Data Structures Lab 1 Analysis | Cooper Conway | 06/02/2024

Discussion and Justification of Data Structure Used

The data structure that I used for converting a prefix expression to its postfix expression is an unbounded stack. This is the Last In First Out (LIFO) data structure discussed in module 2 of this course where items can be pushed (or inserted) to the top of the stack and popped (or deleted) from the top of the stack. I also defined methods to peek at the top element, check if the stack is empty or not, and empty the stack. For prefix notation, operators precede their operands while operators follow their operands in postfix notation. When reading right to left in prefix notation, the next operator will always be applied to the next two operands. Hence, a stack's LIFO nature makes sense for this problem by simplifying the pairing of a given operator with the next two operands while preserving previous operands in the correct order until they are also combined with an operator. Additionally, each element is processed in constant time because the needed element will always be the last added element in the stack. This is more efficient than using other data structures like a linked list or an array because no indexing to a specified stored space is needed. The last item added to the data is the item that will be used next, exactly as a stack is structured.

Description and Justification of Design Decisions

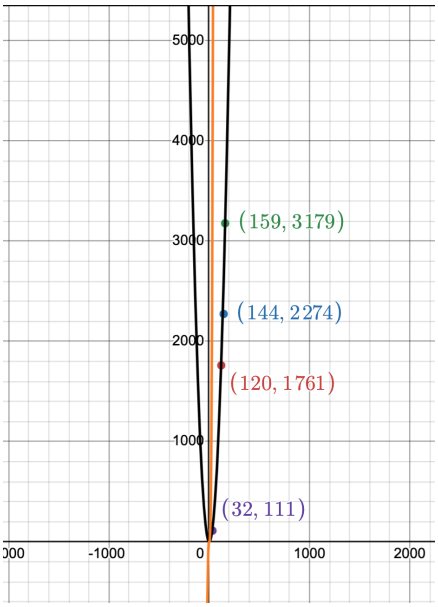
I designed my program for converting prefix to postfix as two nested for loops: one outer for loop that loops over the lines of the text file and one inner for loop that loops over the characters of the line. I chose to implement my own created string method to remove tabs, spaces, and new line characters so I would not have to consider those in the conversion algorithm (more on this in the *What I Learned and What I Would Do Differently* section). I also implemented my own created method that reverses the order of the inputted string since prefix expressions are meant to be read from right to left. I implemented an error flag (*err_flag*) to help with skipping over printing a calculated postfix expression when an input is invalid. There was also another design decision in the calling of the *get_count* function which I will discuss in the *Program Enhancement* section.

Efficiency with respect to Time and Space

In theory with how I designed my program, I am looping over (most of) each character in the input file three times: once to strip the string of spaces, tabs, and new line characters using my developer created *strip_string* function, twice to reverse the string to read from right to left, and a third time to perform the conversion algorithm. And so with three loops over each character, the time complexity is theoretically $O(n^3)$. To note, there is a possibility for this to be worse if more inputs result in certain errors where another loop over an invalid expression is performed (This is discussed further in the *Program Enhancement* section). The space complexity will be O(n) which is linearly related to the number of characters, n, in the input file.

To test the theoretical $O(n^3)$ time complexity in practice, I implemented a start and stop timer to get the time complexity in nanoseconds for a given input file. I also made a *get_input_size* function to count all characters in the input file including spaces, new line characters, and tab characters. This represents the space complexity. I tested 4 different inputs (the RequiredInput.txt plus three more) in my analysis.

Each coordinate pair (x,y) represents a different input where x is the input size of the inputted text file in number of characters and y is run time in microseconds. The black curve is $y = \frac{1}{7}x^2$ and the orange curve is $y = \frac{1}{10}x^3$. This was pleasantly surprising that the program performed better than what I calculated as the worst case. Since the *strip_string* loop removes empty characters, the *reverse_string* loop and conversion loop may perform better by having fewer characters to loop over. Based on this analysis, the program appears to follow a time complexity more like $O(n^2)$.



Recursion vs Iteration

My iterative solution loops over each character of each line in the file from right to left and utilizes a stack to combine operators with operands. A recursive solution could be implemented to remove the first character (which should be an operator) then calls itself recursively to get the next two operands. Assemble the removed operator with the two operands and return. The last non-operator character is the base case. Although my iterative solution using stacks had more than one pass through to handle reversing the string and removing space characters, the theoretical time complexity for iteration over each character once would be O(n). In this recursive method, each character is read one time in the input expression so the time complexity would also be O(n). I find the iterative solution more intuitive and therefore easier to understand and implement. However, since both solutions have the same time complexity, I don't think one solution would necessarily be better than another, at least in runtime.

Program Enhancement

The enhancement that I implemented into my program that goes above and beyond the requirements of this assignment lies within my error messages. In general, for a prefix expression to be able to be converted to a postfix expression, the number of operators must be equal to the number of operands minus 1. Errors in a valid prefix expression can include not enough operators, too many operators, or encountering an operator before two operands. On top of the descriptive error messages alerting the user when and why their prefix

expression is invalid, I implemented a *get_count* function that also outputs to the user the number of operators (**ops**) and the number of operands (**alpha**) that are present in their input expression. This enhancement will significantly help the user when providing correct and valid prefix expressions as input.

The implementation for when to call this *get_count* function when an error occurs did require some design decision making. I initialize a count for operators and operands as I loop through each character. If I loop through all characters and am still left with more than 1 item in the stack, then **ops** < **alpha** - 1 and the string is invalid. I have already calculated both counts because I have already looped through all the characters, and I can output that to the user. If **ops** > **alpha** - 1, then the input will error while in the middle of the line. I have not looped over all characters and do not have the counts to output to the user. Hence, I call the *get_count* function here. Then, I loop over all the characters to get the counts, and this includes looping over characters I have already read. While I assign the same counting variables to the fully calculated counts to save space, I do lose some time complexity for looping again; however, I am able to output to the user the counts for operators and operands for that input. I think this possible loss in time complexity is worth having immediate knowledge of the counts as an end user for fixing invalid inputs.

What I Learned and What I Would Do Differently

The algorithm to convert directly from prefix to postfix was not too difficult to figure out, and the logic was not too difficult to implement. I quite enjoyed this part. I learned how much other additional processing goes into putting together a fully documented, error handling, and input parsing program. I learned that this part is what gave me some difficulty as well as the thought process of finding input edge cases that would break my program and error handling them accordingly. I also learned how to take a stack and implement it practically in code. I think understanding things in practice and understanding things in theory can be two different things, and I was able to learn about implementing an ADT we discussed in lecture into my program. I also learned that decisions made on the design of your code can have an impact on the efficiency of the program. As mentioned in the *Description and Justification of Design Decisions* section, I developed my own method to strip away spaces, tabs, and new line characters from each expression. I found this to be intuitive so that it would be easier when processing the characters needed for the conversion. This ended up costing me an extra multiplication by n in time complexity due to looping over the input expression before the loop that implemented the conversion algorithm. Something I would do differently moving forward is I would disregard these spaces, tabs, and new line characters during the conversion on a single pass. This would theoretically make my program run more efficiently.