|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| ADT | Load factor  / L&M | File1.dat | File2.dat | File3.dat | File4.dat |
| Skip List |  | 0.32 | 0.23 | 0.26 | 0.45 |
| BST |  | >5min | 222.247 | >5MIN | 0.27 |
| AVL |  | 0.39 | 0.299 | 0.3156 | 0.47 |
| Splay |  | 0.13 | 0.104 | 0.1002 | 0.32 |
| BTree | L=1,M=3 | 0.28 | 0.405 | 0.246 | 0.52 |
|  | L=200,M=3 | 0.29 | 0.49 | 0.416 | 0.65 |
|  | L=2,M=1000 | 1.44 | 2.07 | 1.3893 | 1.93 |
|  | L=200,M=1000 | 0.63 | 0.695 | 0.709 | 0.779 |
| Separate Hash | 0.5 | 0.152 | 0.114 | 0.11 | 0.17 |
|  | 1 | 0.146 | 0.118 | 0.107 | 0.21 |
|  | 10 | 0.149 | 0.136 | 0.102 | 0.25 |
|  | 100 | 0.243 | 0.654 | 0.16 | 1.01 |
|  | 1000 | 2.39 | 1.958 | 1.05 | 1.58 |
| Quad Hash | 0.1 | 0.09 | 0.084 | 0.084 | 0.089 |
|  | 0.25 | 0.0886 | 0.083471 | 0.083 | 0.083 |
|  | 0.5 | 0.0882 | 0.083321 | 0.086 | 0.087 |
|  | 1 | 0.12 | 0.10042 | 0.102 | 0.11 |
|  | 2 | 0.138 | 0.1085 | 0.109 | 0.109 |
| QuadPtr hash | 0.1 | 0.155 | 0.1193 | 0.123 | 0.13 |
|  | 0.25 | 0.134 | 0.107236 | 0.11 | 0.12 |
|  | 0.5 | 0.122 | 0.10408 | 0.104 | 0.11 |
|  | 1 | 0.152 | 0.11865 | 0.11 | 0.12 |
|  | 2 | 0.172 | 0.125578 | 0.12 | 0.13 |
| Binary heap |  | 0.07 | 0.172 | 0.174 | 0.182 |

File1:

Big O in File1:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | skiplist | BST | AVL | Splay tree | BTree | Separate chaining | Quad  Hash/  Quadptr | heap |
| single | logN | N | logN | 1 | logm N | 1+λ | 1 | 1 |
| File | NlogN | N^2 | NlogN | N | NlogmN | N | N | N |

The cost of Insertion by Skip list for single element is O(logN), hence the total costs of Insertion is O(NlogN) for all files. For File1.dat, since we insert the number from 1—250,000, The order is from smaller to bigger, hence during the insertion, they will never go back to left pointer. Though skiplist and AVL are both O(logN) for single insertion, AVL need to count the height for each node and do rotation to re-balance tree. Hence, AVL is slower than skiplist.

BST is the slowest. Since insertion is from 1-500000, BST will become a linked list. Every time inserting a new number, BST needs to traverse a whole linked list to insert a new number which is O(N). For whole file, it needs O(n^2).

AVL is the third slowest. Splay tree is faster than AVL. Due to two reasons, first, AVL needs to count height for every node. This maintenance costs extra time, comparing to splay tree which is not necessary to count height for each node. Second, since insertion is from 1-500000. Every time splay tree insert a new number, the new number will be the only one in right children. It only needs to do a single rotation to bring the new number to the root. This action only takes constant time, hence O(1) for single insertion and O(N) for the whole file. Actually, even without the second reason, the AVL will slower than splay tree. We can see this in file4. Even the insertion is random number, splay tree is still faster than AVL, mainly due to the first reason which is AVL needs to count height for each node.

BTree is much faster than AVL and BST, especially when M=3 and L=1. At first, I think the BTree with M=1000 and L=200 will be the fastest, mainly because the insertion code at internalnode and leavenode starts by checking the tail of the array and since in file1, insertion number become larger and larger, inserting will be done only O(1) which simply insert at the tail of the array. In this setting, the less the height the BTree is , the less time the insertion will cost, because traversing between different level will cost more time. However, it turns out that BTree is fastest when M=3 and L=1. After checking the code, I found that when leaf size is not equal to 1, insertion will cause BTree to split. After split, the left leaf will be half empty. When trying to insert a new number, the BTree will insert number to left( addtoleft() ) and call the function resetMinimum to update the keys. These procedures are quite time consuming. If insertion are 500000 and L=200, it will cause 2499 times splits and call the function resetMinimum 249900 times. However, if L=1, the leaf will never split. Hence no need to call function addtoleft and resetMinimum. In another words, the less the splits, the less running time. We can get the following conclusions:

1. If M is the same, the bigger the leafsize(except L=1), the less running time.
2. If L is the same, the bigger the M, the less running time.(As I mentioned above, if M is bigger, the less height the Btree will have.)
3. If L=1, split will never happen. Hence, when L=1, the BTree will be faster.

The reason no matter what M and L are, the BTree is still faster than AVL and BST is that the insertion always at the tail of the array.

Separate chaining hash is O(1+ **λ),** this Big O could explain the data that hash table become slower from load factor 1 to 1000. The bigger the load factor, more element need to traverse through the list( needs to check duplication). At first, I thought there should be no difference between **λ**=1 and **λ**=0.5. In both situations, there will be only one element in the list. However, it turns out that λ=0.5 is faster. I think it is mainly because when λ=1, initialization will cost more time to create a larger size hash table.

Quad Hash is even much faster than separate chaining. It is because Quad hash does not need to traverse any list(even the list only contain 1 element). This saves time. The data for different load factors are as expect. Since the hash table will rehash when the element in the hash table exceed half the table size, load factor=0.5 will never rehash; load factor=1, rehash 2 times; load factor=2, rehash 4 times. The rehash explains why the running time become bigger when load factors become bigger from 1 to 2. The reason for load factor=0.5 is faster than load factor=0.1 and 0.25 is the same for separate chaining. Initialization will cost more time to create a larger size hash table.

QuadPtr is much slower than Quad. It proves that creating new objects takes more time than copying integer into the array.

Heap is the fastest ADT in File1. It is because the new number is inserted at the tail of array and it will not change the position when it compare to its parent. Hence we have O(1) for single insertion and O(N) for whole File.

File2:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Skip list | BST | AVL | Splay | BTree | Separate  chaining | Quad | heap |
| Single  Delete | logN | 1 | logN | 1 | L+Mlogm­(N/L) | 1 | 1 | logN |
| File  delet | NlogN | N | NlogN | N | N(L+Mlogm­(N/L)) | N | N | NlogN |

I did not list insertion, since it is the same with file1.

Deletion in File2:

Skip list deletion in File2 is O(logN). File2 is deleting from 1 to 125,000. It does not need to traverse whole list to delete the max number. However, file3 does need. Hence, File2 is best situation for skip list. Like insertion, skip list deletion is still faster than AVL for the same reason which is AVL need to do counting and rebalance the tree.

BST deletion in File2 is O(1), since BST here is a link list and begin with1(root is 1). Deletion is from 1 to 250000, so each deletion is simply delete the root.

AVL is O(logN), the performance is as expect. It is also slow because it after every deletions, it has to balance the tree. Besides, since deletion is from 1-250000, every deletion have to traverse the longest path in the tree which is logN-1.

BTree deletion is slow in file2. It is because deletion is from 1 to 250000. It has to traverse each element in the array in both leafnode and internalnode. Since BTree search the array from the end, it has to traverse the whole list to find the minimum. Another reason the BTree is slow is that when the count is smaller than the half size of L/M, it has to borrow from the right sibling. L=2 is the worst case, every deletion will cause 500 times borrow. Besides, if it is InternalNode, it also need to update the keys.

Splay Tree is the fastest. Although at first several deletions have to use O(n) rotation to bring 1 to the root. However, after 20 deletions(log250,000), the distance between every node in the tree and root is decrease to (½)^20  of the initial distance. And after that, almost every deletion only cost O(1). Hence, the amortized big O for single deletion is O(1) and for whole file is O(N).

The separate chaining is as expect. Since every insertion is at the head of the list. The smallest element must in the tail of every list. Hence, deletion from1 to 2500000 will cause every deletion traverse the whole list which is O(1+ **λ)**.

Quad hash is the third fastest as expect. The single deletion is O(1). For whole file, it is O(N). The different running time for different load factors is due to insertions.

Heap is the second fastest. It is due to the path for deletion is logN long and only two times comparison in every level. Hence, the single deletion for heap is O(logN) and O(NlogN) for whole file.

File3:

Insertion is the same with File1.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Skip list | BST | AVL | Splay | Btree | separate  chaining | Quad  hash | heap |
| single deletion | logN | N | logN | 1 | logm(N/L) | 1 | 1 | logN |
| File deletion | NlogN | N^2 | NlogN | N | N\*logm(N/L) | N | N | NlogN |

Deletion:

BST is the slowest. After 25000 insertions, BST is a linked list from 1 to 25000(1 is root). In file3, deletion is from 25000 to 1, so every deletion needs to traverse whole list which is O(N). Hence, for whole file deletion, it takes O(N^2).

AVL is still slower than splay tree. Since AVL is balanced and sorted, every deletion need to traverse Θ(logN-1) long path. After a deletion, AVL has to count the height and re-balance. This procedure is the reason it is slower than splay.

BTree performance is quite good in File3. Since File3 deletion is from 25000 to1. Since search starts from the end of the array, Btree only needs to compare 1 time in each intertnalNode and get down to to the next level and finally delete the last element in leafNode. In the case M=1000 and L=200, after only two comparisons, deletion can be done. BTree with M=1000 and L=200 should be the fastest. However, the data shows M=3, L=1 is the fastest. The reason is simply the same in File1. Some deletion may cause borrow, and calling borrow is quite time consuming (it will call restminimum again). Again, in the case M=3 and L=1, there is no need to borrow. Hence, it is the fastest one among BTrees.

Splay tree is fastest without any doubt. After insertion, the splay tree is a sorted linked list (25000 is the root). Every deletion is simply delete the root and makes its child become root. Hence single deletion is O(1) and for whole file is O(N).

Quad hash and separate chaining is the same for file2 and file3.

Heap performance is the same as it in File2 because heap do not influenced by the order of number.

File4:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Skip | BST | Splay | Btree | Sep chain | quad | AVL | heap |
| insertion | logN | logN | logN | L+Mlogm(N/L) | 1 | 1 | logN | 2 |
| deletion | logN | logN | logN | L+Mlogm(N/L) | 1 | 1 | logN | logN |
| File | NlogN | NlogN | NlogN | N(L+Mlogm(N/L)) | N | N | NlogN | NlogN |

Insertion:

Skip list is third slowest ADT in File 4. Since the number are random, the insertion will sometimes go back to the left pointer because we cannot promise the number to insert or number to delete is always bigger than the number already in the list.

Since the insertion number is random in file4. BST will have excellent performance and it is indeed the fast ADT among all the tress. BST will grow in a quite balanced way. Hence, all single insertion and deletion will be O(logN).

AVL is slow mainly because it needs to count the height for every node and do rotations while the tree is not balance. Since those procedure is quite same in File2 and File3. The runtime is almost the same.

Splay tree in File4 is not as fast as it in File 2 and File3. It is because the insertion will need O(logN) to move the element to the root.

The running time for BTree in File4 should be longer than it in File3 and shorter than in File2. File 3 is the best situation for BTree. Since the number is random, a number could be searched at the mid of the leafnode or intertnalnode which is M/2 and L/2. Since every time split, internalNode need to update two array, hence we want smaller M size if L is the same. If M is the same, we want L is bigger, hence we will have less split and save the time(do not need to call the function resetMinmum).

The quad and separate chaining do not effected by the “random number”. Hence, their performance is the same as it in the File2 and Fil3.

Deletion:

Since BST is quite balance in File4, the single deletion is O(logN). AVL and Splay will have O(logN) as their insertion’s big O. Heap and probing will not influenced by the “random number”. Hence, the deletion is the same as they are in File 2 and 3.