Eduardo Ortiz
U0922628

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?

The load factor is the ratio of how full the hash map is. It essentially sets how full a hash map has to be before it must rehash and grow in size. In quadratic probing if the load factor is too big the chance for Collison and clustering increases as the number of available spots is smaller and the table is getting closer to full. Quadratic probing works by incrementing the distance from x(the index) by squares i.e quadratic fashion thus a bigger table to work with more likely ensure that a collision will not occur by spreading values out diminishing primary clustering found in linear probing. In linear probing collision is worked through by checking the next available spot in the table until an open value is found. As a consequence as the table fills up more (the load factor increases) the chance of collision is heavily increased. This differs from quadratic however in that it suffers from primary clustering as values take on spots close to one another. As a consequence the load factor must be low, more optimally below .8 for after that the chance of collision is very high. In our case, the hash table is rehashed when λ = 0.5 at that time, there are approximately N/2 inserts and the total cost is N/2 + N per add. As the load factor grows larger, the hash table becomes less efficient as n is getting bigger. The constant runtime of a hash table assumes that the load factor is kept below some bound. Taking all of this in however,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.

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 2;
      }
```

The Hashing function I used for the badhash as seen above would always return 2 regardless of the string this is very bad as you will get collisions for every added element into the table.

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) {
            int hash = 0;
            int prime = 11;
            char[] word = item.toCharArray();
            if (word.length <= 2)
            {
```

```
            hash = word[0] * prime;
        }
        else if (word.length == 3)
        {
            hash = word[1] * prime;
        }
        else  {hash = word[3] * prime;}return hash;}
```

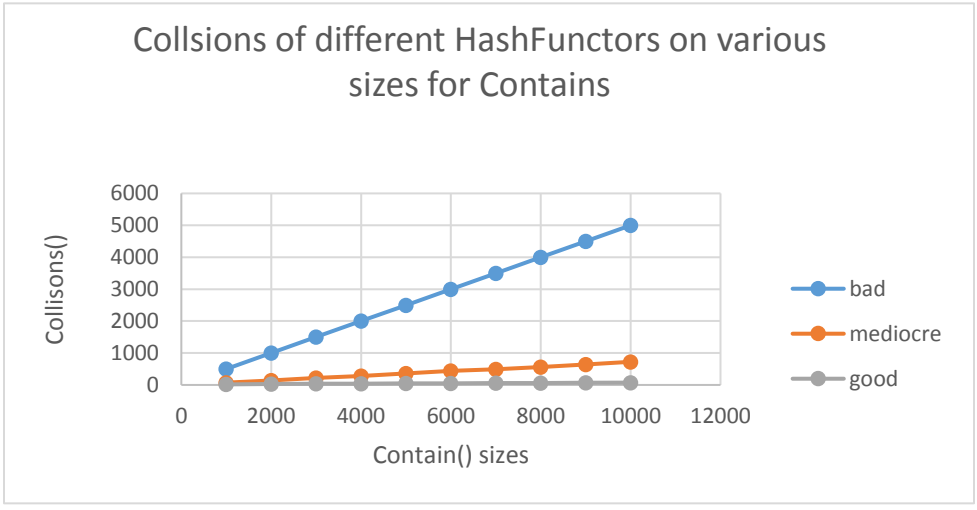
The mediocre hash functor as seen above is better than the bad one in that it returns similar values depending on the length of the word, however these values are not unique and result in quite a few collisions if two words of length 1 or 2 are encountered or any words above 3 characters in length.

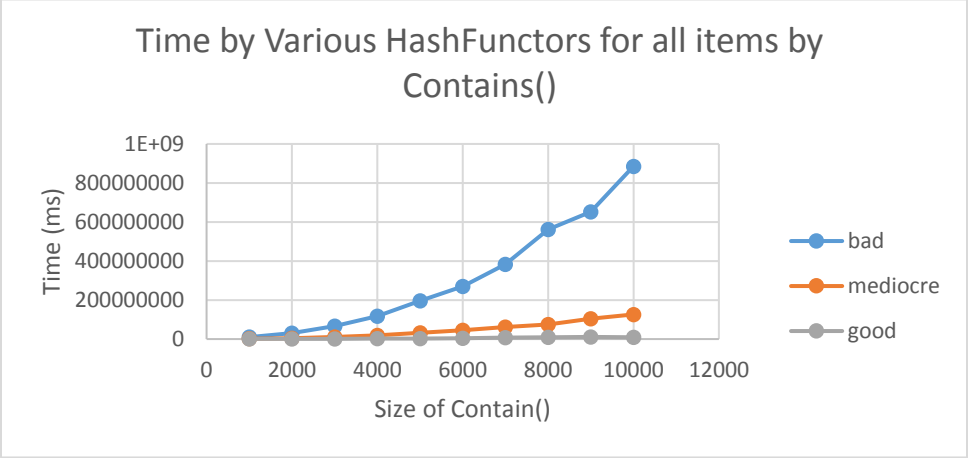
4. Give and explain the hashing function you used for GoodHashFunctor. Be sure to discuss why you expected it to perform well (i.e., result in few or no collisions).

```
public int hash(String item) {
            int hash = 0;
            char[] word = item.toCharArray();
            for(char c: word)
            {
                hash = hash + c;
            }
            return hash;
        }
```

The good hash functor I used gave each word a different value depending on how many characters they had in. This was good because varying sizes of words would attain different values. When juxtaposed to the bad and mediocre it outperformed better as it was not limited by words of only 1, 2 and other sizing. It still however encountered collisions as word size can only vary by so much.

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.





A recommendation for this experiment is to create two plots: one that shows the number of collisions incurred by each hash function for a variety of hash table sizes, and one that shows the actual running time required by each hash function for a variety of hash table sizes. You may use either type of table for this experiment.

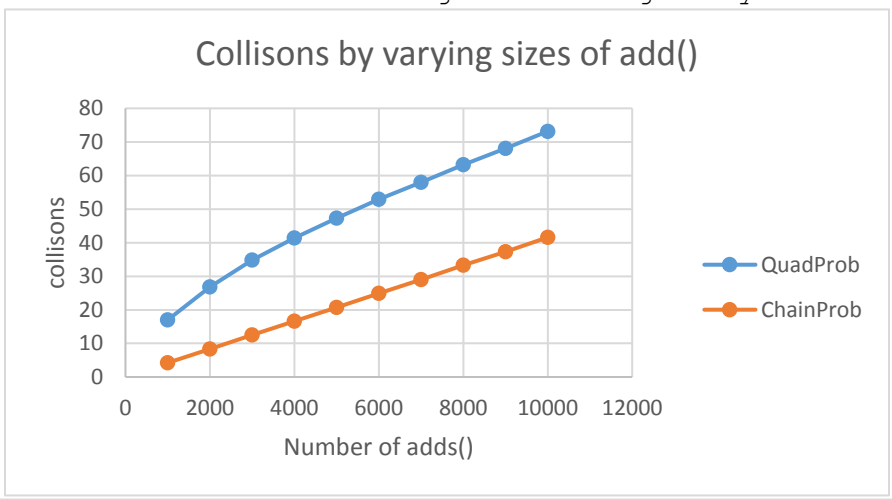
To begin we utilized the timing class given to us by Ryan during one of our earlier labs. We decided to proceed by following the recommended directions given to us for the assignment. We created a quadratic Probing table as the base and added n values (the size of the list) to it in a random order by utilized a private randomization helper method in the same class. We then proceeded to time the amount of time it would take to add each consecutive add 100x incrementing each value by 1000 up to 10000. We then added the adds for each 100 tests at a value and divided by the number of tests 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. This very simply shows the impact a good vs bad hash functor can have on the total performance on the hash table. Going back to the initial question we can see the bad functor accounts for a tremendously less efficient hash table then the other two encountering substantially more collisions than the other two. This again is because it is returning the same index value for any item brought it while the other two are more similar the good functor bests the mediocre simply because it offers more variability in terms of inputted values and because of that the number of collisions and time is lower than the others.

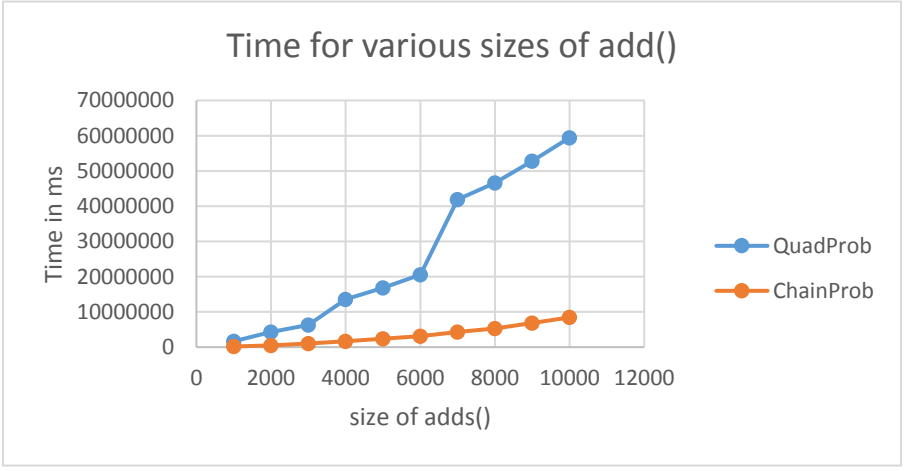
6. Design and conduct an experiment to assess the quality and efficiency of each of your two hash tables. Carefully describe your experiment, so that anyone reading this document could replicate your results. Plot the results of your experiment. Since the organization of your plot(s) is not specified here, the labels and titles of your plot(s), as well as, your interpretation of the plots is critical.

A recommendation for this experiment is to create two plots: one that shows the number of collisions incurred by each hash table using the hash function in GoodHashFunctor, and one that shows the actual running time required by each hash table using the hash function in GoodHashFunctor.

To begin we utilized the timing class given to us by Ryan during one of our earlier labs. We decided to proceed by following the recommended directions given to us for the assignment. For the number of collisions we created two different hashTables one of the chainProbing method and

the other of the quadraticProber. We then added the same amount of n about a 100x for n and then averaged the number of collisions per table to post on the graph. For the timing of various add we simply recreated the experiment above but instead of calculating for number of collisions we sought to see how long it would take for each hashTable to do so. We used the same hash functor for both to maintain the different hashtables as the only controlled variable. As we can see form above the quadratic Probe was outperformed by the Chain Prob in terms of time and collisions. In regards to collisions this may have been a result of the incremental way in which we added values to each table. We used a nonrandom order which benefited the chainProb in that each value was next in line per say in the list thus creating a constant rate of collisions as can be seen in the graph. While the QuadProb was stuck squaring each gotten value and causing a slightly less efficient performance. The number of collisions for the quadratic also appears to be slowly incrementing as size goes up as should be expected while the chainProb maintains a constant increase. In regards to timing we see this the real discrepancy appear. As We added values in a linear fashion the chain pro attained a significantly more efficient graph as we were constantly adding to the end of hashTable, thus resulting in a constant increase in size and minimal rehashing. The quadratic however can be seen to take several huge jumps as the rehash and growth take place thus appearing to be less efficient. I think the way in which we added unfairly gave the chainProb an advantage however being able to see the consequence really exemplified the advantage of using a linked list as the backing vs a string array.





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.)

I believe the cost or big-O of both the good and mediocre functions are both big-O (C) as the time cost for each one increases linearly with the size of the list thus indicating the cost per size grows linearly with the size of the list. The good one of course more closely resembles O(c) as it results in less collisions than the other two. The mediocre is slightly higher as its performance is only slightly affected by these extra collisions. The bad functor however appear to attain O(n) as it growing linearly with the list as the other two but because of the erroneous amount of collisions this time is multiplied another time by n thus resulting in the o(n^2) appearance above even though it is not o(n^2) but o(n). I can say that these graphs do indicate what I was initially expecting and confirmed my initial assumptions of what would happen.

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

The greater the load factor the more full the table becomes and the higher the chance a collision will occur simply because most of the available slots are taken. The lower the load factor the more frequently it will rehash and grow which although lowering collisions also lowers performance as the number of times it has to rehash all values increases. When using 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. The logic is similar when we use 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.

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. All you have to do is to reference the same index value of the item you with to delete on the linked list and then delete the specified value there.

```
entries[hashCode(item)].remove(item))
```

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.

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 Any Type)? If so, what changes would you make?

I believe you would be able to implement a generic class for the hash Table. This would be accomplished initially by changing each method declaration to that of a generic one replacing the old string it currently has. Then we would have to create various ways for determining the index of an element depending on what it is. To save time I would assign each element being added the same value as seen in the bad function above. :3 But If I wanted something more functional I would create various methods to determine the index value depending on what type of element was being added.

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

Approximately 15 hours

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.

Quad Table class assignment10.GoodHashFunctor

Column1	Column2
1000	16.9933
2000	26.81985
3000	34.7819
4000	41.405425
5000	47.27892
6000	52.93835
7000	57.9807429
8000	63.181975
9000	68.0934
10000	73.15995

Chain Table class assignment10.GoodHashFunctor

Column1	Column2
1000	4.2325
2000	8.36195
3000	12.4932333
4000	16.646925
5000	20.70236
6000	24.912
7000	28.9633286
8000	33.279375
9000	37.2873111
10000	41.59448

Quad Table class assignment10.GoodHashFunctor

1000	1662589
2000	4297399
3000	6272166
4000	1.36E+07
5000	1.68E+07
6000	2.06E+07
7000	4.19E+07
8000	4.67E+07
9000	5.28E+07
10000	5.94E+07

Chain Table class assignment10.GoodHashFunctor

1000	184832
2000	503831
3000	1070777
4000	1625074
5000	2413747
6000	3116977
7000	4313176
8000	5282923
9000	6883611
10000	8433749

timing for each method on contains

QB Table class assignment10.BadHashFunctor

1000	9968504
2000	3.05E+07
3000	6.73E+07
4000	1.18E+08
5000	1.96E+08
6000	2.70E+08
7000	3.83E+08
8000	5.61E+08
9000	6.53E+08
10000	8.85E+08

QB Table class
assignment10.MediocreHashFunctor

1000	1770789

2000	3751217
3000	1.06E+07
4000	1.94E+07
5000	3.26E+07
6000	4.51E+07
7000	6.16E+07
8000	7.50E+07
9000	1.05E+08
10000	1.26E+08

QB Table class assignment10.GoodHashFunctor

1000	923417
2000	888248
3000	1701155
4000	2503964
5000	3135733
6000	4746673
7000	6915228
8000	8675726
9000	1.10E+07
10000	9363664

QB Table class assignment10.BadHashFunctor

1000	499.5
2000	999.5
3000	1499.5
4000	1999.5
5000	2499.5
6000	2999.5
7000	3499.5
8000	3999.5
9000	4499.5
10000	4999.5

QB Table class
assignment10.MediocreHashFunctor

1000	71.983
2000	139.2921

3000	224.139
4000	281.3718
5000	361.597
6000	445.1804
7000	486.4051
8000	564.1059
9000	642.1102
10000	724.3552

QB Table class assignment10.GoodHashFunctor

1000	17.0807
2000	26.8158
3000	34.76723
4000	41.44463
5000	47.2167
6000	52.76803
7000	57.87707
8000	63.14193
9000	68.07111
10000	73.13698