## Binary Expression Tree (BXT)

A BXT is binary tree in which the leaves are numbers and the nonleaves are arithmetic operators. As you might expect, we will build a binary expression tree, display it, and evaluate it. Unlike TreeLab, we will encapsulate the behavior in a BXT class so that all the methods are instance methods. The teacher has written the driver class with the postfix expressions you are to use.

In order to allow decimals and negative numbers, we require that the tokens in the input string be separated by a space. You may assume that the input postfix expression is well-formed.

**Sample Run**

|  |  |  |
| --- | --- | --- |
| Postfix Exp: 14 -5 /   -5 /  14 Infix order: 14 / -5  Prefix order: / 14 -5  Evaluates to -2.8 ----------------------  **A simple inorder traversal ignores order of operations** | Postfix Exp: 20.0 3.0 -4 + \*   -4  + 3.0 \*  20.0 Infix order: 20.0 \* 3.0 + -4  Prefix order:\* 20.0 + 3.0 -4  Evaluates to -20.0 ------------------------ | Postfix Exp: 2 3 + 5 / 4 5 - \*  5  -  4 \*  5  /  3  +  2 Infix order: 2 + 3 / 5 \* 4 - 5  Prefix order: \* / + 2 3 5 - 4 5  Evaluates to -1.0 ------------------------ |

**Build a BXT**

You need to process the postfix expression String so it forms a binary expression tree stored at root.

Hint 1: The input is a string with spaces. Use str.split(" "); and a for-each loop.

Hint 2: Start with the operators + - \* / %. In postfix notation these operators are **preceded** by   
 two operands (numbers). This suggests a **stack of TreeNodes** would be useful.

Hint 3: Build the tree: each operator needs to be placed in a TreeNode with links to its two children, also   
 in TreeNodes. Then push the TreeNode-with-operator.

Hint 4: Evaluate the tree: each TreeNode contains an Object value. If the value is an operator, it   
 must be cast to a String. If the value is a number, it must be changed to a String, then parsed   
 to a double before it can be evaluated.

Hint 5: A postfix string that has only one item is a special case.

**Display Infix and Prefix orders**

You don't need help with this. Each one needs, as usual, a public no-argument method which calls a private recursive method with arguments.

## Evaluating the Expression

Do this recursively. If the node is an operator, recursively evaluate the left child and the right child, and return the result. Else the node is a number, so it can be converted into a double and returned.

**Extensions**

1. Process the operators ^ and !
2. Create a method inorderTraverseWithParentheses which returns the infix expression that matches the operations in the tree. Use parentheses, but only if necessary. For example,

8 2 3 + \* 🡪 8 \* ( 2 + 3 )

8 2 3 \* + 🡪 8 + 2 \* 3

8 2 3 + + 🡪 8 + ( 2 + 3 ) //matches the operations in tree

**import** java.util.\*;  
/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
Represents a binary expression tree. The BXT can build itself from a postorder expression. It can evaluate and print itself. It also prints an inorder string and a preorder string.   
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  
**public class** BXT   
 {

**public** **static** **final** String operators = "+ - \* / % ^ !";  
 **private** TreeNode root;  
 **public** BXT()  
 {  
 root = null;  
 }  
 /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
 Builds a BXT from a postfix expression. Uses a helper stack of

TreeNodes.  
 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  
 **public void** buildTree(String str)  
 {

}  
 **public double** evaluateTree()  
 {  
 **return** evaluateNode(root);  
 }  
 **private double** evaluateNode(TreeNode root) //recursive  
 {

}  
 **private double** computeTerm(String s, **double** a, **double** b)  
 {

}  
 **private** boolean isOperator(String s)   
 {

}  
 // display() from TreeLab  
 // inorder traverse  
 // preorder traverse  
}