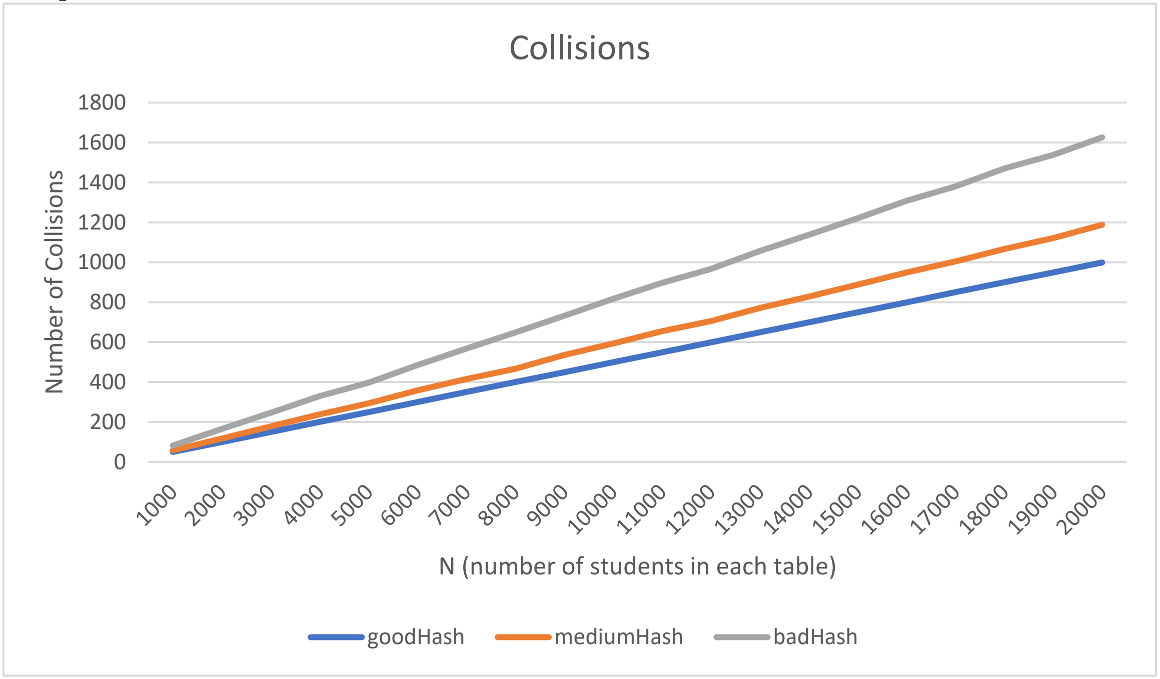
Assignment 9 Analysis

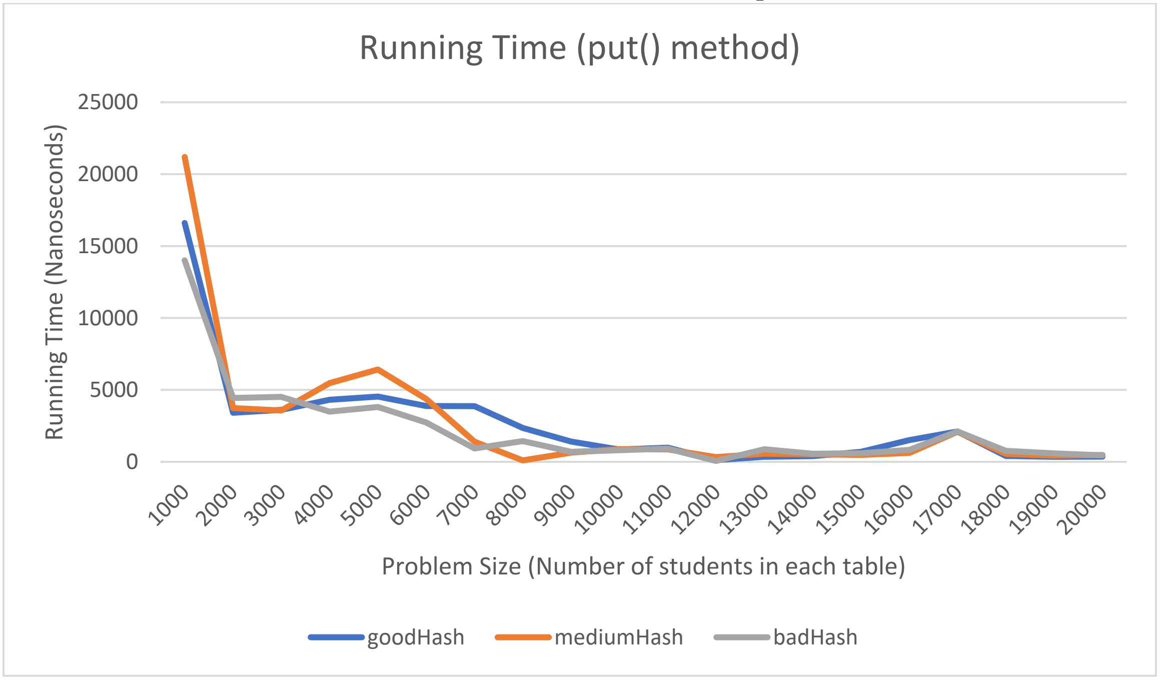
1. We chose separate chaining for our collision resolving strategy. We chose this because the implementation for separate chaining was far easier and more straightforward than quadratic probing. Quadratic probing requires the table to be a prime number and that the load factor is less than 0.5, which means we would have to resize the underlying array a lot more. Additionally, it doesn't seem to be as effective in solving collisions. While it does solve collisions better than linear probing, it still has an issue with secondary clustering. The linked list structure, however, instead treats each array index as a 'bucket' that we can add up to n items to. The allowed load factor is much larger, which means we do not have to resize as often. This takes less time and means there are far less wasted spaces in our underlying array. Additionally, these buckets are created using a linked list which means they can grow dynamically, we don't have to create extra spaces that are wasted—we can add more links only as needed. Even though we do have to keep track of each head node and would need to traverse through the linked list to find the desired value, we still felt it was far easier to implement than quadratic probing. We didn't have to worry about putting a value in a different index than it originally would have gone in, which means searching was far easier. We could get the hash code of the desired item and search that linked list only, where this isn't really possible in quadratic probing due to the possibility of it being stored in a different index. Additionally, it was a little easier for my partner and I to visualize the separate chaining hashTable. The actual code required was pretty similar to what we've already practiced in this class, while the quadratic probing had some elements that we were uncertain about (finding the next index for handling collisions, getting the next prime after resizing, etc.).
2. The first step of my experiment was to add a variable to my hashTable class that could actually keep track of the number of collisions that occur while using my hashTable. I created a private instance variable for this, and in my put method, I set the variable to increment every time it went through an entry in the linked list held at the index and it didn't reset that entry. If the linked list was empty, it would have skipped over that section and there would be no collisions, so there were no possibilities of incorrectly adding a collision if it was the first value. Next, I created two helper methods. One method was called createName() and would create a set of random letters of a random length between one and 10, (the number generated by a random number generator). The second helper method would generate a random uid from 1000000 to 10000000 so each student will have a randomly generated 7 digit uid. From there, I created three hash tables—goodHash, mediumHash, and badHash. I would generate a first name, a last name, and a uid. I would insert each new student into all three hash tables to ensure consistency, alongside a randomly generated gpa for the value of the inserted student key. After I had added N (an integer from 1000 to 20000 that I looped through) students to the hash tables, I printed out the collision count divided by N for each map to compare them to each other. This is a good experiment because we are using the same students for every table, so the number of collisions is an accurate representation of how efficient the hashCode function is for each student. While there will obviously be a lot of collisions due to the fact that we create each table with separate chaining, it is still a good way to

compare between different hashCode functions.



As it is clear from the graph above, the goodHash is superior to both the medium and the bad hashes, as it consistently has the smallest number of collisions.

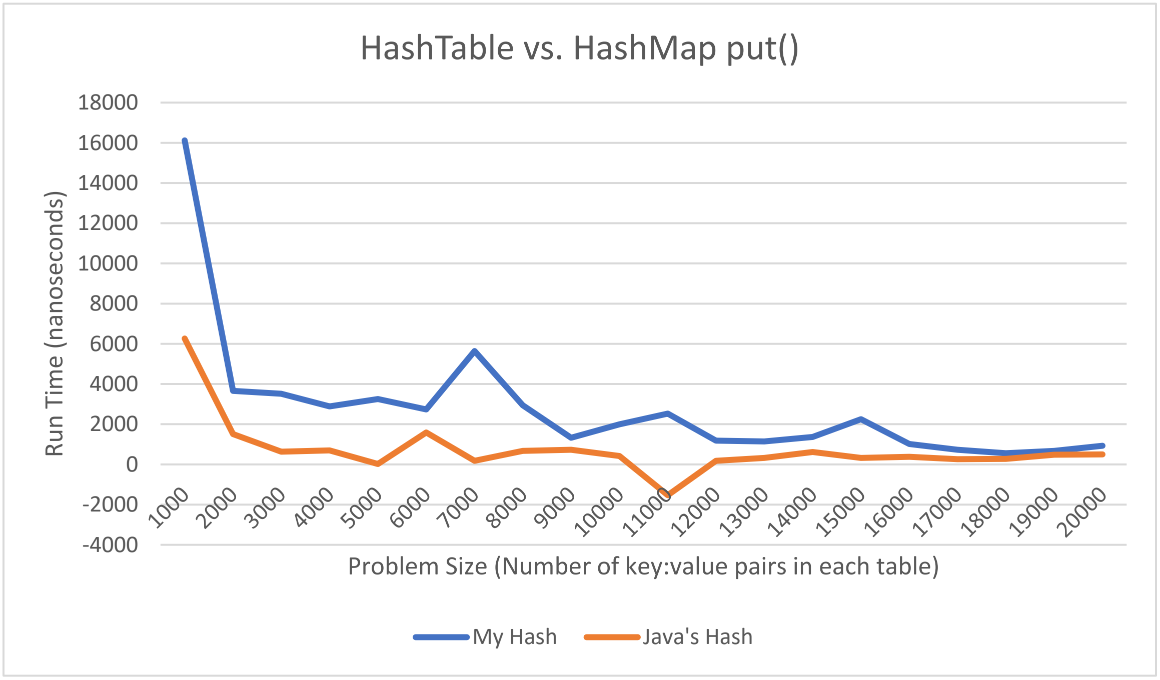
The next experiment I conducted was testing the running time of each hash function. To do this, I created the exact same conditions as above (three hashTables with identical students), and then for each student in the map I tested how quickly it would put a new student in the table. I started with the goodHash table first, then I timed the mediumHash table, then the badHash table last, to ensure each table had their own timer and the timing wasn't the time it took for all three of the tables to call the put method.



The running times of each of these three methods are fairly similar, but it is clear that sometimes, the goodHash method takes a little longer to run than the medium and bad hashes. I believe this is because our goodHash hashFunction() method must take the time to add up each number in the uid, where the medium and bad hash just call length for the first name and last name, and the first name respectively. The goodHash function takes a little longer but produces a better hashTable with less collisions.

3. The goodHash function has a fractionally larger cost than the medium and bad hashes. Its hash function uses a for loop that runs 7 times to get each digit out of the uid. However, in terms of actual complexity, this is so negligible that its complexity is still O(1). The bad and medium hashes are also O(1) and are a little smaller than the good hash function, because both functions just call length for a given string and return that, which has a very small cost. The bad hash just returns the length of the student's last name, while the medium hash returns the sum of the length of both the first and the last name. Each operation is very cost efficient, but not as effective in creating unique hash codes to avoid collisions. Each function performed as I expected it to—the goodHash function was much better than both the medium and the bad hashes, and the medium hash was still better than the bad hash. There were no inconsistencies shown in my experiments, they all performed exactly as expected.
4. The load factor impacts the performance of my hash table in two ways. First, we must ensure we calculate the load factor after each call to put—if the load factor gets above a certain value, then we must expand the underlying array and rehash the entire table. Obviously, this will take a lot more time than any other call to put(), so that is one way the load factor impacts the performance. The second way is that the higher the load factor is, the longer the linked lists contained in each bucket. This impacts the put() and remove() functions because if the linked lists are longer, we need to take more time to search through each value.
5. To compare my hash table to Java's hash table, I created a variable of my hash table and one of Java's hash table. I then called the createName() function as described above to generate a random string of a random length, and timed how long it took to insert that string into my hash table, then separately timed how long it took to insert that string into Java's hash table. The values associated with these keys were random values and were negligible to this experiment. Strings already have a hashCode() function which means this experiment solely timed the efficiency of my put() function against Java's put() function, which is a good indicator of how efficiently values are spread across the hash table, the cost of dealing with collisions, the cost of checking the load factor, expanding

the table, etc.



It is clear from the plot above that Java’s HashMap is far more effective and much faster than my HashTable class. Java’s hash map was so fast the timer couldn’t get an accurate reading, resulting in negative times for a few of the problem sizes.