CHAPTER 45

AVL TREES AND SPLAY TREES

Objectives

- To know what an AVL tree is (§45.1).
- To understand how to rebalance a tree using the LL rotation, LR rotation, RR rotation, and RL rotation (§45.2).
- To know how to design the `AVLTree` class (§45.3).
- To insert elements into an AVL tree (§45.4).
- To implement node rebalancing (§45.5).
- To delete elements from an AVL tree (§45.6).
- To implement the `AVLTree` class (§45.7).
- To test the `AVLTree` class (§45.8).
- To analyze the complexity of search, insert, and delete operations in AVL trees (§45.9).
- To know what a splay tree is and how to insert and delete elements in a splay tree (§45.10).

45.1 Introduction

perfectly balanced
well-balanced tree

Chapter 26 introduced binary search trees. The search, insertion, and deletion times 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 tree
$O(\log n)$

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 $O(\log n)$.

balance factor
balanced
left-heavy
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**.

AVL tree animation

Pedagogical Note

Run from www.cs.armstrong.edu/liang/animation/AVLTreeAnimation.html to see how an AVL tree works, as shown in Figure 45.1.

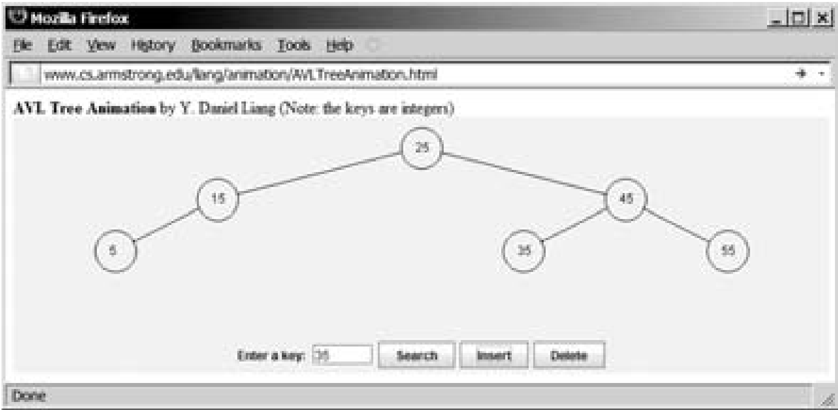


FIGURE 45.1 The animation tool enables you to insert, delete, and search elements visually.

45.2 Rebalancing Trees

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.

LL imbalance
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 45.2(a). This type of imbalance can be fixed by performing a single right rotation at **A**, as shown in Figure 45.2(b).

RR imbalance
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 45.3(a). This type of imbalance can be fixed by performing a single left rotation at **A**, as shown in Figure 45.3(b).

LR imbalance
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 45.4(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 45.4(b).

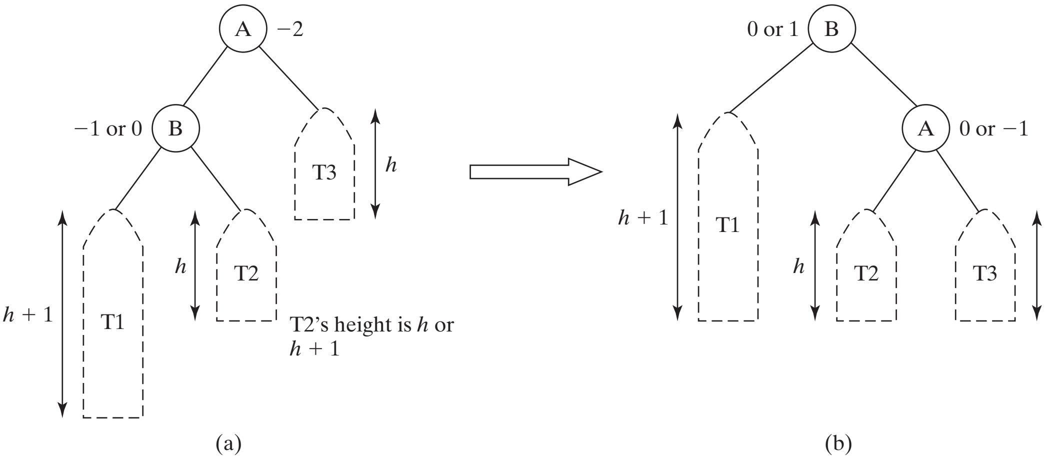


FIGURE 45.2 LL rotation fixes LL imbalance.

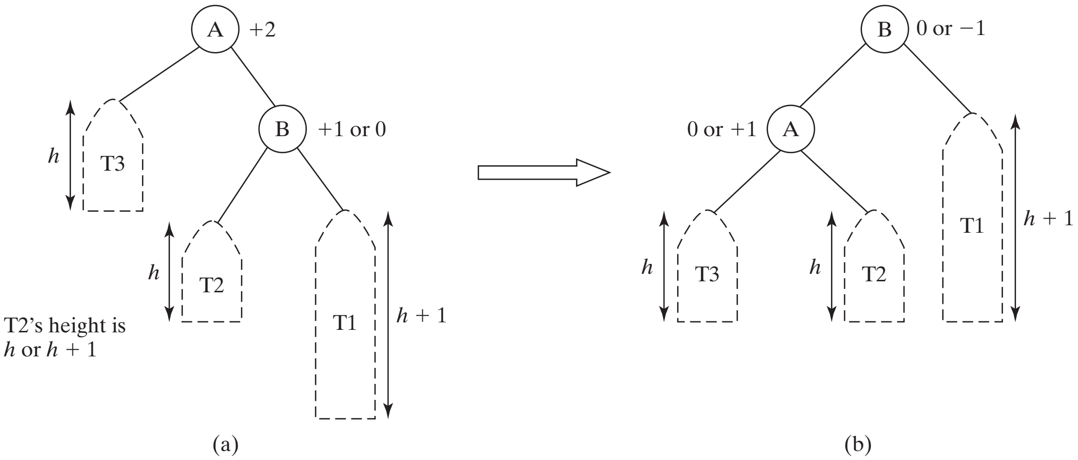


FIGURE 45.3 RR rotation fixes RR imbalance.

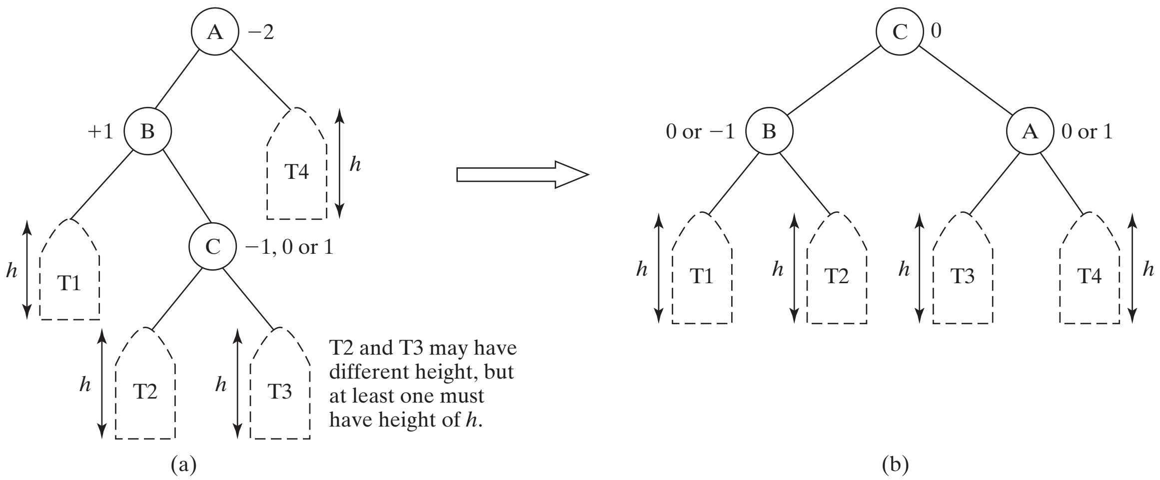


FIGURE 45.4 LR rotation fixes LR imbalance.

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 45.5(a). Assume **B**'s left child is **C**.

RL imbalance

RL rotation

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 45.5(b).

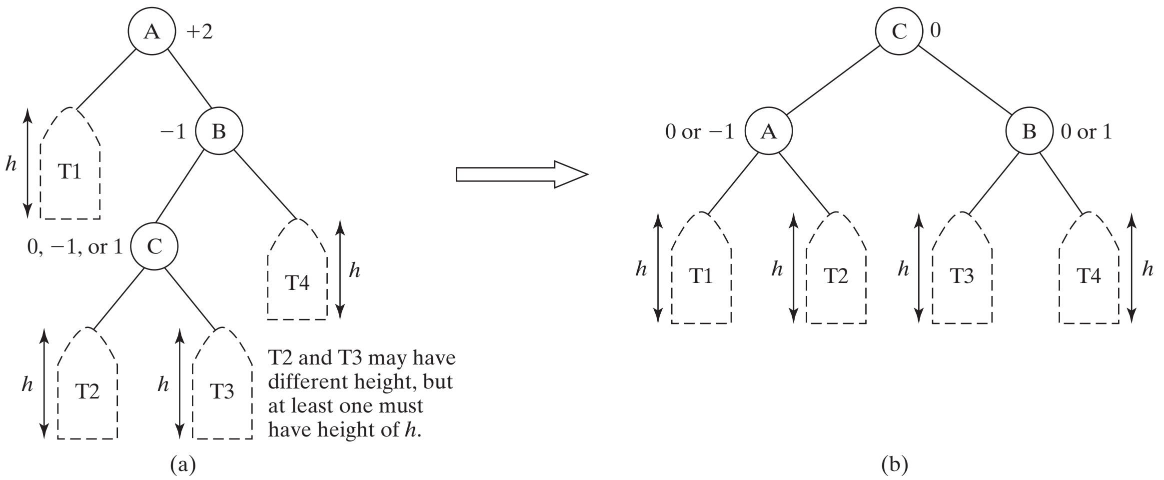


FIGURE 45.5 RL rotation fixes RL imbalance.

45.3 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 45.6. The **BinaryTree** and **TreeNode** classes are defined in §26.2.5.

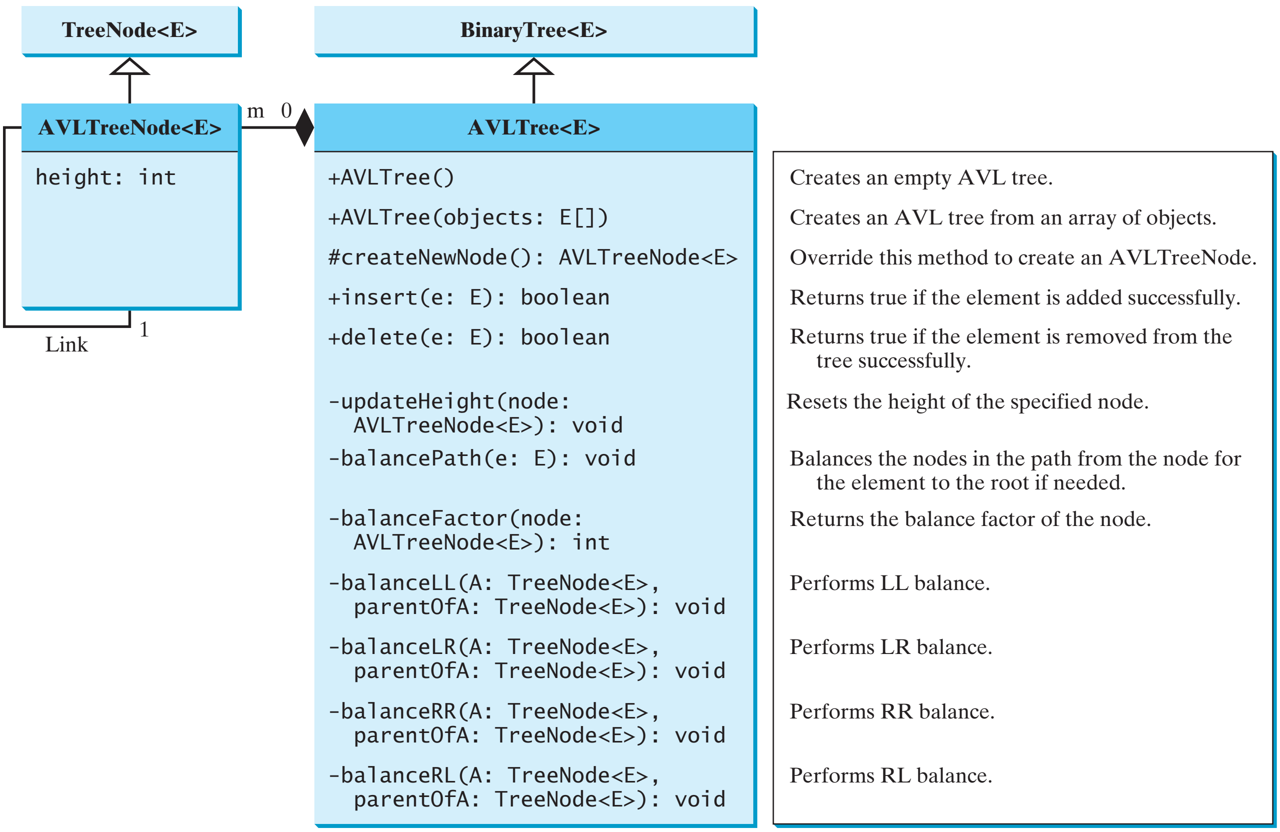


FIGURE 45.6 The **AVLTree** class extends **BinaryTree** with new implementations for the **insert** and **delete** methods.

AVLTreeNode

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 `BinaryTree`. `AVLTreeNode` will be defined as a static inner class in `AVLTree`. `TreeNode` contains the data fields `element`, `left`, and `right`, which are inherited in `AVLTreeNode`. So, `AVLTreeNode` contains four data fields, as pictured in Figure 45.7.

node: AVLTreeNode<E>
#element: E
#height: int
#left: TreeNode<E>
#right: TreeNode<E>

FIGURE 45.7 An `AVLTreeNode` contains protected data fields `element`, `height`, `left`, and `right`.

createNewNode()

In the `BinaryTree` 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 `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.

45.4 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 45.1 Balancing Nodes on a Path

```
balancePath(E e) {
  Get the path from the node that contains element e to the root,       get the path
    as illustrated in Figure 45.8;
  for each node A in the path leading to the root {
    Update the height of A;                                             update node height
    Let parentOfA denote the parent of A,                               get parent node
      which is the next node in the path, or null if A is the root;

    switch (balanceFactor(A)) {                                         is balanced?
      case -2: if balanceFactor(A.left) = -1 or 0
                 Perform LL rotation; // See Figure 45.2                LL rotation
               else
                 Perform LR rotation; // See Figure 45.4                LR rotation
               break;
      case +2: if balanceFactor(A.right) = +1 or 0
                 Perform RR rotation; // See Figure 45.3                RR rotation
               else
                 Perform RL rotation; // See Figure 45.5                RL rotation
```

```
    } // End of switch
  } // End of for
} // End of method
```

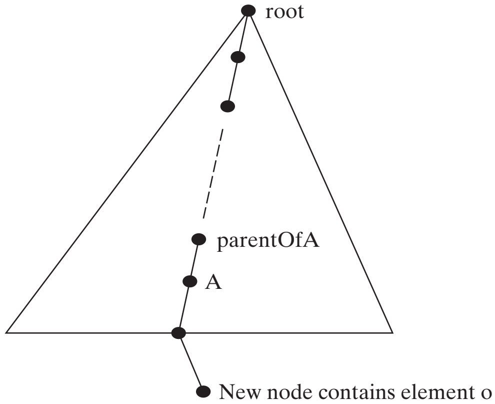


FIGURE 45.8 The nodes along the path from the new leaf node may become unbalanced.

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.

45.5 Implementing Rotations

Section 45.2, "Rebalancing Tree," illustrated how to perform rotations at a node. Listing 45.2 gives the algorithm for the LL rotation, as pictured in Figure 45.2.

LISTING 45.2 LL Rotation Algorithm

left child of A

reconnect B's parent

move subtrees

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
```

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.

45.6 Implementing the `delete` Method

As discussed in §26.3, "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 26.10(a). To delete the current node, simply connect the parent with the right child of the current node, as shown in Figure 26.10(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); // Defined in Listing 45.1
```

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 26.12(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 26.12(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); // Defined in Listing 45.1
```

45.7 The **AVLTree** Class

Listing 45.3 gives the complete source code for the `AVLTree` class.

LISTING 45.3 AVLTree.java

```
public class AVLTree<E extends Comparable<E>> extends BinaryTree<E> {
  /** Create an empty 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
```

no-arg constructor

constructor

create AVL tree node

override insert

balance tree

update node height

balance nodes
get path
consider a node
update height
get height
left-heavy
LL rotation
LR rotation
right-heavy
RR rotation
RL rotation
get balance factor
LL rotation

```
      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 45.2) */
  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);
  }

```

update height

LR rotation

```
  /** Balance LR (see Figure 45.2(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);
  }

```

update height

RR rotation

```
  /** Balance RR (see Figure 45.2(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;
```

update height

```
    updateHeight((AVLTreeNode<E>)A);
    updateHeight((AVLTreeNode<E>)B);
  }

  /** Balance RL (see Figure 45.2(d)) */
```

RL rotation

```
  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
```

update height

```
    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 */
```

override delete

```
  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)
```

```
211     if (current.left == null) {
212       // Connect the parent with the right child of the current node
213       if (parent == null) {
214         root = current.right;
215       }
216       else {
217         if (element.compareTo(parent.element) < 0)
218           parent.left = current.right;
219         else
220           parent.right = current.right;
221
222         // Balance the tree if necessary
223         balancePath(parent.element);                          balance nodes
224       }
225     }
226     else {
227       // Case 2: The current node has a left child
228       // Locate the rightmost node in the left subtree of
229       // the current node and also its parent
230       TreeNode<E> parentOfRightMost = current;
231       TreeNode<E> rightMost = current.left;
232
233       while (rightMost.right != null) {
234         parentOfRightMost = rightMost;
235         rightMost = rightMost.right; // Keep going to the right
236       }
237
238       // Replace the element in current by the element in rightMost
239       current.element = rightMost.element;
240
241       // Eliminate rightmost node
242       if (parentOfRightMost.right == rightMost)
243         parentOfRightMost.right = rightMost.left;
244       else
245         // Special case: parentOfRightMost is current
246         parentOfRightMost.left = rightMost.left;
247
248       // Balance the tree if necessary
249       balancePath(parentOfRightMost.element);                 balance nodes
250     }
251
252     size--;
253     return true; // Element inserted
254   }
255
256   /** AVLTreeNode is TreeNode plus height */
257   protected static class AVLTreeNode<E extends Comparable<E>>   inner AVLTreeNode class
258       extends BinaryTree.TreeNode<E> {
259     int height = 0; // New data field                           node height
260
261     public AVLTreeNode(E o) {
262       super(o);
263     }
264   }
265 }
```

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). constructors

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

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.

insert

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

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 49), check its balance factor (line 53), and perform appropriate rotations if necessary (lines 53–69).

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 45.1–45.4. After the rotation, the heights of nodes **A**, **B**, and **C** are updated for the LL and RR rotations (lines 102, 129, 152, 179).

delete

The **delete** method in **AVLTree** is overridden in lines 187–264. The method is the same as the one implemented in the **BinaryTree** class, except that you have to rebalance the nodes after deletion in two cases (lines 211, 226).

45.8 Testing the AVLTree Class

Listing 45.4 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 45.4 TestAVLTree.java

create an **AVLTree**

insert 34

insert 50

insert 30

insert 10

delete 34

delete 30

delete 50

```
 1 public class TestAVLTree {
 2   public static void main(String[] args) {
 3     // Create an AVL tree
 4     AVLTree<Integer> tree = new AVLTree<Integer>(new Integer[]{25,
 5       20, 5});
 6     System.out.print("After inserting 25, 20, 5:");
 7     printTree(tree);
 8 
 9     tree.insert(34);
10     tree.insert(50);
11     System.out.print("\nAfter inserting 34, 50:");
12     printTree(tree);
13 
14     tree.insert(30);
15     System.out.print("\nAfter inserting 30");
16     printTree(tree);
17 
18     tree.insert(10);
19     System.out.print("\nAfter inserting 10");
20     printTree(tree);
21 
22     tree.delete(34);
23     tree.delete(30);
24     tree.delete(50);
25     System.out.print("\nAfter removing 34, 30, 50:");
26     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();
  }
}
```

delete 5

```
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
```

Figure 45.9 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 45.9(a). **5** is inserted as a left child of **20**, as shown in

Figure 45.9(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 45.9(c).

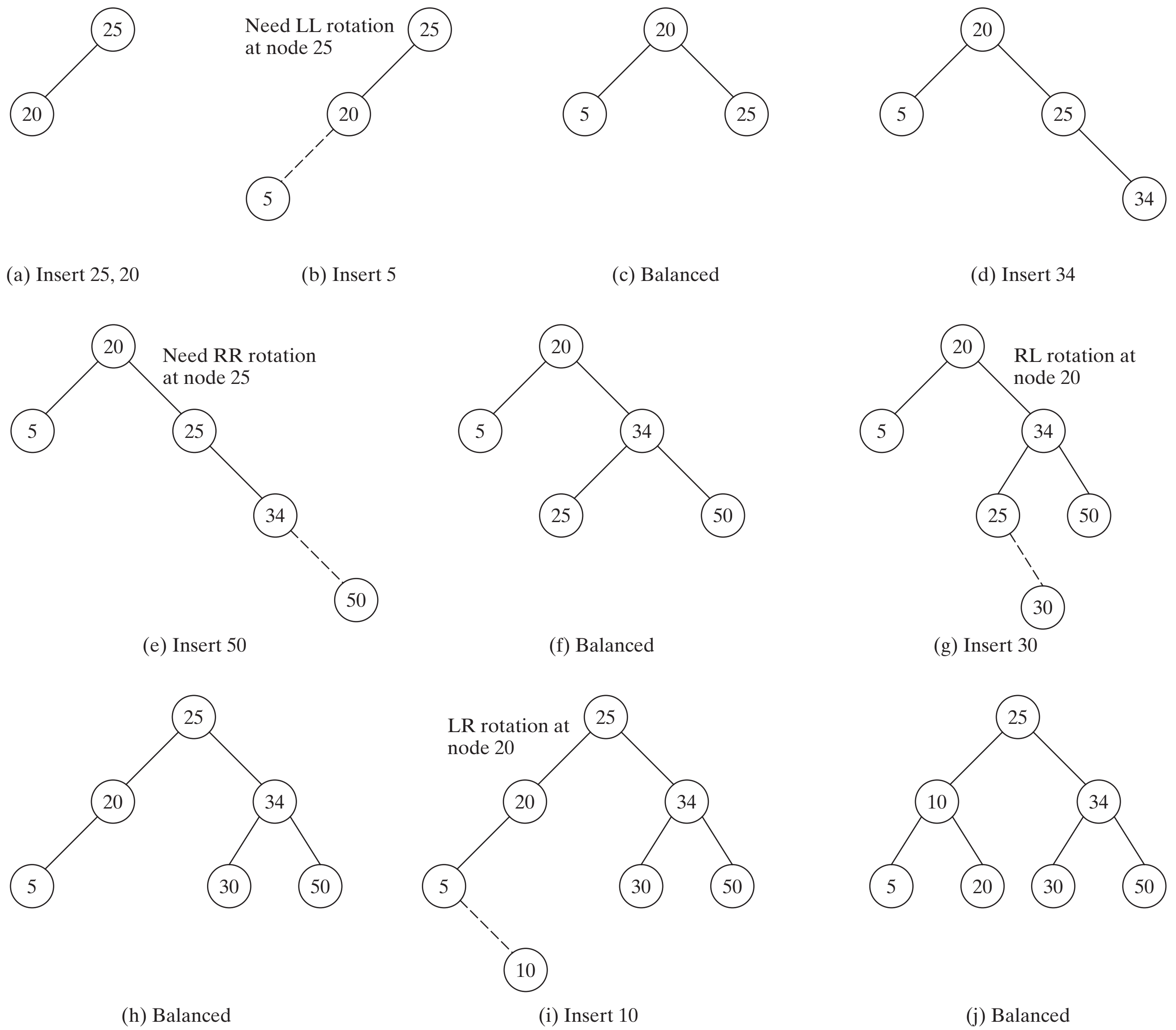


FIGURE 45.9 The tree evolves as new elements are inserted.

After inserting **34**, the tree is shown in Figure 45.9(d). After inserting **50**, the tree is as shown in Figure 45.9(e). 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 45.9(f).

After inserting **30**, the tree is as shown in Figure 45.9(g). The tree is not balanced. Perform an LR rotation to result in an AVL tree, as shown in Figure 45.9(h).

After inserting **10**, the tree is as shown in Figure 45.9(i). The tree is not balanced. Perform an RL rotation to result in an AVL tree, as shown in Figure 45.9(j).

Figure 45.10 shows how the tree evolves as elements are deleted. After deleting **34**, **30**, and **50**, the tree is as shown in Figure 45.10(b). The tree is not balanced. Perform an LL rotation to result an AVL tree, as shown in Figure 45.10(c).

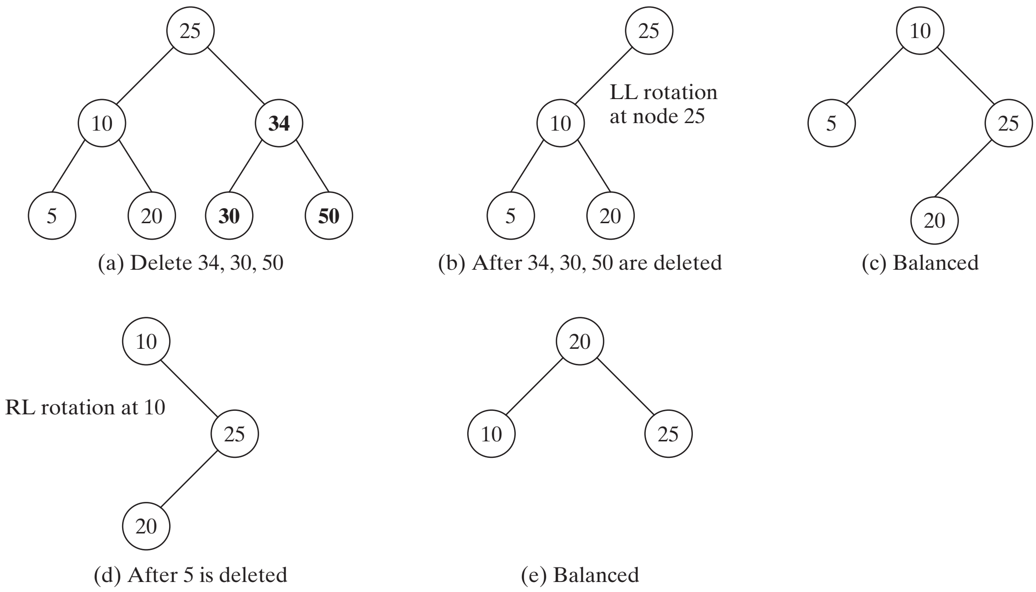


FIGURE 45.10 The tree evolves as the elements are deleted from the tree.

After deleting **5**, the tree is as shown in Figure 45.10(d). The tree is not balanced. Perform an RL rotation to result in an AVL tree, as shown in Figure 45.10(e).

45.9 Maximum Height of an AVL Tree

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 $O(\log n)$. rotation

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 $O(\log n)$.

45.10 Splay Trees

If the elements you access in a BST are near the root, it will take just $O(1)$ time to search for them. Can we design a BST that places the frequently accessed elements near the root? *Splay trees*, invented by Sleator and Tarjan, are a special type of BST for just this purpose. A splay tree is a self-adjusting BST. When an element is accessed, it is moved to the root under the assumption that it will very likely be accessed again in the near future. If this turns out to be the case, subsequent accesses to the element will be very efficient. why splay trees?

splay tree animation

Pedagogical Note

Run from www.cs.armstrong.edu/liang/animation/SplayTreeAnimation.html to see how a splay tree works, as shown in Figure 45.11.

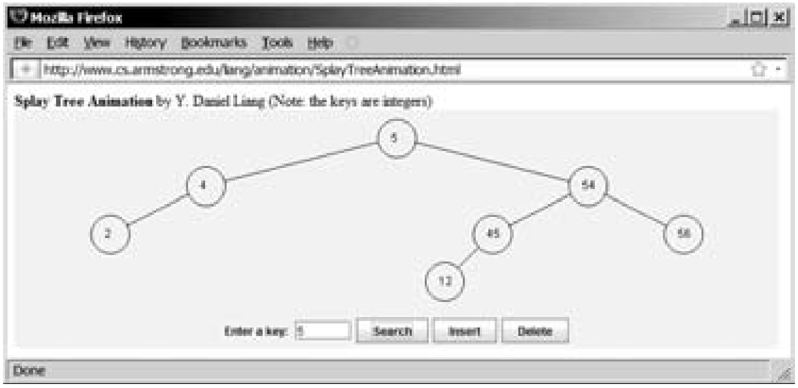


FIGURE 45.11 The animation tool enables you to insert, delete, and search elements visually.

AVL vs. splay

An AVL tree applies the rotation operations to keep it balanced. A splay tree does not enforce the height explicitly. However, it uses the move-to-root operations, called *splaying*, after every access, in order to move the newly-accessed element to the root and keep the tree balanced. An AVL tree guarantees the height to be $O(\log n)$. A splay does not guarantee it. Interestingly, splaying guarantees the average time for search, insertion, and deletion to be $O(\log n)$.

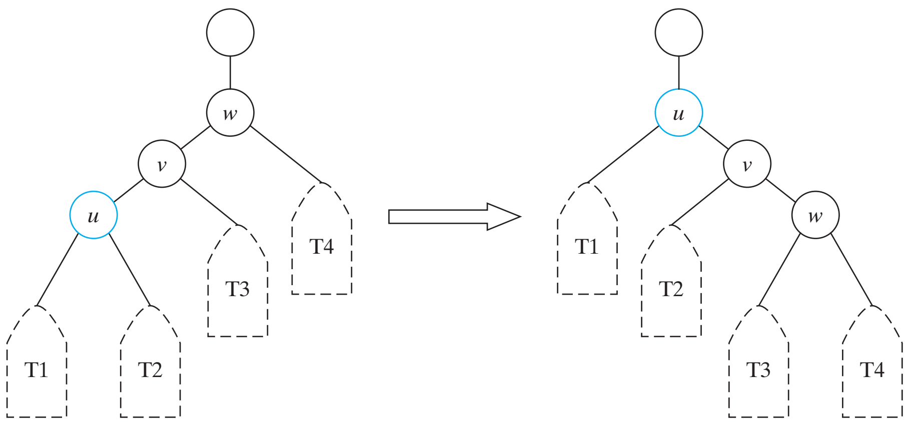
The splaying operation is performed at the last node reached during a search, insertion, or deletion operation. Through a sequence of restructuring operations, the node is moved to the root. The specific rule for determine which node to splay is as follows:

node to splay

`search(element)`: If the element is found in a node u, we splay u. Otherwise, we splay the leaf node where the search terminates unsuccessfully.

`insert(element)`: We splay the newly created node that contains the element.

`delete(element)`: We splay the parent of the node that contains the element. If the node is the root, we splay its left child or right child. If the element is not in the tree, we splay the leaf node where the search terminates unsuccessfully.



(a) (b)

FIGURE 45.12 Left zig-zig restructure.

how to splay

How do you splay a node? Can it be done in an arbitrary fashion? No. To achieve the average $O(\log n)$ time, splaying must be performed in certain ways. The specific operations we perform to move a node u up depends on its relative position to its parent v and its grandparent w. Consider three cases:

zig-zig Case: u and v are both left children or right children, as shown in Figures 45.12(a) and 45.13(a). Restructure u, v, and w to make u the parent of v and v the parent of w, as shown in Figures 45.12(b) and 45.13(b).

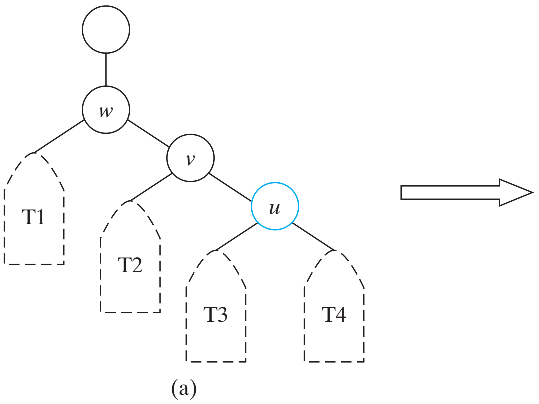


FIGURE 45.13 Right zig-zig restructure.

zig-zag Case: u is the right child of v and v is the left child of w, as shown in Figure 45.14(a), or u is the left child of v and v is the right child of w, as shown in Figure 45.15(a). Restructure u, v, and w to make u the parent of v and w, as shown in Figures 45.14(b) and 45.15(b).

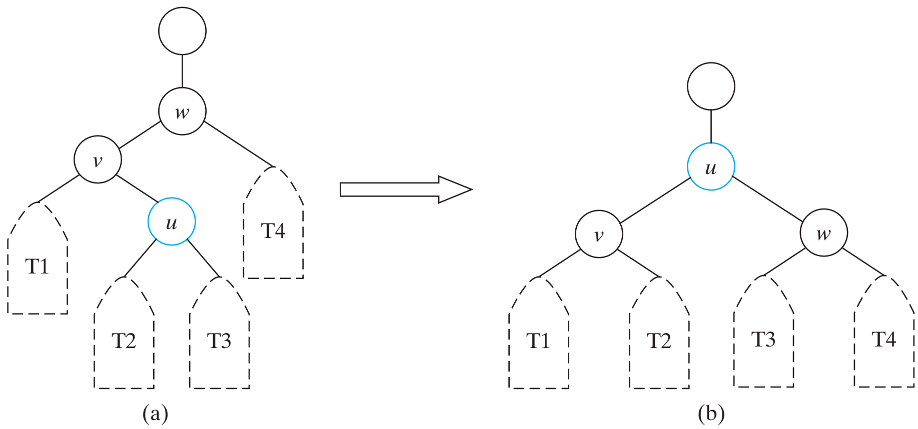


FIGURE 45.14 Left zig-zag restructure.

zig Case: v is the root, as shown in Figures 45.16(a) and 45.17(a). Restructure u and v and make u the root, as shown in Figures 45.16(b) and 45.17(b).

The algorithm for search, insert, and delete in a splay tree is the same as in a regular binary search tree. The difference is that you have to perform the splay operation from the target node to the root. The splay operation consists of a sequence of restructurings. Figure 45.18 shows how the tree evolves as elements **25**, **20**, **5**, and **34** are inserted to the tree.

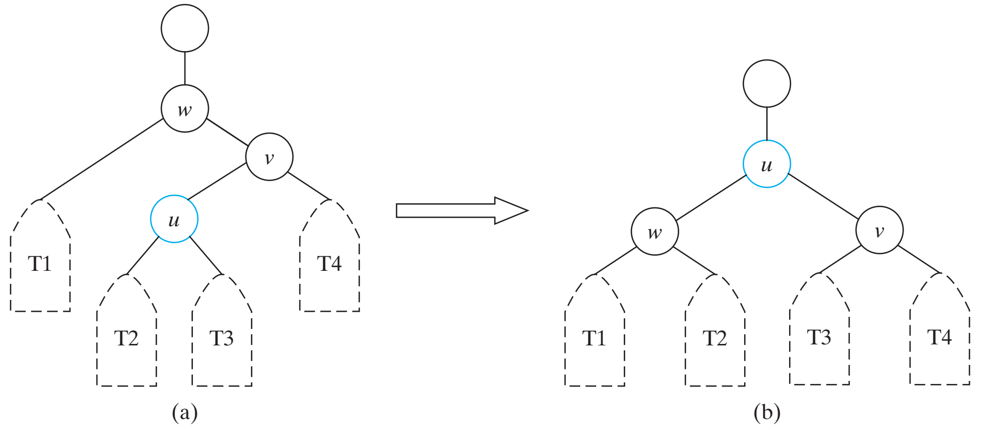


FIGURE 45.15 Right zig-zag restructure.

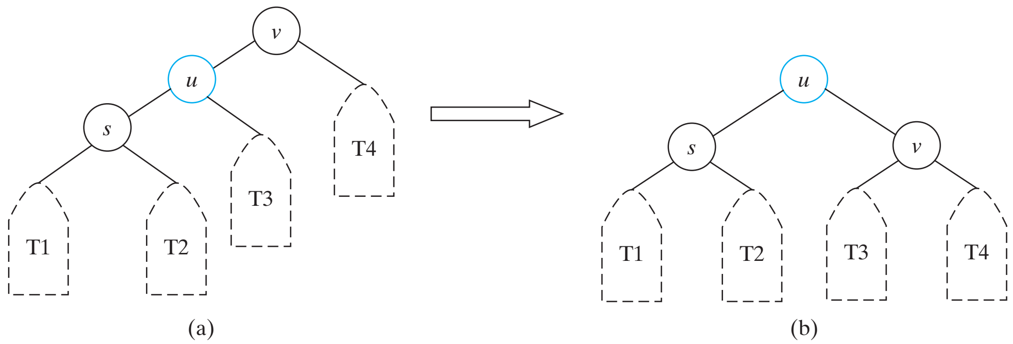


FIGURE 45.16 Left zig restructure.

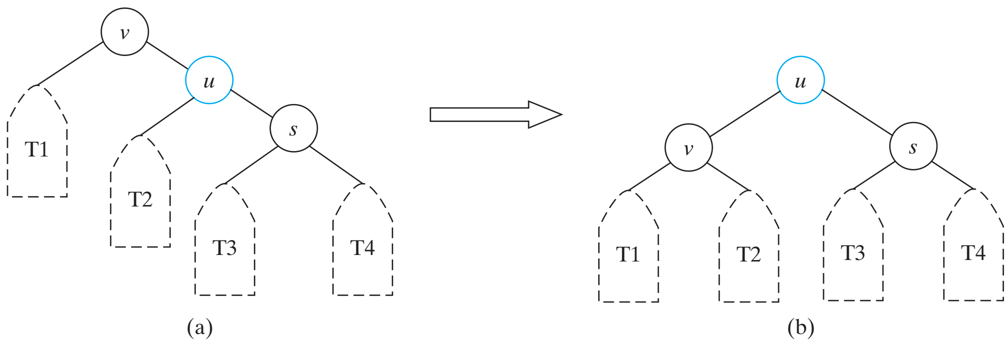


FIGURE 45.17 Right zig restructure.

Suppose you perform a search for element **20** for the tree in Figure 45.19(a). Since **20** is in the tree, splay the node for **20**; the resulting tree is shown in Figure 45.19(c).

Suppose you perform a search for element **21** for the tree in Figure 45.19(c). Since **21** is not in the tree and the last node reached in the search is **25**, splay the node for **25**; the resulting tree is shown in Figure 45.20.

Suppose you delete element **5** from the tree in Figure 45.20(b). Since the node for **20** is the parent node for the node that contains **5**, splay the node for **20**; the resulting tree is shown in Figure 45.21.

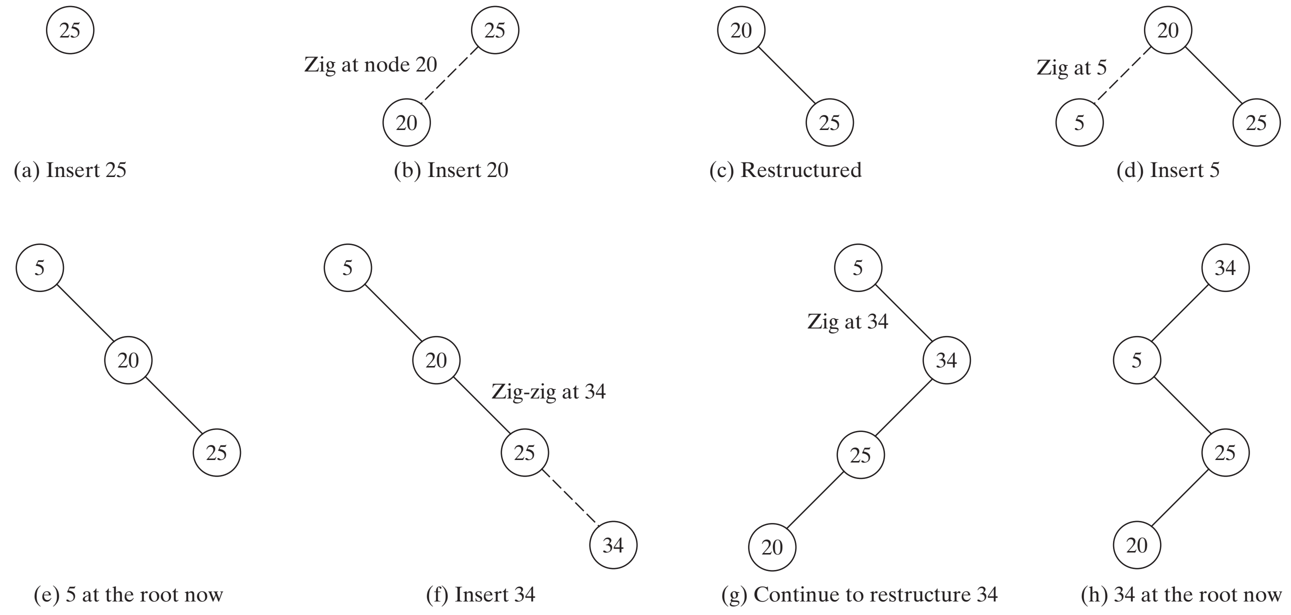


FIGURE 45.18 The tree evolves as new elements are inserted.

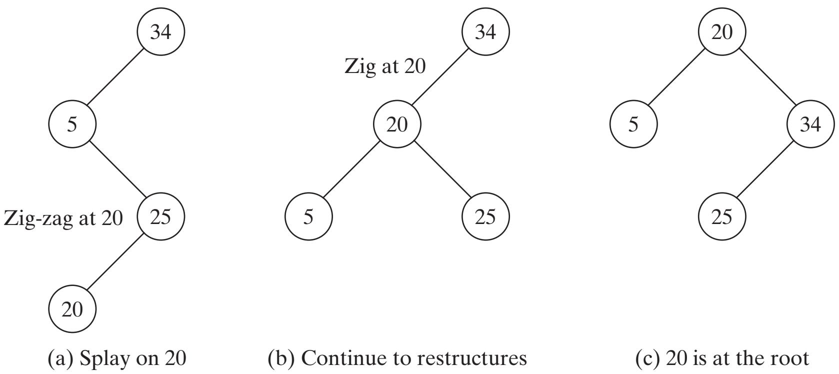


FIGURE 45.19 The tree is adjusted after searching **20**.

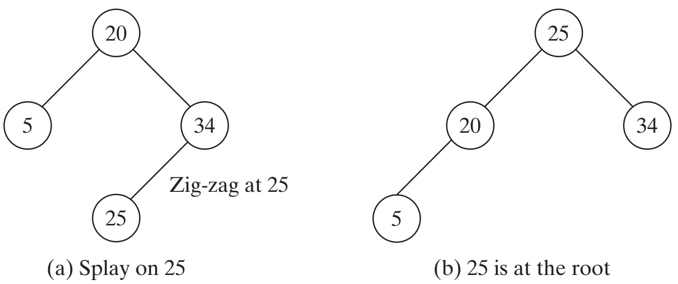
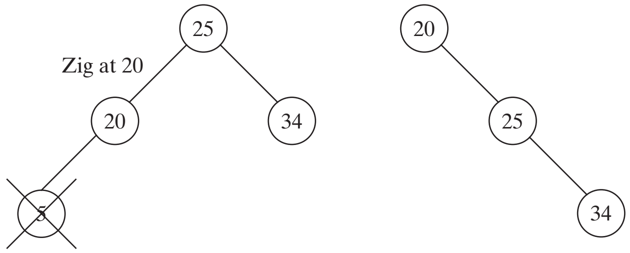


FIGURE 45.20 The tree is adjusted after searching **21**.

When moving a node *u* up, we perform a zig-zig or a zig-zag if *u* has a grandparent, and perform a zig otherwise. After a zig-zig or a zig-zag is performed on *u*, the depth of *u* is decreased by 2, and after a zig is performed, the depth of *u* is decreased by 1. Let *d* denote the depth of *u*. If *d* is odd, a final zig is performed. If *d* is even, no zig is performed. Since a single



(a) Splay on 20

(b) 20 is at the root

FIGURE 45.21 The tree is adjusted after deleting **5**.

average time

zig-zig, zig-zag, or zig operation can be done in constant time, the overall time for a splay operation is $O(d)$. Though the runtime for a single access to a splay tree may be $O(1)$ or $O(n)$, it has been proven that the average time complexity for all accesses is $O(\log n)$. Splay trees are easier to implement than AVL trees. The implementation of splay trees is left as an exercise (see Exercise 45.7).

Key Terms

AVL tree 45–2
LL rotation 45–2
LR rotation 45–2
RR rotation 45–2
RL rotation 45–3
balance factor 45–2
left-heavy 45–2
right-heavy 45–2
rotation 45–12
perfectly balanced 45–2
well-balanced 45–2
splay tree 45–15

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 $O(\log n)$.
6. Splay trees are a special type of BST that provide quick access for frequently accessed elements. The process for inserting or deleting an element in a splay tree is the same as in a regular binary search tree. The difference is that you have to perform a sequence of restructuring operations to move a node up to the root.
7. AVL trees are guaranteed to be well balanced. Splay trees may not be well-balanced, but its average time complexity is $O(\log n)$.

Review Questions

Sections 45.1–45.2

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

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

Sections 45.3–45.8

45.3 Why is the `createNewNode` method protected?

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

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

45.6 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?

45.7 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.

45.8 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.

Section 45.10

45.9 Show the change of a splay tree when `1`, `2`, `3`, `4`, `10`, `9`, `8`, `6` are inserted into the tree, in this order.

45.10 For the tree built in the preceding question, show its change of the tree after attempting to delete `1`, `9`, `7`, `5`, `8`, `6` from the tree in this order.

45.11 Show an example with all nodes in one chain after inserting six elements in a splay tree.

Programming Exercises

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

45.2 (*Comparing performance*) Write a test program that randomly generates 500000 numbers and inserts them into a `BinaryTree`, 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.

45.3*** (*AVL tree animation*) Write a Java applet that animates the AVL tree `insert`, `delete`, and `search` methods, as shown in Figure 45.1.

45.4** (*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.

45.5** (*The* k*th 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 45.22 shows an AVL tree and the `size` value for each node in the tree.

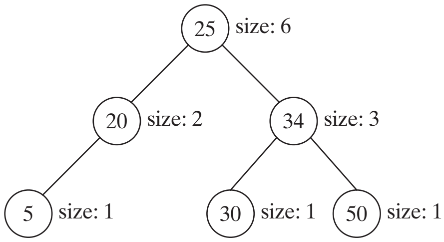


FIGURE 45.22 The `size` data field in `AVLTreeNode` stores the number of nodes in the subtree rooted at the node.

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

```
public E find(int k)
```

The method returns `null` 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 $k \le root.size$, `find(k, root)` can be recursively defined as follows:

$$\text{find}(k, \text{root}) = \begin{cases} \text{root.element, if } A \text{ is null and } k \text{ is 1;} \\ \text{B.element, if } A \text{ is null and } k \text{ is 2;} \\ f(k, A)\text{, if } k <= A\text{.size;} \\ \text{root.element, if } k = A\text{.size} + 1\text{;} \\ f(k - A.size - 1, B)\text{, if } k > A\text{.size} + 1\text{;} \end{cases}$$

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.

45.6** (*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`.

45.7*** (*The `SplayTree` class*) Section 45.10 introduced the splay tree. Implement the `SplayTree` class by extending the `BinaryTree` class and override the `search`, `insert`, and `delete` methods.

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

45.9*** (*Splay tree animation*) Write a Java applet that animates the splay tree `insert`, `delete`, and `search` methods, as shown in Figure 45.11.