York University
CSE 2011 – Assignment 1
Summer 2015 Instructor: Jeff Edmonds

Family Name: _____ Given Name: _____

Student #: _____ CSE Email: _____

Family Name: _____ Given Name: _____

Student #: _____ CSE Email: _____

MaxTime 1) Try	0	
2) Following Pointers	$4 \times 6 = 24$	
3) System Invariants	$2 \times 6 = 12$	
4) Binary Search Tree	$4 \times 6 = 24$	
5) Rotations	12	
6) Rebalancing	$2 \times 6 = 12$	
7) Height of AVL Tree	16	
Total	100	

Before answering these questions (or coding) do the following.

- Read and understand Jeff's three documents:
  http://www.eecs.yorku.ca/∼jeff/courses/3101/ass/steps0.pdf
  http://www.eecs.yorku.ca/∼jeff/courses/3101/ass/steps1.pdf
  http://www.eecs.yorku.ca/∼jeff/courses/3101/ass/steps2.pdf
- Read and understand Jeff's *Binary Tree* document. There are lots of good ideas in there.
- Read and understand the requirements of Assignment 2.
- Run Jeff's MainGUI code that calls the TreeWalker methods that you will have to write in Assignment 2.
- Recall that the Fibonacci sequence is defined with $Fib(i) = Fib(i-1) + Fib(i-2)$, $Fib(0) = 0$, $Fib(1) = 1$, $Fib(2) = 1$, $Fib(3) = 2$, $Fib(4) = 3$, and $Fib(i) \approx 1.618^i$.
- Answer the following questions:

1. Review the section Parsing a String with a Grammar: Try on your own to figure out and draw the tree generated by each of the following. Then check it by running the code.
   IOTree(" (((1 2 null) 4 null) 5 ((null 6 7) 8 (9 10 11)))");
   IOTree(" (x+y*7+4)*z"); (Nothing to hand in).

2. Following Pointers

   (a) Start with the tree " ((1 2 3) 4 (null 5 (6 7 8))) ". (See Fig1.a in the Jeff's *Binary Tree* document.) Suppose that all the code I have given you has nothing *private*. For each of the following, give me method free code. The trick is to follow the line of pointer. See if you can do each in one line.

      i. Change the 2 to a 9.

      ii. Allocate space for a new BTNode. Point the left child pointer of node 3 at it.

      iii. Point the right child pointer of node 3 at node 7.

   (b) Redo the above code that points the right child pointer of node 3 at node 7. This time, however, respect the fact that the fields are all private and use the methods from IOTree (and those inherited from LinkedBinaryTree). Only use the methods from BTNode if absolutely needed. Use as many or as few lines of code as you want. Put the castings in when needed. For every object in the path following the pointers state both its type and what it is in the figure. Try dropping your answer into Eclipse to see if it compiles.

   (c) Draw a picture of the data structure after these three changes from part (a) have been made. If you were to call tree.PrettyPrint(); on this new data structure, what would it print. Why? Show how you would use the command tree = new IOTree(toParse) to produce figure drawn.

   (d) Suppose that the last change was not to point the right child pointer of node 3 pointing at node 7, but was instead to point it at node 2 = node 9.
   Again draw a picture of the data structure.
   Again what would tree.PrettyPrint(); output on this new data structure? Why?

3. System Invariants:

   (a) IOTree implements binary trees. Its *system invariants* would strongly disallow an operation that points the right child pointer of node 3 at node 7. If all we did to this class was to lighten up this restriction, what *Abstract Data Type* would this give you? What restriction would this implementation of this data type oppose on it.

   (b) What *system invariant* restriction would you need to add to IOTree to get the *Abstract Data Type* called a *Stack*?
   In class we said that one of the four operation add/remove a node from head/null side of a linked list took too much time. Which? Why? Would we have that problem here?

4. Searching a Binary Search Tree:
   Read www.eecs.yorku.ca/~jeff/courses/3101/ass/steps1.pdf
   Suppose your current node is the root of a binary search tree. Suppose you do not know the key at any node other than your current node. Suppose you are trying to find a particular key value k, for example $k = 6$. Design an algorithm for walking down the tree with the commands $\swarrow$ and $\searrow$, searching for this key. How do you decide which direction to go? How do you know you will find the node with key k if it exists. At what point do you know that such a node does not exist.
   Hint: Establish and maintain the loop invariant "If k is anywhere in the tree, then it is in the subtree rooted at your current node".

   (a) Establishing the Loop Invariant $\langle pre{-}cond \rangle$ & $code_{pre\text{-}loop} \Rightarrow \langle loop{-}invariant \rangle$:
   The input to the algorithm is a binary search tree and a key k. What action does your algorithm first take? From these alone prove that the loop invariant is then true.

(b) Recall that the contra positive of $A \Rightarrow B$ is $\neg B \Rightarrow \neg A$. What is the contra positive of the loop invariant? (Nothing to hand in).

(c) Obtaining the postcondition $\langle loop{-}invariant \rangle$ & $\langle exit{-}cond \rangle$ & $code_{post\text{-}loop} \Rightarrow \langle post{-}cond \rangle$: There are two good reasons to exit.

- k equals the key in current node, i.e. $k = current'.element().x$.
- $current = null$.

For each of these prove how the loop invariant together with the exit condition ensures that the requirements of the problem have been met.

(d) Maintaining the Loop Invariant $\langle loop{-}invariant' \rangle$ & *not* $\langle exit{-}cond \rangle$ & $code_{loop} \Rightarrow \langle loop{-}invariant'' \rangle$:
Fly in from Mars. Let $current'$ denote your current node. All you know is that the loop invariant is true and the exit condition has not yet been met. In order to make progress, your algorithm must take a step $\swarrow$ or $\searrow$ down the tree. How do you decide which direction to go? Let $current''$ denote the resulting current node. Prove how the fact that the loop invariant is true with respect to $current'$, together with why you chose your direction, ensures that the loop invariant will be true with respect to $current''$.

(e) What is the measure of progress of your algorithm? Prove that each iteration you this measure decreases by at least one. What is the maximum number of iterations that this algorithm can take before the exit condition is met? What is the running time of your algorithm as some feature of the tree given as input?

(f) Convince yourself that you have just proved that your algorithm always finds the key k if its in the tree and correctly reports that it is not there if it is not. (Nothing to hand in).

(g) Try: Read the assignment 1 explanation of how to move the current node to the next node in the sorted order by value (assuming that the tree is a binary search tree). Play around with examples until you understand this and convince yourself that it works. (Nothing to hand in).

5. Rotations: For each cut and paste the string representing the following tree into MainGUI and play with it.
(null 1 (null 2 (null 3 (null 4 (null 5 (null 6 7))))))
How do you use $rotateL$ and $rotateR$ operations to make this tree completely balanced? If you are feeling more bold do this one.

(null 1 (null 2 (null 3 (null 4 (null 5 (null 6 (null 7 (null 8 (null 9 (null 10 (null 11 (null 12 (null 13 (null 14 15)))))))))))))

6. Rebalancing an AVL Tree After an Insert:

Recall the definition of an *AVL Tree*. For each node v, define $height(v)$ to be the height of the subtree rooted at v and define $balanceFactor(v) = height(leftChild(v)) - height(leftChild(v))$ A tree is said to be an *AVL Tree* if and only if for **each** node v in the tree, $balanceFactor(v) \in \{-1, 0, 1\}$.

Recall from Assignment 2 that the operation $rotateL(10)$ changes the left tree in Figure 2 to the right tree and $rotateR(5)$ changes it back. The [..5] subtree moves up one level while the [10..] subtree moves one down. Hopefully, this helps to balance the tree. A potential problem is that the [5..10] subtree does not change its height.

Consider some AVL tree. Label each node v with $balanceFactor(v)$. Verify that it is an AVL Tree. Consider inserting a new node as a leaf. Denote this node ℓ. Note how doing so may change $balanceFactor(v)$ by at most one along the path from ℓ to the root. Denote by z the first node along this path that became too unbalanced, i.e. lowest node for which $balanceFactor(z) \in \{-2, +2\}$. By symmetry assume 2. Denote by y the child of z with the highest subtree. Assume $balanceFactor(y) = \{-1, +1\}$. (Case 0 is similar). Denote by z the child of y with the highest subtree. There are lots of cases, but the following two examples are representative.
- For each cut and paste the string representing the tree into MainGUI and play with it. Find the

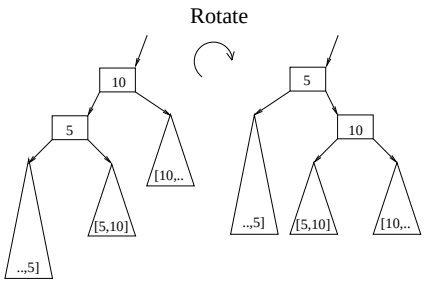

Figure 1: $rotateL(10)$ changes the left tree to right tree and $rotateR(5)$ changes is back.

minimum number of $rotateL(v)$ or $rotateR(v)$ operations needed to make this tree into an AVL tree again.
- Draw on paper (or snip from screen) the tree as it goes through these steps. Label the nodes z, y, and x.
- Explain why in this general case the tree will be an AVL tree after these operations.

(a) Consider the tree " (((0 1 null) 2 3) 6 7) " formed by inserting the value 0 as a leaf into the tree " ((1 2 3) 6 7) ".

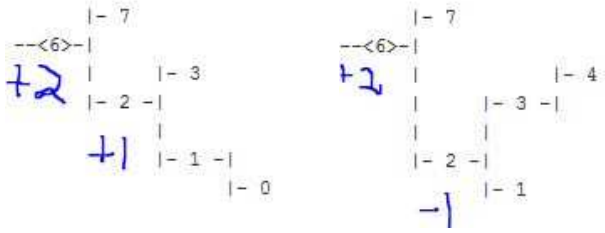

Figure 2: (a) Tree for this question. (b) Tree for next question.

(b) Consider the tree " ((1 2 (null 3 4)) 6 7) " formed by inserting the value 4 as a leaf into the tree " ((1 2 3) 6 7) ".

7. Height of a AVL Tree:
If you want to build a tree with height h that has the maximum number of nodes $n = N(h)$, then your tree should be perfectly balanced. This is built recursively by having a root whose left and right subtrees are trees of height $h-1$ with maximum number of nodes $N(h-1)$.
If you care about this recursive pattern instead of the tree being a binary search tree, then the string producing such a tree will be as follows.

- height $h = 0$ will be "null" with $N(0) = 0$.
- height $h = 1$ will be "1" with $N(1) = 2 \times 0 + 1 = 1$.
- height $h = 2$ will be " (1 2 1) " with $N(2) = 2 \times 1 + 1 = 3$.
- height $h = 3$ will be " ((1 2 1) 3 (1 2 1)) " with $N(3) = 2 \times 3 + 1 = 7$.
- height $h = 4$ will be " (((1 2 1) 3 (1 2 1)) 4 ((1 2 1) 3 (1 2 1))) " with $N(4) = 2 \times 7 + 1 = 15$.

(View this tree by plugging it into the MainGUI program.)
The recurrence relation on this number of nodes is
$N(h) = 2 \times N(h-1) + 1$ and $N(0) = 0$.

It evaluates to $N(h) = 2^h - 1$.
Turning this around gives that if a tree has n nodes then its minimum height is $h \approx \log(n)$.

If you want to build a tree with height h that has the minimum number of nodes $n = N(h)$, then your tree should be as single path. This is built recursively by having a root whose left subtree is a tree of height $h-1$ with minimum number of nodes $N(h-1)$ and whose right subtree is empty. The string producing such a tree will be as follows.

- height $h = 0$ will be "null" with $N(0) = 0$.
- height $h = 1$ will be "1" with $N(1) = 0 + 1 + 0 = 1$.
- height $h = 2$ will be " (1 2 null) " with $N(2) = 1 + 1 + 0 = 2$.
- height $h = 3$ will be " ((1 2 null) 3 null) " with $N(3) = 2 + 1 + 0 = 3$.
- height $h = 4$ will be " (((1 2 null) 3 null) 4 null) " with $N(4) = 3 + 1 + 0 = 4$.

(View this tree by plugging it into the mainGUI program)
The recurrence relation on this number of nodes is $N(h) = N(h-1) + 1$ and $N(0) = 0$.
It evaluates to $N(h) = h$.
Turning this around gives that if a tree has n nodes then its maximum height is $h = n$.

Repeat all of these steps, but now when you want to build a tree with height h that has as few nodes as possible while still being an AVL tree.