Christian Znidarsic
EN.605.202 Data Structures
3/1/22

Lab 1 Analysis

Data Structure Description

The objective of this lab was to use a stack data structure to convert prefix strings into postfix strings directly. I used a stack because it is a quick and efficient method for reading in characters from an input file, and performing operations on those characters one at a time. This problem lends itself to the implementation of a stack because the code works out to be quite simple when a stack is used. The Last In First Out nature of stacks provide two main advantages in this problem:

1. The most recently concatenated strings on the stack are available using a simple pop() command
2. The order of chars can be reversed quickly and efficiently

For this problem, a stack makes sense because when a char is read, it is put directly onto the stack with O(1) efficiency, and when a char is needed, it is removed with O(1) efficiency. No searching is required to access relavent data from the stack as it would for a simple array.

Iterative vs. Recursive

This problem can be solved with either an iterative or a recursive solution. I implemented an iterative approach, which is likely more efficient than an equivalent recursive solution to the same problem. Iterative solutions are generally more efficient that recursive, and use less memory.

For example, my program operates at O(n) space and time complexity. A recursive solution would likely exceed O(n) due to redundancy and the large space requirements of numerous recursive function calls.

A recursive alternative, however, may look simpler in writing, and may be more readable than my current iterative solution.

Design

My program works using three classes: main, preToPost, and Stack.

Main

The main class is simple, and asks the user for input in the form of text files, creating relevant BufferedReader, Scanner and PrintWriter objects accordingly. It then passes these objects as arguments into the static method preToPost located in the PreToPost class. All of this is done inside of a try-catch block to catch FileNotFoundExceptions when the user enters an invalid file path.

Stack

The stack class is the class that specifies Stack objects. It includes the fields int arraySize, which specifies the starting array length to be 100, int head, which specifies the array index of the head of the stack, and String[], which is the array that holds the values in the stack. I chose to implement an array in my stack because it seemed to be the most simple option. Another option would be to implement a linked list using nodes. Stack also includes the methods push(), pop(), and peek().

PreToPost

The PreToPost class is the largest and most crucial class in my program. It takes the BufferedReader, Scanner and PrintWriter objects in from Main. Generally speaking, it takes an input text file containing prefix strings, and it outputs corresponding postfix strings to an output text file.

First, a while loop implements Scanner to copy over all of the text from the input file to the output file. Next, the large while loop implements BufferedReader to iterate across every character in the input file. If the character is an operand, it is pushed onto the stack. If it is an operator, the top two items on the stack are popped, and concatenated together with the operator, and the resulting string is pushed back onto the stack. This process repeats until there is a single string in the stack. This resulting string is printed to the output file.

Other methods in PreToPost are isOperator, which determines if a string is a valid operator as specified in the lab spec, isOperand, which determines if a string is a letter, and isSpaceOrTab, which is used to exclude characters that are spaces or tabs.

Justification

To me this ordering made sense because main is quite short and only performs user I/O and very general operations like calling preToPost.

Stack is a completely self-contained class that has all of the fields and methods needed to implement a stack data structure. Stack objects can be created and manipulated cleanly, and relevant fields and methods that don’t need to be accessed outside of the class are private.

PreToPost is then the most complicated class, and performs most all of the arduous work of performing operations on the input file. It calls Stack but remains separate from it and only uses Stack objects.

Errors Handled

Spaces and tabs

Spaces and tabs are excluded by an if statement before they even have a chance to be pushed onto the stack. This is important because the user may accidentally include spaces or tabs without knowing it, since they may not be easy to see in a text editor. Thus, they are omitted outright upon reading them from input and not processed any further.

Invalid operands

The isOperand method takes a string as an input and determines if it is a valid operand. In this case, valid operands are only letters. Thus, isOperand uses Character.isLetter() to determine if the input is a letter

Invalid operators

In the lab spec, the valid operators are defined to be "+", "-", "*", "/", and "$" for exponentiation. The method isOperator takes a string as input, and outputs true if it is one of these valid operators using a simple switch statement, and false otherwise. This method is implemented in the code, and if it returns true, then two items are popped from the stack, added to the character, and then pushed back on.

Integers

The lab spec lays out that operands are only letters. Thus, integers are not valid operands. They are excluded by default by isOperand because Character.isLetter(char) will return false.

Other Non-Letters Or Operands

Other characters that that are not letters *or* operands are excluded.

Invalid prefix expressions

Invalid prefix expressions will result in either an empty stack, or a stack with more than one item at the end of the main while loop. This is checked for in an if statement, and if either of these two conditions are met, then an error message is output in place of the postfix expression, indicating that the prefix expression was invalid.

FileNotFound exception

When the user enters an invalid path to the terminal, a FileNotFoundException will arise. The main method implements a try-catch block to catch any FileNotFoundExceptions that are thrown. The exception is handled smoothly by requesting the user to re-enter the input file path.

Test Input.txt Key

The input .txt file tests for error handling in the following order:

1. Control (no errors, valid prefix expression)
2. Integer inserted in valid expression
3. Multiple integers in valid expression
4. Only integers
5. Spaces within otherwise valid expression
6. Spaces within and outside of otherwise valid expression
7. Spaces and tabs within and outside of otherwise valid expression

8. Empty line
9. Empty line with spaces and tabs
10. Invalid operand inserted within otherwise valid expression
11. Multiple invalid operands inserted within otherwise valid expression

Enhancements

When an invalid prefix expression is processed, it will either result in an empty stack at the end of the while loop, or a stack with more than one element. After the main while loop, an if statement checks to see if either of these conditions are met. If one of them is, then in place of the corresponding postfix expression, an error message is printed in output that says: "Invalid prefix expression or invalid characters. Valid operators are +, -, *, /, and $."

This enhancement not only accounts for invalid characters in the input file, but it also runs a check to see if the prefix expression entered is valid. This is especially useful because the user may not see that the expression they entered is an invalid prefix expression. It is not intuitive to see, but this enhancement will tell the user if they entered an invalid expression.

Efficiency

Time Complexity

The only part of my program that is not O(1) is in the preToPost method. In this method, there is a while loop that iterates across every character in the input file and pushes them onto a stack. Within that while loop is a second while loop that will eventually pop all of those characters from the stack by the time the program is done running. Thus, the second while loop iterates over all of the items that the first while loop iterates over. This means that the program is approximately O(1) + O(2n) = O(n).

Space Complexity

Each character read in from the input file is first stored in the preStack, and is then pushed into the main stack. The rest of the space requirements are variables of O(1). Because every element from input is stored in two separate stacks, each element must be stored twice, resulting in O(2n). In total, the space complexity for my program is O(1) + O(2n) = O(n).

What I Would Do Differently

If I were to do this lab again, one of the main things I would like to implement would be differentiating to the user what the error was with their given input. As it is right now, my program returns the same error message for when an integer is entered in a prefix expression, and when an invalid prefix expression is entered. The reason for this is because when an integer is included in a prefix expression, it is read and discarded before being pushed onto the preStack. Thus, when an integer is discarded, often times the resulting expression will be an invalid prefix expression because it is likely that the user mistakenly entered an integer instead of a character or operator. When the digit is removed, it leaves an incorrect expression.

I tried for a long time to implement a third stack, into which incorrect characters would be pushed. I got it to work, however the output was having some issues. It seemed to pick up characters beyond what I expected. The output seemed to have extra newlines so results were all indented. To me it seemed like "\n" newlines were being pushed onto the stack, and when the contents of the errorStack were printed, newlines were printed as well, causing incorrect spacing in the output. I tried for a long time to filter out newlines from being pushed onto the stack, using if statements and dedicated methods specifically to determine if a character is a newline. But despite my best efforts, I could not get rid of the problem. I thought maybe they were "\r" lines as well, but whenever I ran a check for them, the program wouldn't work at all.

I wanted to include uniquely taylored error messages to represent these various user input errors, but I was unable to get this feature to work in the time I had for this lab. This would be my priority if I were to re-do this lab, in order to enhance the user experience.