CS2020: Data Structures and Algorithms (Accelerated)

Problems 12–13

Due: February, 23:59

Overview. In this problem, we revisit the binary search tree. After fixing any problems with your binary search tree implementation from Problem Set 3, you will improve your binary search tree so that it maintains a special *weight-balance* property. You will then (as a bonus problem) show that the weight-balanced implementation guarantees that the average cost of an insertion is $O(\log n)$, and that the cost of a search is always $O(\log n)$.

Collaboration Policy. As always, you are encouraged to work with other students on solving these problems. However, you **must** write up your solution **by yourself**. In addition, when you write up your solution, you **must** list the names of every collaborator, that is, every other person that you talked to about the problem (even if you only discussed it briefly). Any deviation from this policy will be considered cheating, and will be punished severely, including referral to the NUS Board of Discipline.

Problem 12. (Weight-Balanced Binary Search Trees)

In Problem 6 on Problem Set 3, you implemented a basic binary search tree. In this problem, you will extend your implementation so that the tree always remains balanced.

For the purpose of this problem, we will use a different type of balancing than we did in lecture. An AVL tree is always *height-balanced*. Recall that a tree is *height-balanced* if, for every node v, $|height(v.left) - height(v.right)| \leq 1$ (i.e., the height of a node's left child and right child differ by at most 1).

For this problem, we will maintain a *weight-balanced* tree, **not a height-balanced tree**. Recall in Problem 6, we defined the weight of a node v as the total number of keys in the sub-tree rooted at v. Formally, we defined $weight(v)$ as follows:

- $weight(v) = 1$ if v is a leaf, i.e., has no children.
- $weight(v) = weight(v.left) + 1$ if v has a left child, but no right child.
- $weight(v) = weight(v.right) + 1$ if v has a right child, but no left child.
- $weight(v) = weight(v.left) + weight(v.right) + 1$ if v has both a left and a right child.

We say that a node v is *weight-balanced* if the ratio of the children's weight is at most a factor of 2. Formally, we say that a node v is *weight-balanced* if it satisfies at least one of the following conditions:

1. Node v has no left child and $weight(v.right) = 1$.
2. Node v has no right child and $weight(v.left) = 1$.
3. Node v has both a left child and a right child, and $1/2 \leq weight(v.left)/weight(v.right) \leq 2$.

If a node v does not satisfy one of these three conditions, then we say it is *unbalanced*

The first two conditions deal with the case where v has only one child. In this case, the child must be a leaf (i.e., have weight 1); otherwise, node v is unbalanced. The last condition says that a node v is balanced if the weight of its children differs by at most a factor of 2.

We say that a tree is *weight-balanced* if every node in the tree is weight-balanced. (Notice that a tree may be height-balanced, but not be weight-balanced! Is it possible to have a tree that is weight-balanced but not height-balanced?) We have several examples of balanced and unbalanced trees:

- In Figure 1 and Figure 6, there are two examples of weight-balanced trees.
- In Figure 2, there is an example of a tree that violates condition 2: node 12 has no left child, and the weight of its right child is > 1.
- In Figure 4 and Figure 5, there are two examples of trees that violate condition 3. In Figure 4, node 13 has two children, one with with 1 and one with weight 3. Thus:

$$weight(v.left)/weight(v.right) < 1/2,$$

  violating condition 3. In fact, node 11 is also unbalanced, in that it has a left child with weight 2 and a right child with weight 5. (In addition, node 15 is unbalanced as well: it has

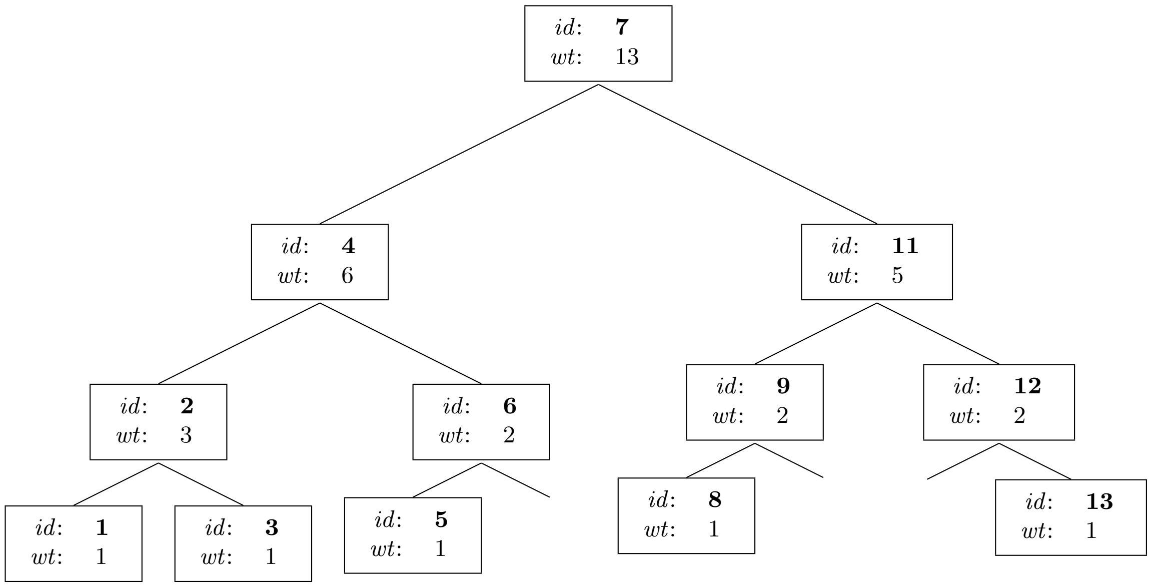

Figure 1: A tree that is weight-balanced, i.e., for every node v: $1/2 \leq w(v.left)/w(v.right) \leq 2$.

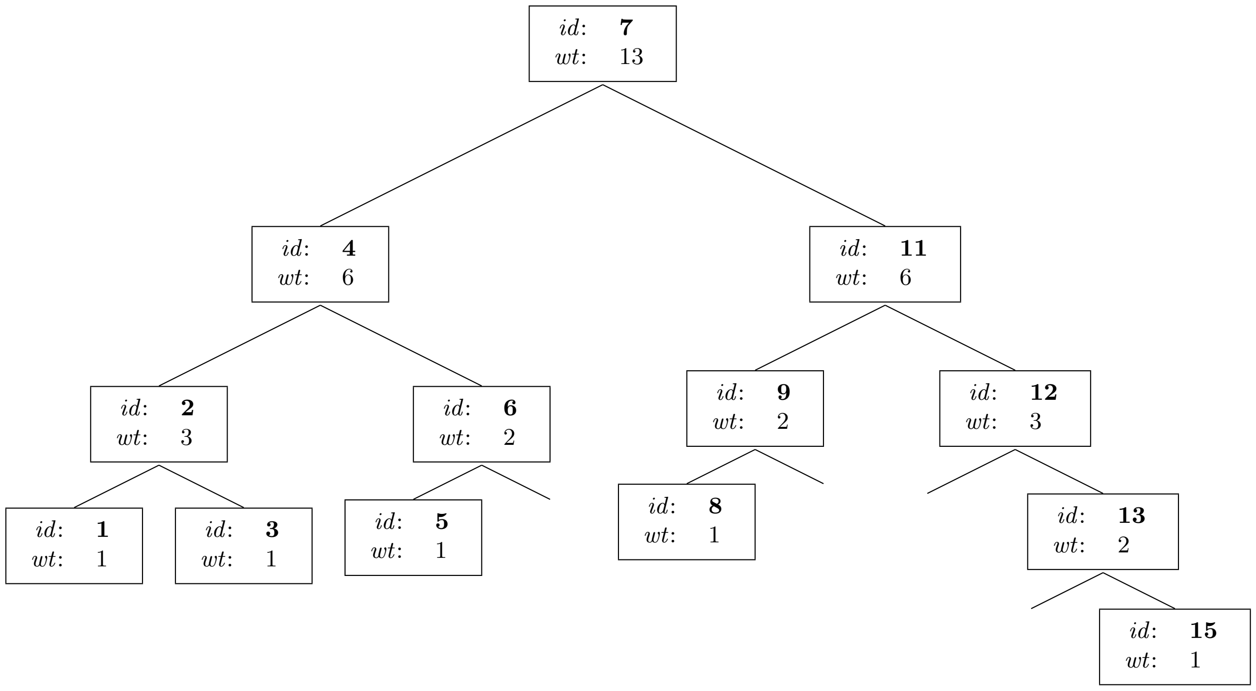

Figure 2: After inserting key 15, the tree is no longer weight-balanced. Notice that the node with identifer 12 has one child with weight 2 and one child with weight 0. Node 12 should be rebalanced.

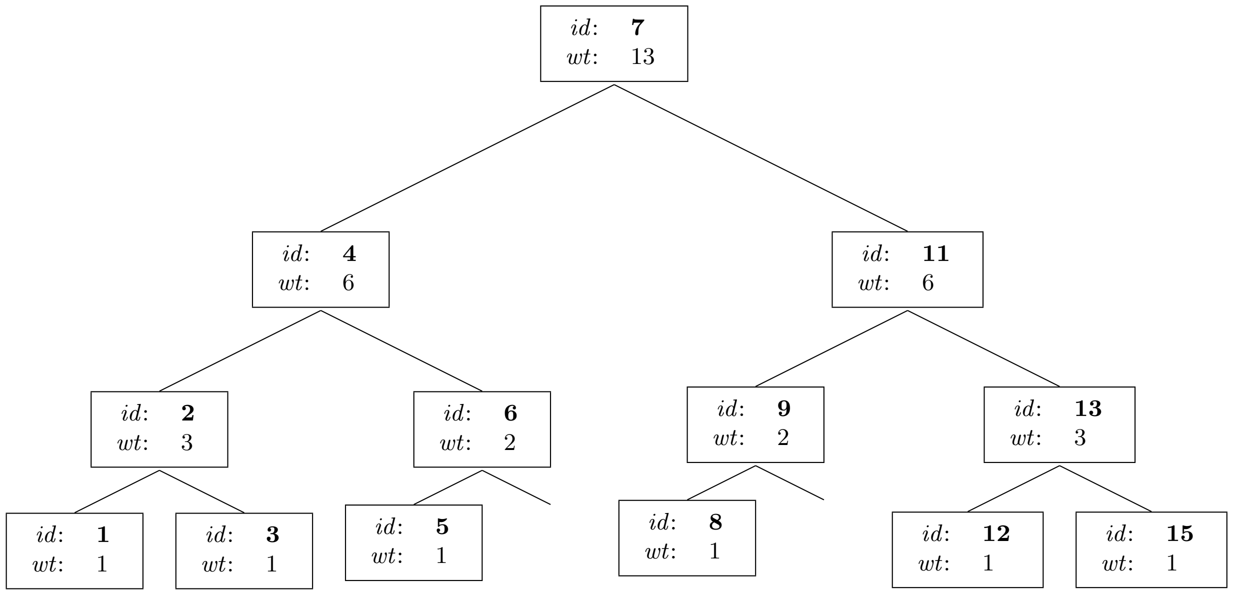

Figure 3: After rebalancing node 12, the tree is balanced again.

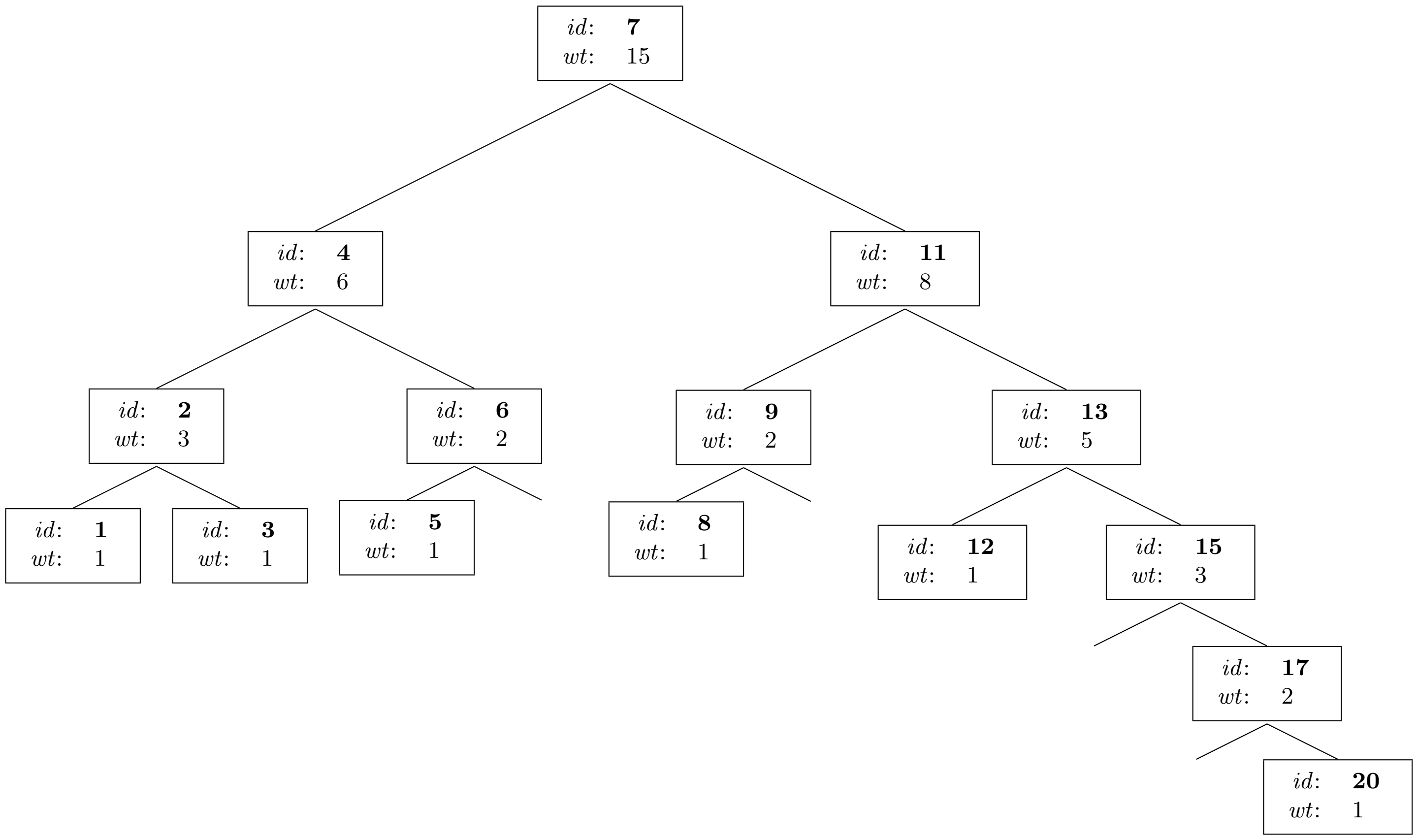

Figure 4: Now, we insert another two keys: 17 and 20. Now, the tree is no longer weight-balanced. Notice that there are three different nodes that are unbalanced: 11, 13, and 15. These three can be rebalanced in any order.

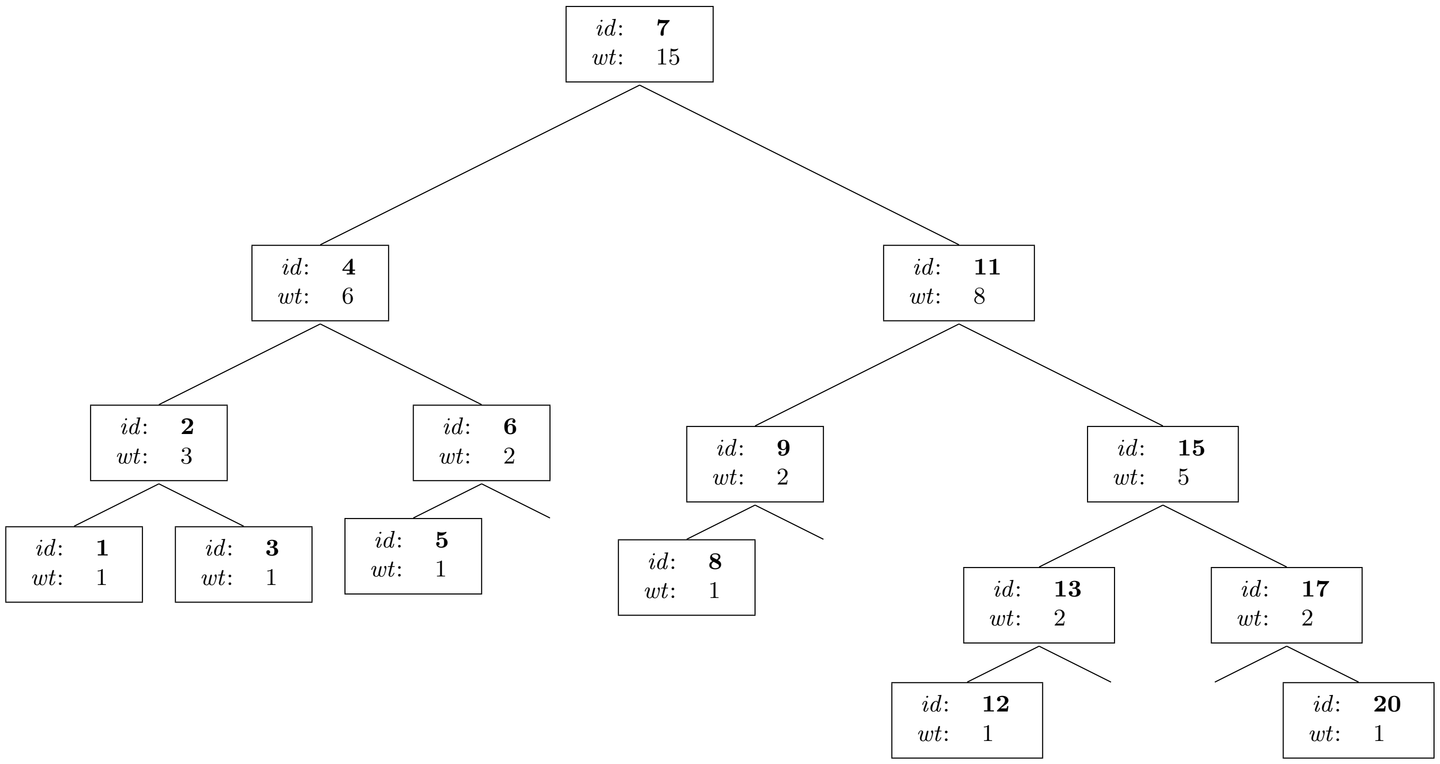

Figure 5: Here, we have rebalanced node 13. Notice that the tree is still unbalanced: node 11 still needs to be rebalanced.

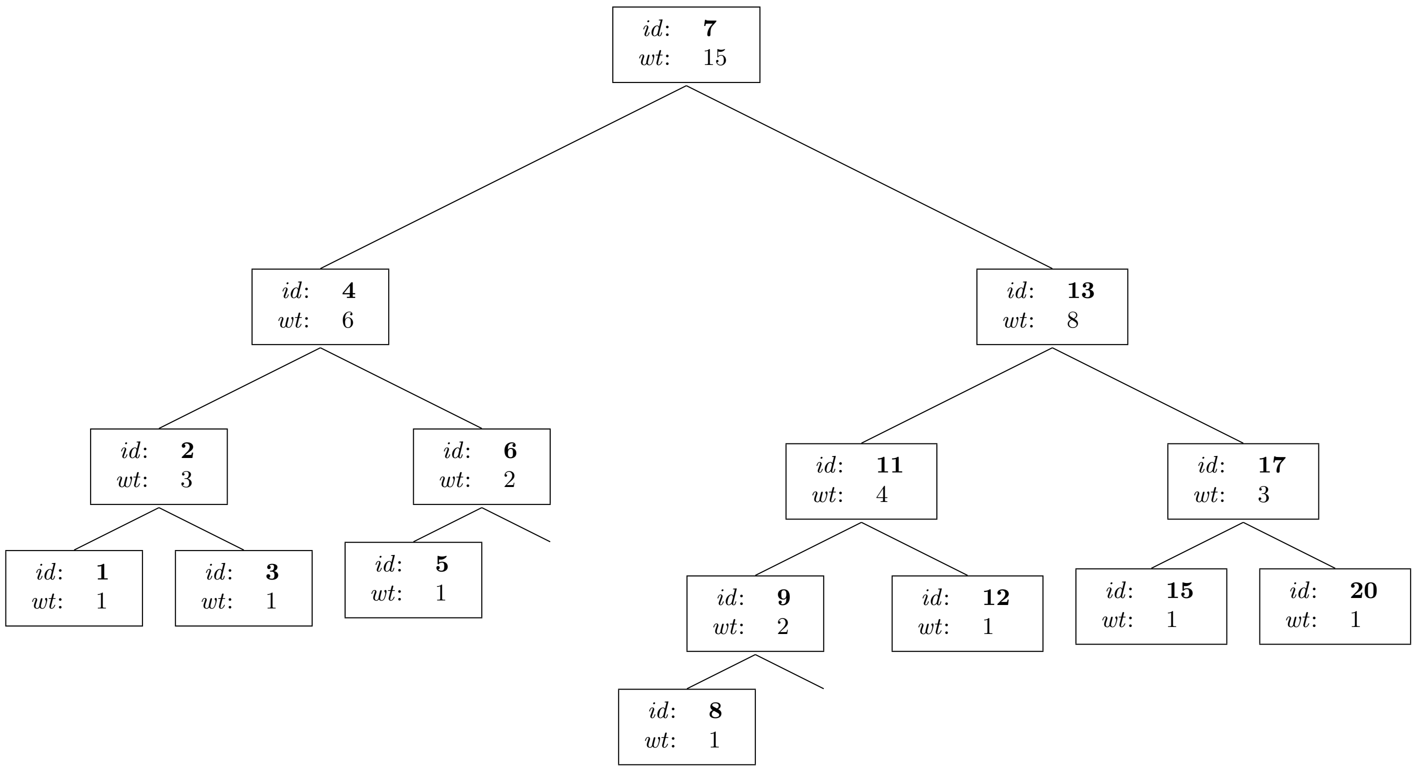

Figure 6: Here, we have rebalanced node 11. Now the tree is balanced again.

no left child, but a right child of weight 2.) In Figure 5, node 11 is unbalanced, while the other nodes are all balanced.

Problem 12.a. Your first task is to fix your solution for Problem 6. Address any comments from your tutor, and ensure that your basic binary search tree passes all the test cases specified in the solutions for Problem 6. Please use the attached unit test code. Submit again your basic binary search tree class.

Problem 12.b. Next, extend your basic binary search tree class in such a way as to ensure that the binary search tree always remains weight-balanced. For example, if your basic binary search tree is implemented in class `TreeNode`, then you should declare:

```
class BalancedTreeNode extends TreeNode {
...
}
```

In order to ensure that the tree remains weight-balanced, you must perform rebalancing operations after some insertions to restore the balance. In your extended class, create a new `insert` method (overriding the old `insert` method) that first inserts the key, and then checks whether the tree remains balanced. (Notice that after a node v is inserted, the only nodes that may need to be rebalanced are on a leaf-to-root path from v to the root.)

When your `insert` method discovers a node needs to be rebalanced, the rebalancing operation on a node v should operate as follows:

- If v is already balanced, do nothing.
- If v is unbalanced, then use the `getTreeKeys` function to get a sorted list of all the keys in the sub-tree rooted at v.
- Find the median key in the (sorted) list. Set the key of v equal to the median key in the list.
- Use the `buildTree` routine to build a new tree from the keys in the sorted list that precede the median. Set this new tree to be the left sub-tree of v.
- Use the `buildTree` routine to build a new tree from the keys in the sorted list that come after the median. Set this new tree to be the right sub-tree of v.

Notice that you only rebalanced nodes that are already unbalanced. When this rebalancing operation is complete, node v is weight-balanced again. Make sure that the weights are all correctly updated during an insertion.

In Figure 1, there is an example of a weight-balanced tree. In Figure 2, we are in the process of inserting node 15. Node 15 has been inserted as a child of node 13. This insertion has led node 12 to be unbalanced. We thus rebalance node 12, which leads to Figure 3. In Figure 4, we have inserted two further nodes: 17 and 20. Both have been inserted as children of 15. This again has led to the tree being unbalanced: nodes 11, 13, 15 are all unbalanced. In cases such as this, your program may balance these three nodes in any order. (Note that it is more efficient to balance the highest node first: if your rebalance 11, then you will not need to rebalance 13 or 15. However, it is simpler to implement the `insert` such that it rebalances the lowest nodes first. In this case, you walk down the tree to perform the insertion, and you walk back up the tree in order to update the weights and perform the rebalancing. Either solution is acceptable.) In Figure 5, we have rebalanced node 13, yet the tree is still unbalanced. In Figure 6, we have rebalanced 11 and the tree is again in balance.

Submit for this problem your extended class for a weight-balanced tree.

Problem 12.c. Write a set of tests that verifies the operation of your weight-balanced tree. First, write (in English) a list of test-cases, indicating the properties and cases being checked by each test. (For example, your test code might try inserting all the integers from 1 to 100 (in either forward or reverse order), and ensuring that the resulting tree is weight-balanced.) Then, implement your tests in Java. Submit for this part both a textual description of your tests, along with test code that verifies correct operation.

Problem 12.d. Profile your code for several values of n (i.e., inserting different numbers of elements into the tree). If $T(n)$ is the measured running time for n elements, graph $2^{T(n)/n}$, using the data from your profiler executions. Is the result linear? What does this mean? (Note that you may or may not get good results for this part, depending on how well your profiler is working.)

Problem 13. (Analyzing Weight-Balanced Trees: Bonus Problem)

In this problem, we will analyze the performance of your weight-balanced tree in order to show that it ensures good performance. We will use a type of *amortized analysis* known as the *accounting method*. Throughout this problem, assume that the weight-balanced binary search tree is implemented as described in Problem 12.

Problem 13.a. Prove that every search operation runs in $O(\log n)$ time by showing that a weight-balanced tree has height $< 2\log n$.

Problem 13.b. For the purpose of this problem, we will assume that every node in the tree has a *bank account* which can hold (virtual) dollars. Initially, every node begins with zero dollars in its account.

Whenever a node is inserted, it is provided with $c\log n$ dollars to spend (for some constant c). During the insertion, it will deposit these dollars in the bank accounts of various nodes.

Whenever a node is rebalanced, it will spend the money that has accumulated in its bank account. For every step of the rebalancing operation, it will spend one dollar. (You will show later that there is always enough money to pay for the rebalancing operation.) Thus if a rebalance operation on node v takes ℓ steps, then we deduct ℓ dollars from the node v's account.

Prove that *if the bank accounts always remain* ≥ 0, then the total cost for inserting k keys into an empty tree (summing the number of steps for each of the individual insertions) is $O(k\log n)$.

Problem 13.c. How many steps (asymptotically) does it take to rebalance a node with weight w?

Problem 13.d. Assume that after we rebalance a node v, it has no money left in its bank account, i.e., its account is zero. Notice that after a rebalance, the tree rooted at v is (almost perfectly balanced (due to the `buildTree` routine). Assume that next time node v is rebalanced, it has weight w. Show that at least $(w-1)/3$ nodes were inserted at v in between the two rebalance operations.

Problem 13.e. Assume that when a node is inserted, it begins with a budget of $12c\log n$. As the insert proceeds, it deposits 6c dollars in the bank account of each node on the path from root to leaf that is traversed during the insertion. Prove that the bank account of each node is always ≥ 0 (i.e., there is always enough money at v to pay for the rebalancing of v).

Problem 13.f. Conclude your proof by arguing that inserting k keys takes $O(k\log n)$ steps (including the cost of rebalancing during insertions).