浙江大學

Skip Lists

Advanced Data Structures and Algorithm Analysis
Research Project 7

朱璟森	童鑫远	陶泓羽
3170104166	3170103148	3170102625

June 10, 2019

Contents

1 Introduction

1.1 Background of Skip List

Linked List is a common linear data structure, with an $O(N)$ complexity for search, which is inefficient. To speed it up, we can use balanced search trees such as AVL Tree, Red-Black Tree, Splay Tree, etc., with an $O(\log N)$ complexity for search, insertion and deletion.

However, balanced search trees are complicated and difficult to implement. Fortunately, we have **Skip List**. Based on randomization, skip list supports both searching and insertion in $O(\log N)$ expected time, but easier to implement. Skip List is widely used in databases such as *Redis* and *LevelDB*.

1.2 Problem Description

This project requires us to introduce the skip lists, and to implement insertion, deletion, and searching in skip lists. A formal proof is expected to show that the expected time for the skip list operations is $O(\log N)$. We also generate test cases of different sizes to illustrate the time bound.

2 Algorithm Specification

2.1 Description of Skip List

The structure of an ordinary ordered Linked List is as follows (H and T refers to the head node and tail node of the list, which contains no valid value):

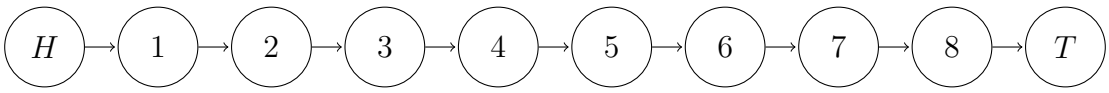

Figure 1: Ordered Linked List

If we want to find 7 in the list, we'll have to sequentially search $1 \to 2 \to \ldots \to 7$, which is $O(N)$. But in an ordered array, we can use binary search to reduce the complexity to $O(\log N)$. But a linked list doesn't support randomized access, thus we can't use binary search method.

A skip list is bulit in levels. If we store some internal nodes in a higher level, we can support a searching method similar to binary search.

Now we search 7 again, starting from head node H:

1. Compare 4 and 7, 7 > 4; compare 8 and 7, 7 < 8, go down at 4
2. Compare 6 and 7, 7 > 6; compare 8 and 7, 7 < 8, go down at 6
3. Compare 7 and 7, 7 = 7, found!

The process of finding 7 is as figure 2:

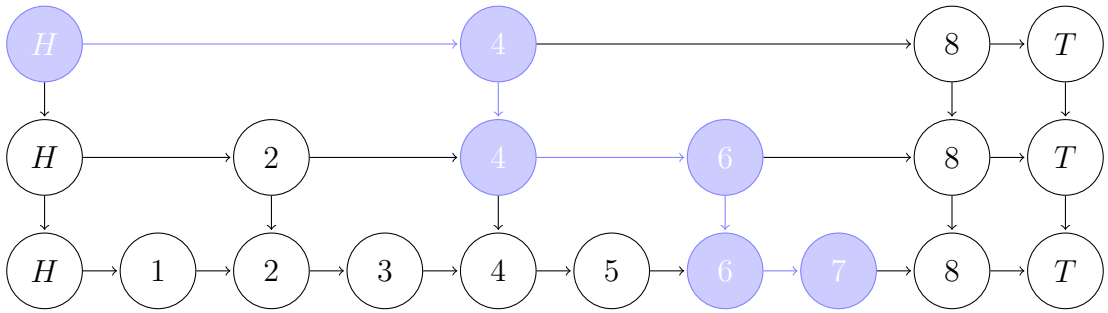

Figure 2: Structure of Skip List and the search path of finding 7

In other words, skip list stores some indices of the binary search, which combines the structure of linked list and the method of binary search.

2.2 Data Structure of skip list

Firstly, we define the structure of nodes in the skip list:

```
struct SkipListNode {
    int value;
    int level;
    SkipListNode **next_nodes;
};
```

`value` refers to the value in the node, and `level` refers to the level. The difference between our implementation of skip list and ordinary linked list is that we store the successor nodes in an array, so that the same node in different levels shares the same storage space. `next_nodes[i]` is the successor node in level `i`. For example, for the node 4 in figure 2, `next_nodes[0]` is 5, `next_nodes[1]` is 6, and `next_nodes[2]` is 8. In this way, we can decrease the space complexity and reduce the structure in figure 2 to as follows:

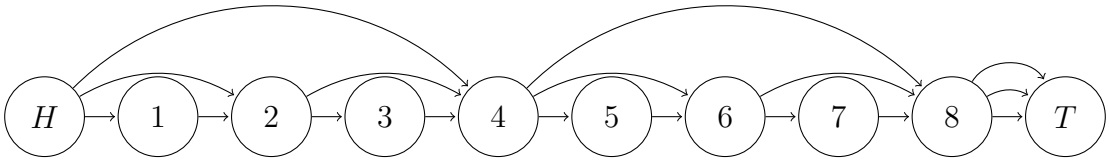

Figure 3: Skip List with shared nodes

Then, we define the structure of the skip list:

```
class SkipList {
    public:
        SkipList(int max_level); //Constructor
        void Insert(int value);
        void Delete(int value);
```

```
6         SkipListNode *Find(int value);
7         void show(); //to print the list
8     private:
9         int Max_Level;
10        SkipListNode *head;
11        SkipListNode *tail;
12        static int random_level(int max); //Determine the
              level of a node randomly
13 };
```

2.3 Operations on skip list

2.3.1 Search

A brief description of search is shown by a specific case in the last section. Now let's introduce it formally.

1. Start at the $node = H$ and $level = max\_level - 1$
2. Move ahead to next node until tail or $value \leq node.value$
3. If $value = node.value$ return $node$, else $level = level - 1$ and repeat step 2
4. If $level = 0$ and still not found, return $null$

In pseudo-code:

Algorithm 1 Skip List: Find

```
1:  procedure FIND(value)
2:      node ← H
3:      for level ← max_level − 1 to 0 do
4:          while node.next_nodes[level] ≠ T and node.value < value do
5:              node ← node.next_nodes[level]
6:          end while
7:          if node.value = value then
8:              return node
9:          end if
10:     end for
11:     return null
12: end procedure
```

2.3.2 Insertion

There are 3 steps when we insert a new value into a skip list: finding the predecessors, creating a new node and finally insert into the list:

1. Finding the predecessors:
   Firstly, we need to locate the position of the newly-inserted node in the list by

finding its predecessors. This step is similar to the find operation, just to find the last node whose value is less than the new value in each level.

2. Creating a new node:
   In this step, we construct a new node and choose a number k with a randomized strategy (which will be mentioned below) as the number of levels of the node.
3. Inserting into the list:
   For level 0 to k, insert the new node after its predecessor just like ordinary linked list.

In pseudo-code:

Algorithm 2 Skip List: Insert

```
1:  procedure INSERT(value)
2:      //Step 1: Finding the predecessors
3:      predecessors ← new Node[Max_level]
4:      node ← H
5:      for level ← max_level − 1 to 0 do
6:          while node.next_nodes[level] ≠ T and node.value < value do
7:              node ← node.next_nodes[level]
8:          end while
9:          predecessors[level] ← node
10:     end for
11:     //Step 2: Creating a new node
12:     k ← RANDOM_LEVEL(Max_level)
13:     new_node ← new Node(value, k)
14:     //Step 3: Inserting into the list
15:     for level ← 0 to k do
16:         Insert new_node after predecessors[level]
17:     end for
18: end procedure
```

For example, if we want to insert 9 into the list in figure 3, and suppose we get $k = 2$, then the process is as follows (blue nodes and arrows denotes the search path to locate the predecessor, yellow node is the predecessor, red node denotes the new node. Dashed arrows and red thick arrows denote the old and updated "next" relations):

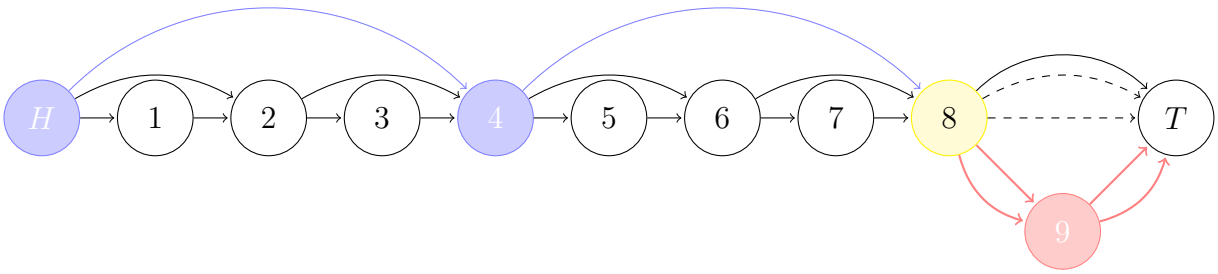

Figure 4: Process of inserting 9

2.3.3 Deletion

The process of deletion is similar to insertion in 2 steps: finding the predecessor and delete the node.

Algorithm 3 Skip List: Delete

```
1:  procedure DELETE(value)
2:      //Step 1: Finding the predecessors
3:      predecessors ← new Node[Max_level]
4:      node ← H
5:      for level ← max_level − 1 to 0 do
6:          while node.next_nodes[level] ≠ T and node.value < value do
7:              node ← node.next_nodes[level]
8:          end while
9:          if node.next_nodes[level] ≠ tail and node.next_nodes[level].value =
    value then
10:             predecessors[level] ← node
11:         else
12:             predecessors[level] ← null
13:         end if
14:     end for
15:     //Step 2: Deleting the node
16:     for level ← 0 to k do
17:         Delete new_node after predecessors[level]
18:     end for
19: end procedure
```

For example, if we want to insert 6 in the list in figure 3, then the process is as follows (blue nodes and arrows denotes the search path to locate the predecessor, yellow node is the predecessor, red node denotes the new node. Dashed arrows and red thick arrows denote the old and updated "next" relations):

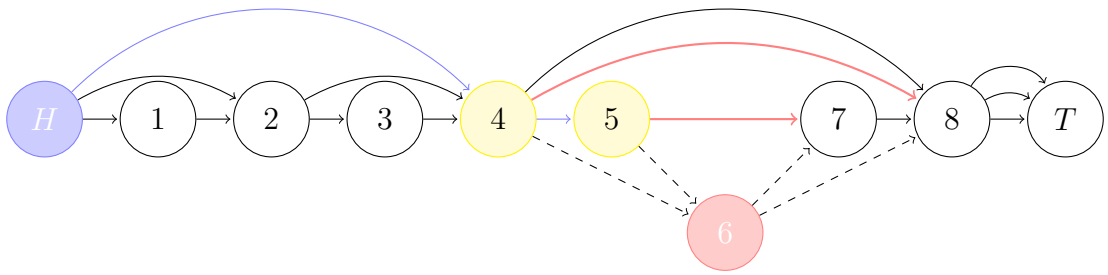

Figure 5: Process of deleting 6

2.3.4 Randomization

Finally, we introduce the random strategy to determine the level k of a new node in the process of insertion.

k is a random number between 0 and Max_level. However, in order to make the expected result to similate binary-search, we don't generate uniform random (i.e. `k=rand()%Max_level`), but make k follows **Geometric Distribution** with $p = \frac{1}{2}$. The algorithm is like flipping a coin — If the coin faces up, the level increases by 1, or else stop at current level:

Algorithm 4 Random Level

```
1:  procedure RANDOM_LEVEL
2:      level ← 0
3:      while level < Max_level − 1 do
4:          if Random{0, 1} then
5:              level ← level + 1
6:          else
7:              break
8:          end if
9:      end while
10:     return level
11: end procedure
```

Suppose the maximum level is M, then

$$P(k = n) = \begin{cases} (\frac{1}{2})^{n+1}, \ n < M - 1 \\ (\frac{1}{2})^{n} = (\frac{1}{2})^{M-1}, \ n = M - 1 \end{cases}$$

If the size of the skip list is N, then the expected number of nodes in the k_{th} level is $E(N_k) = \frac{N}{2^k}$, each level decreases by $\frac{1}{2}$, just like the process of binary search.

3 Testing Results

In this section, we test the running time of insertion, deletion and searching. We use `time()` function in C standard library to record the total time of N operations (insertion/deletion/search). For small N, the running time is too short (less than 1 ms), we make several iterations of the same operations and compute the average time.

As insertion/deletion will modify the skip list, the size of the skip list will change after each operation. Thus, we only record the total running time of N insertions/deletions, instead of dividing N to get the average time per operation (which is meaningless because the size of the skip list isn't fixed). But for searching, the size of the skip list doesn't change after each operation, so we can divide N to get the average time per operation.

3.1 Insertion

In this test, we insert N values from 0 to $N-1$ into a skip list with $Max\_level = 32$ in an order of

- Increasing, i.e. $0, 1, 2, \ldots, N-1$
- Decreasing, i.e. $N-1, N-2, \ldots, 1$
- Random

And we record the total time of N insertions. The result is as follows:

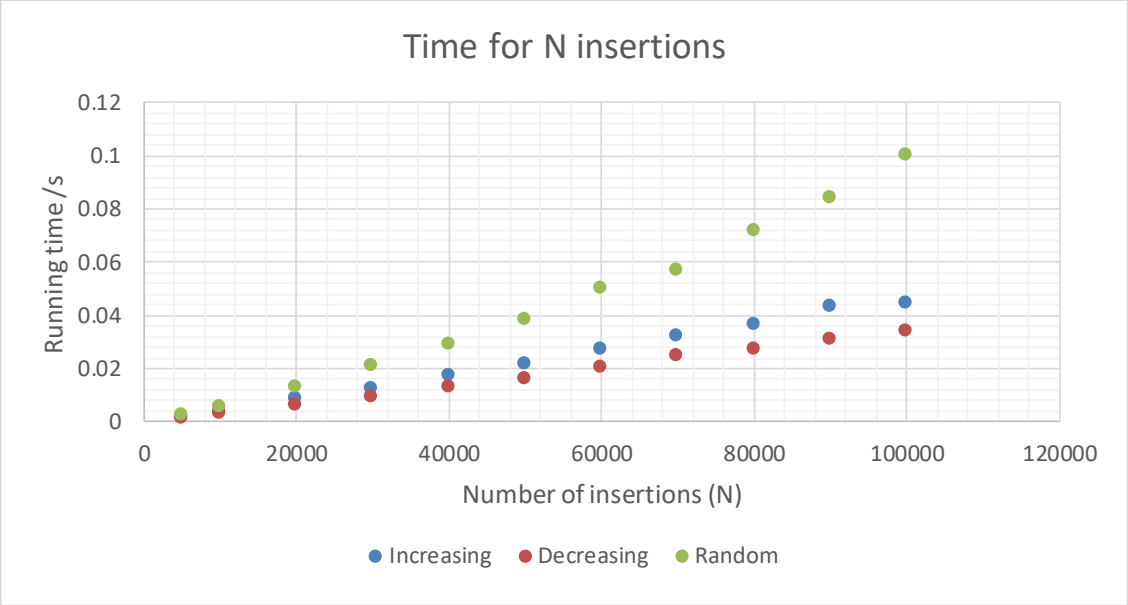

Figure 6: Running time for 3 insert orders

3.2 Deletion

We first create a skip list with $Max\_level = 32$ and insert N values from 0 to $N-1$ into it. Then, we delete the values in an order of

- Increasing, i.e. $0, 1, 2, \ldots, N-1$
- Decreasing, i.e. $N-1, N-2, \ldots, 1$
- Random

We use the same method as we did in the insertion test to record the total time of N deletions. The result is as follows:

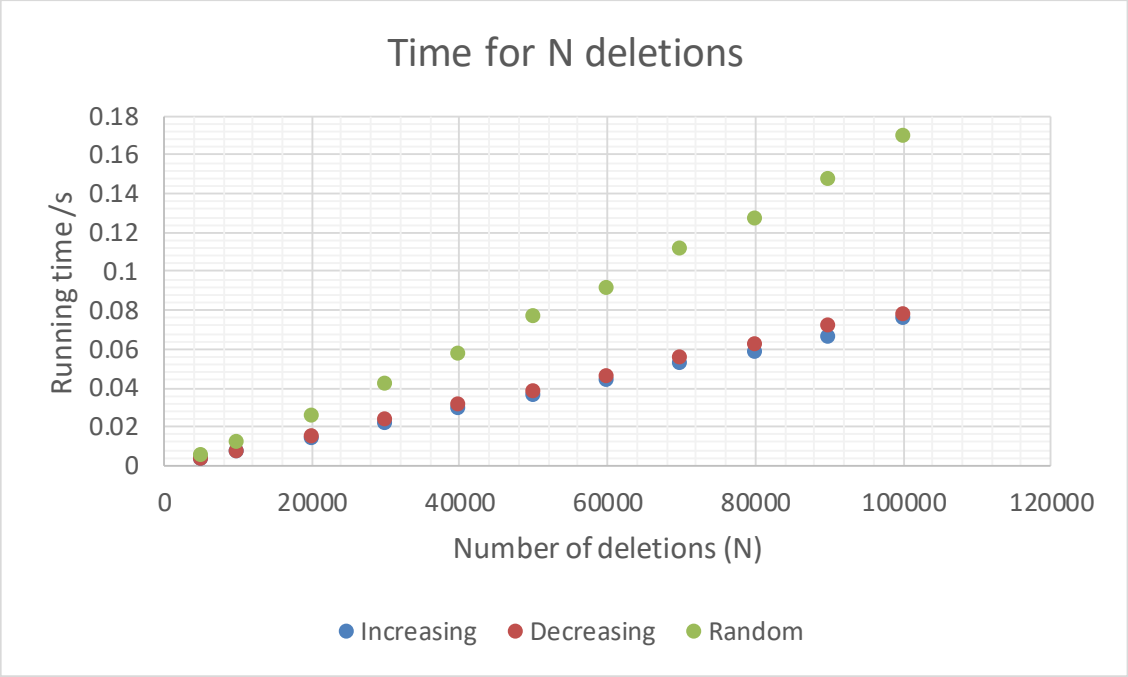

Figure 7: Running time for 3 delete orders

3.3 Search

Finally, we test the time of the find operation.

We first create a skip list with $Max\_level = 32$ and insert N values from 0 to $N - 1$ into it. Then, we find a random value between $[0, N)$ from the skip list for N times. The average time per searching is $\frac{\text{total time}}{N}$. The result is as follows:

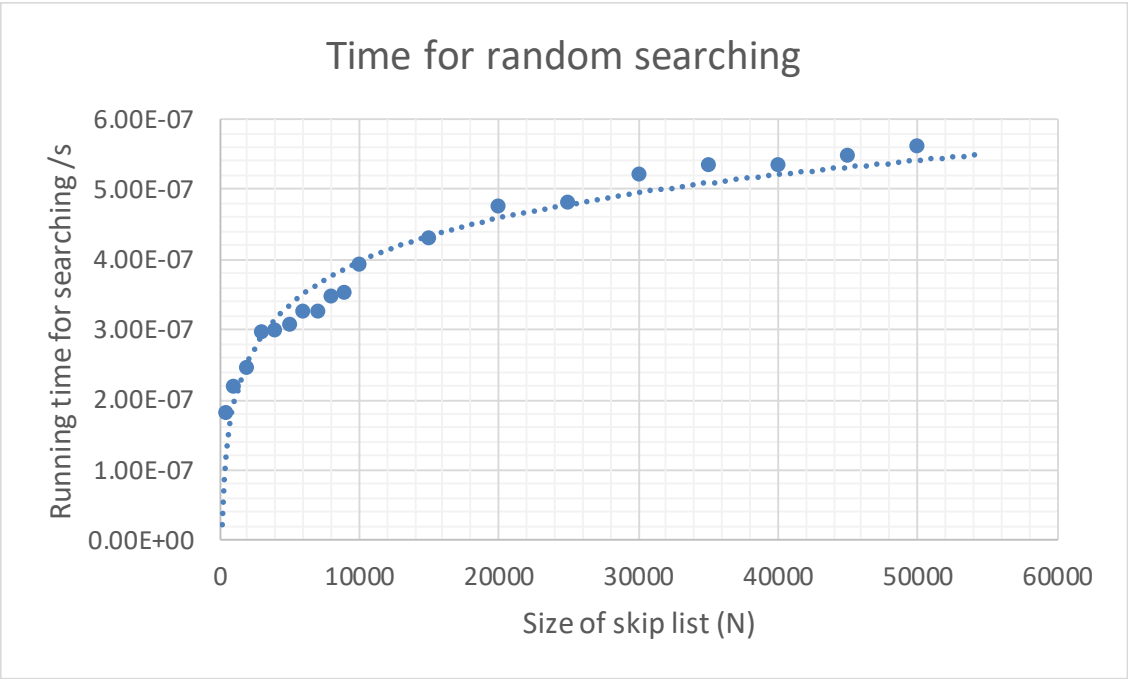

Figure 8: Running time for searching

From the figure, we observe that the time complexity of finding seems to be $O(\log N)$.

4 Analysis and Comments

Because skip list is based on a randomized algorithm, all the complexities analyzed below is the mathematical expectation. The worst-case complexity might be worse, but only with a very small probability.

4.1 Time Complexity

In this section, we will prove that the expectation of time complexity of search, insertion and deletion is $O(\log N)$.

4.1.1 The expected Max_level

Let p denotes the probability of increasing the level in the random_level algorithm 4 (In our implementation, $p = \frac{1}{2}$). For an arbitary node, let k denotes the level of it, then $P(k = i) = p^i(1-p)$(i.e. Geometric distribution). Then, the expected number of elements in the k_{th} level (denoted N_k) is

$$E(N_k) = \begin{cases} N,\ k = 0 \\ N(1 - \sum_{i=0}^{k-1} p^i(1-p)) = N(1 - \frac{(1-p)(1-p^k)}{1-p}) = Np^k,\ k \geq 1 \end{cases}$$

To make the search efficient, we want the highest level $M-1$(M is the total number of levels) has an expected $O(1) = a$ number of elements, i.e.

$$\begin{aligned} E(N_{M-1}) &= Np^{M-1} = a \\ \log_p N + M - 1 &= \log_p a \text{ (Logarithm on both sides)} \\ \therefore M &= -O(\log_p N) = O(\log_{\frac{1}{p}} N) \end{aligned}$$

Therefore, the Max_level of the skip list can be set to $\log_{\frac{1}{p}} N$ to obtain maximum expected efficiency. In computer, the range of integer is $[0, 2^{32})$, so Max_level is often set to be 32 in practical.

4.1.2 The expected search cost

We use a reverse analysis method to analyze the cost. Suppose we have found the node in the k_{th} level, we analyze the search path backwards, going up and left to the head H. At an arbitary node x in the i_{th} level in the backward-path, there are 2 situations:

1. The level of node x is exactly i, then the next step is to go left.
2. The level of node x is larger than i, then the next step is to continue to climb up.

Let's take the case of finding 7 in section 2 as an example. The dashed arrows denote the path backwards. The red 6 and 4 nodes satisfy 2 situations above respectively:

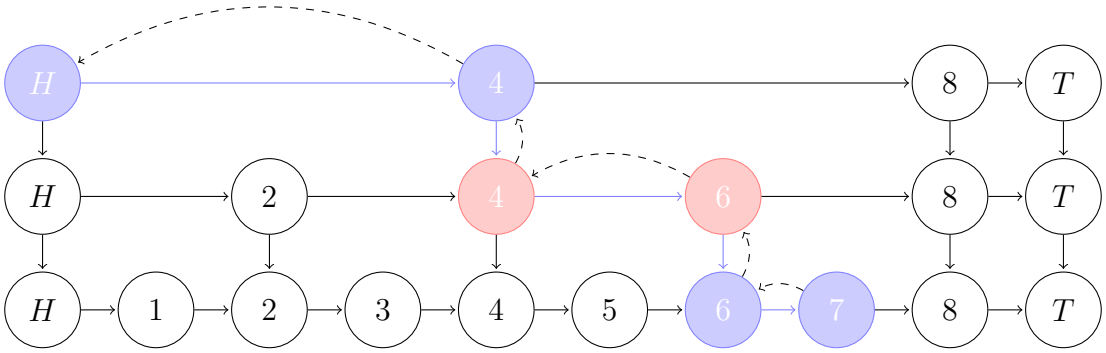

Figure 9: The search path backwards from 7

The red node 6 is in level 1, equals to the maximum level of itself, so satisfies case 1; while the red node 4 is in level 1 but the maximum level of itself is 2, which satisfies case 2.

Because the level follows geometric distribution, in an arbitary node x, it has a probability of $1-p$ to satisfy case 1 while p to satisfy case 2. Let $C(k)$ denotes the expected cost of search path that climbs up k level, then

$$
\begin{aligned}
C(0) &= 0 \\
C(k) &= (1-p)(1+\text{cost in situation 2}) + p(1+\text{cost in situation 1}) \\
&= (1-p)(C(k)+1) + p(C(k-1)+1) \\
\therefore C(k) &= \frac{1}{p} + C(k-1) \\
\therefore C(k) &= \frac{k}{p}
\end{aligned}
$$

Because $k \leq Max\_level = O(\log_{\frac{1}{p}} N)$, thus $C(k) = O(\log N)$. The time complexity of search is proportional to the cost of the search path, thus we finish the proof that the expected time of search is $O(\log N)$.

4.1.3 The expected insertion/deletion cost

The insertion/deletion takes 2 steps:

1. Find the predecessor
2. Insert/Delete

Step 1 costs $O(\log N)$ while step 2 costs $O(k) \leq Max\_level = O(\log N)$ where k is the level of the node to insert/delete, thus the expected time of insertion/deletion is also $O(\log N)$.

4.2 Space complexity

The level L follows geometric distribution, so the expectation of L is

$$E(L) = \sum_{k=0}^{\infty} kp^k(1-p) = \frac{1}{1-p}$$

For each node, it costs $O(L)$ space to store the successors array `next_nodes`. Therefore, the space complexity of a skip list is $O(NL) = O(\frac{N}{1-p}) = O(N)$. The smaller p is, the less space the skip list costs.

Here, we assume that the maximum level is infinity. If max level is finite (denoted M), we have

$$P(L=k) = \begin{cases} p^k(1-p),\ k < M-1 \\ 1 - \sum_{k=0}^{M-2} p^k(1-p) = p^{M-1},\ k = M-1 \end{cases}$$

When $M = O(\log N)$ is large, $(M-1)p^{M-1} \to 0$, the bias caused by assuming infinity level can be ignored. Thus, for skip list with finite level, the space complexity is still $O(N)$.

4.3 Further thinking

As an emerging data structure, skip list exudes its unique brilliance with its high efficiency and simplicity. Compared to balanced search trees, skip list is not inferior to them.

For a long time, in order to optimize the efficiency of searching, a lot of sophisticated tree structures are designed. Despite their efficiency, their large programming complexity is unacceptable. Skip list not only describes the structure itself, but also "skips" the stereotype of the "tree" structure. As a data structure, skip list has been widely used to describe an ordered set instead of using trees in more and more areas. As a way of thinking, "skipping the stereotype" leads mankind to making progress one after another!

Appendices

Source Code

Due to limited space, the complete source code isn't included in this report. You can find the source code in `skip_list.cpp`, thanks.

Author List

- Programmer: Tao Hongyu
- Tester: Tao Hongyu, Zhu Jingsen
- Report Writer: Zhu Jingsen
- PPT and Presentation: Tong Xinyuan

Declaration

We hereby declare that all the work done in this project titled "Skip Lists" is of our independent effort as a group.