Assignment 10 Analysis

Brayden Wright
u0895942

1. **What does the load factor λ mean for each of the two collision-resolving strategies and for what value of λ does each strategy have good performance?**

In the quadratic probing strategy, λ is defined as the fraction of the table that is currently occupied. The closer to half the size of the table that is full, the better the performance seems to be when using the quadratic probing strategy. In the case of the separate chaining strategy, λ is defined as the average length of the linked lists (the chains). For best performance when using chain hashing, one should rehash when λ (the average length of the chains) grows past a predetermined threshold.

2. **Give and explain the hashing function you used for BadHashFunctor.**

My BadHashFunctor hashing function is really simple. All it does to compute a string's hash is return the string's length. Bad performance was expected because there are many potential strings that are the same length, thus would produce the same hash and cause a collision. As expected, this hashing function produces a really large number of collisions.

3. **Give and explain the hashing function you used for MediocreHashFunctor.**

This is my MediocreHashFunctor hashing function:

```
return (item.length()*13) + (item.length() > 0 ? item.charAt(0) : 1)
```

It takes the input string's length and multiplies it by two. After, it then adds the value of the first character in the string, or simply 1 if the string is empty. This method of hashin is a little better than the bad method I implemented mostly due to simply adding another variable that can change the hash. After some tests, it does reduce the number of collisions by a significant amount, though there are still many.

4. **Give and explain the hashing function you used for GoodHashFunctor.**

My GoodHashFunctor hashing function is modelled after results I got from doing some quick research on good hashing functions for hash tables. This is most of the function:

```
for (char chr : item.toCharArray()) {
   hash = (hash*101)+chr;
}
```

Brayden Wright
u0895942

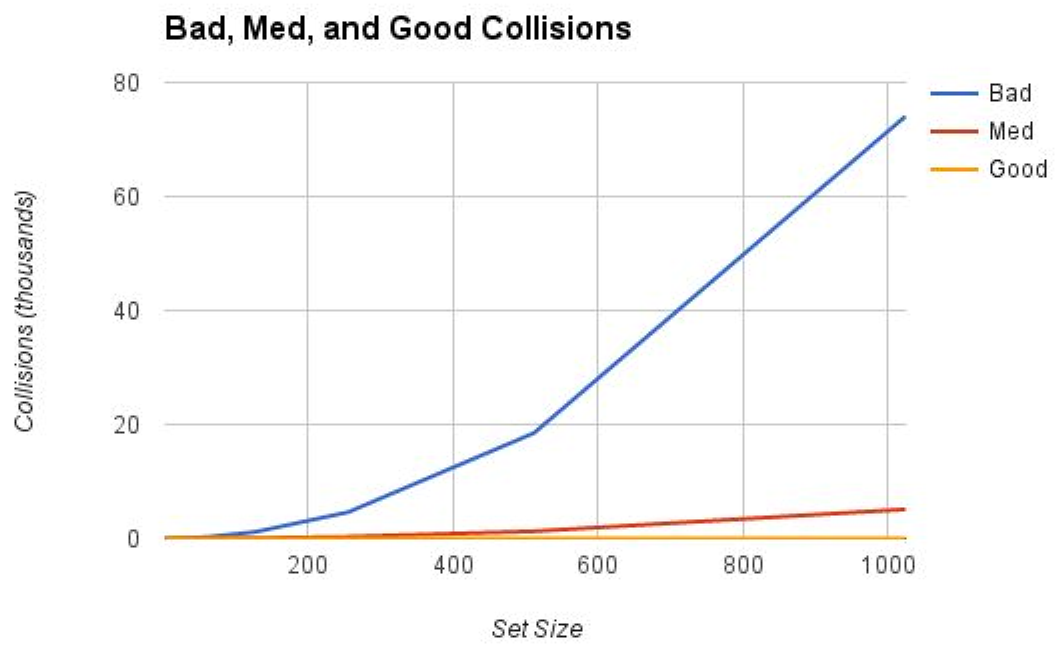
This hashing function does a couple of things. First, it initializes an integer named 'hash' and sets it equal to a prime number, 13 in my code. Then, for each character in the passed string, the hash gets set to the hash multiplied by another prime, in my case 101, and has the current character value added on. When done with every character in the passed string, the hashing function returns the absolute value of the hash.

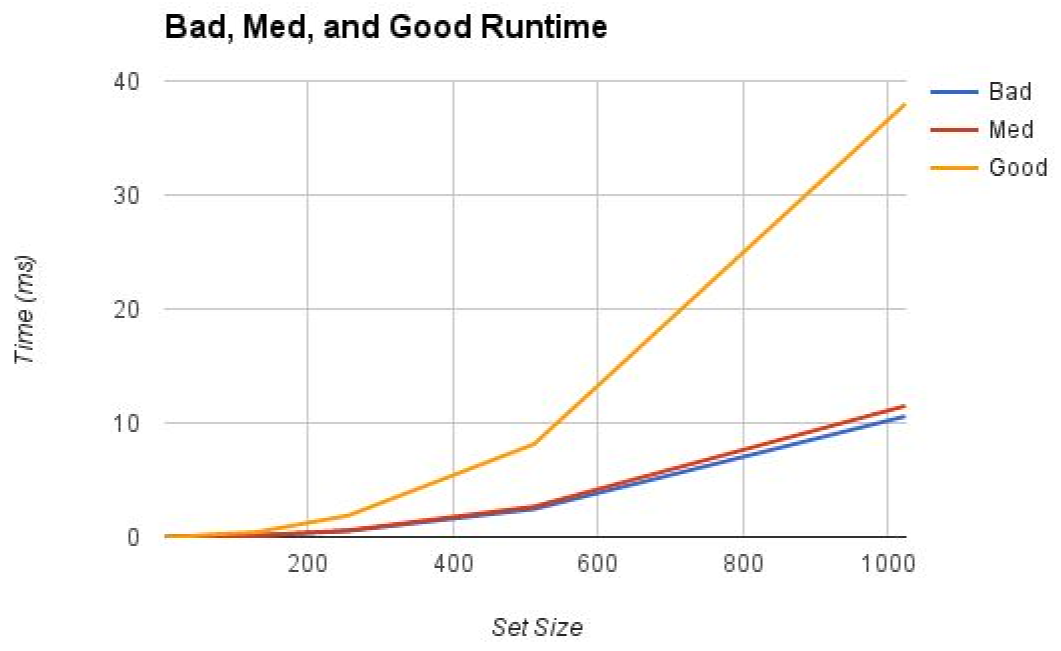
The primes are used in hopes that the resulting hash will be reasonably unique. This is backed by the idea that primes, when multiplied by each other, are more likely to generate a somewhat unique number than if non-primes were used. We need these reasonably unique hashes to avoid collision as much as possible.

5. **Design and conduct an experiment to assess the quality and efficiency of each of your three hash functions.**

I tested the the quality of each of the three hash functions by performing the hashing every item in a set of a predetermined set size that increases periodically, doing so one-thousand iterations for each set size. The number of collisions would be determined by the total number of collisions that occurred divided by the number of iterations, in this case one-thousand. Thus getting an average number of collisions per set size. The better the hash function, the fewer number of collisions that occurred, with the good hashing function producing no collisions on average. The graph on the next page titled, "Bad, Med, and Good Collisions" illustrates this.

The efficiency test was performed in the same manner, same values and everything, with the only difference being that I tracked the average time to hash every item in a set instead of the number of collisions. Something that surprised me at first, though makes sense upon reflection, was that the better the hash function, the slower it ran with the good hash function taking nearly twice as long as the bad hash function. The graph on the next page titled, "Bad, Med, and Good Runtime" illustrates this.

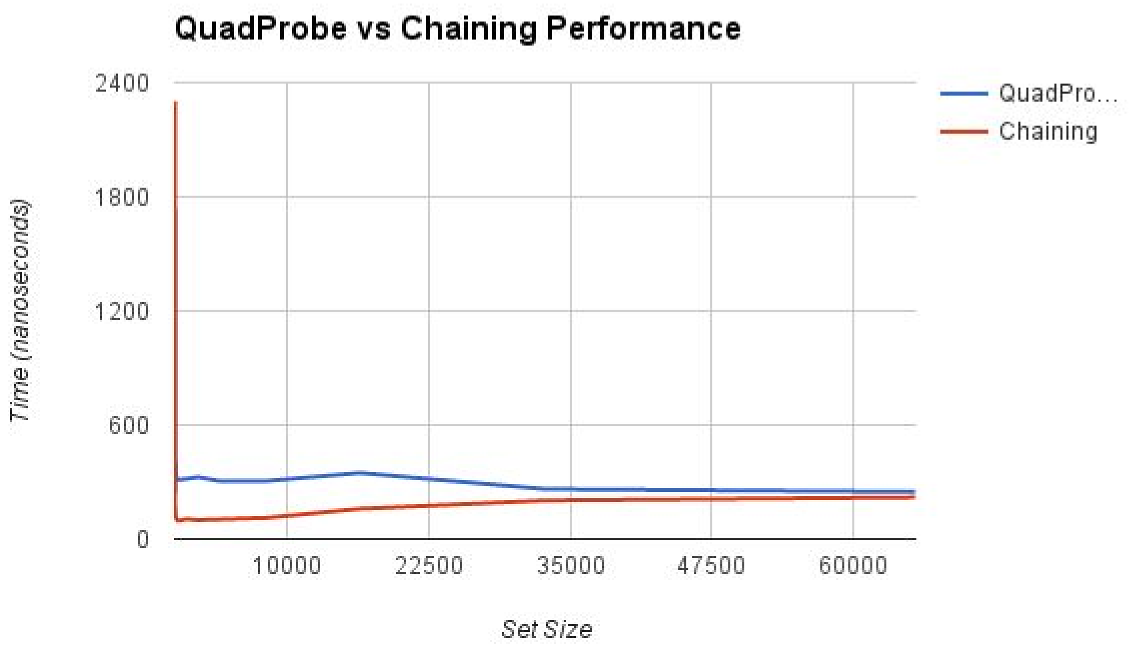




6. **Design and conduct an experiment to assess the quality and efficiency of each of your two hash tables.**

For the purposes of testing these two hash table types, they are going to use the good hashing method I made. Since they will be using the good hashing method, and that method appears to produce zero collisions on average, the quality in regards to collisions is the same.

When timing, I timed adding every item from a list of strings to the table and took the average. The list of strings would increase in size and every size would be iterated over one-thousand times. The tables were initialized with a capacity of at least the string set's size divided by two. I was expecting the chaining hash table to perform significantly faster than the quadratic probing table, but it turns out they're fairly close. The chaining hash table did seem to outperform the quadratic hashing table by just a little bit though. Considering the runtimes were gathered in nanoseconds, and there's a variance of only about two-hundred nanoseconds, I would say they both run in about O(N) time, with fluctuations due to my computer's processor.



7. What is the cost of each of your three hash functions (in Big-O notation)?

All of my hashing functions seem to be O(N), where N represents the size of a string, as far as I can tell. The bad hashing function relies on the length, or number of characters, in the passed string; my mediocre hashing function relies on the same thing, just with an added constant; and my good hashing function has to loop through every character in the given string. So, they all rely on the length of the passed string to some degree.

After some testing, the performance of each of these hashing functions is about as expected. The only one that is not quite as expected is the mediocre hashing function, which produced fewer collisions than I anticipated.

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

The load factor has a significant effect on the performance of my hash tables. For the quadratic probing hash table, the load factor is about half the capacity of the underlying array. If the total capacity was only increased by a set amount each time the load capacity was reached, then the performance would suffer since more resizing would have to occur. With the chaining hash table, the load factor is the average length of all the chains. Since there wasn't a rehashing requirement for the chaining hash table though, I didn't implement one and thus the load factor isn't taking into much consideration. I instead just initialize the chaining hash table so that the total number of chains is one-half the total size of the input set of strings.

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

I would implement a remove method for these hash tables by adapting a variation of my contains method. Instead of returning true though, in the case of the quadratic probing hash table, I would simply check if the item is stored, set the stored item to null, decrement the size count, and then return true. Of course, false would be returned if the item was not found to be within the quadratic probing hash table. This would effectively remove the item from the quadratic probing hash table. For the chaining hash table, I would check if the item is stored, remove it from the chain, then return true. Again, if the item was not found to be within the chaining hash table, false would be returned.

10. As specified, your hash table must hold String items. Is it possible to make your implementation generic? If so, what changes would you make?

Brayden Wright
u0895942

Yes, I think it would be possible to make the hash table implementations generic, though it would be a bit difficult. Instead of generating the hashes by looking at the characters in a string, the hashes would probably instead have to be implemented in a way that uses the passed item's byte value or a specially crafted comparator of some sort. Other than changing the hashing function itself though, and making the backing array and linked lists support generics, I don't think anything else would need to be changed.

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

I think I spent somewhere between six and ten hours on this assignment.