Landon Crowther
U0926601
4/7/17

HW 10 Analysis – Hashing:

Testing

To ensure that our implementations worked properly, we performed the following tests:

Crack Class

This class was a bit harder to test. We tested the file readers by loading the given sample files, and manually assuring that everything was read properly.

For the brute force tests, we added the hashed version simple strings ("a", "b", "cd") to a HashSet. Then, we called the brute force attack method, and ensured that all three passwords were "cracked"

This test was also repeated using values from the "dictionary" file

To test the MD5 hash algorithm, we loaded a few words from the "a_few_words" files, hashed them, and ensured that their hash was equivalent to the corresponding "a_few_hashes" files.

HashMap Class:

To test the HashTables, we created a Junit test file, where each test should work on all three implementations. Some of the tests go as follows:

insert – to test this method, we added pairs of keys and values, and assured that the size increased properly

find – to test find, we first tried to find an element in an empty HashMap, and confirmed that the result was null. Then, we inserted a pair. After inserting, we ensured that the size was equivalent to 1, then, we found the key and assured that the find method found the key, and returned the value.

clear – to test clear, we added three elements to the HashMap, asserted that the size was equal to 3. Then, we cleared the HashMap. After clearing, we tested for size 0, and tested that null was returned when finding a previously entered element.

capacity – to test capacity for the chain implementation, we ensured that the capacity was equivalent to the original size. For the other implementations, we tested that the capacity was equivalent to the first prime number proceeding the input size. Even after adding 1-2 elements, the capacity remained the same (the size never had to change because not enough elements were added).

size – to test size, we first checked for size 0. Then after adding an element, we ensured the size was 1.

print_stats – for this method, we added four values, and then called the print_stats method. We then assured that the returned ArrayList had the correct stats for each implementation (chain, linear, quadratic).

resize – we first tested that the capacity was the original input size (or next prime for linear/quadratic). Then, added a few elements. The resize method was then called, using a number greater than the original capacity. After calling the resize method, we tested the find method to ensure that the elements were still there, and then asserted that the capacity had indeed changed.

Disable Resize at size 79:

The following results were obtained after following the procedure indicated by the analysis question. Note, these results are in the following order: linear, quadratic, chaining.

Landon Crowther
U0926601
4/7/17

```
------------ Hash Table Info ------------
  Average collisions: 3.5822784810126582
  Average Hash Function Time: 641.6455696202531
  Average Insertion Time: 3859.0506329113923
  Average Find Time: NaN
 Percent filled : 100.0
  Size of Table  : 79
  Elements       : 79
----------------------------------------

------------ Hash Table Info ------------
  Average collisions: 588.6455696202531
  Average Hash Function Time: 138.50632911392404
  Average Insertion Time: 99052.12658227848
  Average Find Time: NaN
 Percent filled : 100.0
  Size of Table  : 79
  Elements       : 79
----------------------------------------

------------ Hash Table Info ------------
  Average collisions: 0.5569620253164557
  Average Hash Function Time: 276.9746835443038
  Average Insertion Time: 8567.518987341773
  Average Find Time: NaN
 Percent filled : 100.0
  Size of Table  : 79
  Elements       : 79
----------------------------------------
```

These results (although they changed slightly each time the method was ran) make sense. Due to the small amount of values being added, the linear implementation ended up having the best insertion time. This was because when a bucket was full, finding the next empty bucket linearly was relatively quick. Because the quadratic implementation skipped around so much, it took a while to find an empty bucket as the data structure got more and more full. The chaining insertion time wasn't far behind the linear, but the quadratic implementation time was much greater than the other two.

As far as collisions go, the chain implementation by far had the fewest collisions. The reason that the chain implementation has any collisions is because before an element was added, our method ensures that the key doesn't already exist, which causes a few collisions. Because 79 was a relatively small number, the linear version had much fewer collisions than the quadratic. This is because, like mentioned above, as the HashMap became closer to "full," the program had to jump around a lot more until an empty bucket could be found.

Overall this was a decent test of performance. Another metric that would be useful would be testing with a much larger size (say 20,000+). If comparing the quadratic vs, the linear, the quadratic would (initially) perform better at finding empty buckets. However, as the the HashTable started filling up, the same problem of "bouncing around" would continue to occur.

Linear vs. Quadratic vs. Chaining

Although there is no "best" implementation, each has its own perks. For relatively small data structures, the linear implementation may work better than the quadratic. This is because checking each element in a relatively small data structure doesn't take very much time. In a larger data structure, the quadratic implantation would be much more effective. If collisions are the biggest concern, the chain structure by far is the best way to go. The chain only had 0.5 collisions on average, compared to the 3.5 or 588 from the linear and quadratic.

Remove

Although a remove function wasn't required, it wouldn't be too hard to implement. It would be easiest

to remove from the chain implementation, as the bucket containing the pair would be guaranteed. You would simply remove the pair from the LinkedList.
The quadratic or linear implementations would be a bit more difficult. The most guaranteed (although not efficient) method would be to remove the pair, and then rebuild the HashMap. This would guarantee that everything is find-able in the future, but wouldn't be time/memory efficient. An alternative would be to make an alternative find method, that returns the location of the where the element exists. Then simply clear this element. That way the structure of the rest of the elements wouldn't get messed up.

Dependency:
For the most part, the hash tables aren't very dependent on the array size. The only time the array size makes a difference is in the insert function. The hash value is modded with the capacity, and that determines which index that will be added. If each hash value is unique, then a large capacity would theoretically decrease the likelihood for collisions. The hash function is extremely important to table efficiency. If the hash function gives relatively similar values, the efficiency will decrease, as the hashes will all try to go into the same bucket. On the other hand, if the hash function makes a wide variety of hashes, the efficiency (in theory, given a large enough capacity) will increase, as there will be less collisions. Personally, I believe a larger capacity is more important than a good hash function. This is because if a good hash function were used on a small capacity array, there is a high possibility of repeated indices after the mod function is used. On the other hand, if the quadratic implementation was used on a large capacity array, although the initial indices may overlap, finding an empty bucket will go relatively quickly.

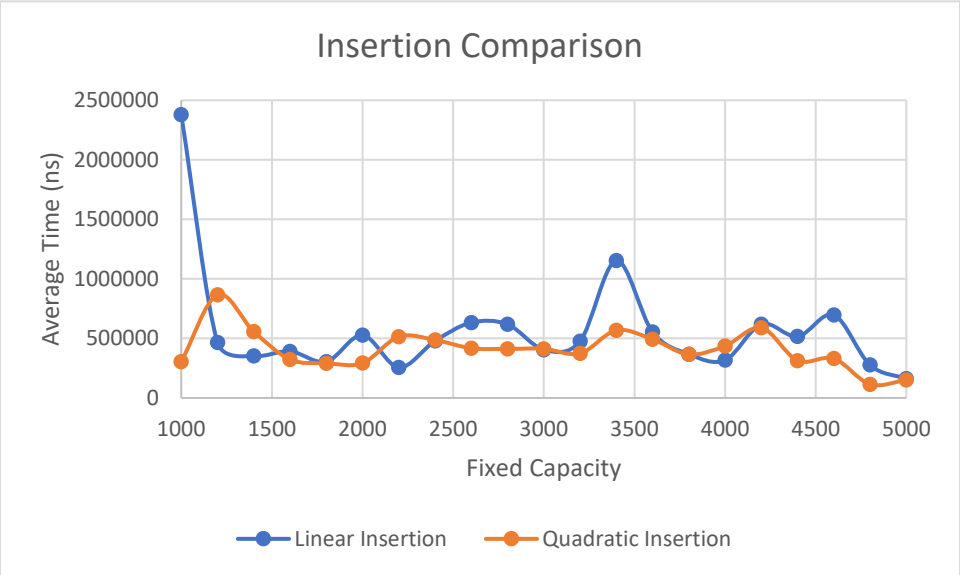
Other Thoughts:
This project didn't take too much time. A good amount of time was made during the linear implementation. However, once this was created, it took less than 2 minutes to create the quadratic implementation. As far as the chaining goes, a large amount of the code was identical to the linear implantation. The only major differences were in the insert and find method, where instead of cycling through the array, adding/searching through a list was substituted. Total time spent on the code portion was probably 7 hours or so. This was about the same amount of time I anticipated, as my partner and I both were very efficient with our time.

Brute Force:
The brute force algorithms, although not very efficient, do end up "cracking" the password. I say that they are not very efficient because every possibility must be checked; however, I can't think of any "better" implementations that would guarantee success. Implementing a recursive method turned out to be very effective, as it would be nearly impossible to write the correct number of for loops without prior knowledge of the length of the password. It was difficult at first using the StringBuilder and figuring out the best way to write the method, but once we figured it out it worked great.
Without a doubt, the HashMap implementation was more effective than the ArrayList implementation. Although no official timing tests were conducted, I would hypothesize that many possible passwords would take much longer on the ArrayList as opposed to the HashMap.

Landon Crowther
U0926601
4/7/17

Timing Analysis:

The following figures were produced by calling insert and find on the list of names at given capacities, as described in the problem.

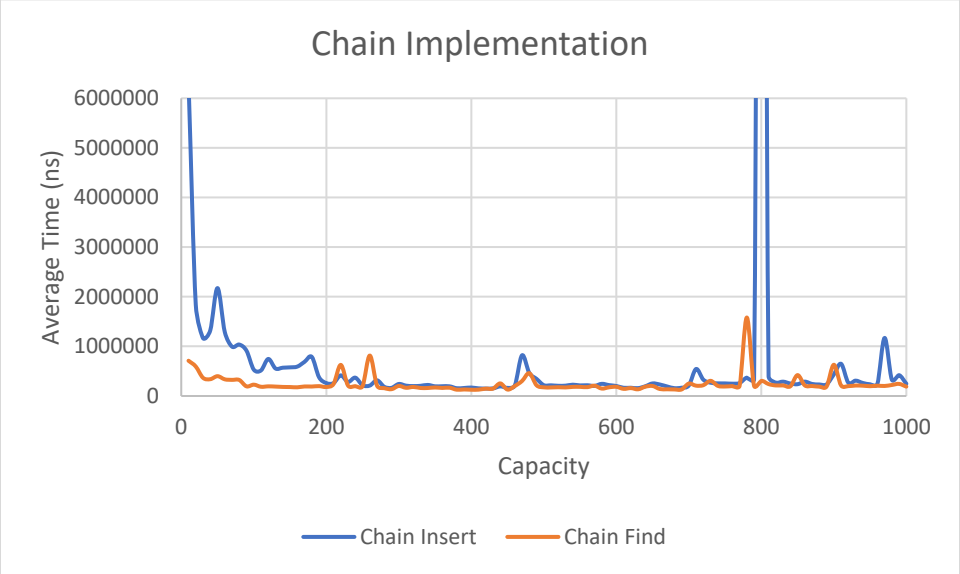


For insertion, the linear and quadratic were similar as capacity changed, but quadratic seemed to be quicker overall. This behavior is likely due to a poor hash algorithm, as when the capacity is small, it seems like there are a lot of collisions, *especially* for the linear version. My guess is that the first 1000 elements or so gave the same index after being hashed, which caused a lot of collisions. As the capacity increased, the speed increased, and fluctuated slightly. The speed seemed to be very inconsistent, as there are random spikes in time around 3500 elements. Overall, the average time decreased with an increase in capacity, which would be expected.

For find, the quadratic and linear were almost the same. The only main differences are a huge spike in time for the linear implementation at low capacity. My guess would be that the low capacity caused a lot of collisions. Further experimentation with a hash algorithm may provide better results.

Landon Crowther
U0926601
4/7/17

The following figure was produced using the assignment specifications for the chain implementation:



note – to make the data more visible, two of the spikes were omitted, as they skewed the rest of the data.

As seen by the data, both insert and find were relatively consistent, and didn't change much with capacity. (Disregarding the spike at low capacity and 800 capacity). As far as the spikes go, the initial spike makes sense, as there will be more collisions with a small array. However, I am unsure why there is a spike around 800 elements. My best guess would be some sort of coincidence with the hash algorithm. Although the time was relatively consistent with capacity (which is good), it wasn't *faster* than the non-chain versions above. Perhaps this is because searching a LinkedList can be time intensive.

Final Note

The figures that are required in the assignment description are in the "Documents" folder, however they are saved as .pdf files, rather than .ps files. I am unsure what a .ps file is, or how to generate it.