CST2550

Software Engineering Management and Development

Coursework

Autumn/Winter term
2024/2025

Date of Submission: 16/04/25

Student Name: Koller Melanie Turinabo Bugingo

Student ID Number: M00972547

Student Name: Sarah Mariam Hossen Goburdhun

Student ID Number: M00939801

Student Name: Favour Esset Esset

Student ID Number: M00933241

Student Name: Samuel Kingsley-Ogarashi

Student ID Number: M00931065

Student Name: Iwuagwu Nkem

Student ID Number: M00897932

Lab Tutor: Mr Karel Veerabudren

Table of Contents

Introduction

This report presents the development of the GoCar Rental System project, a software developed to streamline the operations of a car rental business. The application supports key functionalities, including client registration, rental processing, and car management. Developed using Agile methodology, there was a focus on iterative development, regular feedback and adaptive planning through daily meetings. A custom-built hash table employing separate chaining was implemented into the program to optimise the data retrieval, insertion, and deletion processes with an SQL database to act as a storage. Rigorous testing was carried out using the xUnit framework, focusing on validating input data and ensuring reliable ID generation.

The report outlines the design rationale, the logic behind the algorithms, the testing procedures, and the critical reflections on the project development cycle.

Design

Justification of Selected Data Structures

A hash table was selected for the storage and retrieval of essential entities such as car records, client profiles, and rental transactions due to its efficiency and scalability. According to Karimov (2020), a hash table typically consists of any array of lists, with each data value mapped at a distinct index derived from its key using a hash function. This form of *associative mapping* allows for quick access to elements by establishing direct connections between keys and values, a strategy frequently utilised in programming due to its speed and efficiency compared to other data structures (Tapia-Fernández, García-García and García-Hernandez, 2022).

Hash tables provide an average-case time complexity of O(1) for Insert, Search, and Delete operations, (Karimov, 2020), which is vital for ensuring responsiveness and scalability in a system with high data retrieval rates. This attribute makes it ideal for the GoCar system, where user interactions need to be fast and efficient.

Hash function

Modulo division was selected as the hashing operation. This technique calculates the address by dividing the key by the size of the array and using the remainder as the index. According to Bhullar et al. (2016), this is a simple and efficient method, where the address is calculated using the formula: *Address = key % list size.* This method is easy to implement and works well with hash tables that employ arrays of prime number sizes, leading to a more even distribution of keys and less clustering.

Collision Resolution with Separate Chaining

To address collisions, defined as instances where keys in the hash table generate the same hash value (Misto, 2024), separate chaining together with a custom linked list was used. This strategy was preferred over open addressing due to its greater adaptability to datasets of unknown or increasing size (Nielsen, 2009). It also prevents clustering —where multiple keys hash to the same or nearby indices, leading to a dense concentration of entries in certain parts of the table (Misto, 2024).

Separate chaining technique using linked lists allows for constant-time insertions at the end of each chain and manageable linear-time searches, provided that the hash function evenly distributes keys across buckets (Bible and Moser, 2023). This data structure enables multiple records that hash to the same index to coexist within the same bucket, preventing data overwrites and preserving the integrity of the stored information. According to GeeksforGeeks (2025), this method effectively handles collisions, ensuring that the hash table remains functional and dependable even under high-load conditions.

Resizing Strategy and Load Factor Management

The hash table also implements a resizing strategy that ensures performance remains efficient as the data increases. Two main principles are applied:

- **Prime Number Bucket Sizes:** Employing arrays sized with prime numbers can enhance the distribution of keys within the table, leading to better performance and more reliable hash table operations. As noted by Bhullar et al. (2016), using prime-sized arrays can help distribute inputs more uniformly because prime numbers are spaced in a way that reduces repetitive patterns in key distribution. This approach aids in reducing both collisions and clustering, thereby boosting the overall effectiveness of the hash table.
- **Load Factor Threshold**: The hash table resizes when the load factor surpasses 0.75. According to Misto (2024), it's essential to keep the load factor in the optimal range, generally between 0.6 and 0.75—to maintain performance. When this limit is exceeded, expanding to a larger table facilitates quick insertions and reduces collisions, ensuring that the system functions efficiently as more entries are introduced.

In conclusion, a hash table was selected as the primary data structure for this project due to its well-established effectiveness and flexibility in scenarios requiring real-time data access (Karimov, 2020). Its ability to provide rapid and consistent operations for inserting, searching, and deleting makes it especially appropriate for applications needing quick user interactions, such as the console-based GoCar system (Tapia-Fernández, García-García and García-Hernandez, 2022). By incorporating separate chaining with linked lists, this structure efficiently handles collisions by permitting multiple records to occupy the same index without erasing existing entries (Bible and Moser, 2023). This method enhances scalability, particularly when managing unpredictable or increasing datasets, and avoids the limitations typically associated with open addressing (Misto, 2024).

Moreover, the design features a resizing mechanism based on load factor thresholds and prime-number bucket sizes to ensure stable performance and equitable key distribution (Karimov, 2020). While separate chaining involves a memory trade-off due to the overhead of linked list node pointers (Nielsen, 2009), this disadvantage is counterbalanced by the structure's greater reliability, modularity, and improved handling of collisions. In summary, the robust performance traits and dynamic adaptability of the hash table validate its choice as the foundation for GoCar's data storage strategy.

Analysis of Key Functionalities (CRUD Operations) in Data Structures

Linked List

Insert-Node Algorithm

- The **AddLast method**, adds a new node with a given key and value to the end of the linked list.
- Complexity *(Bible and Moser, 2023)*:
    - Average Case: O(1), inserting at the start(head) of the linked list.
    - Worst Case: O(n), traversing the entire list (searching through all the nodes) until desired insertion point is found.

Pseudocode

```
FUNCTION AddLast(key, value)
  CREATE a new node with key and value
  IF head is null THEN
    SET head = new node
  ELSE
    SET current = head
    // "Next" points to the following node
```

```
    WHILE current.Next is not null
      SET current = current.Next
    END WHILE
    SET current.Next = new node
  END IF
  INCREMENT node count

END FUNCTION
```

Search-Node Algorithm

- The **Find method**, searches for a node in the linked list that has a key matching the input.
- Complexity *(Bible and Moser, 2023)*:
    - Average Case: O(n), traverse depending on the position of node in the list.
    - Worst Case: O(n), if the key is at the end or not found, the entire list must be traversed.

Pseudocode:

```
FUNCTION Find(key)
  SET current = head
  WHILE current is not null
    IF current.Key equals key THEN
      RETURN current
    END IF
    SET current = current.Next
  END WHILE
  RETURN null
END FUNCTION
```

Delete-Node Algorithm

- The **Remove method**, searches for a node with a specific key and removes it from the linked list.
- Complexity *(Bible and Moser, 2023)*:
    - Average Case: O(n), traversing part of the list to find the node.
    - Worst Case: O(n), if the key is at the end or not found, the entire list must be traversed.

Pseudocode

```
FUNCTION Remove(key)
  IF head is null THEN
    RETURN false
  END IF
  IF head.Key equals key THEN
    SET head = head.Next
    DECREMENT node count
    RETURN true
  END IF
  SET current = head
  WHILE current.Next is not null
    IF current.Next.Key equals key THEN
      SET current.Next = current.Next.Next
```

```
            DECREMENT node count
            RETURN true
        END IF
        SET current = current.Next
    END WHILE
    RETURN false
END FUNCTION
```

Hash Table

Insertion Algorithm

- Adds new key-value pair or updates the value if the key exists.
- Complexity *(Misto, 2024):*
    - Average Case: O(1)
    - Worst Case: O(n), in the event of many collisions.

Pseudocode

```
FUNCTION INSERT (key, value)
    IF current load factor >= load factor threshold THEN
        CALL ResizeHashTable()
    END IF
    SET bucketIndex = GetHashCode(key)
    IF bucket[bucketIndex] is null THEN
        CREATE a new linked list at bucket[bucketIndex]
    END IF
    SET node = FIND node with matching key in bucket[bucketIndex]
    IF node exists THEN
        UPDATE node's value with the new value
    ELSE
        ADD new node with (key, value) to bucket[bucketIndex]
        INCREMENT node count
    END IF
END FUNCTION
```

Search Algorithm

- Returns the value linked to a specific key.
- Complexity *(Misto, 2024):*
    - Average Case: O(1)
    - Worst Case: O(n), when multiple keys hash to the same bucket resulting in a long chain of nodes.

Pseudocode

```
FUNCTION SEARCH (key)
    SET bucketIndex = GetHashCode(key)
    SET bucket = buckets[bucketIndex]
    IF bucket is not null THEN
        SET node = FIND node with matching key in bucket[bucketIndex]
        IF node is not null THEN
```

```
        RETURN node.Value
      END IF
    END IF
    // If bucket is null or node not found
    THROW KeyNotFoundException OR RETURN default value
END FUNCTION
```

Delete Algorithm

- Removes a key-value if it exists in the hash table.
- Complexity *(Misto, 2024):*
    - AverageCase: O(1)
    - Worst Case: O(n), when traversing a long chain of linked lists.

Pseudocode

```
FUNCTION DELETE (key)
    SET bucketIndex = GetHashCode(key)
    SET bucket = buckets[bucketIndex]
    IF bucket is not null THEN
      IF bucket.Remove(key) is true THEN
        // Node was found and removed
        DECREMENT node count
        RETURN true
      END IF
    END IF
    RETURN false
END FUNCTION
```

Testing

Testing Approach

For the testing of the rental car application, unit testing was selected as the primary testing methodology. Unit testing is a software testing technique in which individual units of an application, such as functions or methods, are tested in isolation to verify that they run as expected. It was implemented due to its effectiveness in identifying bugs early in the development cycle (Robillard, 2022). By testing individual units of code in isolation, errors can be promptly detected and resolved before it can lead to more complex issues. Consequently, the quality of the code is also enhanced, making it more maintainable and reliable.

Agile methodology was also used during development, which emphasises flexibility, iterative planning, and continuous feedback (GeeksForGeeks, 2025). This approach allowed the team to adapt quickly to new requirements, divide our tasks into manageable sprints, and incorporate regular feedback across the project's lifecycle. According to Rasnacis and Berzisa (2017), the success of Agile methodologies – such as Scrum, extreme programming, and Kanban – hinges on the preparedness of the project team. In support of this, Agile practices were thus adapted to suit the different facets of the team, such as size, training, and motivation, to ensure smooth and effective performances across all members.

Summary of test cases

Test Area	Test Type	Test Input(s)	Expected Output	Rationale
carId validation	Positive	Y664-AA69, Z555-BB55	True	Validates accepted carId format.
	Negative	1234-5678, YYYY-9999, XX-1234	False	Catches incorrect carId format.
fuelType validation	Positive	Diesel, Petrol, Electric, Hybrid	True	Accepts valid fuel types.
	Negative	Water, Gasoline123, null value	False	Rejects invalid input or empty values.
year validation	Positive	2000, 2025	True	Accepts realistic years.
	Negative	2027, 1488	False	Detects future years or unrealistic past years.
phoneNumber validation	Positive	12345678, 87654321	True	Accepts exactly 8-digit numbers.
	Negative	12345, 0123456789	False	Rejects numbers of the wrong length.
email format validation	Positive	zook@gmail.com, hello@world.co.uk	True	Accepts standard formats for email validation.
	Negative	flookfailcat.com, flook@@gmail.com	False	Catches invalid email formats.
Rental date validation	Positive (Future)	25-12-2030, 29-02-2028	True	Validates future dates and leap year dates.
	Negative (Past)	01-01-2000, 15-03-2015	False	Detects past dates.
	Invalid format/logic	2025-01-01, 29-02-2025, 12-13-2025	False	Rejects dates with the wrong format and non-leap year logic.
clientId validation	Empty table	-	AB0	Verifies the ID starts at 0 when no clients exist.
	Single entry increment	YZ100	YZ101	Confirms correct incrementation when a single id is present.
	High value handling	YZ100, YZ101, YZ102	YZ103	Tests whether the generator can handle ids over 100
rentalId validation	Empty table	-	R0	Verifies the ID starts at 0 when no rentals exist.
	Single entry increment	R1	R2	Confirms correct incrementation when a single id is present.
	High value handling	R100, R101	R102	Tests whether the generator can handle ids over 100

	Large dataset	R0 – R1000	R1000	Tests whether the generator can handle a large volume of rentals efficiently

Conclusion

The GoCar Rental System was successfully developed to streamline the core functionalities required by a car rental business, including client registration, rental processing, and vehicle administration. The project applied object-oriented programming, custom hash tables, an integrated SQL database, and rigorous testing. This resulted to the creation of a functional and well-structured system.

However, the development was not without its limitations. The most significant issue was the graphical user interface (GUI), which suffered from navigational flexibility – specifically, the inability for users to return to previous screens after making a selection. This was due to insufficient planning during the design phase.

Future projects with similar implementations would benefit from a greater emphasis on early-stage interface prototyping, and algorithmic planning to improve navigation. Integrating more advanced interface technologies such as Windows Forms or web-based frameworks would thus result in a better user experience. Additionally, validation methods would be used on file entries to ensure correct data is read into the database in order to prevent errors.

References

Bhullar, R.K., Pawar, L., Kumar, V. and Anjali, 2016. *A novel prime numbers based hashing technique for minimizing collisions*. In: *2016 2nd International Conference on Next Generation Computing Technologies (NGCT)*. IEEE, pp.522–527. https://doi.org/10.1109/NGCT.2016.7877471

Bible, P.W. and Moser, L., 2023. *Linked lists*. In: *An Open Guide to Data Structures and Algorithms*. Available at: https://pressbooks.palni.org/anopenguidetodatastructuresandalgorithms/chapter/linked-lists/

GeeksforGeeks, 2025. *Introduction to Hashing*. [online] GeeksforGeeks. Available at: https://www.geeksforgeeks.org/introduction-to-hashing-2/?ref=gcse_outind

GeeksforGeeks (2025) What *is Agile Methodology?* Available at: What is Agile Methodology? | GeeksforGeeks (Accessed: 11 April 2025)

Karimov, E., 2020. *Hash Table*. In: E. Karimov, ed. *Data Structures and Algorithms in Swift: Implement Stacks, Queues, Dictionaries, and Lists in Your Apps*. Berkeley, CA: Apress, pp.55–60. Available at: https://doi.org/10.1007/978-1-4842-5769-2_7

Misto, S., 2024. *Efficiency in Hash Table Design: A Study of Collision Resolution Strategies and Performance (*Dissertation). Available at: https://urn.kb.se/resolve?urn=urn:nbn:se:hig:diva-45903

Nielsen, F., 2009. *Linked Lists*. In: *A Concise and Practical Introduction to Programming Algorithms in Java*. London: Springer London, pp.1–23. https://doi.org/10.1007/978-1-84882-339-6_7

Rasnacis, A. and Berzisa, S. (2017) 'Method for Adaptation and Implementation of Agile Project Management Methodology', *Procedia Computer Science*, 104, pp.43–50. Available at: https://doi.org/10.1016/j.procs.2017.01.055 (Accessed: 12 April 2025)

Robillard, M.P. (2022). 'Unit Testing', in *Introduction to Software Design with Java.* Cham: Springer International Publishing, pp. 99 – 124. Available at: https://doi.org/10.1007/978-3-030-97899-0_5 (Accessed: 12 April 2025)

Tapia-Fernández, S., García-García, D. and García-Hernandez, P., 2022. Key concepts, weakness and benchmark on hash table data structures. Algorithms, 15(3), p.100. Available at: https://doi.org/10.3390/a15030100