1. **Problem Statement**

Create a program that takes an infix equation as input, the processes it into an expression tree and prints the traversals in prefix, infix, and postfix.

1. **Requirements**
   1. **Assumptions**

* A minimum of one space will be on each side of a character in the file
* Each line in the input contains only one infix expression to be processed
* Only single uppercase characters are used as operands
  1. **Specifications**
* Print a welcome message when the program starts
* Must read inputs from a file
  + Echo all inputs back to the user
  + User should enter the file name
    - Verify that the file exists
    - Verify the file is of the correct type (.dat file extension)
  + Check that the file is in the correct format
    - Each line should contain only one expression
    - Characters should have one space between them
* Convert the expression to postfix and print the operations that would be performed to solve the expression
* Build an expression tree of the equation
  + Traverse the tree and print the results in infix, postfix, and prefix
  + Evaluate the expression using prefix notation
    - Print each step of the evaluation and the corresponding result
  + Print the tree structure to the screen
* Expression variable values
  + A = 1, B = 2, C = 3, …, I = 9
* Operators are:
  + \*, /, +, -
* Print all errors and outputs that are printed to the screen to a file called output.dat as well
  + Verify that parentheses match up
  + Check for 2 operators for each operand
* Print a message stating that program has completed

1. **Decomposition Diagram**

Main

Process

Output

User enters the filename for the input file

Converts each expression to postfix and traverse it through an expression tree

Reads the expressions from the input file

Checks the file for errors

Evaluate the expression using prefix notation

Print all results and errors to the screen and to a file

Input

1. **Test Strategy**

* Valid
  + Test valid expressions and formats that should NOT throw any errors.
* Invalid
  + Test invalid expressions and formats that SHOULD throw errors.
* File Handling
  + Test file functions

1. **Test Plan Version 1**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Test Strategy | Test Number | Description | Input | Expected Output | Actual Output | Pass/Fail |
| Valid | 1.1 | Test expression with 1 operator and no parentheses or operands |  |  |  |  |
| Valid | 1.2 | Test expression with 2 values, + operator, no parentheses |  |  |  |  |
| Valid | 1.3 | Test expression with 2 values, - operator, no parentheses |  |  |  |  |
| Valid | 1.4 | Test expression with 2 values, \* operator, no parentheses |  |  |  |  |
| Valid | 1.5 | Test expression with 2 values, / operator, no parentheses |  |  |  |  |
| Valid | 1.6 | Test expression with 2 values, 1 operator, all enclosed in parentheses |  |  |  |  |
| Valid | 1.7 | Test expression with 3 values, operation with 2 values enclosed in parentheses, first operation added to 3rd operator |  |  |  |  |
| Valid | 1.8 | Test expression with 3 values, operation with 2 values enclosed in parentheses, first operation divided by 3rd operator |  |  |  |  |
| Valid | 1.9 | Test expression with 3 values, operation with 2 values enclosed in parentheses, first operation multiplied by 3rd operator |  |  |  |  |
| Valid | 1.10 | Test expression with 3 values, operation with 2 values enclosed in parentheses, first operation subtracted from 3rd operator |  |  |  |  |
| Valid | 1.11 | Test expression with a combination of operations inside and outside of the parentheses |  |  |  |  |
| Valid | 1.12 | Test order of operations at the same level: \*, /. +, - |  |  |  |  |
| Valid | 1.13 | Test order of operations at the same level: +, -, \*, / |  |  |  |  |
| Valid | 1.14 | Test order of operations at the same level: \*, +, /, - |  |  |  |  |
| Valid | 1.15 | Test order of operations at the same level: +, /. +, \* |  |  |  |  |
| Valid | 1.16 | Test order of operations between operations separated by parentheses (-)\*(+) |  |  |  |  |
| Valid | 1.17 | Test order of operations between operations separated by parentheses (\*)-(/) |  |  |  |  |
| Valid | 1.18 | Test order of operations between operations separated by parentheses (\*)/(+) |  |  |  |  |
| Invalid | 2.1 | Test an expression with 2 operands, 1 operator, and only an opening parenthesis at the begging and no closing parenthesis |  |  |  |  |
| Invalid | 2.2 | Test an expression with 2 operands, 1 operator, and only a closing parenthesis at the end |  |  |  |  |
| Invalid | 2.3 | Test an expression with 2 operands and no operators |  |  |  |  |
| Invalid | 2.4 | Test an expression with 1 operand and 1 operator |  |  |  |  |
| Invalid | 2.5 | Test an expression with no operands and 1 operator |  |  |  |  |
| Invalid | 2.6 | Test an expression with only a set of parentheses |  |  |  |  |
| Invalid | 2.7 | Test an expression with two operations and a missing closing parenthesis |  |  |  |  |
| Invalid | 2.8 | Test an expression with two operations and a missing opening parenthesis |  |  |  |  |
| Invalid | 2.9 | Test an expression with two operations enclosed in parentheses and no operator between them |  |  |  |  |
| File Handling | 3.1 | Input a file name that does not exist |  |  |  |  |
| File Handling | 3.2 | Input a file with a type other than “.dat” |  |  |  |  |
| File Handling | 3.3 | Input a file of type “.dat” that exists |  |  |  |  |
| File Handling | 3.4 | Input a file of type “.dat” that exists, but is empty |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |

1. **Initial Algorithm**

* \*\*\*NOTE: A call to the display function of the Logging class whenever displaying values to the screen. \*\*\*
* Main Method (called on program startup)
  + Log variable of type Logging
  + Call the openFile function of Log
    - Pass “output.dat” as a parameter
  + If log getFileOpen is true
    - Display “Output file opened successfully: output.dat”
  + Else
    - Display “Output file cannot be opened: output.dat”
  + Display a welcome message to the user
  + Ask the user to enter the location of the file they want to run through the program. (ex: [c://users//](../../../../c://users//)…//filename.dat)
  + If the file does not exist
    - Display an error message that it doesn’t exist and ask the user to enter another file name
  + If the file is empty
    - Display an error message that the file is empty and ask the user to enter another file name
  + If the file is of the wrong type (not “.dat”)
    - Display an error message that the file is of the wrong type and ask the user to enter another file name
  + Open the file and print a message that the file was opened successfully
  + For each line in the input file
    - read the line in and print it to the screen
    - string variable for the postfix expression
    - pass the expression to the function that converts infix to postfix
      * set the postfix variable equal to the return of this function
    - if the postfix variable is empty (an error must have occurred)
      * move to the next line in the file
    - if the postfix vairable is not empty (means it was converted successfully)
      * call the function displayPostfixOperations and pass the postfix string variable as a parameter
      * create a new node for the root of the expression tree called root
      * set the root node equal to the return of the convertPostfixToTree function
      * Call the displayTree function and pass the root node in as a parameter
      * string variable for postfix, set equal to the return of getPostfixFromTree (root node)
      * display the postfix string
      * string variable for prefix, set equal to the return of getPrefixFromTree (root node)
      * display the prefix string
      * string variable for infix, set equal to the return of getInfixFromTree (root node)
      * display the infix string
      * string solution variable equals the return of solutionFromPrefix (prefix string)
      * display solution variable “Final Result: <solution>”
  + Display “Thank you for using the program! Goodbye!”
* Function getTreeHeight
  + Accepts the root of a tree as an input parameter
  + Integer for height of tree
  + Integer for leftH and rightH
  + if root left child is not null
    - set leftH to getTreeHeight (left child)
  + if root right child is not null
    - set rightH to getTreeHeight (right child)
  + if rightH > leftH
    - set height to rightH
  + else set height to leftH
  + return height
* Function displayTree
  + Accepts the root of a tree as an input parameter and integer for the spacing
  + If the root is not null
    - Increase spacing by 5
    - Recursively call display tree on the right child of the root node
    - Display a new line
    - While index is less that spacing
      * Display a space
      * Increase index by 1
    - Display the character in the root and add a new line
    - Recursively call display tree on the left child of the root node
* Function solutionFromPrefix
  + Accepts an expression string in prefix notation
  + Create an integer stack
  + For each character in the input string
    - If character is an operand
      * Integer value equal to the return of getVariableValue (operand character)
      * Push integer value to the stack
    - Else
      * Integer1 equal to first pop off the stack
      * Integer2 equal to second pop off the stack
      * If character = “+”
        + Push integer1+interger2 to the stack
      * If character = “-“
        + Push integer1-integer2 to the stack
      * If character = “\*”
        + Push integer1\*integer2 to the stack
      * If character = “/”
        + If integer2 is not 0

Push integer1/integer2 to the stack

* + - * + Else

Display “ERROR: Cannot solve, division by zero: <integer1> / <integer2>”

Return “error”

* + Return the top of the stack as a string variable
* Function getVariableValue
  + Accepts a character as an input parameter
  + A=1, B=2, C=3, D=4, …, I=9
  + If input is not A through I
    - Display “Error: Invalid character value <character>, defaulting to -1 [getVariableValue]”
    - Return -1
  + Return the corresponding integer value for the inputted character
* Function getPostfixFromTree (recursive function)
  + Accepts a pointer to the tree root as an input parameter
  + Create an empty string variable
  + If root node is null
    - Display “Error: root node is null”
    - Return an empty string value
  + Else
    - If node left child is not null
      * Add the return of getPostfixFromTree(node left child) to the string variable
    - If node right child is not null
      * Add the return of getPostfixFromTree(node right child) to the string variable
    - Add the character value of the node to the string variable
    - Return the string variable
* Function getPrefixFromTree (recursive function)
  + Accepts a pointer to the tree root as an input parameter
  + Create an empty string variable
  + If root node is null
    - Display “Error: root node is null”
    - Return an empty string value
  + Else
    - Add the character value of the node to the string variable
    - If node left child is not null
      * Add the return of getPrefixFromTree(node left child) to the string variable
    - If node right child is not null
      * Add the return of getPrefixFromTree(node right child) to the string variable
    - Return string variable
* Function getInfixFromTree
  + Accepts a pointer to the tree root as an input parameter
  + Create an empty string variable
  + If root node is null
    - Display “Error: root node is null”
    - Return an empty string value
  + Else
    - If node left child is not null
      * Add the return of getInfixFromTree(node left child) to the string variable
    - Add the character value of the node to the string variable
    - If node right child is not null
      * Add the return of getInfixFromTree(node right child) to the string variable
    - Return string variable
* Function convertPostfixToTree
  + Accepts a postfix string as an input
  + Create a new tree node stack
  + Create a new root node for the tree
  + For each character in the postfix expression
    - Create a new tree node and set the character value to the current char in the expression
    - If character is an operand
      * Push the node to the stack
    - If character is an operator
      * Pop a value from the stack and set the left node of the current node equal to the value that was popped
      * Pop the next value from the stack and set the right node of the current node equal to the value that was popped
      * Push the current node to the stack
  + Return the root node
* struct that is for a node in the expression tree
  + pointer to left and right node
  + value that contains the character operand or operator
* Function to display the postfix operations
  + accepts an expression string in postfix notation as an input parameter
  + create a new queue object, queue1
  + create a new queue object, queue2
  + for each character in the input string
    - if character is an operand
      * en-queue the character in queue1
    - if the character is an operator
      * if queue1 is not empty
        + en-queue the character in queue1
        + create a new string variable
        + while the queue1 is not empty

dequeue from queue1 and add it to the string variable

* + - * + display the value of the string variable as an operation
        + en-queue the operation that is in the string variable to queue2
      * if queue1 is empty and queue2 is not empty
        + en-queue the character to queue2
        + create a new string variable
        + while queue2 is not empty

dequeue from queue 2 and add it to the string variable

* + - * + display the value of the string variable as an operation
        + en-queue the string value of the operation to queue2
      * if queue1 is empty and queue2 is empty
        + display an error that there are no operands for the operation to be performed on, and display the character.
        + return from the function, do not continue processing the inputted postfix expression
* class for a string queue (used for determining the operation steps of expressions)
  + variable that points to the first element in the queue
  + struct object for the queue
    - string variable for the operation
    - variable that points to the next value in the queue
  + function to en-queue a string
    - accepts a string input paramter
    - if the first element is empty
      * create a new struct object and set the string variable equal to the string input
      * set the first element equal to the new struct object
    - if the first element is not empty
      * create a new struct object and set the string variable equal to the string input
      * variable that points to a struct object, set equal to the first element, called current
      * while current is not null
        + set current equal to current next
      * set current equal to the new struct object that contains the input string
  + function to de-queue the first element from the queue
    - if the queue is not empty
      * create a new struct object that points to the next object of first element
      * create a string variable that is equal to the string in the first element
      * set the first element variable equal to the new struct object
      * unallocate the memory for the new struct object and delete the object
      * return the string variable
    - if the queue is empty
      * display “Error: cannot dequeue from an empty queue.”
      * return an empty string
  + function to check if the queue is empty
    - if the first element is null, then return true
    - if the first element is not null, then return false
* Function to convert infix to postfix
  + accepts an expression string as an input parameter
  + create an empty variable for the postfix expression string
  + create an empty character stack
  + variable for the index of the previous character, operands and operators
  + variable for the parentheses count
  + Boolean variable for errors, set to false
  + for each character index
    - if character is a space
      * Change index to the next character
      * If previous character is an operand and current character is an operand
        + display an error that there are two operands in a row in the expression
        + set error variable to true
      * if previous character is an operator and current character is an operator or “)”
        + display an error that there are invalid operands for the operator
        + set error variable to true
      * if previous character is an “)” and the current character is an “(“
        + display an error that there is no operator for the operands
        + set error variable to true
      * if character index is a space
        + display an error that there are two spaces in a row \*\*\*
        + set error variable to true
      * if character index is out of bounds
        + if the postfix string variable is empty

display an error that the expression on the current line is empty

set error variable to true

* + - * + if the postfix string variable is not empty

the string is done being parsed

* + - * If character index is an operand (call checkOperand function)
        + Add the character to the end of the postfix string variable
      * If character index is an operator (call the checkOperator function)
        + While the stack is not empty and top of stack is not “(“ and top of stack is high precedence than the operator

Pop the top of the stack and add it to the postfix sting variable

* + - * + Push the current operator character to the stack
      * If the character index is a “(“
        + push it to the character stack
        + increase parentheses count by 1
      * if the character index is a “)”
        + while the stack is not empty and the top of the stack is not “(“

pop the top character from the stack and add it to the postfix string variable

* + - * + pop the top character from the stack to discard it, it should be a “(“
        + decrease the parentheses count by 1
      * increase the character index by 1
  + While the stack is not empty
    - Pop the rest of the values from the stack and add them to the string variable
  + if parentheses count is not equal to 0
    - display an error that there are mismatched parentheses
    - set error variable to true
  + if error variable is false
    - return postfix
  + if error variable is true
    - return an empty string
* Function to check if a character is an uppercase operand
  + accepts a single character parameter
  + if the input character is >= ‘A’ and <= ‘Z’, return true
  + otherwise return false
* Function to check if a character is an operator
  + accepts a single character parameter
  + if input is a +, -, /, \*
    - return true
  + otherwise return false
* Function to get the weight of the operators
  + accepts a single character parameter
  + if character is ‘+’ or ‘-’
    - return 1
  + if character is ‘\*’ or ‘/’
    - return 2
  + else
    - display “Error: Invalid operator input [getWeight]”
    - return -1
* Template class for the stack (to allow for different variable types))
  + private variable that points to the top object of the stack
  + create a structure within the class that contains
    - variable of the template type
    - variable that points to the next object in the stack
  + function push to the stack
    - accepts a value of template type as an input parameter
    - if stack is empty
      * create a new struct object and set the variable value equal to the input parameter
      * set the private top variable equal to the newly created struct object
    - if the stack is not empty
      * create a new struct object and set the variable value equal to the input parameter
      * set the next variable in the new object equal to the private top object
      * set the top object equal to the new object
  + function to check if stack is empty
    - if private top variable is a valid value that points to a struct object, return false
  + Function to pop the top value off the stack
    - if the stack is not empty
      * create a new struct variable and set it equal to the top variable
      * set the top variable equal to next value of top variable
      * unallocate memory for the new struct variable
  + Function to return the value of the top of the stack
    - if stack is not empty
      * return the variable value of the top of the stack
    - if stack is empty, return a null value
* Class called logging (used to log data to the output file and to the screen
  + Private variables:
    - String filename
    - Variable for the file stream
    - Boolean for file open
  + Get/set functions for private variables
  + Function openFile
    - String variable for the file name as an input parameter
    - If file is not open
      * Attempt to open/create the specified file
      * File stream variable equal to the stream of the file that was opened
    - Else
      * Display “Output file is already open: <filename>”
  + Function closeFile
    - If the file is open
      * Close the file stream that corresponds to the file stream variable
    - If the file is closed
      * Display “File is already closed: “filename”
  + Function display
    - Accepts a string to display as an input parameter
    - Print the input string to the console screen on a new line
    - If the file is open
      * Write the input string to a new line in the file

1. **Test Plan Version 2**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Test Strategy | Test Number | Description | Input | Expected Output | Actual Output | Pass/Fail |
| Valid | 1.1 | Test expression with 1 operator and no parentheses or operands | A | Valid Statement  Postfix: A  Operations: A  Expression Tree: A  Prefix: A  Infix: A  Postfix: A  Final result: 1 |  |  |
| Valid | 1.2 | Test expression with 2 values, + operator, no parentheses | A + B | Valid Statement  Postfix: AB+  Operations: AB+  Expression Tree:  +  A B  Prefix: +AB  Infix: A+B  Postfix: AB+  Final result: 3 |  |  |
| Valid | 1.3 | Test expression with 2 values, - operator, no parentheses | A – B | Valid Statement  Postfix: AB-  Operations: AB-  Expression Tree:  -  A B  Prefix: -AB  Infix: A-B  Postfix: AB-  Final result: -1 |  |  |
| Valid | 1.4 | Test expression with 2 values, \* operator, no parentheses | A \* B | Valid Statement  Postfix: AB\*  Operations: AB\*  Expression Tree:  \*  A B  Prefix: \*AB  Infix: A\*B  Postfix: AB\*  Final result: 2 |  |  |
| Valid | 1.5 | Test expression with 2 values, / operator, no parentheses | D / B | Valid Statement  Postfix: DB/  Operations: DB/  Expression Tree:  /  D B  Prefix: /DB  Infix: D/B  Postfix: DB/  Final result: 2 |  |  |
| Valid | 1.6 | Test expression with 2 values, 1 operator, all enclosed in parentheses |  |  |  |  |
| Valid | 1.7 | Test expression with 3 values, operation with 2 values enclosed in parentheses, first operation added to 3rd operator | ( A \* B ) + C | Valid Statement  Postfix: AB\*C+  Operations: AB\*  AB\*C+  Expression Tree:  +  \* C  A B  Prefix: +\*ABC  Infix: A\*B+C  Postfix: AB\*C+  Final result: 5 |  |  |
| Valid | 1.8 | Test expression with 3 values, operation with 2 values enclosed in parentheses, first operation divided by 3rd operator | ( C \* B ) / C | Valid Statement  Postfix: CB\*C/  Operations: CB\*  CB\*C/  Expression Tree:  /  \* C  C B  Prefix: /\*CBC  Infix: C\*B+C  Postfix: CB\*C+  Final result: 2 |  |  |
| Valid | 1.9 | Test expression with 3 values, operation with 2 values enclosed in parentheses, first operation multiplied by 3rd operator | ( A \* B ) \* C | Valid Statement  Postfix: AB\*C\*  Operations: AB\*  AB\*C\*  Expression Tree:  \*  \* C  A B  Prefix: +\*ABC  Infix: A\*B\*C  Postfix: AB\*C\*  Final result: 6 |  |  |
| Valid | 1.10 | Test expression with 3 values, operation with 2 values enclosed in parentheses, first operation subtracted from 3rd operator | ( A \* B ) - C | Valid Statement  Postfix: AB\*C-  Operations: AB\*  AB\*C-  Expression Tree:  -  \* C  A B  Prefix: -\*ABC  Infix: A\*B-C  Postfix: AB\*C-  Final result: -1 |  |  |
| Valid | 1.11 | Test expression with a combination of operations inside and outside of the parentheses | A + ( A \* B ) / B | Valid Statement  Postfix: AB\*C/A+  Operations: AB\*  AB\*C/  AB\*C/A+  Expression Tree:  +  / A  \* B  A B  Prefix: +\*ABC  Infix: A\*B/B+A  Postfix: AB\*C/A+  Final result: 2 |  |  |
| Valid | 1.12 | Test order of operations at the same level: +, \*, / | B + A \* C / B | Valid Statement  Postfix: AB\*C/B+  Operations: AB\*  AB\*C/  AB\*C/B+  Expression Tree:  +  / B  \* B  A C  Prefix: +/\*ACBB  Infix: A\*C/B+B  Postfix: AB\*C/B+  Final result: 3 |  |  |
| Valid | 1.13 | Test order of operations at the same level: +, \*, /, - | B + A \* C / B - D | Valid Statement  Postfix: AB\*C/B+D-  Operations: AB\*  AB\*C/  AB\*C/B+  AB\*C/B+D-  Expression Tree:  -  + D  / B  \* B  A C  Prefix: -+/\*ACBBD  Infix: A\*C/B+B  Postfix: AB\*C/BD+-  Final result: -2 |  |  |
| Valid | 1.14 | Test order of operations at the same level: \*, +, /, - | B \* A + C / B - D | Valid Statement  Postfix: BA\*CB/+D-  Operations: BA\*  CB/  BA\*CB/+  BA\*CB/+D-  Expression Tree:  -  + D  \* /  B A C B  Prefix: -+\*BA/CBD  Infix: B\*A+C/B-D  Postfix: BA\*CB/+D-  Final result: -1 |  |  |
| Valid | 1.16 | Test order of operations between operations separated by parentheses (-)\*(+) | ( B – A ) \* ( C + D ) | Valid Statement  Postfix: BA-CD+\*  Operations: BA-  CD+  BA-CD+\*  Expression Tree:  \*  - +  B A C D  Prefix: \*-BA+CD  Infix: B-A\*C+D  Postfix: BA-CD+\*  Final result: 7 |  |  |
| Valid | 1.17 | Test order of operations between operations separated by parentheses (\*)-(/) | ( B \* A ) - ( D / B ) | Valid Statement  Postfix: BA\*DB/-  Operations: BA\*  DB/  BA\*DB/+  Expression Tree:  +  \* /  B A D B  Prefix: +\*BA/DB  Infix: B\*A+D/B  Postfix: BA\*DB/+  Final result: 0 |  |  |
| Valid | 1.18 | Test order of operations between operations separated by parentheses (\*)/(+) | ( B \* A ) / ( D - B ) | Valid Statement  Postfix: BA\*DB-/  Operations: BA\*  DB-  BA\*DB-/  Expression Tree:  /  \* -  B A D B  Prefix: /\*BA-DB  Infix: B\*A/D-B  Postfix: BA\*DB-/  Final result: 1 |  |  |
| Invalid | 2.1 | Test an expression with 2 operands, 1 operator, and only an opening parenthesis at the begging and no closing parenthesis | ( A + B | Invalid statement  Error: Mismatched parentheses |  |  |
| Invalid | 2.2 | Test an expression with 2 operands, 1 operator, and only a closing parenthesis at the end | A + B ) | Invalid Statement  Error: Mismatched parentheses |  |  |
| Invalid | 2.3 | Test an expression with 2 operands and no operators | B C | Invalid Statement  Error: No valid operator |  |  |
| Invalid | 2.4 | Test an expression with 1 operand and 1 operator | A + | Invalid Statement  Error: Invalid number of operands |  |  |
| Invalid | 2.5 | Test an expression with no operands and 1 operator | + | Invalid Statement  Error: Invalid number of operands |  |  |
| Invalid | 2.6 | Test an expression with only a set of parentheses | ( ) | Invalid Statement  Error: No valid operation inside parentheses |  |  |
| Invalid | 2.7 | Test an expression with two operations and a missing closing parenthesis | ( A + B ) – ( C \* D | Invalid Statement  Error: Mismatched parentheses |  |  |
| Invalid | 2.8 | Test an expression with two operations and a missing opening parenthesis | ( A + B ) – C \* D ) | Invalid Statement  Error: Mismatched parentheses |  |  |
| Invalid | 2.9 | Test an expression with two operations enclosed in parentheses and no operator between them | ( A + B ) ( C – D ) | Invalid Statement  Error: Missing Operator |  |  |
| Invalid | 2.10 | Test the function checkOperator by inputting a non-operator value | A | Function returns false |  |  |
| Invalid | 2.11 | Test the function getWeight by inputting a non-operator value | B | Function returns -1 and displays an error: “Error: Invalid operator input [getWeight]” |  |  |
| Invalid | 2.12 | Test the function getCharacterValue by inputting a character outside of the range of valid character operands | ] | Function returns -1 and displays an error: “Error: Invalid character value <character>, defaulting to -1 [getVariableValue]” |  |  |
| Invalid | 2.13 | Attempt to process an operation that will divide by zero | A / ( A – A ) | Valid Statement  Postfix: AA-A/  Operations: AA-  AA-A/  Expression Tree:  /   * A   A A  Prefix: /-AAA  Infix: A-A/A  Postfix: AA-A/  “ERROR: Cannot solve, division by zero: <integer1> / <integer2>”  Final result: error |  |  |
| Invalid | 2.14 | Pass a null pointer to a root node as an input parameter for the function displayTree | Null pointer | “Error: Cannot display tree, null root node” |  |  |
| Invalid | 2.15 | Attempt to dequeue an element from an empty queue | N/A | Dequeue function returns an empty value.  Display an error: “Error: cannot dequeue from an empty queue.” |  |  |
| Invalid | 2.16 | Pass a null pointer to a tree root node as a parameter to the getPostfixFromTree function | Null pointer | Returns an empty string and displays an error: “Error: Root node is empty” |  |  |
| Invalid | 2.17 | Pass a null pointer to a tree root node as a parameter to the getPrefixFromTree function | Null pointer | Returns an empty string and displays an error: “Error: Root node is empty” |  |  |
| Invalid | 2.18 | Pass a null pointer to a tree root node as a parameter to the getInfixFromTree function | Null pointer | Returns an empty string and displays an error: “Error: Root node is empty” |  |  |
| Invalid | 2.19 | Pop a value off of an empty stack | N/A | Returns a null value |  |  |
| Invalid | 2.20 | Call the top function on an empty stack | N/A | Top returns a null value |  |  |
| File Handling | 3.1 | Input a file name that does not exist | Somefile.dat | Error: Input file does not exist |  |  |
| File Handling | 3.2 | Input a file with a type other than “.dat” | Somefile.txt | Error: Invalid input file type |  |  |
| File Handling | 3.3 | Input a file of type “.dat” that exists | Input.dat | Open file successfully |  |  |
| File Handling | 3.4 | Input a file of type “.dat” that exists, but is empty | Input.dat | Error: empty input file |  |  |
| File Handling | 3.5 | Attempt to open an output file that is opened by another program | output.dat | Should still open the file and take control of writing to it |  |  |

1. **Code**

//Program Name: Expression processing

//Programmer Name: Jason Hogan

//Description: Takes an infix equation as input, the processes it into an expression tree and prints the traversals in prefix, infix, and postfix.

//Date Created: 2/7/2019

#include <string>

#include <cstring>

#include <iostream>

#include <fstream>

#include <iterator>

#include <stdio.h>

#define FILENAME "output.dat"

using namespace std;

struct treeNode{

treeNode\* left = NULL;

treeNode\* right = NULL;

char value;

};

// Description: Stack data structure that is used to handle any type of datatypes

template <typename T>

class tStack

{

private:

class node {

public:

T value;

node\* next = NULL;

};

node\* topNode = NULL;

public:

tStack()

{

}

~tStack()

{

}

// Description: Checks if the stack is empty

bool isEmpty() {

if (topNode == NULL) {

return true;

}

else {

return false;

}

}

// Description: Add a value to the top of the stack

void push(T input) {

if (isEmpty()) {

node\* newNode = new node();

newNode->value = input;

topNode = newNode;

}

else {

node\* newNode = new node();

newNode->value = input;

newNode->next = topNode;

topNode = newNode;

}

}

// description: Removes value from the stack and returns it

T pop() {

T value = topNode->value;

if (!isEmpty()) {

node\* temp = topNode;

topNode = topNode->next;

delete(temp);

}

return value;

}

// Description: Returns the top value on the stack

T top() {

return topNode->value;

}

};

// Description: Logs the inputted string to the output file and to the console screen

void log(const std::string &input){

ofstream filestream(FILENAME, std::ios\_base::app|std::ios\_base::out);

filestream.flush();

cout.flush();

std::cout << input << endl;

filestream << input << endl;

filestream.close();

}

// Description: Checks if the inputed character is a valid operator

bool isOperator(char input){

if (input == '+' || input == '-' || input == '/' || input == '\*'){

return true;

}

else{

return false;

}

}

// Description: Checks if the inputed character is a valid operand

bool isOperand(char input){

if(input >= 'A' && input <= 'Z'){

return true;

}

else{

return false;

}

}

// Description: Returns the weighted precidence of the inputted character operator

int getOperatorWeight(char input){

if (input == '+' || input == '-'){

return 1;

}

else if (input == '\*' || input == '/'){

return 2;

}

else{

log("[ERROR][getOperatorWeight] Cannot get weight, invalid operator: " + input);

return -1;

}

}

// Description: Converts a postfix string into a tree structure \*\*Assumes the input is already in proper postfix notation\*\*

treeNode\* convertPostfixToTree(string postfix) {

tStack<treeNode\*> treeStack;

treeNode\* root = new treeNode();

for (int i = 0; i < postfix.size(); i++) {

treeNode\* node = new treeNode();

node->value = postfix[i];

// check if operand

if (isOperand(postfix[i])) {

treeStack.push(node);

}

// check if operator

if (isOperator(postfix[i])) {

treeNode\* temp = treeStack.pop();

node->right = temp;

temp = treeStack.pop();

node->left = temp;

treeStack.push(node);

}

}

root = treeStack.pop();

return root;

}

// Description: Converts the infix expression into a postfix expression and returns a string for the postfix. Using stacks.

string infixToPostfix(string infix){

std::string postfix = "";

tStack<char> stack;

int parCount = 0;

int prevIndex = 0;

bool errors = false;

for (int i = 0; i <= infix.size(); i++){

// checks if the character value is a space

if(infix[i] == ' '){

i += 1;

if (i > infix.length() - 1 ){

if (isOperator(infix[prevIndex])) {

errors = true;

log("[ERROR] Missing second operand at indexes: " + to\_string(prevIndex) + ',' + to\_string(i));

}

else if (postfix.size() == 0){

errors = true;

log("[ERROR] Empty line!");

}

else {

// all done

break;

}

}

else if (isOperand(infix[prevIndex]) && isOperand(infix[i])){

errors = true;

log("[ERROR] Two operands cannot be next to each other! At indexes: " + to\_string(prevIndex) + ',' + to\_string(i));

}

else if (isOperator(infix[prevIndex]) && infix[i] == ')'){

errors = true;

log("[ERROR] Missing operand! At indexes: " + to\_string(prevIndex) + ',' + to\_string(i));

}

else if (infix[prevIndex] == ')' && infix[i] == '('){

errors = true;

log("[ERROR] Missing operator! At indexes: " + to\_string(prevIndex) + ',' + to\_string(i));

}

else if (infix[i] == ' '){

errors = true;

log("[ERROR] Invalid spacing! At indexes: " + to\_string(prevIndex) + ',' + to\_string(i));

}

else if (isOperand(infix[i])){

postfix += infix[i];

}

else if (isOperator(infix[i])){

while(stack.isEmpty() == false && stack.top() != '(' && getOperatorWeight(stack.top()) >= getOperatorWeight(infix[i])){

postfix += stack.pop();

}

stack.push(infix[i]);

}

else if(infix[i] == '('){

stack.push(infix[i]);

parCount += 1;

}

else if (infix[i] == ')'){

while(stack.isEmpty() == false && stack.top() != '('){

postfix += stack.pop();

}

if (stack.isEmpty() == false){

stack.pop();

}

parCount -= 1;

}

}

else if (infix[i] != ' '){

errors = true;

log("[ERROR] Invalid spacing! At indexes: " + to\_string(prevIndex) + ',' + to\_string(i));

}

prevIndex = i;

}

while (stack.isEmpty() == false){

postfix +=stack.pop();

}

if(parCount != 0){

errors = true;

log("[ERROR] Mismatched Parentheses!");

}

if(errors) {

return "";

}

else{

return postfix;

}

}

// Description: Returns a string with properly formatted operations that the postfix expression will perform according to the order of operations

string getPostfixOperations(string postfix) {

string operations;

string operation1;

string operation2;

tStack<string> stack;

for (int i = 0; i < postfix.size(); i++) {

if (isOperand(postfix[i])) {

stack.push(string(1, postfix[i]));

}

else if (isOperator(postfix[i])) {

operation1 = stack.pop();

operation2 = stack.pop();

string temp = operation2 + operation1 + postfix[i];

stack.push(temp);

operations += temp + "\n";

}

}

return operations;

}

// Description: Returns a string that represents a tree structure in a horizontal format with the root node all the way to the left.

string visualizeTree(treeNode\* root, int spacing) {

string temp;

if (root != NULL) {

spacing += 5;

temp += visualizeTree(root->right, spacing);

temp += "\n";

for (int i = 0; i < spacing; i++) {

temp += " ";

}

temp += string(1,root->value);

temp += visualizeTree(root->left, spacing);

}

return temp;

}

// Description: Traverses the tree and returns it in postfix notation

string getPostfixFromTree(treeNode\* root) {

string postfix;

if (root == NULL) {

return "";

}

else {

if (root->left != NULL) {

postfix += getPostfixFromTree(root->left);

}

if (root->right != NULL) {

postfix += getPostfixFromTree(root->right);

}

postfix += string(1,root->value);

}

return postfix;

}

// Description: Traverses the tree and returns it in prefix notation

string getPrefixFromTree(treeNode\* root) {

string prefix;

if (root == NULL) {

return "";

}

else {

prefix += string(1, root->value);

if (root->left != NULL) {

prefix += getPrefixFromTree(root->left);

}

if (root->right != NULL) {

prefix += getPrefixFromTree(root->right);

}

}

return prefix;

}

// Description: Traverses the tree and returns it in prefix notation

string getInfixFromTree(treeNode\* root) {

string infix;

if (root == NULL) {

return "";

}

else {

if (root->left != NULL) {

infix += getInfixFromTree(root->left);

}

infix += root->value;

if (root->right != NULL) {

infix += getInfixFromTree(root->right);

}

}

return infix;

}

//Description: Returns integer value of the character

int getVariableValue(char input) {

if (input >= 'A' && input <= 'Z') {

return input - 64;

}

else {

log("[ERROR][getVariableValue] Invalid Character, defaulting to -1: " + string(1, input));

return -1;

}

}

// Description: Prints the operations of a prefix expression and displays the solution of it

void solutionFromPrefix(string prefix) {

tStack<string> opStack;

tStack<int> intStack;

int curVal = 0;

for (int i = prefix.size() - 1; i >= 0; i--) {

if (isOperand(prefix[i])) {

curVal = getVariableValue(prefix[i]);

opStack.push(string(1,prefix[i]));

intStack.push(getVariableValue(prefix[i]));

}

else {

if (opStack.isEmpty() == false) {

string temp = string(1,prefix[i]);

int tmpInt;

string str1, str2;

str1 = opStack.pop();

temp += str1;

str2 = opStack.pop();

temp += str2;

opStack.push(temp);

// handle operations

if (prefix[i] == '+') {

int int1 = intStack.pop();

int int2 = intStack.pop();

int tmp = int1 + int2;

intStack.push(tmp);

}

else if (prefix[i] == '-') {

int int1 = intStack.pop();

int int2 = intStack.pop();

int tmp = int1 - int2;

intStack.push(tmp);

}

else if (prefix[i] == '\*') {

int int1 = intStack.pop();

int int2 = intStack.pop();

int tmp = int1 \* int2;

intStack.push(tmp);

}

else if (prefix[i] == '/') {

int int1 = intStack.pop();

int int2 = intStack.pop();

if (int2 != 0) {

int tmp = int1 / int2;

intStack.push(tmp);

}

else {

log("[ERROR] Cannot divide by zero!");

return;

}

}

curVal = intStack.top();

log(temp + " = " + to\_string(intStack.top()));

}

}

}

log("Final Solution: " + to\_string(curVal));

}

// Description: The main method of the application, Runs on startup.

int main() {

log("\nWelcome to the best program ever!");

log("Please enter an input file name: ");

string inputFileName;

getline(cin, inputFileName);

ifstream inFile(inputFileName);

if (!inFile) {

log(inputFileName + " Does not exist!");

system("PAUSE");

return 1;

}

else if (inFile.peek() == ifstream::traits\_type::eof()) {

log(inputFileName + " File is empty!");

system("PAUSE");

return 1;

}

log(inputFileName + " File was opened successfully!");

string line;

while (getline(inFile, line)) {

log("===============================================");

log("Input: " + line);

string postfix;

postfix = infixToPostfix(line);

if (postfix != "") {

log("\nPostfix: " + postfix);

string operations = getPostfixOperations(postfix);

log("\nOperations: " + operations);

treeNode\* root = convertPostfixToTree(postfix);

string visTree = visualizeTree(root, 1);

log("\nTree: \n" + visTree);

string postfixTree = getPostfixFromTree(root);

string infixTree = getInfixFromTree(root);

string prefixTree = getPrefixFromTree(root);

log("\nPostfix from tree: " + postfixTree);

log("Prefix from tree: " + prefixTree);

log("Infix from tree: " + infixTree);

solutionFromPrefix(prefixTree);

}

}

log("Thank you for using the program! Have a great day!");

system("PAUSE");

return 1;

}

1. **Updated Algorithm**

* \*\*\*NOTE: A call to the display function of the Logging class whenever displaying values to the screen. \*\*\*
* Main Method (called on program startup)
  + ~~Log variable of type Logging~~
  + ~~Call the openFile function of Log~~
    - ~~Pass “output.dat” as a parameter~~
  + ~~If log getFileOpen is true~~
    - ~~Display “Output file opened successfully: output.dat”~~
  + ~~Else~~ 
    - ~~Display “Output file cannot be opened: output.dat”~~
  + ~~Display a welcome message to the user~~
  + Log “Welcome to the program!”
  + Ask the user to enter the location of the file they want to run through the program. (ex: [c://users//](../../../../c://users//)…//filename.dat)
  + If the file does not exist
    - Display an error message that it doesn’t exist and ask the user to enter another file name
  + If the file is empty
    - Display an error message that the file is empty and ask the user to enter another file name
  + If the file is of the wrong type (not “.dat”)
    - Display an error message that the file is of the wrong type and ask the user to enter another file name
  + Open the file and print a message that the file was opened successfully
  + For each line in the input file
    - read the line in and print it to the screen
    - string variable for the postfix expression
    - pass the expression to the function that converts infix to postfix
      * set the postfix variable equal to the return of this function
    - if the postfix variable is empty (an error must have occurred)
      * move to the next line in the file
    - if the postfix vairable is not empty (means it was converted successfully)
      * call the function displayPostfixOperations and pass the postfix string variable as a parameter
      * create a new node for the root of the expression tree called root
      * set the root node equal to the return of the convertPostfixToTree function
      * Call the displayTree function and pass the root node in as a parameter
      * string variable for postfix, set equal to the return of getPostfixFromTree (root node)
      * display the postfix string
      * string variable for prefix, set equal to the return of getPrefixFromTree (root node)
      * display the prefix string
      * string variable for infix, set equal to the return of getInfixFromTree (root node)
      * display the infix string
      * string solution variable equals the return of solutionFromPrefix (prefix string)
      * display solution variable “Final Result: <solution>”
  + Display “Thank you for using the program! Goodbye!”
* Function getTreeHeight
  + Accepts the root of a tree as an input parameter
  + Integer for height of tree
  + Integer for leftH and rightH
  + if root left child is not null
    - set leftH to getTreeHeight (left child)
  + if root right child is not null
    - set rightH to getTreeHeight (right child)
  + if rightH > leftH
    - set height to rightH
  + else set height to leftH
  + return height
* Function displayTree
  + Accepts the root of a tree as an input parameter and integer for the spacing
  + If the root is not null
    - Increase spacing by 5
    - Recursively call display tree on the right child of the root node
    - Display a new line
    - While index is less that spacing
      * Display a space
      * Increase index by 1
    - Display the character in the root and add a new line
    - Recursively call display tree on the left child of the root node
* Function solutionFromPrefix
  + Accepts an expression string in prefix notation
  + Create an integer stack
  + Create string stack
  + For each character in the input string going in reverse order
    - If character is an operand
      * Integer value equal to the return of getVariableValue (operand character)
      * Push integer value to the stack
      * Push the current character to the string stack
    - Else if string stack is not empty
      * Temp string equal to current char value
      * String1 equal to return of string stack pop
      * String 2 equal to return of string stack pop
      * Add string1 and string2 to the temp string
      * Push temp string to the string stack
      * Integer1 equal to first pop off the stack
      * Integer2 equal to second pop off the stack
      * If character = “+”
        + Push integer1+interger2 to the int stack
      * If character = “-“
        + Push integer1-integer2 to the int stack
      * If character = “\*”
        + Push integer1\*integer2 to the int stack
      * If character = “/”
        + If integer2 is not 0

Push integer1/integer2 to the int stack

* + - * + Else

Display “ERROR: Cannot solve, division by zero: <integer1> / <integer2>”

Return “error”

* + - * Log “<temp string> = <int stack top>”
  + Return the top of the stack as a string variable
* Function getVariableValue
  + Accepts a character as an input parameter
  + A=1, B=2, C=3, D=4, …, I=9
  + If input is not A through Z
    - Display “Error: Invalid character value <character>, defaulting to -1 [getVariableValue]”
    - Return -1
  + Return the corresponding integer value for the inputted character
* Function getPostfixFromTree (recursive function)
  + Accepts a pointer to the tree root as an input parameter
  + Create an empty string variable
  + If root node is null
    - Display “Error: root node is null”
    - Return an empty string value
  + Else
    - If node left child is not null
      * Add the return of getPostfixFromTree(node left child) to the string variable
    - If node right child is not null
      * Add the return of getPostfixFromTree(node right child) to the string variable
    - Add the character value of the node to the string variable
    - Return the string variable
* Function getPrefixFromTree (recursive function)
  + Accepts a pointer to the tree root as an input parameter
  + Create an empty string variable
  + If root node is null
    - Display “Error: root node is null”
    - Return an empty string value
  + Else
    - Add the character value of the node to the string variable
    - If node left child is not null
      * Add the return of getPrefixFromTree(node left child) to the string variable
    - If node right child is not null
      * Add the return of getPrefixFromTree(node right child) to the string variable
    - Return string variable
* Function getInfixFromTree
  + Accepts a pointer to the tree root as an input parameter
  + Create an empty string variable
  + If root node is null
    - Display “Error: root node is null”
    - Return an empty string value
  + Else
    - If node left child is not null
      * Add the return of getInfixFromTree(node left child) to the string variable
    - Add the character value of the node to the string variable
    - If node right child is not null
      * Add the return of getInfixFromTree(node right child) to the string variable
    - Return string variable
* Function convertPostfixToTree (Assumes the input is in proper postfix notation.)
  + Accepts a postfix string as an input
  + Create a new tree node stack
  + Create a new root node for the tree
  + For each character in the postfix expression
    - Create a new tree node and set the character value to the current char in the expression
    - If character is an operand
      * Push the node to the stack
    - If character is an operator
      * Pop a value from the stack and set the left node of the current node equal to the value that was popped
      * Pop the next value from the stack and set the right node of the current node equal to the value that was popped
      * Push the current node to the stack
  + Pop the top node off the stack and set the root node equal to this. (There should only be one node on the stack)
  + Return the root node
* struct that is for a node in the expression tree
  + pointer to left and right node
  + value that contains the character operand or operator
* Function getPostfixOperations
  + accepts an expression string in postfix notation as an input parameter
  + string for postfixOperations
  + create a new queue object, queue1
  + create a new queue object, queue2
  + for each character in the input string
    - if character is an operand
      * en-queue the character in queue1
    - if the character is an operator
      * if queue1 is not empty
        + en-queue the character in queue1
        + create a temp string variable
        + while the queue1 is not empty

dequeue from queue1 and add it to the string variable

* + - * + add a new line to the operations string, then add the temp string to the end of this string
        + en-queue the operation that is in the string variable to queue2
      * if queue1 is empty and queue2 is not empty
        + en-queue the character to queue2
        + create a temp string variable
        + while queue2 is not empty

dequeue from queue 2 and add it to the temp string variable

* + - * + ~~display the value of the string variable as an operation~~
        + add a new line to the operations string, then add the temp string to the end of this string
        + en-queue the string value of the operation to queue2
      * if queue1 is empty and queue2 is empty
        + display an error that there are no operands for the operation to be performed on, and display the character.
        + return from the function, do not continue processing the inputted postfix expression
    - return operations string
* ~~class for a string queue (used for determining the operation steps of expressions)~~
  + ~~variable that points to the first element in the queue~~
  + ~~struct object for the queue~~
    - ~~string variable for the operation~~
    - ~~variable that points to the next value in the queue~~
  + ~~function to en-queue a string~~
    - ~~accepts a string input paramter~~
    - ~~if the first element is empty~~
      * ~~create a new struct object and set the string variable equal to the string input~~
      * ~~set the first element equal to the new struct object~~
    - ~~if the first element is not empty~~
      * ~~create a new struct object and set the string variable equal to the string input~~
      * ~~variable that points to a struct object, set equal to the first element, called current~~
      * ~~while current is not null~~
        + ~~set current equal to current next~~
      * ~~set current equal to the new struct object that contains the input string~~
  + ~~function to de-queue the first element from the queue~~
    - ~~if the queue is not empty~~
      * ~~create a new struct object that points to the next object of first element~~
      * ~~create a string variable that is equal to the string in the first element~~
      * ~~set the first element variable equal to the new struct object~~
      * ~~unallocate the memory for the new struct object and delete the object~~
      * ~~return the string variable~~
    - ~~if the queue is empty~~
      * ~~display “Error: cannot dequeue from an empty queue.”~~
      * ~~return an empty string~~
  + ~~function to check if the queue is empty~~
    - ~~if the first element is null, then return true~~
    - ~~if the first element is not null, then return false~~
* Function to convert infix to postfix
  + accepts an expression string as an input parameter
  + create an empty variable for the postfix expression string
  + create an empty character stack
  + variable for the index of the previous character, operands and operators
  + variable for the parentheses count
  + Boolean variable for errors, set to false
  + for each character index
    - if character is a space
      * Change index to the next character
      * If previous character is an operand and current character is an operand
        + display an error that there are two operands in a row in the expression
        + set error variable to true
      * if previous character is an operator and current character is an operator or “)”
        + display an error that there are invalid operands for the operator
        + set error variable to true
      * if previous character is an “)” and the current character is an “(“
        + display an error that there is no operator for the operands
        + set error variable to true
      * if character index is a space
        + display an error that there are two spaces in a row \*\*\*
        + set error variable to true
      * if character index is out of bounds
        + if the postfix string variable is empty

display an error that the expression on the current line is empty

set error variable to true

* + - * + if the postfix string variable is not empty

the string is done being parsed

* + - * If character index is an operand (call checkOperand function)
        + Add the character to the end of the postfix string variable
      * If character index is an operator (call the checkOperator function)
        + While the stack is not empty and top of stack is not “(“ and top of stack is high precedence than the operator

Pop the top of the stack and add it to the postfix sting variable

* + - * + Push the current operator character to the stack
      * If the character index is a “(“
        + push it to the character stack
        + increase parentheses count by 1
      * if the character index is a “)”
        + while the stack is not empty and the top of the stack is not “(“

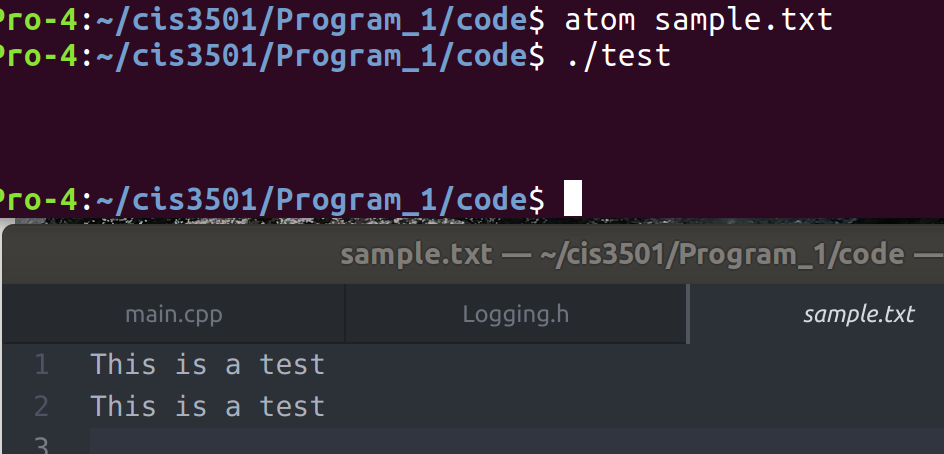
pop the top character from the stack and add it to the postfix string variable

* + - * + pop the top character from the stack to discard it, it should be a “(“
        + decrease the parentheses count by 1
      * ~~increase the character index by 1~~
    - if character at current index is not a space
      * set errors to true
      * log “[ERROR] Invalid spacing! At indexes: index1, index2”
  + While the stack is not empty
    - Pop the rest of the values from the stack and add them to the string variable
  + if parentheses count is not equal to 0
    - display an error that there are mismatched parentheses
    - set error variable to true
  + if error variable is false
    - return postfix
  + if error variable is true
    - return an empty string
* Function to check if a character is an uppercase operand
  + accepts a single character parameter
  + if the input character is >= ‘A’ and <= ‘Z’, return true
  + otherwise return false
* Function to check if a character is an operator
  + accepts a single character parameter
  + if input is a +, -, /, \*
    - return true
  + otherwise return false
* Function to get the weight of the operators
  + accepts a single character parameter
  + if character is ‘+’ or ‘-’
    - return 1
  + if character is ‘\*’ or ‘/’
    - return 2
  + else
    - display “Error: Invalid operator input [getWeight]”
    - return -1
* Template class for the stack (to allow for different variable types))
  + private variable that points to the top object of the stack
  + create a structure within the class that contains
    - variable of the template type
    - variable that points to the next object in the stack
  + function push to the stack
    - accepts a value of template type as an input parameter
    - if stack is empty
      * create a new struct object and set the variable value equal to the input parameter
      * set the private top variable equal to the newly created struct object
    - if the stack is not empty
      * create a new struct object and set the variable value equal to the input parameter
      * set the next variable in the new object equal to the private top object
      * set the top object equal to the new object
  + function to check if stack is empty
    - if private top variable is a valid value that points to a struct object, return false
  + Function to pop the top value off the stack
    - if the stack is not empty
      * create a new struct variable and set it equal to the top variable
      * set the top variable equal to next value of top variable
      * unallocate memory for the new struct variable
  + Function to return the value of the top of the stack
    - if stack is not empty
      * return the variable value of the top of the stack
    - if stack is empty, return a null value
* Function called log
  + Accepts a reference to a string value as input
  + Opens the file defined in compilation in append mode
  + Writes the input string to the console
  + Writes the input string to the file stream
  + Closes the file stream
* ~~Class called logging (used to log data to the output file and to the screen~~
  + ~~Private variables:~~
    - ~~String filename~~
    - ~~Variable for the file stream~~
    - ~~Boolean for file open~~
  + ~~Get/set functions for private variables~~
  + ~~Function openFile~~
    - ~~String variable for the file name as an input parameter~~
    - ~~If file is not open~~
      * ~~Attempt to open/create the specified file~~
      * ~~File stream variable equal to the stream of the file that was opened~~
    - ~~Else~~
      * ~~Display “Output file is already open: <filename>”~~
  + ~~Function closeFile~~
    - ~~If the file is open~~
      * ~~Close the file stream that corresponds to the file stream variable~~
    - ~~If the file is closed~~
      * ~~Display “File is already closed: “filename”~~
  + ~~Function display~~
    - ~~Accepts a string to display as an input parameter~~
    - ~~Print the input string to the console screen on a new line~~
    - ~~If the file is open~~
      * ~~Write the input string to a new line in the file~~

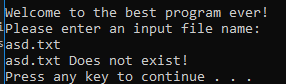
1. **Test Plan Version 3**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Test Strategy | Test Number | Description | Input | Expected Output | Actual Output | Pass/Fail |
| Valid | 1.1 | Test expression with 1 operator and no parentheses or operands | A | Valid Statement  Postfix: A  Operations: A  Expression Tree: A  Prefix: A  Infix: A  Postfix: A  Final result: 1 | Input: A  Postfix: A  Operations:  Tree:  A  Postfix from tree: A  Prefix from tree: A  Infix from tree: A  Final Solution: 1 | Pass |
| Valid | 1.2 | Test expression with 2 values, + operator, no parentheses | A + B | Valid Statement  Postfix: AB+  Operations: AB+  Expression Tree:  +  A B  Prefix: +AB  Infix: A+B  Postfix: AB+  Final result: 3 | Input: A + B  Postfix: AB+  Operations: AB+  Tree:  B  +  A  Postfix from tree: AB+  Prefix from tree: +AB  Infix from tree: A+B  +AB = 3  Final Solution: 3 | Pass |
| Valid | 1.3 | Test expression with 2 values, - operator, no parentheses | A – B | Valid Statement  Postfix: AB-  Operations: AB-  Expression Tree:  -  A B  Prefix: -AB  Infix: A-B  Postfix: AB-  Final result: -1 | Input: A - B  Postfix: AB-  Operations: AB-  Tree:  B  -  A  Postfix from tree: AB-  Prefix from tree: -AB  Infix from tree: A-B  -AB = -1  Final Solution: -1 | Pass |
| Valid | 1.4 | Test expression with 2 values, \* operator, no parentheses | A \* B | Valid Statement  Postfix: AB\*  Operations: AB\*  Expression Tree:  \*  A B  Prefix: \*AB  Infix: A\*B  Postfix: AB\*  Final result: 2 | Input: A \* B  Postfix: AB\*  Operations: AB\*  Tree:  B  \*  A  Postfix from tree: AB\*  Prefix from tree: \*AB  Infix from tree: A\*B  \*AB = 2  Final Solution: 2 | Pass |
| Valid | 1.5 | Test expression with 2 values, / operator, no parentheses | D / B | Valid Statement  Postfix: DB/  Operations: DB/  Expression Tree:  /  D B  Prefix: /DB  Infix: D/B  Postfix: DB/  Final result: 2 | Input: D / B  Postfix: DB/  Operations: DB/  Tree:  B  /  D  Postfix from tree: DB/  Prefix from tree: /DB  Infix from tree: D/B  /DB = 2  Final Solution: 2 | Pass |
| Valid | 1.7 | Test expression with 3 values, operation with 2 values enclosed in parentheses, first operation added to 3rd operator | ( A \* B ) + C | Valid Statement  Postfix: AB\*C+  Operations: AB\*  AB\*C+  Expression Tree:  +  \* C  A B  Prefix: +\*ABC  Infix: A\*B+C  Postfix: AB\*C+  Final result: 5 | Input: ( A \* B ) + C  Postfix: AB\*C+  Operations: AB\*  AB\*C+  Tree:  C  +  B  \*  A  Postfix from tree: AB\*C+  Prefix from tree: +\*ABC  Infix from tree: A\*B+C  \*AB = 2  +\*ABC = 5  Final Solution: 5 | Pass |
| Valid | 1.8 | Test expression with 3 values, operation with 2 values enclosed in parentheses, first operation divided by 3rd operator | ( C \* B ) / C | Valid Statement  Postfix: CB\*C/  Operations: CB\*  CB\*C/  Expression Tree:  /  \* C  C B  Prefix: /\*CBC  Infix: C\*B+C  Postfix: CB\*C+  Final result: 2 | Input: ( C \* B ) / C  Postfix: CB\*C/  Operations: CB\*  CB\*C/  Tree:  C  /  B  \*  C  Postfix from tree: CB\*C/  Prefix from tree: /\*CBC  Infix from tree: C\*B/C  \*CB = 6  /\*CBC = 2  Final Solution: 2 | Pass |
| Valid | 1.9 | Test expression with 3 values, operation with 2 values enclosed in parentheses, first operation multiplied by 3rd operator | ( A \* B ) \* C | Valid Statement  Postfix: AB\*C\*  Operations: AB\*  AB\*C\*  Expression Tree:  \*  \* C  A B  Prefix: +\*ABC  Infix: A\*B\*C  Postfix: AB\*C\*  Final result: 6 | Input: ( A \* B ) \* C  Postfix: AB\*C\*  Operations: AB\*  AB\*C\*  Tree:  C  \*  B  \*  A  Postfix from tree: AB\*C\*  Prefix from tree: \*\*ABC  Infix from tree: A\*B\*C  \*AB = 2  \*\*ABC = 6  Final Solution: 6 | Pass |
| Valid | 1.10 | Test expression with 3 values, operation with 2 values enclosed in parentheses, first operation subtracted from 3rd operator | ( A \* B ) - C | Valid Statement  Postfix: AB\*C-  Operations: AB\*  AB\*C-  Expression Tree:  -  \* C  A B  Prefix: -\*ABC  Infix: A\*B-C  Postfix: AB\*C-  Final result: -1 | Input: ( A \* B ) - C  Postfix: AB\*C-  Operations: AB\*  AB\*C-  Tree:  C  -  B  \*  A  Postfix from tree: AB\*C-  Prefix from tree: -\*ABC  Infix from tree: A\*B-C  \*AB = 2  -\*ABC = -1  Final Solution: -1 | Pass |
| Valid | 1.11 | Test expression with a combination of operations inside and outside of the parentheses | A + ( A \* B ) / B | Valid Statement  Postfix: AB\*C/A+  Operations: AB\*  AB\*C/  AB\*C/A+  Expression Tree:  +  / A  \* B  A B  Prefix: +\*ABC  Infix: A\*B/B+A  Postfix: AB\*C/A+  Final result: 2 | Input: A + ( A \* B ) / B  Postfix: AAB\*B/+  Operations: AB\*  AB\*B/  AAB\*B/+  Tree:  B  /  B  \*  A  +  A  Postfix from tree: AAB\*B/+  Prefix from tree: +A/\*ABB  Infix from tree: A+A\*B/B  \*AB = 2  /\*ABB = 1  +A/\*ABB = 2  Final Solution: 2 | Pass |
| Valid | 1.12 | Test order of operations at the same level: +, \*, / | B + A \* C / B | Valid Statement  Postfix: AB\*C/B+  Operations: AB\*  AB\*C/  AB\*C/B+  Expression Tree:  +  / B  \* B  A C  Prefix: +/\*ACBB  Infix: A\*C/B+B  Postfix: AB\*C/B+  Final result: 3 | Input: B + A \* C / B  Postfix: BAC\*B/+  Operations: AC\*  AC\*B/  BAC\*B/+  Tree:  B  /  C  \*  A  +  B  Postfix from tree: BAC\*B/+  Prefix from tree: +B/\*ACB  Infix from tree: B+A\*C/B  \*AC = 3  /\*ACB = 1  +B/\*ACB = 3  Final Solution: 3 | Pass |
| Valid | 1.13 | Test order of operations at the same level: +, \*, /, - | B + A \* C / B - D | Valid Statement  Postfix: AB\*C/B+D-  Operations: AB\*  AB\*C/  AB\*C/B+  AB\*C/B+D-  Expression Tree:  -  + D  / B  \* B  A C  Prefix: -+/\*ACBBD  Infix: A\*C/B+B  Postfix: AB\*C/BD+-  Final result: -2 | Input: B + A \* C / B - D  Postfix: BAC\*B/+D-  Operations: AC\*  AC\*B/  BAC\*B/+  BAC\*B/+D-  Tree:  D  -  B  /  C  \*  A  +  B  Postfix from tree: BAC\*B/+D-  Prefix from tree: -+B/\*ACBD  Infix from tree: B+A\*C/B-D  \*AC = 3  /\*ACB = 1  +B/\*ACB = 3  -+B/\*ACBD = -1  Final Solution: -1 | Pass |
| Valid | 1.14 | Test order of operations at the same level: \*, +, /, - | B \* A + C / B - D | Valid Statement  Postfix: BA\*CB/+D-  Operations: BA\*  CB/  BA\*CB/+  BA\*CB/+D-  Expression Tree:  -  + D  \* /  B A C B  Prefix: -+\*BA/CBD  Infix: B\*A+C/B-D  Postfix: BA\*CB/+D-  Final result: -1 | Input: B \* A + C / B - D  Postfix: BA\*CB/+D-  Operations: BA\*  CB/  BA\*CB/+  BA\*CB/+D-  Tree:  D  -  B  /  C  +  A  \*  B  Postfix from tree: BA\*CB/+D-  Prefix from tree: -+\*BA/CBD  Infix from tree: B\*A+C/B-D  /CB = 1  \*BA = 2  +\*BA/CB = 3  -+\*BA/CBD = -1  Final Solution: -1 | Pass |
| Valid | 1.16 | Test order of operations between operations separated by parentheses (-)\*(+) | ( B – A ) \* ( C + D ) | Valid Statement  Postfix: BA-CD+\*  Operations: BA-  CD+  BA-CD+\*  Expression Tree:  \*  - +  B A C D  Prefix: \*-BA+CD  Infix: B-A\*C+D  Postfix: BA-CD+\*  Final result: 7 | Input: ( B - A ) \* ( C + D )  Postfix: BA-CD+\*  Operations: BA-  CD+  BA-CD+\*  Tree:  D  +  C  \*  A  -  B  Postfix from tree: BA-CD+\*  Prefix from tree: \*-BA+CD  Infix from tree: B-A\*C+D  +CD = 7  -BA = 1  \*-BA+CD = 7  Final Solution: 7 | Pass |
| Valid | 1.17 | Test order of operations between operations separated by parentheses (\*)-(/) | ( B \* A ) - ( D / B ) | Valid Statement  Postfix: BA\*DB/-  Operations: BA\*  DB/  BA\*DB/+  Expression Tree:  +  \* /  B A D B  Prefix: +\*BA/DB  Infix: B\*A+D/B  Postfix: BA\*DB/+  Final result: 0 | Input: ( B \* A ) - ( D / B )  Postfix: BA\*DB/-  Operations: BA\*  DB/  BA\*DB/-  Tree:  B  /  D  -  A  \*  B  Postfix from tree: BA\*DB/-  Prefix from tree: -\*BA/DB  Infix from tree: B\*A-D/B  /DB = 2  \*BA = 2  -\*BA/DB = 0  Final Solution: 0 | Pass |
| Valid | 1.18 | Test order of operations between operations separated by parentheses (\*)/(+) | ( B \* A ) / ( D - B ) | Valid Statement  Postfix: BA\*DB-/  Operations: BA\*  DB-  BA\*DB-/  Expression Tree:  /  \* -  B A D B  Prefix: /\*BA-DB  Infix: B\*A/D-B  Postfix: BA\*DB-/  Final result: 1 | Input: ( B \* A ) / ( D - B )  Postfix: BA\*DB-/  Operations: BA\*  DB-  BA\*DB-/  Tree:  B  -  D  /  A  \*  B  Postfix from tree: BA\*DB-/  Prefix from tree: /\*BA-DB  Infix from tree: B\*A/D-B  -DB = 2  \*BA = 2  /\*BA-DB = 1  Final Solution: 1 | Pass |
| Invalid | 2.1 | Test an expression with 2 operands, 1 operator, and only an opening parenthesis at the begging and no closing parenthesis | ( A + B | Invalid statement  Error: Mismatched parentheses | Input: ( A + B  [ERROR] Mismatched Parentheses! | Pass |
| Invalid | 2.2 | Test an expression with 2 operands, 1 operator, and only a closing parenthesis at the end | A + B ) | Invalid Statement  Error: Mismatched parentheses | Input: A + B )  [ERROR] Mismatched Parentheses! | Pass |
| Invalid | 2.3 | Test an expression with 2 operands and no operators | B C | Invalid Statement  Error: No valid operator | Input: B C  [ERROR] Two operands cannot be next to each other! At indexes: 1,3 | Pass |
| Invalid | 2.4 | Test an expression with 1 operand and 1 operator | A + | Invalid Statement  Error: Invalid number of operands | Input: A +  [ERROR] Missing second operand at indexes: 3,5 | Pass |
| Invalid | 2.5 | Test an expression with no operands and 1 operator | + | Invalid Statement  Error: Invalid number of operands | Input: +  [ERROR] Missing second operand at indexes: 1,3 | Pass |
| Invalid | 2.6 | Test an expression with only a set of parentheses | ( ) | Invalid Statement  Error: No valid operation inside parentheses | Input: ( )  [ERROR] Empty line! | Pass |
| Invalid | 2.7 | Test an expression with two operations and a missing closing parenthesis | ( A + B ) – ( C \* D | Invalid Statement  Error: Mismatched parentheses | Input: ( A + B ) - ( C \* D  [ERROR] Mismatched Parentheses! | Pass |
| Invalid | 2.8 | Test an expression with two operations and a missing opening parenthesis | ( A + B ) – C \* D ) | Invalid Statement  Error: Mismatched parentheses | Input: ( A + B ) - C \* D )  [ERROR] Invalid spacing! At indexes: 19,20  [ERROR] Mismatched Parentheses! | Pass |
| Invalid | 2.9 | Test an expression with two operations enclosed in parentheses and no operator between them | ( A + B ) ( C – D ) | Invalid Statement  Error: Missing Operator | Input: ( A + B ) ( C - D)  [ERROR] Missing operator! At indexes: 9,11  [ERROR] Invalid spacing! At indexes: 17,18  [ERROR] Invalid spacing! At indexes: 18,19 | Pass |
| Invalid | 2.10 | Test the function checkOperator by inputting a non-operator value | A | Function returns false | Returned false | Pass |
| Invalid | 2.11 | Test the function getWeight by inputting a non-operator value | B | Function returns -1 ~~and displays an error: “Error: Invalid operator input [getWeight]”~~ | Returned -1 | Pass |
| Invalid | 2.12 | Test the function getCharacterValue by inputting a character outside of the range of valid character operands | ] | Function returns -1 and displays an error: “Error: Invalid character value <character>, defaulting to -1 [getVariableValue]” | Function returns -1 and displays an error: “Error: Invalid character value <character>, defaulting to -1 [getVariableValue]” | Pass |
| Invalid | 2.13 | Attempt to process an operation that will divide by zero | A / ( A – A ) | Valid Statement  Postfix: AA-A/  Operations: AA-  AA-A/  Expression Tree:  /   * A   A A  Prefix: /-AAA  Infix: A-A/A  Postfix: AA-A/  “ERROR: Cannot solve, division by zero: <integer1> / <integer2>”  Final result: error | Input: A / ( A - A )  Postfix: AAA-/  Operations: AA-  AAA-/  Tree:  A  -  A  /  A  Postfix from tree: AAA-/  Prefix from tree: /A-AA  Infix from tree: A/A-A  -AA = 0  [ERROR] Cannot divide by zero! | Pass |
| Invalid | 2.14 | Pass a null pointer to a root node as an input parameter for the function displayTree | Null pointer | Returns empty string value | Returned empty value | Pass |
| Invalid | 2.15 | Attempt to dequeue an element from an empty queue | N/A | Dequeue function returns an empty value.  Display an error: “Error: cannot dequeue from an empty queue.” | “[Error] Cannot dequeue from an empty queue.” | Pass |
| Invalid | 2.16 | Pass a null pointer to a tree root node as a parameter to the getPostfixFromTree function | Null pointer | Returns an empty string | Returned an empty string | Pass |
| Invalid | 2.17 | Pass a null pointer to a tree root node as a parameter to the getPrefixFromTree function | Null pointer | Returns an empty string | Returned an empty string | Pass |
| Invalid | 2.18 | Pass a null pointer to a tree root node as a parameter to the getInfixFromTree function | Null pointer | Returns an empty string | Returned an empty string | Pass |
| Invalid | 2.19 | Pop a value off of an empty stack | N/A | Returns a null value | Null | Pass |
| Invalid | 2.20 | Call the top function on an empty stack | N/A | Top returns a null value | Null | Pass |
| File Handling | 3.1 | Input a file name that does not exist | Somefile.dat | Error: Input file does not exist | <File> does not exists! | Pass |
| File Handling | 3.3 | Input a file of type “.dat” that exists | Input.dat | Open file successfully | File opened | Pass |
| File Handling | 3.4 | Input a file of type “.dat” that exists, but is empty | Input.dat | Error: empty input file | File is empty error | Pass |
| File Handling | 3.5 | Attempt to open an output file that is opened by another program | output.dat | Should still open the file and take control of writing to it | Opened file | Pass |

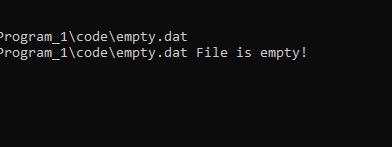
1. **Screenshots**



Test case 3.5 PASS



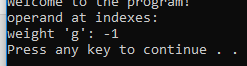
Test case 3.1 PASS



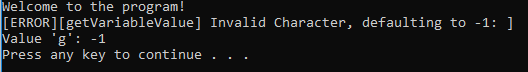
Test Case 3.4 PASS



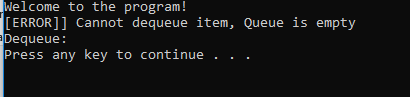
Test case 2.10 PASS



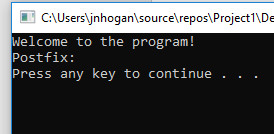
Test case 2.11 PASS



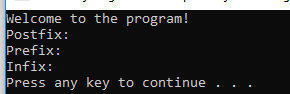
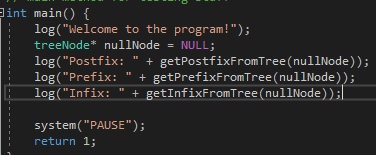
Test case 2.12 PASS



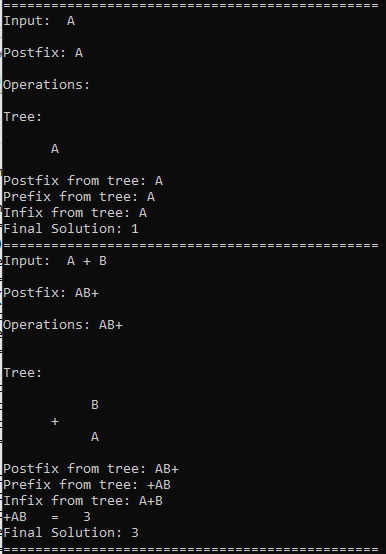
Test case 2.15 PASS



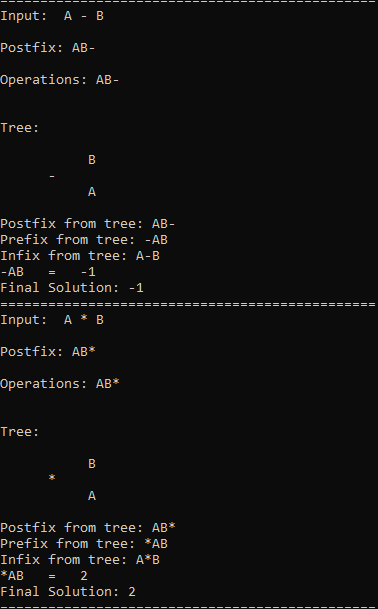
Test case 2.16 PASS



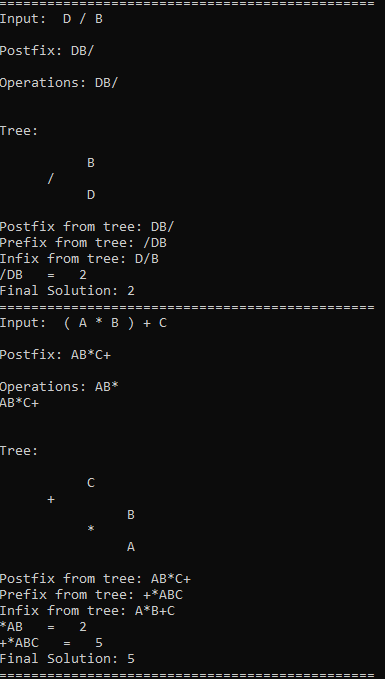
Test case 2.16, 2.17, 2.18 PASS



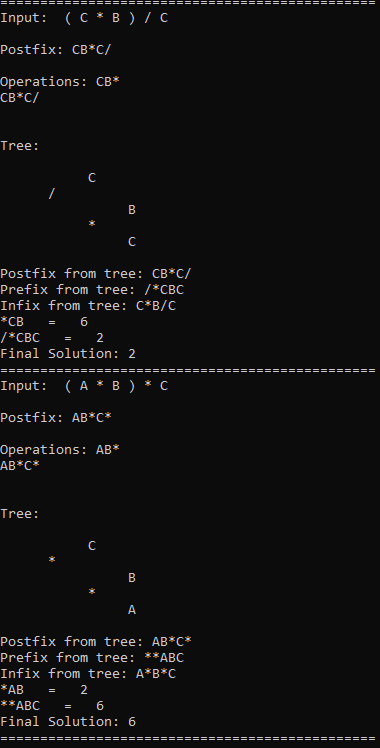
Test cases 1.1, 1.2 PASS



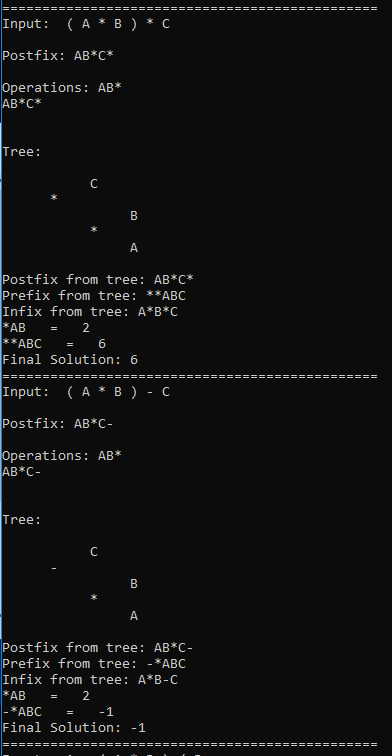
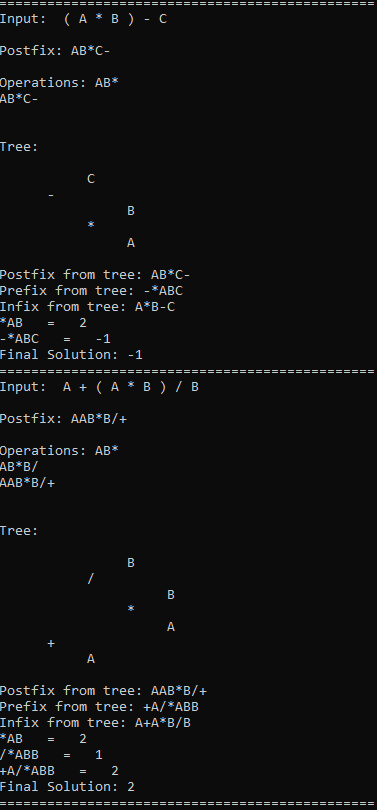
Test cases 1.3, 1.4 PASS



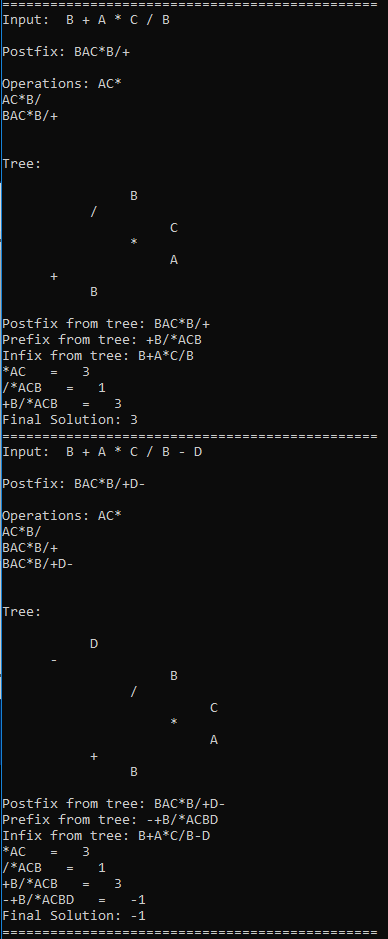
Test cases 1.5, 1.7 PASS



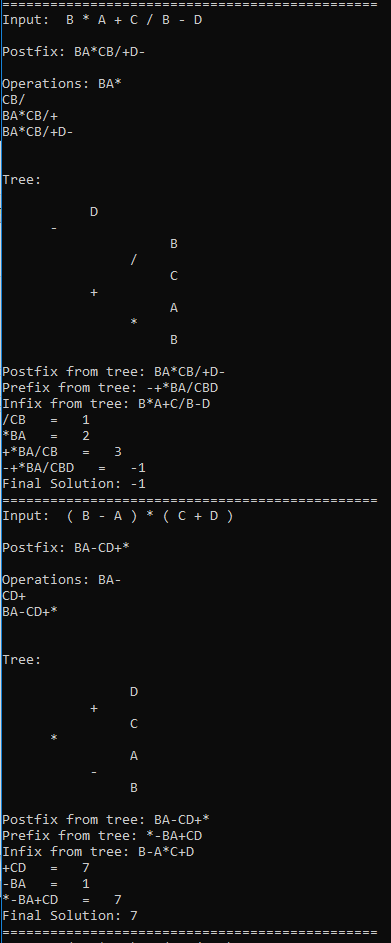
Test cases 1.8, 1.9 PASS



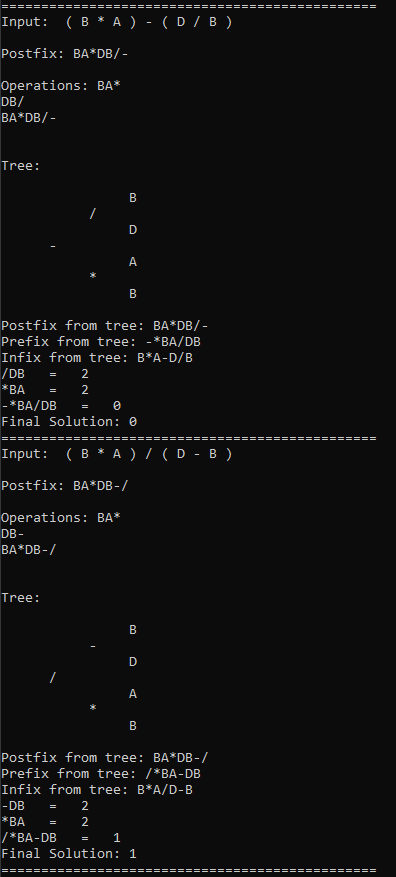
Test case 1.10, 1.11 PASS



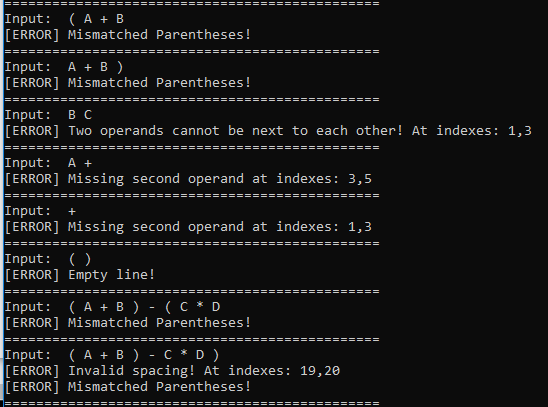
Test cases 1.12, 1.13 PASS



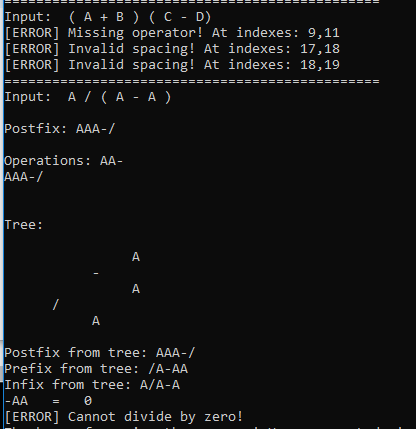
Test cases 1.14, 1.16 PASS



Test cases 1.17, 1.18 PASS



Test cases 2.1, 2.2, 2.3, 2.4, 2.5, 2.6, 2.7, 2.8 PASS



Test cases 2.9, 2.13 PASS

1. **Error Log**

|  |  |  |
| --- | --- | --- |
| Error Type | Cause of Error | Solution to Error |
| Logic | The postfix operations were being printed in the wrong order | Instead of adding the first operand popped to the second, it needed to add the second operation to the first operation |
|  |  |  |

1. **Status**

The application has been tested and should accurately evaluate infix expressions of any size and display the tree, postfix, infix, prefix, operations, and the evaluation of it.