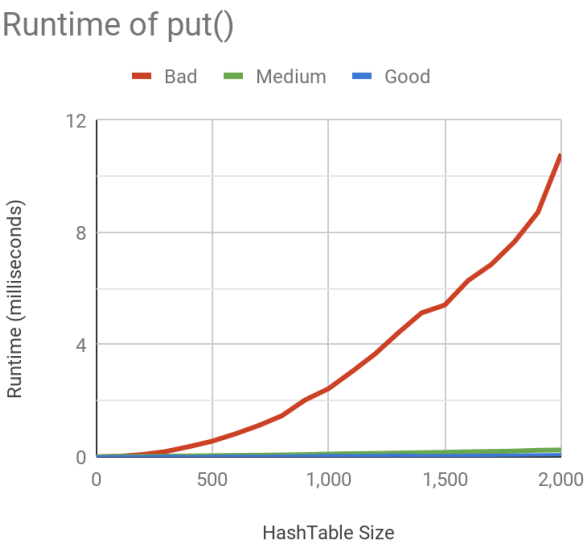
Assignment 9: Hash Tables

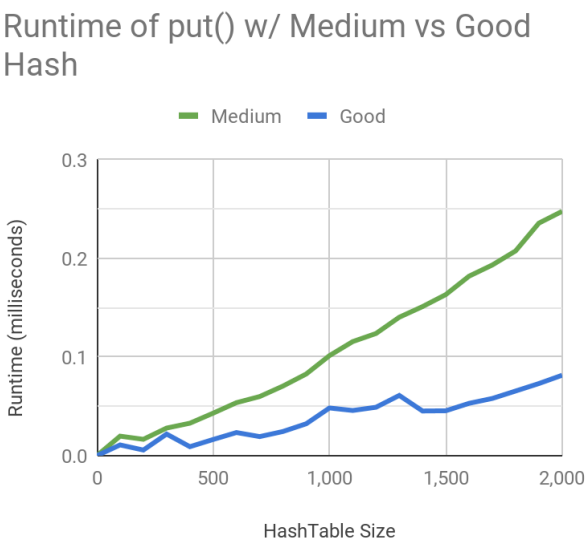
Analysis Document by Qianlang Chen (u1172983)

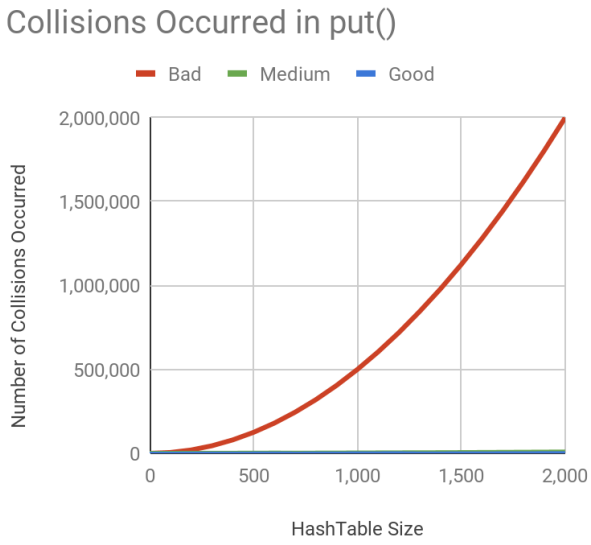
As in the previous 3 programming assignments, I gladly had Kevin Song as my programming partner in this one again. Kevin did a great job at understanding the problem and coming up with proper solutions, as well as following my commands and giving me ideas. I plan to work with him in the next assignment, along with in the final project if it is also going to be a group project.

Facing the problem given in the programming assignment, namely which strategy should be used to implement and store the data in our *HashTable*, we discussed and chose **Separate Chaining**, because of its two significant advantages. First of all, since the collisions which happen when data with the same hashcode or chain index (the index of the linked-list in the backing array) are stored using linked-list, **the collisions do not affect other chains** along with the data stored in them. Secondly, because of the mechanism of linked-lists, **it does not waste much space** compared to using Linear- or Quadratic-Probing, where much space is left out in the backing array as compensation to performance.

To further understand how the quality of the hash-codes generated by the keys inserted affects the performance of our *HashTable*, we constructed three different classes that have the same structure — *StudentBadHash*, *StudentMediumHash*, and *StudentGoodHash* — each of them generates different quality of hash-codes. The *StudentBadHash* class uses the length of the name of the student as the hash-code. *StudentMediumHash* uses the sum of the ASCII values of the name as the hash-code. Finally, *StudentGoodHash* gives each position of the characters in the string a weight before summing up the ASCII values, with the UID of the student added to the sum. We timed and counted the number of collisions occurred in the *put()* method with different numbers of keys, and here are our results:







The graphs show that the performance of our *HashTable* very much depends on the quality of the hash-codes of the keys. When the hash-codes are so bad that too many collisions occur, our *HashTable* performs no much different than a linked-list, since it cannot take full advantage of different hash-codes to quickly locate the key that it is looking up. In our experiment, particularly, we generated all equal-length strings as the students' first and last

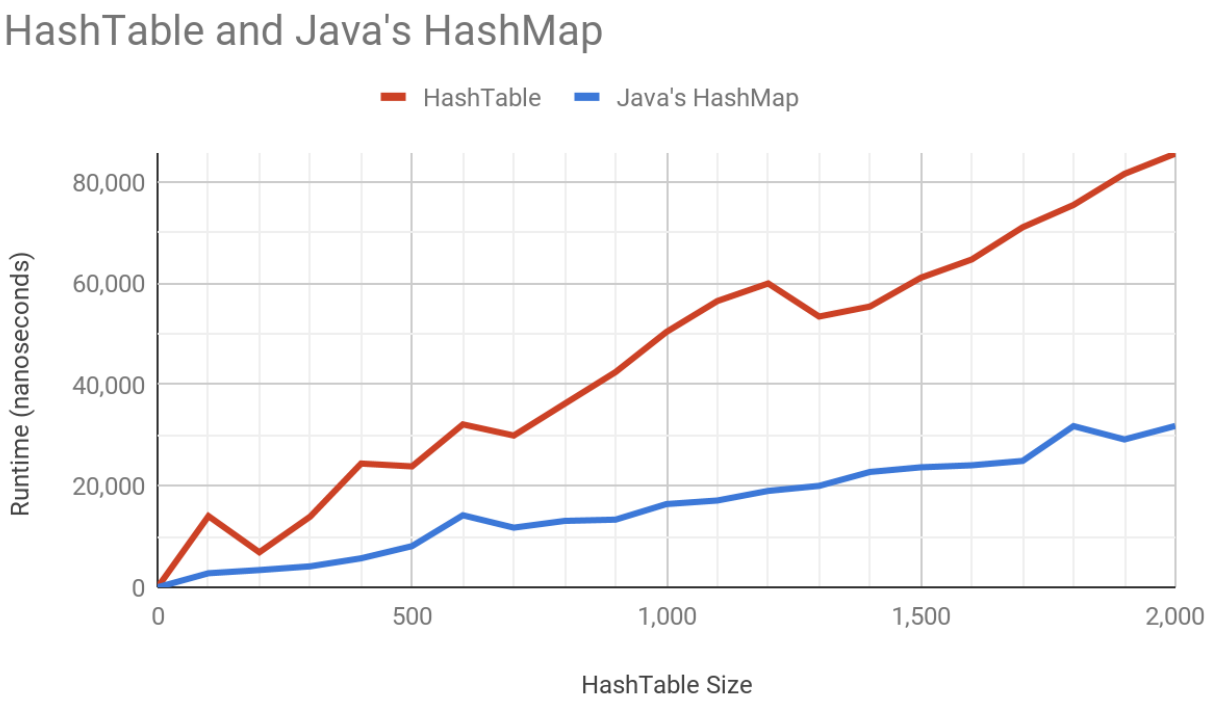
names, and our *StudentBadHash* class simply takes the sum of the lengths of the strings as the hash-code, which makes the performance the worst-case-possible, where every key has the same hash-code, making all keys on one single chain. Runtime-wise, since we timed the *put()* method for a set of N keys, the actual runtimes of the *put()* method using different student classes should be the runtime we see from the graphs divided by N, which then gives a **constant runtime for medium and good classes, and a linear runtime for the bad class**. One thing that we have to admit is that it does have some cost to generate a hash-code for each key — the bad class generates in constant time, and the medium and good classes generate in linear time relative to the length of the student's name. However, the number of collisions they bring match our predictions, according to our reasoning above.

One other critical question that we asked ourselves when designing the *HashTable* class was when to expand the backing array, since the higher the load factor is, the higher the chance of getting collisions in any action to our *HashTable*, which directly affects the runtime performance. We set a threshold of 5, which made the backing array expand when the average length of the individual linked-lists (chains) reached 5, but we were not sure if this was a good choice. Therefore, we timed the *contains()* method for different size of HashTables (with String keys) using different thresholds ranged from 1 to 10. Note that in the table, N represents *HashTable* sizes, the numbers 1~10 on the top represent different threshold sizes, and the rest numbers record the runtime (in nanoseconds) of putting N String keys into the *HashTable*:

N	1	2	3	4	5	6	7	8	9	10
250	83	68	68	66	52	48	48	49	56	49
500	56	53	54	61	54	54	55	53	54	60
750	49	50	52	49	59	49	50	52	47	55
1,000	52	51	72	52	58	57	51	51	52	61
1,250	60	75	59	61	57	58	57	57	64	57
1,500	73	50	111	59	51	51	52	62	74	56
1,750	60	59	62	57	63	62	62	62	71	56
2,000	65	59	61	61	82	58	59	61	66	62

As we can see from the table, the runtime behavior is **constant for all examined thresholds, and they pretty much equal**. However, using a low threshold does mean that the backing array needs to expand more frequently, which requires more memory.

Finally, we compared the runtime performance of our *HashTable* to Java's native *HashTable* — the *HashMap* class. We timed the put() method for different numbers of String keys N, and here is the result we have got:



The graph shows that the runtime behavior is **constant for both classes** (the *put()* method is run N times, making the behavior linear, and we divided that by N to get its correct behavior). However, Java's native *HashMap* class provides better performance than our *HashTable*.