Assignment 2

Betty Zhang - 30040611
William Chan - 30041834
Umair Hassan - 30047693
CPSC 331

November 11, 2018

1. Use the inequalities and equations at lines (1) − (4) to prove that

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

Proof: We prove that $s_i \geq F_{i+1} + F_{i+2} - 1$ by induction on i. The strong form of mathematical induction will be used, and the cases that $i = -1$ and $i = 0$ will be considered as basis.

Basis:

Suppose $i = -1$. Then

$$s_i = s_{-1} = 0$$
$$F_{i+1} = F_0 = 0$$
$$F_{i+2} = F_1 = 1$$
$$F_{i+1} + F_{i+2} - 1 = F_0 + F_1 - 1 = 0$$
$$0 = 0$$

so $s_i = s_{-1} = F_0 + F_1 - 1 = F_{i+1} + F_{i+2} - 1$ which is the same as $s_{-1} \geq F_0 + F_1 - 1$.
Therefore the statement $s_i \geq F_{i+1} + F_{i+2} - 1$ is satisfied in the case where $i = -1$.

Suppose $i = 0$. Then

$$s_i = s_0 = 1$$
$$F_{i+1} = F_1 = 1$$
$$F_{i+2} = F_2 = F_0 + F_1 = 0 + 1 = 1$$
$$F_{i+1} + F_{i+2} - 1 = F_1 + F_2 - 1 = 1$$
$$1 = 1$$

so $s_i = s_0 = F_1 + F_2 - 1 = F_{i+1} + F_{i+2} - 1$ which is the same as $s_0 \geq F_1 + F_2 - 1$.
Therefore the statement $s_i \geq F_{i+1} + F_{i+2} - 1$ is satisfied in the case where $i = 0$.

Inductive Step: Let k be an integer such that $k \geq 0$.

Inductive Hypothesis: Suppose i is an integer such that $-1 \leq i \leq k$,
then $s_i \geq F_{i+1} + F_{i+2} - 1$

Inductive Claim: $s_{k+1} \geq F_{k+2} + F_{k+3}$ - 1

Since $s_{i+1} \geq min(s_{i-1} + s_{i+1}, 2s_{i+1})$ for $i \geq 0$, we will use two cases, if the minimum value is the first or second equation. If they are both equal, we can use either cases for the proof.

Case 1: $s_{k-1} + s_k + 1 \leq 2s_k + 1$,

$$\text{So } s_{k+1} \geq s_{k-1} + s_k + 1$$
$$\text{Since } k \geq 0,\ k - 1 \geq -1, \text{ we can use the inductive hypothesis to get}$$
$$s_{k-1} + s_k + 1 \geq F_k + F_{k+1} - 1 + F_{k+1} + F_{k+2} - 1 + 1$$
$$= F_{k+1} + F_k + F_{k+2} + F_{k+1} - 1$$
$$\text{Since } k \geq 0,\ k + 2 \geq 2, \text{ we can use the given Fibonacci Numbers to get}$$
$$F_k + F_{k+1} = F_{k+2} \text{ and } F_{k+1} + F_{k+2} = F_{k+3}$$
$$\text{Thus,}$$
$$s_{k-1} + s_k - 1 \geq F_{k+2} + F_{k+3} - 1$$
$$s_{k+1} \geq F_{k+2} + F_{k+3} - 1$$

Case 2: $s_{k-1} + s_k + 1 > 2s_k + 1$,

$$\text{So } s_{k+1} \geq 2s_k + 1$$
$$\text{Using our Inductive Hypothesis,}$$
$$2s_k + 1 \geq 2(F_{k+1} + F_{k+2} - 1) + 1$$
$$= F_{k+2} + (F_{k+1} + F_{k+2}) + F_{k+1} - 1$$
$$\text{Since } k + 2 \geq 2, \text{ we can use the given Fibonacci Numbers to get}$$
$$= F_{k+2} + F_{k+3} + F_{k+1} - 1$$
$$\text{Since } k \geq 0, \text{ given the definition of the Fibonacci's Numbers, } F_{k+1} \geq 0, \text{ so}$$
$$\geq F_{k+2} + F_{k+3} - 1$$
$$s_{k+1} \geq F_{k+2} + F_{k+3} - 1$$

By mathematical induction on i, $s_i \geq F_{i+1} + F_{i+2} - 1$ for all $i \geq -1$.

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.

Answer: The height of each node is calculated by the number of nodes on longest path from that node to leaf node. If a new leaf is inserted and it does not have a sibling, then the height of the new leaf's parent will increase by 1. Likewise, the height of every node that lies on the path from the new leaf up to the root may increase by 1 if the path between such node and the newly added leaf is the longest path. All nodes not in the path from the new leaf to the root do not change because the number of nodes in their path hasn't changed. Balance factor is the difference between the height of the left subtree and height of the right subtree, adding a leaf in either of the subtrees would change their height and therefore change the balance factor. If the leaf added is not in either of the subtrees of the node(hence not changing the height of any of the nodes in the subtrees), the balance factor does not change.

3. Explain briefly why the balance factors of the nodes on the above path are all either 2, 1, 0, -1, or -2.

Answer: Given the definition and invariant of an AVL trees, for all AVL trees not in the middle of an operation, the balance factors of all of its nodes are either -1, 0 or 1. When a new leaf is inserted, the height of the nodes on the above path can increase by 1 or can remain the same. If a new leaf was inserted to left subtree of a node, the balance factor could increase by 1, because since the balance factor is the left subtree subtract right subtree, and the largest the balance factor before an operation can be is 1, the maximum possible balance after the insertion is 2. If new leaf was inserted to right subtree of a node, the balance factor could decrease by 1, the lowest the balance factor before an operation can be is -1, so the lowest the new balance factor can be is -2. If the balance factor is 2 or -2, we use rotations to make the balance factors back to -1, 0 and 1, so that the nodes can never get a balance factor of -3 or 3 since it can only vary by 1, and after an operation we balance height of the tree.

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.

Answer: The height of a node can be calculated by adding 1 to its biggest subtree (the subtree with the larger height), if neither the right or left subtrees height is changed then the node's height is not changed neither. If the height of v was not changed and the height of its sibling was not changed then the parent nodes height was not changed, and all the nodes above are not changed because none of the nodes have been changed below them. The balance factor is similar, since the balance factor relies on the height of the subtrees. If the height and balance factor was not changed, neither did the node change.

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.

Answer: The height of a node is the depth of the subtree with this node as the root. So if the depth of its subtrees with the node as the root aren't changed, its height is still the same. No new leaf is added nor any operations were done on the subtrees for the nodes T_2 and T_3, if you look at them as the root for their respective subtrees. Since T_2 and T_3 as the roots of their subtrees are not changed, their heights are the same as before and after the rotation done on the whole tree. The height of node α is the higher height of the left subtree (T_2) or right subtree (T_3) + 1. Since T_2 and T_3 are both $h-1$ before, and do not change their height in this rotation, the height of node α is $h-1+1$ which is h. The balance factor is the height of the left subtree - the height of the right subtree. The heights for both

T_2 and T_3 are $h-1$, and so the balance factor is $(h-1)-(h-1)=0$.

6. Use this to confirm that the node β has height $h+1$ and balance factor 0 after this rotation as well.

Answer: Similar to question 5, for node T_1, the rotation did not affect the nodes in its subtrees relative to T_1 or move anything below it, so its height is still h after the rotation. Since we know after the rotation T_1 and α both have height h and β is the parent of both, β's height is just one more than either of the subtrees, so it is $h+1$. Like in question 5, the subtrees T_1 and α have the same height h, so the balance factor for the parent β is $h-h=0$.

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.

Answer: To be an AVL tree, all the nodes must have a balance factor of -1, 0, or 1. β and α have balance factors of 0 after the rotation. Since we know a leaf was added and the balance factor of β after the operation is 1, so then the height of the left subtree (T_1) increased. Since T_2 and T_3 did not have a leaf added nor have nodes in their subtrees change position the after the rotation, the nodes in the subtrees of T_2 and T_3 still all have balance factors of -1, 0, or 1. For T_1 we are assuming T_1 has been height balanced after the insertion because α is the deepest node on this path whose balance factor is either 2 or 2, then since the rotation at α did not affect the nodes under T_1, they should all be balanced still.
The height of the problem nodes above β would now also be adjusted (decreased by one) and the balance factors of all problem nodes is now correct: now that β is the new root of the subtree that α was previously the root of and β has the height of $h+1$, which is the same as what the subtree had before the insertion of the new node and when the tree was still an AVL tree (when α has the height of $h+1$)
Note: according to the diagram and the details on insertion, α had height $h+2$ before the **rotation** and the height of $h+1$ before the insertion, which is different from what is stated in the question

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 of 1, 0, 1 and height h after this adjustment.

Answer: Since before the rotations γ had height h, the max height of T_{2a} is $h-1$, and T_1 had height $h-1$. since β is parent to both T_{2a} and T_1 after the rotations and nothing below T_{2a} and T_1 was changed by the rotations, β's height $= h-1+1=h$. Same thing with α, before T_3 and T_{2b} had height of $h-1$, did not change nodes below them, α is the parent of both after, so αs height $= h-1+1=h$. β is parent to T_1 and T_{2a} after, T_1 and T_{2a} have height of $h-1$, β's balance factor = height of T_1 - height of T_{2a}= $(h-1)-(h-1)=0$. Same with α. α is parent to T_{2b} and T_3 after, T_3 and T_{2b} have height of $h-1$, α's balance factor = height of T_3 - height of T_{2b} $= (h-1)-(h-1)=0$.

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.

Answer: From question 8 we found α and β have height h, γ is parent to them so γs height is just one more of either α or β, γ's height is $h+1$. γ's balance factor is height of β - height of $\alpha = h - h = 0$. In question 8 we found that α and β have balance factors of 0, so now all we need is for the subtrees to be balanced. Since α is the deepest node on this path whose balance factor is either 2 or 2 before the rotations, and T_1, T_{2a}, T_{2b}, and T_3 (T nodes) are below α, they must have had balance factors of -1, 0, or 1. Also all the nodes under the T nodes also are all balanced because they are also below α, and not affected by the rotations, their heights didn't change by rotations. All the nodes have balance factors of -1, 0, or 1 so this is an AVL tree.
The height of the problem nodes above γ would now also be adjusted (decreased by one) and the balance factors of all problem nodes is now correct: now that γ is the new root of the subtree that α was previously the root of and γ has the height of $h+1$, which is the same as what the subtree had before the insertion of the new node and when the tree was still an AVL tree (when α has the height of $h+1$)

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.

Answer: Say that you had an AVL tree and then you inserted a leaf. Let α be the deepest node with a balance factor of -2, and height $h+2$ after the insertion. Let T_1 be the left subtree of α with height $h-1$. Let β be the root of the right subtree of α with height $h+1$.
Right-Right Case: Let T_2 be the left subtree of β with height $h-1$. Let T_3 be the right subtree of β with height h. To height balance this tree do a left rotation at α.
Right-Left Case: Let T_2 be the left subtree of β with height h. Let T_3 be the right subtree of β with height $h-1$. To height balance this tree do a right rotation at β then a left rotation at α.

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 α 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.

Answer: After the rotation at α, α is the parent of the root of the T_2 (as its left child) and root of the T_3 (as its right child). T_2 and T_3 had heights of $h+1$ and $h-1$ respectively, and the rotation did not change any nodes under them relative to T_2 and T_3, so the heights are unchanged after the rotation. So α's height is just one more than T_2's height (because T_2 has higher height than T_3), so α's height $= h+1$. αs balance factor = height of T_2 - height of $T_3 = h - (h-1) = 1$.

12. Explain briefly why β has height $h + 2$ and balance factor 1 after this adjustment.

Answer: After the rotation, β is the parent of the root of T_1 (as its left child) and α (as its right child), and we know from question 11, α has height of $h + 1$. T_1 had height of h before the rotation, and since the rotation did not affect the nodes under T_1 relative to T_1, they are the same height afterwards. Since α has higher height than T_1, β's height is just one more than α's height, so β's height $= h + 1 + 1 = h + 2$. β's balance factor = height of T_1- height of $\alpha = h - (h + 1) = -1$.

13. Explain briefly why this (entire) binary search tree is an AVL tree after this adjustment.

Answer: We know from questions 11 and 12, β and α have balance factors -1 and 1 respectively. Before the rotation α was the deepest node with a balance factor of -2 or 2, and since T_1, T_2, T_3, and all their sub nodes were under α, they must all of had balance factors of -1, 0, or 1. The rotation did not affect the heights of T_1, T_2, T_3, and the nodes under them, the balance factors are the same after the rotation, which were balanced before. All the nodes in the tree have a balance factor of -1, 0, or 1, thus it is an AVL tree. Furthermore, after the rotation, $beta became the new root of the the subtree, and the height of β is h + 2, which is the same as the height of α before the deletion and when the AVL Dictionary invariant was stratified.

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.

Answer: Say that you had an AVL tree and then you inserted a leaf. Let α be the deepest node with a balance factor of -2, and height $h + 2$ after the insertion. Let T_1 be the left subtree of α with height $h - 1$. Let β be the the right subtree of α with height $h + 1$.
Right-Right Case: Let T_2 be the left subtree of β with height $h - 1$. Let T_3 be the right subtree of β with height h. To height balance this tree do a left rotation at α. Continue to trace the path, looking for balance factor of -2 or 2 and balancing them, until we reach the root or until we reach a right-equal case.
Right-left Case: Let T_2 be the left subtree of β with height h. Let T_3 be the right subtree of β with height $h-1$. To height balance this tree do a right rotation at β then a left rotation at α. Continue to trace the path, looking for balance factor of -2 or 2 and balancing them, until we reach the root or until we reach the right-equal case.
Right-Equal Case: Let T_2 and T_3 be subtrees of β both with height h. To height balance this tree do a left rotation at α.

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).

Answer: Assuming the AVL Dictionary invariants are satisfied and the heights and balance factors of all the nodes in this AVL tree are available before the operations, start the insertion with the "set" method, a key k of type K and a value v of type V is given as input. The algorithm should check if the tree is empty. In the case that the tree is empty, where root is *null*, then a new node storing the key k and value v will be created and become the root of the tree. Otherwise, continue to rest of algorithm with key k , value v and root x as input into the change method.
The "change" method will take a key k of type K, a value v of type V, and a node x as input.

Let h be the key stored at x.

- If k is less than h, then it is necessary to continue with the left subtree of the subtree with root x. If the left child of x is *null*, such that the left subtree is empty, then a new node storing k and v, whose left and right children are *null* and whose parent is x, should be added as the new left child of x. If a new node is inserted, a helper method, "insertion adjust", will be called with the inserted node as input, to adjust the tree so that the loop invariants (specifically the balance factors of each nodes) still satisfies after the insertion. The height of all the nodes in the AVL should be updated after the adjustment. Otherwise, the algorithm should be called recursively with k, v, and the existing left child of x as inputs.

- If k is greater than h, then it is necessary to continue with the right subtree of the subtree with root x. If the right child of x is *null*, such that the right subtree is empty, then a new node storing k and v, whose left and right children are *null* and whose parent is x, should be added as the new right child of x. If a new node is inserted, a adjustment helper method will be called with the inserted node as input, to adjust the tree so that the loop invariants (specifically the balance factors of each nodes) still satisfies after the insertion. The height of all the nodes in the AVL should be updated after the adjustment. Otherwise, the algorithm should be called recursively with k, v, and the existing right child of x as inputs.

- If k is equal to h, so that it is stored at x, the value stored at x should be changed to v. Let the newly inserted node be a.

The adjustment method will take an AVL node x as an input, in this case, the node x is the newly inserted node from the insertion method. The method will begin by updating the height of all the nodes in the subtree where x is the root. Starting from node x and moving up to the parent of x. The loop ends when the root of the tree is reached or when a problem node is identified and adjusted (using the break statement after the problem is node is fixed). Let α be the problem node where the balance factor of the node is not -1, 0 or 1. There are two cases for α:

- α has a balance factor of 2, which means that height of the the left subtree of α has increased by one after the insertion. Let β be the left child of α, there are two cases for β:
  - β has a balance factor of 1. In this case, we perform a right rotation on α to correct the AVL tree. We can now exit the loop.
  - β has a balance factor of 1. In this case, we perform a left rotation on β followed by a right rotation on α to correct the AVL tree. We can now exit the loop.
- In the case where α has a balance factor of -2, the height of the right child of α had increased by one after the insertion. Let γ be the right child of α, there are two cases for γ:
  - γ has a balance factor of 1. In this case, we perform a right rotation on γ followed by a left rotation on α to correct the AVL tree. We can now exit the loop.
  - γ has a balance factor of 1. In this case, we perform a left rotation on α to correct the AVL tree. We can now exit the loop.

After the adjustments are made and the heights of all nodes in the AVL tree is updated, the AVL Dictionary invariants should be satisfied once again with the new node inserted or value of the node with key k updated.

Pseudocode for the adjustment method for insertion:

```
insertionAdjust (AVLNode x){
        while (x is not null){
            update height of nodes in subtree of x
                if (the balance factor of x = 2){
                        y = the left child of x
                        if (the balance factor of y = 1){
                                left rotation on x
                        } else if (the balance factor of y = -1){
                                left rotation on y
                                right rotation on x
                        }
                        exit the loop (hence exiting the method)
                }else if (the balance factor of x = -2){
                        z = the right child of x
                        if (the balance factor of z = 1){
                                right rotation on z
                                left rotation on x
                        }else if (the balance factor of z = -1){
                                left rotation on x
                        }
                        exit the loop (hence exiting the method)

                }
                x = the parent of x
        }
}
```

The implementation of insertion for an AVL tree is simpler because it relies on the heights of the subtree to get its balance factor and there are only 4 cases in total. Deletion for red black is much more complicated because we have to consider 6 different cases (where two of them has 2 subcases)

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).

Answer: Assuming the AVL Dictionary invariants are satisfied and the heights and balance factors of all the nodes in this AVL tree are available before the operations. The algorithm begins by receiving a key k of type K through the "remove" method supplied by the dictionary. The key is then passed to the a sub method, "deleteFromSubtree", with the key k and root of the AVL dictionary as input.

The "deleteFromSubtree" method takes in a key k of type K and an AVL node x as input. The method performs similar to the the search algorithm: the key k is compared to h, if k is less than h, the method is called recursively with k and the left child of x as input. If k is greater than h, the method is called recursively with k and the right child of x as input. The node associate with k as the key does not exist if at some point the method has a null node for x as input, then a NoSuchElementException will the thrown. Otherwise, if at any point k is equal to h, we have found the node we must delete. The value of the node v of type V is stored before the deletion the value can be returned at the end of the algorithm. A "deleteNode"method is called with the AVL node that has to be deleted as input to perform the deletion.

The "deleteNode" takes an AVL node x as input, there are a few cases for x:

1. The node x is a leaf, so its left child and right child are both *null*. If the nodes parent is also *null*, then this node is the only node in the tree, in this case, the root of the tree is set to *null* and and the deletion is completed. If the parent is not *null*, and x is a left child, then the left child of the parent will now be set to *null*. If the parent is not *null* and x is a right child, the right child of the parent will now be set to *null*.

2. The node x has a *null* left child and a non-*null* right child. If x's parent is *null*, then the right child of x is promoted as the root of the tree and the parent of the promoted node is set to *null*. If x's parent is not *null* and x is a right child of its parent, then the right child of x promoted to be the right child of the parent. If x's parent is not *null* and x is a left child of the parent, the right child of x is promoted to be the left child of the parent. The parent of the promoted node is updated to x's parent as well.

3. The node x has a *null* right child and a non-*null* left child. If x's parent is *null*, then the left child of x is promoted to be the root of the tree, the parent of the promoted node is set to *null*. If x's parent is not *null* and x is a right child of its parent, then the right child of x promoted to be the left child of the parent. If x's parent is not *null* and x is a left child of the parent, the left child of x is promoted to be the left child of the parent. The parent of the promoted node is updated to x's parent as well.

4. Neither of x's children are *null*. Let y be the node strong the smallest key at the right subtree of x, this node will be the successor of x. Once y is located, the key and values

of y is copied to x, "replacing" the node x. The "deleteNode" method is then called again recursively with the successor y as input. Since y does not have a left child (as it is the smallest node in the subtree), this execution of the algorithm will either meet case 1 where x is a leaf or case 2 where x has a *null* left child and a non-*null* right child.

After x is deleted from the tree, the height of the nodes of the subtree where the parent is the root are then updated. An extra adjustment function is required with the promoted node as input (or the parent of the deleted node if no node was promoted) to make the proper adjustment so that the AVL Dictionary invariants are still satisfied.

The adjustment method deletion performs similarly to the adjustment for insertion, except in this case, the balance factor of every node in the path between the promoted node a and the root of the tree is checked for adjustment. As well, the height of each nodes affected by the adjustment (rotations) will be updated in each iteration of the loop.

Explanation for the adjustment method (pseudo-code for this method is also provided below): The method takes an AVL node x as an input. The method begins by checking the balance factor of x, and continues to move up to the parent of x until the root is reached. Adjustments are required when the balance factor of x is not -1, 0 or 1. There are two cases:

- When x has a balance factor of 2, which means that height of the the right subtree of x has decrease by one after deletion or an adjustment during this execution of the method. Let y be the left child of x, there are three cases for y:
  - y has a balance factor of 1. In this case, we perform a right rotation on x to correct the AVL tree.
  - y has a balance factor of 1. In this case, we perform a left rotation on y followed by a right rotation on x to correct the AVL tree.
  - y has a balance factor of 0. In this case, we perform a right rotation on x. Since it is shown in question 13 that this operation is sufficient to ensure the tree once again is an AVL tree, we can now update the height of all nodes in the tree and exit the method.
- In the case where x has a balance factor of -2, the height of the left child of x had decreased by one after the deletion or an adjustment of its descendants. Let z be the right child of x, there are three cases for z:
  - z has a balance factor of 1. In this case, we perform a right rotation on z followed by a left rotation on x to correct the AVL tree.
  - z has a balance factor of 1. In this case, we perform a left rotation on x to correct the AVL tree.
  - z has a balance factor of 0. In this case, we perform a left rotation on x. Since this operation result in a situation similar to the "left equal" case where the operation is sufficient to ensure the tree once again is an AVL tree, we can now update the height of all nodes in the tree and exit the method.

The heights of each nodes are updated after each adjustment and the loop continues moving up the AVL tree from the problem node until the root is reached. At the end of the execution, the AVL Dictionary invariants should be satisfied once again with the new node inserted or value of the node with key k updated.

Pseudo code for adjustment method of deletion:

```
deletionAdjust(AVLNode x){
        while (x is not null){
                if (the balance factor of x = 2){
                        y = the left child of x
                        if (the balance factor of y = 1){
                                left rotation on x
                        } else if (the balance factor of y = -1){
                                left rotation on y
                                right rotation on x
                        } else if (balance factor of y =0 ){
                    right rotation on x
                    update the height of the nodes
                    exit the loop and method
                }
                    }else if (the balance factor of x = -2){
                        z = the right child of x
                        if (the balance factor of z = 1){
                                right rotation on z
                                left rotation on x
                        }else if (the balance factor of z = -1){
                                left rotation on x
                        }else if (balance factor of z = 0){
                    left rotation on x
                    update height of the nodes
                    exit the loop and method
                }
                    }
                    update heights of nodes in the subtree of x
                    x = the parent of x
        }
}
```

The implementation of deletion for an AVL tree is simpler because it relies on the heights of the subtree to get its balance factor and there are only 6 cases in total. Deletion for red black is much more complicated because we have to consider 16 different possible combination of color and the black height of the problem node and it's parent, it's sibling and children of its sibling.

Extra: changes made in the AVLDictionary.java drom the BSTDictionary.java (the actual source code is submitted in D2l)

- new properties added to each node (already provided by the assignment): height as at attribute. height( ) and balanceFactor ( ), which returns the height and the balance factor of the AVL node
- new methods added:
  - rotateLeft (AVLNode x) : to perform left rotation on x
  - rotateRight(AVLNode x): to perform right rotation on x
  - updateHeight (AVLNode x): to update the height of nodes in the subtree where x is the root
  - insertionAdjust(AVLNode x): an adjustment method for insertion, used to check through the nodes in the path between the newly added node and the root to find problem that does not satisfy the required balance factor and make the proper adjustments
  - deletionAdjust(AVLNode x): an adjustment method for deletion, used to check through the nodes in the path between the newly added node and the root to find problem that does not satisfy the required balance factor and make the proper adjustments
- change to insertion method: the insertionAdjust method is called when a node is inserted
- change to deletion method: the deletionAdjust method is called when a node is deleted

References:
Lecture Notes #9 - 12
Tutorial Solutions #10 - 13
Assignment #2 documents
- Test java files
- AVLUtilities.java
BSTDictionary.java
https://www.geeksforgeeks.org/avl-tree-set-1-insertion/