Self adjusting Skip Lists: Adaptive Structures for Efficient Element Retrieval

Abstract

This scientific report presents an investigation into self-adjusting skip lists and their variations, with a focus on optimizing element retrieval efficiency when the same element is queried multiple times. Skip lists are probabilistic data structures known for their logarithmic time complexity in search, insertion, and deletion operations. In this study, three variations of self-adjusting skip lists were implemented and compared with the basic skip list. These variations included promoting the queried element to the next higher level, promoting it to a random higher level, or promoting it directly to the top level.

The performance of these self-adjusting skip lists was evaluated using randomly generated numbers and exponentially distributed numbers. Metrics such as side and down movement were used to assess retrieval efficiency. The experimental results demonstrated the effectiveness of the self-adjusting skip lists in improving retrieval performance compared to the basic skip list. Variation (1), promoting the element to the next higher level, consistently reduced the average number of movements required for subsequent queries. Variation (2), involving random promotion, yielded mixed outcomes depending on query patterns and data distribution. Variation (3), promoting the element to the top level, proved to be the most effective strategy, significantly reducing traversal steps.

Through rigorous experimentation and analysis, this study provides insights into the trade-offs and benefits of each self-adjusting variation. The impact of data distribution and query patterns on the performance of self-adjusting skip lists is also discussed. The findings highlight the potential of these variations as efficient and scalable data structures for applications involving repeated element retrievals. By elucidating the mechanisms underlying these variations, this research contributes to informed decision-making in selecting the most suitable self-adjusting strategy based on specific application requirements.

1 Introduction

Skip lists are widely recognized probabilistic data structures known for their efficient search, insertion, and deletion operations, all achieved with logarithmic time complexity. These characteristics have made skip lists valuable in various domains. However, when the same element is queried multiple times, traditional skip lists may exhibit suboptimal retrieval efficiency, leading to unnecessary traversals.

To address this limitation, self-adjusting skip lists have been proposed as an extension to the basic skip list structure. Self-adjusting skip lists dynamically adapt their structure in response to query patterns, aiming to optimize element retrieval performance in scenarios involving repetitive queries. In this study, we explore and compare three variations of self-adjusting skip lists, each employing a different strategy for promoting elements that are queried again.

The objective of this research is to evaluate the impact of these self-adjusting variations on retrieval efficiency compared to the traditional skip list structure. We consider two different data distributions: randomly generated numbers and exponentially distributed numbers. By measuring metrics such as side and down movement, we aim to assess the performance of the self-adjusting skip lists in terms of the number of traversals required for subsequent queries. Through rigorous experimentation and analysis, we aim to provide insights into the benefits and trade-offs associated with each self-adjusting variation. Additionally, we investigate the influence of query patterns and data distribution on the performance of these variations. The findings of this study will contribute to a better understanding of self-adjusting

skip lists and their potential as efficient and scalable data structures for applications that involve repeated element retrievals. By elucidating the mechanisms underlying these variations, we enable informed decision-making in selecting the most suitable self-adjusting strategy based on specific application requirements.

2 Preliminaries

The problem addressed in this scientific report is the efficient retrieval of elements in skip lists, particularly in scenarios where the same element is queried multiple times. Although skip lists offer logarithmic time complexity for search operations, they do not inherently adapt to repeated queries, potentially resulting in unnecessary traversal steps and sub optimal performance. Therefore, the problem at hand is to enhance skip lists with self-adjusting mechanisms that promote frequently queried elements to higher levels, aiming to minimize the number of movements required for subsequent queries. This research aims to explore and evaluate different variations of self-adjusting skip lists to identify the most effective strategy for optimizing element retrieval in multiset environments.

3 Algorithm & Implementation

This research project focuses on the implementation and analysis of a skiplist data structure along with three additional variations aimed at optimizing element retrieval efficiency. The variations investigated in this study include the Next Higher Level Promotion, Random Higher Level Promotion, and Top Level Promotion strategies within the skiplist framework.

3.1 Traditional Skip-list

A skip list is a probabilistic data structure that provides an efficient alternative to balanced search trees, such as binary search trees or AVL trees. It offers logarithmic time complexity for search, insertion, and deletion operations, making it a valuable data structure for a variety of applications.

At its core, a skip list is a data structure composed of multiple linked lists, referred to as levels. Every level is a standard sorted linked list where each element contains a key-value pair, with keys used for ordering. In addition to the forward pointers that enable skipping multiple elements, each element in a level also has a backward pointer, making it a doubly linked list. By utilizing both forward and backward pointers, skip-lists optimize element retrieval by minimizing the number of comparisons needed for traversal.

The runtime complexity of skip lists is O(log n) on average for search, insertion, and deletion operations. This performance is guaranteed in expectation and with high probability. In rare cases, the worst-case time complexity can be O(n). This efficiency arises from the skip list's ability to skip multiple elements during traversal, resulting in a reduced number of comparisons required for these operations.

Regarding space complexity, skip lists require additional memory compared to ordinary linked lists due to the maintenance of multiple levels. However, the space overhead is generally acceptable, particularly for large datasets. The space complexity of a skip list is O(n), where n represents the number of elements in the skip list. Importantly, the height of a skip list remains below O(log n) with high probability.

3.2 Adjustment for Controlling Skiplist Height

The adjustment made in the skiplist implementation aimed to address the growth of the skiplist's height as the number of elements increased. Specifically, it involved reducing the probability of further promotion to 1/logn (where logn represents the logarithm of the number of elements in the skiplist) after a specific height (e.g., 19). It happens after the size of the skiplist exceeds 1 million.

It was intended to find a balance between the skiplist's time complexity and space complexity.

Regarding time complexity:

The reduction in promotion probability beyond a certain height had the potential to restrict the creation

of additional levels in the skiplist. With an increasing skiplist height, search, insertion, and deletion operations could experience higher time complexity due to the need for traversing more levels. The aim of controlling the skiplist's height was to manage the time complexity of these operations. By reducing the promotion probability after a certain height, excessive levels could be prevented, potentially improving the average time complexity of operations.

Regarding space complexity:

Skiplists required more memory compared to ordinary linked lists due to the overhead associated with maintaining multiple levels. By reducing the probability of further promotion beyond a specific height, the goal was to limit the creation of additional levels. This adjustment aimed to manage the skiplist's space usage, particularly for large datasets, by keeping the number of levels and additional memory requirements within acceptable limits.

The adjustment in the skiplist implementation sought to find equilibrium between the time complexity and space complexity of the data structure. By controlling the skiplist's height through a reduction in promotion probability beyond a certain point, the intention was to mitigate potential increases in time complexity while ensuring reasonable space utilization.

Algorithm 1 locate Key (Search)

```
0: procedure LOCATE KEY(key, exact_match, first_occur)
0:     no_of_ops ← 0
0:     pointer ← self.top_left_element
0:     while pointer.below ≠ None do
0:         pointer ← pointer.below
0:         no_of_ops ← no_of_ops + 1
0:     end while
0:     while pointer.after.key ≤ key do
0:         pointer ← pointer.after
0:         no_of_ops ← no_of_ops + 1
0:         if first_occur and pointer.key = key then
0:             return pointer, no_of_ops
0:         end if
0:     end while
0:     if exact_match and pointer.key ≠ key then
0:         return None, no_of_ops
0:     else
0:         return pointer, no_of_ops
0:     end if
0: end procedure=0
```

Algorithm 2 Insertion

```
0: procedure INSERT(key, value)
0:   (found_element, no_of_ops) ← self.locate_key(key)
0:   pointer ← found_element
0:   if found_element.key = key then
0:     old_value ← found_element.value
0:     found_element.value ← value
0:     return old_value
0:   end if
0:   self.elements_count ← self.elements_count + 1
0:   level ← 0
0:   element ←self.insert_after_above (pointer,None, key, value, level)
0:   prob ← 0.5
0:   count ← 1
0:   while random.random() > prob do
0:     count ← count + 1
0:     level ← level + 1
0:     while pointer.above = None do
0:       pointer ← pointer.before
0:     end while
0:     pointer ← pointer.above
0:     element ←self.insert_after_above(pointer, element, key, value, level)
0:     if count ≥ self.levels_count then
0:       self.insert_top_level()
0:       if self.levels_count > 15 then
0:         logn ← log2(self.elements_count)
0:         prob ← 1 − (1/logn)
0:       end if
0:     end if
0:   end while
0:   return no_of_ops
0: end procedure=0
```

Algorithm 3 Deletion

```
0: procedure REMOVEELEMENT(key)
0:   pointer, key, value, depth_of_node ← self.find_first_occurrence(key)
0:   if pointer.key ≠ key then
0:     raise Exception('NOT_FOUND')
0:   end if
0:   while pointer ≠ None do
0:     depth_of_node ← depth_of_node − 1
0:     if pointer.before.key = −∞ and pointer.after.key = ∞ then
0:       pointer.before.above ← None
0:       pointer.after.above ← None
0:       self.levels_count ← self.levels_count − 1
0:       self.top_left_element ← pointer.before
0:     end if
0:     pointer.before.after ← pointer.after
0:     pointer.after.before ← pointer.before
0:     self.decrease_level_count(depth_of_node)
0:     pointer ← pointer.below
0:   end while
0:   self.elements_count ← self.elements_count − 1
0: end procedure=0
```

3.3 Skiplist: Promote to next higher level

The Promote-One-Level Skip List is an extension of the traditional skip list data structure. In this variation, when an element is searched for and found, it is promoted to one level higher than its current level.
When searching for an element, the algorithm follows the same procedure as the regular skip list. It starts from the top left element and traverses down the levels, searching for the desired key.If the element is found during the search, it is promoted to one level higher than its current level. This promotion involves inserting the element in the level above its current position. The element is inserted between the node before it and the corresponding node above it.
By promoting the found element, the Promote-One-Level Skip List aims to reduce future search times for that element. As the element moves up one level, subsequent searches for the same element will have a

higher chance of finding it sooner, potentially saving traversal time in lower levels.

The promotion process incurs additional overhead in terms of space complexity and insertion operations when an element is found. However, this trade-off is justified by the improved search efficiency, particularly for popular elements.

Algorithm 4 Promote One Level Skiplist

```
0: function SEARCH_AND_INSERT(skiplist, key, value)
0:   pointer, target_key, target_value, level, no_of_ops ← FIND_FIRST_OCCURRENCE(skiplist, key)
0:   if pointer = None then
0:     no_of_ops ← INSERT(skiplist, key, value)
0:     skiplist.num_of_comparison ← skiplist.num_of_comparison + no_of_ops
0:   else
0:     skiplist.num_of_comparison ← skiplist.num_of_comparison + no_of_ops
0:     key ← target_key
0:     value ← target_value
0:     if skiplist.levels_count − 1 > level then
0:       found_before ← pointer.before
0:       while found_before.above is None do
0:         found_before ← found_before.before
0:       end while
0:       found_before ← found_before.above
0:       pointer ← INSERT_AFTER_ABOVE(found_before, pointer, key, value, level)
0:       testPointer ← pointer
0:       if testPointer.key ≠ −∞ then
0:         while testPointer.key ≠ −∞ do
0:           testPointer ← testPointer.before
0:         end while
0:         if testPointer.below = None then
0:           print "stop"
0:         end if
0:       end if
0:       if level ∈ skiplist.level_element_count and level − 1 ∈ skiplist.level_element_count then
0:         if skiplist.level_element_count[level] = skiplist.level_element_count[level − 1] then
0:           REMOVE_LEVEL(skiplist, level, pointer)
0:         end if
0:       end if
0:     end if
0:   end if
0: end function=0
```

3.4 Skiplist: Promote to random higher level

The Promote-Random-Level Skiplist algorithm extends the traditional skiplist data structure by promoting elements to a random number of levels whenever they are searched. This extension aims to optimize the search efficiency, particularly for frequently accessed elements. In this subsection, we will provide an overview of the algorithm and discuss its potential runtime improvements with different streams of numbers.

When an element is searched or inserted, the search and insert method is called. The algorithm begins with a standard search operation, and if the element is not found, it is inserted using the standard insertion procedure. However, if the element is found, it undergoes a promotion process to higher levels. By flipping a coin and comparing the result to a threshold of 0.5, the algorithm determines whether the element should be promoted to a higher level. If the element reaches or surpasses the highest level, a new top level is added. This random promotion strategy improves search efficiency for frequently accessed elements and offers adaptability to varying element popularity, leading to potential runtime improvements in different number streams.

In scenarios where the popularity of elements varies greatly, the Promote-Random-Level Skiplist can adapt dynamically and optimize the structure for specific elements. Frequently accessed elements will be promoted to higher levels, enabling faster search operations. Less popular elements will remain at lower levels, preventing unnecessary promotion and maintaining efficient search times. This adaptability of the algorithm to different levels of element popularity can result in significant runtime improvements.

Algorithm 5 Promote to random higher level

```
0: function SEARCH_AND_INSERT(key, value)
0:   element, target_key, target_value, level, no_of_ops ← FIND_FIRST_OCCURRENCE(key)
0:   pointer ← element
0:   if pointer = None then
0:     no_of_ops ← INSERT(key, value)
0:     self.num_of_comparison ← self.num_of_comparison + no_of_ops
0:   else
0:     self.num_of_comparison ← self.num_of_comparison + no_of_ops
0:     key ← target_key
0:     value ← target_value
0:     while random.random() > 0.5 and self.levels_count − 1 > level do
0:       while pointer.above is None do
0:         pointer ← pointer.before
0:       end while
0:       pointer ← pointer.above
0:       element ← INSERT_AFTER_ABOVE(pointer, element, key, value, level)
0:       if level ∈ self.level_element_count and level − 1 ∈ self.level_element_count then
0:         if self.level_element_count[level] = self.level_element_count[level − 1] then
0:           REMOVE_LEVEL(level, pointer)
0:         else
0:           level ← level + 1
0:         end if
0:       end if
0:       if level ≥ self.levels_count then
0:         INSERT_TOP_LEVEL
0:       end if
0:     end while
0:   end if
0: end function=0
```

3.5 Skiplist: Promote to top level

The Promote-to-Top-Level Skiplist algorithm is an extension of the traditional skiplist data structure. In this variation, whenever an element is searched for, it is promoted to the top level of the skiplist. This algorithm aims to improve the search efficiency for

frequently accessed elements and reduce the average search time. In this subsection, we will provide an overview of the algorithm, discuss its runtime analysis, explore its potential improvements, and consider the trade-offs associated with this approach.

The algorithm begins with a standard search operation to locate the element within the skiplist. If the element is not found, it is inserted into the skiplist at the lowest level using the standard insertion procedure. However, if the element is found, it undergoes a promotion process to the top level. The element is moved to the highest level by iteratively inserting it after the corresponding nodes at each level. This promotion ensures that frequently accessed elements are placed at the top level, allowing for faster search operations in subsequent accesses.

The runtime analysis of the Promote-to-Top-Level Skiplist algorithm is influenced by both the structure of the skiplist and the distribution of element accesses. In the worst-case scenario where all elements are accessed uniformly, the average search time remains worse compared to a traditional skiplist. This is because every element is promoted to the top level, essentially transforming the skiplist into an array-like structure with a search time of O(n). However, the algorithm exhibits significant improvements in scenarios where certain elements are accessed more frequently than others. By promoting frequently accessed elements to the top level, the search time for these elements is reduced to O(1), resulting in an overall enhancement of search efficiency. It is important to note that the benefits of this algorithm come at the expense of increased space complexity and insertion overhead for promoted elements.

Algorithm 6 Promote-to-Top-Level Skiplist

```
0: function SEARCH_AND_INSERT(key, value)
0:   pointer, target_key, target_value, level, no_of_ops ← FIND_FIRST_OCCURRENCE(key)
0:   if pointer = None then
0:     no_of_ops ← INSERT(key, value)
0:     self.num_of_comparison ← self.num_of_comparison + no_of_ops
0:   else
0:     key ← target_key
0:     value ← target_value
0:     self.num_of_comparison ← self.num_of_comparison + no_of_ops
0:     while self.levels_count − 1 > level do
0:       found_before ← pointer.before
0:       while found_before.above is None do
0:         found_before ← found_before.before
0:       end while
0:       found_before ← found_before.above
0:       pointer ← INSERT_AFTER_ABOVE(found_before, pointer, key, value, level)
0:       if level ∈ self.level_element_count and level − 1 ∈ self.level_element_count then
0:         if self.level_element_count[level] = self.level_element_count[level − 1] then
0:           REMOVE_LEVEL(level, pointer)
0:         else
0:           level ← level + 1
0:         end if
0:       end if
0:     end while
0:   end if
0: end function=0
```

3.6 Algorithm engineering ideas used

The process of promoting elements to higher levels in the skiplist involves a check to determine if the current level and the promoted level have the same number of elements. If this condition is satisfied, the upper level is removed from the skiplist. This approach aims to optimize both time and space efficiency by eliminating unnecessary overhead.

The average time complexity of the promotion and removal process depends on the distribution of ele-

ment accesses in the skiplist. If the access pattern is random or follows a relatively balanced distribution, the average time complexity can be considered relatively efficient. This is because, on average, the number of elements at each level will be proportional to the overall size of the skiplist. Thus, the promotion and removal process would typically involve iterating through a small number of elements at each level, resulting in a time complexity of approximately O(log n), where n is the total number of elements in the skiplist.
However, The worst-case time complexity occurs when all elements in the skiplist are accessed uniformly, resulting in each element being promoted to the top level.Therefore, the worst-case time complexity can be considered as O(n), where n is the total number of elements in the skiplist.
The space efficiency of this approach is generally improved by removing redundant levels from the skiplist. When elements are promoted to the top level, the removal of redundant levels ensures that only the necessary levels are retained, reducing the overall memory footprint. By removing these levels, the skiplist can achieve a more compact structure, resulting in improved space efficiency. The amount of space saved depends on the number of redundant levels and the size of the skiplist. In practice, the space efficiency gain can vary, but it generally leads to reduced memory usage compared to a skiplist that retains all levels regardless of their occupancy.

4 Experimental Evaluation

4.1 Hardware

Table 1 shows the specifications of the computer.

Table 1: Computer Specifications

Chip	Intel core i5 8300H(2.3 GHz)
Core	8 CPU
Memory	LPDDR5 16GB
Storage	1TB HARD DISK

4.2 Preliminary Tests

In the preliminary test phase, we conducted experiments using randomly generated numbers to evaluate the functionality of the skiplist data structure and its variations. Specifically, we tested the insertion and search operations to ensure that the skiplist and its different variations performed as expected. By inserting random numbers into the skiplist and subsequently searching for specific elements, we verified the correct functioning of the implemented functionalities.
This preliminary test phase served as an initial assessment of the skiplist and its variations, allowing us to confirm their basic operations and establish a foundation for further experimentation and analysis.

4.3 Data

In order to conduct our experiments, we employed two distinct types of data streams: 1) uniformly random generated numbers and 2) numbers generated with an exponential distribution. Our dataset consisted of 100k numbers, which were used to test all variations of skiplists. We aimed to evaluate the performance of each skiplist variation by measuring the time and movements required for searching and inserting elements.
The first data stream comprised numbers generated uniformly at random. This distribution ensured that the elements were evenly spread throughout the skiplist, offering a balanced scenario for testing. By utilizing this stream, we aimed to assess the overall efficiency and effectiveness of the skiplist variations when subjected to a uniformly random access pattern.
The second data stream consisted of numbers generated with an exponential distribution. This distribution introduced a skewed pattern of element popularity, with some elements being more frequently accessed than others. By incorporating this stream, we aimed to evaluate the adaptability and performance of the skiplist variations in scenarios where certain elements are accessed more frequently, simulating real-world access patterns that are often unevenly distributed.

To ensure comprehensive testing, we generated a dataset of 100k numbers for each data stream. We then applied the same dataset to each skiplist variation, measuring the time required for both searching and inserting elements. Additionally, we tracked the movements made during these operations, providing insights into the efficiency and effectiveness of each skiplist variation.
By utilizing these two diverse data streams and conducting rigorous experiments, we aimed to gain a thorough understanding of how each skiplist variation performs under different access patterns. The comprehensive evaluation of time, movements, and overall efficiency allows for informed analysis and comparison of the skiplist variations, ultimately aiding in identifying the most suitable variation for specific use cases.

4.4 Results

The experiments conducted on variations of skiplists involved a dataset consisting of 100k numbers ranging from 1 to 1000. Two different types of data distributions were examined: uniformly random numbers and exponentially distributed numbers. In the random number stream, there were 1000 unique numbers, while the exponential number stream contained approximately 35 unique numbers.
In figure 1 and 3 we can see that For the uniformly random number stream, it was found that the basic skiplist performed the most efficiently, exhibiting the shortest execution time and the least number of movements. On the other hand, the variations of skiplists, namely promote to top, promote one level up, and promote to random level, required more time to complete the experiment and resulted in a higher number of movements. Notably, the promote to top variation had the longest execution time.
In figure 2 and 4 we can see that When tested with exponentially distributed numbers, where certain numbers occurred more frequently than others, the basic skiplist showed the longest execution time. In contrast, the variations of skiplists demonstrated shorter execution times. Specifically, promoting elements to one level higher than the current level resulted in the shortest execution time. Comparing the movements in the skiplists, the basic skiplist had the highest number of movements, while the variations of skiplists showed fewer movements overall. Notably, all the variations had similar numbers of movements during the experiment.

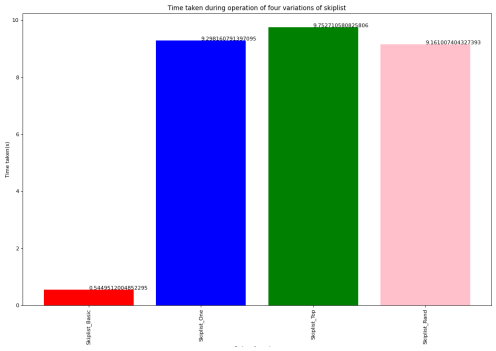

Figure 1: Time taken during operation of four variations of skiplist with random stream of numbers

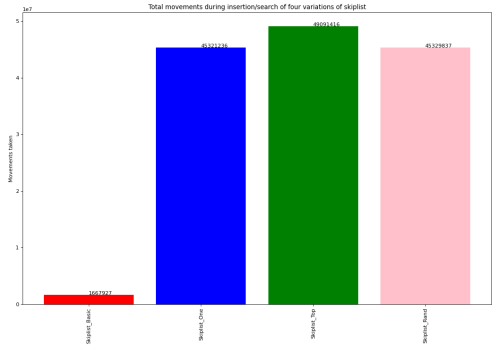

Figure 2: Movements during operation of four variations of skiplist with exponential stream of numbers

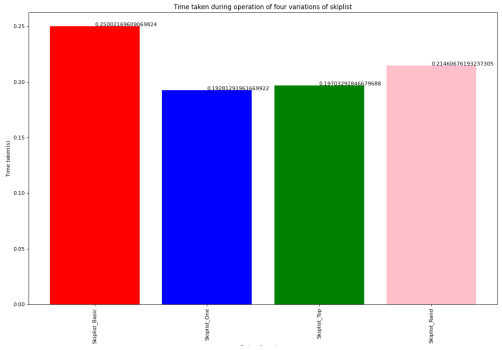

Figure 3: Time taken during operation of four variations of skiplist with exponential stream of numbers)

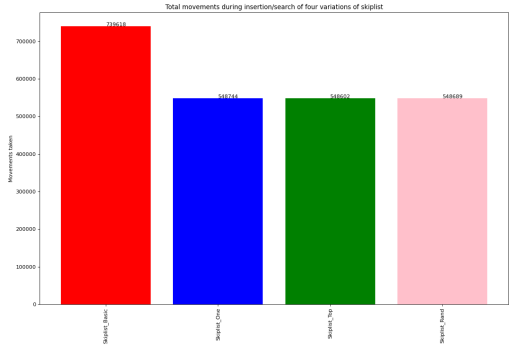

Figure 4: Movements during operation of four variations of skiplist with random stream of numbers

5 Discussion

5.1 Execution Time and Movements in Skiplist Variations with Uniformly Random Numbers

In our experiments, we observed the performance of various skiplist variations when operating on a stream of 100,000 uniformly random numbers within the range of 1 to 1000. The basic skiplist demonstrated the shortest execution time compared to the variations, while the "promote to top" variant exhibited the longest execution time. Additionally, the basic skiplist required the fewest movements during execution, indicating its efficiency in handling uniformly random number streams.

5.2 Impact of Data Distribution on Skiplist Performance

Our findings emphasize the crucial role of data distribution in skiplist performance. With uniformly random number streams, the basic skiplist excelled in terms of execution time and movements. However, when dealing with exponentially distributed number streams, the variations proved to be more efficient, outperforming the basic skiplist in terms of execution time and reducing movements.

6 Conclusion

Our experiments on skiplist variations using different data distributions provide valuable insights into their performance characteristics. The choice of skiplist variant depends on the nature of the data stream. For uniformly random number streams, the basic skiplist offers optimal performance with shorter execution times and minimal movements. Conversely, when dealing with data streams that exhibit non-uniform distributions, such as the exponential number stream tested in our experiments, the variations of skiplists prove to be more efficient, reducing execution time and minimizing movements. These findings highlight the importance of considering data characteristics when selecting an appropriate skiplist variant for a given application.
By understanding the strengths and weaknesses of different skiplist variations, researchers and practitioners can make informed decisions in choosing the most suitable variant to optimize performance based on the specific characteristics of their data.

7 Bibliography

[2] [1] [3]

References

[1] Funda Ergün, Süleyman Cenk Sahinalp, Jonathan Sharp, and Rakesh K. Sinha. Biased skip lists for highly skewed access patterns. In *Revised Papers from the Third International Workshop on Algorithm Engineering and Experimentation*, ALENEX '01, page 216–230, Berlin, Heidelberg, 2001. Springer-Verlag.

[2] William Pugh. Skip lists: a probabilistic alternative to balanced trees. *Commun. ACM*, page 668–676, 1990.

[3] Z. Xie, Q. Cai, H. Jagadish, B. Ooi, and W. Wong. Parallelizing skip lists for in-memory multi-core database systems. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*, pages 119–122, Los Alamitos, CA, USA, 2017. IEEE Computer Society.