Algorithm & Data Structures Report

Section 1: The essence of your solution to the challenge in a single paragraph.

The solution tackles the challenge of evaluating postfix expressions in C++ by implementing a postfix expression evaluator. The program takes a user-input postfix expression containing variables (A-Z) and numeric constants and evaluates it, considering user-provided values for the variables. It supports basic arithmetic operations (+, -, *, /), modulus (%), power (^), equal (==), factorial (!) and three custom operators that i created (@, #, &). The essence lies in using a stack-based approach for expression evaluation and handling variables through an array-based symbol table. The primary objective of the solution is to develop a robust postfix expression evaluator in C++, capable of handling arithmetic operations and variables within an expression. This program not only serves as a tool for mathematical computations but also enables the incorporation of user-specified values for variables. By utilizing a stack-based approach and an array-based symbol table, the solution efficiently processes postfix expressions, providing users with a flexible and powerful tool for mathematical expressions with variables.

Section 2: An explanation of the original algorithms of your solution in non-technical language. You do not need to describe the workings of any standard algorithm that might comprise part of your solution.

The core algorithm uses a stack to process postfix expressions sequentially. It loops through each character of the expression, pushes the numeric operands onto the stack and processes them, and performs the operation when the operator is found. The value of the variable is looked up from an array-based symbol table. This algorithm guarantees the correct order of operations and repeatedly evaluates the expression. The program also includes custom operators that extend basic arithmetic functionality. The core algorithm is based on a batch-based methodology for systematically evaluating postfix expressions. It loops through each character in the expression to distinguish between numeric constants and variables. Numeric operands are seamlessly pushed onto the stack while variable values are retrieved from an array-based symbol table. The code systematically iterates through each character in the postfix expression, intelligently handling operands, operators, and variable assignments. The use of a stack facilitates the storage of intermediate results, ensuring a streamlined evaluation process. Variable assignments are seamlessly integrated into the main loop, enabling user interaction. The switch statement efficiently manages various operators, including custom ones, executing the corresponding operations. I created my own custom operators which further augment the basic arithmetic capabilities, allowing for increased versatility. The simplicity of the approach lies in its clarity and ease of comprehension, fostering accessibility for users seeking a reliable postfix expression evaluator.

Section 3: Pseudocode for each original algorithm. You are urged to follow the pseudocode conventions of Cormen at al, Chapter 2. Pay great attention to how you lay out your pseudocode. In particular, take care to use structured indentation. Pseudocode that is not correctly indented, or is otherwise unreadable, will not be marked.

Pseudocode for Algorithm :

```
Algorithm EvaluatePostfix(exp: string) returns integer
    Initialize stack st
    for each character c in exp
        if c is digit
            operand = c - '0'
            st.push(operand)  // Push numeric operand onto the stack
        else if c is letter
            variableValue = variables[c - 'A']
            st.push(variableValue)  // Push variable value onto the stack
        else
            switch c
                case '+':
                    val1 = st.top()
                    st.pop()
                    val2 = st.top()
                    st.pop()
                    st.push(val2 + val1)  // Add two operands and push result
                case '-':
                    val1 = st.top()
                    st.pop()
                    val2 = st.top()
                    st.pop()
                    st.push(val2 - val1)  // Subtract val1 from val2 and push result
                case '*':
```

```
            val1 = st.top()
            st.pop()
            val2 = st.top()
            st.pop()
            st.push(val2 * val1)  // Multiply two operands and push result
        case '/':
            val1 = st.top()
            st.pop()
            val2 = st.top()
            st.pop()
            st.push(val2 / val1)  // Divide val2 by val1 and push result
        case '%':
            val1 = st.top()
            st.pop()
            val2 = st.top()
            st.pop()
            st.push(val2 % val1)  // Modulo val2 by val1 and push result
        case '^':
            val1 = st.top()
            st.pop()
            val2 = st.top()
            st.pop()
            st.push(pow(val2, val1))  // Raise val2 to the power of val1 and push
result
        case '=':
            val1 = st.top()
            st.pop()
            val2 = st.top()
            st.pop()
            st.push(val2 == val1)  // Push boolean result of equality check
        case '!':
```

```
            val = st.top()
            st.pop()
            st.push(factorial(val))  // Push factorial of val onto the stack
        // Custom operator symbols that I created by playing around with the values and operators
        case '@':
            val1 = st.top()
            st.pop()
            val2 = st.top()
            st.pop()
            st.push(val2 * 2 + val1)  // Custom operator: Multiply val2 by 2, add val1, and push result
        case '#':
            val1 = st.top()
            st.pop()
            val2 = st.top()
            st.pop()
            st.push(val2 - val1 * 3)  // Custom operator: Subtract 3 times val1 from val2 and push result
        case '&':
            val1 = st.top()
            st.pop()
            val2 = st.top()
            st.pop()
            st.push(val2 + val1 / 2)  // Custom operator: Add val2 to half of val1 and push result
    return st.top()  // Return the final result
```

Pseudocode for Symbol Table:

```
Algorithm InitializeSymbolTable()
    Create an array symbolTable[MAX_VARIABLES]
    Set all elements in symbolTable to 0
```

```
    Return symbolTable

Algorithm AssignValueToVariable(symbolTable: array, variable: char, value: int)
    Let index be the position of variable in the alphabet (A=0, B=1, ..., Z=25)
    Set symbolTable[index] to value

Algorithm RetrieveValueFromVariable(symbolTable: array, variable: char) returns int
    Let index be the position of variable in the alphabet (A=0, B=1, ..., Z=25)
    Return symbolTable[index]

-- Usage in the main program --
SymbolTable symbolTable = InitializeSymbolTable()

For each character c in exp
    If c is a letter
        Print "Enter value for variable c: "
        Read user input as value
        AssignValueToVariable(symbolTable, c, value)
    End If
End For
```

Explanation of Algorithm Pseudocode:

The pseudocode reflects the logic of the original algorithm, emphasizing structured indentation for clarity. It outlines the processing of each character in the postfix expression, distinguishing between digits, letters, and operators. The switch statement guides the execution flow, ensuring proper handling of various operations, including custom operators and the factorial function. Moreover , the pseudocode outlines the postfix expression evaluation algorithm. It initializes a stack and iterates through each character in the input expression. Depending on the type of character (digit, letter, or operator), the algorithm performs specific actions such as pushing operands onto the stack or applying operators to the top elements of the stack. The switch statement handles various operators, including the custom operators (@, #, &).

Explanation for Symbol Table Pseudocode:

The pseudocode outlines a simple symbol table implementation to manage variable assignments. The InitializeSymbolTable algorithm creates an array symbolTable with a size of MAX_VARIABLES and initializes all elements to 0. The AssignValueToVariable algorithm allows assigning a value to a variable, where the position of the variable in the alphabet determines the index in the symbol table. The RetrieveValueFromVariable algorithm retrieves the value of a variable based on its position in the alphabet. In the main program, the symbol table is initialized, and for each variable encountered in the postfix expression, the user is prompted to enter a value, which is then assigned to the corresponding variable in the symbol table.

Section 4: A list of the data structures in your solution. Explain why each chosen data structure is suitable for the task.

Firstly, a stack is used to store and manage intermediate results during the postfix expression evaluation, showcasing its suitability for this iterative, last-in-first-out scenario. Secondly, an array is employed to represent variables (A-Z), providing a straightforward and efficient means to store and retrieve variable values. The choice of data structures plays a pivotal role in the efficiency and functionality of the solution. The stack serves as a dynamic structure for managing operands during expression evaluation, while the array-based symbol table efficiently stores and retrieves variable values. This selection aligns with the specific requirements of the task, providing a well-balanced approach to handling both numerical and symbolic components within postfix expressions. It relies on two primary data structures: a stack and an array. The stack is instrumental in managing the order of operations during expression evaluation. It allows for the systematic handling of operands and operators, ensuring the correct sequence of calculations. The array, serving as a symbol table, efficiently stores and retrieves variable values. The choice of these data structures is driven by their appropriateness for the specific tasks they perform within the postfix expression evaluation process. Last but not least I have also used recursive functions.The factorial function is implemented using recursion to calculate the factorial of a number and is called when the '!' operator is found in a postfix expression.This provides a clean, modular way to calculate factorials, and improves code readability and maintainability by separating the factorial calculation logic from the main evaluation loop (e.g., evaluatePostfix).

Section 5: Commented source code and a link to a short (< 5 minute) video that shows your code in execution. Ensure that source code is an exact implementation of your pseudocode. An efficient way to demonstrate the relationship between pseudo and actual code is to use lines of pseudocode as comments directly above their implementation in source code. The source code must be visible in the video, as well as the command line interface.

My Code in C++ App :

```cpp
#include <iostream>

#include <stack>
#include <cctype>
#include <cmath>
using namespace std;

const int MAX_VARIABLES = 26;

int variables[MAX_VARIABLES] = {0};

int factorial(int n) {
    if (n == 0 || n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}

int evaluatePostfix(string exp) {
    stack<int> st;

    for (int i = 0; i < exp.size(); ++i) {
        char currentChar = exp[i];

        if (isdigit(currentChar)) {
            int operand = currentChar - '0';
            st.push(operand);
        } else if (isalpha(currentChar)) {
            int variableValue = variables[currentChar - 'A'];
            st.push(variableValue);
        } else {
            switch (currentChar) {
                case '+': {
                    int val1 = st.top();
                    st.pop();
                    int val2 = st.top();
                    st.pop();
                    st.push(val2 + val1);
                    break;
                }
                case '-': {
                    int val1 = st.top();
                    st.pop();
                    int val2 = st.top();
                    st.pop();
                    st.push(val2 - val1);
```

```
            break;
        }
        case '*': {
            int val1 = st.top();
            st.pop();
            int val2 = st.top();
            st.pop();
            st.push(val2 * val1);
            break;
        }
        case '/': {
            int val1 = st.top();
            st.pop();
            int val2 = st.top();
            st.pop();
            st.push(val2 / val1);
            break;
        }
        case '%': {
            int val1 = st.top();
            st.pop();
            int val2 = st.top();
            st.pop();
            st.push(val2 % val1);
            break;
        }
        case '^': {
            int val1 = st.top();
            st.pop();
            int val2 = st.top();
            st.pop();
            st.push(pow(val2, val1));
            break;
        }
        case '=': {
            int val1 = st.top();
            st.pop();
            int val2 = st.top();
            st.pop();
            st.push(val2 == val1);
            break;
        }
        case '!': {
            int val = st.top();
            st.pop();
            st.push(factorial(val));
            break;
        }
```

```
                // Custom operator symbols that I created by playing around
with the values and operators
                case '@': {
                    int val1 = st.top();
                    st.pop();
                    int val2 = st.top();
                    st.pop();
                    st.push(val2 * 2 + val1);
                    break;
                }
                case '#': {
                    int val1 = st.top();
                    st.pop();
                    int val2 = st.top();
                    st.pop();
                    st.push(val2 - val1 * 3);
                    break;
                }
                case '&': {
                    int val1 = st.top();
                    st.pop();
                    int val2 = st.top();
                    st.pop();
                    st.push(val2 + val1 / 2);
                    break;
                }
            }
        }
    }

    return st.top();
}

int main() {
    string exp;
    cout << "Enter postfix expression: ";
    cin >> exp;

    for (char &c : exp) {
        if (isalpha(c)) {
            cout << "Enter value for variable " << c << ": ";
            cin >> variables[c - 'A'];
        }
    }

    cout << "Postfix evaluation: " << evaluatePostfix(exp) << endl;
    return 0;
}
```

Video Link:

https://drive.google.com/file/d/1nW2SJPiNNK_owp7qkrqOX_ioZvnsjVuB/view?usp=sharing

Section 6: Point out any defects of your design and/or implementation. Suggest remedies for these shortcomings.

One notable defect is the lack of comprehensive error handling for invalid expressions. Notably, the program lacks robust error handling, especially concerning invalid expressions and potential division by zero scenarios. Additionally, the input validation for user-assigned variable values is minimal, assuming correctness at all times. The documentation for custom operators (@, #, &) is insufficient, leaving room for ambiguity in understanding their intended behavior. To address this, we propose implementing checks for expression validity, such as ensuring the correct number of operands for each operator and validating proper variable input. Additionally, we recognize the need for enhanced error reporting to provide users with informative messages regarding the nature of errors during expression evaluation. Our solution's operator support is somewhat limited, and we suggest extending it to accommodate a broader range of operators and potentially allowing users to define custom functions. Lastly, input validation for variable values could be improved by implementing checks for valid numeric input. These proposed remedies aim to fortify the solution's robustness, user-friendliness, and versatility.

Conclusion:

The presented report encapsulates a comprehensive overview of our postfix expression evaluator, detailing its essence, algorithms, pseudocode, data structures, source code, and potential areas for improvement. The solution strikes a balance between technical depth and non-technical clarity, providing a holistic understanding of the implemented system.