06-26956/26955 *Software Workshop*
Spring Semester 2015-16

The University of Birmingham
School of Computer Science

Work Sheet 2: Recursion and Trees (Parts A and B)

(Assessed Exercise: 16.7% of the Spring Term continuous assessment)

Assigned: Thursday, 21 January
Due: Tuesday, 2 February, 11:59pm.
(5% late submission penalty per 24 hours or part thereof. No submission after 24 hours.)
(Junit testing and documentation mandatory.)

You should use the class `Tree` provided in the Worksheet2 starter pack. The starter pack includes a template for a class `Worksheet2`, where you should write all your methods.

In this Worksheet, *you must not use assignment statements to change the value of any variables.* Recursion is the main vehicle by which you will get computations done. You can use local variable declarations with initialisations. But once initialized, the variables keep those values till the end of their scope. They do not change. You cannot use any loop statements either. Any solution containing an assignment statement or a loop statement will receive a mark of 0.

Part A: Binary trees and Binary search trees

Exercise 1: Negate a tree (5%)

```
static Tree negateAll(Tree a)
```

Given a tree of integers a, write a method that returns a new tree containing all the elements of a with their sign negated, i.e., positive integers become negative and negative integers become positive.

Exercise 2: Mirror image (10%)

```
static Tree mirror(Tree a)
```

Given a tree a, construct and return a tree that is the mirror image of a along the left-right axis. Here is an example:

The tree on the left is the mirror image of the tree on the right, and vice versa.

Exercise 3: Postorder traversal (10%)

```
static List postorder(Tree a)
```

Given a tree a, produce and return a list containing the values in a by trvaersing the nodes in *postorder*, i.e., for every node, all the values in the left subtree should be listed first, then all the values in the right subtree and then finally the value in the node itself.

Hint: Recall the method for inorder traversal done in the Lecture.

Exercise 4: Check for positive (5%)

```
static boolean allPositive(Tree a)
```

Given a tree of integers a, return a boolean value indicating whether *all* the values in its nodes are positive, i.e., ≥ 0.

Binary search trees

As dissussed in the Lecture, a tree is a binary search tree if, *for every node*,

- all the values stored in the left subtree of the node are less than the value at the node, and
- all the values stored in its right subtree of the node are greater than the value at the node.

Exercise 5: Test for the search tree property (10%)

```
static boolean isSearchTree(Tree a)
```

Given a tree of integers a, write a method that returns a boolean value indicating whether a is a binary search tree.
Hint: You may need helper functions to write this method. Document any helper functions you define.

Exercise 6: Traversing binary search trees (10%)

```
static void printDescending(Tree a)
```

Given a binary search tree of integers a, write a method that prints the values stored in it in *descending order*. Do this *without building a separate list of the values.*

Exercise 7: Maximum value in a search tree (10%)

```
static int max(Tree a)
```

Assuming that the argument tree a is a binary search tree, write an efficient method to find the maximum value stored in the tree. Your method must not visit and compare all the nodes in the tree. Rather, it it must traverse at most one path in the tree from the root node. This should work in $O(\log\ n)$ time for a balanced binary tree.
(**Hint:** In a binary search tree, all the values in the left subtree of a node are less than or equal to the value in the node. So, the maximum value can't be in the left subtree, right? Where can it be?)

Exercise 8: Deleting a value in a search tree (15%)

```
static Tree delete(Tree a, int x)
```

Assuming that the argument tree a is a binary search tree, this method must delete the value x from a and return the resulting tree. If there are multiple copies of x in the tree, you need to delete only one copy of x. The original tree a must not be altered. Rather, you should build a new tree that contains all the values of a except for one copy of x. The resulting tree must be again a binary search tree.
Your algorithm must take time proportional to the height of the tree, which is normally $O(\log n)$.
(**Hint:** As discussed in Lecture, the node of x may have two subtrees. In that case, the node cannot be simply deleted. Rather, you need to replace x in that node with the maximum value of the left subtree.)

Part B: Height-balanced trees (AVL trees)

Height-balanced trees A height-balanced binary tree is a binary tree in which, for every node, the left and right subtrees have *a difference in height of at most 1*. Insertion/deletion in a height-balanced tree may in general destroy the height-balanced property. In that case, we can use rotations discussed in Lecture to rebalance the tree and obtain a height-balanced tree again. Please consult any Data Structures text book as well as online resoures for reference material.

AVL trees A height-balanced binary search tree is called an "AVL tree" (named after its inventors, Adelson-Velsky and Landis).
For checking the height-balanced property of nodes after insertions/deletions, one needs an efficient method to obtain the height of a tree. Explicit calculation requires $O(n)$ time, which is too expensive. For this purpose, the `Tree` class given to you has been extended with an instance variable to store the height of the tree:

```
protected final int height;
```

It also has an instance method

```
public int getHeight();
```

that returns the stored height value.

Exercise 9: Checking for height-balanced property (5%)

```
static boolean isHeightBalanced(Tree a)
```

Given a tree of integers a, check to see if it is height-balanced, returning a boolean value.

Exercise 10: Insertion/deletion with height-balancing (20%)

```
static Tree insertHB(int x, Tree a)
static Tree deleteHB(Tree a, int x)
```

Write modified versions of `insert` and `delete` methods that maintain the height-balanced property of trees. You should assume that the input trees are height-balanced and produce results that are height-balanced.
Both the methods should work in $O(\log\ n)$ time.
You may use the `isHeightBalanced` method in `assert` statements to ensure that your code works correctly.

End notes

- The `Tree` class provided in the starter pack includes a `toString` that gives a pretty-printed version of a tree as a string.
- The `Tree` class also includes an `equals` method. So, you can use `assertEquals` in your tests directly on trees. You should *not* convert trees to strings for checking equality.