Ryan Osler Section: 101
Jackson Bilelo Section: 202
CSCI 2270 - Data Structures

Hash Tables Final Project

Purpose:

The purpose of this project was to read in a data set, send the values through a given hash function, and use certain collision mechanisms that allowed us to organize the data when the index of two values are equal. Combining the two hash functions with the four collision mechanisms individually, we need to assess the two data files to investigate certain execution times to insert, search, and delete a certain node in our hash table. After coding each function, we need to assess the data given from our algorithm and come up with a conclusion on the execution time of these functions.

Procedure:

Before truly starting the project, we thought about any extra functionality that might be going into the collision resolution mechanics implementation. From there we were able to devise a basic outline of a Hash table-class with a header file, driver, and implementation file. Each combination of hash function and collision mechanisms have their own driver file. We could've put them all into one file, but for organization and preference we decided to break it up. This way, when we wanted to collect the data on a certain combination, we could run the driver file and collect the data in an output. We further developed the skeleton of our code, and then would continuously compile and run our program so that we didn't have to deal with small yet fatal errors that come from early stages of coding. We looked at the data set and supplemental info and got a better idea of how to lay the project out as well. After finalizing the driver file and the outline of what we wanted to do, we proceeded with first implementing the functions that we wanted to run. We thought that the functions were the most important and difficult part of this project. The seven driver files that were created were all relatively similar once we found the pattern that we wanted with data streaming.

Because we initialize our table as a vector with pointers, we need to create a structure that will hold all the data for every node in our structures. In our structure, we have a key value, a next value used for linked list, and a left and right child used for a binary search tree.

First a hash function is used to input a key and put our index. The first data structure that we implemented for each index in the hashing tree was a linked list. The linked list is one of the simpler ways of retrieving and inserting the data we wanted to manipulate. After obtaining the index for which our value would be inserted into, we would then traverse the linked list to the end and insert our new value. Making our hashing table a pointer at every index allows us to use the vector index as a pointer to our root node of the structure, and in this case the head of the linked list. After inserting the data, we proceed to search the tree for these same values. Searching the table for our vector only requires the index and key value. From there, we proceed to traverse the linked list until our value is found and we return it. To delete a value from our table,

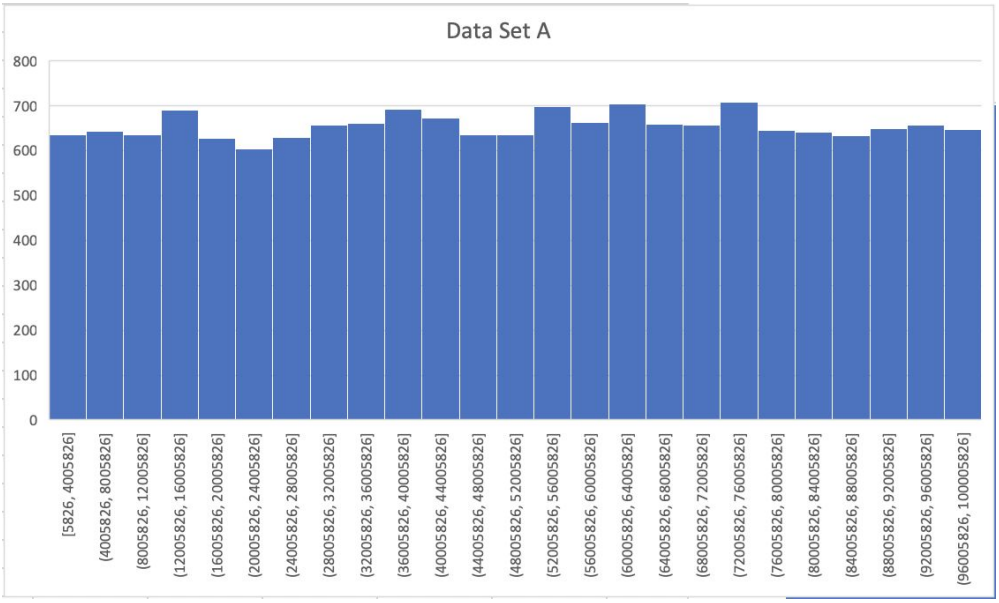
we first search for the value and if it exists, we traverse the given linked list and delete it and using our trail node we make sure the linked list is rearranged after the deletion.

The next data structure we used to organize our data was a binary search tree. Again, the index our table points to the root node of our structure, and in this case is the root of our search tree. After obtaining the certain index, we then traverse our tree comparing our key at every node until we find our NULL position in the tree where our data is then inserted. The same goes for searching in a search tree. We obtain the index, then from the root of the tree, we traverse the tree until we find our node and return it. For deleting a node in search is more difficult. We first search our given tree for the node and if it exists, we delete it within a given case. Since the nodes below a parent node are dependent upon the parent for searching and insertion, we need to have multiple cases for how we delete our nodes. First, if our node is the root node and it has no children, we can delete it and make the index equal to NULL. If our node only has one child, left or right, we switch the parent node that we want to delete with the value of our child node and delete the child node. If our node has two children, we need to find the next highest value in the tree to replace it. To do this, we need to traverse right one node, then traverse left until we find the smallest value. After this is done, our original parent node's data is replaced with the minimum value node we just found and the minimum value node is deleted.

The next way of organizing and retrieving the data in our table is the mechanism of linear probing. In this mechanism, there is no data structure attached to each index. Instead, we have one data value per node. To do this in an organized way, we first obtain the original index that corresponded to our value. If this value is empty, we add our value to that node. If our original index is occupied, we then increment our index by one until we find an open location to insert or value. To then search for this data, we first inspect our original index, and if we find it we return the node. If we don't find our value, we increment the index by one until we either find an open spot, or we find our value. One thing i found that could be a problem here, is if a value that was originally in the way of our value that we inserted, is then removed from the table and a vacant spot is created. To protect against that, we need to search the entire list until we either find our value or we return back to our original index. To then delete a piece of data in our table, we first search for it with our past function, then delete the node if we found our value.

The final, and most confusing collision mechanism is the cuckoo hashing. To start, we create two hash tables, each with their own hash function. To insert data into this dual table, we first obtain the first index of the first table and check and see if it is vacant. If the node is open, we add our value to the first open node. If it is taken, we check our second index. If that is open we insert our data. If the second index is also occupied, we take the value that is currently there and we find that values alternative index on the other table. If that location is also taken we replace the value and relocate the value that was there. We continue this loop until we find an open location for a value. If this loop is infinite, we need to rehash our table by increasing the size and creating two new hash functions. To search for a node is much simpler, we simply check both of our index locations and if our value is there, we return it and if not we return NULL. To delete the node, we search for it and delete the node and return it to NULL.

Data:



Data set A is graphed above. With such a large set of numbers it is tough to visualize the data as is, but for the most part we can see that the data is relatively similar in size. With large values in a large data set with a large table the data is relatively spread out which should keep the times relatively similar and easier to judge times of insertion, search, and deletion.

Results:

One thing that we can see from the graphs that is most surprising is that we would expect the data to trend upward as we increased our load factor. Throughout the course of this project we have come across many obstacles that we had to think through and one of the big obstacles that we could not figure out how to do which attested to this trend was when we added new values to up the load factor. When we wrote our code, we would open the text file, insert 100 values, close and reopen file, search for those 100, close and reopen file, then delete these 100. We figured this was the best way to make sure we were really got a good time in searching for these values. Then when we added 1000 values to get a load factor of 0.1, we started from the beginning of the data file. Then when we wanted to insert 100 values, we could insert the next 100 numbers but if we didn't close the file we wouldn't have been able to search or delete 100 numbers that were in the list. So, every time we returned back to the beginning of the .csv file to insert search and delete numbers that were actually in the list. But, by doing this, these numbers were already in the file and therefore made each mechanism look similar. If we had more time to learn and get help, we may have been able to possibly create an array of the data file that would allow us to pick and choose certain values from the file that would allow us to get a better look at the execution time of the collision mechanisms.

Another issue that came up for linear probing was the inability to properly rehash the table when the load factor neared toward 1. We only had one input into our hash function, the key value, which was called in the driver file before we called our insert linear probing function. To fix this value, we thought of two solutions. One was to have two inputs into the hash function, one being the key value and one being the table size. This way, we can rehash the table and any future numbers that came into the function were hashed accordingly with a new hash function. The second idea was if we made a helper function that recursively rehashed the table and changed the hash function. This option was less plausible but possible. We were unable to insert it into our code as we had a certain deadline for the project and wished to get as much done as we could to present.

We also were unable to properly implement the cuckoo insert function. We were able to properly code a search and delete function for this mechanism as the value we were looking for is in either of the two possible hashing tables. We are familiar with how the insert function works with the replacement of the nodes so that each value was in one of its two possible index positions but the execution of the code was poor.

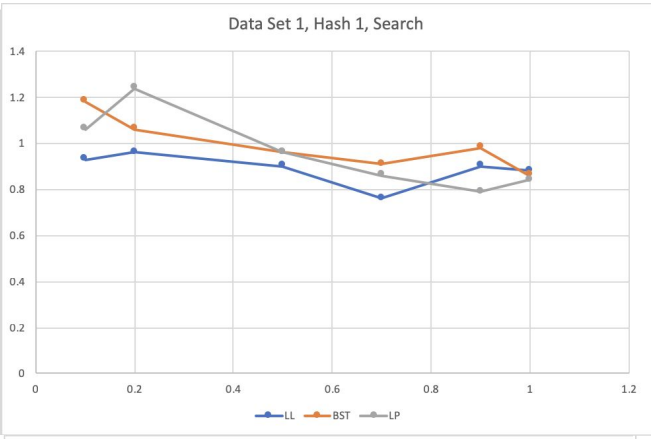
Since we did not have the opportunity to properly collect data and analyze it, we made some predictions and educated guesses on what we thought was the fastest and most efficient collision mechanism out of the four options. We found that it very much so depends on the load factor of the table.

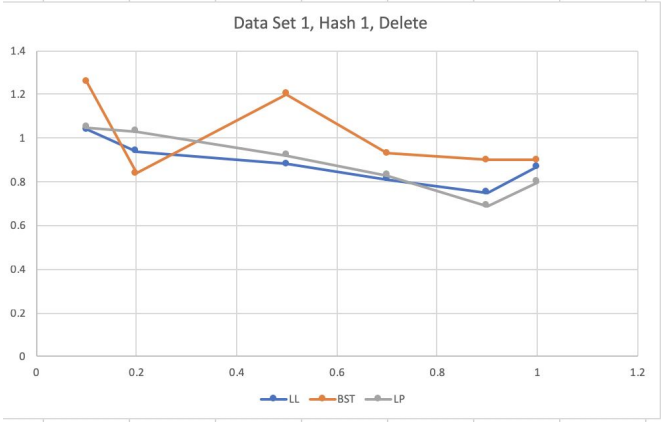
When the table is run at low load factors, certain mechanisms such as linear probing and cuckoo hashing are the most efficient ways of inserting and searching for data. Since there aren't many elements in the table/tables at this time, there is a smaller chance that there will be a value in our desired index. This way, we can insert our value with minimal adjustments such as incrementing the index or replacing until an open index is found. With the linked list and binary search trees there could be a couple steps taken before we arrive at our value to insert.

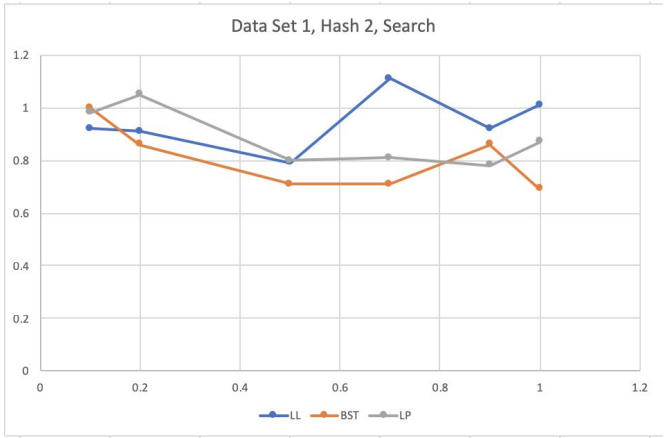
Once we hit higher load factors, we start to see that search trees start to become quicker at inserting and searching for values depending on the data set. With the ability to have the data be in a defined index, we can traverse through and find our data. At high load factors a binary search tree is the most efficient way of finding data compared to a linked list. With a linked list, we have to search through an entire list to find our value. With a search tree we have to traverse through the values without having to traverse unnecessary values.

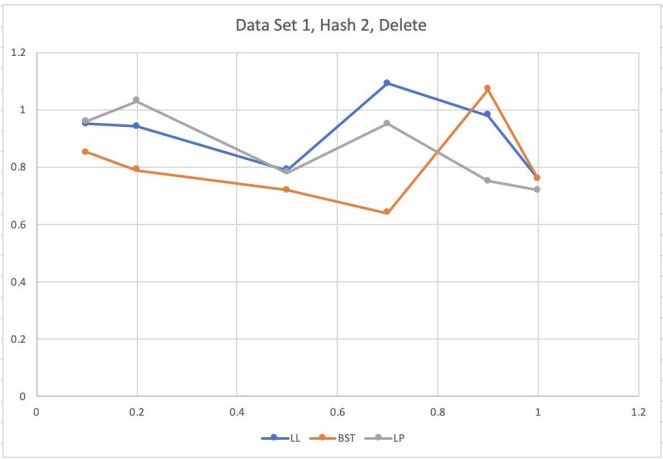
When it comes to searching or deleting values at any load factor, cuckoo hashing is the most efficient because we are guaranteed that the value is in one of its two alternative locations. It may be most efficient in searching and deleting but it has the potential to be very poor when it comes to inserting. Again, depending on the data set and the hash function, linear probing and cuckoo can be better collision mechanisms than the linked list. For inserting data, on average a binary search tree is best, but at low load factors cuckoo and linear probing are efficient. When it comes to searching and deleting data, cuckoo hashing is the best due to is guarantee of position. Overall, the cuckoo collision mechanism is the most efficient shortly followed by the binary search tree.

Appendix of Graphs:

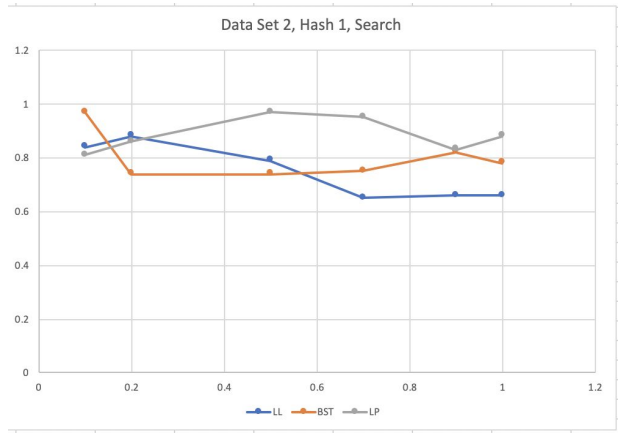






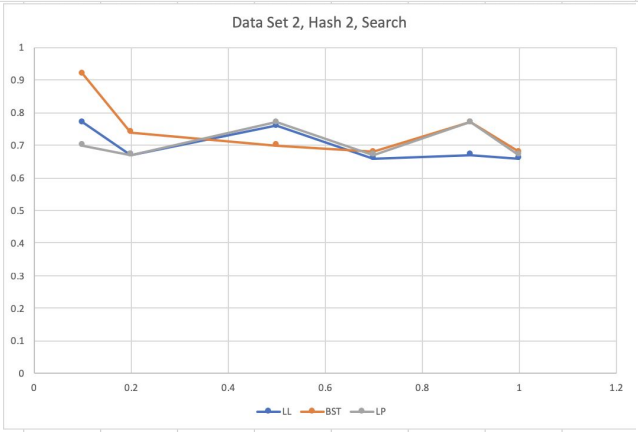


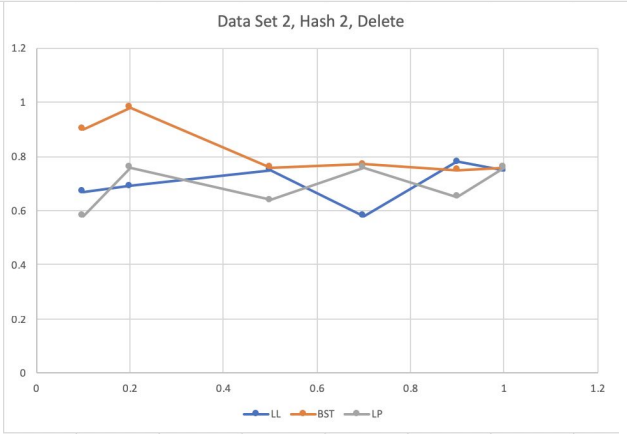

Data Set 2, Hash 1, Insert
1.4
1.2
1
0.8
0.6
0.4
0.2
0
0
0.2
0.4
0.6
0.8
1
1.2
LL
BST
LP


Data Set 2, Hash 1, Search
1.2
1
0.8
0.6
0.4
0.2
0
0
0.2
0.4
0.6
0.8
1
1.2
LL
BST
LP


Data Set 2, Hash 1, Delete
1.2
1
0.8
0.6
0.4
0.2
0
0
0.2
0.4
0.6
0.8
1
1.2
LL
BST
LP


Data Set 2, Hash 2, Insert
1.2
1
0.8
0.6
0.4
0.2
0
0
0.2
0.4
0.6
0.8
1
1.2
LL
BST
LP


Data Set 2, Hash 2, Search
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0
0.2
0.4
0.6
0.8
1
1.2
LL
BST
LP


Data Set 2, Hash 2, Delete
1.2
1
0.8
0.6
0.4
0.2
0
0
0.2
0.4
0.6
0.8
1
1.2
LL
BST
LP