Code Structure Overview

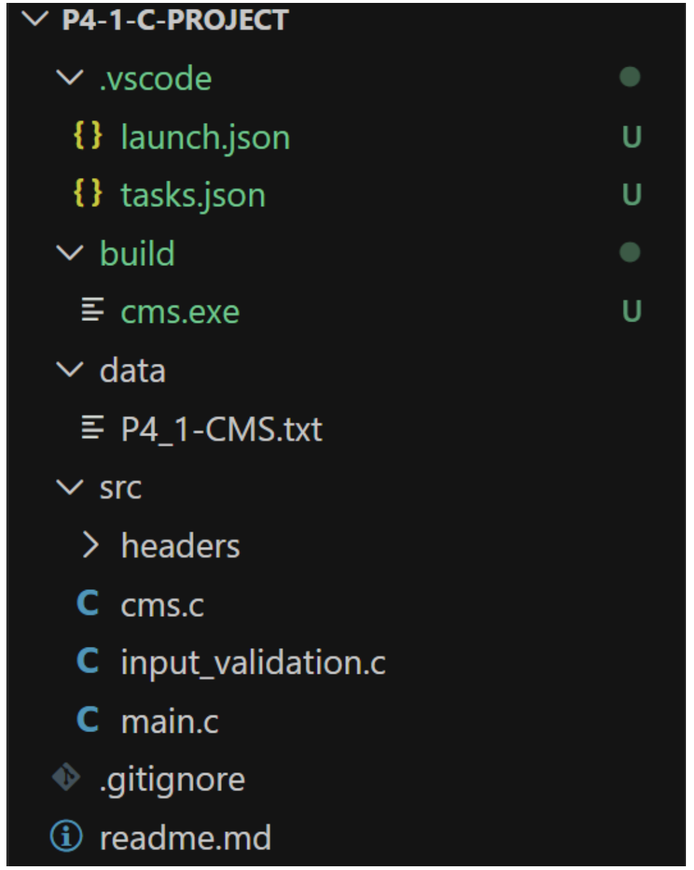
C Project: Class Management System (CMS)

Team: P4-1

File Organization

Project Structure Overview

- Clear project folder layout
- /src → C source files
- /headers → function declarations
- /data → database file
- /build → compiled output



Student Data Structure

What the StudentRecord struct stores

- Student ID
- Student Name
- Programme
- Mark

Reasons for choosing this struct

- Keeps all student information in one place
- Makes storing each record in the linked list simple
- Helps with searching, sorting, and displaying
- Ensures all functions use the same data format

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

Node Structure Used in CMS

- A single Node lets each student record exist in both the linked list and the hash table
- *next → maintains record order in the linked list
- *hashNext → links recodes inside the hash table for collision handling and fast lookups
- The dual pointer design supports both traversal and O(1) ID search

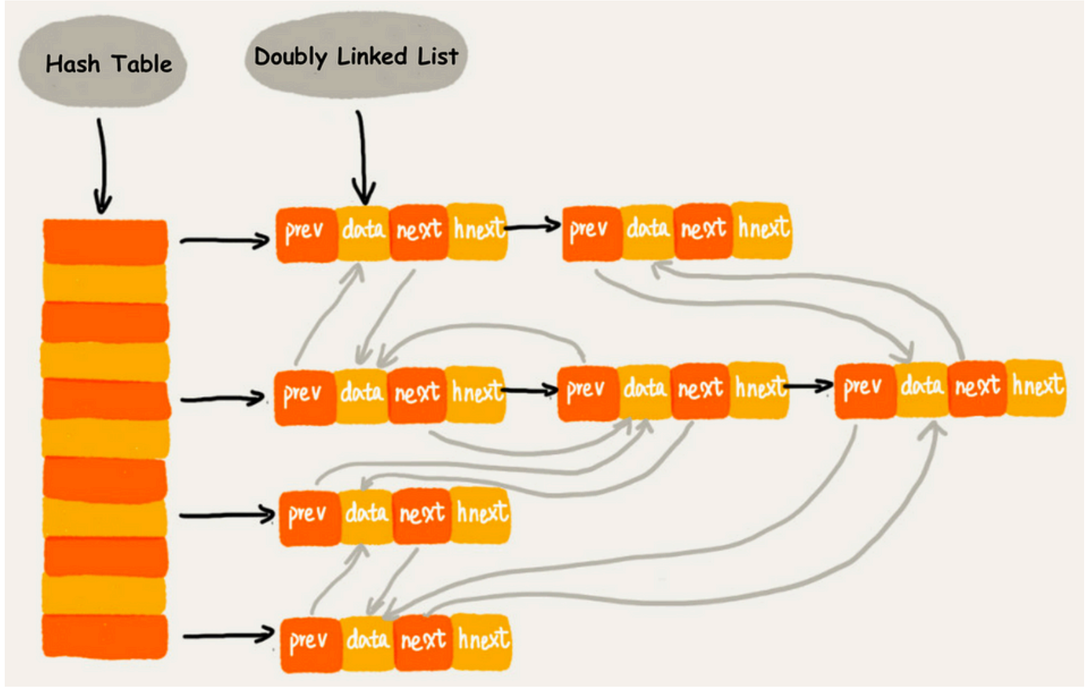
```
typedef struct Node {
    StudentRecord data;
    struct Node *next;
    struct Node *hashNext;
} Node;
```

Linked List for Ordered Traversal

- Maintains record order (file load or insertion)
- Efficient dynamic resizing (no fixed limit)
- O(1) append using tail pointer
- Used by:
  - SHOW ALL
  - SHOW SUMMARY
  - SAVE (writing back to file)
  - DELETE (removing from main list)

Hash Table for Fast ID Lookup

- Maps student ID → bucket index
- Enabled fast search in O(1) average time
- Uses separate chaining (hashNext)
  - Used by:
  - INSERT (duplicate ID detection)
  - QUERY
  - UPDATE
  - DELETE



Source: https://blog.devgenius.io/why-hash-tables-and-linked-lists-are-often-used-together-d30992fa18a7

```
void hashInsert(Node *node) {
    int index = hash(node->data.id);
    node->hashNext = hashTable[index];
    hashTable[index] = node;
}
```

Action Structure for Undo/Redo

Implemented using a stack-based mechanism

User operation creates an Action snapshot

- oldData → state before operation
- newData → state after operation
- type → identifies action category

Allowing precise reversal or reapplication of:

- Insert
- Update
- Delete
- Restore

```c
typedef struct Action {
    ActionType type;
    StudentRecord oldData;
    StudentRecord newData;
    struct Action *next;
} Action;
```

Action Structure for Undo/Redo

Implemented using a stack-based mechanism

Why a Stack?

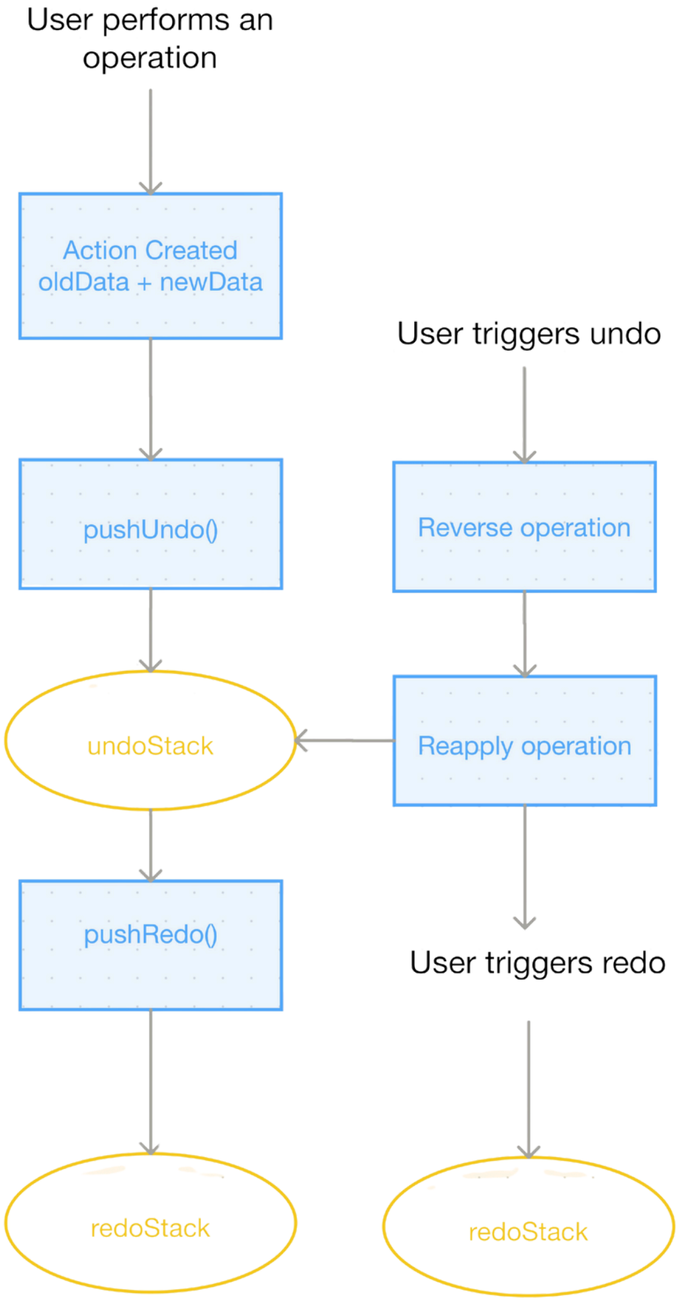
- Uses Last-In First-Out logic
- Ensure the most recent edit is undone first

How it enables full reversal

- Undo pops from undostack → applies reverse operation
- Redo pops from redostack → reapplies action
- Both stacks work together to allow multi-level undo/redo

Design rationale

- Action structure → memory efficient, accurate, and reversible



How commands are processed

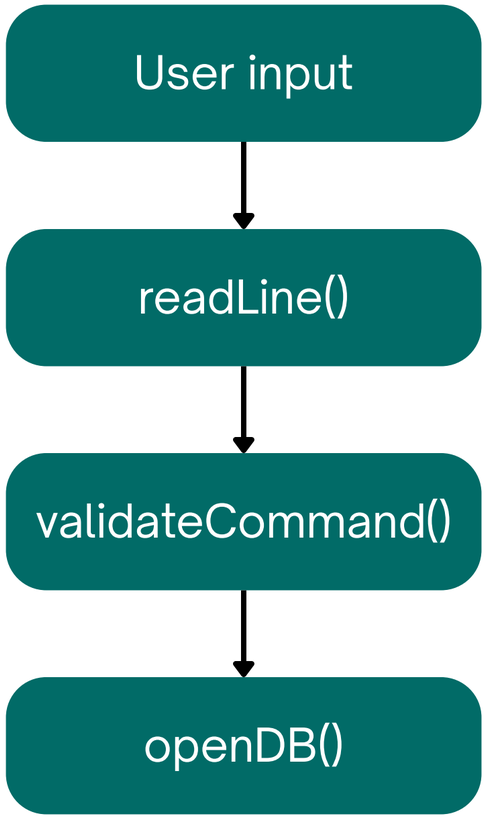
Commands that do not require parsing

readline()

- Dynamic buffer resizing
- Reads user commands of any length to prevent memory overflow

validateCommand(input, "OPEN")

- Trims leading whitespace
- Convert command to upper case
- Compare with actual command “OPEN”
- E.g “ oPeN ” → “oPeN” → “OPEN”



How commands are processed

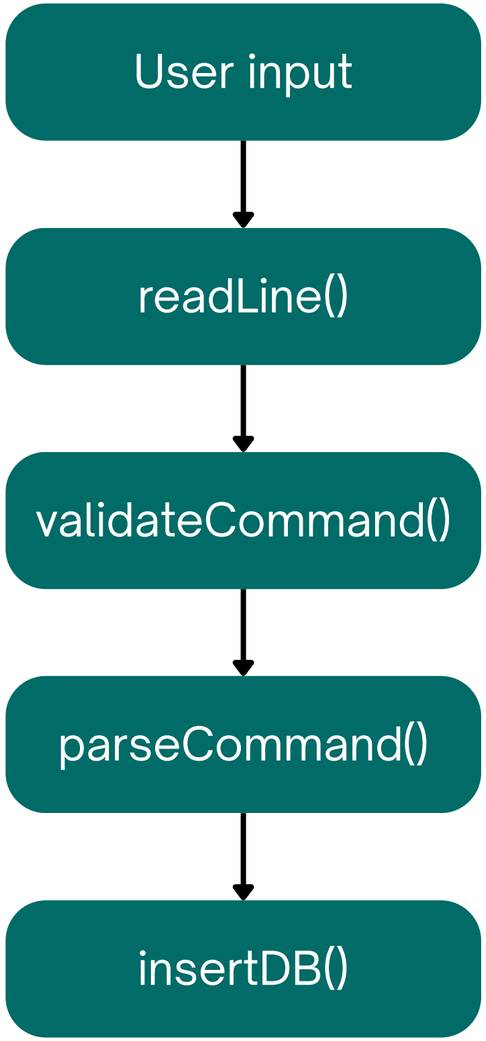
Commands that require parsing : parseCommand()

validateCommand(input, "INSERT")

- “ InsErt ID=2000001   NAME=John   DoeMark=50 "
- “InsErt ID=2000001 NAME=John DoeMark=50”
- “InsErt"
- “INSERT”

parseCommand()

- Extract key/value pairs from the full input and populate fields
- validateID()
- validateMark()
- toTitleCase()
- INSERT + ID = "2000001" + NAME = "John Doe" + MARK = "50"



How commands are processed

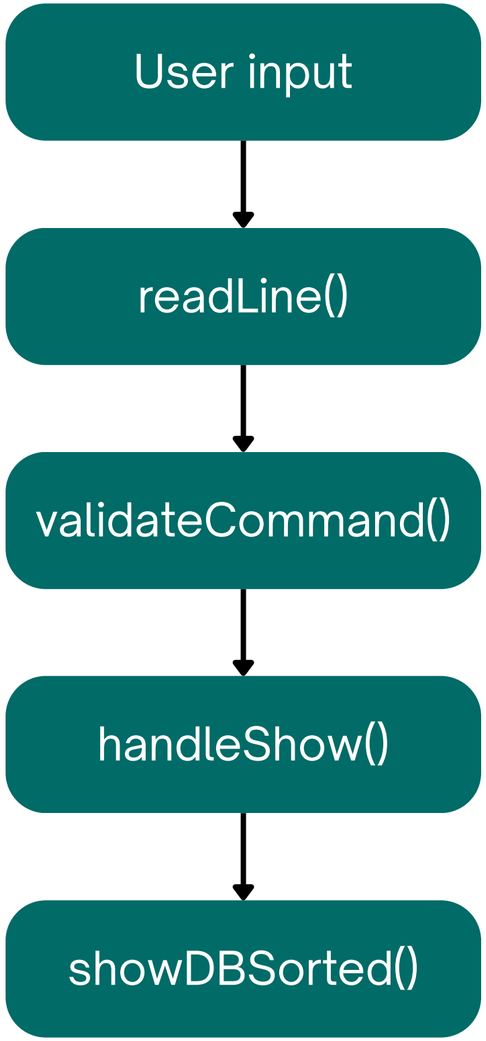
Commands that require parsing : handleShow()

validateCommand(input, "SHOW")

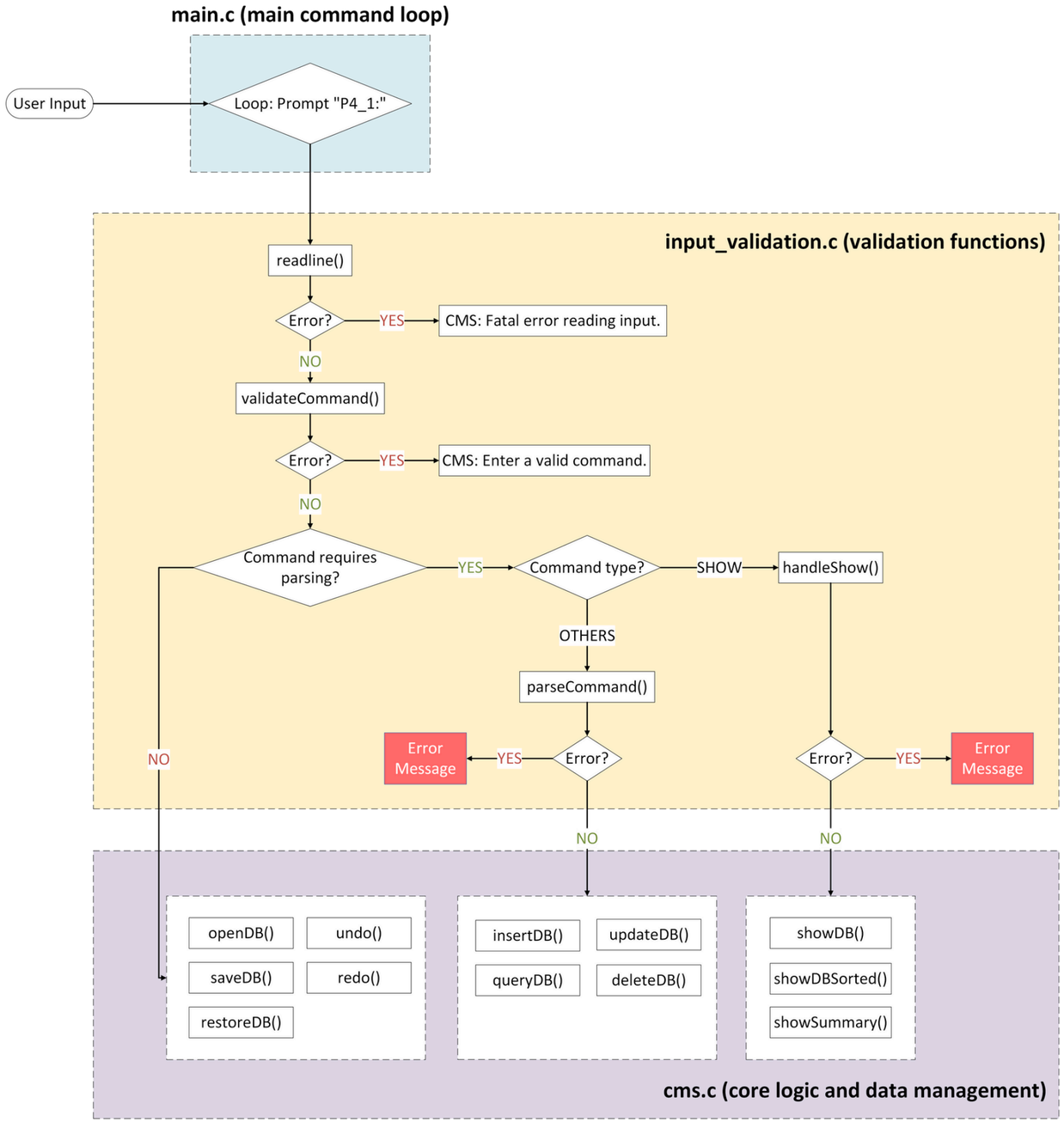
- “ Show aLL sOrt by id aSc "
- “Show aLL sOrt by id aSc"
- “Show”
- “SHOW”

handleShow()

- String is duplicated and tokenized using strtok(buf, " ")
- if (strcasecmp(token, "ALL") == 0) → TRUE
- next token = "SORT" → TRUE
- next token = "BY" → TRUE
- next token = "ID" → VALID field
- next token = "ASC" → VALID order
- showDBSorted(sortByID = 1, ascending = 1);



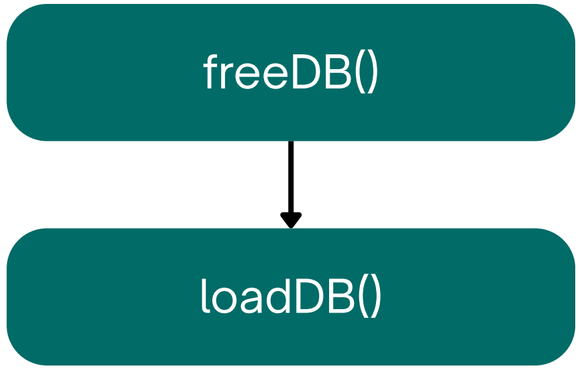
How commands are processed (flowchart)



File Reading/Writing Implementation

Loading the Database (loadDB)

- Mode: "r" (read-only)
- Clear previous records freeDB()
- Read file line by line
- "r" prevents modification of backup file during restore



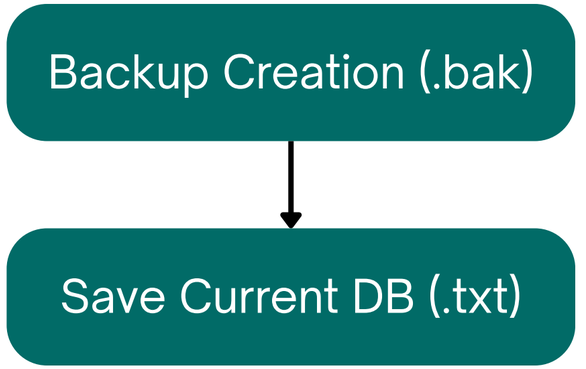
loadDB() → Clear pre-existing in memory database → Read file → Linked List & Hash Table
"2000001\tJohn Doe" → Node{id=2000001, name="John Doe"}
"2000002\tJane Smith\t90" → Node{id=2000002, name="Jane Smith", mark=90}

File Reading/Writing Implementation

Saving the Database (saveDB)

Backup Creation

- Original: "r", Backup: "w"
- Copy characters using fgetc/fputc
- Copy each character from original → backup using fgetc/fputc



Save Current Database

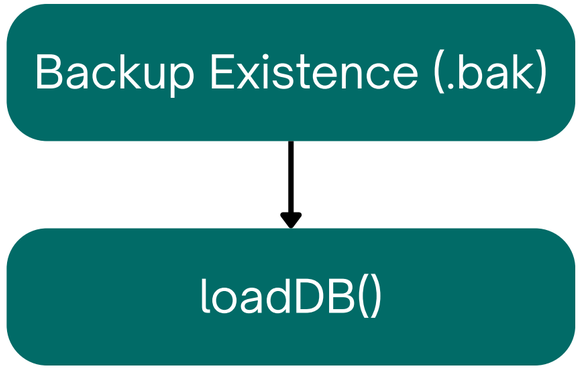
- Mode: "w" (overwrites file)
- Write linked list as tab-delimited lines
- Destructive write ensures file matches current DB
- Backup prevents data loss

Linked List → fopen("w") → File Lines:
[2000001, John Doe] → "2000001\tJohn Doe\n"
[2000002, Jane Smith, 90] → "2000002\tJane Smith\t90\n"

File Reading/Writing Implementation

Restoring the Database (restoreDB)

- Mode: "r" (read-only)
- Check if the backup file exists.
- Load backup via loadDB() → populate linked list & hash table
- "r" prevents modification of backup file during restore



Backup File → loadDB() → Linked List & Hash Table
"2000001\tJohn Doe" → Node{id=2000001, name="John Doe"}
"2000002\tJane Smith\t90" → Node{id=2000002, name="Jane Smith", mark=90}

Thank You!