Christian Znidarsic
EN.605.202 Data Structures
3/29/22

Lab 2 Analysis

Data Structure Description

The objective of this lab was to use recursion to convert prefix strings to postfix strings directly. We were instructed not to use stacks, like we did in Lab 1. This fundamentally changed the way that this problem was solved, as some of the procedures used in the previous lab were no longer applicable.

Even though we could not use stacks for this lab, I still used multiple queues for the analysis of the user input prefix functions. These stacks did not play a part in the conversion process other than holding and feeding the input characters to the recursive function that did the conversion. For example, a queue was used to store all of the input characters, one by one, as they were read from the input text file. While characters were added, information about the expression was kept in other local variables. This information was used to determine whether the input expression was a valid prefix expression, or whether it contained invalid characters, etc. In this way, the queues were not used fundamentally for the conversion from prefix to postfix.

The queue data structure lends itself to this application because of its first in first out (FIFO) nature. Therefore, when the input expression is read character by character, each character is stored in the queue, when the queue is dequeued completely, the order of outputs from the queue is the same order that the input expression was in originally. In other words, reading from the queue is like reading the input expression front to back. This is opposed to a stack, because when you load an expression into a stack, when you read from the stack you are reading the original expression from back to front.

The queue that I used implemented a linked list. This required a QueueNode class to be defined in addition to a Queue class. A linked implementation made sense because items could be added and removed efficiently without worrying about indices or bounds like an array implementation would.

Iterative vs. Recursive

This problem can be solved either iteratively or recursively. In Lab 1, we solved it iteratively, and for this lab I solved it recursively. Getting to solve the same problem using these two different methods has taught me a lot about the differences between them. In this section I discuss the differences between these two methods.

General Description

In the iterative solution heavily utilized stacks. The input expressions were read one character at a time and pushed onto a stack. This stack was then pushed onto another stack to reverse the order of characters. Then a characters were popped, concatenated, and pushed back onto the stack in such a way that by the end, the last remaining item on the stack was the resulting postfix expression.

In contrast, the recursive solution does not utilize stacks. Characters are read one at a time from a queue containing the input expression. If the current character is an operator, then two more items are read from the queue, and concatenated together with the current character and returned. If any of those two next items are operators, then they also check the next two items in the queue. This represents the recursive case of the function because it calls itself. The stopping case of the function is if the character that is read is an operand. Then the function simply returns the operand.

Christian Znidarsic
EN.605.202 Data Structures
3/29/22

Simplicity

Compared to the iterative solution, the recursive solution is much more simple to write. It also appears more simple in code and is easier for a reader to follow. The actual code to convert from prefix to postfix is in the range of 15 lines, whereas the actual code doing the conversion using the iterative case is closer to 100 lines.

Effectiveness

Both the iterative and the recursive cases are equally effective. They both produce the same results and are equally accurate.

Time Complexity

The iterative solution I used in Lab 1 had a time complexity of O(n). The recursive solution I used in this lab also has a time complexity of O(n). Thus, neither solution has an advantage over the other when it comes to computational complexity.

My Thoughts

In the end I thought that the recursive implementation was far more simple to code, and it looks better in code as well. In the future I would continue to be wary of recursion however, simply because of the lack of efficiency that often arises. I would say that I enjoyed writing the recursion more because there were fewer edge cases to handle and it was more simple to think about than the iterative approach with stacks. I also enjoyed using queues because they allowed me to read the input in the same order after loading it into the queue.

Design

My program works using four classes: main, preToPost, Queue, and QueueNode.

Main

The main class is simple, and asks the user for input in the form of text files, creating relevant BufferedReader PrintWriter objects accordingly. It then passes these objects as arguments into the static method preToPost located in the PreToPost class. All of this is done inside of a try-catch block to catch FileNotFoundExceptions when the user enters an invalid file path.

Queue

The Queue class specifies Queue objects. It includes two QueueNode fields, one is called front, and one is called rear. The constructor method sets two properties this.front and this.rear to both be null. This is the case when the queue is empty. The Queue class implements a linked list to store nodes. This allows items to be added and removed in O(1) time, and without indices. This also means that there are virtually no size limitations to the queue.

The method enqueue() takes the key value of a new node as input, and adds it to the rear of the queue. Because this method

The dequeue() method removes the node at the head of the queue.

The isEmpty method returns true when the queue has no values in it, and false when there is at least 1 node in the queue.

QueueNode

The QueueNode class is the smallest class in the program. It simply specifies a queue node. Each node has two properties: key and next. Key holds the data of the node, and next is a pointer to the next node.

PreToPost

The PreToPost class is the largest and most crucial class in my program. It takes the BufferedReader and PrintWriter objects in from Main. Generally speaking, it takes an input text file containing prefix strings, and it outputs corresponding postfix strings to an output text file.

This method works by reading from the input text file using BufferedReader. It reads the input one character at a time. Depending on what the character is, different actions are taken with it. If it is a space or tab, it is ignored completely and the next character is read. If it is an invalid character, it is pushed to both the inputQueue and errorQueue. If it is a valid character, then it is pushed only onto the inputQueue. If a newline is read, then that means that a complete expression has been read.

After the newline, if the errorQueue contains any invalid characters, a corresponding error message is printed, along with the incorrect character(s). If the prefix expression was determined to be invalid, then an error message is printed that describes what conditions the expression did not meet in order to be deemed valid. If none of those conditions are true, then the input expression is assumed to be valid, and it is passed in the inputQueue to convertExpression.

convertExpression is the iterative function that actually does the conversion from prefix to postfix. Characters are read one at a time from a queue containing the input expression. If the current character is an operator, then two more items are read from the queue, and concatenated together with the current character and returned. If any of those two next items are operators, then they also check the next two items in the queue. This represents the recursive case of the function because it calls itself. The stopping case of the function is if the character that is read is an operand. Then the function simply returns the operand.

Other methods in PreToPost are isOperator, which determines if a string is a valid operator as specified in the lab spec, isOperand, which determines if a string is a letter, and isSpaceOrTab, which is used to exclude characters that are spaces or tabs.

Justification

To me this ordering made sense because main is quite short and only performs user I/O and very general operations like calling preToPost.

Queue and QueueNode are both completely self-contained classes that have all of the fields and methods needed to implement a Queue data structure together. Queue objects can be created and manipulated cleanly.

PreToPost is then the most complicated class, and performs the brunt of the work of converting expressions from the input file. It calls Queue but remains separate from it and only uses Queue objects. The PreToPost class contains both the preToPost method to handle file I/O and error checking, as well as the convertExpression method, which does the actual conversion from prefix to postfix recursively. Keeping the recursive method outside of the main preToPost method allows the recursive function to call itself without repeating other operations that only need to be called once.

Errors Handled

Spaces and tabs

Spaces and tabs are excluded by an if statement before they even have a chance to be pushed onto the stack. This is important because the user may accidentally include spaces or tabs without knowing it, since they may not be easy to see in a text editor. Thus, they are omitted outright upon reading them from input and not processed any further.

Blank lines

If the user input text file contains lines that are blank, or lines that contain only spaces and/or tabs, then that line is excluded using an if statement. Then, in the output, the blank line is excluded completely. This allows for user error and also cleans up the output file.

Invalid operands

The isOperand method takes a string as an input and determines if it is a valid operand. In this case, valid operands are only letters. Thus, isOperand uses Character.isLetter() to determine if the input is a letter.

Invalid operators

In the lab spec, the valid operators are defined to be “+”, “-“, “*”, “/”, and “$” for exponentiation. The method isOperator takes a string as input, and outputs true if it is one of these valid operators using a simple switch statement, and false otherwise.

Integers

The lab spec lays out that operands are only letters. Thus, integers are not valid operands. They are excluded by default by isOperand because Character.isLetter(char) will return false.

Invalid prefix expressions

Invalid prefix expressions are handled by preToPost. When the user input is read and loaded into inputQueue, the function keeps track of whether or not the input is valid. It does this using a counter. The counter is initially set equal to 1. When an operator is read, 1 is added to the counter. When an operand is read, 1 is subtracted from the counter. The method monitors two conditions to determine whether the input is a valid prefix expression.

1. If the counter is not equal to zero when the entire expression has been read, it is not a valid prefix expression.
2. If the counter dips to zero or less *before* the last character in the input is read, then it is not a valid prefix expression.

By determining whether an expression is valid before converting it, then the program can ensure that only valid prefix expressions are passed as arguments to convertExpression. This makes the convertExpression recursive function simple and readable. It also allows total control over what is and what isn't valid.

If an input expression is deemed invalid, a relevant error message is printed to output that specifies the criterion that wasn't met.

FileNotFound exception

When the user enters an invalid path to the terminal, a FileNotFoundException will arise. The main method implements a try-catch block to catch any FileNotFoundExceptions that are thrown. The exception is handled smoothly by requesting the user to re-enter the input file path.

Test Input.txt Key

The input .txt file tests for error handling in the following order:

1. Control (no errors, valid prefix expression)
2. Single invalid character inserted in otherwise valid expression
3. Multiple invalid characters inserted in otherwise valid expression
4. A blank line to demonstrate how the program disregards blank lines
5. A blank line with spaces and tabs
6. Expression with too many operands
7. Expression with too many operators
8. Expression with correct number of operands/operators, but in invalid order
9. Only integers
10. Spaces within otherwise valid expression
11. Spaces and tabs within otherwise valid expression
12. Control (no errors, valid prefix expression)
13. Control (no errors, valid prefix expression)

Improvements from Lab 1

Christian Znidarsic
EN.605.202 Data Structures
3/29/22

In my last lab I was able to do most of what I set out to do. In this lab, however, I was able to improve upon some things and add some functionality that makes my Lab 2 stronger than Lab 1.

Improved invalid character feedback

In the last lab I could not figure out how to return invalid characters to the user. I tried for a long time to implement this feature but could not get it to run cleanly. This time I was able to make it work and it adds a lot of functionality to this project.

When an invalid character is read, it is enqueued to the errorQueue *and* the inputQueue. If at the end of the expression, the errorQueue is not empty, then that means that at least one invalid character was included in the expression. Instead of passing this invalid expression to convertExpression, an error message is written to output, and the contents of errorQueue are printed along with it. This way, the user is told what the error is, and can also see each invalid character that the expression contained. This allows the user to adjust and fix the invalid characters after running the program.

Improved invalid prefix expression feedback

In the last lab, a single error message was output to the user to say that the input was an invalid expression. In this lab, there are three specific error messages to signify an invalid expression.

1. "This is an invalid prefix expression. Too many operands."
2. "This is an invalid prefix expression. Too many operators."
3. "This is an invalid prefix expression. Check order of operands and operators."

There are three different ways a prefix expression can be invalid, even with all valid characters. My updated program determines and outputs in which specific way the expression is invalid.

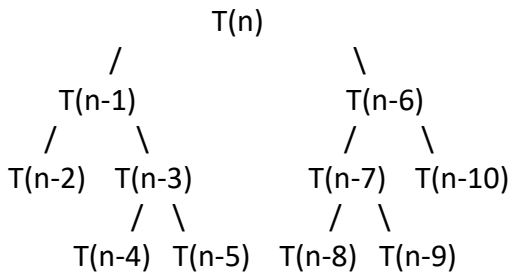
Improved output formatting

In the previous lab, the output file contained the inputs printed above the outputs on different lines. This made it difficult to compare the prefix expressions to the postfix expressions. Now, printf() was utilized to print both a prefix expression and its resulting postfix expression (or error message) on the same line. This way the user can clearly see the before and after of each input expression. This is also especially useful for erroneous expressions, because the user can see the incorrect expression directly next to its error message. This makes it easier for the user to see what was wrong with the expression so they can correct it before running the program again.

Efficiency

Time Complexity

The program has two areas that are not O(1). The first area is where each character is read from input. This process is O(n) because operations are performed for each input value. The second area is the recursive function convertExpression. This expression is also O(n) because a recursive call is made once for every character in the input. A recursive tree for the expression "*-A/BC-/AKL" is shown below to help visualize this idea.



As you can see, one recursive call is made for every item in the input expression. And because the complexity of all other non-recursive actions within the function are O(1), the time complexity of the function is O(n). Thus, the program has no actions that are greater than O(n), so it is O(n).

Space Complexity

Each character read from the input file is stored in the inputQueue and/or the errorQueue. Other variables are all O(1). Thus, in worst case scenario, all items are stored twice resulting in O(2n). Thus, the program has space complexity of O(1) + O(2n) = O(n).

Reflection

What I would do differently

After Lab 1, there were several items that I had wished to do differently. Fortunately, Lab 2 gave me the chance to improve these areas. I was able to fix what I set out to fix, which can be seen in greater detail in the “Improvements from Lab 1” section.

How I felt about Lab 2 versus Lab 1

I was much more confident going into Lab 2 than I was going into Lab 1. This is partly because I could reuse a significant amount of file I/O and main method code from Lab 1. I was also much more confident in my ability to write recursion than I was when I wrote Lab 1. I was excited to implement a queue, as I had never actually coded one before. In fact, I had never coded a linked implementation of anything before, so it was great to get to make a real node class and use it in a linked list. It was quite simple once I got the hang of it.