York University
CSE 2011 – Assignment 2
Summer 2015 Instructor: Jeff Edmonds

Family Name: _____ Given Name: _____

Student #: _____ CSE Email: _____

Family Name: _____ Given Name: _____

Student #: _____ CSE Email: _____

1) root (lines of code = 7)	11	
2) parent (lines of code = 9)	11	
3) leftChild (lines of code = 8)	11	
4) rotateR (lines of code = 15)	12	
5) first (lines of code = 4)	11	
6) next (lines of code = 13)	11	
7) set (lines of code = 11)	11	
8) insert (lines of code = 5)	11	
9) delete (lines of code = 8)	11	
Total	100	

1. You are write a set of methods for a new class called *TreeWalker* that extends my *IOTree* class. As such, a TreeWalker object consists of a binary tree that can be inputted and printed in a pretty way. Your class will will have three additional *system invariants*.

   (a) The tree is displayed on the screen as done in Figure 1.

   (b) Exactly one of its nodes is marked as the *current node*. This acts as a cursor.

   (c) The nodes of the tree will be order so that it is a *binary search tree*.

   Each constructor for your class must ensure that it establishes each of these invariants. Each of your methods must ensure that it maintains them, i.e. if these system invariants are true before it is executed then they will be true after. Your code does not attempt to maintain it, but the user may wish to

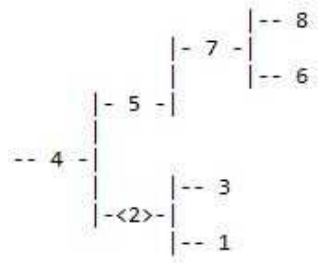

Figure 1: A tree created by parsing the string "((1 2 3) 4 (null 5 (6 7 8)))". You have to rotate your head to see it. As the user walks around, the current node is indicated with < .. >.

maintain the additional system invariant that the tree is an *AVL tree*. Your class will provide the tools for doing this.

I provide the class MainGUI which consists of an *event* processing loop. It waits for one of the following commands from the user and then calls the appropriate method from your TreeWalker class. Note how the system invariants for the class become the loop invariants for this loop.

Each of the methods here takes the current position as the input and returns the new current position. They **do not** change the current flag in these nodes and do not reprint the tree. This is done in GUI with the following lines of code.

```
tree.notCurrent(current);
current = tree.leftChild(current);
tree.makeCurrent(current);
write.setText(tree.PrettyPrint());
```

To help you, I have also given you the code for *deleteNoRight*. It contains may of the technques you will need.

root,parent,leftChild,rightChild ($\cdot, \uparrow, \swarrow, \searrow$): Moves the current node to its root, parent, left child, or right child as the case may be.

Levels: When there is not such a root, parent, left child, or right child to move to:

(a) First simply have the current node not move. No error needs to be given.

(b) If you have more time, automatically create the node needed. For example, when at the root, moving up creates a new root. The previous root will be the new root's left child. As another example, if the tree is the empty tree (created with "null"), then it is ok for the methods parent, leftChild, and rightChild to crash, but the method root should create a root node with the value 100.

(c) If you have managed to write it, use the method *set* to set its value.

rotateL, rotateR: Suppose the current node is that with value 10 in the left tree of Figure 2. Suppose the user's command is rotateL (clockwise). Your code should change the tree to that on the right side of the figure with the current node being that with the value 5. If the user's command is then to rotateR (counter clockwise), then the tree changes back. The purpose of this is that the [..5] subtree moves up one level while the [10..] subtree moves one down. Hopefully, this helps to balance the tree.

first,last,next,previous ($\Leftarrow, \Rightarrow, \leftarrow, \rightarrow$): These commands moves the current node to the first, last, next or previous node in the sorted order by value (assuming that the tree is a binary search tree). This is the same as tree's *infix Traversal order*. The first node is found by starting at the root of

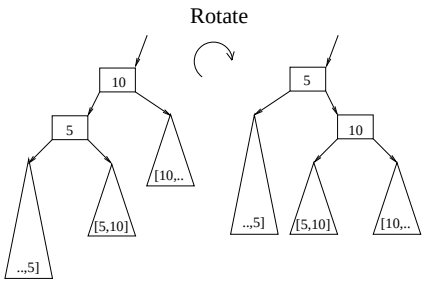

Figure 2: $rotateL(10)$ changes the left tree to right tree and $rotateR(5)$ changes is back.

the tree and going left as far as you can go. The next in this order is as follows. If your current node has a right child, then go to this right child and then from there follow the path left left left as far as you can go. If your current node does not have a right child, travel *up and left* as far as you can and then *up and right*. Said another way follow the path up parent parent parent. Stop the first time that the node you came from is the left node of where you currently are. If this up and left process goes past the root, then where you started already was the last node in this order. In this case, automatically create the node needed with an appropriate binary search value. Hint: use getLeft or getRight already written to create this node.

set: The current code is automatically given a new value that will help move the tree towards being a binary search tree, i.e. to an integer value that is between its previous and next node's value in the infix traversal order. Hint: Use the already written methods *next*, *previous*, $first$, and *last*. (This routine is not called in MainGUI.)

Insert After: A new node is created after the current node according to the tree's *infix Traversal order*, i.e. go right and then left left left and put the new node there. Shorter code uses the routines that you have already written. If the current node does not have a right child then the method *rightChild* already written will create this node. Otherwise, the method *next* takes you right left left left. Then $leftChild$ will create the node.

Delete: Delete the current node. If the current node does not have a right child, then it's parent adopts its left child. If no left child, it's parent adopts its right. The adopted child becomes the current node (unless it is null then the parent does.) If the current node two children, then the *next* node is right left left left. This node does not have a left child. Hence, it can be deleted by having its parent adopt its right child. Before deleting *next*, its *element* contents is moved to the current node. This remains the current node. To help you, I give you the code for $deleteNoRight$. You write $deleteNoLeft$ symmetrically. Then use these in *delete*.

Have Fun: