Abeal Sileshi
December 5, 2020
CSCI 2270
Final Project Write Up

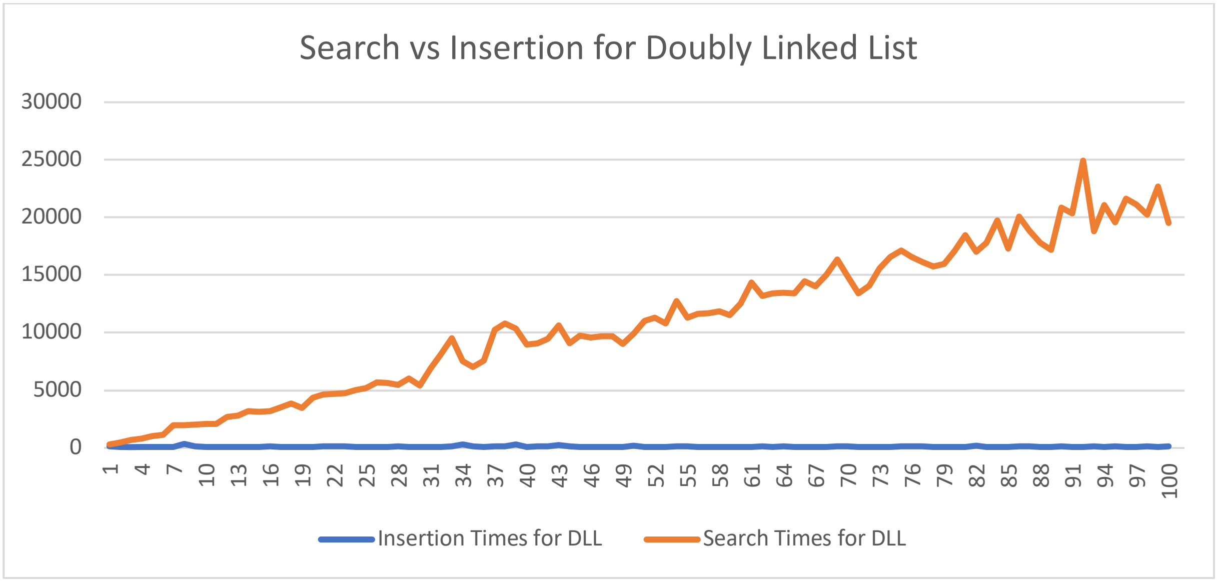
Part A

In Part A, we compare linked list performance with the hash table. Theoretically, a linked list as O(1) insertion time since we are inserting at a known position. My implementation of the linked list added each node at the tail of the linked list that had a tail pointer to identify the tail. So all I had to do was assign the old tail's 'next' to the new node, and then update the tail. However, the linked list has O(n) search time complexity. When searching for a number in my implementation you must start at the head of the linked list and traverse the whole list until you find the node. The worst case is that the node one is searching for is at the very end of the linked list.

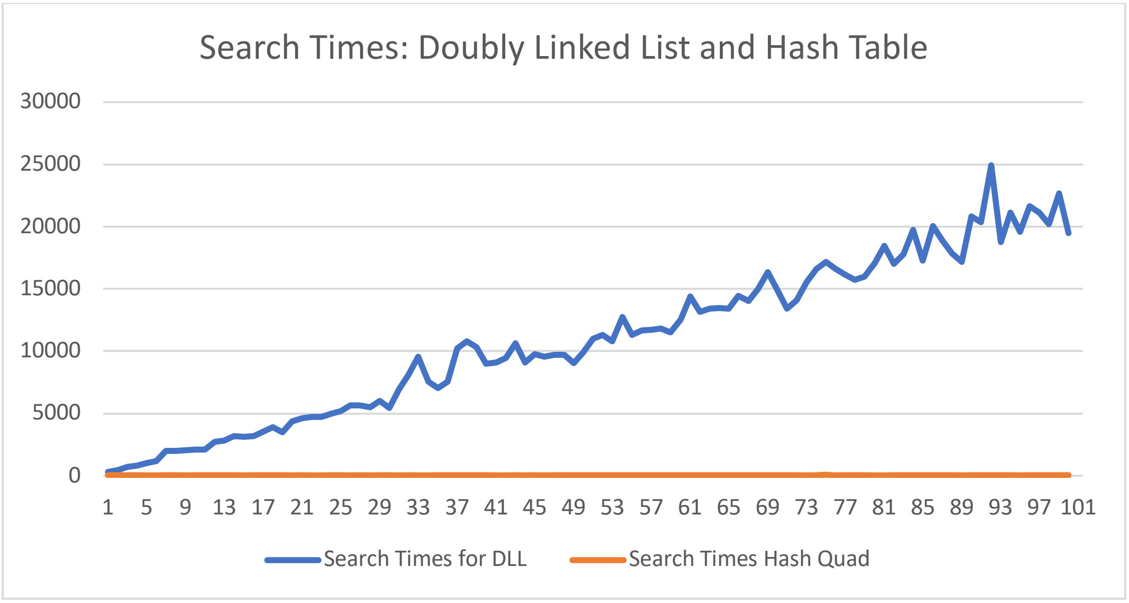
The hash table on the other side has O(1) search and insertion time complexity. But as we will see from my findings, the hash table doesn't always have O(1) time complexity. The worst-case scenario for hash tables is O(n) because of collisions. When multiple keys have the same hash value, you'll need to search n positions to find the key.

My findings indicate that a hash table is a much more efficient data structure. In all scenarios it beat the linked list in terms of performance speed.

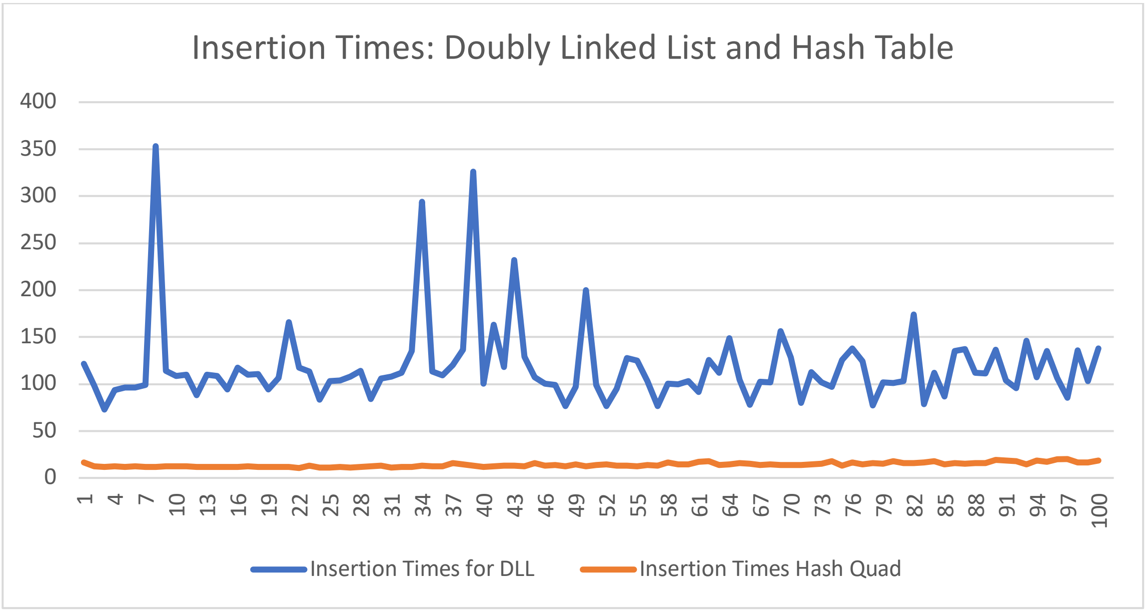
Note: all speeds are measured in nanoseconds. (The y-axis.)



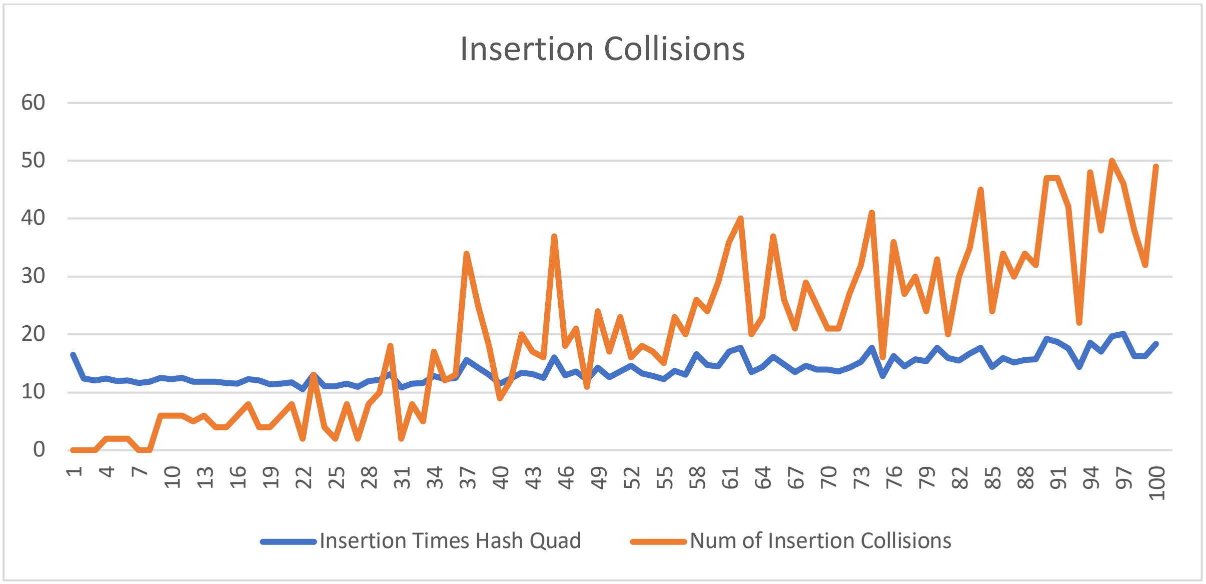
The insertion time is constant because we always insert at the tail of the linked list in my implementation. However, search times continue to grow because the linked list becomes bigger and bigger and you must therefore traverse more nodes.



With Hash Table having O(1) time complexity, it's no surprise that search times for LL is so much slower. Notice the LL search line seems to have an almost linear shape- thanks to its O(n) time complexity



This graph is interesting because it compares the LL Insertion's O(1) performance to the hash table's O(1) performance. While this as close as the linked list will get to the hash table in speed, it still loses in the end.



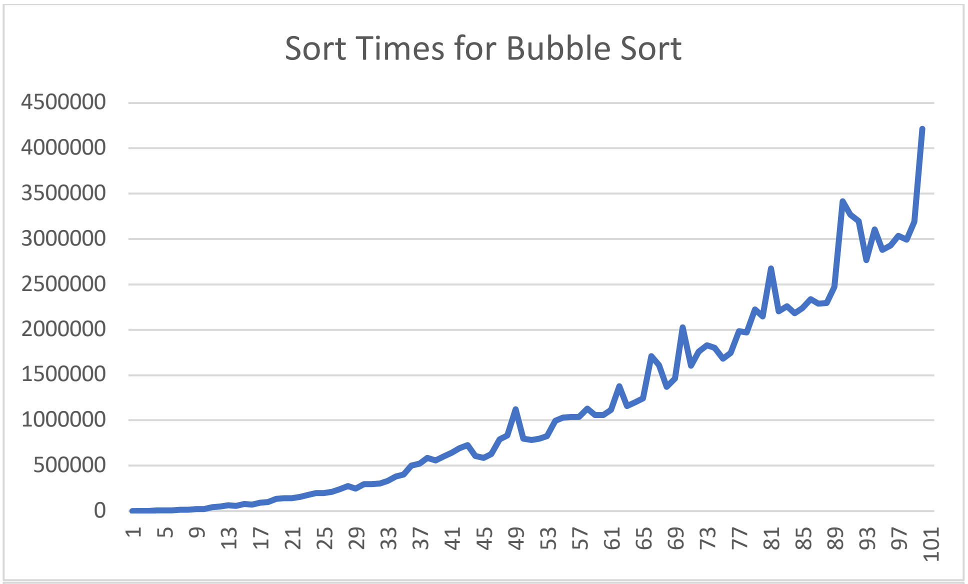
Collisions start from zero, because when you're inserting 100 elements into an array of 40009, it's unlikely to collide. But the collisions steadily increase as the hash table has more values.

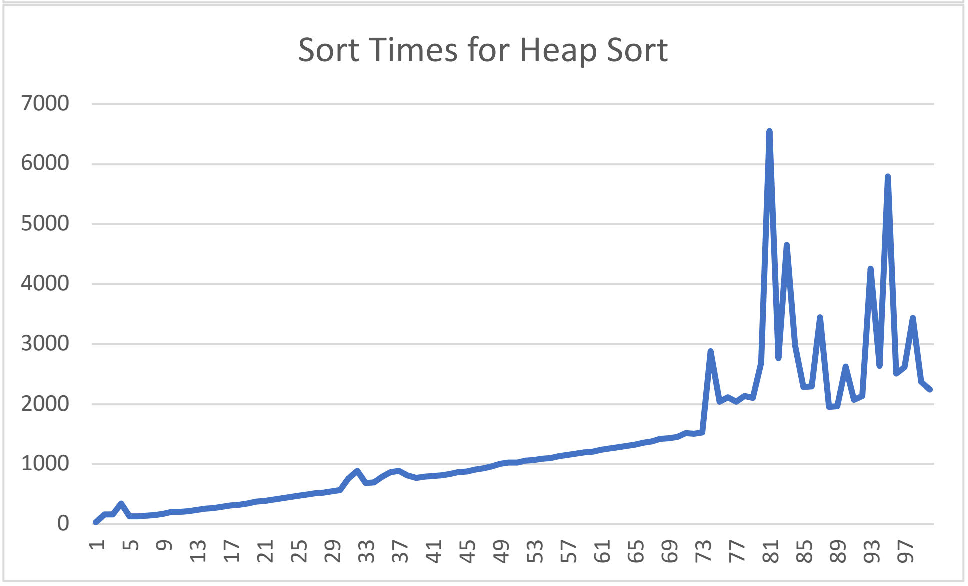
Part B

In Part B, we compare the performance of the bubble sort algorithm with the heap sort algorithm. The bubble sort is one of the slower sorting algorithms, with a time complexity of O(n^2). This is because it must check so many indices and swap them if necessary. While on the other hand, the heap sort is more efficient with a time complexity of O($nlogn$). The heap sort algorithm is more strategic in nature with its methodological way of organizing the data. The bubble sort is more brutish, it forces its way through the array swapping values- it's not very strategic.

Visually, can see in bubble sort's graph its shaped quadratically. It's the slowest algorithm compared in the project. Its times can reach many million nanoseconds! In comparison, the heap sort only reaches a few thousand nanoseconds in our timings. This makes the distinction between the two data structures clear- heap sort is the better data structure.

All speeds are still timed in nanoseconds.


Sort Times for Bubble Sort
4500000
4000000
3500000
3000000
2500000
2000000
1500000
1000000
500000
0
1
5
9
13
17
21
25
29
33
37
41
45
49
53
57
61
65
69
73
77
81
85
89
93
97
101


Sort Times for Heap Sort
7000
6000
5000
4000
3000
2000
1000
0
1
5
9
13
17
21
25
29
33
37
41
45
49
53
57
61
65
69
73
77
81
85
89
93
97