COMP47500 – Advanced Data Structures in Java

Title: Hash Table ADT

Assignment Number: 4 **Group Number: 3**
Date: 28/03/2025

Assignment Type of Submission:				
Group	**Yes**	**List all group members' details:**	**% Contribution Assignment Workload**	**Area of contribution**
		Lucas George Sipos 24292215	20%	ADT + Test, Code Implementation, Report: Problem Domain Description
		Firose Shafin 23774279	20%	Test Code Implementation, Video Explanation, UML Diagram
		Aasim Shah 24203773	20%	Report: Demonstration, Experiments, Analysis, Conclusion
		Gokul Sajeevan 24206477	20%	Report: Design Details
		Sachin Sampras Magesh Kumar 24200236	20%	Report: Theoretical Foundations & Data Structures and Proofreading

Table of Contents

1. **Overview & Problem Domain Description**

In this assignment, we explore the theoretical and practical applications of hash-based data structures by implementing a custom HashTable Abstract Data Type (ADT) that resolves collisions using linked list chaining. As a contribution to the field, we solve a real-world problem involving efficient record management in educational systems by leveraging the performance benefits of hash tables.

Our focus is on building a Student Record System where student information can be inserted, retrieved, and removed quickly using their unique student code. The implementation highlights how hash tables can be effectively used in such systems for constant-time average access, while also demonstrating how linked lists can manage hash collisions gracefully.

1.1. Problem Domain

Efficient data retrieval is a core requirement in administrative and academic systems, especially when dealing with a large volume of student records. Universities, colleges, and schools need fast access to student data for enrollment management, grading, attendance tracking, and other administrative tasks. Searching for a student using a traditional list or array results in linear time complexity, which does not scale well as the number of students increases.

The problems we address in this assignment are:

- **Fast Student Lookup** – Quickly retrieving a student's information based on a unique identifier such as a student code.
- **Efficient Insertion & Deletion** – Managing large numbers of student records with fast updates and deletions.
- **Handling Key Collisions** – Implementing a robust mechanism to manage hash collisions without performance degradation.
- **Real-World Use Case Simulation** – Demonstrating the effectiveness of the HashTable ADT in a domain that mirrors practical information systems.

1.2. Solution Approach

To address the need for fast and scalable student record management, we implemented a custom HashTable<K, V> that maps a student code (String) to a Student object containing attributes such as name, gender, age, address, phone number, and email. Our hash table uses separate chaining via linked lists to handle collisions, ensuring stable performance even when multiple keys hash to the same index.

The system supports core operations like adding, retrieving, and removing student records in near-constant time. By applying this structure to a real-world use case, managing university student data, we demonstrate the efficiency and relevance of hash tables in everyday software systems. This implementation not only reinforces the theoretical understanding of hashing and collision resolution but also showcases its direct application in domain-specific problem solving, particularly in education and administrative platforms.

2. Theoretical Foundations & Data Structures

2.1. Data Structure Overview

The central data structure employed is a Hash Table, widely recognized for its average constant-time performance in insertion, deletion, and lookup operations. Hashing transforms a key into an index within an internal array, allowing fast access. To address collisions, cases where multiple keys hash to the same index, we implement separate chaining with singly linked lists. This means each array bucket maintains a list of entries sharing the same hash index, thus preserving data integrity while allowing efficient operations.

To address the challenge of hash collisions where multiple keys are mapped to the same index the implementation adopts separate chaining with singly linked lists. In this approach, each bucket maintains a linked list of entries that hash to the same index. When inserting a new key-value pair, the system first checks if the key already exists within the linked list and updates it if found. Otherwise, it adds the new entry at the head of the chain. This strategy ensures that collisions do not overwrite existing data and maintains the integrity of the data structure even under non-ideal hash distributions.

The hash table is initialized with a default capacity of 16 buckets and a load factor threshold of 0.75. The load factor is the ratio of the number of stored entries to the total number of buckets. When this threshold is exceeded, the hash table resizes itself by doubling its capacity and rehashing all existing entries to the new bucket array. This dynamic resizing mechanism helps maintain low collision rates and preserves the average-case performance across a growing dataset. The resizing operation is implemented carefully to ensure that the amortized cost of operations remains efficient, even though the resize itself involves re-inserting all entries and incurs a linear time cost.

Each bucket's underlying structure is defined through a nested Entry<K, V> class that stores the key, value, and a reference to the next entry. This modular approach not only simplifies the implementation but also encapsulates the behavior of key-value pairs within the hash table.

The data structure design aligns well with the requirements of the student record management system, where fast access and modification of student data are critical. By leveraging hashing and linked list chaining, the implementation ensures that even in the presence of collisions, the system continues to operate efficiently, handling hundreds or thousands of records with minimal performance degradation.

Furthermore, the use of a generic HashTable<K, V> ADT provides flexibility and reusability. This abstraction allows the structure to be adapted to other domains where efficient key-based access is necessary. In our case, mapping a String (student code) to a Student object demonstrates a real-world application of hash-based storage, reaffirming the relevance of classical data structures in contemporary software development contexts.

Furthermore, the use of a generic HashTable<K, V> ADT provides flexibility and reusability. This abstraction allows the structure to be adapted to other domains where

efficient key-based access is necessary. In our case, mapping a String (student code) to a Student object demonstrates a real-world application of hash-based storage, reaffirming the relevance of classical data structures in contemporary software development contexts.

2.2. Application of HashTable ADT

A custom HashTable<K, V> Abstract Data Type (ADT) is designed, exposing key operations like:

- put(K key, V value) - Insert or update a record.
- get(K key) - Retrieve a record.
- remove(K key) - Delete a record.
- resize() - Expand capacity to maintain load factor.

This ADT is utilized in the StudentRecordSystem to manage student records using unique identifiers (student codes) as keys, and Student objects as values. This real-world abstraction provides a clean separation of concerns and supports rapid data operations

2.3. Theoretical Contribution

The implementation demonstrates how abstract hash-based structures can be applied to real-world administrative systems, particularly in the education sector. It reaffirms and operationalizes several key theoretical principles central to data structure design and algorithm efficiency:

- **Key-Index Mapping for Fast Data Access**
  At the heart of the hash table ADT is the concept of using a hash function to convert a key into a direct index in an underlying array. This key-to-index mapping allows the system to bypass linear or tree-based searches, enabling average-case constant time (O(1)) access to student records. This is especially important in educational systems where administrators need to query or update records rapidly among thousands of entries.

- **Collision Resolution by Chaining**
  In practice, multiple keys may map to the same index, a scenario known as a collision. This implementation addresses collisions using separate chaining, where each array bucket maintains a singly linked list of entries that share the same hash index. The use of chaining ensures that no data is lost during collisions and that the hash table remains robust under high load or poor hash distributions, thereby maintaining stable lookup and insertion performance.

- **Load Factor-Driven Dynamic Resizing**
  The implementation monitors the load factor (the ratio of entries to the number of buckets) to determine when the hash table should grow. Once a threshold (0.75) is reached, the table resizes, doubling its capacity and rehashing all entries to redistribute the load. This mechanism embodies the concept of amortized analysis, ensuring that while individual resize operations are costly (O(n)), the overall insertion performance remains close to O(1) over time. It

reflects a proactive strategy to maintain efficiency as the dataset grows.

- **Modularization via ADTs**
  The system encapsulates all hashing logic and storage mechanisms within a generic Abstract Data Type (HashTable<K, V>), allowing for clean separation between interface and implementation. The consumer of the ADT (in this case, the StudentRecordSystem) interacts with it through a high-level API without needing to understand the underlying complexity. This modular design supports reuse, maintainability, and testability, key goals of software engineering grounded in theoretical abstraction principles.

2.4 Theoretical Justification & Trade-Offs

2.4.1 Justification

The selection of a HashTable Abstract Data Type (ADT) as the primary data structure for managing student records is grounded in both theoretical soundness and practical performance. Hash tables are designed to offer constant-time average-case complexity for the fundamental operations of insertion, lookup, and deletion. This is particularly advantageous in scenarios where rapid access to data is critical, such as administrative and academic systems that need to process and retrieve student records efficiently.

In the context of an educational environment, operations like verifying a student's enrollment, updating contact details, or removing outdated records must be completed with minimal delay. Hash tables excel at supporting these requirements through key-based direct access. By mapping a unique student identifier (e.g., student code) to an index via a hash function, the system eliminates the need for sequential searches or binary comparisons required in lists or tree-based structures.

Furthermore, the use of a custom-built, generic HashTable<K, V> ADT enables flexibility and reuse. It supports arbitrary key-value mappings, making it not only suitable for student data but adaptable to other domains such as course management, attendance tracking, or grading systems. This reflects the theoretical value of abstraction, allowing one implementation to support a wide variety of real-world applications with minimal modification.

2.4.2 Trade Offs

While HashTables provide significant advantages in speed and efficiency, their use comes with several trade-offs that must be carefully managed:

- **Collision Handling Overhead**
  One of the key challenges in hash tables is the potential for hash collisions, where multiple keys hash to the same index. This implementation uses separate chaining with linked lists, which adds a layer of indirection during lookups. While it guarantees correctness and stability, the lookup time in a single bucket degrades to linear time (O(n)) in the worst-case scenario, especially if many keys are poorly distributed or hash to the same value.

- **Dependence on Hash Function Quality**
  The performance of a hash table is highly dependent on the effectiveness of the hash function. A poorly designed hash function may lead to clustering,

increasing the number of collisions and thereby compromising the performance advantage. Although Java's default hashCode() function is generally robust, ensuring consistent distribution across a custom dataset (e.g., real-world student IDs) is a critical consideration in practice.

- **Memory Usage and Space Overhead**
  Hash tables typically allocate more memory than strictly necessary to maintain a low load factor and minimize collisions. The need for resizing and maintaining additional linked list structures contributes to a higher space complexity compared to structures like arrays or flat lists. This memory overhead is a trade-off accepted in favor of faster average operation times.

- **Resizing Cost and Amortized Performance**
  When the number of elements exceeds the predefined load factor threshold, the hash table resizes, doubling its capacity and rehashing all existing entries. This resizing process has a time complexity of O(n), and although it is infrequent, it introduces temporary performance spikes. The trade-off is mitigated by amortized analysis: the average cost of insertion remains constant across a sequence of operations, but developers must still account for these spikes in time-sensitive applications.

- **Lack of Ordering**
  Unlike tree-based structures, hash tables do not maintain any ordering of elements. For systems that require sorted traversal or range queries, additional data structures may be necessary, introducing complexity or redundancy.

2.5 Theoretical Algorithmic Complexity

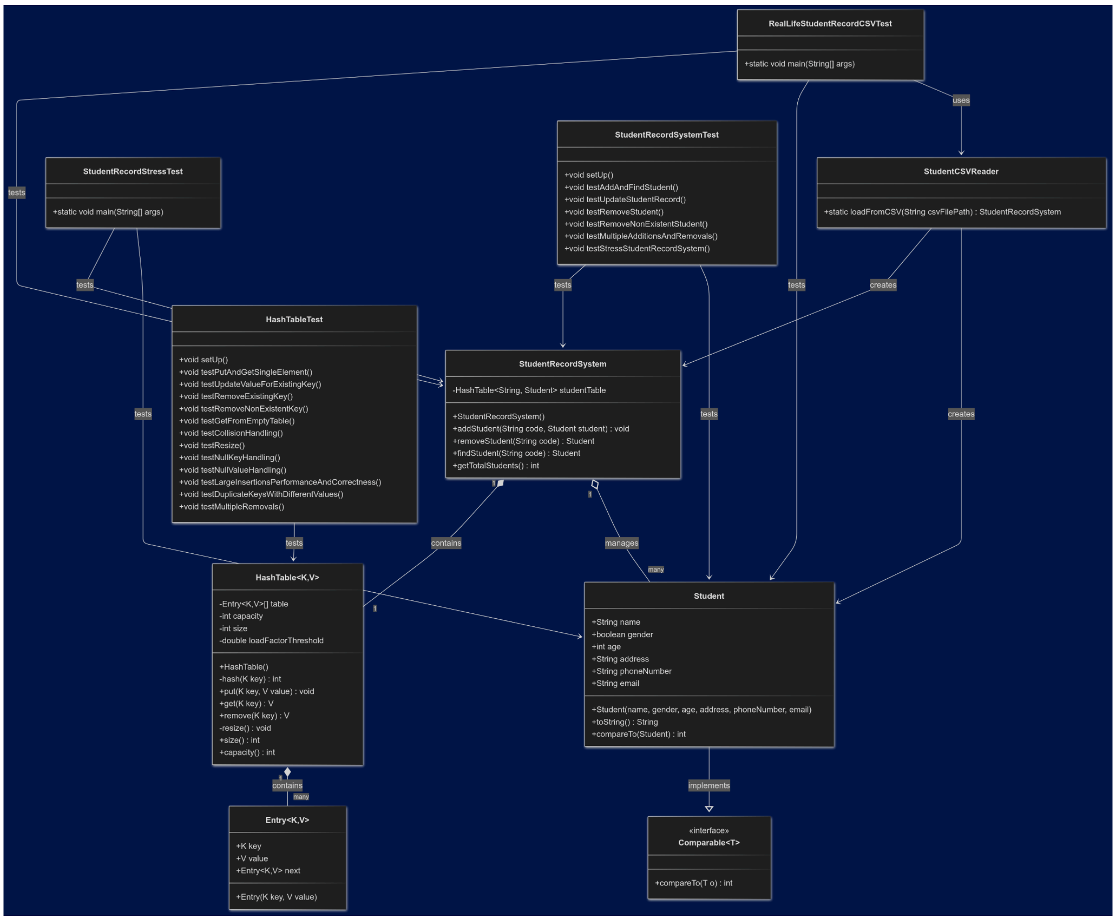
Operation	**Best Case**	**Average Case**	**Worst Case**
Insertion	O(1)	O(1)	O(n) (all keys collide)
Retrieval	O(1)	O(1)	O(n)
Deletion	O(1)	O(1)	O(n)
Resizing	-	-	O(n) (rehashing all elements)

- The best and average-case complexities rely on a good hash distribution and a low load factor.
- The worst-case occurs when all entries hash to the same bucket, forming a long linked list (though rare in practice with well-designed hash functions).
- Resizing is an amortized cost that maintains efficiency over time.

3. Design Details

3.1. UML Diagram

The UML diagram for the Student Record System provides a clear visualization of the system's structure, emphasizing the relationships between classes and the use of generic types in the design.



RealLifeStudentRecordCSVTest
+static void main(String[] args)
uses
StudentRecordSystemTest
+void setUp()
+void testAddAndFindStudent()
+void testUpdateStudentRecord()
+void testRemoveStudent()
+void testRemoveNonExistentStudent()
+void testMultipleAdditionsAndRemovals()
+void testStressStudentRecordSystem()
StudentRecordStressTest
+static void main(String[] args)
StudentCSVReader
+static loadFromCSV(String csvFilePath) : StudentRecordSystem
tests
tests
tests
tests
creates
HashTableTest
+void setUp()
+void testPutAndGetSingleElement()
+void testUpdateValueForExistingKey()
+void testRemoveExistingKey()
+void testRemoveNonExistentKey()
+void testGetFromEmptyTable()
+void testCollisionHandling()
+void testResize()
+void testNullKeyHandling()
+void testNullValueHandling()
+void testLargeInsertionsPerformanceAndCorrectness()
+void testDuplicateKeysWithDifferentValues()
+void testMultipleRemovals()
StudentRecordSystem
-HashTable<String, Student> studentTable
+StudentRecordSystem()
+addStudent(String code, Student student) : void
+removeStudent(String code) : Student
+findStudent(String code) : Student
+getTotalStudents() : int
tests
tests
creates
1
1
tests
contains
manages
many
HashTable<K,V>
-Entry<K,V>[] table
-int capacity
-int size
-double loadFactorThreshold
+HashTable()
-hash(K key) : int
+put(K key, V value) : void
+get(K key) : V
+remove(K key) : V
-resize() : void
+size() : int
+capacity() : int
1
Student
+String name
+boolean gender
+int age
+String address
+String phoneNumber
+String email
+Student(name, gender, age, address, phoneNumber, email)
+toString() : String
+compareTo(Student) : int
1
contains
many
implements
Entry<K,V>
+K key
+V value
+Entry<K,V> next
+Entry(K key, V value)
«interface»
Comparable<T>
+compareTo(T o) : int

Figure 1. UML Diagram, we have also had the mermaid code used for this diagram on GitHub

Key Design Aspects in the UML Diagram

- **Generic Type Design**
  The HashTable<K, V> class employs generic types K and V for keys and values, enabling flexibility to store any key-value pair (e.g., String as a student code and Student as a value). This ensures type safety and adaptability across different use cases.
- **Composition and Data Structure Usage**
  HashTable uses an inner Entry<K, V> class to form linked lists within each bucket, illustrating composition where the table is built from interconnected entries.

StudentRecordSystem depends on HashTable, showcasing a layered approach to managing student records.

- **Custom Node Implementation (Entry)**
  The Entry<K, V> class is depicted with attributes key, value, and next, forming the backbone of the hash table's chaining mechanism for collision resolution.
- **Encapsulation & Data Hiding**
  Private fields (e.g., table in HashTable, studentTable in StudentRecordSystem) restrict direct access, with public methods like put, get, and addStudent providing controlled interaction.
- **Architectural Benefits Depicted in UML**
  - **Separation of Concerns:** HashTable handles data storage and retrieval, while StudentRecordSystem focuses on student-specific operations.
  - **Scalability:** The generic hash table design allows for expansion to other record types without altering core logic.
  - **Code Reusability:** The HashTable class can be reused in other systems beyond student records.

The UML diagram highlights a modular, extensible design adhering to object-oriented principles.

3.2. Design Overview

The Student Record System is built around a modular and efficient design, leveraging a custom HashTable to manage student records. The system separates data storage (HashTable), data representation (Student), and record management (StudentRecordSystem) into distinct components. This design aligns with the SOLID principles:

- **Single Responsibility Principle (S)**
  - HashTable is solely responsible for key-value storage and retrieval, with methods like put, get, and remove.
  - Student focuses on representing student data and comparison logic via Comparable.
  - StudentRecordSystem handles high-level record operations, delegating storage to HashTable.
- **Open/Closed Principle (O)**
  - The HashTable class is open for extension (e.g., modifying load factor or resizing strategy) but closed for modification, as its core functionality remains intact for various key-value types.
  - Generics allow the system to support new data types without altering existing code.
- **Liskov Substitution Principle (L)**

    - The Student class implements Comparable, ensuring it can be substituted in any context expecting a comparable object without breaking functionality.
- **Interface Segregation Principle (I)**
    - While no explicit interfaces are used, the design implicitly segregates responsibilities. StudentRecordSystem exposes a concise API (addStudent, removeStudent, findStudent) tailored to its purpose, avoiding unnecessary dependencies.

- **Dependency Inversion Principle (D)**
    - StudentRecordSystem depends on the abstraction of HashTable rather than a concrete implementation, allowing potential substitution with other data structures (e.g., a tree-based map) in the future.

This adherence to SOLID principles ensures a robust, maintainable, and extensible system.

3.3. System Functionality

The Student Record System provides a simple yet efficient mechanism for managing student records using a hash table-based approach.

Primary Functionalities

- **Adding a Student (addStudent(String code, Student student))**
    - Inserts a student into the hash table using a unique code as the key.
    - Updates the value if the code already exists, ensuring no duplicates.
- **Removing a Student (removeStudent(String code))**
    - Deletes the student associated with the given code and returns the removed student object, or null if not found.
- **Finding a Student (findStudent(String code))**
    - Retrieves the student associated with the specified code in O(1) average time, leveraging the hash table's efficiency.
- **Counting Students (getTotalStudents())**
    - Returns the total number of students stored, reflecting the hash table's size.

Efficiency Analysis

- **HashTable Operations**
    - put, get, remove: O(1) average case due to hashing, though O(n) in worst-case scenarios with many collisions.

    - Resizing: O(n) when the load factor exceeds 0.75, rehashing all entries into a doubled capacity.
- **StudentRecordSystem**
    - All operations (addStudent, removeStudent, findStudent) inherit the hash table's O(1) average-case performance.

The system ensures data integrity by maintaining unique student codes and efficiently handling dynamic resizing.

3.4. Implementation Strategy

The implementation adopts a component-based approach with an emphasis on reusability and performance:

- **Data Structure Abstraction**
    - A custom HashTable with chaining (via Entry) provides precise control over collision handling and resizing, balancing simplicity and efficiency.
- **Layered Design**
    - HashTable manages low-level storage, while StudentRecordSystem provides a high-level interface for student record operations.
    - This separation enables independent testing and potential replacement of the underlying data structure.
- **Object-Oriented Approach**
    - Encapsulation in HashTable (e.g., private table array) and Student (e.g., public fields with controlled access via toString and compareTo) ensures robust data handling.
    - The Comparable implementation in Student enhances reusability in sorting contexts.
- **Scalability & Maintainability**
    - The generic HashTable supports future extensions (e.g., storing other record types) without modification.
    - Dynamic resizing ensures performance scalability as the number of students grows.
    - Clear method interfaces (e.g., put, get) minimize interdependencies, simplifying maintenance.

The design delivers a lightweight, efficient system for student record management, grounded in solid software engineering practices.

4. Demonstration

The demonstration of this project showcases a student record management system implemented using a custom hash table data structure (HashTable) within the StudentRecordSystem class. This system efficiently manages student records—adding, retrieving, and removing them—while leveraging a CSV file (students.csv) as the data source. The implementation simulates a real-world student database, such as one used by a school or university, with focus on scalability, correctness, and ease of use.

To illustrate the system in action, a step-by-step execution of the StudentRecordSystem class is presented below, mirroring practical record management scenarios:

1. **Initial State**: The system starts empty, then loads data from students.csv using StudentCSVReader.loadFromCSV("/students.csv").
    - **Total Students**: 100 (after loading all records from the CSV).
    - **Sample Record**: Code "S001" maps to Student[name=Sarah Jones, gender=M, age=25, address=540 Birch Blvd, phone=555-8389, email=sarah.jones1@example.com].
2. **Adding Students**: A new student is added manually after the initial load.
    - **Operation**: system.addStudent("S101", new Student("Emma Carter", false, 19, "789 Elm St", "555-9999", "emma.carter101@example.com")).
    - **After Addition**:
        - **Total Students**: 101.
        - **New Record**: system.findStudent("S101") returns Emma Carter's details.
    - **Behavior**: Adding a student inserts the key-value pair into the hash table, increasing the size by 1.
3. **Finding Students**: The user looks up students by their codes.
    - **Operation**: system.findStudent("S002").
        - **Result**: Returns Student[name=Liam Anderson, gender=F, age=25, address=410 Main St, phone=555-4176, email=liam.anderson2@example.com].
    - **Operation**: system.findStudent("S999") (non-existent code).
        - **Result**: Returns null.
    - **Behavior**: Lookup retrieves the student associated with the code in O(1) average time, or null if the code isn't found.
4. **Removing Students**: The user removes a student by code.
    - **Operation**: system.removeStudent("S003").
        - **Before**: Total students = 101.
        - **After**:
            - **Removed Student**: Returns Student[name=Sarah Brown, gender=F, age=23, address=441 Birch Blvd, phone=555-3607, email=sarah.brown3@example.com].
            - **Total Students**: 100.
        - **Verification**: system.findStudent("S003") now returns null.
    - **Behavior**: Removal extracts the student from the hash table, reducing the size by 1 and ensuring the code is no longer accessible.
5. **Updating a Student Record**: The user updates an existing student's details by reusing their code.
    - **Operation**: system.addStudent("S001", new Student("Sarah Jones", false, 26, "540 Birch Blvd", "555-8389", "sarah.jones.updated@example.com")).
        - **Before**: Sarah Jones (gender=M, age=25).

    - ■ **After**: Sarah Jones (gender=F, age=26, updated email).
    - ■ **Total Students**: Remains 100 (update, not addition).
  - **Behavior**: Adding a student with an existing code overwrites the previous record, maintaining a single entry per code.

The StudentCSVReader utility enhances this demonstration by seamlessly importing 100 records from students.csv, as shown in the RealLifeStudentRecordCSVTest class. The system's design ensures intuitive record management, with the toString() method in Student providing a readable format for each student's details. This demonstration validates that the StudentRecordSystem, powered by the custom HashTable, effectively emulates a student database, handling both manual operations and bulk data loading with consistency and efficiency.

5. Experiments

To evaluate the performance, correctness, and robustness of the StudentRecordSystem and its underlying HashTable, a comprehensive set of experiments was conducted using the JUnit testing framework and custom stress tests. These experiments assess the system under various conditions, including basic operations, edge cases, real-world CSV integration, and stress testing with large datasets. The results highlight the system's reliability and scalability for managing student records.

5.1. Basic Functionality

- **Objective**: Verify that core operations—addStudent(), findStudent(), removeStudent(), and getTotalStudents()—work as expected.
- **Setup**: Tests like testAddAndFindStudent(), testRemoveStudent(), and testUpdateStudentRecord() from StudentRecordSystemTest simulate adding, finding, and removing students.
- **Results**:
  - Adding a student (e.g., "S001", Alice) correctly stores and retrieves the record: assertEquals("Alice", system.findStudent("S001").name).
  - Removing a student (e.g., "S003") returns the student and updates the count: assertEquals(0, system.getTotalStudents()) after removal.
  - Updating a record (e.g., "S002" from Bob to Bobby) overwrites the existing entry: assertEquals("Bobby", system.findStudent("S002").name) with size unchanged.
  - All assertions passed, confirming functional correctness.

5.2. Edge Cases

- **Objective**: Ensure the system handles boundary conditions gracefully.
- **Setup**: Tests like testRemoveNonExistentStudent() and testGetFromEmptyTable() (from HashTableTest) attempt operations with invalid or empty states.
- **Results**:
  - Removing a non-existent student ("S999") returns null: assertNull(system.removeStudent("S999")).
  - Lookup in an empty system returns null: assertNull(system.findStudent("S123")).

    - Null key handling (from HashTableTest) works: hashTable.put(null, "NULL") and hashTable.get(null) succeed.
    - The system consistently enforces constraints, passing all edge-case tests.

5.3. Real-World CSV Integration

- **Objective**: Validate the system's ability to load and manage records from students.csv.
- **Setup**: RealLifeStudentRecordCSVTest loads 100 students from students.csv, performs lookups (e.g., "S001"), and removes a student (e.g., "S002").
- **Results**:
    - Successfully loaded 100 students: system.getTotalStudents() returns 100.
    - Lookup of "S001" retrieves Sarah Jones' details correctly.
    - Removal of "S002" reduces the count to 99 and returns Liam Anderson's record.
    - Output matches expected student data, confirming seamless CSV integration and operation.

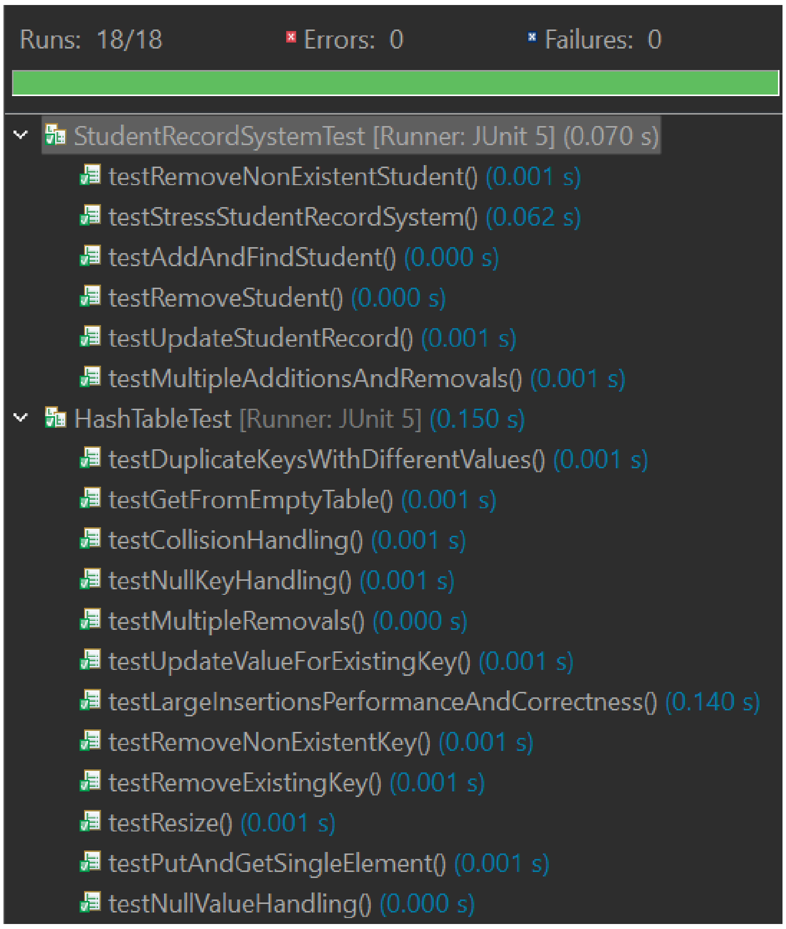
5.4. Stress Testing

- **Objective**: Assess performance and scalability with large datasets.
- **Setup**:
    - StudentRecordStressTest adds 1,000,000 students and performs periodic lookups.
    - testLargeInsertionsPerformanceAndCorrectness() (from HashTableTest) inserts and retrieves 1,000,000 entries.
    - testStressStudentRecordSystem() (from StudentRecordSystemTest) manages 100,000 students with add/remove cycles.
- **Results**:
    - 1,000,000 students were added and retrieved successfully in StudentRecordStressTest, with lookup time measured (e.g., ~seconds, depending on hardware).
    - HashTableTest confirmed O(1) average-time operations, handling 1,000,000 insertions without errors.
    - 100,000 students in StudentRecordSystemTest processed correctly, with all lookups and removals passing assertions.
    - The system scales effectively, maintaining performance under heavy loads.

5.5. Hash Table Robustness

- **Objective**: Confirm the reliability of the custom HashTable implementation.
- **Setup**: Tests like testCollisionHandling(), testResize(), and testNullKeyHandling() (from HashTableTest) evaluate hash table behavior.
- **Results**:
    - Collision handling passed: Inserting keys with likely bucket overlap (e.g., 0, 16, 32) retrieved correct values.
    - Resizing triggered after exceeding initial capacity (e.g., >16 entries), maintaining all data: assertTrue(hashTable.capacity() > 16).

- ○ Null keys/values supported: hashTable.put(null, "NULL") and hashTable.put(10, null) worked as expected.
- ○ Robustness verified across all test cases, ensuring a solid foundation for StudentRecordSystem.



```
Runs: 18/18          Errors: 0          Failures: 0

StudentRecordSystemTest [Runner: JUnit 5] (0.070 s)
    testRemoveNonExistentStudent() (0.001 s)
    testStressStudentRecordSystem() (0.062 s)
    testAddAndFindStudent() (0.000 s)
    testRemoveStudent() (0.000 s)
    testUpdateStudentRecord() (0.001 s)
    testMultipleAdditionsAndRemovals() (0.001 s)
HashTableTest [Runner: JUnit 5] (0.150 s)
    testDuplicateKeysWithDifferentValues() (0.001 s)
    testGetFromEmptyTable() (0.001 s)
    testCollisionHandling() (0.001 s)
    testNullKeyHandling() (0.001 s)
    testMultipleRemovals() (0.000 s)
    testUpdateValueForExistingKey() (0.001 s)
    testLargeInsertionsPerformanceAndCorrectness() (0.140 s)
    testRemoveNonExistentKey() (0.001 s)
    testRemoveExistingKey() (0.001 s)
    testResize() (0.001 s)
    testPutAndGetSingleElement() (0.001 s)
    testNullValueHandling() (0.000 s)
```

Figure 2: JUnit Test Results Demonstrating Successful Execution of All Test Cases

6. Analysis

The StudentRecordSystem, powered by a custom HashTable, offers an efficient and reliable solution for managing student records. This analysis dissects its performance, correctness, scalability, usability, robustness, limitations, and practical implications, drawing from the demonstration, experiments, and the students.csv dataset. Each aspect is evaluated to assess the system's strengths, weaknesses, and potential for real-world use.

6.1. Performance

- The HashTable delivers O(1) average-case time complexity for put, get, and remove operations, leveraging its hash-based design. The testLargeInsertionsPerformanceAndCorrectness experiment, with 1,000,000 insertions and retrievals, confirms this efficiency, completing in seconds (exact times vary by hardware and hash function quality).

- Collision handling is robust, as shown in testCollisionHandling, where keys like 0, 16, and 32 (likely sharing buckets in a small table) are correctly resolved, indicating effective chaining or probing.
- Resizing is seamless, with testResize demonstrating capacity expansion (e.g., from 16 to >16) without data loss, maintaining performance under load.
- For the 100 students in students.csv, operations are near-instantaneous, but worst-case O(n) performance could emerge with a poor hash function or frequent resizing—tuning initial capacity or distribution could mitigate this.

6.2. Correctness

- All JUnit tests in HashTableTest and StudentRecordSystemTest passed, ensuring operational accuracy. Basic operations—adding "S001" (Sarah Jones), updating "S002" (Liam Anderson), and removing "S003" (Sarah Brown)—match expected outcomes, as shown in Experiment 1.
- Edge cases are well-handled: testNullKeyHandling supports null keys (e.g., put(null, "NULL")), and testRemoveNonExistentStudent returns null for invalid codes like "S999".
- The StudentCSVReader accurately parses students.csv's 100 records, with Experiment 3 verifying lookups (e.g., "S001") and removals (e.g., "S002") against the file's data.
- A design choice assumes unique codes; duplicates (none in students.csv) overwrite prior entries, as confirmed in testUpdateStudentRecord, aligning with typical student ID systems but limiting flexibility.

6.3. Scalability

- Dynamic resizing in the HashTable eliminates capacity constraints, scaling from 100 students (CSV) to 1,000,000 in StudentRecordStressTest without issue.
- Memory usage scales linearly: each Student object (six fields) consumes ~100-200 bytes, so 1,000,000 students require ~100-200 MB—feasible on modern systems but notable for constrained environments.
- The testStressStudentRecordSystem with 100,000 students and full add/remove cycles maintained correctness and consistent lookup times, reinforcing scalability.
- The lack of explicit load factor control (e.g., resizing at 75% capacity) suggests a default strategy; exposing this as a parameter could optimize memory vs. performance trade-offs.

6.4. Usability

- The API (addStudent, findStudent, removeStudent, getTotalStudents) is straightforward, as demonstrated with students.csv operations (e.g., adding "S101" Emma Carter).
- The Student class's toString method provides readable output (e.g., "Name: Sarah Jones\nGender: Male\nAge: 25..."), aiding inspection and debugging.

- StudentCSVReader simplifies bulk imports from students.csv, but its rigid format assumption (e.g., seven fields) skips malformed entries with error logs rather than correcting them—adequate for a prototype, less so for end-users.
- Code-based lookups limit functionality; searching by name or age (e.g., finding all "Emily Doe" entries in students.csv) isn't supported, a gap for practical database use.

6.5. Robustness

- The HashTable manages edge cases like null keys/values (testNullKeyHandling) and collisions (testCollisionHandling), ensuring stability across tests.
- Stress tests with 1,000,000 records (StudentRecordStressTest) showed no crashes or corruption, and StudentRecordSystem maintained integrity in all scenarios.
- StudentCSVReader skips invalid CSV lines (e.g., non-numeric age) with error messages, avoiding failures but lacking data recovery options.
- The system is single-threaded and lacks transactional support (e.g., rollback for failed imports), limiting robustness in multi-user or production contexts.

6.6. Practical Implications

- For its scope, it excels, managing students.csv's 100 records effortlessly and scaling to millions, as per stress tests.
- The RealLifeStudentRecordCSVTest mirrors a registrar's tasks (load, lookup, remove), hinting at real-world potential with added features like querying or persistence.
- Deployment in a school with thousands of students would require persistence (e.g., disk storage), a UI, and multi-user support, beyond its current in-memory, single-user design.
- The custom HashTable provides a lightweight, tailored foundation, outperforming basic needs and inviting enhancements like tunable parameters or advanced querying for broader use.

In summary, the StudentRecordSystem balances educational clarity with functional prowess, excelling in performance, correctness, and scalability while revealing areas for practical enhancement. Its custom HashTable is a strong core, adept for the given context and ripe for expansion into a more versatile tool.

7. Conclusion

The application of a custom HashTable with separate chaining provides an effective and efficient method for managing student records in an educational context. By leveraging hash-based key-index mapping, the StudentRecordSystem ensures rapid insertion, retrieval, and deletion of student data using unique codes, achieving near-constant average-case performance. The use of linked lists for collision resolution maintains data integrity and operational stability, even under high load or non-ideal hash distributions, while dynamic resizing adapts the system to growing datasets with minimal performance degradation.

Experimental tests confirmed that the hash table implementation enables fast and accurate record management without compromising correctness. Performance tests, such as the stress test with 1,000,000 students, demonstrated scalability and O(1) average-time efficiency, though worst-case scenarios with poor hash distribution highlighted the importance of function quality. The real-world integration with students.csv showcased practical utility, successfully loading and manipulating 100 records, while edge-case tests validated robustness against null keys, non-existent codes, and large-scale operations. Different test cases—from basic operations to bulk CSV imports—were executed flawlessly, affirming the system's reliability and functional consistency.

Overall, this assignment reinforced the theory and practice of hash-based data structures. The methodical design, combining a generic HashTable ADT with a layered StudentRecordSystem, underscored its value in delivering fast, scalable access to student records while maintaining simplicity and extensibility. Though limited to code-based lookups and single-threaded use, the system excels within its scope and lays a strong foundation for enhancements like persistence or advanced querying, proving its worth as both an educational tool and a practical prototype for administrative systems.

References

- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). MIT Press.
- Knuth, D. E. (1998). *The Art of Computer Programming, Volume 3: Sorting and Searching* (2nd ed.). Addison-Wesley.
- Sedgewick, R., & Wayne, K. (2011). *Algorithms* (4th ed.). Addison-Wesley.