CPSC 331 — Assignment #2
AVL Trees

About This Assignment

This assignment can be completed by groups of up to three students and it is due by 11:59pm on Monday, November 5.

AVL Trees: Definitions and Fundamental Properties

Consider a binary search tree T. The ***height*** of a node in T is the *depth of the subtree* with this node as root. For example, the heights of the nodes in the tree in Figure 1 (on page 2) are as shown in the picture.

The ***balance factor*** of a node in a binary search tree T is the difference between the depth of the *left* subtree of this node (which is -1, if the left subtree is empty) and the depth of the *right* subtree of this node (which is -1, if the *right* subtree is empty).It follows by this definition that all *leaves* of a binary search tree have balance factor 0.

The balance factors of the nodes in the tree in Figure 1 are as shown in Figure 2 (on page 2).

A binary search tree is an ***AVL Tree*** if the balance factors of all of its nodes are either -1, 0 or 1 — so that the absolute value of the difference in heights of the left and right subtrees of any node is at most one.

One can see by examination of the Figure 2 that the binary search tree shown in this figure is an AVL tree. On the other hand, the binary search tree shown in Figure 3 (on page 3) is *not* an AVL tree because the node storing key 6 has balance factor -2.

Note that, since an AVL tree is just a regular binary search tree that satisfies an additional property, the algorithm to ***search*** for a given key in a binary search tree can be used to search for a given key in an AVL tree, without change. You should recall that the number of steps used

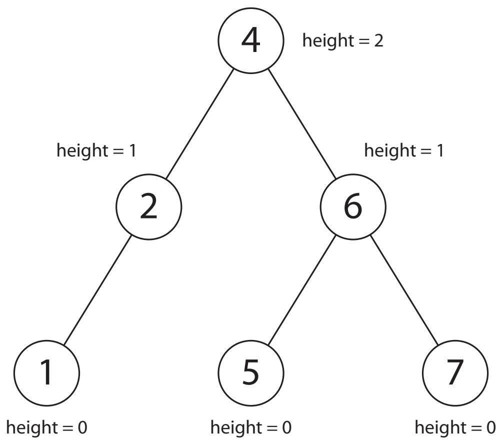

Figure 1: Heights of Nodes in a Binary Search Tree

by this algorithm is linear in the depth of the tree, in the worst case. As defined here, that is the same as the *height* of the root of the tree.

Now, for $i \geq 0$, let s_i be the ***minimum size*** of an AVL tree whose depth is i. Since the only

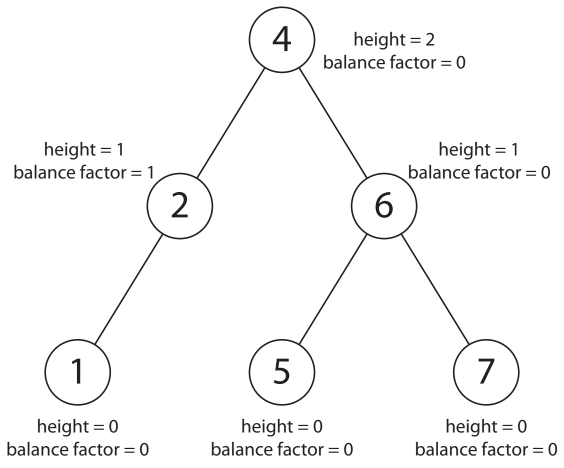

Figure 2: Balance Factors of Nodes in a Binary Search Tree

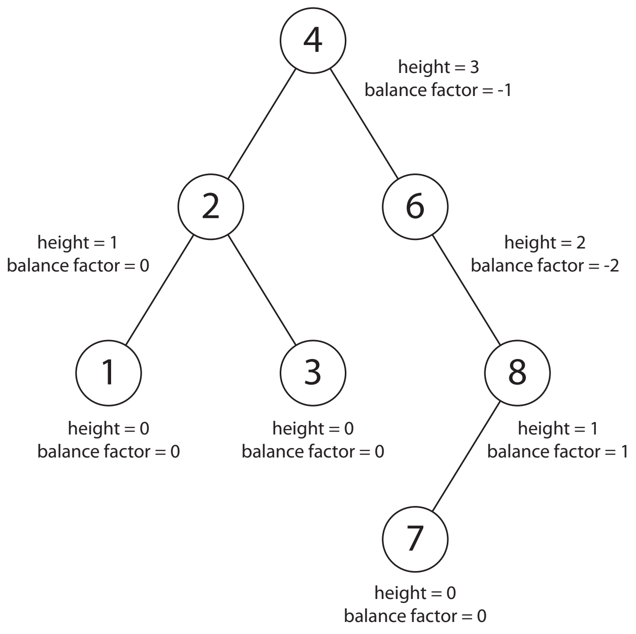

Figure 3: A Binary Search Tree That is Not an AVL Tree

binary search tree (or AVL tree) with depth -1 is the empty tree,

$$s_{-1} = 0. \tag{1}$$

Similarly, the only binary search trees (or AVL trees) with depth 0 are trees whose root is a leaf, so that they only have one node:

$$s_0 = 1. \tag{2}$$

Finally, suppose that i is an integer such that $i \geq 0$ and consider an AVL tree with depth $i+1$. One can see by the definition of an AVL tree that the left and right subtrees of the root are AVL trees as well (since the definition of an AVL tree concerned the balance factors of *all* of the nodes in the tree). Now, since the entire tree has depth $i+1$, one of the left or right subtrees must have depth i, and the other must have depth *at most* i. On the other hand, since the balance factor of the root is either -1, 0 or 1, a consideration of these cases confirms that the depth of the other subtree of the root is either i or $i-1$.

If the other subtree has depth $i-1$ then there are at least $s_{i-1} + s_i + 1$ nodes in the entire tree. On the other hand, if the other subtree has depth i then there are at least $2s_i + 1$ nodes in the entire subtree. Thus if $i \geq 0$ then

$$s_{i+1} \geq \min(s_{i-1} + s_i + 1, 2s_i + 1). \tag{3}$$

Now recall the ***Fibonacci numbers:*** For $i \geq 0$,

$$F_i = \begin{cases} 0 & \text{if } i = 0, \\ 1 & \text{if } i = 1, \\ F_{i-2} + F_{i-1} & \text{if } i \geq 2. \end{cases} \tag{4}$$

1. Use the inequalities and equations at lines (1)–(4), above, to prove that

$$s_i \geq F_{i+1} + F_{i+2} - 1 \qquad \text{for every integer } i \geq -1. \tag{5}$$

You will need to use mathematical induction on i — but the proof is not particularly long or hard to discover!

Now — as noted earlier in the course

$$F_i = \frac{1}{\sqrt{5}} \left(\left(\frac{1+\sqrt{5}}{2} \right)^i - \left(\frac{1-\sqrt{5}}{2} \right)^i \right) \tag{6}$$

for every integer $i \geq 0$. Notice that

$$\left| \frac{1+\sqrt{5}}{2} \right| \approx 1.618 > 1 \quad \text{and} \quad \left| \frac{1-\sqrt{5}}{2} \right| \approx 0.618 < 1.$$

These can be used, along with the inequality at line (5) and equation (6), to prove that

$$s_i \geq \frac{1}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2} \right)^{i+2} \tag{7}$$

for all sufficiently large integers i. It follows that if T is an AVL tree with size n and depth d then, for sufficiently large d,

$$n \geq s_d \geq \frac{1}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2} \right)^{d+2}$$

and straightforward algebraic manipulations (beginning with a consideration of the logarithm of both sides of the above inequality) are sufficient to establish that

$$d \in O(\log_2 n).$$

Indeed, the maximum depth of any AVL tree with size n is ***slightly smaller*** than the maximum depth of any ***red-black tree*** with size n — AVL trees are slightly more "balanced," in the worst case than red-black trees.

Of course, this information is only useful if it is possible to perform ***insertions*** and ***deletions*** in AVL trees (producing AVL trees as a result) as well.

1 Rotations

Please review the definitions of ***rotations*** included in the lecture notes on red-black trees: Exactly the same rotations are used to perform insertions and deletions in AVL trees too.

2 Insertions

Suppose we wish to ***insert*** a key `k` into an AVL tree. Just as for red-black trees, the insertion algorithm for AVL trees should *begin* with an application of the algorithm for insertion into binary search trees.

If `k` is already stored in the tree then, when a `Dictionary` is being represented and this is part of a `set` operation, then the corresponding value at the node should simply be changed, as needed, and no other steps are required: The load factors of nodes have not been changed. Similarly, if an ordered set is being represented then an exception should be thrown and the tree should not be changed at all — nothing more needs to be done in this case either.

Otherwise a new leaf, storing `k`, is included in this binary search tree.

2. Explain ***briefly*** why the only nodes in this tree whose heights or balance factors might have changed lie on the path from the new leaf (storing `k`) up to the root of this tree.
3. Explain ***briefly*** why the balance factors of the nodes on the above path are all either 2, 1, 0, -1, or -2.
4. Explain why the following is also true: If v is some node on this path in the tree, and the height of v has not been changed, then the heights and balanced factors of all the nodes on the path that are ***above*** v (that is, closer to the root) have not been changed, either.

Sometimes you are lucky and the result is still an AVL tree. In general, though the binary search tree resulting from this first part of the operation might ***not*** be an AVL tree: Some of the nodes on the path, mentioned above, really *might* have balance factor either 2 or -2.

Let α be the ***deepest*** node on this path whose balance factor is either 2 or -2 — noting that

- it is possible to find α by beginning at the leaf storing `k`, and moving up toward the root, recomputing heights and balance factors until either α is discovered or the root is reached; and
- if you discover that no such node α exists then you will have confirmed that the tree *is* still an AVL tree — so that nothing more must be done.

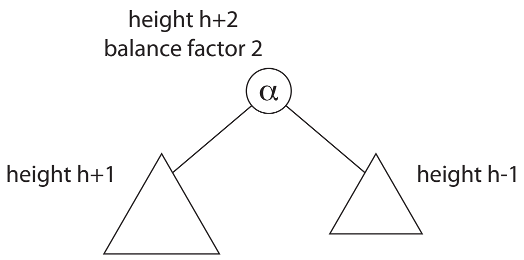

Figure 4: Case: Node α has Balance Factor 2

Suppose that the tree is *not* already an AVL tree, so that some such node α exists. Either the balance factor of α is 2, or it is -2.

If the balance factor of α is 2 then, since the heights of each of the left and right subtrees are each at least -1, the height of α is currently equal to $h+2$ for some integer $h \geq 0$. The *left subtree* of α has height $h+1$ and the *right subtree* of α has height $h-1$, as shown in Figure 4. Now, using the fact that this tree was an AVL tree ***before*** this operation, and that heights of trees cannot have *decreased* because of it, we may conclude that α must have had balance factor 1 and height $h+1$ before this operation. The height of the right subtree cannot have changed, and the height of the *left* subtree must have been increased from h to $h+1$ as a result of this operation.

Let β be the *left child* of α. Note that the heights of one of β's subtrees must have been increased from $h-1$ to h as a result of this insertion operation, since β had height h before the operation and height $h+1$ after it.

Since β has height h before the operation, the other subtree of β (which was not changed by the insertion) could not have had height more than $h-1$ before the operation. On the other hand, it cannot have height *less* than $h-1$ before or after the operation either — for, otherwise, the balance factor of β would be either -2 or 2 after the operation — contradicting the choice of α as the ***deepest*** node in the tree for which this is true.

Thus ***both*** subtrees of β had height $h-1$ before this operation, and β had balance factor 0 before it.

The heights of *one* of the subtrees of β increased from $h-1$ to h as a result of the insertion. The balance factor of β *after* the operation is 1 if the height of the left subtree increased, and the balance factor of β after the operation is -1 if the height of the right subtree increased, instead.

Let us first consider the case that the height of the *left* subtree of β has increased, so that β

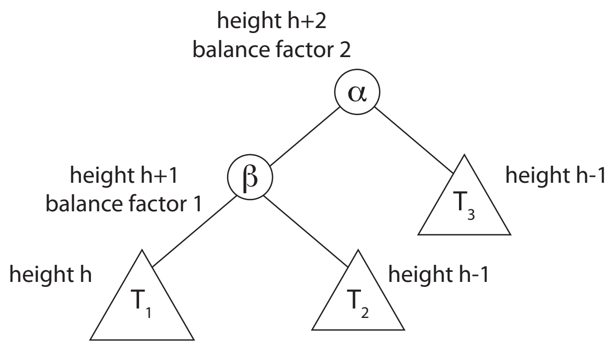

Figure 5: The “Left-Left” Case: After Insertion but Before Adjustment

has balance factor 1 after the insertion. This ***left-left case*** is as shown in Figure 5, above. In this case, a ***right rotation at α*** produces the binary search tree shown in Figure 6.

5. Comparing Figures 5 and 6 as needed, confirm that node α has height h and balance factor 0 after the rotation that has now been described.
6. Use this to confirm that the node β has height $h+1$ and balance factor 0 after this rotation as well.
7. Recalling that the node α had height $h+1$ before this rotation, ***briefly*** explain why this binary search tree is an AVL tree, once again, after this rotation.

Suppose, instead,that the height of the *right* subtree of β has increased, so that β has balance

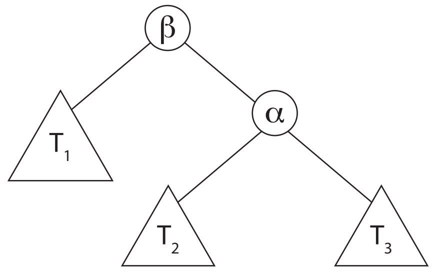

Figure 6: The “Left-Left” Case After Adjustment

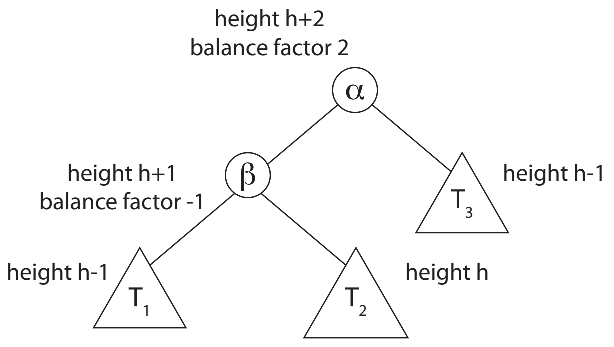

Figure 7: The "Left-Right" Case: After Insertion but Before Adjustment

factor -1 after this operation. This ***left-right case*** is as shown in Figure 7.

Since $h \geq 0$ and the right subtree of β has height h, this subtree is nonempty. Let γ be the right child of β, so that the subtree with root α is as shown in Figure 8.

Once again, the fact that α is the ***deepest*** node in this binary search tree whose balance factor is not in $\{-1, 0, 1\}$ can be used to argue that the subtrees T_{2a} and $T_{2,b}$ each have height either $h-2$ or $h-1$, and (since γ has height h) at least one of them has height $h-1$.

With that noted, suppose that we continue by performing a ***left rotation*** at β, followed by a

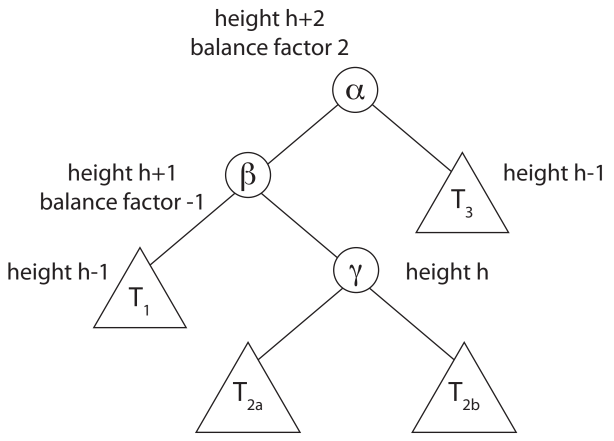

Figure 8: The "Left-Right" Case, Expanded

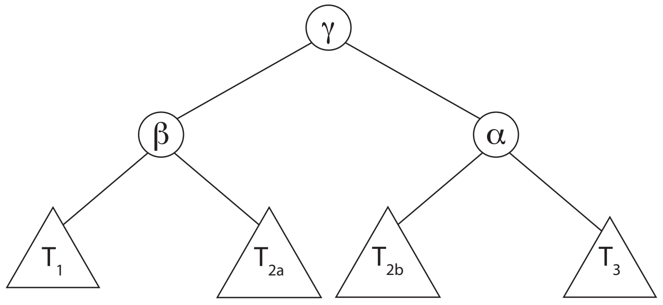

Figure 9: The “Left-Right” Case After Adjustment

right rotation at α. The resulting tree is as shown in Figure 9.

8. Use the above information (including a comparison of Figures 8 and 9, as needed) to confirm that α and β each have balance factor in $\{-1, 0, 1\}$ and height h after this adjustment.

9. Explain why γ has height $h + 1$ and balance factor 0 after this operation. Using this, explain why the resulting tree is an AVL tree after this operation.

10. Now consider the case that α has balance factor -2, so that the subtree with root α is as shown in Figure 10. Describe two more cases (that should probably be called the ***right-right case*** and the ***right-left case*** that might arise, corresponding this one, and describe the adjustments that can be used to produce AVL trees when these cases arise.

    Note: This should not be difficult!

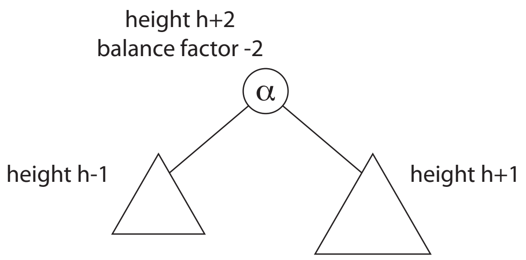

Figure 10: Case: Node α has Balance Factor -2

3 Deletions

Suppose we wish to ***delete*** a key `k` from the set stored in an AVL tree. Just as for red-black trees, the deletion algorithm for AVL trees should *begin* with an application of the algorithm for deletion from binary search trees.

If `k` is not stored in the tree then an exception should be thrown and the tree should not be changed — so that it will still be an AVL tree after the operation, and nothing more needs to be done.

Otherwise some node `y` is deleted from the binary search tree and another node `x` is promoted to replace it.

Note: It will probably be helpful to review the lecture notes about both

- deletions from binary search trees, and
- deletions from red-black trees (which refers to exactly the same nodes `y` and `x`)

before proceeding — `y` is ***not*** always the node that stored the key to be deleted, and you will only be able to understand (and correctly implement) the algorithm for deletion from AVL trees if you can correctly identify these nodes.

It is possible to prove that following properties — which resemble properties you have already been asked to prove, for insertions:

- The only nodes in this tree whose heights or balance factors might have changed lie on the path from the promoted node `x` up to the root of this tree.
- The balance factors of the nodes on this path are either -2, -1, 0, 1 or 2.
- If v is some node on this path in this tree, the height of v has not been changed so far, ,and its balance-factor is -1, 0 or 1 — so that it is not necessary to *change* its height in order to correct its balance-factor — then the heights and balance factors of all the nodes ***above*** v on this path have not been changed.

Let α be the ***deepest*** node on this path whose balance factor is either 2 or -2 — noting that

- it is possible to find α by beginning at the promoted node `x`, and moving up toward the root, recomputing heights and balance factors until either α is discovered or the root is reached; and
- if you discover that no such node α exists then you will have confirmed that the tree *is* still an AVL tree — so that nothing more must be done.

Once again, suppose that α has balance factor 2, so that the subtree with root α is as shown in Figure 4 on page 6. In this case — since this was caused by a ***deletion*** instead of an ***insertion*** — we know the following.

- The *left subtree* of α was ***not*** changed by the first part of this operation, so that it also had height $h+1$ before the deletion began.
- On the other hand, the *right subtree* of α ***was*** changed — it had height h before this deletion began.
- The ***height*** of α was not changed — it also had height $h+2$ (and balance factor 1) before this deletion began.

As in the discussion of insertions, let β be the left child of α. We do not know as much about β as we did when considering insertions, because the ***right*** subtree of α was changed by the first part of the deletion operation instead of the left. However, since α has (still) been defined to be the ***deepest*** node in the tree whose balance factor is either 2 or -2, we know that the balance factor of β is either 1, 0, or -1.

Suppose that the balance factor of β is 1, so that the subtree with root α is as shown in Figure 5 on page 7. One can show, once again, that a ***right rotation at α*** is sufficient to ensure that all of the nodes that are now in the subtree with root β have balance factors -1, 0, or 1.

Unfortunately ***the entire tree still might not be an AVL tree*** because the height of β *after* this adjustment is ***one less than*** the height that α had before it! Therefore, there might still be one or more nodes ***above*** β in the tree with balance factors 2 or -2.

Suppose instead that the balance factor of β is -1, so that the right subtree of β is nonempty; once again, let γ be the right child of β, so that the subtree with root α is as shown in Figure 8 on page 8. As in the case for insertions, a ***left rotation at β***, followed by a ***right rotation at α*** is sufficient to ensure that all of the nodes that are now in the subtree with root γ have balance factors -1, 0, or 1.

Once again, though, ***the entire tree still might not be an AVL tree*** because the height of γ *after* this adjustment is ***one less than*** the height that α had before it! Therefore, there might still be one or more nodes ***above*** γ in the tree with balance factors 2 or -2.

Finally, suppose that the balance factor of β is 0, so that the subtree with root α is as shown in Figure 11 on page 12.

11. Suppose that you perform a ***right rotation at α*** — the same adjustment as for the "Left-Left Case" — so that the subtree with root β after this adjustment is as shown in Figure 6 on page 7.

    Explain ***briefly*** why α has height $h+1$ and balance factor 1 after this adjustment.
12. Explain ***briefly*** why β has height $h+2$ and balance factor -1 after this adjustment.
13. Explain ***briefly*** why this (entire) binary search tree is an AVL tree after this adjustment.

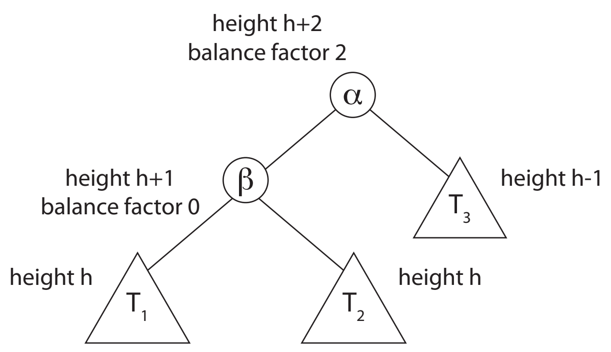

Figure 11: The “Left-Equal” Case

14. Now consider the case that α has balance factor -2, instead, so that the subtree with root α is as shown in Figure 10 on page 9. ***Briefly*** describe another three cases (that might be called the “Right-Right Case,” the “Right-Left Case,” and the “Right-Equal Case”) corresponding to this, along with the adjustments that should be made for each.

4 Implementing an AVL Tree

15. ***Briefly*** describe the structure of an algorithm for an ***insertion*** into an AVL tree. You may do this by writing a few paragraphs explaining what the algorithm does or by giving pseudocode for it. You should assume that the ***heights*** and ***balance factors*** of all the nodes in the AVL tree are available before the operation (and that these should also be updated during it).

    Note: If you have worked through the previous part of the assignment then this should not be difficult — and you should find (and report) that an algorithm for an insertion into an AVL tree is at least a bit ***simpler*** than the algorithm for an insertion into a red-black tree.

16. ***Briefly*** describe the structure of an algorithm for a ***deletion*** from an AVL tree. As above, you may do this by writing a few paragraphs explaining what the algorithm does or by giving pseudocode for it. You should assume that the ***heights*** and ***balance factors*** of all the nodes in the AVL tree are available before the operation (and that these should also be updated during it).

    Note: If you have worked through the previous part of the assignment then this should not be difficult — and you should find (and report) that an algorithm for a deletion from an AVL tree is similar to, but at least a bit ***simpler*** than the algorithm for a deletion from a red-black tree.

17. The following programs are now available

    - `Dictionary.java`: The same interface as was provided with Lecture #9, when binary search trees were introduced.
    - `AVLDictionary.java`: An incomplete class that uses an AVL tree to implement a `Dictionary`. This is based heavily on the `BSTDictionary.java` program that was provided as a supplement for Lecture #10.

    In order to complete the `AVLDictionary.java` program you will need to complete the following methods:

    - `search`: A method that should replace the method provided by the `BSTDictionary` class with the same name.
    - `rotateLeft`: A method that should carry out a left rotation at a given node. Since this method should be private and you should only be calling it yourselves you may assume that left rotations are always possible when this is called.
    - `rotateRight`: A method that should carry out a right rotation at a given node. As for left rotations you may assume that right rotations are always possible when this is called.
    - `change`: A method that should replace the method provided by the `BSTDictionary` class with the same name.
    - `deleteFromSubtree`: A new method that should replace the method provided by the `BSTDictionary` class with the same name.
    - `deleteNode`: A new method that should replace the method provided by the `BSTDictionary` class with the same name.
    - `successor`: Once again, a new method that should replace the method provided by the `BSTDictionary` class with the same name.

    You might also find that coding is simplified if a ***small*** number of additional methods (called by one or more of the above ones) is also included. Please ***do not*** change the other utility methods that have been fully implemented, because they will be used to test your code.

    Provide a complete `AVLDictionary.java` class for assessment. Your written solutions should also include a reasonably ***brief*** description of what you changed, in the `BSTDictionary.java` program, when completing the `AVLDictionary.java` program.