Project 6: The Height of a Binary Tree

1 Background

In Chapter 12, we discussed the height of a binary tree. In the extreme case, it can be quite high, when it goes up to a linear list.

We pointed out, in Theorem 12.4, that the expected height of a randomly built BST with \( n \) keys is \( O(\log n) \). We now want to confirm this result with some experiments, similar to what we did in Project 2.

This project on a linked structure based tree implementation, and the last two projects, will also set the stage for the next one, the mother of all the projects.

2 What to do?

2.1 Get familiar with the binary tree structure

Many of you implemented your own “circular doubly linked list” when completing Project 5. However, for this one, you have to stick with the following specification to give it a uniform flavor, which you need for the next project.

To start, a node in a binary tree looks like the following, in terms of Java:

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

    public BinaryTreeNode(int key){
        this.key=key;
        left=right=parent=null;
    }
}
```

while a binary tree looks like the following:

```java
public class BinaryTree {
    BinaryTreeNode root;

    public BinaryTree(){
        root=null;
    }
}
```
You then have to implement at least the **Insertion** operation, as contained in the lecture notes, either recursively, or non-recursively.

You also want to write a method `int treeHeight(BinaryTreeNode root)` that gives the height of any tree with its top node referred by the `root` parameter.

**Question:** How to call such a function, given the above declaration of a binary tree?

The gist of this algorithm is the following: the height of a binary tree is one plus the maximum of the heights of its left and right subtrees.

**Question:** What is the height of an empty binary tree? ☺

You might want to play with a few real binary trees to come up with a correct base case to complete a recursive definition of the `treeHeight` function.

*Notice that, as Dan w/ B pointed out in a class, the running time of this `treeHeight` function is $\Theta(n)$ since it has to find the height of each and every node before finding out the height of the whole tree.*

Incidentally, it is also an example of PreOrder traversal of a binary tree.

### 2.2 What to do?

1. Generate a random permutation of the integers 1 through $n$, $n \in [10, 50, 100, 200, 500, 1000]$.

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

3. Use the `treeHeight` method you wrote in Sec. 2.1 to find out the height of this just generated tree. For every $n$, repeat this experiment for at least $n$ different lists of numbers\(^1\) 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 $\lceil \log_2(n + 1) \rceil$.

### 3 What to hand in?

Email me the source code of your program, together with a lab report \(^2\) 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.

---

\(^1\)Keep in mind that there are $n!$ such lists. For example, when $n = 10$, there are 3,628,800 such lists. ☺

\(^2\)You should send in 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 the two “curves”.

---

2