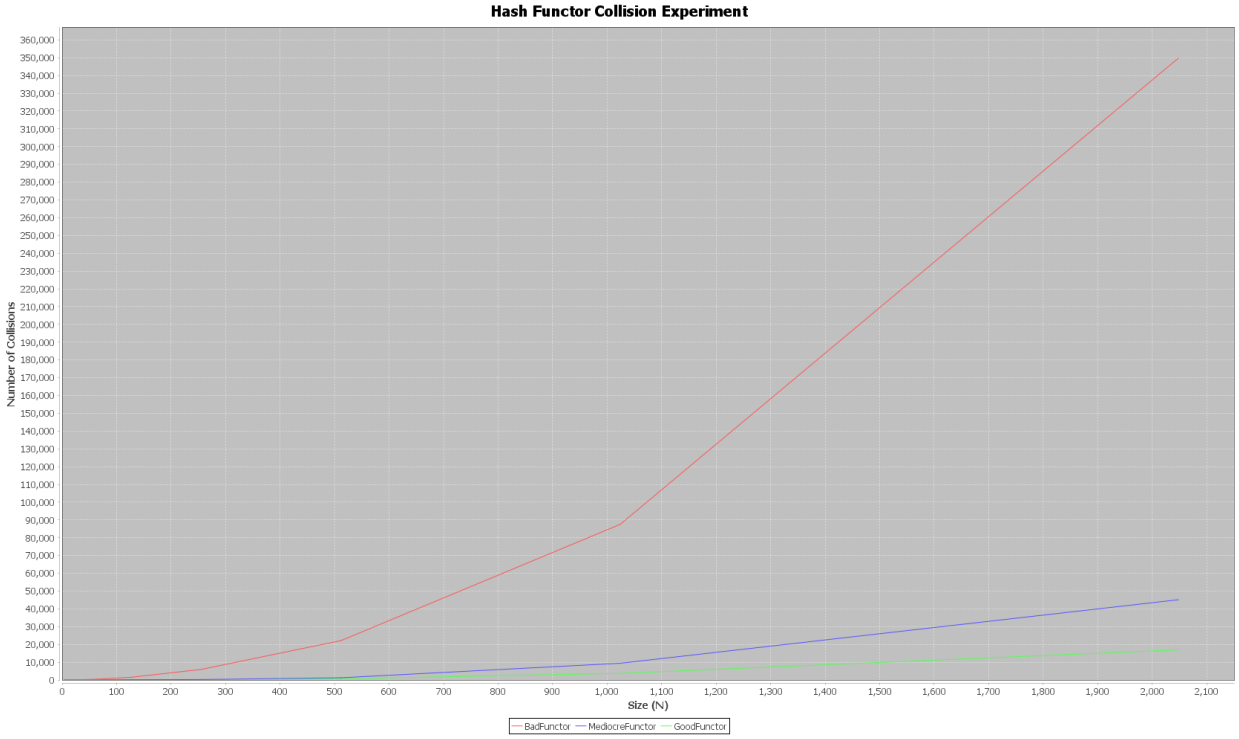
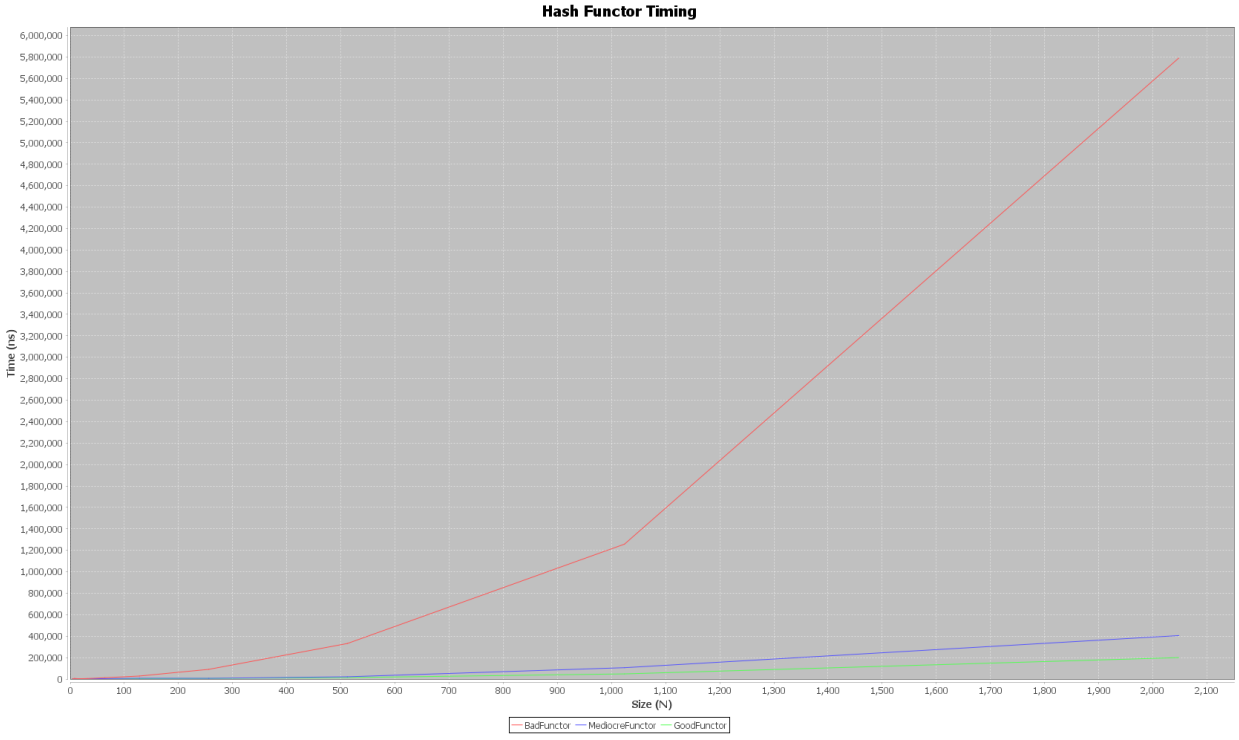
Braeden Barwick

1. The load factor for a quadratic probing table is just the percentage of space filled in the array, and a good factor is about .5, or 50%. The load factor for a chaining table is more complex. For my implementation, I used the average length of each list in the table to be the load factor, and whenever the load factor exceeded 10 (10 items per index in the table) I would rehash the table.
2. For the badFunctor I simply returned the length of the string. I expected this to make a lot of collisions because I limited the size of my strings to be between 3 and 10 characters, which means only 8 possible hashes exist.
3. For the mediocreFunctor, I added up the values of each character in the string. I knew this would be a LOT better than the bad functor because integer values for characters vary wildly, but I knew it would result in a lot of clumping because strings of similar lengths would have very closely grouped hashes.
4. For the goodFunctor, I took the sum of each character multiplied by 7 to the power of the index of the current character. By multiplying the character value by a prime number it results it more diversified hashes, and the exponent gives each number a much bigger variance, resulting in less clumping.

5.



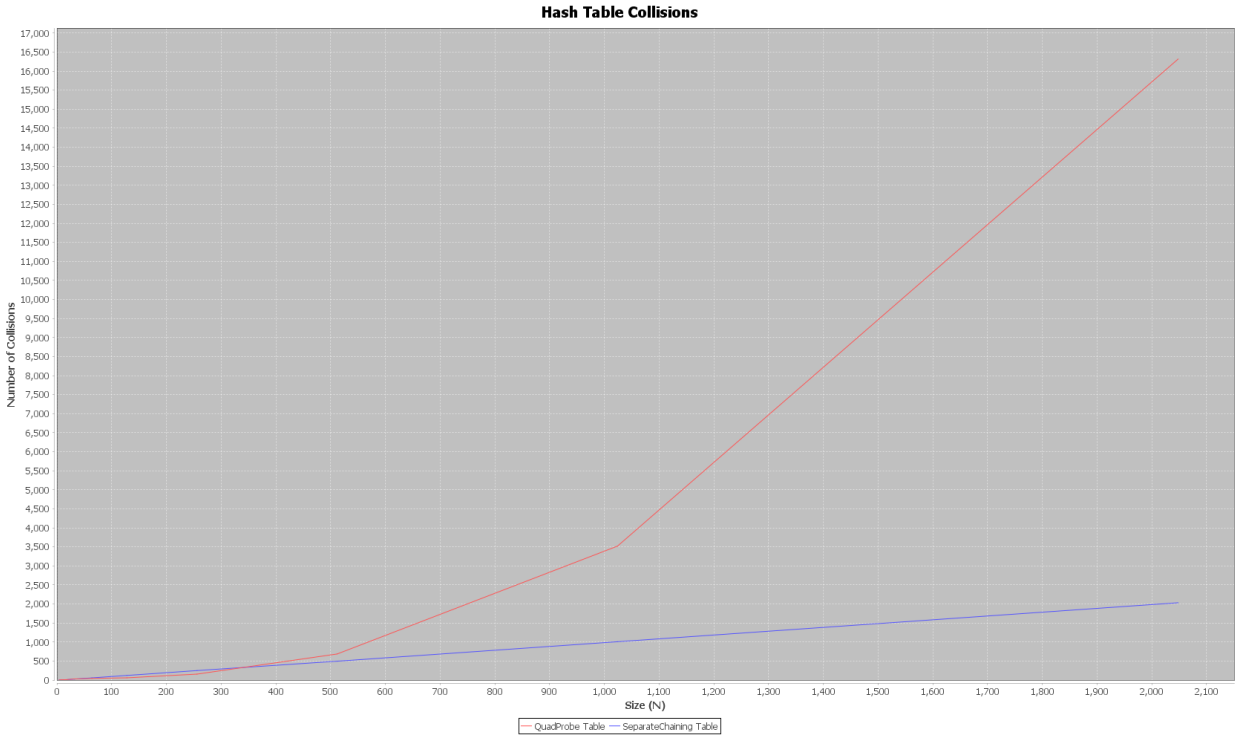


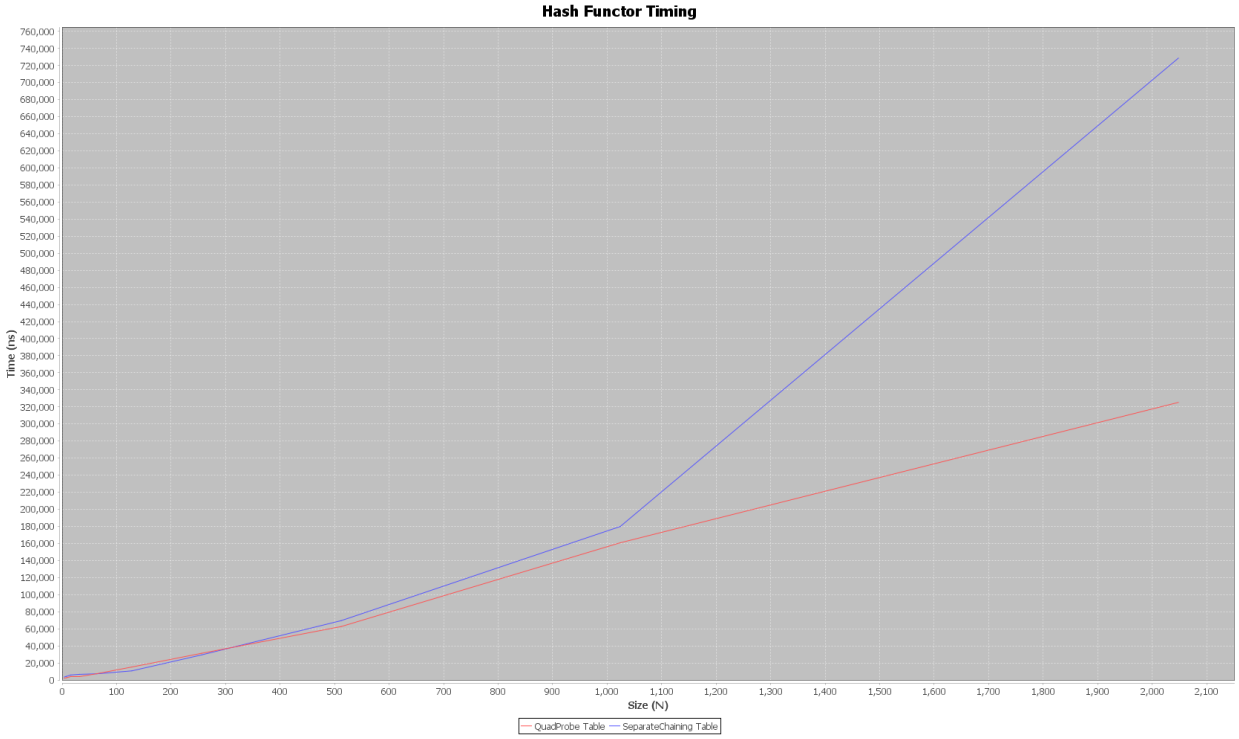
The images didn't turn out great, but the red lines represent the bad functors, the mediocre functors are the blue lines, and the good functors are the green lines. For the collision experiment, I kept track of the number of collisions while adding random string lists to the table of increasing size. As expected, the bad functor got increasingly worse as the number of items increased, because strings would only hsah to the first 8 indices of the array. The mediocre functor performed very well, and I'm not sure how I could

produce a more evenly distributed result, but it didn't do as well as my good functor.

For the timing, I recorded the run time required to add each list of increasing size to a table using the bad, mediocre, and good functors respectively. We can see that the time required increases directly proportional to the number of collisions incurred.

6.





For the hash table experiments, the red line represents the Quadradic Probing table, and

the blue line represents the Separate Chaining table. For the collision experiment, it can be hard to qualify what counts as a collision. For a quad probe table a collision is every time an item searches over an already full array, but for a chaining table a collision is only counted if the item is added to a list that already exists. This makes the separate chaining table seem much more efficient compared to the quad probe table, but the timing experiment tells a much different story. Because of the increasing lengths of the lists in a chaining table, the actual run time increases much faster than the number of collisions, while in a quad probe table, the run time increases slower, proportionally, to the number of collisions incurred. So, for a data structure that you want to add many things to, a separate chaining table can be very good for that, as there is a maximum of 1 collision on each insertion. However, for searching a table, quadratic probing produces faster results.

7. The cost of my bad functor is O(c), because it just returns the length of the string, one operation. The mediocre and good functors are O(N) because they step through each character in the string and perform one operation on it. My functors all produced results that I expected although I struggled to find a mediocre functor that produced results that were close to halfway between the good and bad functors. It seemed like you either had a bad functor, or a functor that was almost good, or a good functor.
8. The load factor has a big impact on the performance of the hash table, because it controls when the table must rehash, a linear cost function. The fewer times the table must execute that O(N) function, the faster it can go.
9. Removing items from a separate chaining table would be fairly simple. You would hash the item, search for it in the corresponding array list, and remove it. That means that the timing of the remove function would rely on the quality of your functor and load factor. For quadratic probing it would be a bit more complicated. I would replace the strings in the array list with objects that contain their hash value as well as their string value. That way when something is deleted, I would just set that objects string value to null in the array. So when you search for an item, you can still pass over an empty object with the same hash value, and you can still overwrite any null items with the same hash value.
10. It would be possible to make the hash tables generic, but it would have to rely on the assumption that whatever item being passed into it has a hash function. All java objects have a hash function, so you would be safe in implementing a hash table of type Object. Instead of using a string hash functor, you would have to use a functor given with the object type. Otherwise implementation would remain the same.
11. I spent about 5 hours on this assignment.