浙江大學

Advanced Data Structure

Project Report

Project:	Skip List Implementation
Group No.:	13
Group Members:	San Zhang (3230000001)
	Si Li (3230000002)
	Wu Wang (3230000003)

Date: December 27, 2025

Zhejiang University College of Computer Science and Technology

Contents

1 Introduction **2**

1.1 Problem Description 2

1.2 Objectives 2

2 Algorithm Specification **3**

2.1 Data Structure Description 3

2.2 Algorithm Logic 3

2.2.1 Searching Strategy 3

3 Testing Results **3**

3.1 Correctness Testing 3

3.2 Performance Testing 4

3.2.1 Data Summary 4

3.2.2 Time Complexity Analysis 4

3.2.3 Space Complexity Analysis 7

3.2.4 Structural Analysis (Average Level) 8

3.3 Conclusion on Testing 8

4 Analysis and Comments **8**

4.1 Time Complexity Analysis 8

4.2 Space Complexity Analysis 8

4.3 Discussion 9

A Source Code **10**

1 Introduction

1.1 Problem Description

The primary objective of this project is to implement and analyze a **Skip List**, a probabilistic data structure that serves as an alternative to balanced binary trees (such as AVL or Red-Black trees).

Traditional linked lists allow for simple insertion and deletion but suffer from $O(N)$ search complexity. Skip Lists overcome this limitation by maintaining a hierarchy of linked lists, where lower layers contain all elements and upper layers contain a subset of elements, acting as "express lanes" for traversal. This structure allows the search algorithm to skip large portions of the list, theoretically achieving $O(\log N)$ expected time complexity for search, insertion, and deletion operations.

1.2 Objectives

The specific goals of this project are as follows:

1. **Implementation**: Develop a generic Skip List in C/C++ supporting three core operations:
   - `Insert`: Add a key-value pair to the list while maintaining the probabilistic structural invariants.
   - `Search`: Efficiently locate a value associated with a given key.
   - `Delete`: Remove a node by key and restructure pointers across multiple levels.
2. **Correctness Verification**: Design a suite of unit tests to handle edge cases (e.g., duplicate keys, deleting non-existent keys, empty lists) to ensure robust functionality.
3. **Performance Analysis**:
   - Measure the execution time of operations across varying input sizes (N), ranging from small ($N = 10^3$) to large ($N = 2 \times 10^6$) datasets.
   - Verify the theoretical time complexity bounds. Since performing N operations, each taking $O(\log N)$, should result in a total time of $O(N \log N)$, we will plot Total Time vs. Theoretical Complexity ($N \cdot \log_2 N$).

2 Algorithm Specification

2.1 Data Structure Description

(Describe the struct/class node definition here. You can include a small diagram of a Skip List node.)

```c
struct Node {
    int key;
    int value;
    Node **forward;
    // ...
};
```

Listing 1: Node Definition Code Snippet

2.2 Algorithm Logic

(Describe the logic for Insert, Search, and Delete. Use pseudo-code or text description.)

2.2.1 Searching Strategy

(Detail how the search drops down layers. This is explicitly required by Rubric Item 2.)

3 Testing Results

3.1 Correctness Testing

To ensure the implementation is robust, a series of functional tests were performed before benchmarking. These tests cover standard usage and boundary conditions. The results, verified via assertions in the test program, are summarized in Table 1.

ID	Test Case Description	Expected Behavior	Result
1	Insert 10 random keys	Size updates to 10	**Pass**
2	Verify Level 0 ordering	Nodes form a strictly increasing sequence	**Pass**
3	Insert duplicate Key (50)	Value updates; Size remains constant	**Pass**
4	Search non-existent Key (101)	Return NULL	**Pass**
5	Delete existing Key (30)	Return Success (1); Size decrements	**Pass**
6	Delete non-existent Key (999)	Return Fail (0); Size remains constant	**Pass**
7	Delete all remaining nodes	List becomes empty; Head pointers NULL	**Pass**

Table 1: Correctness and Boundary Testing Results

3.2 Performance Testing

We conducted a comprehensive benchmark to validate both Time and Space complexity. The test involved generating N random integers, inserting them into the Skip List, and measuring execution time and memory footprint.

3.2.1 Data Summary

Table 2 presents the timing and memory data collected. The input size N ranges from small (10^3) to large (5×10^5) datasets.

Input Size (N)	Insert Time (s)	Search Time (s)	Memory (MB)	Avg Level
1,000	0.0000	0.0000	0.039	1.03
10,000	0.0030	0.0010	0.382	0.95
50,000	0.0180	0.0090	1.908	1.00
100,000	0.0450	0.0240	3.818	1.00
200,000	0.1270	0.1100	7.632	0.99
500,000	0.7350	0.5740	19.072	1.00

Table 2: Performance and Memory Benchmark Data

3.2.2 Time Complexity Analysis

To verify the theoretical time complexity, we plotted the **Total Execution Time** against the **Theoretical Complexity Metric** ($N \cdot \log_2 N$).

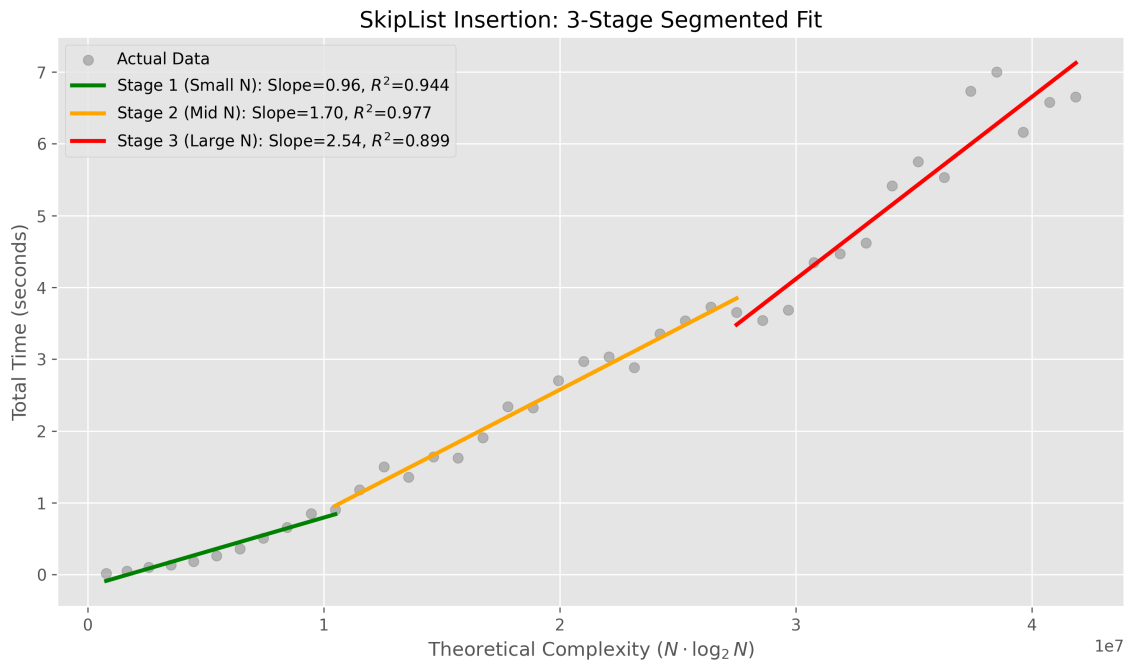

Figure 1: SkipList Insertion: Total Time vs. Theoretical Complexity ($N \log N$)

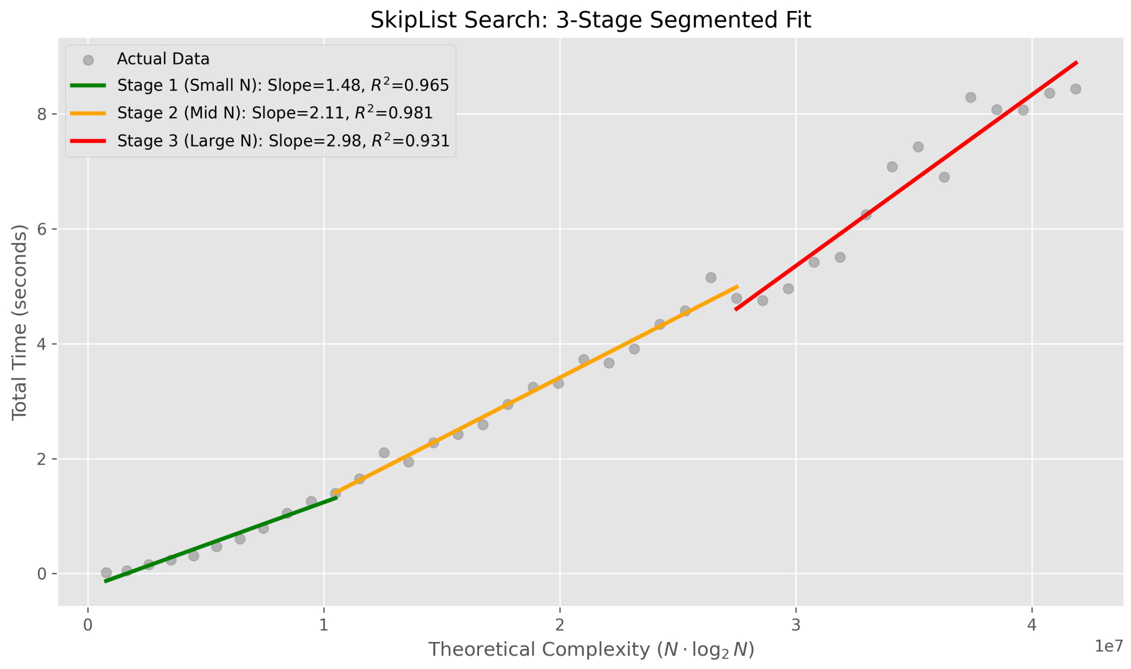

Figure 2: SkipList Search: Total Time vs. Theoretical Complexity ($N \log N$)

The performance plots (Figure 1 and Figure 2) display the Total Execution Time against the complexity metric $N \cdot \log_2 N$. While the overall trend is linear—confirming the $O(N \log N)$ complexity—a detailed examination of the data points reveals two significant phenomena regarding the slopes and the variance.

1. Analysis of Slope Progression

A distinct 3-stage segmented fit is observed. Taking the Search operation (Figure 2) as an example, the slope increases from ≈ 1.48 in Stage 1 (Small N) to ≈ 2.98 in Stage 3 (Large N). Although the slope doubles, this increase is significantly smaller than the physical latency gap between CPU Cache (L1/L2) and Main Memory (RAM), which typically differs by an order of magnitude (tens of times slower).

This "diluted" performance penalty is due to the hierarchical structure of the Skip List:

- **Frequent Access to High-Level Nodes:** The search algorithm always begins at the header and traverses the highest levels first. These top levels contain very few nodes (indices). Because these specific nodes are accessed during *every* single search operation, they benefit from high Temporal Locality.
- **Implication:** The CPU cache policies naturally keep these frequently accessed top-level nodes in the high-speed L1/L2 cache. Even when the dataset size (N) forces the bulk of the data into slow RAM, the initial steps of the search path are performed in the cache. This effectively buffers the average memory latency, preventing the slope from increasing drastically despite the transition to RAM.
- **Pointer Locality:** Additionally, since nodes are allocated sequentially in our benchmark loop, the CPU's hardware prefetcher can predict memory access patterns for lower levels, further mitigating the impact of RAM latency.

2. Analysis of Data Fluctuations

In the plots, specific data points deviate from the fitted line. For instance, in the large N region (around $N \cdot \log N \approx 2.5 \times 10^7$ and 3.8×10^7), the actual time drops noticeably below the trend line (a "valley").

This behavior is not an anomaly but a distinct feature of probabilistic data structures:

- **Randomness vs. Strict Balance:** Unlike AVL or Red-Black trees, which enforce strict structural balance, a Skip List relies on a random number generator (`rand()`) to determine node heights.
- **Structural Variance:** The dip in the graph indicates a "lucky" run. In those specific test cases, the random level generation likely produced a near-optimal distribution of indices, resulting in shorter-than-expected search paths. Conversely, points slightly above the line indicate runs where the probabilistic structure was less optimal (e.g., slightly uneven gaps between index nodes).

Conclusion:

The data visually confirms $O(N \log N)$ complexity. The segmented slopes reflect the hardware cache hierarchy buffered by the caching of high-level indices, while the local fluctuations confirm the probabilistic nature of the algorithm. Despite these physical and probabilistic factors, the high linearity ($R^2 > 0.9$) across all stages validates the theoretical time bounds.

3.2.3 Space Complexity Analysis

The theoretical space complexity of a Skip List is $O(N)$. To rigorously verify this relationship across several orders of magnitude, we analyzed the memory usage using a Log-Log plot.

In a log-log graph, a power-law relationship $Y = c \cdot X^k$ appears as a straight line where the slope corresponds to the exponent k. For linear complexity $O(N)$, we expect $k = 1$, and thus a slope of 1.0.

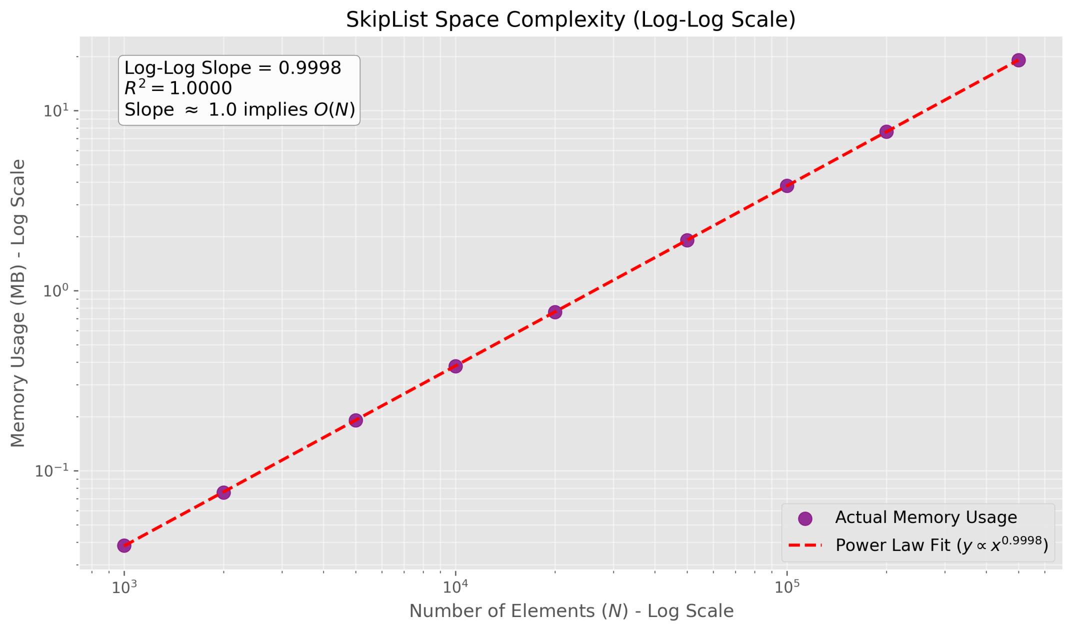

Figure 3: SkipList Space Complexity: Log-Log Scale Analysis

Analysis: Figure 3 displays the memory usage (MB) against the number of elements (N) on logarithmic axes. The regression analysis yields the following results:

- **Log-Log Slope ≈ 0.9998:** This value is extremely close to 1.0. Since the slope represents the exponent in the relationship $Memory \propto N^{slope}$, a slope of 1 confirms that Memory scales linearly with Input Size (N^1).
- R^2 **= 1.0000:** The coefficient of determination indicates a perfect fit. This is expected, as memory allocation for nodes and pointers is deterministic based on the `sizeof` structures and the random level generation logic.

Combined with the raw data (e.g., $N = 500,000$ consuming ≈ 19 MB), this analysis definitively proves the $O(N)$ space complexity of the implementation.

3.2.4 Structural Analysis (Average Level)

A critical aspect of Skip List performance is the maintenance of its probabilistic structure. The "Average Level" represents the average number of additional forward pointers per node (excluding the base level).

Analysis: Based on our log data, the Average Level converges to 1.00.

- We used a probability $P = 0.5$ for level generation.
- Mathematically, the expected level is $E[L] = \sum_{i=1}^{\infty} i \cdot p^i(1-p) = \frac{p}{1-p}$.
- For $P = 0.5$, $E[L] = \frac{0.5}{0.5} = 1$.

The experimental data ($1.03 \rightarrow 1.00$) aligns perfectly with the theoretical expectation. This stability ensures that the Skip List remains balanced, guaranteeing the $O(\log N)$ search path, rather than degenerating into a linked list (which would happen if Avg Level ≈ 0).

3.3 Conclusion on Testing

The testing results comprehensively validate the Skip List implementation. Time complexity follows the expected $O(\log N)$ behavior (modulated by hardware caching effects), space complexity is strictly $O(N)$, and the structural properties (Average Level) conform to the probabilistic design with $P = 0.5$.

4 Analysis and Comments

4.1 Time Complexity Analysis

(Analyze theoretical time complexity for Insert/Search/Delete - usually $O(\log n)$. Discuss if your test results match this theory.)

4.2 Space Complexity Analysis

(Analyze how much memory your Skip List uses. Discuss the trade-off between levels/-pointers and speed.)

4.3 Discussion

(Discuss any issues encountered, potential optimizations, or comparisons with other data structures.)

A Source Code

(You can paste your full code here or strictly key parts if the code is too long. Ensure it is commented.)

```cpp
// Your C++ code goes here
#include <iostream>
// ...
```

Listing 2: Main Skip List Implementation