CHAPTER 20

AVL Trees

Objectives

- To know what an AVL tree is (§20.1).
- To understand how to rebalance a tree using the LL rotation, LR rotation, RR rotation, and RL rotation (§20.2).
- To know how to design the AVLTree class (§20.3).
- To insert elements into an AVL tree (§20.4).
- To implement node rebalancing (§20.5).
- To delete elements from an AVL tree (§20.6).
- To implement the AVLTree class (§20.7).
- To test the AVLTree class (§20.8).
- To analyze the complexity of search, insert, and delete operations in AVL trees (§20.9).
20.1 Introduction

Key Point: AVL Tree is a balanced binary search tree.

Chapter 19 introduced binary search trees. The search, insertion, and deletion time for a binary tree depend on the height of the tree. In the worst case, the height is $O(n)$. If a tree is perfectly balanced—i.e., a complete binary tree—its height is $\log n$. Can we maintain a perfectly balanced tree? Yes. But doing so will be costly. The compromise is to maintain a well-balanced tree—i.e., the heights of two subtrees for every node are about the same.

AVL trees are well balanced. AVL trees were invented in 1962 by two Russian computer scientists G. M. Adelson-Velsky and E. M. Landis. In an AVL tree, the difference between the heights of two subtrees for every node is 0 or 1. It can be shown that the maximum height of an AVL tree is $O(\log n)$.

The process for inserting or deleting an element in an AVL tree is the same as in a regular binary search tree. The difference is that you may have to rebalance the tree after an insertion or deletion operation. The balance factor of a node is the height of its right subtree minus the height of its left subtree. A node is said to be balanced if its balance factor is -1, 0, or 1. A node is said to be left-heavy if its balance factor is -1. A node is said to be right-heavy if its balance factor is +1.

Pedagogical NOTE

Run from

[www.cs.armstrong.edu/liang/animation/AVLTreeAnimation.htm](http://www.cs.armstrong.edu/liang/animation/AVLTreeAnimation.htm)

1 to see how an AVL tree works, as shown in Figure 20.1.
Figure 20.1

The animation tool enables you to insert, delete, and search elements visually.

20.2 Rebalancing Trees

Key Point: After inserting or deleting an element from an AVL tree, if the tree becomes unbalanced, perform a rotation operation to rebalance the tree.

If a node is not balanced after an insertion or deletion operation, you need to rebalance it. The process of rebalancing a node is called a rotation. There are four possible rotations.

**LL Rotation:** An LL imbalance occurs at a node $A$ such that $A$ has a balance factor $-2$ and a left child $B$ with a balance factor $-1$ or $0$, as shown in Figure 20.2a. This type of imbalance can be fixed by performing a single right rotation at $A$, as shown in Figure 20.2b.
Figure 20.2

LL rotation fixes LL imbalance.

RR Rotation: An RR imbalance occurs at a node $A$ such that $A$ has a balance factor $+2$ and a right child $B$ with a balance factor $+1$ or $0$, as shown in Figure 20.3a. This type of imbalance can be fixed by performing a single left rotation at $A$, as shown in Figure 20.3b.

Figure 20.3

RR rotation fixes RR imbalance.
**LR Rotation:** An *LR imbalance* occurs at a node $A$ such that $A$ has a balance factor $-2$ and a left child $B$ with a balance factor $+1$, as shown in Figure 20.4a. Assume $B$’s right child is $C$. This type of imbalance can be fixed by performing a double rotation at $A$ (first a single left rotation at $B$ and then a single right rotation at $A$), as shown in Figure 20.4b.

![LR Rotation Diagram](image)

Figure 20.4

*LR rotation fixes LR imbalance.*

**RL Rotation:** An *RL imbalance* occurs at a node $A$ such that $A$ has a balance factor $+2$ and a right child $B$ with a balance factor $-1$, as shown in Figure 20.5a. Assume $B$’s left child is $C$. This type of imbalance can be fixed by performing a double rotation at $A$ (first a single right rotation at $B$ and then a single left rotation at $A$), as shown in Figure 20.5b.

![RL Rotation Diagram](image)
**Figure 20.5**

*RL rotation fixes RL imbalance.*

**Check point**

20.1 What is an AVL tree? Describe the following terms: balance factor, left-heavy, and right-heavy.

20.2 Show the balance factor of each node in the tree, as shown in Figure 20.6.

![AVL Tree Diagram](attachment:avl_tree.png)

(a) (b)

**Figure 20.6**

*A balance factor determines whether a node is balanced.*

20.3 Describe LL rotation, RR rotation, LR rotation, and RL rotation for an AVL tree.

20.4 For the AVL tree in Figure 20.6a, show the new AVL tree after adding element 89. What rotation did you perform in order to rebalance the tree? Which node was unbalanced?

20.5 For the AVL tree in Figure 20.6a, show the new AVL tree after adding element 40. What rotation did you perform in order to rebalance the tree? Which node was unbalanced?
20.6 For the AVL tree in Figure 20.6a, show the new AVL tree after adding element 40. What rotation did you perform in order to rebalance the tree? Which node was unbalanced?

20.7 For the AVL tree in Figure 20.6a, show the new AVL tree after adding element 40. What rotation did you perform in order to rebalance the tree? Which node was unbalanced?

20.8 For the AVL tree in Figure 20.6a, show the new AVL tree after adding element 47. What rotation did you perform in order to rebalance the tree? Which node was unbalanced?

20.9 For the AVL tree in Figure 20.6a, show the new AVL tree after adding element 81. What rotation did you perform in order to rebalance the tree? Which node was unbalanced?

20.10 For the AVL tree in Figure 20.6b, show the new AVL tree after adding element 200. What rotation did you perform in order to rebalance the tree? Which node was unbalanced?

### 20.3 Designing Classes for AVL Trees

Key Point: Since an AVL tree is a binary search tree, `AVLTree` is designed as a subclass of `BST`.

An AVL tree is a binary tree. So you can define the `AVLTree` class to extend the `BST` class, as shown in Figure 20.7. The `BST` and `TreeNode` classes are defined in §26.2.5.
AVLTree

AVLTree()
createNewNode(): AVLTreeNode
insert(e: object): bool
delete(e: object): bool

updateHeight(node: AVLTreeNode):
None
balancePath(e: object): None

balanceFactor(node: AVLTreeNode): int
balanceLL(A: TreeNode, parentOfA:
TreeNode): None
balanceLR(A: TreeNode, parentOfA:
TreeNode): None
balanceRR(A: TreeNode, parentOfA:
TreeNode): None
balanceRL(A: TreeNode, parentOfA:
TreeNode): None

1

Figure 20.7

The AVLTree class extends BST with new implementations for the insert and delete methods.

In order to balance the tree, you need to know each node’s height. For convenience, store the height of each node in AVLTreeNode and define AVLTreeNode to be a subclass of BinaryTree.TreeNode. Note that TreeNode is defined as a static inner class in BST. AVLTreeNode will be defined as a static inner class. TreeNode contains the data fields element, left, and right, which are inherited in AVLTreeNode. So, AVLTreeNode contains four data fields, as pictured in Figure 20.8.

node: AVLTreeNode

element: E
height: int
left: TreeNode<E>
right: TreeNode<E>

Figure 20.8

An AVLTreeNode contains protected data fields element, height, left, and right.
In the **BST** class, the `createNewNode()` method creates a `TreeNode` object. This method is overridden in the **AVLTree** class to create an `AVLTreeNode`. Note that the return type of the `createNewNode()` method in the **BinaryTree** class is `TreeNode`, but the return type of the `createNewNode()` method in **AVLTree** class is `AVLTreeNode`. This is fine, since `AVLTreeNode` is a subtype of `TreeNode`.

Searching an element in an **AVLTree** is the same as searching in a regular binary tree. So, the `search` method defined in the **BST** class also works for **AVLTree**.

The `insert` and `delete` methods are overridden to insert and delete an element and perform rebalancing operations if necessary to ensure that the tree is balanced.

### 20.4 Overriding the `insert` Method

**Key Point:** *Inserting an element into an AVL tree is the same as inserting it to a BST, except that the tree may need to be rebalanced.*

A new element is always inserted as a leaf node. The heights of the ancestors of the new leaf node may increase, as a result of adding a new node. After insertion, check the nodes along the path from the new leaf node up to the root. If a node is found unbalanced, perform an appropriate rotation using the following algorithm in Listing 20.1.

#### Listing 20.1 Balancing Nodes on a Path

```python
balancePath(e):
    # Get the path from the node that contains element e to the root, as illustrated in Figure 20.9
    for each node A in the path leading to the root:
        # Update the height of A
        let parentOfA denote the parent of A, which is the next node in the path, or None if A is the root

        if balanceFactor(A) == -2:
            if balanceFactor(A.left) == -1 or 0:
                Perform LL rotation # See Figure 20.2
            else:
                Perform LR rotation # See Figure 20.4
        else balanceFactor(A) == +2:
            if balanceFactor(A.right) == +1 or 0:
                Perform RR rotation # See Figure 20.3
```
Perform RR rotation  # See Figure 20.3

else:
    Perform RL rotation  # See Figure 20.5

Figure 20.9

The nodes along the path from the new leaf node may become unbalanced.

The algorithm considers each node in the path from the new leaf node to the root. Update the height of the node on the path. If a node is balanced, no action is needed. If a node is not balanced, perform an appropriate rotation.

20.5 Implementing Rotations

Key Point: An unbalanced tree becomes balanced by performing an appropriate rotation operation.

Section 20.2, “Rebalancing Tree,” illustrated how to perform rotations at a node. Listing 20.2 gives the algorithm for the LL rotation, as pictured in Figure 20.2.

Listing 20.2 LL Rotation Algorithm

\[
\text{balanceLL}(A, \text{parentOfA}): \\
\text{Let } B \text{ be the left child of } A. \\
\text{if } A \text{ is the root:} \\
\text{Let } B \text{ be the new root} \\
\text{else:} \\
\text{if } A \text{ is a left child of parentOfA:} \\
\text{Let } B \text{ be a left child of parentOfA} \\
\text{else:} \\
\text{Let } B \text{ be a right child of parentOfA}
\]
Make T2 the left subtree of A by assigning B.right to A.left
Make A the left child of B by assigning A to B.right
Update the height of node A and node B

Note that the height of nodes A and B may be changed, but the heights of other nodes in the tree are not changed. Similarly, you can implement the RR rotation, LR rotation, and RL rotation.

### 20.6 Implementing the delete Method

**Key Point:** Deleting an element from an AVL tree is the same as deleting it from a BST, except that the tree may need to be rebalanced.

As discussed in Section 20.3, “Deleting Elements in a BST,” to delete an element from a binary tree, the algorithm first locates the node that contains the element. Let current point to the node that contains the element in the binary tree and parent point to the parent of the current node. The current node may be a left child or a right child of the parent node. Three cases arise when deleting an element:

**Case 1:** The current node does not have a left child, as shown in Figure 20.10a. To delete the current node, simply connect the parent with the right child of the current node, as shown in Figure 20.10b.

The height of the nodes along the path from the parent up to the root may have decreased. To ensure the tree is balanced, invoke

```
balancePath(parent.element)  # Defined in Listing 20.1
```

**Case 2:** The current node has a left child. Let rightMost point to the node that contains the largest element in the left subtree of the current node and parentOfRightMost point to the parent node of the rightMost node, as shown in Figure 20.12a. The rightMost node cannot have a right child but may have a left child. Replace the element value in the current node with the one in the rightMost node, connect the parentOfRightMost node with the left child of the rightMost node, and delete the rightMost node, as shown in Figure 20.12b.
The height of the nodes along the path from parentOfRightMost up to the root may have decreased. To ensure that the tree is balanced, invoke

\[
\text{balancePath(parentOfRightMost)} \quad \# \text{ Defined in Listing 20.1}
\]

20.7 The AVLTree Class

Key Point: The AVLTree class extends the BST class to override the insert and delete methods to rebalance the tree if necessary.

Listing 20.3 gives the complete source code for the AVLTree class.

```
Listing 20.3 AVLTree.py

from BST import BST
from BST import TreeNode

class AVLTree(BST):
    def __init__(self):
        BST.__init__(self)

    # Override the createNewNode method to create an AVLTreeNode
    def createNewNode(self, e):
        return AVLTreeNode(e)

    # Override the insert method to balance the tree if necessary
    def insert(self, o):
        successful = BST.insert(self, o)
        if not successful:
            return False # o is already in the tree
        else:
            self.balancePath(o) # Balance from o to the root if necessary
        return True # o is inserted

    # Update the height of a specified node
    def updateHeight(self, node):
        if node.left == None and node.right == None: # node is a leaf
            node.height = 0
        elif node.left == None: # node has no left subtree
            node.height = 1 + (node.right).height
        elif node.right == None: # node has no right subtree
            node.height = 1 + (node.left).height
        else:
            node.height = 1 + max((node.right).height, (node.left).height)

    # Balance the nodes in the path from the specified
    # node to the root if necessary
    def balancePath(self, o):
        path = BST.path(self, o);
        for i in range(len(path) - 1, -1, -1):
            A = path[i]
            self.updateHeight(A)
            parentOfA = None if (A == self.root) else path[i - 1]
            if self.balanceFactor(A) == -2:
```

© Copyright 2012 by Pearson Education, Inc. All Rights Reserved.
if self.balanceFactor(A.left) <= 0:
    self.balanceLL(A, parentOfA) # Perform LL rotation
else:
    self.balanceLR(A, parentOfA) # Perform LR rotation
elif self.balanceFactor(A) == +2:
    if self.balanceFactor(A.right) >= 0:
        self.balanceRR(A, parentOfA) # Perform RR rotation
    else:
        self.balanceRL(A, parentOfA) # Perform RL rotation

# Return the balance factor of the node
def balanceFactor(self, node):
    if node.right == None: # node has no right subtree
        return -node.height
    elif node.left == None: # node has no left subtree
        return +node.height
    else:
        return (node.right).height - (node.left).height

# Balance LL (see Figure 14.2)
def balanceLL(self, A, parentOfA):
    B = A.left # A is left-heavy and B is left-heavy
    if A == self.root:
        self.root = B
    else:
        if parentOfA.left == A:
            parentOfA.left = B
        else:
            parentOfA.right = B
    A.left = B.right # Make T2 the left subtree of A
    B.right = A # Make A the left child of B
    self.updateHeight(A)
    self.updateHeight(B)

# Balance LR (see Figure 14.2(c))
def balanceLR(self, A, parentOfA):
    B = A.left # A is left-heavy
    C = B.right # B is right-heavy
    if A == self.root:
        self.root = C
    else:
        if parentOfA.left == A:
            parentOfA.left = C
        else:
            parentOfA.right = C
    A.left = C.right # Make T3 the left subtree of A
    B.right = C.left # Make T2 the right subtree of B
    C.left = B
    C.right = A
    # Adjust heights
    self.updateHeight(A)
    self.updateHeight(B)
    self.updateHeight(C)

# Balance RR (see Figure 14.2(b))
def balanceRR(self, A, parentOfA):
    B = A.right # A is right-heavy and B is right-heavy
    if A == self.root:
        self.root = B
    else:
if parentOfA.left == A:
    parentOfA.left = B
else:
    parentOfA.right = B

A.right = B.left # Make T2 the right subtree of A
B.left = A
self.updateHeight(A)
self.updateHeight(B)

# Balance RL (see Figure 14.2(d))

def balanceRL(self, A, parentOfA):
    B = A.right # A is right-heavy
    C = B.left # B is left-heavy

    if A == self.root:
        self.root = C
    else:
        if parentOfA.left == A:
            parentOfA.left = C
        else:
            parentOfA.right = C

    A.right = C.left # Make T2 the right subtree of A
    B.left = C.right # Make T3 the left subtree of B
    C.left = A
    C.right = B

    # Adjust heights
    self.updateHeight(A)
    self.updateHeight(B)
    self.updateHeight(C)

# Delete an element from the binary tree.
# Return True if the element is deleted successfully
# Return False if the element is not in the tree

def delete(self, element):
    if self.root == None:
        return False # Element is not in the tree

    # Locate the node to be deleted and also locate its parent node
    parent = None
    current = self.root
    while current != None:
        if element < current.element:
            parent = current
            current = current.left
        elif element > current.element:
            parent = current
            current = current.right
        else:
            break # Element is in the tree pointed by current

    if current == None:
        return False # Element is not in the tree

    # Case 1: current has no left children (See Figure 23.6)
    if current.left == None:
        # Connect the parent with the right child of the current node
        if parent == None:
            root = current.right
        else:
            if element < parent.element:
                parent.left = current.right
            else:
                parent.right = current.right
# Balance the tree if necessary
self.balancePath(parent.element)

else:
    # Case 2: The current node has a left child
    # Locate the rightmost node in the left subtree of
    # the current node and also its parent
    parentOfRightMost = current
    rightMost = current.left

    while rightMost.right != None:
        parentOfRightMost = rightMost
        rightMost = rightMost.right # Keep going to the right

    # Replace the element in current by the element in rightMost
    current.element = rightMost.element

    # Eliminate rightmost node
    if parentOfRightMost.right == rightMost:
        parentOfRightMost.right = rightMost.left
    else:
        # Special case: parentOfRightMost is current
        parentOfRightMost.left = rightMost.left

    # Balance the tree if necessary
    self.balancePath(parentOfRightMost.element)

self.size -= 1 # One element deleted

return True # Element inserted

# AVLTreeNode is TreeNode plus height

class AVLTreeNode(TreeNode):
    def __init__(self, e):
        self.height = 0 # New data field
        TreeNode.__init__(self, e)

The **AVLTree** class extends **BST** (line 4). Its constructor invokes its superclass’s constructor to initialize root and size property of a binary tree (line 6).

The **createNewNode()** method defined in the **BST** class creates a **TreeNode**. This method is overridden to return an **AVLTreeNode** (lines 9–10). This is a variation of the Factory Method Pattern.

Design Pattern: Factory Method Pattern
The **Factory Method pattern** defines an abstract method for creating an object, but lets subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

The **insert** method in **AVLTree** is overridden in lines 13–20. The method first invokes the **insert** method in **BST**, then invokes **balancePath(o)** (line 18) to ensure that the tree is balanced.
The balancePath method first gets the nodes on the path from the node that contains element o to the root (line 36). For each node in the path, update its height (line 39), check its balance factor (line 42), and perform appropriate rotations if necessary (lines 42–51).

Four methods for performing rotations are defined in lines 63–140. Each method is invoked with two TreeNode arguments A and parentOfA to perform an appropriate rotation at node A. How each rotation is performed is pictured in Figures 20.1–20.4. After the rotation, the heights of nodes A, B, and C are updated for the LL and RR rotations (lines 76, 98, 116, 138).

The delete method in AVLTree is overridden in lines 145–203. The method is the same as the one implemented in the BST class, except that you have to rebalance the nodes after deletion in two cases (lines 177, 200).

20.8 Testing the AVLTree Class

Key Point: This section gives an example of using the AVLTree class.

Listing 20.4 gives a test program. The program creates an AVLTree initialized with an array of integers 25, 20, and 5 (lines 6–7), inserts elements in lines 11–20, and deletes elements in lines 24–30.

Listing 20.4 TestAVLTree.py

```python
from AVLTree import AVLTree

def main():
    tree = AVLTree()
    tree.insert(25)
    tree.insert(20)
    tree.insert(5)
    print("After inserting 25, 20, 5:")
    printTree(tree)
    
    tree.insert(34)
    tree.insert(50)
    print("After inserting 34, 50:")
    printTree(tree)

main()
```
```python
tree.insert(30)
print("After inserting 30")
printTree(tree)

tree.insert(10)
print("After inserting 10")
printTree(tree)

tree.delete(34)
tree.delete(30)
tree.delete(50)
print("After removing 34, 30, 50:")
printTree(tree)

tree.delete(5)
print("After removing 5:")
printTree(tree)

def printTree(tree):
    # Traverse tree
    print("Inorder (sorted): ", end = "")
    tree.inorder()
    print("\nPostorder: ", end = "")
    tree.postorder()
    print("\nPreorder: ", end = "")
    tree.preorder()
    print("\nThe number of nodes is " + str(tree.getSize()))

main() # Call the main function

Sample output
After inserting 25, 20, 5:
Inorder (sorted): 5 20 25
Postorder: 5 25 20
Preorder: 20 5 25
The number of nodes is 3

After inserting 34, 50:
Inorder (sorted): 5 20 25 34 50
Postorder: 5 25 34 20
Preorder: 20 5 34 25 50
The number of nodes is 5

After inserting 30
Inorder (sorted): 5 20 25 30 34 50
Postorder: 5 20 30 50 34 25
Preorder: 25 20 5 34 30 50
The number of nodes is 6

After inserting 10
Inorder (sorted): 5 10 20 25 30 34 50
Postorder: 5 20 10 30 50 34 25
Preorder: 25 10 5 20 34 30 50
The number of nodes is 7

After removing 34, 30, 50:
```

Inorder (sorted): 5 10 20 25
Postorder: 5 20 25 10
Preorder: 10 5 25 20
The number of nodes is 4

After removing 5:
Inorder (sorted): 10 20 25
Postorder: 10 25 20
Preorder: 20 10 25
The number of nodes is 3

Figure 20.10 shows how the tree evolves as elements are added to the tree. After 25 and 20 are added, the tree is as shown in Figure 20.10a. 5 is inserted as a left child of 20, as shown in Figure 20.10b. The tree is not balanced. It is left-heavy at node 25. Perform an LL rotation to result an AVL tree, as shown in Figure 20.10c.

After inserting 34, the tree is shown in Figure 20.10d. After inserting 50, the tree is as shown in Figure 20.10e. The tree is not balanced. It is right-heavy at node 25. Perform an RR rotation to result in an AVL tree, as shown in Figure 20.10f.

After inserting 30, the tree is as shown in Figure 20.10g. The tree is not balanced. Perform an LR rotation to result in an AVL tree, as shown in Figure 20.10h.

After inserting 10, the tree is as shown in Figure 20.10i. The tree is not balanced. Perform an RL rotation to result in an AVL tree, as shown in Figure 20.10j.
Figure 20.10

The tree evolves as new elements are inserted.

Figure 20.11 shows how the tree evolves as elements are deleted. After deleting 34, 30, and 50, the tree is as shown in Figure 20.11b. The tree is not balanced. Perform an LL rotation to result an AVL tree, as shown in Figure 20.11c.

After deleting 5, the tree is as shown in Figure 20.11d. The tree is not balanced. Perform an RL rotation to result in an AVL tree, as shown in Figure 20.11e.
20.9 Maximum Height of an AVL Tree

Key Point: Since the height of an AVL tree is $O(\log n)$, the time complexity of the search, insert, and delete methods in $AVLTree$ is $O(\log n)$.

The time complexity of the search, insert, and delete methods in $AVLTree$ depends on the height of the tree. We can prove that the height of the tree is $O(\log n)$.

Let $G(h)$ denote the minimum number of the nodes in an AVL tree with height $h$. Obviously, $G(1)$ is 1 and $G(2)$ is 2. The minimum number of nodes in an AVL tree with height $h \geq 3$ must have two minimum subtrees: one with height $h - 1$ and the other with height $h - 2$. So,

$$G(h) = G(h - 1) + G(h - 2) + 1$$
Recall that a Fibonacci number at index \( i \) can be described using the recurrence relation

\[
F(i) = F(i - 1) + F(i - 2)
\]

So, the function \( G(h) \) is essentially the same as \( F(i) \). It can be proven that

\[
h < 1.4405 \log(n + 2) - 1.3277
\]

where \( n \) is the number of nodes in the tree. Therefore, the height of an AVL tree is \( O(\log n) \).

The search, insert, and delete methods involve only the nodes along a path in the tree. The updateHeight and balanceFactor methods are executed in a constant time for each node in the path. The balancePath method is executed in a constant time for a node in the path. So, the time complexity for the search, insert, and delete methods is \( O(\log n) \).

**Check point**

20.11 Why is the createNewNode method protected?

20.12 When is the updateHeight method invoked? When is the balanceFactor method invoked? When is the balancePath method invoked?

20.13 What are the data fields in the AVLTreeNode class? What are data fields in the AVLTree class?

20.14 In the insert and delete methods, once you have performed a rotation to balance a node in the tree, is it possible that there are still unbalanced nodes?

20.15 Show the change of an AVL tree when inserting 1, 2, 3, 4, 10, 9, 7, 5, 8, 6 into the tree, in this order.

20.16 For the tree built in the preceding question, show its change after 1, 2, 3, 4, 10, 9, 7, 5, 8, 6 are deleted from the tree in this order.

**Key Terms**
Chapter Summary

1. An AVL tree is a well-balanced binary tree. In an AVL tree, the difference between the heights of two subtrees for every node is 0 or 1.

2. The process for inserting or deleting an element in an AVL tree is the same as in a regular binary search tree. The difference is that you may have to rebalance the tree after an insertion or deletion operation.

3. Imbalances in the tree caused by insertions and deletions are rebalanced through subtree rotations at the node of the imbalance.

4. The process of rebalancing a node is called a rotation. There are four possible rotations: LL rotation, LR rotation, RR rotation, and RL rotation.

5. The height of an AVL tree is $O(\log n)$ . So, the time complexities for the search, insert, and delete methods are $O(\log n)$.

Multiple-Choice Questions

See multiple-choice questions online at

liang.armstrong.edu/selftest/selftestpy?chapter=20.
Programming Exercises

*20.1 (Displaying AVL tree graphically) Write an applet that displays an AVL tree along with its balance factor for each node.

20.2 (Comparing performance) Write a test program that randomly generates 500000 numbers and inserts them into a BST, reshuffles the 500000 numbers and performs search, and reshuffles the numbers again before deleting them from the tree. Write another test program that does the same thing for an AVLTree. Compare the execution times of these two programs.

**20.3 (Parent reference for BST) Suppose that the TreeNode class defined in BST contains a reference to the node’s parent, as shown in Exercise 7.20. Implement the AVLTree class to support this change. Write a test program that adds numbers 1, 2, ..., 100 to the tree and displays the paths for all leaf nodes.

**20.4 (The kth smallest element) You can find the kth smallest element in a BST in $O(n)$ time from an inorder iterator. For an AVL tree, you can find it in $O(\log n)$ time. To achieve this, add a new data field named size in AVLTreeNode to store the number of nodes in the subtree rooted at this node. Note that the size of a node $v$ is one more than the sum of the sizes of its two children. Figure 20.12 shows an AVL tree and the size value for each node in the tree.

![AVL Tree with size values](image-url)
The size data field in AVLTreeNode stores the number of nodes in the subtree rooted at the node.

In the AVLTree class, add the following method to return the $k$th smallest element in the tree.

```python
def find(k)
```

The method returns None if $k < 1$ or $k >$ the size of the tree. This method can be implemented using a recursive method `find(k, root)` that returns the $k$th smallest element in the tree with the specified root. Let $A$ and $B$ be the left and right children of the root, respectively. Assuming that the tree is not empty and $k \leq root.size$, `find(k, root)` can be recursively defined as follows:

```
find(k, root) =
    root.element, if $A$ is null and $k$ is 1;
    B.element, if $A$ is null and $k$ is 2;
    f(k, $A$), if $k$ $\leq$ $A$.size;
    root.element, if $k$ = $A$.size + 1;
    f($k - A$.size - 1, $B$), if $k$ > $A$.size + 1;
```

Modify the insert and delete methods in AVLTree to set the correct value for the size property in each node. The insert and delete methods will still be in $O(\log n)$ time. The find($k$) method can be implemented in $O(\log n)$ time. Therefore, you can find the $k$th smallest element in an AVL tree in $O(\log n)$ time.

***20.5 (AVL tree animation) Write a program that animates the AVL tree insert, delete, and search methods, as shown in Figure 20.1.

**20.6 (Closest pair of points) Section 23.8 introduced an algorithm for finding a closest pair of points in $O(n \log n)$ time using a divide-and-conquer approach. The algorithm was implemented
using recursion with a lot of overhead. Using the plain-sweep approach along with an AVL tree, you can solve the same problem in $O(n \log n)$ time. Implement the algorithm using an AVLTree.