Ashley Grevelink u0749357
Assignment 10 Analysis
November 8, 2016

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?

The load factor for quadratic probing means (size / capacity) for the array. With separate chaining, the load factor is the average size of the LinkedLists which have at least one item. A load factor below 0.5 is best for quadratic probing. Rehashing was not required for the separate chaining hash table, so I do not know a good load factor for this.

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).

```
public int hash(String item) {
      return 5;
}
```

This hashing function returns a prime constant number and yields a collision N – 1 times—basically every time. I used this bad hashing function for comparative reasons to my mediocre and good hashing functions.

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).

```
public int hash(String item) {
      return item.length();
}
```

This hashing function returns the length of the item, which varies by item, offering some hope to avoid collisions. However, there are many types of lists where the item lengths will be similar, if not the same.

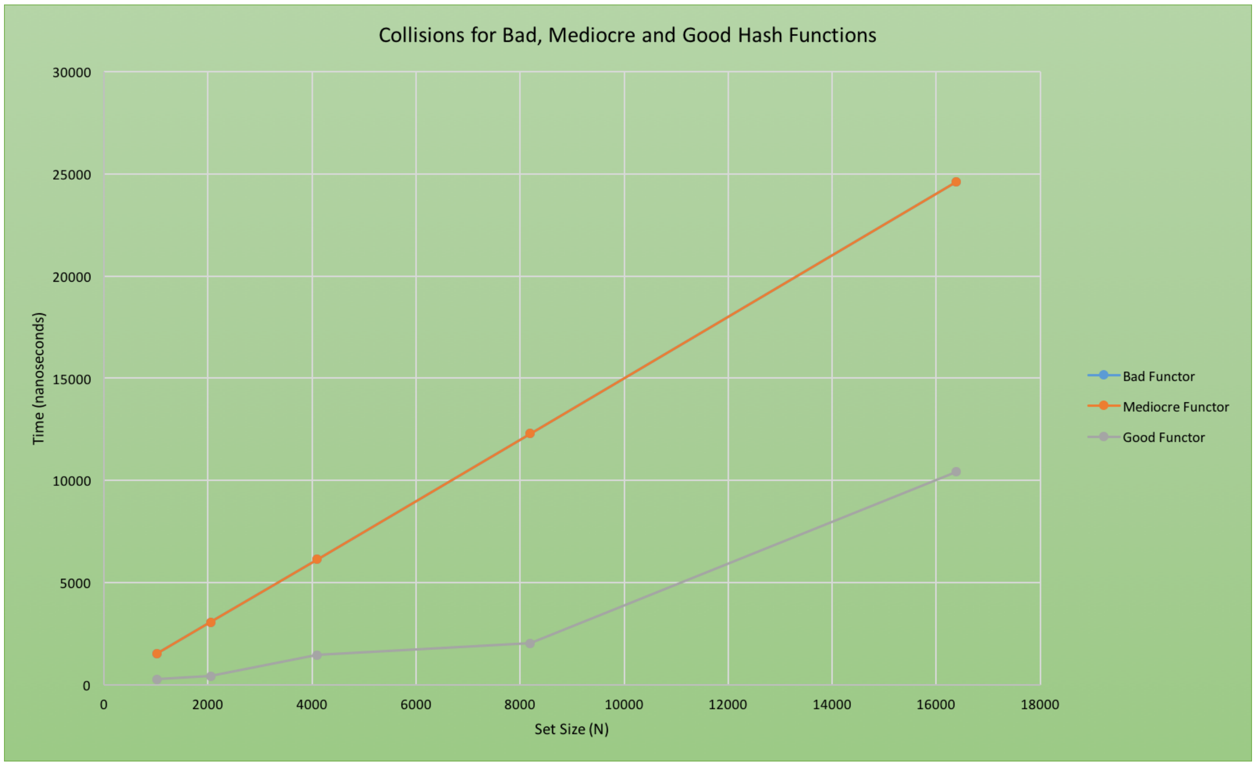
4. Give and explain the hashing function you used for GoodHashFunctor. Be sure to discuss why you expected it to perform well (i.e., result in few or no collisions).

```
public int hash(String item) {
      int hash = 5;
      for (int index = 0; index < item.length(); index++) {
          hash = hash*23 + item.charAt(index);
      }
      return hash;
```

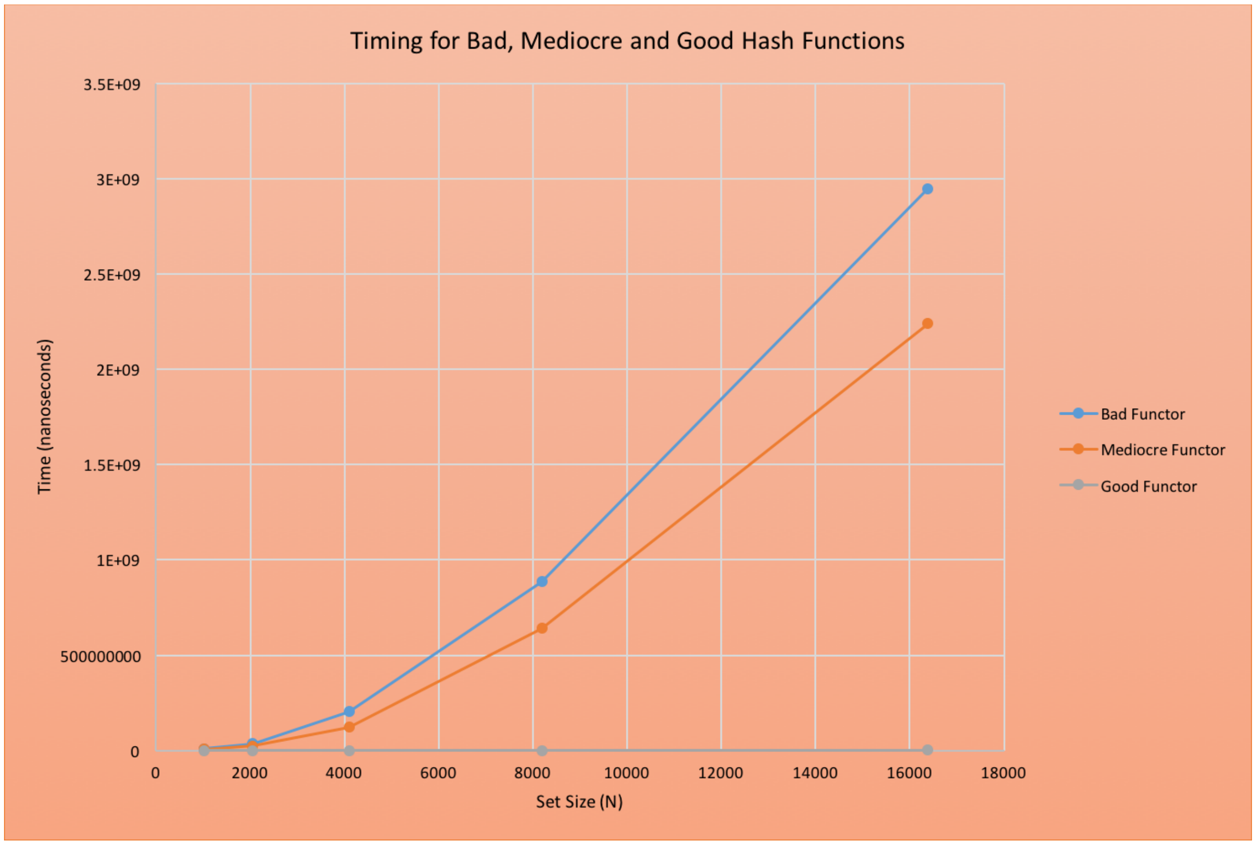
}
For my good hashing function, I studied hashing function theory online and found one where the item length is used as well as each of the characters of the item, in this case a String. It has a loop which iterates through each characters and multiplies the char by a prime number, in this case 23. This had far fewer collisions than the mediocre hashing function, because many operations are performed on two different aspects of the unique item being passed to the function.

5. Experiment to assess the quality and efficiency of each of your three hash functions:

I measured the number of collisions for each of my hash functions for the set sizes (N) between 10^10 and 10^14. I duplicated this 5 times to help smooth outliers. I created an ArrayList of ints for every number in the current set, and then timed addAll() with this ArrayList on a QuadProbeHashTable. I also counted the number of collisions for each of the three hash functions in separate plots. Now I have six sets of data: three with timing data for addAll() and three for collision numbers.



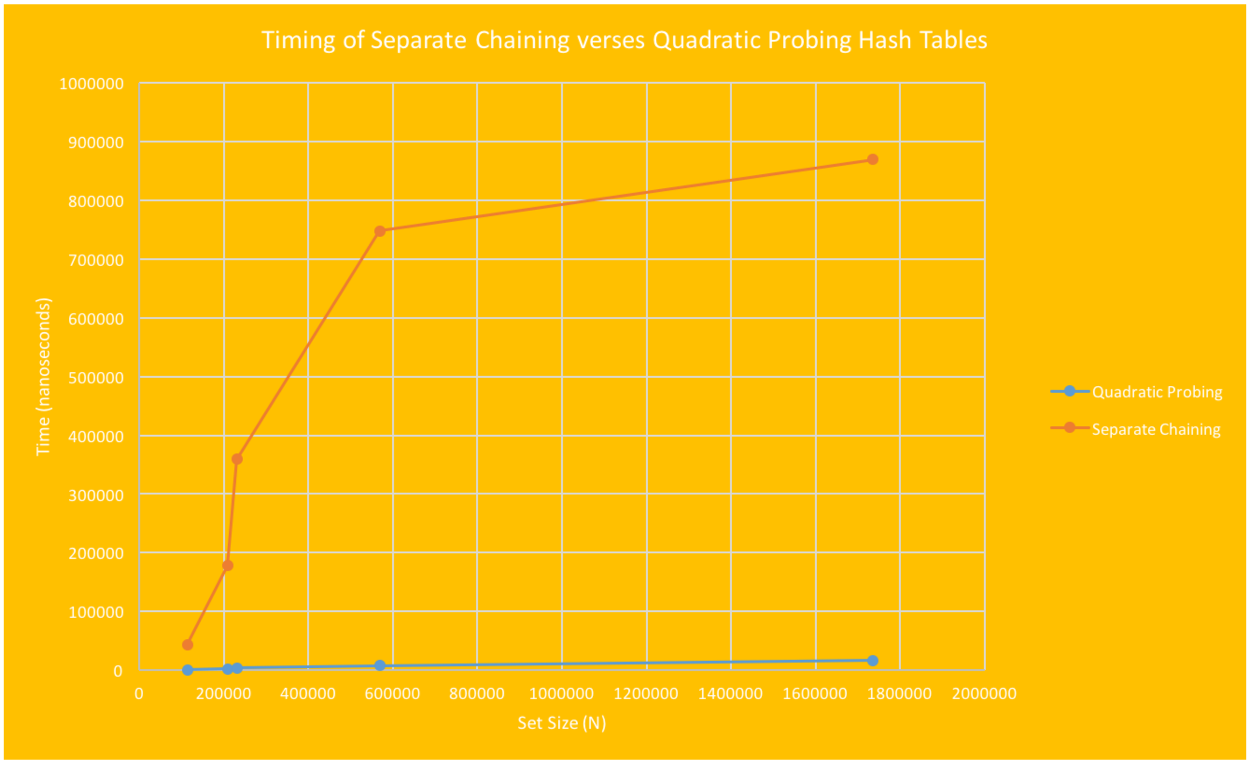
My mediocre and bad functors actually had the same results due to the nature of this specific testing set of integers. In other situations, they would not necessary have such similar results, but in this case, my mediocre functor was basically a constant due to the data. The good functor showed much better results.



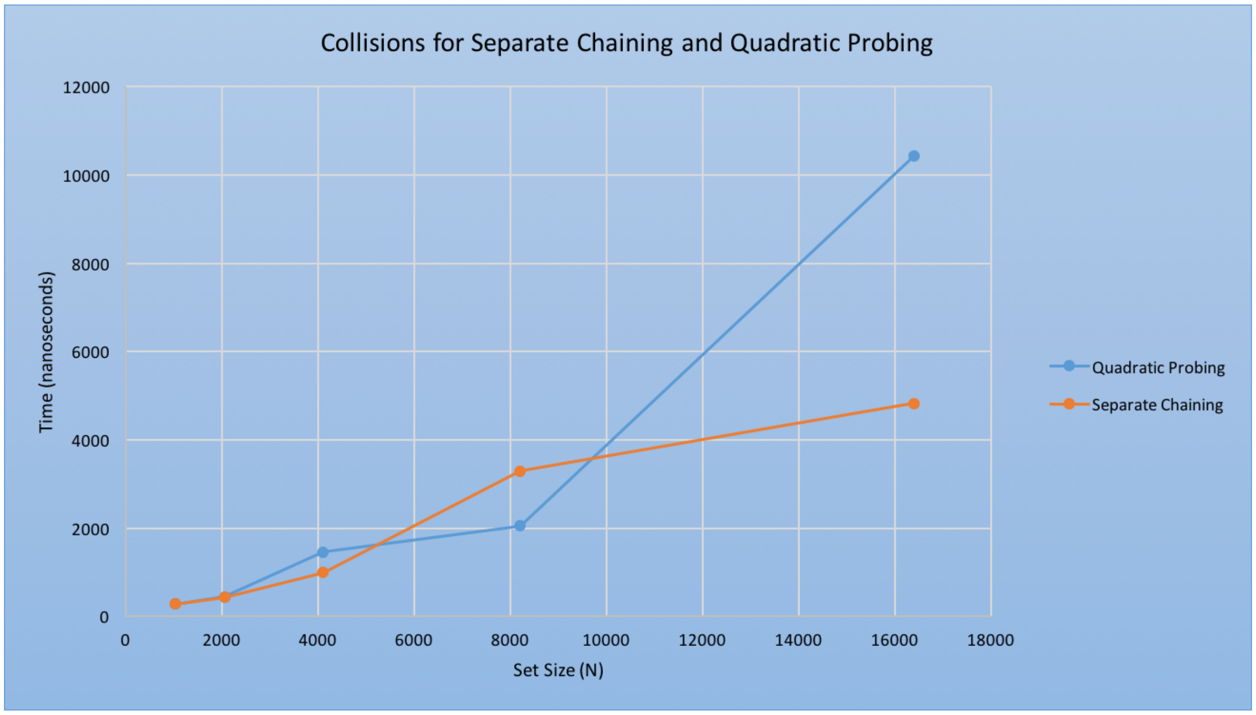
My bad and mediocre functors varied slightly on this graph in time even though they did not vary in collisions. Compared to the linear, or even possibly exponential, nature of the bad and mediocre functors, the good functor appeared to have a complexity of O(c).

6. Experiment to assess the quality and efficiency of each of your two hash tables:

I measured the number of collisions for with my good hash function for the set sizes (N) between 10^10 and 10^14. I duplicated this 5 times to help smooth outliers. I created an ArrayList of ints for every number in the current set, and then timed addAll() with this ArrayList on a QuadProbeHashTable, then a ChainingHashTable. I also counted the number of collisions in a separate plot. I gathered four sets of data for this experiments: two of timing and two of collisions.



In the case of number of collisions, separate chaining appears to have O(logN) behavior, whereas quadratic probing has O(c) complexity. Quadratic probing has much better complexity in terms of number of collisions. This is probably because I did not implement resizing and rehashing for my chaining hash table.



In terms of time, quadratic probing and separate chaining were virtually equal and appeared linear, but at this small scale it is wrong to make a definitive conclusion.

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.)

Good: O(c)
Mediocre: O(N)
Bad: O(N)

My mediocre hash function did not result in the number of collisions I desired. It was linear time with the bad hash function, which I did expect. I hope I could test my mediocre functor on a different type of data—perhaps URLs—to see if the results differ and perhaps come closer to O(c).

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

My quadratic probing hash table seemed to be extremely efficient as it kept its load factor below 0.5. Its complexity seemed to be O(c) as desired. The load factor was clearly too high for my chaining table and it would be interesting to experiment with different load factors for this type of hash table.

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

As discussed in class, I would add a Boolean value to each index in my array to mark whether or not the value had been removed or not. I would keep values in my table as placeholders until a time of rehashing.

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 could make my implementation generic. Not much of the code would actually change, because the hashing and unhashing of the values makes the code naturally very generic. I would have to add more type parameters considering hierarchies and wildcards, unchecked type casting, and suppression of raw types, along with defining a variable for my type of data. This would affect my functor class type parameters as well, but definitely could be accomplished.

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

About 8 hours.