ADT	File #	Time
SkipList	File 1- File 2- File 3- File 4-	2.51743 1.9138 1.98224 6.35026
Binary Search Tree	File 1- File 2- File 3- File 4-	> 5min > 5min > 5min > 5min
AVL Tree	File 1- File 2- File 3- File 4-	2.80077 2.32432 2.46857 7.17726
Splay Tree	File 1- File 2- File 3- File 4-	1.3857 1.24147 1.15236 4.87556

ADT	File #	Lambda & Time	Lambda & Time	Lambda & Time	Lambda & Time	Lambda & Time
BTree	File 1- File 2- File 3- File 4-	M = 3; L = 1 2.78361 2.9349 2.14673 6.52721	M = 3; L = 200 1.76446 2.61082 2.43456 3.42218	M = 1000; L = 2 6.81547 9.68687 7.11289 11.0496	M = 1000; L = 200 4.57759 5.27524 5.69937 5.97571	N/A
Separate Chaining Hash	File 1- File 2- File 3- File 4-	λ = 0.5 1.8583 1.53088 1.41431 2.44045	λ = 1 1.80868 1.36461 1.28113 2.57569	λ = 10 1.69772 1.35626 1.36789 2.51082	λ = 100 3.5192 2.90571 1.92848 8.29333	λ = 1000 29.8878 24.9964 13.4805 73.7917
Quadratic Probing Hash	File 1- File 2- File 3- File 4-	λ = 2 1.39157 1.1787 1.12032 1.50697	λ = 1 1.22982 1.07401 1.06934 1.49913	λ = 0.5 1.20569 0.993828 0.995879 1.37519	λ = 0.25 1.00533 1.00015 0.991821 1.37613	λ = 0.1 0.998194 0.98049 1.0323 1.39573
Binary	File 1-	N/A	N/A	N/A	N/A	N/A

Heap	1.00016 File 2- 1.61496 File 3- 1.58857 File 4- 2.1123					
Quadratic Probing Ptr Hash	File 1- File 2- File 3- File 4-	$\lambda = 2$ 2.02192 1.46642 1.36962 2.25195	$\lambda = 1$ 1.7557 1.3489 1.35009 2.07393	$\lambda = 0.5$ 1.45088 1.2366 1.20023 1.99871	$\lambda = 0.25$ 1.58224 1.38087 1.28175 2.09164	$\lambda = 1$ 1.79935 1.38478 1.30819 2.228

BIG O-Table

ADT	Individual Insertion	Individual Deletion	Entire Series of Insertions	Entire Series of Deletions	Entire File
Binary Search Tree	File 1- O(n) File 2- O(n) File 3- O(n) File 4- O(n)	File 1- N/A File 2- O(1) File 3- O(n) File 4- O(log n)	File 1- O(n) File 2- O(n) File 3- O(n) File 4- O(n)	File 1-N/A File 2- O(1) File 3- O(n) File 4- O(log n)	File 1- O(n^2) File 2- O(n^2) File 3- O(n^2) File 4- O(n^2)
AVL Tree	File 1- O(log n) File 2- O(log n) File 3- O(log n) File 4- O(log n)	File 1- N/A File 2- O(log n) File 3- O(log n) File 4- O(log n)	File 1- O(nlog n) File 2- O(nlog n) File 3- O(nlog n) File 4- O(nlog n)	File 1- N/A File 2- O(nlog n) File 3- O(nlog n) File 4- O(nlog n)	File 1- O(nlog n) File 2- O(nlog n) File 3- O(nlog n) File 4- O(nlog n)
Splay Tree	File 1- O(n) File 2- O(n) File 3- O(n) File 4- O(n)	File 1- N/A File 2- O(n) File 3- O(1) File 4- O(log n)	File 1- O(n) File 2- O(n) File 3- O(n) File 4- O(n)	File 1- N/A File 2- O(n) File 3- O(1) File 4- O(log n)	File 1- O(n) File 2- O(n) File 3- O(n) File 4- O(n)
Skiplist	File 1- O(log n) File 2- O(log n) File 3- O(log	File 1- N/A File 2- O(log n) File 3- O(log n)	File 1- O(nlog n) File 2- O(nlog n) File 3- O(nlog	File 1- N/A File 2- O(nlog n) File 3- O(nlog n)	File 1- O(nlog n) File 2- O(nlog n) File 3- O(nlog

	n) File 4- O(log n)	File 4- O(log n)	n) File 4- O(nlog n)	File 4- O(nlog n)	n) File 4- O(nlog n)
BTree	File 1- O(n) File 2- O(n) File 3- O(n) File 4- O(n)	File 1- N/A File 2- O(1) File 3- O(n) File 4- O(log n)	File 1- O(n) File 2- O(n) File 3- O(n) File 4- O(n)	File 1-N/A File 2- O(1) File 3- O(n) File 4- O(log n)	File 1- O(n^2) File 2- O(n^2) File 3- O(n^2) File 4- O(n^2)
Separate Chaining Hash	File 1- O(1) File 2- O(1) File 3- O(1) File 4- O(1)	File 1- N/A File 2- O(λ) File 3- O(λ) File 4- O(λ)	File 1- O(1) File 2- O(1) File 3- O(1) File 4- O(1)	File 1- N/A File 2- O(λ) File 3- O(λ) File 4- O(λ)	File 1- O(λ) File 2- O(λ) File 3- O(λ) File 4- O(λ)
Quadratic Probing Hash	File 1- O(1) File 2- O(1) File 3- O(1) File 4- O(1)	File 1- N/A File 2- O(λ) File 3- O(λ) File 4- O(λ)	File 1- O(1) File 2- O(1) File 3- O(1) File 4- O(1)	File 1- N/A File 2- O(λ) File 3- O(λ) File 4- O(λ)	File 1- O(λ) File 2- O(λ) File 3- O(λ) File 4- O(λ)
Binary Heap	File 1- O(log n) File 2- O(log n) File 3- O(log n) File 4- O(log n)	File 1- N/A File 2- O(log n) File 3- O(log n) File 4- O(log n)	File 1- File 2- O(n logn) File 3- O(nlog n) File 4- O(nlog n)	File 1- N/A File 2- O(nlog n) File 3- O(nlog n) File 4- O(nlog n)	File 1- O(n) File 2- O(n) File 3- O(n) File 4- O(n)
Quadratic ProbingPtr Hash	File 1- O(1) File 2- O(1) File 3- O(1) File 4- O(1)	File 1- N/A File 2- O(λ) File 3- O(λ) File 4- O(λ)	File 1- O(1) File 2- O(1) File 3- O(1) File 4- O(1)	File 1- N/A File 2- O(λ) File 3- O(λ) File 4- O(λ)	File 1- O(λ) File 2- O(λ) File 3- O(λ) File 4- O(λ)

ESSAY

In the Skiplist, it is basically an organization of linked list. So, in file 1 we see that the file is inserting a million integers, from 0 to one less than a million. It is a linked list, so it goes from 0 to the million, which takes a little bit of time. The File 2 goes from 0 to half a million and then deletes the values that were inserted in the order that they were inserted. The File 3 has a half million insertions, and the deletions go from the bottom to the top. The File 4 has a half million integers inserted randomly. The deletion of these random numbers take longer than the rest of the operation, because the random insertions/deletions messes with where the pointer has to go to delete the number.

Binary Search Tree for File 1 takes a huge amount of time because it's a huge linked list on the right, that goes deep in the RAM. It takes a while to traverse the whole list. For File 2, it takes a while to create the list and also takes a while to delete the list. Since, we are inserting about half a million integers and deleting those integers, it is reasonable to assume that it will take a while. For File 3 and File 4, it follows from the previous two files that it will also take a while, because it becomes a huge linked list and is difficult to traverse and delete the value that has been randomly allocated into the list.

AVL tree works by balancing the number inserted, however the number inserted in File 1 the numbers are all in increasing order so that essentially means that accessing the disk is gonna take a while. Since the file is all insertions, it's gonna take the longest because of the creation of pointers and the allocation of those pointers. The next file which is File 2 is insertions in order and then deletions in that order, since we aren't creating and allocating as many pointers as those in File 1. File 3 is essentially similar to File 2, but the difference between the files is the order in which the numbers are deleted. Since the order is different for file 3 and it is the opposite way in

which the pointers are allocated then it will take longer. File 4 takes the longest amount of time, because allocation of pointers takes a while and it's random, since it is random the deletion also takes a while because the program has to search for that pointer, from there the pointer also has to be deleted.

Splay Tree works by splaying up the recently inserted items. The items inserted are put in and from there they are splayed up to the Tree. It's no wonder why File 1 takes a short amount of time. It takes a short amount of time because of the memory allocation of the integer values in a pointer. File 2 has to allocate less memory space because File 2 is going to half a million integers. The deletions would take a while because it is deleting in the order, in which it is inserted. That process takes a while because the last items on the splay are the items that are being deleted. File 3 is similar to File 2 the only difference is the order of deletions of the inserted integers. Since File 3 is deleted in the opposite order in which it is inserted it is much faster than File 1 and much faster than File 2. File 4 takes the longest, because of the random order of insertion and the random order of deletion. It's that way because of the allocation of pointers needed for the integer values and the deletion of said values.

BTree is similar to the Binary Tree in many respects other than the time. File 4 takes the longest no matter what the M and L are. BTree makes the pointers by allocating space to the root. The bigger the root the more disk accesses that will need which in turn takes up time that is why in general the BTree with the biggest M takes the largest time. The opposite result is true, because the file with the largest amount of leaf nodes also takes the shortest amount of time. That is why in general the files with a large amount of leaf nodes take a shorter amount of time compared to those with small leaf nodes. The M and L that takes the largest amount of time, are when the M = 1000 and the L = 2. That means that the disk has to create a 1000 root nodes some

of which might not be used. In each of these nodes they are stringed together or are siblings of each other. At each sibling there are two children pointers.

Separate Chaining Hash works miracles by creating new pointers in the mod of the lambda, which we are allocating the new values into. The generality with the values is that the bigger the lambda the longer it takes to traverse the list and the longer the time it takes to add the values to the actual hash array, because it takes a while to locate and to be added into the hash. This is the general theme for separate hashing. It's easier to allocate the pointers, instead of potentially traversing the list and adding the values into the appropriately modded spot.

Quadratic Probing Hash works similar to the separate chaining hash but the difference lies in the way that the separate chaining hash and the quadratic probe deal with the collisions or cache misses. The quadrating probing, however, works differently. It works by inserting the value at the mod of the table size. If the slot is filled, then the integer is added to a 1 then modded, if there is still a cache miss. It is then added to a 4, then modded. It keeps being added to values and keeps being modded by the table size until it hits a free cache. That process happens for all the lambdas, that is why the lambdas that are shorter have a shorter time to access the cache and see if there is a miss to create a bigger cache and the from that a new hash table is created. This new hash table is bigger and doesn't hit as much caches as the previous hash table. This is also why the lambdas with the shorter numbers take much less time than those with larger numbers. Those with larger numbers have to traverse the modded list.

Binary Heap is short in the sense that it takes a short amount of time to work. It works by allocating memory to the heap and is essentially a linked list. The shortest binary heap which works is the one which is all insertions. This works by inserting values into the pointer and creating and allocating more pointers to the long linked list which is caused by inserted numbers.

The largest are those that take the longest amount of time to allocate and delete, which is File 4 because of the randomness of the insertion and deletion there is no proper way to delete or point to the actual value in the list and then travers that list in a proper way.

Quadratic Probing Pointer Hash works similar to the separate chaining hash but the difference lies in the way that the separate chaining hash and the deal with the collisions or cache misses. The quadrating probing, however, works differently. It works by inserting the value at the mod of the table size. If the slot is filled, then the integer is added to a 1 then modded, if there is still a cache miss. It is then added to a 4, then modded. It keeps being added to values and keeps being modded by the table size until it hits a free cache. Then it allocates memory in the stack. That process happens for all the lambdas, that is why the lambdas that are shorter have a shorter time to access the cache and see if there is a miss to create a bigger cache and then from there a new hash table is created. This new hash table is bigger and doesn't hit as much caches as the previous hash table. This is also why the lambdas with the shorter numbers take much less time than those with larger numbers. Those with larger numbers have to traverse the modded list and are put in via pointers.