**Recursive Prefix to Postfix Analysis**

**Brandon L Morrow**

**Recursive Analysis**

Recursion can be a very useful tool when converting prefix to postfix expressions. I was able to quickly code a recursive prefix to postfix conversion that was a much more concise solution than with stacks. This program demonstrates the power of recursion, and although powerful is something that I would use in moderation in the future due to its complexity when debugging. I believe a joint approach using iteration and recursion to be most useful in the future.

This program processes the file to a postfix expression as it reads the file. The readLine method processes each line to postfix using the preToPost method, and checks for errors in the postfix expression using the error checking method. The readLine method is recursive and reads through the entire file. An added feature is that it is compatible with Windows and Linux files. This is because linux files end with \n and windows \r\n. The code is concise, but extremely difficult to debug.

I was able to quickly get the prefix to postfix conversion method running, but initially I was not able to debug the code properly to get it to read the entire file. It took quite a bit of time to debug and get the program running for the whole file. I eventually found a way to read every line using recursion. I was made very much aware of one of the biggest shortfalls of using recursion – debugging. What I thought would be a one-day fix turned into a multi-day headache trying to follow the input stream through the methods. When the method calls itself, and then other methods within itself, it becomes really confusing working through the debugger. It took days to get the readLines method correct using a recursive method to read through the entire file.

Error handling is also more difficult using recursion vs stacks. Stacks are easy since during the conversion if there are any remaining elements in the conversion stack then it is an invalid postfix/prefix expression. Strings are harder to validate since you must build a separate method to check that there is an operator at the beginning/end, and you must parse through the string to compare the number of operators and operands.

In order to confirm my findings for big-O, I built a test into the program for the time it takes for execution. I started the timer at the beginning of the program and stopped it at the end of the program. This confirmed my finding that the complexity of the program is O(N) which can be clearly seen at higher lines of input. I have attached a comparison of iterative vs recursive O(N). We can clearly see that both are O(N) time as expected. While both are equal in time they are not equal in memory management.

The stack forces the user to clear out the accumulator while the recursive method makes it difficult to find a way to clear out the accumulator to avoid memory issues with large input files. The stack allows for the strings to be stored inside of the data structure, while the recursive call stores the postfix expression as a accumulated string. While there are solutions to avoid memory issues it is more difficult when compared with stacks.

|  |  |  |
| --- | --- | --- |
| **Iterative Runtime O(N) Analysis** | | |
| **Lines Prefix** | **Time (msec)** | **Ratio** |
| **100000** | **550** | **0.0055** |
| **1000000** | **4228** | **0.004228** |
| **2000000** | **10304** | **0.005152** |
| **3000000** | **13938** | **0.004646** |

|  |  |  |
| --- | --- | --- |
| **Recursive Runtime O(N) Analysis** | | |
| **Lines Prefix** | **Time (msec)** | **Ratio** |
| **500** | **16** | **0.032** |
| **1000** | **36** | **0.036** |
| **1500** | **47** | **0.031** |
| **2000** | **62** | **0.031** |

What I Learned

I have a much better understanding of how recursion can be used instead of iterative loops. This also demonstrated the power of recursion as I wrote the preToPost conversion method almost immediately after starting the project with only a few lines of code. I was shocked by how much easier it was to use recursion for something like this. The rest of my time coding was to fix a bug that was causing the rest of the file to not be read.

I learned that using both iterative and recursive techniques together can be the best way to code certain problems. When I first started coding I quickly thought of a recursive way to convert a prefix expression to postfix, but was having difficulty finding a recursive way to get the program to read the whole file. I ended up finding a recursive way to loop through the whole file but would have been an extremely easy to code had I been able to use an iterative loop of the preToPost method for each line.

I also learned that recursive methods can have memory problems with the system. Unlike with a stack you have to manually clear the accumulator so that there are not memory issues for large input files.

I have truly seen that big O notation is a pretty accurate representation for runtime efficiency. I have also seen that the recursive method is O(N) complexity like it is with the iterative method. With some cases there are no negative effects of using a recursive method vs an iterative, and there can be benefit to recursion by cutting down the lines of code required.

What I Might do differently Next Time

I need to practice debugging and to stop to make sure that I understand what is happening with the input sequentially. I had moments where I was not sure what the errors were, and I should have slowed down more and thought of a solution before trying to implement a fix. I caught myself at times guessing a solution instead of really thinking through what was causing the error. I ended up writing everything out very slowly and was able to fund a solution.

I will make sure that I write out on paper more conceptually what I am trying to do. I knew what I wanted to do in my head, but I had to do more trial-and-error coding than I would have liked. This is exactly what happened last project, but I did improve in that I made a good initial plan. I actually got the initial code correct like I predicted but did not go back to writing it out when I started the debugging process. Had I gone straight back to the conceptual phase it would have drastically sped up the process.

Justification for Design Decisions

There are three classes in this problem. These classes are readLine, preToPost, and a method to validate the postfix. The main method for Project 2 implements the readLine method to reduce clutter in the main and compartmentalize the conversion. This provides a clean separation between the specifics of the project assignment and the other methods that could possibly be reused in other problems. The readLine method is designed to read all of the lines, and can read both Windows OS and Linux OS files. The preToPost does the conversion, and the validate postfix validates that the output is correct.

This design is efficient in that it allows additional methods to be easily built and implemented. A postfix to prefix class could be implemented and tied into the readline method. A validate prefix method could be implemented to validate the prefix before the conversion. It is compartmentalized in a way that allows for the easy addition of other features.

Issues of Efficiency

The data is processed as it is read from the file so the access time of the filereader is O(1) and the recursive loops make the complexity O(N). The runtime directly scales with the size of the file.

Recursive methods can have memory problems whereas stacks are easier to use to avoid memory problems. Using recursion, you have to manually clear the accumulator so that there are not memory issues for large input files. This can happen with stacks but is easier to manage since instead of an accumulator string you are pushing each line into a stack. Again, memory problems are avoidable but take a more conscious effort.

This program processes the file to a postfix expression as it reads the file. It does this by passing the inputstream to the recursive method preToPost which then rearranges the character stream to a postfix expression. This is then recursively iterated through each line using the readline method. Generally, O(N) appears to be the best-case scenario for this program when processing the prefix data from a file. The runtime scales proportionally to the amount of data in the file.