In File1.dat, we are inserting numbers in order from 1 to 2,500,000. First, let us compare the trees and the skip list. The slowest data structure was the binary search tree which did not finish. It was the slowest because every number that we insert is greater than the previous number that we inserted, so the binary search tree will always insert to the right of the tree, effectively creating a long LinkedList. This makes a single insertion O(n), and a series of insertions O(n2).

Next, we can compare the skip list, the AVL tree and the splay tree. Even though the skip list and the AVL tree have the same time complexity, O(nlog(n)), the AVL tree was noticeably slower. The reason is because the AVL tree must rebalance the tree at every other insertion, since we are always inserting values to the right side of the tree. On the other hand, the skip list does not have to rebalance itself at all, so it uses significantly less constant operations at every insertion, making it faster. Finally, the splay tree was the fastest of these three because it was O(1) for each individual insertion. This is because we only need 1 ‘zig left’ rotation to insert the larger element to the top of the tree, since the element is always inserted to the right side of the tree and rotations cause the other elements to always be on the left side of the tree.

Next, the BTree, even though the Big-O was the same, had different speeds for different M’s and L’s. The fastest M and L was when M=3 and L=200. This makes sense since M=3 and L=1 would require too much splitting and allocation of memory for new internal nodes and leaf nodes. M=1000 and L=2 has a similar problem, but in addition, splits of an internal node would be very costly because you would have to transfer half of the leaf nodes (500 leaf nodes) to the new internal node each time a split occurs. M=1000 L=200 would also have to transfer 500 leaf nodes at a split, however it is slightly faster than M=1000 L=2 because splits are less common since each leaf node can hold more items. Finally, the last thing to note is that the BTree performs very well on insertions. This is because whenever it inserts a new element in to the BTree, it compares starting from the largest element of the array. Since we are inserting larger values each time, the comparisons will be O(1) for each insertion. That’s why we dropped the +L from the Big-O notation. In general, for BTrees, a larger M slows performance, so L should be greater than M to maximize speed.

Finally, the Binary Heap was the fastest test for File1.dat because no reordering is necessary. We are inserting them one by one into the bottom of tree (array), without ever having to shuffle the newly inserted data structure back to the top, since we insert the smallest values first. We only need a one comparison with its direct parent to know that that’s where the element belongs. This makes it a very fast O(n) operation for File1.dat.

Now, we must compare the hash tables for File1.dat. For separate chaining, each number is sequentially added to the head of a linked list at the index of the hash table. Therefore, the time complexity should be O(1) in general. However, in this code, each insertion takes O(λ) time because for some strange reason the code checks to see if a duplicate element already exists in the linked list. *(SeparateChaining.cpp, line 57)* This means that on average you will have to traverse through λ items in the linked list to see if it already exists in the hash table before you can insert it. That’s why the elements with the largest load factor (λ) are the slowest – searches for duplicates slow it down. Finally, there was one unexpected result in separate chaining that is not explained by λ. For some reason, a load factor of 0.5 took longer than a load factor of 1.0. This might have something to do with cache misses. The test with a load factor of 0.5 requires a 5,000,000-element array, while a load factor of 1.0 requires a 2,500,000-element array. These are very large arrays, so the difference in time might be because all the elements of the array cannot fit into the cache at once.

Next, we must compare quadratic probing with quadratic probing pointer hash tables. For the quadratic probing, the times increase as the load factor gets bigger. This makes sense, because if the load factor is greater than 0.5, the table would have to rehash. For example, the hash table that has a load factor of 2.0 would have to rehash 4 times, leading to slower times. However, unlike the separate chaining, larger table sizes did not affect times. A load factor of 0.1 was just as efficient as a load factor of 0.25. This makes sense, since in both tests minimize collisions. For the quadratic probing pointer hash table, the same general pattern holds. The larger the load factors, the longer the times, and again this makes sense because there were more collisions and rehashing must occur. However, the major difference between quadratic probing and quadratic probing using pointers is that it must dynamically allocate memory every time we rehash to copy the objects over to the new hash table, and we have to deallocate the old objects. Compare this to quadratic probing where the values are stored directly into an array that already exists. Therefore, dynamic memory significantly slows down the insertions for the pointer hash table. Thus, the pointer hash has two competing forces. When the load factor increases, the run times slow down, and when the amount of dynamic memory allocation increases, the run times slow down. The best results occurred in the middle where λ=0.5.

Finally, comparing separate chaining, quadrating probing and quadratic probing using pointers, we can see that in general, separate chaining is the slowest, quadratic probing using pointers is the second slowest, and the regular quadratic probing is the fastest. This makes sense because separate chaining requires traversing through all the linked lists to see if the element already exists before inserting, while both versions of quadratic probing do not. Also, the quadrating probing using pointers is slower than regular quadratic probing because it is creating objects using dynamic allocation and it must dynamically allocate new objects every time it rehashes. These reasons account for the differences in the hash table times.

In File2.dat, we had 1,250,000 insertions in order from 1 to 1,250,000, followed by 1,250,000 deletions in order from 1 to 1,250,000. For the insertions, all of the reasons that we gave for File1.dat also hold for File2.dat. The only difference is that we are inserting less numbers. However, we still need to discuss how the deletions affected the times.

As mentioned earlier, the binary search tree mimics a linked list since it always inserts the numbers to the right of the tree. After the insertions finish, the root of the binary search tree contains a 1, followed by a 2, 3, 4 etc… Therefore, deletions will be really fast - an O(1) operation, since the next deleted node will always be at the root of the tree. Even though the test did not finish, I suspect that File2.dat would be faster than File1.dat for the binary search tree. However, it was still the slowest test because insertions are O(n2).

Next, we can compare deletions for the skip list, the AVL tree and the splay tree. First comparing the skip list and AVL tree, we notice that the big O is still O(log(n)), just like it was for the insertions, since in a worst-case scenario we would have to traverse the log(n) items to get to the item that we are trying to delete. However, if we compare the times for File1.dat and File2.dat, we notice that File2.dat finished faster for both data structures. The reason for this is that the height of the AVL tree and the length of the skip list is now log(1250000) instead of log(2500000). The shorter height means less traversals for each insertion and deletion, which results in faster times. For AVL trees, it will take either log(n) or log(n-1) steps to delete each element, since the smallest value will end up furthest away from the root of the tree. Also note that for the skip list, the deletions are always going to occur at the leftmost node, so there will be less than log(n) traversals to find the first item. All you would have to do is keep going down the height of the skip list until you reach the first node. Finally, the splay tree was the fastest from the trees because insertions are O(1). Once all the insertions are complete, the splay tree will have all of its nodes on the left side. Therefore, the first deletion would take O(n) since you would have to traverse through n items just to get to 1. However, once you bring the 1 up to the root and delete it, splaying it up to the root will bring all the numbers close to 1 also closer to the root, so 2,3,4 etc. will be near the root and deleting them will be an O(1) operation. Because of amortized analysis, the total series of deletions will still be O(n).

The deletions in the BTree are worse than the insertions. The big O is n(mlogm(n/l) + l) since searching for an item begins at the maximum value in the internal nodes (*InternalNode.cpp, line 177*), but we’re trying to delete the minimum each time. Also, whenever we delete the minimum value, we must shift L-1 items down one. Shifting L-1 items is O(l). That’s why we add +L to our big O for deletions. M=3 L=1 and M=3 L=200 are significantly faster than M=1000 L=2 and M=1000 L=200 for this reason. Deleting elements with a high M means shifting many items in the internal nodes. Also, just like with insertions, deleting nodes and transferring children to a new parent explains why M=3 L=1 is faster than M=3 L=200 and why M=1000 L=200 was faster than M=100 L=2, since a higher L means more elements per leaf node.

For binary heap, each deletion is O(logn) because in a worst-case scenario, deleting an item and replacing it with the right minimum might require a lot of swapping to get it back to the state where the top node contains the smallest number and the lower nodes contain larger numbers. Overall, this might take log(n) swaps (the height of the tree). Nevertheless, it is still one of the fastest tests because insertions are O(1).

Finally, we must compare the deletions of the hash tables. In separate chaining, we are trying to delete items that we inserted first. Because we inserted them first, they will be at the ends of the linked lists, so we will have to traverse the entire load factor + 1 to delete them. This results in O(λ+1). For either quadratic probing, a single deletion is O(1) and it is not affected by the load factor. The times are different only because insertion speed depends on the load factor. It takes the same time to delete all the elements in both data structures. However, calling destructors for quadratic probing using pointers slows it down a bit.

The same reasoning that we used to explain insertions for File1.dat also explains insertions for File3.dat. However, we still need to explain how the deletions in the reverse order affect the times, and we must compare File3.dat to File2.dat. The BST is more inefficient for File3.dat than File2.dat. The BST acts like a linked list with the smallest element at the head node and the largest element at the tail, so you need to traverse the entire list to delete an item, so deletions are O(n2). This explains why the test took too long. It is incredibly inefficient for both insertions and deletions. However, I predict that this test would have been faster than File1.dat, since the height of the ‘tree’ is 1,250,000 instead of 2,500,000.

Next, for the skip list, AVL tree and splay tree, the skip list was O(log(n)) because now it’s deleting from the end of the list. Since there are horizontal transversals involved in addition to movements down the skip list, it will take on average more moves to reach the last element than the first element. This explains why the skip list was slower for File3.dat than File2.dat. The AVL tree is slower for File3.dat than File2.dat. It is doing the same procedure, only a mirror version of it - starting from the right side of the tree instead of the left. Having to transverse down log(n) items for each deletion makes it O(log(n)). However, because it’s always deleting a maximum node, it doesn’t have to reposition as many children or rebalance as often. This explains why File3.dat was slower than File2.dat for AVL trees. Finally, the splay tree was incredibly fast for File3.dat. This makes sense, since insertions are O(1), and deletions are also O(1) since we are deleting the largest element, which is always found at root. Compared to File3.dat, File2.dat was slower for the splay tree because it had to work hard at the beginning to bring items to the top. However, once it did, amortized cost became constant O(1). In File3.dat, the splay tree always deleted at the root node, so it didn’t have to work to bring the nodes to the top.

The BTree deletions for File3.dat were faster than for File2.dat. This makes sense, since in File3.dat, we are deleting from the top of the list, so we never shift elements in the leaf nodes (*LeafNode.cpp, line 135*). However, the times are still slower than File1.dat because finding the index in the leaf node for insertion starts at count-2 and goes to 0 (*LeafNode.cpp, line 61*). However, for deletions, it starts searching for that index at 0 for the leaf nodes (*LeafNode.cpp, line 128*). Since we are either inserting or deleting a maximum value, it will be located closer to count-2. Thus, for File3.dat deletions, the search for the correct index takes longer. However, internal node searching is still extremely fast for both File1.dat and File3.dat (an O(1) operation), since it begins searching at count-2 for both insertions and deletions.

For File3.dat, the binary heap ignores the specific deletions, since it only has a deleteMin() function. As discussed for File2.dat, deletions at the root are a O(log(n)) operation. Both File2.dat and File3.dat act exactly the same way for insertions and deletions.

Finally, for the hash tables, the same arguments that we made for File2.dat also apply to File3.dat. The only difference is that deletions are O(1) for separate chaining instead of O(λ+1), because the largest numbers will always be at the front of the list. That’s why we see a slight speed improvement for separate chaining in File3.dat compared to File2.dat. Quadratic probing deletions take the same time regardless of the order in which you delete them. That’s why the times are very similar between File1.dat and File2.dat.

Finally, we’re left with File4.dat, which inserts and deletes random numbers. The BST was the fastest tree. This makes sense because the insertions and deletions are random, so it should automatically create a fairly balanced tree, and random deletions should automatically keep the tree balanced. AVL trees and splay trees are slower because they perform unnecessary checks at each step to make sure that the tree is balanced. These checks are not needed because the randomness keeps the tree balanced automatically. Next, the skip list got a lot slower because the data is no longer conveniently found at the beginning (File2.dat) or end (File3.dat) of the split list. It’s like somewhere in the middle, so this slows down performance.

As for the BTree, performance was the worst compared to the other files. This is because M and L no longer can take advantage of the way in which we are searching and inserting/deleting items. On average, an insertion will require M/2 checks to find the correct location for insertion, and M/2 elements must be shifted so the new element could be inserted into the proper location. The same idea applies to the leaf nodes.

Next, binary heap was slower for File4.dat because insertions are no longer O(1) since numbers are randomly inserted. In a worst-case scenario, the insertions will take O(log n) due to shuffling. However, the average case for insertion is still O(1), since there’s a 50% chance the inserted element will be larger than the current element, so the expected value is 1\*.5 + 2\*.25 + 3\*.125... = 2. It takes 2 steps on average to insert a random value into a binary heap.

Finally, the explanations for File2.dat and File3.dat also apply to File4.dat. In general, the order does not matter for insertions and deletions in hash tables.