Brayden Carlson
u0959889

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

In a quadratic probing hash table, the load factor is the number of items divided by the total number of available slots in the array. It represents how full the table is from 0 to 1. In separate chaining, the load factor is the average size of the linked lists. For both kinds of hash tables, the lower the load factor, the better the performance.

2. 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).

My BadHashFunctor simply uses the string's character length for the hash. I expect this to be bad, because many strings inputted will probably have the same length.

3. 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).

My MediocreHashFunctor uses the sum of the string's characters ASCII values for the hash. For example the hash value of "cat" would be (cat) = (99 + 97 + 116) = 312. I expect this to be much, much better than the BadHashFunction due to there being many more potential hashes.

4. 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).

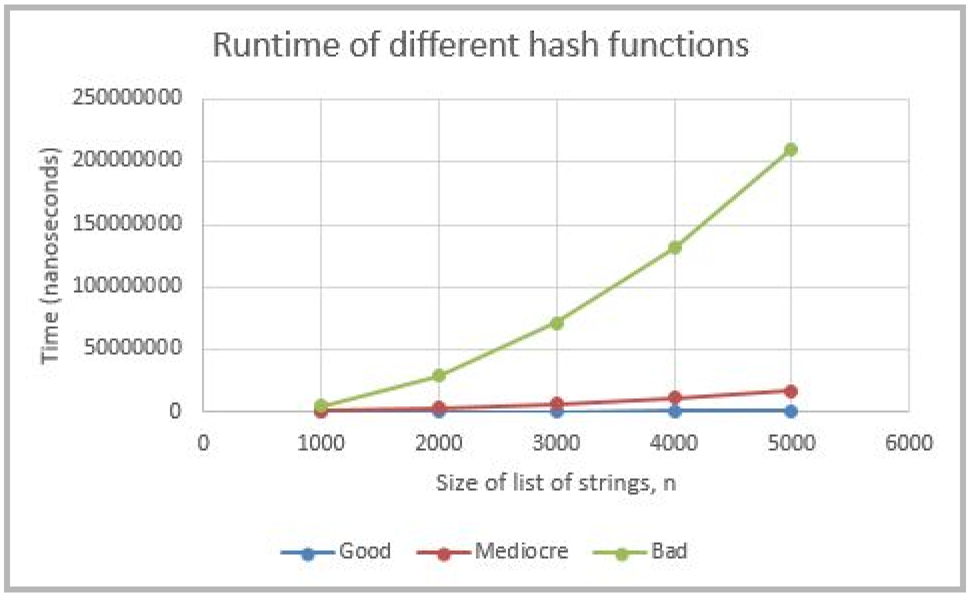
My GoodHashFunctor finds the string's hash by using Java's algorithm to find string hashes. I don't use Java's method directly, however, I instead implemented the algorithm myself according to this:

$$h(s) = \sum_{i=0}^{n-1} s[i] \cdot 31^{n-1-i}$$

I expect this to do better than the other two functors because it's the actual hash function Java uses and probably has had a lot of thought put into it.

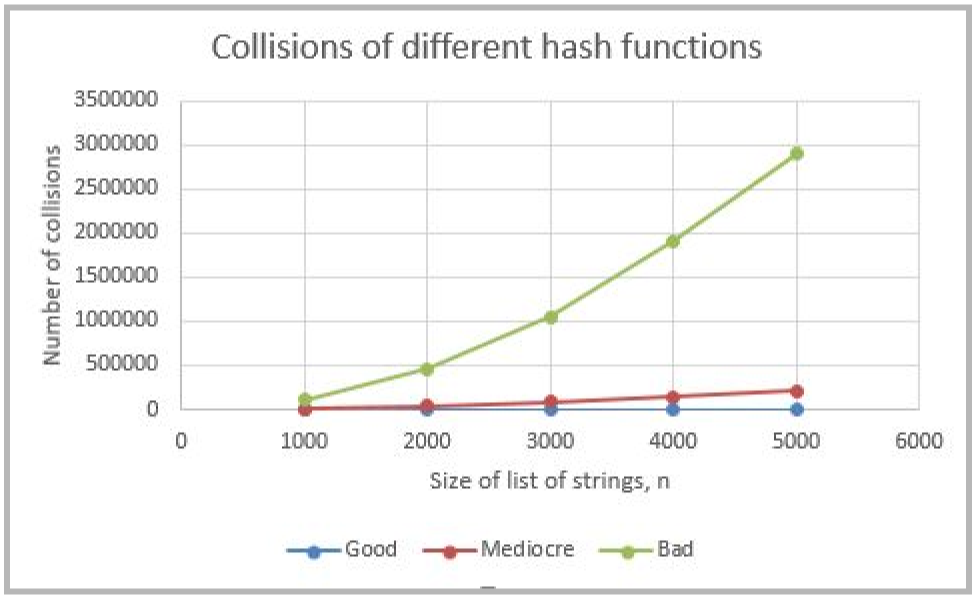
5. Design and conduct an experiment to assess the quality and efficiency of each of your three hash functions. Carefully describe your experiment, so that anyone reading this document could replicate your results. Plot the results of your experiment.

Experiment 1: Use a hash table that uses quadratic probing. Time the time it takes for the hash table to add a list of random strings of size n using each of the three hash functions. Have the hash tables start at an initial capacity of 1000. Have the length n go from 1000 to 5000 and increment by 1000. Plot the results.



As seen, the three functions performed exactly as expected in terms of runtimes, with the bad hash function performing significantly worse than the other two.

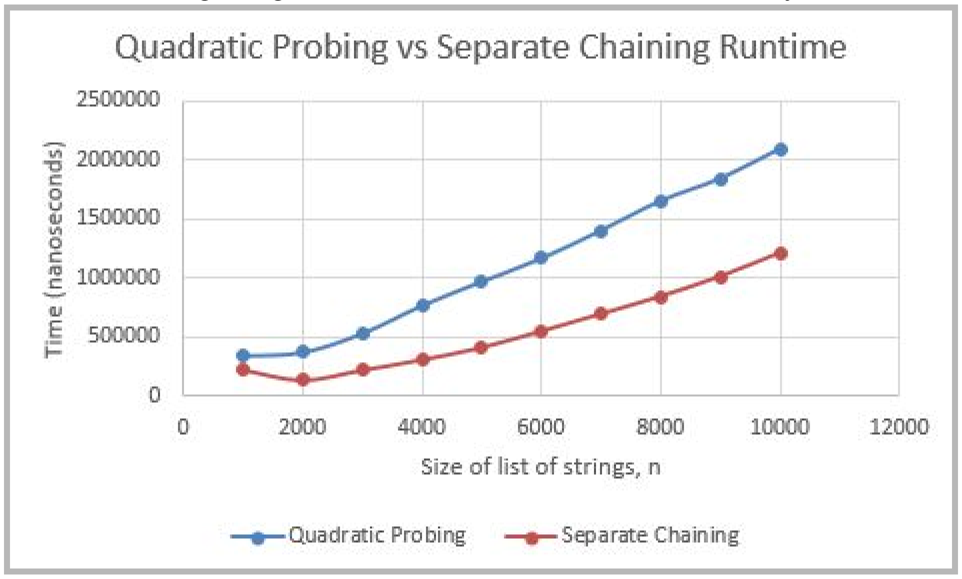
Experiment 2: Use a hash table that uses quadratic probing. Add a list of random strings of size n to the hash table using each of the three hash functions, and find the number of collisions that occur when doing so for each one. Have the hash tables start at an initial capacity of 1000. Have the length n go from 1000 to 5000 and increment by 1000. Plot the results.



This is also as expected. This graph almost exactly matches up with the graph that measures each hash function’s runtime. This shows that the number of collisions is a good indicator of the runtime of the hash function when used in a hash table.

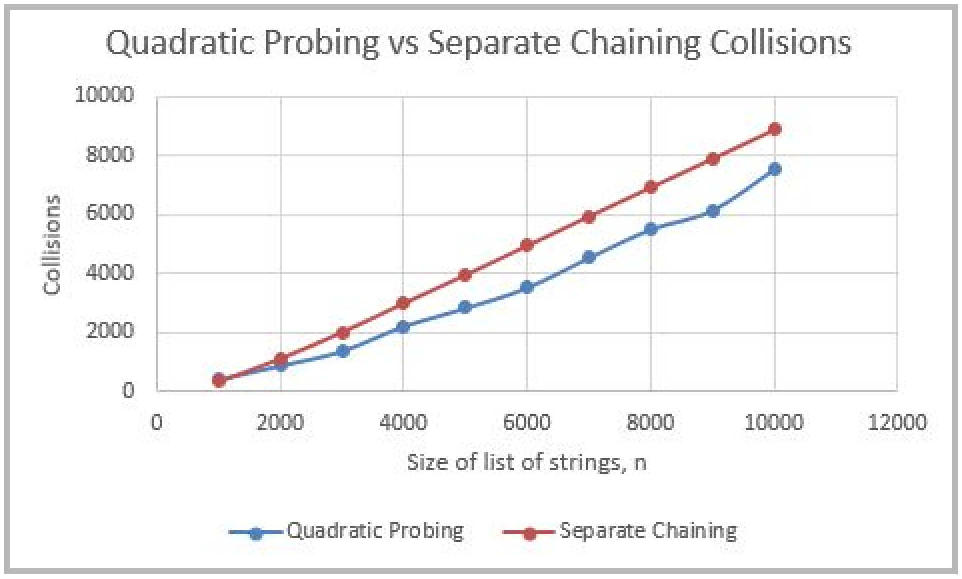
6. Design and conduct an experiment to assess the quality and efficiency of each of your two hash tables. Carefully describe your experiment, so that anyone reading this document could replicate your results. Plot the results of your experiment.

Experiment 1: Find the runtime of adding a list of random strings of size n to both a hash table that uses quadratic probing and a hash table that uses separate chaining. Use the GoodHashFunctor for both. Have the hash tables start at an initial capacity of 1000. Have length n go from 1000 to 10000 and increment by 1000. Plot the results.



As seen here, the separate chaining ran quite a bit faster than the quadratic probing. I believe this is most likely because I didn't implement a resizing/rehash in my separate chaining hash table. The quadratic probing hash table has to re-add all of its elements every time the load factor goes above 0.5 causing the slower performance. Another reason for the difference in runtime is in the way linked lists work. Even if collisions occur in the separate chaining hash table, the complexity of adding an element to the end of a linked list is always O(c), and therefore collisions have no effect when adding items to a separate chaining hash table, making it much faster.

Experiment 2: Find the number of collisions that occur between a hash table that uses quadratic probing and one that uses separate chaining when adding a list of random strings of size n. Use the GoodHashFunctor for both. Have the hash tables start at an initial capacity of 1000. Have length n go from 1000 to 10000 and increment by 1000. Plot the results.



In terms of collisions, separate chaining appears to have the most by a bit. This is again probably because I didn't implement my separate chaining hash table to resize. The more capacity a hash table has, the fewer collisions it's going to have, and because the quadratic probing hash table resizes as it goes, it has the least collisions. You can actually see they start off almost exactly the same in terms of collisions, but then I assume based on my theory that the quadratic probing table starts rehashing and starts having less collisions than the separate chaining table.

7. 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.)

Bad - O(c). This hash function only took the string's length.
Mediocre - O(n). This hash function had to iterate through each character in the string so it could add it to the summation.
Good - O(n). This hash function also had to iterate through each character in the string.

Overall, all three of the functions performed exactly as I expected. The bad function performed the poorest by far, the good function performed the best, and the mediocre fell between the two, but much closer to the good function in terms of performance. Even though the bad function has the fastest complexity in terms of big-o, it is much more about the number of collisions each function causes that determines which is truly faster.

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

The load factor determines the average number of spaces in the array that must be examined when adding/finding an element in a hash table. The smaller the load factor, the more empty the array currently is, and thus, less collisions and a faster runtime overall. If the load factor is getting too big, there will be more collisions and a slower performance. This is why rehashing and resizing is important in hash tables.

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

For the hash table using quadratic probing, you would have to use "lazy deletion." When an element is requested to be deleted, instead of being removed from the array, it is simply flagged as deleted. It won't actually be deleted until the hash table is rehashed.

A remove method for the separate chaining hash table would be much more simple. Just simply find the linked list it is contained within using its hash and invoke linkedlist's remove method.

10. 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 it would be possible. In the hash table class you would only have to make the methods and class generic for it to work (changed the method headers and variable types to generic). However, you would have to come up with new hash functions because the one's currently used would only work for strings.

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

9 hours.