Timetest

| | ADT | Op | File1.dat | File2.dat | File3.dat | File4.dat |
|---|---|---|---|---|---|---|
| 6 | Skip List | O(Single Insert) | logN | logN | logN | logN |
| | | O(Single Delete) | x | logN | logN | logN |
| | | O(Insert Series) | NlogN | NlogN | NlogN | NlogN |
| | | O(Delete Series) | x | NlogN | NlogN | NlogN |
| | | O(File) | NlogN | NlogN | NlogN | NlogN |
| 7 | Binary Search Tree | O(Single Insert) | N | N | N | logN |
| | | O(Single Delete) | x | 1 | N | logN |
| | | O(Insert Series) | N^2 | N^2 | N^2 | NlogN |
| | | O(Delete Series) | x | N | N^2 | NlogN |
| | | O(File) | N^2 | N^2 | N^2 | NlogN |
| 8 | AVL Tree | O(Single Insert) | logN | logN | logN | logN |
| | | O(Single Delete) | x | logN | logN | logN |
| | | O(Insert Series) | NlogN | NlogN | NlogN | NlogN |
| | | O(Delete Series) | x | NlogN | NlogN | NlogN |
| | | O(File) | NlogN | NlogN | NlogN | NlogN |
| 9 | Splay Tree | O(Single Insert) | 1 | 1 | 1 | logN (amortized) |
| | | O(Single Delete) | x | logN (amortized) | 1 | logN (amortized) |
| | | O(Insert Series) | N | N | N | NlogN (amortized) |
| | | O(Delete Series) | x | NlogN (amortized) | N | NlogN (amortized) |
| | | O(File) | N | NlogN (amortized) | N | NlogN (amortized) |
| 10 | BTree | O(Single Insert) | log(M/2)(2N/L) | log(M/2)(2N/L) | log(M/2)(2N/L) | log(M/2)(2N/L) |
| | | O(Single Delete) | x | log(M/2)(2N/L) | log(M/2)(2N/L) | log(M/2)(2N/L) |
| | | O(Insert Series) | Nlog(M/2)(2N/L) | Nlog(M/2)(2N/L) | Nlog(M/2)(2N/L) | Nlog(M/2)(2N/L) |
| | | O(Delete Series) | x | Nlog(M/2)(2N/L) | Nlog(M/2)(2N/L) | Nlog(M/2)(2N/L) |
| | | O(File) | Nlog(M/2)(2N/L) | Nlog(M/2)(2N/L) | Nlog(M/2)(2N/L) | Nlog(M/2)(2N/L) |
| 11 | Separate Chaining Hash | O(Single Insert) | 1 | 1 | 1 | 1 |
| | | O(Single Delete) | x | 1 | 1 | 1 |
| | | O(Insert Series) | N | N | N | N |

| | | | | | | |
|---|---|---|---|---|---|---|
| | | O(Delete Series) | x | N | N | N |
| | | O(File) | N | N | N | N |
| 12 | Quadratic Probing Hash | O(Single Insert) | 1 | 1 | 1 | 1 |
| | | O(Single Delete) | x | 1 | 1 | 1 |
| | | O(Insert Series) | N | N | N | N |
| | | O(Delete Series) | x | N | N | N |
| | | O(File) | N | N | N | N |
| 13 | Binary Heap | O(Single Insert) | 1 | 1 | 1 | logN |
| | | O(Single Delete) | x | logN | logN | logN |
| | | O(Insert Series) | N | N | N | NlogN |
| | | O(Delete Series) | x | NlogN | NlogN | NlogN |
| | | O(File) | N | NlogN | NlogN | NlogN |
| 14 | Quadratic Probing Pointer Hash | O(Single Insert) | 1 | 1 | 1 | 1 |
| | | O(Single Delete) | x | 1 | 1 | 1 |
| | | O(Insert Series) | N | N | N | N |
| | | O(Delete Series) | x | N | N | N |
| | | O(File) | N | N | N | N |

Runtimes

| | ADT | | File1.dat | File2.dat | File3.dat | File4.dat |
|---|---|---|---|---|---|---|
| 6 | Skip List | | 1.05468 | 0.711153 | 0.911297 | 2.76489 |
| 7 | Binary Search Tree | | > 5 mins | > 5 mins | > 5 mins | 1.67931 |
| 8 | AVL Tree | | 1.53181 | 1.07072 | 1.13605 | 2.54195 |
| 9 | Splay Tree | | 0.409516 | 0.31693 | 0.250898 | 2.28943 |
| 10 | Btree | M = 3 , L = 1 | 1.04512 | 1.65522 | 0.923685 | 3.00635 |
| | | M = 3 , L = 200 | 0.70588 | 1.79186 | 1.42653 | 2.25878 |
| | | M = 1000 , L = 2 | 4.80462 | 7.14405 | 4.74048 | 0.25216 |
| | | M = 1000 , L = 200 | 2.54201 | 3.57591 | 3.12975 | 3.4342 |
| 11 | Separate Chaining Hash | load factor | | | | |
| | | 0.5 | 0.436023 | 0.339591 | 0.468029 | 1.14118 |
| | | 1 | 0.403328 | 0.312295 | 0.549896 | 0.98031 |
| | | 10 | 0.443589 | 0.390717 | 0.658889 | 1.30257 |
| | | 100 | 4.36396 | 5.15203 | 5.10558 | 6.34577 |
| | | 1000 | 45.3495 | 40.1335 | 38.612 | 56.6433 |
| 12 | Quadratic Probing Hash | load factor | | | | |
| | | 2 | 0.375712 | 0.285173 | 0.283085 | 0.381214 |
| | | 1 | 0.313767 | 0.262596 | 0.250807 | 0.337287 |
| | | 0.5 | 0.225148 | 0.20846 | 0.20451 | 0.281842 |
| | | 0.25 | 0.224669 | 0.20987 | 0.205086 | 0.279495 |

| | | | | | | |
|---|---|---|---|---|---|---|
| | | 0.1 | 0.227322 | 0.212353 | 0.206441 | 0.283149 |
| 13 | Binary Heap | | 0.19969 | 0.537579 | 0.537282 | 0.655376 |
| 14 | Quadratic Probing Pointer Hash | load factor | | | | |
| | | 2 | 0.468744 | 0.344229 | 0.457223 | 0.734606 |
| | | 1 | 0.421448 | 0.322549 | 0.460769 | 0.578461 |
| | | 0.5 | 0.33178 | 0.271411 | 0.396501 | 0.621331 |
| | | 0.25 | 0.356749 | 0.279683 | 0.365561 | 0.467001 |
| | | 0.1 | 0.429107 | 0.317566 | 0.408352 | 0.516418 |

## Skip List

Skip Lists are linked lists that incorporate binary search in all its operations. Hence, the order of operations does not matter and the total time complexity for all four files is O(NlogN). The results show that File1.dat takes longer than File2.dat and File3.dat, but less than File4.dat, thus revealing that the deletions in File2.dat and File3.dat are unusually fast.

For the non-random files, to find the position of the maximum being inserted, the skip list moves right until it finds a NULL pointer, and then moves down. Thus, File2.dat and File3.dat never back-up during their insertion. File2.dat is faster than File3.dat because the path across the list is the minimum for each item deleted.

For File4.dat, the deletion and insertion paths must back-up at each level since they are randomized. This repeated overshooting by File4.dat explains why its operations takes longer than those of File2.dat and File3.dat.

## Binary Search Tree

The Binary Search tree had the longest run time compared to all other ADTs for File1.dat, File2.dat and File3.dat. This is because the order of insertion caused the formation of a right-skewed tree, essentially forming an ascending order linked list. This is the worst-case scenario for binary search trees since the tree must be traversed all the way to then end for each insertion, since there is no tail pointer provided. Thus, insertions for File1.dat, File2.dat and File3.dat are O(N^2) since a right-skewed tree of height N elements is produced by insertion.

Theoretically, File2.dat should perform faster than File3.dat. In File2.dat, deletions are performed in ascending order, while deletions are performed in descending order for File3.dat. Deletions involve removing the root and promoting the minimum value of the right subtree. Hence File2.dat should perform faster than File3.dat since deletions can be performed in order with O(1) time complexity for each deletion, while the tree must be traversed all the way to the end for each deletion File3.dat. However, the actual results were unclear since both files took longer than 5 minutes to execute and had to be terminated prematurely.

Since File4.dat has randomized operations, the binary search tree is more likely to be balanced. In this case, the cost of all operations is O(NlogN), therefore performing much faster than the first three files.

## AVL Tree

An AVL tree is identical to a binary search tree, except that for every node in the tree, the height of the left and right subtrees can differ by at most 1. The AVL tree is self-balancing, thus insertions and deletions warrant updating the balance of the tree such that the balance condition is always met.

Since the AVL tree is rebalanced after each operation, the order of insertions and deletions do not matter. Since the tree is balanced, insertions and deletions in all four files have O(NlogN) time complexity.

Despite having the same time complexity across all four files, it is noteworthy that File4.dat took significantly longer than File2.dat and File3.dat. This is likely attributed to the ascending order insertions in the first three files; since insertions in File4.dat are randomized, rebalancing involves backing up to check larger values.

## Splay Tree

The splay tree  guarantees at most O(MlogN) time for M consecutive tree operations, although this guarantee does not preclude the possibility that any single operation might take Θ(N) time. Thus, a splay tree has an O(logN) amortized cost per operation.

Splay trees are characterized by the condition that each time a node is accessed, the node is pushed to the root by a series of AVL tree rotations. Thus, if a particularly deep node is accessed, the tree is restructured in the process of splaying the node and accessing the remaining nodes on that path is made cheaper.

Insertions in File1.dat, File2.dat and File3.dat each have O(1) time complexity, since a single insertion in ascending order only required insertion into the right subtree followed by a single rotation. Since insertions are in ascending order for File1.dat, File2.dat and File3.dat, after the final insertion, the resulting tree contains integers in descending order with the largest value at the root and the first value in a leaf. This explains why File2.dat took longer than File3.dat; for File3.dat, no splaying was needed since the maximum value was already at the root. On the other hand, File2.dat performs ascending deletions, which require that the minimum value be splayed up to the root for each deletion. However, in this process, the tree becomes more balanced over time and the resulting time complexity is amortized O(NlogN). Using similar logic, the cost of insertions and deletions in File4.dat is also amortized O(NlogN), since splaying values causes the tree to become more balanced over time.

## BTree

A B-tree of order M is an M-ary tree such that increased branching allows for decreased depth, to reduce the number of required disk accesses.

All operations are of O(Nlog(M/2)(2N/L)) (the maximum height of a Btree with N elements) complexity since the tree must be traversed all the way to the leaves for each operation. It is noteworthy that for small values of M, the size of L does not significantly affect performance. Whereas for large values of M, large values of L allow faster performance. This is attributed to the fact that checking leaf nodes are less expensive than checking internal nodes. Hence, the larger the number of leaf nodes on a  given level, i.e., the larger the branching, the fewer the number of internal nodes that must be checked. When the number of leaves is the same, larger values of M provide for better performance, again because this increases the number of nodes on the same level.

Comparing File2.dat, File3.dat and File4.dat, File3.dat has the fastest run time across all M and L values. This is because deleting in descending order ensures that values are removed from the bottom, such that the remaining values do not need to be shifted. Insertions across all files have the same time complexity as insertion is always at the end of a leaf node.

Comparison between Trees and Skip List

Evidently, the Binary Search Tree performed worst across File1.dat, File2.dat, and File3.dat, since the order of insertions resulted in the wort case scenario – a skewed tree that resembles a linked list. However, the Binary Search Tree outperforms all other trees and Skip List since insertions and deletions are randomized, so the tree is more likely to be balanced. Moreover, the Binary Search Tree does not have to worry about additional conditions such as performing rotations, splaying values, or keeping track of height.

The Splay Tree performed best across File1.dat, File2.dat, and File3.dat. Its O(1) insertion cost for ascending insertions and amortized O(logN) deletion cost allows very fast performance over time.

The AVL Tree has the advantage that all operations require balancing. Thus, regardless of order, the time complexity an operation will always be O(logN), thus allowing for greater consistency.

Comparison between Hash tables

For the Separate Chaining Hash, Quadratic Probing Hash, and Quadratic Probing Pointer Hash, insertions and deletions have O(1) time complexities since they can easily be accessed using a hash function, such that search operations are also of O(1) time complexity.

However, the Separate Chaining hash is slower than the Quadratic Probing Hash, and Quadratic Probing Pointer Hash, since Separate Chaining involves keeping a list of all elements that hash to the same value. As the number of elements in the hash table increases, the number of collisions also increase and the Separate Chaining lists increase in length, which increases the time required for searching and rehashing the elements in the lists.

Considering the effects of the load factor, it should be noted that the load factor is inversely proportional to the table size. For separate chaining, a smaller load factor produced better performance. As the load factor increases for the Separate Chaining hash, the table size decreases, and thus collisions are more frequent. Thus, the size of its lists are increased and runtime also increases as it takes longer to traverse the lists in order to find and rehash elements. For the Quadratic Probing Hash and Quadratic Probing Pointer Hash, as the load factor decreases, the table size increases and collisions are less frequent since the probability of finding an empty space in the hash increases.

Additionally, it is noteworthy that quadratic probing outperformed Separate Chaining on all four files even though Separate Chaining merely involves an insert into the front of the list determined by the hash function. Inspection of the code for Separate Chaining reveals that the insert function checks lists for duplicate values, which likely slows down runtime, especially for larger load factors, where collisions are more frequent, and lists are increased in size.

Comparison between Quadratic Probing Hash and Quadratic Probing Pointer Hash

The Quadratic Probing Hash performed better than the Quadratic Probing Pointer Hash. This is because the Quadratic Probing Pointer Hash must create new objects during inserts and must deal with pointer accesses search operations, which takes additional time in comparison to the Quadratic Probing Hash, which can simply copy integers into existing arrays.

Binary Heap

The Binary Heap is characterized by the heap-order property, such that, for every node X, the key in the parent of X is smaller than (or equal to) the key in X, except for the root. Thus, insertions in File1.dat, File2.dat, and File3.dat have O(1) time complexities, since the elements are simply inserted into the heap in ascending order such that the succeeding value is always less than its parent.  However randomized insertions in File4.dat contributed to a longer runtime since the heap must be traversed such that the heap-order property is met, which contributes to an O(logN) time complexity.

For deletions on all four files, in the worst case, the new root must be swapped with its child on each level until the bottom level of the heap is reached. Hence, a delete operation has a time complexity relative to the height of the tree, or O(log n).