Jeremy M. Hein

CSCI 2270

Final Project Report

FINAL PROJECT (DEFAULT)

Purpose:

The purpose of this analysis is to examine the effectiveness of different hash table collision methodologies as well as the effectiveness of two different hash functions, defined as follows:

- **h(x)** = x mod TABLE_SIZE
- **h'(x)** = Floor(x / TABLE_SIZE) mod TABLE_SIZE

We have examined chaining with a Linked List, chaining with a Binary Search Tree ("BST"), Linear Probing, and Cuckoo Hashing. In order to compare the effectiveness of each methodology, we recorded the execution time on the three operations of hash tables (insert, lookup, and delete) across methodologies and hash functions.

Collision Methodologies:

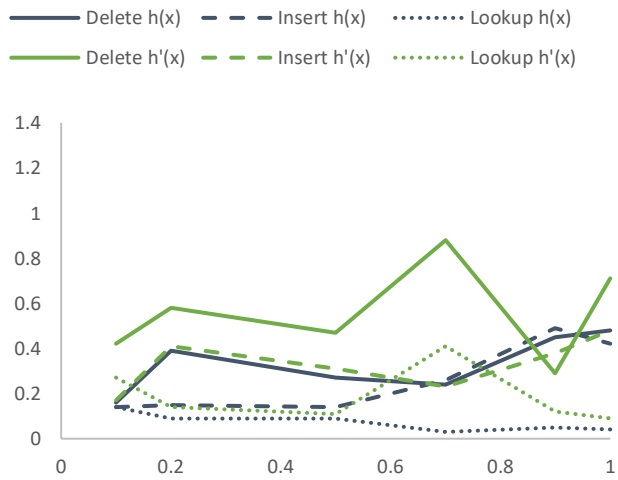
Chaining with a Linked List: In this methodology, our hash table class creates an array of structs. Within each struct is a key and a pointer to the next struct or node. As the name suggests, each bucket of the table (or index) acts as the head of a Linked List data structure. When a collision occurs, we simply add a new node to the list at that index. Resolving collisions in this fashion means our performance is reliant upon the length of our Linked Lists. The longer the lists, the more time it will take on average to perform operations.

Chaining with a BST: Similar to Linked List chaining, except each index of the hash table acts as the root of a BST rather than the head of a Linked List. When a collision occurs, a new node is added to the tree. Resolving collisions in this fashion means our performance is reliant upon the depth of our trees. The deeper the tree, the more time it will take on average to perform operations.

Linear Probing: Linear probing is an open addressing methodology where if a collision occurs, we look for the next available open index and place the struct in that address. In practice, this means the run time is extremely small at lower load factors, but as the table fills, it takes much longer to find an open index and thus the run time is much greater.

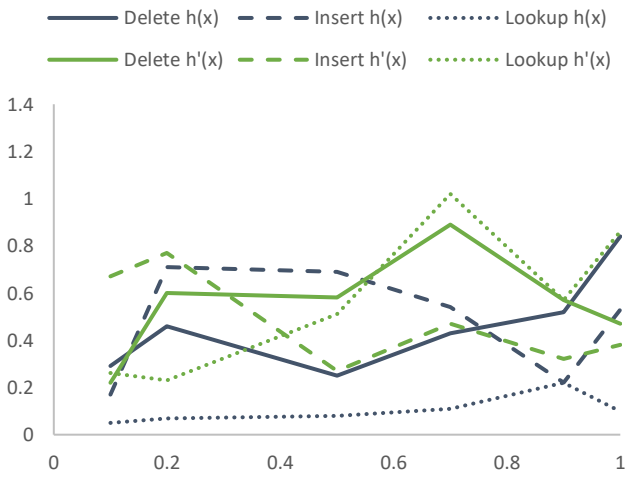
Cuckoo Hashing: Cuckoo hashing is another open addressing methodology in which the hash table class maintains two hash tables, one per hash function. If the index in the first table is unavailable, we place the struct in its alternative index in the second table. If both indices are full, we displace the element at the first index and check its alternative address. We continue this process of displacement until we find an open index. If we enter an infinite cycle of displacements, we resize the table and rehash all of the keys. Cuckoo hashing is extremely efficient for the lookup and delete functions, as the program only needs to check two locations for the element. The insert function is a bit more complex but still comparable to the other collision resolution methods.

Chaining with a Linked List – Data Set A:

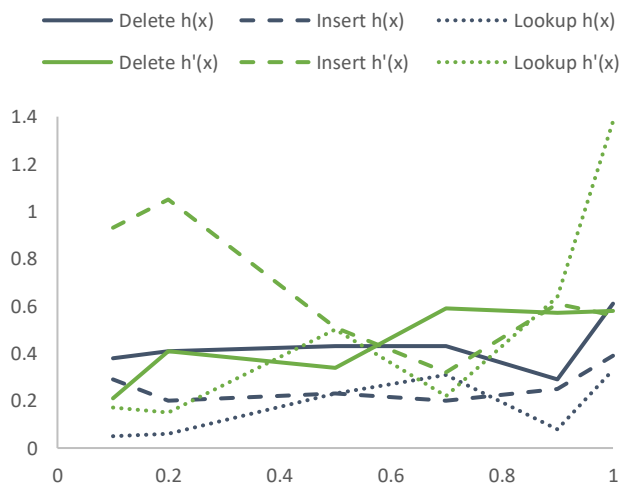


Chaining with a Linked List – Data Set C:

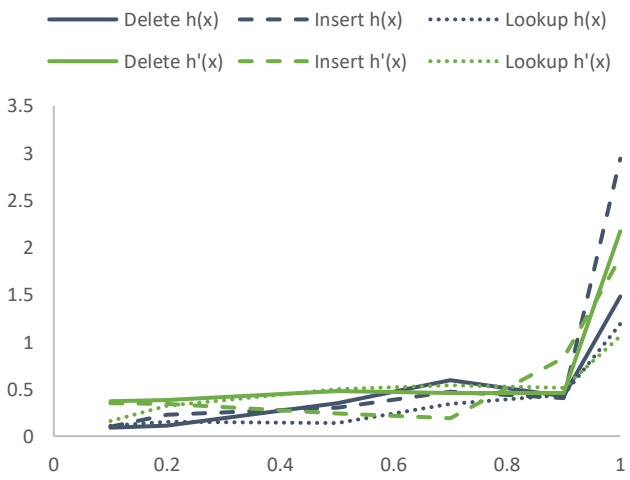
Chaining with a BST – Data Set A:



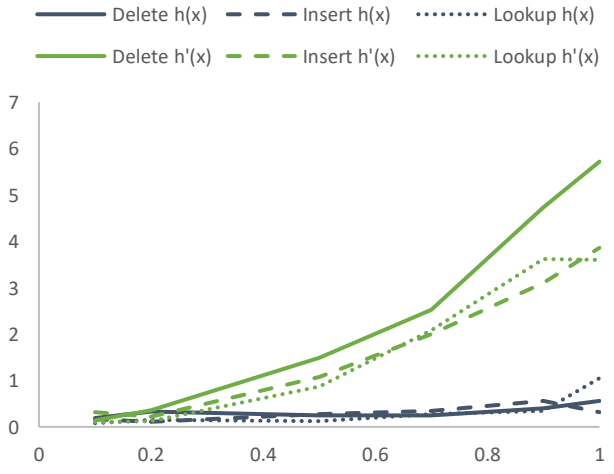
Chaining with a BST – Data Set C:



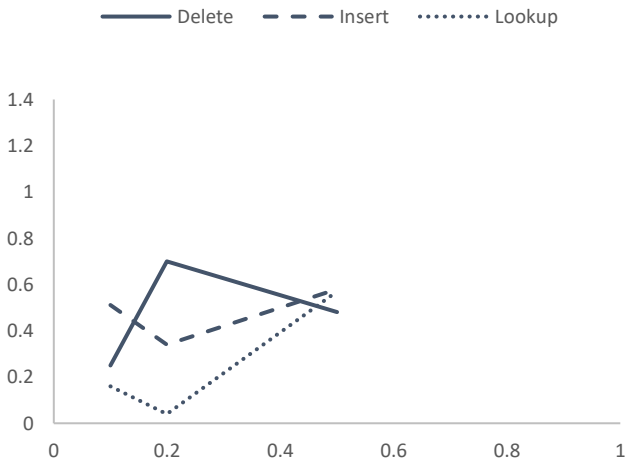
Linear Probing – Data Set A:



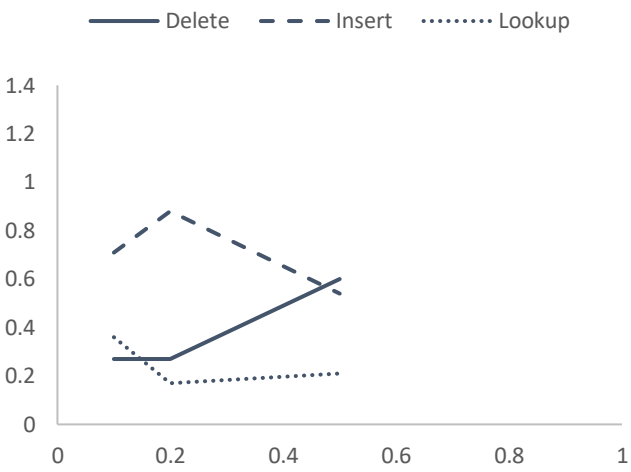
Linear Probing – Data Set C:



Cuckoo Hashing – Data Set A:



Cuckoo Hashing – Data Set B:



Discussion of Results:

Chaining Techniques: In both data sets and chaining techniques, we can see that the h(x) hash function, on average, is more efficient in terms of run time for the various table operations as compared to the h'(x) hash function. We don't see a large difference in performance as the load factor increases using chaining. This is likely because the lists or trees at each index are still relatively small. In addition, we can see some fluctuations in run time within each operation as the load factor increases. This is expected, as we are choosing a new set of random integers for each series of operations, some of which may have longer lists or trees to cycle through. When compared to linear probing or cuckhoo hashing, the chaining techniques will perform better at higher load factors and worse at lower load factors.

Linear Probing: As is the case in the chaining techniques, the h(x) hash function appears to perform better, on average, than the h'(x) has function. We would expect a relatively linear relationship between run time and load factor in this technique, because rather than chaining when we have a collision, we must search for the next open index. The more full the hash table, the longer we will have to search for an open index. The run time data presented above confirms that expectation, especially at a load factor of 1.0 when the table is full. When compared to chaining methods, linear probing will perform better when load factors are small, but will perform worse at higher load factors.

Cuckoo Hashing: In cuckoo hashing, we would expect to see the insert function require more run time than the search and delete functions. That is largely the case in the data presented above, with the exception the outlier data point at load factor 0.2 in data set A. The run time at lower load factors is similar to that of the other collision resolution methods. Note that because of table rehashing, the table is unable to reach a load factor in excess of 0.5. The rehashing function resizes the table to the next highest prime number, which would result in hundreds of rehashes in order to exhaust the list of numbers in each of the data sets.