605.202: Data Structures

Ziyu(Yvonne) Lin

Lab1 Analysis

Lab1 Analysis

1. Description of Data Structures

The main data structure used in this lab - transforming prefix to postfix is a stack. A stack follows the Last-In-First-Out (LIFO) principle, similar to a pile of books, where the last book placed on top is the first one removed.

In this implementation, I have built my own custom stack class instead of using Java's built-in Stack class, which is a great learning experience for me to build all the methods from scratch and gain a deeper understanding of stack operations. The stack is implemented using a linked list way, where each element in the linked list is treated as a node. In the Stack class, I have a private Node class so later every element could be treated as a node.

The stack class includes the following methods, which we have learned from the previous module and the zy book:

- isEmpty(): it checks whether the stack contains any nodes;
- push(T data): it pushes a new node on the top of the stack and returns nothing;
- pop(): it removes and returns the top node from the stack;
- peek(): without modifying the stack, it returns the top node of the stack;
- size(): it returns the size of the stack.

2. Justification of the data structure choices and implementation

In this problem, the stack data structure is ideal. There are several reasons. First, the prefix requires a right-to-left read, and a stack class could help us store the operands and pop the appropriate operators and operands to postfix expression when we encounter a new operand in our scan. Since the stack naturally has a LIFO order, it could help us pop the most recent operands and operators when needed and this is what we need in the postfix expression.

Implementing the stack using linked list is ideal as well. Unlike an array-based stack, a linked list stack does not require a predefined size and can thus grow dynamically. Array-based stack needs resizing when full, which is O(n) time complexity.

3. Discussion of Appropriateness to the Application

This application includes the following method:

- isOperator(): it determines whether the character is an operator or not.
- toPostfix(): it takes the prefix expression and converts it to postfix expression.

The conversion from prefix to postfix expressions will first read the prefix expression from right to left. If it encounters operands, the script will push the operands to the stack. If it encounters the operators, pop two operands from the stack and form a new postfix expression by adding the operator after the two operands. And then push the postfix expression back to the stack. If we finish this process, then the last and the only element is the postfix expression we want.

It is always good to add error-checking since we are unaware of the user input. First, we determine whether we have only one last element in the element. If we do not, the script will raise an illegal state exception saying there is more than one element in the stack at the end. Second, every time the script encounters an operator, we determine if there are more than 2 operators to pop. If there are less than 2 operators, the script will raise an illegal state exception saying there are less than two elements in the stack and thus not enough operands for operators.

Using stack is very beneficial for this application. The reason why we choose stack over other data structures is that the LIFO structure ensures the correct order of operators and operands when we pop or push the element. And FIFO or other data structures could not do it.

4. Design Decisions and Justifications

- Use stack

Using a stack will naturally solve this problem compared to using other data structures like queues. Operands are pushed onto the stack and operators pop the last operands and combine them into a postfix expression. The LIFO characteristics of the stack naturally ensures what is put into the stack last will be examined first, and this property solves this problem.

- Modularization

This program follows a modular approach. And actually writing this program has been a great learning experience for me to think through about code modularization, as I don't have much experience with it. The stack is a separate class, creating a template of an object, which allows other programs to use it in the future. And the prefix to postfix expression is also a class. Inside this class, there is the first method called isOperator(), which helps to determine whether the char is an operator or not. And creating a separate method the script builds more clarity. Then there is the prefix to postfix function, which translates a prefix input to postfix expression. This method uses the stack class and the toOperator() method.

- Error handling

Error handling has been a valuable practice for me to carefully consider different cases. One scenario is when an expression contains invalid characters, which can be identified by checking if each character is a digit, letter, or valid operator. If none of these conditions are met, the expression is marked as invalid. Another case occurs when an expression is unbalanced. If there is more than one item in the stack, that means there are more operands than the operators. If there are less than 2 items in the stack when we are trying to pop the top 2 items from the stack, that means there are more operators than the operands. Another error handing is the file handling. File handling errors such as missing files, permission issues, or failures in reading and writing are managed by catching IOExceptions and providing meaningful error messages.

- Command-line arguments

Using command-line arguments for specifying input and output files makes the program more flexible and user-friendly. Instead of hardcoding file names, the program takes input and output file names as arguments, allowing users to process multiple files dynamically. I was hardcoding the filenames before, but listening to the recorded office hours, I found out that I could do things differently. And this is a great practice for me.

5. Efficiency with respect to both time and space

- Time complexity

This application has the following methods: the script traverses the prefix string from right to left, and this is O(n); the script uses push method to add the operand on the top of the stack, and this is O(1) in time complexity; the script uses pop method to pop 2 operands from the stack, and this is O(1) in time complexity as well. The overall problem solution is driven by the problem itself, not the data structure. So, this application is O(n) in time complexity.

- Space complexity

This script traverses the prefix string from right to left, and stores operands, or operators or subexpressions in the stack. The stack takes space of O(n). There is another postfix string that stores the postfix string expressions we get in the middle, and this takes space of O(n). The overall problem is driven by the stack data structure. So, this application takes O(n) of space complexity.

6. Recursive Approach

Almost all iterative approaches could be expressed in a recursive approach. This application is one of the good examples. It is not straightforward for me to think of this problem in a recursive approach. I take some time to think through it and I think I figured it out! One point that does not come intuitively for me is that I think in this application, we need to evaluate the next two operands before the final expression, since an operator always requires two operands. So we need to recursively process them (two operands/ subexpressions) before applying the operator. Another point is that we process the prefix from left to right, because operators need to wait for both operands.

The base case of the recursion is that if the current character is an operand, then return it. The recursive case is that we process the first operator, recursively evaluate the next two operands or subexpressions, (each operator leads to two recursive calls), and then construct the postfix expression by appending the operands before the operator. And finally, we return the postfix expression.

There is no definitive advantage of one approach over the other. Compared to the iterative approach, the recursive approach uses implicit stack frames during the recursive calls. So the memory usage is more than O(n), the space usage in the iterative case. Time complexity is also O(n), same as the iterative case. The recursive approach may be more readable since it naturally breaks the problem down into smaller subproblems. And one potential drawback is that deep recursion may cause a stack overflow. Therefore, it is really the personal preference to choose a recursive approach or an iterative approach.

7. What I learned

Converting a prefix to postfix is very helpful for my understanding. Initially, although I understood that a stack was the appropriate data structure for this problem, I struggled with how to implement it effectively. However, by carefully analyzing the logic and gradually working through the program, I was able to understand how the LIFO (Last-In-First-Out) structure of the stack helps in correctly formulating the postfix expression.

Additionally, implementing the stack from scratch was surprisingly insightful experience. Previously, I had relied on built-in stack implementation. I understood how to use stacks but not how they worked internally. By manually implementing push, pop, peek, and size operations using a linked list-based stack, I gained a much deeper understanding of this data structure.

Another key takeaway from this project was the importance of modularity in software design. Instead of embedding the stack logic directly into the conversion function, I structured the program by placing the stack in a separate class, making the PrefixToPostfix class more reusable and maintainable.

Moreover, applying the theoretical concepts from class and homework to real code has been incredibly rewarding. While concepts like prefix to postfix expressions seemed like something I have already learned from the lectures, actually implementing them in code helped solidify my understanding.

8. What I might do differently next time

One improvement I would make is comparing my algorithm with Java's standard implementation to analyze performance differences between my implementation and the more general one designed by the language developers. This would help me evaluate whether my solution is optimal or if there are improvements to be made.

Another enhancement would be adding unit tests to the program. I have never worked with unit tests before, but after hearing my classmates discuss them during office hours, I realize they are highly structured and valuable in real-world applications. I would like to take the time to learn about unit testing and implement it in my future projects to improve code reliability and maintainability.

9. Enhancements

In this practice, I also implemented the postfix to prefix functionality. I placed both prefix to postfix and postfix to prefix conversions in the same class, NotationConverter, as they share the same fundamental purpose—transforming one notation into another. If I were the user, I feel calling the methods from the same class would be very intuitive.

And also implementing postfix to prefix functionality helped me think through the problem more thoroughly by approaching it from the reverse perspective. I tested the conversion using the same input file, and everything appears correct since the transformations successfully revert the expressions back to their original forms!

10. Addition input

These are more inputs I generated, which I think will be representative of some error scenarios:

- A - single letter
- +-*/ - only operators
- ++AB – more operators than operands
- +ABCD – more operands than operators
- +AB – simple correct expression