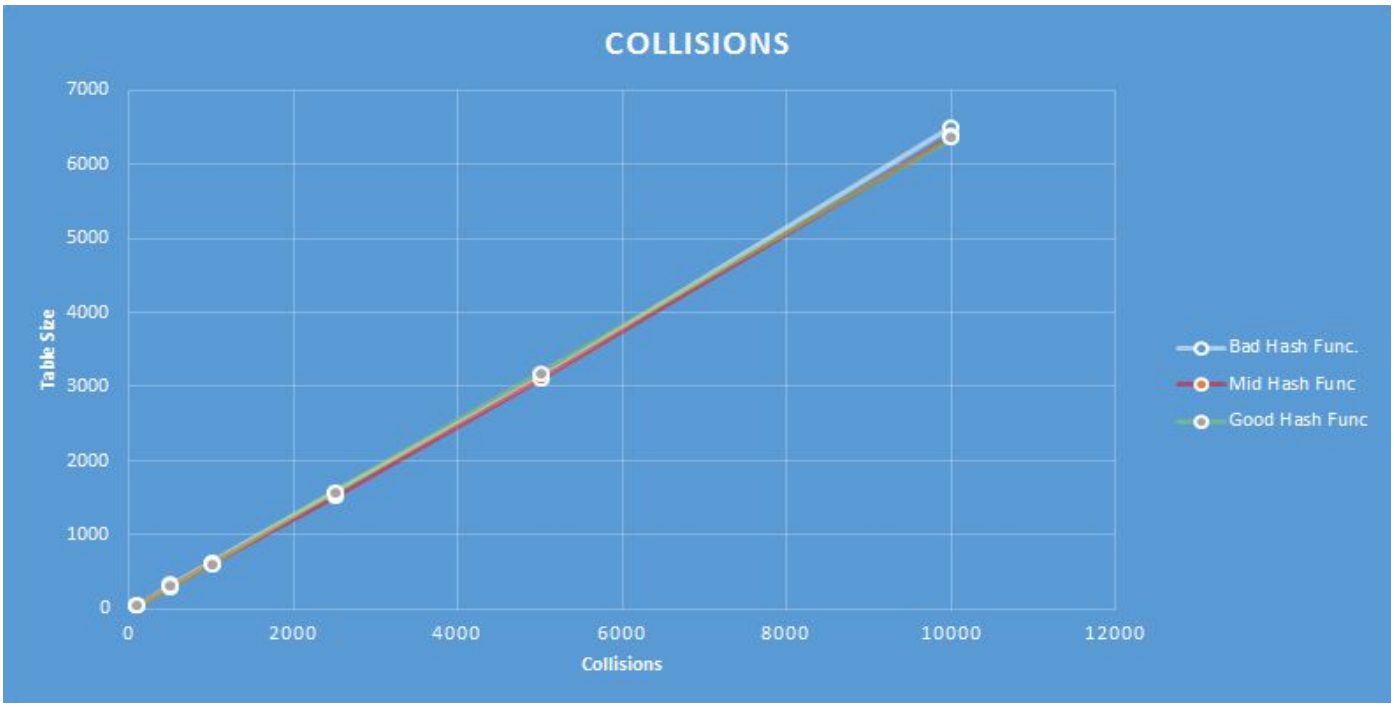
Cooper Pender (u0843147)
11/05/2016
Assignment 10 Analysis

1. Load factor for quadratic probing, basically will regrow and rehash the backing array when too many spots have been occupied. This is done to not only reduce the number of collisions that occur, but also ensure no failures happen when inserting. Load factor for separate chaining will rehash the array of linked lists when the average length of those lists becomes too large. For quadratic probing, a load value of .5 or less will yield good results whereas for separate chaining a load value of under 1 will result in good performance.

2. The bad hashing function I implemented sums up the ASCII values from individual characters of that string, and mod the result. I expect this to be a poor hashing function, since words with similar characters and frequency will result in the same hash, and thus more collisions.

3. For the mediocre hashing function I took an initial value of 7 and 3 times, I multiplied it by 11, and then added it to the current character's ASCII value, finally modding it by the capacity. I expect this to perform better than the last hashing function since it no longer uses a sum, which should eliminate the previous problems.

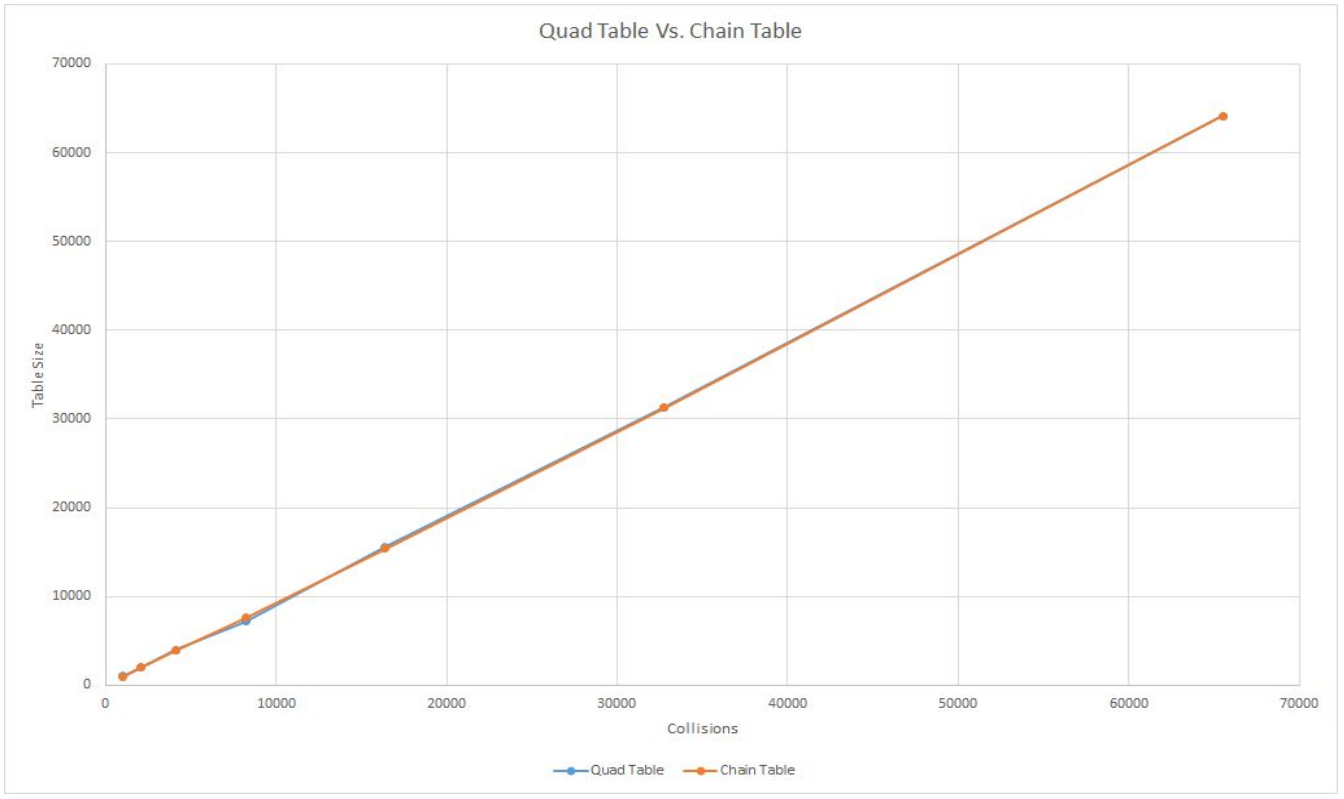
4. The good hashing function I wrote does the same work as the previous function, but instead of multiplying by 11, it multiplies by 128. I expect this to produce more unique hashes and thus less collisions due to multiplying by a higher number each time.

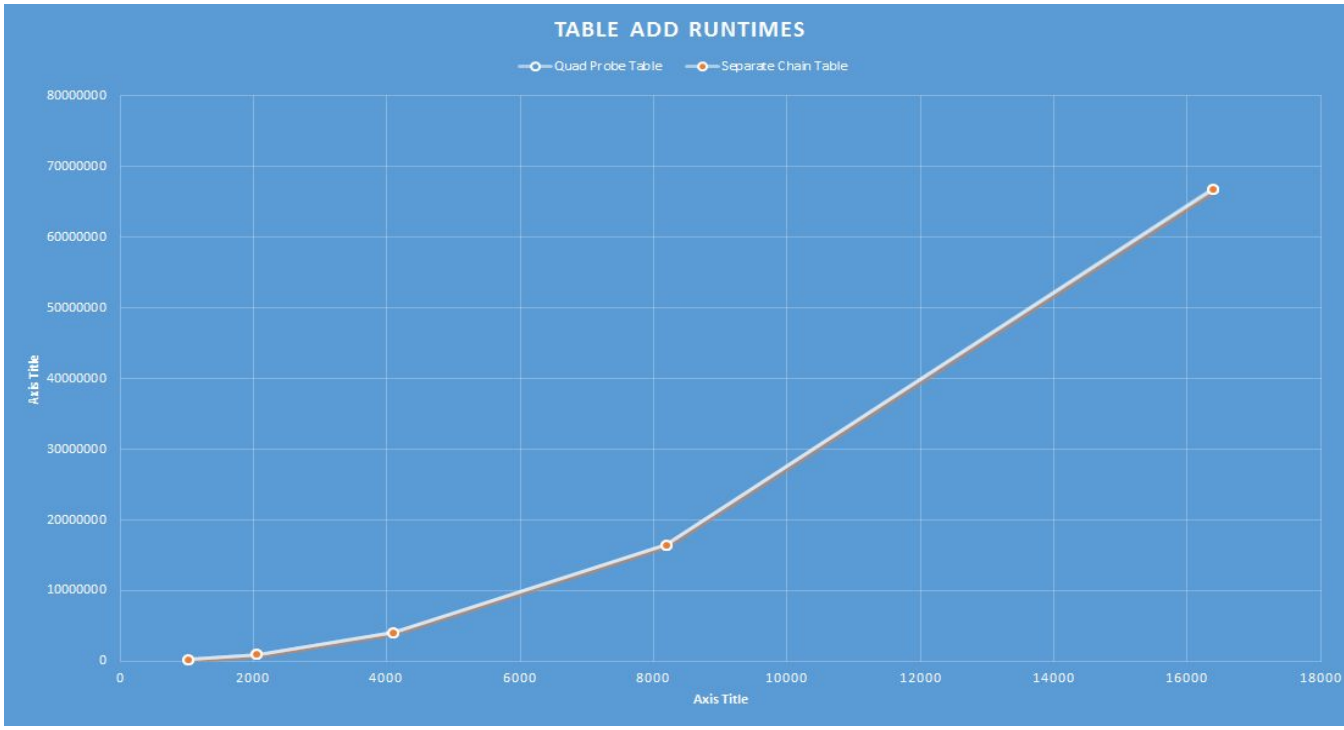


5. For my experiment I tested all three of my hashing functions using a class I wrote, that both detects the number of collisions incurred by each hash, and also tests the running time of each hash function. First I tested different sizes of tables and then I went in and added that same amount of random strings to the table. Inside my table classes, I have a method which returns the number of collisions the table encountered during this process. That was how I conducted this experiment. For timing I simply created a table of the varying sizes, and then added a certain number of strings to it using the different hash functions.
6. The first chart shows the number of collisions each hash function produced. I was surprised at how similar each hash function performed in terms of collisions. I was expecting the difference between the bad and good hash function to be much larger, but this was not the case.

The second chart shows the running time of each hash function. As expected the bad and mediocre hash functions exhibit something closer to **O(N)** running time since they rely on iterating through the full string, whereas the good hashing function shows somewhere between **O(c)** and **O(logN)** behavior.

7. For my experiment on testing the two hash tables I tested two things, first the number of collisions each table ran into while using the good hash function, and the running time for performing an insert on each table. Testing the collisions was very similar to how I tested them in question 5. I used roughly the same procedure as well for the timing, just making sure to use both the chaining and quad tables.





8. My bad and mediocre hash functions have a complexity of **O(N)** where **N** is the number of characters in the string. My good hash function has a **O(C)** complexity, since it only takes the first half of the characters into account. Each of my hash functions performed as I predicted. I also assumed that the worse the hash function was, the more likely

collisions were to happen. Each hash function resulted in roughly the number of collisions I expected, although they ended up being much more close than I thought they would be.
The load factor, when above the earlier specified numbers can have a significant impact on the performance of the hash tables, and could even cause the quadratic probing table to fail during an insert.

9. Implementing a remove method in a hash table would be fairly trivial. You would need to just look up the key, and change the data at that point to null. If you wanted to rehash afterwards that would also be beneficial.

10. I don't believe it would be possible to implement a generic hashtable. This is because each type of data will most likely need its own hashing function. If we were to specify a generic hashing function, then yes it would be possible, but based off of our implementation, no it would not be feasible.

11. I spent around 8 or so hours on this assignment.