Andres Montealegre & Manny Navarrete
Data Structures - Godley & Zagrodzki
December 8th, 2019

Hashing Implementation and Analysis

For the Final Project, Hashing Implementation and Analysis, we were tasked with designing and analyzing various collision resolution methods including Linked List, Binary Search Tree, Linear Probing, and Cuckoo hashing. Each approach allowed us to implement basic data structures, learned throughout the course, as well as practice hash function creation. Additionally, we were instructed to then test and analyze each function, search, insertion, and deletion, within each implementation. After completing all four collision methods, we timed each function at various load factors (0.1, 0.2, 0.5, 0.7, 0.9, and 1) and plotted them against each other, as outlined below. The load factor was calculated by multiplying the number of buckets (10,009) by the given load factor decimal.

We began with a Linked List Hash table, which utilized chaining to store the data across the buckets. Before insertion and deletion, this the table first searched for duplicates. In order to search for a given key, our function hashed the key (key % tableSize for H, (key/tableSize) % tableSize for H') and found the index at which they key was stored. This index stores a Linked List. We then created a variable node that traversed this linked list. If the node key does not equal the key we are searching for, it moves on to the next key, and so on. If the traverse node reached the end of the Linked List (i.e. traverse == NULL), it does not find a duplicate and returns a null node. However, if a duplicate is found, the function returns the node with the given key. In order to insert, the function uses the search function to make sure no duplicates are found. If none are found, the insert first hashes the imputed key and stores the result as an index. The function then goes to the bucket at the index found. If the bucket is empty (null), a new node is created and stored with the imputed key. If the bucket is not empty, a node traverses the list and stores the new node at the end of the Linked List. In order to delete, the function uses the search function to see if the key is inside the hashtable. If the key is found in search, the delete function hashes the key and moves to the index. Once at the index, a node traverses the Linked List until it finds the key. The node is the only one in the List, the bucket is set to null and the node is deleted. If not, the previous node is then set to point to the node after the node we wish to deleted. The node with the imputed key is then deleted and the function returns true.

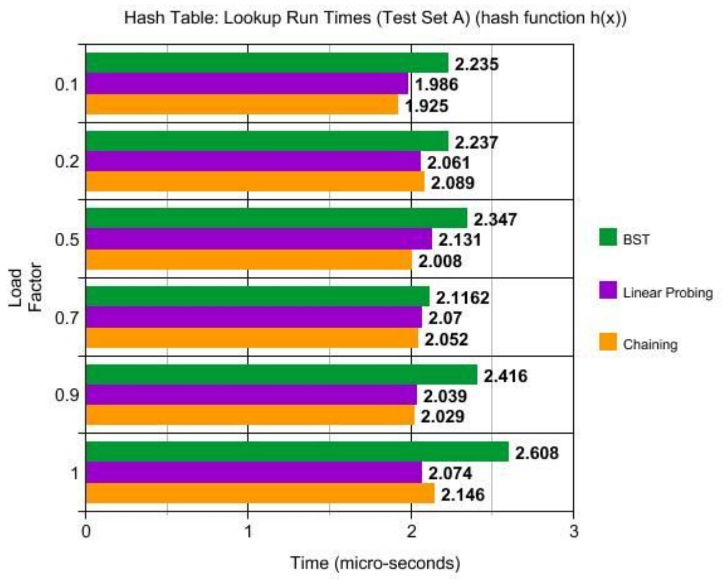
We then created a Binary Search Tree Hash table, which also used chaining to store the data. The BST Table worked very similarly to the Linked List Table. To search, the data structure hashed the imputed key and stored the index found. It then located the bucket with the given index. Inside the bucket, we had a Binary Search Tree. To navigate the tree, we used inequalities. If the key is less than the node, move to the left child. If the key is greater than the node, move to the right child. The function continues this method until either the key is found, or the node reaches a Null space. If null, the key is not inside the hash table and null is returned. If the key is found, it's node is returned. In order to insert, we make sure no duplicates are found in search. We then hash the imputed key and store the index returned. Using the inequalities (key less than -> leftChild, key greater than -> rightChild), we then traversed the BST at the index's bucket. Once we reach the first null node, we create a new node and set its parent to the previous node. Additionally, we make sure to set the parent's child (either right or left according to key value) to the new created node. Insert then returns true. To delete, we use search to make sure they key is within the hash table. If found, the delete function hashes the key and goes to the bucket of the index. Once at the bucket, delete checks if the key that needs to be deleted is the root of the BST. If it's the root, the function checks how many children the root has. If it has one child, that child is set to the root, and the old root is deleted. If it has no children, root is deleted and the bucket becomes null. If it has two children, delete calls a recursive function that finds the minimum value in the right side of the node and replaces the old node. All the parents and children are all reset and the old node is deleted. If the key is not the root, the recursive function above is called as well.

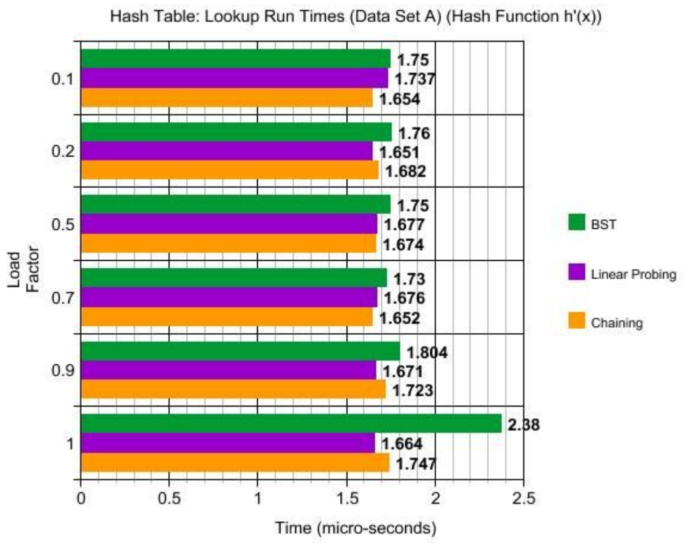
Our third data structure was a Linear Probing Hash table. To search, this data structure hashes the key and goes to the specified index. If the key is not stored at that bucket, it then moves to the next bucket in the hash table. The collision method continues to move to the next bucket until the key is found. If it reaches the last bucket, we reset the index and it begins searching at index zero. We placed a counter so if 10,009 comparisons (table size) are made and the key is not found, we return false. If the key is found return the node. To insert, we make sure no duplicates are found using search. Then we hash the function and go to the specified bucket. If the bucket is empty, create and store a new node with the specified key. If the bucket is full, traverse the table, starting at the hashed index, until the first empty bucket is found then store the new node. To delete, we use search to make sure the key is in the table. Then delete hashes the key and goes to that index. If the key is at the index, delete it and set the bucket to null. If not, traverse the table until key is found then set that bucket to null and delete the node.

Our last data structure was a cuckoo hash table, which uses two codependent hash tables that work together to store the data. The first hash table uses H = (Key % tableSize) to find the index and the second uses H' = (Key / tableSize) % tableSize. To search, we hashed the key using H and found the index. If the key is found at the index's bucket, that node is returned. If the key is not stored in the bucket it is rehashed using H'. The new index is then used for the second hash table. Using the new index, we look at the bucket in the second hash table. If the key is there, it's node is returned. If the key is not found in the second hash table, it is not in either hash table, and search returns null. To insert, the function hashes the key using H. If the bucket at this index is empty, the new key is inserted. If it is full, we rehash using H'. If the new index in hash table two is empty, the key is inserted at the bucket in the second hash table. If the bucket in hash table two is also full, we return to the original bucket. We then insert the imputed key in hash table one and pick up the node that was previously stored at this bucket. We then check the pick- up node's second bucket. If that one is full, we insert it in hash table two and once again, pick up the stored node. With the second node we picked up, we continue this process until all nodes are placed without any conflicts. If there is a case where insert runs infinitely (tableSize**2 comparisons), we resize the table. The table is resized to the next prime number. To delete, we first search to make sure the key is within the hash tables. We then hash the key using H and store the index. If the key is found at the index in hash table one, the bucket is set to null and the node is deleted. If it wasn't found, it is rehashed using H'. Using the new index, we then find the key in hash table two, delete it, and set the bucket to null.

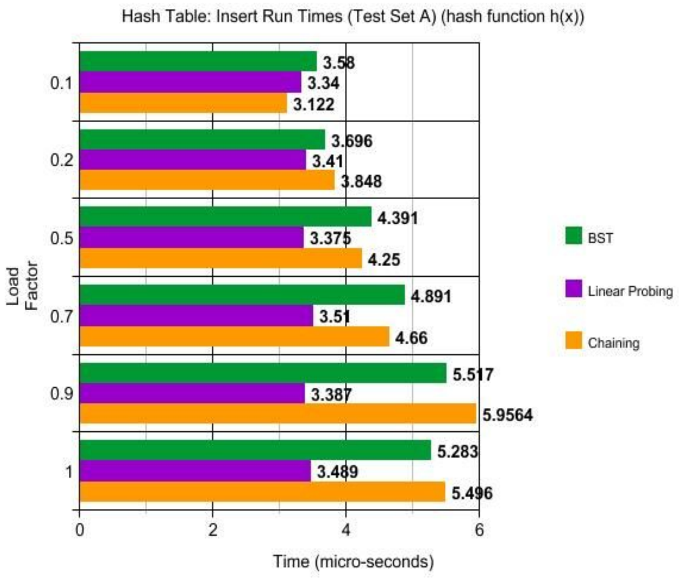
After creating each of the four collision resolution methods, we began timing the run time of each of the functions at different load factors. Seeing these graphs, allows us to understand our functions' efficiency. To time the functions, we used a clock to time 100 searches, insertions, and deletions for each method. We then repeated these steps 100 times and calculated the average run time and standard deviation of each. We then changed the load factors by inserting different amounts of numbers, before running the clock (i.e. Load factor of .1 = 1000 numbers inserted). All data found can be found in RAW_DATA_Lookup, RAW_DATA_Insert, and RAW_DATA_Delete.

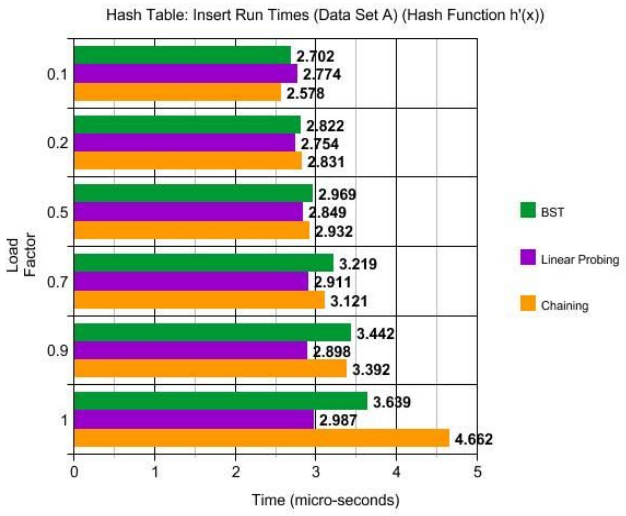
We started of by timing the search function when using both hash methods H and H'. Since, search is used in both insert and delete, we expected search to have the fastest runtime, which it did. As shown, the runtime of all the methods was around 2 nanoseconds. Furthermore, we were very pleased to find that the data from every load factor and implementation had a low standard deviation of .53 microseconds. This steady rate across all the data is because the search functions only slightly differed between implementation. Additionally, search differs little between load factors, because our search still compares null buckets. However, in H' search we see much faster times when compared to H. This difference in time is due to the fact that H' creates much smaller indices. Since the hash H' function creates smaller indices, the search function finds and returns nodes much faster. Although, cuckoo hashing is not pictured, it would theoretically have the fastest search time. The cuckoo, when implemented correctly, only makes two comparisons, and should be able to return faster than any other implementation. BST consistently had the slowest search time. We believe it is the slowest because it chains, unlike Linear Probing, and it has a much more complex traversal (more comparisons) than a Linked List.



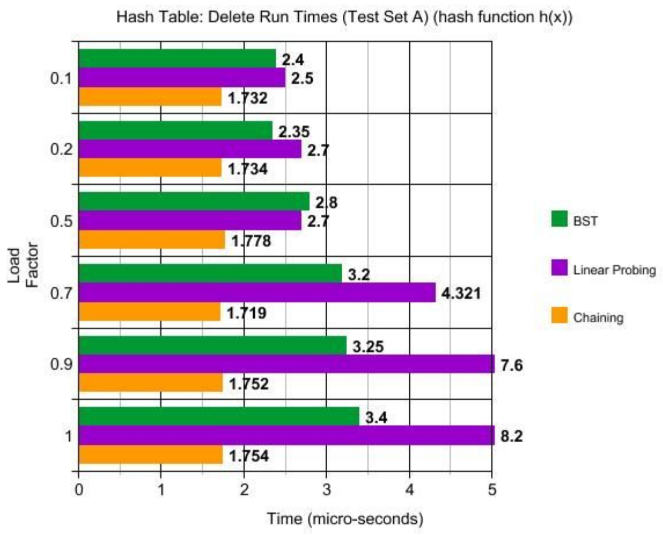


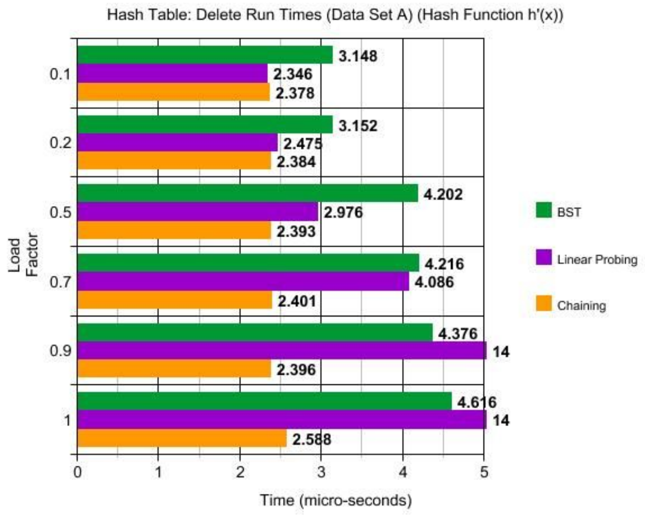
We then began timing the insertion of our four implementations, when using both hash Functions H and H'. Since, insertion includes our search function, we expected it to take longer, which happened. Immediately, we noticed that load factor had a dramatic effect on the run times of BST and Linked List. Since both utilize chaining, they become harder to traverse as more data is added. If any numbers overlap in the hash function, they have to chain the inserted number. On the other hand, Linear Probing has a much more consistent run time between load factors. This can be attributed to the way it inserts in a linear, non chaining way. Since it compares every bucket, no matter what load factor, it stays consistent. However, there can be times it has to compare every bucket, increasing run time dramatically. Cuckoo hashing can have very different insertion run times depending on the data set. We had trouble finding a method that would be able to identify if it ran infinitely to know when to resize the hash table. Since the cuckoo function can have instances where the table has to be resized often, it's run times can differ dramatically. Despite this, all our hash tables had a standard deviation of only 0.86 microseconds, meaning each method worked consistently throughout each load factor. When comparing H and H' we noticed that H' had lower insertions on average. This difference can be attributed to the lower indices that makes searching (used in insert) a lot faster.





Lastly we timed the deletion of our hash table implementations; again using both hash functions H and H'. We saw that the load factor had a direct impact on some of the implementations more than the other. Even with this being the case, our average standard deviation for deletion was below 0.7 microseconds for both hash functions, meaning our results were consistent throughout out 100 tests. Regarding linear probing, we believe the execution time increased dramatically when the hash table was almost full because as most of the indexes are being filled, we have more indexes to iterate through to make sure a number exists or doesn't exist in the hash table. As for our chaining implementation, the load factor had little-to-no effect on the average execution time, as there was really no penalty for having a more full table, other than having to traverse what is likely a small linked list in the bucket. Finally, BST





is directly affected by the load factor of the hash table because we have to traverse more complex trees to find and delete the desired key as the load factor increases. Theoretically, cuckoo should be the fastest deletion function across all implementations. Since cuckoo deletion only takes a maximum of two comparisons, it can complete this function quickly. When comparing H and H’ we noticed that deletion increased in average run time. We believe this is due to the fact that the data is becoming more centralized throughout the hash table. This means that the chaining implementations start to have more complex traversals, increasing run time.