1 Analysis

Building a binary expression tree with given postfix expression can be done using LIFO stack scheme. To define the stack for the tree, I used linked list containning a list of tree nodes. A node is defined as either an operand or operator, stores the value, and stores three pointers left, right, and next. The left pointer points to left child of the binary subtree while the right pointer points to the right child. The extra pointer next is used to point to the next node that follow after a node in the tree stack. The tree stack contains a single pointer top pointing the top of the stack or the end of the linked list. To build the expression tree with postfix expression, the algorithm scans the expression from left to right. If the node is a operand, the algorithm pushes the node to the tree stack. Else, whenever it encouters a operator node, it pops the top two nodes left and right from the stack. At this instance, if the stack has less than two nodes, the expression is incomplete so the algorithm stops and continue to the next expression. The operator node's left and right pointers are assigned to the left and right nodes that were popped. Then the operator node is pushed to the stack. This process repeat until the end of the expression. If the expression is correct, the tree stack should now only contain the root of the tree. The binary expression tree is now complete. This algorithm to build the binary expression tree has a linear complexity of $O(n)$, where n is the number of nodes.

Traversing the expression tree to infix expression is a simple task. To do this, the algorithm recursively going through the left and right nodes starting at the root node. The base case of the recursion is when the current node have no children. So, after a left subtree solved, it output (display) the current node value then proceed to solve the right subtree. Basically, the order of outputing follows as left subtree, operator, then right subtree. The infix traversal takes complexity of $O(n)$.

Similarly, prefix expression traversal is done using recursive call through the right and left subtree from the root node. The output order follows as operator, left subtree, right subtree. However, an additional step is required to evaluate the expression using prefix. In this case, when an operand is reached (base case), the algorithm returns the operand's value. If the node is a operator, the algorithm uses the results from left subtree and right subtree then evaluate left and right solution with the operator. The prefix traversal also takes complexity of $O(n)$.

2 Results

All of the results from the examples are as expected even with invalid expression. From the result below, if the postfix expression were to be evaluate, the result will be the same as the prefix expression. This can be done by hand. For infix expression, eventhough the expression is not always correct (without parenthesis), if it were to evaluate during the traversal, the result is also the same. This can be done by following through the expression tree.

As discussed before, the expression tree is builded using LIFO stack. Additionally, all the of traversal are using depth-first search (DFS) scheme and only the order of outputing is difference. It is a DFS because it starts at the root then always visiting the left subtree first. When it reach a leaf node, it trace back to the lastest discovering node to process the right subtree. Thus, LIFO stack and DFS are the two important concepts for this program.

```
----------New Expression-----------
Postfix:
+
Error: Missing operand/s for '+'


----------New Expression-----------
Postfix:
2 3 5 * + 9 -

Binary Tree:
'--[-]
    |--[+]
    |   |--[2]
    |   '--[*]
    |       |--[3]
    |       '--[5]
    '--[9]

Infix:
2 + 3 * 5 - 9
Prefix:
- + 2 * 3 5 9
Result: 8


----------New Expression-----------
Postfix:
F A B L E * - / +

Binary Tree:
'--[+]
    |--[F]
    '--[/]
        |--[A]
        '--[-]
            |--[B]
            '--[*]
                |--[L]
                '--[E]

Infix:
F + A / B - L * E
Prefix:
+ F / A - B * L E
Result: 69
```

```
----------New Expression-----------
Postfix:
10 99 33 / 5 167 * - +

Binary Tree:
'--[+]
    |--[10]
    '--[-]
        |--[/]
        |   |--[99]
        |   '--[33]
        '--[*]
            |--[5]
            '--[167]

Infix:
10 + 99 / 33 - 5 * 167
Prefix:
+ 10 - / 99 33 * 5 167
Result: -826


----------New Expression-----------
Postfix:
F A B L - / E * + S
Error: Invalid expression 'S'


----------New Expression-----------
Postfix:
FB FA +

Binary Tree:
'--[+]
    |--[FB]
    '--[FA]

Infix:
FB + FA
Prefix:
+ FB FA
Result: 140
```

3 Source Code

All of the header files are stored in the include folder and source files are stored in src folder, which include the expression tree, stack, and node classes. The example input files are stored in

the examples folder. The root folder includes the main.cpp, Makefile, and the executable. The program takes in the a file. Then it begin by reading the file line by line. For each line, it build a expression tree and display the tree using ExprTree class's member (postfix_to_tree()). Note, the tree is displayed horizontally instead of vertically. Also, it outputs the infix and prefix expressions. All outputs are outputed to standard out. For linux machine, the result can be same by the following command: "./expr_tree <input_file> >> <output_file>"