CSCI 2270: Data Structures Final Project

Authors: Madisen Purifoy-Frie and Trevor Green

Purpose

The purpose of this assignment is to compare the performance characteristics of Hash Tables using different hash functions, load factors, and collision resolution mechanisms. The ultimate goal of this assignment is to find the most time-efficient hash function and collision resolution mechanism combination for the operations insert lookup and delete. We are using four different collision resolution mechanisms including; cuckoo hashing, inserting into a linked list, inserting into a binary search tree, and linear probing. We perform 100 inserts, lookup and delete operations using two different hash functions, and each of the four collision resolution mechanisms. Then we take the mean and standard deviation runtime of each operation at four different load factors. The pairing that produces the lowest mean and standard deviation, at each respective load factor is the most time-efficient function.

Implementation Theory

1. Linear probe. The linear probe first assigns all the values in the table to -1 within the constructor. To insert a value, it first checks the location given by the hash function. If that location is full it loops through the table to find the next open location. If it reaches the end of the table, it wraps around to the beginning of the table.
2. Linked List. The Linked List first assigns all the values in the table to -1 within the constructor. To insert a value is first checks the location given by the hash function. If that location is full, it adds the value to the end of a linked list.
3. Binary search tree. the BST first assigns all the values in the table to -1 within the constructor. To insert a value is first checks the location given by the hash function. If that location is full, it adds the value to a binary search tree.
4. Cuckoo Hash. The Cuckoo Hash first assigns all the values in table 1 and table 2 to -1 within the constructor. To insert a value, it runs the value through the first hash function, if the location obtained by the first value is full, it runs the value through the second hash function. If both these locations are full, it replaces the value in location 1 with the value that is to being inserted. Then, it places the value that was in location 1 into Table 2 using the second hash function. It stores the replaced value in a temp variable. If that temp is -1 we are done, if not, it repeats the process only in the opposite direction. It recursively calls this function until an empty spot in the array is found. In the case that the original hash location and the hash location of the second temp variable are the same, the program goes into an infinite loop. If this happens, we double the table size, create a new array with the larger table size, then hash all the numbers in the original array into the new array. Finally, we set the old array equal to the new array. We continue doubling the array until all the values can be inserted without entering an infinite loop.

Analysis

The first collision resolution to be implemented in the code was the Linear Probing. This resolution was arguably the easiest to code as it utilized the already existing hash table which has a struct of a vector. Calling the linear probing collision resolution on data set A produced promising results. The collision resolution was used 5305 times for data set A and 5317 times for data set B throughout and the times it took to insert, search, and delete for the different load factors is shown below in Figure 1.

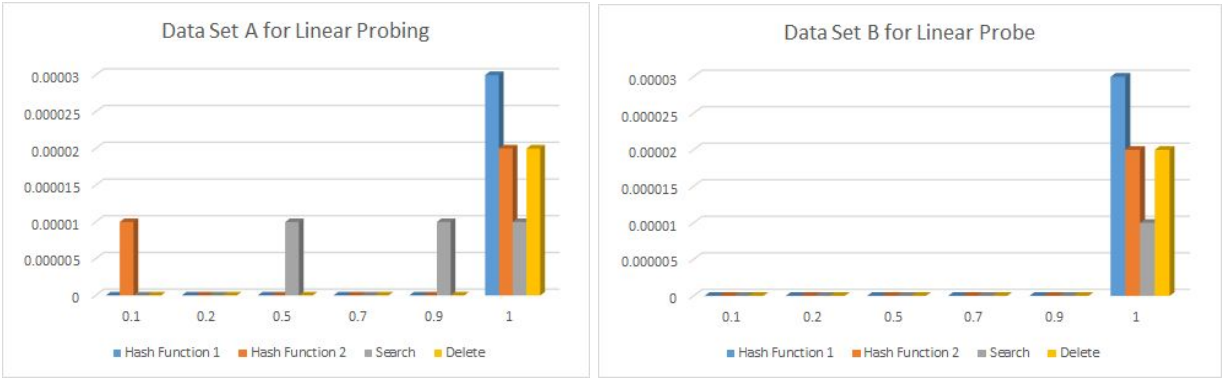

Figure 1: Data Set A and B for Linear Probing collision resolution

The second collision resolution to be implemented in the code is Linked List. When this collision resolution was called on both data sets, the code utilized it a total of 4171 times for data set A and 4233 for data set B as it inserted numbers to a load factor of 1 as seen below in Figure 2.

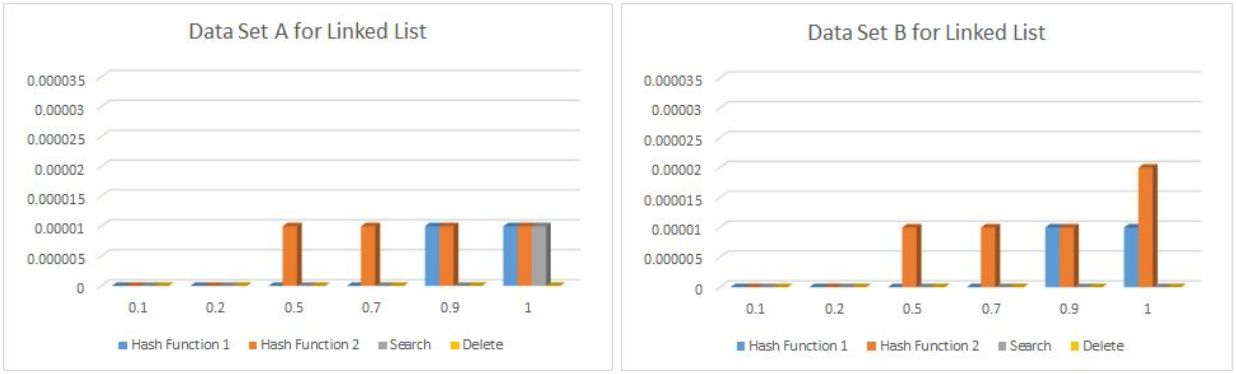

Figure 2: Data Set A and B for Linked List collision resolution

The third collision resolution to be implemented in the code was the Binary Search tree. We predicted that it would either the binary search tree or the linear probe collision resolution, but the time results for the binary search tree proved it was a very effective method.

Figure 3: Data Set A and B for Binary Search Tree collision resolution

The fourth and final resolution to be implemented in the code was the Cuckoo Hash which uses both hash functions to insert the numbers. This method was very complicated to code. We were unable to get it to compile without any errors for all of the load factors; however, the figure below includes the load factors that the code was able to run through before seg faulting.

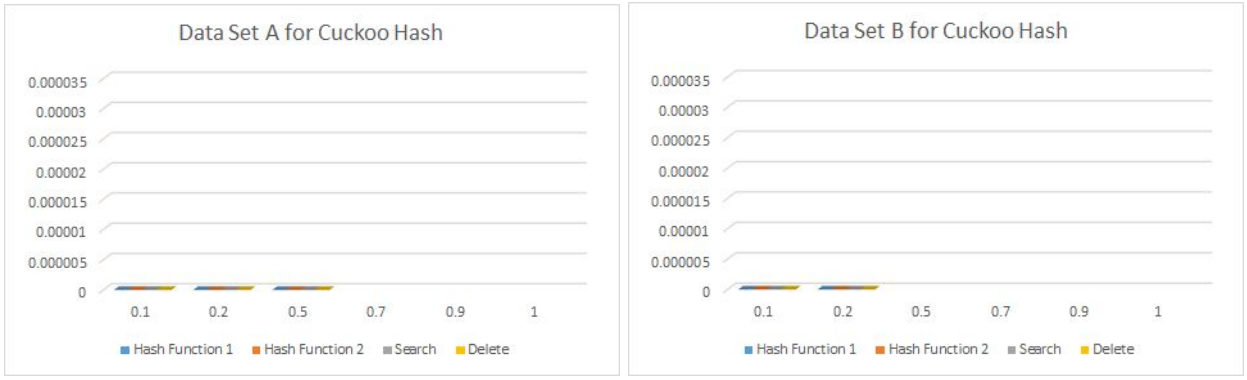

Figure 4: Data Set A and B for Cuckoo Hash collision resolution

Results

It can be concluded that the Binary Search Tree collision resolution was the most time-effective of the four collision resolutions. The time it took to insert, search, and delete 100 elements at six different load factors was so small that C++ double was not precise enough to register it, rounding it down to 0 seconds. The linear probe took very little time like the binary search tree method until it reached a load factor of 1. At this point, it took a considerable amount of time to insert, search, and delete.

Trial and Errors

There were a few aspects of the project that could have been more refined or provided more structure. The first problem was with the time code that was provided to us. Multiple methods were attempted to try and get the clock to register a time that wasn't zero seconds. The method that we had to settle on was recording the time it took to insert 100 elements and dividing it by 100 to get the average time it took to insert 1 element. This was the same for deleting and searching. However, as seen in the figures, it only worked for some of the bigger functions. The figures seem quite deceiving to say that the time it took to insert, search, and delete for different load factors was zero seconds, but there was nothing that we did that fixed this issue. The second issue was with the cuckoo hash. This method was not difficult to implement as a few sources on the internet explained it well; however, when it came to the issue that some numbers sent the code into an infinite loop. Two keys ended having the same hash value for both tables which prompts the code to insert them in an infinite loop. None of the numerous attempts that we tried to use to resolve this issue proved to be an ultimate solution.

Conclusion

This project focused on comparing different collision resolutions based on the knowledge of data structures that were taught this semester for hash tables. It specifically focused on the time it takes for the different collision resolution at different load factors. In the end, we concluded that the best collision resolution is the Binary Search Tree because it took the least amount of time between the other methods we were able to compare. Even with the difficulties we faced with the time issue and the Cuckoo Hash implementation, we feel like we fully understand the project and can explain any aspect of it at various levels of expertise.