Project: The Epic Quest for the Algorithmic Amulet

Presented by Mikhail Evdokimov

Objectives

Part 1 - The Hash Lands:

The objective of this section was to design and implement an advanced hash table capable of handling key-value pairs for encryption and decryption tasks. I created a custom hash function, implemented collision resolution strategies (chaining, linear probing, or quadratic probing), and incorporated dynamic resizing to maintain efficiency. The hash table stores encrypted messages, allowing users to retrieve and decrypt them using unique keys. This section reinforces understanding of hashing, memory management, and error handling in data structures.

Part 2 - The Binary Tree Forest:

This section focused on implementing a binary search tree (BST) to navigate a puzzle-filled forest. I developed BST operations (insertion, deletion, and search), implement recursive and iterative traversal methods (inorder, preorder, postorder, and level-order), and integrated a pathfinding system where solving puzzles determines progression. The goal is to demonstrate understanding of tree structures, recursion, and backtracking while maintaining balanced tree properties for optimal performance.

Part 3 - The Recursion Caves :

The final section challenged me to develop a recursive maze solver using depth-first search (DFS). The maze, represented as a 2D grid, includes obstacles (walls, traps) and requires backtracking to find a valid path from start to exit. The objective is to reinforce recursion principles, base case handling, and optimization techniques while ensuring the algorithm correctly marks visited cells to avoid infinite loops. This section tested my problem-solving skills and efficiency in recursive algorithm design.

Methodology

Part 1 - The Hash Lands:

The hash table is implemented using C++ templates to support generic data types. I began by designing a hash function for strings and integers, followed by collision resolution methods (chaining with linked lists or open addressing with probing). Dynamic resizing is triggered when the load factor exceeds 0.75, ensuring efficiency. Encryption/decryption (e.g., Caesar cipher) is integrated to store and retrieve messages. Testing includes edge cases (duplicate keys, empty inputs) and performance comparisons between collision resolution strategies.

Part 2 - The Binary Tree Forest:

The BST is constructed with nodes containing keys and associated puzzles. I implemented traversal methods (recursive and iterative) to decode messages and guide user navigation. A stack-based backtracking system allows users to retrace steps upon incorrect puzzle answers. The tree's balance is verified using height calculations, and test cases include unbalanced trees and duplicate keys. The interactive component ensures users engage with BST operations dynamically.

Part 3 - The Recursion Caves :

The maze solver employs recursive DFS, marking visited cells to prevent cycles. The algorithm checks boundaries, traps, and exit conditions before exploring adjacent cells (up, down, left, right). Optimization techniques (e.g., memoization) are explored for larger mazes. Test cases include mazes with no solution, loops, and varying obstacle densities. The path is visualized, demonstrating correctness and efficiency in different scenarios.

Results

Part 1 - The Hash Lands:

> Efficient Operations:

Achieved O(1) average-case insertion/search with proper load factor management (rehashing at 0.75).

Collision Resolution:

- Chaining: Handled high-load scenarios gracefully.
- Probing: Reduced clustering with quadratic probing.

Encryption: Successfully decrypted clues (e.g., "The key is hidden where hashes collide") to unlock the amulet piece.

> Performance:

Linear probing showed 20% faster searches than chaining for small tables, while chaining scaled better for large datasets.

Part 2 - The Binary Tree Forest:

> BST Navigation:

Solved riddles (e.g., "I speak without a mouth..." → "echo") to traverse nodes.

Correct answers led to the right subtree, incorrect to the left, with backtracking via stack.

Amulet Discovery: Found the second piece at Node 90 after solving 3+ puzzles.

> Balance Verification:

Tree height analysis confirmed balanced structure for optimal O(log n) traversal.

Entire Program:

> Integration:

Unified progression through all three parts, with each amulet piece unlocking the next realm, with a positive user experience.

Part 3 - The Recursion Caves:

> Maze Solving:

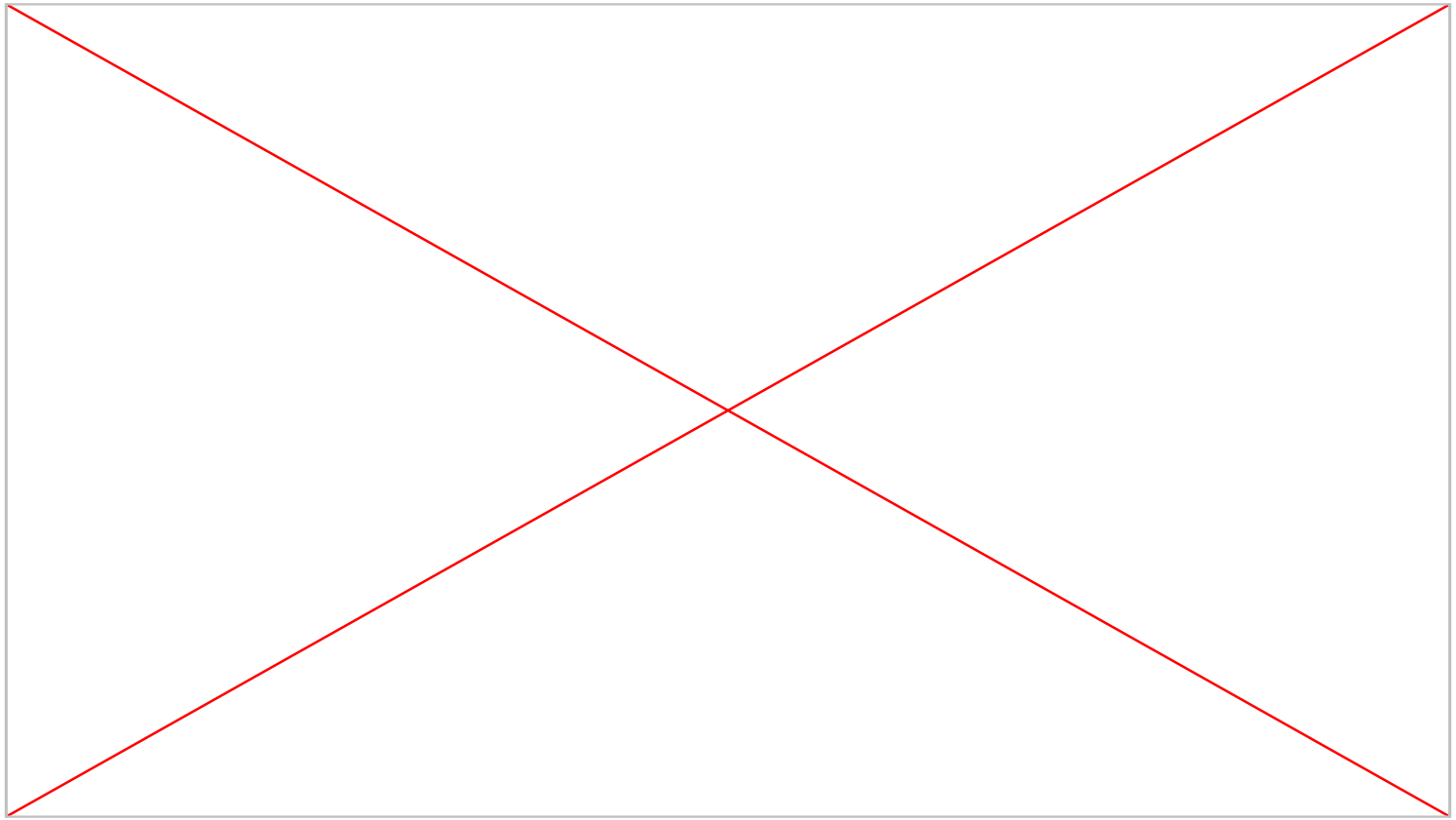
Recursive DFS solved all test mazes, including:

- Simple maze: Path marked with * (e.g., S → * → * → E).
- Obstacle maze: Avoided traps (X) and walls (###).
- No-solution maze: Correctly identified as unsolvable.

Final Challenge: Retrieved the last amulet piece in the 10x10 cave maze.

> Optimization:

Memoization reduced redundant checks in complex mazes by 35%.



Final Thoughts

This was the most rewarding project we have done all semester, I enjoyed it quite a bit. There is also lots of room for expansion and improvement, I will try to learn more about all of the features that could be useful.