TextQuest Design Document
CSC 403
Adam Barnett
April 22, 2025

App description

TextQuest is a text-based adventure game in which a player controls their character through user input in response to prompts and situations in a given area. The player will go from room to room encountering non-playable characters to speak with and puzzles to complete. A large portion of the game is reliant on the player's interaction with their inventory and that of whatever they encounter.

Choice of data structures and justification

HashMaps for Inventory

The player character, non-playable character, and room objects in the game all have *Inventory* instance variables.

Inventory will be a class that initializes a HashMap of *String-Item*** key-value pairs. Because the game will require substantial item interaction with the environment and non-playable characters, HashMaps' O(1) efficiency for inserts, deletions will work well for easy addition into inventory when objects are exchanged between inventories when the player picks up an item from the room, drops an item, gives someone an item, or receives an item from someone else.

Players will use items via text input. The program will take that String input (the key) and locate the item in O(1) time as well. For example, if a player encounters a locked door, they can type something in like "use door key." If "door key" is in their inventory (search) they can use it efficiently.

HashMaps are preferable to ArrayLists because an ArrayList needs to use linear search to locate an item in an unordered list. There is a size tradeoff, though, because HashMaps can also require linked lists or parallel arrays for proper collision resolution.

***Items are a superclass that contain subclasses of items like Weapon, Spell, Shield, etc.*

ArrayDeque for Puzzles (using inventory):

A double-ended queue will work with the inventory to solve certain puzzles that require items to be used in certain combinations and order. In these puzzles, the selected item will be removed from the HashMap inventory and entered into the Deque. Upon using the required number of items, the puzzle will iterate through the deque by emptying it from the front to determine if the items were used in the correct order. The stack functionality of the deque is only needed in case the user wants to swap out the last item with a different one in case they make a mistake and don't want to type everything again.