Koon Chua
EN 605.202.81
7/3/2018
Lab 1 Analysis

I. Introduction:

In Lab 1, our objective was to classify a given string into one or more languages L. This consists of a few key steps:

- Iterating through an input file where each line is a string.
- Creating our own stack ADT to be used for language classification
- Processing each string using stack manipulation to identify language patterns
- Classifying each input string and writing the results in a cogent format for review

With the objectives clearly laid out, I created four classes to tackle each individual sub-problem. ReadWrite combined both the reading/writing processes into a single class. It uses Java's built-in File class to open the input and out files, Scanner class to parse through the input file, and PrintWriter class to write and close the output file. The output file is identical to the input file, but with the results of the language tests concatenated to the end.

The Stack_LL class contains a linked-list implementation of a stack. It contains the requisite methods: isEmpty, push, pop, peek, empty, and getSize.

LanguageEval is where all of the stack manipulation and language classification occurs. By creating auxiliary stacks to hold all instances of 'A' and 'B', we can create methods to measure the occurrences of each without explicitly counting them.

The main driver class Lab1Main initializes instances of these classes, and also captures errors due to invalid input/output files or incorrect parameters.

II. Implementation Justification:

When we look at the character components of each string, we are only concerned with one character at a time. The push and pop methods of a stack have the perfect functionality for our goal. I decided on a linked-list implementation for two main reasons:

1. Dynamic Allocation: Each string may have different lengths, this way we can feely add additional elements into our stack without fear of stack overflow.

2. Sequential Access: Given the context of our problem, sequential access is preferred since it forces us to maintain the relationship between neighboring characters.

III. Recursion Comparison:

For a recursive solution, our recursive "step" would be to continuously pop off the top element of the stack until our stopping condition is met (i.e. when the stack is empty, invalid character found, etc). I found the iterative solution more intuitive while conceptualizing and debugging, however it is certainly possible that implementing some methods recursively would reduce unnecessary code.

IV. Lessons Learned:

Because we were had to rely solely upon stack manipulation, it made a seemingly simple objective quite tricky. The lab required creative applications of auxiliary stacks, and the capturing the subtle differences between each language was challenging. Additionally, having to create my own stack class gave me a newfound appreciation for Java's standard class (which has both dynamic allocation and random access due to its vector implementation). Overall, I was able to deepen my understanding of both the linked-list and stack ADT through this lab.

V. Efficiency:

Due to the nature of a stack, I believe this process was quite efficient. Without needing to query the stack, push and pop operations are performed in O(1) time. This makes parsing each input very fast, and the cost to evaluate is primarily dependent on the size of the input file and the operators within each input string.

VI. Future Iterations:

Going forward, I definitely want to create a fully-fleshed out recursive implementation for this problem. Similarly, It might be interesting to try and replicate this lab as we continue to learn about other ADTs. This would give us a practical way to discover the pros and cons of different data structures, while keeping the objective the same.