Project 6: The Height of a Binary Tree

1 Why do we do it?

Tree is a very useful general data structure. We want to have some experience of working with it, partly to prepare for the forthcoming project 7.

On Page 6 of the HeapSort notes, we discussed the height of a binary tree. In the extreme case, it can be quite high, when it goes up (comes down?) to a linear list.

It is stated, in Theorem 12.4 on Page 33 of the BST notes, that the expected height of a randomly built BST with n keys is $O(\log n)$. We now want to confirm/demonstrate this result with some experiments, similar to what we did in Projects 2, 3, and 4.

The core of this investigation is a method, `treeHeight(BinaryTreeNode)`, which will return the height of a tree, rooted at root, when called with (Cf. Figure 1)

```
treeHeight(bst1.root)
```

This operation is certainly useful when deciding which node in either the left subtree, or the right subtree, should be used in the Deletion operation for the BST structure (Cf. Exercise 12.3-6).

2 What to do?

You need to implement the BST structure, as well as the needed operations, mainly insertion and `treeHeight`.

2.1 Get familiar with the binary tree structure

To complete this project, you have to stick with the following specification as is. To start, as shown on Page 4 of the BST notes, a node in a binary tree looks like the following, in terms of Java:

```
public class BinaryTreeNode {
    int key;
    BinaryTreeNode left, right, parent;
}
```

---

1To refresh, the height of a node in a tree is the number of edges on the longest simple path from this node to a leaf in the tree, and the height of a tree is the height of its root. Thus, e.g., for the tree as shown in Figure 1, the height of the node containing 3 is 0, that of the one holding 4 is 2, and the height of this tree is that of 9, i.e., 3.
public BinaryTreeNode(int key){
    this.key=key;
    left=right=parent=null;
}
}

And a binary tree looks like the following:

public class BinaryTree {
    BinaryTreeNode root;

    public BinaryTree(){
        root=null;
    }
}

Figure 1 shows an example, where the parent pointers are missing.

To make yourself comfortable with tree operations, you want to look at the figure on Page 27 of the BST notes, and understand it in light of the non-recursive insertion operation related discussion, starting on Page 24.

2.2 Implementation of the BST operations

You then have to implement at least the aforementioned Insertion operation, and another method, int treeHeight(BinaryTreeNode root), that gives the height of any tree with its top node referred by the root parameter.

Question 1: What are the heights of the trees rooted at 3, 6, 4 and 9, as shown in Figure 1, in this order? Why? You need to work this out by hand, then confirm it with the treeHeight operation that you will develop later.
Question 2: How to call the function $treeHeight$, given the above declaration of a binary tree, as given in Section 2.1?

Question 3: You can certainly convert the non-recursive one as given on Page 25 of the BST notes into a Java method. On the other hand, if you decide to do it recursively, how do you deal with the base case, i.e., what is the height of an empty binary tree? And how do you come up with a general case when the tree is not empty?

You might want to play with a few real binary trees, including the one as shown in Figure 1, to complete a recursive definition of the $treeHeight$ function.

Question 4: Is your completed definition of the $treeHeight$ method a PreOrder, InOrder, PostOrder, or the LevelOrder, traversal of the tree? Why?

3 What to do?

1. Generate a random permutation of $n$ distinct integers, $n \in \{10, 50, 100, 200, 500, 1000\}$.

2. Insert $n$ nodes containing the respective keys generated in the previous step into an initially empty binary search tree, according to the random order.

3. Use the $treeHeight$ method you wrote in Section 2.2 to find out the height of this just generated tree. For every $n$, repeat this experiment for at least $n$ different lists of numbers $^2$ to find out the average of the measured heights for that specific value of $n$.

4. For each value of $n$, compare the just obtained average height with the theoretical figure of $\lceil \log_2(n+1) \rceil$ (Cf. Theorem 12.4 as stated on Page 33 of the BST notes).

4 What to hand in?

Source code of your program, together with a lab report $^3$ on the output, containing a table showing, for each $n \in \{10, 50, 100, 200, 500, 1000\}$, the experimental data and the theoretical data, as well as their comparison, either in chart or data showing that their constant ratio $^4$.

5 A general grading guideline

$\geq 1$: A syntactically correct and executable program, containing both the $insert$ and the $treeHeight$ functions.

---

$^2$Keep in mind that there are $n!$ such lists. For example, when $n = 10$, there are 3,628,800 such lists.

$^3$You should include a spreadsheet, which shows the theoretical result and the practical data, and their comparison. You should also send in a chart, showing the comparison of these two “curves”.

$^4$For examples, check out the sampler for the earlier projects.
≤ 3: Correct solutions to the questions as posed in Section 2.2

≤ 5: Collection of the practical and theoretical data regarding the heights of randomly generated BST in a spreadsheet and justification of their relationship with, e.g., charts.