README
- COS226 Final Project -
Maxwell Morin - Dylan Lougee

To use, just run the Main.java file. This will open a new GUI window containing our program.

In the GUI window, there are multiple screens displaying information. The main windows with the black background
is where the ScapeGoat Tree will be drawn out.

On the lower left, this is the information window that displays the current step, such as 'Added node', or 'Removed node', etc...

In the lower section of the main window is where you can either Add, Remove, or Find, within the ScapeGoat Tree.
To use any of the three functions, click inside the corrosponding box and enter any value, and press enter. The
ScapeGoat tree will then perform that said function and display it in the tree and provide a snypase of what
occured in the information window.

On the lower right, there is a window with three buttons next to it that can be used at any time the tree is
populated. To use, just click any of the three buttons (IN ORDER, PRE-ORDER, or POST-ORDER) and it will display
the said function in the window to the right.

There is a HELP function built in that will open a window that displays HOW-TO information about the ScapeGoat
Tree application. To open the HELP window, you have to click any of the three insertion boxes and then press
F11. If you are not clicked in an insertion window then there is no key monitoring happening and the window
will not receive the command to open.

The reason why we choose a ScapeGoat Tree is because it is a tree we have not covered, and because it is a well
balanced tree. It is based off of a Binary Search Tree and offers O(log n) for insertion, deletion, and search.
Here are proofs deminstrating that. These examples derrived from
http://en.wikipedia.org/wiki/Scapegoat_tree

INSERTION COST:
Define the Imbalance of a node v to be the absolute value of the difference in size between its left node and
right node minus 1, or 0, whichever is greater. In other words:

I(v) = max(|left(v) - right(v)| - 1, 0) \

Immediately after rebuilding a subtree rooted at v, I(v) = 0.

Lemma: Immediately before rebuilding the subtree rooted at v,
I(v) = \Omega (|v|) \
(\Omega \ is Big O Notation.)

Proof of lemma:

Let v_0 be the root of a subtree immediately after rebuilding. h(v_0) = log(|v_0| + 1) \ .
If there are \Omega (|v_0|) degenerate insertions (that is, where each inserted node increases the height by 1),
then
I(v) = \Omega (|v_0|) \ ,
h(v) = h(v_0) + \Omega (|v_0|) \ and
log(|v|) \le log(|v_0| + 1) + 1 \ .

Since I(v) = \Omega (|v|) before rebuilding, there were \Omega (|v|) insertions into the subtree rooted at v that
did not result in rebuilding. Each of these insertions can be performed in O(log n) time. The final insertion that
causes rebuilding costs O(|v|). Using aggregate analysis it becomes clear that the amortized cost of an insertion
is O(log n):

{\Omega (|v|) O(\log{n}) + O(|v|) \over \Omega (|v|)} = O(\log{n}) \

DELETION COST:
Suppose the scapegoat tree has n elements and has just been rebuilt (in other words, it is a complete binary tree).
At most n/2 - 1 deletions can be performed before the tree must be rebuilt.
Each of these deletions take O(\log{n}) time (the amount of time to search for the element and flag it as deleted).
The n/2 deletion causes the tree to be rebuilt and takes O(\log{n}) + O(n) (or just O(n)) time.
Using aggregate analysis it becomes clear that the amortized cost of a deletion is O(\log{n}):

{\sum_{1}^{{n \over 2}} O(\log{n}) + O(n) \over {n \over 2}} = {{n \over 2}O(\log{n}) + O(n) \over {n \over 2}} = O(\log{n}) \

SEARCH COST:
Lookup is not modified from a standard binary search tree, and has a worst-case time of O(log n).
This is in contrast to splay trees which have a worst-case time of O(n).
The reduced node memory overhead compared to other self-balancing binary search trees can further improve locality of reference and caching.