**Programming Exercises for Chapter 9**

THIS PAGE CONTAINS several exercises for Chapter 9 in [Introduction to Programming Using Java](http://math.hws.edu/eck/cs124/javanotes7/index.html). For each exercise, a link to a possible solution is provided. Each solution includes a discussion of how a programmer might approach the problem and interesting points raised by the problem or its solution, as well as complete source code of the solution.

**Exercise 9.1:**

In many textbooks, the first examples of recursion are the mathematical functions *factorial* and *fibonacci*. These functions are defined for non-negative integers using the following recursive formulas:

factorial(0) = 1

factorial(N) = N\*factorial(N-1) for N > 0

fibonacci(0) = 1

fibonacci(1) = 1

fibonacci(N) = fibonacci(N-1) + fibonacci(N-2) for N > 1

Write recursive functions to compute factorial(N) and fibonacci(N) for a given non-negative integer N, and write a main() routine to test your functions. Consider using the *BigInteger* class (see [Exercise 8.2](http://math.hws.edu/eck/cs124/javanotes7/c8/ex2-ans.html))

(In fact, *factorial* and *fibonacci* are really not very good examples of recursion, since the most natural way to compute them is to use simple for loops. Furthermore, *fibonacci* is a particularly bad example, since the natural recursive approach to computing this function is extremely inefficient.)

[See the Solution](http://math.hws.edu/eck/cs124/javanotes7/c9/ex1-ans.html)

**Exercise 9.2:**

[Exercise 7.7](http://math.hws.edu/eck/cs124/javanotes7/c7/ex7-ans.html) asked you to read a file, make an alphabetical list of all the words that occur in the file, and write the list to another file. In that exercise, you were asked to use an ArrayList<String> to store the words. Write a new version of the same program that stores the words in a binary sort tree instead of in an arraylist. You can use the binary sort tree routines from [*SortTreeDemo.java*](http://math.hws.edu/eck/cs124/javanotes7/source/chapter9/SortTreeDemo.java), which was discussed in [Subsection 9.4.2](http://math.hws.edu/eck/cs124/javanotes7/c9/s4.html#recursion.4.2).

[See the Solution](http://math.hws.edu/eck/cs124/javanotes7/c9/ex2-ans.html)

**Exercise 9.3:**

Suppose that linked lists of integers are made from objects belonging to the class

class ListNode {

int item; // An item in the list.

ListNode next; // Pointer to the next node in the list.

}

Write a subroutine that will make a copy of a list, with the order of the items of the list reversed. The subroutine should have a parameter of type *ListNode*, and it should return a value of type *ListNode*. The original list should not be modified.

You should also write a main() routine to test your subroutine.

[See the Solution](http://math.hws.edu/eck/cs124/javanotes7/c9/ex3-ans.html)

**Exercise 9.4:**

[Subsection 9.4.1](http://math.hws.edu/eck/cs124/javanotes7/c9/s4.html#recursion.4.1) explains how to use recursion to print out the items in a binary tree in various orders. That section also notes that a non-recursive subroutine can be used to print the items, provided that a stack or queue is used as an auxiliary data structure. Assuming that a queue is used, here is an algorithm for such a subroutine:

Add the root node to an empty queue

while the queue is not empty:

Get a node from the queue

Print the item in the node

if node.left is not null:

add it to the queue

if node.right is not null:

add it to the queue

Write a subroutine that implements this algorithm, and write a program to test the subroutine. Note that you will need a queue of *TreeNodes*, so you will need to write a class to represent such queues.

(Note that the order in which items are printed by this algorithm is different from all three of the orders considered in [Subsection 9.4.1](http://math.hws.edu/eck/cs124/javanotes7/c9/s4.html#recursion.4.1).)

[See the Solution](http://math.hws.edu/eck/cs124/javanotes7/c9/ex4-ans.html)

**Exercise 9.5:**

In [Subsection 9.4.2](http://math.hws.edu/eck/cs124/javanotes7/c9/s4.html#recursion.4.2), I say that "if the [binary sort] tree is created by inserting items in a random order, there is a high probability that the tree is approximately balanced." For this exercise, you will do an experiment to test whether that is true.

The depth of a node in a binary tree is the length of the path from the root of the tree to that node. That is, the root has depth 0, its children have depth 1, its grandchildren have depth 2, and so on. In a balanced tree, all the leaves in the tree are about the same depth. For example, in a perfectly balanced tree with 1023 nodes, all the leaves are at depth 9. In an approximately balanced tree with 1023 nodes, the average depth of all the leaves should be not too much bigger than 9.

On the other hand, even if the tree is approximately balanced, there might be a few leaves that have much larger depth than the average, so we might also want to look at the maximum depth among all the leaves in a tree.

For this exercise, you should create a random binary sort tree with 1023 nodes. The items in the tree can be real numbers, and you can create the tree by generating 1023 random real numbers and inserting them into the tree, using the usual treeInsert() method for binary sort trees. Once you have the tree, you should compute and output the average depth of all the leaves in the tree and the maximum depth of all the leaves. To do this, you will need three recursive subroutines: one to count the leaves, one to find the sum of the depths of all the leaves, and one to find the maximum depth. The latter two subroutines should have an int-valued parameter, depth, that tells how deep in the tree you've gone. When you call this routine from the main program, the depth parameter is 0; when you call the routine recursively, the parameter increases by 1.

[See the Solution](http://math.hws.edu/eck/cs124/javanotes7/c9/ex5-ans.html)

**Exercise 9.6:**

The parsing programs in [Section 9.5](http://math.hws.edu/eck/cs124/javanotes7/c9/s5.html) work with expressions made up of numbers and operators. We can make things a little more interesting by allowing the variable "x" to occur. This would allow expression such as "3\*(x-1)\*(x+1)", for example. Make a new version of the sample program [*SimpleParser3.java*](http://math.hws.edu/eck/cs124/javanotes7/source/chapter9/SimpleParser3.java) that can work with such expressions. In your program, the main() routine can't simply print the value of the expression, since the value of the expression now depends on the value of x. Instead, it should print the value of the expression for x=0, x=1, x=2, and x=3.

The original program will have to be modified in several other ways. Currently, the program uses classes *ConstNode*, *BinOpNode*, and *UnaryMinusNode* to represent nodes in an expression tree. Since expressions can now include x, you will need a new class, *VariableNode*, to represent an occurrence of x in the expression.

In the original program, each of the node classes has an instance method, "double value()", which returns the value of the node. But in your program, the value can depend on x, so you should replace this method with one of the form "double value(double xValue)", where the parameter xValue is the value of x.

Finally, the parsing subroutines in your program will have to take into account the fact that expressions can contain x. There is just one small change in the BNF rules for the expressions: A <factor> is allowed to be the variable x:

<factor> ::= <number> | <x-variable> | "(" <expression> ")"

where <x-variable> can be either a lower case or an upper case "X". This change in the BNF requires a change in the factorTree() subroutine.

[See the Solution](http://math.hws.edu/eck/cs124/javanotes7/c9/ex6-ans.html)

**Exercise 9.7:**

This exercise builds on the previous exercise, [Exercise 9.6](http://math.hws.edu/eck/cs124/javanotes7/c9/ex6-ans.html). To understand it, you should have some background in Calculus. The derivative of an expression that involves the variable x can be defined by a few recursive rules:

* The derivative of a constant is 0.
* The derivative of x is 1.
* If A is an expression, let dA be the derivative of A. Then the derivative of -A is -dA.
* If A and B are expressions, let dA be the derivative of A and let dB be the derivative of B. Then the derivative of A+B is dA+dB.
* The derivative of A-B is dA-dB.
* The derivative of A\*B is A\*dB + B\*dA.
* The derivative of A/B is (B\*dA - A\*dB) / (B\*B).

For this exercise, you should modify your program from the previous exercise so that it can compute the derivative of an expression. You can do this by adding a derivative-computing method to each of the node classes. First, add another abstract method to the *ExpNode* class:

abstract ExpNode derivative();

Then implement this method in each of the four subclasses of *ExpNode*. All the information that you need is in the rules given above. In your main program, instead of printing the stack operations for the original expression, you should print out the stack operations that define the derivative. Note that the formula that you get for the derivative can be much more complicated than it needs to be. For example, the derivative of 3\*x+1 will be computed as (3\*1+0\*x)+0. This is correct, even though it's kind of ugly, and it would be nice for it to be simplified. However, simplifying expressions is not easy.

As an alternative to printing out stack operations, you might want to print the derivative as a fully parenthesized expression. You can do this by adding a printInfix() routine to each node class. It would be nice to leave out unnecessary parentheses, but again, the problem of deciding which parentheses can be left out without altering the meaning of the expression is a fairly difficult one, which I don't advise you to attempt.

(There is one curious thing that happens here: If you apply the rules, as given, to an expression tree, the result is no longer a tree, since the same subexpression can occur at multiple points in the derivative. For example, if you build a node to represent B\*B by saying "new BinOpNode('\*',B,B)", then the left and right children of the new node are actually the same node! This is not allowed in a tree. However, the difference is harmless in this case since, like a tree, the structure that you get has no loops in it. Loops, on the other hand, would be a disaster in most of the recursive tree-processing subroutines that we have written, since it would lead to infinite recursion. The type of structure that is built by the derivative functions is technically referred to as a directed acyclic graph.)

[See the Solution](http://math.hws.edu/eck/cs124/javanotes7/c9/ex7-ans.html)