File 3 included 125,000 inserts in order from 1 to 125,000 and then 125,000 deletions in order from 125,000 to 1. From the Tree based data structures, the Binary Search Tree preformed the worst for File 3. The program was taking longer than 5 min and was terminated. The BST algorithm has a worst case scenario Big – O of O(n) and I believe it was fully realized because numbers were in increasing order, each insertion created a new right child and consequently the next insertion had to be done after a traversal through every single element already in the list. The AVL tree was a lot faster due to its Big-O of O(log (n)) and since it is self balancing unlike the BST. The Splay Tree, with the same Big-O as the AVL worked almost twice as fast as the AVL tree since after each insertion and deletion the most recent element was brought to the top (and consecutive elements nearby) and so there was very little traversal and element access. The binary heap, also with a Big-O of O(log(n)) preformed worse than the splay tree even though they share the same Big O for both insertion and deletion, because the deletion operation for the binary heap was slower than the splay tree which had fewer amounts of traversals to access the element to be deleted. The BTree ADT with M=3 , L=1 had a big-O of O(log(n)) and preformed around the same than the rest of the trees (except the binary search tree) . The skip list ADT had a Big-O of O(n) and surprisingly preformed well with a time close to the rest of the tree times. This was because deletions took less time than insertions which is why the skip list preformed worse for file 1 than it did for files 2 and 3.

The BTree ADT was tested using four different M and L combos for file 3. 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) but was still slower (for file 3) than combo 1 because the deletions were ordered so that the elements most recently put into the tree were deleted first. Combo 3 (M=1000,L=2) resulted in a very large big-O which is why it was almost 3 times slower than combos 1 and 2. Combo 4's (M=1000, L=200) big-O was also large but smaller than combo 3’s.

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. Also, the separate chaining hash table worked better for file 3 than it did for file 2 because the deletions were from 125,000 and since the list was first in last out, the element finding for the deletions took less time. Also deletions was faster than the insertion operation because a deletion did not require traversing through an entire list to see if the element existed.

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. Also again, file 3 was slightly faster than file 1 because the quadratic probing hash used lazy deletion and so the deletion times were faster than the insertion times.

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. But 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 file 3.