Travis Taylor Cassity

Assignment 10 Analysis Document

November 9th, 2016

1. For quadratic, Lambda means how full the actual array itself is. From what I can find Googling and tinkering with it myself, generally, we grow the list at 70% Lambda. Though after some experimenting on my own, growing it at much lower percentages (35%~) prevented overloading when **adding many duplicates.** Otherwise, it took so long for it to find a new slot with Quadratic Probing, the integer would typically overload first. This obviously is not a good idea if you're expecting a lot of duplicates (then again, Hash Tables in general would be very bad if you were expecting a lot of duplicates)

For separate chaining, Lambda still means about the same. If the array of LinkedLists was, say half-way full (say you have an array of 10 cells, and 1 cell has 5 items in it), you would grow it then. You would still want to grow the array at about the same percentage (70%~). As an extra tidbit, from reading a few Stack Overflow threads, it seems using a Separate Chaining hash table is preferable when dealing with 10 fewer items.

An extra 'overall' note to add is I found several pages insisting that it 'all depends.' It depends on how your hashing works, how far it skips, and what kind of data you're dealing with (duplicates or not, for example). While 70% seemed to be the 'standard(ish)' percentage across a number of pages, it can be very situational.

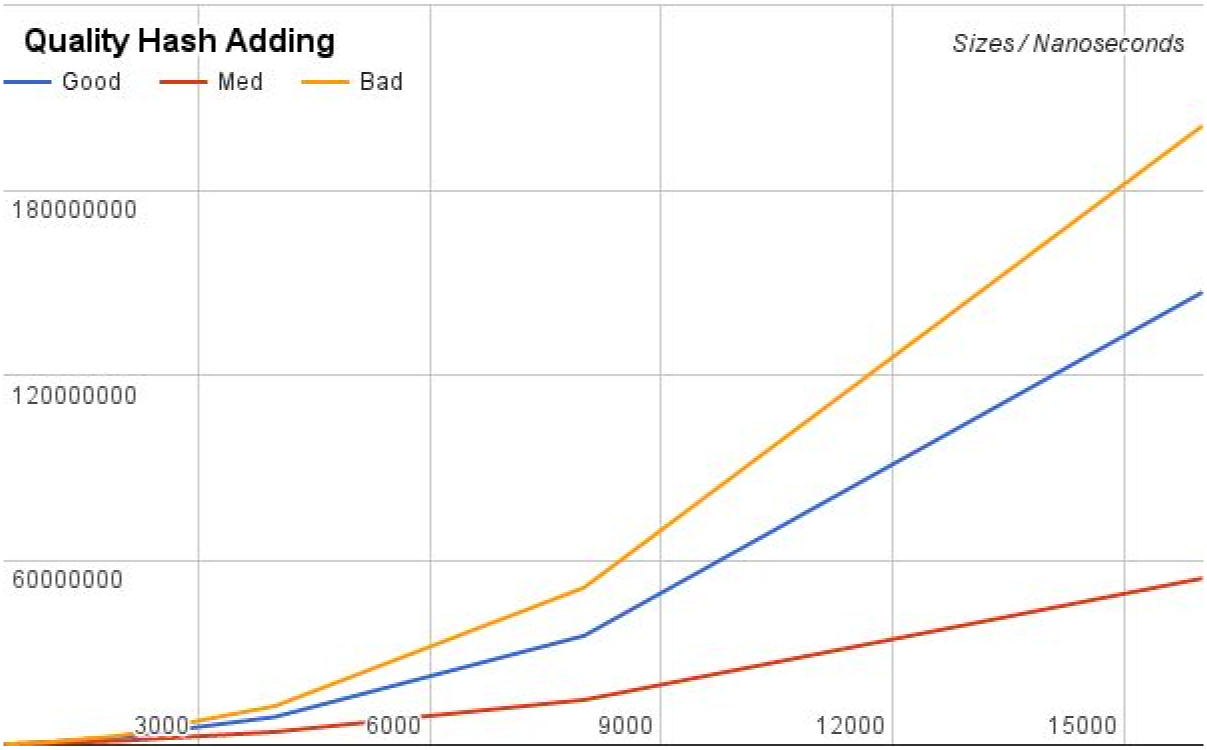
2. All my BadHash does is take the first character of the string and convert it to an integer. That's is. As you might example, this is pretty horrible; while it'll get the job done, it's going to make your hash table into a "dictionary" in a sense. All of the A words will be in a cluster, all of the B words will be in a cluster, etc. It is frankly horrible.

3. My version will convert the whole word into an integer and return that. This is okay - it's more processing power than the bad hash, and it'll give you something different for every non-anagram word, but it's still not as varied as it could be. While modding, you're likely to get a lot of similar numbers, especially if you're adding words alphabetically. (ex 'cat' will be very very close to 'bat', and made lead to some probing)

4. I'm hoping this one isn't too process-intensive, but it does some pretty different stuff, so hopefully it's pretty good. It adds all the characters up in the word as an integer, doubles it then finds the next prime number, squares it, mods it by the word length, then adds back on the 'word total'. This is enough of slight value-sensitive operations that, in my mind, will cause it to be very varied, even with similar words ('bat', 'cat'), as even one integer of difference can give back a totally different number.

5. Experiment process

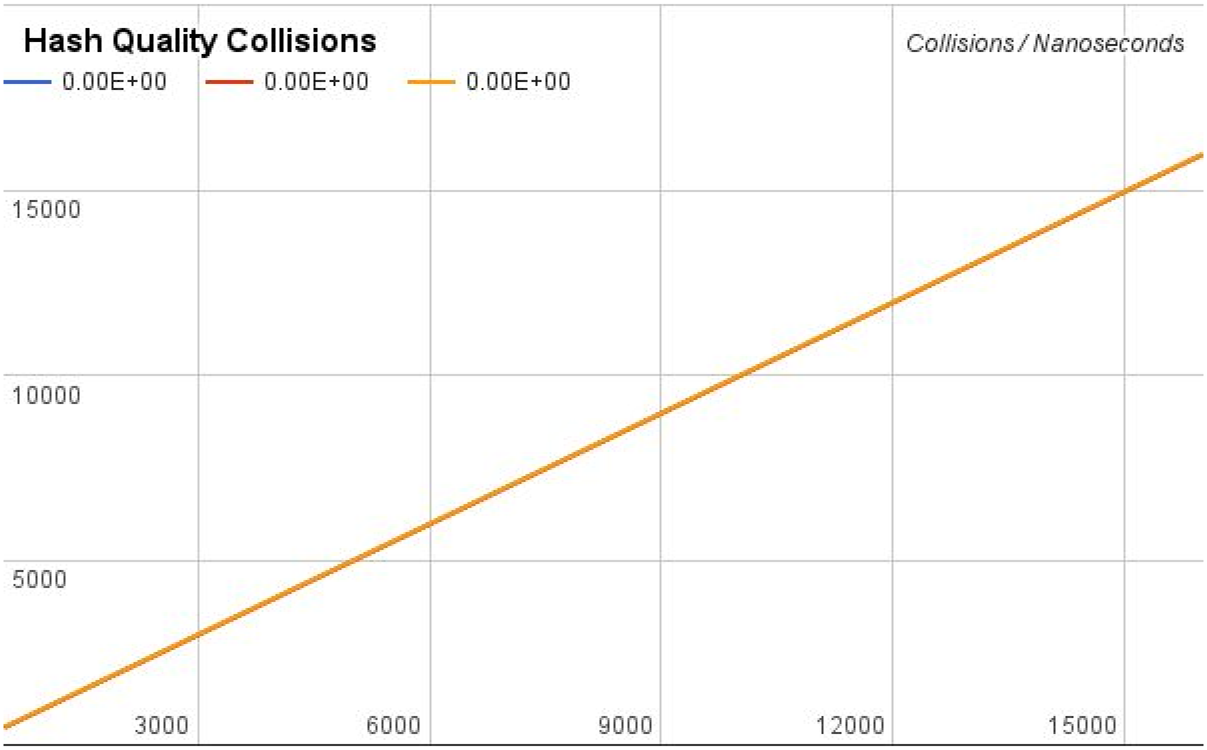
Using Quadratic Probing, added every word in a file to the hash table, starting with 500 words, 1000 words, 2000 words, etc up to 16000. Each timing used either of my ‘good’, ‘medium’, and ‘bad’ hashers. This was done 5000 times per sample size, then averaged.



Timing Interpretations

(Note that, as stated above, these all used the exact same processes besides the Hashers.) As evidenced above, the Bad hasher is, as expected, the worst. Though interestingly, the Good hasher is not as good as the Bad hasher. I’ll admit that I’m not 100% sure on what that is, but my guess would be that my “good” hasher just isn’t complex enough. It takes the first and last letter of the word, converts it to an integer, and squares it. The medium hasher converts each character into the word, adds them up, and returns that number. As this is complexity O(N), I figured my O(C) complexity for Good would not only be better, but more “random”, as even the slightest numerical change would return something completely different.

Evidently, this is not the case. As I think about it more, this does make some sense, as the first and last letter of words in the English language aren't often so different. I would likely change the formula a bit in the future, such as using the same process, but instead of squaring it, bring it to the Nth power, N = length of the word. This should still maintain an O(C) complexity (depending on how Java finds the length of a string, not sure on the details currently), and still bring a degree of predictable 'randomness' to it.



Collisions

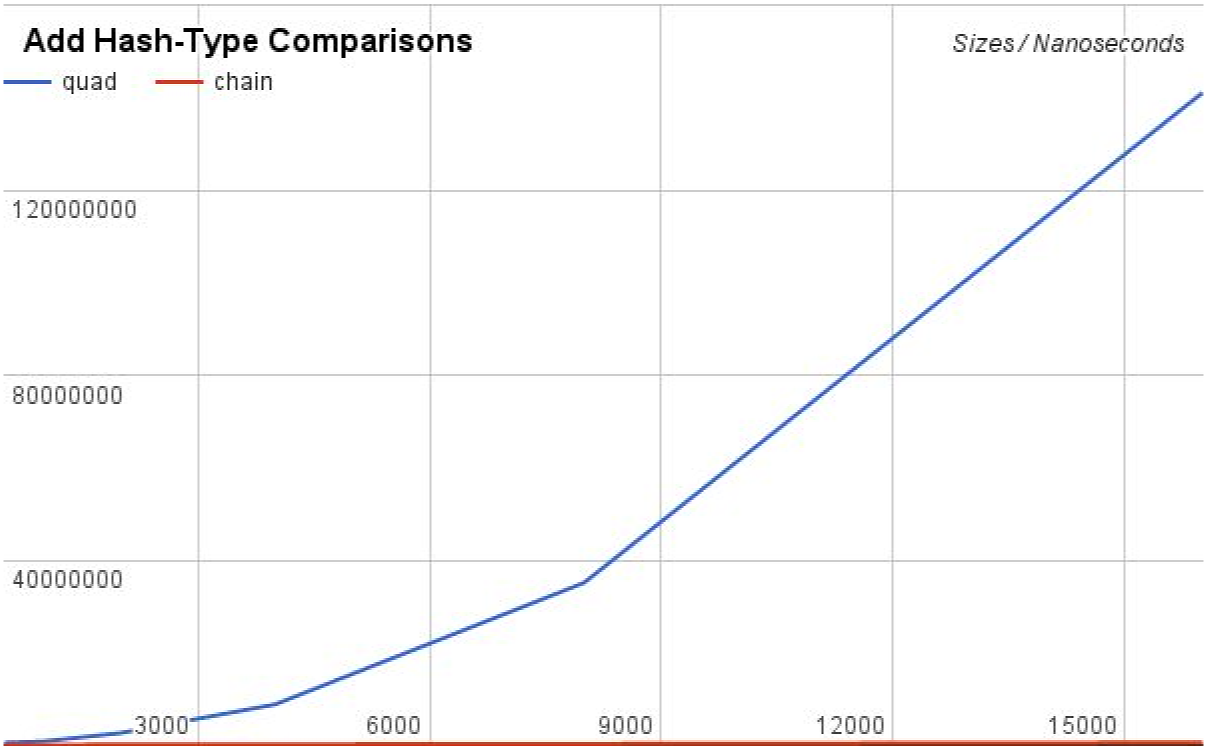
While the above lines all grow in the same fashion, there are some slight differences. The badhasher always had a few more collisions, while the 'mediocre' had the least amount of collisions, and the 'good' had the medium amount of collisions, though only slightly.

Example, at list size of 1000:

Good	**Mediocre**	**Bad**
976	966	983

All data points were the same as above; while all 3 methods were very close, there was a very slight difference. The Bad, again, makes sense, as it would have a collision with any word starting with the same letter, and as I discussed in the "Timing Interpretations" (and is evident for that part's graph), the Mediocre hash method has proven to be the 'best' for all adding functions. Despite there being a small difference in collisions, they have a huge impact on the overall speed of the function, as shown in the graph for "Timing Interpretations."

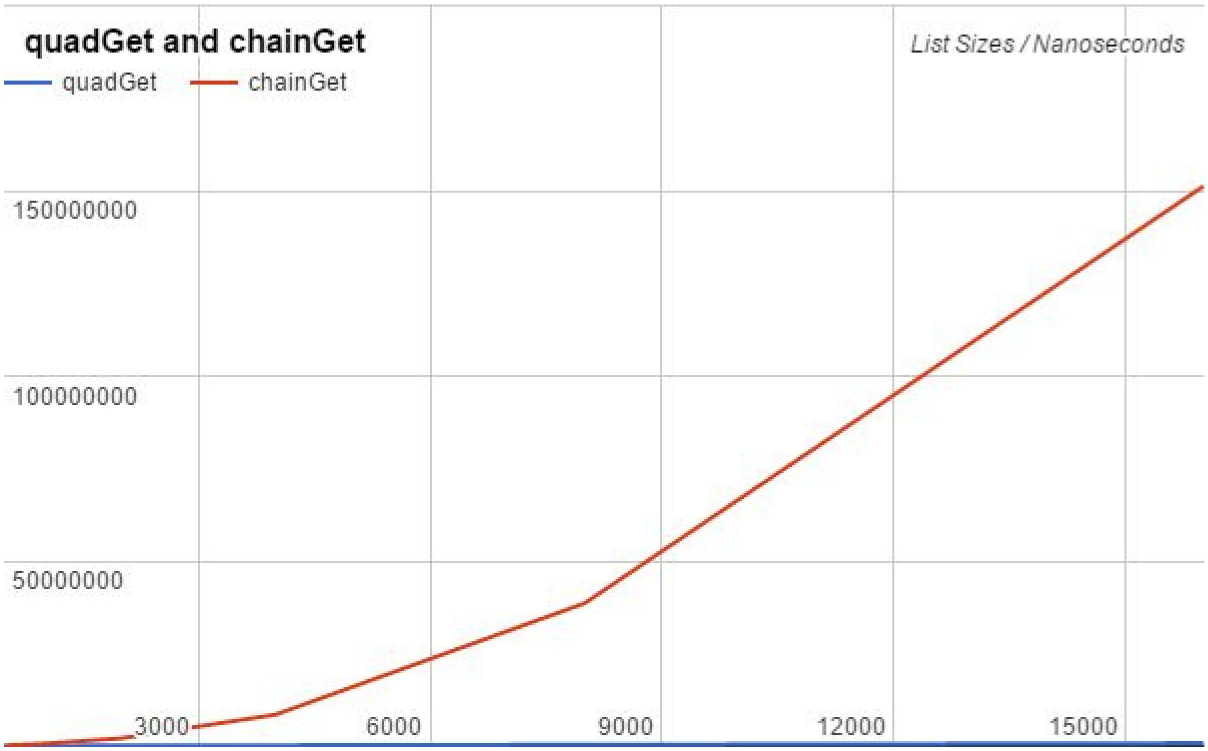
6. Explanation: Both used my “best” hash function and added all the words in several different text files. Starting from 500, 1000, 2000, etc up to 16000. There is an additional graph for using “contains” on all the added words, in the same order they were added. This was then averaged out over 5000 iterations.



Add Timing

Note: my Chaining hash table **does not** grow in cell size as this questions does not require it and it’s midterms season. Thus, the adding method is *significantly* better. Even if a collisions occurs, the time to add is still O(C), as it is simply added onto the end of the Linked List at that index location. It will never take more operations to add that.

The Quadratic hash table, on the hand, does grow. Additionally, as it must jump to another index when there is a collision, the time to add is not always O(N). (And as discussed in #5, due to the poor hasing and Quadratic probing in my code, it will almost always collide). As a result, this makes the Adding for the Quadratic hash table significantly worse. Much, much worse, with a complexity of O(n).

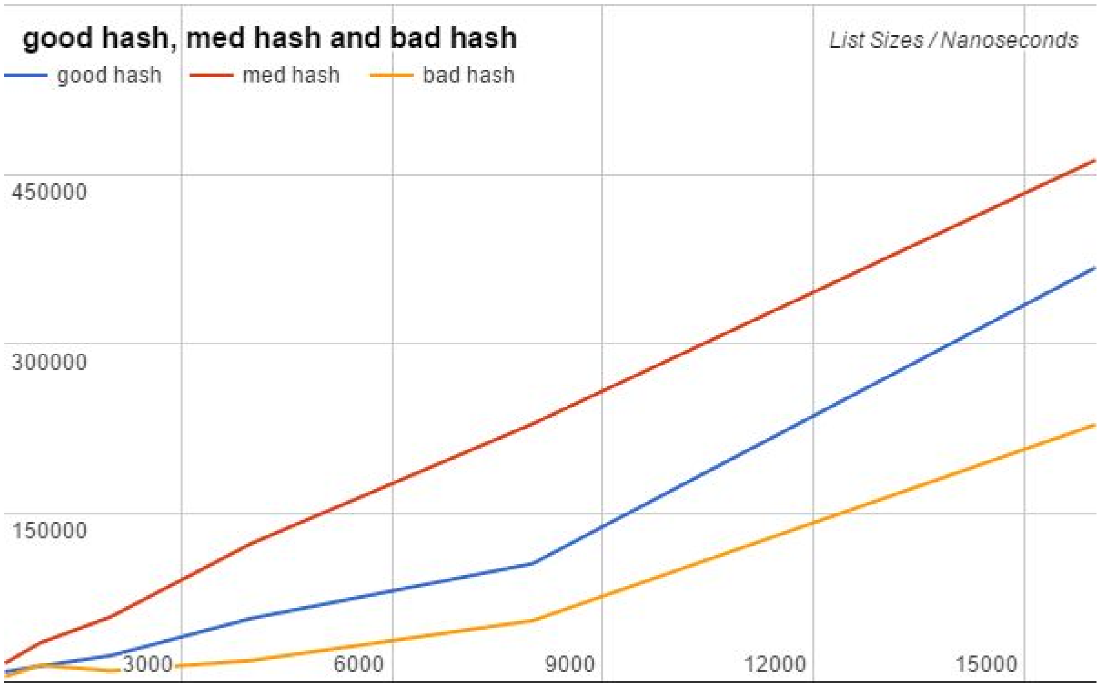


Get Timing

As you might expect from my explanation above, the Get timers are the complete opposite. Since my Chain's array does not grow, the time to grab a word increases by O(N), as it must traverse down a LinkedList O(N) times to find the item it is searching. On the plus side, since I do not grow the array and as it does not need to check multiple index locations, the complexity will never be worse than O(N).

But the Quadratic prober 'knows' where everything is, it just has to keep hashing the item until it finds it in the list. The best case complexity for Quadtratic is O(C), where the worse case is O(N), but in most cases, it finds the item very quickly. Hence, it's essentially a flat line on the graph above.

7.



Complexity

- Bad: O(C). Due to just grabbing the first letter and hashing it.
- Med: O(N). Due to adding up the int value of each letter through the word.
- Good: O(C). Due to adding up the int value of the first and last letter of the word, and performing several O(C) operations on it.

Collisions

I discussed the difference in collisions in-depth on question #5, under "Collisions". The shortened version is "Bad has the most collisions (expected), good has the 'middle' amount of collisions (unexpected), and medium has the least amount of collisions (unexpected).

However, one irk I have about this is *all* of my hash functions have a large number of collisions. Indeed, for a sample size of 1000 words, all 3 hash functions have over at least 900+ collisions, meaning nearly every single word collided. What this tells me is A) My hash methods are probably *all* terrible, and B), I have a bad Quadratic Probing method (Which is done by adding index + iteration^2. Ex., 2 + 2^2, 2 + 3^2, etc). In my research, this had been suggested as the best answer by several online answer and tutorial sites (and a number of Stackoverflow threads), but perhaps it is not. I can't imagine most hash tables have as many collisions as mine does.

While I did expect the Bad to collide extremely often, I figured the other two would so so more rarely, maybe ½ of the time on average. As it stands, however, all three methods collide at least 90% of the time.

8. In my code, the Quadtraic hash table will grow its array, but the Chaining hash table will not. As the Lambda (load) increases in the case of adding, the Quadratic table has to grow more often, which is expensive. As the load increases for the Chaining table, it makes no difference in adding. Its complexity remains O(C), as it needs only slap the item onto the end of the linked list at that location.
However, the Contains functions are somewhat backwards. Since the Chaining table does not grow sits array size, the LinkedLists begin to become combersome at large list sizes. What might have once been O(C) to find an item may now be O(N), as it is forced to slowly walk through every single node at the index until it finds the item it is looking for.
On the flip side, as the Quadratic table does not need to grow, it simply follows the hashing formula until it finds the item, which is a significantly faster process than ever having to copy over the whole array, making grabbing data for a Quadratic table significantly faster than a Chaining table, even with an increased Lambda (load size).

9. This would be super easy. For Separate Chaining, you'd simply hash the item, iterate through the LinkedList at that index, and remove it.
For Quadratic probing, it would be just a little bit more complex. You'd hash through the table as if you were adding it, storying every item you found that *wasn't* what you were trying to remove. One you found the item to remove, set the index to null. You'd keep hashing that item until you found an empty index, still storing each item. Then you'd rehash them all back into the table.
Which means if you're expecting to do a lot of removing from your table, you might want to consider using Separate Chaining.

10. Sure it is! Since the hash function is imported into the classes, it'd be very easy. I'd have to change the array types and method signatures to Generic, and that's really about it. Everything else is handled by the hasher, and so long as that returns an integer, there is very little extra work on my end that'd have to be done to make this work with generics.

11. About 15.