Tree Structure:

Expression tree is a binary tree with two types of nodes, leaves which are operands (they can be literal values or variables) and internal nodes which are operators. The incomplete src codes are an extension of the BinaryTree which we worked on in class and in Lab6. To deal with different type of nodes, you can change the code and add different fields to class BinaryNode: (this is just a suggestion, you can change and add more fields)

```
class BinaryNode
{
      private int value; // the associated value of this subtree
      private String token; // 'a', '+', '3'
      private boolean isOperator; // internal or leaf node
      private BinaryNode right; // right child
      private BinaryNode left; // left child
…
}
```

Note that this is not a generic code anymore. This class is specifically designed for the node in an expression tree.

Handle Leaf Nodes

To keep track of variables, you can create a hashtable (HashMap in Java).

How to Create and use a HashMap:

```
// create a map:
HashMap<String, Integer> VariableTable = new HashMap<String, Integer>();

// add a new variable:
VariableTable.put("a", 3);

// check a value of a variable:
int x = VariableTable.get("a")
```

Then use them when you are building the tree. If the token e is a variable (e.g. "a"), get the value from the hash table, if it is a literal integer (e.g. "45") convert the string to integer value (you can use: `Integer.parseInt("45")`)

- Create an empty stack for operators opStack
- Create an empty stack for subtrees treeStack
- Read the infix expression from left to right (let e be the next token in the infix expression)
  - **if e is an operand, build a tree with e as root and push the tree to treeStack**
  - if e is a left parentheses, "(", push it to the opStack
  - if e is an operator:
    - if opStack is empty, push e to the opStack
    - otherwise, pop operators of greater or equal precedence from the opStack
      For each operator,op, that you popped, pop two trees from treeStack, (t1, t2)
      Build a new tree with op as the root, t2 as left subtree and t1 as right subtree
      Push the new tree to treeStack
    - You should stop when you see an operator of lower precedence or "(" and then push e to the opStack
  - if e is ")" , pop operators from the opStack until you see matched "(" (NOTE: you should also pop "(")
    For each operator, op, that you popped, build a tree: (similar to previous case)
    pop two trees from treeStack, (t1, t2)
    Build a new tree with op as the root, t2 as left subtree and t1 as right subtree
    Push the new tree to treeStack
- When you reach to the end of infix expression, pop all operators in opStack:
  For each operator, op, that you popped, build a tree: (similar to previous case)
  pop two trees from treeStack, (t1, t2)
  Build a new tree with op as the root, t2 as left subtree and t1 as right subtree
  Push the new tree to treeStack

- at the end there should be just one tree in treeStack, and it will be the full expression tree