

CHAPTER

9

AVL Trees

Objectives

- To know what an AVL tree is (§9.1).
- To understand how to rebalance a tree using the LL rotation, LR rotation, RR rotation, and RL rotation (§9.2).
- To know how to design the AVLTree class (§9.3).
- To insert elements into an AVL tree (§9.4).
- To implement node rebalancing (§9.5).
- To delete elements from an AVL tree (§9.6).
- To implement the AVLTree class (§9.7).
- To test the AVLTree class (§9.8).
- To analyze the complexity of search, insert, and delete operations in AVL trees (§9.9).

9.1 Introduction

<Side Remark: well-balanced tree>

Chapter 7 introduced binary trees. The search, insertion, and deletion time for a binary tree is dependent on the height of the tree. In the worst case, the height is $O(n)$. If the tree is well-balanced, i.e., the heights of two subtrees for every node are about the same, then the height of the entire tree is $O(\log n)$.

<Side Remark: AVL tree>

<Side Remark: $O(\log n)$ >

AVL trees are well-balanced. AVL trees were invented by two Russian computer scientists G. M. Adelson-Velsky and E. M. Landis in 1962. 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 $O(\log n)$.

<Side Remark: balance factor>

<Side Remark: balanced>

<Side Remark: left-heavy>

<Side Remark: right-heavy>

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.

9.2 Rebalancing Trees

<Side Remark: rotation>

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.

<Side Remark: LL imbalance>

<Side Remark: LL rotation>

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 9.1(a). This type of imbalance can be fixed by performing a single right rotation at A, as shown in Figure 9.1(b).

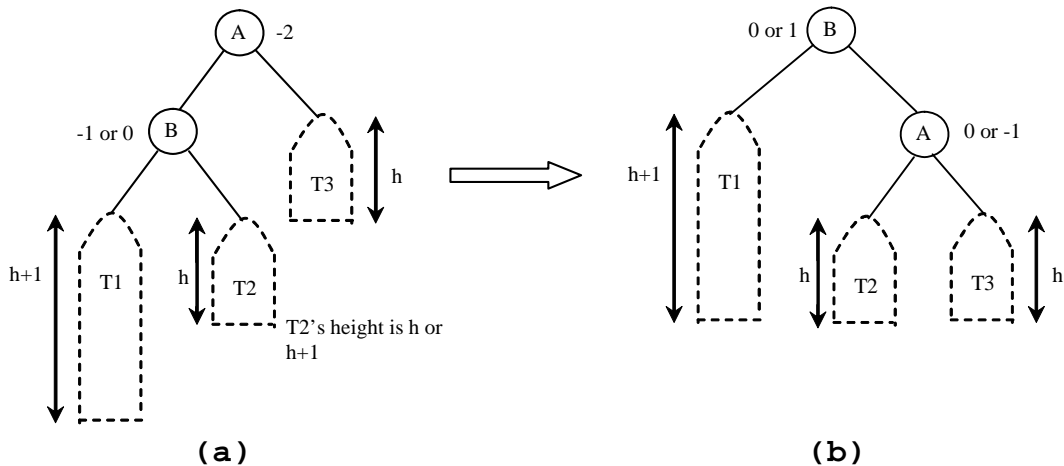


Figure 9.1
LL rotation fixes LL imbalance.

<Side Remark: RR imbalance>

<Side Remark: RR rotation>

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 9.2(a). This type of imbalance can be fixed by performing a single left rotation at A, as shown in Figure 9.2(b).

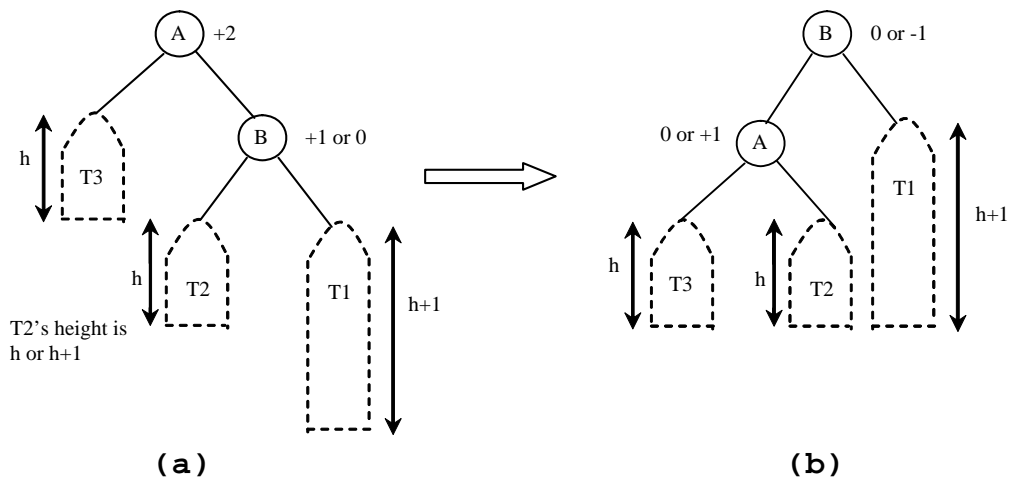


Figure 9.2
RR rotation fixes RR imbalance.

<Side Remark: LR imbalance>

<Side Remark: LR rotation>

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 9.1(a). 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 9.3(b).

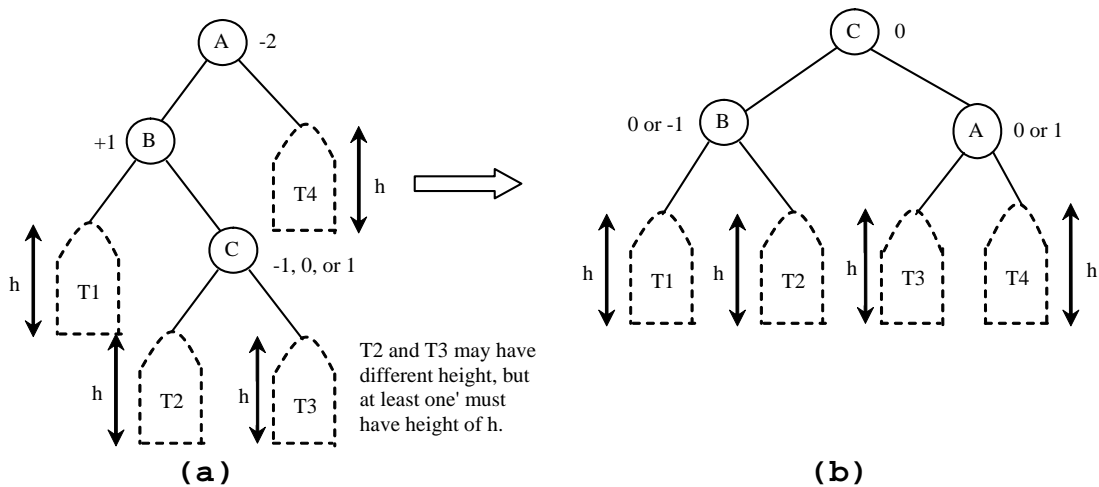


Figure 9.3
LR rotation fixes LR imbalance.

<Side Remark: RL imbalance>
 <Side Remark: RL rotation>

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 9.1(a). 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 9.4(b).

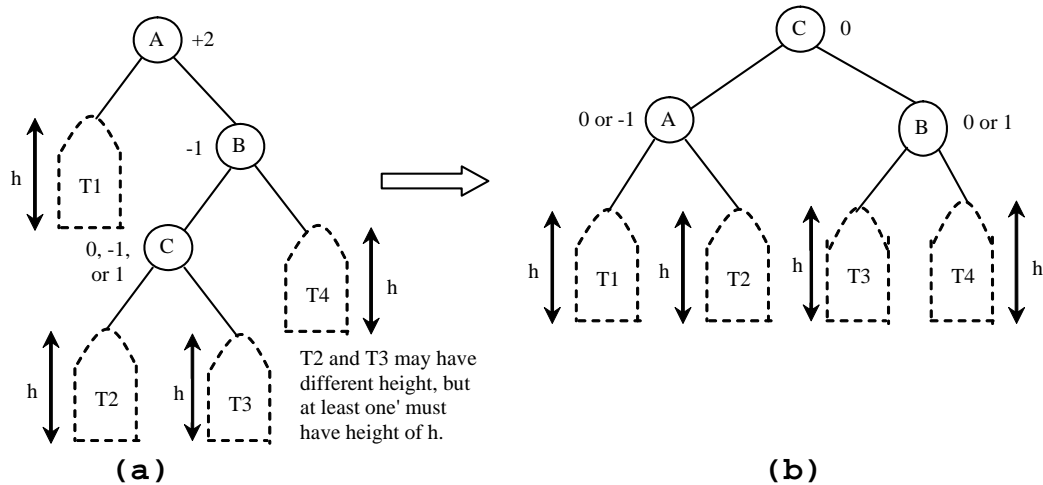


Figure 9.4
RL rotation fixes RL imbalance.

9.3 Designing Classes for AVL Trees

An AVL tree is a binary tree. So you can define the AVLTree class to extend the BinaryTree class, as shown in Figure 9.5.

***New Figure
 <PD: UML Class Diagram>

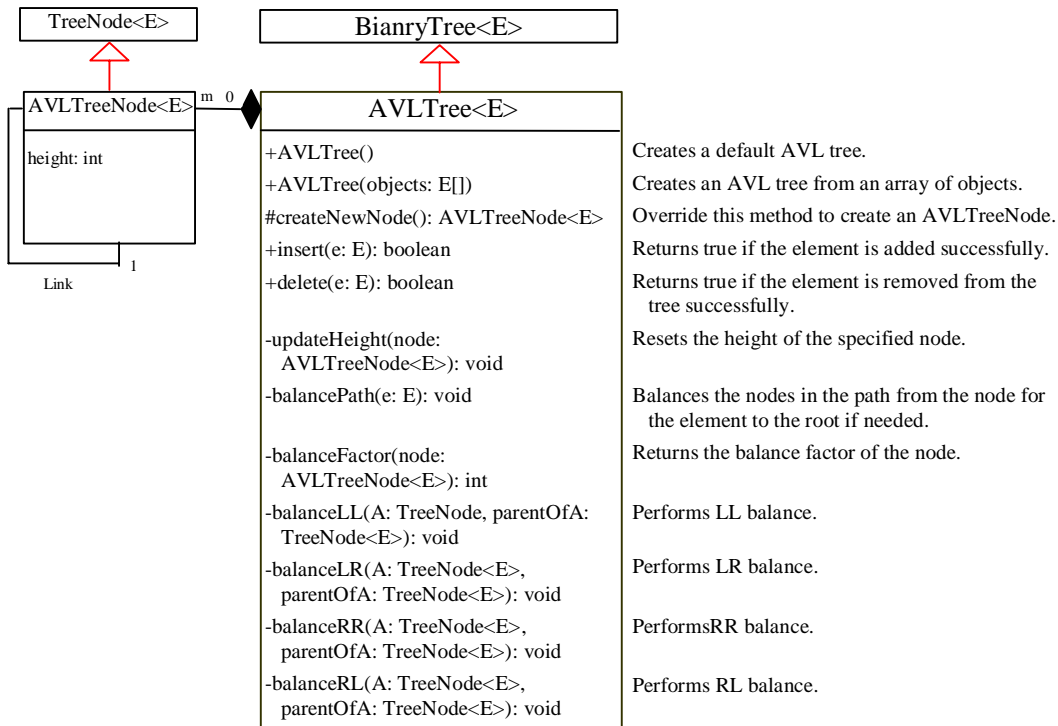


Figure 9.5

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

<Side Remark: AVLTreeNode>

In order to balance the tree, each node's height must be stored. So, the AVLTreeNode is defined to extend BinaryTree.TreeNod. Note that TreeNod is defined as a static inner class in BinaryTree. AVLTreeNode will be defined as a static inner class in AVLTree. Note that TreeNod contains the data fields element, left, and right, which are inherited in AVLTreeNode. So, AVATreeNode contains four data fields, as pictured in Figure 9.6.

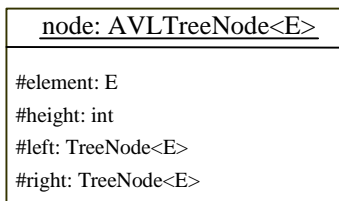


Figure 9.6

An AVLTreeNode contains protected data fields element, height, left, and right.

<Side Remark: createNewNode()>

In the BinaryTree class, the createNewNode() method creates a TreeNod 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 TreeNod, but the return type of the createNewNode() method in AVLTree class is AVLTreeNode. This is fine, since AVLTreeNode is a subtype of TreeNod.

Searching an element in an AVL is the same as searching in a regular binary tree. So, the search method defined in the BinaryTree 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.

Pedagogical NOTE

<side remark: AVL tree animation>

Run from

www.cs.armstrong.edu/liang/jds/exercisejds/Exercise9_3.html to see how an AVL tree works, as shown in Figure 9.8.

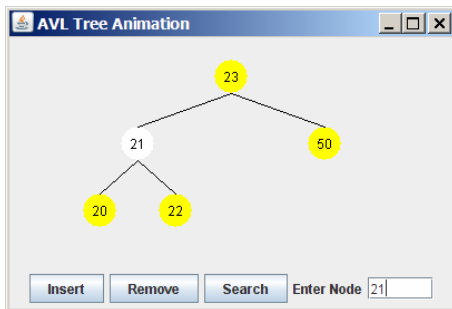


Figure 9.8

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

***End NOTE

9.4 Overriding the insert Method

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:

Listing 9.1 Balancing Nodes on a Path

```
***PD: Please add line numbers in the following code***
***Layout: Please layout exactly. Don't skip the space. This
is true for all source code in the book. Thanks, AU.
<Side Remark line 2: get the path>
<Side Remark line 5: update node height>
<Side Remark line 6: get parent node>
<Side Remark line 9: is balanced?>
<Side Remark line 11: LL rotation>
<Side Remark line 13: LR rotation>
<Side Remark line 16: RR rotation>
<Side Remark line 18: RL rotation>
```

```
balancePath(E o) {
```

```

    Get the path from the node that contains element o to the root,
    as illustrated in Figure 9.8;
    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 null if A is the root;

        switch (balanceFactor(A)) {
            case -2: if balanceFactor(A.left) = -1 or 0
                    Perform LL rotation; // See Figure 9.1
                else
                    Perform LR rotation; // See Figure 9.3
                break;
            case +2: if balanceFactor(A.right) = +1 or 0
                    Perform RR rotation; // See Figure 9.2
                else
                    Perform RL rotation; // See Figure 9.4
        } // End of switch
    } // End of for
} // End of method

```

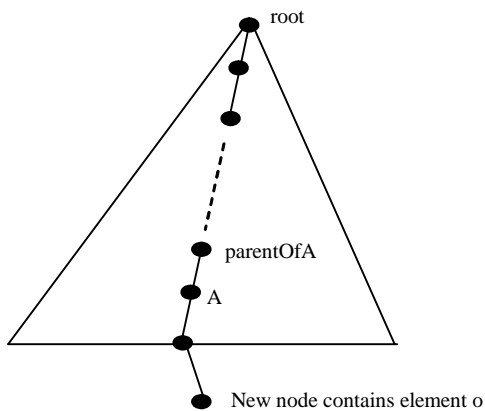


Figure 9.8

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.

9.5 Implementing Rotations

§9.2, "Rebalancing Tree," illustrated how to perform rotations at a node. Listing 9.2 gives the algorithm for the LL rotation, as pictured in Figure 9.1.

Listing 9.2 LL Rotation Algorithm

```

***PD: Please add line numbers in the following code***
***Layout: Please layout exactly. Don't skip the space. This
is true for all source code in the book. Thanks, AU.
<Side Remark line 2: left child of A>

```

<Side Remark line 4: reconnect B's parent>

<Side Remark line 13: move subtrees>

<Side Remark line 15: adjust height>

```
balanceLL(TreeNode A, TreeNode parentOfA) {  
  Let B be the left child of A.  
  
  if (A is the root)  
    Let B be the new root  
  else {  
    if (A is a left child of parentOfA)  
      Let B be a left child of parentOfA;  
    else  
      Let B 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;  
} // End of method
```

Similarly, you can implement the RR rotation, LR rotation, and RL rotation.

9.6 Implementing the delete Method

As discussed in §7.3, "Deleting Elements in a Binary Search Tree," 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 7.8(a). To delete the current node, simply connect the parent with the right child of the current node, as shown in Figure 7.8(b).

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);
```

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 7.10(a). 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 7.10(b).

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

```
balancePath(parentOfRightMost);
```

9.7 The AVLTree Class

Listing 9.3 gives the complete source code for the AVLTree class.

Listing 9.3 AVLTree.java

*****PD: Please add line numbers in the following code*****
*****Layout: Please layout exactly. Don't skip the space. This is true for all source code in the book. Thanks, AU.**

<Side Remark line 2: no-arg constructor>
<Side Remark line 7: constructor>
<Side Remark line 12: create AVL tree node>
<Side Remark line 17: override insert>
<Side Remark line 22: balance tree>
<Side Remark line 29: update node height>
<Side Remark line 45: balance nodes>
<Side Remark line 46: get path>
<Side Remark line 48: consider a node>
<Side Remark line 49: update height>
<Side Remark line 50: get height>
<Side Remark line 54: left-heavy>
<Side Remark line 56: LL rotation>
<Side Remark line 59: LR rotation>
<Side Remark line 62: right-heavy>
<Side Remark line 64: RR rotation>
<Side Remark line 67: RL rotation>
<Side Remark line 74: get balance factor>
<Side Remark line 85: LL rotation>
<Side Remark line 102: update height>
<Side Remark line 106: LR rotation>
<Side Remark line 129: update height>
<Side Remark line 135: RR rotation>
<Side Remark line 152: update height>
<Side Remark line 157: RL rotation>
<Side Remark line 179: update height>
<Side Remark line 187: override delete>
<Side Remark line 224: balance nodes>
<Side Remark line 249: balance nodes>
<Side Remark line 257: inner AVLTreeNode class>
<Side Remark line 259: node height>

```
public class AVLTree<E extends Comparable<E>> extends BinaryTree<E> {  
    /** Create a default AVL tree */  
    public AVLTree() {  
    }  
  
    /** Create an AVL tree from an array of objects */  
    public AVLTree(E[] objects) {  
        super(objects);  
    }  
  
    /** Override createNewNode to create an AVLTreeNode */  
    protected AVLTreeNode<E> createNewNode(E o) {  
        return new AVLTreeNode<E>(o);  
    }  
  
    /** Override the insert method to balance the tree if necessary */  
    public boolean insert(E o) {  
        boolean successful = super.insert(o);
```

```

    if (!successful)
        return false; // o is already in the tree
    else {
        balancePath(o); // Balance from o to the root if necessary
    }

    return true; // o is inserted
}

/** Update the height of a specified node */
private void updateHeight(AVLTreeNode<E> node) {
    if (node.left == null && node.right == null) // node is a leaf
        node.height = 0;
    else if (node.left == null) // node has no left subtree
        node.height = 1 + ((AVLTreeNode<E>)(node.right)).height;
    else if (node.right == null) // node has no right subtree
        node.height = 1 + ((AVLTreeNode<E>)(node.left)).height;
    else
        node.height = 1 +
            Math.max(((AVLTreeNode<E>)(node.right)).height,
                ((AVLTreeNode<E>)(node.left)).height);
}

/** Balance the nodes in the path from the specified
 * node to the root if necessary
 */
private void balancePath(E o) {
    java.util.ArrayList<TreeNode<E>> path = path(o);
    for (int i = path.size() - 1; i >= 0; i--) {
        AVLTreeNode<E> A = (AVLTreeNode<E>)(path.get(i));
        updateHeight(A);
        AVLTreeNode<E> parentOfA = (A == root) ? null :
            (AVLTreeNode<E>)(path.get(i - 1));

        switch (balanceFactor(A)) {
            case -2:
                if (balanceFactor((AVLTreeNode<E>)A.left) <= 0) {
                    balanceLL(A, parentOfA); // Perform LL rotation
                }
                else {
                    balanceLR(A, parentOfA); // Perform LR rotation
                }
                break;
            case +2:
                if (balanceFactor((AVLTreeNode<E>)A.right) >= 0) {
                    balanceRR(A, parentOfA); // Perform RR rotation
                }
                else {
                    balanceRL(A, parentOfA); // Perform RL rotation
                }
            }
        }
    }
}

/** Return the balance factor of the node */
private int balanceFactor(AVLTreeNode<E> node) {
    if (node.right == null) // node has no right subtree
        return -node.height;
    else if (node.left == null) // node has no left subtree
        return +node.height;
    else
        return ((AVLTreeNode<E>)node.right).height -
            ((AVLTreeNode<E>)node.left).height;
}

/** Balance LL (see Figure 9.1) */
private void balanceLL(TreeNode<E> A, TreeNode<E> parentOfA) {
    TreeNode<E> B = A.left; // A is left-heavy and B is left-heavy

    if (A == root) {
        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
    updateHeight((AVLTreeNode<E>)A);
    updateHeight((AVLTreeNode<E>)B);
}

/** Balance LR (see Figure 9.1(c)) */
private void balanceLR(TreeNode<E> A, TreeNode<E> parentOfA) {
    TreeNode<E> B = A.left; // A is left-heavy
    TreeNode<E> C = B.right; // B is right-heavy

    if (A == root) {
        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
    updateHeight((AVLTreeNode<E>)A);
    updateHeight((AVLTreeNode<E>)B);
    updateHeight((AVLTreeNode<E>)C);
}

/** Balance RR (see Figure 9.1(b)) */
private void balanceRR(TreeNode<E> A, TreeNode<E> parentOfA) {
    TreeNode<E> B = A.right; // A is right-heavy and B is right-heavy

    if (A == root) {
        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;
    updateHeight((AVLTreeNode<E>)A);
    updateHeight((AVLTreeNode<E>)B);
}

/** Balance RL (see Figure 9.1(d)) */
private void balanceRL(TreeNode<E> A, TreeNode<E> parentOfA) {
    TreeNode<E> B = A.right; // A is right-heavy
    TreeNode<E> C = B.left; // B is left-heavy

    if (A == root) {
        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
updateHeight((AVLTreeNode<E>)A);
updateHeight((AVLTreeNode<E>)B);
updateHeight((AVLTreeNode<E>)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 */
public boolean delete(E element) {
    if (root == null)
        return false; // Element is not in the tree

    // Locate the node to be deleted and also locate its parent node
    TreeNode<E> parent = null;
    TreeNode<E> current = root;
    while (current != null) {
        if (element.compareTo(current.element) < 0) {
            parent = current;
            current = current.left;
        }
        else if (element.compareTo(current.element) > 0) {
            parent = current;
            current = current.right;
        }
        else
            break; // Element is in the tree pointed by current
    }

    if (current == null)
        return false; // Element is not in the tree

    // Case 1: current has no left children (See Figure 23.6)
    if (current.left == null) {
        // Connect the parent with the right child of the current node
        if (parent == null) {
            root = current.right;
        }
        else {
            if (element.compareTo(parent.element) < 0)
                parent.left = current.right;
            else
                parent.right = current.right;
        }
    }

    // Balance the tree if necessary
    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
    TreeNode<E> parentOfRightMost = current;
    TreeNode<E> rightMost = current.left;

    while (rightMost.right != null) {
        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
    balancePath(parentOfRightMost.element);
}

size--;
return true; // Element inserted
}

/** AVLTreeNode is TreeNode plus height */
protected static class AVLTreeNode<E> extends Comparable<E>>
    extends BinaryTreeNode<E> {
    int height = 0; // New data field

    public AVLTreeNode(E o) {
        super(o);
    }
}

super(o);
}
}

```

<Side Remark: constructors>

The `AVLTree` class extends `BinaryTree`. Like the `BinaryTree` class, the `AVLTree` class has a no-arg constructor that constructs an empty `AVLTree` (lines 3-4) and a constructor that creates an initial `AVLTree` from an array of elements (lines 7-9).

<Side Remark: createNewNode()>

The `createNewNode()` method defined in the `BinaryTree` class creates a `TreeNode`. This method is overridden to return an `AVLTreeNode` (lines 12-14). This is a variation of the Factory Method Pattern.

Design Pattern: Factory Method Pattern

<Side Remark: *Factory Method Pattern*>

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

*****END OF Design Pattern Box**

<Side Remark: insert>

The `insert` method in `AVLTree` is overridden in lines 17-26. The method first invokes the `insert` method in `BinaryTree`, and then invokes `balancePath(o)` (line 22) to ensure that tree is balanced.

<Side Remark: balancePath>

The `balancePath` method first gets the nodes on the path from the node that contains element `o` to the root (line 46). For each node in the path, update its height (line 51), checks its balance factor (line 53), and perform appropriate rotations if necessary (lines 53-69).

<Side Remark: rotations>

Four methods for performing rotations are defined in lines 85-182. 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 9.1-9.4. After the rotation, the height of node A, B, and C is updated for the LL and RR rotations (lines 102, 129, 152, 179).

<Side Remark: delete>

The delete method in AVLTree is overridden in lines 189-274. The method is the same as the one implemented in the BinaryTree class except that you have to rebalance the nodes after deletion in three cases (lines 226, 247, 269).

9.8 Testing the AVLTree Class

Listing 9.4 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 9.4 TestAVLTree.java

*****PD: Please add line numbers in the following code*****
*****Layout: Please layout exactly. Don't skip the space. This is true for all source code in the book. Thanks, AU.**

<Side Remark line 4: create an AVLTree>

<Side Remark line 9: insert 34>

<Side Remark line 10: insert 50>

<Side Remark line 14: insert 30>

<Side Remark line 18: insert 10>

<Side Remark line 22: delete 34>

<Side Remark line 23: delete 30>

<Side Remark line 24: delete 50>

<Side Remark line 28: delete 5>

```
public class TestAVLTree {
    public static void main(String[] args) {
        // Create an AVL tree
        AVLTree<Integer> tree = new AVLTree<Integer>(new Integer[]{25,
            20, 5});
        System.out.print("After inserting 25, 20, 5:");
        printTree(tree);

        tree.insert(34);
        tree.insert(50);
        System.out.print("\nAfter inserting 34, 50:");
        printTree(tree);

        tree.insert(30);
        System.out.print("\nAfter inserting 30");
        printTree(tree);

        tree.insert(10);
        System.out.print("\nAfter inserting 10");
        printTree(tree);
    }
}
```

```

        tree.delete(34);
        tree.delete(30);
        tree.delete(50);
        System.out.print("\nAfter removing 34, 30, 50:");
        printTree(tree);

        tree.delete(5);
        System.out.print("\nAfter removing 5:");
        printTree(tree);
    }

    public static void printTree(BinaryTree tree) {
        // Traverse tree
        System.out.print("\nInorder (sorted): ");
        tree.inorder();
        System.out.print("\nPostorder: ");
        tree.postorder();
        System.out.print("\nPreorder: ");
        tree.preorder();
        System.out.print("\nThe number of nodes is " + tree.getSize());
        System.out.println();
    }
}

```

<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 50 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

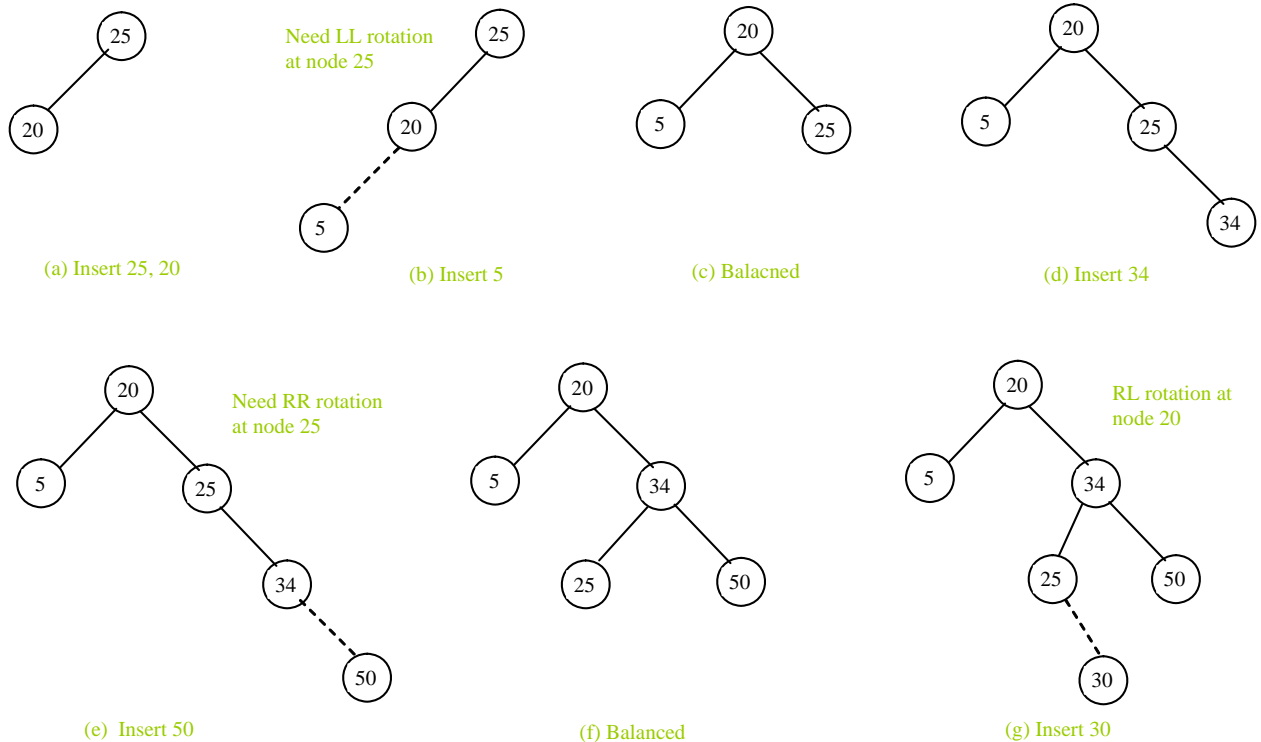
<End Output>

Figure 9.8 shows how the tree evolves as elements are added to the tree. After 25 and 20 are added to the tree, the tree is shown in Figure 9.8(a). 5 is inserted as a left child of 20, as shown in Figure 9.8(b). 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 9.8(c).

After inserting 34, the tree is shown in Figure 9.8(d). After inserting 50, the tree is shown in Figure 9.8(e). The tree is not balanced. It is right-heavy at node 25. Perform an RR rotation to result an AVL tree, as shown in Figure 9.8(f).

After inserting 30, the tree is shown in Figure 9.8(g). The tree is not balanced. Perform an LR rotation to result an AVL tree, as shown in Figure 9.8(h).

After inserting 10, the tree is shown in Figure 9.8(i). The tree is not balanced. Perform an RL rotation to result an AVL tree, as shown in Figure 9.8(j).



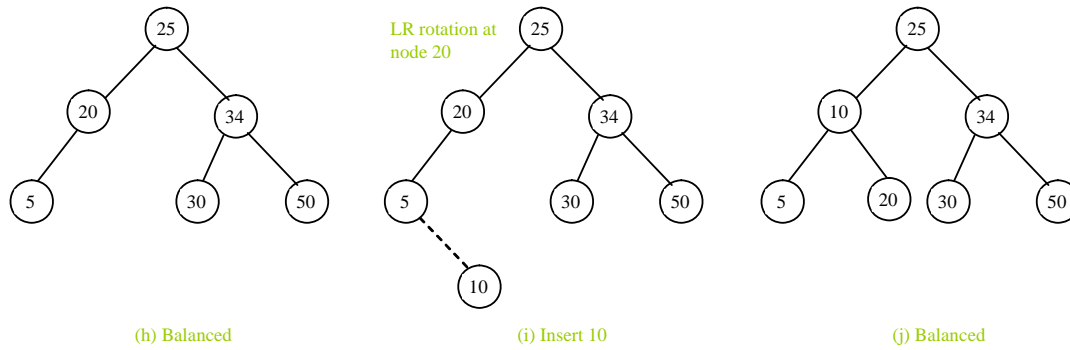


Figure 9.8

The tree evolves as new elements are inserted.

Figure 9.9 shows how the tree evolves as elements are deleted. After deleting 34, 30, and 50, the tree is shown in Figure 9.9(b). The tree is not balanced. Perform an LL rotation to result an AVL tree, as shown in Figure 9.9(c).

After deleting 5, the tree is shown in Figure 9.9(d). The tree is not balanced. Perform an RL rotation to result an AVL tree, as shown in Figure 9.9(e).

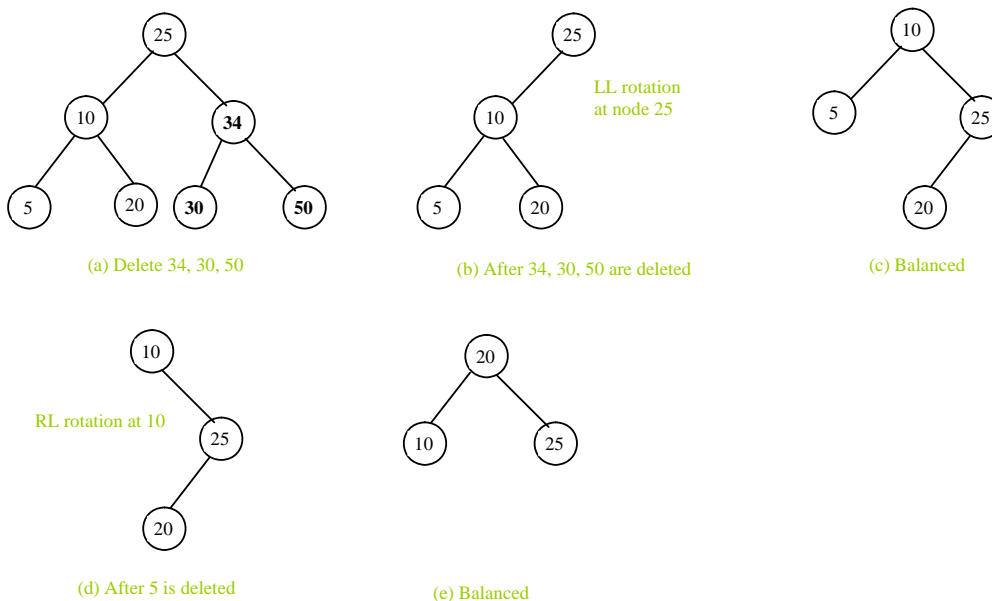


Figure 9.9

The tree evolves as the elements are deleted from the tree.

9.9 Maximum Height of an AVL Tree

<Side Remark: rotation>

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 $O(\log n)$.

Key Terms

*****PD: Please place terms in two columns same as in intro6e.**

- LL rotation
- LR rotation
- RR rotation
- RL rotation
- balance factor
- left-heavy
- right-heavy
- rotation

Chapter Summary

- 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.
- 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 process of rebalancing a node is called a *rotation*. There are four possible rotations: LL rotation, LR rotation, RR rotation, and RL rotation.
- The height of an AVL tree is $O(\log n)$. So, the time complexity for the search, insert, and delete methods are $O(\log n)$.

Review Questions

Sections 9.1-9.2

9.1

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

9.2

Describe LL rotation, RR rotation, LR rotation, and RL rotation.

Sections 9.3-9.8

9.3

Why is the createNewNode method protected?

9.4

When is the updateHeight method invoked? When is the balanceFacotor method invoked? When is the balanacePath method invoked?

9.5

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

9.6

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

9.7

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

9.8

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

Programming Exercises

9.1*

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

9.2

(*Comparing performance*) Write a test program that randomly generates 500000 numbers and inserts into a BinaryTree and then delete all these numbers from the tree. Write another test program that do the same thing for AVLTree. Compare the execution time of these two programs.

9.3***

(*AVL tree animation*) Write a Java applet that animates the AVL tree insert, delete, and search methods, as shown in Figure 9.8.

9.4**

(*Parent reference for BinaryTree*) Suppose that the TreeNode class defined in BinaryTree contains a reference to the node's parent, as shown in Exercise 7.17. 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.