# Technical Report:

## 1. Game introduction

Mystery Library Puzzle is a single-player, console-based puzzle game in which the player explores a magical library by moving from book to book and solving riddles. Each book is represented as a node in a linked list, and the structure of the library is randomized every time the program runs, so each playthrough offers a different path. The goal of the game is to start from one of four entrance books, navigate through a series of intermediate EASY and HARD books, and finally reach and clear one of two final books that contain the ultimate riddles.

The game focuses on two main learning objectives: practicing linked list data structures (including branching using multiple pointers) and integrating them with interactive gameplay logic. The riddle-based interaction demonstrates how data (clues) and structure (nodes and pointers) work together to control progression through a non-linear game world.

## 2. Data structures used

The implementation relies only on pointer-based linked structures and simple helper containers:

- **Clue struct**
    - Fields: string problem, string solution.
    - Purpose: Encapsulates a single riddle and its answer, so game logic does not need to know text details.
- **Book struct**
    - Fields:
        - int bookID – unique identifier of the book.
        - string bookType – "ENTRANCE", "INTERMEDIATE", or "FINAL".
        - string difficulty – "EASY", "HARD", or "NONE" for non-intermediate nodes.
        - Book* next1, Book* next2 – outgoing pointers used to represent one or two possible paths.

    - Clue clue1, Clue clue2 – riddles stored in the book. EASY books use both; HARD books use only clue1.
    - bool visited, bool cleared – track whether the player has seen and solved the book.
    - Book* nextInList – pointer used internally to maintain the list of intermediate books.
  - Purpose: Represents both shelves and books in the game as nodes in a linked structure.
- **BookList struct**
  - Fields: Book* head, Book* tail, int count.
  - Member functions: addBook, getBookAt, shuffleBooks.
  - Purpose: Maintains a singly linked list of all intermediate books to support easy insertion and a shuffle operation.
- **ClueDatabase struct**
  - Arrays: easyClues[8], hardClues[6], finalClues[2].
  - Counters: easyCount, hardCount, finalCount, and indices easyIndex, hardIndex.
  - Functions: shuffleEasyClues, shuffleHardClues, getNextEasyClue, get NextHardClue, getFinalClue.
  - Purpose: Stores all riddles and provides them in randomized order to books.
- **Standard library helpers**
  - <set> is used in cleanup to avoid double delete when different paths converge to shared nodes.
  - <algorithm> is used for swap and to transform strings to lowercase for case-insensitive answer checking.

**3. Room node design**

In this game, a “room” is modeled as a Book node. Each book behaves like a room with doors (pointers) leading to the next rooms:

- Entrance rooms:
    - bookType = "ENTRANCE".
    - difficulty = "NONE".
    - One outgoing pointer next1 leading to the first intermediate room.
    - No riddles; they act as starting points.
- Intermediate rooms:
    - bookType = "INTERMEDIATE".
    - difficulty is "EASY" or "HARD".
    - EASY rooms:
        - Two outgoing pointers next1 and next2, each leading to a different next book.
        - Two clues (clue1, clue2); the chosen path decides which clue the player must solve.
    - HARD rooms:
        - Only one logical outgoing path via next1.
        - One clue (clue1) that must be solved to continue.
- Final rooms:
    - bookType = "FINAL".
    - Two special nodes with IDs 100 and 101 (Ancient Tome and Forbidden Scroll).
    - One final riddle stored in clue1.
    - No outgoing pointers; they terminate the path.

The node design keeps gameplay-related state (visited, cleared), structural links (next1, next2), and content (clues) inside a single structure, which simplifies traversal and rendering.

**4. Linked list diagrams**

You should draw these diagrams by hand or using a diagram tool, then screenshot them for the PDF. A verbal description to guide your drawing:

- **Intermediate book list (internal list)**
    - A simple singly linked list using nextInList:
        - head -> [Book 10] -> [Book 11] -> ... -> [Book N] -> nullptr
    - This list is used only for generation and shuffling.
- **Gameplay graph using next1/next2**
    - Four entrance nodes on the left, each with one arrow (next1) into the intermediate layer.
    - Intermediate nodes arranged roughly in a chain but with branches:
        - HARD nodes: one outgoing arrow next1.
        - EASY nodes: two outgoing arrows next1 and next2 creating forks in the path.
    - All branches eventually point to one of the two FINAL nodes on the right.

Include at least:

- One diagram for the Book structural fields.
- One diagram for the intermediate list via nextInList.
- One diagram showing an example playthrough path from an entrance to a final book.

**5. Flowchart of gameplay**

Create a flowchart with these main steps:

1. Start → show welcome message.
2. Ask player to choose entrance book (1–4) → validate input.
3. Set currentBook to chosen entrance; mark visited = true.
4. While currentBook != nullptr:
    - Display book info (ID, type, difficulty, status).
    - If bookType == FINAL:
        - Ask final riddle.
        - If correct → show win message → End.
        - Else → show game over → End.
    - Else if bookType == ENTRANCE:
        - Wait for Enter → move currentBook = currentBook->next1 → mark visited.
    - Else if bookType == INTERMEDIATE and difficulty == EASY:
        - Show two possible next books (if not null).
        - Ask player for path choice (1 or 2) with validation.
        - Call solveClue with clue1 or clue2.
        - If wrong → game over → End.
        - If correct → mark cleared, set currentBook to selected next pointer, mark visited.
    - Else if bookType == INTERMEDIATE and difficulty == HARD:
        - Call solveClue with clue1.
        - If wrong → game over → End.
        - If correct → mark cleared → wait for Enter → currentBook = currentBook->next1, mark visited.

5. After loop → print “GAME COMPLETE”.

**6. Initialization logic**

Initialization is handled in main and buildLibrary:

- main
    - Calls srand(time(0)) to seed the random generator.
    - Declares an array of 4 Book* for entrance books and two Book* for final books.
    - Calls buildLibrary, then startGame, then cleanupLibrary.
- buildLibrary

1. Creates four entrance books with IDs 1–4 (ENTRANCE).

2. Creates two final books with IDs 100 and 101 (FINAL).

3. Generates a random number of intermediate books between 8 and 15 using int numberOfBooks = 8 + rand() % 8;.

4. For each intermediate book:
    - Chooses difficulty randomly (EASY or HARD).
    - Creates a Book node with type INTERMEDIATE and adds it to the BookList.

5. Instantiates ClueDatabase, which:
    - Fills arrays with easy, hard, and final riddles.
    - Shuffles easy and hard arrays in its constructor.

6. Calls assignCluesToBooks to give intermediate books their clues and assign final clues to the two final books.

7. Shuffles the intermediate list using BookList::shuffleBooks, which randomly swaps node contents (including IDs, difficulty, and clues).

8. Calls connectBooks to:

        - Link each entrance’s next1 to the first few intermediate books.
        - For each intermediate node, set its next1/next2 to either subsequent intermediate nodes or directly to one of the final books, depending on position and difficulty.

**7. Clue system explanation**

The clue system stores every riddle in ClueDatabase and attaches them to books during initialization:

- **Storage**
    - Easy clues: 8 short riddles in easyClues.
    - Hard clues: 6 more challenging riddles in hardClues.
    - Final clues: 2 special riddles in finalClues reserved for the final books.
- **Randomization**
    - In the constructor
      of ClueDatabase, shuffleEasyClues() and shuffleHardClues() use a Fisher–Yates style shuffle so that the order of riddles changes each run.
    - When assignCluesToBooks is called, each EASY book receives two consecutive easy clues, and each HARD book receives one hard clue via getNextEasyClue() or getNextHardClue(); indices wrap around using modular arithmetic.
- **Answer checking**
    - During gameplay, solveClue chooses clue1 or clue2 depending on the selected path or difficulty.
    - The function prints clue.problem, reads the full line as the player’s answer, and converts both the player’s answer and the stored solution to lowercase using std::transform.
    - If the strings match exactly, the function prints a success message and returns true; otherwise it prints the correct answer and returns false.

    - The calling code then either opens the next path (on success) or ends the game (on failure).

**8. Score system (including hint penalties)**

The current implementation does not have a numeric score or hint system; the game outcome is binary (win or game over). To satisfy this report section, you can explain the current state and propose an extension:

- **Current state**
    - No explicit score variable is stored.
    - The only "performance" feedback is the number of steps printed when the final book is cleared (stepsCount).
    - There is no hint command, so no hint penalties are applied.
- **Suggested extension (for documentation)**
  You can describe how a score system could be added without changing the core structure:
    - Add int score and int hintsUsed variables in startGame.
    - On each correct answer, increase score (e.g., +10 for EASY, +20 for HARD, +50 for final).
    - Add a hint option in solveClue that reveals part of the answer and subtracts points (e.g., −5 per hint).
    - At the end of the game, display total score and total hints used.

Mention clearly in the PDF that these are "planned features" if you do not implement them in code.

**9. Conclusion**

This project successfully demonstrates how linked lists and pointer-based branching can be used to represent a non-linear puzzle game world. The game dynamically builds a randomized network of books, assigns randomized riddles, and allows the player to traverse this structure by solving clues at each step. Although the current version focuses on core traversal and riddles rather than scoring or hints, the modular design of Book, BookList, and ClueDatabase makes it straightforward to extend the system with additional mechanics such as scoring, inventory, or a richer hint system in future work.