EN 602.202 Introduction to Data Structures

Lab 2 Analysis

Implementation of Prefix to Post Fix using List and Recurrsion

By Eduardo Carrasco

Data: 3/18/25

Summary of Abstract Data Structure and Justification.

This Lab 2 is a continuation of Lab 1 where we focus on taking a prefix expression and converting it to a postfix expression. This time around I choose to use a simple list using a stack method. The focus of this lab was implementing improvements from the previous labs as well as using recursion to convert the inputs.

The key improvements from this lab 2 are as follows:

1) Using recursion instead of a for loop to take a stack of prefix to postfix express.
    a. A more efficient and understandable way of implementing the prefix expression.

2) Unit Test for various test cases for testing prefix expression to postfix conversion.
    a. Including a unit test file makes it more

3) Postfix to Prefix conversion.
    a. At times you may not just want to test forward and backwards for the expressions.

4) Python virtual environment.
    a. A virtual environment is a standard approach of usage for python modules.

Stack Implementation

In terms of how the overall structure of the code works, it is read in the prefix in.txt file line by line. Each line is then converted to a stack Abstract Data Structure. For example if one of the lines read in is ***+AB*** . This will be converted to stack.

The way the algorithm then looks for a few keep operators within what the input read line is. These characters are checked by character in the code. If the algorithm finds: +, -, *, $, or /. It starts to parse stack method or looking for another letter.

Time and Space complexity

In terms of efficiency of the code well it is rather slow but through. Let's cover time complexity first. First, let's start with how each line is read and converted to a stack. Since it converts each character into each line of input it has a complexity of **O(n)** time where **n** is the length of the line or number of characters being read in. For simplicities sake let us say n is also the length of the longest line in the input text. Secondly, we also have multiple lines being read in. Let us again say that this represents a line of m which is read in O(m) times. Which can yield a time complexity of O(n*m) or if n and m are equal, O(n^2) time.

The same can be said about the space complexity of this approach. Each line is read in character by character and converted into a stack of space of O(n) where n is the number of characters in the stack. The same is true for each line being read in. This we can of size O(m) where m is number of lines being read. Which can yield a space complexity of O(n*m) or if n and m are equal, O(n^2) space complexity.

Reflections on Possible improvements.

- Add a GUI to the program.
- Add method to the module to convert a directory of files from prefix expressions to postfix expressions.
- Add a infix conversion of prefix to infix expressions.
- Better error handling such as string values being valid inputs.

As we add more possible test cases and examples for use cases of this lab2.

Justification for the Algorithmic Design

Having a predefined stack class in the files named prefix_to_postfix_stack.py helped to break the Abstract data structure into more manageable pieces. At this point in time, we have not covered trees or tree variations. I also do not think a Linked List would be appropriate due to having to loop though each time to find specific characters. This linked list would have added to the memory and time complexity.

Secondly the choice to read each line by line is slower but allows the algorithm to check each character in the final step.

The last step was the most complicated; this is where you need to do a check for special operator characters of “+”, “-”, “/”, “*”, and “$”. Once found the algorithm must pop the following values.