|  |  |  |
| --- | --- | --- |
| File # | ADT # | Time (in seconds unless mentioned otherwise) |
| File1.dat | 6 (Skip List) | 0.77602 |
| File2.dat | 6 | 0.642524 |
| File3.dat | 6 | 0.733271 |
| File4.dat | 6 | 3.02888 |
| File1.dat | 7 (Binary Search Tree) | >5minutes |
| File2.dat | 7 | >5minutes |
| File3.dat | 7 | >5minutes |
| File4.dat | 7 | 1.63552 |
| File1.dat | 8 (AVL trees) | 1.31107 |
| File2.dat | 8 | 0.995486 |
| File3.dat | 8 | 1.04426 |
| File4.dat | 8 | 2.47701 |
| File1.dat | 9 (Splay Trees) | 0.34065 |
| File2.dat | 9 | 0.278832 |
| File3.dat | 9 | 0.226742 |
| File4.dat | 9 | 2.25398 |
| File1.dat | 13 (Binary Heap) | 0.177683 |
| File2.dat | 13 | 0.516528 |
| File3.dat | 13 | 0.516594 |
| File4.dat | 13 | 0.633893 |

**For ADT# 10 (B Tree)**

|  |  |  |  |
| --- | --- | --- | --- |
| File # | M | L | Time (in seconds) |
| File2.dat | 3 | 1 | 1.49947 |
| File2.dat | 3 | 200 | 1.63145 |
| File2.dat | 1000 | 2 | 6.54626 |
| File2.dat | 1000 | 200 | 3.33739 |
| File1.dat | 3 | 1 | 0.884175 |
| File1.dat | 3 | 200 | 0.877121 |
| File1.dat | 1000 | 2 | 4.34338 |
| File1.dat | 1000 | 200 | 2.44051 |
| File4.dat | 3 | 1 | 3.02644 |
| File4.dat | 3 | 200 | 2.20074 |
| File4.dat | 1000 | 2 | 7.15857 |
| File4.dat | 1000 | 200 | 3.52807 |
| File3.dat | 3 | 1 | 0.8048 |
| File3.dat | 3 | 200 | 1.28163 |
| File3.dat | 1000 | 2 | 4.24302 |
| File3.dat | 1000 | 200 | 3.13529 |

**For #12(Quadratic Probing Hashes), #14(Quadratic Probing Pointers)**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| File # | ADT # | Load 1  (λ =2)  Timings (in sec) | Load 2  (λ=1)  Timings (in sec) | Load 3  (λ=0.5)  Timings (in sec) | Load 4  (λ=0.25)  Timings (in sec) | Load 5  (λ=0.1)  Timings (in sec) |
| File1.dat | 12 | 0.345951 | 0.300765 | 0.210778 | 0.210932 | 0.21113 |
| File1.dat | 14 | 0.432999 | 0.383601 | 0.308396 | 0.328762 | 0.404417 |
| File2.dat | 12 | 0.270154 | 0.247354 | 0.202491 | 0.202328 | 0.202352 |
| File2.dat | 14 | 0.326045 | 0.303272 | 0.258647 | 0.267815 | 0.304499 |
| File3.dat | 12 | 0.270078 | 0.247707 | 0.20307 | 0.203225 | 0.20279 |
| File3.dat | 14 | 0.329064 | 0.305817 | 0.261822 | 0.268412 | 0.305751 |
| File4.dat | 12 | 0.341587 | 0.318846 | 0.266957 | 0.268905 | 0.268928 |
| File4.dat | 14 | 0.667158 | 0.651205 | 0.556983 | 0.464828 | 0.495578 |

**For #11(Separate Chaining Hash Tables)**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| File# | Load 1  (λ =0.5)  Timings (in sec) | Load 2  (λ =1)  Timings (in sec) | Load 3  (λ =10)  Timings (in sec) | Load 4  (λ =100)  Timings (in sec) | Load 5  (λ =1000)  Timings (in sec) |
| File1.dat | 0.391123 | 0.379297 | 0.410957 | 4.06215 | 45.1938 |
| File2.dat | 0.269326 | 0.261077 | 0.30851 | 2.84972 | 25.0639 |
| File3.dat | 0.268786 | 0.261438 | 0.281861 | 1.89968 | 20.0691 |
| File4.dat | 0.650349 | 0.791851 | 1.12292 | 5.24416 | 45.2764 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| FILE#.dat | ADT# | INDIV  INSERTIONS | INDIV  DELETIONS | ALL  INSERTIONS | ALL  DELETIONS | WHOLE FILE |
| 1 | 6(Skip List) | O(logN) | - | O(NlogN) | - | O(NlogN) |
| 2 | 6 | O(logN) | O(logN) | O(NlogN) | O(NlogN) | O(NlogN) |
| 3 | 6 | O(logN) | O(logN) | O(NlogN) | O(NlogN) | O(NlogN) |
| 4 | 6 | O(logN) | O(logN) | O(NlogN) | O(NlogN) | O(NlogN) |
| 1 | 7(Binary Search Tree) | O(N) | - | O(N2) | - | O(N2) |
| 2 | 7 | O(N) | O(1) | O(N2) | O(N) | O(N2) |
| 3 | 7 | O(N) | O(N) | O(N2) | O(N2) | O(N2) |
| 4 | 7 | O(log N) | O(log N) | O(N log N) | O(N log N) | O(N log N) |
| 1 | 8 (AVL Trees) | O(logN) | - | O(NlogN) | - | O(NlogN) |
| 2 | 8 | O(logN) | O(logN) | O(NlogN) | O(NlogN) | O(NlogN) |
| 3 | 8 | O(logN) | O(logN) | O(NlogN) | O(NlogN) | O(NlogN) |
| 4 | 8 | O(logN) | O(logN) | O(NlogN) | O(NlogN) | O(NlogN) |
| 1 | 9 (Splay Trees) | O(1) | - | O(N) | - | O(N) |
| 2 | 9 | O(1) | O(N) for first delete as 1 is right at bottom  O(1) for all other deletes as it moves after 1 is splayed to the root | O(N) | O(N) | O(NlogN)  (average of deletes and insertions) |
| 3 | 9 | O(1) | O (1) | O(N) | O(N) | O(N) |
| 4 | 9 | O(logN) | O(logN) | O(NlogN) | O(NlogN) | O(NlogN) |
| 1 | 10 (BTree) | O(L+MlogM/2(N/L)) | - | O(NM\*logM/2(N/L) + NL) | - | O(NM\*logM/2(N/L) + NL) |
| 2 | 10 | O(L+MlogM/2(N/L)) | O(L+MlogM/2(N/L)) | O(NM\*logM/2(N/L) + NL) | O(NM\*logM/2(N/L) + NL) | O(NM\*logM/2(N/L) + NL) |
| 3 | 10 | O(L+MlogM/2(N/L)) | O(L+MlogM/2(N/L)) | O(NM\*logM/2(N/L) + NL) | O(NM\*logM/2(N/L) + NL) | O(NM\*logM/2(N/L) + NL) |
| 4 | 10 | O(L+MlogM/2(N/L)) | O(L+MlogM/2(N/L)) | O(NM\*logM/2(N/L) + NL) | O(NM\*logM/2(N/L) + NL) | O(NM\*logM/2(N/L) + NL) |
| 1 | 11 (Separate Chaining)  let load factor be L | In general  O(L + 1) | - | O(N\*L + N) | - | O(N\*L + N) |
| 2 | 11 | In general  O(L + 1) | In general  O(L + 1) | O(N\*L + N) | O(N\*L + N) | O(N\*L + N) |
| 3 | 11 | In general  O(L + 1) | In general  O(L + 1) | O(N\*L + N) | O(N\*L + N) | O(N\*L + N) |
| 4 | 11 | In general  O(L + 1) | In general  O(L + 1) | O(N\*L + N) | O(N\*L + N) | O(N\*L + N) |
| 1 | 12 (Quadratic Probing) | O(1/1-L) | - | O(N/1-L) | - | O(N/1-L) |
| 2 | 12 | O(1/1-L) | O(1/1-L) | O(N/1-L) | O(N/1-L) | O(N/1-L) |
| 3 | 12 | O(1/1-L) | O(1/1-L) | O(N/1-L) | O(N/1-L) | O(N/1-L) |
| 4 | 12 | O(1/1-L) | O(1/1-L) | O(N/1-L) | O(N/1-L) | O(N/1-L) |
| 1 | 13 (Binary Heap) | O(1) | - | O(N) | - | O(NlogN) in general or else O(N) worst case |
| 2 | 13 | O(1) | O(logN) | O(N) | O(NlogN) | O(NlogN) |
| 3 | 13 | O(1) | O(logN) | O(N) | O(NlogN) | O(NlogN) |
| 4 | 13 | O(1) | O(logN) | O(N) | O(NlogN) | O(NlogN) |
| 1 | 14 (Quadratic Probing Pointer)  let L be load factor | O(1/1-L) | - | O(N/1-L) | - | O(N/1-L) |
| 2 | 14 | O(1/1-L) | O(1/1-L) | O(N/1-L) | O(N/1-L) | O(N/1-L) |
| 3 | 14 | O(1/1-L) | O(1/1-L) | O(N/1-L) | O(N/1-L) | O(N/1-L) |
| 4 | 14 | O(1/1-L) | O(1/1-L) | O(N/1-L) | O(N/1-L) | O(N/1-L) |

**Skip List**

Skip List is the data structure which uses a different algorithm to insert and delete. This algorithm works on the concept of levels and implements probability and hence it always has the time complexity of O (log N) whatsoever for a single insertion and single deletion. So, for all the files, the skip list data structure has a O (N log N) complexity for the whole program and this does not change for any file. Therefore, for the files 1, 2, 3 and 4, all of them run the similar time except file4 which runs slower because it involves random deletion and therefore more time is taken to complete that operation.

**Binary Search Tree**

Binary Search Tree is a data structure which cannot have more than two children. Now, in case of all the files except File4.dat, there is an ordered insertion (in increasing order) and therefore the binary search tree works like a large linked list. However, in File 1,2,3 the value to be inserted is first found where to insert and then it is inserted. The insert () algorithm of BST is recursive and it checks through left and then the right child recursively to find the position. Hence, the time for inserting a single value is O (height of the tree), which is O(N) in this case as all the values are arranged in like a linked list format and height is N for all the files but file4.dat. Similarly, single deletions for file3 is O(N) because it deletes from the reverse order and therefore the code has to delete the values from the bottom most leaf in the tree (as 1.25 million is found at the very end). But for file2.dat, the BST deletes values in the same order of insertion and therefore it only takes O(1) to delete the root and make the child as the new root and hence overall deletion is O(N). Hence, it had a time complexity of O(N2) for the whole program for the files1,2 and 3. Therefore, all these files took a lot of time greater than five minutes to compile and hence was the slowest structure amongst all the other ADTs. However, it ran quickly for file4.dat because it was random insertions and deletions, and therefore insertions and deletions take only O(logN) as the average height of a BST is log N. The program takes only O(NlogN) time, which means it is faster than the time taken for the other three files. Hence, BST is the slowest amongst all the other ADTs except for file4.dat.

**AVL Trees**

AVL trees are special self-balancing Binary Search Trees which always ensure that the height of left and right subtree don’t differ by a maximum of 1. Hence, whenever a new element is inserted or when an existing element is deleted, the tree always self-balances itself to form a balanced tree and therefore it requires only O(log N) time to insert and delete a single value as the balancing nature of the tree makes it easier to insert and delete the values as it is just traversal through the height of a given tree which is log N in average. Hence, for all the files, regardless of the order of insertions and deletions, the total time for the whole program to compile is O(N log N) for an AVL tree.

**Splay Tree**

Splay tree works with the concept of always splaying an item to the top (the root) whenever an item is inserted or to be deleted. For files 1,2,3 since it is ordered insertions, the tree just splays the new value which is larger than the current root is easily splayed through a single rotation and therefore insertions of a single element is O(1) for these three files. For deletion in files 2 and 3, the values to be deleted in file 2 are in same order of insertions and hence the value to be deleted at first is 1 and it is right at the bottom of the tree. Therefore, the time taken to splay 1 alone is O(N) as it has move through the height of the tree which behaves more like a linked list in this case. However, after 1 is splayed as the root and it is deleted, the tree has the minimum of the right sub tree (which is 2) as the new root as it is the next value. From this point, the values to be deleted subsequently are the root in the next steps and now it takes only O (1) for all the other deletions. Meanwhile, for file3.dat, the values to be deleted are in reverse order of deletions and therefore the value to be deleted are from the root and therefore done very quickly O(1) time. For files 1 and 3, the whole program takes O(N) time to compile while File2.dat has a total time of O(NlogN) as deletions and insertions don’t have a constant variation and therefore the average of it yields O(NlogN) and it runs longer than the first two files. For file4.dat, the insertions and deletions are random and hence they have a time complexity of O(log N) for each value as the value has to be traversed through an average tree height of log N and therefore it takes O(NlogN) time for the whole program for this file.

**B Trees**

B Trees are a complex data structure and these structures were tested with different values of M and L and it yielded different results for different values of M. Clearly, for all the files, if the value of M is larger, then the time taken to insert as well as delete is more because the work done to check the value to insert it or delete it has to involve searching through each child of an internal node and hence the time taken depends linearly on the value of M. Also, it can be calculated that the time for each deletion and insertion is O(M\*logM/2N/L + L) as it has to traverse to each leaf of each child and then has to look into the items of each leaf. Therefore, B Trees take shorter time to compile if M is small. In particular, the deletions for M=1000 and L=2 took a longer time as there can be transfer of items and splitting of nodes which all depend at the cost of N.

**Separate Chaining Hash Tables**

For each file, it is observed that the separate chaining always depends on the load factor of the hash table and if the number of collisions become more frequent (the load factor is low), then the time taken is substantially more than excepted as there are more collisions which make it slow to compile. For files 1,2,3 and 4, clearly the insertion after finding the index is just in front of the list and hence it is just O (1). Similarly, deletions also work in a similar way of finding the value first and then removing it. Hence, the time complexity of each insertion and deletion is O(L + 1). Therefore, it largely depends on L (load factor ) and hence it takes more time for more collisions.

**Binary Heap**

This ADT is another special ADT where always the minimum is at the root and the elements are filled to the leftmost available space and if it is smaller than a parent element, it is moved up. Therefore, for all the files, the numbers are inserted at O (1) time as it just fills in the available left most space. However, deletion is always involving the deletion of the minimum and therefore there is a rotation once the root value is removed. Clearly, this is same for all the files as it just involves the rotation of the new smallest value to the top as the root and hence involves O (height = log N) time per deletion. Clearly file1 is the fastest because the numbers are inserted in sequential order which makes it the fastest amongst the four files, while the deletions in the other files make it run slowly compared to the first file.

**Quadratic Probing and Quadratic Pointer Probing Hash Tables**

These two data structures have the same big oh values but the significant difference in compiling for the same value of load factor is found where quadratic probing is faster than quadratic pointer probing. This is because there is no memory allocation and deallocation in quadratic probing which does not result into extra time taken. Therefore, quadratic pointer probing is slower than quadratic probing. Each file takes almost the same time for a given load factor for each separate ADT. Now, quadratic probing is better than separate chaining because it reduces the amount of collisions and rehashing always decreases the probability of collision.

**Conclusion:**

Overall, amongst the skip lists and trees, we can find that AVL and splay trees are faster than simple skip lists because trees have an average height of log N and the insertions and deletions for a tree can be O(1) (in case of splay for files 1,2,3; insertion is O(1)) and hence trees work faster than skip list.

Amongst trees, splay trees are the fastest and then it is AVL and then B Trees and finally BSTs. This is because splay trees just involving splaying the value to be deleted or just inserted to the top and then performing the operation while AVL is rotation which keeps always the height to be log N. Meanwhile,

B Trees and BSTs are slow because they involve finding the position to insert or to find the to-be-deleted item and then move the items over in case of B Trees which takes more time. Also, B Trees with a larger M value, take more time than those with small M value. BSTs include a lot of recursion to find the proper position to insert the value and they also have to adjust the spacing of the tree after deletion. Hence BST are the slowest.

Quadratic Probing is faster than Quadratic Pointer probing as the latter involves a lot of dereferencing pointers and also memory allocation which can increase the time taken to compile. On the other hand, separate chaining hash tables are slower than Quadratic Probing because separate chaining involves more collisions than quadratic probing which can increase the time taken for insertions and deletions. Meanwhile, they are faster than the Quadratic Pointer Probing tables as the latter involve more of memory allocation and dereferencing while the former does not.