605.202: Introduction to Data Structures

Claire Harelson

JHU ID: charels1

Lab 2 Analysis

Due Date: March 28, 2023

Dated Turned In: March 28, 2023

Lab 1 Analysis

A recursive approach to this problem was relatively similar to using a stack data structure. They possess a similar functionality in that while a stack follows the last in first out order, a recursive approach will work differently to essentially carry out the same function. This is because recursion will call through each individual conversion until it hits the base case, in which it is signaled to traverse back through the problem to produce the result. In this way it simplifies the larger problem by solving the smaller individual problems that comprise it. Although recursion may be more efficient than an iterative approach in some cases, I personally prefer the iterative approach and think it is better and easier. I believe recursion is a silly way to solve a problem and honestly, I hope to never use it again! After spending days attempting to solve this problem recursively, I was still unable to. I doubt this will ever come up in my intended career field, so I truly do not want to waste another minute of my time losing my mind over this.

I approached my program design by creating a recursive function designed to accept a prefix string as a parameter, conduct some error checking to ensure the statement is valid, then carry out the recursive call to return the converted postfix string. By defining the acceptable operands and operators allowed in a statement I created a handful of conditional statements to either allow the variables to be called by the recursive function or to raise an assertion error to the end user. I then included some code to carry out the actual conversion of the valid prefix string into the postfix string and return this result. The assertion errors in this function were designed and implemented to stop the program from running and alert the user that there is an issue with their input. However, if the input has no issues, then the file gets processed, and the output is written to a text file. Included in the outputs are the statements in incorrect postfix form as well as the corresponding runtime metrics for each conversion. My ‘postfix’ strings have all operands listed followed by all operators. This is obviously not correct as some operators should be within the operands.

Although I did not successfully fulfil the specific requirements of the lab to implement recursion to convert prefix expressions to postfix expressions, I was still able to make a somewhat functioning module. It accepts the specified operators as well as single letter operands. In addition to the basic requirements, I added several enhancements as well. I incorporated the use of some assertion errors to raise an alert to the user regarding specific issues. This included a check to ensure that there was the proper ratio of operands to operators in a statement and that only the specified accepted variables were included. Thus, this carries out error handling for any non-acceptable input types. I accounted for several other forms of user ‘error’, which included the acceptance of spaces or newline characters as well as the inclusion of a ‘^’ to represent a ‘$’ variable to account for various methods of writing exponentiation. My code is also set to handle the input of lowercase letters and convert them to upper case in the output for uniformity. I also used a try-except block in the __main__ script to alert the end user in the case of having an incorrect input file name or file path.

From this project I learned how recursion compares to using a stack and how to implement both of these techniques to convert prefix statements to postfix statements. I can also see how recursion could be utilized for a number of other purposes as well as it is somewhat versatile. I also better understand the importance of dividing my code into a package containing multiple scripts and various input files. The use of modularity helps to organize a project much more efficiently and allows an end user to better comprehend the program. This is especially useful if the intention is to just use a portion of a program rather than its entirety. I can see where this may come in handy for use in future labs.

For my next project I intend to take an approach of spending more time writing out pseudocode and an outline before diving right in to coding in my IDE. This will likely save me time in the long run as it will provide me a method to consider the ‘big picture’ concepts as a whole before working out the details of the implementation. I also would like to improve my error handling for file IO and include a broader range of assertions for different types of issues. More specific to this project though, I would write into the program a method for handling parenthesis. I know this would be an incredibly useful feature and unfortunately, I was unable to find the time to implement it. Another supplementary feature I would have liked to incorporate is a method to convert from prefix to infix and an additional method to convert from infix to postfix. Although the assignment asked for a direct conversion from prefix to postfix, I believe having this feature for infix could be very useful as well.

It is interesting to compare the recursive approach with the use of a stack to solve this problem. I believe each approach could be the better choice in different scenarios. In terms of extreme cases, there is a possibility of getting a stack overflow with recursion as you may not know how many calls will be made. With a stack, you can define a max height or allow the stack to be of infinite size. In many cases I think iteration has the potential to be more efficient in terms of time and space.

In terms of time efficiency with this program, I tried to measure the observed efficiency in nanoseconds of how long it took to convert each statement. This isn’t a particularly great way to analyze this, but I did notice somewhat of a correlation in terms of the size of a conversion with the time it took to complete. With the exception of some outliers, it is apparent that an increased problem size took longer to convert, as one may anticipate. The theoretical efficiency of this program is dependent on the inclusion of factors that influence runtime, such as loops. I have a number of loops and conditional statements in my program that may or may not execute depending on the input. I believe both the time and space complexity for this recursive approach would be O(n). This aligns with my observed efficiency as the size of the input influenced the time to execute the program.