Abdulaziz Aljanahi
U0901606

1) The load factor for the quadratic probing represents a percentage of how full the table is, the higher the load factor the higher the chances of collisions happening. And it has a good performance for a load factor that is equal to half the array size. Any higher and it starts causing a lot of issues. For the separate chaining the load factor represents, how the size of the separate chain inside the array. A good load factor is around 10, however the chaining method's load factor performs at a linearly, so the optimal load factor of separate chaining may vary.
2) The bad hashing function I picked, simply divides the number of characters in a word by 2, and returns that number as a hash code. This is bad since the average word length in the English language is 5 letters, while the longest common words are around 10 letters long. So, there will be a lot of collisions. The function got 1278810 while inserting 3000 words.
3) The mediocre function I chose does 5 things:
    i) first it checks if the word length is even or odd.
    ii) If even, checks if the letter is bigger than m (middle letter in the alphabet) a prime number multiplied by the length, and the ASCII code of the middle letter in the word is added to the multiplied length.
    iii) if the letter is smaller than m does the same thing as (ii) however multiplies by a different prime number.
    iv) If the word length is odd, does the same thing as (ii) & (iii) however uses 2 different prime numbers.

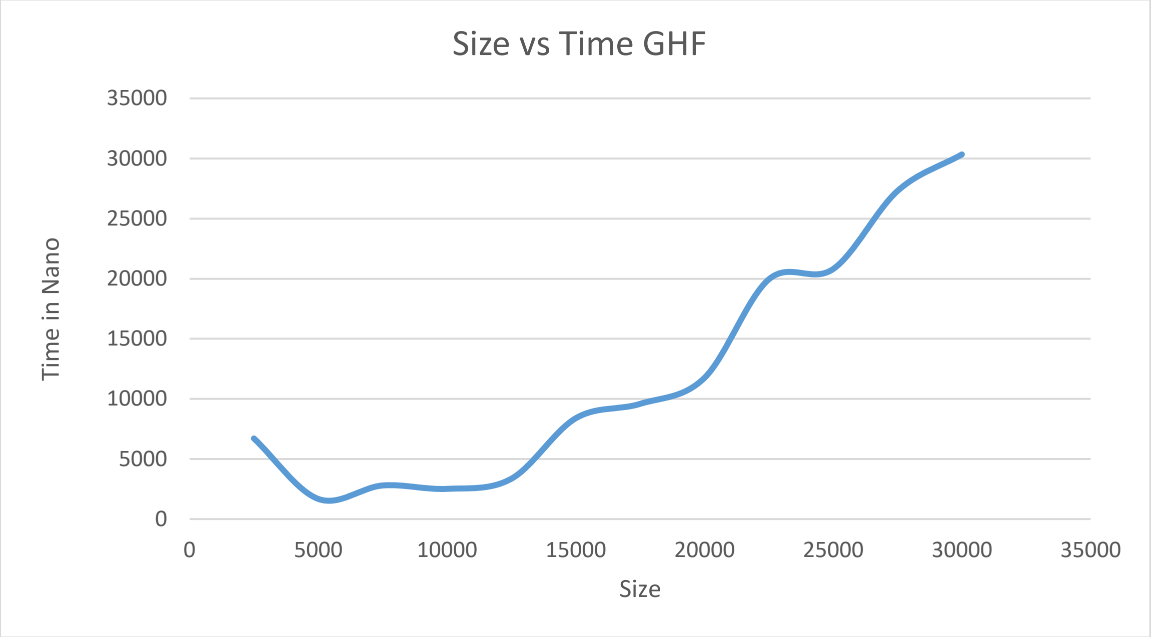
    The prime numbers I chose were 11,47,2,27 these picks were random, I just made sure that they were prime numbers, to reduce the number of duplicates.

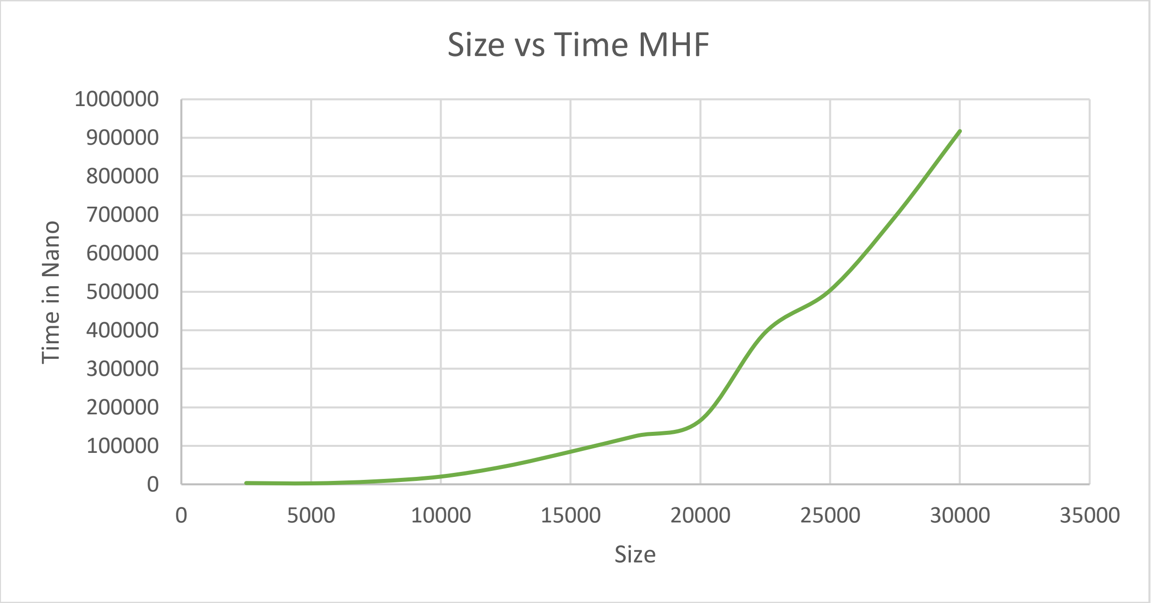
    This function is mediocre, since it still generally depends on the length of the word. However, the addition of the ASCII code of the middle char, and choosing different prime numbers, makes it much less likely to produce duplicate codes. The function got 80698 while inserting 3000 words.

4) The good hash function I chose takes a word, and splits it into two halves, the first half is converted to upper case, and the second is converted to lower case. Then generates a generates the code by multiplying each ASCII character code by dividing it by half and, multiplying it with 37.
The same thing is done for the other lower case half; however, it is multiplied by 13.
This function is good since it does not depend on the word length, it creates the code by generating a number from each character's code. This helps create almost a unique number for each word, the multiplication and division only adds more complexity to the number. Also, the fact that each half is converted to lower and higher case, reduces the chances of having the same code if two words share the same letters. This function got 3561 collisions while adding 3000 words to it.

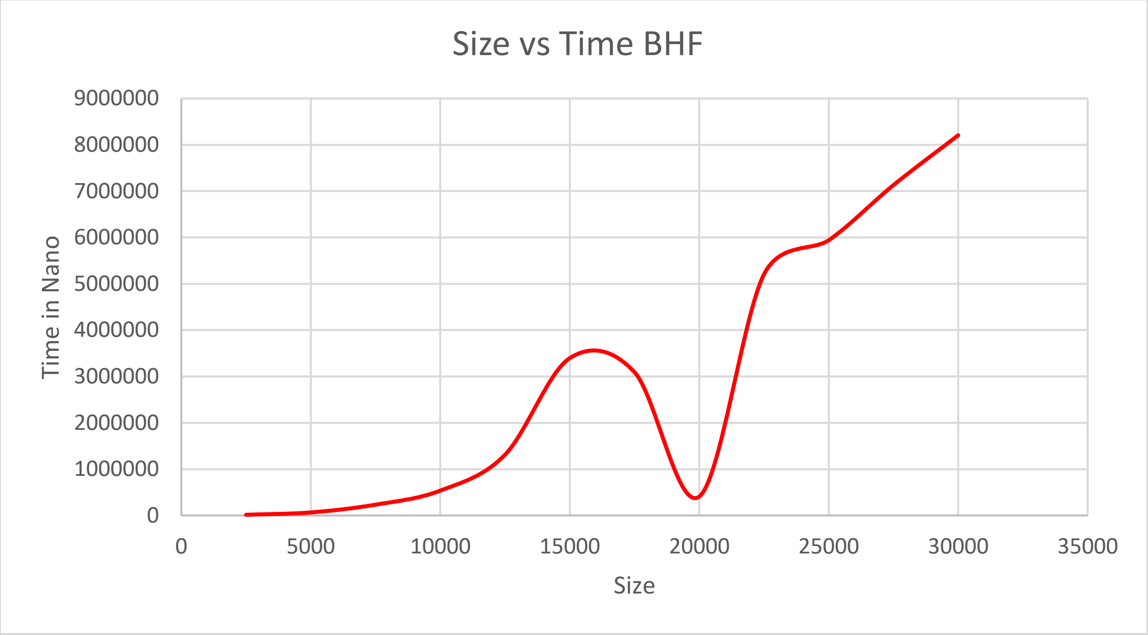
5) The experiment to measure the performance of the hash functions was simply creating multiple arrays ranging from size 2500 to 30000, and increasing by an increment of 2500. With 10 letter words, randomly generated words in them. These words are then added to the hash table, and the average time of addition is taken, alongside the number of collisions.

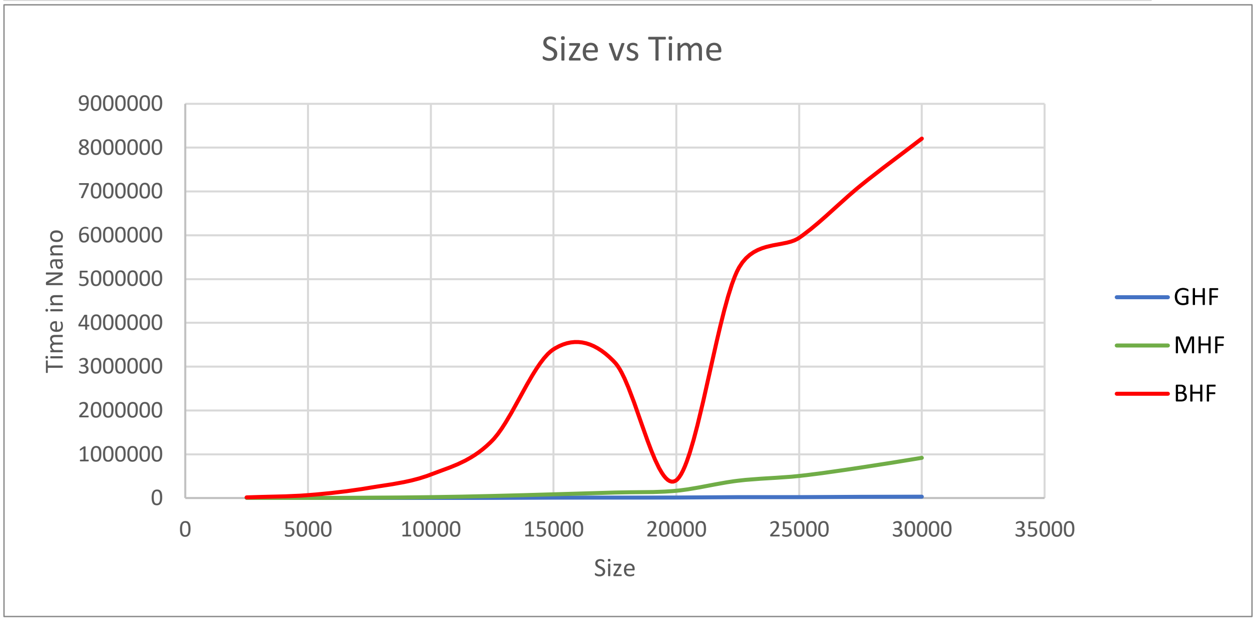
Abdulaziz Aljanahi
U0901606





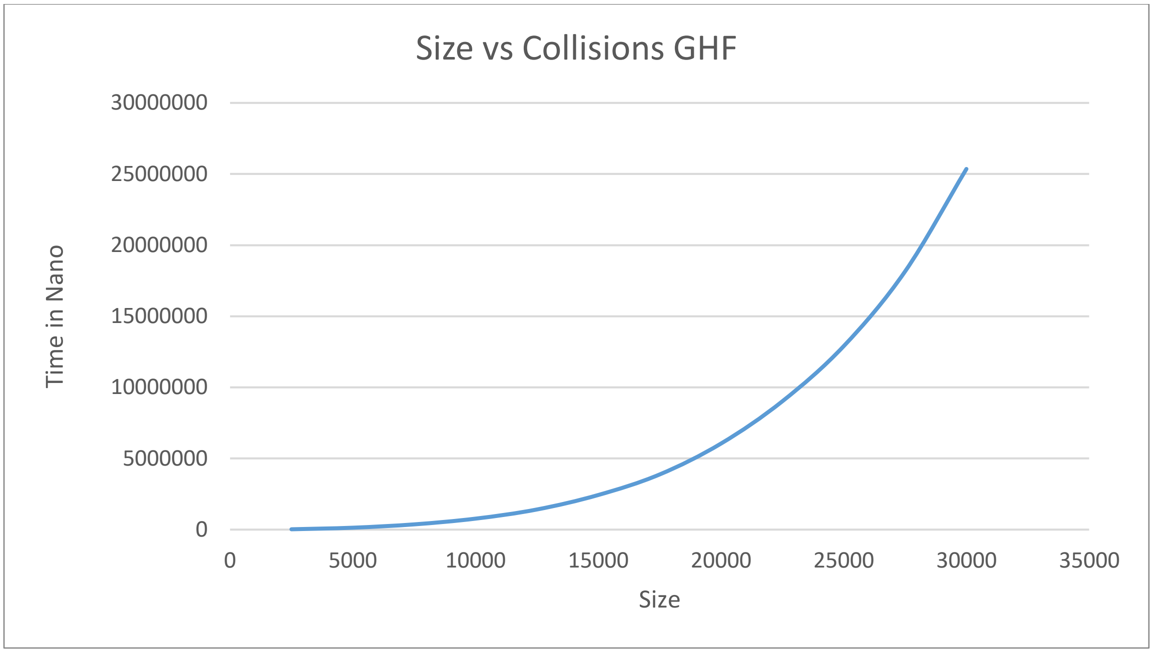
Abdulaziz Aljanahi
U0901606

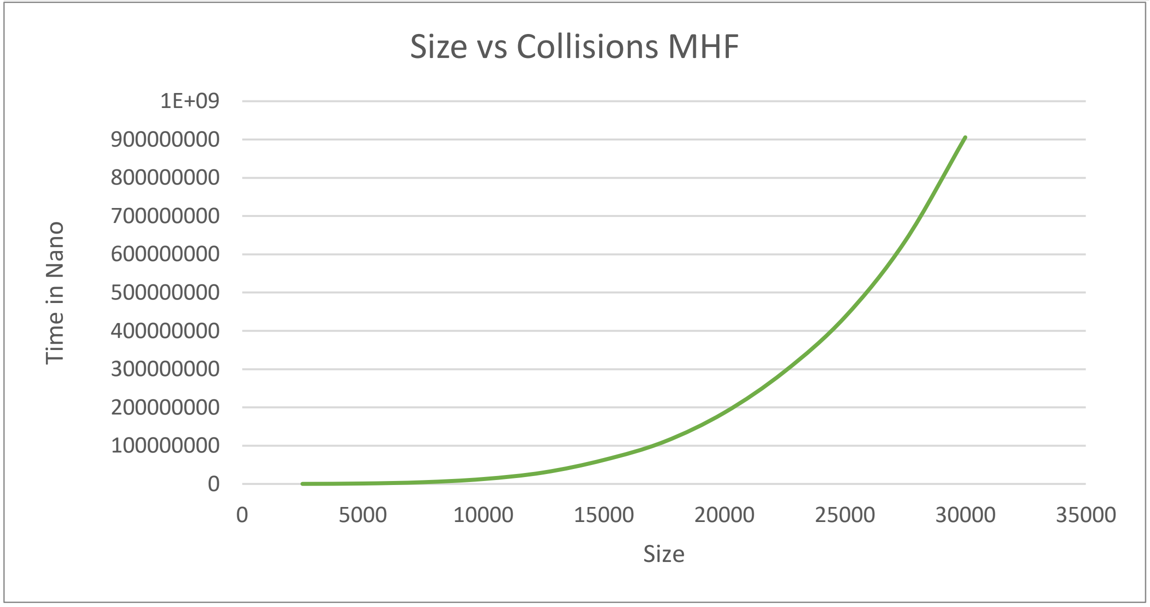




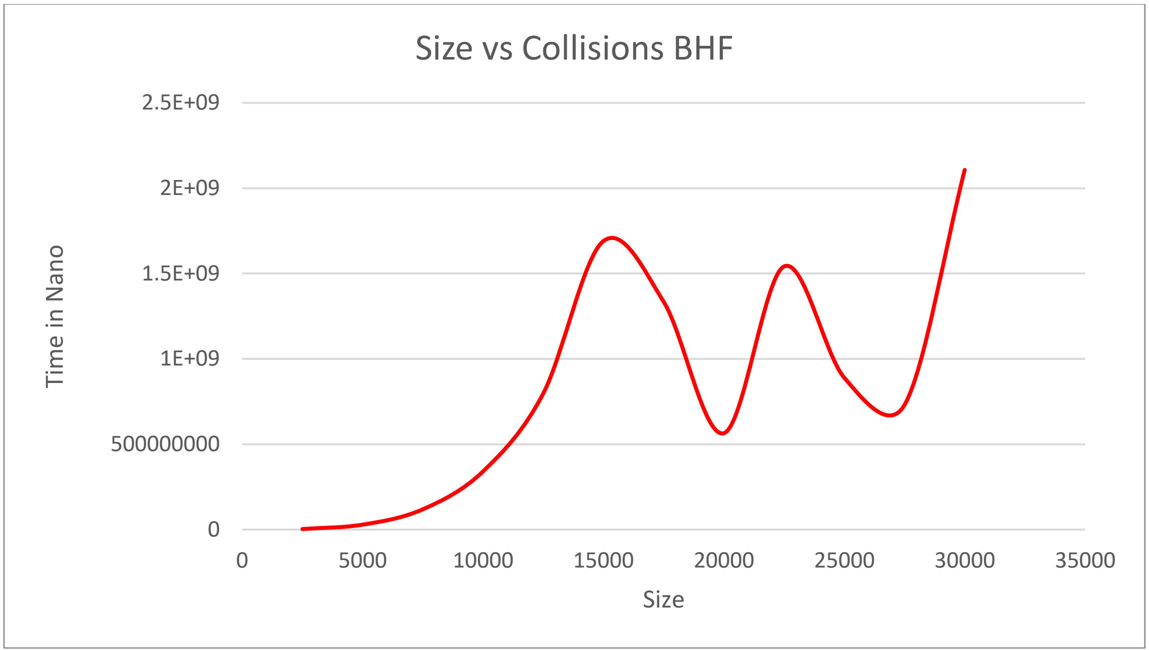
These graphs represent the size vs time of each hash function. When compared with each other, the GHF performs significantly better than the BHF, and has an O(C) behavior. Even though the time is not significantly better the MHF, meaning that the GHF could be optimized a little more, it is still better, and as the size gets larger the difference would be higher.

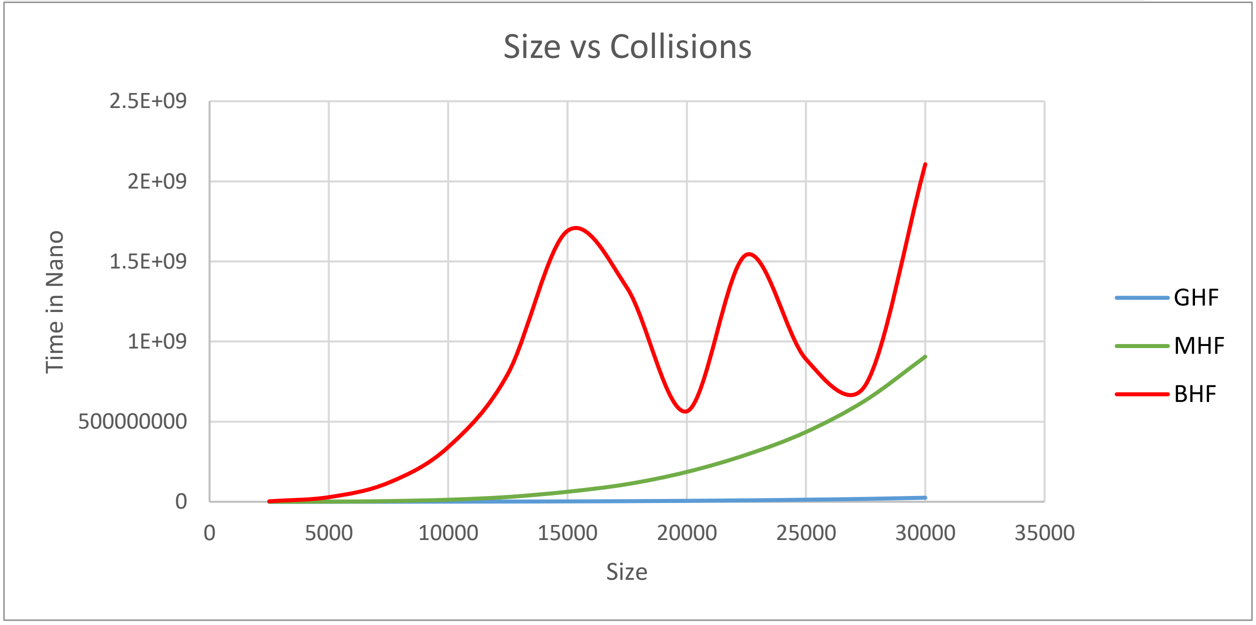
Abdulaziz Aljanahi
U0901606





Abdulaziz Aljanahi
U0901606

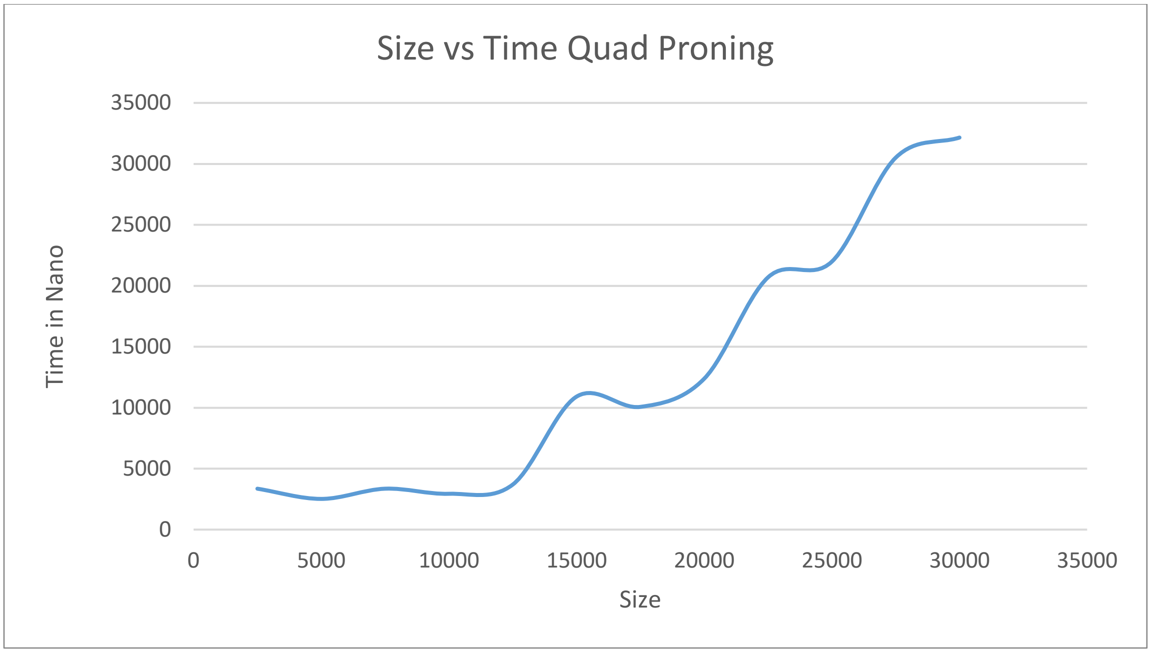


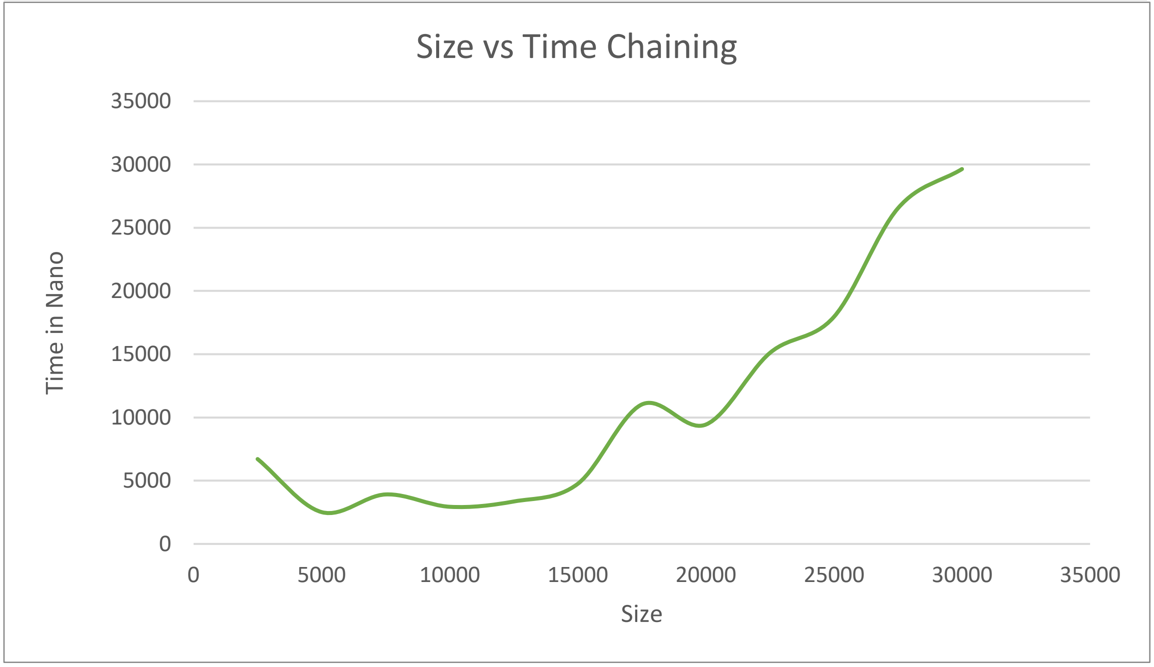


These graphs represent the size vs collisions of each hash function. When compared with each other, again the GHF performs significantly better than the BHF, and has an O(C) behavior. It also performs significantly better than the MHF.

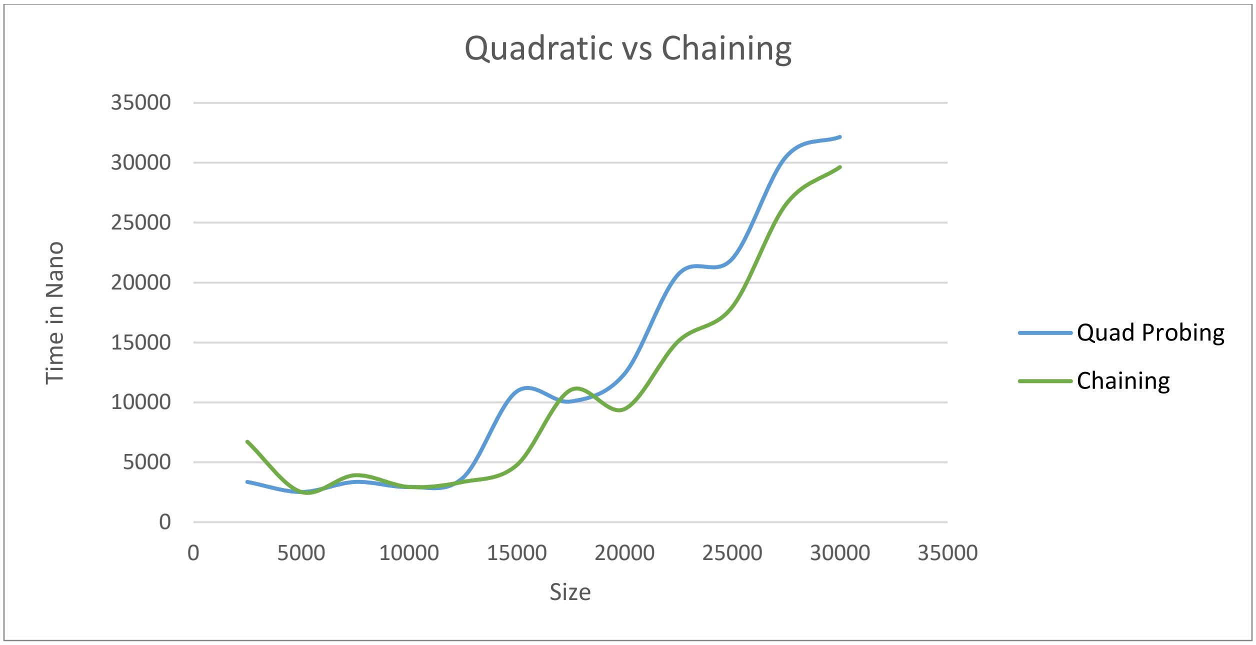
6) The experiment to measure the performance of the quadratic probing and Separate Chaining functions was similar to the previous one, it starts by creating multiple arrays ranging from size 2500 to 30000, and increasing by an increment of 2500. With 10 letter words, randomly generated words in them. These words are then added to the hash Quadratic Probing table, and the average time of addition is taken, alongside the number of collisions. The same process is repeated for the Separate Chaining table

Abdulaziz Aljanahi
U0901606



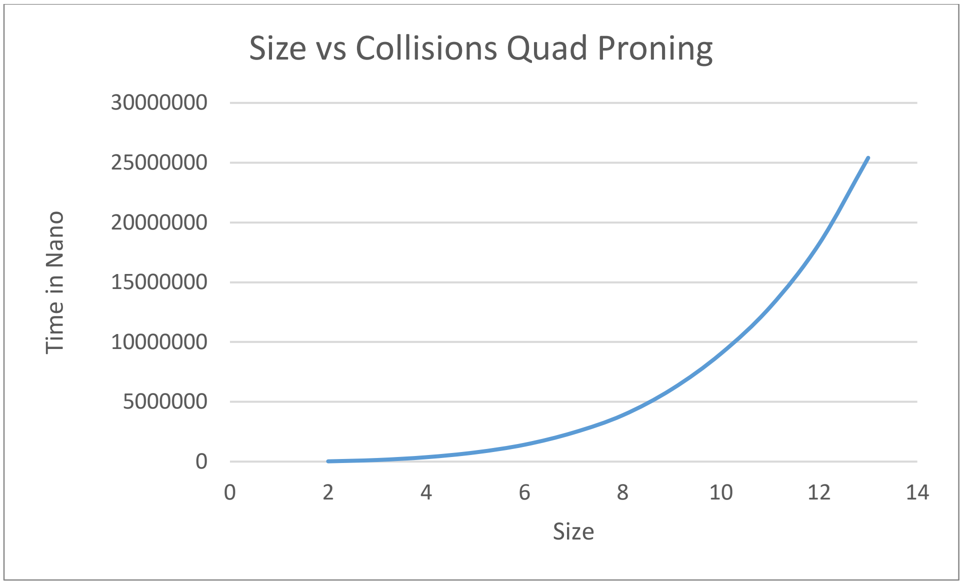


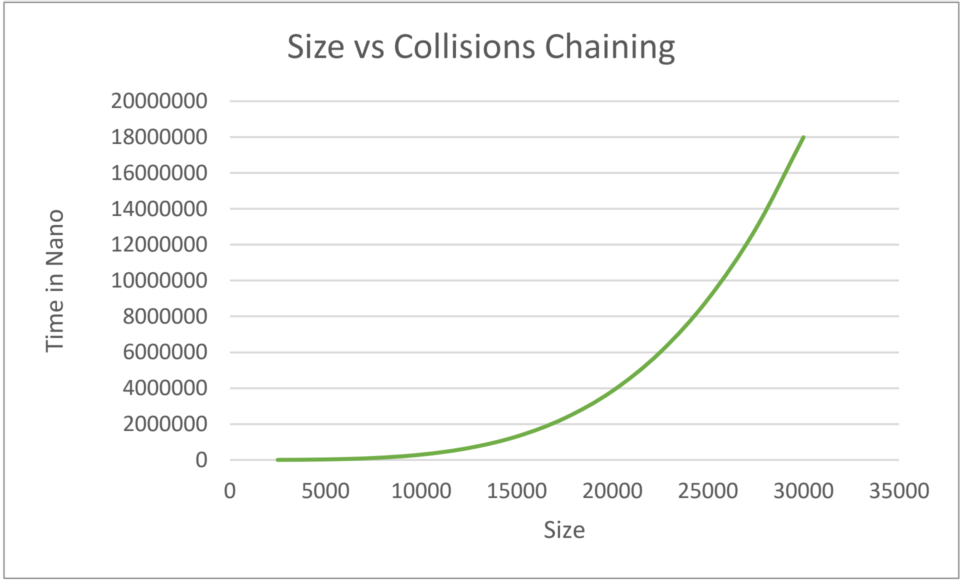
Abdulaziz Aljanahi
U0901606



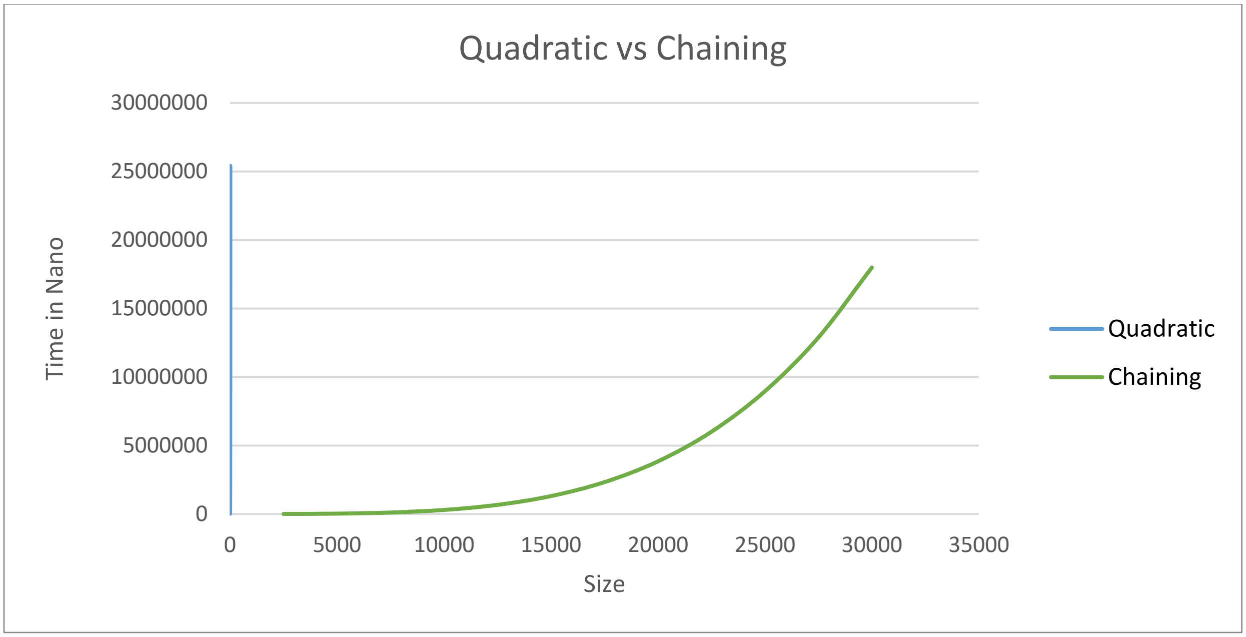
Both the quadratic probing and the chaining method have very similar runtimes when it comes to addittion, the chaining seems slightly better in the graph, however that's a very small difference that could be ignored.

Abdulaziz Aljanahi
U0901606





Abdulaziz Aljanahi
U0901606



The collisions on the Quadratic probing and Separate chaining methods have a very similar behavior. However, when compared next to each other the Quadratic probing is much worse. Than the chaining. This could be attributed to how I chose to implement the chaining method. My chaining method has a linked list in each space, where I would simply add the array. And the collisions are simply calculated, by checking if the current Hashcode has an array in it, add the size of the array to the number of collisions.

7) The average runtime of a hash-table is O(C), and when looking at the good hash function's graph, it has an O(C). and the worst case is O(n) and when looking at the BHF even though it has an abrupt dip in the middle, it still has a linear behavior to it if the dip is ignored. So yes they behaved as expected.
8) The higher the load factor λ the more chances of collisions, the higher chance of collision the longer the runtime. This is apparent when looking at the individual graphs of the functions, the more items are added to the table the longer the run time it for the operation.
9) I've spent around 17 hours on this assignment.