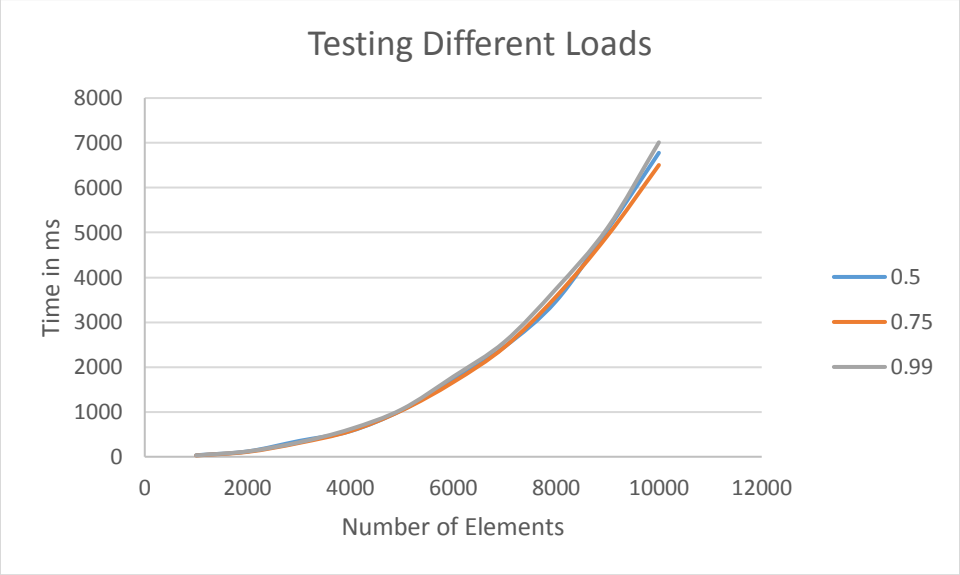
Jordan Gardner

Assignment 10

U0566259

The load factor for the quadratic probing strategy measures how full the array is. When the array reaches a certain percentage of full then it rehashes. For the separate chaining strategy the program rehashes when the linked lists at each hash become too large. The best load factor I found was 0.75 because this had the best times associated with the add function without allocating a very large amount of storage.

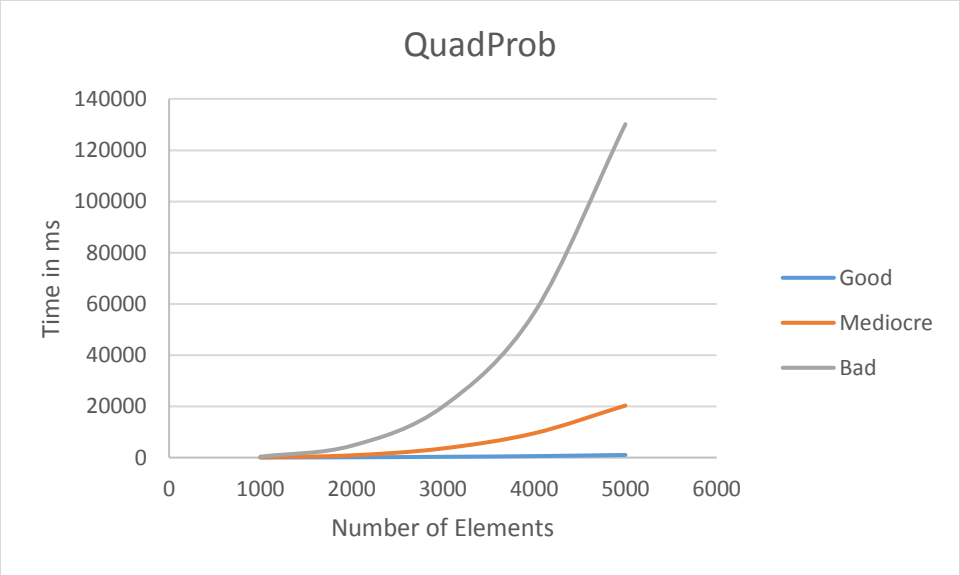


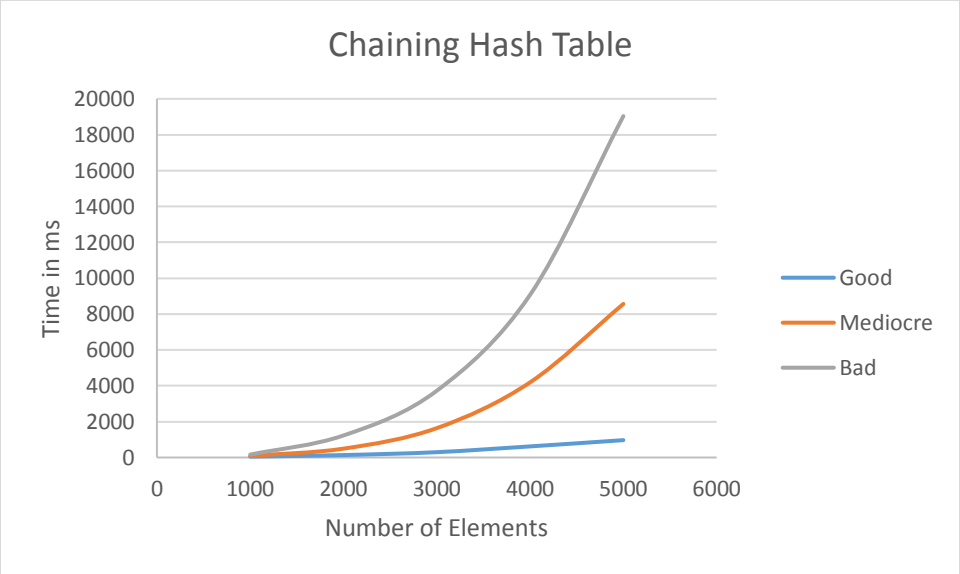
The bad hash functor I used was just the char At method. This I considered bad because there are many strings that have the same beginning character therefore causing many collisions.

In this functor I decided to use the length method in combination with the char At method used in the bad functor. This time I wanted reverse the characters in the string, cut some out, and then return this as a hash. The word read backwards, plus missing characters, is probably a decent hash if not a great one.

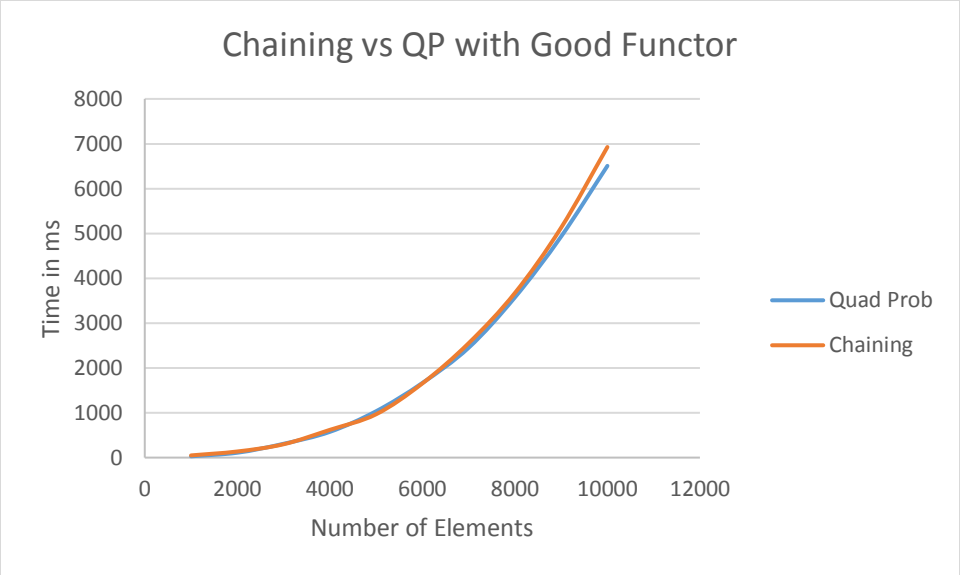
And lastly I used a character array to create a hash that continues to add a hash to hash + each of the characters in the string. This I consider the best because it takes in every character and hashes it then adds it to the already hashed characters. It's kind of hard to explain but I think this is my best functor.

What I did here is ran each functor for the two different hash table creating strategies. I added strings to a list then from that list I added them into a hash table using one of the two strategies. I timed how long it took to add a certain number of elements to the table then increased that number, and timed again. I used the best load threshold to test each of my functors. As you can see the graphs properly demonstrate that the best functor hashed and added the same number of elements in much shorter amount of time than the other two functors. This was the case in each method, the Quad Prob method and the Chaining method. I did not create a graph that shows collisions but from this data shown the collisions were much much higher on the bad functor in comparison to the good functor.





This experiment was conducted in the same way as previously mentioned however, the comparison is between the two methods used to created hash tables. I used the Good Functor and the best threshold for each of these methods and tested them up to 10,000 elements added. Both of them performed very closely to one another but it seems that Quad Prob may perform better with my functor for larger data sizes. The number of collisions would be relatively the same for the two methods due to the fact they are both using the same threshold and the same functor. Most of the time I would assume very little collisions and optimal run time unless there was a problem with my functor.



Yes the problem size and functor are directly proportional to the length of the string. This tells me that the longer the string the more intricate hash and therefore less collisions. The expected Big-O notations for hash tables are in average case C and in worse case its N. When I was conducting this experiment I intended to see straight lines that occurred at different time levels indicating a big O of C but different time taken depending on the functor. However when I conducted the experiments adding elements to the hash tables then increasing elements it seemed that it was not the case. The reason it may look like n^2 is because I'm testing the number of elements rather than time it takes to add each individual item. For example each item added may be c, so does that mean that if I were to iterate over 10,000 adding functions would the big O still be c. I think that for each individual add it would be C notation because that is how the hash function is designed to work, specific locations, and easy access. However, for multiple iterations of this process will take additional time hence why you see my graphs following a curve.

As you can see what I demonstrated in problem 1. It does affect the run time and performance of the program, however not as much as I expected. It does seem that there is a proper balance between size of object and how many locations you want to create vs when the program will need to rehash. After researching this I think javas Hash function rehashes around 71% full. This is very similar to what I found in problem 1.

Remove would be ok to implement as long as you are using the same functor to add. In this case you could easily find the hash and remove the item at that hash. First you would have to make a comparison and check the item at the initial hash to make sure it is what you want to remove. Otherwise you would do what we learned in class and make that hash with a Boolean indicator rather than removing that data. Then using whatever technique you used to add data into the table you would use the same technique to remove by stepping through until you found the object you need to remove.

I'm sure it is possible, however, I feel that my functors would have to change as well as some of my implementation. Unfortunately, you would have to develop a functor to deal with all the data types primitives and strings and that would be difficult. Furthermore if you had a data sample that included a variety of types it would be even more difficult to hash everything the same way. For example if you needed to hash an int then a character then a string would access to these be possible if they were hashed differently? I think the problem lies in the functor, may take some more thought.

Approximately 10 hours