Name: _____

Please note that much of the programming portion of this assignment is devoted to getting familiar with the class accounts and submission procedure; thus, it is much simpler than the programming assignments that will follow. You should submit

1. a hard-copy of pages 1–11 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 on the class Web site.

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 ⋆ are optional but you are encouraged to answer them for extra credit.

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

2. (1 pt.) Read the material on the class Web site. Sign your name here to indicate that you have understood that material: _____

3. (1 pt.) Change the passwords on your Gandalf (Unix) and PC cluster accounts. Fill in the following information. (Change your passwords and note *yes* in the last column. Do *not* write your old or new passwords here.)

Account	User Name	Password Changed? (yes/no)
Gandalf	_____	_____
PC cluster	_____	_____

4. (1 pt.) Post a message on the class newsgroup.

5. (1 pt.) Package and submit your solutions to the programming questions. *After* uploading your jar file to the FTP server, complete the following:
   File name: _____ Size, in bytes: _____

6. (5 pts.) Do there exist non-identical functions f and g such that f is $O(g)$ and g is $O(f)$? If so, provide examples of f and g and indicate why they satisfy the required conditions. If not, explain why such f and g do not exist.

7. (20 pts.) Consider the loops L_1, L_2, and L_3 outlined below, where `A` is an array of one million elements:

```
L1:  for(int i = 0; i < 1000000000; i++) x = i/2;
L2:  for(int i = 0; i < 1000000000; i++) A[i / 1000] = i/2;
L3:  for(int i = 0; i < 1000000000; i++) A[i % 1000] = i/2;
```

   (a) Estimate the running times of L_1, L_2, and L_3. The estimates should be in units of real time, such as milliseconds. *Explain in detail* how you arrive at these estimates. Clearly state all assumptions and values of relevant parameters such as the programming environment, CPU type, speed, and cache structure, and main-memory bandwidth. [Hint: You may use the class discussion on this topic as a template.]

(b) Implement L_1, L_2, and L_3 in the environment you describe above and measure the running times experimentally. Describe your experimental setup briefly. Compare your experimental results to your estimates. Explain any significant differences.

8. (30 pts.) Let us represent the empty binary tree by $\emptyset$ and a nonempty binary tree with root 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:

$$f_1(t) = \begin{cases} (n, r, l) & \text{if } t = (n, l, r) \\ \emptyset & \text{otherwise} \end{cases}$$

$$f_2(t) = \begin{cases} (n, f_2(r), f_2(l)) & \text{if } t = (n, l, r) \\ \emptyset & \text{otherwise} \end{cases}$$

$$f_3(t) = \begin{cases} (n, f_3(r), l) & \text{if } t = (n, l, r) \\ \emptyset & \text{otherwise} \end{cases}$$

$$f_4(t) = \begin{cases} (n, r, f_4(l)) & \text{if } t = (n, l, r) \\ \emptyset & \text{otherwise} \end{cases}$$

$$f_5(t) = \begin{cases} (n_1, (n, l, l_1), r_1) & \text{if } t = (n, l, (n_1, l_1, r_1)) \\ \emptyset & \text{otherwise} \end{cases}$$

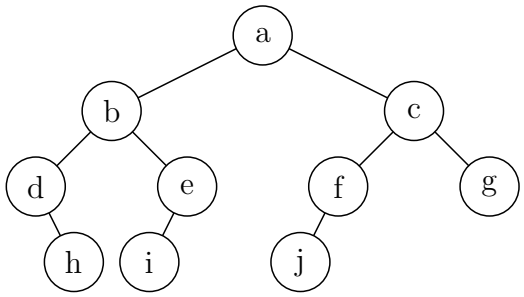
$$f_6(t) = \begin{cases} (n_1, l_1, (n, r_1, r)) & \text{if } t = (n, (n_1, l_1, r_1), r) \\ \emptyset & \text{otherwise} \end{cases}$$

$$f_7(t) = \begin{cases} (n_2, (n_1, l_1, l_2), (n, r_2, r)) & \text{if } t = (n, (n_1, l_1, (n_2, l_2, r_2)), r) \\ \emptyset & \text{otherwise} \end{cases}$$

$$f_8(t) = \begin{cases} (n_2, (n, l, l_2), (n_1, r_2, r_1)) & \text{if } t = (n, l, (n_1, (n_2, l_2, r_2), r_1)) \\ \emptyset & \text{otherwise} \end{cases}$$

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

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



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]

9. (40 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) $f_1(f_1(t)) \stackrel{?}{=} t$

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

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

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

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

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

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

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

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

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

[additional space for answering the earlier question]

[additional space for answering the earlier question]

10. (20 pts.) $\star$ Consider the set S of finite strings composed of the characters a and b, such as $abaab$, and $aaaaaa$. Let $\cdot$ denote the concatenation operator on such strings. Thus $a \cdot ab = aab$ and $abaab \cdot aaaaaa = abaabaaaaaa$. Does S contain strings x and y such that $x \cdot a \cdot y = y \cdot b \cdot x$? Justify your answer.

11. (50 pts.) Interfaces `Tree` and `TreeNode` are described below using the tree depicted in Figure 18.3 in the textbook[1] as a running example. Additional examples and definitions may be found on pages 600–601 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 first-child/next-sibling method described in the textbook.[2].

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.

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.

getFirstChild returns the leftmost 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]`.

[1] Mark Allen Weiss, *Data Structures and Problem Solving Using Java*, 3rd edition (Addison-Wesley, 2006), p.598.

[2] *Idem*, p. 597.

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 simple JUnit[3] test `TestMyTree` has been included as an example. At a *minimum* your tree should pass these tests. While it is not required that you use JUnit testing, it is recommended. It is your responsibility to make sure that each method of your tree produces the correct results. The test program given here is only a *basic example* to test your tree's implementation.

Submission Files that should be submitted for this question include:

`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.

12. (40 pts.) Implement the functions $f_1, \ldots, f_8$ of Question 8. 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.

13. (10 pts.) Implement the function g_1 of Question 8. 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.[4]]

[3] `http://www.junit.org/`
[4] Weiss, *op. cit.*, Section 4.8, p. 137.

14. (20 pts.) ⋆ Write a program that reads a positive integer n from standard input and writes to standard output a representation of of all distinct 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 $t = \emptyset$ (the empty tree), then $L(t) = \{\}$.
    - If t has root n and children $C = \{c_1, c_2, \ldots, c_k\}$, let $d_1, d_2, \ldots, d_k$ be the listing of the children C in sorted order. Then

      $$L(t) = \texttt{(n, } \{L(d_1)\texttt{, } L(d_2)\texttt{, } \ldots\texttt{, } L(d_k)\}\texttt{)}$$

      where, by slight abuse of notation, $L(d_i)$ denotes the linear textual representation of the subtree rooted at child d_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 viewed as strings. The output should contain nothing else (such as spurious newline characters, prompts, or informative messages). Sample inputs and outputs for $n = 1 \ldots 3$ appear below.

    Input: 1
    Output:
    ```
    (1, {})
    ```

    Input: 2
    Output:
    ```
    (1, {(2, {})})
    (2, {(1, {})})
    ```

    Input: 3
    Output:
    ```
    (1, {(2, {(3, {})})})
    (1, {(3, {(2, {})})})
    (2, {(1, {(3, {})})})
    (2, {(3, {(1, {})})})
    (3, {(1, {(2, {})})})
    (3, {(2, {(1, {})})})
    (1, {(2, {}), (3, {})})
    (2, {(1, {}), (3, {})})
    (3, {(1, {}), (2, {})})
    ```