Assignment 10 – Analysis

Ben Figlin (u1115949)

Nov 9, 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 λ for the quadratic probing is the fraction of the table that is full.
   For the separate chaining, it is the average chain length.
   The quadratic probing had a good performance for $\lambda < 0.5$, because the average amount of examined cells will be just 2 or less.
   The separate chaining strategy performance is not as severely affected by the λ value, and it is always proportional to λ. In general, $\lambda>1$ means that there is more than one immediate item in each slot on average, so performance begins to decrease due to the necessary scanning of the linked list chains, so as long as $\lambda<=1$, the performance should be optimal on average.

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).**

   ```
   return item.length();
   ```

   The hashing function simply returns the length of the string.
   It is very bad, because strings with similar lengths will create collisions, even if the string content is completely different. Another concern is that it will not spread out evenly in cases where the length of the strings is in some small range (smaller than the size of the array, so they will all be attempted to be placed at the beginning of 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).**

   ```
   int hash = 0;
   for (int i = 0; i < item.length(); i++) {
       hash += item.charAt(i);
   }
   return hash;
   ```

   This function simply adds the values of the characters together.
   In this case, it will calculate different values for different strings, even if they have the same length. But for strings with similar letters and different ordering of the letters, the result will be the same and a collision will happen. Also, the resulting value will be much

smaller for small strings, so it is not good when there are a lot of small strings in large tables.

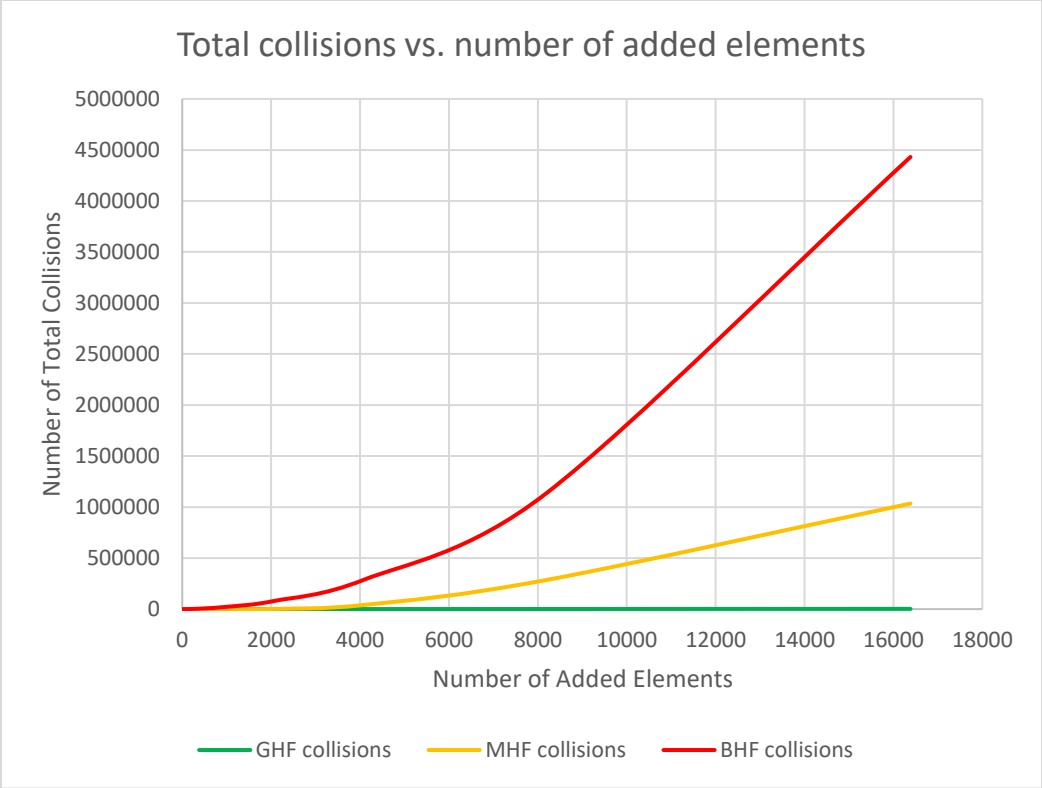
4. **Give and explain the hashing function you used for GoodHashFunctor. Be sure to discuss why you expected it to perform well (i.e., result in few or no collisions).**

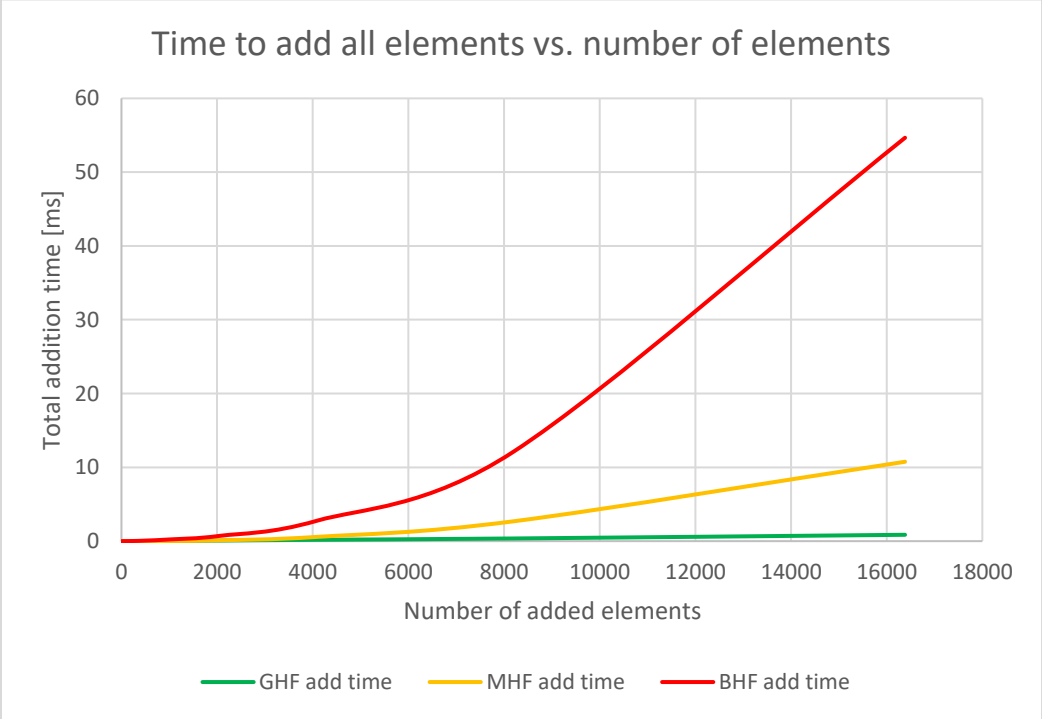
```
int hash = 37;

for (int i = 0; i < item.length(); i++) {
    hash = hash*53 + item.charAt(i);
}

return hash > 0 ? hash : -hash;
```

In this function, we start with an initial value of some prime number, and with each character of the string, we will multiply the hash result by another prime number and add the character value to that, eventually creating a very large number with no repeating patterns due to the use of prime numbers. Finally, we return a positive number, because this function can originally return negative values due to the very large numbers and potential overflow of the integer value.

This function returns numbers without any clear patterns, it is sensitive to character ordering, so will produce different results for ‘cat’ and ‘act’, and string length does not affect the result as much as with the other functions, because the numbers become very large very fast, even with small strings.

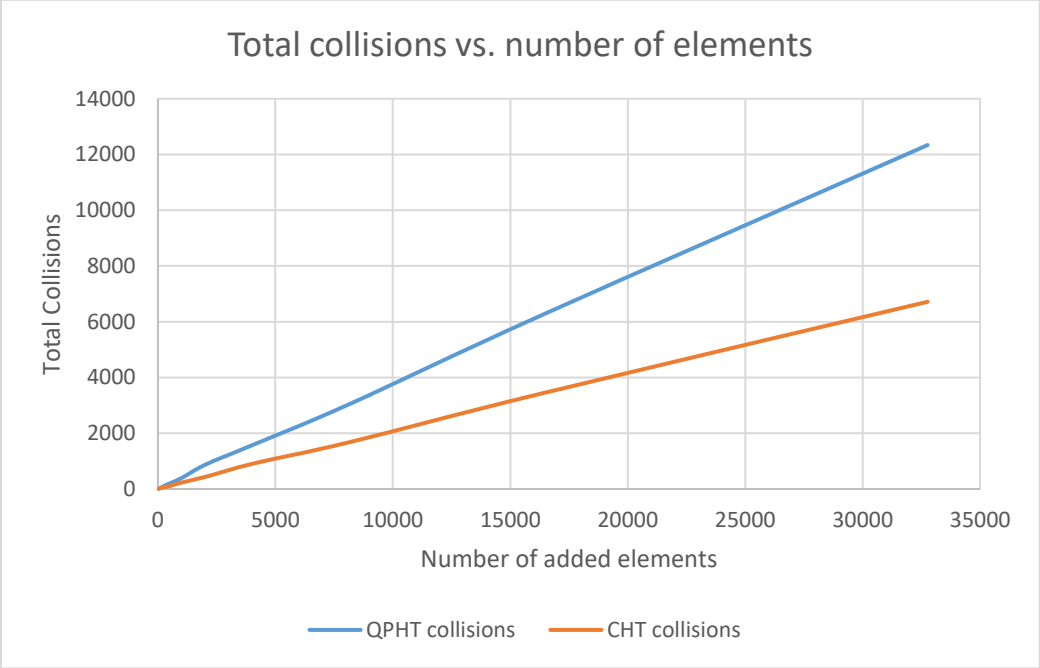
5. **Design and conduct an experiment to assess the quality and efficiency of each of your three hash functions. Carefully describe your experiment, so that anyone reading this document could replicate your results. Plot the results of your experiment. Since the organization of your plot(s) is not specified here, the labels and titles of your plot(s), as well as, your interpretation of the plots is critical.**
    1. For each time the hash table needs to perform probing, we count it as collision.
    2. We create a hash table that is large enough (size=N*4 so $\lambda <= 0.25$) so we don’t need to resize the table during the addition.
    3. For each type of hash functor, we add N random items (same random items for each of the functors) to the hash table. We measure the time it took to add all elements.
    4. We read the number of collisions that occurred during the addition of all elements.
    5. We write a row with the two measured parameters for each functor, together with the number of items (N).
    6. Repeat step 2, with N = N*2. Repeat until there are enough data points.
    7. Plot the results of all of the collisions on one chart, and the timing results on another chart.

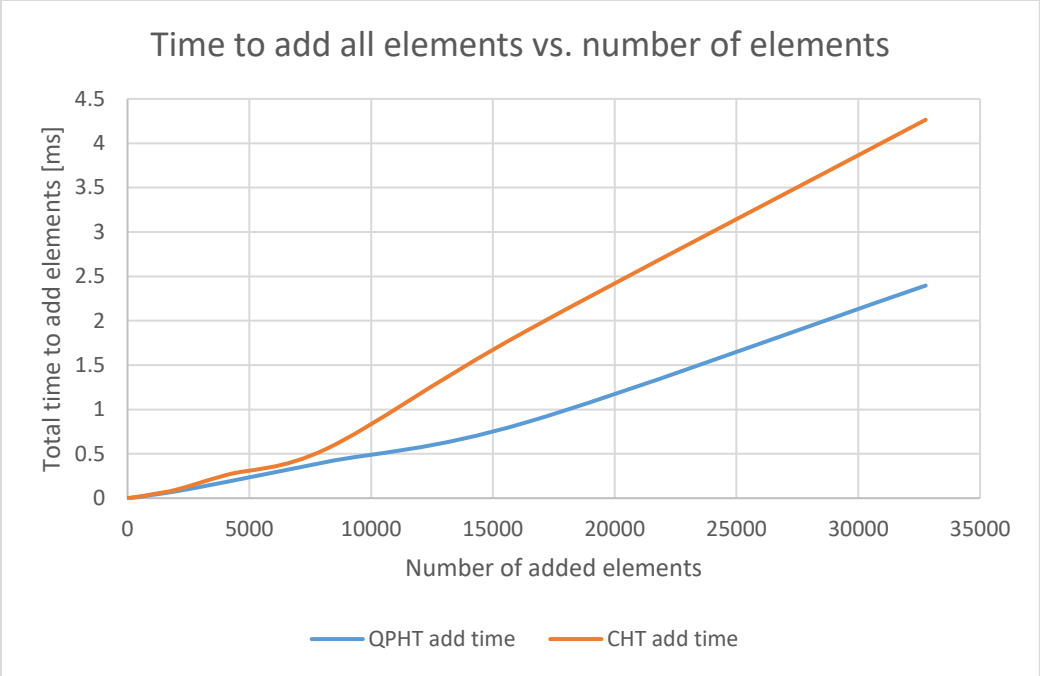




We can see how the GHF is so much better than the others.

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.**
    1. For each time the hash table needs to perform probing, we count it as collision. For the chaining hash table, we count collisions according to the index of the added item in the linked list (if it's the first item, the index is 0, so no collisions happened)
    2. For each type of hash table, we create a hash table with the size of 2*N (to avoid resizing in the QP hash table).
    3. For each hash table, we add N random items (same random items for both tables), and we measure the time took to add those items.
    4. We read the number of occurred collisions after adding the items.
    5. We write a row with the two measured parameters for each table, together with the number of items (N).
    6. Repeat step 2, with N = N*2, until there are enough data points
    7. Plot the results of all of the collisions on one chart, and the timing results on another chart.





We can see that the difference is not major between the two tables. The QP hash table is slightly faster, even though it has more collisions.
But we must not forget that the tables are underutilized and we are only comparing the raw functionality of the add method:

a. The QPHT does not perform any resizing (which takes a lot of time), because we assume that in real life the resizing will be rare.
b. The CHT has a λ<=0.5, which is below the point of ~1 where the linked lists start to introduce delays, so we can have a fair comparison between same size tables.

8. **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.)**

The BHF complexity is O(c) because it only uses the .length() method, which has a constant complexity (Java stores and tracks the length value in a variable).

The MHF and GHF are both iterating through the string characters and performing N operations, so this makes the complexity of them both be O(N), while GHF is slightly slower because with each iteration it does some extra calculations.

Each of the hash functions performed exactly as I expected! GHF showed excellent results, BHF showed terrible results and MHF showed something in the middle between the two.

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

The load factor affects the QP hash table severely when it starts climbing towards 1, as the number of probing increases rapidly, in an exponential fashion.
On the other hand, as the load factor of the CHT increases, the performance decreases linearly in a proportional way, and not exponentially.

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

For the quadratic probing, each cell will now contain a node type that has data and a flag that indicates if this cell is used or not. We can then simply go to the hashed index of the item to be removed just like with the contains method (and do quadratic probing if needed, until we find the item), and simply set the cell flag of the found item to unused. We stop probing if the item to be removed is found, or if we reach a null cell (null cells are cells that were never used, and will always exist as we still maintain $\lambda<0.5$). We will not change the size variable when marking the flag as unused or used, but only when adding items to null cells, so we can always keep >50% of null cells for the λ calculation.
For the separate chaining, we simply find the item just like with the contains method, and remove it from the linked list in the usual way.

11. 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?

Yes, it is possible. We just need to change all the String types to generic types, and to use an appropriate hash functor for each type (because currently our hash calculations are specific to the structure of a string). We can also simply use the type's built-in hashing function, assuming that the types we will use will have this implemented.

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

I spent around 6-8 hours on this assignment.