ECS60

Xie Zhou

Id: 912143385

Timetest3 Write-up

Average Run time for each ADT for each file

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | File1  1-500000 | File2  1-250000 1-250000 | File3  1-250000 250000-1 | File4  random |
| Skip list | 0.164993 | 0.122467 | 0.144549 | 0.372126 |
| Binary search tree | >5 min | >5min (321.41s) | >5 min | 0.225844 |
| AVL tree | 0.320399 | 0.247626 | 0.237967 | 0.361071 |
| Splay tree | 0.067159 | 0.071733 | 0.087287 | 0.318899 |
| B tree (M 3 L 1) | 0.185283 | 0.383481 | 0.223774 | 0.737866 |
| B tree (M 3 L 200) | 0.186231 | 0.380587 | 0.228964 | 0.425359 |
| B tree (M 1000 L 2) | 0.952558 | 1.38893 | 0.869804 | 1.3195 |
| Btree (M1000 L 200) | 0.509891 | 0.667517 | 0.596792 | 0.655787 |
| Separate Chaining Hash (LoadFactor 0.5) | 0.091728 | 0.080198 | 0.092371 | 0.207317 |
| Separate Chaining Hash (LoadFactor 1) | 0.085516 | 0.098795 | 0.123024 | 0.194913 |
| Separate Chaining Hash (LoadFactor 10) | 0.148746 | 0.141624 | 0.130138 | 0.195448 |
| Separate Chaining Hash(LoadFactor 100) | 0.535227 | 0.504393 | 0.214836 | 0.703568 |
| SeparateChainingHash(LoadFactor 1000) | 8.0845 | 6.13509 | 1.94885 | 5.77272 |
| Quadratic probing Hash (LoadFactor 2) | 0.078427 | 0.119233 | 0.110068 | 0.131543 |
| Quadratic probing Hash (LoadFactor 1) | 0.072428 | 0.105859 | 0.11053 | 0.133463 |
| Quadratic probing Hash (LoadFactor 0.5) | 0.043728 | 0.08951 | 0.059324 | 0.068592 |
| Quadratic probing Hash (LoadFactor 0.25) | 0.055846 | 0.088575 | 0.090122 | 0.079597 |
| Quadratic probing Hash (LoadFactor 0.1) | 0.045341 | 0.042458 | 0.046755 | 0.065827 |
| Binary heap | 0.037392 | 0.122043 | 0.104662 | 0.127575 |
| Quadratic probing ptr hash (LoadFactor 2) | 0.173815 | 0.109375 | 0.106924 | 0.173839 |
| Quadratic probing ptr hash (LoadFactor 1) | 0.138291 | 0.09269 | 0.127111 | 0.11068 |
| Quadratic probing ptr hash (LoadFactor 0.5) | 0.064595 | 0.074697 | 0.116133 | 0.088748 |
| Quadratic probing ptr hash (LoadFactor 0.25) | 0.076131 | 0.078364 | 0.057965 | 0.092001 |
| Quadratic probing ptr hash (LoadFactor 0.1) | 0.094279 | 0.088319 | 0.067661 | 0.110346 |

Performance tables for each ADT -- Big(O)s:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Skip list (Big-O) | File1 | File2 | File3 | File4 |
| Individual insertion | log(N) | log(N) | log(N) | log(N) |
| Individual deletion |  | log(N) | log(N) | log(N) |
| Entire series of insertions | Nlog(N) | Nlog(N) | Nlog(N) | Nlog(N) |
| Entire series of deletions |  | Nlog(N) | Nlog(N) | Nlog(N) |
| Entire file | Nlog(N) | Nlog(N) | Nlog(N) | Nlog(N) |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Binary search tree (Big-O) | File1 | File2 | File3 | File4 |
| Individual insertion | N | N | N | log(N) |
| Individual deletion |  | 1 | 1 | log(N) |
| Entire series of insertions | N^2 | N^2 | N^2 | Nlog(N) |
| Entire series of deletions |  | N | N | Nlog(N) |
| Entire file | N^2 | N^2 | N^2 | Nlog(N) |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| AVL tree (Big-O) | File1 | File2 | File3 | File4 |
| Individual insertion | log(N) | log(N) | log(N) | log(N) |
| Individual deletion |  | log(N) | log(N) | log(N) |
| Entire series of insertions | Nlog(N) | Nlog(N) | Nlog(N) | Nlog(N) |
| Entire series of deletions |  | Nlog(N) | Nlog(N) | Nlog(N) |
| Entire file | Nlog(N) | Nlog(N) | Nlog(N) | Nlog(N) |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Splay tree (Big-O) | File1 | File2 | File3 | File4 |
| Individual insertion | 1 | 1 | 1 | log(N) |
| Individual deletion |  | 1st -> N  the rest-> 1 | 1 | log(N) |
| Entire series of insertions | N | N | N | Nlog(N) |
| Entire series of deletions |  | N | N | Nlog(N) |
| Entire file | N | N | N | Nlog(N) |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| B tree (Big-O) | File1 | File2 | File3 | File4 |
| Individual insertion | Mlog(N/L)+L | Mlog(N/L)+L | Mlog(N/L)+L | Mlog(N/L)+L |
| Individual deletion |  | Mlog(N/L)+L | Mlog(N/L)+L | Mlog(N/L)+L |
| Entire series of insertions | MNlog(N/L)+L | MNlog(N/L)+L | MNlog(N/L)+L | MNlog(N/L)+L |
| Entire series of deletions |  | MNlog(N/L)+L | MNlog(N/L)+L | MNlog(N/L)+L |
| Entire file | MNlog(N/L)+L | MNlog(N/L)+L | MNlog(N/L)+L | MNlog(N/L)+L |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Separate Chaining Hash | File1 | File2 | File3 | File4 |
| Individual insertion | λ+1 | λ+1 | λ+1 | λ+1 |
| Individual deletion |  | λ+1 | 1+1 | λ＋1 |
| Entire series of insertions | N(λ+1) | N(λ+1) | N(λ+1) | N(λ+1) |
| Entire series of deletions |  | N(λ+1) | N(1+1) | N(λ+1) |
| Entire file | N(λ+1) | N(λ+1) | N(λ+1) | N(λ+1) |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Quadratic probing hash (Big-O) | File1 | File2 | File3 | File4 |
| Individual insertion | 1 | 1 | 1 | 1 |
| Individual deletion |  | 1 | 1 | 1 |
| Entire series of insertions | N | N | N | N |
| Entire series of deletions |  | N | N | N |
| Entire file | N | N | N | N |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Binary Heap (Big-O) | File1 | File2 | File3 | File4 |
| Individual insertion | 1 | 1 | 1 | log(N) |
| Individual deletion |  | 1 | N | log(N) |
| Entire series of insertions | N | N | N | Nlog(N) |
| Entire series of deletions |  | N | N^2 | Nlog(N) |
| Entire file | N | N | N^2 | Nlog(N) |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Quadratic probing pointer hash | File1 | File2 | File3 | File4 |
| Individual insertion | 1 | 1 | 1 | 1 |
| Individual deletion |  | 1 | 1 | 1 |
| Entire series of insertions | N | N | N | N |
| Entire series of deletions |  | N | N | N |
| Entire file | N | N | N | N |

The **skip list** was relatively fast at running time for all four files. This was due to its uniform complexity of O(log(N)) when accessing data. However, skip list did spend more time when running file4, which consisted of random insertions and deletions. This was because random accesses required longer time to reach through the file. Because data in file 1, file 2 and file 3 was sorted, the average running time of O(log(N)) was guaranteed.

The **binary search tree** operations were mostly slow except for file 4. For file 1, 2 and 3, the running times all exceeded 5 minutes. The total running time was very slow because operations in all three files are all in sorted order. In file 1, the insertions from 1 to 500,000 made the binary search tree in the worst form—it only had its right children and grandchildren, with a depth of 500,000. Because of the ascending pattern of the numbers, every insertion had time complexity of O(N). The total complexity then became O(N^2). Similarly, in file 2 and 3, the insertions were all very time consuming. The deletions in file 2 was faster than those in file 3 because they were in reversed order, making every deletion of time complexity of O(1). However, the total running times for the two files were still very slow (O(N^2)) due to the slow insertions. For file 4, binary search tree performed much better due to the random insertions and deletions. The running time was actually the fastest among all trees. Because the operations were random, the tree was well balanced, making each individual access time O(log(N)) and total running time O(Nlog(N)). Therefore, binary search tree is the best data structure to be used for random data types.

The **AVL tree** operations on file 1, 2, 3, 4 were not very fast, but quite consistent. Even though it had individual insertion and deletion complexities of O(log(N)), because it needed to be balanced after every insertion or deletion, a large number of rotations are needed. AVL trees were slower than skip list for this reason. Clearly, AVL trees cannot be particularly fast with any types of data. However, they can guarantee stability.

The **splay tree** operations were very fast for file 1, 2, and 3. On these three files, it out performed skip list. For file 4, it had only average performance. In file 1, all insertions, which were from1 to 500,000 were done in O(N) time. Because the i-1^th insertion was splayed up to the root, the ith insertion only needed to be added to the right branch of the root which only requires O(1) time. In file 2, after the first deletion, which brought up the “1” from the bottom of the tree, the following entries were also brought next to “1”. Therefore, the remaining deletions all had time complexity of O(1). In file 3, the deletions are all in descending order (same as the tree’s order from root); thus, the running time complexity was O(N) as well. Therefore, operations on these three files were much faster than skip list’s O(Nlog(N)). In file 4, because the insertions and deletions were random, the time complexity for each access was regularly O(logN).

The **B tree** had very different performances on file 1, 2, 3, and 4. It performed best on file 1. For insertions from 1 to 500,000, the B tree split whenever it truly needed to. For file 2, and 3, the operations also weren’t slow because the splits and merges are also sufficient. File 3 out performed file 2 because deletions in file 3 were from right to left—250,000 to 1 (the nodes looked to the left sibling first). File 4 was the slowest because random insertions and deletion caused many more unnecessary splitting of the nodes.

Looking at running times of B trees with different M and L values, the slowest was the B tree with M=1000 and L=2; it is the worst case because the leaves needed to split very often and that slowed the operation down. The best case was when M = 3, L = 200, because in this case, number of splits was minimized. The worse case was when M = 1000. In this case, when inserting or deleting, the whole key list had to be traversed through. Because when M = 1000, the list was very long, it was very time consuming.

The **binary heap** had a very fast running time on file 1 due to its special structure(beat skiplist). Because the two children were always smaller than their parent and the tree was implemented in an array, every insertion to the end of the entry list was O(1) (where skip list required O(logN)). File 2 was slower because the array needed to be adjusted when deleting the smallest number in the array (root) every time (similar to skip list). File 3 was faster than file 2 because the array did not need to be adjusted since after deletion was from the end of the list (beat skip list). For file 4, which consisted of random accesses, the running time was similar to file 2 because after locating the entry to be deleted (O(logN)), the binary heap often needed to adjust itself (still bear skip list). Clearly, binary heaps are a better choice when dealing with ascending-ordered data.

Hash tables were mostly faster than trees (with proper load factors), with complexities of O(1).

The **separate chaining hash** had very short running time on all four files because each insertion is O(λ+1). Particularly, with load factor > 10, for file 3 it should be the faster than file 2 (even though not the fastest accoding to the recorded time) because the deletions in file 3 started from 250,000 to 1, which made each deletion happen at the beginning of the linked list (O(1+1)). In file 2 (deletion from 1 to 250,000), each deletion was O(λ+1). For file 4, which consisted of random accesses, the running time was much longer than the other three files. This was because when randomly deleting an entry, searching through the lists takes longer.

As the load factors got larger, the running time increased. This was because going through a larger linked list correspondent to a spot in the hash table took a longer time. However, if load factor was equal to 0.5, rehashing became more frequent. Therefore, the perfect load factor for separate chaining hash isλ= 1.

The **quadratic probing Hash** was faster than separate chaining hash because accessing the hash table doesn’t need to go through a linked list. It was the fastest among all ADTs with a total time complexity of O(1) as well. Quadratic probing hash performed best on file 1, which requires only insertions. For file 2, 3, and 4, deletions took a little more time. But operations on all files are fast in general.

As load factors got closer to 0.1, the running time was increased because more rehashing was completed, which was time consuming. The best load factor was 0.5. The running time forλ> 0.5 increased again. That was because spaces weren’t enough in the hash table, and more collisions occurred.

The **quadratic probing pointer** hash was also fast in general but much slower than quadratic probing hash on all four files. This was because having a pointer point to a memory location and deconstructing it took more time. Quadratic probing pointers behaved in similar pattern as quadratic probing hash did. The influence of load factor on it was also similar to that on quadratic probing.