**Brandon L Morrow**

**Pre to Postfix Conversion Using Stacks Analysis**

**Completed: October 04, 2022**

**Project 1 Analysis**

The stack data Structure seems to be a very useful tool in the analysis of prefix, postfix, and infix expressions. While there are other techniques that are easier to program, they are often times not as dynamic as the stack data structure. This program demonstrates the use of a stack to convert from prefix to postfix expressions from an input file. The stack class was overridden with my stack class, and is not the built in java class.

The stack itself is implemented as a generic java class/object. The stack class contains a node inner class that is used to store node data objects inside the stack. The node objects are used to store information inside the stack and when items are pushed into the stack it clears the next node for a new input. This stack is designed in a way that it could be used for other projects in the same way that the built in Java class can be used for other projects

This program processes the file to a postfix expression as it reads the file. It does this by placing each character into a stack, and then processing the stack for each line. This allows for error checking to occur inside the conversion method for each line to ensure that a valid prefix expression is being converted to a postfix expression. An iterative approach was taken as opposed to a recursive method using a while loop. A recursive method would have cut down some lines of code, but not significantly enough to show a real benefit over an iterative approach.

In order to confirm my findings for big-O, I built a test into the program for the time it takes for execution. I checked the program runtime at the midway point after the input stream, and then again after file output. This confirmed my finding that the complexity of the program is O(N) which can be clearly seen at higher lines of input.

**Runtime O(N) Analysis**

|  |  |  |
| --- | --- | --- |
| **Lines Prefix Input** | **Time (msec)** | **Ratio** |
| **100000** | **550** | **0.0055** |
| **1000000** | **4228** | **0.004228** |
| **2000000** | **10304** | **0.005152** |
| **3000000** | **13938** | **0.004646** |
| **4000000** | **19594** | **0.004899** |

What I Learned

I have a much better understanding of the stack data structure and the nature of the nodes. I implemented the node subclass after a lot of thought and a hint from the office hours. This forced me to really think about the object-oriented side of the data structure and how the nodes need to be organized.

That type conversion can be difficult to handle when creating and using data structures. An example is that I had to push the input stream into a character stack, but then had to convert it to a string stack for the actual pre to post conversion. This was somewhat confusing at first because with pseudo-code previously I never had to think about what data type I was placing into the stack. Generics were not the magical solution I had hoped they would be when I first created the stack class.

Like the example in Project 0 stated I now have truly seen that big O notation is actually a pretty accurate representation for runtime efficiency. I was surprised that after coding a timer into the program that the complexity came out exactly as I expected. I expected there to be more variance.

What I Might do differently Next Time

What I will do differently is that I will spend more time writing out the methods required. I should have listened to the classic warning of “do not dive into coding without writing out good pseudo-code”. I did a decent job of preparing what methods I wanted to use, but I should have typed out more detailed pseudo code before going straight into programming.

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. An example is that I had the node class separately and not as a subclass of Stack. This just caused me some issues when I was trying to call the next Node and lead to me confusing myself. After a lot of trial and error and a lot of wasted lines of code I narrowed it down to an inner class and cut down the methods to only what I needed.

I would think about error handling more in depth and include it in my initial design decisions. I initially coded the pre to postfix method and had it working, but I immediately noticed that in the output file there were invalid prefix expressions. These invalid prefix expressions were causing the extra operator to have a line of its own. This forced me to rewrite how I handled the conversion which took almost an entire day. Eventually I was able to figure out a solution by analyzing one line of characters at a time from the file, but if I had taken the time to think about more user errors I would have known to do that from the beginning.

Justification for Design Decisions

There are three classes in this problem. The main method for the Project 1 application is used to outline the solution algorithm while hiding the details in other methods as much as possible. This provides a clean separation between the specifics of the project assignment and the other classes that could possibly be reused in other problems.

The generic Stack class overrides the Java built in class and is used to store characters in the prefix expression. The Node class is an inner class inside the Stack class and is used to store the objects pushed into the stack.

The Main class contains the main method, and contains the methods preToPostString, and isOperator. The preToPostString method takes an input prefix character stack and converts it to a postfix string. This string is then returned to the main method so that it can be pushed into the output stack. The isOperator method is used inside the preToPostString method.

Issues of Efficiency

A stack is a relatively straightforward way to calculate the postfix. The access time to the stack is O(1), so the overall problem solution is driven by the data file itself, not the data structure.

This program processes the file to a postfix expression as it reads the file. It does this by placing each character into a stack, and then processing the stack for each line. It could be made more efficient by doing only doing the error checking during the reading of the file, and then processing the entire character stack to a postfix expression. Generally, O(N) appears to be the best-case scenario for this program when processing data from a file. The runtime scales proportionally to the amount of data in the file.