Project 3: Expression Tree

DUE: Sunday, April 8th at 11:59pm

Basic Procedures

You must:

- Fill out a readme.txt file with your information (goes in your user folder, an example readme.txt file is provided)
- Have a style (indentation, good variable names, etc.)
- Comment your code well in JavaDoc style (no need to overdo it, just do it well)
- Have code that compiles with the command: javac *.java in your user directory

You may:

- Add additional methods, fields, and classes, however these must be private (or package default for fields/methods inside nested classes).

You may **not**:

- Make your program part of a package.
- Add additional public methods, variables, or classes.
- Use any built in Java Collections Framework classes anywhere in your program (e.g. no ArrayList, LinkedList, Stack, Queue, etc.) or add any additional import statements.
- Alter any method signatures defined in this document or the template code.
- Add any additional libraries/packages which require downloading from the internet.

Setup

- Download the project3.zip and unzip it. This will create a folder `section-yourGMUUserName-p3`
- Replace "`section`" with `z001, z002, k003`, and `z004` for the 4 sections respectively.
- Rename the folder replacing `yourGMUUserName` with the first part of your GMU email address/netID. Example: `k003-jkrishn2-p3`
- Complete the `readme.txt` file (an example file is included)

Submission Instructions

- Make a backup copy of your user folder!
- Remove all test files, jar files, class files, etc.
- You should just submit your java files and your readme.txt
- Zip your user folder (not just the files) and name it as "`section-yourGMUUserName-p3.zip`" (no other type of archive) where "`yourGMUUserName`" is your GMU email address / netID.
- Submit to blackboard.

Grading Rubric

Due to the complexity of this assignment, an accompanying grading rubric pdf has been included with this assignment. Please refer to this document for a complete explanation of the grading.

Topics Covered

Trees, Recursion, Stacks, and Queues

Overview

In this project, you will be working with expression trees represented in First-Child-Next-Sibling format. Basic features of the tree include:

- As an expression tree, every node represents either an operator or an operand. We restrict our operands to be integers only. For operators, we only consider the ones included in the table below. Refer Weiss 11.2.4 (pp. 468) for more details.
- As a First-Child-Next-Sibling tree, each node keeps two links, one to its left child (if it is not a leaf) and one to its right sibling (if it is not the rightmost sibling) in the normal binary expression tree. Refer Weiss 18.1.2 (pp. 653) for more details.

Operator	Definition	Example	Example Evaluation
+	Addition	5 + 3	8
-	Subtraction	5 - 3	2
*	Multiplication	5 * 3	15
/	(integer) Division	5 / 3	1
%	Remainder	5 % 3	2
~	Negation	~ 5	-5

Examples: Representations of the expression **`(5+3)*((5*10)/2)`** is shown below.

Figure 1: Binary Tree Representation

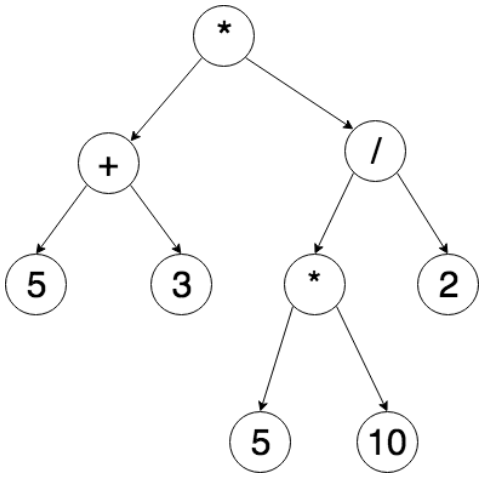
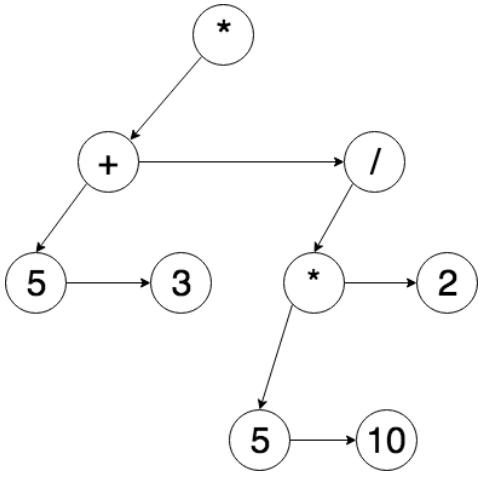


Figure 2: First-Child-Next-Sibling Tree Representation



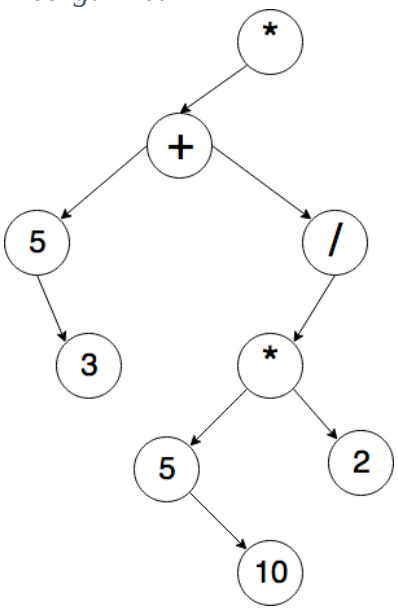
A tree node in both representations, maintain three fields: value and links to its two children. A binary tree maintains **`left`** and **`right`** children while a FCNS tree maintain **`firstChild`** and **`nextSibling`** children. Thus, in a binary tree representation **`+.left`** is **`5`** and **`+.right`** is **`3`** while in the FCNS representation **`+.firstChild`** is **`5`** and **`+.nextSibling`** is **`/`**. Note that the root nodes in both representations are the same.

Classes Overview

ExpressionBinaryTree and **BinaryTreeNode (ExpressionBinaryTree.java)**: These two classes are used to represent expressions as binary trees. They are already implemented and provided to you. Do not make any modification to them.

Stack (Stack.java) and **Queue (Queue.java)**: These two are generic implementation of stack and queue data structures respectively. You will find them useful in multiple tasks you need to implement in this project.

Figure 3: FCNS Tree reorganized



ExpressionFCNSTree and **FCNSTreeNode (ExpressionFCNSTree.java)**: These two classes are used to represent expressions as FCNS-format trees. FCNSTreeNode is already implemented and provided to you. You should use it without making any modification. **ExpressionFCNSTree** is the main class that you will need to implement. We have provided the signature and description of all required methods in the template Java file. Below is the list of main tasks.

Task 1: Reporting tree properties, including the size, height, and the number of tree nodes with certain features.

Task 2: Generating string representation of the tree. You will need to generate a string representing pre-order, post-order, and level-order (breadth-first) tree traversal of the FCNS tree respectively. Check the template file for format requirements and examples. For example, based on the FCNS tree in Figure 4, we should have the following string representations:

- Prefix: "+ % 5 3 10 "
- Postfix: "3 5 10 % + "
- Level-order: "+ % 5 10 3 "

Task 3: Construct a FCNS tree from an expression in prefix notation using method buildTree(). This method should accept a String parameter filename, open the file and construct a FCNS expression tree based on the file contents. Each input file contains a numeric expression in prefix notation. For example, `* + 5 3 / * 5 10 2` represents `(5+3)*((5*10)/2)` and corresponds to a FCNS tree shown in Figure 2. You can assume that the input file is error-free and that operators/operands are separated by a space. More examples are included in Figure 3 and 4.

Figure 3: FCNS for `* ~ 5 10`

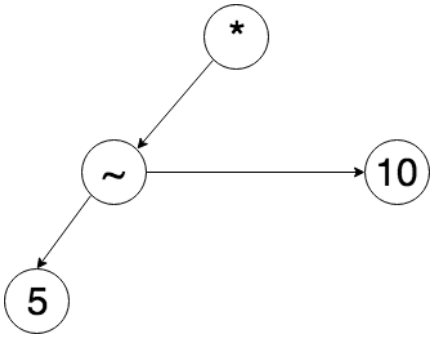
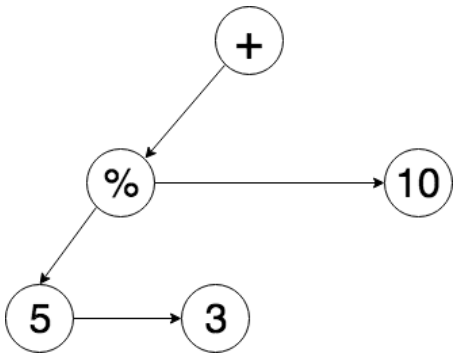


Figure 4: FCNS for `+ % 5 3 10`



Allowed Operators: add(+), subtract(-), multiply(*), divide(/), mod(%), negate(~). See the previous table for definitions and examples. Note that the tree supports unary operator ~ which can complicate your code.

Allowed Operands: integers (positive, negative, zero).

Task 4: Construct a binary tree from the FCNS tree using buildBinaryTree(). You will need to create the corresponding binary tree from the FCNS tree. For example, given the FCNS tree in Figure 2, a binary tree as in Figure 1 should be constructed and the root should be returned.

Task 5: Simulate in-order binary tree traversal with the FCNS tree using toStringPrettyInFix(). You will need to generate an infix notation of the expression. This is essentially simulating the in-order traversal of the corresponding binary expression tree. Note that you are NOT allowed to first construct the binary tree and then perform an in-order traversal to generate the string. The infix notation must be generated from the FCNS tree directly without building a binary tree. Parentheses need to be inserted to make the expression correct and human-readable. For example, the string generated from Figure 2 should be **`(5+3)*((5*10)/2)`**.

Task 6: Evaluate the expression to an integer from FCNS tree, both recursively and iteratively. In this task, you will compute the result of the expression using the FCNS tree.

Task 6.1: Recursive Implementation. Implement method evaluate() for this task. The method walks through the tree and update two attributes for individual nodes: an integer **value** and a boolean flag **nan** to indicate whether the expression is not-a-number. Normally, the value of an operand node is the integer value of that operand; the value of an operator is the integer value associated with sub-expression rooted at that node. For example, the value of each tree node in Figure 4 after we perform evaluate() is given in the following table:

Node symbol	5	3	%	10	+
Node.value	5	3	2	10	12
Reason	operand	operand	5 % 3	operand	(5%3)+ 10 = 2+10

The special situation not-a-number is triggered when a division has a zero divisor. Then the not-a-number flag needs to be propagated following this rule: any operation applying on a not-a-number yields a not-a-number result. When a node is marked as not-a-number, its value should be null.

Task 6.2: **Iterative Implementation.** Implement method evaluateNonRec() for this task. This method returns an integer value without changing any tree node attributes. For this task, you can assume all expressions are valid and there is no division-by-zero.

Sample inputs and results:

Input (prefix notation)	**Pretty-Infix notation (or Binary Tree Infix Traversal with parenthesis)**	**FCNS Infix notation (**implementation not required**)**
`* + 5 3 / * 5 10 2`	`((5+3)*((5*10)/2))`	`5 3 + 5 10 * 2 / *`
`* ~ 5 10`	`((-5) * 10)`	`5 ~ 10 *`
`+ % 5 3 10`	`((5 % 3) + 10)`	`5 3 % 10 +`
`/ 10 ~ * 5 2`	`(10 / -(5 * 2))`	`10 5 2 * ~ /`

Table continued…

FCNS Postfix notation	FCNS Level Order	Result
`3 5 10 5 2 * / + *`	* + 5 / 3 * 5 2 10	200
`5 10 ~ *`	* ~ 5 10	-50
`3 5 10 % +`	+ % 5 10 3	12
`2 5 * ~ 10 /`	/ 10 ~ * 5 2	-1

Big-O

Template given to you in the starter package contains instructions on the REQUIRED Big-O runtime for a subset of methods. For those methods, your implementation should not have a higher Big-O and you will be graded on this.

Testing

Test cases will not be provided for this project. However, feel free to create test cases by yourselves. In addition, the main methods provided along with the template classes contain useful code to test your code. You can use command like "`java ExpressionFCNSTree`" to run the testing defined in main( ). You could also edit main( ) to perform additional testing. As always, a part of your grade will be based on automatic grading using test cases that are not provided to you. A set of expression files are provided to you for testing purpose.