CS20 Final Project Report (TUI Card Game)

For my final project in CS20 Data Structures and Algorithms I decided to make a game that would incorporate the various concepts we learned over the course of the semester. The game's general idea is that you have two players playing, and each player takes their turn in succession with each other. First, I will discuss the various problems that would need to be solved, and then how they were solved.

A general description of the game is as the player you choose a set of cards to play against the other player. Amongst these cards are Sword, Shield, Healing, and Poison. During your turn you choose the cards to play, and they are added to the queue, but be careful because once you've chosen one you can't go back and replace it. It's part of the strategy. Now once you've chosen all your cards, it's time to end your turn and inflict damage on the enemy or give yourself minor buffs such as shielding or healing.

The first problem I thought of when coming up with this idea is how I would deal with the "inventory". In the beginning I thought that it would be dealt entirely with a closed hash table. My original motivation for this is that entries are stored via keys/values and has very fast lookup. Insertion was of no concern as this would only happen max 6 times in the entire program. So, I used this to store only the data that describes each card's effects and not the player's actual cards. This way when I wanted to retrieve this information all I need to do is query the hash table for a key, and I'll get a result very efficiently. I ended up just using a simple array for the actual contents of the inventory though, as it didn’t make sense to use a more complex data structure for that.

The next problem I had was how to deal with card selection and how to play the cards against the opposing player. In this case I ended up using a circular queue, as they have a simple array as the underlying data structure and lend themselves very well to ordered insertion/removal. Whatever you enqueue will dequeue in the opposite order, as a FIFO (first in first out) data structure should. An added motivation for using a queue with an underlying array is that I knew beforehand there would be a maximum number of elements in the queue. While linked lists are better “in general”, arrays can be more efficient when you can anticipate a maximum size which I could. Most if not all functions in the queue ended up being constant time.

Next, a major part of the game is that you can give each player varying effects. These effects are tied directly to the types of cards. But the question is how do you store these effects in an interesting way? Well, I figured that using some variation of the stack data structure would be cool because of its LIFO (last in first out) property. It would give some extra strategy to the gameplay, by making the order that the effects are pushed matter. Now, getting into some specifics about this stack, I decided to make it doubly linked with header and trailer nodes. By using header and trailer nodes pushing and popping from the stack becomes a constant time operation.

Finally, there is one last feature of the game. This is a win/loss tracker. Essentially it displays how many games you've won and lost, but in a special way. Having each win and loss intermixed with each other based on "which" game was won seemed a little strange. So, I opted to implement it with a sorted list. The underlying data structure for this list is a double linked list. Nothing special like the player effects, just a normal double linked list. There are a couple interesting aspects of this class. One being that it was implemented recursively. Linked lists in general lend themselves well to recursive implementations due to their "linear tree structure". Another implementation detail that sort of caught me off guard was that sorted lists don't implement all of the typical List ADT functions that we defined in class. So while writing it I opted to immediately throw in those default list functions, stating that they are invalid operations for the Sorted List implementation of the List ADT. Instead I supported slightly modified functions, which have the same names with different parameters.

This project was a challenge, and throughout it I ran into so many different weird and frustrating problems. However, one of the main things that I personally realized is that recursion is a really cool thing! The way it works kind of lets you essentially list out the different edge cases in a linear fashion, where iterative implementations do not shine.

P.S. There are bound to be logic issues in the gameplay as the flow of the game is quite complex. I am very aware of them.