EECS 268

Lab5 Writeup

Chris Watkins

1. Describe and justify how you plan to implement the stack interface (e.g. nodes, arrays, save it all to file?). Use details from the problem to justify your answer.

My implementation of the stack interface utilizes nodes, with each node housing a string. I read the input file values directly into a linked list of nodes (type string) before adding any elements to the stack. I found this method intuitive and effective. My string implementation uses functions similar to the linkedList class for push, pop and peek. Since the stack is a last in first out data structure, each node<string> added to the stack contains a name of a movie attendee, and the node is added to the top of the stack. Peek simply returns the name housed in the top of the stack. Since the stack in this problem has a fixed size, an array implementation could have been possible as well. I simply declared a maxSize in the constructor of the stack, and made sure the size of the stack never exceeded the maxSize (10) when the push method was called.

2. Describe and justify how you plan to implement the queue interface (e.g. nodes, arrays, save it all to file?). Use details from the problem to justify your answer.

My implementation of the queue interface was similar to that of my stack Interface – nodes housing a string value of a movie attendee. The queue has an unlimited size so a node structure was a clear choice. The queue is a first in first out data structure, so my enqueue() method added a node to the back of the queue instead of the front. Similarly, my dequeue() method removed the front of the queue. Pointers were utilized to keep track of the front and back of the queue. Peek() returned the name of the person at the front of the queue using the getValue() function from the node class.

3. Create an implementation plan, listing out the order in which you plan to implement the classes. If some classes will be developed in parallel, discuss this.

After creating the stack and queue interfaces, I turned to building a general structure of the derived (from runtime_error) class PreconditionViolationException (both .h and .cpp files), as I would need to utilize this class when implementing my interfaces. I developed the Stack and Queue class in parallel, as they share many similarities. I utilized my PreconditionViolationException class while implementing the functions that could possibly violate one of the defined errors given by the lab assignment (i.e. pop() on empty stack). Once both Stack and Queue classes successfully implemented their respective interface and were structured to correctly handle all exceptions I declared and implemented my Theater class. This class creates a linkedList of the input data during construction as well as a empty Stack and Queue, before letting the run() function implement all of my Stack and Queue methods based on the commands given by the input file.

4. After completing your implementation, describe in a one or more paragraphs what the most difficult part of the implementation for you was. There isn't a right or wrong answer to this. Please do discuss. We're interested in response like "The Stack class was hard."

I used a node based structure for both the Stack and Queue and found this made the actual implementation of the interfaces intuitive and straightforward for the problem at hand (running the movie theatre). I did, however, run into some issues regarding how things were deconstructed when either enqueue() or pop() was called and had to coordinate with my Node class destructor in a way that was compatible and symbiotic. I also struggled in initially getting the syntax correct for how to handle each exception using the created derived exception class. Once, I had done it correctly once, future implementation was easy, but I had issues initially getting it right.