

COMP 250 Winter 2017 - Homework #4

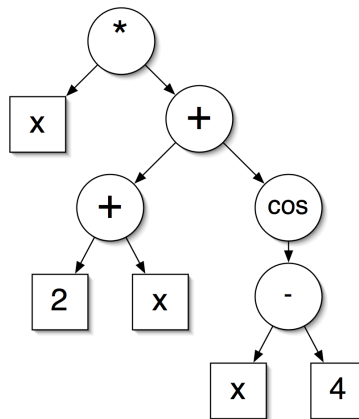
Due date: March 20th, 23:59pm

IMPORTANT NOTES:

- For questions 1:
 - Submit only your ExpressionTree.java
 - Do not change the class name, the file name, the method names, or the methods' arguments. However, you can create new methods if you want.
 - Do not include packages
 - You will get zero for this question if you violate any of the above, because this will break our automated marking program
- For questions 2-5: Submit a single PDF document.

1) (50 points) Manipulation of symbolic expressions

In mathematics, a symbolic expression is an expression that contains variables like x , y , ... For example, $4 * x + 3 * y$ and $\sin(2 * x) + e^x$ are symbolic expressions. We have seen in class how such expressions can be represented using a (improper) binary tree, where internal nodes correspond to operator (+, -, *, cos, sin, exp) and the children of a node are the operands. For example, the expression $x * ((2 + x) + \cos(x - 4))$ can be represented as:



The code available at <http://www.cs.mcgill.ca/~blanchem/250/hw4> provides you with a binary tree data structure for storing expressions. It contains a constructor that takes as input a string describing an expression and returns the root of the binary tree storing that expression. The string describing the expression is in the following format.

- (1) The only operators considered are add, mult, minus, sin, cos, and exp.
- (2) Each operator is followed by its operand(s), in parentheses. If there are two operands, they are separated by a comma.
- (3) We only consider expressions with a single variable called x .

Thus, the mathematical expression $x * ((2 + x) + \cos(x - 4))$ should be written as: `mult(x,add(add(2,x),cos(minus(x,4))))`. So,

`new treeNode("mult(x,add(add(2,x),cos(minus(x,4)))");` returns the root of the tree above. Another example of notation: the expression $\cos(3.1416 \cdot x + e^{\sin(x-1)})$ should be written as: `cos(add(mult(3.1416,x),exp(sin(minus(x,1)))))`

You are also provided with a method `deepcopy()` which returns a copy of the subtree rooted at the node on which it is called. More precisely, it builds a completely new tree, with new nodes, that is a copy of the original. Make sure you understand what it does, as it is going to be very useful. Finally, the `treeNode` class is equipped with a `toString()` method which prints the whole subtree rooted at a node. This will be useful for debugging.

Two useful things in Java:

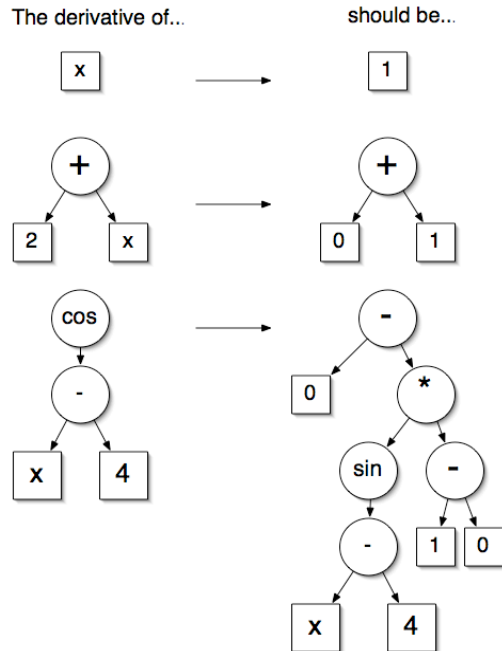
- 1) If you want to test whether two objects of type `String` are equal, use the `equals` method. For example:

```
String word1 = "hello";  
String word2 = "goodbye";  
if ( word1.equals(word2) ) { ...}  
if ( word1.equals("hello") ){...}
```
- 2) All nodes in our expression tree are stored as objects of type `String`. If an object of type `String` actually describes a number, it can be turned into a `double` as follows:

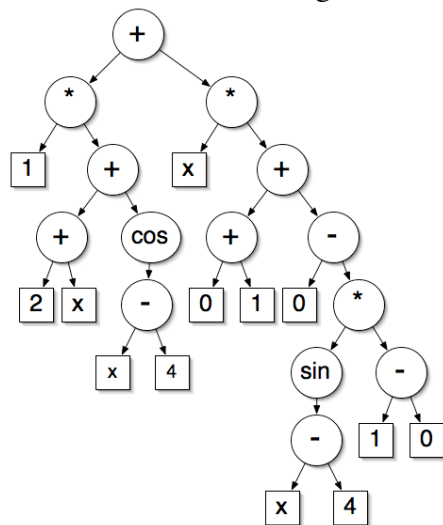
```
String myNumber = "3.1416";  
double x=Double.parseDouble(myNumber);
```

a) (15 points) Complete the `evaluate(double x)` method, which returns the value of the expression rooted at the `treeNode` on which it called, for a particular value of `x`. For example, `evaluate(1)` on the tree above should return approximately 2.01.... For `sin` and `cos`, angles are measured in radians.

b) (25 points) Complete the `differentiate()` method, which returns a new expression tree describing the derivative of the expression described by the tree rooted at the `treeNode` on which it is called. Notice that the original tree should remain intact and that the tree containing the derivative should share no node with the original. For example:



The derivative of the original tree is thus:



Notice that the tree above may seem more complicated than necessary. However, it is the one you will most likely end up with after differentiating the original expression. (It could be slightly different but equivalent, and that would be OK). Do not worry about simplifying the expression described by the tree.

Again, that sounds really hard, but it's not, because the rules of derivation lend themselves very well to the tree representation we are using. Indeed, the chain rule for derivation is a recursive algorithm!

$$d/dx (f(x) + g(x)) = d/dx f(x) + d/dx g(x)$$

$$d/dx (f(x) - g(x)) = d/dx f(x) - d/dx g(x)$$

$$d/dx (f(x) * g(x)) = (d/dx f(x)) * g(x) + f(x) * (d/dx g(x))$$

$$d/dx (\sin(f(x))) = \cos(f(x)) * d/dx f(x)$$

$$d/dx (\cos(f(x))) = - \sin (f(x)) * d/dx f(x)$$

$$d/dx (\exp(f(x))) = \exp(f(x)) * d/dx f(x)$$

with the base cases:

$$d/dx (x) = 1$$

$$d/dx (c) = 0 \text{ for any constant } c$$

2) (16 points) Binary search trees

Consider a binary search tree that contains n nodes with keys $1, 2, 3, \dots, n$.

The shape of the tree depends on the order in which the keys have been inserted in the tree.

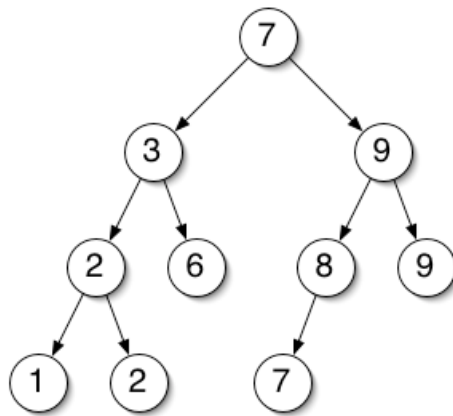
- a) In what order should the keys be inserted into the binary search tree to obtain a tree with minimal height?
- b) On the tree obtained in (a), what would be the worst-case running time of a find, insert, or remove operation? Use the big-Oh notation.
- c) In what order should the keys be inserted into the binary search tree to obtain a tree with maximal height?
- d) On the tree obtained in (c), what would be the worst-case running time of a find, insert, or remove operation? Use the big-Oh notation.

3) (20 points) Tree traversals

Consider the following pair of recursive algorithms calling each other to traverse a binary tree.

```
Algorithm weirdPreOrder(treeNode n)
  if (n != null) then
    print n.getValue()
    weirdPreOrder( n.getRightChild() )
    weirdPostOrder( n.getLeftChild() )
```

```
Algorithm weirdPostOrder(treeNode n)
  if (n != null) then
    weirdPreOrder( n.getRightChild() )
    weirdPostOrder( n.getLeftChild() )
    print n.getValue()
```



- (5 points) Write the output being printed when weirdPreOrder(root) is executed on the following binary tree.
- (5 points) Write the output being printed when weirdPostOrder(root) is executed.
- (5 points) Consider the binary tree traversal algorithm below.

Algorithm queueTraversal(treeNode n)

Input: a treeNode n

Output: Prints the value of each node in the binary tree rooted at n

Queue q ← new Queue();

q.enqueue(n);

while (! q.empty()) **do**

 x ← q.dequeue();

print x.getValue();

if (x.getLeftChild() != null) **then** q.enqueue(x.getLeftChild());

if (x.getRightChild() != null) **then** q.enqueue(x.getRightChild());

Question: Write the output being printed when queueTraversal(root) is executed.

d) (4 points) Consider the binary tree traversal algorithm below.

Algorithm stackTraversal(treeNode n)

Input: a treeNode n

Output: Prints the value of each node in the binary tree rooted at n

Stack s \leftarrow new Stack();

s.push(n);

while (! s.empty()) **do**

 x \leftarrow s.pop();

print x.getValue();

if (x.getRightChild() != null) **then** s.push(x.getRightChild());

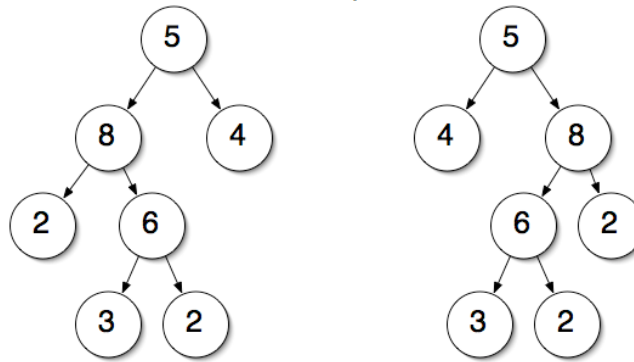
if (x.getLeftChild() != null) **then** s.push(x.getLeftChild());

Question: Write the output being printed when stackTraversal(root) is executed. This is the equivalent of what traversal method seen previously in class?

4) (14 points) Tree isomorphism

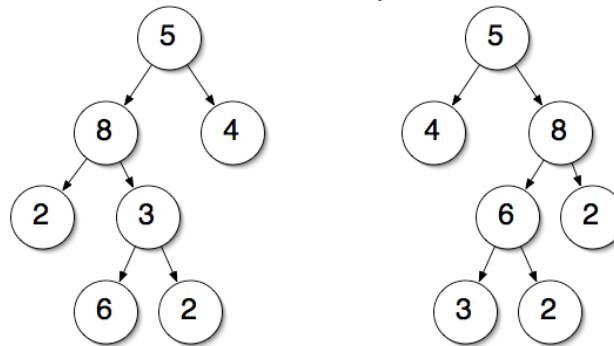
Two unordered binary trees A and B are said to be *isomorphic* if, by swapping the left and right subtrees of certain nodes of A, one can obtain a tree identical to B. For example, the following two trees are isomorphic:

Two isomorphic trees



because starting from the first tree and exchanging the left and right subtrees of node 5 and of node 8, and leaving the children of node 6 unchanged, one obtains the second tree. On the other hand, the following two trees are not isomorphic, because it is impossible to rearrange one into the other:

Two non-isomorphic trees



Question: Write the pseudocode of a recursive algorithm that tests if the trees rooted at two given treeNodes are isomorphic. Hint: if your algorithm takes more than 10 lines to write, you're probably not doing the right thing.