Hashing Efficiency

Dr. John Matta
Ryan Bender
CS 340 - 002

Preview:

The purpose of this project is to gain a stronger understanding of hashing and hash tables. Hash tables are a complex data structure that allows data to be stored and retrieved extremely efficiently. The hash tables in the project use linked lists to handle collisions when different data is addressed to the same location. For this project, I inputted the novel *Little Women* into a hash table to store each word individually. Each word was hashed to create the key for the word, and the value in that location is how many times it occurred in the novel. To test out how effective different table sizes are, I tested the hash table in sizes 500, 1000, 2000, 5000, and 10000.

Data and Analysis:

Hash tables are extremely efficient for inserting and searching for data. Since the table is created from an array, any given index can be accessed – both for inserting and searching – in one instruction. For these tables, since they handle collisions through linked lists, they will typically take a few more instructions. The program must search through the linked list in an address until it finds the value it was looking for, so the number of steps for a given word would be how far down the list it is. Having a larger hash table would decrease this time since there would be more unique addresses to store data but also increase the amount of memory an empty table to take up. The average times to insert a word into the table can be found by dividing the sum of total number of steps by the number of occurrences for all words. This equation gives us a good scale of how efficient each table size is. The smallest table averages about 3.1 steps to access each word, while the largest table took 1.1 steps. Both values are extremely low, indicating effective use of the hash table.

The ideal number of steps to access a given word would also improve from a larger table. In the ideal table, each linked list would be the same length in every address in the array. To find out how many steps an ideal table would take, you can divide the number of words in the novel by the size of the array. This result would give you the maximum number of instructions it would take to find a word. For example, the array of size 500 would take 22 steps at most. Since there are 10933 words in the novel, you would divide 10933 by 500. The other tables sizes of 1000, 2000, 5000 and 10000 would take 11, 5.5, 2, and 1.1 steps respectively.

Comparing the results from the hash tables sizes I created in this experiment, the tables for the novel *Little Women* did a good job at creating an efficient table. This novel has 10933 unique words in it, and almost 200000 words in the whole novel. With data this size and using the smallest hash table, an average access time of 3.1 steps is more than acceptable. These results largely surpassed my expectations. I expected the average access times from the different sizes to be much more different due to the large size difference in the tables. I expected the largest table to far outperform the smallest, but it only beat the smallest table by 2 steps. This result is likely due to the frequently used words being used early in the novel, resulting in them being near the front of their linked lists. If they are near the front, the length of the list does not matter because the algorithm stops searching once it finds the value it is looking for. Despite these positive results, a different novel or changing the way collisions are linked could have had a different result. These results will be tested further in this report when changing the method of the linked list.

If new words were added to the end of the linked lists, the expected behavior of the algorithm should behave in the same way. The algorithm searches through the linked lists by traversing the lists until the next pointer is null. When a new word is added, the tail pointer is changed to a new word and that new tail points to null. The next iteration of the search will proceed on the same path but just one step further to the new node until it sees a null pointer.

To determine which words in the linked lists were first in the list, you can compare the number of steps to the occurrences of the word. If the occurrences are the same as the steps, it means that the word in the spot is in the front of the list. You can find out what spot each word is in by dividing the steps by the occurrences and the resulting number shows its position in the linked list for that address.

Discovery:

To explore hash tables more, I tested how changing the linked lists to add new words to the front of the list instead of the back would affect the average access times for an arbitrary index. At first glance, it may not seem like it would affect the result much, but after reviewing the original algorithm, words that appear first in the novel typically were used more frequently than words that had their first appearance later in the novel. This means that every time a frequently used word that appeared early in the novel needed to be accessed, it would be at the end of the linked list, slowing down the access time.

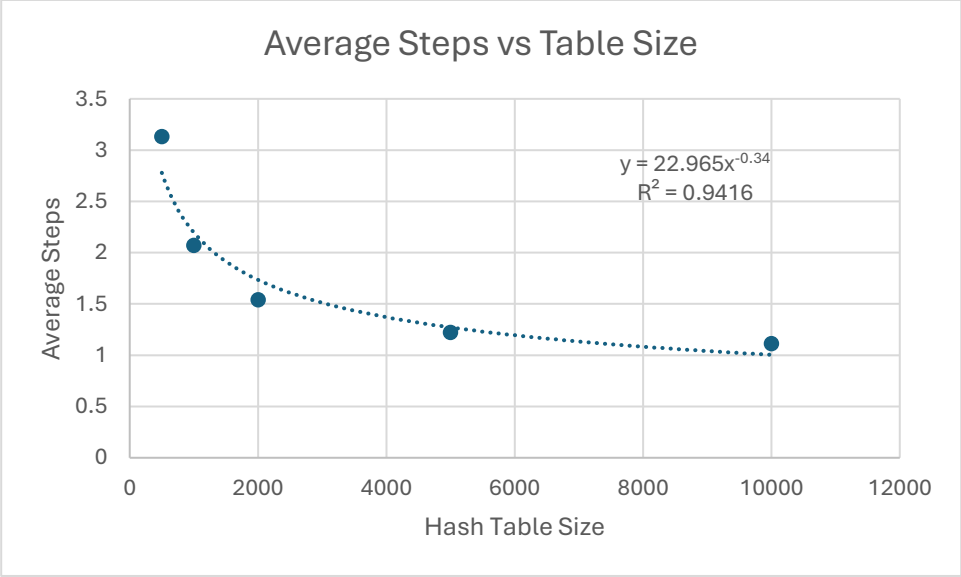
When testing this new collision handling method, the access times were much slower than the original linked list method. The average time of the table of size 500 was much slower at an average of 13.9 steps, while the other sizes were nearly double the early tests at 7.3, 4.1, 2.4, and 1.7 steps. This increase can largely affect the runtime of the program when the data is much larger than the size of this novel. In the best scenario, the

number of steps is the same as the original hash table, but actual outcomes vary much more. This variation makes sense, and I am not surprised by the results of it. I expected it to take longer when you change the way new words are added. For example, if a word is added in the front of a linked list, but it only appears once, it must be iterated through every time data in the linked list is being searched for, but it is rarely the target data. This slows down the search by causing rarely used data to have to be looked at during every iteration through the list. This practice is impractical and inefficient, and I would not recommend it be implemented when creating a hash table.

For this method of chaining, adding words onto the end of the linked list should still work if it is entered in correctly. The program should still search through the list until it either finds the value it is looking for or finds a null pointer on the end of the list. Additionally, the first words entered in the linked list can be found by navigating to the end and the tail value is the first word entered in the list. Its next pointer should be null, verifying that it is the tail. It does require you to look through the entire list to find that value, so it is much slower than the first method.

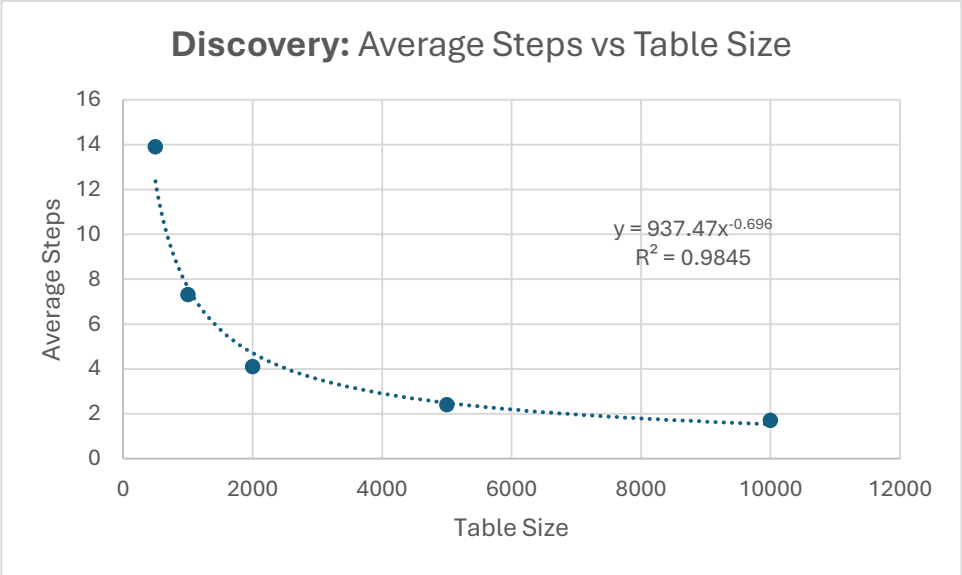
Comparing Data:

Upon examining the relationship between average number of steps and the size of the hash table, the data shows a strong correlation. Using the original chaining method, the number of steps decreased as the size of the array increased. This follows my belief that a larger array will have shorter linked list, resulting in faster insertion and search times. Below is the graph for this data.



Additionally, the experimental method of adding new words to the front of the linked list showed a similar trend, despite taking more steps. Both graphs show a strong

correlation, and the data I discussed above backs up that a larger array will decrease the average number of steps, even if the linked list is not handled optimally.



Summary:

Hash tables are an effective way to store data most of the time. For random data sets, hash tables should be able to handle large amounts of data effectively and accurately. Given the data from hashing *Little Women*, the hashing function and collision handling used in this project was effective. I learned that words used frequently often have their first appearance early on in novels, and the data from this book backs that claim up. I believe that hashing has good use and works well in real-life situations. For example, if a bus system must quickly access a rider's ID number, hash tables have quick lookup speeds and can keep up with the steady flow of passengers.

The success of a hashing function depends on many variables, but the largest of those is how well its hash spreads out the data. If the hash produces the same value for every input, the resulting table would just be one long linked list, which would lose its effectiveness. Ultimately, a good hash function depends on the data it is storing. The programmer creating the function needs to have an idea of what the data will look like, so the hash will be able to spread out the data evenly. An ideal hash table would contain a linked list of even lengths for every index in the array. The more off balance the list lengths are, the less effective the table will be at insertions and searches, which causes the primary purpose of a hash table to fail.