605.202 DATA STRUCUTRES

GUAN YUE WANG

JHU ID: gwang39

LAB 1 ANALYSIS

JUNE 29, 2017

General Program Design

The program simulates a language parser and evaluates if input language belongs to language L. It consists of 3 classes, LanguageParser class evaluates the intake String and provides results, Stack class helps handle data manipulation and validation process, OutputList class acts like an array list which holds the final evaluation results for each input String for future file output. Lastly, the main entry point can be ran with or without input file path command line argument. If the user chooses to provide file path and name in command line, the program would take input from there to process, if it is left blank, program can also prompt user for input location later in the console.

The overall program is controlled by a while loop to read input text file line by line. During this process, LanguageParser reads each line and then implements test individually for L1 - 16 using class methods. Then the evaluation results are printed and returned to OutputList (an array list). After the iteration, all evaluation results in the OutputList would be printed and exported to a text file.

Data Structure Implementation And Approriateness

There are two major data structures in the program, stack and array list. Both data structures are implemented using array because our input size tends to be small and content is likely to be simple string and char. From a coding perspective, array implementation is also easy to write compared with other options available such as linked list.

Stack is implemented as an array of char with size default to 80 char and top pointer initiated as -1. There are five methods in this class including push, pop, isEmpty, peek, display. Whenever an element is pushed or popped, methods check for array out of bound exception first for overflow and underflow issue to make sure the method can be correctly executed.

Stack is selected to handle and validate data due to multiple considerations. First of all, our elements are all single char which can be easily pushed as popped as a single element to the stack. Secondly, language validation usually involves either quantity check or pattern check which is a good fit for stack data structure. To illustrate, quantity can be simply tested by storing each type of letter in different stacks and popping them at the same time till one is empty. Then is Empty check is used to see whether the stack size is the same or not. On the other hand, pattern can also be check more efficiently using stack. For example, when you have pattern in one stack and all characters in the other stack, whole string's pattern can be effectively checked by popping pattern stack together with each fragment of the string and validate popped element char by char. Please see Appendix 1 to see how stack is utilized to validate L1 - L6 when we process a correct case.

Other than the stack data structure, array list is also implemented using array to perform data storage for test results during the iterative validation process. There are four methods in this class: add, get, increaseSize, and getLastIndex. This array list has the functionality to double its size whenever array reaches its maximum so we can minimize the risk of running of space to store results.

Array list is chosen as a storage space for output results as it provides more flexibility to save data when we don't know the upper limit of the data file. As data storage happens during the iterative process, we

can just add our results into the array list along the evaluation process without concerning about the size of the array. However, in the real life it does have a maximum depending on the memory and processing power of the computer running the program.

Alternative Approach

This whole program relies on iterative structure to either loop through the char array or the stack. However, recursion can also be a good way to handle different parts of the program. For example, when we read the text file, we can recursively read the character till we reach the end of the line. Other than that, when handle the item, we can also use recursion to do the validation by popping necessary stacks till one of the stack is running of item to pop. Also, in the LanguageParser class, when we do test L1-L6, we can recursively run next available test till there's no test left to do. Lastly, we can recursively add results to summary text file from the array list till the next element is null. Overall, there are a lot of opportunities to utilize recursion in this program. Potentially, it might cost more processing power and space but the final code would look more elegant and interesting.

Time And Space Efficiency

Regarding time complexity, both data structures have O(1) so the complexity depends more on the program itself. The main program has a while loop to read line and pass to LanguageParser. After that, the LanguageParser mostly use single loop to handle string but L3test requires the usage of double while loops to process items. Consequently, the complexity of the program is O(n^3). From an optimization perspective, as only one language test requires double while loop operation, if we can improve that a bit and turn the method into a single loop process, we would be able to reduce the complexity to O(n^2). With a small data input and good processing power, this does not make a significant difference; however, if we consider large data size, the efficiency difference can be meaningful. Please see Appendix 2 for an demonstration of current time complexity using real run time records with different data size.

From a space efficiency perspective, each line of input data is stored in string first, pass into stack as array of char with default size 80 char. And then the output is stored into an array list with array size sets to 20 String initially but has the capability to double its size whenever maximum is reached. In order to improve the space efficiency, we might want to get a better estimate for string length and test volume so we can decrease the default size of both stack and array list class. Also, perhaps we can implement the same functionality for array in the stack class as array list, initialize the size with 10 char but grant it the ability to increase char array size whenever string length is greater than the array size. However, this would likely have negative impact on time efficiency of the program.

Learning And Takeaway

There are many takeaways from this lab. One of the most important ones is how the time efficiency of the whole program can be impacted by a small block of the code. With one of the blocks in inner loop is O(n^2), the whole program is negatively impacted and becomes O(n^3). If we can optimize this block to O(n), then the whole program's complexity can be improved to O(n^2) instead of O(n^3) which can be a

significant time saving when data size is large. Next time, I will probably keep this in mind when I design the overall algorithm and try to keep the complexity low for each module and improve the overall efficiency of the program.

In addition, other things I might do differently next time are the implementations of the stack and array list. I would like to try how to use other implementations such as linked list to implement the stack and the array list. Although it's a bit harder to write, it would be really beneficial to know the difference in the implementation procedure and how it forms each data structure as well as what role node plays in the whole process. On the other hand, things I would do the same is the same is to provide flexibility in the program for users. In this lab, I'm able to provide both command line and console prompt for user file input and also allow them choose output method and location in the console. I believe those are all valuable features in the real world. Last but not the least, I really see the OOP concept's benefit in this program, by having a separate class handles language evaluation, the whole program becomes cleaner and easier to follow and modify. This is also something I would keep doing.

Enhancement

There are several enhancements in the program to improve the process flow in the aspect of general design, user I/O, error handling, and program content.

To begin with, the general design of the program implements OOP concept by having a language parser available to read input, evaluate language, and generate test results. Compared to put those features as functions in main class, it brings encapsulation to make sure the structure and flow of the program is cleaner and easier to follow. In addition, array list class is implemented through array to perfectly fit program's need for output storage the results is saved through a convenient way during the iterative process after reading and evaluating the language.

As a part of the design, there's also enhancement at user I/O level, it grants user the flexibility to choose their way for importing and exporting files. To be more specific, one option they have is to state input file location in the command line argument so the program gets the file directly from there. Also, they can just run the program without any command line argument and the program would prompt user for file location and read from there. On the other hand, while exporting the results to output file, user also has the options to choose whether they want to provide output locations or save under the same folder as input file but with modified name (i.e./input1.txt becomes /input1 _outputGW.txt). In the real world, I believe this improved user experience can increase the adoption rate and flatten the learning curve.

As a part of all above designs, exception handlings are utilized to prevent unforeseen errors generated while people are running the program in different cases. This includes IOE exceptions when input and output files, stack overflow and underflow checks in stack class, and array out of bound validations in array list class.

In addition to design, program content is also enriched by adding two more languages, one evaluates AnBmC(m=n) pattern and the other check if language is symmetry at certain point. Both requires different and interesting utilization of stacks.

Appendix

Appendix 1. Demonstration on how correct language is handled for L1 - L6

L1 BAAABBAB - pop both stack to check the quantity of As and Bs

A	B
A	B
A	B
A	B

L2 AAAABBBB - pop both stack to check the quantity of As and Bs

A	B
A	B
A	B
A	B

L3 AAAABBBBBBBB - pop both stack (once for A, twice for B) to check the quantity of As and Bs

	B
	B
	B
	B
A	B
A	B
A	B
A	B

L4 ABBBABBBABBB - pop fragment by fragment to check stackPattern vs stackLanguage equality

	B
	B
	B
	A
B	B
B	B
B	B
A	A

L5 AAABBCCCCC - pop C stack while popping (A stack and then B stack)

		C
		C
A		C
A	B	C
A	B	C

L6 ABCCDCCBA - pop left side reverse into stack 1 and right side into stack 2, and then pop both stacks one by one to check equality

A	A	
B	B	
C	C	
C	C	D

Appendix 2. Current Time Complexity O(n^3) with real program run time tracking

n	**Run time (microseconds)**
1	1200
10	1700
50	4000
100	6700
500	20000
1000	100000