LAB 1 ANALYSIS REPORT

by Chih Yuan Chan ID: 28A8CA

Date: 03/01/2023

1. PROBLEM DEFINITION

1.1 General approach

The problem is to write Python code to convert a Prefix expression into an equivalent Postfix expression using a Stack ADT.

The adopted general approach to solve this problem is to progressively iterate through the Prefix expression from right to left (instead of left to right), reading character by character.

This order naturally arranges the Prefix expression into an initial "almost Postfix" form (where the operators are positioned to the left of operands), which in turn facilitates the following repeatable sequences to be executed:

- **STEP 1-** *If the read character is an operand, push it onto the stack*
- **STEP 2-** *If the read character is an operator, pop two operands from the stack*
- **STEP 3** - *Concatenate the two operands and operator into a string, in the following order:*

  operand1 + operand2 + operator
- **STEP 4** - *Push the resultant string back to the stack*
- **STEP 5** -*Repeat the above steps until the end of the Prefix expression is encountered*

The repeated steps naturally suit and justify the use of a conventional iterative approach.

1.2 Adopted Abstract Data Structure (ADT)

The Stack is the ideal ADT for this algorithm for the following reasons:

- Pushing the characters from the Prefix String (left to right) onto the stack naturally arranges the characters in the correct reverse (right to left) order as they are popped.
- A stack naturally maintains the precedence of the operators by storing them in the order they are encountered as the stack is traversed.
- Once the entire Prefix expression has been traversed, the stack naturally stores the characters in the correct Postfix order, which can be efficiently retrieved by sequentially popping the characters from the stack.
- The algorithm does not require intermediate elements within the stack to be accessed, only the top (head) element in the stack. Pushing and popping from the top of the stack can be implemented with O(1) time and space complexity and this is highly efficient.
- The code implementation for using a stack to convert prefix to postfix is highly readable and easy to follow / understand.

1.3 Implementation of ADT and justification

The chosen implementation of the Stack ADT is the singly linked list (SLL).

The alternative implementation option in Python is the "list" data structure.

1.3.1 Algorithmic Complexity considerations (SLL vs list)

The most frequently executed operations in the Prefix to Postfix conversion algorithm are "Push" and "Pop" operations.

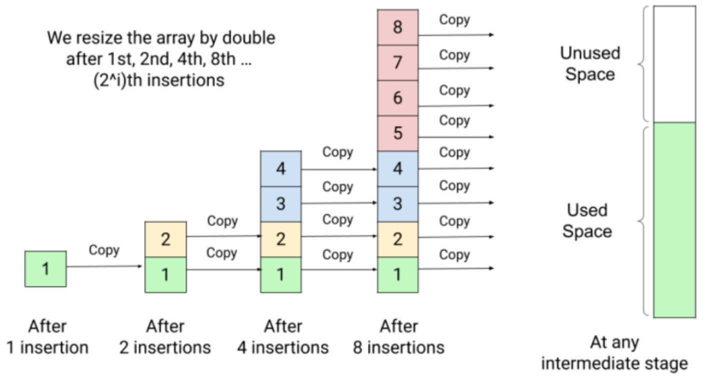
In the case of the singly-linked list, the push and pop operations have O(1) time and space complexity since the head node of the linked list can be implemented to correspond to the top of the stack.

The Python "list" data structure uses a Dynamic Array (ref 1). Inserting (pushing) and deleting (popping) from the front of the list will incur a worst case time complexity of O(n) since this will require a reorganization of the list. However, pushing and popping can be implemented using the "append" and "pop" methods respectively, and these can be implemented to access the back of the list, so potentially O(1) efficiency is also achievable.

It should be noted that the singly linked lists may have poor cache locality, which can reduce their speed in some instances and potentially cause them to be slower than dynamic arrays (ref 2). This occurs when accessing one element in a singly linked list requires following the next pointer to the next node in memory and this location may not be adjacent to the current node.

So for the case of time complexity, the SLL and the Python List have comparable efficiency in the best case scenarios.

However, for Space Complexity, the SLL outperforms the Python List because the dynamic allocation (doubling of memory whenever the previous array size is exhausted) means that some memory is spare and wasted as the data storage increases prior to the next memory doubling operation.



From: https://www.enjoyalgorithms.com/blog/dynamic-array

By contrast, SLL memory is only allocated for each element and the next pointer, whereas dynamic arrays allocate memory for a fixed number of elements.

Although the SLL may not be superior to the Python List for Time Complexity performance, it is more efficient than the Python List for Space Complexity performance, hence, it has been selected as the implementation for the Stack.

2. CODE IMPLEMENTATION

2.1 Additional Algorithmic Complexity considerations

Push and Pop operations are frequently executed but each operation only has O(1) Space and Time complexity.

The two other dominant operations (in terms of Time and Space complexity) that contribute to the upper bound performance of the overall algorithm performance are the following:

1. Reading and Writing Operations

   Reading and Writing Operations involve traversing the Prefix string in a single loop and using push and pop operations to store and read from the Prefix expression string. Hence these are O(n) operations (discounting other constant time sub-operations such as reading and writing the single character etc.)

2. Deep copying of the stack

   When functions are called to modify the stack, such as applying pop operations to examine the contents of the stack without replacement, the original stack is modified in Python which demonstrates that the object is passed by reference.

   Therefore, to avoid modifying the original stack, modifications are carried out on a deep copy of the stack instead of the original.

   As modification operations are performed on the stack frequently throughout the code, the deep copy operation is executed frequently and is a major contributor affecting Time and Space complexity performance of the algorithm.

   Since a new node needs to be created for each element and to store them in memory, the space complexity of a deep copy for a singly linked list implementation of a stack is O(n), where n is the number of elements in the stack. Therefore, the space complexity of a deep copy is also linear with respect to the number of elements in the stack.

In summary, from the above considerations, the overall performance of the Prefix to Postfix algorithm is generally a constant multiple of n (based on the total sum of reading and writing and deep copy operations run in sequence), giving rise to O(n) for the overall time and space complexity performance.

2.2 Incorporated Code Enhancements

The implemented Python code incorporates the following enhancements:

- Additional Prefix to Infix conversion with parentheses bracketing
- Comprehensive file IO error checking:
  - Incorrect file path. Reproduces input file path for troubleshooting.
  - Incorrect file type for both input and output files.
  - Blank file error due to empty file.
- Comprehensive error checking and reporting for the Prefix expression:
  - Flexible accommodation of whitespace (whitespace is ignored in the output and the Prefix expression is reported without whitespace)
  - Catching of illegal symbols and reporting in the output file
  - Checking to ensure that the first and last characters in the Prefix expression are an operator and operand respectively
  - Checking for correct number of operators (one less than total number of operands)

2.3 Lessons Learnt

The following new knowledge was obtained from this programming exercise:

- Understanding the differences in space complexity for implementing a Stack using a singly linked list versus a dynamic array and the considerations for pushing and popping from the front, versus the back of the data structure on algorithmic complexity performance. In general (from ref 3):

<u>Dynamic Array</u>	
Pushing (Appending)	: O(n) (O(1) from the end)
Popping	: O(n) (O(1) from the end)
<u>Singly Linked List</u>	
Pushing	: O(n) (O(1) at the beginning)
Popping	: O(n) (O(1) at the beginning)

- Understand dynamic memory allocation for a linked list versus fixed memory allocation for an array implementation of a stack.
- Understanding that modifications to a stack are passed by reference and how to implement a stack deep copy to bypass this issue.
- Implications of frequency of Python garbage collection operations to clean up unused object storing variables on space efficiency.
- Implement a Prefix to Postfix conversion using recursion.
- Understand how to optimise a conventional recursion algorithm using helper functions (memorization). Use tail recursion to eliminate the call stack build up.
- How to modularize and package code in Python
- Implement comprehensive exception handling for file IO using standard Exception Classes and printing to stderr.
- Understand the advantages of using an argument parser to facilitate command line operations.

2.4 Future Considerations

Many copies of the original character stack were made by deep copy, and some of the copies were not completely empty at the end of an operation, which could mean that excess space is used to store these redundant copies. In the future, the code should be optimized to undertake deep copies only when absolutely necessary, and to ensure that any duplicates are emptied (deleted) by the Python garbage collector at a regular frequency.

I would also have liked to run some comparison performance metrics on an implementation of the stack using a dynamic array (list) versus the singly linked list using a varying size of dataset to obtain a feel for the performance.

Lastly, I would have liked to implement the ability to perform Prefix to Postfix conversions for numeric values and producing a final numeric value. This would also account for the possible use of more than single digit values, which will require some form of concatenation when reading from the Prefix string.

3. DISCUSSION OF ALTERNATIVE RECURSIVE APPROACH

3.1 Overview

The structure of the prefix to postfix conversion algorithm also naturally suits a recursion approach since the prefix notation may be defined recursively (refer below algorithm).

In this case the Prefix expression will be traversed / read from left to right and the following sequence of operations may be repeated recursively (ref 4):

- **STEP 1** - *The first character in a Prefix expression is always operator. The recursive method removes the first character and save it to a variable, say 'op'.*
- **STEP 2** - *Next the method calls itself twice recursively to get the following two operands, say 'op1' and 'op2'.*
- **STEP 3** - *A list is declared to store the resultant postfix sub-expression, which is formed by combining:*
  op1 + op2 + op
- **STEP 4** - *The base case of the recursion is reached when a sub-expression is reduced to a single operand, which is then added to the postfix expression.*
- **STEP 5** - *When the last recursion returns from the call stack, the list has the final postfix expression.*

The size of the data set is reduced by 1 with each successive iteration and since the work done at each level is O(1), the Time and Space complexity is O(n).

3.2 Comparison against conventional iteration approach

The adoption of either a recursive or iterative approach to converting Prefix to Postfix has its advantages and disadvantages.

As demonstrated in the previous section, the prefix to postfix conversion has a natural recursive structure and adopting recursion can be natural and intuitive.

The recursive code can also be very concise versus the iterative approach.

However, the use of the iterative approach eliminates the need to make repeated function calls, which is a memory overhead.

The main disadvantage of the recursion implementation is the requirement of memory overhead by making calls to the function stack and subsequent potential for stack overflows, particularly when large prefix expressions are passed.

Notwithstanding, this shortcoming may be largely overcome by implementing a tail-recursive algorithm instead of a conventional recursive approach (ref 5). This would involve invoking the use of helper functions to "remember" the results of preceding function operations, thereby reducing calling the stack only once (ref 6).

In summary, neither approach can be considered 'better' in all categories and an efficient implementation is possible in both cases.

REFERENCES:

1. (2022, May 23). How to Add Elements in List in Python? Scaler Topics. Retrieved March 1, 2023, from https://www.scaler.com/topics/how-to-add-elements-in-list-in-python/
2. (n.d.). Types of Linked Lists, Memory allocation, and cache locality. Learnsteps. Retrieved March 1, 2023, from https://www.learnsteps.com/types-of-linked-lists/
3. Wengrow, J. (2018). A Common Sense Guide to Data Structures and Algorithms (2nd ed., p. 238). The Pragmatic Programmers.
4. (2022, July 11). Prefix to postfix (2 solutions) – stack and recursion. La Viviene Post. Retrieved March 1, 2023, from https://www.lavivienpost.com/convert-prefix-to-postfix-code/
5. (n.d.). Tail call elimination. Functional Programming in Elm. Retrieved March 1, 2023, from https://functional-programming-in-elm.netlify.app/recursion/tail-call-elimination.html
6. Eliasen, D. (2018, November 19). How to build up an intuition for recursion. Free Code Camp. Retrieved March 1, 2023, from https://www.freecodecamp.org/news/how-to-build-up-an-intuition-for-recursion-986032c2f6ad/