Laboratory 11: Expression Trees and Binary Search Trees

Introduction

Trees are nonlinear objects that link nodes together in a hierarchical fashion. Each node contains a reference to the data object, a reference to the left subtree, and a reference to the right subtree. Methods that process trees are often recursive. There are three primary patterns of recursive tree methods: inorder, preorder, and postorder.

For inorder methods, the sequence is to do the recursive call on the left subtree, then process the node, then do the recursive call on the right subtree.

For preorder methods, the sequence is to process the node, then do the recursive call on the left subtree, then do the recursive call on the right subtree.

For postorder methods, the sequence is to do the recursive call on the left subtree, then do the recursive call on the right subtree, then process the node.

Also, recursive methods are usually declared private, and require a public non-recursive method to initiate them.

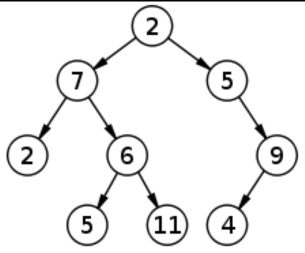
For example, a common way to display the nodes of a tree in order would involve the following two methods:

```
public void inOrder(){
      inOrderSub( root );  // start the recursive partner off at the root node
}

public void inOrderSub(TreeNode p){
      if (p != null){
            inOrderSub(p.getLeft());  // print the left subTree
            System.out.println( p.getElement()); // print the node data
            inOrderSub(p.getRight());  // print the right subTree
      }
}
```

The other two traversal patterns would simply change the order of the three statements inside the if statement.

To make sure you understand these basic principles, please print out the contents of the following tree as it would appear using an inOrder, preOrder, and postOrder traverse:

	InOrder: PreOrder: PostOrder:

Part A: Binary Search Trees

Although Binary Search Trees are a more complex than Expression Trees, we will begin with a simplified version of a BST so you can get a sense of how trees work in general. Part B will ask you to develop an expression tree implementation.

In class Demos

We develop a simple (non-generic) Binary Search Tree of String class:

1) basic linking of TreeNodes
2) recursive methods: printing in order and in graphic form
3) iterative methods: add, contains, get
4) inorder, preorder and postorder traversal methods
5) returning an iterator of tree data

Development Tasks

1) Complete the three traversal methods inOrder, preOrder and postOrder
2) Make and test a method `count`, to count the number of nodes in a BST. This method will call its recursive partner `countAux` on `root`. `CountAux` will recursively count the nodes in its left and right subTrees, then return the total of them both plus 1 to include itself.
3) Make a method to compute the height of a BST. This method counts the number of nodes on the longest path from the root to any leaf node in the tree. This statistic is significant because the amount of time it can take to find an item in the tree is related to the height of the tree.

   ```
   int height()
   // pre: none
   // post: returns the height of the tree
   ```

   You can compute the height of a tree using a postorder traversal, and the following recursive definition of height:

   ```
   heightAux(subTree) = 0          if  subTree is null
                  else  =  1 + the larger of heightAux( subTree.left) and heightAux(subTree.right)
   ```

4) Copying trees is a tricky operation, since if you are not careful, your copy will be "entangled" with the original tree (the nodes in the two trees will refer to the same data objects). Make a `copy` method for your BST. This method should make and return a complete copy of the original tree where every `String` data in every `TreeNode` has been duplicated (this is a "deep copy"). To duplicate a `String`, you can invoke the `String` copy constructor method, similar to the code that is underlined below:

   ```
   TreeNode newNode = new TreeNode(new String(subTree.data));
                        // using String copy constructor to duplicate a String
   ```

   (you may be familiar with `clone`, but `String` does not implement `clone` since `String`s are immutable, and there are a number of reasons to avoid writing clone for your own classes. For now, it's best to use the example above to duplicate a `String` object.

The usual way to copy a tree is with a recursive pair of methods that implement a pre-order traversal. The `copy` method for BST can simply create a new empty BST, then invoke the `copyAux` method passing `newBST` and `root` as parameters. The `copyAux` method will then visit the data in its `subTree` parameter and add a copy of that data into the `newBST`, and invoke itself again on the left and right `subTree`s in preorder fashion.

Back in `copy`, after the recursive `copyAux` method completes, the `newBST` is returned.

Test your method using the code provided.

Part B: Expression Trees

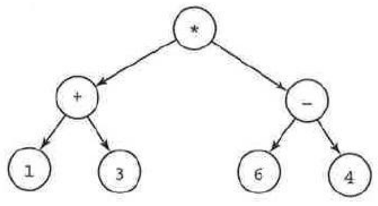
Expression trees are another great way to explore the use of heirarchical data structures. In this section you will

1) create an implementation of the Expression Tree ADT using a linked tree structure.
2) define a build method that builds an expression tree from a pre-fix arithmetic expression
3) define an evaluate method that evaluates an expression tree

OVERVIEW

Although you ordinarily write arithmetic expressions in linear form, you treat them as hierarchical entities when you evaluate them. When evaluating the following arithmetic expression, for example, (1+3)*(6-4)

you have to evaluate the two subexpressions in parentheses before multiplying their results together. You can express the above in the following tree form, indicating the order of operations by level in tree (the lowest levels must be evaluated before the higher levels).



We will develop an expression tree where each node contains a char variable indicating either a digit or an operator. Then we will define methods to build and evaluate expression trees.

1. To begin, open the ExpressionTree project folder. Examine the two different program shells. The TreeNode class is contained inside the ExprTree class as it was for the BST.

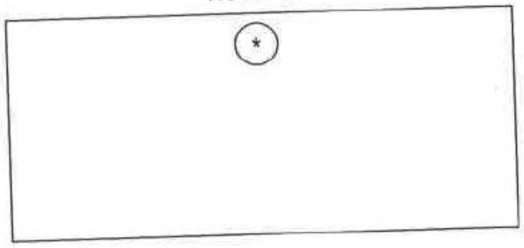
2. Now examine the TestExprTree program. This program accepts a string representing a prefix expression, then passes a StringTokenizer based on this expression to the build method for the ExprTree object, which passes it on to its recursive partner buildSub.

   We commonly write arithmetic expressions in infix form-that is, with each operator (*,/,-,+) placed between two operands (numbers) such as (1+3)*(6-4)

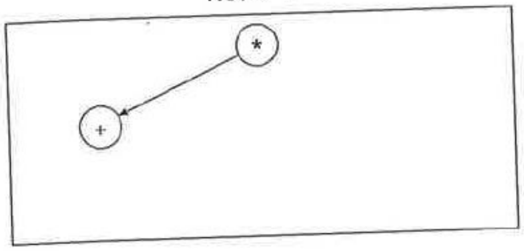
To aid the construction of our tree, we will request the expression to be entered in prefix form, so the previous expression will be typed in as follows: * + 1 3 – 6 4   (but with no spaces)

As the buildSub method recursively processes the StringTokenizer expression, the tree is constructed, one node at a time, as the following sequence of diagrams indicates:

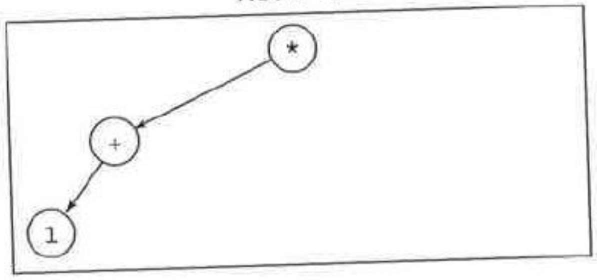
Read '*'



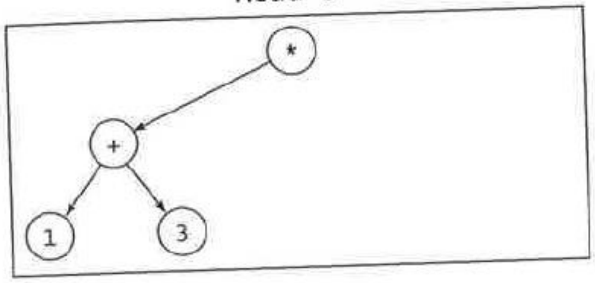
Read '+'



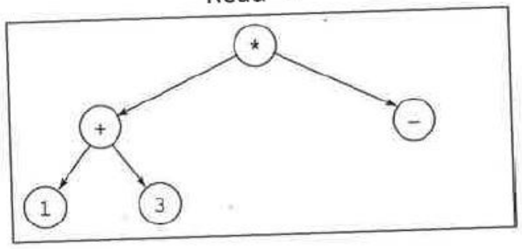
Read '1'



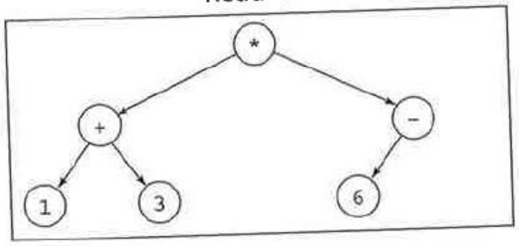
Read '3'



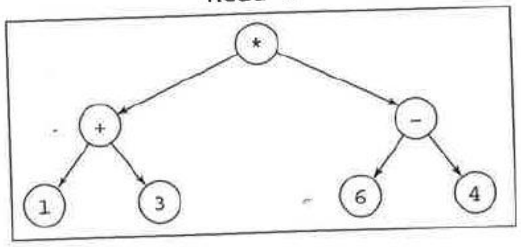
Read '–'



Read '6'



Read '4'



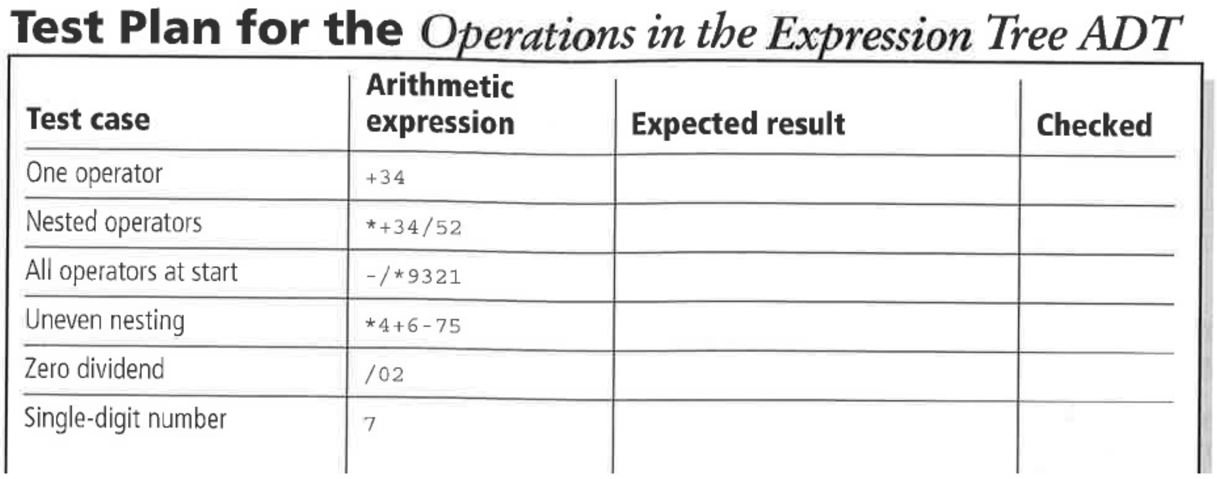
Algorithm for buildSub:

- Read the next arithmetic operator or numeric value.
- Create a node containing the operator or numeric value.
- i f the node contains an operator
    - then Recursively build the subtrees that correspond to the operator's operands.
- else the node is a leaf node. No additional recursive calls needed.

The StringTokenizer object is a convenient aid for us to extract characters from our expression string. It works like the String iterator you built a couple labs back. By passing the StringTokenizer object to the recursive build method you have a convenient way for the recursive calls to consume the expression in the proper sequence as the tree is being built.

Which traversal method is indicated by the algorithm listed above?

Write the build method and its recursive partner buildSub, then view how the tree appears in the console when it's displayed. Here are some sample expressions you can try out to see how they appear.



Test Plan for the ***Operations in the Expression Tree ADT***

Test case	Arithmetic expression	Expected result	Checked
One operator	`+34`		
Nested operators	`*+34/52`		
All operators at start	`-/*9321`		
Uneven nesting	`*4+6-75`		
Zero dividend	`/02`		
Single-digit number	`7`		

3. The `expression` method prints outputs the corresponding arithmetic expression in fully parenthesized form. Implement this method now and test it.

4. The `evaluate` method evaluates the expression that the tree represents. To write this method you'll need to convert the numeric char data into double, which you can do with the statement

```
tokenVal = (double)Character.getNumericValue(thisToken);
```