Individual Report | Ivan Tran - S3849505

<u>Major enhancement 1: Doubly-Linked list |</u>

Files: LinkedList (h, cpp), LinkedListSingle (h, cpp) LinkedListDouble (h, cpp)

I have replaced the linkedlist with an abstract class “LinkedList.h” and 2 derived classes “LinkedListSingle” and “LinkedListDouble”. The reason I have done this is for code reusability. When you analyze tasks required by the vending machine, most of the task only requires you to traverse the list one way. Hence only about 15 percent of the code needs to be rewritten, that being “addLL”, “removeLL” and the node classes. I have also made the node class an abstract class. This is done to maintain the “head” variable within the linked list abstract class.

For the AddLL, sorting is done through insertion for both singly and doubly. However, the implementations are both different. The variable “prev” allows more freedom when writing. You do not need to keep track of “previous” which is the advantage of using a doubly linked list. RemoveLL has the same principles as AddLL. You do not need to keep track of previous values in order to delete something. This enhancement can be seen through the option “display items”. Once selected, it will prompt the user 3 options: Ascending, descending and menu. And to see insertion, it can be seen through the “add item” option. It will order by alphabet (non capital sensitive) no matter how many times you insert.

<u>Major enhancement 2: Command Design Pattern</u>

Command: Command (h, cpp), Invoker: Menu (cpp), Receiver: VendingMachine (h, cpp)

There are 2 new classes (Command and Menu). Essentially command creates command objects that represent kind of like a message. They are objects used to tell the receiver to execute a task. And the ones that send these commands are the invokers (Menu). Vending machine class is essentially broken up into tasks/methods. Each method executing a task for example: Adding an item, removing an item etc. There are a total of 10 including toggling enhancements. This method is great for readability and efficient in terms of reducing redundancy.. Spotting errors becomes easier and when modifying a task, it will unlikely interfere with other objects/methods. Each class has a role which also makes planning easier.

<u>Minor enhancement: Color</u> File: ColorOutPut (h)

I have created a class that solely works on changing the color of strings. This is done by adding onto the string or integer. I have used a generic or template to create my color class. I have used this so that strings or numeric values are able to pass through the methods without having to rewrite methods or approaching the overload method. This class also works hand to hand with toggling enhancement. Basically all functionality uses this class to create colors for outputs. Obviously the template follows an assumption that it is either a “string” or “numeric” else there is a chance it may create errors.

<u>Overall:</u>

Modifying milestone 2 wasn’t difficult. After reading through the code, I have noticed some errors. I have nothing to compare it to, hence, it's hard to determine whether modifying m2 could have been easier or harder. I think we should have structured more or at least focused on structuring at the very beginning. It would have made things easier, such as having all error messages in one file. It would have made coloring easier or even instant implementation. This can be said for success messages too and displaying lists.The code implementation is a little bit messy, this could have also been avoided if everything was planned and structured.