Name: ____ **Start work early.**

You should submit

1. a hard-copy of pages 1–8 of this assignment with your answers filled in, and
2. an electronic package that contains the source files for your work on the programming questions, by following the submission procedure described in class.

You are welcome to use any inanimate resources (e.g., books, Web sites, publicly available code) to help you with your work. However, *all such help must be clearly noted* in your submissions. Further, no matter what you use, *you must be able to explain, in detail, how it works.* (You may be called upon to explain your homework in person.) Refer to the class policy for details, and ask for clarifications if you are unsure if something is allowed.

Questions marked with a $\star$ are optional but you are strongly encouraged to answer them for extra credit.

1. (1 pt.) Write your name in the space provided above.

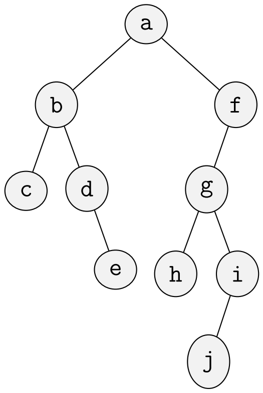
2. (10 pts.) Do there exist functions f and g such that f is $O(g)$ and g is $O(f)$ **and** such that there are no constants c_1 and c_2 for which $f(x) = c_1 g(x) + c_2$? If so, provide examples of f and g **and** explain in detail why they satisfy the required conditions. If not, explain why such f and g do not exist.

3. (30 pts.) Let us represent the empty binary tree by $\perp$ and a nonempty binary tree with root label n, left subtree l, and right subtree r by the triple (n, l, r). Using this notation, we may define the following functions on binary trees:

$$
\begin{aligned}
f_1(t) &= \begin{cases} (n, r, l) & \text{if } t = (n, l, r) \\ \perp & \text{otherwise} \end{cases} \\
f_2(t) &= \begin{cases} (n, f_2(r), f_2(l)) & \text{if } t = (n, l, r) \\ \perp & \text{otherwise} \end{cases} \\
f_3(t) &= \begin{cases} (n, f_3(r), l) & \text{if } t = (n, l, r) \\ \perp & \text{otherwise} \end{cases} \\
f_4(t) &= \begin{cases} (n, r, f_4(l)) & \text{if } t = (n, l, r) \\ \perp & \text{otherwise} \end{cases} \\
f_5(t) &= \begin{cases} (n_1, l_1, (n, r_1, r)) & \text{if } t = (n, (n_1, l_1, r_1), r) \\ \perp & \text{otherwise} \end{cases} \\
f_6(t) &= \begin{cases} (n_1, (n, l, l_1), r_1) & \text{if } t = (n, l, (n_1, l_1, r_1)) \\ \perp & \text{otherwise} \end{cases} \\
f_7(t) &= \begin{cases} (n_2, (n_1, l_1, l_2), (n, r_2, r)) & \text{if } t = (n, (n_1, l_1, (n_2, l_2, r_2)), r) \\ \perp & \text{otherwise} \end{cases} \\
f_8(t) &= \begin{cases} (n_2, (n, l, l_2), (n_1, r_2, r_1)) & \text{if } t = (n, l, (n_1, (n_2, l_2, r_2), r_1)) \\ \perp & \text{otherwise} \end{cases}
\end{aligned}
$$

The following function g_1 takes a binary tree t and another function f as arguments:

$$
g_1(t, f) = \begin{cases} f((n, g_1(l, f), g_1(r, f))) & \text{if } t = (n, l, r) \\ \perp & \text{otherwise} \end{cases}
$$



For the binary tree t_1 depicted above, depict the 10 trees $f_1(t_1), \ldots, f_8(t_1)$, $g_1(t_1, f_3)$, and $f_2(f_3(t_1))$ using the usual representation. Label your trees clearly.

[additional space for answering the earlier question]

[additional space for answering the earlier question]

4. (50 pts.) For each of the following, indicate whether the claimed identity is true. The free variables, such as t and n, are universally quantified. *Justify your answers.*

   (a) $f_1(f_1(t)) \overset{?}{=} t$

   (b) $f_2(f_2(t)) \overset{?}{=} t$

   (c) $f_2((n, l, r)) \overset{?}{=} f_1((n, f_1(l), f_1(r)))$

   (d) $f_2((n, l, r)) \overset{?}{=} f_1((n, f_2(l), f_2(r)))$

   (e) $f_2(t) \overset{?}{=} f_3(f_4(t))$

   (f) $f_3(f_4(f_3(f_4(t)))) \overset{?}{=} t$

   (g) $f_2(t) \overset{?}{=} g_1(t, f_1)$

   (h) $f_5(f_6(t)) \overset{?}{=} t$

   (i) $f_5(f_5(t)) \overset{?}{=} f_7(t)$

   (j) $g_1((g_1(t, f_5)), f_6) \overset{?}{=} t$

[additional space for answering the earlier question]

[additional space for answering the earlier question]

5. (9 pts.) Refer to the *first-child/next-sibling* method of implementing trees.[1] Consider an analogous *last-child/previous-sibling* method in which each node stores a link to its last (rightmost, in sibling order) child and another link to its previous (or left) sibling.

   Depict the implementation of the tree of Figure 18.1 in the textbook[2] using this method (by analogy with Figure 18.3 in the textbook).

[1] Mark Allen Weiss, *Data Structures and Problem Solving Using Java*, 4th edition (Addison-Wesley, 2010), § 18.1.2, p. 653.

[2] *Idem*, p.652.

6. (50 pts.) Interfaces Tree and TreeNode are described below using the tree depicted in Figure 18.1 in the textbook[3] as a running example. Additional examples and definitions may be found on pages 652–653 of the textbook. A more detailed description appears in in the files Tree.java and TreeNode.java that are in the *Sample Code* section of the class Web site. Implement these interfaces using the last-child/previous-sibling method of Question 5.

Interface Tree:

getRoot returns the root of the tree. For our example tree, it returns the node containing a.

size returns the number of nodes in the tree, 11 in our example.

height returns the length of the path from the root to the deepest leaf, which is 3 in our example (the path to K).

getPreOrder returns a list of elements in preorder: [A, B, F, G, C, D, H, E, I, J, K] in our example.

getPostOrder returns a list of elements in postorder: [F, G, B, C, H, D, I, K, J, E, A] in our example.

makeEmpty removes all nodes from the tree.

isEmpty returns true iff the tree contains no nodes.

height takes a node as input and returns its height in the tree. For nodes B and F of our example, this method returns 1 and 0, respectively.

depth takes a node as input and returns its depth in the tree. For nodes A and K of our example, this method returns 0 and 3, respectively.

You should also create a public constructor for the tree that takes a TreeNode that is to be the root of the tree.

Interface TreeNode:

getElement returns the element contained in the node; A for the root of our example tree.

setElement sets the element of this node to the non-null element given as argument.

getLastChild returns the rightmost child of this node.

getChildren returns this node's children represented as a list.

addChild takes a non-null TreeNode and adds it as the rightmost child of this node.

size returns the number of nodes in this node's subtree (including this node).

height returns the length of the path from this node to its subtree's deepest leaf.

getPreOrder is similar to getPreOrder in Tree, but it returns only this node's subtree in preorder. In our example, invoking this method on node E yields [E, I, J, K].

getPostOrder returns this node's subtree in postorder. In our example, invoking this method on node E yields [I, K, J, E].

[3] *Idem.*

toString This method is not actually specified in `TreeNode`, since every Object in Java has a `toString` method specified in the class Object. You should redefine `toString` in `MyTreeNode` to display the node's element.

In addition, you should create a constructor that takes an element and sets it as the `TreeNode`'s element.

TestMyTree A sample JUnit[4] test `TestMyTree` appears in the *Sample Code* section of the class Web site. JUnit testing is not required, but is recommended. It is your responsibility to make sure that each method of your tree produces the correct results in all cases, not just on the sample test program.

Submission Submit a single *jar* file, named using the convention

`cos226-hw01-`*`lastname-firstname-mynum`*`.jar`

where *lastname* and *firstname* are replaced with the appropriate values and where *mynum* is an arbitrary 4-digit string of your choice. This jar file should contain the following:

`README:` a file that describes the files in the submission and how to use them.

`MyTree.java:` your implementation that extends `Tree.java`.

`MyTreeNode.java:` your concrete implementation that extends `TreeNode.java`.

`TestMyTree.java:` your test program that illustrates the correct working of your implementation.

`Tree.java:` the Tree interface specification.

`TreeNode.java` the TreeNode interface specification.

`Makefile:` a configuration file that ensures that running the standard *make* command results in the complete build of your implementation.

Important: Check very carefully that your submission contains *only source files*, not object files such as `.class` files. You will lose many points if this requirement is not met! As a general guide, a source file is something you or someone else generated by typing; these are the only files that should be part of your submission. If in doubt, ask for clarifications.

7. (40 pts.) Implement the functions $f_1, \ldots, f_8$ of Question 3. That is, define an interface `F1To8` that includes methods corresponding to these functions, and implement that interface. Include both the file defining your interface and the files used by your implementation with the rest of your submission. Be sure to describe them appropriately in the README file.

8. (10 pts.) Implement the function g_1 of Question 3. That is, add a method corresponding to this function to the `F1To8` interface above, implement it, and include the necessary files in your submission. [Hint: Functors.[5]]

[4] `http://www.junit.org/`

[5] Weiss, *op. cit.*, § 4.8, p. 157.

9. (40 pts.) $\star$ Write a program that reads a positive integer n from standard input and writes to standard output a representation of all distinct rooted, ordered, labeled trees on the set $\{1, 2, \ldots, n\}$ of vertices. Package and submit your implementation as in earlier questions, including the appropriate instructions in the README file.

For the output, use the following linear textual representation $L(t)$ of a tree t:

- If t is empty then $L(t) =$ `()`.
- If t has root n and children, in sibling order, $C = (c_1, c_2, \ldots, c_k)$, then

$$L(t) = \texttt{(n, (}L(c_1)\texttt{, }L(c_2)\texttt{, ..., }L(c_k)\texttt{))}$$

where, $L(c_i)$ denotes the linear textual representation of the subtree rooted at child c_i. There is a single space after each comma in the representation.

The output should contain the representation of one tree on each line and should be sorted by lexicographic order on the linear representations when treated as strings. The output should contain nothing else (such as spurious newline characters, prompts, or informative messages). Sample inputs and outputs for $n = 1, 2, 3$ appear below.

Input: 1
Output:

```
(1, ())
```

Input: 2
Output:

```
(1, ((2, ())))
(2, ((1, ())))
```

Input: 3
Output:

```
(1, ((2, ((3, ())))))
(1, ((2, ()), (3, ())))
(1, ((3, ((2, ())))))
(1, ((3, ()), (2, ())))
(2, ((1, ((3, ())))))
(2, ((1, ()), (3, ())))
(2, ((3, ((1, ())))))
(2, ((3, ()), (1, ())))
(3, ((1, ((2, ())))))
(3, ((1, ()), (2, ())))
(3, ((2, ((1, ())))))
(3, ((2, ()), (1, ())))
```