



# INF1002 Programming Fundamentals C Project

Project Name: Class Management System (CMS)

Date of submission: 25/11/2025

Team: P4-1

Team Member Name	Student ID	Emails
Chew Shu Wen	2503112	2503112@sit.singaporetech.edu.sg
Adora Goh Shao Qi	2502513	2502513@sit.singaporetech.edu.sg
Calson See Jia Jun	2502169	2502169@sit.singaporetech.edu.sg
Au Myat Yupar Aung	2502143	2502143@sit.singaporetech.edu.sg
Chung Kai Sheng Desmond	2502081	2502081@sit.singaporetech.edu.sg

# Table of Contents

<b>Table of Contents</b>	<b>2</b>
<b>1. Introduction</b>	<b>3</b>
<b>2. System Design and Implementation</b>	<b>3</b>
2.1 Overall Design/Program Flow	3
2.2 Data Structure Design	4
2.3 Key Functions	6
2.3.1 Open Database (openDB / loadDB)	6
2.3.2 Show Records (showDB)	7
2.3.3 Insert Record (insertDB)	7
2.3.4 Query Record (queryDB)	8
2.3.5 Update Record (updateDB)	8
2.3.6 Delete Record (deleteDB)	9
2.3.7 Save Record (saveDB)	10
2.4 Enhancement Features	10
2.4.1 Show Records sorted by ID or Mark (showDBsorted)	10
2.4.2 Show Summary of Records (showSummary)	11
2.4.3 Restore (restoreDB)	11
2.4.4 Undo / Redo (undo/redo)	12
<b>3. Test Cases and Validation</b>	<b>12</b>
3.1 Reflections from Testing	12
3.2 Test Table	13
<b>4. Team Contributions</b>	<b>16</b>
<b>Appendix</b>	<b>18</b>

# 1. Introduction

The Class Management System (CMS) is a command-line program written in C that manages student records stored in a text file. It was created to demonstrate core programming skills such as file handling, memory management, input validation, and structured program design.

The system allows users to load a file, add records, update existing entries, delete records, view stored data, and save changes. Commands follow a structured format, and the system validates inputs to prevent incorrect or incomplete data from being processed.

A key enhancement in this system is the Undo/Redo feature. It allows users to reverse or reapply recent actions such as insert, update, delete, and restore. This improves usability and prevents accidental data loss.

The goal of the project was to build a functional, reliable, and organised system that supports record management through a text-based interface. The scope covers command processing, data validation, persistent storage, and additional safety features, while keeping the program modular for future improvements.

# 2. System Design and Implementation

## 2.1 Overall Design/Program Flow

The CMS operates as a continuous command loop in **main()**, running until the **QUIT** command is entered. User input is captured using **readLine()** from **input\_validation.c**, which dynamically allocates memory to support variable-length inputs. The input is then processed through **validateCommand()** to identify supported commands (e.g., **OPEN**, **INSERT**, **UPDATE**, **SAVE**, etc.). Commands requiring additional arguments are further parsed using **parseCommand()** or **handleShow()** before being executed by the relevant database function. After each command completes, the system outputs a status message, cleans temporary memory, and returns to the loop for the next user instruction.

The program is structured into three main modules, illustrated in Figure 1, which highlights their relationships:

1. **Main Application Loop (main.c):** Manages initialization, command routing, and cleanup via **freeDB()**, interfacing with both the database and input modules.
2. **Input and Parsing (input\_validation.c):** Handles dynamic input, command validation, and parameter parsing to ensure all values passed to the system are safe and well-formatted.
3. **Core Database Management (cms.c):** Implements operations including file I/O (**loadDB**, **saveDB**), undo/redo, and additional features, using hybrid data structures for efficient record storage and lookup.

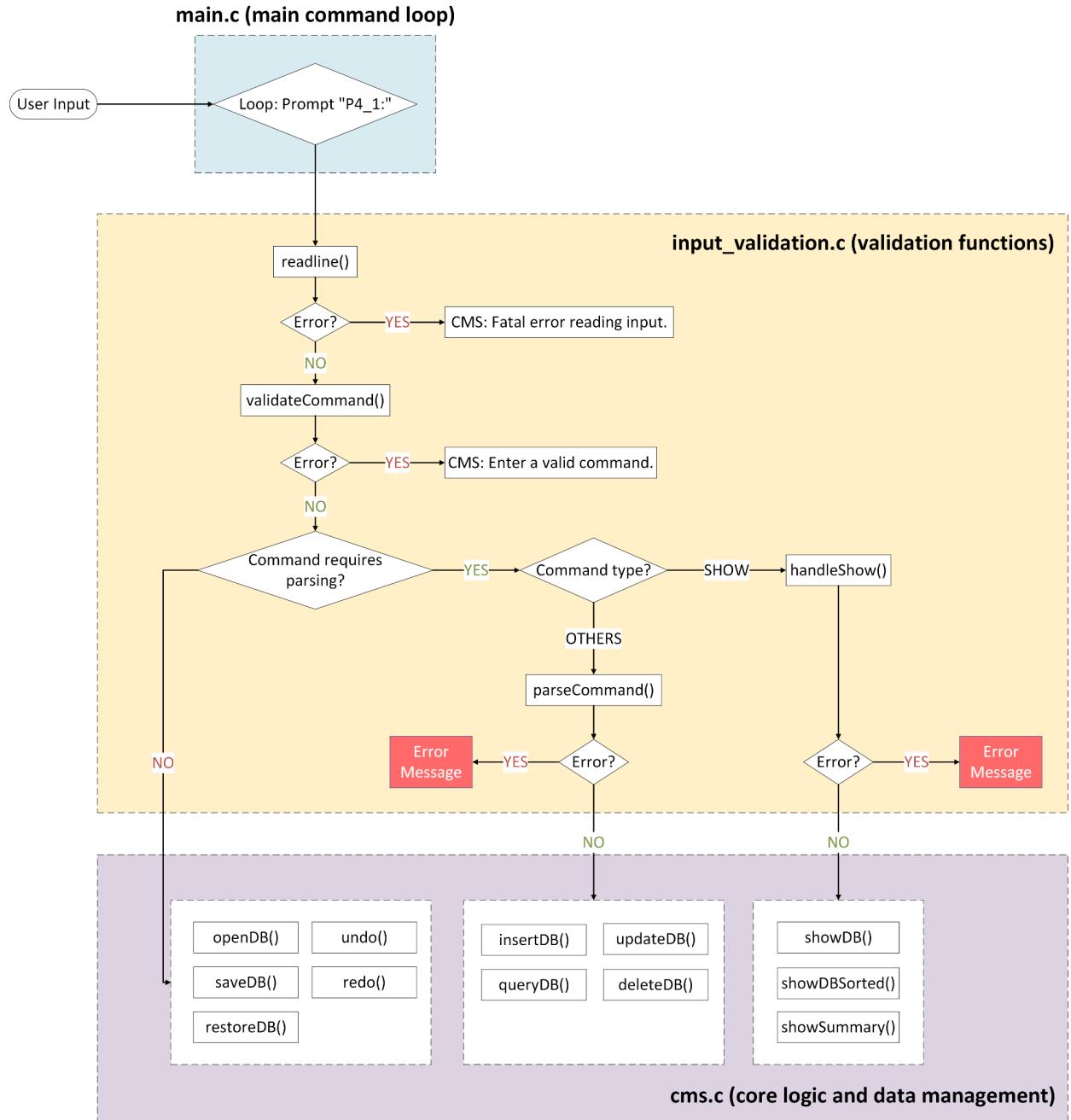


Figure 1 (System Design)

## 2.2 Data Structure Design

The CMS employs a hybrid data structure approach, combining a linked list for ordered storage with a hash table for fast record lookup. This design ensures efficient insertion, deletion, and querying while supporting ordered traversal and advanced features like undo/redo and sorting.

## Student Record Structure

```
typedef struct StudentRecord {  
    int id;  
    char name[MAX_NAME];  
    char programme[MAX_PROGRAMME];  
    float mark;  
} StudentRecord;
```

This structure directly represents a student entry, making the system easy to understand and maintain. **Fixed-size character arrays** were chosen to avoid the complexity of dynamic string allocation and to reduce risks such as memory leaks and fragmentation. More flexible alternatives like dynamically allocated strings or key-value storage were rejected because they add unnecessary overhead and complicate numeric comparisons (e.g., **sorting marks**).

## Linked List Node and Hash Table Integration

```
typedef struct Node {  
    StudentRecord data;  
    struct Node *next;  
    struct Node *hashNext;  
} Node;
```

Each record is stored in a Node, which supports two forms of access:

- next enables traversal as a standard linked list, supporting ordered operations like SHOW ALL.
- hashNext supports separate chaining in the hash table for fast QUERY, UPDATE, and DELETE operations.

Other options like arrays would waste space or require resizing, while a pure hash table would make sequential or sorted output difficult while this hybrid design balances efficiency and usability.

## Hash Table

```
Node *hashTable[TABLE_SIZE] = {0};
```

The hash table provides **O(1)** average lookup time using ID-based hashing. Separate chaining simplifies collision handling and supports frequent insert/delete operations without rehashing. Open addressing was considered but rejected due to more complex deletion rules and inability to maintain list order.

## Action Structure for Undo/Redo

```
typedef struct Action {  
    ActionType type;
```

```
StudentRecord oldData;
StudentRecord newData;
struct Action *next;
} Action;
```

**Undo/redo** is implemented with stack behavior (**LIFO**). Each action stores both old and new record states, allowing accurate reversal of insert, update, delete, and restore operations. Full database snapshots were rejected due to high memory cost, and storing only operation types was insufficient to restore multi-field records.

## 2.3 Key Functions

Before execution, every command goes through multiple processing stages to ensure safety, correctness, and consistency.

### **Input Handling (readLine())**

User input is first captured using the `readLine()` function, which reads characters dynamically from `stdin`. Unlike fixed-size input functions (e.g., `scanf()` or `gets()`), `readLine()` expands its buffer automatically using `realloc()`, preventing buffer overflow and allowing variable-length input.

Key behaviour:

- Grows dynamically when input exceeds buffer size
- Stops at newline (\n) or EOF
- Ensures null-termination and checks for allocation failures

### **Command Validation (validateCommand)**

Once input is captured, it is passed to `validateCommand()`. This stage verifies:

- The command keyword is valid and case-insensitive
- Leading/trailing whitespace is ignored
- No unsupported or extraneous arguments are included

This early validation prevents malformed commands from reaching core logic. Further argument-level checks are handled by downstream functions such as `parseCommand()`, `handleShow()`, and helper validators (`validateID()`, `validateMark()`, `toTitleCase()`).

### 2.3.1 Open Database (openDB / loadDB)

The `OPEN` operation initializes the CMS by loading the database from persistent storage into memory. It ensures that subsequent operations such as `INSERT`, `UPDATE`, `DELETE`, or `QUERY` can access the existing student records efficiently. `openDB()` serves as the user-facing interface, while `loadDB()` performs the actual reading and parsing of the database file.

Logical Flow:

1. openDB (which internally calls loadDB) first checks whether a database has already been loaded to prevent redundant loading.
2. If not, it calls loadDB with the filename of the database. Inside loadDB, any pre-existing in-memory database is cleared, including freeing memory of nodes and resetting hash table entries.
3. The function opens the specified file and reads it line by line.
4. For each record line, it splits the data fields based on tab delimiters, converts them into their appropriate data types, and creates a new linked list node.
5. Each node is appended to the main linked list and simultaneously inserted into a hash table to enable O(1) lookups by student ID.
6. Finally, the function flags the database as loaded and prints a success message.

Validation:

1. Checks whether the database file exists and is accessible before attempting to open.
2. Any failure to open the file (missing, permission denied) is handled with an error message.
3. Memory allocation for linked list nodes is validated; failure triggers an error and aborts the load.
4. Each record is parsed:
  - a. ID is converted to an integer and validated with **validateID()**.
  - b. Name and programme default to empty strings if missing.
  - c. Mark defaults to 0.0 if missing, otherwise validated numerically within 0–100 via **validateMark()**.
5. Duplicate IDs checked via **findNode()** when populating the hash table.

### 2.3.2 Show Records (showDB)

Simply prints all records in the linked list in insertion order.

Logical Flow:

1. Checks if the linked list is empty.
2. If there are no records, it prints a message indicating that there are no records to display. Otherwise, it traverses the list starting from head, visiting each node sequentially.
3. For each node, it prints the stored data (ID, name, programme, mark) in a tabular format.
4. The function uses **printNodeList()** to dynamically calculate column widths based on the longest name or programme string to ensure proper alignment.

### 2.3.3 Insert Record (insertDB)

This function adds a new student record to the database, both in the linked list and the hash table for efficient lookups.

Parameters:

- **newID:** The 7-digit identifier of the student.

- **newName**: Student's name (optional, can be empty).
- **newProgramme**: Student's programme (optional, can be empty).
- **newMark**: Student's mark (optional, defaults to 0.0 if empty).
- **isUndoRedo**: Flag to indicate if the operation is part of an undo/redo action.

Logical Flow:

1. The function first checks if a node with the same ID exists via **findNode()**.
2. If not, a new Node is allocated and populated with the provided data.
3. The node is appended to the linked list tail to preserve order and inserted into the hash table for future retrieval.
4. If this is a standard insertion (not undo/redo), an action record is pushed onto the undo stack to enable reversal of the operation.
5. Finally, the database is marked as modified.

Validation

1. All inputs are validated through **parseCommand()** before **insertDB()** is called.
2. ID: Checked with **validateID()** (7 digits starting with '2').
3. **parseCommand()** ensures fields (ID/ NAME/ PROGRAMME/ MARK) are optional and can be empty (**OPTIONAL\_ALLOWED\_EMPTY**).
4. Name/Programme: Optional field, length limited to **MAX\_NAME/MAX\_PROGRAMME**, converted to title case.
5. Mark: Optional field, validated with **validateMark()**, numeric 0-100.
6. Duplicate IDs checked via **findNode()** to prevent insertion conflicts.
7. Memory allocation for new nodes is verified else failure prevents insertion.

### 2.3.4 Query Record (queryDB)

To retrieve and display a specific student record by ID.

Parameters:

- **id**: The 7-digit student ID to look up.

Logical Flow:

1. If the record exists, it prints the student's ID, Name, Programme, and Mark in a tabular format.
2. Otherwise, it notifies the user that the record does not exist.

Validation:

1. ID: Checked with **validateID()** (7 digits starting with '2').
2. **parseCommand()** ensures no extra fields are provided (**OPTIONAL\_NONE**).
3. **findNode()** ensures the record exists before displaying it.

### 2.3.5 Update Record (updateDB)

To modify one or more fields of an existing student record.

Parameters:

- **id**: ID of the record to update.
- **name**: Optional new name.
- **programme**: Optional new programme.
- **mark**: Optional new mark.
- **isUndoRedo**: Flag indicating if this update is part of undo/redo.

Logical Flow:

1. The function locates the record by ID.
2. If found, it selectively updates each field only if the new data is provided.

Validation:

1. All inputs are validated through **parseCommand()** before insertDB() is called.
2. ID: Checked with **validateID()** (7 digits starting with '2').
3. At least one of name, programme, or mark must be provided (**OPTIONAL\_REQUIRED**).
4. Name/Programme: length limited to **MAX\_NAME/MAX\_PROGRAMME**, converted to title case.
5. Mark: validated with **validateMark()**, numeric 0-100.
6. Duplicate IDs checked via **findNode()** to prevent insertion conflicts.
7. Memory allocation for new nodes is verified else failure prevents update.

### 2.3.6 Delete Record (deleteDB)

To remove a student record from both the linked list and hash table.

Parameters:

- **id**: The ID of the record to delete.
- **confirm**: A flag to indicate whether user confirmation is required.
- **isUndoRedo**: Whether the deletion is part of undo/redo.

Logical Flow:

1. The function iterates through the linked list to locate the node.
2. Upon finding it, confirmation is optionally requested.
3. The node is unlinked from both the list and hash table, memory is freed, and an undo action is pushed if applicable.
4. The tail pointer is updated if the deleted node was at the end of the list.

Validation:

1. ID: Checked with **validateID()** (7 digits starting with '2').
2. **parseCommand()** ensures no extra fields are provided (**OPTIONAL\_NONE**).
3. **findNode()** ensures the record exists before displaying it.
4. Optional confirmation input validated with **validateCommand()** for "Y" or "N".

### 2.3.7 Save Record (saveDB)

To persist the in-memory database to disk, ensuring changes are not lost.

Logical Flow:

1. Before overwriting the main database file, a backup is created (**P4\_1-CMS.bak**).
2. The system compares the current in-memory database with the existing file content.  
If both versions match, meaning no new edits have been made, no write occurs
3. The main database file is opened in write mode, and all nodes in the linked list are written sequentially.
4. Fields are separated by tabs, preserving the original structure.
5. After writing, the file is closed and the dbModified flag is reset.

Validation:

1. Checks that the database is loaded.
2. Verifies that the file can be opened for writing.
3. Handles potential write errors and ensures proper file closure.
4. Backup file creation (**P4\_1-CMS.bak**) is checked for successful write.

## 2.4 Enhancement Features

### 2.4.1 Show Records sorted by ID or Mark (showDBsorted)

Displays all records sorted either by ID or by mark, in ascending or descending order.

Parameters:

- **sortByID (boolean):** Determines whether sorting is based on the student ID (true) or the mark (false).
- **ascending (boolean):** Determines whether the sort order is ascending (true) or descending (false).

Logical Flow:

1. First creates a clone of the linked list (**Node\* cloneList(Node\* original)**) so that the original order remains intact.
2. It then applies a merge sort on the cloned list, using either ID or mark as the sorting key based on sortByID.
3. During sorting, it compares nodes' values and arranges them in ascending or descending order as indicated by the ascending parameter.
4. After sorting, it traverses the sorted list and prints each record in a formatted table, similar to showDB().
5. Once printing is complete, the cloned list is freed to avoid memory leaks.
6. This function operates on a copy of the data and does not modify the original linked list.

Validation:

1. **handleShow()** verifies that the sort field is either **ID** or **MARK**.

2. Sort order must be **ASC** or **DESC** (any other value is rejected).
3. Any extra trailing arguments are flagged as invalid.
4. Sorting parameters are checked before calling `showDBSorted()`.

### 2.4.2 Show Summary of Records (showSummary)

Displays statistical summaries of the records, including count, average, minimum, and maximum marks, and optionally lists top and bottom performers. Can filter results by a specific programme.

Parameters:

- **programmeFilter (string or NULL):** If a programme name is provided, only records matching that programme are included; otherwise, all records are considered.

Logical Flow:

1. `showSummary()` traverses the linked list, either considering all nodes or filtering nodes by the provided `programmeFilter`.
2. As it visits each node, it aggregates data and counts the number of matching records, sums the marks for average calculation, and updates minimum and maximum values.
3. Optionally, it identifies the records corresponding to the highest and lowest marks.
4. Once traversal is complete, it calculates the average mark and prints the summary statistics in a clear format, optionally highlighting the top and bottom performers.
5. This function reads the list but does not alter it.

Validation:

1. **handleShow()** parses optional programme filter.
2. Programme filter length is checked to ensure it does not exceed **MAX\_PROGRAMME**.
3. Unknown filter keys are rejected.
4. Empty or whitespace-only filters are treated as no filter.

### 2.4.3 Restore (restoreDB)

Restores the database to the state of the last backup file.

Parameters:

- **isUndoRedo:** Indicates if the operation is part of undo/redo.

Logical Flow:

1. The function checks if a backup exists.
2. If so, it loads the backup file into memory, replacing the current linked list and hash table.
3. The operation is optionally pushed to the undo stack.

Validation:

1. Optional confirmation input validated with **validateCommand()** for "Y" or "N".
2. Checks the existence of the backup file before restoring.

3. Memory allocation for new nodes is validated.

#### 2.4.4 Undo / Redo (undo/redo)

Allows reversing or reapplying the most recent action for INSERT, UPDATE, DELETE, or RESTORE.

Logical Flow:

1. undo() pops the top action from the undo stack and performs the inverse operation.
2. redo() pops from the redo stack and reapplies the operation.
3. Both functions update the respective stacks to allow multiple consecutive undo/redo operations.

Validation:

1. Stacks (undoStack and redoStack) are checked for emptiness before any action.
2. Individual undo/redo operations validate that the action can be safely reversed/applied.

## 3. Test Cases and Validation

### 3.1 Reflections from Testing

Testing exposed several issues across the system, all of which were resolved to improve stability and usability. Early input handling failures occurred when users entered additional whitespace around fields (**e.g., Programme=**), causing commands to process incorrectly. This was resolved by consistently trimming and normalizing all user input.

The UNDO/REDO mechanism also surfaced issues, particularly with memory handling. A mismanaged pointer referencing `action->newData.id` for the delete operation produced garbage output such as: CMS: **UNDO → Undid DELETE (ID -1163005939)**. This was corrected by properly initializing data structures and ensuring pointers referenced valid memory.

Sorting initially mutated the main linked list, which unintentionally affected the hash table state. To prevent structural corruption, sorting was refactored to operate on deep copies instead. Minor formatting inconsistencies were also discovered, showDBSorted displayed long text fields incorrectly compared to showDB, and error messages were refined for clarity and consistency.

Another important finding involved the SAVE function. Originally, every save operation overwrote both the active database and the backup file, even when no changes had occurred. This rendered the backup redundant and undermined the RESTORE feature. The logic was updated so the system compares in-memory data with existing stored content and skips writing if no modifications are detected.

Overall, the testing phase not only identified key defects but also strengthened data integrity, user experience, and confidence in the system's operational reliability.

### 3.2 Test Table

Test Case ID	Description	Input Command(s)	Expected Output	Reason for Test	Actual Result
TC001	Show all without opening DB	SHOW ALL	No records loaded. Open and load the database first.	Test command behavior when DB not loaded	Pass
TC002	Open database	OPEN	The database file "P4_1-CMS.txt" is successfully opened.	Test initialization and loading	Pass
TC003	Show all records	SHOW ALL	Here are all the records found in the table "StudentRecords"	Test normal SHOW	Pass
TC004	Insert valid record	INSERT ID=2000001 Name=John Doe Programme=Computer Science Mark=88.0	Record with ID=2000001 inserted.	Test normal insertion	Pass
TC005	Insert with missing optional fields	INSERT ID=2000003 programme=name=mark=	Record with ID=2000003 inserted.	Test optional field handling	Pass
TC006	Insert duplicate ID	INSERT ID=2000001 Name=Other Programme=CS Mark=60	Record with ID=2000001 already exists.	Test duplicate prevention	Pass
TC007	Insert invalid ID	INSERT ID=2000@04 Name=A Programme=B Mark=50	Invalid command. ID must be 7 digits starting with '2'.	Test ID validation	Pass
TC008	Insert name/programme with special chars	INSERT ID=2000005 Name=!\$ %& /() =?* '<> # ; ^23~ @` `Programme=!\$ %& /() =?* '<> # ; ^23~ @` ``	Record with ID=2000005 inserted.	Test special character handling	Pass
TC009	Name/Programme too long	INSERT ID=2000006 Name=abc def ghi jkl mno pqrs tuv wxyz ABC DEF GHI JKL MNO PQRS TUV WXYZ !"§ \$%& /() =?* '<> # ; ^23~ @` `.Programme=abc def	Invalid command. Name is too long (Max 99 characters).	Test name/programme length validation	Pass

		ghi jkl mno pqrs tuv wxyz ABC DEF GHI JKL MNO PQRS TUV WXYZ !"§ \$%& /() =?* '<> # ; ^~ @` .			
TC010	Invalid mark (non-numeric)	INSERT ID=2000007 Mark=abc	Invalid command. Mark must be numeric.	Test mark validation	Pass
TC011	Invalid mark (out of range)	INSERT ID=2000020 Mark=150	Invalid command. Mark must be between 0 - 100.	Test mark validation	Pass
TC012	Show sorted by ID descending	SHOW ALL SORT BY ID DESC	Here are all the records sorted by ID DESC from the table "StudentRecords".	Test sorting	Pass
TC013	Show sorted by mark descending	SHOW ALL SORT BY MARK DESC	Here are all the records sorted by mark DESC from the table "StudentRecords".	Test sorting	Pass
TC014	Show invalid field	SHOW ALL SORT BY AGE	Invalid sort field. Use ID or MARK.	Test field validation	Pass
TC015	Show invalid order	SHOW ALL SORT BY ID SIDEWAYS	Invalid sort order. Use ASC or DESC.	Test order validation	Pass
TC016	Show missing sort field	SHOW ALL SORT BY	Missing sort field (ID or MARK).	Test command syntax	Pass
TC017	Show summary	SHOW SUMMARY	Here are summary statistics from the table "StudentRecords"	Test statistical summary	Pass
TC018	Update single field	UPDATE ID=2000001 Name=Jenny	The record with ID=2000001 is successfully updated.	Test single-field update	Pass
TC019	Update all fields	UPDATE ID=2000001 Name=Alex Tan Programme=Software Engineering Mark=75	The record with ID=2000001 is successfully updated.	Test multi-field update	Pass
TC020	Update non-existent ID	UPDATE ID=2999999 Name=Ghost	The record with ID=2999999 does not exist.	Test invalid ID handling	Pass

TC021	Update with no optional fields	UPDATE ID=2000001	At least one of NAME, PROGRAMME, or MARK must be provided for UPDATE.	Test optional fields	Pass
TC022	Using update command without required ID	UPDATE name=john	Missing required ID.	Test update without ID	Pass
TC023	Query existing ID	QUERY ID=2000001	The record with ID=2000001 is found in the data table.	Test normal query	Pass
TC024	Query with trailing arguments	QUERY ID=2000001 name=john	Invalid command. Only ID allowed.	Test parsing	Pass
TC025	Delete with spaces around '='	DELETE ID =2000003 Y	Invalid command. No space allowed before '='.	Test parsing	Pass
TC026	Delete record w invalid confirmation	DELETE ID=2000001 no	Invalid input. The deletion is cancelled.	Invalid confirmation input	Pass
TC027	Delete record (cancel N)	DELETE ID=2000002 N	The deletion is cancelled.	Test user cancellation	Pass
TC028	Delete record (confirm Y)	DELETE ID=2000001 Y	The record with ID=2000001 is successfully deleted.	Test delete with confirmation	Pass
TC029	Save current state	save	The database file "P4_1-CMS.txt" is successfully saved.	Test normal save	Pass
TC030	Save current state	save	Error saving the database file.	Test saving corrupted/missing database file	Pass
TC031	Save when no changes exist	save	No changes detected. Nothing to save.	Test that save skips unchanged DB	Pass
TC032	Restore backup (confirm Y)	RESTORE Y	Database successfully restored from backup. Changes are not saved yet.	Test backup restoration	Pass
TC033	Restore from	RESTORE	Backup file "P4_1-CMS.bak"	Handle missing backup	Pass

	unknown backup	Y	does not exist. Cannot restore.	file	
TC034	Undo last action	UNDO	UNDO -> Undid DELETE (ID 2000001).	Test undo functionality for delete operation	Pass
TC035	Undo with nothing to undo	UNDO	Nothing to undo.	Test empty undo handling	Pass
TC036	Redo last action	REDO	REDO -> Redid DELETE (ID 2000001).	Test redo functionality for the delete operation	Pass
TC037	Redo with nothing to redo	REDO	Nothing to redo.	Test empty redo handling	Pass
TC038	Quit with unsaved changes	QUIT Y	WARNING: You have unsaved changes. Are you sure you want to quit? Type "Y" to confirm or "N" to cancel.	Test quit warning for unsaved data	Pass
TC039	Quit with saved changes	QUIT Y	Are you sure you want to quit? There are no unsaved changes. Type "Y" to confirm or "N" to cancel.	Test normal quit confirmation	Pass

## 4. Team Contributions

Name	Contribution
Chew Shu Wen	Project structure setup for compiling multiple C files in Visual Studio Validation module logic for all commands (ensuring correct input handling) Integration of all core functions into a cohesive system Development of core functions: OPEN, SHOW ALL, INSERT, QUERY DELETE, UPDATE, SAVE. Enhancement features: Sorting functionality (SHOW ALL SORT BY ID/MARK DESC/ASC) Unique features: UNDO, REDO, RESTORE Design and collection of test cases for validation

Adora Goh Shao Qi	<p>Developed core functions DELETE and SAVE, ensuring correct removal, memory cleanup, and file writing.</p> <p>Implemented validation checks for IDs, confirmation prompts, and command inputs to strengthen system specifically for DELETE function</p> <p>Created a test script, enabling batch execution of test cases to validate core and enhancement features efficiently.</p>
Calson See Jia Jun	<p>Implemented several key core functions in the CMS systems, including the OPEN command that loads the database into memory and the SHOW ALL command that displays all student records. Also developed the sorting enhancement feature, allowing records to be sorted by ID or mark in ascending or descending order. Assisted the team in validating overall system behaviour by checking for logical consistency across functions.</p>
Au Myat Yupar Aung	<p>Implemented the full INSERT function, including the input validation specifically for the function itself.</p> <p>Implemented the SUMMARY statistics features, covering both the overall summary and programme-specific summary, to enhance data analytics capabilities</p> <p>Conducted the testing and validation phase, designing comprehensive test cases and automating them using command files to efficiently run hundreds of test cases within seconds</p> <p>Supported the integration of the enhancement features by assisting in logic verification and ensuring compatibility with core functions</p>
Chung Kai Sheng Desmond	<p>Developed the UPDATE function and ensured the function works by running the function with comprehensive test cases.</p>
AI teammate	<p>Guided project structure, input validation, and debugging approaches. Helped the team identify potential risks such as buffer issues and unexpected input, and suggested difficult test cases for stronger validation. It also assisted in simplifying explanations for the report. The AI did not write the code but supported decision-making by offering ideas and clarifying concepts.</p>

## Appendix

Link to video presentation + system demonstration:

[https://youtu.be/fpuOYx\\_Hax4](https://youtu.be/fpuOYx_Hax4)