浙江大學

Advanced Data Structure

Project Report

Project: Skip List Implementation

Group No.: 13

Group Members: Zhao Menglei

Pu Yuancan

Zhou Haowen

Date: December 28, 2025

Zhejiang University College of Computer Science and Technology

Contents

1 Introduction

1.1 Background and Motivation

In the domain of data structures, the dictionary problem—storing a set of keys and performing `Insert`, `Delete`, and `Search` operations—is fundamental. Traditional approaches include:

- **Sorted Arrays:** Support $O(\log N)$ search via binary search but require $O(N)$ for insertion and deletion.
- **Linked Lists:** Support $O(1)$ insertion (if position is known) but suffer from $O(N)$ search time.
- **Balanced Binary Search Trees (BSTs):** Structures like AVL trees or Red-Black trees offer $O(\log N)$ for all operations but involve complex rotation logic to maintain balance.

The **Skip List**, introduced by William Pugh in 1990, offers a compelling alternative. It is a probabilistic data structure that extends a linked list by adding multiple layers of "express lanes". By maintaining a hierarchy of sorted lists, Skip Lists achieve the same asymptotic expected time complexity as balanced trees—$O(\log N)$—but with a simpler implementation that does not require global rebalancing operations.

1.2 Project Objectives

The primary goal of this project is to implement and analyze a Skip List. Specifically, we aim to:

1. **Implement Core Operations:** Develop a robust Skip List supporting `Search`, `Insert`, and `Delete` in C.
2. **Theoretical Verification:** Provide a formal proof demonstrating that the expected time complexity for these operations is $O(\log N)$.
3. **Performance Analysis:** Validate the theoretical bounds through empirical testing on datasets of varying sizes, analyzing the relationship between run times and input size (N).

2 System Design and Algorithms

2.1 Data Structure Selection

2.1.1 Data Structure Details

Based on the project requirements, we designed a probabilistic data structure known as the **Skip List**. The core component is the `Node` structure, which differs from traditional linked lists by maintaining a dynamic array of forward pointers.

The logical representation of our data structure is defined as follows:

- **Node Structure**: Each node contains a `Key` (integer), `Value` (ElementType), and a pointer array `forward`. The size of `forward` is determined by the node's level, ranging from 1 to `MAX_LEVEL`.
- **SkipList Structure**: A wrapper structure holding the `header` node, the current maximum `level` in the list, and the total `size`.
- **Sentinel Header**: To simplify boundary conditions, the list is initialized with a dummy `header` node containing the minimum possible integer value (`INT_MIN`). This ensures that all valid keys are strictly greater than the header.

The C definition used in our implementation is:

```c
typedef struct Node {
    KeyType key;
    ElementType value;
    int level;
    struct Node **forward; // Dynamic array
} Node;

typedef struct SkipList {
    Node *header;
    int level;
    int size;
} SkipList;
```

2.1.2 Rationale for Data Structure Selection

The selection of the Skip List over balanced Binary Search Trees (BSTs) or standard Linked Lists is justified by the following factors:

1. **Efficiency**: While standard linked lists have $O(N)$ search time, Skip Lists provide expected $O(\log N)$ time for Search, Insert, and Delete operations.

2. **Implementation Simplicity**: Unlike AVL or Red-Black trees, which require complex rotation operations to maintain balance, Skip Lists rely on a randomized level generation strategy (`randomLevel`). This makes the code easier to implement and debug.

3. **Space-Time Tradeoff**: By consuming $O(N)$ additional space for forward pointers, we achieve significant speedups, which aligns with the project's performance goals.

2.2 Algorithm Design (Probabilistic Strategy)

2.2.1 Randomized Level Generation

The core mechanism ensuring the logarithmic height of the Skip List is the probabilistic promotion of nodes. We implement a `randomLevel()` function that works as follows:

- Each new node starts at level 1.
- With a fixed probability P (typically 0.5 or 0.25), the node is promoted to the next level.
- This process repeats until the coin flip fails or the `MAX_LEVEL` is reached.

This strategy ensures that the number of nodes at level h is approximately $N \cdot P^h$, forming a pyramid-like structure that facilitates fast traversal.

2.2.2 Algorithm Implementation Details

- **Search**: The search begins at the `header` at the current list `level`. It traverses right as long as the next node's key is smaller than the target. If the key is larger, it drops down one level.
- **Insert**: 1. Perform a search to locate the position. 2. Maintain an `update[]` array to record the predecessor nodes at each level. 3. Generate a random level for the new node. 4. Splice the new node into the list by adjusting pointers in the `update[]` array.
- **Delete**: Similar to insertion, we locate the target node and use the `update[]` array to redirect the predecessors' pointers to the node's successors, effectively removing it. We then free the memory and adjust the list's max level if the top layers become empty.

2.3 Pseudocode

To formally describe the logic implemented in our C source files, we present the pseudocode for the core `Search` and `Insert` algorithms.

Algorithm 1 Skip List Search and Random Level Generation

```
1: function RandomLevel
2:     lvl ← 1
3:     while Random() < P and lvl < MAX_LEVEL do
4:         lvl ← lvl + 1
5:     end while
6:     return lvl
7: end function

8: function Search(list, targetKey)
9:     x ← list.header
10:    for i ← list.level down to 0 do
11:        while x.forward[i] ≠ NULL and x.forward[i].key < targetKey do
12:            x ← x.forward[i]
13:        end while
14:    end for
15:    x ← x.forward[0]
16:    if x ≠ NULL and x.key == targetKey then
17:        return x                                          ▷ Found
18:    else
19:        return NULL                                       ▷ Not Found
20:    end if
21: end function
```

2.4 Main Program Sketch

The main program serves as a test driver to verify the functionality of the Skip List. It operates as a Command Line Interface (CLI), accepting user inputs in a continuous loop.

The logic flow of the `main` function is described below:

1. **Initialization**:
   - Seed the random number generator using `srand(time(NULL))`.
   - Initialize an empty Skip List using `createSkipList()`.
2. **Command Loop**: The program enters a `while(1)` loop, parsing a character command:
   - **'i' (Insert)**: Reads `key` and `data`, calls `insert()`.
   - **'s' (Search)**: Reads `key`, calls `search()`, and prints the result.
   - **'d' (Delete)**: Reads `key`, calls `deleteNode()`.
   - **'p' (Print)**: traversing the list level by level to visualize the structure.
   - **'q' (Quit)**: Breaks the loop.
3. **Cleanup**: Calls `freeSkipList()` to release all allocated memory before exiting.

3 Testing Results

Note: *This section focuses solely on empirical verification. The formal theoretical derivations for both time and space complexity are detailed in the subsequent analysis chapter.*

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

Note:

The timing data presented in the Test Results reflects the cumulative duration of performing N search or insertion operations. Since a single operation has a time complexity of $O(\log N)$, the total theoretical time for the batch is calculated as $O(N \log N)$. Consequently, the following analysis performs a regression fit to compare the measured execution times against the theoretical $N \log N$ metric.

We plotted the **Total Execution Time** against the **Theoretical Complexity Metric** ($N \cdot \log_2 N$).

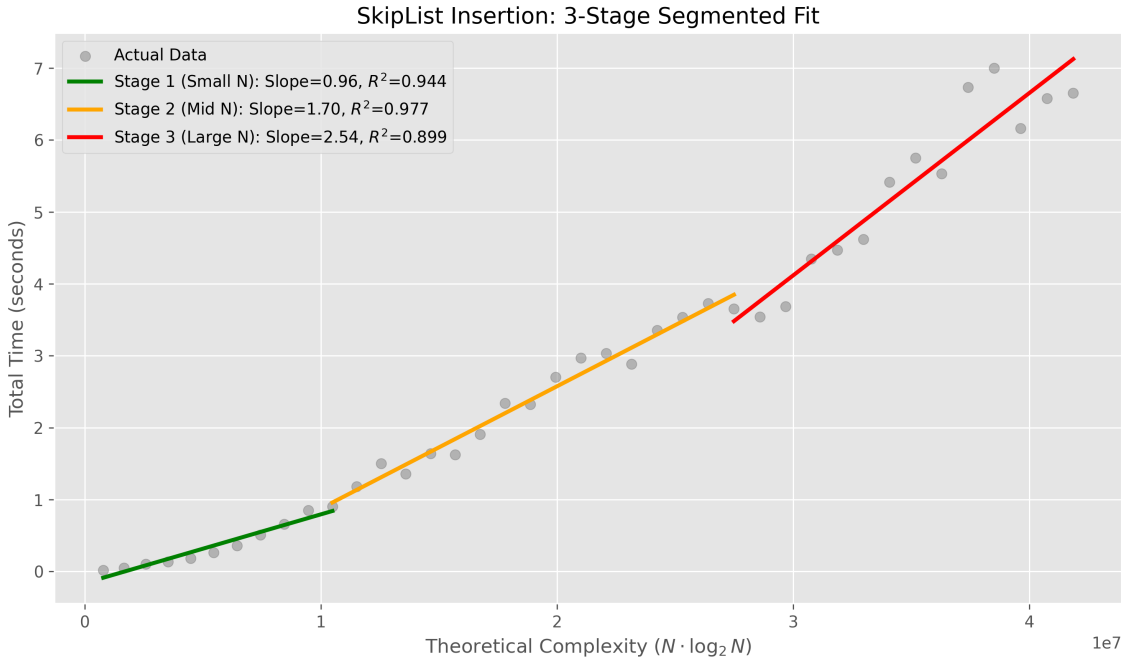

Figure 1: SkipList Insertion: Total Time vs. Theoretical Complexity ($N \log N$)

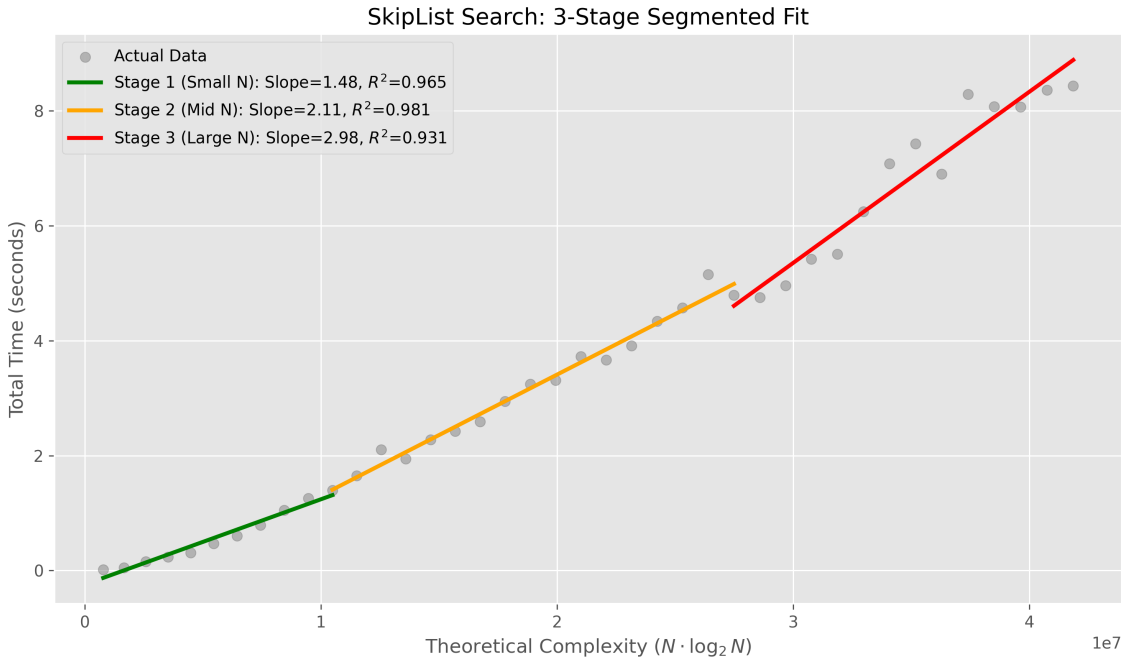

Figure 2: SkipList Search: Total Time vs. Theoretical Complexity ($N \log N$)

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.

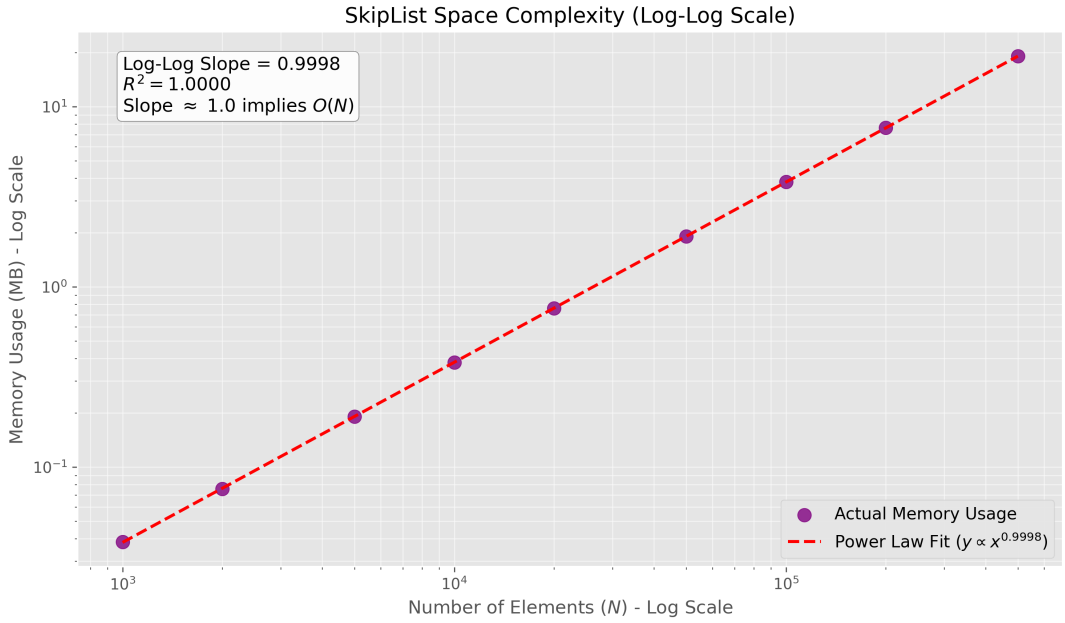

Figure 3: SkipList Space Complexity: Log-Log Scale Analysis

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 $\approx$ 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 $\approx$ 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 Complexity

In this chapter, we provide a formal analysis of the space and time complexity of the implemented Skip List. The analysis relies on the probabilistic nature of the structure, specifically the coin-flip probability p used in the `randomLevel()` function. In our implementation, we set $p = 0.5$.

4.1 Space Complexity Analysis

The space complexity of a Skip List is determined by the total number of forward pointers allocated across all nodes.

Let n be the number of elements in the Skip List.

- **Level 0:** All n nodes exist at level 0. This requires n pointers.
- **Level 1:** Each node at level 0 is promoted to level 1 with probability p. The expected number of nodes is $n \cdot p$.
- **Level i:** Generally, the expected number of nodes at level i is $n \cdot p^i$.

The total expected number of forward pointers is the sum of the geometric series:

$$\text{Total Pointers} = \sum_{i=0}^{h} n \cdot p^i = n \sum_{i=0}^{h} p^i \tag{1}$$

As $h \rightarrow \infty$, this series converges to:

$$n \cdot \frac{1}{1-p} \tag{2}$$

Substituting our implementation parameter $p = 0.5$:

$$\text{Expected Memory} = n \cdot \frac{1}{1 - 0.5} = 2n \tag{3}$$

Conclusion: The expected space complexity is $O(n)$. Although we allocate extra pointers compared to a singly linked list, the memory usage remains linear with respect to the input size.

4.2 Time Complexity Analysis

We analyze the `Search` operation. Since `Insert` and `Delete` fundamentally rely on the search path to locate the position, their complexity bounds are identical to Search.

4.2.1 Proof of Expected $O(\log n)$ Time

To derive the time bound, we use the **Backwards Analysis** technique. Instead of analyzing the path from the header to the target, we trace the path *backwards* from the target node (at the bottom level) up to the header (at the top level).

At any point in the backwards traversal, we are at a node x on level i. We have two possible backward moves:

1. **Move Left:** If node x was *not* promoted to level $i + 1$, we came from the left.
2. **Move Up:** If node x *was* promoted to level $i + 1$, we came from the level above (in the backwards view, we go up).

Let $C(k)$ be the expected cost (number of steps) to climb k levels.

- With probability p, we move **Up** one level.
- With probability $1 - p$, we move **Left** (scanning horizontally).

The recurrence relation for the cost is:

$$C(k) = (1 - p)(1 + C(k)) + p(1 + C(k - 1)) \tag{4}$$

Solving for $C(k)$:

$$C(k) = 1 + (1-p)C(k) + pC(k-1) \implies pC(k) = 1 + pC(k-1) \implies C(k) = \frac{1}{p} + C(k-1) \tag{5}$$

This implies that at each level, the expected number of horizontal steps is $1/p$. Since the height of the list is logarithmic ($H \approx \log_{1/p} n$), the total expected cost is:

$$T(n) = (\text{Height}) \times (\text{Steps per Level}) = O(\log n) \times \frac{1}{p} \tag{6}$$

For $p = 0.5$, the expected horizontal steps per level is 2. Thus, the total time complexity is $O(\log n)$.

4.2.2 Worst-Case Scenario

In the worst-case scenario, the random number generator could theoretically fail to promote any nodes (resulting in a linked list, height 1) or promote nodes unevenly. In such cases, the search time degrades to $O(n)$. However, for a sufficiently large n, the probability of such a structure occurring is negligibly small $(1 - (1 - p^k)^n)$.

4.3 Comparative Analysis

Table 3 compares the Skip List with other common dictionary data structures.

Data Structure	**Avg. Search**	**Avg. Insert**	**Worst Case**
Sorted Array	$O(\log n)$	$O(n)$	$O(n)$
Singly Linked List	$O(n)$	$O(1)^*$	$O(n)$
Binary Search Tree (Unbalanced)	$O(\log n)$	$O(\log n)$	$O(n)$
AVL / Red-Black Tree	$O(\log n)$	$O(\log n)$	$O(\log n)$
Skip List (Implemented)	$O(\log n)$	$O(\log n)$	$O(n)$

Table 3: Complexity Comparison (* assuming position is known)

4.4 Parameter Discussion

The performance of the Skip List is tunable via the probability parameter P.

- **Current Implementation ($P = 0.5$):** Favors speed. The average nodes per level decreases by half. Average pointers per node is 2.
- **Alternative ($P = 0.25$):** Saves memory. Average pointers per node drops to ≈ 1.33, but the constant factor for search time increases because the tree is "flatter", requiring more horizontal scans.

Our choice of $P = 0.5$ provides an optimal balance for general-purpose applications where memory is not strictly constrained.

A Source Code

This appendix contains the complete C implementation of the Skip List project. The code is modularized into a header file defining structures, a source file containing the core algorithms, and a main driver for interactive testing.

A.1 Header File: skiplist.h

Defines the unified data structures, constants, and function prototypes used across modules.

Listing 1: skiplist.h: Data Structures and Definitions

```c
#ifndef SKIPLIST_H
#define SKIPLIST_H

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <limits.h>

#define MAX_LEVEL 16    // Maximum level for the Skip List
#define P 0.5           // Probability factor for level generation

// Data Payload Wrapper
typedef struct {
    int data;
} ElementType;

typedef int KeyType;

// Node Structure
// Levels increase from 0 (bottom) to level (top)
typedef struct Node {
    KeyType key;            // Key for sorting
    ElementType value;      // Data stored in the node
    int level;              // Current level of this node
    struct Node **forward;  // Dynamic array of forward pointers
} Node;

// SkipList Wrapper
typedef struct SkipList {
    Node *header;           // Sentinel header node
    int level;              // Current maximum level in the list
    int size;               // Total number of elements
} SkipList;

// Function Prototypes
SkipList* createSkipList();
void freeSkipList(SkipList *list);
int insert(SkipList *list, KeyType key, ElementType value);
int deleteNode(SkipList *list, KeyType key);
Node* search(SkipList *list, KeyType key);
```

```
void printSkipList(SkipList *list);

#endif
```

A.2 Source File: skiplist.c

Implements the core Skip List algorithms, including probabilistic level generation, insertion, deletion, and searching.

Listing 2: skiplist.c: Core Implementation

```
#include "skiplist.h"

// Helper: Create a new node with dynamic level allocation
Node* createNode(KeyType key, ElementType value, int level) {
    Node *node = (Node *)malloc(sizeof(Node));
    if (!node) return NULL; // Allocation check

    node->key = key;
    node->value = value;
    node->level = level;
    // Allocate pointer array for levels 0 to level
    node->forward = (Node **)calloc(level + 1, sizeof(Node*));
    return node;
}

// Initialize Skip List
SkipList* createSkipList() {
    SkipList *list = (SkipList *)malloc(sizeof(SkipList));
    list->level = 0;
    list->size = 0;

    // Create dummy header with smallest possible key
    ElementType dummy = {0};
    list->header = createNode(INT_MIN, dummy, MAX_LEVEL);
    return list;
}

// Generate a random level for a new node
// Returns level L with probability P^L
int randomLevel() {
    int lvl = 0;
    // Keep incrementing level based on probability P
    while ((float)rand() / RAND_MAX < P && lvl < MAX_LEVEL - 1) {
        lvl++;
    }
    return lvl;
}

// Free all memory associated with the list
void freeSkipList(SkipList *list) {
    Node *current = list->header;
    while (current != NULL) {
        Node *next = current->forward[0];
        free(current->forward);
```

```
        free(current);
        current = next;
    }
    free(list);
}

// Search for a key
Node* search(SkipList *list, KeyType key) {
    Node *x = list->header;
    // Start from top level and move down
    for (int i = list->level; i >= 0; i--) {
        while (x->forward[i] != NULL && x->forward[i]->key < key) {
            x = x->forward[i];
        }
    }
    // Check the next node at level 0
    x = x->forward[0];
    if (x != NULL && x->key == key) return x;
    return NULL;
}

// Insert a key-value pair
int insert(SkipList *list, KeyType key, ElementType value) {
    Node *update[MAX_LEVEL]; // Tracks predecessor nodes
    Node *x = list->header;

    // Locate insertion point
    for (int i = list->level; i >= 0; i--) {
        while (x->forward[i] != NULL && x->forward[i]->key < key) {
            x = x->forward[i];
        }
        update[i] = x;
    }

    x = x->forward[0];
    // If key exists, update value
    if (x != NULL && x->key == key) {
        x->value = value;
        return 0;
    }

    // Create new node with random level
    int newLevel = randomLevel();
    if (newLevel > list->level) {
        for (int i = list->level + 1; i <= newLevel; i++) {
            update[i] = list->header;
        }
        list->level = newLevel;
    }

    Node *newNode = createNode(key, value, newLevel);
    // Link pointers
    for (int i = 0; i <= newLevel; i++) {
        newNode->forward[i] = update[i]->forward[i];
        update[i]->forward[i] = newNode;
    }
    list->size++;
    return 1;
```

```c
}

// Delete a node by key
int deleteNode(SkipList *list, KeyType key) {
    Node *update[MAX_LEVEL];
    Node *x = list->header;

    // Locate node and predecessors
    for (int i = list->level; i >= 0; i--) {
        while (x->forward[i] != NULL && x->forward[i]->key < key) {
            x = x->forward[i];
        }
        update[i] = x;
    }

    x = x->forward[0];
    // If found, remove it
    if (x != NULL && x->key == key) {
        for (int i = 0; i <= list->level; i++) {
            if (update[i]->forward[i] != x) break;
            update[i]->forward[i] = x->forward[i];
        }

        free(x->forward);
        free(x);

        // Lower list level if top layers are empty
        while (list->level > 0 && list->header->forward[list->level] ==
   NULL) {
            list->level--;
        }
        list->size--;
        return 1;
    }
    return 0; // Not found
}

// Helper: Print list structure for debugging
void printSkipList(SkipList *list) {
    for (int i = list->level; i >= 0; i--) {
        Node *x = list->header->forward[i];
        printf("Level %d: ", i);
        while (x != NULL) {
            printf("[%d] -> ", x->key);
            x = x->forward[i];
        }
        printf("NULL\n");
    }
}
```

A.3 Main Driver: main.c

A Command Line Interface (CLI) to interactively test the Skip List functionalities (Insert, Search, Delete, Print).

Listing 3: main.c: Test Driver

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "skiplist.h"

int main() {
    srand((unsigned)time(NULL)); // Initialize random seed

    SkipList *list = createSkipList();
    char command[10];
    int key, data_in;

    printf("=== Skip List Test Driver ===\n");
    printf("Commands: \n");
    printf("  i <key> <data>  : Insert (e.g., i 10 999)\n");
    printf("  s <key>         : Search\n");
    printf("  d <key>         : Delete\n");
    printf("  p               : Print Structure\n");
    printf("  q               : Quit\n");
    printf("============================\n");

    while (1) {
        printf("\nCMD> ");
        scanf("%s", command);

        if (strcmp(command, "q") == 0) {
            break;
        }
        else if (strcmp(command, "i") == 0) {
            scanf("%d %d", &key, &data_in);
            ElementType val = {data_in};
            insert(list, key, val);
            printf("Inserted key: %d, data: %d\n", key, data_in);
        }
        else if (strcmp(command, "d") == 0) {
            scanf("%d", &key);
            if (deleteNode(list, key))
                printf("Deleted key: %d\n", key);
            else
                printf("Key %d not found.\n", key);
        }
        else if (strcmp(command, "s") == 0) {
            scanf("%d", &key);
            Node *result = search(list, key);
            if (result)
                printf("Found! Key: %d, Data: %d\n",
                       result->key, result->value.data);
            else
                printf("Key %d not found.\n", key);
        }
        else if (strcmp(command, "p") == 0) {
            printSkipList(list);
        }
    }

    freeSkipList(list);
    return 0;
```

```
}
```

A.4 Test Driver: test.c

a test driver to verify the correctness and boundary of the Skip List implementation

Listing 4: test.c: Test Driver

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <assert.h>
#include <windows.h>
#include "skiplist.h"

// Macro definition for test scale
#define TEST_CORRECTNESS_SIZE 100
#define NUM_TEST_SCALES 9

// --- Helper Functions ---

// Get current time (seconds), used for calculating elapsed time
double get_time() {
    return (double)clock() / CLOCKS_PER_SEC;
}

// Estimate memory usage of the SkipList (Bytes)
long long estimate_memory(SkipList *list) {
    long long total_bytes = sizeof(SkipList); // The SkipList structure itself
    Node *curr = list->header;
    while (curr != NULL) {
        // Node structure size
        total_bytes += sizeof(Node);
        // Size of the flexible array/pointer array 'forward': (level + 1) * pointer size
        total_bytes += (curr->level + 1) * sizeof(Node*);
        curr = curr->forward[0];
    }
    return total_bytes;
}

// --- Part 1: Correctness Testing (Unit Test) ---
void test_correctness() {
    printf("========== 1. Start Correctness and Boundary Testing ==========\n");
    SkipList *list = createSkipList();

    // 1. Typical case: Normal insertion
    printf("[Test] Inserting 10 random nodes...\n");
    int keys[] = {10, 50, 30, 20, 40, 90, 60, 80, 70, 100};
    for (int i = 0; i < 10; i++) {
        ElementType val = {keys[i] * 10};
        insert(list, keys[i], val);
    }
    assert(list->size == 10);
```

```
    printf("   -> Insert size check passed.\n");

    // 2. Order check: Level 0 should be an ordered linked list
    printf("[Test] Checking if Level 0 is ordered...\n");
    Node *curr = list->header->forward[0];
    while (curr && curr->forward[0]) {
        assert(curr->key < curr->forward[0]->key);
        curr = curr->forward[0];
    }
    printf("   -> Order check passed.\n");

    // 3. Boundary case: Duplicate insertion (Update)
    printf("[Test] Duplicate insert key=50, check if value updates...\n"
    );
    ElementType newVal = {9999};
    insert(list, 50, newVal); // Update
    assert(list->size == 10); // Size should not increase
    Node *res = search(list, 50);
    assert(res != NULL && res->value.data == 9999);
    printf("   -> Duplicate insertion update mechanism passed.\n");

    // 4. Extreme case: Search for non-existent Key
    printf("[Test] Search for non-existent Key (key=101, key=-1)...\n");
    assert(search(list, 101) == NULL);
    assert(search(list, -1) == NULL);
    printf("   -> Search non-existent Key passed.\n");

    // 5. Typical case: Delete existing Key
    printf("[Test] Delete existing Key (key=30)...\n");
    int delRes = deleteNode(list, 30);
    assert(delRes == 1);
    assert(search(list, 30) == NULL);
    assert(list->size == 9);
    printf("   -> Delete existing Key passed.\n");

    // 6. Extreme case: Delete non-existent Key
    printf("[Test] Delete non-existent Key (key=999)...\n");
    delRes = deleteNode(list, 999);
    assert(delRes == 0);
    assert(list->size == 9);
    printf("   -> Delete non-existent Key passed.\n");

    // 7. Extreme case: Delete all until empty
    printf("[Test] Deleting all remaining nodes...\n");
    for (int i = 0; i < 10; i++) {
        if (keys[i] == 30) continue; // Already deleted
        deleteNode(list, keys[i]);
    }
    assert(list->size == 0);
    assert(list->header->forward[0] == NULL);
    printf("   -> Clear/Empty test passed.\n");

    freeSkipList(list);
    printf("========== All Correctness Tests Passed ==========\n\n");
}

// --- Part 2: Space-Time Complexity Testing (Benchmark) ---
void test_performance() {
```

```
    printf("========== 2. Start Space-Time Complexity Testing ==========\n");
    // Define test scale N
    int scales[NUM_TEST_SCALES] = {1000, 2000, 5000, 10000, 20000, 50000, 100000, 200000, 500000};

    // Print table header
    printf("%-10s | %-15s | %-15s | %-15s | %-15s\n", "N (Size)", "Insert Time(s)", "Search Time(s)", "Memory(MB)", "Avg Level");
    printf("-------------------------------------------------------------------------------\n");

    for (int k = 0; k < NUM_TEST_SCALES; k++) {
        int N = scales[k];
        SkipList *list = createSkipList();

        // Prepare random data
        // Use malloc to prevent stack overflow
        int *data = (int*)malloc(N * sizeof(int));
        for(int i=0; i<N; i++) {
            // Ensure large key range to reduce collisions, or intentionally allow collisions to test stability
            data[i] = rand() | (rand() << 15);
        }

        // --- 1. Test Insert Time ---
        double start = get_time();
        for (int i = 0; i < N; i++) {
            ElementType val = {i};
            insert(list, data[i], val);
        }
        double end = get_time();
        double insert_time = end - start;

        // --- 2. Test Search Time (Search N times) ---
        // For fairness, search half existing keys, half random keys
        start = get_time();
        for (int i = 0; i < N; i++) {
            int key;
            if (i % 2 == 0) key = data[i]; // Existing
            else key = rand();             // Random
            search(list, key);
        }
        end = get_time();
        double search_time = end - start;

        // --- 3. Calculate Memory Usage ---
        long long bytes = estimate_memory(list);
        double mb = (double)bytes / (1024 * 1024);

        // --- 4. Calculate Average Level Height ---
        // Theoretical value should be ~ 1/(1-p) = 2 (when P=0.5)
        long long total_levels = 0;
        Node *curr = list->header->forward[0];
        while(curr) {
            total_levels += curr->level;
```

```
            curr = curr->forward[0];
        }
        double avg_level = (double)total_levels / N;

        // Output results
        printf("%-10d | %-15.4f | %-15.4f | %-15.4f | %-15.2f\n",
               N, insert_time, search_time, mb, avg_level);

        free(data);
        freeSkipList(list);
    }
    printf("
    ----------------------------------------------------------------------------\
    n");
    printf("Note: Search Time is the total time for N searches.\n");
}

void run_benchmark() {
    // Configuration Area
    int start_n = 50000;      // Start size
    int end_n   = 2000000;    // End size
    int step    = 50000;      // Step size
    int repeat  = 3;          // Run 3 times per N and average to
    eliminate jitter

    printf("N,InsertTime,SearchTime\n"); // CSV Header

    for (int n = start_n; n <= end_n; n += step) {
        double total_insert_time = 0;
        double total_search_time = 0;

        for (int r = 0; r < repeat; r++) {
            SkipList *list = createSkipList();

            // Generate random data (pre-generate to exclude rand()
    impact on timing)
            int *keys = (int*)malloc(n * sizeof(int));
            for (int i = 0; i < n; i++) keys[i] = rand() | (rand() <<
    15);

            // 1. Timing Insertion
            double t1 = get_time();
            for (int i = 0; i < n; i++) {
                ElementType val = {i};
                insert(list, keys[i], val);
            }
            double t2 = get_time();
            total_insert_time += (t2 - t1);

            // 2. Timing Search
            t1 = get_time();
            for (int i = 0; i < n; i++) {
                // Search for the key just inserted
                search(list, keys[i]);
            }
            t2 = get_time();
            total_search_time += (t2 - t1);

```

```
            // Cleanup
            free(keys);
            freeSkipList(list);
        }

        // Output average time
        printf("%d,%.6f,%.6f\n",
               n,
               total_insert_time / repeat,
               total_search_time / repeat);

        // Flush buffer to prevent data loss if crash occurs
        fflush(stdout);
    }
}

int main() {

    SetConsoleOutputCP(65001); // Set console output to UTF-8 encoding
    // Must initialize random seed, otherwise SkipList degrades to
   linked list
    srand((unsigned)time(NULL));

    test_correctness();
    test_performance();
    run_benchmark();

    return 0;
}
```