Adrian Bollerslev

When you are satisfied that your program is correct, write a brief analysis document. The analysis document is 30% of your Assignment 10 grade. Ensure that your analysis document addresses the following.

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

Load factor is the ratio of, the total number of elements in the hash table, over the table's array size. In general, we want to keep the load factor small in order to ensure good performance. However, we must also consider two important factors: a good hash function that distributes evenly across the entire array, and, a good collision function to prevent clustering. When the load factor becomes too large (bigger than 0.5) we must rehash. This involves creating a larger array and moving every element into the new array. Rehashing takes O(N) because we have to walk through the entire (size N) original array. If the hash table is rehashed when $\lambda = 0.5$ then there are approximately N/2 inserts. In that case, the total cost is N/2 + N. As the load factor grows larger, the hash table becomes slower. The constant runtime of a hash table assumes that the load factor is kept below some bound. As the ratio becomes higher than this bound, the time for a lookup begins to grow with each new entry. A low load factor is not beneficial either, as this means a higher proportion of unused areas in the hash table. However, there is not necessarily any reduction in search cost. This results in wasted memory.

If we have a higher load factor, we have more elements in the array and therefore a higher likelihood of collisions. Depending on whether we are using quadratic probing or separate chaining, these collisions will affect runtime in different ways. With quadratic probing a low load factor is a requirement because without it we run the risk of filling the array. We can only shift the value over so many times through multiplying it by itself. If our array size is a prime number, it gives us more potential buckets to use for collisions. However, there are still a finite number of collisions we can handle before the hash set begins functioning incorrectly with quadratic probing. With separate chaining the load factor will not actually make the program function incorrectly, but, it will drastically affect runtime. For each collision, we receive we add the value onto a linked list. This means that if the entire table was inside one bucket we would receive O(N) runtime through traversing the Linked List as oppose to the constant runtime we expect.

In summary, the load factor does not have a direct effect on the runtime of quadratic probing or separate chaining, but it does increase the chance of collisions. These collisions do have a direct effect on runtime of separate chaining because the more collisions the longer we must search through a linked list. The collisions past a certain point do not allow us to save more values with quadratic probing because we will run out of buckets in the table to save our value.

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).

The hashing function we designed for BadHashFunctor literally puts all of the hashes in the exact same bucket. This results in only collisions for every value. This defeats the purpose of a Hash Table and does certainly not provide constant runtime. A non-uniform distribution increases the number of collisions and the cost of resolving them. This results in heavy clustering and heavy performance issues. The Separate Chaining Hash must scan through the entire Linked List while the Quadratic Probing Hash must face clustering issues and possible performance issues if rehashing does not occur.

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).

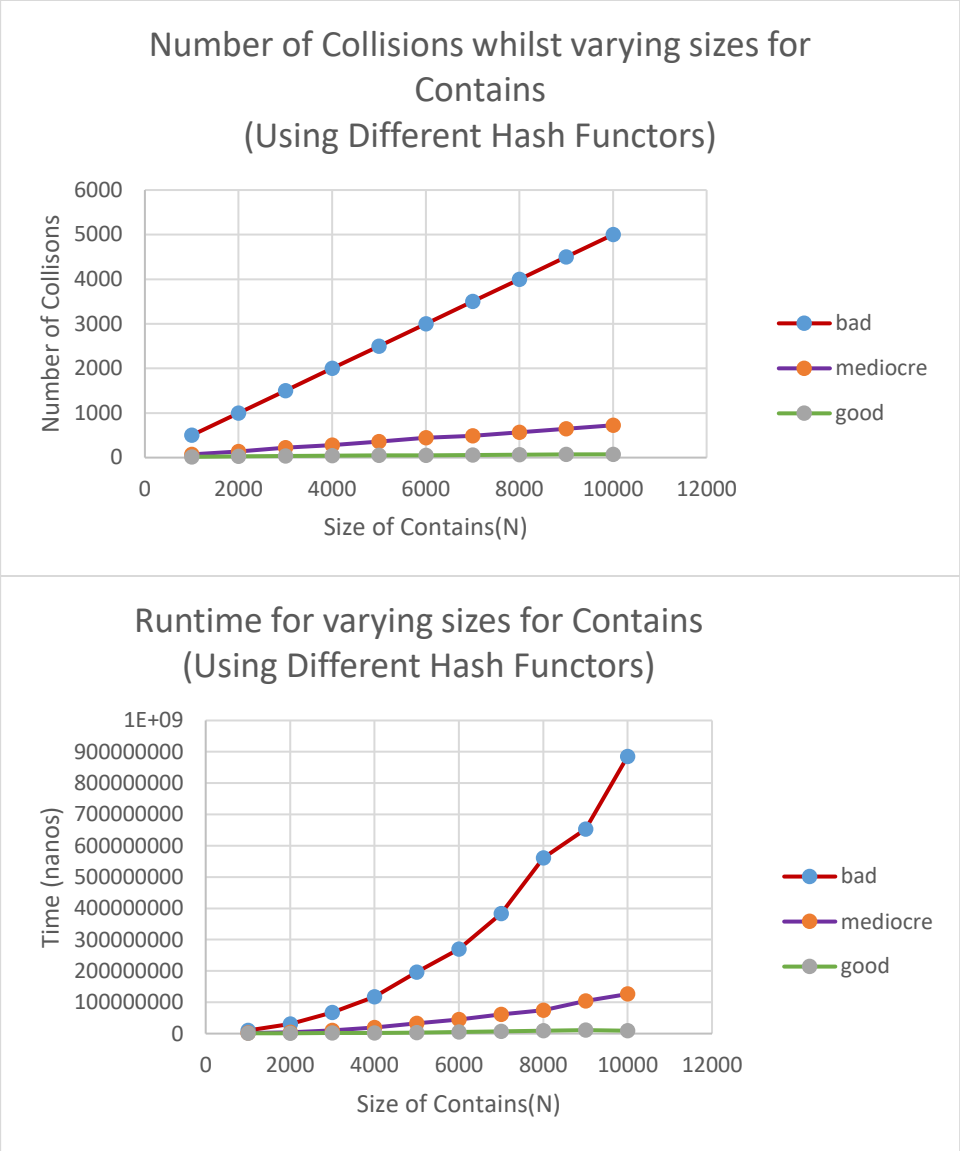
This hashing function works, but it is dubious at best. It only uses one character of the String and multiplies it by the same value every time. This will result in some collisions. This function could be improved by using various multipliers or by using more than one character. It does create some variance but not enough to truly give us a uniform distribution.

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).

Good hashing allows for constant lookups in all cases. This is in contrast to most chaining and open addressing methods, where the time for lookup is low on average, but may be O(*n*). A good Hashing Function ensures that the values are uniformly distributed and therefore avoids collisions and clustering issues. Our Good Hash function ensures a more uniform distribution through looking at each character of the string individually. This ensures that for every string even changing one character will result in a different hash.

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. Since the organization of your plot(s) is not specified here, the labels and titles of your plot(s), as well as, your interpretation of the plots is critical.

To begin we used the timing from a previous lab. Our work was much in line with what was recommended. We created a quadratic Probing table as the base and added N values to it in a random order. We then timed each consecutive add of size N with a sample size of 100. N increased in increments of 1000 up to 10000. We then added the contains method for each 100 tests to a total and divided by 100 to obtain the average collision value per hash Functor. In regards to the timing we simply created a QuadProb with the stated size and then timed how long it would take to call contains for every single value of n. As we can see from the tables above each function can be clearly distinguished from one another as the size of the array and contains is increased.

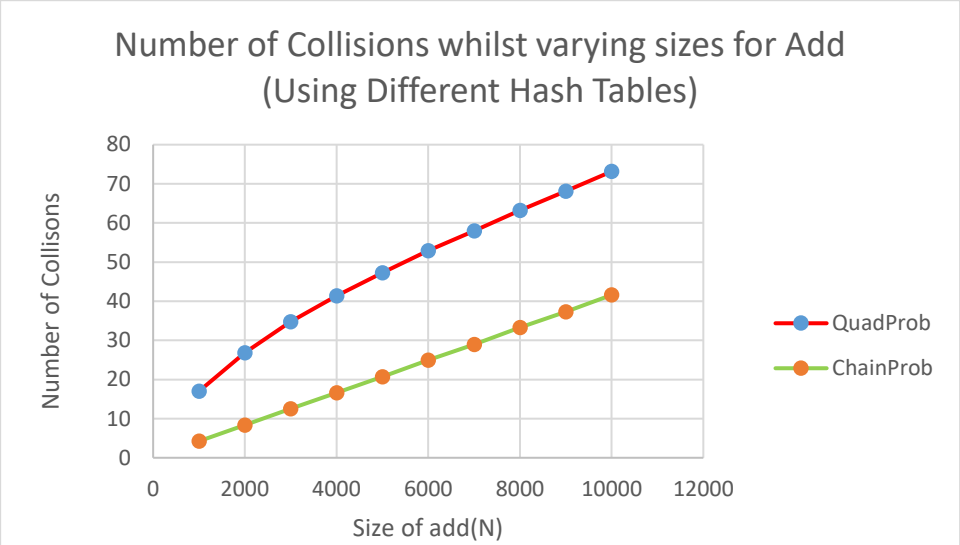


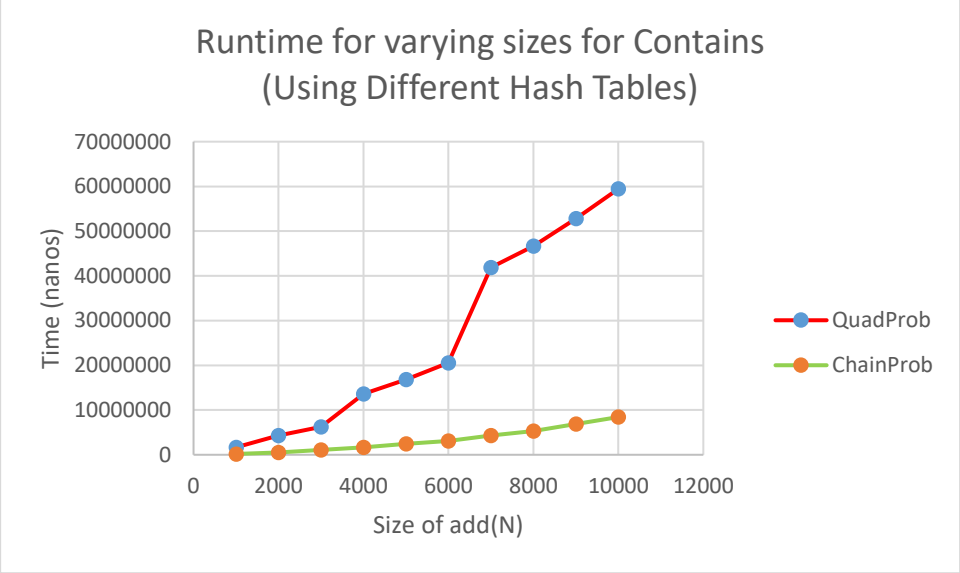
These graphs demonstrate the importance of a good hashing function. The bad functor is guaranteed poor performance because all values are given the same hash code. The bad functor ensures that each item has the exact same index value. The mediocre functor offers some variability which decreases the amount of collisions and increases the runtime. The good functor has drastically increased the runtime and number of collisions because each value is given a more unique index value. Here we can directly see how a hashing function affects the runtime of a HashTable.

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. Since the organization of your plot(s) is not specified here, the labels and titles of your plot(s), as well as, your interpretation of the plots is critical.

To begin we used the timing from a previous lab. Our work was much in line with what was recommended. We created one Chaining class and one Quadractic Probing class. Then we

added N elements to a list, for 100 samples and counted the collisions. We used the same hash functor for both hash tables.





The Chaining Class performed better than the QuadProb class with less collisions and faster runtime. The collisions may have been due to the way we set up the experiment. The sequential order created a constant amount of collisions for the Chaining Class. The QuadProb would likely begin to outperform the Chaining class at very large values as it appears logarithmic while the Chaining class has a constant slope. The Chain class has a much faster runtime because it is constantly adding values to the HashTable. The Quad Prob class must rehash and increase in size resulting in slow-downs of runtime. The way the Quad class must rehash severely affected its runtime however this is a real limitation of the class and therefore must be accounted for in the data.

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.)

The three Hash Functions have vastly different runtimes
The BadHashFunctor has a runtime of O(N+M) because it must scan through the entire HashTable each time. The values are not given a unique Hashcode.
The MediocreHashFunctor has a runtime of O(N/2) this is due to the way that most values are not given a unique index and therefore the HashTable must continuously scan through the entire table before rehashing itself.
The GoodHashFunctor has a runtime of O(C) because nearly every value is given a unique index and therefore it allows the HashTable to have constant runtime.

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

When using separate chaining for collision resolution in a hash table, the time required to search/remove an element from the table is O(n/m)O(n/m), that is, it scales linearly with our load factor.

Therefore, we don't want our load factor to get too high before we expand the table, or else the time required to search for or remove an element of our hash table will increase.

The logic is similar when we use linear/quadratic probing for collision resolution, though the effects of the load factor are far more dramatic. For each of these, the average number of probes required for an unsuccessful search in our hash table will increase exponentially as our load factor gets close to 1, while increasing rather slowly in approach to a load factor of 0.5.

So, regardless of our means of collision resolution, waiting until we've "filled" our hash table before increasing its size will result in lengthy searches.

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

It would be rather easy to implement a remove method inside the Separate Chaining Hash Table.
One would simply reference the LinkedList of values stored with the same hashcode and remove the specified value from the list:
entries[hashCode(item)].remove(item))
This would be encapsulated inside a Boolean return method.

However, it would also be easy for the Quad Probe Hash Table.
If(items[getIndex(item)].compareTo(item) == 0)

Then set the item equal to null thereby deleting it.
This would also be encapsulated inside a Boolean return 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?

I believe that all classes in java implement a .hashCode method. Therefore we could use this to generate a hashCode along with the algorithm we have already created. I do not believe it would be very difficult to “convert” our classes to generics, but, I may be overlooking something.

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

13

Programming partners are encouraged to collaborate on the answers to these questions. However, each partner must write and submit his/her own solutions.

Upload your solution (.pdf only) here by 11:59pm on November 9.