605.202: Introduction to Data Structures

Dominic Sciara

Project 1 Analysis

Due Date: June 30, 2020

Dated Turned In: June 30, 2020

Project 1 Analysis

Description

A stack is a container of objects that are inserted and removed according to the last-in first-out (LIFO) principle. This LIFO characteristic of a stack makes it useful for parsing strings with certain patterns, including mathematical expressions. For this particular project, stacks were used to compare the different counts and orderings of characters within a string.

To put it simple, the stack data structure was used in this case to store ‘A’ characters and pop them off as ‘B’ characters appeared in the string. It made it easy to tell if there is the same number ‘A’ characters and ‘B’ characters within a given string because at the end of the program, the stack would just be empty. Of course there was more intricate uses of the stacks for the different languages tested but the underlying application was very similar in each language.

Implementation

Stacks are typically implemented by linked lists or arrays. I chose to use the linked list implementation for this project because of the dynamic memory allocation characteristic of linked lists. This allowed me to not have to waste space by making an overly allocated array for each stack. This also saved me the time of implementing a dynamically allocated array, which also could waste space. The linked list implementation avoids stack overflow errors, which would occur in the array implementation if there weren’t sufficient memory allocated.

Not knowing how long the input strings are going to be was the major deciding factor for my choice. Also, I didn’t see a use case for random access in parsing these strings, so the benefits seemed one-sided to me.

Efficiency

Space

- For languages 1,2,3 and 5 there was only 1 stack being used and the stack would store up to the whole the string in the worst case scenario. Other than the stacks, there was only constant memory allocated to various variables. Thus making the space complexity for those languages: **O(n)**.
- For language 4, there were two stacks being used which also, at most, had the entire input string stored in 1 stack. There was never a case where both stacks had more than half the entire string at a time. Language 4 also only had various constant memory allocation besides the stack. Thus the space complexity for language 4 was: **O(n).**
- For the overall problem, there was never more than 1 language being tested at a time. After each language test, the local stack being used for calculations would be deconstructed and the memory would be freed up again. Thus the overall space complexity fo the problem was: **O(n)**.

Time

- For languages 1,2,3 and 5 there was never any nested loops. There was only the main loop that iterated over every character of the input string and then constant time operations within that main loop. Thus the time complexity of these languages are: **O(n).**
- Language 4 also had a main for loop that iterated over each character in the array. There was also a while loop within the for loop that would get hit in most cases. If this while loop was

hit, an operation that took at most **O(n-1)**. Thus the overall time complexity for this language is: O((n-1) * n) = **O(n^2).**
- Lastly, the overall problem ran each of these language tests, 1 at a time, for each string in the input file. Thus the time complexity is: O(m * (n +n+n+n+n^2)) = **O(m*n^2)**. (where m is the number of strings in the input file and n is the length of the longest string)

What I learned
I learned a lot of basics for the java syntax in doing this project. I am not all that familiar with the language so doing file i/o and getting practice using classes for everything really helped me further my abilities with the language. Coming from using python and java script regularly humbled me in doing this project.

Specifically, I learned how to run java programs from the command line, as well as pass in arguments from the command line. I also learned how to read and write to files, and all that comes along with the reading and writing. I learned the specific use cases for the keywords of java like public, private, static, etc… Again coming from JavaScript and python, encapsulation wasn't achieved from protected class methods.

What I Might do differently Next Time
Next time I will definitely start off by mapping my algorithms out on paper. I found myself developing algorithms for the language tests as I was coding and getting lost a lot. For language 4, I had to stop coding and sit down and write it all down with a paper and pen, it was a lot easier to code after. I also might consider typing up a solution in a language I am more familiar with, like python, and then translate it over. The extra practice wouldn't hurt and I feel like the translation would really help me understand the java language even more.

Justification for Design Decisions

There are four classes in this problem. The main entry point for the application (Project 1) is used to read the input from the file , call the necessary functions to get the correct out, format the output and lastly, write the output to the file. This helps isolate each of the classes more, allowing for the use of these classes in other programs.

The StackNode class is a prerequisite for the Stack class, it is the foundational building blocks of the stack. It is where the data values are stored, along with pointers to maintain correct ordering.

The Stack class is the core data structure used in this assignment. I implemented a stack that is capable for storing chars only but a more general stack could have been implanted.

The LanguageTest class holds the specific algorithms for each language. This class implements the stack class for each of its algorithms. The main driver refers to this class only for the test results for each string, again creating a nice separation, allowing for different implementations of the tests in another version of this assigmnet.

Recursive Solution

Since function calls of a program are stored in a stack and execute in the LIFO order, the language-test algorithms, which uses stacks, could instead be remodeled to use the call stack through recursion. Stacks can be used to implement a nonrecursive version of a recursive algorithm. Essentially, the stack contents would be held in each subsequent function call. Pushing items to the stack would mean making function calls and popping items would be return from functions. The recursive solution to this assignment would have similar time complexity but a worse space complexity because of the memory expense of the call stack.

Enhancements

I decided to make my output file a little more comprehensive by not only outputting whether a string passed the language tests, but also outputting why the string failed.
I also provided extensive commenting and input testing.