File 4 included 125,000 random inserts and the same amount of random deletes. The Binary Search Tree preformed well for file 4 and did not need to be terminated like the rest of the files operations. The BST algorithm has a worst case scenario Big – O of O (n) and it preformed well since the insertions were random, it did not need to traverse through every single element and so did not. The AVL tree with a Big 0 of 0(log (n)) was a lot slower than the BST this time for File 4 because it was self-adjusting and since the insertions and deletions were random, the amount of adjustments could have been (and was) very large. The Splay Tree, with the same Big-O as the AVL worked slightly faster than the AVL tree since after each insertion and deletion the most recent element was brought to the top but since the insertions and deletions were both randomized, the zig zag and zig zig operations took longer making the Splay tree also work slower than it did for the other files. The binary heap, also with a Big-O of O (log (n)) preformed the best out of all the tree operations because it was a min heap did not need to traverse the tree in order to delete since it used a deleteMin function for every deletion, which deleted the smallest element already at the top of the heap giving it a deletion O (1). The BTree ADT with M=3, L=1 had a big O of O (log (n)) for both insertion and deletion and preformed slightly worse than the rest of the trees. The skip list ADT had a Big-O of O (n) but preformed similar to the AVL and Splay trees because the insertion and deletions were randomized which helped it realize its average O (log (n)).

The BTree ADT was tested using four different M and L combos for file 4. The first was with (M=3, L=1). The time complexity for each combo’s insertion and deletion was O (m \* log (m) \* log (n/L)) but the deletion operations required more frequent node splitting which is why file 2 and 3 were slower than file 1 for BTree combos. The big O came out to be O (7.7) for combo 1. For combo 2 (M=3,L=200) it came out to be O (4.4), which is why it was faster. Combo 3 (M=1000,L=2) resulted in a very large big O, which is why it was 3 times slower than combo 2 was. Combo 4's (M=1000, L=200) big O was also large but smaller than combo 3’s. File 4 took longer time for each of the combos compared to files 1,2, and 3 because the randomized inserts and deletions had the potential to cause more frequent element trading to sibling nodes and splits.

The Separate chaining hash table data structure was tested with 5 different load factors: .5,1,10,100,1000 respectively. The big –O for the insertion and deletion from the separate chaining hash table in this program was O (λ + 1) due to a find in the insertion function because it used a linked list. Which is why we see similar times for the load factors of .5,1 and 10 but as the load factors got bigger (100 and 1000) so did the amount of time it took to run the program because this means the tables are getting smaller resulting in more collisions which calls for more frequent traversing through linked lists with multiple elements. Also, the separate chaining hash table worked worse for file 4 than it did for the other files because the deletions and insertion were random and so had the potential to have more collisions.

The Quadratic probing hash table data structure was also tested with 5 load factors: .1, .25, .5, 1, 2 respectively. The big – O for insertion and deletion for the quadratic probing hash was O (1/1-λ). The runs with the smaller load factors had the faster run times but as seen by the data, the times didn’t differ by much since they were all around O (1). Hence, the runs with the higher load factors had the slower run times. Also, since the “Load Factor = # of elements / Table Size”. The bigger the load factor, the smaller the table size which meant more frequent rehashing operations also resulting in the higher load factor operations to be slower. File 4 was slightly slower than the rest of the files because similar to the separate chaining hash, the random inserts had more potential for collisions.

Since the separate chaining hash table used a linked list, it had O (λ + 1) and so was slower than the quadratic probing hash for the same load factors (.5 and 1). The Quadratic pointer probing hash performance across its tested load factors was similar to that of the regular quadratic probing hash. However, it was slower than the quadratic probing hash. Even though both data structures used an array and similar find methods to place and delete the elements into the tables and had similar big O for insertion and deletion (O (1/1- λ), the pointer probing hash had an extra comparison within its find method checking for a null pointer making it a tad bit slower than the quadratic probing hash table. The quadratic probing hashes did not however use lazy deletion like the separate chaining hash table but also did not rehash for deletion and so were ultimately faster than the separate chaining hash table for all the files.