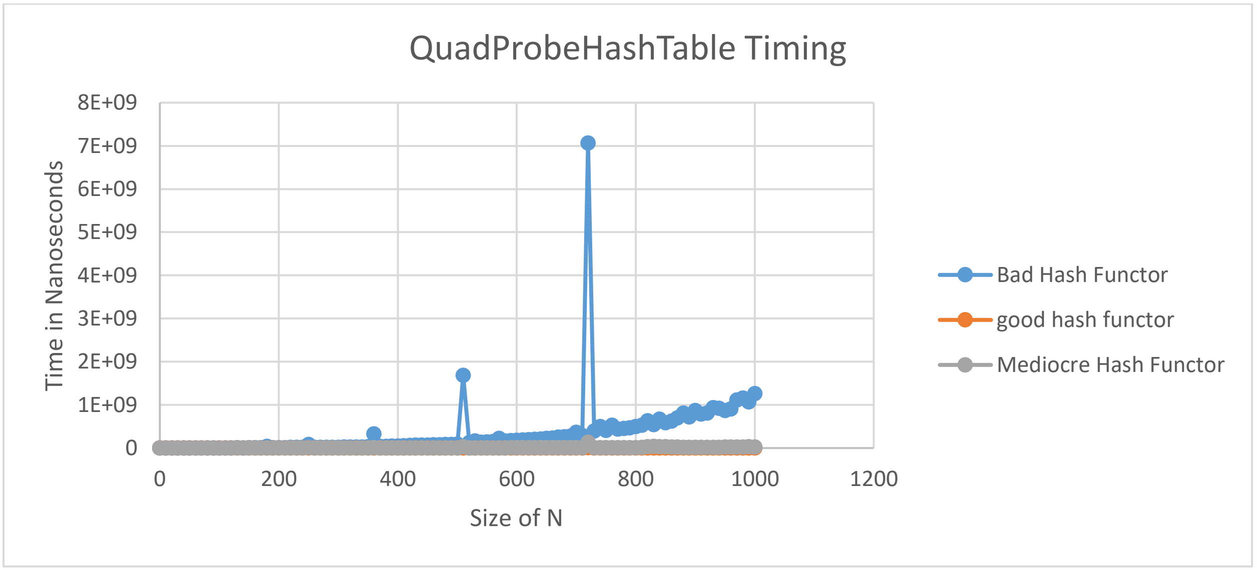
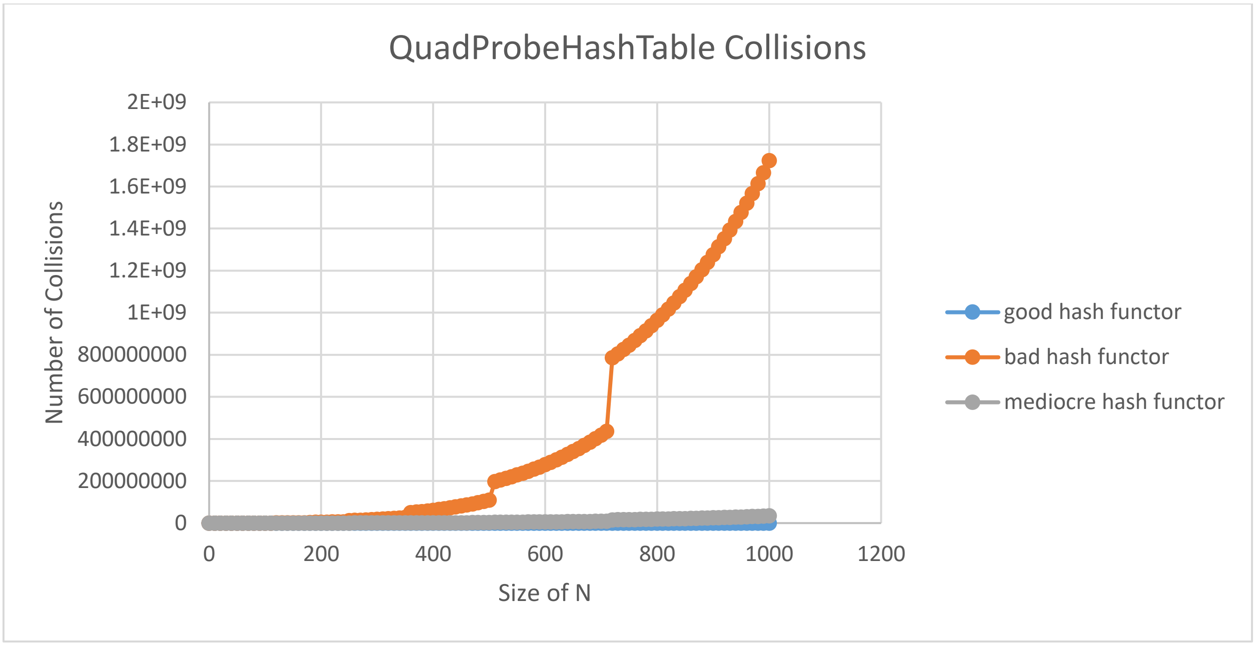
Chenxi Sun

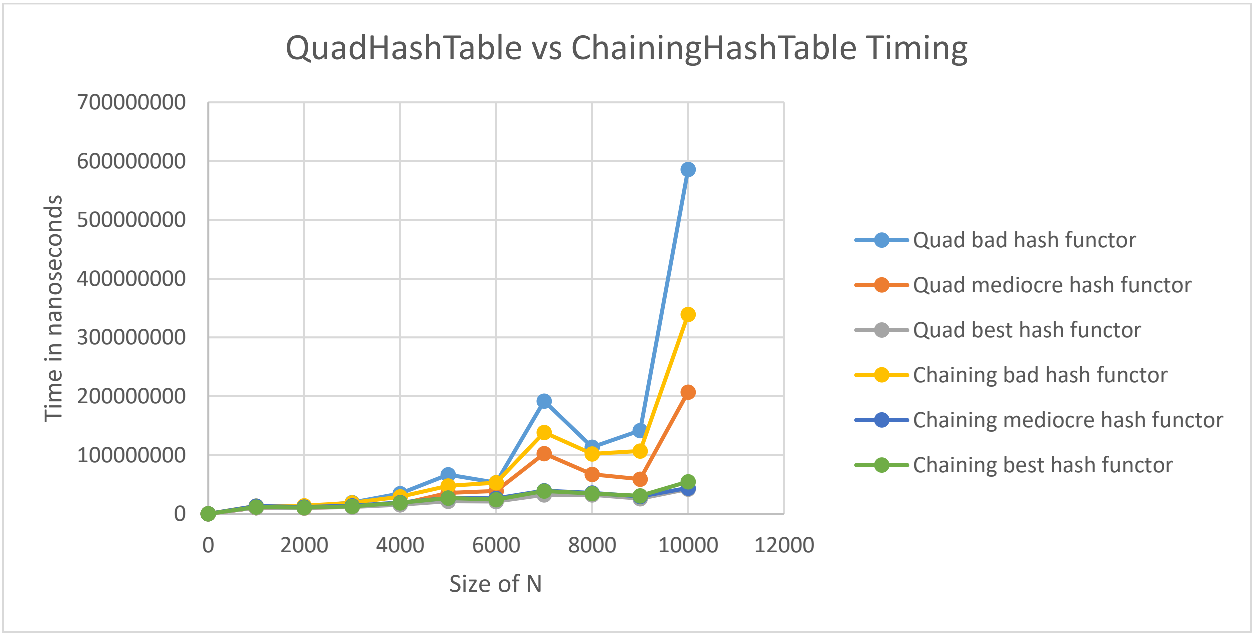
Assignment 10 Document Analysis

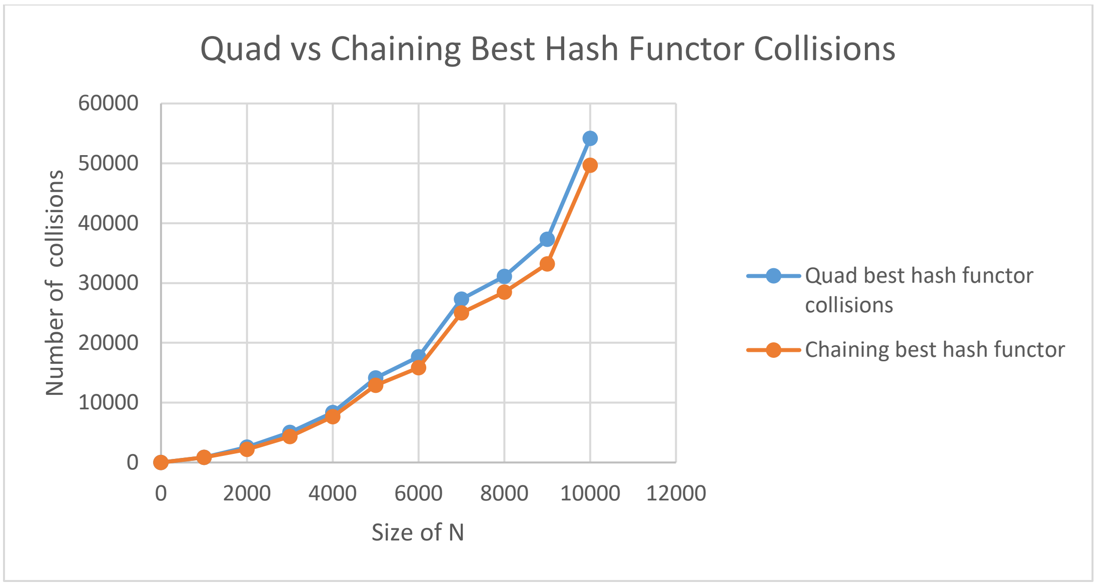
1. The load factor of λ means that it takes average of 1/(1- λ) times to input into the hashtable successfully. For QuadraticProbeHashTable the loadfactor should be less than .5 as it should take 2 tries for the item to be hashed into the table, otherwise the Big O would not be constant and slow. For ChainingHashTable a load factor of 1 should be used as this would also limit the average linkedlist's size in the array to 1.
2. For the bad hash function, I decided to use the lengths of the strings. This is a bad hash function in that it will hash to the same place on the table if the strings inserted have all the same length. This will force the table to use quad probing constantly.
3. For the mediocre hash function, I used a for loop and added all of the values of the chars contained in the string. This is a decent hash function for string in that if the string is same length the hash code will still be different.
4. For the good hash function, I used the same concept of the mediocre hash function but I timed all of the char values in the string by 37. This concept is that if you don't time it by a large value of prime number the difference would still be too small due to the similarities of strings. If you times the hash value by 37, a prime number, not only would the difference by magnified due to the prime number the hash code will also be more unique.





5. The efficiency and performance of all three hash functors are shown in the plot above. From the plot the bad hash functor has the highest running time as size increased and there is an occasional blip, which could mean that the hashtable is resizing. Depending on the size of the table resizing an array can be costly in performance. The fastest time is the good hash functor. For the second plot, the number of collisions for each hash functor is shown. The number of collisions for bad hash functor is particularly high, for good hash functor and mediocre hash functor the number is quite low. The way I completed this experiment is that I created a generate random string method that takes the string length as a parameter. The length I used is 5 and I hash the random string into the hash table from 0 to 1000 at an increase of 10. I outputted the timing and collisions number onto the console and plotted the data.





6. For both the Quad hash table and Chaining hash table, the performance and efficiency are shown in the 2 plots above. One of the plot show the running time of the quad hash table and chaining hash table for the bad, mediocre and best hash functors. In all three cases the performance of chaining hash table out performs quad hash table. For the numbers of collision the quad and chaining hash tables exhibit about the same numbers of collisions. The reason why Quad is much slower is that it has to probe and the arithmetic of probing can add up and will slow the process of hashing. For chaining hash table there is no such problem, it simply has to dynamically increase the linkedlist and add to it. Space wise it is better to use a quad hash table as the chaining hash table has to create an array of lists which is space heavy. The way that I did this experiment is that I used a random string generator function and input a Random number from the function Random.nextInt(100). By putting 100 into Random.nextInt() I generate a random number between the value of 0 to 100 and put that into the parameter of randomString. This will create a random string with length of 0 to 100. I then hash it into the 2 different hash tables and plotted the data.
7. The cost of the bad hash functor is constant of O(C). The cost of the mediocre and good hash functor is just the length of the string or O(N), as it has to add up each of the char value in a for loop. The 3 different hash functors did what they were expected and this is show in the 2 plots above number 5. The bad hash functor takes longer and has exponentially more collisions than mediocre or best hash functors. Mediocre has 2 times as collisions as best hash functors as shown in the plot above number 5 and 6.
8. The load factor for the quad hash table is just .5 and it is basically for how full the table is, so size/capacity. For the chaining hash table the load factor limit is 1 and is also size/capacity as it represent the average size of the list in the array of lists. The load factor is important as it deals with secondary clustering by increasing the table capacity and rehashing it limits the amount of collisions and quad probing to a minimum value. It will also resolve infinite loop as the table is twice the number inserted due to resizing at load factor limit of .5 for quad and 1 for chaining.

9. For quad it is much easier to implement remove by using false remove. This implies that you mark a value of -1 value in the hash table and ignores it in the hashing function for removal. The reason you want to do this is because if you actually delete the value in the array, you then have to shift the array which is O(N). If you don't, you will risk into running nullpointer exception. For chaining you can just use list.remove() function since the list size is small and the big O is still O(c).
10. Yes, it is possible to make the hash table generic. Instead of Set<String> for interface you would do Set<E> and then change the hash functor to generic also by doing functor<E>. You will also have to personally design a hash functor that will use comparator to create a hash value. So on the interface you will also need to do implements Comparator<? Super E> .
11. I spent about 5 hours on the assignment, most of it is spent on doing the document analysis.