COP 4530
Project 2
Spring 2024

Instructions

For Programming Project 2, you will implement a Deque (Double-Ended Queue) and use that data structure to write a class that can convert between the three common mathematical notation for arithmetic.

The three notations are:

Postfix (Reverse Polish) Notation:

Operators are written after operands `A B - C + == (A - B) + C`

Infix Notation:

The standard notation we use where the operator is between the operands.

Prefix (Polish) Notation:

Operators are written before operands: `* - A B C == (A - B) * C`

- The converter will be able to convert from one of those three to any other. The input will be a string written in a single notation, and the output will be the conversion to the specified notation.

- The input string for the functions will only ever contain the standard four arithmetic operators (* /+ -), an operand denoted by a letter (upper or lower case), a left or right parentheses (only round), or spaces.

- Whitespace (one or more characters of space) separate all operand and operators, while parentheses must have whitespace on the outside (between operators), and
  inside parentheses whitespace is optional.
- Parentheses are only used in infix strings.

Note: Parentheses must have operators between them, as all operations must be binary and not
implied. For example:
Valid Postfix: `c d / a b * r r * / *`
Valid Prefix: `* + A B - C D`
Valid Infix: `(( X + B ) * ( Y - D ))`
Invalid Postfix:`c d a b * r r * / *` (Operators don't match up with operands)
Invalid Prefix: `* ^ A B & C D` (Invalid Characters)
Invalid Infix: `((a / f) ((a * b) / (r * r)))` (No operator between parentheses)

- The output string should always separate all operand and operators by ONLY one space.
- The interior of a parenthesis should have no whitespace between the letter and the parenthesis or another parenthesis, while the exterior of a parenthesis should be separated by ONLY one space from an operator and none for another parenthesis. For example:
  Valid Output: `((x / y) - g)`
  Valid Output: `((x / y) - (a * b))`
  Valid Output: `x y * g / h +`

If your output does not conform to this standard, you will not pass the tests required for this Project. Again, there is an abstract class for you to inherit that has you implementing the following methods

Abstract Class Methods

`std::string postfixToInfix(std::string inStr)`
This method takes in a string of postfix notation and returns a string in the infix notation

`std::string postfixToPrefix(std::string inStr)`

This method takes in a string of postfix notation and returns a string in the prefix notation

`std::string infixToPostfix(std::string inStr)`
This method takes in a string of infix notation and returns a string in the postfix notation

`std::string infixToPrefix(std::string inStr)`
This method takes in a string of infix notation and returns a string in the prefix notation

`std::string prefixToInfix(std::string inStr)`
This method takes in a string of prefix notation and returns a string in the postfix notation

`std::string prefixToPostfix(std::string inStr)`
This method takes in a string of prefix notation and returns a string in the postfix notation

These methods will all be instance methods of the class `NotationConverter` (which must be called "NotationConverter"). You will also need to implement a fully functional deque using a doubly linked list. When writing the methods to convert between the notations, you can ONLY use your deque and strings for storing data in the algorithms. You may not use any C++ Standard Library containers, such as the STL Deque, Stack, Queue, Vector, or List.

Examples

Below are some examples of how your code should run. The driver file can also be used to get an
idea of how the code should run.

```
NotationConverter nc;
std::string examplePost = "c d / a b * r r * / *";
nc.postfixToInfix(examplePost) // Infix: ((c / d) * ((a * b)
/ (r * r)))
nc.postfixToPrefix(examplePost) // Prefix * / c d / * a b *
r r
std::string examplePre = "* + A B - C D”;
nc.prefixToInfix(examplePre) // Infix: ((A + B) * (C - D))
nc.prefixToPostfix(examplePre) // Postfix A B + C D - *
std::string exampleInfix = "((a / f) ((a * b) / (r * r)))";
nc.infixToPostfix(exampleInfix) // Postfix: a f / a b * r r
* /
nc.infixToPrefix(exampleInfix) // Prefix / a f / * a b * r r
```

Deliverables

Please submit complete projects as zipped folders. The zipped folder should contain:

1. `NotationConverter.hpp` (Your source file for NotationConverter class)
2. `NotationConverter.cpp` (Your header file for NotationConverter class)

Hints

- It would be a good idea to decode the input string and place it into a deque to make it easier to read for the various methods

- When dealing with infix notations, remember that operator precedence is very important.

- A deque can operate like a stack or a queue, queues and stacks can be considered specializations of a deque.

- You may not need to write a separate algorithm for all six methods. For example, when going from postfix to prefix, you can go from postfix to infix and infix to prefix if those methods are already implemented.

Rubric

Any code that does not compile will receive a zero for this project.

Test cases: 24* 3.75= 90
Documentation: 10
———
Total 100