EN.605.202.81: Introduction to Data Structures

Peter Sullivan

Lab 1 Analysis

Due Date: 7/1/24

**Lab 1 Analysis**

The lab assignment requested that we set up a process for maneuvering from a prefix expression to an postfix expression. There are many ways that this could be implemented. In order to go from a prefix to postfix we need to process the prefix expression from right to left. This really means were reading in the string in reverse. While reading in the string we are keeping track of operands and keeping track of operators. When an operator comes up, this is when we take two of the most recent items that were processed, and we join those into a new operand.

For example, we first process A, then B, and then a + comes up. The next step is to take the previous operands A and B and combine those into a new operand using the operator. The way that we combine those is dictated by what type of expression we are trying to create. If we are trying to create a postfix expression then we would create an operand as AB+. For infix we would create (A+B). The type of expression dictates the process of combining operands. When we combine those operands, we can think of the previous three elements as 1 new element or operand. This means that as we continue to process the data and a new operator appears, we will treat the previous three operands as one operator. So if we continue parsing and we see a C appear and then an \*. This means we take the AB+ and combine with the C and \*. This becomes AB+C\* for postfix. Or ((A+B) \*C) for infix.

Why am I showing the process of processing a expression from postfix to prefix? This is because in order to understand the best implementation process, we first need to understand the process. In order to process a prefix to postfix expression, I decided to use a stack (as advised for the lab). The way that the stack works is we process the expression character by character. If we are going from prefix to postfix, we process the prefix expression from right to left. As we process the characters, operands are pushed onto the stack. This allows us to grab the most recent items easily from the stack. When an operator appears, we pop the most recent two items on the stack and combine those items with the operator to create a single operand that is then pushed back on the stack. We continue this process until there are no more characters in the prefix expression. As we process and combine each character, we should end up with only one single item in the stack, which should be the postfix representation of the intiaal prefix expression.

A stack is a very easy way to grab the most recent items. No indexing or slicing is required, so this is a very good solution for this problem. Another option would be using a deque, though we would have to make sure that we are grabbing the most recent instead of the fifo that a deque offers.

Another option is to use a recursive solution. This would be an acceptable way to process the prefix expression. To implement a recursive solution we would have the function start parsing through the prefix and slowly start creating the postfix. While parsing, when ever we ever we combine to create a new operand such as a +b. We could then send over the prefix unprocessed and the postfix processed to the function to be called again. We would iteratively keep combining until the base case where the prefix string has been completely processed and the postfix string is now complete.

**Complexity**

*Recursive Implementation*

The recursive solution would slowly process through the string which would give us a O(N) complexity. We would then call the function multiple times until we process the string, so the complexity would be O(N)\*M. M would be the amount of calls required for the base case to be reached. As the expression gets very long the M can be ignored and the complexity would become O(N).

*Stack Implementation*

The expression class is doing most of the work for my program. If we are looking at prefix to post fix method in the expressions.py file, I first initialize an empty stack **(complexity O(1)).** I then loop through the input expression in reverse. In order to read in reverse, I use the reverse order method that I created. This method implements a stack to reverse the order of the string. In this method, I have two loops that loads the stack and unloads the stack. The complexity of reverse order would be **O(N)\*2.** I then have another for loop, where I load up the stack for each item. I then either push or pop twice. If I pop twice from the stack, I then initialize an operand object. I then push the operand object to the stack using the operand combine object. All of these are O(1) complexity. Assuming that we are able to convert to a postfix, I then return the popped stack. Here is an outline of the complexity assuming the worst case:

* Initiate stack
  + stack = StackArray()
  + O(1)
* Reverse String:
  + reverse\_order(self,string):
  + O(N)\*2
* Loop through reverse string:
  + For loop
  + O(N)
* Pop twice
  + pop2 = stack.pop()
  + O(1)+O(1)
* Initiate Operand Object
  + operand = Operands(pop1,pop2,char,'Pre\_Post')
  + O(1)
* Push operand combined
  + stack.push(operand.combine\_operands())
  + O(1)+O(1)
* Return pop
  + Return stack.pop()
  + 0(1)

The complexity would be O(N)\*2 +O(N) +O(7). This would turn into O(N) complexity.

**Both the recursive and iterative process would have a complexity of O(N).**

If I had to choose between a stack solution or a recursive solution. I would probably go with the recursive solution due to its simliciity. We are only doing one combine operand operation at a time, which is very easy to keep track of. Both solutions work and have a similar time complexity so if we are going by just time complexity I would go with the recursive solution. If we were to look at space complexity that would be a different issue. Each time we run the recursive function we would be implementing a new prefix and postfix strings. This could be very large space consuming if the strings are incredibly large. While the stack solution would not require us to keep creating new prefix and posfix strings or stacks. If we are purely looking at time complexity, either stack or recursive are a great solution. Personally I would choose recursive as it would be a cleaner and simpler way to parse through the expression. If we are looking at time and space complexity, stack is the process to go with!

**Lessons Learned**

If I were going to do this project again, I would spend much more time on the test cases/edge cases. I think it would have been a lot easier for me if I had spent more time trying to understand method of how the prefix is processed to prefix instead of trying to make the code work.

I’ve never used the arg\_parser before, and I am very happy I’ve learned this package. It’s incredibly useful to be able to create a package or program that can be run by the command line. I also found the pathlib package very useful.

I had a difficult time making the project into a module. I had first created the process and had it running via a juptyer notebook, which made it easier for testing. For future labs, I want to create test cases at the beginning. I then want to start the project as a module. That way for testing purposes, I can utilize the command line to process inputs and start debugging immediately.

I learned the hard way that project verisioning is important. I made changes to the project and then suddenly nothing was working. After many control Z’s, I was then able to run the module again. As instructed by the professor, I utilized git and github for versioning on the lab. I started to work on my lab in smaller pieces, creating a branch for each enhancement, and pushes those changes into the master branch after testing. This made me feel a lot more comfortable about trying changes, when I knew that I had a working version saved in the master branch.