Balanced Binary Search Trees

In the previous section we looked at building a binary search tree. As we learned, the performance of the binary search tree can degrade to $O(n)$ for operations like `get` and `put` when the tree becomes unbalanced. In this section we will look at a special kind of binary search tree that automatically makes sure that the tree remains balanced at all times. This tree is called an **AVL tree** and is named for its inventors: G.M. Adelson-Velskii and E.M. Landis.

An AVL tree implements the Map abstract data type just like a regular binary search tree, the only difference is in how the tree performs. To implement our AVL tree we need to keep track of a **balance factor** for each node in the tree. We do this by looking at the heights of the left and right subtrees for each node. More formally, we define the balance factor for a node as the difference between the height of the left subtree and the height of the right subtree.

$$balanceFactor = height(leftSubTree) - height(rightSubTree)$$

Using the definition for balance factor given above we say that a subtree is left-heavy if the balance factor is greater than zero. If the balance factor is less than zero then the subtree is right heavy. If the balance factor is zero then the tree is perfectly in balance. For purposes of implementing an AVL tree, and gaining the benefit of having a balanced tree we will define a tree to be in balance if the balance factor is -1, 0, or 1. Once the balance factor of a node in a tree is outside this range we will need to have a procedure to bring the tree back into balance. *Figure 1* shows an example of an unbalanced, right-heavy tree and the balance factors of each node.

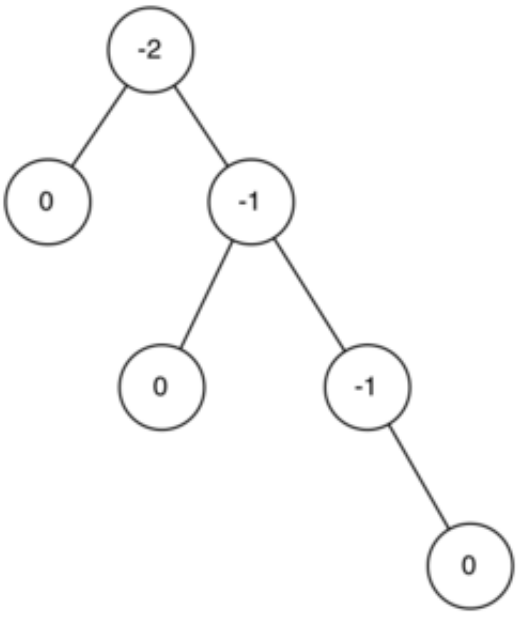

Figure 1: An Unbalanced Right-Heavy Tree with Balance Factors

AVL Tree Performance

Before we proceed any further lets look at the result of enforcing this new balance factor requirement. Our claim is that by ensuring that a tree always has a balance factor of -1, 0, or 1 we can get better Big-O performance of key operations. Let us start by thinking about how this balance condition changes the worst-case tree. There are two possibilities to consider, a left-heavy tree and a right heavy tree. If we consider trees of heights 0, 1, 2, and 3, *Figure 2* illustrates the most unbalanced left-heavy tree possible under the new rules.

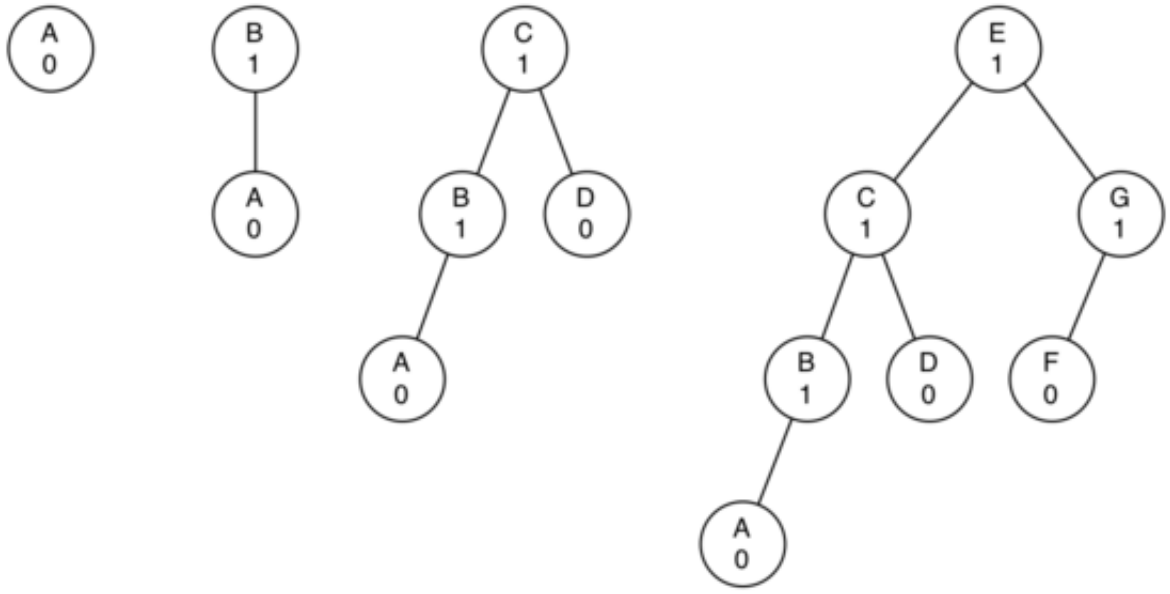

Figure 2: Worst-Case Left-Heavy AVL Trees

Looking at the total number of nodes in the tree we see that for a tree of height 0 there is 1 node, for a tree of height 1 there is $1 + 1 = 2$ nodes, for a tree of height 2 there are $1 + 1 + 2 = 4$ and for a tree of height 3 there are $1 + 2 + 4 = 7$. More generally the pattern we see for the number of nodes in a tree of height h (N_h) is:

$$N_h = 1 + N_{h-1} + N_{h-2}$$

This recurrence may look familiar to you because it is very similar to the Fibonacci sequence. We can use this fact to derive a formula for the height of an AVL tree given the number of nodes in the tree. Recall that for the Fibonacci sequence the i_{th} Fibonacci number is given by:

$$\begin{aligned} F_0 &= 0 \\ F_1 &= 1 \\ F_i &= F_{i-1} + F_{i-2} \text{ for all } i \geq 2 \end{aligned}$$

An important mathematical result is that as the numbers of the Fibonacci sequence get larger and larger the ratio of F_i/F_{i-1} becomes closer and closer to approximating the golden ratio Φ which is defined as $\Phi = \frac{1+\sqrt{5}}{2}$. You can consult a math text if you want to see a derivation of the previous equation. We will simply use this equation to approximate F_i as $F_i = \Phi^i/\sqrt{5}$. If we make use of this approximation we can rewrite the equation for N_h as:

$$N_h = F_{h+2} - 1, h \geq 1$$

By replacing the Fibonacci reference with its golden ratio approximation we get:

$$N_h = \frac{\Phi^{h+2}}{\sqrt{5}} - 1$$

If we rearrange the terms, and take the base 2 log of both sides and then solve for h we get the following derivation:

$$\log N_h + 1 = (H+2)\log\Phi - \frac{1}{2}\log 5$$
$$h = \frac{\log N_h + 1 - 2\log\Phi + \frac{1}{2}\log 5}{\log\Phi}$$
$$h = 1.44 \log N_h$$

This derivation shows us that at any time the height of our AVL tree is equal to a constant(1.44) times the log of the height of the tree. This is great news for searching our AVL tree because it limits the search to $O(\log N)$.

AVL Tree Implementation

Now that we have demonstrated that keeping an AVL tree in balance is going to be a big performance improvement, let us look at how we will augment the procedure to insert a new key into the tree. Since all new keys are inserted into the tree as leaf nodes and we know that the balance factor for a new leaf is zero, there are no new requirements for the node that was just inserted. But once the new leaf is added we must update the balance factor of its parent. How this new leaf affects the parent's balance factor depends on whether the leaf node is a left child or a right child. If the new node is a right child the balance factor of the parent will be reduced by one. If the new node is a left child then the balance factor of the parent will be increased by one. This relation can be applied recursively to the grandparent of the new node, and possibly to every ancestor all the way up to the root of the tree. Since this is a recursive procedure let us examine the two base cases for updating balance factors:

- The recursive call has reached the root of the tree.
- The balance factor of the parent has been adjusted to zero. You should convince yourself that once a subtree has a balance factor of zero, then the balance of its ancestor nodes does not change.

We will implement the AVL tree as a subclass of `BinarySearchTree`. To begin, we will override the `_put` method and write a new `updateBalance` helper method. These methods are shown in *Listing 1*. You will notice that the definition for `_put` is exactly the same as in simple binary search trees except for the additions of the calls to `updateBalance` on lines 7 and 13.

Listing 1

```
def _put(self,key,val,currentNode):
    if key < currentNode.key:
        if currentNode.hasLeftChild():
                self._put(key,val,currentNode.leftChild)
        else:
                currentNode.leftChild = TreeNode(key,val,parent=currentNode)
                self.updateBalance(currentNode.leftChild)
    else:
        if currentNode.hasRightChild():
                self._put(key,val,currentNode.rightChild)
        else:
                currentNode.rightChild = TreeNode(key,val,parent=currentNode)
                self.updateBalance(currentNode.rightChild)

def updateBalance(self,node):
    if node.balanceFactor > 1 or node.balanceFactor < -1:
        self.rebalance(node)
        return
    if node.parent != None:
        if node.isLeftChild():
                node.parent.balanceFactor += 1
        elif node.isRightChild():
                node.parent.balanceFactor -= 1

        if node.parent.balanceFactor != 0:
                self.updateBalance(node.parent)
```

The new `updateBalance` method is where most of the work is done. This implements the recursive procedure we just described. The `updateBalance` method first checks to see if the current node is out of balance enough to require rebalancing (line 16). If that is the case then the rebalancing is done and no further updating to parents is required. If the current node does not require rebalancing then the balance factor of the parent is adjusted. If the balance factor of the parent is non-zero then the algorithm continues to work its way up the tree toward the root by recursively calling `updateBalance` on the parent.

When a rebalancing of the tree is necessary, how do we do it? Efficient rebalancing is the key to making the AVL Tree work well without sacrificing performance. In order to bring an AVL Tree back into balance we will perform one or more **rotations** on the tree.

To understand what a rotation is let us look at a very simple example. Consider the tree in the left half of *Figure 3*. This tree is out of balance with a balance factor of -2. To bring this tree into balance we will use a left rotation around the subtree rooted at node A.

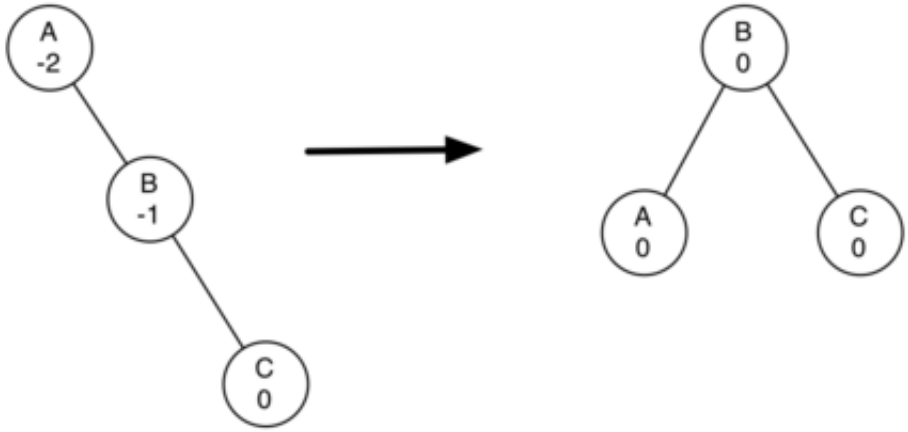

Figure 3: Transforming an Unbalanced Tree Using a Left Rotation

To perform a left rotation we essentially do the following:

- Promote the right child (B) to be the root of the subtree.
- Move the old root (A) to be the left child of the new root.
- If new root (B) already had a left child then make it the right child of the new left child (A). Note: Since the new root (B) was the right child of A the right child of A is guaranteed to be empty at this point. This allows us to add a new node as the right child without any further consideration.

While this procedure is fairly easy in concept, the details of the code are a bit tricky since we need to move things around in just the right order so that all properties of a Binary Search Tree are preserved. Furthermore we need to make sure to update all of the parent pointers appropriately.

Lets look at a slightly more complicated tree to illustrate the right rotation. The left side of *Figure 4* shows a tree that is left-heavy and with a balance factor of 2 at the root. To perform a right rotation we essentially do the following:

- Promote the left child (C) to be the root of the subtree.
- Move the old root (E) to be the right child of the new root.
- If the new root(C) already had a right child (D) then make it the left child of the new right child (E). Note: Since the new root (C) was the left child of E, the left child of E is guaranteed to be empty at this point. This allows us to add a new node as the left child without any further consideration.

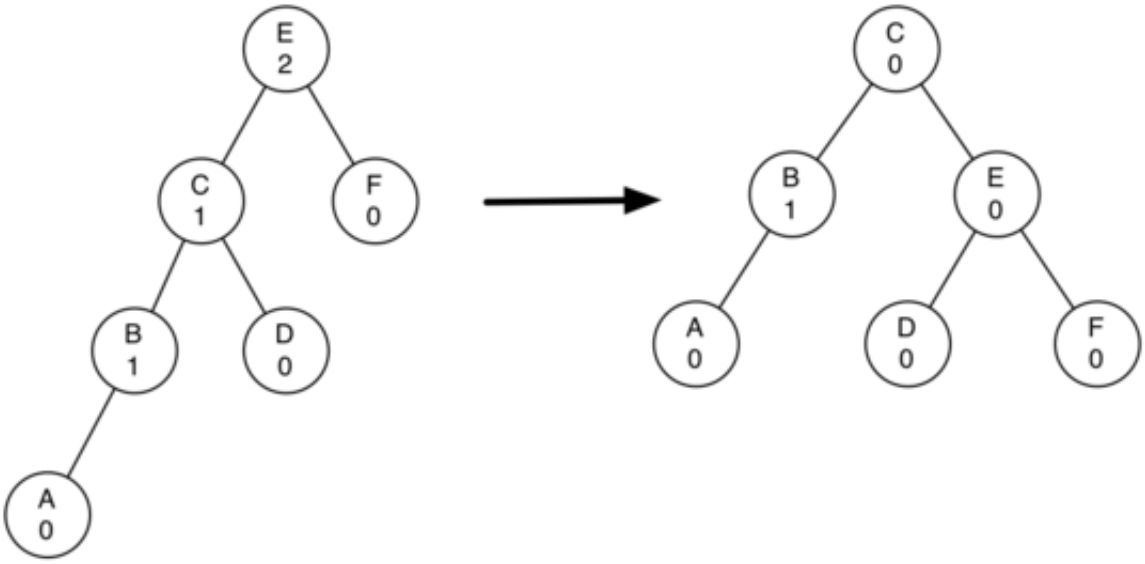

Figure 4: Transforming an Unbalanced Tree Using a Right Rotation

Now that you have seen the rotations and have the basic idea of how a rotation works let us look at the code. *Listing 2* shows the code for both the right and the left rotations. In line 2 we create a temporary variable to keep track of the new root of the subtree. As we said before the new root is the right child of the previous root. Now that a reference to the right child has been stored in this temporary variable we replace the right child of the old root with the left child of the new.

The next step is to adjust the parent pointers of the two nodes. If `newRoot` has a left child then the new parent of the left child becomes the old root. The parent of the new root is set to the parent of the old root. If the old root was the root of the entire tree then we must set the root of the tree to point to this new root. Otherwise, if the old root is a left child then we change the parent of the left child to point to the new root; otherwise we change the parent of the right child to point to the new root. (lines 10-13). Finally we set the parent of the old root to be the new root. This is a lot of complicated bookkeeping, so we encourage you to trace through this function while looking at *Figure 3*. The `rotateRight` method is symmetrical to `rotateLeft` so we will leave it to you to study the code for `rotateRight`.

Listing 2

```
def rotateLeft(self,rotRoot):
    newRoot = rotRoot.rightChild
    rotRoot.rightChild = newRoot.leftChild
    if newRoot.leftChild != None:
        newRoot.leftChild.parent = rotRoot
    newRoot.parent = rotRoot.parent
    if rotRoot.isRoot():
        self.root = newRoot
    else:
        if rotRoot.isLeftChild():
                rotRoot.parent.leftChild = newRoot
        else:
            rotRoot.parent.rightChild = newRoot
    newRoot.leftChild = rotRoot
    rotRoot.parent = newRoot
    rotRoot.balanceFactor = rotRoot.balanceFactor + 1 - min(newRoot.balanceFacto
r, 0)
    newRoot.balanceFactor = newRoot.balanceFactor + 1 + max(rotRoot.balanceFacto
r, 0)
```

Finally, lines 16-17 require some explanation. In these two lines we update the balance factors of the old and the new root. Since all the other moves are moving entire subtrees around the balance factors of all other nodes are unaffected by the rotation. But how can we update the balance factors without completely recalculating the heights of the new subtrees? The following derivation should convince you that these lines are correct.

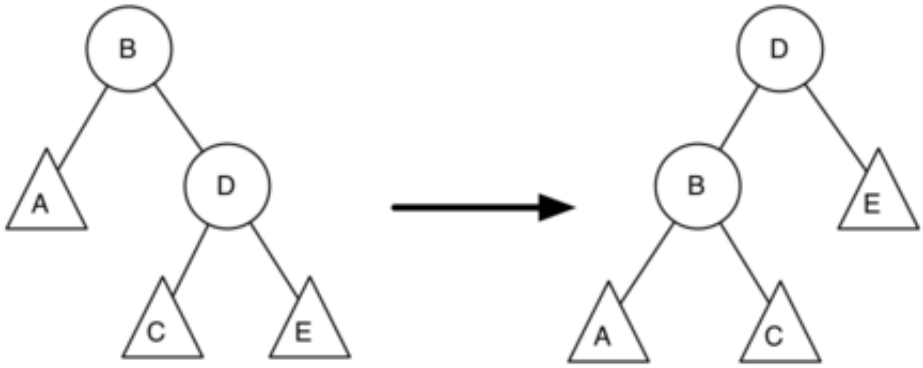

Figure 5: A Left Rotation

Figure 5 shows a left rotation. B and D are the pivotal nodes and A, C, E are their subtrees. Let h_x denote the height of a particular subtree rooted at node x. By definition we know the following:

$$newBal(B) = h_A - h_C$$
$$oldBal(B) = h_A - h_D$$

But we know that the old height of D can also be given by $1 + max(h_C, h_E)$, that is, the height of D is one more than the maximum height of its two children. Remember that h_c and h_E hav not changed. So, let us substitute that in to the second equation, which gives us

$$oldBal(B) = h_A - (1 + max(h_C, h_E))$$

and then subtract the two equations. The following steps do the subtraction and use some algebra to simplify the equation for $newBal(B)$.

$$\begin{aligned} newBal(B) - oldBal(B) &= h_A - h_C - (h_A - (1 + max(h_C, h_E))) \\ newBal(B) - oldBal(B) &= h_A - h_C - h_A + (1 + max(h_C, h_E)) \\ newBal(B) - oldBal(B) &= h_A - h_A + 1 + max(h_C, h_E) - h_C \\ newBal(B) - oldBal(B) &= 1 + max(h_C, h_E) - h_C \end{aligned}$$

Next we will move $oldBal(B)$ to the right hand side of the equation and make use of the fact that $max(a, b) - c = max(a - c, b - c)$.

$$newBal(B) = oldBal(B) + 1 + max(h_C - h_C, h_E - h_C)$$

But, $h_E - h_C$ is the same as $-oldBal(D)$. So we can use another identity that says $max(-a, -b) = -min(a, b)$. So we can finish our derivation of $newBal(B)$ with the following steps:

$$\begin{aligned} newBal(B) &= oldBal(B) + 1 + max(0, -oldBal(D)) \\ newBal(B) &= oldBal(B) + 1 - min(0, oldBal(D)) \end{aligned}$$

Now we have all of the parts in terms that we readily know. If we remember that B is `rotRoot` and D is `newRoot` then we can see this corresponds exactly to the statement on line 16, or:

```
rotRoot.balanceFactor = rotRoot.balanceFactor + 1 - min(0,newRoot.balanceFactor)
```

A similar derivation gives us the equation for the updated node D, as well as the balance factors after a right rotation. We leave these as exercises for you.

Now you might think that we are done. We know how to do our left and right rotations, and we know when we should do a left or right rotation, but take a look at *Figure 6*. Since node A has a balance factor of -2 we should do a left rotation. But, what happens when we do the left rotation around A?

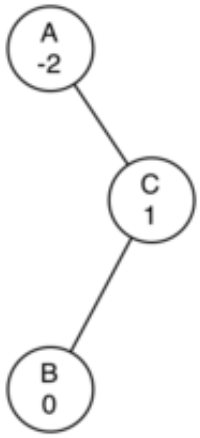

Figure 6: An Unbalanced Tree that is More Difficult to Balance

Figure 7 shows us that after the left rotation we are now out of balance the other way. If we do a right rotation to correct the situation we are right back where we started.

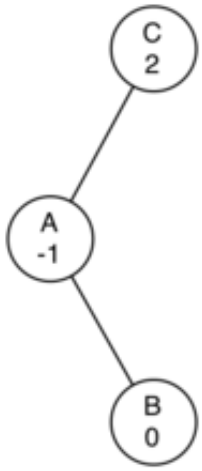

Figure 7: After a Left Rotation the Tree is Out of Balance in the Other Direction

To correct this problem we must use the following set of rules:

- If a subtree needs a left rotation to bring it into balance, first check the balance factor of the right child. If the right child is left heavy then do a right rotation on right child, followed by the original left rotation.
- If a subtree needs a right rotation to bring it into balance, first check the balance factor of the left child. If the left child is right heavy then do a left rotation on the left child, followed by the original right rotation.

Figure 8 shows how these rules solve the dilemma we encountered in *Figure 6* and *Figure 7*. Starting with a right rotation around node C puts the tree in a position where the left rotation around A brings the entire subtree back into balance.

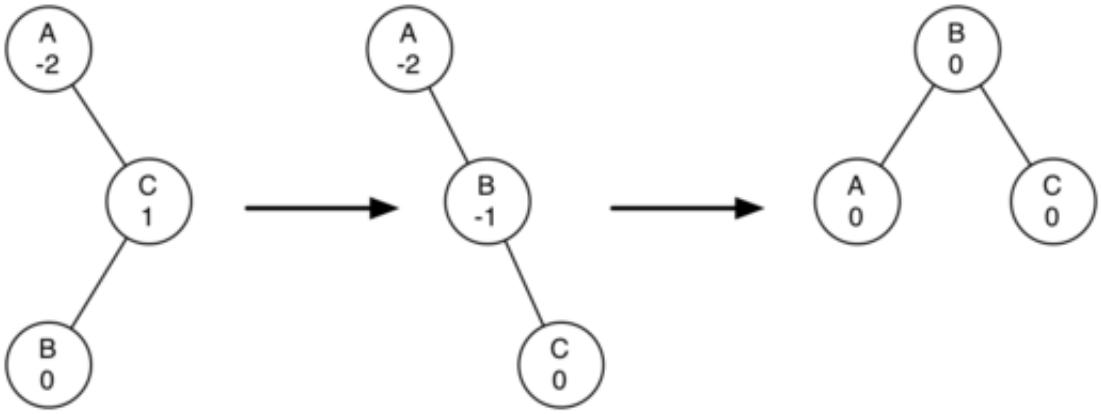

Figure 8: A Right Rotation Followed by a Left Rotation

The code that implements these rules can be found in our `rebalance` method, which is shown in *Listing 3*. Rule number 1 from above is implemented by the `if` statement starting on line 2. Rule number 2 is implemented by the `elif` statement starting on line 8.

Listing 3

```
def rebalance(self,node):
  if node.balanceFactor < 0:
        if node.rightChild.balanceFactor > 0:
           self.rotateRight(node.rightChild)
           self.rotateLeft(node)
        else:
           self.rotateLeft(node)
  elif node.balanceFactor > 0:
        if node.leftChild.balanceFactor < 0:
           self.rotateLeft(node.leftChild)
           self.rotateRight(node)
        else:
           self.rotateRight(node)
```

The *discussion questions* (treeexercises.html#tree-discuss) provide you the opportunity to rebalance a tree that requires a left rotation followed by a right. In addition the discussion questions provide you with the opportunity to rebalance some trees that are a little more complex than the tree in *Figure 8*.

By keeping the tree in balance at all times, we can ensure that the `get` method will run in order $O(log_2(n))$ time. But the question is at what cost to our `put` method? Let us break this down into the operations performed by `put`. Since a new node is inserted as a leaf, updating the balance factors of all the parents will require a maximum of $log_2(n)$ operations, one for each level of the tree. If a subtree is found to be out of balance a maximum of two rotations are required to bring the tree back into balance. But, each of the rotations works in $O(1)$ time, so even our `put` operation remains $O(log_2(n))$.

At this point we have implemented a functional AVL-Tree, unless you need the ability to delete a node. We leave the deletion of the node and subsequent updating and rebalancing as an exercise for you.

Summary of Map ADT Implementations

Over the past two chapters we have looked at several data structures that can be used to implement the map abstract data type. A binary Search on a list, a hash table, a binary search tree, and a balanced binary search tree. To conclude this section, let's summarize the performance of each data structure for the key operations defined by the map ADT (see *Table 1*).

Table 1: Comparing the Performance of Different Map Implementations

operation	Sorted List	Hash Table	Binary Search Tree	AVL Tree
put	$O(n)$	$O(1)$	$O(n)$	$O(\log_2 n)$
get	$O(\log_2 n)$	$O(1)$	$O(n)$	$O(\log_2 n)$
in	$O(\log_2 n)$	$O(1)$	$O(n)$	$O(\log_2 n)$
del	$O(n))$	$O(1)$	$O(n)$	$O(\log_2 n)$