|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ADT Menu | File1 | File2 | File3 | File4 |
| 1.Skip List | 0.2 | 0.143 | 0.166 | 0.209 |
| 2.Binary Search Tree | 507.059 | 73.315 | 185.722 | 0.133 |
| 3.AVL Tree | 0.222 | 0.182 | 0.191 | 0.24 |
| 4.Splay Tree | 0.08 | 0.058 | 0.054 | 0.168 |
| 5.BTree | | | | |
| M=3,L=1 | 0.174 | 0.24 | 0.145 | 0.279 |
| M=3,L=200 | 0.163 | 0.29 | 0.245 | 0.321 |
| M=1000,L=2 | 0.85 | 1.225 | 0.811 | 1.122 |
| M=1000,L=200 | 0.366 | 0.407 | 0.413 | 0.457 |
| 6.Separate Chaining Hash | | | | |
| 0.5 | 0.09 | 0.067 | 0.069 | 0.099 |
| 1 | 0.083 | 0.07 | 0.062 | 0.107 |
| 10 | 0.083 | 0.076 | 0.063 | 0.104 |
| 100 | 0.154 | 0.201 | 0.108 | 0.257 |
| 1000 | 1.457 | 1.529 | 0.828 | 1.71 |
| 7.Quadratic Probing Hash | | | | |
| 2 | 0.079 | 0.069 | 0.067 | 0.069 |
| 1 | 0.073 | 0.061 | 0.057 | 0.06 |
| 0.5 | 0.051 | 0.049 | 0.048 | 0.055 |
| 0.25 | 0.048 | 0.051 | 0.049 | 0.049 |
| 0.1 | 0.05 | 0.047 | 0.048 | 0.051 |
| 8.Binary Heap | 0.047 | 0.104 | 0.102 | 0.111 |
| 9.Quadratic Probing Pointer Hash | | | | |
| 2 | 0.105 | 0.08 | 0.076 | 0.1 |
| 1 | 0.089 | 0.075 | 0.077 | 0.092 |
| 0.5 | 0.072 | 0.07 | 0.068 | 0.078 |
| 0.25 | 0.077 | 0.07 | 0.069 | 0.081 |
| 0.1 | 0.088 | 0.079 | 0.078 | 0.089 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| ADT Menu | File | Individual insertion | Individual deletion | Entire series of insertions | Entire series of deletions | Entire file |
| 1.Skip List | 1 | logN |  | NlogN | NlogN | NlogN |
|  | 2 | logN | logN | NlogN | NlogN | NlogN |
|  | 3 | logN | logN | NlogN | NlogN | NlogN |
|  | 4 | logN | logN | NlogN | NlogN | NlogN |
| 2.Binary Search Tree | 1 | N |  | N\*N |  | N\*N |
|  | 2 | N | 1 | N\*N | N | N\*N |
|  | 3 | N | N | N\*N | N\*N | N\*N |
|  | 4 | logN | logN | NlogN | NlogN | NlogN |
| 3.AVL Tree | 1 | logN |  | NlogN |  | NlogN |
|  | 2 | logN | logN | NlogN | NlogN | NlogN |
|  | 3 | logN | logN | NlogN | NlogN | NlogN |
|  | 4 | logN | logN | NlogN | NlogN | NlogN |
| 4.Splay Tree | 1 | 1 |  | N |  | N |
| (amortized d) | 2 | 1 | logN | N | NlogN | NlogN |
|  | 3 | 1 | 1 | N | N | N |
| (amortized) | 4 | logN | logN | NlogN | NlogN | NlogN |
| 5.BTree | 1 | logN |  | NlogN |  | NlogN |
|  | 2 | logN | logN | NlogN | NlogN | NlogN |
|  | 3 | logN | logN | NlogN | NlogN | NlogN |
|  | 4 | logN | logN | NlogN | NlogN | NlogN |
| 6.Separate Chaining Hash | 1 | λ |  | Nλ |  | Nλ |
|  | 2 | λ | λ | Nλ | Nλ | Nλ |
|  | 3 | λ | λ | Nλ | Nλ | Nλ |
|  | 4 | λ | λ | Nλ | Nλ | Nλ |
| 7.Quadratic Probing Hash | 1 | λ |  | Nλ |  | Nλ |
|  | 2 | λ | λ | Nλ | Nλ | Nλ |
|  | 3 | λ | λ | Nλ | Nλ | Nλ |
|  | 4 | λ | λ | Nλ | Nλ | Nλ |
| 8.Binary Heap | 1 | 1 |  | N |  | NlogN |
|  | 2 | 1 | logN | N | NlogN | NlogN |
|  | 3 | 1 | logN | N | NlogN | NlogN |
|  | 4 | logN | logN | logN | NlogN | NlogN |
| 9.Quadratic Probing Pointer Hash | 1 | λ |  | Nλ |  | Nλ |
|  | 2 | λ | λ | Nλ | Nλ | Nλ |
|  | 3 | λ | λ | Nλ | Nλ | Nλ |
|  | 4 | λ | λ | Nλ | Nλ | Nλ |

1. Binary search tree

For binary search tree, the tree does not need to be balanced. File1, 2 and 3 insert sorted elements, from the smallest number to the largest. So the number inserted is never smaller than the last one. This results in that the binary tree always creates a new level to store only one element. Then, the binary tree is extended like a single linked list. Thus, the big O of inserting sorted numbers is N. When it comes to deleting, file 2 delete from the smallest to the largest number. After insertion, the smallest number is at the top of the tree. So in the deletion of file2, it is only need to remove the root and use the nearest element as the root. Thus the big O of the deletion of file 2 is 1. On the other hand, file 3 requires deleting from the largest number to the smallest, which is from the deepest level to the highest. Thus the big O of the deletion of file 3 is N. File4 is totally random. Therefore, the insertion and the deletion of file4 are the average that is logN. Then, every big O of inserting and deleting series of elements is N times inserting and deleting individual elements.

1. AVL Tree

An AVL tree is a binary tree with a balance condition. The balance condition must be easy to maintain, and it ensures that the depth of the tree is logN. An AVL tree except that for every node in the tree, the height of the left and right subtrees can differ by at most 1. Thus, all the tree operations can be performed in logN time. Thus, no matter the order of insertion and deletion, the big O is always log N. Therefore, the big O four files to insert and delete individual element is logN, and for series of elements and the whole files is NlogN.

1. Splay Tree

Splaying the tree for a certain element rearranges the tree so that the element is placed at the root of the tree. In file1, 2 and 3, the insertion of elements is in order, from the smallest number to the largest. The number inserted is always larger than the last inserted number and it is at the root of the tree. So the tree is like a line and the largest number is at the top and elements become smaller when they go deeper. Thus, the insertion of these file is O(1). After insertion, it comes to deletion. File 3 deletes from the largest number to the smallest and because the largest number is always at the top of the tree. So it is only need to delete and change the root without rotation. The big O the deletion of file3 is 1. On the other hand, in File2, every time deleting an element makes the tree nearly a balanced binary tree. Therefore the deletion of file2 is logN. File 4 is random. Thus, the big o of the insertion and the deletion of file4 are logN.

1. Binary Heap

Binary heap is an array. Because the insertion of file1, 2 and 3 is in order, it is only need to put the element to the end of the array. So the big O of the insertion of these files is 1. To delete the number in Binary Heap need to find the number in the tree first. Because elements are in different places of the tree, the big O of the deletion of these files is the average logN. File 4 is random, so the big O of both insertion and deletion is logN.

1. Hash table

The hash tables have faster operations, O(λ), than any of the other data structures we’ve studied. Λ is # elements/tablesize. So the big O of all the hash table is O(λ) to insert and delete an single element and O(Nλ) to insert and delete series of elements and the whole file.

1. Separate Chaining Hash

When the load factor is 1, the table size equals to the number of elements. Since elements are different, each space of the array contains only one linked list node. Thus, the big O of it is 1. When the factor becomes larger, the table size becomes smaller, which means, each space of the array contains more linked list nodes. Thus, when the load factor is larger, it needs more time to run.

1. Quadratic Probing Hash & Quadratic Probing Pointer Hash

Quadratic Probing Pointer Hash takes longer time of Quadratic Probing Hash. Using a Quadratic Probing Pointer Hash waste too much space. Usually, there is more than four times the space needed since any time there is two times the space needed the program double the table size. So if use an array of pointer can save the space. To sum up the Quadratic Probing Pointer Hash needs less space than Quadratic Probing Hash. Because when using the Quadratic Probing Pointer Hash, need to access the element through the pointer first, Quadratic Probing Pointer Hash needs more time to run than Quadratic Probing Hash. Quadratic Probing Pointer Hash needs time to dereference the pointer.

1. BTree

Suppose that we have more data than can fit in main memory, and, as a result, must have the data structure reside on disk. Then we use BTree. If we have more branching, we have less height. An M-ary tree has height that is roughly log(M,N). The M-ary search tree is balanced in some way, it could degenerate into a linked list. When L are the same, the best case height of a B-tree is log(m,n+1), and the worst height case is log(m/2,(n+1)/2+1), so M changes with N. When M does not change, the larger L needs less split and merge. So when m=1000, L=200 runs faster than L=2.

1. Skip list

The big O of the skip list is always logN, but skip list do not care about the order. So there is no big difference between the running time of file2 and 3. Comparing to the AVL tree, avl tree also does not care about the order of elements, but avl tree is slower than skip list, because avl tree needs to rotate and change pointers and roots. Binary search tree cares about the order. This results in the extreme cases take place, such as file 1, 2 and 3, since binary search tree can be unbalanced. Splay tree has its own rule to keep self-balanced, so in sorted order the big O of splay tree can be 1. By using splay tree, file 2 runs faster than using skip list. Btree is slower than skip list in most case, because it stores things on the disk. It is faster to access memory than to access disk.