Data Structures Lab 2 Analysis | Cooper Conway | 07/23/2024

Discussion and Justification of Data Structure Used

There was not a "data structure" used for converting prefix expressions to postfix for this lab. Recursion was the main tool that was used to perform the conversion so no further data structure was really needed. The input to the function was a string data type and as characters were read from the string, they were passed back into the same function recursively or combined with other characters to form a string. Then, that combined string was passed back into the function recursively until the final postfix expression was returned. Recursion made sense for this conversion because each time an operator was read, two corresponding operands needed to be found. If another operator was found when looking for the previous operator's two operands, then the process starts over and two operands need to be found for that operator first to form operands for the previous operator. Hence, this process works well with recursion.

Description and Justification of Design Decisions

I designed my program to recursively convert prefix expressions to postfix using a for loop over the lines of the input file inside of a starter function (*starter_func)* where each line is passed into a recursive conversion function (*convert_prefix_to_postfix_recursive*). The recursive conversion function reads the first character of the string which should be an operator. It then aims to find two operands. It reads the next character. If this is an operator, the function gets called again. If not, it reads that character as the first operand, and it reads the next character. If this next character is an operator, the function is called again. If not, that character is the second operand and the postfix expression is returned as operand 1, operand 2, and operator. Hence, after an operator is read, the base case is that 2 operands must be hit. To read characters, a *get_valid_char* function is called. This function skips over whitespace characters, and it checks that characters are alphabet letters or valid operators. A -1 is passed back up the recursion in place of the read in character if any kind of error involving the string becoming empty or there still being characters in the string after the conversion is complete to indicate that the current input line is not in valid prefix form. A -2 is passed back up the recursion in place of the updated string (*in_str*) along with the character read to indicate an invalid character was found that way the character can be outputted to the user in the error. Additionally, I check that the input line contains valid characters before calling the recursion function to handle skipping over blank lines. I also designed my program to perform the conversion of prefix to infix notation alongside the prefix to postfix conversion to improve time complexity. I will discuss this further in the *Program Enhancement* section. I also implemented my own created method to remove the first character (or 0th index) of a string since characters needed to be removed from the string after they were assigned elsewhere.

Program Enhancement

The enhancement that I implemented into my program that goes above and beyond the requirements of this assignment is providing the equivalent infix expression for each valid prefix input. This provides all three inline arithmetic expressions discussed in lecture to the user: prefix (inputted by the user), postfix (calculated by the program), and infix (calculated by the program). I originally created a separate function to convert from prefix to infix using recursion (*prefix_to_infix_recursive*). After successfully creating this function for valid prefix expression inputs, I realized it uses the exact same algorithm as the recursive prefix to postfix conversion used where the only difference being the order in which the 2 operands and operator were combined: operand 1, operand 2, operator for postfix versus operand 1, operator, operand 2 for infix.

Hence, I decided to include parenthesis around the characters each time for the infix notation and then return that as a third argument in the tuple within the same prefix to postfix function. I could have called my prefix to infix using recursion function separated and only handled prefix to postfix in its respective function; however, it was possible to do the conversion in parallel since the logic was the same, and it was more time efficient because each character was only read one time.

Recursion vs Iteration

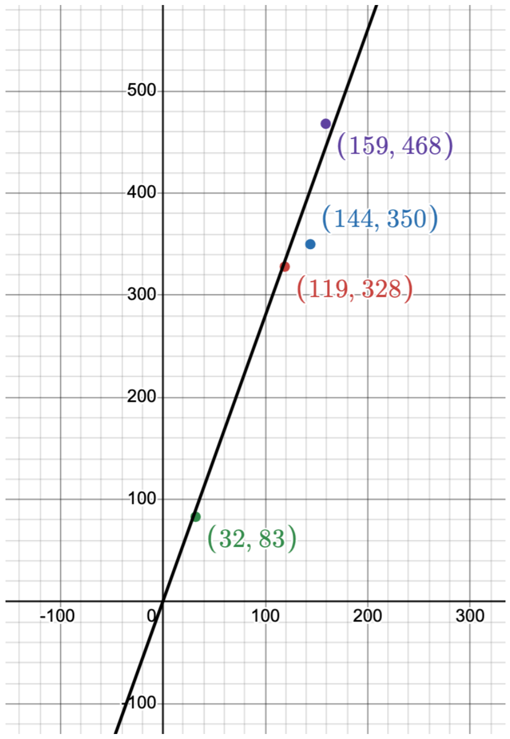
In the iterative approach, characters of an input prefix string are read from right to left and a stack is utilized where characters are pushed to and popped from the stack to combine operators with their operands. In the recursive approach, characters are read from left to right, and each time another operator is hit, the function recursively calls itself to find corresponding operands passing postfix expressions back up the recursion until the final postfix expression is formed. In theory, I believe the time complexity for both methods is O(n) because each character of the string only needs to be read one time so neither method is particularly better in time complexity. From a coding perspective, I found the recursive method easier to assemble the logic for (separate from the error handling). This may have been due to doing the iterative approach first so when it came to applying recursion, it felt easier. The iterative method needed to have the input string reversed and include several if checks for different steps depending on the character where some parts of the string were converted to postfix and some parts were still in the stack. The recursive method seemed easier to write because it could be read from left to right where each time an operator was hit the function calls itself again and the returned postfix expression became an operand for the previous run of the function. Additionally, it made the conversion more clear that non-converted components remained in the original string. One part that I did find more difficult in the recursive method was error handling (other than simple invalid character errors). When an issue emerged in the iterative approach, it was more intuitive and simpler to identify why based on the status of the stack being empty or not being empty after processing the full string. The iterative method processed all elements of the input so a diagnosis could be established. For the recursive method, the conversion was finished when it found two operands for the original first operator and not necessarily when it read all characters from the input string. This created some ambiguity between possible issues with the validity of inputs. Hence, I decided to implement a standard "invalid prefix expression" error because it was difficult to identify the specific problem for edge case inputs.

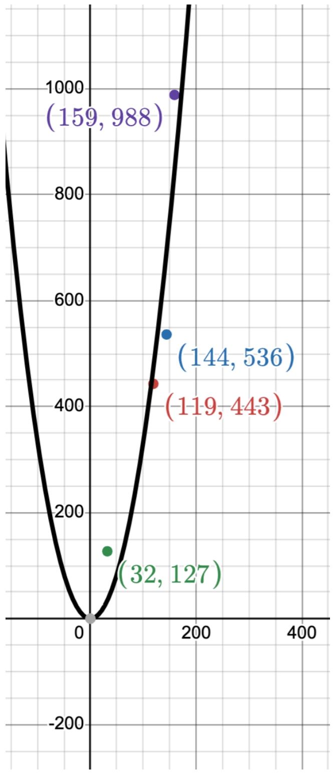
Efficiency with respect to Time and Space

In theory, the conversion from prefix to postfix using recursion loops through and reads each character of the input one time throughout the recursive process. Hence, I would expect my program to be O(n) in time complexity. To note, when a character is read during the conversion, it will need to be removed from the string. I initially used the built-in method of [1:] indexing into the string, and it allowed my program to properly remove the 0th index character efficiently. However, I also wrote my own method (*remove_first_character*) to remove the first character (0th index) of a string for the scope of this assignment. In this method, the program needs to loop over the string each time a character is read in order to remove it. When the function is called, there will always be one less character in the string than the previous call. So if there are n characters in the string on the first call, there are n-1 characters in the second call, n-2 in the third, and so on. Hence, the time complexity will be n+(n-1)+(n-2)+...+2+1 assuming all characters are read and removed. This is equivalent to the sum of the first n integers which equals n(n+1)/2

= O(n^2). Additionally, I would expect the space complexity to also be O(n) and linearly related to the number of characters, n, in the input file.

I did a study where I tested my program using the built-in string method of [1:] to remove the first character versus using my own function and evaluated the time complexity. 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 left graph shows results for the built-in string method and the right graph shows results for the string method I wrote.





The left graph with the built-in string method has the black line representing $y = \frac{1}{30}x^2$ while the right graph which uses the string method I wrote has the black line representing $y = 2.8x$. This follows what I expected for the recursive conversion to take O(n) but be O(n^2) using my string method due to the extra looping over characters.

What I Learned and What I Would Do Differently

The logic of converting prefix to postfix using recursion required some extra thought to figure out, but implementing the logic itself was not too difficult. As mentioned previously, I found it easier to implement the conversion using recursion over using the iterative method. This may be due to the fact that I used the iterative method first, and so I already dove deep into the basics of the conversion for that assignment. I learned that logic in how variables are passed between a function recursively can be a little confusing. For example, on my first draft of the program, I did not pass the updated string (*in_str*) in the return and so after an operand consisting of another postfix expression was returned, those characters were still in the working string. I found the debugger extremely useful for following how recursion works and helped me realize the updated string also needed to be in the return statement. I also learned how difficult it can be to use counters or trackers in a recursive process as indicators would need to be passed the entire way back up the recursive "tree" to return to the original call of the function. Something I would do differently would be to try to use a module to hold a global variable for counts of characters. I would also implement a better mapping of indicators for flagging different errors from invalid inputs and this may have helped me identify more specific error messages for the user.