Project 7. Due to Friday, November 13, 11:59pm.

Total: 100 points

The objective of this assignment is to compare the performance of inorder traversal for two types of binary trees: standard BST and threaded BST. After implementing both classes you need to compare the running time of both traversals.

I'm not asking you to plot anything this time. Nevertheless, you need to run your traversals on a number of inputs and determine when TBST starts performing better than BST. State it in **HW7.txt**

What to submit:

1. BST.java (starter code is provided)
2. TBST.java
3. Main.java (starter code is provided)
4. HW7.txt

Part 1. Binary Search Tree (30 points)

.

In the first part you need to implement a binary search tree that supports the following methods:

1. Insert a node in BST (use recursion)
2. In-order traversal (use recursion)

Everything is pretty standard and you can find the starter code in the homework directory.

Part 2. Threaded Binary Search Tree (wiki) (60 points)

In the second part you need to implement a Threaded binary search tree that supports the following methods:

1. Insert a node in TBST
2. In-order traversal (no recursion)

What is a Threaded Binary Tree?

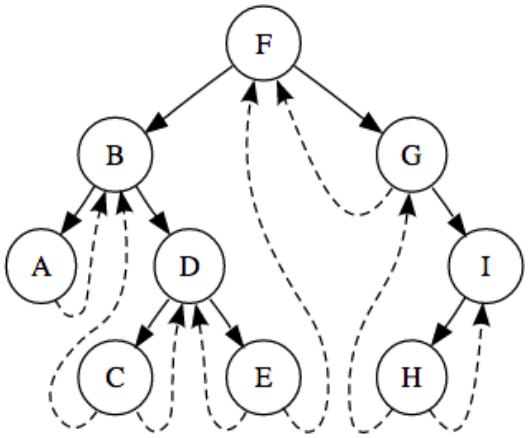
Definition:

"A binary tree is *threaded* by making all right child pointers that would normally be null point to the inorder successor of the node (**if** it exists), and all left child pointers that would normally be null point to the inorder predecessor of the node."

In Binary Search Tree, Inorder Successor of an input node can be defined as the node with the **smallest** key greater than the key of input node.

For example, in the picture below the inorder successor for node C is D (D is greater than C, and it is the smallest one). It is like tightest upper bound.

Inorder predecessor of the node has the opposite meaning: node with **largest** key smaller than the key in the input node. Inorder predecessor for C is B. It is like tightest lower bound.



Purpose of TBT:

Threaded binary tree allow **fast** traversal: given a pointer to a node in a threaded tree, it is possible to cheaply find its in-order successor (and/or predecessor).

You need to modify your BST.java class to incorporate additional pointers.
Make sure that your in-order traversal method is iterative so that you can take advantage of the threads.

Part 3. (Optional) HW2.2 Improvements

This part is optional. If you are unsatisfied with your score or think you could get more points by completing extra credit problems you can do so.

You have a chance to get some points back by improving your code for part 2 (Generic version). The points you can get back are:

1. **Compilation**. Your code **must** compile. No excuses are allowed.
2. **MasterTest** points. I will provide a .class that you can use to test your implementation.
3. **StudentTest** points. Make sure your DDL12 passes all of your own tests.
4. **BrokenCode** points. I will give you .class of our linked list implementation so you can test your own tester.

5. **10orMore** tests. If you did not have enough tests in your tester there is a change to make it better.
6. **Iterator test points**. Make sure you test every method in your iterator implementation.

You will have to submit it in a separate assignment on Vocareum called HW2_Regrade.

Part 4. Extra credit problems.

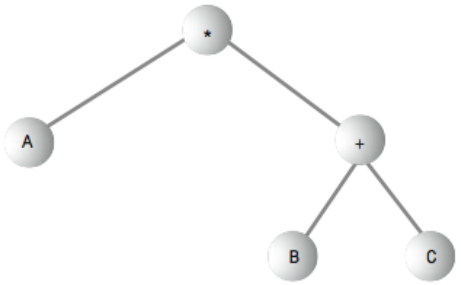
If you implemented extra credit problems, please mention it in your HW7.txt file.

1. (10 points) As you probably noticed in hw4, you stack/queue added more locations than necessary. Incorporate a proper BFS and DFS to keep track of your locations. Both searches need to be implemented in order to receive full credit. The logistics of the methods are up to you. The only thing I want you to be consistent with is the order you explore locations: left, up, right, down. The output structure should be the same as in HW4.

2. (15 points) Postfix binary tree.

Preambule (it is lengthy but easy):

A binary tree (not a binary search tree) can be used to represent an algebraic expression that involves the binary arithmetic operators +, –, /, and *. The root node holds an operator, and the other nodes hold either a variable name (like A, B, or C), or another operator. Each subtree is a valid algebraic expression.




```
Infix: A*(B+C)
Prefix: *A+BC
Postfix: ABC+*
```

For example, the binary tree shown above represents the algebraic expression A*(B+C). This is called *infix* notation; it's the notation normally used in algebra. Traversing the tree inorder will generate the correct inorder sequence A*B+C, but you'll need to insert the parentheses yourself.

Traversing the tree shown above using preorder would generate the expression

*A+BC

This is called *prefix* notation. One of the nice things about it is that parentheses are never required; the expression is unambiguous without them. Starting on the left, each operator is applied to the next two things in the expression. For the first opera tor, *, these two things are A and +BC. For the second operator, +, the two things are B and C, so this last expression is B+C in inorder notation. Inserting that into the original expression *A+BC (preorder) gives us A*(B+C) in inorder. By using different traversals of the tree, we can transform one form of the algebraic expression into another.

For the tree above, visiting the nodes with a postorder traversal would generate the expression

ABC+*

This is called *postfix* notation. It means "apply the last operator in the expression, *, to the first and second things." The first thing is A, and the second thing is BC+.

BC+ means "apply the last operator in the expression, +, to the first and second things." The first thing is B and the second thing is C, so this gives us (B+C) in infix. Inserting this in the original expression ABC+* (postfix) gives us A*(B+C) postfix.

What needs to be implemented.

You can construct a tree like above using a postfix expression as input (I will use a text file as a command line argument). Here are the steps when we encounter an operand:

1. Make a tree with one node that holds the operand.
2. Push this tree onto the stack.

Here are the steps when we encounter an operator:

1. Pop two operand trees B and C off the stack.
2. Create a new tree A with the operator in its root.
3. Attach B as the right child of A.
4. Attach C as the left child of A.
5. Push the resulting tree back on the stack.

When you're done evaluating the postfix string, you pop the one remaining item off the stack. Somewhat amazingly, this item is a complete tree depicting the algebraic expression. You can see the prefix and infix representations of the original postfix (and recover the postfix expression) by traversing the tree.

When the tree is generated, traverse in order and output each visited node. The output sequence is your answer.

Submit: Prefix.java