COMP47500 ADVANCED DATA STRUCTURES IN JAVA

ASSIGNMENT - 1

INFIX TO POSTFIX EXPRESSION CONVERSION AND EVALUATION USING STACK

Assignment Type of Submission:			
Group	Yes/No	**List all group members' details:**	**% Contribution Assignment Workload**
COMP47500 **Advanced Data Structures** **Group Assignment - 1**	Yes	Student Name: Avantika Sivakumar Student ID: 23205789	33.33% (Code implementation, material gathered for report, video explanation)
		Student Name: Ria Rahul Asgekar Student ID:23203987	33.33% (Results evaluation, material gathered for report, video explanation)
		Student Name: Jeevan Raju Ravi Govind Raju Student ID: 23207314	33.33% (Problem Identification, material gathered for report, video explanation)

1. Problem Domain Description:

Infix to Postfix Expression Conversion and Evaluation using stack data structure in Java.

In Computer Science, the infix to postfix expression conversion is also called Reverse Polish Notation or RPN. It is a fundamental operation used in parsing and evaluating mathematical expressions. This is the process of converting an expression written in infix notation, where operators are placed between operands, to postfix notation, where operators follow their operands.

The evaluation of postfix expressions involves interpreting and calculating the result of the expression based on postfix notation. This process typically utilises a stack data structure to manage operands and intermediate results during evaluation.

For this assignment, our problem domain includes:

1.1. Infix to Postfix Conversion:

- Implementing a Java program using stack data structure to convert infix expressions to postfix notation.
- Handling operators precedence and associativity rules (e.g., parentheses, multiplication, division, addition, and subtraction).
- Ensuring proper handling of parentheses to maintain the correct order of operations.
- Utilising a stack data structure to store operators temporarily during conversion.

1.2. Infix and Postfix Expression Evaluation:

- Develop a Java program to evaluate infix/postfix expressions to get a single result.
- Implementing a stack data structure to manage operands/operators and intermediate results during evaluation.
- Performing arithmetic operations (addition, subtraction, multiplication, division) according to infix/postfix notation.
- Handling potential errors such as division by zero or invalid expressions.

1.3. User Interface:

- Providing a test.java interface for users to input infix expressions for running this program.
- Displaying the steps involved during conversion.
- Displaying the corresponding postfix notation after conversion.
- Showing the result of postfix expression evaluation.

1.4. Error Handling:

- Detecting and handling various types of errors such as invalid input expressions or division by zero during evaluation.
- Providing informative error messages to users.

1.5. Testing and Validation:

- Conducting thorough testing to ensure the correctness and robustness of the implementation.
- Validating the program with different types of infix expressions, including edge cases and complex expressions(using alphabets, invalid inputs, etc).
- Verifying that the converted postfix expressions are correct and that the evaluation produces accurate results.

1.6. Benefits:

1. No need for the concept of parentheses and precedence rules etc. in a postfix expression.
2. Any formula can be expressed without parenthesis in a postfix expression.
3. It is very convenient for evaluating formulas on a computer with stacks.
4. Postfix is slightly easier to evaluate for machines, although conversion takes extra time.
5. It reflects the order in which operations are performed.

6. You need not worry about the left and right associativity.

By addressing these aspects within the problem domain, this Java program aims to provide a reliable and efficient solution for converting infix expressions to postfix notation and evaluating the resulting expressions and displays impact towards this specific problem domain.

2. Theoretical Foundations of the Data Structure(s) utilised

The **stack** data structure is widely utilised in various algorithms and applications. Its theoretical foundations revolve around its applications in managing data in a Last-In-First-Out (LIFO) manner. LIFO implies that the element that is inserted last, comes out first and FILO implies that the element that is inserted first, comes out last. It is a linear data structure, meaning it follows a particular order in which operations are performed.

Real-life examples of a stack: a deck of cards, piles of books, piles of money.
This example allows you to perform operations from one end only, like when you insert and remove new books from the top of the stack.

Theoretical foundations of the stack data structure:

2.1. Definition:

- A stack serves as a collection of elements with two primary operations: push and pop.
- The push operation adds an element to the top of the stack, while the pop operation removes the top element from the stack.
- Stacks typically allow access only to the top element (the most recently added) via a peek or top operation.
- Stacks can be implemented using various underlying data structures such as arrays, linked lists, or dynamic arrays.

2.2. LIFO Principle:

The core principle of stacks is **Last-In-First-Out (LIFO**), meaning the last element added to the stack is the first one to be removed.
This property makes stacks suitable for conversion and modelling scenarios where elements are added and removed in a reverse order of their arrival.

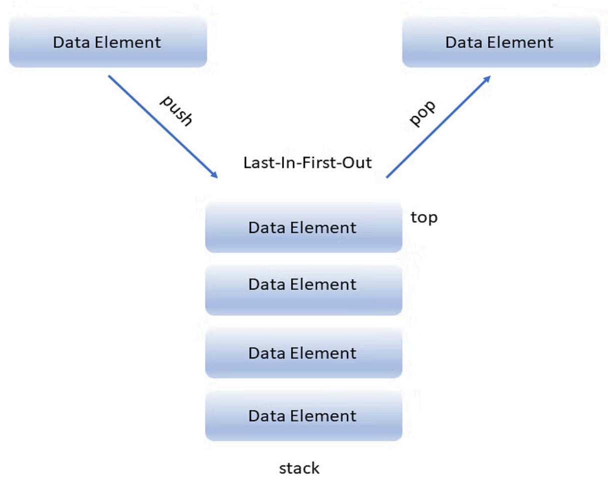


Fig. 1: Diagram depicting flow of data using a stack

2.3. Operations:

- **Push**: Adds an element to the top of the stack.
- **Pop**: Removes and returns the top element of the stack.
- **Peek** (or Top): Returns the top element of the stack without removing it.
- **isEmpty**: Checks if the stack is empty.
- **Size**: Returns the number of elements in the stack.

Push Operation: One of the fundamental operations to insert data into a stack. We have only one end to insert an element, this is the top of the stack.

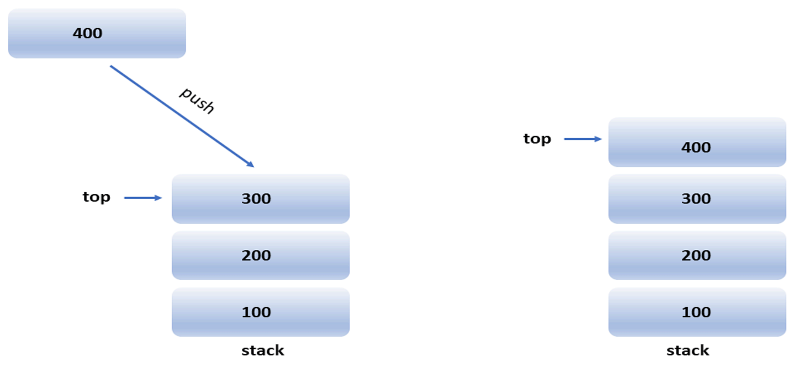


Fig. 2: Diagram depicting insertion operation(push) of an element '400' into a stack

Pop Operation: One of the fundamental operations to delete/remove data from a stack. We have only one end to delete an element, this is the top of the stack.

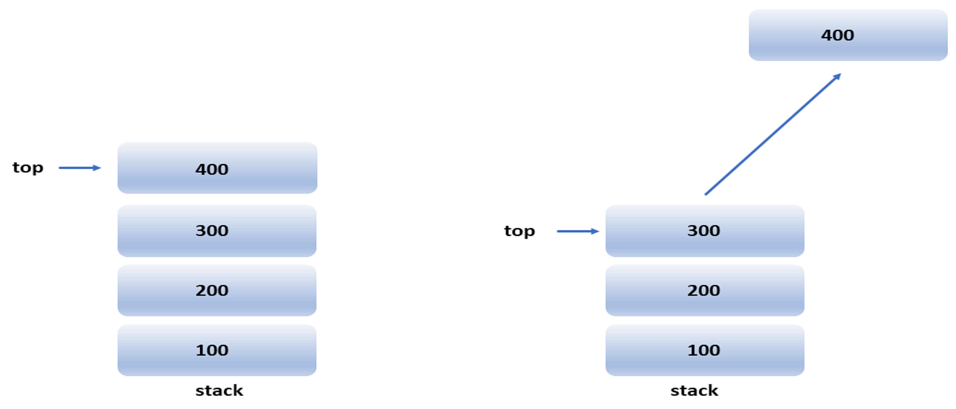


Fig. 3: Diagram depicting deletion operation(pop) of an element '400' from a stack

3. Applications of Stack:

1. **Expression evaluation:** Used during parsing and evaluating expressions, such as infix to postfix conversion and postfix expression evaluation.
2. **Function call management:** In programming languages to manage function calls, storing information such as local variables and return addresses.
3. **Undo mechanisms:** In applications that require undo functionality, such as text editors, where the history of operations is stored on a stack. This aligns with LIFO Principle so a direct application.
4. **Backtracking algorithms:** Utilised in backtracking algorithms to maintain the state of choices made during exploration of a search space.

In the context of this problem domain of infix to postfix conversion and postfix expression evaluation, stacks play a crucial role in managing operators and operands during both conversion and evaluation processes.

4. Time and Space Complexities:

4.1. Space Complexity:
The space complexity of an array-based stack is **O(n)**, where n is the maximum number of elements the stack can hold.
This is because the array needs to allocate memory for a fixed-size array to hold the elements.

4.2. Time Complexity:
- **Push Operation: O(1)** amortised time complexity. Appending an element to the end of an array usually takes a constant time. However, in some cases, when the array needs to be resized (e.g., when it reaches capacity), the time complexity can become **O(n)** due to the need to copy elements to a new array.

- **Pop Operation: O(1)**. Removing the last element of an array takes a constant time.
- **Peek Operation: O(1).** Accessing the last element of an array takes a constant time.
- **Search Operation: O(n)**. Searching for an element in an unsorted array-based stack may require iterating through all elements in the worst case.

5. Advantages of infix to postfix conversion:

Reverse Polish Notation (RPN), also known as postfix notation, and the process of converting infix expressions to postfix notation play significant roles in computer science and mathematics.

5.1. Simplicity and Clarity:

RPN makes expressions simpler by eliminating the requirement for parentheses and precedence rules. Each operator immediately follows its operands, resulting in a more clear and unambiguous syntax.

5.2. Ease of Evaluation:

Postfix expressions can be evaluated efficiently using a stack-based algorithm, making them particularly suitable for implementation in computing systems.
The evaluation algorithm for postfix expressions is straightforward and requires minimal processing overhead, leading to faster computation compared to infix expressions. A single left-to-right scan is enough for postfix evaluation.

5.3. Elimination of Ambiguity:

Converting infix expressions to postfix notation removes ambiguity that may arise due to operator precedence and parentheses. Each operator's position unambiguously defines its precedence and ensures a consistent evaluation order.

5.4. Compiler Design and Parsing:

RPN is commonly used in compiler design and parsing algorithms, where expressions need to be analysed and processed efficiently.
Converting infix expressions to postfix notation is an essential step in building parsers for programming languages, mathematical software, and expression evaluation engines.

5.5. Calculator Implementation:

Postfix notation is often used in calculator implementations because it simplifies expression input and evaluation.
Calculators benefit from postfix notation as it allows users to enter expressions without the need for parentheses or explicit precedence rules.

5.6. Expression Trees:

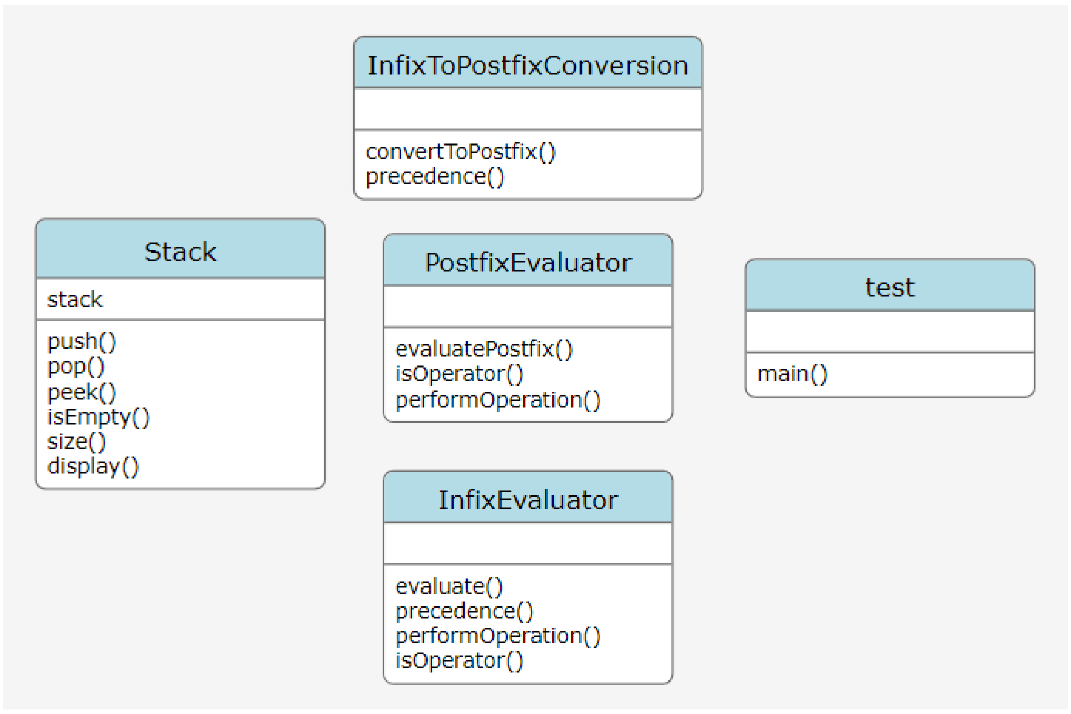
In some applications, postfix expressions are used to represent expression trees, which are data structures used to represent mathematical expressions.
Postfix notation naturally lends itself to building expression trees, where each operator becomes a node and each operand becomes a leaf.

6. History about postfix notation:

The first computer to employ postfix notation, in Germany, was Konrad Zuse's Z3 in 1941 and Z4 in 1945. Arthur Burks, Don Warren, and Jesse Wright introduced the reverse Polish system again in 1954, then Friedrich L. Bauer and Edsger W. Dijkstra recreated it separately in the early 1960s to decrease computer memory access and evaluate expressions on the stack. During the 1970s and 1980s, Hewlett-Packard utilised RPN in all of their desktop and handheld calculators, and it is still used in certain versions in the 2020s. In computer science, stack-oriented programming languages like Forth, dc, STOIC, PostScript, RPL, and Joy employ reverse Polish notation.

7. Analysis/Design (UML Diagram(s))

Below is an example of a simplified UML (Unified Modeling Language) diagram depicting the design of a Java program for infix to postfix expression conversion and evaluation using a stack data structure:



Explanation:
There are 4 main classes implementing the algorithms, and the test class is used for results and analysis.
The stack class has a data member stack, of type ArrayList. The stack has member functions like push(), pop(), and peek(), to insert, delete, and view the element at the top of the stack.
The InfixToPostfixConversion class has a method convertToPostfix() that takes in the infix expression as input, and uses a stack to convert it to a postfix expression. This method also considers the precedence of operators to compute the equivalent postfix expression.

The PostfixEvaluator and InfixEvaluator classes have functions to compute the final result of the expressions. They have functions to perform each unit expression, that is, compute the result of 2 operands and an operator.
The test class has a main method that calls the three classes that use stacks, to get the results, and also the time taken by each algorithm.

This UML diagram provides a high-level view of the design, showing the key classes and methods involved in the process.

Example output:

An infix expression is a mathematical expression where operators are placed between operands. It's the standard way we write expressions in mathematics. For example:

Infix expression: 3 + 4 * 2 / (1 - 5)^2

The repeated scanning makes it very inefficient. Infix expressions are easily readable and solvable by humans whereas the computer cannot differentiate the operators and parentheses easily so, it is better to convert the expression to postfix form before evaluation.

In contrast, a postfix expression, also known as Reverse Polish Notation (RPN), is a mathematical expression in which every operator follows all of its operands. This notation eliminates the need for parentheses to denote the order of operations. Here's the same expression in postfix notation:

Postfix expression: 3 4 2 * 1 5 - 2 ^ / +

To evaluate a postfix expression, you would scan it from left to right, pushing operands onto a stack and applying operators whenever they are encountered.

Let's briefly go through the evaluation steps for the above postfix expression:

Algorithm 1: Infix to Postfix Conversion
In postfix evaluation, we scan the expression from left to right. When an operand is encountered, we push it onto the stack. When an operator is encountered, we pop the required number of operands from the stack, perform the operation, and push the result back onto the stack. We continue this process until we've evaluated the entire expression, resulting in the final answer.
Below are the steps to implement the above idea:

1. Scan the infix expression **from left to right**.
2. If the scanned character is an operand, put it in the postfix expression.
3. Otherwise, do the following
    - If the precedence and associativity of the scanned operator are greater than the precedence and associativity of the operator in the stack [or the stack is empty or the stack contains a ‘**(**‘], then push it in

the stack. [‘**^**‘ operator is right associative and other operators like ‘**+**‘,’**–**‘,’*****‘ and ‘**/**‘ are left-associative].

        - Check especially for a condition when the operator at the top of the stack and the scanned operator both are ‘**^**‘. In this condition, the precedence of the scanned operator is higher due to its right associativity. So it will be pushed into the operator stack.
        - In all the other cases when the top of the operator stack is the same as the scanned operator, then pop the operator from the stack because of left associativity due to which the scanned operator has less precedence.
    - Else, Pop all the operators from the stack which are greater than or equal to in precedence than that of the scanned operator.
        - After doing that Push the scanned operator to the stack. (If you encounter parenthesis while popping then stop there and push the scanned operator in the stack.)
4. If the scanned character is a ‘**(**‘, push it to the stack.
5. If the scanned character is a ‘**)**’, pop the stack and output it until a ‘**(**‘ is encountered, and discard both the parenthesis.
6. Repeat steps **2-5** until the infix expression is scanned.
7. Once the scanning is over, Pop the stack and add the operators in the postfix expression until it is not empty.
8. Finally, print the postfix expression.

Algorithm 2: Postfix Evaluation

1. First initialise an empty stack to store the operands.
2. Then we scan the expression from left to right.
3. For each character in the expression:
    - If it is an operand, we push it onto the stack.
    - If it is an operator, we pop the required number of operands from the stack, perform the operation, and push the result back onto the stack.
4. We continue this process until the entire expression is scanned.
5. Once the expression is fully evaluated, the result will be the only item left on the stack and displayed as a single value.

Algorithm 3: Infix Evaluation

1. Create two stacks: one for operands and one for operators.
2. Iterate through each character of the input expression(Steps 3-6).
3. If the character is a digit, push it onto the operand stack after converting it to an integer.
4. If the character is an opening parenthesis '(', push it onto the operator stack.
5. If the character is a closing parenthesis ')', perform operations until the corresponding opening parenthesis '(' is found.
6. When encountering an operator:

a. While the operator stack is not empty and the precedence of the current operator is less than or equal to the precedence of the operator on the top of the stack, perform the operation.

b. Push the current operator onto the operator stack.

7. After iterating through the expression, perform any remaining operations in the operator stack.
8. Pop from stack to get result.

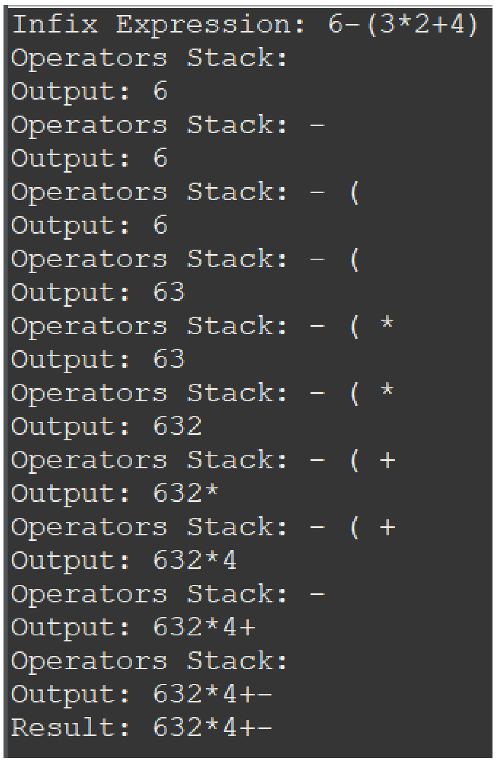
Additional methods:

Precedence function: Assign precedence values to operators (+, -, *, /, ^).

Perform operation function: Evaluates a simple expression with two operands and one operator.

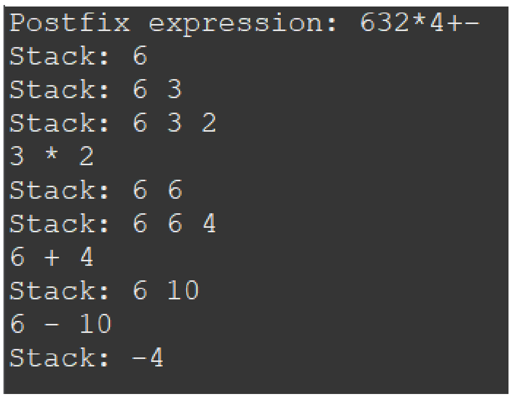
Is operator function: Check if a given character is one of the supported operators (+, -, *, /, ^).

9. RUN:



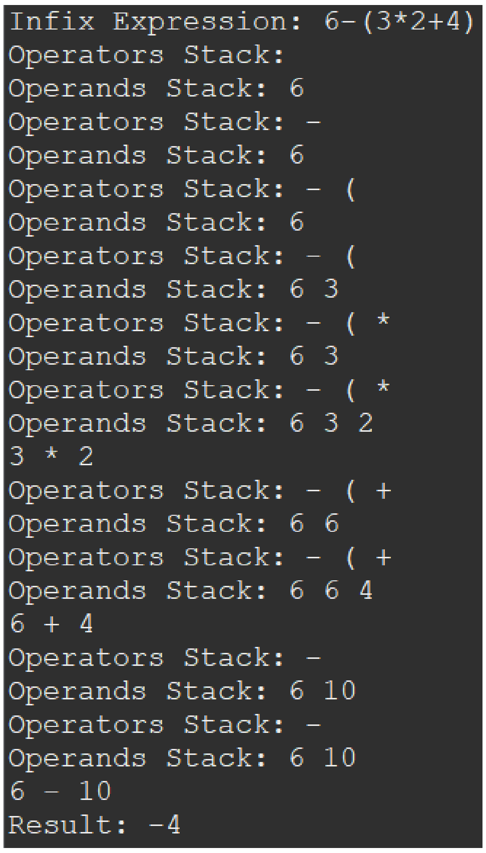
```
Infix Expression: 6-(3*2+4)
Operators Stack:
Output: 6
Operators Stack: -
Output: 6
Operators Stack: - (
Output: 6
Operators Stack: - (
Output: 63
Operators Stack: - ( *
Output: 63
Operators Stack: - ( *
Output: 632
Operators Stack: - ( +
Output: 632*
Operators Stack: - ( +
Output: 632*4
Operators Stack: -
Output: 632*4+
Operators Stack:
Output: 632*4+-
Result: 632*4+-
```

Fig. 4: Infix to postfix conversion using stack



```
Postfix expression: 632*4+-
Stack: 6
Stack: 6 3
Stack: 6 3 2
3 * 2
Stack: 6 6
Stack: 6 6 4
6 + 4
Stack: 6 10
6 - 10
Stack: -4
```

Fig. 5: Postfix expression evaluation



```
Infix Expression: 6-(3*2+4)
Operators Stack:
Operands Stack: 6
Operators Stack: -
Operands Stack: 6
Operators Stack: - (
Operands Stack: 6
Operators Stack: - (
Operands Stack: 6 3
Operators Stack: - ( *
Operands Stack: 6 3
Operators Stack: - ( *
Operands Stack: 6 3 2
3 * 2
Operators Stack: - ( +
Operands Stack: 6 6
Operators Stack: - ( +
Operands Stack: 6 6 4
6 + 4
Operators Stack: -
Operands Stack: 6 10
Operators Stack: -
Operands Stack: 6 10
6 - 10
Result: -4
```

Fig. 6: Evaluation of infix expressions using stack

10. SET OF EXPERIMENTS RUN AND RESULTS:

The test cases consist of infix expressions of varied lengths/number of operators and operands.

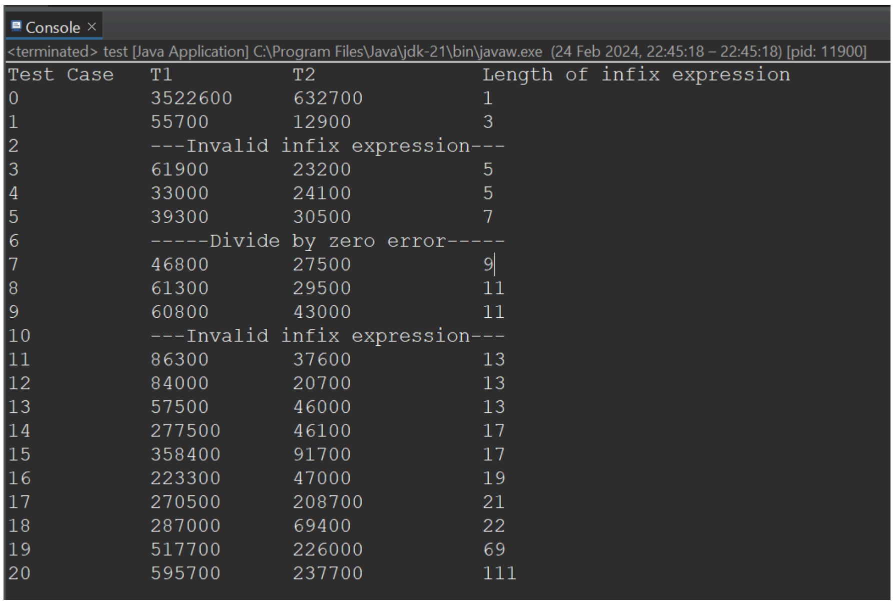
infixExpressions = { "7", "6/2", "a+1", "1+4*7", "5+8-6", "5-0-9*6", "2+3+4-5/0", "6-(3*2+4)", "3+8-3*(4+8)", "4+3/3+9*2/1", "9*8+3/1)-5*2", "6+3+5+4-5*7*3", "(1+2+3)/(1*3)", “(4/2-1*8/2)/1", "4+(3-2+(7*3+3/1))", "5*(9+4-2+4*7)*3-5", "(9+7+9*2+5)*(2+6-4)", "8*8*5*3/2-9+2+6*(4+5)", "9/4+8-5*7-9+2/4(7+5*2)", “(5*8+3)(4*7*1/3)/(2+1-5-4/2)(2+6)-2*5+7*(8+9/3+6)+(5+4*(8+6/2+7/2)/3)", "5+3+6*(9+2*3*4)-5(2+3-4*7)-5*4(8/4)+2-3+7-4+9*0-(8+9/3)+(3*(8+6/2+7/2)/3)*(4+8-6+3)(2+5-3+9-2)(2+1-5-4/2)*(2+6)" };

The test cases also include edge cases such as:

1. Invalid Expressions(contains other characters, unbalanced parentheses): "a+1", "9*8+3/1)-5*2"
2. Expressions that give arithmetic errors(divide by zero error): "2+3+4-5/0"

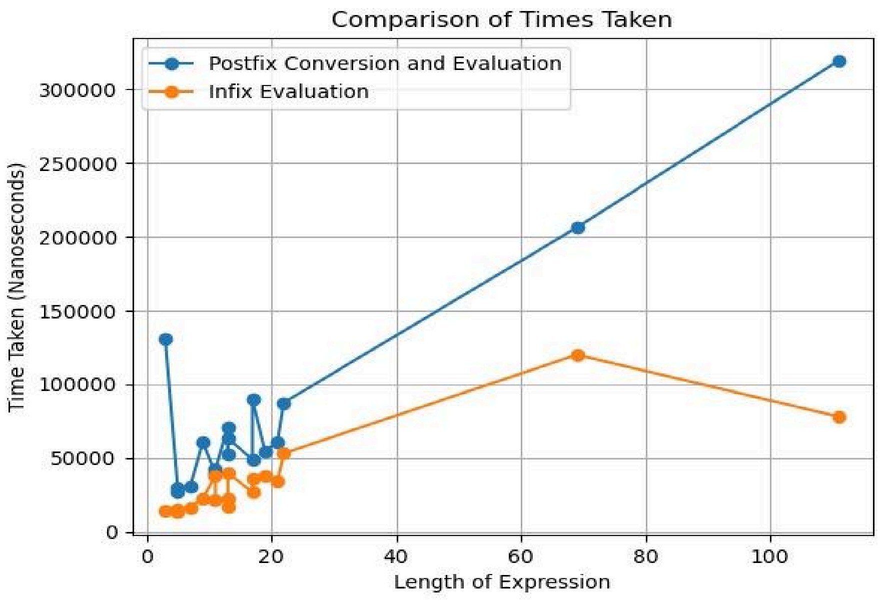
Each of these infix expressions is passed to the 3 algorithms and the time taken for converting the expression to postfix and evaluating it, and the time taken for directly evaluating the infix expressions is tabulated, along with the length of the expression.

Results of run time obtained:



```
Console ×
<terminated> test [Java Application] C:\Program Files\Java\jdk-21\bin\javaw.exe  (24 Feb 2024, 22:45:18 – 22:45:18) [pid: 11900]
Test Case    T1           T2              Length of infix expression
0            3522600      632700          1
1            55700        12900           3
2            ---Invalid infix expression---
3            61900        23200           5
4            33000        24100           5
5            39300        30500           7
6            -----Divide by zero error-----
7            46800        27500           9
8            61300        29500           11
9            60800        43000           11
10           ---Invalid infix expression---
11           86300        37600           13
12           84000        20700           13
13           57500        46000           13
14           277500       46100           17
15           358400       91700           17
16           223300       47000           19
17           270500       208700          21
18           287000       69400           22
19           517700       226000          69
20           595700       237700          111
```

Graph: The results are plotted for better visualisation.



The first data point is omitted from the graph as that test case takes extra time, since it is the first iteration of the loop and the code is optimised by the JIT compiler during the initial runs. It is for this reason that the second data point in our table(first in our graph) is also considerably higher than the others. It is observed that as the length of the expression increases, the time taken for the algorithm also increases. Conversion to postfix and then evaluation of the postfix expression takes longer than directly evaluating the infix expression using two stacks.

Code Implementation GitHub (link):
https://github.com/RiaAsgekar/COMP47500_Assignment1

Set of Experiments run and results:
https://github.com/RiaAsgekar/COMP47500_Assignment1/blob/main/src/assignment1/test.java

Comments: test.java has test cases of infix expressions covering various complexities and edge cases and the algorithms(infix evaluation, infix to postfix and postfix evaluation) are run and the time taken is tabulated.

Video of the Implementation running

Zoom (link & password):
https://ucd-ie.zoom.us/rec/share/Wl61A4mhAvQoA8RzK1eKAzHCdAOBm5nxv1K9e9ejhxvBcWSTpHg97EQAI5NAqALt.5toNZFZ8UWH7agno

Passcode: **NhtM7q&a**

Comments:
Ria gives an overview of our chosen problem domain, explains stacks, principle involved, program structure.

Jeevan explains about the three algorithms involved:
1. Infix to postfix expression conversion with example output
2. Postfix expression evaluation with example
3. Infix expression evaluation with example

Avantika implements the program and talks about the test cases involved. Highlighting the important test cases, edge cases handled like check input for valid characters, division by zero and so on.

References:

1. https://www.simplilearn.com/tutorials/data-structure-tutorial/stacks-in-data-structures#:~:text=Real%2Dlife%20 examples%20of%20a,the%20top%20of%20the%20stack.
2. https://en.wikipedia.org/wiki/Stack_(abstract_data_type)

3. https://www.geeksforgeeks.org/convert-infix-expression-to-postfix-expression/
4. https://stackoverflow.com/questions/66736516/infix-to-postfix-conversion-in-java-using-stack-and-queue-adts
5. https://www.prepbytes.com/blog/stacks/infix-to-postfix-conversion-using-stack/
6. The Art of Computer Programming, Volume 1: Fundamental Algorithms by Donald Knuth
7. *Introduction to Algorithms by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein*