605.202: Data Structures

Peter Rasmussen

Lab 1 Analysis

Due Date: March 2, 2021

Dated Turned In: March 2, 2021

Lab 1 Analysis

This analysis examines the 1) use of the stack data structure to convert prefix expressions into postfix equivalents and 2) code used to perform that conversion. I justify the use of stacks as an appropriate data structure for this lab because of stacks' last-in-first-out property, which is well-suited for reversing sequences. A Python package, called as a module, performed the conversion using an array implementation of the stack. Once I figured out how to perform the prefix-to-postfix conversion, I saw how recursion might be preferrable to the iterative approach used in this lab. The program also generated complexity statistics, which I used to analyze the program's time and space efficiency. The complexity statistics implied a space complexity of O(N), which did not surprise me. I was surprised that the statistics indicated a runtime complexity of O(N)[1]; I had initially expected a runtime complexity greater than that based on the design of my code. I was also surprised that my usual approach to problem solving involving code – to begin coding immediately – was not optimal. I learned that next time I should sketch out a solution more completely – or completely – before diving into the coding.

The rest of this analysis covers the following:

- Justification of the design of the program.
- Description and justification of stack implementation.
- Argument that stacks were a good choice for this lab.
- Discussion of the use of recursion in lieu of iteration.
- Discussion of the time and space efficiency of the program.
- Summarization of lessons learned.
- Description of what I could have done differently.

Justification of the Design of the Program

The program is organized as a Python package[2], which has the benefit of making the code portable as the user does not have to worry about relative path imports, an issue that – for the developer – is obscured by modern IDEs, which infer a project's structure. In this design, the user can execute the program from the command line from any directory so long as the path to the module and other required inputs are provided.

The package is comprised of the following modules:

__main__.py

This module gives the package the portability / modularity described above. It "the entry point into [the] program when the module is executed as a standalone program" (Scott Almes, proj0 Python package).

__init__.py

This module "expose[s] what functions, classes, etc. are available to other scripts when the module is imported" (Scott Almes, proj0 Python Package). Specifically, the __init__.py limits imports to the class necessary to execute the program, PrefixConverter (described later), instead of importing all the classes unnecessarily.

[1] I tested values of N up to 10,000, so it is possible the shape of the runtime curve could change as N increases, but I doubt that based on inspection of the results.
[2] Per Scott Almes, proj0 Python package: "Packages are groups of modules that work together to act as a library or program".

stack.py
This module provides Stack, StackUnderflowError, and StackOverflowError classes. The Stack class is the fundamental data structure of this program and includes all of the methods specified in that data structure's ADT. The Stack class is implemented using an array of user-specified size pre-allocation. The StackUnderflowError and StackOverflowError classes are custom classes that catch instances when an empty stack is popped or a full stack is pushed, respectively.

prefix_converter.py
This module converts a file of newline-delimited prefix expressions, when possible, into their postfix equivalents. The module provides the PrefixConverter and PrefixSyntaxError classes. The PrefixConverter class performs the actual conversion and returns a newline-delimited string of prefix and postfix outputs. The class is designed to catch prefix statements with syntax errors before they are converted into postfix, and writes outputs in a human-readable manner, as indicated in the example output file below.

Example output file:

```
# Peter Rasmussen, Lab 1
# Input file: /path/to/required_input.txt
# Output file: /path/required_output.txt

Line 1: Prefix: -+ABC, Postfix: AB+C-
Line 2: Prefix: -A+BC, Postfix: ABC+-
Line 3: Prefix: /A+BC +C*BA  , Postfix: PrefixSyntaxError('Column 11: Too few operators, ...


@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
Complexity outputs
Function: convert_prefix_input       Time (ns): 10375000  Loops: 3
Function: convert_prefix_stack       Time (ns): 7775000   Loops: 9
Function: _convert_prefix_stack      Time (ns): 7190000   Loops: 8
```

Header statements make up the first four lines of the output file. Prefix processing outputs are listed line by line thereafter. Each line of prefix output begins with the line number of the corresponding prefix expression to make inspection easier for the user. Then, the original prefix statement is echoed. Finally, the postfix expression is written. Below the conversion outputs are complexity outputs: time and number of loops, a crude proxy for space complexity.

Prefix statements with syntax errors are not converted into postfix. Instead, an error message encapsulated in PrefixSyntaxError object is written to in lieu of a postfix expression.

A readme summarizes the code and explains how to run the program.

<u>Description and justification of stack implementation.</u>

This is a traditional stack that includes all required[3] and nice-to-have methods from the data structure's ADT, implemented using an array, that has a homogenous datatype. The stack accepts strings, integers,

[3] Push, pop, is_empty, and the constructor.

floats, lists, and Booleans, although in this package we only use strings. The class includes an additional 1) display method that shows the elements top to bottom and 2) to_string method that converts stacks to strings.

The array implementation would usually provide the benefit of random access; however, I did not make use of that feature in this lab.

Argument That Stacks Were a Good Choice for This Lab

We begin our argument by restating the problem, which basically is to convert a prefix expression into a postfix expression. In prefix statements, operators precede the operands and in postfix statements, operators follow the operands. Although the postfix equivalent of a prefix statement is not simply its reverse, the reading of the terms *is* reversed: postfix statements are read left-to-right while prefix statements are read right-to-left. So, to convert a prefix statement into a postfix statement (or vice-versa), we need a data structure that can facilitate this reversal. The stack data structure is well-suited for this purpose because its last-in, first-out property, which makes the data structure useful for reversing sequences.

Discussion of the Use of Recursion in Lieu of Iteration

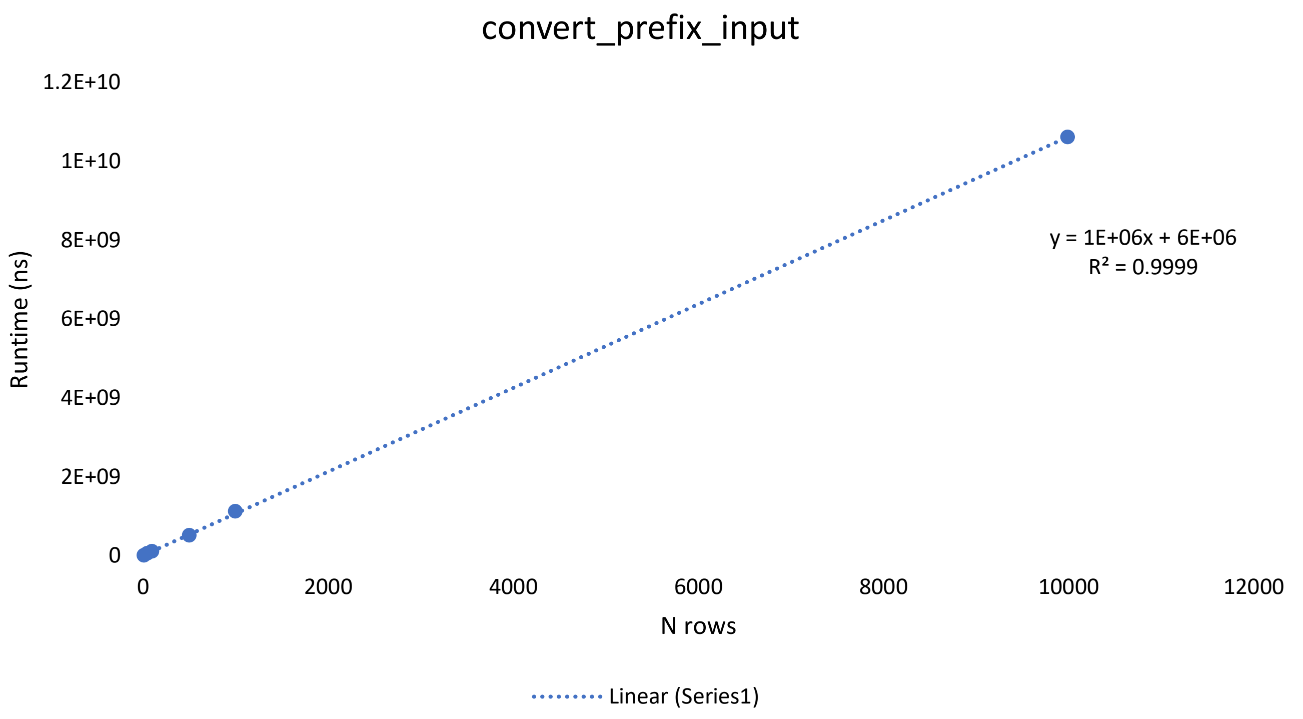
As I mentioned in the introduction, I saw how recursion could be used instead of iteration when I figured out how to perform the conversion. The example of a conversion from prefix to postfix. In the iterative version, a succession of passes through the string are made, combining operator-operand-operand terms and rearranging them, until the conversion is complete. Instead of making a succession of loops over the string, a recursive function could continue to call itself until all the operator-operand-operand terms are combined and rearranged.

++-ABC+DE
++(AB-)C(DE+)
+(AB-)+C(DE+)
+(AB-)(CDE++)
(AB-)+(CDE++)
AB-+CDE++

Discussion of the time and space efficiency of the program

Runtime efficiency

The figure below illustrates the runtime efficiency of the program. On the x-axis is N, the number of rows, and on the y-axis is the runtime. I ran the program for file sizes of 10, 50, 100, 500, 1000, and 10,000 rows of prefix terms for a fixed number of characters in each row. The increase in runtime is strikingly linear. This initially surprised me because I used nested while loops to process the outputs, but on second thought I think this problem is not really O(N) but O(M^2*N), where M is the number of characters in each row and N is the number of rows. I did not adjust M, so this is a constant thus reducing runtime in my tests to O(N). The M^2 is due to the fact that I used two while loops to perform the conversion.



Space Efficiency

The space efficiency of the stack is O(M): As the number of characters M increases the space complexity increases linearly with it. The space complexity of the PrefixConverter is technically O(M*N), but in practice, because M tends to be constant (at least in the cases tested) the complexity is O(N).

<u>Summarization of Lessons Learned and What I Could Have Done Differently</u>

After spending so much time studying and implementing them, I have a much better feel for how stacks work, and how to use them to solve problems.

I wish I had written out the algorithm by hand completely before I started coding. That was my biggest mistake in this lab and one that I will not repeat ☺. I tend to forget just how time-consuming code documentation is and will also remember that for the next lab.

I have developed production-level Python code before, but I actually had never made my own Python package. I learned a lot just by looking at Scott's code, which was plainly explained. I intend to develop my Python programs as packages going forward.