Cameron Pickle

1. Who is your programming partner? Which of you submitted the source code of your program?

Daniel Avery. I will submit the code.

2. Evaluate your programming partner.

He does a good job and works hard in the class.

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

In quadratic probing λ means the amount of the array that is full. In order to have a good performance for this strategy λ should be 0.5 or less. This will help to reduce clustering and collision.

4. 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).

We used the length of the given String. We expected this to perform poorly because many strings will have the same length. Most words will have very close lengths and there will be many collisions. We only had a variation of strings from length one to ten and so every string after the first ten had collisions.

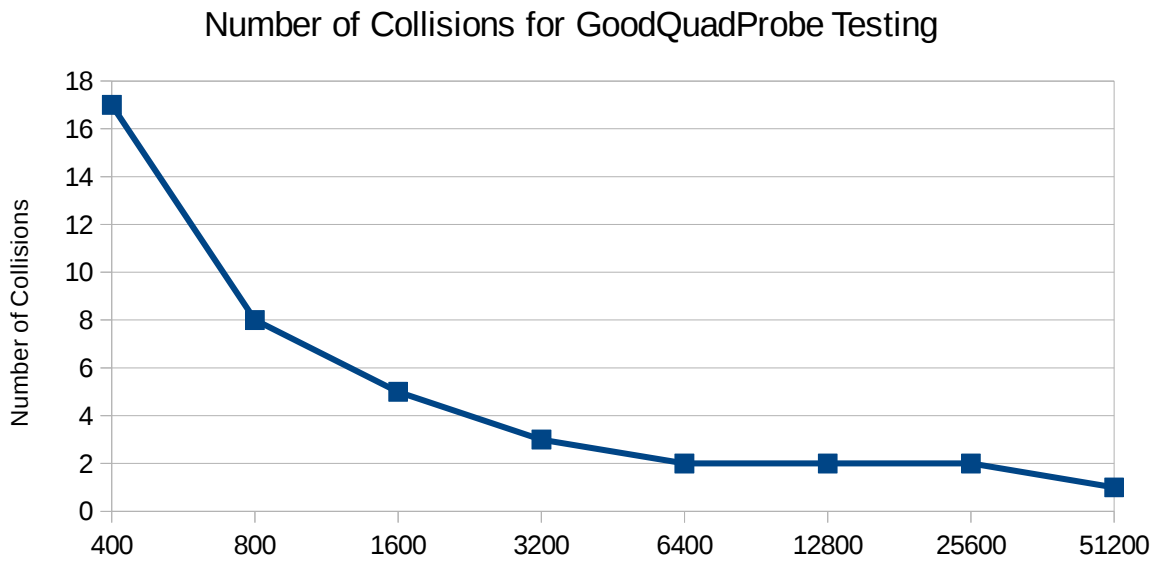
5. Give and explain the hashing function you used for MediocreHashFunctor. Be sure to discuss why you expected it to perform moderately (i.e., result in some collisions).

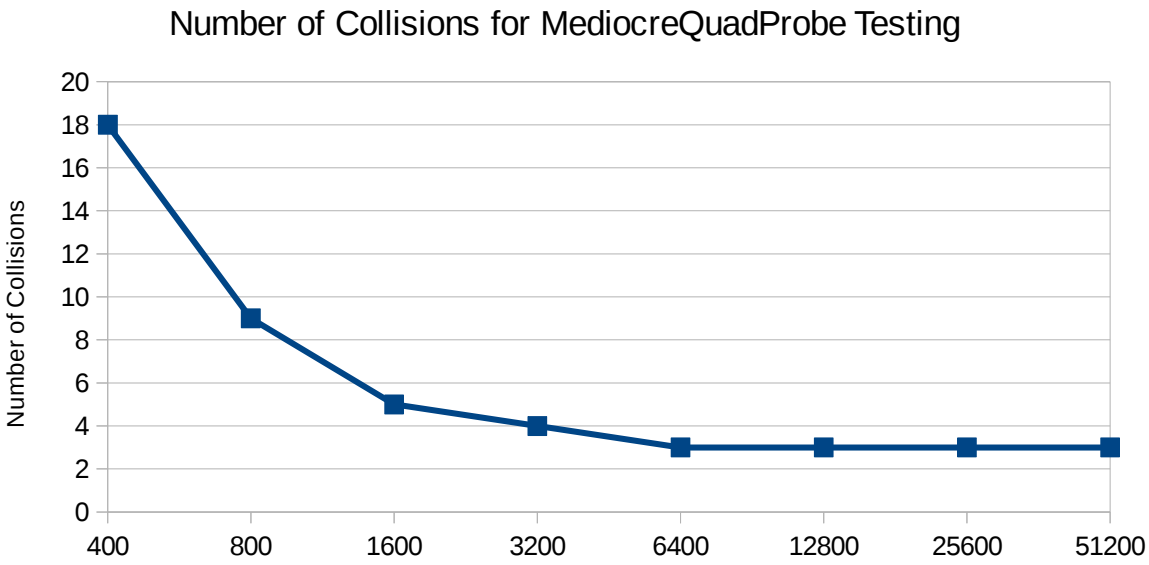
The Mediocre functor will have a medium amount of collisions but increase as the size of n increases due to the smaller possible outcomes of the functor. We designed it so that we used a combination of the ascii value of the first and last char along with the length of the string. By doing this we have approximately 26*26*10 possible outcomes of the functor. This is a pretty large number to give decent results.Also by thinking about the method used to create this functor the only probable collisions will be words that are anagrams. While there are still a lot of anagrams it is relatively small in comparison to the whole number of words.

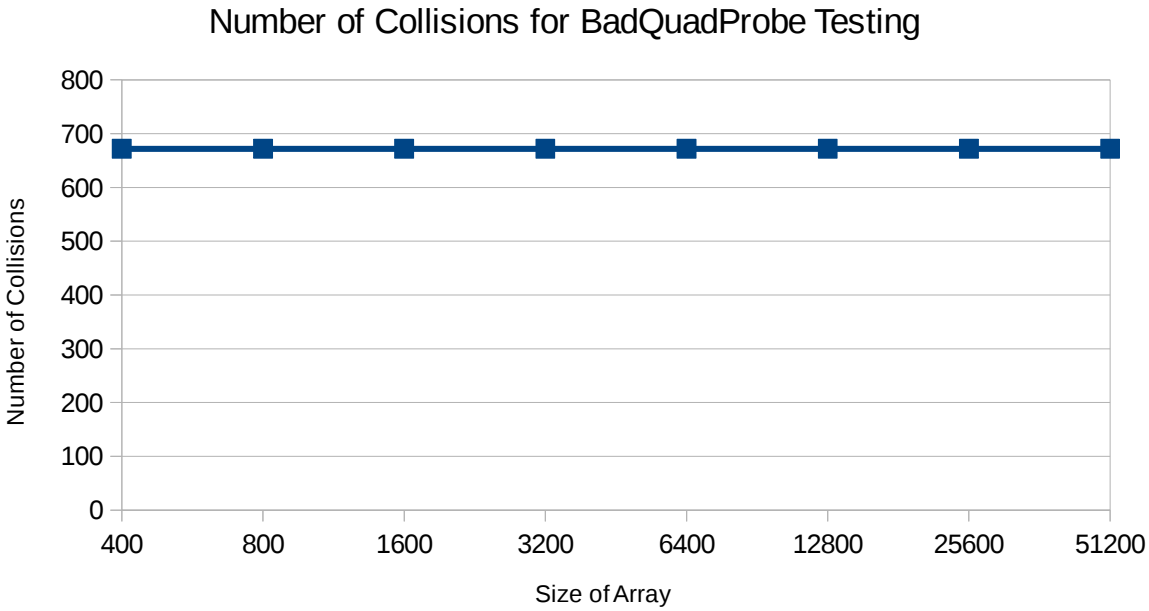
6. Give and explain the hashing function you used for GoodHashFunctor. Be sure to discuss why you expected it to perform well (i.e., result in few or no collisions).

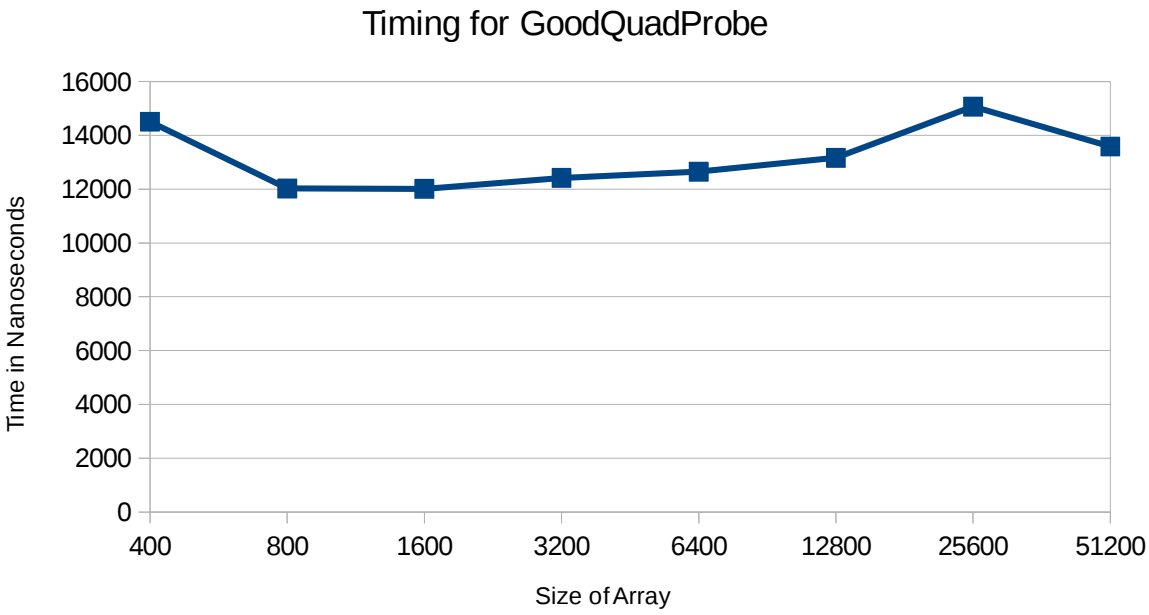
For our good hash function we took all the ascii values of each char in the string and multiplied them to a prime number and then performed a few more operations using prime numbers to get as diverse results as possible. This gave us the best results for a has functor. We were inspired to perform those operations based on similar operations we saw while searching good hash functors online.

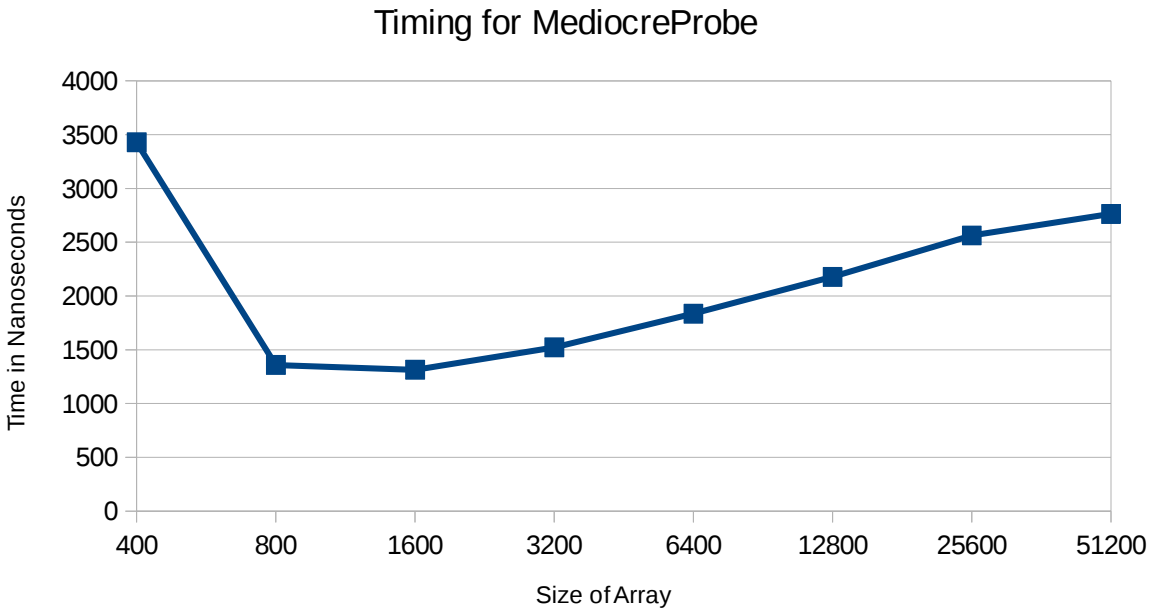
7. Design and conduct an experiment to assess the quality and efficiency of each of your three hash functions. Carefully describe your experiment, so that anyone reading this document could replicate your results. Plot the results of your experiment. Since the organization of your plot(s) is not specified 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.

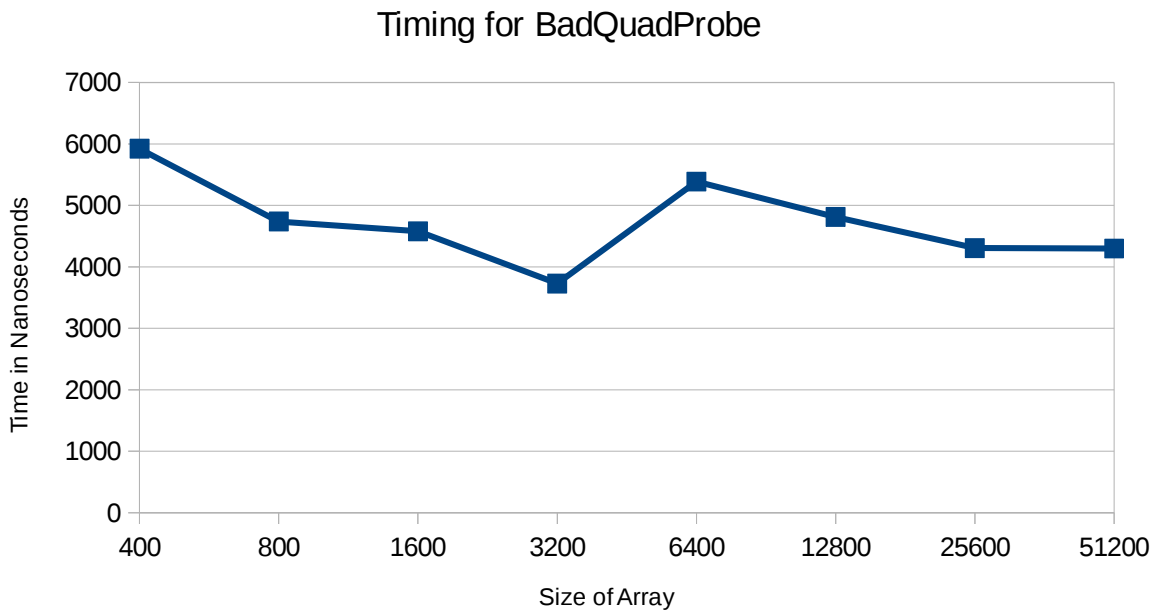

Number of Collisions for GoodQuadProbe Testing
Number of Collisions
18
16
14
12
10
8
6
4
2
0
400
800
1600
3200
6400
12800
25600
51200


Number of Collisions for MediocreQuadProbe Testing
Number of Collisions
20
18
16
14
12
10
8
6
4
2
0
400
800
1600
3200
6400
12800
25600
51200


Number of Collisions for BadQuadProbe Testing
Number of Collisions
800
700
600
500
400
300
200
100
0
400
800
1600
3200
6400
12800
25600
51200
Size of Array


Timing for GoodQuadProbe
Time in Nanoseconds
16000
14000
12000
10000
8000
6000
4000
2000
0
400
800
1600
3200
6400
12800
25600
51200
Size of Array

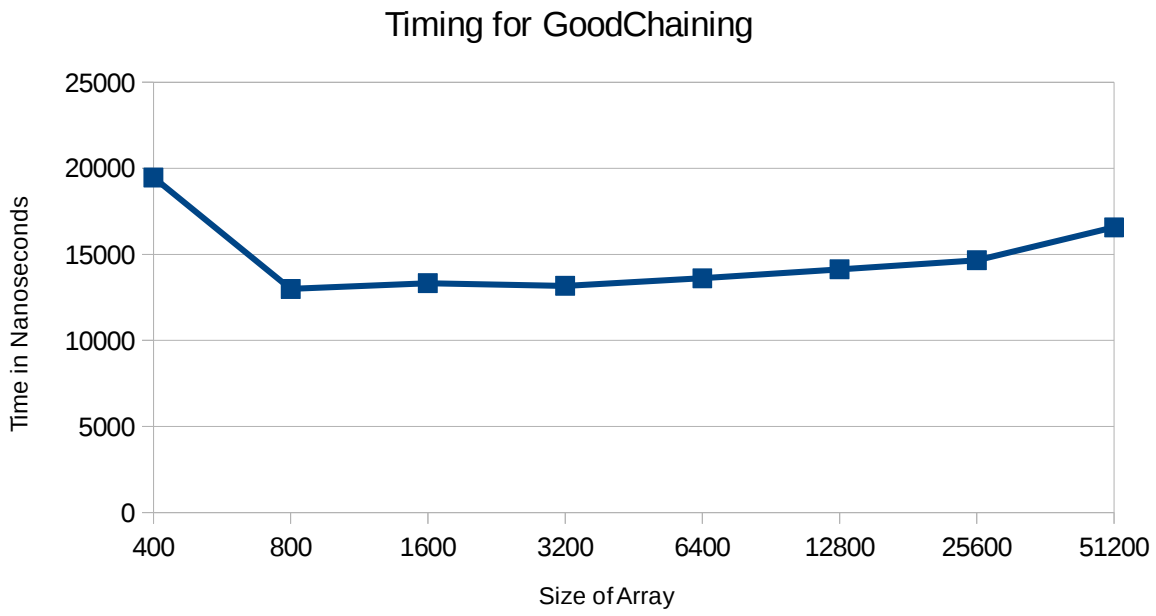

Timing for MediocreProbe
Time in Nanoseconds
4000
3500
3000
2500
2000
1500
1000
500
0
400
800
1600
3200
6400
12800
25600
51200
Size of Array

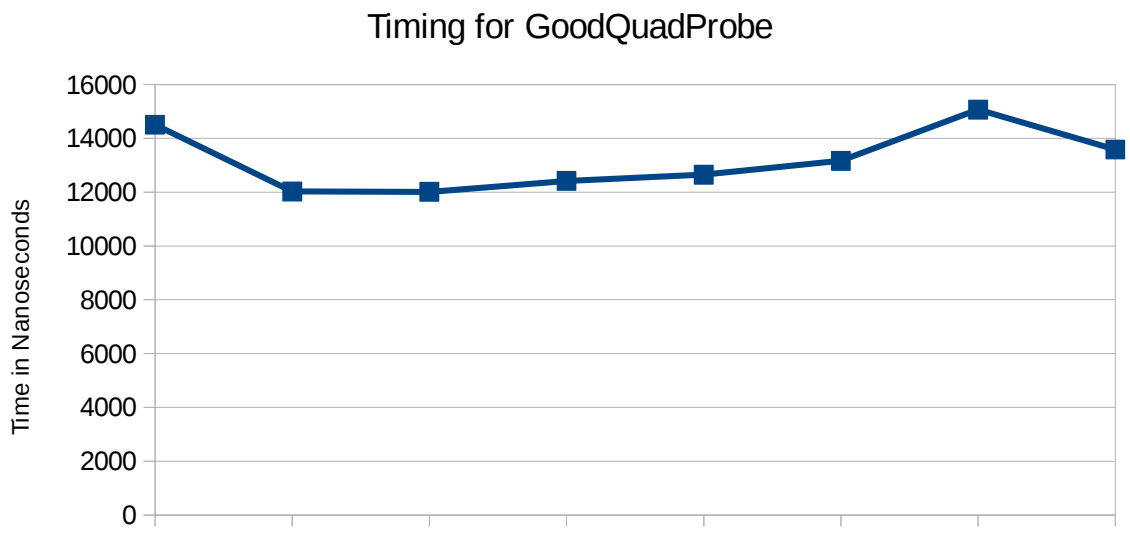


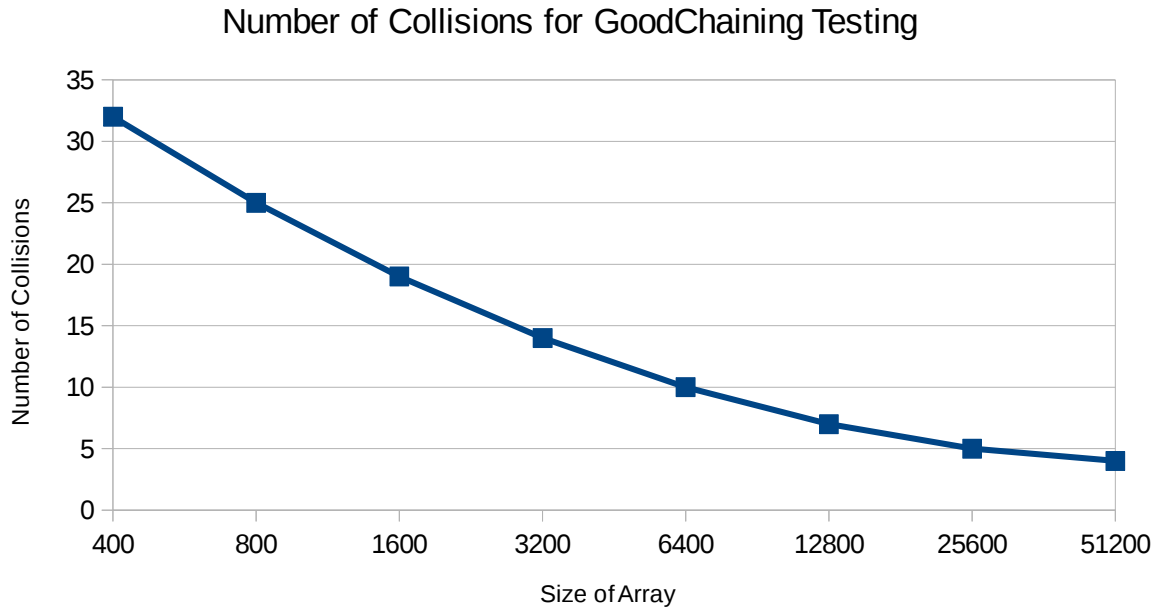
The results for the collisions happened exactly as we expected. The Bad hash functor resulted in only 10 distinct values causing many collisions in all table sizes because the hash functor had a limit of ten values. The Mediocre hash functor has much better results by multiplying the length, and ascii value of the first and last char of the string together. This offers approximately 26*26*10 possible values which is considerably larger. The Good hash functor has the greatest numerical diversity by using the ascii value of every char in the string and multiplying it by a prime number and then performing other operations with prime numbers to obtain as diverse of results as possible. The actual values of the collisions were quite low and decreased as the array size increased, just as expected.

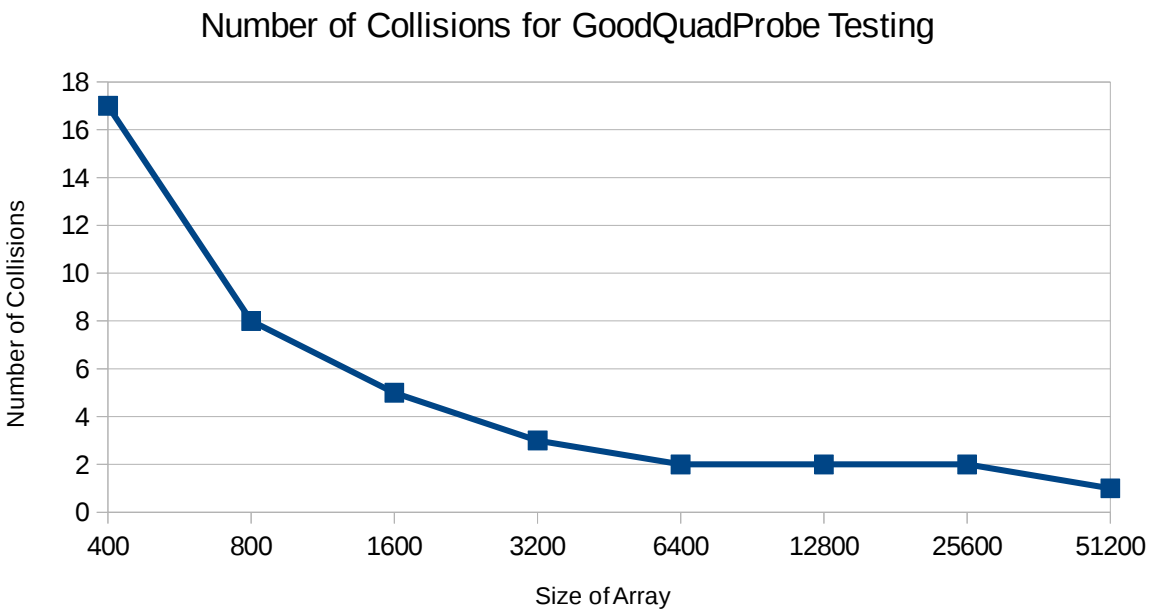
The timing also worked how we expected. The charts for the most part showed how the hash tables were a constant time. The variance in the timing would come from the background tasks such as from the OS or other running applications that may spike the cpu use.

8. 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.


Timing for GoodChaining
25000
20000
15000
10000
5000
0
Time in Nanoseconds
400
800
1600
3200
6400
12800
25600
51200
Size of Array


Timing for GoodQuadProbe
16000
14000
12000
10000
8000
6000
4000
2000
0
Time in Nanoseconds


Number of Collisions for GoodChaining Testing
35
30
25
20
15
10
5
0
Number of Collisions
400
800
1600
3200
6400
12800
25600
51200
Size of Array



The results of these experiments did not come out quite as I expected. I thought that the chaining hash table would be faster and have less collisions. I thought that by having the values stack up in arraylists they would be less likely to have collisions because of clustering. In the actual results it clearly shows that the quad probing has less collisions and also has a faster running time than the chaining hash table.

9. 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.)

All of the functions for our hash table performed in O(C) time as expected. The hash functor made a difference and the load factor also had an effect on the performance but the functions themselves all were O(C). The collisions also meet our expectations. The Bad hash functor resulted in only 10 distinct values causing many collisions in all table sizes because the hash functor had a limit of ten values. The Mediocre hash functor has much better results by multiplying the length, and ascii value of the first and last char of the string together. This offers approximately 26*26*10 possible values which is considerably larger. The Good hash functor has the greatest numerical diversity by using the ascii value of every char in the string and multiplying it by a prime number and then performing other operations with prime numbers to obtain as diverse of results as possible. The actual values of the collisions were quite low and decreased as the array size increased, just as expected.

10. How does the load factor λ affect the performance of your hash tables?

The higher the load factor the slower the hash table is. You can see this in the timing results that the hash tables perform slower when they have a smaller size and are being filled in comparison to the hash tables that are larger they are a lot slower as seen in the timing graphs.

11. Describe how you would implement a remove method for your hash tables.

In order to implement a remove you would have to add a boolean value to each position in the array to show whether it has been deleted or not. Then when you add a new value if a item has been deleted you may replace it in the array as if it were an empty space. Also when searching you would just skip over that space but continue in the quadratic manner to assure that the value isn't farther on in the array.

12. 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?

Yes, there are two main things that would have to be changed to have a generic implementation. We could have to change the back array storage to be of generic type. Also you would have to have a hash functor that would guarantee a hash value for any type of object given.

13. How many hours did you spend on this assignment?

We spent about 8 hours on this assignment.