Balanced Binary Search Trees 10

In Chap. 6 binary search trees were defined along with a recursive insert algorithm. The discussion of binary search trees pointed out they have problems in some cases. Binary search trees can become unbalanced, actually quite often. When a tree is unbalanced the complexity of insert, delete, and lookup operations can get as bad as $\Theta(n)$. This problem with unbalanced binary search trees was the motivation for the development of height-balanced AVL trees by G.M. Adelson-Velskii and E.M. Landis, two Soviet computer scientists, in 1962. AVL trees were named for these two inventors. Their paper on AVL trees [1] described the first algorithm for maintaining balanced binary search trees.

Balanced binary search trees provide $\Theta(log\ n)$ insert, delete, and lookup operations. In addition, a balanced binary search tree maintains its items in sorted order. An infix traversal of a binary search tree will yield its items in ascending order and this traversal can be accomplished in $\Theta(n)$ time assuming the tree is already built.

The HashSet and HashMap classes provide very efficient insert, delete, and lookup operations as well, more efficient than the corresponding binary search tree operations. Heaps also provide $\Theta(log\ n)$ insert and delete operations. But neither hash tables nor heaps maintain their elements as an ordered sequence. If you want to perform many insert and delete operations and need to iterate over a sequence in ascending or descending order, perhaps many times, then a balanced binary search tree data structure may be more appropriate.

10.1 Chapter Goals

This chapter describes why binary search trees can become unbalanced. Then it goes on to describe several implementations of two types of height-balanced trees, AVL trees and splay trees. By the end of this chapter you should be able to implement your own AVL or splay tree datatype, with either iteratively or recursively implemented operations.

K.D. Lee and S. Hubbard, *Data Structures and Algorithms with Python*,
Undergraduate Topics in Computer Science, DOI 10.1007/978-3-319-13072-9_10

10.2 Binary Search Trees

A binary search tree comes in handy when a large number of insert, delete, and lookup operations are required by an application while at times it is necessary to traverse the items in ascending or descending order. Consider a website like Wikipedia that provides access to a large set of online materials. Imagine the designers of the website want to keep a log of all the users that have accessed the website within the last hour. The website might operate as follows.

- Each visitor accesses the website with a unique cookie.
- When a visitor accesses the site their cookie along with a date and time is recorded in a log on the site's server.
- If they have accessed the site within the last two hours their cookie and access time may already be recorded. In that case, their last access date and time is updated.
- Every hour a snapshot is generated as to who is currently accessing the site.
- The snapshot is to be generated in ascending order of the unique cookie numbers.
- After a patron has been inactive for at least an hour, according to the snapshot, their information is deleted from the record of website activity log.

Since the site is quite large with thousands, if not tens of thousands or more, people accessing it every hour, the data structure to hold this information must be fast. It must be fast to insert, lookup, and delete entries. It must also be quick to take snapshot since the website will hold up all requests while the snapshot is taken.

If the number of users that come and go during an hour on a site like Wikipedia is typically higher than the number that stay around for long periods of time, if may be most efficient to rebuild the tree from the activity log rather than delete each entry after it has been inactive for at least an hour. This would be true if the number of people still active on the site is much smaller than the number of inactive entries in the snapshot of the log. In this case, rebuilding the log after deleting inactive patrons must be fast as well.

A binary search tree is a logical choice for the organization of this log if we could guarantee $\Theta(log\, n)$ lookup, insert, and delete along with $\Theta(n)$ time to take a snapshot. However, a binary search tree has one big problem. Recall that as the snapshot is taken the log may be rebuilt with only the recently active users and furthermore the cookies will be accessed in ascending order while rebuilding the log.

Consider the insert operation on binary search trees shown in Sect. 10.2.1. When the binary search tree is rebuilt the items to insert into the new tree will be added in ascending order. The result is an unbalanced tree.

10.2.1 Binary Search Tree Insert

```
def __insert(root,val):
    if root == None:
        return BinarySearchTree.__Node(val)
```

```
4       if val < root.getVal():
5           root.setLeft(BinarySearchTree.__insert(root.getLeft(),val))
6       else:
7           root.setRight(BinarySearchTree.__insert(root.getRight(),val))
8       return root
```

If items are inserted into a binary search tree in ascending order the effect is that execution always progresses from line 2 to 4, 6, 7 and 8. The result on line 7 puts the new value in the right most location of the binary search tree, since it is the largest value inserted so far. The resulting tree is a stick extending down and to the right. Without any balance to the tree, inserting the next bigger value will result in traversing each and every value that has already been inserted to find the location of the new value. This means that the first value takes zero comparisons to insert, while the second requires one comparison to find its final location, the third value requires two comparisons, and so on. The total number of comparisons to build the tree is $\Theta(n^2)$ as proved in Chap. 2. This complexity will be much too slow for any site getting a reasonable amount of activity in an hour. In addition, when the height of the binary search tree is n, where n is the number of values in the tree, the look up, insert, and delete times are $\Theta(n)$ for both the worst and average cases. When the tree is a stick or even close to being a stick the efficiency characteristics of a binary search tree are no better than that of a linked list.

10.3 AVL Trees

A binary search tree that stays balanced would provide everything that is required by the website log described in the last section. AVL trees are binary search trees with additional information to maintain their balance. The height of an AVL tree is guaranteed to be $\Theta(log\ n)$ thus guaranteeing that lookup, insert, and delete operations will all complete in $\Theta(log\ n)$ time. With these guarantees, an AVL tree can be built in $\Theta(n\ log\ n)$ time from a sequence of n items. Moreover, AVL trees, like binary search trees, can be traversed using an inorder traversal, yielding their items in ascending order in $\Theta(n)$ time.

10.3.1 Definitions

To understand how AVL trees work, a few definitions are in order.

Height(*Tree*): The height of a tree is one plus the maximum height of its subtrees. The height of a leaf node is one.
Balance(*Tree*): The balance of a node in a binary tree is height(right subtree)—height(left subtree).
AVL Tree: An AVL tree is a binary tree in which the balance of every node in the tree is -1, 0 or 1.

10.3.2 Implementation Alternatives

Looking back at Chap. 6 and the implementation of binary search trees, inserting a value into a tree can be written recursively. Inserting into an AVL tree can also be implemented recursively. It is also possible to implement inserting a value into an AVL tree iteratively, using a loop and a stack. This chapter explores both alternatives.

Additionally, the balance of an AVL tree can be maintained using either the height of each node in the tree or the balance of each node in the tree. Implementations of AVL tree nodes store either their balance or their height. As values are inserted into the tree, the balance or height values of affected nodes are updated to reflect the addition of the new item in the tree.

10.3.3 AVLNode with Stored Balance

```python
class AVLTree:
    class AVLNode:
        def __init__(self,item,balance=0,left=None,right=None):
            self.item = item
            self.left = left
            self.right = right
            self.balance = balance

        def __repr__(self):
            return "AVLTree.AVLNode("+repr(self.item)+",balance="+
                   repr(self.balance)+",left="+repr(self.left)+
                   ",right="+repr(self.right)+")"
```

Whether implementing insert recursively or iteratively, the *Node* class of Chap. 6 must be extended slightly to accommodate either the balance or the height of the node. Consider the code fragment in Sect. 10.3.3. The first implementation of AVLTree that we'll explore is a balance storing iterative version of the algorithm. Notice that the AVLNode implementation is buried inside the AVLTree class to *hide* it from users of the AVLTree class. While Python does not actually prevent access to the AVLNode class from outside the AVLTree class, by convention users of the AVLTree data structure should know to leave the internals of the tree alone. AVL trees are created by users of this data structure, but not AVL nodes. The creation of nodes is handled by the ALVTree class.

The AVLNode constructor has default values for balance, left, and right which makes it easy to construct AVLTrees when debugging code. The *__repr__* function prints the AVLNode in a form that can be used to construct such a node. Calling *print*(*repr*(*node*)) will print a node so it can be provided to Python to construct a sample tree. The *repr*(*self.left*) and *repr*(*self.right*) are recursive calls to the *__repr__* function, so the entire tree is printed rooted at *self*. From Chap. 6 the same *__iter__* function will work to traverse an AVLTree. The iterator function will yield all the values of the tree in ascending order.

Examples in this chapter will refer to balance of nodes in an AVL Tree. It turns out that storing the balance of a node is sufficient to correctly implement height balanced

AVL Trees, but perhaps a bit more difficult to maintain than maintaining the height of each node in the tree. Later in the chapter modifications to these algorithms are discussed that maintain the height of each node. Whether storing height or balance in AVL Trees, the complexity of the tree operations is not affected.

10.3.4 AVL Tree Iterative Insert

As described in the last section, there are two variants to the insert algorithm for height balanced AVL trees. Insert can be performed iteratively or recursively. The balance can also be stored explicitly or it can be computed from the height of each subtree. This section describes how to maintain the balance explicitly without maintaining the height of each node.

Iteratively inserting a new value in a height balanced AVL tree requires keeping track of the path to the newly inserted value. To maintain that path, a stack is used. We'll call this stack the *path stack* in the algorithm. To insert a new node, we follow the unique search path from the root to the new node's location, pushing each node on the path stack as we proceed, just as if we were adding it to a binary search tree.

As we proceed along the path to the new node's destination, we push all the nodes we encounter onto the *path stack*. We insert the new item where it should be according to the binary search tree property. Then, the algorithm proceeds popping values from the path stack and adjusting their balances until a node is found that has a balance not equal to zero before being adjusted. This node, which is the closest ancestor with non-zero balance, is called the *pivot*. Based on the pivot and the location of the new value there are three mutually exclusive cases to consider which are described below. After making the adjustments in case 3 below there may be a new root node for the subtree rooted at the pivot. If this is the case, the parent of the pivot is the next node on the path stack and can be linked to the new subtree. If the path stack is empty after popping the pivot, then the root of the tree was the pivot. In this case, the root node of the AVL tree can be made to point to the new root node in the tree. As mentioned above, one of three cases will arise when inserting a new value into the tree.

Case 1: No Pivot There is no pivot node. In other words the balance of each node along the path was 0. In this case just adjust the balance of each node on the search path based on the relative value of the new key with respect to the key of each node. You can use the *path stack* to examine the path to the new node.

This case is depicted in Fig. 10.1 where 39 is to be added to the AVL tree. In each node the value is on the left and the balance is given on the right. Each of the nodes containing 10, 18, and 40 are pushed onto the *path stack*. The balance of the new node containing 39 is set to 0. The new balance of the node containing 40 is -1. The node containing 18 has a new balance of 1. The balance of the root node after the insert is 1 because 39 is inserted to the right of it and therefore its balance increases by one. The new value is inserted to the left of the node containing 40, so its balance decreases by one. Figure 10.2 depicts the tree after inserting the new value.

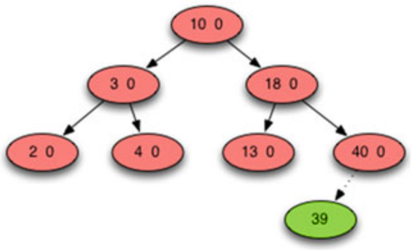

Fig. 10.1 AVL Tree Case 1—No Pivot Node

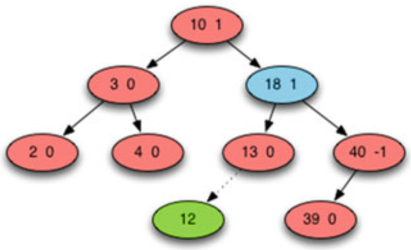

Fig. 10.2 AVL Tree Case 2—No Rotate

Case 2: Adjust Balances The pivot node exists. Further, the subtree of the pivot node in which the new node was added has the smaller height. In this case, just change the balance of the nodes along the search path from the new node up to the pivot node. The balances of the nodes above the pivot node are unaffected. This is true because the height of the subtree rooted at the pivot node is not changed by the insertion of the new node.

Figure 10.2 depicts this case. The item with *key* 12 is about to be added to the AVL tree. The node containing the 18 is the pivot node. Since the value to be inserted is less than 18 and the balance of the node containing 18 is 1, the new node could possibly help to better balance the tree. The AVL tree remains an AVL tree. The balance of nodes up to the pivot must be adjusted. Balances above the pivot need not be adjusted because they are unaffected. Figure 10.3 depicts what the tree looks like after inserting 12 into the tree.

Case 3: The pivot node exists. This time, however, the new node is added to the subtree of the pivot of larger height (the subtree in the direction of the imbalance). This will cause the pivot node to have a balance of −2 or 2 after inserting the new node, so the tree will no longer be an AVL tree. There are two subcases here, requiring either a *single rotation* or a *double rotation* to restore the tree to AVL status. Call the child of the pivot node in the direction of the imbalance the *bad child*.

Subcase A: Single Rotation This subcase occurs when the new node is added to the subtree of the bad child which is also in the direction of the imbalance. The *solution*

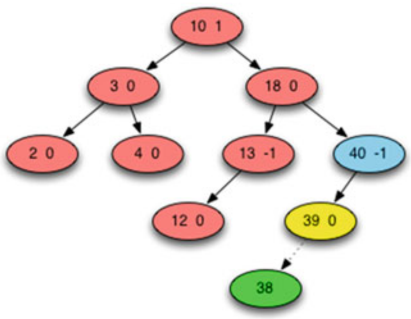

Fig. 10.3 AVL Tree Case 3A—Single Rotation

is a rotation at the pivot node in the opposite direction of the imbalance. After the rotation the tree is still a binary search tree. In addition, the subtree rooted at the pivot will be balanced once again, decreasing its overall height by one.

Figure 10.3 illustrates this subcase. The value 38 is to be inserted into the tree to the left of the node containing 39. However, doing so would result in the balance of the node containing 40 to decrease to −2, which is the closest ancestor with improper balance and the pivot node. The yellow node is the *bad child*. In addition, the 38 is being inserted in the same direction as the imbalance. The imbalance is on the left and new new value is being inserted on the left. The solution is to rotate the subtree rooted at 40 to the right, resulting in the tree pictured in Fig. 10.4.

Subcase B: Double Rotation This subcase occurs when the new node is added to the subtree of the bad child which is in the opposite direction of the imbalance. For this subcase, call the child node of the bad child which lies on the search path the *bad grandchild*. In some cases, there may not be a bad grandchild. In Fig. 10.4 the bad grandchild is the purple node. The solution is as follows:

1. Perform a single rotation at the bad child in the direction of the imbalance.
2. Perform a single rotation at the pivot away from the imbalance.

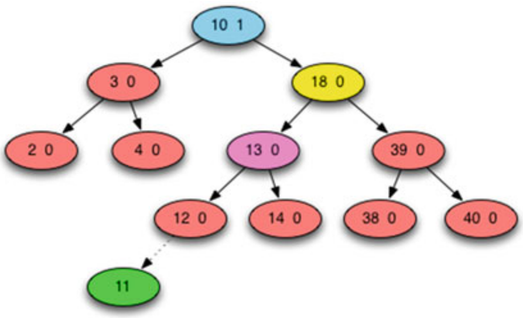

Fig. 10.4 AVL Tree Case 3B—Double Rotation

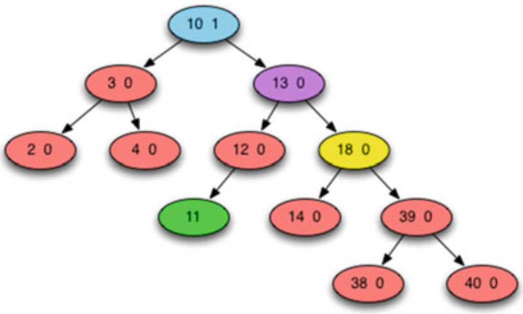

Fig. 10.5 AVL Tree Case 3B Step 1 Rotate Toward

Again, the tree is still a binary search tree and the height of the subtree in the position of the original pivot node is not changed by the double rotation. Figure 10.4 illustrates this situation. The pivot in this case is the root of the tree. The node containing 18 is the bad child. The bad grandchild is the node containing 13 (Fig. 10.5).

The imbalance in the tree is to the right of the pivot. Yet the 11 is being inserted to the left of the bad child. The first step is a rotation to the right at the bad child. This *brings the* 11 *up*, somewhat helping to balance the right side of the tree. The second step, depicted in Fig. 10.6 rotates to the left at the pivot bringing the whole tree into balance again.

The trickiest part of this algorithm is updating the balances correctly. First, the pivot, bad child, and bad grandchild contain the balances that may change. If there is no bad grandchild then the pivot's and bad child's balances will be zero. If there is a bad grandchild, as is the case here, then there is a little more work to determining the balances of the pivot and the bad child. When the bad grandchild exists, its balance is 0 after the double rotation. The balances of the bad child and pivot depend on the direction of the rotation and the value of the new item and the bad grandchild's item. This can be analyzed on a case by case basis to determine the balances of both the pivot and bad grandchild in these cases. In the next section we examine how the balances are calculated.

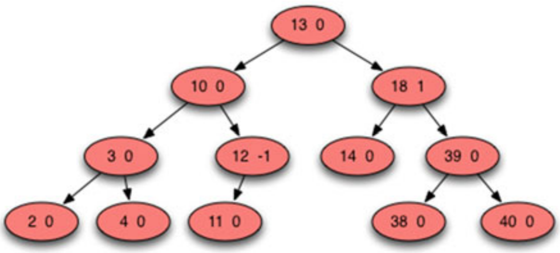

Fig. 10.6 AVL Tree Case 3B Step 2 Rotate Away

10.3.5 Rotations

Both cases 1 and 2 are trivial to implement as they simply adjust balances. Case 3 is by far the hardest of the cases to implement. Rotating a subtree is the operation that keeps the tree balanced as new nodes are inserted into it. For case 3 A the tree is in a state where a new node is going to be added to the tree causing an imbalance that must be dealt with. There are two possibilities. Figure 10.7 depicts the first of these possible situations. The new node may be inserted to the left of the bad child, A, when the subtree anchored at the pivot node is already weighted to the left. The pivot node, B, is the nearest ancestor with a non-zero balance. For node B to have balance -1 before inserting the new node its right subtree must have height h while its left subtree has height $h + 1$. Adding the new node into the subtree of the bad child would result in the pivot having balance -2 which is not allowed. The right rotation resolves the problem and maintains the binary search tree property. The subtree *T2* moves in the rotation but before the rotation all values in *T2* must have been less then *B* and greater than *A*. After the rotation this would also be true which means it remains a binary search tree.

Inserting a value to the right of the bad child when the imbalance is to the right results in an analogous situation requiring a left rotation. Notice that in either rotation the balance of nodes *A* and *B* are zero. This only applies to case 3 A and does not hold in the case of a double rotation (Fig. 10.8).

Again, the balance of both nodes, the pivot and the bad child, become zero after the rotation in either direction. Case 3 A is not possible under any other circumstances.

For case 3B we must deal not only with a pivot and bad child, but also a bad grandchild. As described in the previous section, this case occurs when inserting a

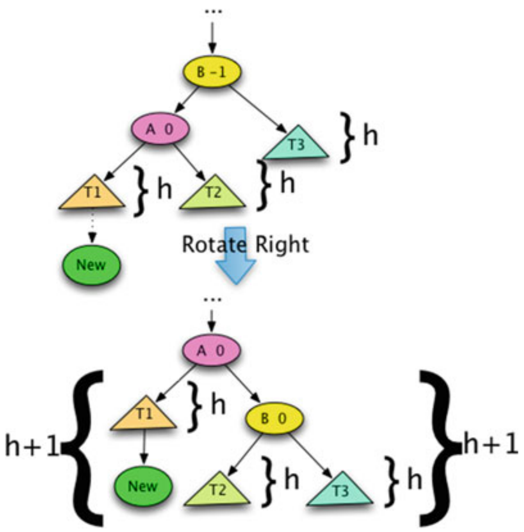

Fig. 10.7 AVL Tree Case 3A Right Rotation

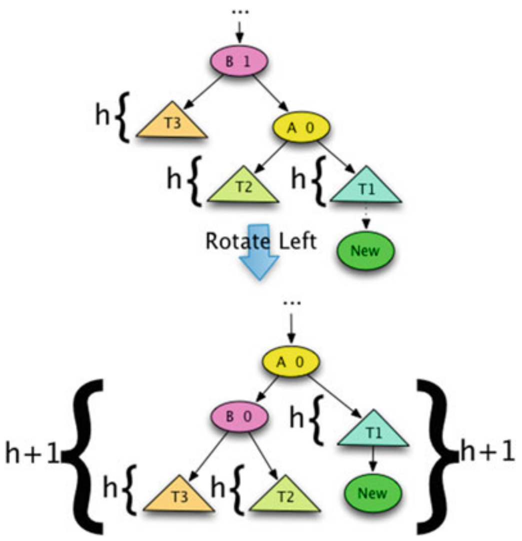

Fig. 10.8 AVL Tree Case 3A Left Rotation

new value under a bad child in the opposite direction of the imbalance. For instance, the subtree in Fig. 10.9 is weighted to the left and the new node is inserted to the right of the bad child. An analogous situation occurs when the subtree is weighted to the right and the new node is inserted into the left subtree of the bad child. When either situation occurs a double rotation is needed to bring it back into balance.

Figure 10.9 show that there are two possible subcases. There are actually three possible subcases. It is possible there is no bad grandchild. In that case, the newly inserted node will end up in the location that would have been occupied by the bad grandchild. Otherwise the new node might be inserted to the left or right of the bad grandchild, which is node C in Fig. 10.9. Either way, the first step in Fig. 10.9 is to rotate left at the bad child, node A. Then a right rotation at the pivot, node B, completes the rebalancing of the tree.

Again, the trickiest part of this implementation is the calculation of the balance of each node. The bad grandchild and new pivot node, node C in Fig. 10.9, always has a balance of 0. If there is no bad grandchild, then the new pivot node is the newly inserted value. If there was a bad grandchild, and if the new item was less than the bad grandchild's item, the balance of the bad child is 0 and the balance of the old pivot is 1. If the new item was inserted to the right of the bad grandchild then the balance of the bad child is −1 and the balance of the old pivot is 0. All other balances remain the same including balances above the pivot because the overall height of the tree before inserting the new value and after inserting the new value has not changed.

Again, an analogous situation occurs in the mirror image of Fig. 10.9. When a new value is inserted into a left subtree of a bad child which is in the right subtree of the pivot and which is already weighted more heavily to the right, then a double rotation is also required, rotating first right at the bad child and then left at the pivot.

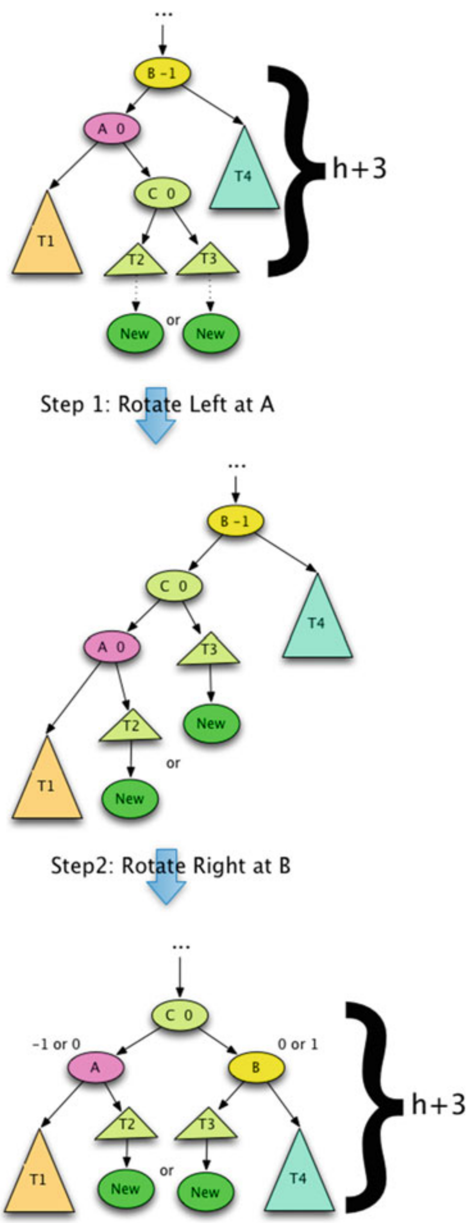

Fig. 10.9 AVL Tree Case 3B Steps 1 and 2

10.3.6 AVL Tree Recursive Insert

When implementing a recursive function it is much easier to write as a stand-alone function as opposed to a method of a class. This is because a stand-alone method may be called on nothing (i.e. *None* in the case of Python) while a method must always have a non-null *self* reference. Writing recursive functions as methods leads to special cases for *self*. For instance, the *insert* method, if written recursively, is easier to implement if it calls *__insert* as its recursive function. The *__insert* function of Sect. 10.2.1 won't suffice for height balanced AVL trees. The insert algorithm must take into account the current balance of the tree and operate to maintain the balance as we discussed in the three cases presented in the previous section.

10.3.7 The Recursive Insert AVL Tree Class Declaration

```python
class AVLTree:
    class AVLNode:
        def __init__(self,item,balance=0,left=None,right=None):
            self.item = item
            self.left = left
            self.right = right
            self.balance = balance

        # Other methods to be written here like __iter__ and
        # __repr__. See Chap. 6

    def __init__(self,root=None):
        self.root = root

    def insert(self, item):

        def __insert(root,item):
            ... # Code to be written here

            return root

        self.pivotFound = False
        self.root = __insert(self.root,item)

    def __repr__(self):
        return "AVLTree(" + repr(self.root) + ")"

    def __iter__(self):
        return iter(self.root)
```

The shell of the recursive implementation is given in Sect. 10.3.7. The algorithm proceeds much like a combination of the three cases presented above along with the implementation of insert presented in Sect. 10.2.1. There is no *path stack* in the recursive implementation. Instead, the run-time stack serves that purpose. Between lines 5 and 6 or lines 7 and 8 of Sect. 10.2.1 there is an opportunity to rebalance the tree as the code returns and works its way back up from the recursive calls. As each call returns, the balances of each node can be adjusted accordingly. Adjusting balances before returning implements cases one and two as described earlier in the chapter. Case three is detected when a balance of −2 or 2 results from rebalancing. In that case the pivot is found and rebalancing according to case 3 can occur.

Should a pivot be found, no balancing need occur above the pivot. This is the use of the *self.pivotFound* variable initialized on line 22 of the code in Sect. 10.3.7. This flag can be set to *True* to avoid any balancing above the pivot node, should it be found. Balances are adjusted just as described in the case by case analysis earlier in the chapter. In the worst case the balances of the pivot and bad child will need to be adjusted.

Implementing both the iterative and the recursive versions of insert into AVL trees helps illustrate the special cases that must be handled in the iterative version, while the recursive version will not need special cases. The recursive version does not need special case handling because of the way the *__insert* works. The function always is given the root node of a tree in which to insert the new item and returns the root node of the tree after inserting that item. Since it works in such a regular way, special case handling is not necessary.

10.3.8 Maintaining Balance Versus Height

The two implementations presented in this chapter, the recursive and iterative insert algorithms for AVL trees, maintained the balance of each node. As an alternative, the height of each node could be maintained. In this case, the height of a leaf node is 1. The height of any other node is 1 plus the maximum height of its two subtrees. The height of an empty tree or *None* is 0.

10.3.9 AVLNode with Stored Height

```python
class AVLNode:
    def __init__(self,item,height=1,left=None,right=None):
        self.item = item
        self.left = left
        self.right = right
        self.height = height

    def balance(self):
        return AVLTree.height(self.right) - AVLTree.height(self.left)
```

If the height of nodes is maintained instead of balances, all heights on the path to the new item's inserted location must be adjusted on the way back up the tree. Unlike balances, it is not possible to stop adjusting heights at the pivot node. After rotation the height of the pivot and bad child must also be recomputed as the rotation may change their height. Since heights are computed bottom-up, all heights on the path, including the heights of the pivot and bad child should be recomputed in a bottom-up fashion. The code in Sect. 10.3.9 provides a partial declaration of an AVLNode storing the height of the tree tree rooted at the node. In this implementation the balance of any node can be computed from the heights of the two subtrees.

10.3.10 Deleting an Item from an AVL Tree

Deleting a value from an AVL tree can be accomplished in the same way as described in programming problem 2 from Chap. 6. However, it is necessary to adjust balances on the way back from deleting the final leaf node. This can be done either by maintaining a *path stack* if delete is implemented iteratively or by adjusting balances or heights while returning from the recursive calls in a recursive implementation of delete.

In either case, when adjusted balance of a node on the path reaches 2, a left rotation is required to rebalance the tree. If the adjusted balance of a node on the path results in -2, then a right rotation is required. These rotations may cascade back up the path to the root of the tree.

10.4 Splay Trees

AVL trees are always balanced since the balance of each node is computed and maintained to be either -1, 1 or 0. Because they are balanced they guarantee $\Theta(log\ n)$ lookup, insert, and delete time. An AVL tree is a binary search tree so it also maintains its items in sorted order allowing iteration from the smallest to largest item in $\Theta(n)$ time. While there doesn't seem to be many downsides to this data structure there is a possible improvement in the form of splay trees.

One of the criticisms of AVL trees is that each node must maintain its balance. The extra work and extra space that are required for this balance maintenance might be unnecessary. What if a binary search tree could maintain its balance *good enough* without storing the balance in each node. Storing the balance of each node or the height of each node increases the size of the data in memory. This was a bigger concern when memory sizes were smaller. But, maintaining the extra information takes extra time as well. What if we could not only reduce the overall data size but eliminate some of the work in maintaining the balance of a binary search tree.

The improvement to AVL trees incorporates the concept of *spatial locality*. This idea reflects the nature of interaction with large data sets. Access to a large data set is often localized, meaning that the same piece or several pieces of data might be accessed several times over a short period of time and then may not be accessed for some time while some other relatively small subset of the data is accessed by either inserting new values or looking up old values. *Spatial Locality* means that a relatively small subset of data is accessed over a short period of time.

In terms of our example at the beginning of this chapter, a tree containing cookies may have cookies that are assigned when a user first visits a website. A user coming into the website will interact for a while and then leave, probably not coming back soon again. The set of users who are interacting with the web server will change over time but it is always a relatively small subset compared to the overall number of entries in the tree. If we could store the cookies of the recent users closer to the top of the tree, we might be able to improve the overall time for looking up and inserting a new value in the tree. The complexity won't improve. Inserting an item will still take $\Theta(log\ n)$ time. But the overall time to insert or lookup an item might improve a little bit. This is the motivation for a splay tree.

In a splay tree, each insert or lookup moves the inserted or looked up value to the root of the tree through a process called *splaying*. When deleting a value, the parent may be splayed to the root of the tree. A splay tree is still a binary search tree. Splay trees usually remain well-balanced but unlike an AVL tree, a splay tree does not contain any balance or height information. Splaying a node to the root involves a series of rotates, much like the rotates of AVL trees, but with a slight difference.

It is interesting to note that while splay trees are designed to exploit *spatial locality* in the data, they are not dependent on spatial locality to perform well. Splay trees function as well or better than AVL trees in practice on completely random data sets.

There are several things that are interesting about splay trees.

- First, the splaying process does not require the balance or any other information about the height of subtrees. The binary search tree structure is good enough.
- Splay trees don't stay perfectly balanced all the time. However, because they stay relatively balanced, they are balanced enough to get an average case complexity of $\Theta(log\ n)$ for insert, lookup, and delete operations. This idea that they are good enough is the basis for what is called *amortized complexity* which is discussed later in Chap. 2 and later in this chapter.
- Splaying is relatively simple to implement.

In this text we cover two bottom-up splay tree implementations. Splay trees can be implemented either iteratively or recursively and we examine both implementations. In Chap. 6 binary search tree insert was implemented recursively. If splaying is to be done recursively, the splay can be part of the insert function. If written iteratively, a stack can be used in the splaying process. The following sections cover both the iterative and recursive implementations. But first we examine the rotations that are used in splaying.

10.4.1 Splay Rotations

Each time a value is inserted or looked up the node containing that value is splayed to the top through a series of rotate operations. Unlike AVL trees, a splay tree employs a double rotation to move a node up to the level of its grandparent if a grandparent exists. Through a series of double rotations the node will either make it to the root or to the child of the root. If the splayed node makes it to the child of the root, a single rotation is used to bring it to the root.

The single rotate functions are often labelled a *zig* or a *zag* while the double rotations are called *zig-zig* or *zig-zag* operations depending on the direction of the movement of the splayed node. Sometimes the node moves with a zig-zag motion while other times it moves with a zig-zig motion.

Splaying happens when a value is inserted into, looked up, or deleted from a splay tree. When a value is looked up either the searched value is splayed to the top or the would-be parent of the value if the value is not found in the tree. Deletion from the tree can be implemented like delete from any other binary search tree as described in problem 2 of Chap. 6. When a value is deleted from a binary search tree the parent of the deleted node is splayed to the root of the tree.

The example in Fig. 10.14 depicts the splay operations that result from inserting the green nodes into a splay tree. When 30 is inserted, it is splayed to the root of the tree as appears in the second version of the tree (the red nodes). When 5 is inserted, it is splayed to the root as well. Moving 5 to the root is accomplished through a zig-zig rotation called a double-right rotation. Splaying the 8 to the root is the result of a zig-zag rotation called a right-left rotation. When the 42 is splayed to the root it is a double-left rotation followed by a single left rotation.

Splaying the 15 to the root is accomplished by doing a double-right rotation followed by a left-right rotation. The double-right is often called a zig-zig rotation as is the double-left rotation. The left-right and right-left rotations are often called zig-zag rotations. The end result in each case has the newly inserted node, or looked up node, splayed to the root of the tree.

Figures 10.10, 10.11, 10.12 and 10.13 depict these splay operations. Figures 10.12 and 10.13 give some intuitive understanding of why splay trees work as well as they

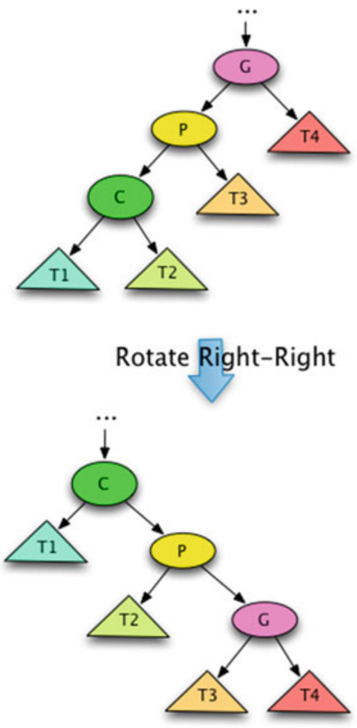

Fig. 10.10 Splay Tree Double-Right Rotate

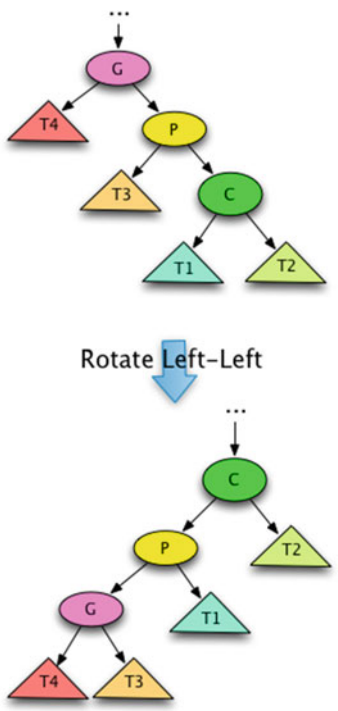

Fig. 10.11 Splay Tree Double-Left Rotate

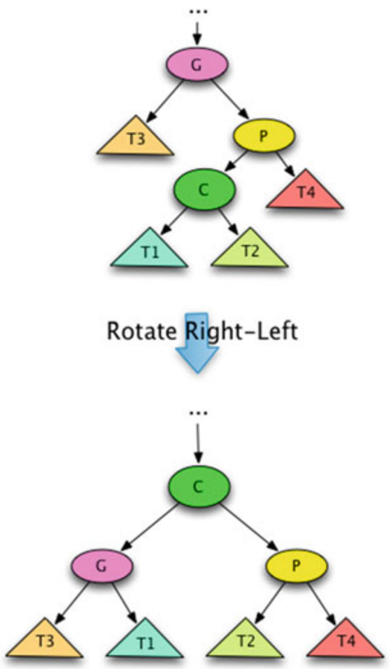

Fig. 10.12 Splay Tree Right-Left Rotate

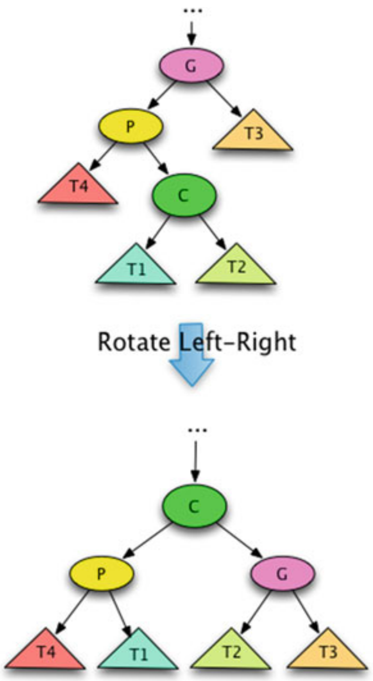

Fig. 10.13 Splay Tree Left-Right Rotate

do. After the rotate operations depicted in Figs. 10.12 and 10.13 the subtree rooted at the child appears to be more balanced than before those rotations.

Notice that doing a left-right rotation is not the same as doing a left rotation followed by a right rotation. The splay left-right rotate yields a different result. Likewise, the splay right-left rotate yields a different result than a right followed by a left rotation. Splay zig-zag rotates are designed this way to help balance they tree. Figures 10.12 and 10.13 depict trees that might be slightly out of balance before the rotation, brought into much better balance by the right-left rotation or the left-right rotation.

10.5 Iterative Splaying

Each time a value is inserted or looked up it is splayed to the root of the splay tree through a series of rotations as described in the previous section. The double rotation operations will either move the value to the root or the child of the root of the tree. If the double rotates result in the newly inserted value at the child of the root of the tree, a single rotate is used to move the newly inserted value to the root as depicted in Fig. 10.14 when 30 and 15 are inserted into the splay tree.

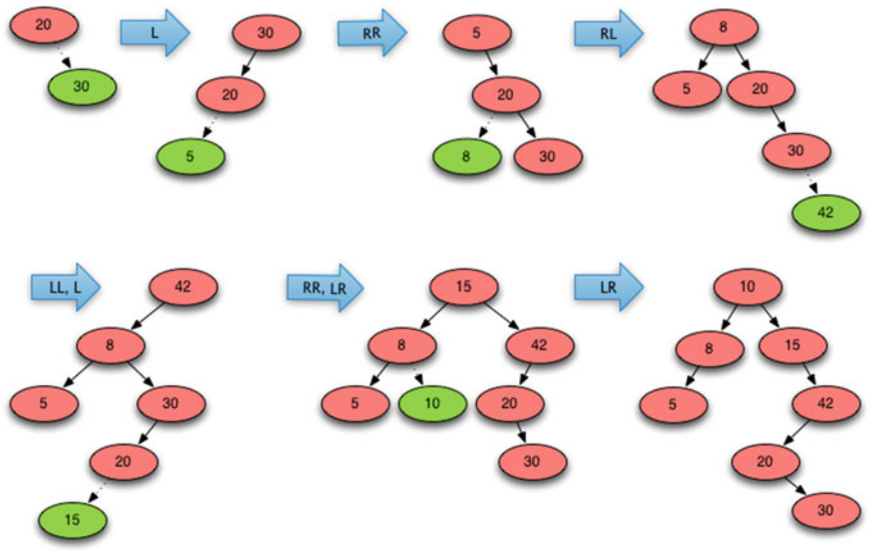

Fig. 10.14 Splay Tree Example

Inserting a new value into a binary search tree without recursion is possible using a while loop. The while loop moves from the root of the tree to the leaf node which will become the new node's parent at which point the loop terminates, the new node is created, and the parent is hooked up to its new child.

After inserting the new node, it must be splayed to the top. To splay it is necessary to know the path that was taken through the tree to the newly inserted node. This path can be recorded using a stack. As the insert loop passes through another node in the tree, it is pushed onto the stack. The end result is that all nodes, from the root to the new child, on the path to the new child are pushed onto this path stack.

Finally, splaying can occur by emptying this path stack. First the child is popped from the stack. Then, the rest of the stack is emptied as follows.

- If two more nodes are available on the stack they are the parent and grandparent of the newly inserted node. In that case a double rotate can be performed resulting in the root of the newly rotated subtree being the newly inserted node. Which double rotation is required can be determined from the values of the grandparent, parent, and child.
- If only one node remains on the stack it is the parent of the newly inserted node. A single rotation will bring the newly inserted node to the root of the splay tree.

Implementing splay in the manner described here works well when looking up a value in the tree, whether it is found or not. When a value is found it will be added to the path stack. When a value is not found, the parent should be splayed to the top, which naturally occurs when the looked up value is not found because the parent will be left on the top of the path stack when splaying is performed.

One method of deleting a node from a splay tree is accomplished by deleting just as you would in a binary search tree. If the node to delete has zero or one child it is trivial to delete the node. If the node to delete has two children, then the leftmost value in its right subtree can replace the value in the node to delete and the leftmost value can be deleted from the right subtree. The parent of the deleted node is splayed to the top of the tree.

Another method of deletion requires splaying the deleted node to the root of the tree first. Then the rightmost value of the left subtree is splayed to the root. After splaying the left subtree, its root node's right subtree is empty and the original right subtree can be added to it. The original left subtree becomes the root of the newly constructed splay tree.

10.6 Recursive Splaying

Implementing splaying recursively follows the recursive insert operation on binary search trees. The splaying is combined with this recursive insert function. As the recursive insert follows the path down the tree it builds a rotate string of "R" and "L". If the new item is inserted to the right of the current root node, then a left rotate will be required to splay the newly inserted node up the tree and an "L" is added to the rotate string. Otherwise, a right rotate will be required and an "R" is added to the rotate string.

As the recursive insert function returns, the path to the newly inserted node is retraced by the returning function. The last two characters in the rotate string dictate what double rotation is required. A dictionary or hash table takes care of mapping "RR", "RL", "LR", and "LL" to the appropriate rotate functions. The hash table lookup is used to call the appropriate rotation and the rotate string is truncated (or re-initialized to the empty string depending on when "R" and "L" are added to the rotate string). When the recursive insert is finished, any required single rotation will be recorded in the rotate string and can be performed.

It should be noted that implementing splaying using a rotate string and hash table like this requires about one half the conditional statements to determine the required rotations as compared to the iterative algorithm described above. When inserting a new node the path must be determined by comparing the value to insert to each node on the path to its location in the tree. In the iterative description above, the values on the path are again compared during splaying. In this recursive description the new item is only compared to each item on the path once. This has an impact on performance as shown later in the chapter.

Looking up a value using this recursive implementation works similarly to insert either splaying the found value or its parent if it is not found to the root of the tree. Deleting a value again can be done recursively by first looking up the value to delete resulting in it being splayed to the root of the tree and then performing the method of root removal described in the previous section.

10.7 Performance Analysis

In the worst case a splay tree may become a stick resulting in $\Theta(n)$ complexity for each lookup, insert, and delete operation while AVL trees guarantee $\Theta(log\ n)$ time for lookup, insert, and delete operations. It would appear that AVL trees might have better performance. However, this does not seem to be the case in practice. Close to 100,000 insert and 900,000 random lookups were performed in an experiment using a pre-generated dataset. The insert and lookup operations were identified in the dataset with all looked up values being found in the tree. The average combined insert and lookup time were recorded in Fig. 10.15 for an AVL tree, a splay tree implemented iteratively, and the recursive implementation of splay tree insert and lookup. The results show that the recursive splay tree implementation performs better on a random set of values than the AVL tree implementation. The experiment suggests that splay trees also exhibit $\Theta(log\ n)$ complexity in practice for insert and lookup operations.

In Figs. 10.13 and 10.12 we got an intuitive understanding of how splay trees maintain balance through their specialized double rotations. However, it is not a very convincing argument to say that the double rotations appear to make the tree more balanced. This idea is formalized using *amortized complexity*. Amortization, first encountered in Chap. 2, is an accounting term used when an expense is spread over a number of years as opposed to expensing it all in one year. This same principle can be applied to the expense in finding or inserting a value in a Splay Tree. The complete

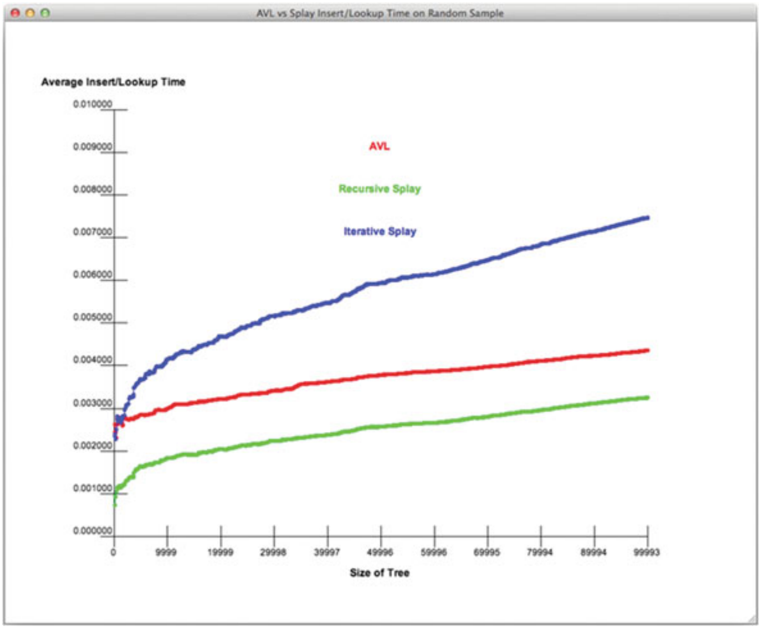

Fig. 10.15 Average Insert/Lookup Time

analysis of this is done on a case by case basis and is not present in this text but may be found in texts on-line. These proofs show that splay trees do indeed operate as efficiently as AVL trees on randomly accessed data. In addition, the splaying operation used when inserting or looking up a value exploits *spatial locality* in the data. Data values that are frequently looked up will make their way toward the top of the tree so as to be more efficiently looked up in the future. While taking advantage of spatial locality is certainly desirable if present in the data, it does not improve the overall computational complexity of splay tree insert and lookup operations.

However, this does not happen in the average case on randomly inserted and looked up values. In fact, the recursive implementation of splay trees presented in the previous section exhibits $\Theta(log\ n)$ average insert and lookup time on a randomly distributed set of values and performs better in a random sample than the AVL tree implementation.

Insert, lookup, and delete operations on an AVL tree can be completed in $\Theta(log\ n)$ time. In the average case this holds for splay trees as well. Traversal of an AVL or splay tree runs in $\Theta(n)$ time and yields its items in ascending or descending order (depending on how the iterator is written). While the quicksort algorithm can sort the items of a list just as efficiently, AVL and splay trees are data structures that allow many insert and delete operations while still maintaining the ordering of their elements. An AVL or splay tree may be a practical choice if a data structure is needed that efficiently implements lookup, delete, and insert operations while also allowing the sequence of values to be iterated over in ascending or descending order. The advantage of AVL trees lies in their ability to maintain the ordering of elements while guaranteeing efficient lookup, insert, and delete complexity. Splay trees work just as well in almost all cases and in the case of the recursive splay tree implementation described in this chapter it performs even better than the AVL Tree implementation on random data sets. The difference in performance between the AVL tree and the recursive splay tree performance numbers is the difference between maintaining the balance explicitly in the AVL tree and getting *good enough* balance in the splay tree.

10.8 Chapter Summary

This chapter presented several implementations of height-balanced AVL trees and splay trees. Recursive and iterative insert algorithms were presented. Both balance maintaining and height maintaining AVL nodes were discussed. The recursive insert algorithms for both AVL and splay trees result in very clean code without many special cases, while the iterative versions needs a few more if statements to handle some conditions. In some instance the iterative version may be slightly more efficient than the recursive version since there is a cost associated with function calls in any language, but the experimental results obtained from the experiments performed in this chapter seem to suggest that the recursive implementations operate very efficiently when written in Python.

10.9 Review Questions

Answer these short answer, multiple choice, and true/false questions to test your mastery of the chapter.

1. What is the balance of a node in an AVL tree?
2. How does the balance of a node relate to its height?
3. How does an AVL tree make use of the balance of a node?
4. What is a pivot node?
5. What is a bad child in relationship to AVL trees?
6. What is the path stack and when is it necessary?
7. After doing a right rotation, where is the pivot node and the bad child in the subtree that was originally rooted at the pivot?
8. Why is the balance of the root of a subtree always 0 after code for case 3 is executed?
9. In the two subcases for case 3, what node becomes the root node of the subtree rooted at the pivot after executing the algorithm on each of the subcases?
10. Why does the AVL tree insert algorithm always completes in $\Theta(log\ n)$ time? Do a case by case analysis to justify your answer for each of the three cases involved in inserting a value.
11. What is the purpose of the rotate string in the recursive insert splay tree implementation?
12. Why does it seem that the recursive splay tree insert and lookup implementation operates faster than the AVL tree implementation?

10.10 Programming Problems

1. Write an AVL tree implementation that maintains balances in each node and implements insert iteratively. Write a test program to thoroughly test your program on some randomly generated data.
2. Write an AVL tree implementation that maintains balances in each node and implements insert recursively. Write a test program to thoroughly test your program on some randomly generated data.
3. Write an AVL tree implementation that maintains heights in each node and implements insert recursively. Write a test program to thoroughly test your program on some randomly generated data.
4. Write an AVL tree implementation that maintains heights in each node and implements insert iteratively. Write a test program to thoroughly test your program on some randomly generated data.
5. Complete programming problem 3. Then implement the delete operation for AVL Trees. Finally, write a test program to thoroughly test your data structure. As values are inserted and deleted from your tree you should test your code to make sure it maintains all heights correctly and the ordering of all values in the tree.

6. Implement two of the programming problems 1–4 in this chapter and then write a test program that generates a random list of integers. Time inserting the values into the first implementation and then time inserting each value into the second implementation. Record all times in the XML format needed by the PlotData.py program from chapter two. Plot the timing of the two algorithms to compare their relative efficiency.
7. Write a splay tree implementation with recursive insert and lookup functions. Implement an AVL tree either iteratively or recursively where the height of each node is maintained. Run a test where trees are built from the same list of values. When you generate the list of values, duplicate values should be considered a lookup. Write the data file with an *L* or an *I* followed by a value which indicates either a *lookup* or *insert* operation should be performed. Generate an XML file in the format used by the PlotData.py program to compare your performance results.
8. Write a splay tree implementation with recursive insert and lookup functions. Compare it to one of the other balanced binary tree implementations detailed in this chapter. Run a test where trees are built from the same list of values. When you generate the list of values, duplicate values should be considered a lookup. Write the data file with an *L* or an *I* followed by a value which indicates either a *lookup* or *insert* operation should be performed. Generate an XML file in the format used by the PlotData.py program to compare your performance results.