Kameron Service
U0963620

Assignment10 Analysis

1. The load factor for quadratic probing is essentially how full your backing array is. It is the amount of items in the array divided by the size of the array. The closer the load factor is to one, the fuller the array is and also, the more collisions you will have. Having a load factor near zero will result in almost no collisions, but will result in a lot of wasted space. So if you're trying to optimize time, keep the load factor very small, but if you're wanting to optimize space, keep the load factor near one. For this assignment we wouldn't allow the load factor to go above .5.

   The load factor for separate chaining also essentially how full the array is, but in a different sense. It is the average amount of items in each linked list in the array. So the more items, the higher the load factor. I ran an experiment to test what the best load factor would be. I created an array list of 100,000 random strings and timed how long it would take to add them all to the hash table. To my surprise, the smaller the load value was, the faster it was able to add. I was surprised by this because the smaller the load factor, the more times you would have to rehash. I figured rehashing so many times would hurt the performance, but a load factor of one turned out to be the best.

2. The bad hashing function that I chose was to simply take the length of the string. This would perform badly because there are many words that have the same length so there would be many collisions. This is definitely my slowest hash function.

3. For the mediocre hash function, I decided just to return the integer value of the first character. This worked much better than the bad hash function because there are not as many words with the same first letter than there are words of the same size. It isn't great however, because words like 'ha' and 'happy' would hash to the same value.

4. For the good hash function, I decided to sum the characters of the string, but instead of only summing them, I multiplied each character by 31 and summed that. I found online that this would result in a good hash function so I tried it. I did conduct an experiment to see if 31 would be the best number to multiply. I started multiplying them all by 1 then I tested 10, 20, 30, etc. up to 100 and it turned out that 31 did in fact give me the best result.

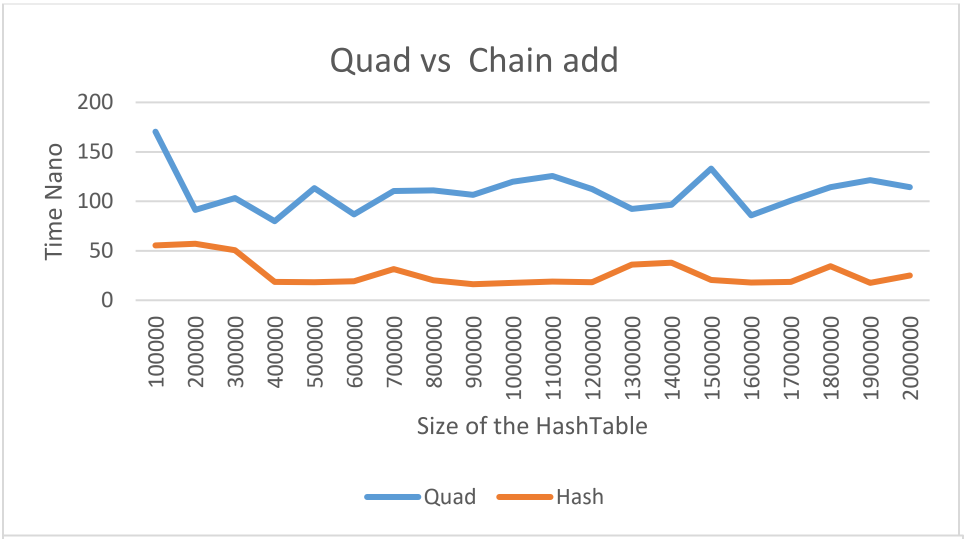
5. To test each of the hash function, I timed adding the word "Kameron" to the hash table of various sizes. I started having the hash table at size 100,000 and went up to 2,000,000. I did this experiment and recorded the results of all 3 functions. The results were exactly as I expected. The bad hash function was terrible and took a very long time, the mediocre was much better but not great, and the good function was much faster. I wasn't able to graph these because each plot gave me a horizontal line. They

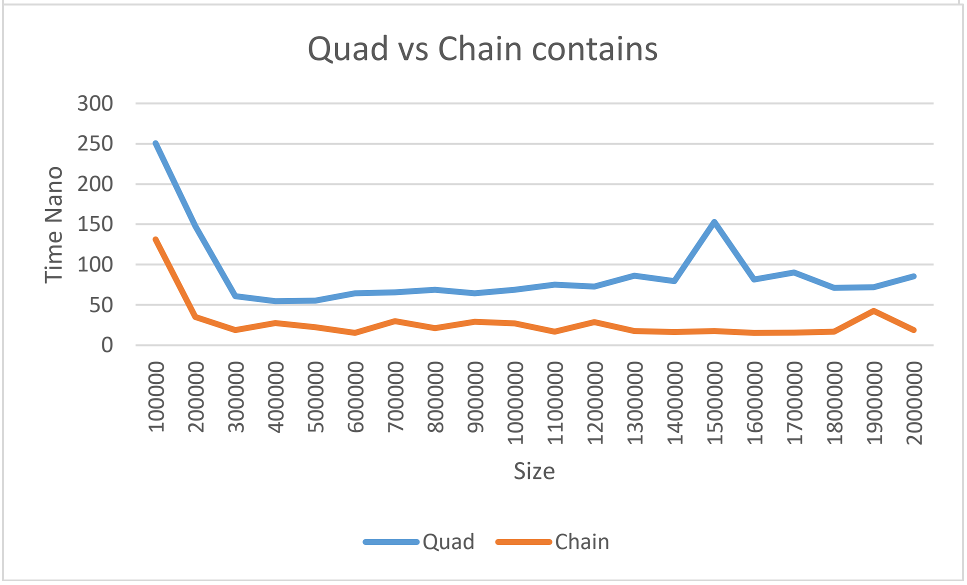
were all technically constant time because they didn't change regarding to N, but the bad and mediocre took much longer that the good. I also recorded how many collisions occurred at each size for each function, but the results were so skewed I couldn't even make a graph that looked good.

Here are the times of the experiment:

Size	Good	Mediocre	Bad
100000	170.374	390.45	65893
200000	91.343	350.516	71326
300000	103.36	402.661	70162
400000	79.912	415.26	69526
500000	113.285	350.15	67148
600000	86.829	415.56	65623
700000	110.266	385.56	68532
800000	111.181	290.16	62356
900000	106.669	405.15	74125
1000000	119.888	477.89	65956
1100000	125.594	432.16	62536
1200000	112.371	315.156	65859
1300000	92.256	502.19	64956
1400000	96.438	426.62	65214
1500000	133.091	375.26	71402
1600000	85.938	305.82	70253
1700000	100.635	415.16	68129
1800000	114.171	550.16	71026
1900000	121.381	390.29	70012
2000000	114.175	465.95	69662

6. To test my two hash tables, I decided to test the add function and the contains function of each. I created a quad hash table and added items from 100,000 to 2,000,000 going by 100,000. Then I tested how long it would take to add one word to the table. I then did the same thing except I used the chain hash table. After I finished with the add method, I did everything the same except timed how long it would take to see if each table contained a word. Both tables performed these method with a constant time performance. However, the hash table that used chaining was a little faster with both methods. I recorded these times and made these plots:





7. For the bad hash and mediocre hash functions, having n be the string length gives a complexity of O(c) because neither hash function changes with a different length string. The good hash function has an O(N) complexity because it goes through each string character by character to get the hash.

8. I explained how the load factor affects performance in question 2.

9. For the quad hash table there would be 2 ways to implement a remove. One method would be to use something similar in the contains method to find the string, and then remove that item from the array and rehash the whole array. Another method would be to create a separate Boolean array that would keep track if an item at a certain index was deleted or not.

For the chain hash table. I would use the string's hash code to find the linked list that contained the string, and use javas remove method to remove the item from the linked list.

10. It would be possible to make this generic. You would have to create different hash functions though, because the hash functions are specific for strings. You would also have to implement E instead of a string in many places, but it wouldn't be too hard.

11. I spent around 10 hours.