CPSC 331: Assignment 2

Matthew Allwright	Karim Beyk	Seth Campbell
30037812	30027342	10152719

November 12, 2018

AVL Tree Defenition and Properties

Question 1: Prove that $s_i \geq F_{i+1} + F_{i+2} - 1$ for all integers $i \geq -1$.

Claim:

$s_i \geq F_{i+1} + F_{i+2} - 1$ for all integers $i \geq -1$.

Proof by Strong Induction on i:

Base Cases ($i = -1, 0$):
Let $i = -1$.
Then $s_i = s_{-1} = 0 \geq 0 = 0 + 1 - 1 = F_0 + F_1 - 1 = F_{i+1} + F_{i+2} - 1$.
Let $i = 0$.
Then $s_i = s_0 = 1 \geq 1 = 1 + 1 - 1 = F_1 + F_2 - 1 = F_{i+1} + F_{i+2} - 1$.
Thus the base cases hold true.

Inductive Step:
Let k be an integer such that $k \geq 0$.
Inductive Hypothesis:
Suppose that for all integers m such that $-1 \leq m \leq k$, $s_k \geq F_{k+1} + F_{k+2} - 1$.
Inductive Claim:
$s_{k+1} \geq F_{k+2} + F_{k+3} - 1$.

According to equation 3 of assignment 2, $s_{i+1} \geq min(s_{i-1} + s_i + 1, 2s_i + 1)$ for all integers $i \geq 0$. This gives us 2 cases:

Case 1 ($s_{k-1} + s_k + 1 < 2s_k + 1$):
Then $s_{k+1} \geq s_{k-1} + s_k + 1$. It then follows that...

$$\begin{aligned} s_{k+1} &\geq s_{k-1} + s_k + 1 \\ &\geq (F_k + F_{k+1} - 1) + (F_{k+1} + F_{k+2} - 1) + 1 \qquad \text{(By the I.H.)} \\ &= (F_{k+2} - 1) + (F_{k+3} - 1) + 1 \\ &= F_{k+2} + F_{k+3} - 1 \end{aligned}$$

Thus the inductive claim is true.

Case 2 ($s_{k-1} + s_k + 1 \geq 2s_k + 1$):
Then $s_{k+1} \geq 2s_k + 1$. It then follows that...

$$\begin{aligned} s_{k+1} &\geq 2s_k + 1 \\ &\geq 2 \cdot (F_{k+1} + F_{k+2} - 1) + 1 && \text{(By the I.H.)} \\ &= 2 \cdot F_{k+3} - 1 \\ &= F_{k+3} + F_{k+3} - 1 \\ &\geq F_{k+2} + F_{k+3} - 1 \end{aligned}$$

Thus the inductive claim is true.

Insertions

Question 2: Briefly explain why the only nodes in a tree whose heights or balance factors might have changed lie on the path from the a leaf up to the root of the tree.

By the definition, node height only depends on the greatest depth of a subtree with the supplied node as the root. The height of a node can only change if a descendant to said node is added or removed from the tree. Any other node that is not an ancestor of the newly inserted/deleted node therefore cannot have had their height altered. Also, since balance factor is defined by the difference in height of a node's children, then if the height of a node is unaltered, the balance factor of the node is also unaltered.

Question 3: Briefly explain why the balance factors of the nodes on the above path are all either 2, 1, 0, -1, or -2.

Assuming the tree was indeed an AVL tree before the insertion, then all nodes of the tree must have a balance factor within $\{-1, 0, 1\}$. By inserting a node, the balance factor of any preceeding node will be modified by a value of either 1 or -1, as only one node was inserted. Thus, the extreme values of the set of possible balance factors may increase or decrease by an absolute value of 1, meaning -1 may become -2 and 1 may become 2. The balance factors of -1, 0, and 1 are also still possible via different combonations of adding or subtracting 1 from any of the previous possible balance factor values.

Question 4: Explain why 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 have not been changed either.

The height of a node depends on the maximum height of its children. If neither of the node's children's heights have changed, then the node's own height cannot have changed. Since the height of v has not been changed, then the height of any ancestor of v cannot have changed either.

The balance factor of a node is defined as the height of the left child minus the height of the right child. If neither of the node's children's heights have changed, then the node's own balance factor cannot have changed. Since the height of v has not been changed, then the balance factor of any ancestor of v cannot have changed either.

Question 5: Comparing Figures 5 and 6, confirm that node α has height h and balance factor 0 after the described rotation.

After the rotation, node α has T_2 as its left child, and T_3 as its right child. No nodes within T_2 or T_3 have been changed, and thus they each maintain the height of $h-1$, as they did before the rotation. Because each child of α has the same height, the balance factor of α is 0. Also, the maximum height of either child of α is $h-1$, meaning that the height of α is indeed one more than the height of its deepest subtree; $(h-1)+1=h$.

Question 6: Confirm that the node β has height $h+1$ and balance factor 0 as well after the described rotation.

We know that subtree T_1 has a height of h before and after the rotation. We also know that α has a height of h. Thus, by similar logic as question 5, we know that β must have a height of one greater than h; $h+1$. Also, the difference in height of each child of β is equal to $h-h$, which results in a balance factor of 0.

Question 7: Explain why the binary search tree is once again an AVL tree after the described rotation.

The right rotation operation on α will preserve the binary tree property. We know that before the insertion, the tree was an AVL tree (because alpha is the deepest node to have a balance factor of 2 or -2). Therefore T_2 and T_3, which were part of the original AVL tree, are by definition also AVL trees themselves. Since nodes α and β now have a balance factor of 0, and no nodes above or below nodes α and β have been modified, the binary search tree is an AVL tree once again because all nodes in the tree have a balance factor within $\{-1, 0, 1\}$.

Question 8: Confirm that α and β each have a balance factor within $\{-1, 0, 1\}$ and height h after this adjustment.

After the rotations, T_1 and T_3 maintain height $h - 1$, and both T_{2a} and T_{2b} maintain height of either $h - 1$ or $h - 2$ (at least one of them must be $h - 1$ because γ had height h before the rotations). Because of this, both nodes α and β will have height h. Also, because there is a chance for either T_{2a} or T_{2b} to have height $h - 2$ (but not both), node β will have a balance factor of either 0 or 1, and node α will have a balance factor of either 0 or -1, both of which will be in $\{-1, 0, 1\}$. Only one of these nodes can have a balance factor that is not 0.

Question 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.

Since both children of γ, nodes α and β, have height h after the rotations, node γ must have height $h + 1$ after the rotations. Seeing as α and β also have the same height as each other, node γ has a balance factor of 0.

Rotations on a tree will preserve its binary tree property. Because, by definition, α is the lowest node on the path from the inserted leaf to the root that has a balance factor of either 2 or -2, we know that all other nodes initially had a balance factor within the range of $\{-1, 0, 1\}$, including nodes inside the subtrees T_1, T_2, and T_3. After the adjustment, the subtrees were unaltered and therefore are still AVL trees. Since nodes α and β now have a balance factor within $\{-1, 0, 1\}$, node γ a balance factor of 0, and no nodes

above or below nodes α, β, and γ have been modified, the binary search tree is an AVL tree once again.

Question 10: 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.

Both of these cases are mirrors of the previous two, and can be solved in a similar way.

Seeing as node α has a balance factor of -2, there must be 2 more descendants on the right of α than there are on the left. Because of this, much like the *left-left* and *left-right* cases, node α must have height of $h+2$ for some integer $h \geq 0$. Using this, we can state that the left child of α (T_1) has height $h-1$, and the right child of α has height $h+1$. We will define the right child of α to be β, and define the left child of α to be T_1. Because α is the *deepest* node with a balance factor of either -2 or 2, both children of node β must have had height $h-1$ before the insertion. If this was not true, then either β would have become the deepest node with a balance factor of either -2 or 2, or β would not have had height h before the insertion. Two cases follow:

Right-Right Case:

In this case, node β will have a balance factor of -1 after the insertion. Since β has a height of $h+1$, the left and right children of β (defined as T_2 and T_3) must have heights $h-1$ and h respectively.

This is solved with a left rotation at α. T_1 will remain the left child of α, with T_2 becoming the right child. Node α then becomes the left child of β, with T_3 remaining the right child.

Because both T_1 and T_2 had height $h-1$, node α now has height h and a balance factor of 0. And since both children of β, T_3 and α, have height h, node β now has height $h+1$ and a balance factor of 0.

Thus this problem node has been fixed.

Right-Left Case:

In this case, node β will have a balance factor of 1 after the insertion. Since β has a height of $h+1$, the left and right children of β (defined as γ and T_3) must have heights h and $h-1$ respectively. We will defined the left

child of γ as T_{2a}, and the right as T_{2b}. In accordance of the height h of γ, at least one of these children must have height $h-1$, while the other can have either $h-1$ or $h-2$.

This is solved with a right rotation at β, followed by a left rotation at α. T_1 will remain the left child of α, with T_{2a} becoming the right child. T_3 will remain the right child of β, with T_{2b} becoming the left child. Nodes α and β then become the left and right children of γ respectively.

Because node γ had height h before the rotations, at least of one T_{2a} and T_{2b} must have height $h-1$. Also, since α was defined as the *lowest* node with a balance factor outside of $\{-1, 0, 1\}$, the remaining child of γ must either have height $h-1$ as well, or height $h-2$ in order to keep γ's balance factor within the acceptable range. Using this, we can determine that node α will have height h and a balance factor of either 0 or 1, and node β will also have height h and a balance factor of either -1 or 0 after the rotations. Examining further, we can also determine that node γ will have height $h+1$ and a balance factor of 0 after the rotations.

Thus this problem node has been fixed.

Deletions

Question 11: Suppose that you perform a right rotation at α so that the subtree with root β after this adjustment is as shown in Figure 6. Briefly explain why α has height $h+1$ and balance factor 1 after this adjustment.

Node α will have children of height h and $h-1$ (from subtrees T_2 and T_3 respectively). Thus the height of α is the max of the children, plus 1; $(h)+1 = h+1$. Also, the balance factor of α will be $h-(h-1)$, which is 1.

Question 12: Briefly explain why β has height $h+2$ and balance factor -1 after this adjustment.

The children of node β have height h and $h+1$ (from subtree T_1 and node α respectively). Thus the height of β is the max of the children, plus 1; $(h+1)+1 = h+2$. Also, the balance factor of β will be $h-(h+1)$, which is -1.

Question 13: Briefly explain why the entire binary search tree is an AVL tree after this adjustment.

The right rotation at α will preserve the binary tree property of the tree. This is because α was the deepest node from the promoted node, and up to the root, that did not have an allowed balance factor. Therefore, β and subtrees T_1, T_2, and T_3 all have balance factors within $\{-1, 0, 1\}$ because they are below α. We have shown in questions 11 and 12 that both β and α have balance factors of -1 and 1 respectively after the rotation. Also, the subtrees T_1, T_2, and T_3 have not been altered through any more insertions/deletions. Therefore, because β, α, and the subtrees T_1, T_2, and T_3 all have balance factors within $\{-1, 0, 1\}$ and the tree is a binary search tree, this tree is indeed an AVL tree.

We also know that the height of α before was $h+2$, whilst the new height of β after the adjustment is also $h+2$. Being the top nodes and having the same height, we know that none of the balance factors above will be changed (as explained in the point about node v if its balance factor is unchanged, then anything above is also unchanged in balance factor). We know that before the deletion, the entire tree was an AVL tree. Therefore, we can conclude that the whole tree is an AVL tree again after the adjustment.

Question 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. Briefly describe another three cases corresponding to this, along with the adjustments that should be made for each.

Right-Left Case: Here, β (the right child of α) has a left child of height h and a right child of height $h-1$. This left child of β (i.e. γ) has at least one child with height $h-1$ and the other is either $h-1$ or $h-2$. An adjustment using a right rotation at β followed by a left rotation at α will restore the tree's AVL properties.
Right-Right Case: Here, β (again, the right child of α) has a right child of height h and a left child of height $h-1$. An adjustment using a left rotation at α will restore the tree's AVL properties.
Right-Equal Case: Here, β (like above) has two children with the same height, h. An adjustment using a left rotation at α will restore the tree's AVL properties.

Implementing an AVL Tree

Question 15: Briefly describe the structure of an algorithm for an insertion into an AVL tree.

```
void insert (Key k, Value v, Node x) throws ElementFoundException {

   if (k < x.key) {

      if (x has no left child){
         Insert a node with key k and value v as the left child of
            x.
      } else {
         insert(k, v, x.left);
      }

   } else if (k > x.key) {

      if (x has no right child){
         Insert a node with key k and value v as the right child of
            x.
      } else {
         insert(k, v, x.right);
      }

   } else { // k == x.key
      throw new ElementFoundException();
      // Or x.value = v; if representing a dictionary
   }

   if (a node y was inserted) {

      while (y != null) {
         int oldHeight = y.height;
         Update the height and balance factor of y.
         if (y.balanceFactor == 2 || y.balanceFactor == -2) {
            Balance the node.
            break;
         }
         if (oldHeight != 0 && oldHeight == y.height) {
            break;
         }
```

```
36          y = y.parent;
37       }
38
39    }
40
41 }
```

Question 16: Briefly describe the structure of an algorithm for a deletion from an AVL tree.

```
1  void delete (Node x) {
2
3     Node p = null;
4
5     if (x has no children) {
6
7        if (x == root) {
8           root = null;
9        } else {
10          p = x.parent;
11          if (x.key < x.parent.key) {
12             p.left = null;
13          } else {
14             p.right = null;
15          }
16          x.parent = null;
17       }
18
19    } else if (x has 1 child) {
20
21       Node c = child of x;
22       if (x == root) {
23          c.parent = null;
24          root = c;
25       } else {
26          p = x.parent;
27          c.parent = p;
28          Assign c as the appropriate child of its new parent.
29          x.parent = null;
30       }
31
```

```
32    } else { // x has 2 children
33
34       Node s = Successor of x;
35       x.key = s.key;
36       x.value = s.value;
37       delete(s);
38
39    }
40
41    while (p != null) {
42       Update the height and balance factor of p.
43       if (p.balanceFactor == 2 || p.balanceFactor == -2) {
44          Balance the node.
45       }
46       if (Balanced a *-Equal case) {
47          break;
48       }
49       p = p.parent;
50    }
51
52 }
```

Question 17: Provide a complete `AVLDictionary.java` class for assessment.

`rotateLeft`:

In the `rotateLeft` method for `AVLDictionary.java`, the only difference from the `rotateLeft` method from `BSTDictionary.java` is that after a node is rotated, the heights of nodes α and β are updated.

`rotateRight`:

In the `rotateRight` method for `AVLDictionary.java`, the only difference from the `rotatRight` method from `BSTDictionary.java` is that after a node is rotated, the heights of nodes α and β are updated.

`change`:

In the `change` method for `AVLDictionary.java`, the only difference from the `change` method from `BSTDictionary.java` is that if a node was inserted, the program loops up the tree from the inserted node to the root. In this loop, the program updates the height of each node, and checks the balance factor of the node as well. If a node has a balance factor of -2 or 2, it performs the appropriate rotations to restore the balance factor to a value within $\{-1, 0, 1\}$.

`deleteNode`:

In the `deleteNode` method for `AVLDictionary.java`, the only difference from the `deleteNode` method from `BSTDictionary.java` is that after a node is deleted, the program loops up the tree from the deleted node to the root. In this loop, the program updates the height of each node, and checks the balance factor of the node as well. If a node has a balance factor of -2 or 2, it performs the appropriate rotations to restore the balance factor to a value within $\{-1, 0, 1\}$.