605.202: Data Structures

Ziyu(Yvonne) Lin

Lab 2 Analysis

Lab 2 Analysis

1. Description of data structures

The main method used in this lab - transforming prefix to postfix - is recursion, differing from what we did in Lab 1, which uses a stack and its Last-In-First-Out (LIFO) principle to store and pop the operands or operators encountered in the prefix expression. In this lab, however, we depart from that approach by leveraging recursion. This way, we eliminate the need for explicit data structures.

In this lab, several data structures have been used:

- Primitive Data Type

The script uses some primitive data structures such as char and int. Characters are used to process each symbol in the prefix expression. Integer variable is used for index, which helps to track the current position in the recursive traversal. see if it is backtracking or forward tracking.

- String

String is used to build the postfix expression during each recursive call. Both the input and output are manipulated as String objects.

- Recursive call stack

The Java call stack implicitly manages the recursive flow and serves the purpose of operand and operators management. This replaces the need for an explicit stack, as used in Lab 1, while still keeping the logical structure of expression evaluation.

2. Justification of the data structure choices and implementation

In this problem, recursion is used to solve the conversion from prefix to postfix by leveraging the data structures mentioned above. There are several reasons for this choice.

First, the recursive method design eliminates the need for any manual data structure such as a stack, simplifying the implementation. And we used simple data types to process the prefix and postfix, which allows efficient processing of the prefix input and the construction of the postfix output. This approach makes the solution both lightweight and straightforward.

3. Discussion of appropriateness to the application

The prefix expression is naturally an expression tree. In this expression tree, operators are nodes and operands are leaves. The recursive approach mirrored this structure. A prefix expression corresponds to a pre-order traversal of the tree, from operator to left subtree to right subtree, A postfix expression can be seen as a post-order traversal, from left subtree to right subtree to

operator. By using recursion, each recursive call can be seen as parsing a subtree. Recursion naturally mirrors the natural structure and traversal of the tree.

Using recursion, this application includes the following method:
- isValidPrefix(String input)
- toPostfixRecursion(String prefix, Index index)
- toPostfix(String prefix)

The first function validates if the input is a valid prefix expression before performing prefix to postfix conversion. This is one important step in error handling, ensuring the input adheres to the structural rules of prefix notation. It validates the number of operands and operators and input structure follows the prefix structure.

The second function is a helper recursion function. The base case of the recursion is that when an operand is encountered, the function simply returns it. The recursive case is that the helper function recursively processes the subexpressions, which represent the left and the right operands of the current operator. Op1 is the left operand and op2 is the right operand. At the end, it combines op1 and op2 and current character in the order: operand1 + operand2 + operator to form the resulting postfix expression. The index input in this function helps to define forward or backward tracking. As for prefix to postfix, we are tracking from left to right, so we are giving this function index = 0. This is because we need to find the operator first and recursively build the expression tree from the root.

The third function is the driver function that initiates the recursion function. In this method, it firstly uses isValidPrefix(prefix) method to validate whether the input is a valid prefix expression. It then calls the recursion function, passing in the input string and initializing the index to 0. This method returns the final postfix expression at the end.

4. Design decisions and justifications

- Use of recursion

The primary choice of this program is using recursion rather than stack structure. This decision is driven by the thought of leveraging natural hierarchical structure of prefix expression tree. Recursion allows us to process the subtree in an intuitive and elegant way, with each recursion handling one node (operator or operand) and its subtrees.

- Modularization

This program follows a modularization approach. Instead of putting everything line by line, this program uses a notation converter class for prefix to postfix transformation method and postfix to prefix transformation method, creating a template of an object, which allows other programs to use in the future.

- Error handling

This program also considers different error handling cases. One case is validating whether the input is a valid prefix before attempting the conversion. The method isValidPrefix() helps

validating. It ensures the number of operands and operators and the input structure follows the prefix structure. In prefix notation, operators appear before their corresponding operands. For a prefix expression to be valid, every operator has exactly two operands. This implementation tracks the number of operands and operators as we scan the expression from left to right. At every step, the implementation also checks whether the number of operands is less than or equal to the number of operators plus one, which is needed for prefix expression. The reasoning behind this is that we want the expression to not overload with operands without enough operators. And the end of the scanning, the program checks whether the total number of operands is exactly one more than the number of operators.

Another error handling case is the file handling. File handling errors are managed by catching IO exceptions. IOException could occur when there are missing files, or permission issues, or failures in reading and writing.

- Command-line arguments

This program uses command-line arguments for specifying input and output files, which makes the program more versatile. Rather than hardcoding file paths or relying on user input during runtime, the use of command-line arguments allows more flexibility of the program. And this is useful for batch processes or automated workflows. This design choice makes the program more adaptable to different usage scenarios, especially in some cases where multiple files need to be processed without manual inputs.

5. Efficiency with respect to both time and space

- Time complexity

The time complexity of the program is O(n). This is because each character in this expression is processed once during the recursive traversal. Think of an expression tree, internal nodes represent operators and leaf nodes represent operands. By using recursion, each node is traversed once. Therefore, the time complexity is O(n).

- Space complexity

The space complexity is determined by the depth of the recursion, which corresponds to the height of the expression tree. Therefore, space complexity is O(h). The space used in this program is not due to the data structures. Rather, the space is implicitly managed by Java call stack during recursion. The maximum of recursive calls on the stack is the depth or the height of the recursion tree, which is O(h).

6. What I learned

Through this program, I gained a deeper understanding of the expression tree. This time we used a recursion to solve this problem. By using recursion, the codes are simple and elegant. Recursion naturally mirrors the structures of expression tree, with internal nodes as operators and leaves as operands. It also naturally follows the hierarchical structures of the tree. I also learned that recursive algorithms in this case avoid using other explicit data structures like stack. Instead,

recursive algorithms use implicit call stacks for temporary storage of immediate values. However, for very deep expression trees, this may cause stack overflow.

In this lab, implementing recursion from scratch helped me better understand the structure of the expression tree and how each subproblem fits into the overall solution. Although recursion does not come naturally to me, this has been a valuable hands-on experience that significantly deepened my understanding of recursive thinking.

7. What I might do differently next time

If I were to do this project again, I would also add methods for infix conversion and evaluation, which allows the program to handle a broader range of input formats. I would also add user-friendly error messages, along with unit tests to ensure each function works correctly under various edge cases.

8. Enhancements

One enhancement I have done is adding postfix to prefix function. Following the same logic used in the prefix to postfix conversion, the postfix to prefix logic will firstly have a isValidPostfix(String input) method to check if the input is a legitimate postfix input. Similar to previous implementation, this conversion also utilized a recursive helper function toPrefixRecursion(String postfix, Index index) and a driver function that calls the recursive function.

To build an expression tree, the recursive function will need to process the operators first. Since operators come last in the postfix expression, index in this case will be the index of the last character. This way, we find the operator first and then recursively build the expression tree.

The base case of this implementation is the same: if the current character is an operand, then return it. This indicates that we have reached the leaf nodes or the bottom of the expression tree. The recursive case is that we recursively process the subexpressions by calling the function itself. Scanning from right to left, the first operand we encounter recursively is the right operand, which is the op2. The second operand we encounter recursively is the left operands which is the op1. And then the final prefix will be the operator + op2 + op1.

The second enhancement I made was to build an expression tree first and then traverse it in different orders. I hadn't realized this approach was possible until I listened to the office hours. After learning about it, I decided to redo the prefix-to-postfix expression conversion using this method. During the office hours, Scott mentioned that we could construct an expression tree from a prefix expression and then use different tree traversal orders to obtain both the prefix and postfix expressions. Although redoing this took me some time, it turned out to be a valuable learning experience. To build the tree recursively, I created a class called PrefixToPostfixConversion. In this class, I created a method called buildExpressionTreeHelper(String prefix, Index index) that constructs the tree recursively step

by step. Then, in the buildExpressionTree(String prefix), the program calls the buildExpressionTreeHelper helper function to recursively construct the tree and then returns the root node. After the tree is built, I use the getPrefix and getPostfix methods to retrieve the respective expressions. In the getPrefix method, the tree is traversed in a pre-order manner to construct the prefix expression, which follows the format: root.c + getPrefix(root.left) + getPrefix(root.right). Similarly, in the getPostfix method, the tree is traversed in a post-order manner to construct the postfix expression, using the format: getPostfix(root.left) + getPostfix(root.right) + root.c.

9. Additional input

In order to troubleshoot and verify, I used the same additional input cases as in the previous implementation. This way, I could compare the recursive approach with the stack-based approach to see if I am doing correctly in the recursion case. Gladly, I have the required input and the additional test inputs the same as those used in the stack-based implementation! Here are my additional 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