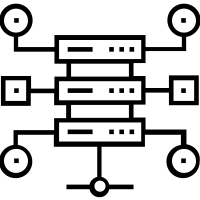
**Assignment No: \_4\_\_**

**Date: 14/ 03 /2025**

**Title: Implementation and Performance Comparison of Tree Data Structures for Efficient File System Management**

(Title based on the application domain and the data structure you will be implementing)

|  |  |  |  |
| --- | --- | --- | --- |
| **Assignment Type of Submission:** |  |  |  |
| **Group** | Yes | Yuxuan Song  24207239  Yijun Liu  24202574  Deepak Shelke  24208478 | **33.33**  **33.33**  **33.33** |

****

1. **Problem Domain Description:**

This report explores the use of hash-based map data structures to efficiently handle employee salary records in a business system. The simulated business scenario involves an enterprise that manages a large number of employee salary entries, which are frequently updated and queried. Specifically, the task focuses on increasing the salaries of employees earning less than 10,000 units, a common operation in payroll adjustment and compensation management systems.

To mimic a real-world environment, a dataset of 10,000 employee records was generated. Each employee was assigned a unique ID in the format of "EMP" followed by a random number, and a salary amount was assigned as a random floating-point number between 3000 and 30,000. These records were saved in a CSV file named employee\_salaries.csv, with each line containing an employee ID and a salary value.

The naive approach of using a simple list for this task would result in O(n) time complexity for each lookup or update, leading to O(n²) total time if salary adjustments are applied across all entries. This is impractical for systems handling large datasets. Instead, hash-based maps were chosen for this experiment due to their average O(1) lookup and update complexity, which makes them highly suitable for real-time employee salary processing.

The program follows a clear structure. First, the EmployeeSystem class generates the CSV data file and reads it into a Map. Once the data is loaded, the system iterates over the entries to apply the salary adjustment: if an employee’s salary is below 10,000 units, it is increased by 10%. Different implementations of the Map interface were tested, including HashMap, LinkedHashMap, and ConcurrentHashMap, to measure both performance and behavior under different access-order and thread-safety scenarios.

The results of the experiment demonstrated clear performance differences among the tested hash-based map implementations. Hashtable delivered the fastest processing time at 15 milliseconds, followed by LinkedHashMap at 20 milliseconds, ConcurrentHashMap at 23 milliseconds, and HashMap at 28 milliseconds. The slight variations in performance can be attributed to the internal design of each map: Hashtable is a legacy synchronized structure, ConcurrentHashMap is optimized for multi-threaded environments, LinkedHashMap maintains predictable insertion order, and HashMap is a general-purpose hash table with no ordering guarantees. Despite these differences, all implementations provided efficient O(1) average-time lookup and update operations, confirming that hash-based maps are highly suitable for handling large-scale salary adjustment tasks in enterprise systems.

1. **Theoretical Foundations of the Data Structure(s) utilised**

This project mainly uses the hash table as the core data structure to store and manage employee salary records efficiently. Both the custom implementation (MyHashTable) and Java’s built-in implementations like HashMap, TreeMap, and LinkedHashMap are explored and tested.

**Hash Table (MyHashTable)**

The MyHashTable class is a simplified version of Java’s HashMap. It uses an array of linked lists (chaining) to handle collisions. Each key is hashed to a specific index in the array, and if multiple keys land in the same index, they are stored in a linked list at that index.

• Time complexity: put(K key, V value): O(1) on average, get(K key): O(1) on average, Worst-case can go up to O(n) if many keys hash to the same index

This approach is simple to implement and is often used in learning environments to understand the basic principles of hashing.

**HashMap**

Java’s HashMap is a hash table-based implementation of the Map interface. Internally, it uses an array of buckets and handles collisions using chaining with linked nodes or balanced trees (after a threshold).

• Provides O(1) performance on average for get() and put().

• Automatically resizes itself when the load factor exceeds 0.75.

Compared to MyHashTable, it is more robust and optimized, especially in large-scale scenarios.

**TreeMap**

Unlike hash-based maps, TreeMap is implemented using a Red-Black Tree, which is a self-balancing binary search tree.

• All entries are sorted based on the natural ordering of keys.

• Provides O(log n) performance for put() and get() operations.

This structure is suitable when sorted data access is required, though it sacrifices some speed compared to hash-based maps.

LinkedHashMap

This is a subclass of HashMap that maintains a doubly-linked list running through its entries, preserving insertion order.

• Offers same time complexity as HashMap (O(1) average).

• Useful when predictable iteration order is required.

Each data structure has its own strengths:

• HashMap is ideal for general-purpose key-value storage.

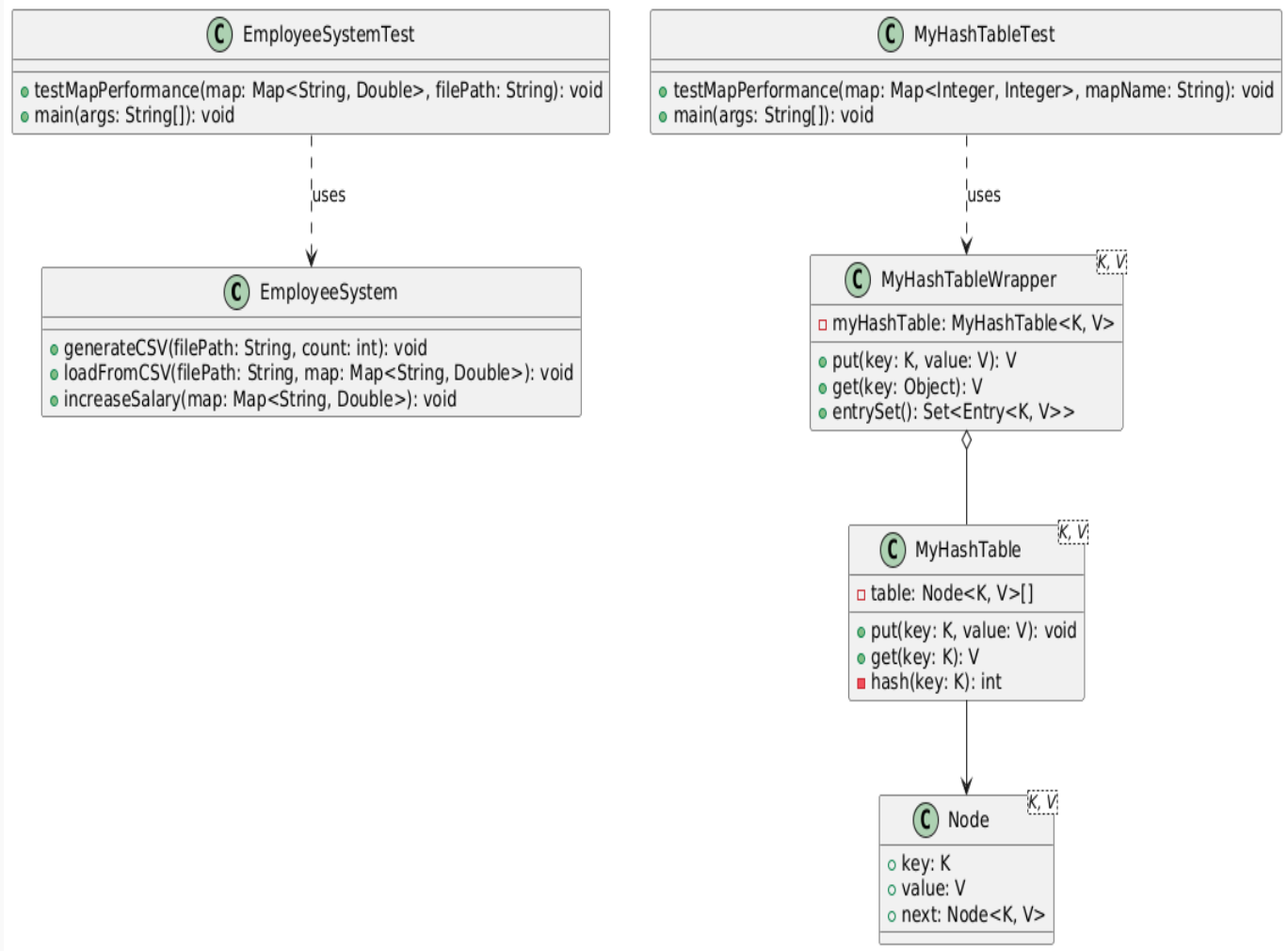
• TreeMap is useful for sorted data.

• LinkedHashMap is great when order matters.

• MyHashTable helps understand how hashing works under the hood.

In this project, performance comparisons between them helped highlight the trade-offs between simplicity, speed, and functionality.

1. **Analysis/Design (UML Diagram(s))**



1. **Code Implementation (please add your TA -** [Furqan.rustam1@gmail.com](mailto:Furqan.rustam1@gmail.com) **– as a collaborator)**

GitHub (link):https://github.com/WolfClarence/Group9\_Assignment4

This is a new repository, please join as the collaborator.

MyHashTable Implementation:

MyHashTable is a custom-built hash table designed to replicate the core behavior of Java's HashMap, with a focus on simplicity and educational value. It uses an array combined with a linked list to resolve hash collisions, avoiding more complex tree structures, which makes it easier to understand and learn.

In the design of a hash table, the core challenge is how to map keys to array indices. MyHashTable implements a hash algorithm similar to Java's HashMap, using a perturbation of the original hash code via key.hashCode() ^ (key.hashCode() >>> 16). This secondary mixing ensures a more uniform distribution of hash values in the array, reducing collisions. The final array index is determined by taking the modulus of the array length.

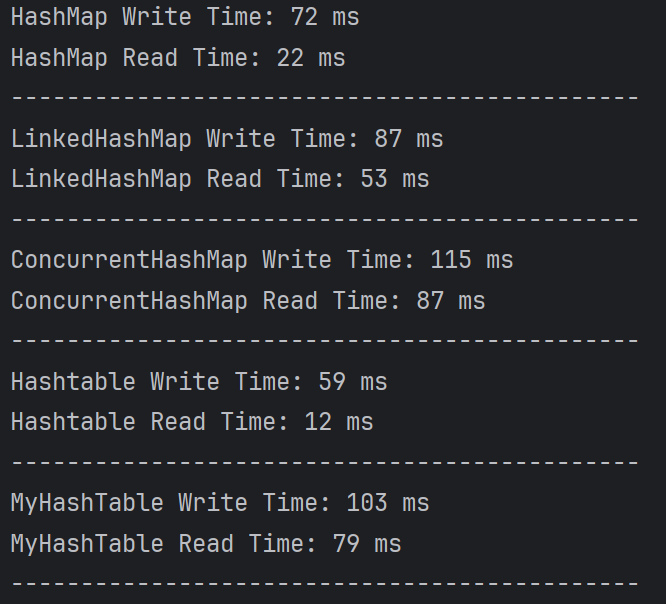
In the write operation(put), MyHashTable first calculates the hash value of the key and determines the target bucket. If the bucket is empty, the new key-value pair is inserted as the head of the linked list. If a hash collision occurs, the linked list is traversed. If an entry with the same key exists, the value is updated; otherwise, the new node is inserted at the head of the list. This approach is simple, and while the average case time complexity for insertion is O(1), in the worst case (with long chains), it can degrade to O(n).

In the read operation(get), the hash value for the key is calculated in the same way as during the write operation. The corresponding bucket's linked list is traversed, comparing each node's key to the target. The read operation has a time complexity of O(1) in the average case, but if there are many hash collisions, it could slow down, depending on the distribution of keys.

To evaluate the performance of MyHashTable, this test compares it with 4 hash-based data structures: HashMap, LinkedHashMap, ConcurrentHashMap, and Hashtable. The test scenario involves executing 1,000,000 write operations followed by 1,000,000 read operations for each data structure, measuring the time taken for each operation. The results show that the official Java hash table implementations are generally highly optimized, particularly in terms of concurrency, memory layout, and data distribution. For example, Hashtable demonstrated the fastest read time, while MyHashTable, due to its use of linked lists for collision resolution and lack of optimization, exhibited higher read and write times. Despite this, MyHashTable correctly performs the required tasks and adheres to the fundamental properties of a hash table.

Through this experiment, the performance gap between industrial-grade implementations and custom implementations becomes clear. MyHashTable, with its simple linked list-based collision resolution, provides a clear demonstration of the basic principles of hash tables, especially in terms of hash algorithms, key lookups, and insertion logic. Although its performance is noticeably slower compared to standard library implementations, it still maintains the expected O(1) average complexity for hash table operations and serves as an excellent exercise for understanding hash table structures and algorithms.

Sample output:



1. **Video of the Implementation running**

youtube link:

Comments:

This is a brief overview of the code execution and result display. The results are shown in the console.

**Please save as pdf and submit on Brightspace**

**Students belonging to the same group** please **submit the same file .**