Data Structures

EN.605.202.81.SP24

Lab 1: Prefix to Postfix Using Stacks

Logan Griffin

2/27/2024

Analysis

Implementation and Design

The main focus of this project was to utilize a self-made stack class to convert a prefix expression into a postfix expression. There are a couple of different ways to implement a stack class. For example, an array or linked list could be used. Although indexing, the main advantage of an array over a linked list, doesn't really make a difference for this application, I chose to use an array. Only having to store an item with an array, rather than an item plus location data, in a linked list, is one advantage that remains for this problem.

The program works by taking an input file and output file. The program will run the number of times equal to the number of lines in the input file. For each line, the valid characters (letters and operators) are taken and pushed to a stack. Since prefix expressions are read from right to left, the stack data structure works well here. Each character can be popped off the stack to simulate reading from right to left. In general, stacks are useful when data needs to be accessed or processed in reverse order.

When the characters are read, the program pushes letters onto a separate stack that will ultimately hold the converted expression. When an operator is encountered, the expression stack is popped twice to obtain the two operands the operator is operating on. The two operands and the operator are then concatenated and pushed back to the expression stack.

There are various checks along the way to flag an invalid statement. The program does not end when an error is encountered. Instead, it will replace any line with a statement saying that specific line was invalid. Spaces are ignored when initially reading in the line.

After a line is converted, there is only one item left in the expression stack. That item contains the entire converted expression. It is popped from the stack and written to the output file before the program tries to read the next line.

Error Handling

Given that the program takes in a random input file without knowing anything else about it, there are many different exceptions that could be encountered. The first exception the program checks for is that there are exactly two arguments: one input and one output file. Once the arguments are confirmed, the program checks to see if those files can even be accessed.

Considering the input file contents, there could be anything: letters, numbers, photos, etc. A single line is attempted to be read, and the program checks for valid characters. There also needs to be an appropriate number of operators for the number of operands. The line will be flagged if there are no

operators, no operands, or the difference between the amount of operands and operators is greater than one.

One of the possible errors that is not handled is having a file that has a blank line in the middle or beginning of the content. Since the program exits the reading loop upon encountering a blank line, a file that starts with a blank line or skips lines would not be read completely, if at all.

Recursive Solution

Converting an expression from prefix to postfix could also be completed recursively. The base case for evaluating a valid prefix expression would be reaching an operator. The algorithm would still take in one character at a time. The method would evaluate the current character and then call itself if a letter was read. Each time the method is called, the next character would be stored. For example, if the prefix expression was +AB, the characters would still be initially added to a stack/array/list. The first time the method is called, the "B" would be stored, the next time the "A", and the third method call would encounter the "+". The third call would trigger the base case, sending the + back to the previous call which is storing the "A", and then both the "A" and "+" would be sent back to the call storing the "B". Finally, the term, in this case the whole expression, would be assembled and returned as AB+.

Like other recursive solutions, the block of code required to accomplish the translation would likely be cleaner and more concise. In terms of storage, the initial stack would be the same. However, the expression stack could be created as the operators are encountered. Since the method itself would be storing the letters/operands, there would be no need to make the expression stack the same length as the line that is read in. This would help with both time and space complexity. There would be fewer operations overall, as well as less pre-allocated storage.

What I Learned

When I first started writing the stack class, I used a while loop to check for the current "top of the stack". Although it seems trivial now, I eliminated all of the loops and used a class variable to keep track of the top. This greatly reduced the amount of code and operations for the program to perform. Having limited coding experience, this project also taught me about how numerous functions work.

If I had to do this problem differently, I would try to extract more blocks of code from the main method and make them into separate methods. This would help, not only me, but anyone else reading the code, more efficiently understand the implementation. Another small change would be to somehow store the type of character (letter, operator, or invalid), after the line is initially read in. This would eliminate the need to check the type a second time when the prefix expression is read again from the stack. Other than that, I would try experimenting with linked lists.

This project provided the first opportunity to work with compiling and running programs from the command line. I learned how to move between directories, the syntax for compiling and running java programs, and the intricacies of file paths and their affect on the program's ability to run.

Description of Data Structures

This project examined the stack data structure. Stacks are a LIFO data structure, where the most recent element added is the first to be removed. Creating our own stack class provided insight to how the data

structure is implemented. Java has its own Stack class that has the same methods. Additionally, Java's Stack class has a search method that returns the place in the stack a specific item can be found.

Efficiency

The efficiency of the program can be examined from a few different viewpoints. If each "run" is considered the processing of a single line, the time complexity is 2N, where N is the number of characters in the in the line. Each character is processed once while being pushed to the stack. Each character is processed a second time after being popped from the stack. This would go up to 3N in the worst case scenario for a single line. A line that contained N letters would be processed completely a 3rd time. Even so, the program would have O(n) complexity.

If a file of N lines with N characters were to be considered, the complexity would increase to O(N^2). A file of N lines with N characters would be the worst case scenario, having the maximum number of lines and characters per line being considered.

Since the stacks are reset for each line, the space complexity would remain the same whether the entire file was part of the run time, or just a single line. The space complexity would consist of the two stacks created, plus the memory allocated to store the temporary variables and other class variables.