The University of Birmingham
School of Computer Science

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

(Assessed Exercise: 1% of the module mark)

Assigned: Thursday, 23 January
Due: Wednesday, 5 February, 2:00pm.
(5% late submission penalty for up to 24 hours. 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. The solutions will be checked against `Worksheet2Interface`, which is also included in the starter pack. *You should not modify the interface file.*

As in Worksheet 2-1, *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. 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<Integer> negateAll(Tree<Integer> 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: Check for even numbers (5%)

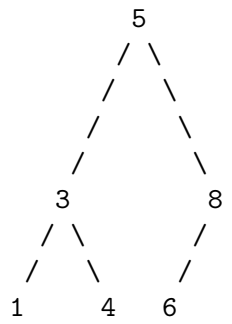
```
static boolean allEven(Tree<Integer> a)
```

Given a tree of integers a, return a boolean value indicating whether *all* the values in its nodes are even, i.e., divisible by 2.

Exercise 3: Depth of a node(10%)

```
static int depth(Tree<Integer> a, int x)
```

The depth of a node is defined as the length of the path (i.e. the number of edges) between the node and the root. For example, in the tree below, node 5 has a depth of 0, while node 6 has a depth of 2.
Given a tree of integers a, which does not contain duplicates, and a node value x, write a recursive program which returns the depth of the given node value. Return -1 if the node value is not in the tree.



Exercise 4: Preorder traversal (5%)

```
static<E> List<E> preorder(Tree<E> a)
```

Given a tree a, produce and return a list containing the values in a by traversing the nodes in *preorder*, i.e., for every node, its own value should be listed first, followed by all the values in the left subtree and finally all the values in the right subtree.
Hint: Recall the method for inorder traversal done in the Lecture.

Binary search trees

As discussed 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 the right subtree of the node are greater than the value at the node.

Exercise 5: Test for the search tree property (15%)

```
static boolean isSearchTree(Tree<Integer> 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<Integer> 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<Integer> 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 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 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<Integer> delete(Tree<Integer> 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. 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 again be 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 lectures, the node containing x may have two subtrees. In that case, the node cannot simply be 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 the rotations discussed in lectures to rebalance the tree and obtain a height-balanced tree again. Please consult any Data Structures text book as well as online resources 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<E> a)
```

Given a tree a (of an arbitrary element type `E`), check to see if it is height-balanced, returning a boolean value.

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

```
static Tree<Integer> insertHB(Tree<Integer> a, int x)
static Tree<Integer> deleteHB(Tree<Integer> 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 binary search trees and produce results that are height-balanced binary search trees.
Both 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` method 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.
- No Junit tests are required for Exercise 6.