605.202: Introduction to Data Structures

Christine Herlihy

Lab #1: Analysis Paper

Due Date: October 6, 2015

Dated Turned In: October 6, 2015

Christine Herlihy
EN 605.202 Data Structures- 10/6/2015

Lab #1 Analysis

I. General Commentary

This project included four primary activities: (1) reading in a set of postfix expressions from an input file; (2) processing each expression using a stack (and generating exceptions as warranted for invalid input); (3) generating representative machine language at each step in the pseudo evaluation process; and (4) sending this language to the output file in a clean and easy-to-read format.

To approximate the Stack class without using Java library functions, I began by creating a MakeStack interface. This helped me ensure that my LinkedStack class would contain all of the necessary methods. I then created a parent class called StackNodeElement, where each StackNodeElement has a string value (i.e. "A", "B" "+"), and a DataValue, set to 1 by default. I also created a StackNode sub-class, where each StackNode object has a String value and a pointer to its next neighbor in the stack. I included the String value so that the StackNode objects could be displayed as upper case letters and operator symbols. This was a necessary step for me to be able to push postfix expression elements onto a stack containing StackNode String values.

I had StackNode inherit from StackNodeElement so that I could implement mathematical logic checks even though the postfix expressions we were required to evaluate did not contain numeric characters as valid input (i.e., I wanted to be able to demonstrate that my division and modulus machine language commands would check to ensure the data value of the operand in the denominator was not equal to 0 before performing the operation and pushing the resulting (undefined) value into the register). Such logic could not be implemented given the format of the input (i.e., the operands are not assigned data values), but the foundation is there and could be elaborated on in future iterations.

I then used the MakeStack interface as a guide to design my LinkedStack class, ensuring that all the requisite methods were implemented (i.e., getSize(), peek(), pop(), push(), isEmpty(), and a customized stackToString()). Next, I used a BufferedReader to read in the postfix expressions from a txt file into a String array, such that each new expression constituted one element of the String array. I also stored a dummy row containing the "#" symbol as a dummy flag in-between each expression in the String array. I then parse each element of the String array, checking each element within each String to see if it is a capital letter (i.e., valid operand), operator, or invalid input. If the element is a valid operand, I push it onto the stack; if it is a valid operator and the stack has at least two operands in it, I pop element A off the stack, pop element B off the stack, perform B op A, and then push the result back onto the stack in the form of a temporary variable. I store each machine instruction line in the output array, and eventually print this output array to the output file. I included additional input expressions.

II. Justification for Design Decisions

A stack makes sense given that our goal is to accept a postfix expression as input, evaluate each element of the postfix expression, generate machine language evaluation instructions, and output these instructions to a file. To perform each of these operations, we are only concerned with the type of item that comes next in the postfix expression (i.e. is it valid, and if valid, is it an operand or an operator?), and thus, the push() and pop() functions of a stack can help us achieve our end goals.

I chose to design my stack using a linked implementation than an array because I did not anticipate that a search method would be needed, or that a search would be performed in the process of converting an individual postfix expression—in fact, the sequential nature of the postfix evaluation process essentially precludes such action. Thus, I was satisfied with sequential access, and not terribly concerned about forgoing the random access capabilities of an array-based implementation. I also opted for a linked implementation because I sought to capitalize on its dynamic allocation with respect to stack size. I wanted to feel free to add several additional input expressions without being concerned about stack overflow.

While my solution is iterative, I did consider a recursive solution. The recursive action you are taking in the iterative solution is the "popping" of the top two elements of the stack each time your parser identifies an operator. Your stopping case would be when there is one element on the stack; if you had one element as input, you would simply return this input, otherwise, you should pop the two elements and perform an operation (provided, of course, that the expression contains a valid and correctly placed operator). To me, it seems that the chief advantage of the iterative solution in this particular situation is that it is less redundant (although this could be mitigated by optimizing for recursion), and easier to de-bug, since it gives you line-by-line results.

III. Lessons Learned

Designing and implementing this project helped me to better understand how a stack ADT translates to a Linked List implementation. Before doing this project, I didn't fully appreciate understand that a stack element's String value and its reference to the next element in the list needed to be stored separately as different types of objects, or how each value could be useful in its own right. Adding in the data value so that I could perform mathematical logic checks further complicated this task.

I also had trouble getting the stack to correctly display the name of a variable as TEMP*n* once an operation had been performed and the temporary variable needed to be displayed within the register. It took me time to realize that I needed to manually push TEMP*n* into the stack in order for it to be displayed as an operand the time a value was popped, and that I would also need to push only a subset of the variable, since I wanted to push TEMP*n* without its previously associated command (i.e. AD, SB, ML, DV, etc.).

Being forced to carry out operations with a stack without using library functions was a good learning experience—it made me appreciate the functional aspects of the Stack class that I might otherwise take for granted. It also made me appreciate the fact that in standard Java, the Stack class is implemented using a vector, rather than an array—you essentially get the benefits of dynamic allocation (with respect to size), along with the benefits of random access/indexing that you get with an array.

IV. Alterations to Consider in Future Iterations

One feature that I considered including and would hope to add in future iterations of this project would be a method to convert the original postfix expression to an infix expression. This infix expression could then be displayed alongside the machine language commands to make the original expression and the resulting output more comprehensible and familiar for the end user. Another modification I would like to make would be to allow the user to assign integer values to a series of capital letter operands (i.e., let A =1, B= 2, C=3, etc.) and receive as output the machine language instructions with values plugged in, alongside the mathematical result of the operation. One final alteration that I think would make the output more readable is to color code the operands in an intuitive way (i.e., A shows up as red, B shows up as yellow; their result in the stack is orange. Given the limited range of primary colors, however, a color gradient might make more sense for larger postfix expressions.

V. Issues of Efficiency

The use of a stack made this process efficient: the cost of each push and pop operation is O(1), since no query of the stack is performed, and one value is either added to or removed from the top of the stack. Thus, the cost to evaluate a given postfix expression is largely a function of how many operands and operators it contains, and the cost to evaluate a set of postfix expressions depends largely on (a) how many expressions you are evaluating, and (b) how many valid operands/operators each contains. Ironically, invalid and/or undefined input speeds up the process, since the evaluation process breaks

upon encountering invalid input (rather than continuing to evaluate each remaining element), and proceeds directly to evaluate the next postfix expression in the input file.