Rob Mullins
Data Structures
3/3/20

Lab 1 Analysis

The purpose of the first lab is to take in an input file with postfix expressions and output them into a file converted to compiler like instructions. I decided to take the approach of using a stack in order to hold the information needed.

What I could do differently next time:

I could have used a linked list but since we did a homework assignment that walked us through the basic steps of how to implement a stack in this regard, I felt a bit more confident doing it this way. A linked list could have been another tool to implement something similar. I chose an iterative solution, where as a recursive solution could have been used as well (more on that later). Another approach to the entire problem would be to create an object stack, where it could take in object types as well as primitives so that we could set a struct/ object with the data and then push the entire thing to the stack. This would allow us to condense information all together and even keep strings as the temp vars.

Justification for Implementation:

A stack is a good method to use as it has a LIFO style implementation. This is useful so that you can push and pop items in order on and of the stack. When looking at different logical implementation I chose to stick with the iterative method which I think is the most concise way of doing this. A recursive implementation can be used here however would make more work than needed. The base case for a recursive call is easy as read and sort the char you read in until there are no more to read. That being said, it would be a lot more difficult to keep track of the stack and contents of it as you recursively go through the requested postfix expression. When using an iterative solution, it is much easier to keep track of the stack and what is in it.

Initially I tried to implement the ‘switch’ style logic to go through and help determine what needed to be sorted where. I originally used a stack that held ints rather than chars. I would take in some chars and convert them to their decimal value and store them that way. In the end I decided to switch this up to using an if statement, and then implementing a char array/stack. I did this as it was becoming a bit difficult changing the input from int to char and back, so I cut out one step. Something that I would like to improve would be the logic to write things out, i.e., the if statements. I think there is a better way to condense this logic into a couple of statements rather than the loads that I am using now.
Some important methods that I implemented was the resize function, as well as the convert function. The resize function is used to resize the stack if I ran out of room to push onto the stack. I started off with a fairly low amount of available room as to conserve memory. I initially thought to upload the entire contents into the stack and the sort through and parse it out once I hit a newline char. I decided to change that approach after talking to Scott A. . He explained that is an approach to doing it however, it

would consume vast amounts of resources if it wasn't just a text file. He also showed even given a semi-inf long expression you may only need 3 spots available to process it all. Another function could be added to remove space availability depending on the size being put in, but I thought that a bit unnecessary as space isn't that big an issue in modern computers. The convert function is the meat of the problem. This hold s all my logic for sorting and writing the determined instructions to the specified output file. This function takes in a char read by main and begins to determine if it is an operator, operand, or neither. After doing this it would either push onto the stack or pop twice. If it was neither it would gracefully fail alerting the user it read something that it could not process. A brief description of my classes and methods are below:

Class stackChar is a class that holds all of my methods for filling and removing from the stack, as well as some other more useful functions. This class contains the constructor of actually creating a blank char[] of stacks. I also have the push function which populates the stack while checking for any overflows. The pop command will remove an item off the stack also checking for an underflow. I went ahead and created the size command which returns the current size which is kept track by pushing and pop. The clear command takes the current contents of the stack and sets everything inside to '0' or ascii null. I also check to see if the stack is empty with an isEmpty function.

Class fileTasks contains a method to create a file and writes to that file. When creating a file it firsts checks to see I the file already exists, if not it will create the file. Looking at the writeFile we open the file write the contents and then close the file again (very slow to repeat).

Class postConvert holds the main part of the logic, the convert function. This function takes in a char and a filename for writing out. It will then check the char to see if it is ascii null or '0' if not it will go through the logic of determining if it is an operand or an operator. If it is neither than it will throw an exception and return the error to the terminal.

Class RMullinsLab1 holds the main function. Here we prompt user for input for an input file to read from and an output one. If it does not accept it, it will throw a traceback. Assuming it is correct it will begin reading char by char skipping the newline char and carriage return. It will then pass the char int other converter and go through the file assuming there are no errors.

Efficiency Issues:

The larges efficiency issue lies with my fileTasks class. This class holds methods that creates and writs to a I given output file. The way I decided to do this was have the file opened and closed each time a write is made which is VERY inefficient. Besides waiting on user input this is the main factor to the amount of time being used to complete the given tasks. As I mentioned above, I think there is a better way to layout the logic from what I did, but this is what I thought of first. I have a lot of duplication which takes up space, not that it is a large contributing factor, but it is a bit annoying in my opinion.

The stack its self is an O(1), while the actual reading and writing to the file I think would be much higher. Without seeing the code I am unsure what the I/O time would be. Another way to increase efficiency is to change the language to C++.

One issue that I have in my logic is holding the temp vars. I only allowed for 1-9 temp vars anything less than that or more than that will not be recognized as a tempN. So given extremely large expressions it would error out/ write something that doesn't make sense. If we implement a more struct/object style way of holding the information we could hold them as a string array or even a string and then convert to char after the fact.

Enhancement:

For the enhancement I decided to implement the '$' as the exponent. Given the logic I used it was very easy to implement, I just added another if statement to compare the chars. I also considered the '0'(zero) and since it is an ascii null I went ahead and ignored it by looking for null's first and then skipping over the logic. I represented the exponent in the instructions by 'EX' similar to the style provided for us like add being 'AD'.