Tony Diep
4-6-17
CS 2420
Analysis Document

Analysis: Hash Tables

We verified the working implementations of our hash tables by constructing tests for executing the double behavior and probing methods. Additionally, we tested for the capacity and the number of entries of the hash tables. In terms of boundary cases, we tested for the statistics for each hash table, verifying that the statistics make logical sense. Although they were not necessarily exhaustive, they covered the basic cases. They were only a couple of boundary cases for doubling behavior and finding keys in the hash tables that were tested.

When first running the three different hash tables at first, we had a couple of problems: the statistics seemed inaccurate in that the three hash tables contained twice the amount of collisions than expected. However, we immediately took note of that and fixed the statistics so that they produce reasonable and logical results.

For the three hash tables, the conducted experiment seems like a fair test. Out of the three different implementations of the hash tables, the chaining hash table is the most effective type of hash table for large quantities of data because it mitigates the fewest amount of collisions compared to linear and quadratic. The linear probing hash table will contain the most collisions since the probing jumps on every single non-empty bucket. However, for small quantities of inserting elements, quadratic probing would be suitable since quadratic would look at every other bucket as opposed to every bucket.

When implementing the remove function, I suggest by computing a hashing index that corresponds to the “bucket” that contains the key with the desired value to remove. Once we retrieve the desired value to remove, we would reinitialize the Pair object to null, indicating that the desired element is removed. This satisfies that the remove operation should be at O(C) complexity.

Each hash table are to an extend dependent on the size of the array. For any given size of the array, every time a Pair object is created, the hash tables check if they are half full then resizes for that case. Even with an array of size 0, the hash tables regrows to the next prime number, which will be 2 in our case.

Hash tables heavily on a good hash function since they will greatly increase the average collisions as well as the time it takes to perform the hash function. A poor hash function would yield many keys pointing to the same hash which would mean more collisions. The hash function should instead find a way to distribute different hash functions more than the number of times inserting Pair objects.

When first looking at the project, I felt overwhelmed by how much implementation is required. But when working on the assignment with my partner, I was able to understand conceptually how should each hash table works. A lot of pseudocode was given so most of the implementation was essentially derived from them. For example, it was much simpler to implement the brute force attack method using recursion after spotting a couple of logical errors in the code. Me and my partner were able to implement the general idea but we later found little bugs and had to fix them immediately. Nonetheless, we were able to complete this project at a reasonable manner. The amount of time spending on this assignment was certainly a lot lower than I anticipated.

Between the brute force and dictionary attack algorithms for cracking passwords, the dictionary attack deems most effective. The Big O complexity of dictionary attack is O(N), where N is the number of words in our collection of hashes. Each word in the dictionary is converted into a hash string using the MD5 algorithm and is used as the key to find if it exists in the hashes collection. The brute force attack takes a lot longer to compute since every permutation of a word’s length is computed then compared to the hashes collection. However, the permutations are calculated for only lowercase letters, and so we can propose that the brute force will take a lot longer because of covering all possible characters in uppercase and lowercase, numbers, and then miscellaneous characters.

Timing and Graphs

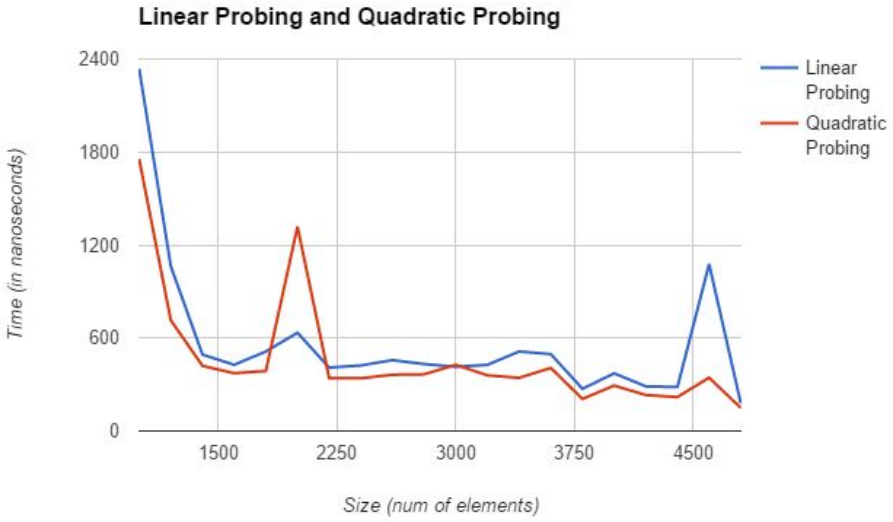

Figure 1: Linear Probing vs. Quadratic Probing

Figure 1 depicts the time between the linear probing and quadratic probing hash table. Around the interval 1500 - 2250 nanoseconds, quadratic probing took a slight long time to find Pair objects. However, after 2250 nanoseconds, the time was shorter compared to linear probing. The graph shown aligns with our hypothesis that quadratic probing is faster since the probing checks every other bin as opposed to every single one.

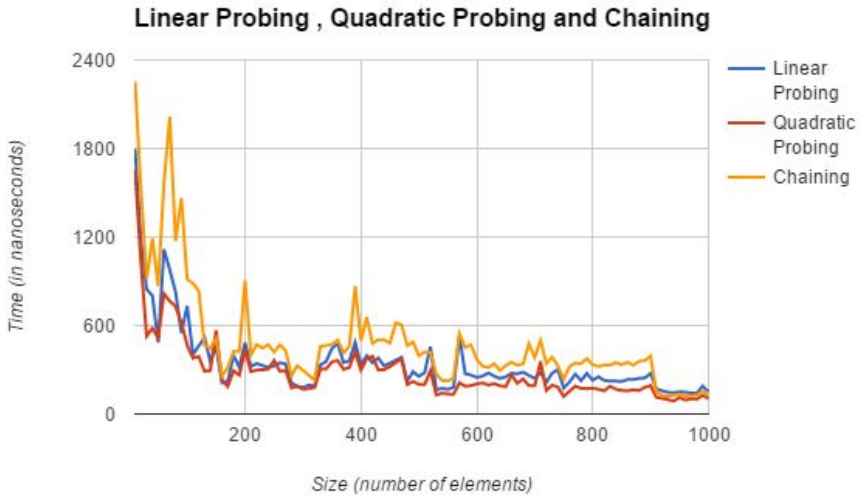

Figure 2: Linear, Quadratic, and Chaining Hash Tables

Figure 2 depicts the time between all three hash tables: linear, quadratic, and chaining to find a key in a Pair object. The chaining hash table took more time to search keys because each bin contains a LinkedList data structure in them, and the cost to search a Pair object's key in each Node of the LinkedList is O(N), where N represents the number of nodes to traverse. Similar to the depiction in Figure 1, quadratic probing took the fastest time to search a key in a Pair object.