Analysis document: HashTable

Graded

Student

HARRY KIM

Total Points

21 / 35 pts

Question 1

Question 1

1 / 1 pt

✔ **– 0 pts** Correct

Question 2

Question 2

5 / 5 pts

✔ **– 0 pts** Correct

Question 3

Question 3

1 / 9 pts

Collisions

✔ **– 2 pts** Plot is missing a descriptive title, labelled axes, or labelled plot lines

---

Timing

✔ **– 1 pt** Incorrect or incomplete description of their experiment

✔ **– 1 pt** Incorrect or incomplete explanation of results

✔ **– 4 pts** No plot

Question 4

Question 4

6 / 6 pts

✔ **– 0 pts** Correct

Question 5

Question 5

1 / 5 pts

✔ **– 4 pts** Major errors, or mostly incomplete

Question 6

Question 6

7 / 9 pts

✔ **– 2 pts** Plot(s) are missing a descriptive title, labelled axes, or labelled plot lines

These methods should be constant not linear.

Question 7

Late Penalty

0 / 0 pts

✔ **– 0 pts** submitted before deadline

Questions assigned to the following page: 1, 2, and 3

Harry Kim
U1226472
11/4/2020
CS 2420

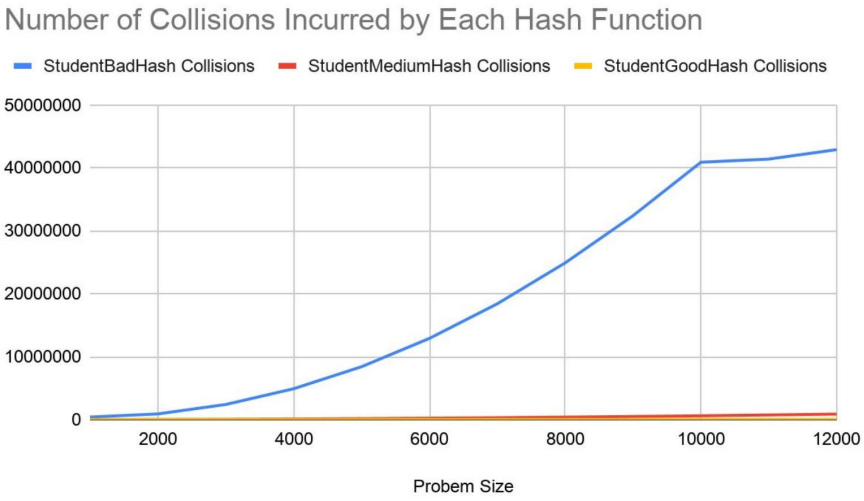
Analysis Document — Assignment 9

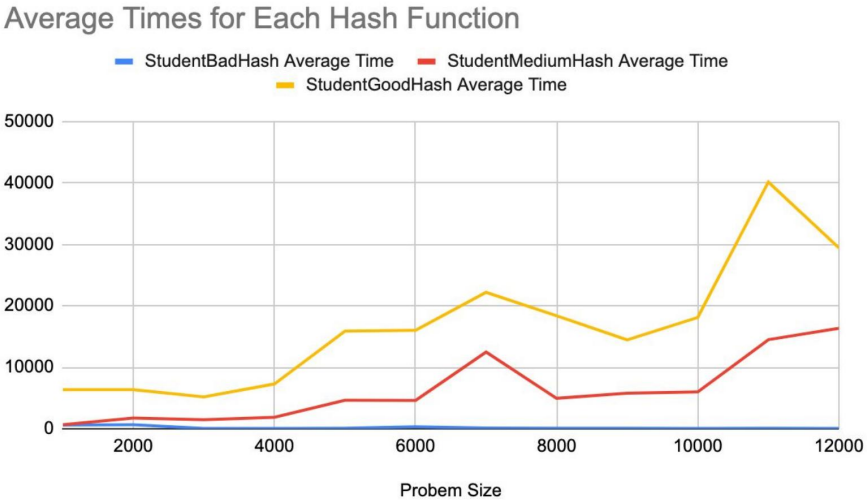
1. I did complete the programming portion of this assignment with a partner. My programming partner is Braden Morfin, and I am the one who submitted the assignment to Gradescope.

2. For this assignment, we chose to implement the separate chaining hashing strategy in order to resolve collisions. We chose to implement the separate chaining hashing strategy because we did not want to compute the next higher prime number when rehashing like we would have to do with quadratic probing. We also felt that an array of linked lists was simpler and cleaner to implement, adding the given value to a specified linked list instead of trying to find the next empty array element, especially when hashing and clamping values. We also took into consideration how the load factor is not bound by the 1.0 and represents the average list length in this hash function.

3. In order to assess the quality and efficiency of each of your three hash functions (via StudentBadHash, StudentMediumHash, StudentGoodHash), we first ran the test through a for loop which incremented the problem size to be used in each loop by sizes of 1,000 and went up to no greater than 20,000. We then created three HashTables of type student and integer (one for each student hash type as stated previously) and populated each student with a random 8 digit identification number along with random strings for their first and last names for the ArrayList elements from 0 to the problem size. We then ran timing tests to create two plots with three lines each: one that shows the number of collisions incurred by each hash function for a range of problem sizes, and one that shows the actual running time required when using each hash function for a range of problem sizes.

Question assigned to the following page: 3





As expected, the number of collisions is highest for StudentBadHash, and lowest for StudentGoodHash because of the way they execute their respective hashCode methods (explained in question 4). The average times for the hash functions for each type of student also make sense as it takes longer to compute a better hashCode method that reduces the number of collisions (each hashCode method cost and function explained in question 4).

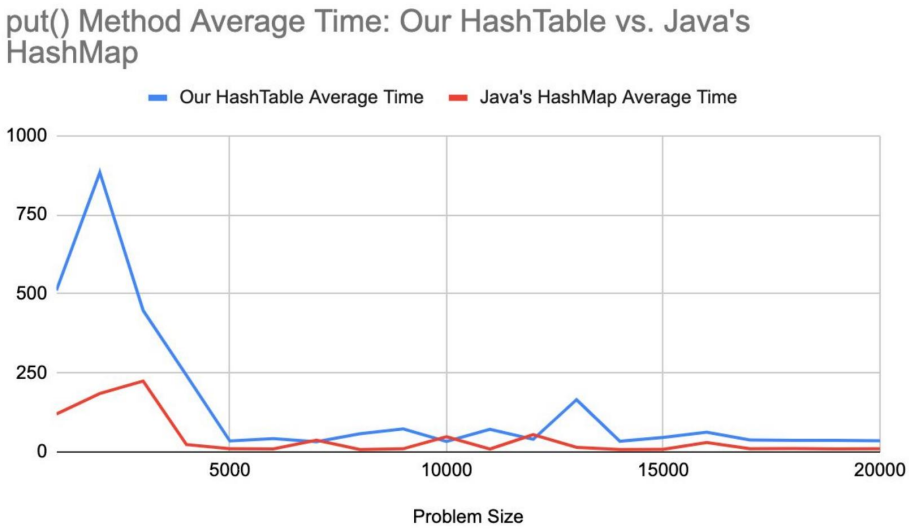
Questions assigned to the following page: 4, 5, and 6

4. The cost of the hash function for StudentBadHash should be constant, as it's only computing a sum, while the cost of the hash function for StudentMediumHash and StudentGoodHash should display linear behavior, as these methods loop through and changes the characters in the inputs to reduce the number of total collisions. As demonstrated by the first graph on problem 3, the number of collisions dramatically decreases as we perform the hash functions for each student. This makes sense as we made the hashCode method for the StudentBadHash class to only look at the sum of the length of all the inputs which led to many collisions trying to put the right value in the HashTable. The StudentMediumHash fairs better since they get the sum of the ASCII values of each character of each input. The StudentGoodHash has the best results since it multiplies each character's position of the inputs by a unique number to ensure a good distribution.

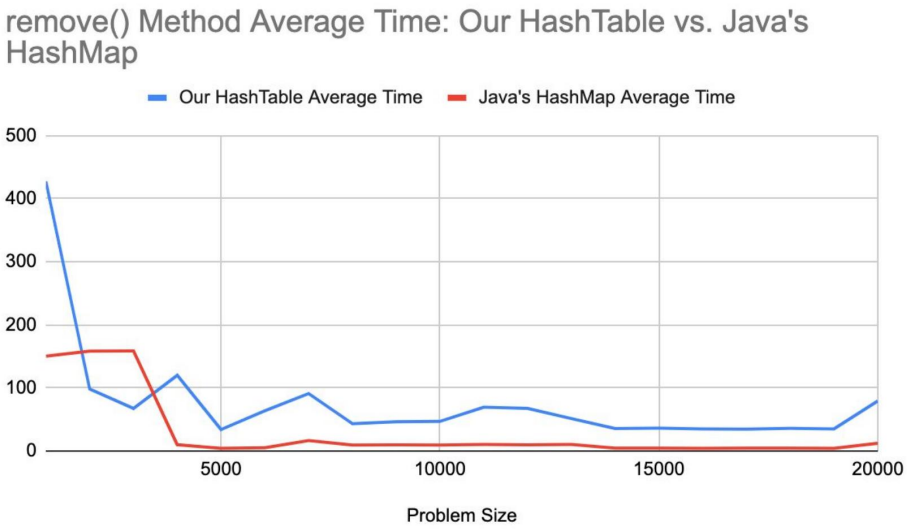
5. Because our code makes use of the separate chaining hashing strategy, the load factor λ doesn't significantly hurt the performance of our hash table. It's not important that the table size is a prime number, as it is in quadratic probing, in order to avoid more collisions. The only time the load factor λ has an effect is when we have it rehash when the number of items divided by the table size reaches a certain threshold (for us, we have it set to 5).

6. In order to compare the performance of your hash table to that of Java's HashMap, we first ran the test through a for loop which incremented the problem size to be used in each loop by sizes of 1,000 and went up to no greater than 20,000. We then created a HashTable, of type Integer for the key and type String for the value, and Java's HashMap, of type Integer for the key and type String for the value. We then populated both the HashTable and HashMap with a key numbered from 0 to the problem size and a randomly generated ten lettered string. We then ran timing tests on the *put*, *remove*, and *get* methods for each Hash to create three plots with two lines each that shows the average times of our HashTable vs. Java's HashMap for computing the *put*, *remove*, and *get* methods.

Question assigned to the following page: 6



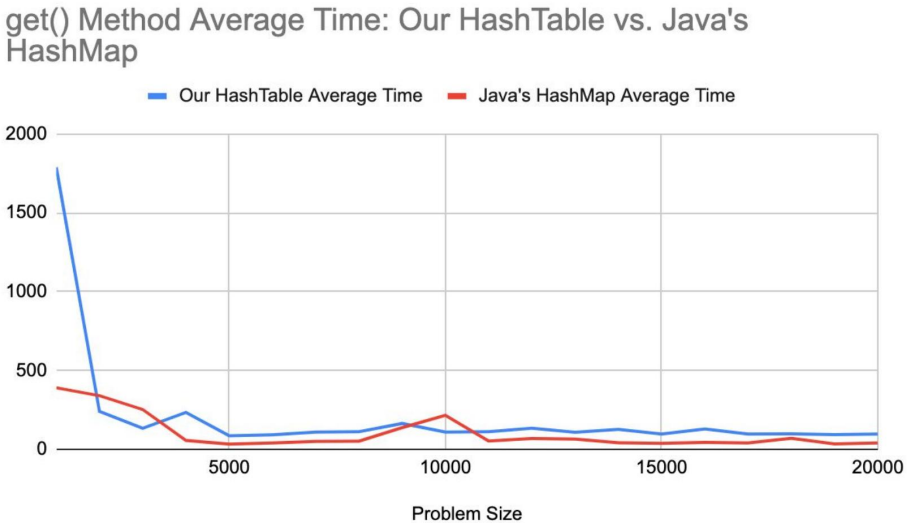
The *put* method should display linear behavior which is reflected within the graph above. This is because the load factor (number of items divided by the length of the array) is to be kept below a bound through rehashing, and the *put* method doesn't need to look through every item to find where to put an item via clamping. It would seem that Java's HashMap is more efficient which means that it's doing something more advanced and elegant than what we were able to program.



The *remove* method should also display linear behavior which is reflected within the graph above. This is because the load factor (number of items divided by the length of the array) is to be kept below a bound through rehashing, and the *remove* method

Question assigned to the following page: 6

doesn’t need to look through every item to find an item and remove it via clamping. It would seem that Java’s HashMap is more efficient which means that it’s doing something more advanced and elegant than what we were able to program.



The *get* method should also display linear behavior which is reflected within the graph above. This is because the load factor (number of items divided by the length of the array) is to be kept below a bound through rehashing, and the *get* method doesn’t need to look through every item to find an item via clamping. It would seem that Java’s HashMap is more efficient which means that it’s doing something more advanced and elegant than what we were able to program.