605.202: Data Structures

Lab 1 Analysis

Renee Ti Chou

Due Date: February 28, 2017

Lab 1 Analysis

Justification for Design Decisions

For this assignment, the data structure used is a stack, which is simple and suitable to hold the operands. The stack is implemented by a linked structure, which cannot actually store heterogeneous items.

I have tried to make the node class to store both string and characteristic values; and utilized the overloading character to write the *push* method that takes a string as well as a character as the parameter. However, the complier did not allow to write an overloading *pop* method that returns a string or a character. Thus, I chose to store string values in the stack because the TEMPn variable can only be stored as a string, and it is easier to convert a character to a string data type.

For the *pop* method, I used the runtime *NoSuchElementException* because it can print out the error message that indicates stack underflow. The stack does not have *peek* method because it can be conduct by the *push* and *pop* methods. The additional *cleanUp* method is useful in the main method in the Lab1 class, where only one line of code is used to empty the stack.

The Lab1 class conduct the main conversion of the postfix expressions. I added an extra instruction which would handle the $ symbol as exponentiation. I also set a rule that only single letters would be accepted as valid operands, and this includes lowercase and uppercase letters. The lowercase letters in the input would be automatically converted to capital letters in the output. In addition, the unintended spaces appearing in the postfix expression would be ignored.

There are mainly three types of error messages for the program. The first one is "*Insufficient operand(s) for:* " followed by an operator. The second is "*Invalid symbol:* " followed by the particular symbol. The last one is "*Insufficient operator for:* " + b + " and " + a, where a and b indicate the operands that cannot be evaluated due to loss of an operator. The operands here include TEMPn variables.

The error messages may be printed out in the middle of the conversion, so there could be an incomplete set of instructions terminated by the error message. This helps backtrack the conversion, which could be useful for the user to figure out, for example, which operator does not have sufficient operands if there are two operators of the same type in the postfix expression.

The stack implementation

In this assignment, the implementation I chose for the stack is the linked structure. The advantage of the linked implementation is that it has no size limitation because of dynamic allocation. Although this structure forces sequential access, there is always a top reference points to the first item in the stack,

and the stack can only pop out one item at a time form the top, so it is not a big issue in this case. However, it requires homogeneity, which is not so convenient when storing both character values that are read from the string line and the TEMPn variables generated by the program at the same time.

Use of a stack for Lab1

A stack can be useful for this postfix expression problem because it is a simple data structure, and it is useful to store the operands. During the evaluation process, only the two values right before the operator are conducted by the operator, and the stack can help temporarily hold the operands until an operator is encountered, which cause the stack to pop out the two most recently stored items for the operation. The evaluation process displays a LIFO order, and thus using a stack could be reasonable.

Potential recursive solution

To solve the problem recursively, we might want to read the postfix expression from the right. Take ABC+*CBA-+* as an example, the first operator encountered would be the multiplier, but since its potential operands still are in the unevaluated forms, the recursive function is called in order to solve the smaller piece of problem. The process continue until the stopping case is met, and that is the situation when the operands are real values that can be directly evaluated.

Comparing to the iterative one, the recursive solution is not so intuitive because it breaks down the problem first rather than directly evaluates from the small expression, which is BC+ in this case. The recursive solution appears to be less space-consuming because the number of the operators is always one less than that of the operands. The recursive solution puts the execution on the system stack and call for the recursive function when the operator cannot operate the postfix expression. However, the difference is so minor that the space issue may not be viewed as the advantage of the recursive solution.

I personally prefer the iterative solution more because the output format and the error messages for the recursive one could be confusing to the user. It may print out the instructions from the middle of the postfix expression, and throw out a message when meeting an error. This could be difficult for the user to backtrack the instructions. After all, the user does not know the actual implementation behind the scene. On the other hand, if the programmer would like to provide lucid error messages, it probably would take much time to address this complex issue.

What I learned

I learned from the example Project 0 how to write a formal documentation for a source code, and how to read and write from named files by using arguments passed from the command line. I also learned how to design the pop method that would throw out an unchecked runtime exception.

Another important technique that I learned is to module the code. As I was addressing the execution which generates the instructions, I found that I kept modifying all the code blocks for the different operators even there is only minor alteration. I realized that this would cause problem if I forgot to fix the code in somewhere because of the redundant work. Thus, I decided to isolate this repetitive piece of code and make it an independent module from the main method.

What I might do differently next time

I might write an array implemented stack next time. I may found that there would be no difference when using the stack in the main method, though the implementation in the stack class is different. The code may be simpler because of the advantage of random access. Also, there is no private class needed for dynamic allocation. I might have to guess in advance how much space is enough for the program, but it should be easy to guess because the postfix expressions usually are not too long. Last, I might have to handle the situation when the stack is overflow.

Another thing I might do differently would be writing a generic stack. Although the generic stack is in essence still homogeneous, it can be widely used in different situations. In this way, I will not have to write an additional stack for another data type. The only thing I have to do is specifying the data type in the declaration of a new stack.

Issues of Efficiency

The time cost to each of the standard methods (*pop*, *push*, and *isEmpty*) in the stack class is constant time O(1). The *size* method also cost constant time, but the cost of the *cleanUp* method is O(n) because of the while loop within the code. Overall, the stack methods will not cost much of time in the program.

The time cost to the Lab1 class in the program mostly depends on while loop which reads in the input data by line, and the for loop that parse the line string into characters. The sum of the cost to the two loops would be about the total characters in the input file, so it is still linear time cost O(n). The *push*, *pop*, and *cleanUp* statements in the main method could also add some cost of time, which may change the constant value of the order in the actual performance, yet overall the program has a linear time cost.

The space cost in this program is mainly based on the size of the stack, while most of other pieces of code have constant space cost O(1). The stack stores operands and pops them out when there are operators read in. The space complexity thus depends on the average of stored items in the stack. For example, the valid expressions could be AB+, ABC++, ABCD+++, and so on. The space used would then be 2, 3, 4, … n units. Therefore, the average space complexity would be approximately O(n/2).