## 12.a:

I've modified the "TreeNode" class so that it can:

1. Save the Object data

2. Create an constructor for it. i.e., **public** TreeNode(**int** key, Object data);

It can pass the test cases now.

### 12.b:

Things I've Changed in the TreeNode.java:

1. I've changed the field modifier from "private" to "protected", in order to access/modify the field
2. In order to avoid type casting, I used the "BalancedTreeNode" class more like a factory -- the additional methods are all static, and primarily used the "TreeNode" class to store data

## 12.c:

Test Case:

1. One Element: root, key = 100
   1. output: root key = 100, weight = 1
2. Then test a normal input:
   1. then insert 12,19, output: root key = 19, weight = 2;
   2. then insert 15, output: root key = 19, weight = 3
   3. then insert 14, output: root key = 15, weight = 4
   4. then insert 13, but check the left child, output: root.left key = 13, weight = 2
3. a list of increasing order elements: 1~200
   1. After each insertion, check if the tree is weight balanced using the test method "testTreeBalanced"; if it cannot pass the test, then throw an exception
   2. Also verify the correctness of insertion, by performing a tree walk to output all the keys
4. Then test a list of decreasing ordered elements: 200~1, following the same testing procedure
5. After that, test a list of random numbers for 200 times, following the same testing procedure

For the Java Code, see the main method of the BalancedTreeNode class.

## 12.d:

Here's the data I got:

|  |  |  |
| --- | --- | --- |
| n | t = time(s) | 2^(T(n)/n) |
| 100 | 0.040443 | 1.000280369 |
| 200 | 0.145360 | 1.000503906 |
| 400 | 0.604627 | 1.001048288 |
| 600 | 0.924337 | 1.001068406 |
| 800 | 1.262695 | 1.001094641 |
| 1000 | 1.682637 | 1.001166996 |
| 1300 | 2.215381 | 1.001181917 |
| 1600 | 2.948135 | 1.001277998 |
| 2000 | 4.007220 | 1.001389761 |

Here's the graph for profiling:

As we can see from the graph, t over n tends to be linear, while over n tends to be constant (considering the fact that the profiler may not be so accurate when n is small).

This implies that, T(n) = O(n)

(Since => => T(n) = n \* log c = O(n) )

## 13.a

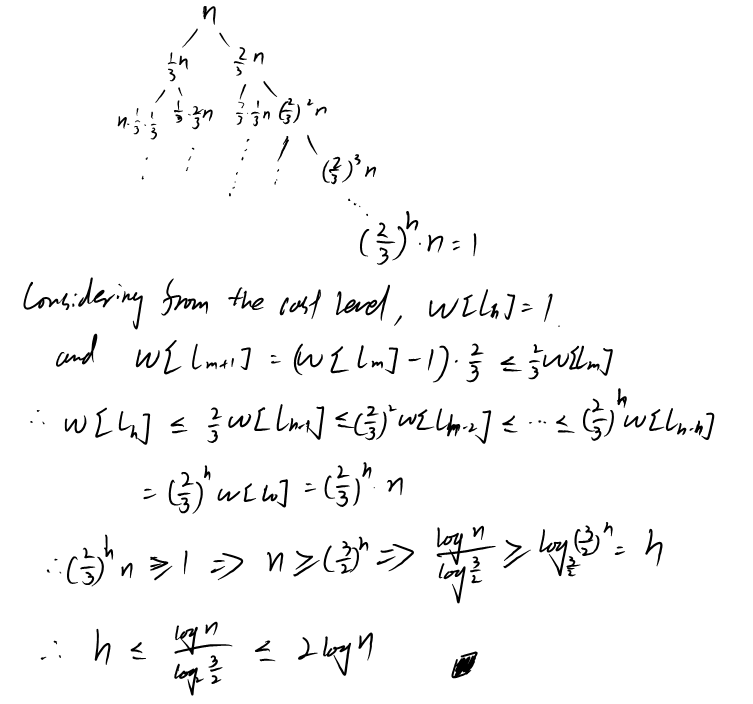
1. denote **ln** as the rightmost node in the nth level of the tree, the root is level 0 -- i.e., l0;
2. denote **w[ln.right]** as the weight of the right child of a node in level n

Without the loss of Generality, assume w[ln. right]

since n is the weight of the root, and base on w[ln.left] <= 2w[ln.right] we can get:

n = w[root] = w[l0] = w[l0.left] + w[l0.right] >= 2w[l0.] + 2w[l]

Consider the worst case, where the weight of the left subtree is exactly half of the right subtree. Like this:



## 13.b:

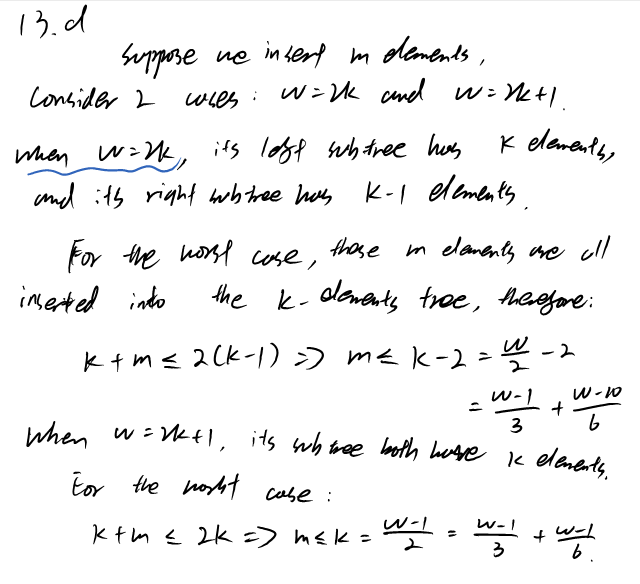
First consider the purely insertion: every time of insertion and element m, at worst it would have to go through all the levels, which pass through O(log n) elements (deposit O(log n) dollars each, k log n dollars in total);

When spending these money during the rebalancing, since the bank accounts always > 0, the total cost for inserting k keys into an empty tree is always less than the money deposited, i.e. O(k log n)

## 13.c:

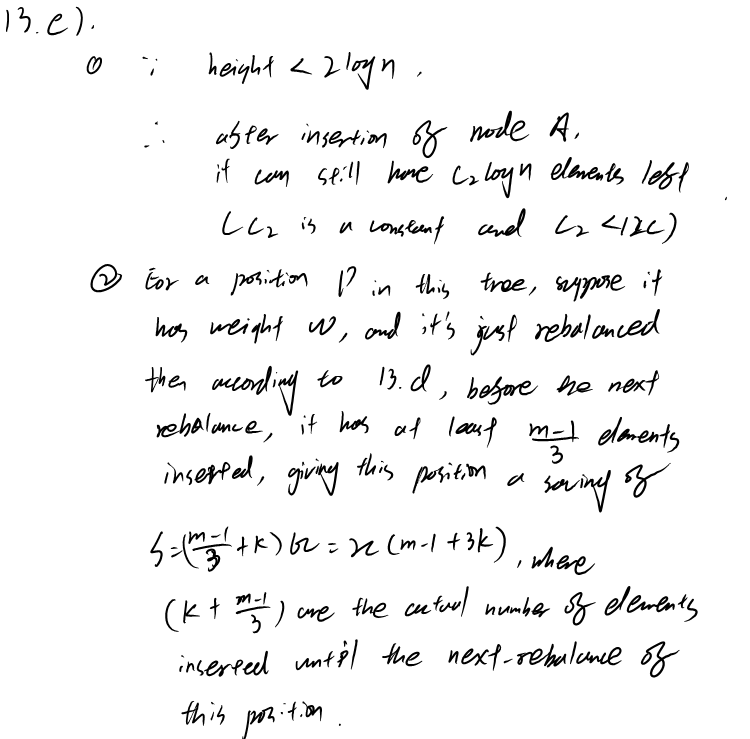
Since the rebalance need to rebuild a tree from node v's sub nodes, and v has w sub-nodes (v included). Therefore it would spend O(w) steps.

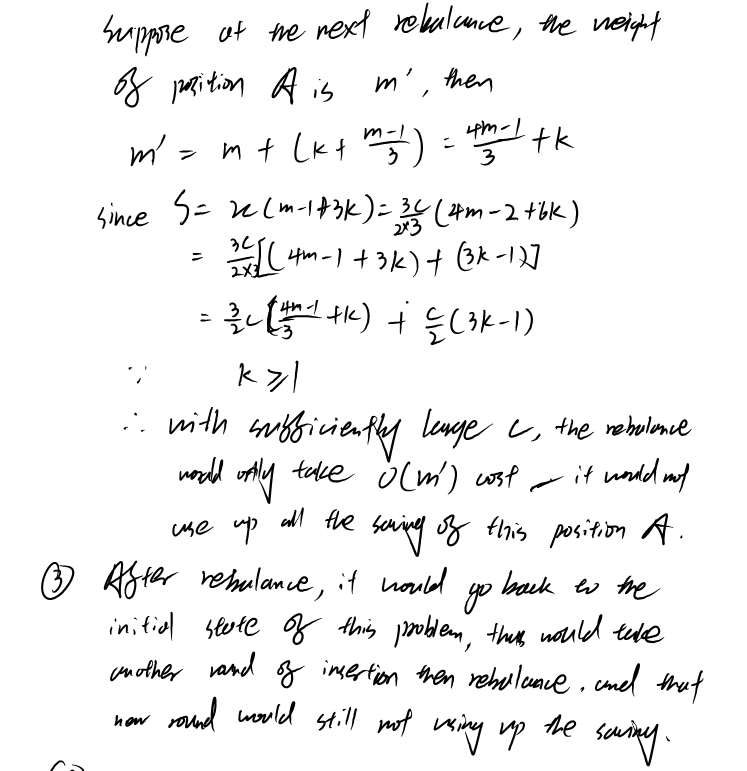
## 13.d:

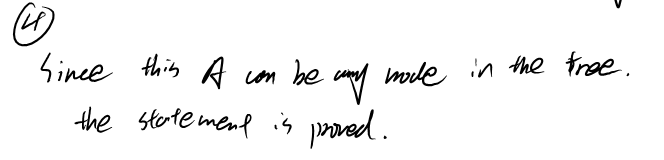


In conclusion, the statement holds as long as the weight is greater than 10.

### 13.e:







### 13.f:

Proven by 13.e, on average, inserting one element would take no more than O(log n) steps.

Therefore, inserting k elements would take at most O(k log n) steps.

References:

For this question, I've looked up the CLRS 2nd ed, Chapter 17 - Amortized

Analysis.

I've mainly discussed the problem with Ding Mingzhe, who remind me with the

wrong understanding of the problem; in addition, I also discussed it with

Jiao Jingping, Zhou Jun