Skip Lists

Xingzhi Zhang **Buyi Lv** **Wei Zhou**

3220104936 **3220102745** **3210103790**

Group 4

1 Introduction

Skip lists are a data structure that offers an alternative to balanced trees. Unlike balanced trees, skip lists employ probabilistic balancing, resulting in simpler and faster algorithms for insertion and deletion operations compared to their balanced tree counterparts.

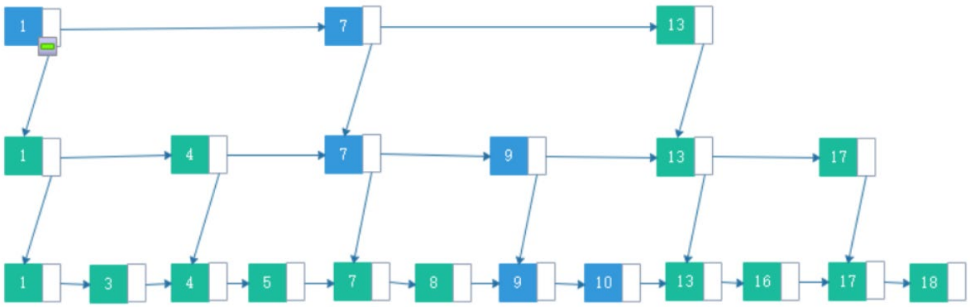
This report aims to introduce skip lists and address the implementation of insertion, deletion, and searching in skip lists. Furthermore, we seek to provide a formal proof demonstrating that the expected time complexity of skip list operations is O(log N). To support our analysis, we generate test cases of varying sizes to illustrate the time-bound characteristics.

2 Data Structure/Algorithms

2.1 Overview

To address the issue of the ordinary singly linked list requiring an $O(n)$ cost for search operations, we introduce a multilevel indexing approach for the list. Here, we first present a naive method.

Specifically, assuming the original list as the 0th level, we choose every other node and create a copy of it, connecting them together to form a new linked list that represents the 1st level. We then repeat the same operation on the 1st level to obtain the 2nd level, and so on.



A skip list possesses several key properties:

- A skip list consists of multiple levels.
- Nodes in each level have a probability of being selected for the next level.
- The first level of the skip list includes all the elements.
- Each level of the skip list functions as a regular list.
- An element in the i-th level always points to another element with the same key using the down pointer.
- Within each level, there exists a head node with the minimum key and a tail node with the maximum key.

2.2 Find

For search operations, starting from the top level, we follow the following procedure: If the value of the next index in the current level is greater than the target value or the index does not exist (assuming the ordered list is sorted in ascending order), it indicates that the target value should fall within the interval formed by the current index and the next index. In this case, we traverse to the next level by following the down pointer of the current index.

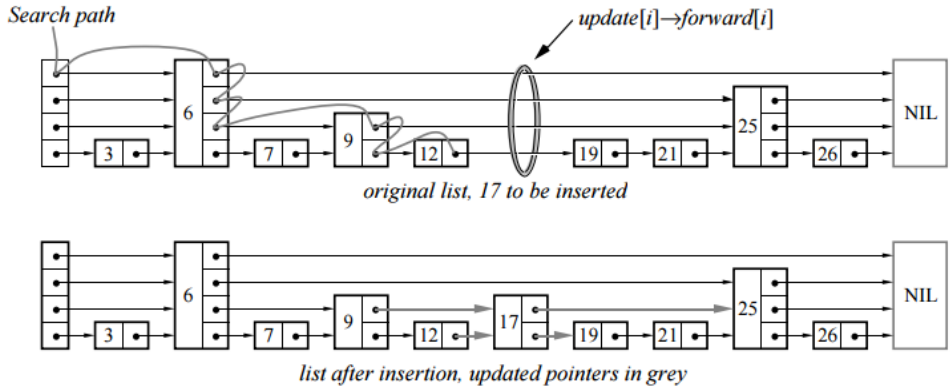
Otherwise, we continue moving along the current level's linked list until the aforementioned condition is met.

If it becomes impossible to move downwards, it signifies that we have reached the 0th level, which corresponds to the original list. If, even at this stage, the target value cannot be found, it implies that the target value does not exist in the list.

2.3 Insert

By using the Find method, we can identify the appropriate position for insertion. It is worth noting that the above method may result in the insertion value falling within the same interval without creating an index, thereby causing the skip list to degrade into a regular linked list. Therefore, we propose some improvements to the indexing strategy of the skip list.

Indexing Strategy: For each element in the *i-th* level, we assign it a probability of p to become an index in the *(i+1)-th* level. In other words, the probability of an element becoming an index in the first level is ***p***, in the second level is p^2, and so on. We will demonstrate the efficiency of this method in subsequent analysis.



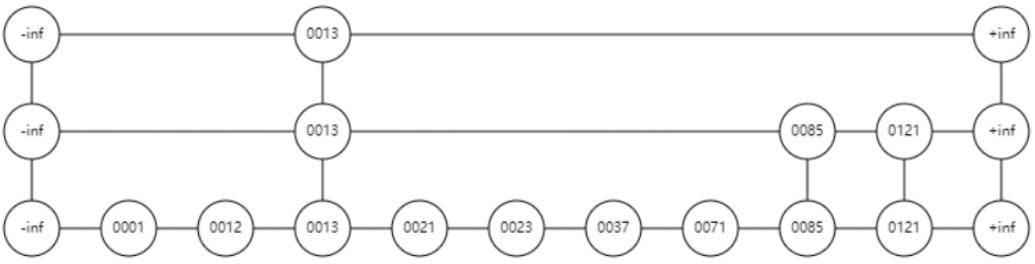
2.4 Delete

To remove a node, we can utilize the Find method to locate the corresponding node. If the node has associated indexes, all indexes must be deleted concurrently. Similar to the search operation, we can initially find the node that points to the node with the specified key, which is the node we want to delete. Then, we update the next pointer of this node to point to the node following the one we want to delete. Finally, we release the memory occupied by the node we wish to delete.

3 Testing Results[1]

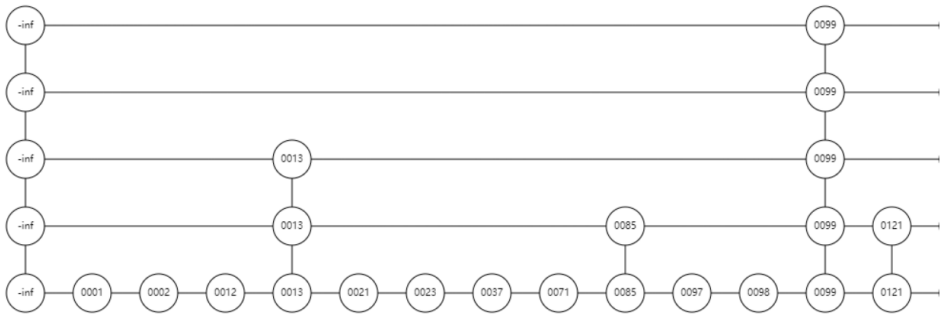
3.1 Test the Correctness

First, we use small-size data to test our program's correctness. When we input data [12, 13, 21, 23, 37, 71, 85, 121，1]. And the result is shown below:

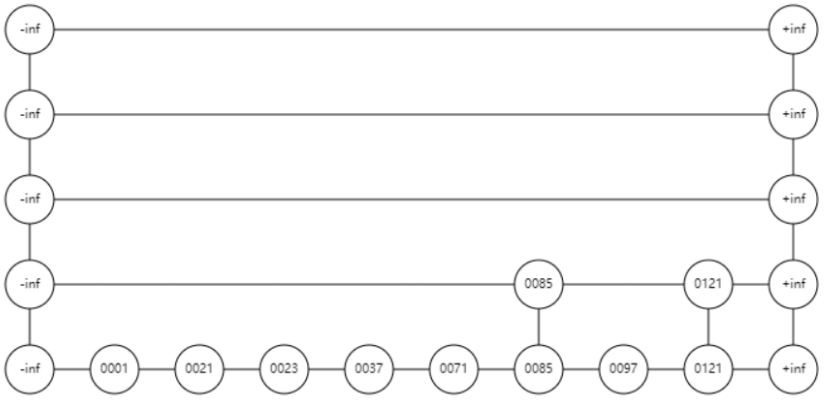


[1] The experiment is conducted in an Ubuntu 20 environment.

When we continue to insert data [2, 99, 98,97, 11]. And the result is shown below:



When we delete [2,99,98,12,13]. And the result is shown below:



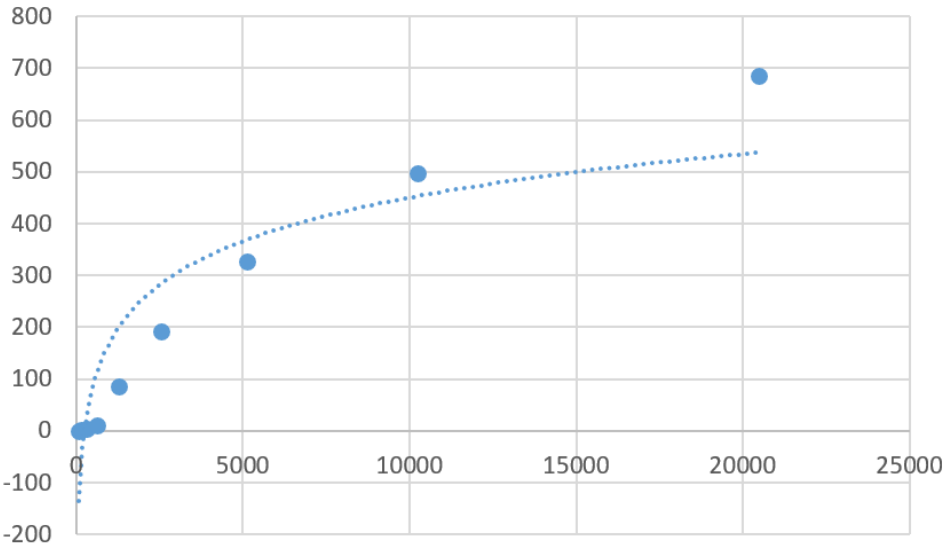
3.2 Test the Time Complexity

To conduct time complexity testing, we employ a random method to generate a series of test cases in this project. When examining data with small input sizes, it is advisable to use multiple test cases within a single test instance. Therefore, we utilize the random method multiple times within each test instance.

Only when inserted, the performance when the inserted value is N is respectively (each is tested for 10 times and record the average time)：

N	80	160	320	640	1280	2560	5120	10240	20480
Time(ms)	0.5	1.6	4.8	13.1	93.1	199.4	395.1	540.5	744.2

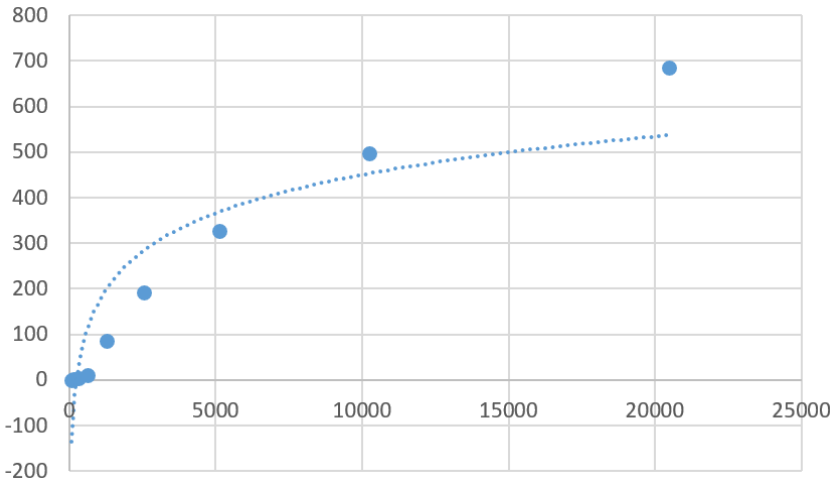
After drawing for figure as follows:



When insertion, access and deletion each account for one third, and the total number of operations is N, the performance is

N	80	160	320	640	1280	2560	5120	10240	20480
Time(ms)	0.3	1.2	4.2	11.4	86.7	191.1	327.1	497.2	685.4

After drawing for figure as follows:



As we can see, the time complexity of all there operations is approximately $O(log\ N)$.

4 Analysis and Comments

4.1 Time Complexity

We only need to analyze the time complexity of the Find operation caused by the indexing strategy used during Insertion since both Insert and Delete operations rely on Find implementation.

Let $C(k)$ represent the expected number of steps required to find the target node when currently at a node in the *k-th* level. Assuming the 0-th level of the list is infinitely long, we pessimistically assume that each node in the *(k-1)-th* level has a probability p of creating an index in the *k-th* level. Therefore, the current node has a probability p of moving to the *(k-1)-th* level, and the remaining probability is to move one step to the right in the current level. Thus, we have $C(k) = pC(k-1) + (1-p)C(k) + 1$. Simplifying this equation, we obtain $C(k) = k/p$.

However, the *0-th* level of the list is not infinitely long but has a length of n. From this, we can determine the maximum number of levels required: Since each node has a probability p of creating an index, in order to minimize the number of nodes in the highest level (having only one node), we require an expected number of $log_{1/p}n$ levels. Therefore, the expected complexity of the Find operation is $C(log_p n) = (log_{1/p} n)/p$. When taking $p = 1/2$, the expected time complexity of the Find operation becomes $O(log_2 n)$, namely $O(logn)$

4.2 Space Complexity

As for space complexity, when $p = 1/2$, the total space occupied is

$$n + np + np^2 + np^3 + \cdots = O(n)$$

5 References

[1] William Pugh, “Skip Lists: A Probabilistic Alternative to Balanced Trees”, 2012.12