Steven Lee
Lab 1 Analysis - Postfix expression interpreter

In order to accomplish the requirements of this assignment, I used a stack data structure with an array implementation. I used a stack because I needed a last-in-first-out order of items, and a stack provides this. Additionally, when I processed an operator in a postfix expression, I could consume the previous variables, and discard them after the operation. This meant that the pop method offered by stacks worked well. The push operation also fit this project because I only needed the topmost operand at any given time, and I could hide the operands beneath the topmost operand. Finally, the isEmpty method provided good error checking, because a correct postfix expression will never result in an empty stack. The reason that this is the case is that, in my program, at least two letters need to be on the stack before the first operator appears. If an operator appears, and the program tries to pop two operands, but the stack is empty, then I know that the input is invalid.

In terms of implementation, I thought an array was the best choice for implementation because it offered several benefits over other implementations, and the drawbacks were not too bad as tradeoffs. First, the overhead for declaring an array is an O(1) operation, because there is one declaration, and each new element in the structure is added to the existing structure. Additionally, I wanted the stack to be stored in the same place in memory because each operator results in two consecutive pop operations, so locating two elements next to each other is faster than locating two elements based on their pointers. For the drawbacks of array declarations, several weaknesses were irrelevant, or mitigated by the specific requirements of this project. In terms of memory required, the array will not result in wasted space in this particular program. This is because, for valid postfix expressions, I know that the maximum size of the array needs to be 1 more than half of the length of the postfix expression. Postfix expressions have one more operand than operator, meaning that the expression will require a maximum size which I can know prior to the declaration of the stack. One tradeoff is that this array size is only needed for certain types of postfix expressions, where all of the operands are declared, then the operators are declared. In other types of postfix expressions, where operators are mixed with operands, there would be some wasted space in the array declaration. Finally, although arrays are slow in dealing with insertion and deletion of elements, this was not a necessary condition for the array, as I could move the header rather than shifting the entire array. This results in what is essentially random access, since I am just giving an index to the array.

For appropriateness of this data structure to the application, the stack interface provides a very good layer of abstraction necessary to process postfix expressions, and it is very helpful to be able to use the push and pop operators in order to manipulate the expression and print the correct results.

My design description is as follows: My main method is declared in the public class Project1. Inside main, I declare my file paths, readers, and writers. I call my other public classes in order to perform the correct manipulations on the given input, and then close the reader and writer. I also catch input and output exceptions in main, and print them to the logger. This works well because my main method specifies where the input and output are located. Since initial input and output locations are necessary in order to run the rest of the program, it makes sense to put those in main. Additionally, if I need to change how the output works or where the input is located, I can do that within the main method without changing any aspect of how the input is processed.

I declare a stack interface in order to specify what a stack needs to do. This is a helpful quality check to make sure that any stack implementation specifies all of the necessary methods and returns the correct values. Since my array stack implements the stack interface, I know that I can use the standard stack operations in other parts of my program, or my program build would fail. The interface specifies standard stack methods; isEmpty(), peek(), push(), and pop().

The public class ArrayStack implements this Stack interface, and writes definitions for each of the stack methods using an array. This class is used for actually declaring stacks, and using them in other parts of the program. The actual array implementation of the stack is located in this class.

The public class RegisterPrinter accepts the postfix expression from the input, and performs the necessary transformations to output a full list of operations which would take place on a postfix expression. The register string variable holds the formatted string of what the output will be, and this is later printed to the output file using the getRegister method. Since we know the length of the input expression, we create an arrayStack object with the maximum length of half of the postfix length plus one. Finally, we create the expression in the constructor, because once we have the postfix expression, we will want to be able to immediately store the necessary output. In this class, I perform error checking using the invalidString() method. I think that doing error checking prior to actually attempting to use the stack or write the output is good because I do not want stack overflow or underflow to happen, and I do not want invalid operations to be written to the register. By checking if the input is valid first, I know that later manipulations will work. The operationWrite method is called whenever an operator is encountered in the postfix expression. This method is responsible for taking the given operator, retrieving the operands, and writing the correct operations to a temporary string. This string is then added to the main register when operationWrite returns its value. I think that having a separate operation for each operator is a good idea because each operator requires several operations in order to print correctly, and encapsulating this in a method makes later debugging easier, and makes the writing operation more reusable in different contexts. Finally, the writeRegisterOperations method returns the full register of operations which would be performed on a postfix expression. The result of this method is what will be written to output on calling the getRegister method. This method iterates through the postfix expression, pushes operands to the stack, and calls operationWrite when it encounters an operator, and this results in the full list of operations being correctly written.

In terms of time efficiency, the least efficient part of my program is the writeRegisterOperations method. This is because the method iterates through the given postfix expression twice. Additionally, the invalidString method contains a nested for loop. However, the inner for loop does not grow with operator size, as it is looping through a fixed list of operators, and is only called if the given character is not a letter. In one case, the input character will not be a letter, and will not be in the list of valid operators. This will result in early stopping because the boolean variable validOperator will never be set to true, but it will still result in iterating through the entire list of operands once. The worst case runtime for the invalidString method will occur if the string is in valid postfix notation which will run in linear time, or O(n). For the writeRegisterOperations method, the worst case runtime occurs when the string is valid, which calls a for loop which will iterate through the length of the string. This will run in linear time, or O(n) as well. The operationWrite method calls a series of constant time operations which are not dependent on the size of the input, so it will run in O(1) regardless of the input size. For this reason, checking if the string is valid and writing the operations to the register runs in O(2n) time. Since only the degree of the nth term is considered, the program should run in O(n) time, with α of 2.

Over the course of this assignment, I learned about how to implement a stack correctly, and the difficulty associated with design decisions in the stack. Initially, I thought I could implement an ArrayStack of type ‘char’ rather than a String, but this did not work because I needed to store the word TEMPn in the stack. I think I could have thought more precisely about what the assignment was asking in this case, because what I needed to do was write operations to a string, not perform arithmetic in postfix notation like I initially thought. In order to finish this type of assignment faster next time, I need to consider what the output means, and how to supply types to perform that output. I also learned a lot about edge cases, mostly by writing invalid input strings. These considerations taught me a lot about how to ensure that user input is valid, and the importance of taking the perspective of a user when designing an application. While making sure that a string is valid seems to be trivial, I ended up spending a larger portion of time than I expected on this part, because I would find other ways to trick my program into accepting invalid input and printing incorrect results. In the end, my error checking function ended up being more complex than I anticipated, so in the future, I will try to come up with a wider range of possible inputs prior to writing code, so I know which errors to detect.

In terms of what I would do differently next time, I think there is a way to get more efficiency by pushing all characters from the postfix expression onto the stack, and then processing the input one character of the time, doing error correction while processing the stack. This would simplify my program by letting the stack be composed of characters, rather than strings, and it would remove the need for a separate pass over the input in order to do error checking. However, this would also result in additional work being done which is not currently necessary in my implementation. For example, if the input is a prefix string, my error checking method finds this very quickly, and can stop processing the rest of the string, and this can be done without ever needing to process the string as though it were postfix. If the entire prefix string were pushed to the stack, this would result in unnecessary operations. I also could have tried having two stacks, one for the characters in the input, which could be a character stack, and a stack of string type for storing temporary variables. This could have provided its own form of error checking, because at the very end, there should be no remaining characters in the character stack, and one remaining “TEMPn” variable in the string stack.

As an enhancement, I display the current string to be processed, so it is easier to understand the operations being performed in the output. Being able to see the current input which is being processed by the program is helpful for human readers, who can verify the operation more easily. After the operations are displayed, I also post the execution time for that operation. This provides interesting comparisons between the time complexities of various operations. For example, the operation completes most quickly in rejecting prefix expressions as invalid. This is most likely because the first character is an operator, which immediately invalidates the whole string. It also seems that longer strings take more time even if they are invalid. I think this is because I get the string length during my program, and getting this value takes longer for longer strings.