**VIETNAM NATIONAL UNIVERSITY, HO CHI MINH CITY**

**UNIVERSITY OF SCIENCE**

**FACULTY OF INFORMATION TECHNOLOGY**

****

**PROJECT REPORT**

**Topic:** Skip List

**Subject:** Data Structures and Algorithms

*Project members:*   *Supervisors:*

Vũ Trần Minh Hiếu (24127003) Mr. Nguyễn Thanh Tình

Hoàng Đức Thịnh (24127240) Ms. Nguyễn Ngọc Thảo

Trần Viết Bảo (24127270)

Table of Contents

[1. Introduction about Skip List 1](#_Toc194947139)

[1.1. History of Skip List 1](#_Toc194947140)

[1.2. Applications of Skip List 1](#_Toc194947141)

[2. Variants and Improvements 2](#_Toc194947142)

[2.1. Deterministic Skip List 2](#_Toc194947143)

[2.1.1. Overview 2](#_Toc194947144)

[2.1.2. Characteristics 2](#_Toc194947145)

[2.1.3. Implementation Methods 2](#_Toc194947146)

[2.1.4. Applications 2](#_Toc194947147)

[2.2. Concurrent Skip List 3](#_Toc194947148)

[2.2.1. Overview 3](#_Toc194947149)

[2.2.2. Characteristics 3](#_Toc194947150)

[2.2.3. Implementation Methods 3](#_Toc194947151)

[2.2.4. Applications 3](#_Toc194947152)

[2.3. Skip Tree 4](#_Toc194947153)

[2.3.1. Overview 4](#_Toc194947154)

[2.3.2. Characteristics 4](#_Toc194947155)

[2.3.3. Implementation Methods 4](#_Toc194947156)

[2.3.4. Applications 4](#_Toc194947157)

[3. Skip List Overview 6](#_Toc194947158)

[3.1. Skip List structure 6](#_Toc194947159)

[3.2. Skip List operations 6](#_Toc194947160)

[3.2.1. Initialization 6](#_Toc194947161)

[3.2.2. Insertion 6](#_Toc194947162)

[3.2.3. Deletion 7](#_Toc194947163)

[4. Skip List Operations Example: Insertion and Deletion 8](#_Toc194947164)

[4.1. Initial Skip List 8](#_Toc194947165)

[4.2. Insertion Process 8](#_Toc194947166)

[4.3. Deletion process: 10](#_Toc194947167)

[5. Time Complexity Analysis 12](#_Toc194947168)

[5.1. Search Operation 12](#_Toc194947169)

[5.2. Insertion Operation 12](#_Toc194947170)

[5.3. Deletion Operation 13](#_Toc194947171)

[5.4. Test whether a Skip List is empty 13](#_Toc194947172)

[5.5. Get the number of elements in a Skip List 13](#_Toc194947173)

[5.6. Build a Skip List from given items 14](#_Toc194947174)

[5.7. Remove all elements from the Skip List 14](#_Toc194947175)

[6. Comparison between Skip List and Regular Linked List 15](#_Toc194947176)

[6.1. Similarities 15](#_Toc194947177)

[6.2. Dissimilarities 15](#_Toc194947178)

[7. Test Cases and Experiment Results 17](#_Toc194947179)

[7.1. Statements 17](#_Toc194947180)

[7.2. Test case and Experiment results 17](#_Toc194947181)

[7.3. Analysis of test case and experiment results 19](#_Toc194947182)

[7.3.1. Algorithm accuracy 19](#_Toc194947183)

[7.3.2. Execution time and complexity analysis 19](#_Toc194947184)

[7.3.3. Conclusion 19](#_Toc194947185)

[8. Appendix 20](#_Toc194947186)

[8.1. Group Information 20](#_Toc194947187)

[8.2. Work Assignment Table and Contribution Evaluation 20](#_Toc194947188)

[8.3. Project requirement completion rate 20](#_Toc194947189)

[9. References 22](#_Toc194947190)

**Table of Tables**

[**Table 1:** Execution time analysis of Skip List operations 18](#_Toc194614253)

[**Table 2:** Group information 20](#_Toc194614254)

[**Table 3:** Work assignment table and contribution evaluation 20](#_Toc194614255)

**Table of Figures**

[**Figure 1:** Skip List 8](#_Toc194614265)

[**Figure 2:** Skip List after Inserting element 40 9](#_Toc194614266)

[**Figure 3:** Skip List after Inserting element 20 10](#_Toc194614267)

[**Figure 4:** Skip List after Deleting element 30 11](#_Toc194614268)

# Introduction about Skip List

## History of Skip List

* Skip Lists were invented by William Pugh in 1989 and first described in his paper "Skip Lists: A Probabilistic Alternative to Balanced Trees," published in the June 1990 issue of Communications of the ACM.
* Pugh developed Skip Lists as a simpler alternative to balanced trees, which required complex rebalancing operations. By using randomization instead of strict balancing rules, Skip Lists achieve good average-case performance while being much easier to implement and maintain.

## Applications of Skip List

* Skip Lists find applications in various domains:
  + **Database Systems**: Used in Redis for sorted sets, MemSQL (now SingleStore), and LevelDB.
  + **Programming Languages and Libraries**: Several standard libraries include skip list implementations, such as Java's ConcurrentSkipListMap and ConcurrentSkipListSet. .NET Framework, and various C++ libraries.
  + **Concurrent Data Structures**: Well-suited for concurrent access due to localized modifications and support for lock-free implementations.
  + **Network Applications**: Used in peer-to-peer systems, distributed hash tables, and content delivery networks.
  + **Memory-Efficient Indexing**: Provides good balance between performance and memory usage in constrained environments.

# Variants and Improvements

## Deterministic Skip List

### Overview

* The Deterministic Skip List is a variant of the classic Skip List, introduced by William Pugh in 1990. Unlike the original Skip List which uses randomization, the Deterministic Skip List employs deterministic rules to build its data structure, ensuring more stable and predictable performance.

### Characteristics

* **Deterministic Structure:** Does not use randomization; nodes are distributed according to fixed rules.
* **Balanced Performance**: Guarantees search time in the worst case.
* **Memory Efficient**: Optimizes memory usage due to the predetermined structure.
* **Guaranteed Balance**: Not affected by uneven distributions as in randomized Skip Lists.

### Implementation Methods

* **Perfect Skip List**: The node appears at level if is a multiple of , creating a perfect structure.
* **Deterministic SkipNet**: Uses partitioning techniques to ensure balance and even distribution.
* **Biased Skip List**: Adjusts the probability of nodes appearing at certain levels based on their positions.

### Applications

* **Databases**: Used in database systems requiring stable performance.
  + **Embedded Systems**: Suitable for systems with limited resources and high determinism requirements.
* **Real-Time Applications**: Ensures consistent response times for real-time applications.
* **Routing Algorithms**: Used in network routing algorithms demanding high reliability.

## Concurrent Skip List

### Overview

* The Concurrent Skip List is a version of the Skip List specifically designed for multi-threaded environments, allowing multiple threads to access and modify the data structure simultaneously without causing consistency issues. This structure combines the advantages of Skip Lists with efficient synchronization mechanisms.

### Characteristics

* **Thread-Safe**: Designed to handle concurrent operations from multiple threads.
* **High Scalability**: Performance increases with multiple participating threads, suitable for high-load systems.
* **No Global Locks**: Uses sophisticated locking or lock-free techniques to avoid bottlenecks.
  + **High Consistency**: Ensures data consistency even with multiple concurrent operations.

### Implementation Methods

* **Lock-free Skip List**: Uses atomic operations such as Compare-And-Swap (CAS) to ensure consistency without locks.
* **Java concurrentSkipListMap**: Implementation in Java's standard library, using advanced concurrency techniques.
* **Optimized Locking Methods**: Uses fine-grained locking to minimize conflicts between threads.
* **Transactional Memory**: Employs transactional memory to simplify concurrent programming.

### Applications

* **Distributed Systems**: Used in distributed systems needing to process multiple concurrent requests.
* **Multi-threaded Caching**: Implements high-performance caching in multi-threaded environments.
* **Online Transaction Systems**: Processes multiple financial transactions simultaneously with high reliability.
* **Java Concurrent Collections**: Implementations such as ConcurrentSkipListMap and ConcurrentSkipListSet in Java.
* **Memory Management Systems**: Used in efficient memory management algorithms for multi-threaded systems.

## Skip Tree

### Overview

* Skip Tree is an extension of the Skip List concept that incorporates tree-like properties to enhance performance and flexibility. While Skip Lists are essentially linked lists with additional forward pointers to allow for fast traversal, Skip Trees combine the simplicity of Skip Lists with the structural advantages of search trees.

### Characteristics

* **Hierarchical Structure**: Organizes data in a multi-level tree-like structure while maintaining Skip List properties.
* **Improved Search Complexity**: Offers search time with better practical performance than standard Skip Lists in certain scenarios.
* **Dynamic Rebalancing**: Many implementations include mechanisms for automatic rebalancing, improving worst-case performance.
* **Space Efficiency**: Some implementations provide better space utilization compared to traditional Skip Lists.

### Implementation Methods

* **Deterministic Skip Trees**: Uses deterministic rules to build the tree structure, similar to Deterministic Skip Lists but with tree-based organization.
* **Probabilistic Skip Trees**: Employs randomization to determine the structure, similar to the original Skip List concept.
* **Self-Adjusting Skip Trees**: Dynamically adjusts the structure based on access patterns to optimize frequently accessed paths.
* **Multi-dimensional Skip Trees**: Extensions that support efficient queries on multiple dimensions or attributes simultaneously.

### Applications

* **Spatial Data Structures**: Used in geographic information systems for efficient spatial queries.
* **Database Indexing**: Provides alternative indexing mechanisms in database management systems.
* **Network Routing**: Applied in advanced routing algorithms that require hierarchical data representation.
* **Memory-Efficient Search Systems**: Utilized in systems where memory overhead must be minimized while maintaining fast search capabilities.
* **Range Query Optimization**: Particularly effective for range-based queries compared to standard Skip Lists.

# Skip List Overview

## Skip List structure

* Consider that each column in the Skip List has the following basic structure:
* : The value stored in the node.
* : An array of cells, each cell contains two pointers: **forward** and **backward** which point to the next and previous element at the same level respectively.
* And the Skip List is structured as follows:
* : A sentinel node marking the beginning of the Skip List.
* : A sentinel node pointing to the last element.
* : Maximum possible levels in the Skip List (usually levels, where is the number of elements). Both head and tail have levels.

## Skip List operations

### Initialization

* For node: All forward pointers point to the tail node’s cell that has the same level.
* For node: All backward pointers point to the head node’s cell that has the same level.
* All other pointers will point to null.

### Insertion

* Consider inserting a new node with value . First, generate a random number of levels for the new node, is less than or equal to . Then follow the following steps:
  + - **Step 0**: Starting from level .
    - **Step 1**: At level , move forward horizontally until we reach node or a node with value , which is greater than , we will put node before node or node by updating **forward** and **backward** pointers of the related nodes.
    - **Step 2**: Decrease by one, if is greater than or equal to zero, go back to **step 1**.

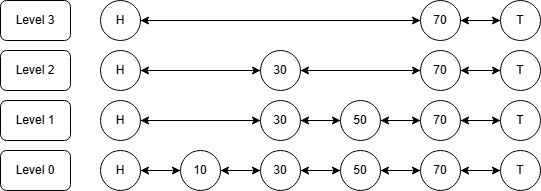
### Deletion

* Consider deleting a node with value .
  + - **Step 0**: Using search operation to locate the position of in Skip List.
    - **Step 1**: At every level of node remove the **forward** and **backward** pointers, and update pointers of related nodes.
    - **Step 2**: Deallocate the block of memory used for node .

# Skip List Operations Example: Insertion and Deletion

## Initial Skip List

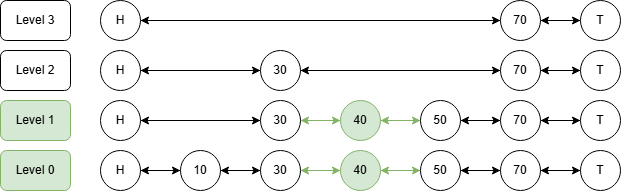
* Begin with this Skip List containing the elements 10, 30, 50, and 70:



**Figure 1:** Skip List

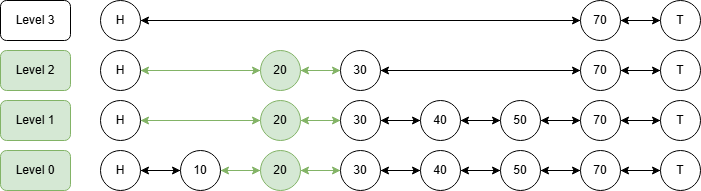
## Insertion Process

* Let’s insert the value 40 into our skip list.
* **Step 1: Determine the random level for the new node**
  + Suppose our coin flips determine that the new node for 40 will have 2 levels, which means it will appear in level 0 and 1.
* **Step 2: Find the insertion position at each level**
  + We start from the highest level (level 3) and the head node:
* **Level 3**: From head, we look for the rightmost node whose value is less than 40. The only node we can move to is 70, but 70 > 40, so we can't move. We drop down to level 2.
* **Level 2**: From head, we see 30 < 40, so we move to 30. From 30, the next node is 70 > 40, so we can't move further. We record 30 as the predecessor of 40 at level 2, then drop down to level 1.
* **Level 1**: From 30, we see 50 > 40, so we can't move. We record 30 as the predecessor of 40 at level 1, then drop down to level 0.
* **Level 0**: From 30, we see 50 > 40, so we can't move. We record 30 as the predecessor of 40 at level 0.
* **Step 3: Insert the new node**
* We insert 40 after its predecessor at level 0 and 1:



**Figure 2:** Skip List after Inserting element 40

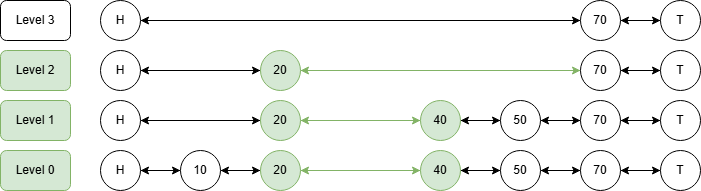
* Now let’s insert 20 with a randomly generated level of 3:
* **Step 1: Determine the random level for the new node**
  + Our coin flips give us a level of 3 for the new node containing 20.
* **Step 2: Find the insertion position at each level**
  + **Level 3:** From head, we can't move (70 > 20). We record head as the predecessor at level 3.
  + **Level 2:** From head, we can't move (30 > 20). We record head as the predecessor at level 2.
  + **Level 1:** From head, we can't move (30 > 20). We record head as the predecessor at level 1.
  + **Level 0:** From head, we see 10 < 20, so we move to 10. From 10, we see 30 > 20, so we can't move. We record 10 as the predecessor at level 0.
* **Step 3: Insert the new node**
  + We insert 20 after its predecessors at all levels



**Figure 3:** Skip List after Inserting element 20

## Deletion process:

* Now let's delete element 30 from our skip list.
* **Step 1: Find the node to delete and its predecessors**
  + Starting from the highest level (level 3) and the head node:
    - **Level 3**: From head, we move to 20. From 20, the next node is 70 > 30, so we record 20 as the predecessor at level 3 and drop down.
    - **Level 2**: From 20, the next node is 30, which is the node we want to delete. We record 20 as the predecessor at level 2.
    - **Level 1**: From 20, the next node is 30, which is the node we want to delete. We record 20 as the predecessor at level 1.
    - **Level 0**: From 20, the next node is 30, which is the node we want to delete. We record 20 as the predecessor at level 0.
* **Step 2: Update the pointers to bypass the node being deleted**
  + At each level where 30 appears, we update the predecessor's pointer to point to 30's successor:
    - **Level 2**: Update 20's level 2 pointer to point to 70 (30's successor at level 2)
    - **Level 1**: Update 20's level 1 pointer to point to 40 (30's successor at level 1)
    - **Level 0**: Update 20's level 0 pointer to point to 40 (30's successor at level 0)
* After deletion, our skip list looks like this:



**Figure 4:** Skip List after Deleting element 30

# Time Complexity Analysis

* Suppose that the number of elements in the Skip List is and the highest level is .

## Search Operation

* **Best-case:**
* The best case occurs when the element we are searching for is at the head of the Skip List and on the highest level. Therefore, we only need one comparison to return the position of that element, and the time complexity will be .
* **Worst-case:**
* The worst case occurs when the Skip List has only one level (which makes our Skip List a Linked List) and the element that we are looking for is at the tail of the Skip List. Because of that, we will have to traverse through the whole list and the time complexity will be .
* **Average-case:**
* When the number of levels of the Skip List are distributed appropriately, making the highest level of the list . Since all the elements are in their proper position, traversing through the list and returning the position of the element we are looking for only takes .

## Insertion Operation

* **Best-case:**
* The best case occurs when the position for the element to be inserted is at the head of the Skip List, but we need to traverse from highest level to lowest level to find the levels to insert, this makes the time complexity
* **Worst-case:** 
  + The worst case occurs when the element has to be inserted to the tail of the Skip List and the number of levels of the list is exactly and the inserted element’s number of levels is also after doing the *coin flips* process.
  + Therefore, traversing through the whole Skip List takes and all levels to finish inserting takes , making the time complexity *.*
* **Average-case:**
* If all the elements and their levels are distributed appropriately, then the process of searching for a proper position for a new element takes and if the number of levels of that new element is also *,* the average time complexity of the insertion operation is .

## Deletion Operation

* **Best-case:**
* The best case is when the element to be deleted is at the head of the Skip List, but we need to traverse from highest level to lowest level to find the levels of the nodes to delete, this make the time complexity of this process .
* **Worst-case:**
* The worst case occurs when the element to be deleted is at the tail of the Skip List and the list’s number of levels if .
* Therefore, we have to traverse through the whole Skip List to the last element and perform the deletion that takes .
* **Average-case:**
* Since the Skip List’s elements and levels are distributed appropriately, the searching process takes and if the number of levels in the list is , then the deletion operation’s time complexity is .

## Test whether a Skip List is empty

* **Time complexity:**
  + Because the implementation in the code uses 2 head and tail nodes, we only need to check if the head node pointer points to the tail node:
    - Yes: Skip List is empty.
    - No: Skip List is not empty.

## Get the number of elements in a Skip List

* **Time complexity:**
  + Because the implementation in the code uses a counter variable that updates the number of elements in the Skip List after each successful insertion and deletion operation.
  + So, every time you need to get the number of elements, just return numElements, so it only takes time complexity.

## Build a Skip List from given items

* **Time complexity:** 
  + Iterate through all the elements to be inserted and perform the insert operation.
  + Each insert has a time complexity of .
  + Let be the number of elements to be inserted, the total time complexity is .

## Remove all elements from the Skip List

* **Time complexity:** 
  + We just need to go through the entire Skip List at the lowest level, for each node, we just need to delete that node and move to the next node, so the time complexity is only .

# Comparison between Skip List and Regular Linked List

## Similarities

* **Node-based** **structure:** Both implementations utilize nodes containing data elements and pointers to organize information.
* **Dynamic size:** Both data structures do not have a fixed size. Therefore, both can dynamically increase or decrease in size during runtime without requiring prior declaration.
* **Linear traversal:** Both data structures can traverse through each element from the beginning to the end of the list by using pointers.
* **No direct access support:** Both data structures do not support random access with time complexity like arrays but instead require linear access.
* **Memory allocation:** Both structures allocate memory dynamically as new elements are added, rather than requiring contiguous memory blocks.

## Dissimilarities

* **Structure complexity:**
* **Regular Linked List:** A single level of connections between nodes.
* **Skip List:** Multiple levels of connections, creating a hierarchy of linked lists. Higher level pointers can “bypass” nodes to traverse faster.
* **Search time complexity:**
* **Regular Linked List:** Slower search, best-case time complexity is and worst-case time complexity is since it has to traverse through the list from the beginning to the end.
* **Skip List:** Faster search, when elements are distributed appropriately, the average-case time complexity will be . This is achieved through pointers at higher levels that allow us to “skip over” numerous nodes of the list, thereby reducing the search space logarithmically.
* **Insertion and deletion time complexity:**
* **Regular Linked List:** It has best-case insertion or deletion of when the node to be inserted or deleted is at the beginning of the list, and worst-case complexity of when the node is at the end of the list. The insertion and deletion operations are simpler because they only require modifying a few pointers before and after the node that needs to be inserted or deleted.
* **Skip List:** When elements are distributed appropriately, it has time complexity of for insertion or deletion operations. However, these operations are more complex since they require updating pointers at multiple levels. The number of levels for a new node is determined randomly, which helps maintain a probabilistically balanced structure.
* **Space complexity:**
* **Regular Linked List:** Lower space complexity, each node of the list contains only a value and a pointer (or two pointers in Doubly Linked List).
* **Skip List:** Higher space complexity for multiple levels and pointers, which helps performing operations faster and more efficiently.
* **Order and Uniqueness:**
* **Regular Linked List:** Will not maintain order and will have duplicate values unless we have conditions for the insertion operation.
* **Skip List:** Designed to keep elements in order and unique (like dictionaries or search trees). Since Skip List is a probabilistic alternative to AVL trees, ordering and uniqueness are core features.
* **Randomness:**
* **Regular Linked List:** Its structure is predictable and only bases on insertion and deletion operations.
* **Skip List:** Hard to predict the structure since each node has its random number of levels.

# Test Cases and Experiment Results

## Statements

* Given a set of integers, implement the following operations:
* : Add the integer to the set .
* : Remove the integer from the set .
* : Find the smallest integer in the set .
* : Find the largest integer in the set .
* : Find the smallest integer in the set that is greater than .
* : Find the smallest integer in the set that is greater than or equal to .
* : Find the largest integer in the set that is smaller than .
* : Find the largest integer in the set that is smaller than or equal to .
* **Notes:**
* For the operation, if already exists in , do not add it again.
* For the operation, if is not in the set , leave the set unchanged.
* For the operations, print **“empty”** if the set is empty.
* For the operations, print **“no”** if no such integer exists in the set .
* **Input:** The input consists of multiple lines. Each line starts with an integer from to , indicating the operation to be performed. The number indicates the end of the input. For operations , the next integer on the line is , which is the parameter of the operation.
* **Output:** For each operation of type , print a line containing the result of the operation.

## Test case and Experiment results

* Let be the number of operations
* All test cases and code can be downloaded from [Google Drive](https://drive.google.com/drive/folders/1uVYldPcJ364Of7J15HXzaaqdb2o5ogIW)

**Table 1:** Execution time analysis of Skip List operations

|  |  |  |  |
| --- | --- | --- | --- |
| **Test case** | **Limit** | **Execution time** | **Result** |
| **1** |  | 0.00007s | Pass |
| **2** |  | 0.00005s | Pass |
| **3** |  | 0.00008s | Pass |
| **4** |  | 0.00023s | Pass |
| **5** |  | 0.00066s | Pass |
| **6** |  | 0.00677s | Pass |
| **7** |  | 0.08691s | Pass |
| **8** |  | 0.28311s | Pass |
| **9** |  | 0.54775s | Pass |
| **10** |  | 1.14352s | Pass |

## Analysis of test case and experiment results

### Algorithm accuracy

* All test case have the result **“Pass”**, meaning that the output file from [Skip List Code](https://drive.google.com/file/d/1ajfh34THdeAFe1u3THPm1nZS2fXPB3En/view?usp=drive_link) matches the answer file from [Brute Force Code](https://drive.google.com/file/d/1gcgoxgOyClnh2Byoke6Mo9TClGLzhbYO/view?usp=drive_link)
* This confirms that the implemented Skip List algorithm produces correct results for all given input datasets

### Execution time and complexity analysis

* For small test cases (, execution time is very low (
* As the number of operations increases, the execution time increases
* In the final test with and , the execution time of the code is
* Given that the computer can perform , and the code has a complexity of , the theoretical execution time of the final test is:
* Thereby we see that the theoretical execution time of the final test is more than the actual execution time. Although this is quite high, the execution time is still within the guaranteed range.

### Conclusion

* **Accuracy:** The Skip List algorithm functions correctly as expected.
* **Performance:** The algorithm runs very fast for small . For large , execution time increases but remains within a reasonable range.
* **Practical Application:** Skip List is an efficient data structure for searching and dynamic data operations. However, for very large datasets (), consider optimizing memory usage and using the improved implementation techniques mentioned earlier to further improve performance.

# Appendix

## Group Information

**Table 2:** Group information

|  |  |  |
| --- | --- | --- |
| **No.** | **Full Name** | **Student ID** |
| 1 | Vũ Trần Minh Hiếu | 24127003 |
| 2 | Hoàng Đức Thịnh | 24127240 |
| 3 | Trần Viết Bảo | 24127270 |

## Work Assignment Table and Contribution Evaluation

**Table 3:** Work assignment table and contribution evaluation

|  |  |  |  |
| --- | --- | --- | --- |
| **No.** | **Member** | **Assigned Task** | **Completion Rate** |
| 1 | Vũ Trần Minh Hiếu | Code, Report | 100% |
| 2 | Hoàng Đức Thịnh | Slide, Code | 100% |
| 3 | Trần Viết Bảo | Slide, Report, Video | 100% |

## Project requirement completion rate

* **Source Code:**
  + The basic operations of Skip List have been fully implemented, including:
    - Checking if the Skip List is empty
    - Counting the number of elements in the Skip List
    - Inserting a new element
    - Deleting an element
    - Constructing a Skip List from given elements
    - Deleting the entire Skip List
  + Additionally, we have implemented some improved operations, such as:
    - Search operation
    - Finding the smallest element that is greater than or equal to
    - Finding the smallest element that is greater than
  + In the function, we also provide several illustrative examples for basic operations such as insertion, deletion, and searching in Skip List to meet the given requirements.
  + **Evaluation:** 100% completed
* **Report:**
  + The report has been completed fully according to the requirements:
    - Introduction to the history and applications of Skip List
    - Overview of some variations and improvements of Skip List and their applications
    - Step by step illustration of insert and delete operations
    - Time complexity analysis of basic operations in Skip List
    - Point out similarities and differences between Skip List and Regular Linked List
  + Additionally, we have implemented some extra features, such as:
    - Explanation of how the Skip List is structured in the attached source code file
    - Explanation of the concept behind implementing basic operations like initialization, insertion, and deletion in Skip List
    - Validation of code correctness through a test case with 10 randomized tests, each varying in the number of operations performed. Furthermore, we provide an analysis and evaluation of the correctness and efficiency of the code when handling large-scale operations
  + **Evaluation:** 100% completed
* **Video:**
  + Presentation video URL: [Presentation video](https://drive.google.com/file/d/1BFSAs5WytBK4Lmn85YHpd4U1uHihJKlD/view?usp=drive_link)
  + **Evaluation:** 80% completed
* **Conclusion:** Approximately 93% of the requirements have been fulfilled.

# References

1. Papadakis, T. (1993). *Deterministic Skip Lists* (Technical Report CS-93-28). University of Waterloo.
2. Wikipedia contributors. (n.d.). *Skip list*. Wikipedia, The Free Encyclopedia.
3. Singh, P. (2024, June 30). *Deterministic Skip List – Advanced Data Structures*.
4. Munro, J. I., Papadakis, T., & Sedgewick, R. (1993). Deterministic Skip Lists. In *Proceedings of the Workshop on Algorithms and Data Structures*.
5. Sasidharan, A. (2023). *Concurrent Deterministic Skiplist and Other Data Structures*.
6. Pugh, W. (1989). *Concurrent maintenance of skip lists*. University of Maryland.
7. Gonzalez-Beltran, A. (2008). *Range queries over skip tree graphs*.