Matt Ceriello

CSCI 2270

Hashing Analysis

December 8, 2019

Hash Table Performance Analysis

Purpose

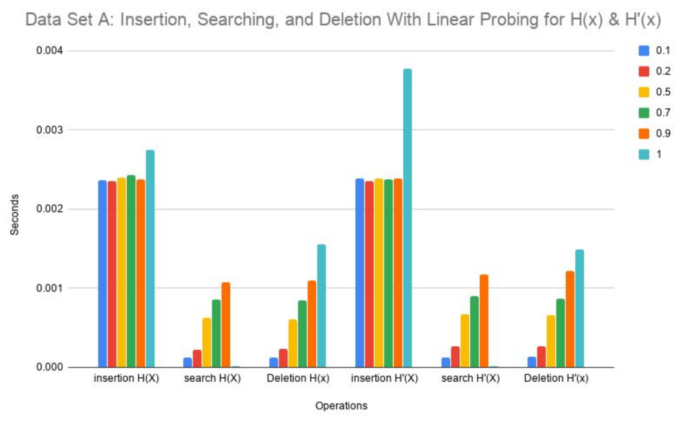
For this project, we are looking into some basic operations of Hash Tables, and how efficiently they perform. Specifically, we are timing how long it takes for specific functions to take place, while increasing the load factor of the hash table. We also want to take notice of how the different collision resolutions will affect the time measurement.

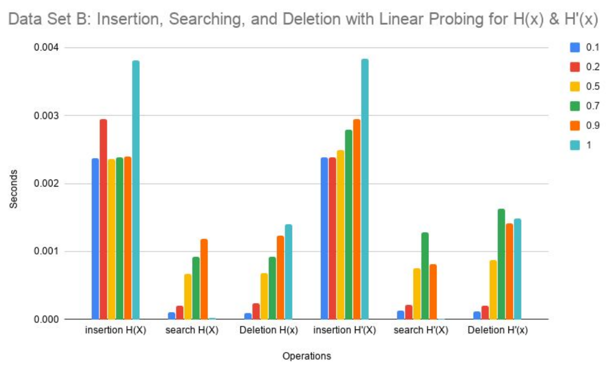
Procedure

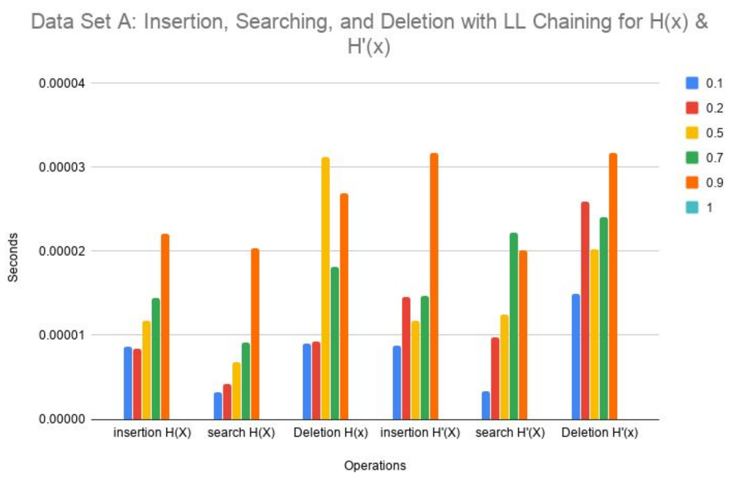
When initially approaching this project, I thought I would be better off sticking to the default project, since I am not as experienced with hash tables as I am with other data structures. The three operations I implemented were insert, search, and delete. All three of these were used in conjunction with the collision resolutions. For searching, I basically just traversed the tree until I found my target, then returned true or false depending on if it was found or not. After that, I wrote up the insert function. For this function, I started by calling my search function to see of this input was a duplicate. If it wasn't a duplicate, I calculated my hash index and tested to see if I could insert it. If not, I had to use the collision resolutions to find another spot in the table(Linear Probing), or chain with a linked list or binary search tree. For my delete function, I first searched to see if the key was already in the table, if it was, I went on and called my delete helper function, which is recursive so I can traverse the existing tree. Based on how many children the target node had, I performed a delete operation on that node. For measuring the insertion, searching, and deletion, I used the chrono clock instead of the timer provided in the rubric.

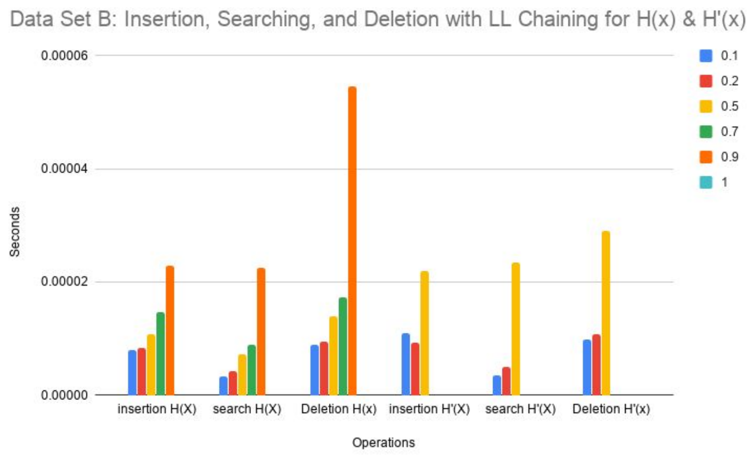
<u>Data/Results</u>

My implementations performed mostly how I expected. The first collision resolution I completed was linear probing. After putting my graphs together, I quickly noticed the difference in time it took for these implementations versus the rest. With linear probing, the key may not be inserted where the hash functions says it's supposed to be. Because of this, I essentially have to traverse the entire array until I find an open slot in the table. This causes operations to take a lot longer. In the graphs below, you can see how much longer insertion takes because of the

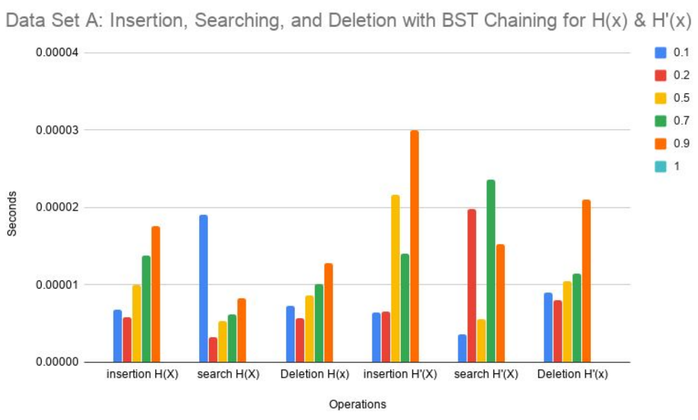


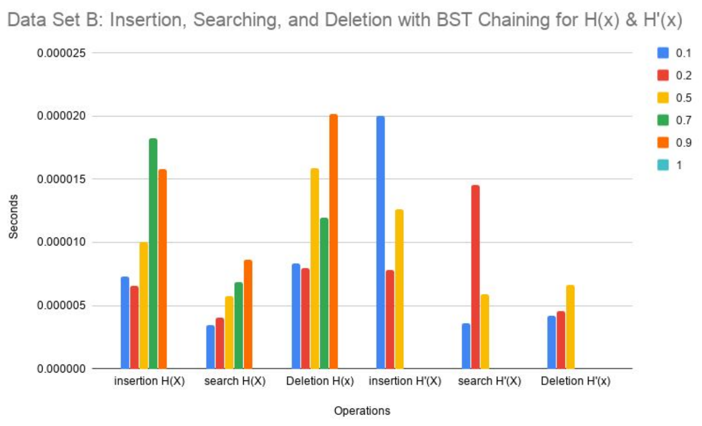




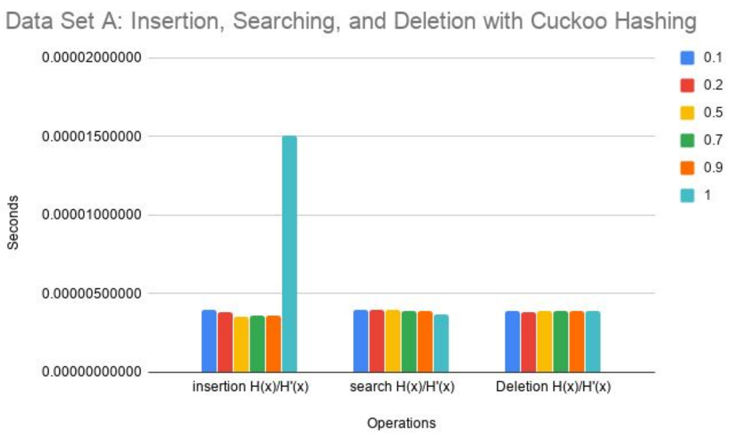


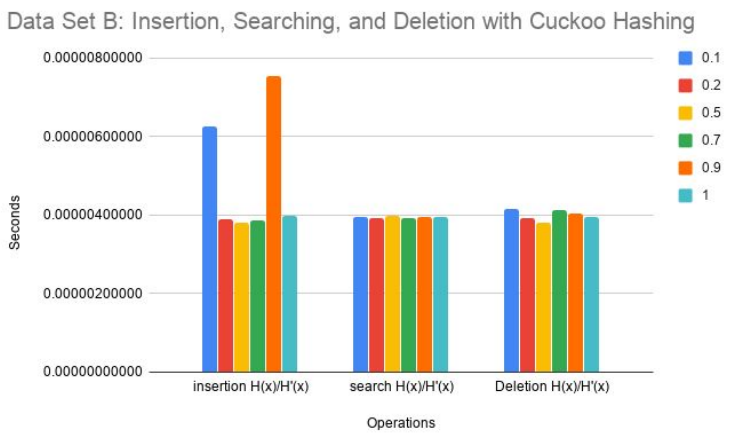
traversal across the array to find an open spot. For my chaining implementations, LL graphs shown above, my load factor seemed to never reach one because not every index in the table is being used. This time, if there is a collision, the key will be added to a linked list starting at the hash table index. The graphs show that as the load factor increases, all the operations take longer. For the graphs of BST chaining, you can see how much faster searching happens compared to





insertion and deletion. Lastly, I attempted Cuckoo Hashing with mild success. When trying to implement this method, I was only able to hash when a cycle didn't occur. If a cycle were to arise, my program would mostly likely quit or enter an infinite loop. However, my function





seemed to work for the provided data.