Leah Dillard

CSCI 2270

Asa Ashraf

December 6, 2020

PART A FIGURES

HASH TABLE WITH OPEN ADDRESSING, QUADRATIC PROBING FIGURES

FIGURE 1. Iterations 0-100

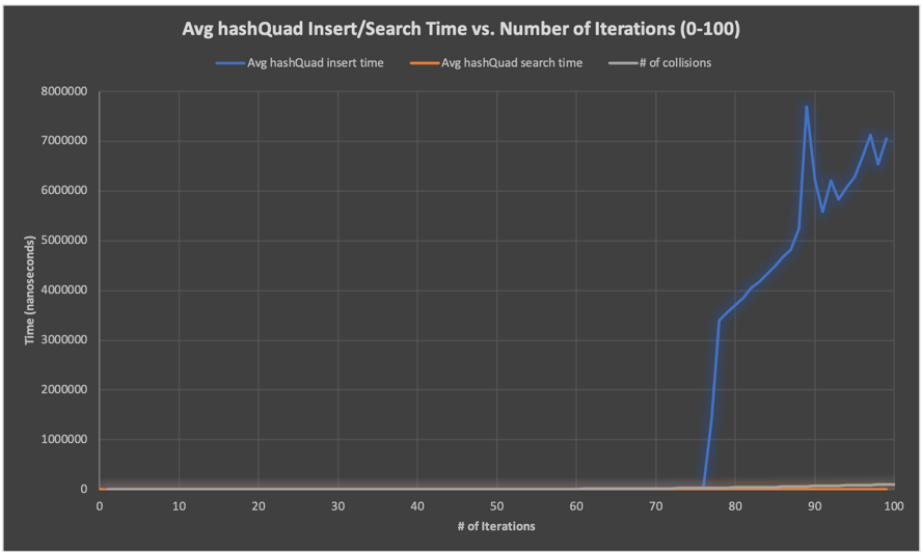
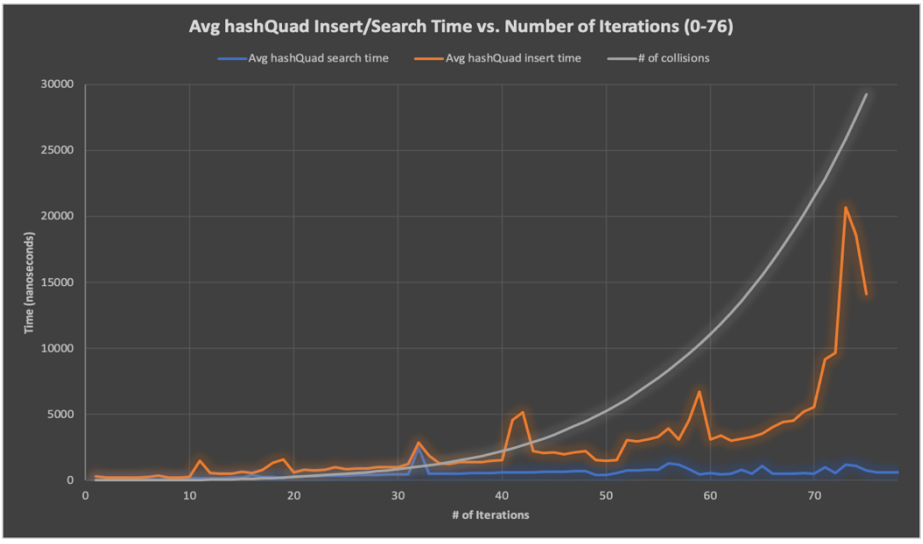


FIGURE 2 (for comparison). Iterations 0-76



DOUBLY LINKED LIST FIGURES

FIGURE 3.

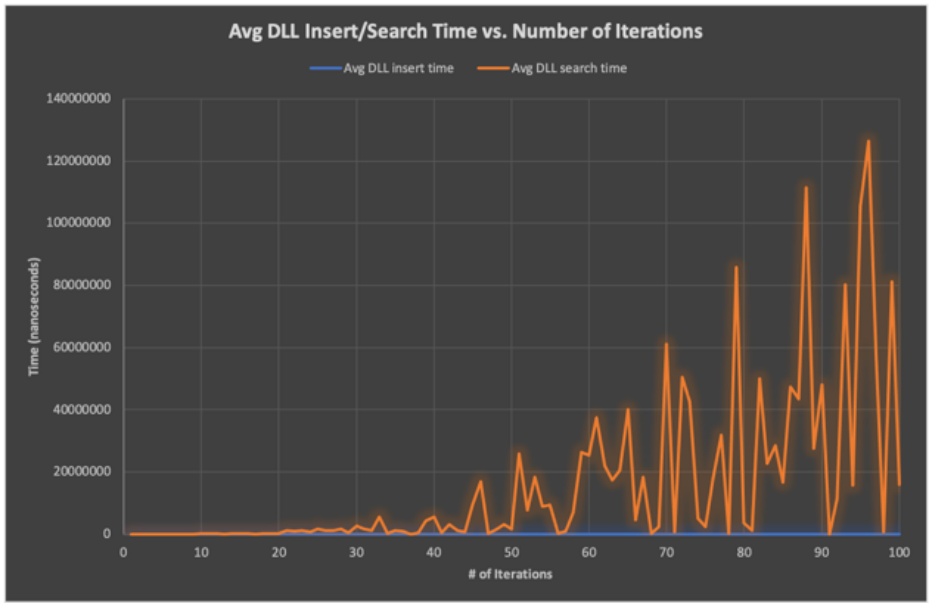
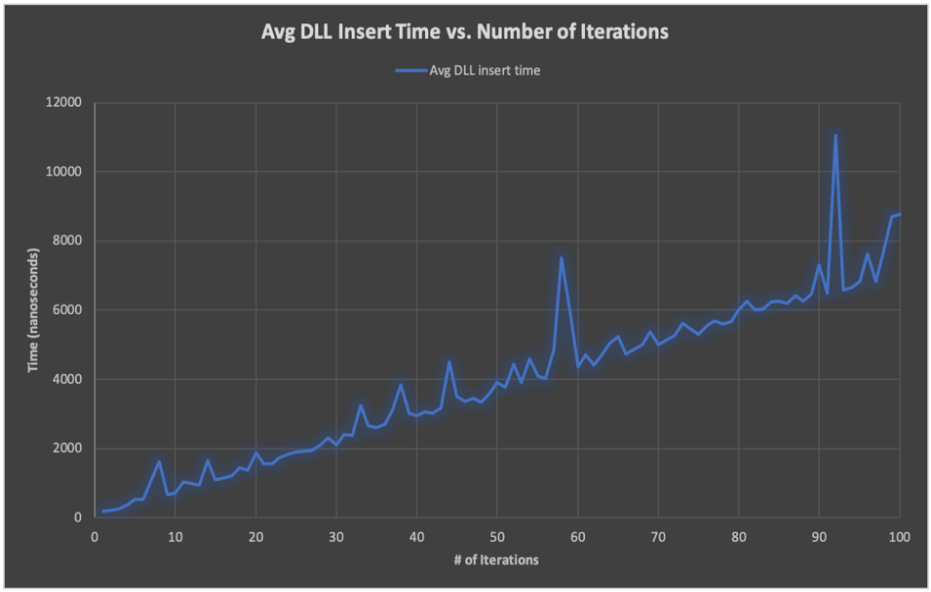


FIGURE 4 (for comparison).



SUMMARY FIGURES FOR INSERT

FIGURE 5.

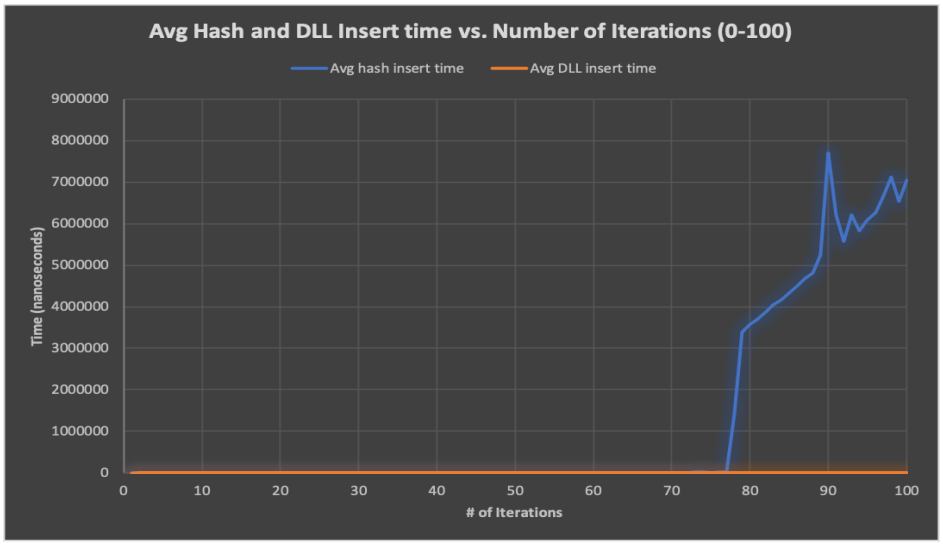
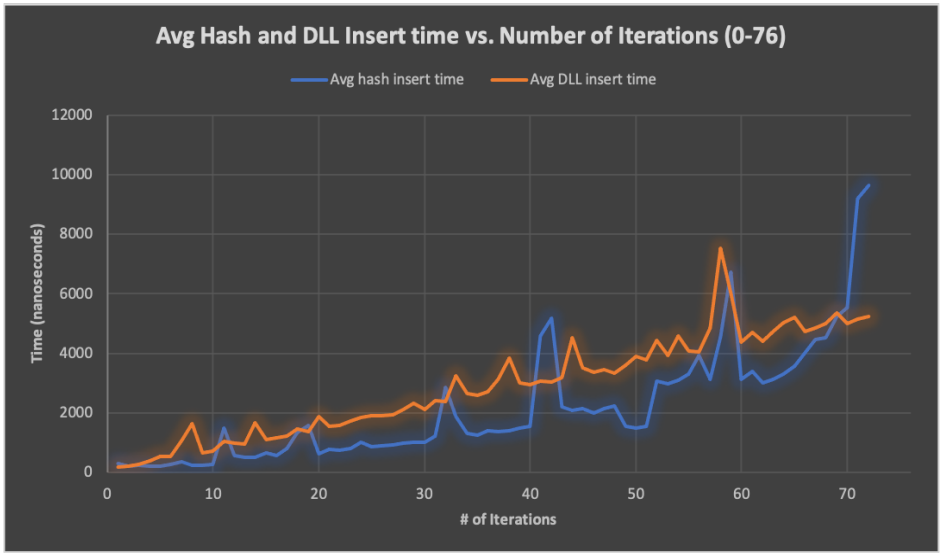


FIGURE 6 (for comparison).



SUMMARY FIGURES FOR SEARCH

FIGURE 7.

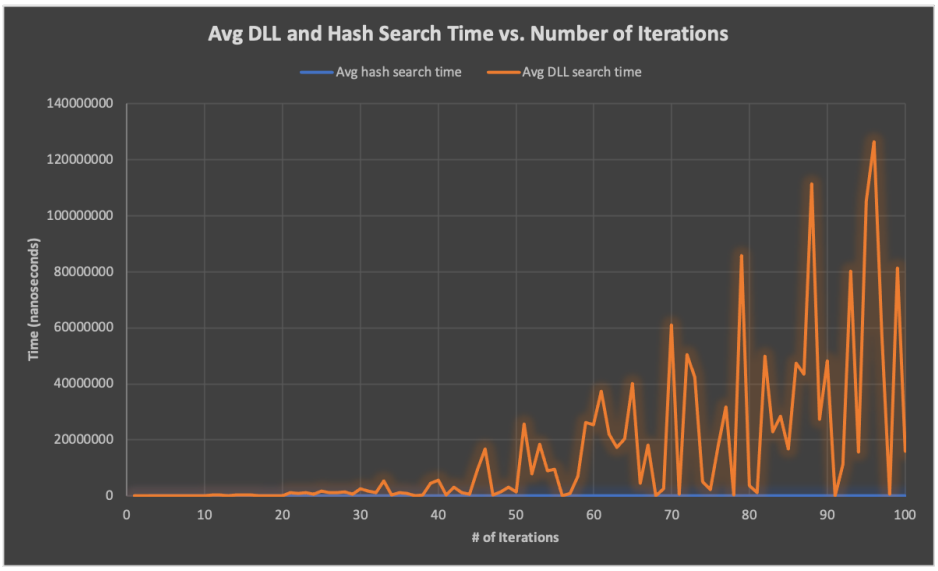
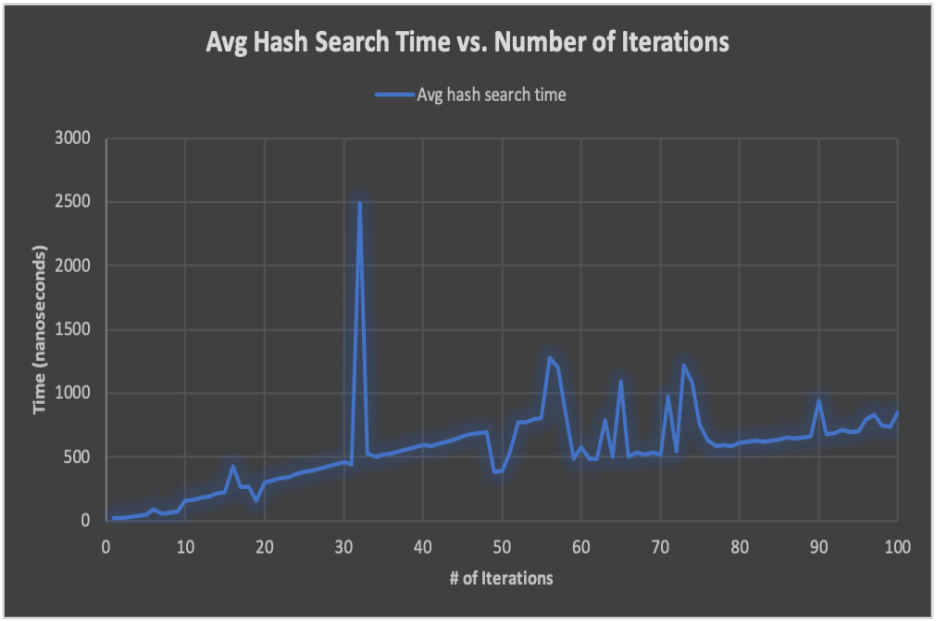


FIGURE 8 (for comparison).



PART A REPORT

Between the hash table with open addressing (using quadratic probing) and the doubly linked list data structures, I found that the hash table had optimal search time while the doubly linked list had optimal insert time, comparatively. I measured the average time in nanoseconds, because it allowed for a more accurate graph output compared to microseconds. In both data structures, either the insert or search time was much greater than its counterpart. For example, the hash table insert time took almost 266 times as long as its search time, seen in Figure 1. Around iteration 76, seen in Figure 2, the insert function spikes up, indicating an increase in the time it took for the algorithm to find an empty bucket. This is because the hash insert algorithm is operating with O(n) time complexity, while the hash search algorithm is operating with O(1) time complexity. The O(n) time complexity is due to the iteration through the entire table quadratically to find an empty bucket if a previously hashed bucket is occupied. Because of this, I hypothesize that the increase in average insert time around iteration 76 has to do with the corresponding swift surge in the number of collisions, which indicates many full buckets and thus more time taken to find an empty bucket.

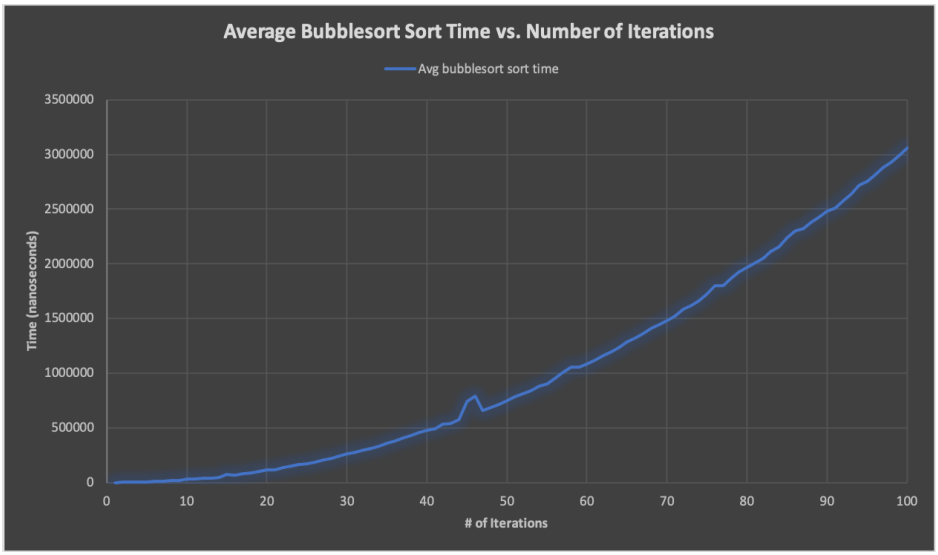
The doubly linked list, on the other hand, had low average insert times and very high average search times. This can be attributed to this data structure having an insert time of O(1), since each node was inserted at the head of the doubly linked list (which only required a pointer to be moved) as seen in Figure 4. The search time complexity was less than optimal at O(n), because in order to find a particular value, the entire linked list needed to be traversed, generally making the search time rise as each iteration was completed, seen in Figure 3. I hypothesize that the search time was higher than the insert time simply due to insertion being at the head of the doubly linked list, while search required traversing the entire list until the correct node was found.

The comparison charts between the doubly linked list and hash table in terms of insert time leads to the conclusion that the doubly linked list had a more optimal average insert time. This was due to its O(1) time complexity compared to the hash search O(n) time complexity discussed previously. Both the hash table and doubly linked list had similar upward trends, but around iteration 76, the hash table's average time dramatically increased. This can be seen in Figure 6 compared to Figure 7. Conversely, the hash table had a more optimal average search time compared to the doubly linked list (Figures 7 and 8). From Figure 7, the data for hash table search can barely be seen on the graph due to the long amount of time it took for the doubly linked list search. In Figure 8, it can be determined that the longest it took for the hash table to find a random value took 2500 nanoseconds. The longest it took for the doubly linked list to find a value was around 120,000,000 nanoseconds.

PART B FIGURES

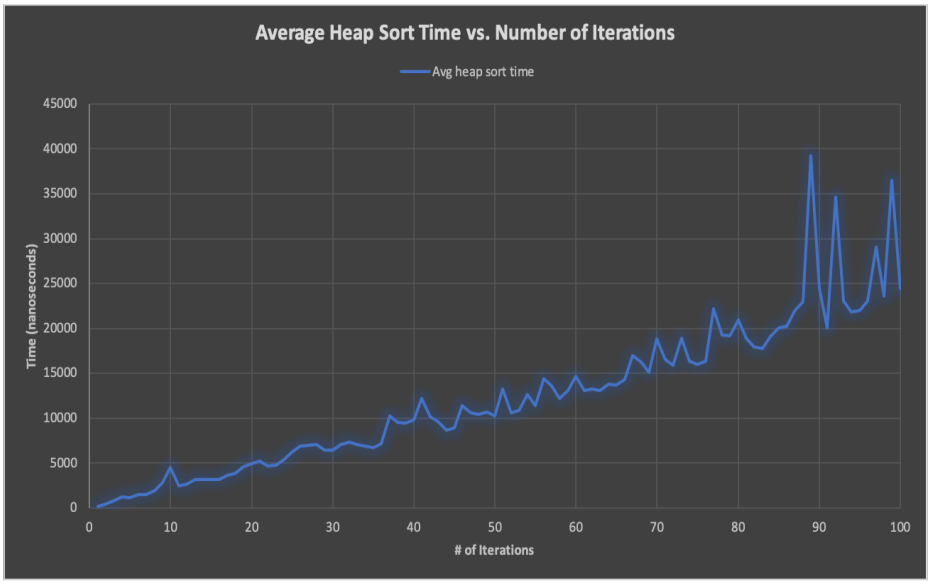
BUBBLE SORT FIGURE

FIGURE 9.



HEAP SORT FIGURE

FIGURE 10.



PART B REPORT

In terms of bubble and heap average sorting time, the heap was able to sort all of the inserted numbers from the data array much faster than bubble sort. In Figures 9 and 10, the sorting algorithm average times in nanoseconds are compared to iteration. It seems that the reason why a heap is the optimum data structure for sorting in this scenario is due to its overall time complexity of O(n log n) compared to the bubble sort time complexity of O(n^2). This difference in time complexity can be attributed to how often the bubble sort algorithm has to swap each value in an array with the value in front of it. If an array is large, this ends up taking quite a long time. The heap, on the other hand, is a complete binary tree in which values are sorted based around parent, left child, and right child indices. In Figure 9, it is apparent that the bubble sort is able to sort through the array with a more consistent increase in average time per iteration. However, this results in the entire sorting process taking more time, as the bubble sort algorithm took about 3,000,000 nanoseconds at its longest (Figure 9). In contrast, the heap sort algorithm only took 40,000 nanoseconds at its peak (Figure 10). I hypothesize that since my heap-sort algorithm relies on sorting by building a max-heap via sub-trees from the bottom to the root, it is much quicker with its average sort time compared to bubble-sorting and swapping through an entire array one-by-one.