605.202: Introduction to Data Structures

Claire Harelson

JHU ID: charels1

Lab 1 Analysis

Due Date: February 28, 2023

Dated Turned In: February 28, 2023

Lab 1 Analysis

A stack ADT was the best choice for this project to convert a string from prefix to postfix form. The stack is useful due to its last in first out order. This allows for items to be pushed onto and popped from the stack in the necessary order. This data structure initially is utilized to reverse a prefix string so that each item in the string gets processed one at a time from right to left. A stack is used again to then convert this prefix string to a postfix string. The features of a stack provide a method to push items on to the stack if they are operands. Once an operator is encountered, two items are popped from the stack and the operand is pushed on to the stack followed by the two operands. This process continues through the entirety of the prefix phrase. Once complete, the stack is popped item by item to reverse the order of the items and place them in a postfix string.

Although a string function could instead be used to reverse the prefix string in a simpler way, the application of the stack to convert the string is a very appropriate use. In terms of actual implementation of the data structure, I created a 'Stack' class to make this an object within my program and set the stack as an empty list. Within the class I defined the ability to pop and push items from the stack using the .pop() and .append() methods, respectfully. Additionally, I incorporated methods to determine if the stack was empty or full, which return a Boolean value. Although I provided the parameter of max height to pass into the Stack class, I chose to use 'inf' for my usage to allow the stack to be any size. This parameter could be utilized with an integer passed in to limit as needed. The last feature is the peek method, which allows a user to see the value at the top of the stack without altering the stack's composition. Although I did not use this feature in my program, it could be incredibly useful for an end user.

I approached the next portion of my program design by creating a Convert class designed to accept a prefix string as a parameter and return a postfix string. Using the Stack class allowed me to successfully carry out this conversion. 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 pushed onto the stack or to raise an assertion error to the end user. The assertion errors were designed and implemented to stop the program from running an 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 output is the correct statements in postfix form as well as the corresponding runtime metrics for each conversion.

I successfully fulfilled the specific requirements of the lab to create my own stack and utilize it to convert prefix expressions to postfix expressions. 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 a specific issue. 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. Additionally, a max overflow error is included if the stack exceeds the expected parameter of stack height. In my usage I set my stacks to infinite height, but if an end user were to pass in a max height value this form of error handling would be necessary. I accounted for several other forms of user 'error', which included the acceptance of spaces or newline characters as well as the conversion of a '^' to 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 the proper approach to implement a stack and utilize it for my desired purpose. However, I can also now see how this particular data structure could be utilized for a number of other purposes as well as it is very 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 Lab 2!

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.

I can see how a recursive solution could work to solve this program as opposed to this iterative one. Rather than using a stack, I could instead recursively move through a prefix statement to convert it to postfix character by character. 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. The time complexity would on average be linear as θ(n) and be influenced by n. Worst case time complexity would be O(n). This aligns with my observed efficiency as the size of the input influenced the time to execute the program. Accessing an item in a stack has a constant time complexity of O(1). Space complexity will be linear and be O(n) for a worst-case scenario. Again, this is influenced by the input size and input contents to be processed.