**Report - Skip List Implementation**

A **skip list** is a [data structure](https://en.wikipedia.org/wiki/Data_structure) that allows fast search within an [ordered sequence](https://en.wikipedia.org/wiki/Ordered_sequence) of elements. Fast search is made possible by maintaining a [linked](https://en.wikipedia.org/wiki/Linked_list) hierarchy of subsequences, with each successive subsequence skipping over fewer elements than the previous one. Searching starts in the sparsest subsequence until two consecutive elements have been found, one smaller and one larger than or equal to the element searched for. Via the linked hierarchy, these two elements link to elements of the next sparsest subsequence, where searching is continued until finally we are searching in the full sequence. The elements that are skipped over may be chosen probabilistically or deterministically, with the former being more common.

It operates with the following complexity:

|  |  |  |
| --- | --- | --- |
|  | Average case | Worst case |
| Space | O(n) | O(n log n) |
| Search | O(n log n) | O(n) |
| Insert | O(n log n) | O(n) |
| Delete | O(n log n) | O(n) |

**Code base:**

Skip list interface: http://www.utdallas.edu/~rbk/teach/2016f/java/code/SkipList.java

Skeleton implementation: http://www.utdallas.edu/~rbk/teach/2016f/java/code/SkipListImpl.java

Driver file: http://www.utdallas.edu/~rbk/teach/2016f/java/code/SkipListDriver.java

References: Pseudo code from Professor’s notes.

https://en.wikipedia.org/wiki/Skip\_list

**Input format**

Input is read through a file. It has type of operation and the value for it respectively.

Ex: Add 1

**Output format**

Count based on the type of operation performed and the time taken.

**The operations are performed as below:**

a. Add x: Add new items to the skip list; contributes 1 to result.

b. Ceiling x: Find least element that is >= x; returned value is added to result, or 0 if no such element.

c. FindIndex x: Find the element at the index position x; returned value is added to result. The first element of the list is at index 0.

d. First: Find the first element; value of returned element is added to result.

e. Last: Find the last element; value of returned element is added to result.

f. Floor x: Find the Greatest element that is <= x, returned value is added to result, or 0 if no such element.

g. Remove x: Remove x from this list; add 1 to result if x is in the list.

h. Contains x: Add 1 to result if x is in the list.

|  |  |
| --- | --- |
| **Sample input**  Add 1  Add 2  Add 13  First  Last  Ceiling 10  Remove 1  Remove 22  End | **Sample output**  31 |

**Results:**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Files | **Skip List** | | | | | | |
| Time | | Memory | | | Output | |
| A1 | 28 msec | | 1 MB / 61 MB | | | 117 | |
| A2 | 31 msec | | 1 MB / 61 MB | | | 584 | |
| A3 | 40 msec | | 1 MB / 61 MB | | | 2583 | |
| A4 | 95 msec | | 2 MB / 61 MB | | | 54501 | |
| A5 | 371 msec | | 12 MB / 61 MB | | | 539108 | |
| A6 | 916 msec | | 24 MB / 77 MB | | | 343528 | |
| A7 | 1102 msec | | 11 MB / 77 MB | | | 858099 | |
| A8 | 7394 msec | | 130 MB / 362 MB | | | 130122 | |
| A9 | 500607 msec | | 51 MB / 165 MB | | | 626281 | |
| B1 | 22 msec | | 1 MB / 61 MB | | | 36 | |
| B2 | 31 msec | | 1 MB / 61 MB | | | 73 | |
| B3 | 43 msec | | 1 MB / 61 MB | | | 150 | |
| B4 | 109 msec | | 2 MB / 61 MB | | | 721 | |
| B5 | 395 msec | | 13 MB / 61 MB | | | 7097 | |
| B6 | 858 msec | | 12 MB / 61 MB | | | 35313 | |
| B7 | 1327 msec | | 18 MB / 77 MB | | | 70918 | |
| B8 | 7824 msec | | 22 MB / 1362 MB | | | 709447 | |
| **Tree Map** | | | | | | | |
| 7472 msec | | 69 MB / 179 MB | 2 msec | 69 MB / 179 MB | 0 msec | | 69 MB / 179 MB |

Conclusion:

Based on the results of memory utilization and execution times from the above table, we can conclude that Skip List is highly efficient than Tree Map.