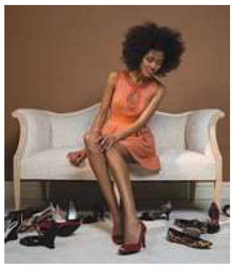
17.2 Binary Trees

A binary tree consists of nodes, each of which has at most two child nodes.

In the following sections, we discuss **binary trees,** trees in which each node has at most two children. As you will see throughout this chapter, binary trees have many very important applications.

In a binary tree, each node has a left and a right child node.

17.2.1 Binary Tree Examples



A decision tree contains questions used to decide among a number of options.

In this section, you will see several typical examples of binary trees. Figure 4 shows a *decision tree* for guessing an animal from one of several choices. Each non-leaf node contains a question. The left subtree corresponds to a "yes" answer, and the right subtree to a "no" answer.

This is a binary tree because every node has either two children (if it is a decision) or no children (if it is a conclusion). Exercises E17.4 and P17.7 show you how you can build decision trees that ask good questions for a particular data set.

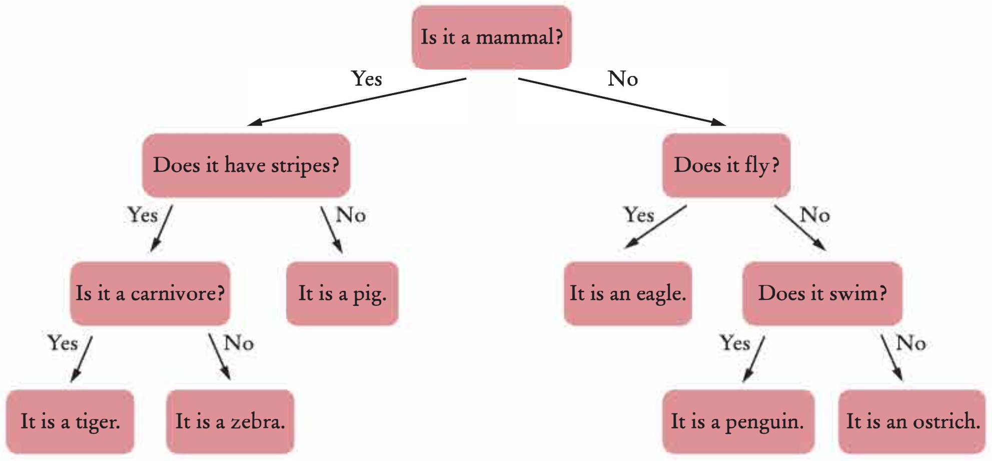


Figure 4 A Decision Tree for an Animal Guessing Game

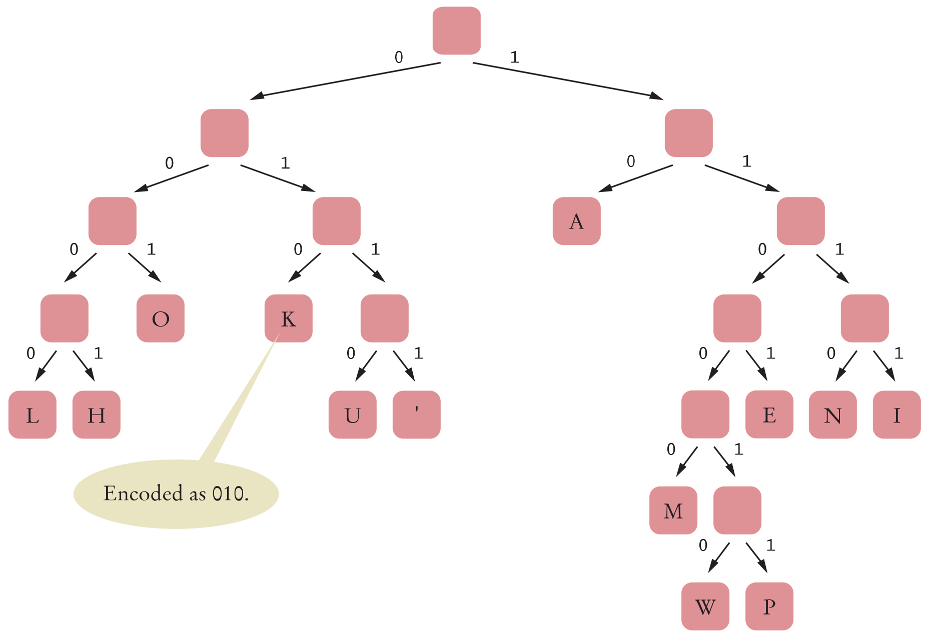


Figure 5 A Huffman Tree for Encoding the Thirteen Characters of Hawaiian Text

In a Huffman tree, the left and right turns on the paths to the leaves describe binary encodings.

Another example of a binary tree is a *Huffman tree*. In a Huffman tree, the leaves contain symbols that we want to encode. To encode a particular symbol, walk along the path from the root to the leaf containing the symbol, and produce a zero for every left turn and a one for every right turn. For example, in the Huffman tree of Figure 5, an H is encoded as 0001 and an A as 10. Worked Example 17.1 shows how to build a Huffman tree that gives the shortest codes for the most frequent symbols.

An expression tree shows the order of evaluation in an arithmetic expression.

Binary trees are also used to show the evaluation order in arithmetic expressions. For example, Figure 6 shows the trees for the expressions

```
(3 + 4) * 5
3 + 4 * 5
```

The leaves of the expression trees contain numbers, and the interior nodes contain the operators. Because each operator has two operands, the tree is binary.

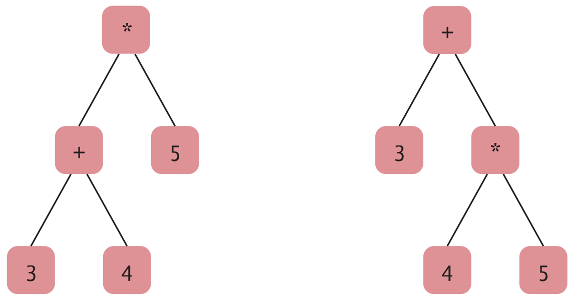


Figure 6 Expression Trees

17.2.2 Balanced Trees

In a balanced tree, all paths from the root to the leaves have approximately the same length.

When we use binary trees to store data, as we will in Section 17.3, we would like to have trees that are *balanced*. In a balanced tree, all paths from the root to one of the leaf nodes have approximately the same length. Figure 7 shows examples of a balanced and an unbalanced tree.

In a balanced binary tree, each subtree has approximately the same number of nodes.

Recall that the height of a tree is the number of nodes in the longest path from the root to a leaf. The trees in Figure 7 have height 5. As you can see, for a given height, a balanced tree can hold more nodes than an unbalanced tree.

We care about the height of a tree because many tree operations proceed along a path from the root to a leaf, and their efficiency is better expressed by the height of the tree than the number of elements in the tree.

A binary tree of height h can have up to $n = 2^h - 1$ nodes. For example, a completely filled binary tree of height 4 has $1 + 2 + 4 + 8 = 15 = 2^4 - 1$ nodes (see Figure 8).

In other words, $h = \log_2(n + 1)$ for a completely filled binary tree. For a balanced tree, we still have $h \approx \log_2 n$. For example, the height of a balanced binary tree with 1,000 nodes is approximately 10 (because $1000 \approx 1024 = 2^{10}$). A balanced binary tree with 1,000,000 nodes has a height of approximately 20 (because $10^6 \approx 2^{20}$). As you will see in Section 17.3, you can find any element in such a tree in about 20 steps. That is a lot faster than traversing the 1,000,000 elements of a list.

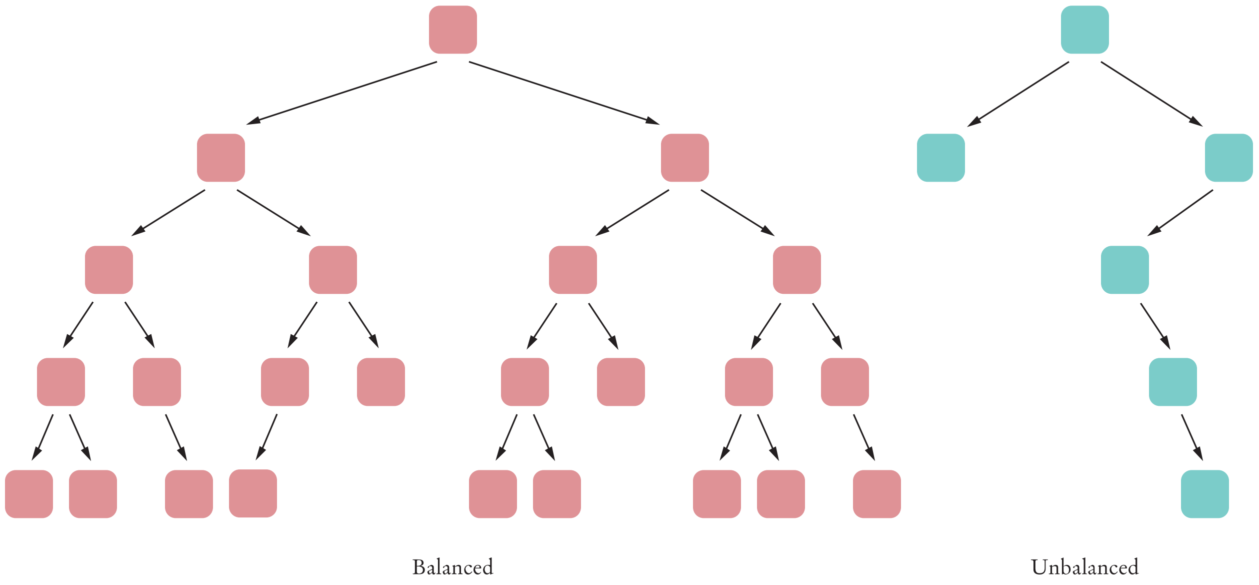


Figure 7 Balanced and Unbalanced Trees

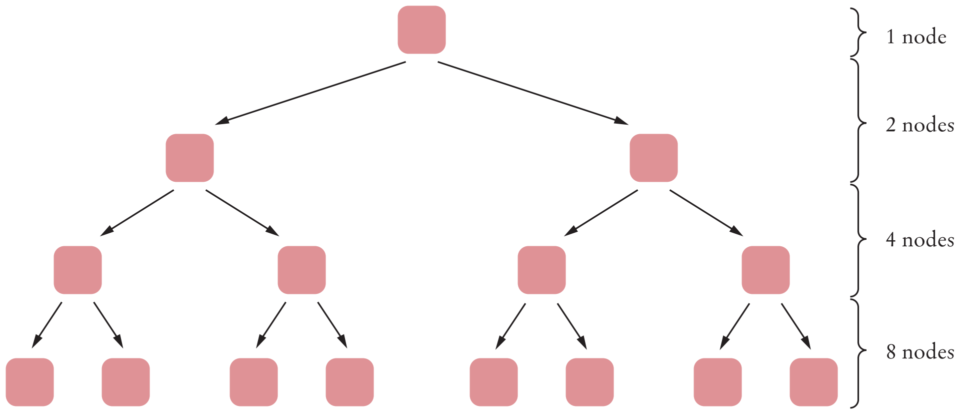


Figure 8 A Completely Filled Binary Tree of Height 4

17.2.3 A Binary Tree Implementation

Every node in a binary tree has references to two children, a left child and a right child. Either one may be `null`. A node in which both children are `null` is a leaf.

A binary tree can be implemented in Java as follows:

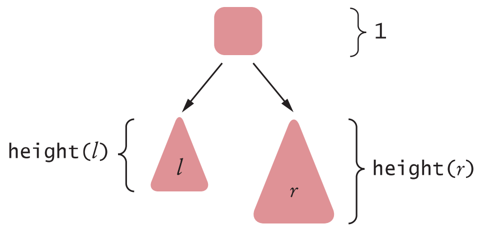
```
public class BinaryTree
{
   private Node root;

   public BinaryTree() { root = null; } // An empty tree

   public BinaryTree(Object rootData, BinaryTree left, BinaryTree right)
   {
      root = new Node();
      root.data = rootData;
      root.left = left.root;
      root.right = right.root;
   }

   class Node
   {
      public Object data;
      public Node left;
      public Node right;
   }

   . . .
}
```

As with general trees, we often use recursion to define operations on binary trees. Consider computing the height of a tree; that is, the number of nodes in the longest path from the root to a leaf.

To get the height of the tree t, take the larger of the heights of the children and add one, to account for the root.

$$\texttt{height}(t) = 1 + \texttt{max}(\texttt{height}(l), \texttt{height}(r))$$

where l and r are the left and right subtrees.



When we implement this method, we could add a height method to the Node class. However, nodes can be null and you can't call a method on a null reference. It is easier to make the recursive helper method a static method of the Tree class, like this:

```
public class BinaryTree
{
   . . .
   private static int height(Node n)
   {
      if (n == null) { return 0; }
      else { return 1 + Math.max(height(n.left), height(n.right)); }
   }
   . . .
}
```

To get the height of the tree, we provide this public method:

```
public class BinaryTree
{
   . . .
   public int height() { return height(root); }
}
```

FULL CODE EXAMPLE
Go to wiley.com/go/javacode to download a program that implements the animal guessing game in Figure 4.

Note that there are two height methods: a public method with no arguments, returning the height of the tree, and a private recursive helper method, returning the height of a subtree with a given node as its root.

SELF CHECK

8. Encode ALOHA, using the Huffman code in Figure 5.
9. In an expression tree, where is the operator stored that gets executed *last*?
10. What is the expression tree for the expression 3 – 4 – 5?
11. How many leaves do the binary trees in Figure 4, Figure 5, and Figure 6 have? How many interior nodes?
12. Show how the recursive height helper method can be implemented as an instance method of the Node class. What is the disadvantage of that approach?

Practice It Now you can try these exercises at the end of the chapter: R17.4, E17.2, E17.3, E17.4.

WORKED EXAMPLE 17.1 **Building a Huffman Tree**

Learn how to build a Huffman tree for compressing the color data of an image. Go to www.wiley.com/go/javaexamples and download Worked Example 17.1.

17.3 Binary Search Trees

A set implementation is allowed to rearrange its elements in any way it chooses so that it can find elements quickly. Suppose a set implementation *sorts* its entries. Then it can use **binary search** to locate elements quickly. Binary search takes $O(\log(n))$ steps, where n is the size of the set. For example, binary search in an array of 1,000 elements is able to locate an element in at most 10 steps by cutting the size of the search interval in half in each step.

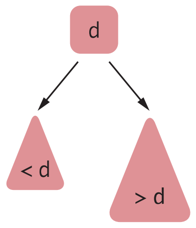
If we use an array to store the elements of a set, inserting or removing an element is an $O(n)$ operation. In the following sections, you will see how tree-shaped data structures can keep elements in sorted order with more efficient insertion and removal.

17.3.1 The Binary Search Property

All nodes in a binary search tree fulfill the property that the descendants to the left have smaller data values than the node data value, and the descendants to the right have larger data values.

A **binary search tree** is a binary tree in which *all nodes* fulfill the following property:

- The data values of *all* descendants to the left are less than the data value stored in the node, and *all* descendants to the right have greater data values.



The tree in Figure 9 is a binary search tree.

We can verify the binary search property for each node in Figure 9. Consider the node “Juliet”. All descendants to the left have data before “Juliet”. All descendants on the right have data after “Juliet”. Move on to “Eve”. There is a single descendant to the left, with data “Adam” before “Eve”, and a single descendant to the right, with data “Harry” after “Eve”. Check the remaining nodes in the same way.

Figure 10 shows a binary tree that is not a binary search tree. Look carefully—the root node passes the test, but its two children do not.

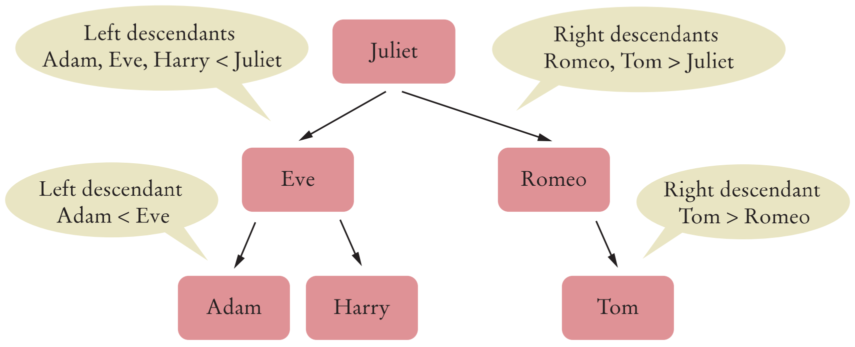


Figure 9 A Binary Search Tree

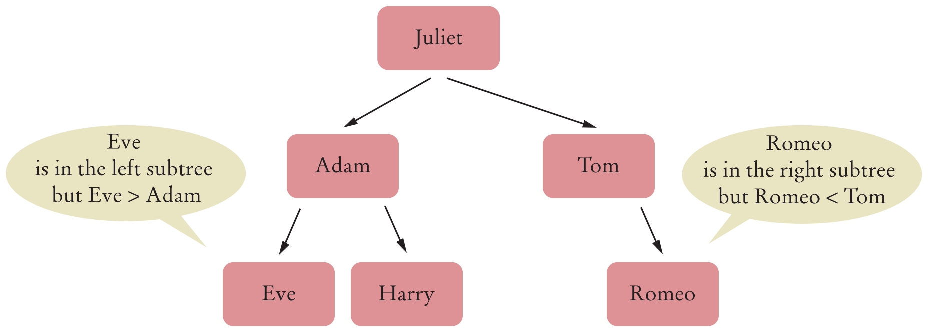

Juliet
Adam
Tom
Eve
Harry
Romeo
Eve is in the left subtree but Eve > Adam
Romeo is in the right subtree but Romeo < Tom

Figure 10 A Binary Tree That Is Not a Binary Search Tree

When you implement binary search tree classes, the `data` variable should have type `Comparable`, not `Object`. After all, you must be able to compare the values in a binary search tree in order to place them into the correct position.

```
public class BinarySearchTree
{
   private Node root;

   public BinarySearchTree() { . . . }
   public void add(Comparable obj) { . . . }
   . . .
   class Node
   {
      public Comparable data;
      public Node left;
      public Node right;

      public void addNode(Node newNode) { . . . }
      . . .
   }
}
```

17.3.2 Insertion

To insert a value into a binary search tree, keep comparing the value with the node data and follow the nodes to the left or right, until reaching a `null` node.

To insert data into the tree, use the following algorithm:

- If you encounter a non-`null` node reference, look at its data value. If the data value of that node is larger than the value you want to insert, continue the process with the left child. If the node's data value is smaller than the one you want to insert, continue the process with the right child. If the node's data value is the same as the one you want to insert, you are done, because a set does not store duplicate values.
- If you encounter a `null` node reference, replace it with the new node.

For example, consider the tree in Figure 11. It is the result of the following statements:

```
BinarySearchTree tree = new BinarySearchTree();
tree.add("Juliet");  1
tree.add("Tom");       2
tree.add("Diana");   3
tree.add("Harry");     4
```

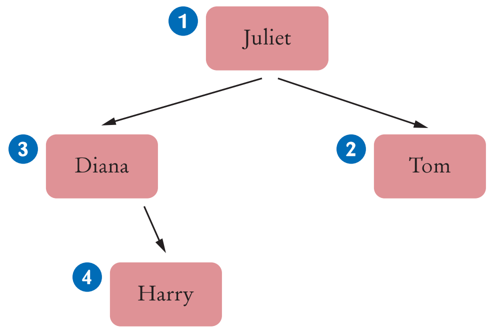


Figure 11 Binary Search Tree After Four Insertions

We want to insert a new element `Romeo` into it:

```
tree.add("Romeo");   5
```

Start with the root node, `Juliet`. `Romeo` comes after `Juliet`, so you move to the right subtree. You encounter the node `Tom`. `Romeo` comes before `Tom`, so you move to the left subtree. But there is no left subtree. Hence, you insert a new `Romeo` node as the left child of `Tom` (see Figure 12).

You should convince yourself that the resulting tree is still a binary search tree. When `Romeo` is inserted, it must end up as a right descendant of `Juliet`—that is what the binary search tree condition means for the root node `Juliet`. The root node doesn't care where in the right subtree the new node ends up. Moving along to `Tom`, the right child of `Juliet`, all it cares about is that the new node `Romeo` ends up somewhere on its left. There is nothing to its left, so `Romeo` becomes the new left child, and the resulting tree is again a binary search tree.

Here is the code for the `add` method of the `BinarySearchTree` class:

```
public void add(Comparable obj)
{
   Node newNode = new Node();
   newNode.data = obj;
   newNode.left = null;
   newNode.right = null;
   if (root == null) { root = newNode; }
   else { root.addNode(newNode); }
}
```

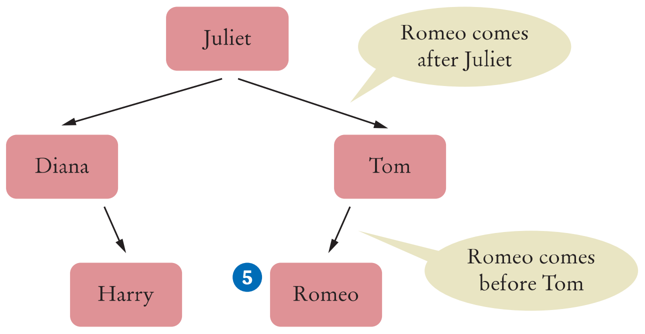


Figure 12 Binary Search Tree After Five Insertions

If the tree is empty, simply set its root to the new node. Otherwise, you know that the new node must be inserted somewhere within the nodes, and you can ask the root node to perform the insertion. That node object calls the `addNode` method of the `Node` class, which checks whether the new object is less than the object stored in the node. If so, the element is inserted in the left subtree; if not, it is inserted in the right subtree:

```
class Node
{
   . . .
   public void addNode(Node newNode)
   {
      int comp = newNode.data.compareTo(data);
      if (comp < 0)
      {
         if (left == null) { left = newNode; }
         else { left.addNode(newNode); }
      }
      else if (comp > 0)
      {
         if (right == null) { right = newNode; }
         else { right.addNode(newNode); }
      }
   }
   . . .
}
```

Let's trace the calls to `addNode` when inserting `Romeo` into the tree in Figure 11. The first call to `addNode` is

```
root.addNode(newNode)
```

Because `root` points to `Juliet`, you compare `Juliet` with `Romeo` and find that you must call

```
root.right.addNode(newNode)
```

The node `root.right` is `Tom`. Compare the data values again (`Tom` vs. `Romeo`) and find that you must now move to the left. Since `root.right.left` is `null`, set `root.right.left` to `newNode`, and the insertion is complete (see Figure 12).

Unlike a linked list or an array, and like a hash table, a binary tree has no *insert positions.* You cannot select the position where you would like to insert an element into a binary search tree. The data structure is *self-organizing;* that is, each element finds its own place.

17.3.3 Removal

We will now discuss the removal algorithm. Our task is to remove a node from the tree. Of course, we must first *find* the node to be removed. That is a simple matter, due to the characteristic property of a binary search tree. Compare the data value to be removed with the data value that is stored in the root node. If it is smaller, keep looking in the left subtree. Otherwise, keep looking in the right subtree.

Let us now assume that we have located the node that needs to be removed. First, let us consider the easiest case. If the node to be removed has no children at all, then the parent link is simply set to `null` (Figure 13).

When the node to be removed has only one child, the situation is still simple (see Figure 14).

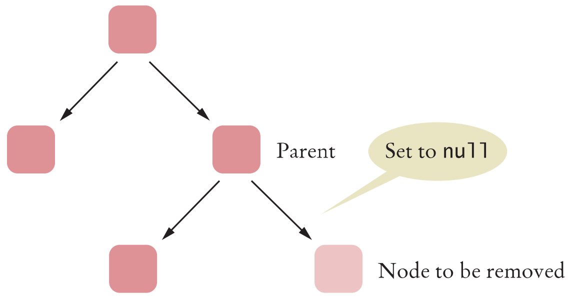


Figure 13 Removing a Node with No Children

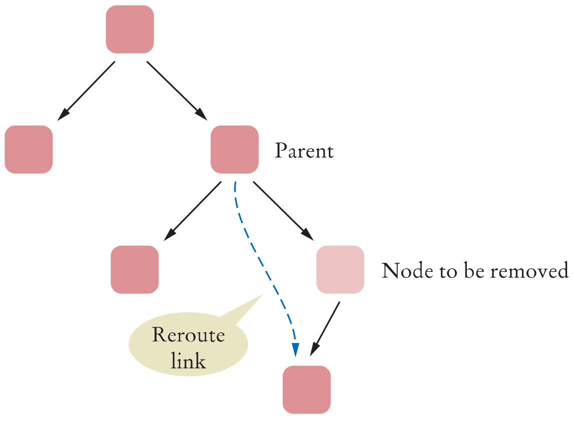


Figure 14 Removing a Node with One Child

When removing a node with only one child from a binary search tree, the child replaces the node to be removed.

When removing a node with two children from a binary search tree, replace it with the smallest node of the right subtree.

To remove the node, simply modify the parent link that points to the node so that it points to the child instead.

The case in which the node to be removed has two children is more challenging. Rather than removing the node, it is easier to replace its data value with the next larger value in the tree. That replacement preserves the binary search tree property. (Alternatively, you could use the largest element of the left subtree—see Exercise P17.5).

To locate the next larger value, go to the right subtree and find its smallest data value. Keep following the left child links. Once you reach a node that has no left child, you have found the node containing the smallest data value of the subtree. Now remove that node—it is easily removed because it has at most one child to the right. Then store its data value in the original node that was slated for removal. Figure 15 shows the details.

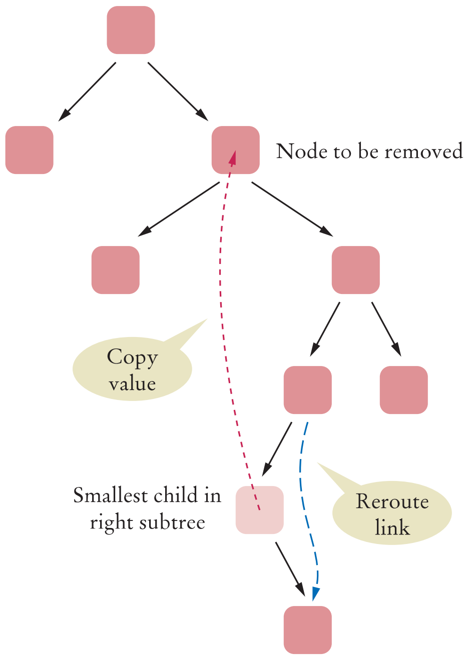


Figure 15 Removing a Node with Two Children

At the end of this section, you will find the source code for the `BinarySearchTree` class. It contains the `add` and `remove` methods that we just described, a `find` method that tests whether a value is present in a binary search tree, and a `print` method that we will analyze in Section 17.4.

17.3.4 Efficiency of the Operations

In a balanced tree, all paths from the root to the leaves have about the same length.

Now that you have seen the implementation of this data structure, you may well wonder whether it is any good. Like nodes in a list, the nodes are allocated one at a time. No existing elements need to be moved when a new element is inserted or removed; that is an advantage. How fast insertion and removal are, however, depends on the shape of the tree. These operations are fast if the tree is balanced.

Because the operations of finding, adding, and removing an element process the nodes along a path from the root to a leaf, their execution time is proportional to the height of the tree, and not to the total number of nodes in the tree.

For a balanced tree, we have $h \approx O(\log(n))$. Therefore, inserting, finding, or removing an element is an $O(\log(n))$ operation. On the other hand, if the tree happens to be *unbalanced*, then binary tree operations can be slow—in the worst case, as slow as insertion into a linked list. Table 2 summarizes these observations.

If a binary search tree is balanced, then adding, locating, or removing an element takes $O(\log(n))$ time.

If elements are added in fairly random order, the resulting tree is likely to be well balanced. However, if the incoming elements happen to be in sorted order already, then the resulting tree is completely unbalanced. Each new element is inserted at the end, and the entire tree must be traversed every time to find that end!

Binary search trees work well for random data, but if you suspect that the data in your application might be sorted or have long runs of sorted data, you should not use a binary search tree. There are more sophisticated tree structures whose methods keep trees balanced at all times. In these tree structures, one can guarantee that finding, adding, and removing elements takes $O(\log(n))$ time. The standard Java library uses *red-black trees*, a special form of balanced binary trees, to implement sets and maps. We discuss these structures in Section 17.5.

Table 2 Efficiency of Binary Search Tree Operations

Operation	Balanced Binary Search Tree	Unbalanced Binary Search Tree
Find an element.	$O(\log(n))$	$O(n)$
Add an element.	$O(\log(n))$	$O(n)$
Remove an element.	$O(\log(n))$	$O(n)$

section_3/BinarySearchTree.java

```
/**
   This class implements a binary search tree whose
   nodes hold objects that implement the Comparable
   interface.
*/
```

```java
public class BinarySearchTree
{
   private Node root;

   /**
      Constructs an empty tree.
   */
   public BinarySearchTree()
   {
      root = null;
   }

   /**
      Inserts a new node into the tree.
      @param obj the object to insert
   */
   public void add(Comparable obj)
   {
      Node newNode = new Node();
      newNode.data = obj;
      newNode.left = null;
      newNode.right = null;
      if (root == null) { root = newNode; }
      else { root.addNode(newNode); }
   }

   /**
      Tries to find an object in the tree.
      @param obj the object to find
      @return true if the object is contained in the tree
   */
   public boolean find(Comparable obj)
   {
      Node current = root;
      while (current != null)
      {
         int d = current.data.compareTo(obj);
         if (d == 0) { return true; }
         else if (d > 0) { current = current.left; }
         else { current = current.right; }
      }
      return false;
   }

   /**
      Tries to remove an object from the tree. Does nothing
      if the object is not contained in the tree.
      @param obj the object to remove
   */
   public void remove(Comparable obj)
   {
      // Find node to be removed

      Node toBeRemoved = root;
      Node parent = null;
      boolean found = false;
      while (!found && toBeRemoved != null)
      {
         int d = toBeRemoved.data.compareTo(obj);
         if (d == 0) { found = true; }
         else
         {
            parent = toBeRemoved;
            if (d > 0) { toBeRemoved = toBeRemoved.left; }
            else { toBeRemoved = toBeRemoved.right; }
         }
      }

      if (!found) { return; }

      // toBeRemoved contains obj

      // If one of the children is empty, use the other

      if (toBeRemoved.left == null || toBeRemoved.right == null)
      {
         Node newChild;
         if (toBeRemoved.left == null)
         {
            newChild = toBeRemoved.right;
         }
         else
         {
            newChild = toBeRemoved.left;
         }

         if (parent == null) // Found in root
         {
            root = newChild;
         }
         else if (parent.left == toBeRemoved)
         {
            parent.left = newChild;
         }
         else
         {
            parent.right = newChild;
         }
         return;
      }

      // Neither subtree is empty

      // Find smallest element of the right subtree

      Node smallestParent = toBeRemoved;
      Node smallest = toBeRemoved.right;
      while (smallest.left != null)
      {
         smallestParent = smallest;
         smallest = smallest.left;
      }

      // smallest contains smallest child in right subtree

      // Move contents, unlink child

      toBeRemoved.data = smallest.data;
      if (smallestParent == toBeRemoved)
      {
            smallestParent.right = smallest.right;
         }
         else
         {
            smallestParent.left = smallest.right;
         }
      }

      /**
         Prints the contents of the tree in sorted order.
      */
      public void print()
      {
         print(root);
         System.out.println();
      }

      /**
         Prints a node and all of its descendants in sorted order.
         @param parent the root of the subtree to print
      */
      private static void print(Node parent)
      {
         if (parent == null) { return; }
         print(parent.left);
         System.out.print(parent.data + " ");
         print(parent.right);
      }

      /**
         A node of a tree stores a data item and references
         to the left and right child nodes.
      */
      class Node
      {
         public Comparable data;
         public Node left;
         public Node right;

         /**
            Inserts a new node as a descendant of this node.
            @param newNode the node to insert
         */
         public void addNode(Node newNode)
         {
            int comp = newNode.data.compareTo(data);
            if (comp < 0)
            {
               if (left == null) { left = newNode; }
               else { left.addNode(newNode); }
            }
            else if (comp > 0)
            {
               if (right == null) { right = newNode; }
               else { right.addNode(newNode); }
            }
         }
      }
   }
```

SELF CHECK

13. What is the difference between a tree, a binary tree, and a balanced binary tree?

14. Are the left and right children of a binary search tree always binary search trees? Why or why not?

15. Draw all binary search trees containing data values A, B, and C.

16. Give an example of a string that, when inserted into the tree of Figure 12, becomes a right child of `Romeo`.

17. Trace the removal of the node "Tom" from the tree in Figure 12.

18. Trace the removal of the node "Juliet" from the tree in Figure 12.

Practice It Now you can try these exercises at the end of the chapter: R17.7, R17.13, R17.15, E17.6.

17.4 Tree Traversal

We often want to visit all elements in a tree. There are many different orderings in which one can visit, or *traverse*, the tree elements. The following sections introduce the most common ones.

17.4.1 Inorder Traversal

Suppose you inserted a number of data values into a binary search tree. What can you do with them? It turns out to be surprisingly simple to print all elements in sorted order. You *know* that all data in the left subtree of any node must come before the root node and before all data in the right subtree. That is, the following algorithm will print the elements in sorted order:

```
Print the left subtree.
Print the root data.
Print the right subtree.
```

To visit all elements in a tree, visit the root and recursively visit the subtrees.

Let's try this out with the tree in Figure 12 on page 773. The algorithm tells us to

1. Print the left subtree of `Juliet`; that is, `Diana` and descendants.
2. Print `Juliet`.
3. Print the right subtree of `Juliet`; that is, `Tom` and descendants.

How do you print the subtree starting at `Diana`?

1. Print the left subtree of `Diana`. There is nothing to print.
2. Print `Diana`.
3. Print the right subtree of `Diana`, that is, `Harry`.

That is, the left subtree of `Juliet` is printed as

```
Diana Harry
```

The right subtree of `Juliet` is the subtree starting at `Tom`. How is it printed? Again, using the same algorithm:

1. Print the left subtree of `Tom`, that is, `Romeo`.
2. Print `Tom`.
3. Print the right subtree of `Tom`. There is nothing to print.

Thus, the right subtree of `Juliet` is printed as

```
Romeo Tom
```

Now put it all together: the left subtree, `Juliet`, and the right subtree:

```
Diana Harry Juliet Romeo Tom
```

The tree is printed in sorted order.

It is very easy to implement this `print` method. We start with a recursive helper method:

```
private static void print(Node parent)
{
   if (parent == null) { return; }
   print(parent.left);
   System.out.print(parent.data + " ");
   print(parent.right);
}
```

To print the entire tree, start this recursive printing process at the root:

```
public void print()
{
   print(root);
}
```

This visitation scheme is called *inorder traversal* (visit the left subtree, the root, the right subtree). There are two related traversal schemes, called *preorder traversal* and *postorder traversal*, which we discuss in the next section.

17.4.2 Preorder and Postorder Traversals

We distinguish between preorder, inorder, and postorder traversal.

In Section 17.4.1, we visited a binary tree in order: first the left subtree, then the root, then the right subtree. By modifying the visitation rules, we obtain other traversals.

In preorder traversal, we visit the root *before* visiting the subtrees, and in postorder traversal, we visit the root *after* the subtrees.

When visiting all nodes of a tree, one needs to choose a traversal order.

```
Preorder(n)
Visit n.
For each child c of n
   Preorder(c).
```

```
Postorder(n)
For each child c of n
   Postorder(c).
Visit n.
```

These two visitation schemes will not print a binary search tree in sorted order. However, they are important in other applications. Here is an example.

In Section 17.2, you saw trees for arithmetic expressions. Their leaves store numbers, and their interior nodes store operators. The expression trees describe the order in which the operators are applied.

Let's apply postorder traversal to the expression trees in Figure 6 on page 767. The first tree yields

```
3 4 + 5 *
```

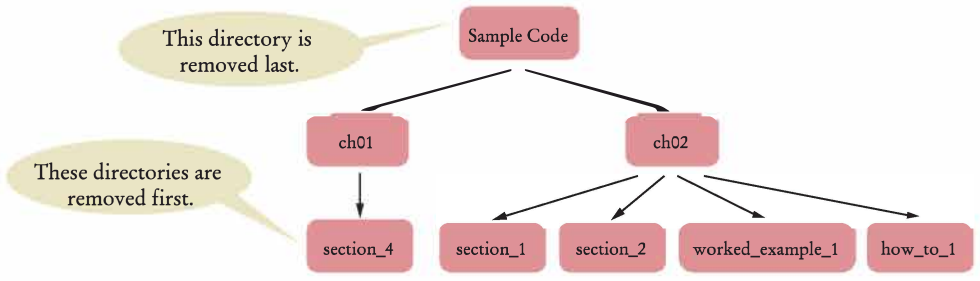
whereas the second tree yields

```
3 4 5 * +
```

Postorder traversal of an expression tree yields the instructions for evaluating the expression on a stack-based calculator.

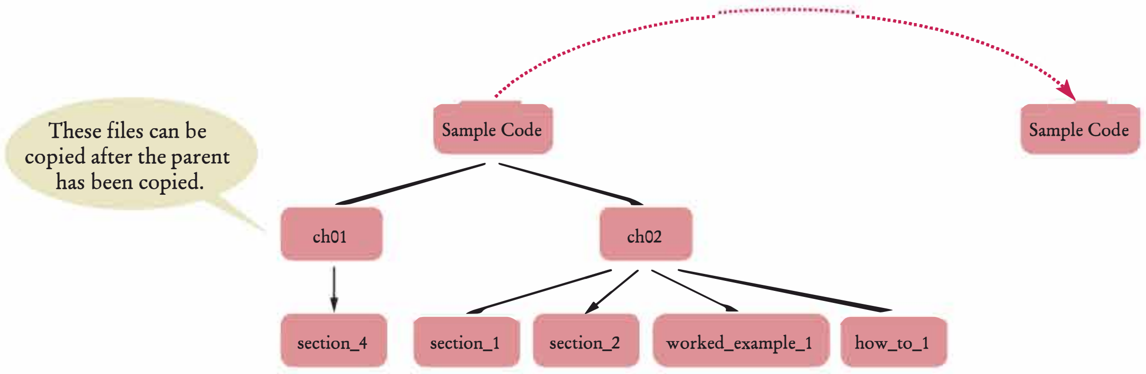
You can interpret the traversal result as an expression in "reverse Polish notation" (see Special Topic 15.2), or equivalently, instructions for a stack-based calculator (see Section 15.6.2).

Here is another example of the importance of traversal order. Consider a directory tree such as the following:



Consider the task of removing all directories from such a tree, with the restriction that you can only remove a directory when it contains no other directories. In this case, you use a postorder traversal.

Conversely, if you want to copy the directory tree, you start copying the root, because you need a target directory into which to place the children. This calls for preorder traversal.



Note that pre- and postorder traversal can be defined for *any* trees, not just binary trees (see the sample code for this section). However, inorder traversal makes sense only for binary trees.