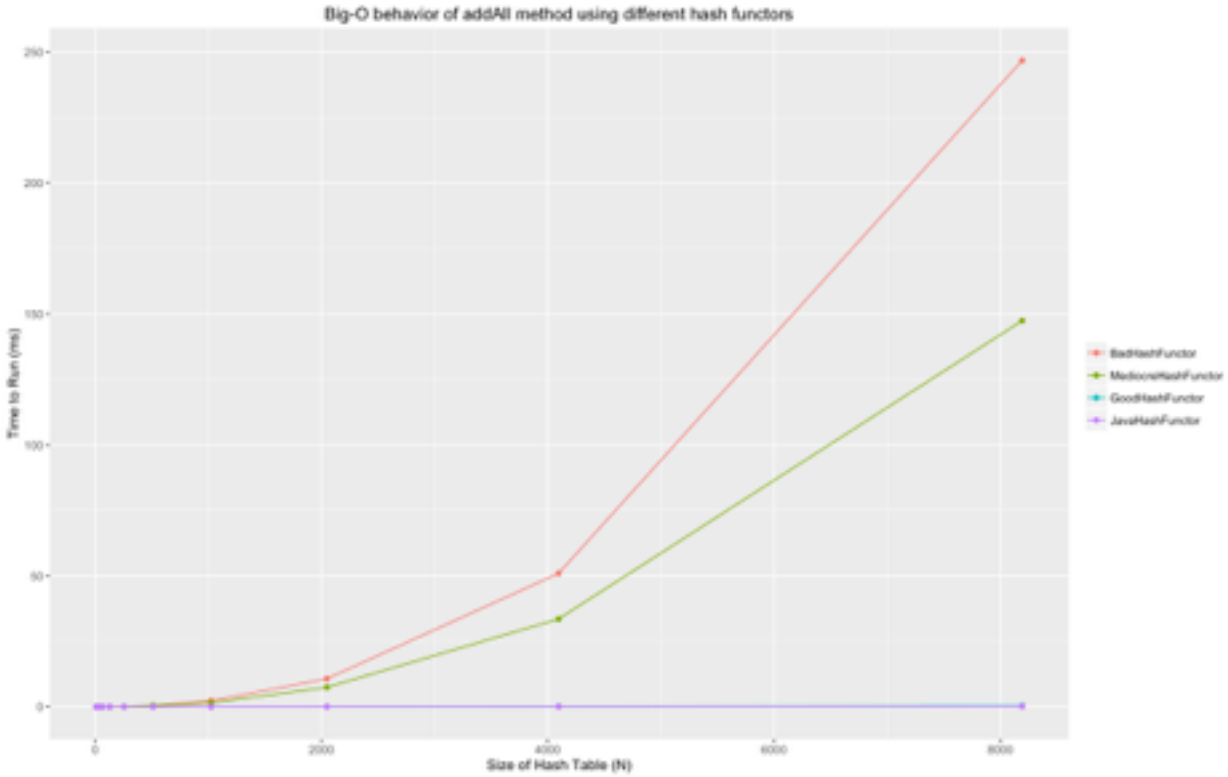
5. Design and conduct an experiment to assess the quality and efficiency of each of your three hash functions. Carefully describe your experiment, so that anyone reading this document could replicate your results. Plot the results of your experiment. Since the organization of your plot(s) is not specified here, the labels and titles of your plot(s), as well as, your interpretation of the plots is critical. A recommendation for this experiment is to create two plots: one that shows the number of collisions incurred by each hash function for a variety of hash table sizes, and one that shows the actual running time required by each hash function for a variety of hash table sizes. You may use either type of table for this experiment.
For the experiment, I calculated the number of collisions caused by different hash functions. A collision was defined as a strings mapping to an array index with an element already in it in the QuadProbeHashTable (when the array is resized, the collisions are reset to zero before the elements are added to the new array). Each data point is the number of collisions that occurred in the QuadProbeHashTable (of initial size N) for adding N random strings of random size

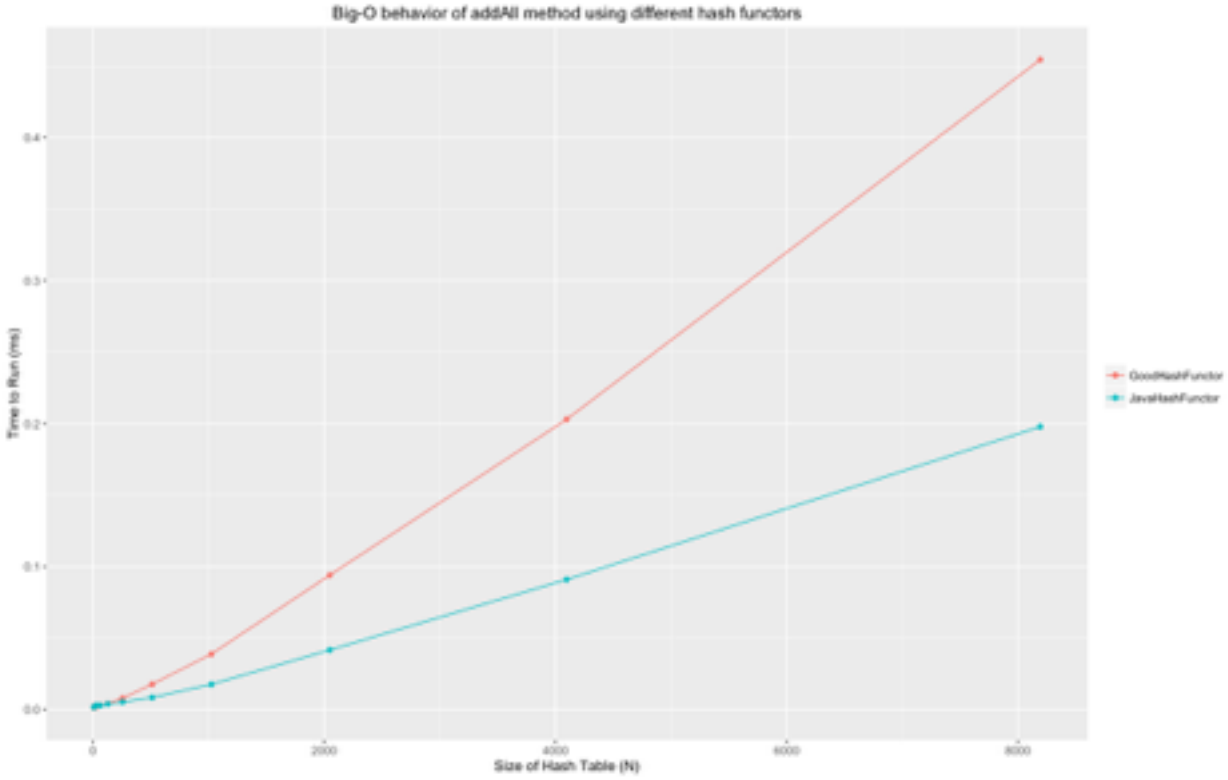


between 1 and 100 (N is between 2^3 and 2^13) using various hash functors. The same random strings were added to the QuadProbeHashTable for testing each different HashFunctor. As can be seen, the BadHashFunctor (red line) causes the most collisions, and the MediocreHashFunctor (green line) causes almost as many collisions. the GoodHashFunctor significantly fewer collisions, and is quite similar to Java's HashFunctor, which is included as a comparison. This similarity is shown best in the second graph comparing the number of collisions caused by GoodHashFunctor and JavaHashFunctor.



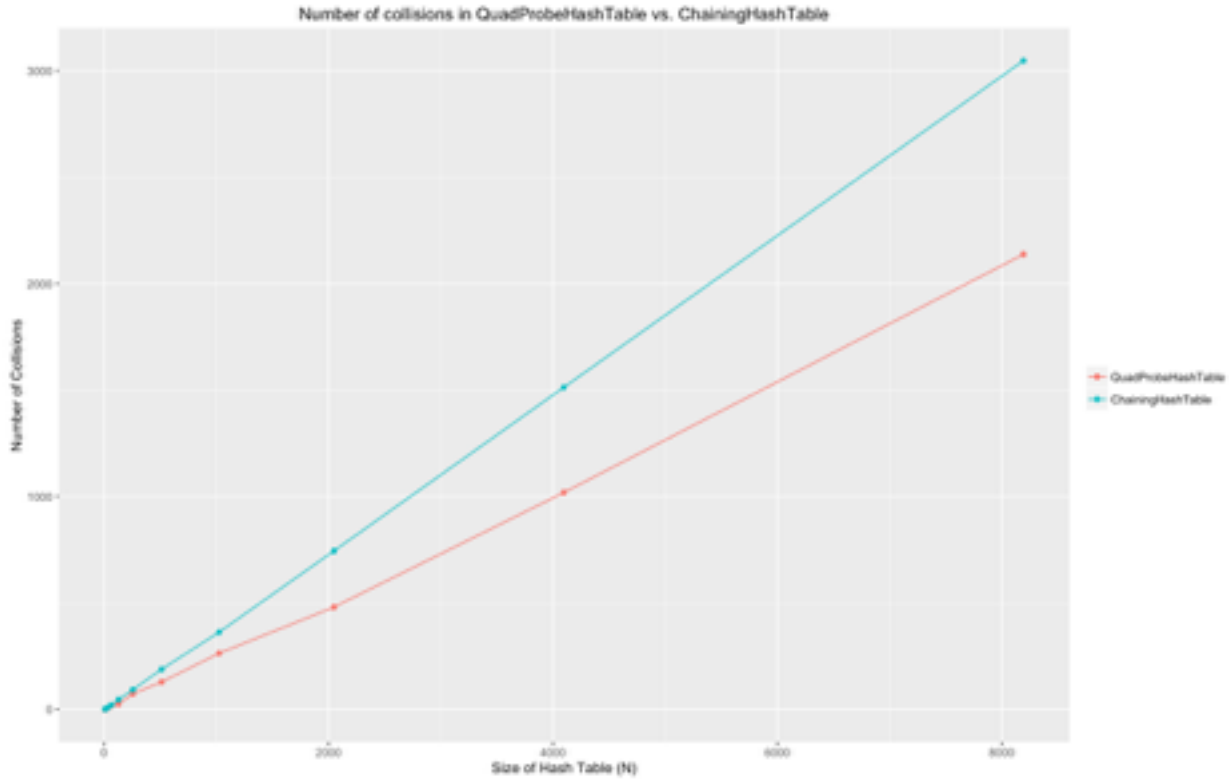
For the second experiment, the amount of time required to add N items to a QuadProbeHashTable using different HashFunctors was calculated. N was between 2^3 and 2^13, increasing by a factor of 2 each time. The same random strings used to calculate the number of collisions were added to the QuadProbeHashTable (of initial size N). As expected, using the BadHashFunctor (red line) took the longest time to add N items to the QuadProbeHashTable, closely followed by the MediocreHashFunction (green line). Using the GoodHashFunctor gave addAll a linear run time, similar to the JavaHashFunctor. The bottom graph excludes the BadHashFunctor and MediocreHashFunctor data, showing the O(N) complexity of addAll (which makes sense because all N items have to be looked at) for QuadProbeHashTables using GoodHashFunctor and JavaHashFunctor

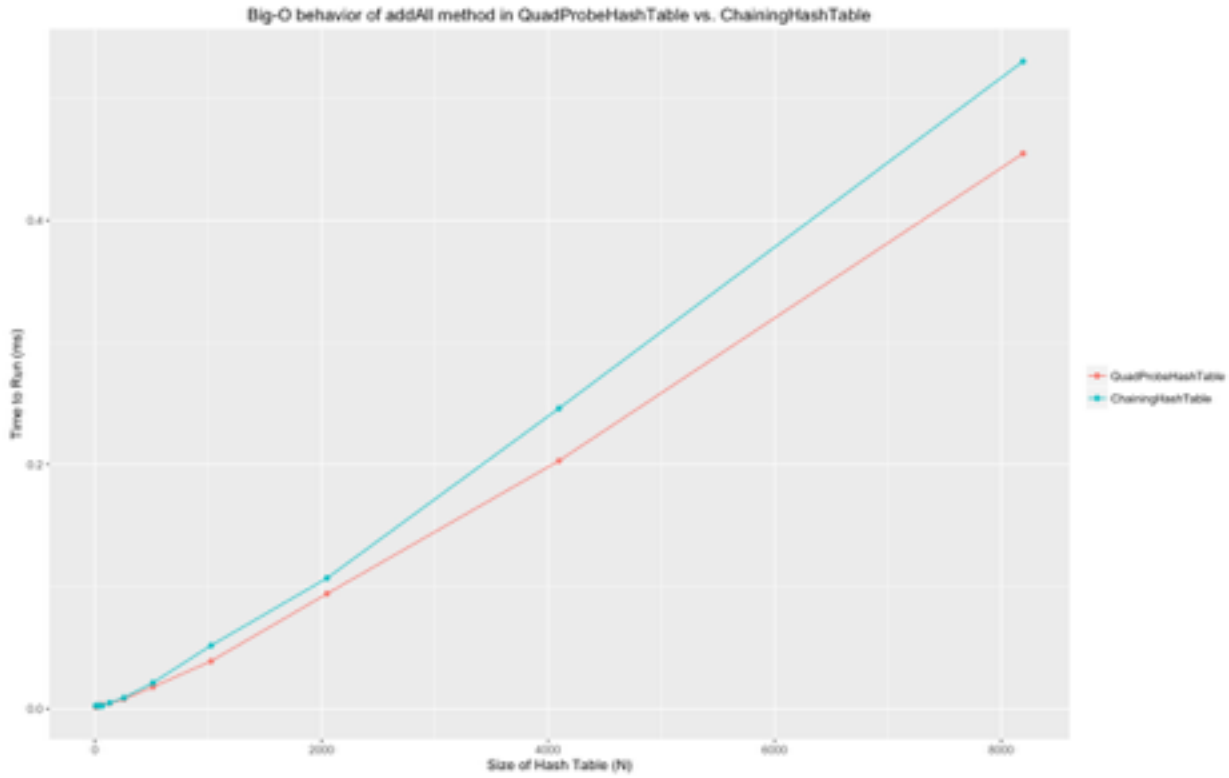




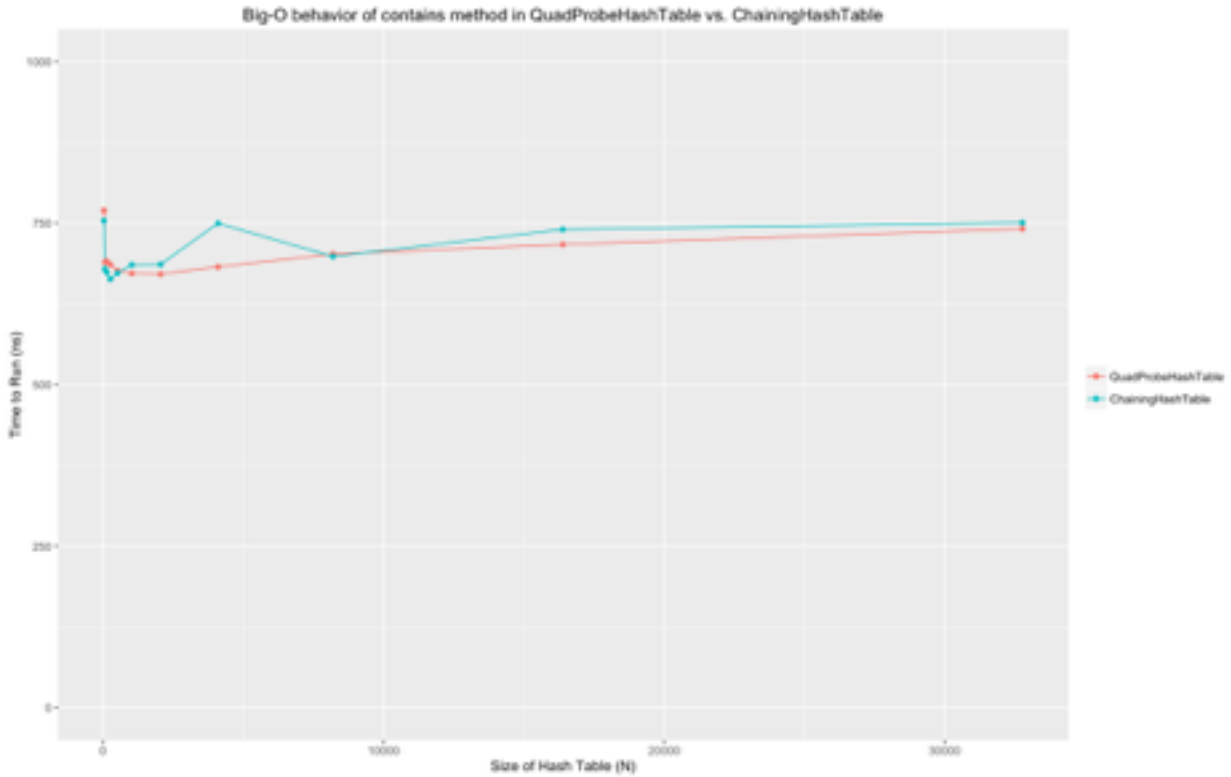
6. Design and conduct an experiment to assess the quality and efficiency of each of your two hash tables. Carefully describe your experiment, so that anyone reading this document could replicate your results. Plot the results of your experiment. Since the organization of your plot(s) is not specified here, the labels and titles of your plot(s), as well as, your interpretation of the plots is critical. A recommendation for this experiment is to create two plots: one that shows the number of collisions incurred by each hash

table using the hash function in GoodHashFunctor, and one that shows the actual running time required by each hash table using the hash function in GoodHashFunctor. First, the number of collisions incurred by each hash table (the number of times the location a string should go already has an element in it) was calculated for hash tables of size N = 2^5 though 2^15 (incrementing by a factor of 2 each time), where the elements were randomly generated strings of length 1-100 and the initial table size was N. As can be seen in the graph below, there were more collisions in the ChainingHashTable (blue) than in the QuadProbeHashTable (red).





Second, the time it took to add an ArrayList of N randomly generated strings of length 1-100 (same strings as used for all other tests) to the QuadProbeHashTable and the ChainingHashTable of initial size N was calculated for N between 2^5 and 2^15 (incrementing by a factor of 2). Surprisingly, the QuadProbeHashTable (red line) had a better run time for this as well, although the complexity for each was O(N) (meaning add() is constant time).
Finally, the efficiency of the contains method was tested by adding the same N randomly generated strings of length 1-100 as in all the other tests to a QuadProbeHashTable or ChainingHashTable of initial size N and then testing how long it took to determine if a randomly generated string of length 1-100 was in the table. Both HashTables had an O(c) contains method, which is to be expected, and the two methods were almost equally efficient.
All in all, the QuadProbeHashTable appears to have better performance than the ChainingHashTable.



7.What is the cost of each of your three hash functions (in Big-O notation)? Note that the problem size (N) for your hash functions is the length of the String, and has nothing to do with the hash table itself. Did each of your hash functions perform as you expected (i.e., do they result in the expected number of collisions)? (Be sure to explain how you made these determinations.)
The BadHashFunctor has complexity O(c) because it just returns the length of the string. It performed about as expected and its use resulted in many collisions.
The MediocreHashFunctor has complexity O(N) because it multiplies the integer values of all the characters in the string. Originally I was unsure about using an O(N) method because I thought it would be significantly slower, but Java's hashcode method for strings is O(N), and for adding smallish strings it will not affect runtime. I was surprised at the poor performance of this

method. I had originally intended for it to be my GoodHashFunctor, but that did not turn out to be the case. It resulted in far more collisions than I anticipated.
The GoodHashFunctor has complexity O(N) as well because it sums all the integer values of the characters in the string. It performed how I expected the MediocreHashFunctor to perform: quite well. It resulted in about as many collisions as Java's String hashcode method, which I took to mean that it was quite adequate.

8. How does the load factor Î» affect the performance of your hash tables?
As the load factor increases, the performance of the hash table worsens. For the ChainingHashTable, more elements have to be looked though in the LinkedList before it can be determined if an element is contained. In the QuadProbeHashTable, the likelihood of an index being open for an element decreases, and more and more often you will have to loop around the array to find an empty spot or determine if the element is contained in the hash table. However, rehashing is also expensive (O(N)) so it should only be done when necessary, which for the QuadProbeHashTable is when the load factor is greater than .5, and for the ChainingHashTable is when the load factor is greater than 10.

9. Describe how you would implement a remove method for your hash tables.
For the ChainingHashTable, it would be extremely easy. You would find the index in the array the item should be at and then call remove on the linked list at that index of the array. For the QuadProbeHashTable, remove would be slightly more challenging because the elements act as placeholders for the quadratic probing. You would not actually remove an element, but mark it as removed. Then when adding elements, you would add an element to an empty spot or a spot with a "removed" element. When determining if an element is contained in the hash table, you would have to find it and check if it has been removed or not.

10. As specified, your hash table must hold String items. Is it possible to make your implementation generic (i.e., to work for items of AnyType)? If so, what changes would you make?
It would be easy to make the implementation generic. All that would have to be done is change QuadProbeHashTable's storage to AnyType[] and ChainingHashTable's storage to LinkedList<AnyType>[]. The add, contains, and remove methods would take an AnyType item instead of a String item. The foreach loops would have to be changed from for(String item: items) to for(AnyType item: items).

11. How many hours did you spend on this assignment?
I spend about 7 hours on this assignment.