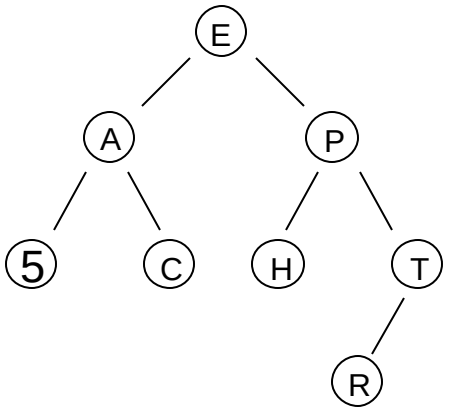


Binary Trees

5.1 Prologue

A ***tree*** is a branching hierarchical structure in which each element except the top one has a link to exactly one element higher in the hierarchy called its ***parent*** (Figure 5-1). The elements of a tree structure are referred to as ***nodes***. The top node in the structure is called the ***root*** of the tree. Any node in the tree may be connected to one or more nodes lower in the hierarchy, called its ***children***. The nodes that have no children are called ***leaves***. There is exactly one path from the root to any node. The intermediate nodes in this path are referred to as the node's ***ancestors*** (i.e., its parent, the parent of its parent, etc.). Trees may not have circular paths.

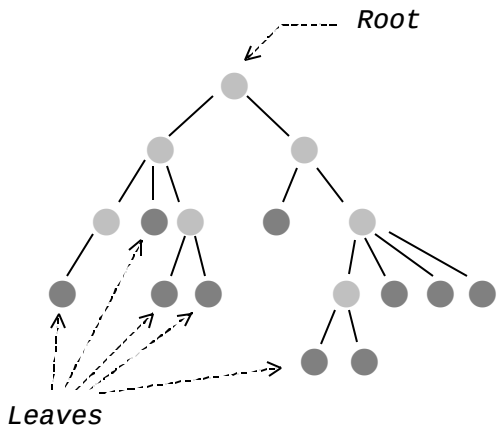


Figure 5-1. A tree structure

As you can see, computer books normally show trees "growing" down, with the root shown on top. This convention probably reflects the fact that we read from the top of the page down and also process trees starting from the root. Trees may be used for representing branching systems or processes, such as organizational charts, game strategies (Figure 5-2), decision charts, class hierarchies in Java, and other hierarchies of objects.

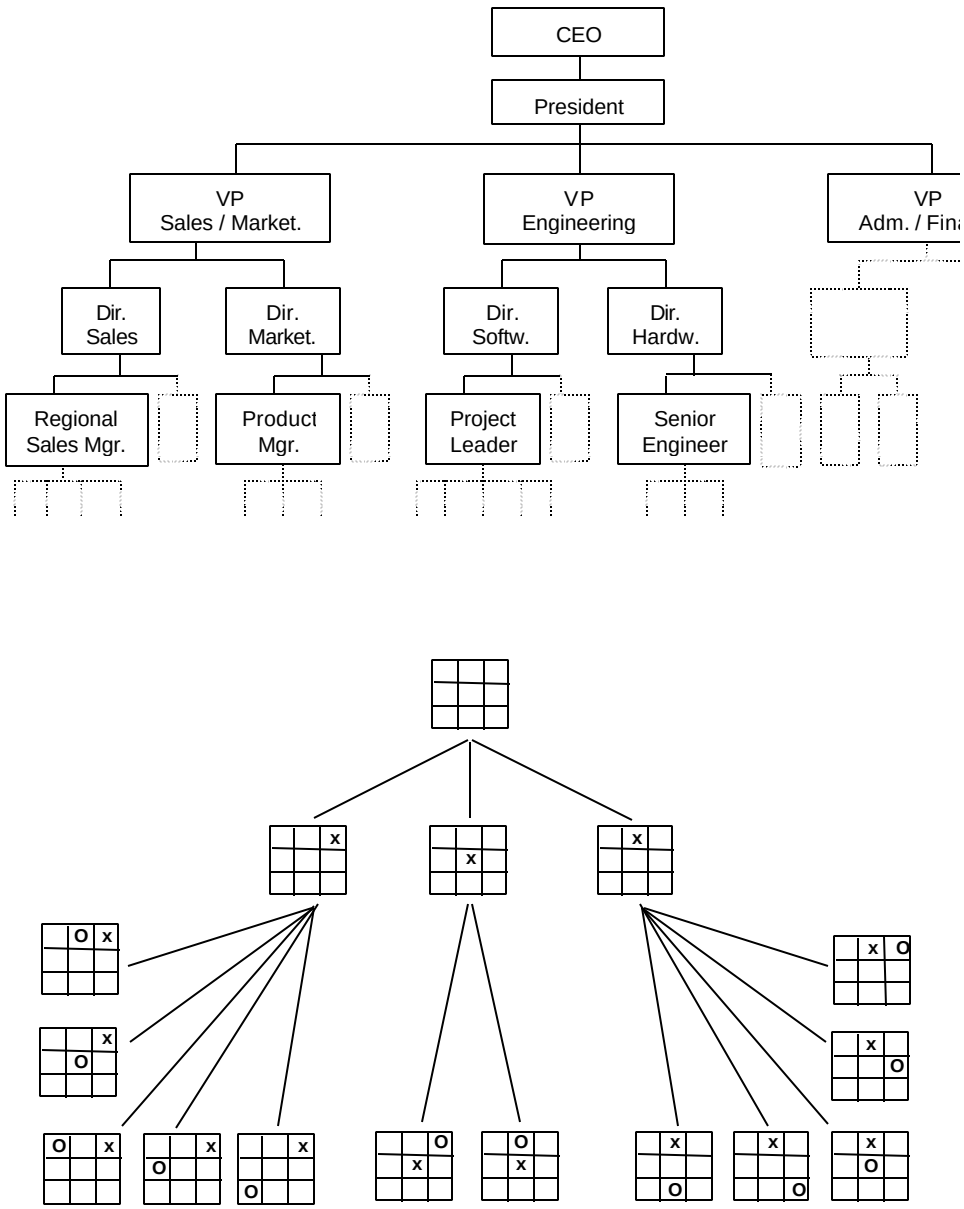


Figure 5-2. Common uses of tree structures

All the nodes in a tree can be arranged in layers with the root at level 0, its children at level 1, their children at level 2, and so on. The level of a node is equal to the length of the path from the root to that node. The total number of levels is called the ***height*** or the ***depth*** of the tree (Figure 5-3).

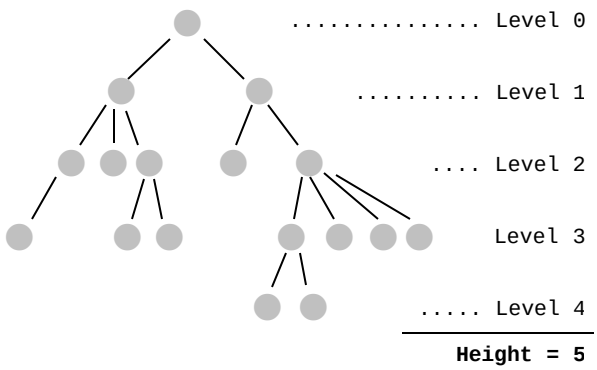


Figure 5-3. Arrangement of tree nodes in levels

One important property of trees is that we can arrange a relatively large number of elements in a relatively ***shallow*** (having a small number of levels) tree. For example, if each node in a tree (except the last level) has two children, a tree with h levels contains $2^h - 1$ nodes (Figure 5-4). Such a tree with 20 levels contains over one million nodes. This property may be utilized for quick searching, data retrieval, decision trees, and similar applications where, instead of going through the whole list and examining all the elements, we can go down the tree and examine just a few. (In strategy games, this property works exactly in reverse and becomes a major stumbling block: if we consider all the possible responses to a given move, then all the responses to those responses, etc., the tree of possible game paths grows so fast that it is not feasible to plan ahead beyond a few moves.)

A list can be viewed as a special case of a tree where the first node is the root, the last node is the only leaf, and all other nodes have exactly one parent and one child. A list has only one node at each level. If a tree degenerates into a near-linear shape with only a few nodes at each level, its advantages for representing a large number of elements in a shallow structure are lost.

A tree is an inherently recursive structure, because each node in a tree can itself be viewed as the root of a smaller tree (Figure 5-5). In computer applications, trees are normally represented in such a way that each node "knows" where to find all its children. In the linked representation, for example, each node, in addition to some

information, holds the list of references to its children. Knowing just the root, we can find all the elements of the tree; and given any node we can find all the nodes in its subtree.

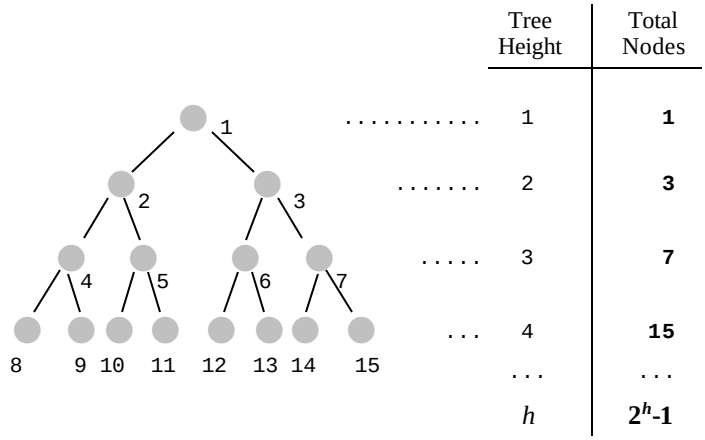


Figure 5-4. A shallow tree can hold many nodes

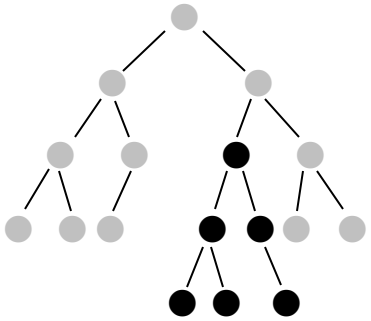


Figure 5-5. Each node in a tree is a root of its own subtree

The recursive branching structure of trees suggests the use of recursive procedures for dealing with them. The following method, for example, allows us to "visit" each node of a tree, a process known as tree ***traversal***:

```
public void traverse (TreeNode root)
{
  // Base case: root == null, the tree is empty -- do nothing
  // Recursive case: tree is not empty
  if (root != null)
  {
    visit(root);
    for (... <each child of the root>)
          traverse (<that child's subtree>);
  }
}
```

This method first "visits" the root of the tree, then, for each child of the root, calls itself recursively to traverse that child's tree. The recursion stops when it reaches a leaf: all its children's trees are empty. Due to the branching nature of the process, an iterative implementation of this method would require your own stack and would be more cumbersome. In this example, therefore, the recursive implementation may actually be slightly more efficient in terms of the processing time, and it does not take too much space on the system stack because the depth of recursion is the same as the depth of the tree, which is normally a relatively small number. The major advantage of a recursive procedure is that it yields clear and concise code.

A tree in which each node has no more than two children is called a ***binary tree***. The children of a node are referred to as the ***left*** child and the ***right*** child. In the following sections we will deal exclusively with binary trees. We will see how a binary tree can be used as a ***binary search tree***. We will examine how a tree can be represented in Java and learn about the Java library classes, `TreeSet` and `TreeMap`, that implement binary search trees. In Chapter 7, we will look at another application of trees: priority queues and heaps.

5.2 Implementation of Binary Trees

The node of a binary tree can be represented as a class, `TreeNode`, which is similar to `ListNode`, a node in a linked list, except that instead of one reference to the next element of the list, a tree node has two references, to the left and right child of that node. In the class `TreeNode` in Figure 5-6, these references are called `left` and `right` and the information held in a node is represented by the object `value`. The class has a constructor that sets these three fields and an accessor and modifier for each of them.

```
public class TreeNode
{
  private Object value;
  private TreeNode left;
  private TreeNode right;

  // Constructor:

  public TreeNode(Object initValue, TreeNode initLeft, TreeNode initRight)
  {
    value = initValue;
    left = initLeft;
    right = initRight;
  }

  // Methods:

  public Object getValue() { return value; }
  public TreeNode getLeft() { return left; }
  public TreeNode getRight() { return right; }
  public void setValue(Object theNewValue) { value = theNewValue; }
  public void setLeft(TreeNode theNewLeft) { left = theNewLeft; }
  public void setRight(TreeNode theNewRight) { right = theNewRight; }
}
```

Figure 5-6. `TreeNode` represents a node in a binary tree[*]

(`Ch05\TreeNode.java`)

[*] Adapted from The College Board's *AP Computer Science AB: Implementation Classes and Interfaces*.

In a linked list each node refers to the next one — a setup suitable for iterations. If, for instance, you need to count the nodes in the list, a `for` loop can do the job:

```
public int countNodes(ListNode head)
{
  int count = 0;
  for (ListNode node = head; node != null; node = node.getNext())
    count++;
  return count;
}
```

It is possible to accomplish the same task recursively, although many people may find it unnecessarily fancy:

```
public int countNodes(ListNode head)
{
  if (head == null)
    return 0;  // base case -- the list is empty
  else
    return 1 + countNodes(head.getNext());
}
```

But for binary trees, recursion is a perfect tool. For example :

```
public int countNodes(TreeNode root)
{
  if (root == null)
    return 0;  // base case -- the tree is empty
  else
    return 1 + countNodes(root.getLeft())
                             + countNodes(root.getRight());
}
```

As we will see in the following section, methods that follow a path in a tree and chose whether to go right or left in each node have simple iterative versions. But recursion is the rule. The base case in recursive methods usually handles an empty tree. Sometimes a method may treat separately another base case, when the tree has only one node, the root. Recursive calls are applied to the left and/or right subtrees.

5.3 Traversals

A method that visits all nodes of a tree and processes or displays their values is called a traversal. It is possible to traverse a tree iteratively, using a stack, but recursion makes it really easy:

```
private void traversePreorder (TreeNode root)
{
  // Base case: root == null, the tree is empty -- do nothing
  if (root != null)
  {
    process(root.getValue());
    traversePreorder(root.getLeft());
    traversePreorder(root.getRight());
  }
}
```

This is called ***preorder traversal*** because the root is visited before the left and right subtrees. In ***postorder traversal*** the root is visited after the subtrees:

```
private void traversePostorder (TreeNode root)
{
  if (root != null)
  {
    traversePostorder(root.getLeft());
    traversePostorder(root.getRight());
    process(root.getValue());
  }
}
```

Finally, in ***inorder traversal***, the root is visited in the middle, between subtrees:

```
private void traverseInorder (TreeNode root)
{
  if (root != null)
  {
    traverseInorder(root.getLeft());
    process(root.getValue());
    traverseInorder(root.getRight());
  }
}
```

If a tree is implemented as a class with the root of the tree hidden in a private field, then the node-visiting method or code becomes restricted to some predefined method or code within the class. If you want to traverse the tree from outside the class, then, just as in the case of linked lists, you need an iterator. It is harder to program an iterator for a tree than for a list: you need to use a stack. Java library classes that implement trees provide iterators for you.

5.4 Binary Search Trees

A ***binary search tree*** (***BST***) is a structure for holding a set of ordered data values in such a way that it is easy to find any specified value and easy to insert and delete values. It overcomes some deficiencies of both sorted arrays and linked lists.

If we have a sorted array of elements, the "divide and conquer" Binary Search algorithm allows us to find any value in the array quickly. We take the middle element of the array, compare it with the target value, and, if they are not equal, continue searching either in the left or the right half of the array, depending on the comparison result. This process takes at most $\log_2 n$ operations for an array of n elements. Unfortunately, inserting values into the array or deleting them from the array is not easy — we may need to shift large blocks of data in memory. The linked list structure, on the other hand, allows us to insert and delete nodes easily, but there is no quick search method because there is no way of getting to the middle of the list easily.

Binary search trees combine the benefits of sorted arrays for quick searching and the benefits of linked lists for inserting and deleting values.

As the name implies, a binary search tree is a kind of a binary tree: each node has no more than two children. The subtree that "grows" from the left child is called the ***left subtree*** and the subtree that "grows" from the right child is called the ***right subtree***. The tree's nodes contain some data values for which a relation of order is defined; that is, for any two values we can say whether the first one is greater, equal, or smaller than the second. The values may be numbers, alphabetized strings, some database record index keys, and so on. Sometimes we informally say that one node is greater or smaller than another, actually meaning that that relationship applies to the data values they contain.

What makes this tree a binary search tree is the following special property: for any node, the value in the node is larger than all the values in this node's left subtree and smaller than all the values in this node's right subtree (Figure 5-7).

A binary search tree is specifically designed to support the "divide and conquer" method. Suppose we need to find a target value. First, we compare the target to the root. If they are equal, the value is found. If the target is smaller, we continue the search in the left subtree. If larger, we go to the right subtree. We will find the target value (or conclude that it is not in the tree) after a number of steps that never exceeds

the number of levels in the tree. If our tree is rather "bushy," with intermediate levels filled to near capacity with nodes, the number of steps required will be close to $\log_2 n$, where n is the total number of nodes.

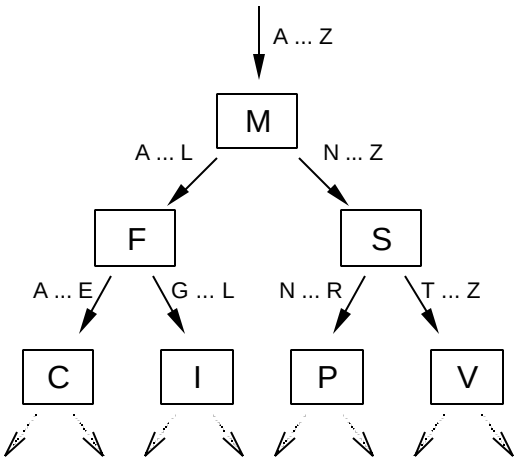


Figure 5-7. The ordering property of a binary search tree

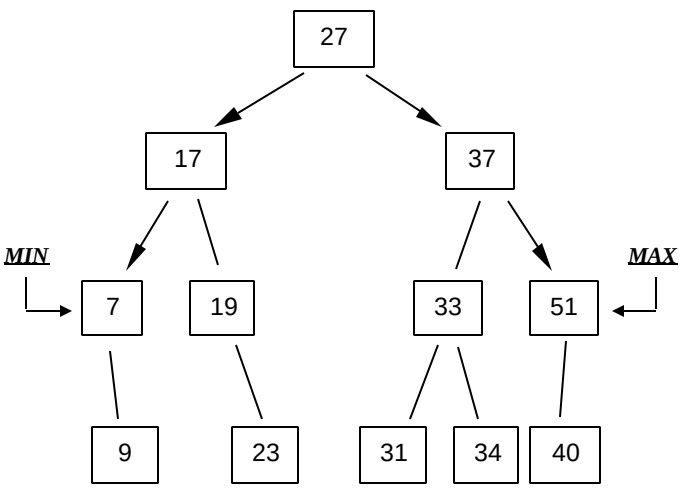


Figure 5-8. Location of the smallest and the largest values in a BST

In a binary search tree, it is also easy to find the smallest and the largest value. Starting at the root, if we always go left for as long as possible, we come to the node containing the smallest value. If we always keep to the right, we come to the node containing the largest value (Figure 5-8). The smallest node, by definition, cannot have a left child, and the largest node cannot have a right child.

The `find` method for a BST can be implemented using either iterations or recursion with equal ease. For example:

```
// Find using iterations:
// ======================

private TreeNode find(TreeNode root, Comparable target)
{
  TreeNode node = root;
  int compareResult;

  while (node != null)
  {
    compareResult = target.compareTo(root.getValue());
    if (compareResult == 0)
      return node;
    else if (compareResult < 0)
      node = node.getLeft();
    else // if (compareResult > 0)
      node = node.getRight();
  }
  return null;
}

// Find using recursion:
// =====================

private TreeNode find(TreeNode root, Comparable target)
{
  if (root == null)           // Base case: the tree is empty
    return null;

  int compareResult = target.compareTo(root.getValue());

  if (compareResult == 0)     // Another base case: found in root
    return root;
  else if (compareResult < 0)
    return find(root.getLeft(), target);  // Search left subtree
  else // if (compareResult > 0)
    return find(root.getRight(), target); // Search right subtree
}
```

5.5 `Comparable` Objects and Comparators

You might have noticed in the previous section that the type of the argument `target` in the `find` method is designated as `Comparable` (pronounced com-pa′-ra-ble). This means that `target` belongs to a class that implements the library interface `Comparable`, which specifies one method, `compareTo`:

```
public int compareTo(Object other);
```

`compareTo` returns 0 if `this` is "equal to" `other`, a negative integer if `this` is "less than" `other`, and a positive integer if `this` is "greater than" `other`. To remember the order of comparison, think of `a.compareTo(b)` as "*a - b*."

`find`'s code calls `target.compareTo(...)` — that's why `target` has to be a `Comparable`, not just an `Object`. Many Java library methods also need to compare objects and expect or assume that they deal with comparable objects. Java library classes such as `String`, `Integer`, `Double`, and `Character`, for which "natural" order makes sense, implement `Comparable`.

You can supply a `compareTo` method for your own class, thus defining a relation of order, a way to compare your class's objects. Then state that your class `implements Comparable`. For example:

```
public class City implements Comparable
{
  ...
  public int compareTo(Object other)
  {
    return getPopulation() - ((City)other).getPopulation();
  }
  ...
}
```

Your class's objects become `Comparable` and you can store them in collections and pass them to methods that expect the `Comparable` data type, such as `find` or `Arrays.sort`.

The `compareTo` argument's type is specified as `Object`, to make the method's prototype independent of a particular class that implements it, but usually it is an object of the same class. `compareTo`'s code casts its argument back into its class's type. For example:

```
return getPopulation() - ((City)other).getPopulation();
```

The order imposed by the `compareTo` method does not have to be totally intuitive. A programmer chooses what to call "smaller" and what "bigger." It may be easier to think of it as "earlier" and "later."

For example, if you swap the operands in the method that compares the populations of two cities —

```
public int compareTo(Object other)
{
  return ((City)other).getPopulation() - getPopulation();
}
```

— then a city with larger population becomes "smaller," that is, stands earlier in the order.

Recall that the `Object` class provides a `boolean` method `equals` that compares this object to another object for equality. Therefore, each object has the `equals` method. When you store comparable objects in a `TreeSet` or use them as keys in a `TreeMap`, it is assumed that the `compareTo` and `equals` methods for these objects agree with each other: that `this.compareTo(other)` returns 0 if and only if `this.equals(other)` returns `true`. However, `TreeSet` and `TreeMap` use only `compareTo` and will work even if `equals` and `compareTo` disagree.

When you define a `compareTo` method for your class, it is a good idea to also define an `equals` method that agrees with `compareTo`.

For example:

```
public class MsgUser implements Comparable
{
  private String screenName;
  ...

  public int compareTo(Object other)
  {
    return screenName.compareTo(((MsgUser)other).screenName);
  }

  public boolean equals(Object other)
  {
    if (other == null)
      return false;
    return compareTo(other) == 0;
  }
}
```

❖ ❖ ❖

Defining `compareTo` gives you one way to compare the objects of your class. For instance, if you define a `compareTo` method for your class `City`, as `this.getPopulation() - other.getPopulation()`, you can sort an array of cities by population in ascending order by calling a library method `Arrays.sort`:

```
City worldCapitals[];
...
// Sort in ascending order by population:
Arrays.sort(worldCapitals);
```

But what if you sometimes need to sort them in ascending order and sometimes in descending order, or alphabetically by name? The `Comparable` abstraction becomes insufficient. Java's response to this dilemma is ***comparator*** objects.

A "comparator" is an object of a class that implements the `Comparator` interface. This interface specifies two methods: `compare` and `equals`. In most cases you don't have to define `equals`, since you can rely on the default method inherited from `Object`.

The `compare` method compares two objects passed to it:

```
public int compare(Object obj1, Object obj2)
```

The result of the comparison is an integer, sort of like `obj1 - obj2`. `obj1` and `obj2` usually belong to the same class, but that class may be different from the comparator's class. For example:

```
public class ComparatorForCityNames
        implements Comparator
{
  public int compare(Object obj1, Object obj2)
  {
    String name1 = ((City)obj1).getName();
    String name2 = ((City)obj2).getName();
    return name1.compareToIgnoreCase(name2);
  }
  ...
}
```

To use this class, create a comparator object and use it to compare cities or pass to constructors or methods that take a `Comparator` argument. For example:

```
ComparatorForCityNames alpha = new ComparatorForCityNames();
...
City worldCapitals[];
...
Arrays.sort(worldCapitals, alpha);
...
int result = alpha.compare(city1, city2);
```

A comparator is an object, not a class, because you may need to pass a comparator to a method and you cannot pass a class to a method.

All comparators from the same class are the same, unless you make the `compare` method dependent on some other properties of the comparator and provide constructors and/or methods for setting these properties. For example:

```
public class ComparatorForCityPopulation
       implements Comparator
{
  private boolean ascending;

  public ComparatorForCityPopulation()
  {
    ascending = true;
  }

  public ComparatorForCityPopulation(boolean asc)
  {
    ascending = asc;
  }

  public void setPopulationOrder(boolean asc)
  {
    ascending = asc;
  }

  public int compare(Object obj1, Object obj2)
  {
    int diff = ((City)obj1).getPopulation() -
                      ((City)obj2).getPopulation();
    if (!ascending)
      diff = -diff;  // flip the result
    return diff;
  }
}
```

The second method of the `Comparator` interface, `equals`, compares comparators! In most cases it is safe to leave it out and to rely on the default `equals` method inherited from the `Object` class. Sometimes you may need to override it to improve performance.

5.6 java.util's TreeSet and TreeMap

Java's util package provides two classes for implementing binary search trees, TreeSet and TreeMap.

TreeSet holds objects; TreeMap holds pairs of objects, a "key" and a "value."

The TreeSet class assumes that the objects stored in the tree have some order or rank relation defined for them and arranges the objects into a BST according to that order. No two values in a BST can have the same rank.

A structure that holds objects, supplies "add," "remove," and "contains" methods, and does not allow duplicate values represents a "Set."

In the java.util package, the set structure is formalized as the java.util.Set interface. Figure 5-9 summarizes a few of its commonly used methods. The add method inserts a given object into the set. If the set already contains an equal object, add ignores the duplicate value and returns false. The remove method removes a given object from the set, and the contains method checks whether a given object is in the set. The iterator method returns an iterator for the set. The order of the values generated by the iterator depends on the class that implements Set.

```
boolean add(Object obj);       // Adds obj to the set
                               //   Returns true if successful,
                               //   false if the object is already
                               //   in the set
boolean remove(Object obj);    // Removes obj from the set
                               //   Returns true if successful,
                               //   false if the object is not found
boolean contains(Object obj); // Returns true if obj is in the set,
                               //   false otherwise
int size();                    // Returns the number of elements
                               //   in the set
Iterator iterator();           // Returns an iterator for the set

Object[] toArray();            // Copies the objects from the set
                               //   into an array and returns
                               //   that array
```

Figure 5-9. Commonly used methods of java.util.Set (and TreeSet)

The `java.util.TreeSet` class implements the `Set` interface, so Figure 5-9 describes `TreeSet`'s methods as well. The `add` method inserts a given object into the tree, and the `remove` method removes a given object from the tree in such a way that the BST ordering property is preserved. For `TreeSet`, the iterator performs inorder traversal of the tree, delivering the values in ascending order.

> **For a BST, inorder traversal (left-root-right) visits the nodes of the tree in ascending order of values.**

(The proof of this fact is left as one of the exercises at the end of this chapter.)

The `TreeSet` class has a no-args constructor that initializes an empty tree and a constructor that takes one argument, a comparator object (see Section 5.5).

Besides BST, there are other ways to implement a set with efficient access to its elements. For example, the `java.util.HashSet` class implements the same interface `Set` but stores elements in a ***hash table***. (Hash tables and Java classes that implement them are the subject of the next chapter.)

`String`, `Character`, `Integer`, and `Double` types of objects are `Comparable`, so you can hold them in a `TreeSet`. In general, any `Comparable` objects may be placed into a `TreeSet` (or you can place any type of objects and supply a comparator). To hold values of primitive data types in a `TreeSet` you need to first convert them into objects using the corresponding wrapper class. For example:

```
TreeSet myTree = new TreeSet();
int m = ...;
...
myTree.add(new Integer(m));
...
if (myTree.contains(new Integer(2004)))
  ...
```

❖ ❖ ❖

A ***map*** establishes a correspondence between the elements of two sets of objects. The objects in one set are thought to be "keys," and the objects in the other set are thought to be "values." Each key has only one value associated with it. For example, a screen name identifies a chat room subscriber; a license plate number identifies a car in the Registry of Motor Vehicles database; a serial number identifies a computer.

"Map" is a data structure that associates keys with their values and efficiently finds a value for a given key. In Java, this functionality is formalized by the `java.util.Map` interface. A few of `Map`'s commonly used methods are shown in Figure 5-10. The `put` method associates a key with a value in the map. If the key was previously associated with a different value, the old association is broken and `put` returns the value previously associated with the key. If the key had no prior association with a value, `put` returns `null`. The `get` method returns the value associated with a given key or `null` if the key is not associated with any value. The `containsKey` method returns `true` if a given key is associated with a value and `false` otherwise.

```
Object put(Object key, Object value);
                                  // Adds key and associated value
                                  //   to the map;
                                  //   Returns the value previously
                                  //   associated with key or null
                                  //   if no value was previously
                                  //   associated with key
Object get(Object key);           // Returns the value associated
                                  //   with key or null if no value
                                  //   is associated with key
boolean containsKey(Object key); // Returns true if key is
                                  //   associated with a value,
                                  //   false otherwise
int size();                       // Returns the number of key-value
                                  //   pairs in the map
Set keySet();                     // Returns the set of keys in
                                  //   the map
```

Figure 5-10. Commonly used methods of `java.util.Map` (and `TreeMap`)

The `java.util.TreeMap` class implements `Map` as a BST ordered by keys. `TreeMap` has a no-args constructor that initializes an empty map. Another constructor takes one argument, a comparator (see Section 5.5). If a map is constructed with the no-args constructor, then the keys must be `Comparable`; if a comparator is supplied to the constructor, then that comparator is used to compare two keys.

The `Map` interface does not specify a method for obtaining an iterator, and the `TreeMap` class does not have one. Instead, you can get the set of all keys by calling the `keySet` method, then iterate over that set. Something like this:

```
TreeMap map = new TreeMap();
SomeType key;
OtherType value;
...
Set keys = map.keySet();
Iterator iter = keys.iterator();
while (iter.hasNext())
{
  key = (SomeType)iter.next();
  value = (OtherType)map.get(key);
  ... // process value
}
```

The values will be processed in the ascending order of keys.

`TreeMap` is more general than `TreeSet`. Both implement BSTs, but in `TreeSet` the values are compared to each other, while in `TreeMap`, no ordering is assumed for the values and the tree is arranged according to the order of the keys. In a way, a set is a special case of a map where a value serves as its own key. In fact, the `TreeSet` class is based on `TreeMap` (or, as Java API puts it, "is backed by an instance of `TreeMap`," i.e., has a `TreeMap` field in it that does all the work). The decision of which class to use, a `TreeSet` or a `TreeMap`, is not always obvious. Sometimes it is possible to include the key in the description of a "value" object and use the `TreeSet` class with an appropriate comparator. But it is often easier to use a `TreeMap`.

5.7 *Lab:* Morse Code

Morse Hall, the Mathematics Department building at Phillips Academy in Andover, Massachusetts, is named after Samuel F. B. Morse, who graduated from the academy in 1805.

In 1838, Samuel Morse devised a signaling code for use with his electromagnetic telegraph. The code used two basic signaling elements: the "dot," a short-duration electric current, and the "dash," a longer-duration signal. The signals lowered an ink pen mounted on a special arm, which left dots and dashes on the strip of paper moving beneath. Morse's code gained wide acceptance and, in its international form, is still in use. (Samuel Morse also achieved distinction as an artist, particularly as a painter of miniatures, and between 1826 and 1845 served as the first president of the National Academy of Design.)

In 1858 Queen Victoria sent the first transatlantic telegram of ninety-eight words to congratulate President James Buchanan of the United States. The telegram started a new era of "instant" messaging — it took only sixteen and a half hours to transmit via the brand new transatlantic telegraph cable.

In this project, we will simulate a telegraph station that encodes messages from text to Morse code and decodes the Morse code back to plain text. The encoding is accomplished simply by looking up a symbol in a `TreeMap` that associates each symbol with its Morse code string. The decoding is implemented with the help of a binary "decoding" tree of our own design. Morse code for each letter represents a path from the root of the tree to some node: a "dot" means go left, and a "dash" means go right. The node at the end of the path contains the symbol corresponding to the code.

The "Telegraph" is implemented in two classes: `Telegraph` and `MorseCode`. In addition, `MorseCode` uses the `TreeNode` class described in Section 5.2. The `Telegraph` class opens two windows on the screen, "London" and "New York," and handles the text entry fields and GUI events in them. We have written this class for you. The `MorseCode` class implements encoding and decoding of text. All the methods in this class are static. The `start` method initializes the encoding map and the decoding tree; the private method `treeInsert` inserts a given symbol into the decoding tree, according to its Morse code string; the public `encode` and `decode` methods convert plain text into Morse code and back, respectively. Your task is to supply all the missing code in the `MorseCode` class.

The `Telegraph` class and the unfinished `MorseCode` class are in the `Ch05\Morse` folder on your student disk. `TreeNode.java` is provided in the `Ch05` folder. Put together a project with `Telegraph`, `MorseCode`, and `TreeNode` and test your program.

5.8 *Case Study and Lab:* Java Messenger

In 1996, AOL introduced its subscribers to the "Buddy List," which allowed AOL members to see when their friends were online. A year later, AOL introduced the *AOL Instant Messenger* (*AIM*). In this case study we implement our own instant "messaging" application, *Java Messenger*. In our application, the same user logs in several times under different screen names, and messages are sent from one window to another on the same screen. Also, in this toy version, all other logged-in users are considered "buddies" of a given user.

Our program can compensate for your friends' absence when your Internet connection is down. But even if you are online, our program has the advantage of always connecting you to a person just as smart and thoughtful as you are. (Another advantage of our program is that it illustrates the use of `java.util`'s `TreeSet` and `TreeMap` classes.)

Our *Java Messenger* application consists of four classes (Figure 5-11). The `Messenger` class initializes the program and implements GUI for logging in and adding new users. The `Server` class defines a server object that keeps track of all the registered users and all currently logged-in users. A server holds screen names and associated registered users in a `TreeMap`. The server maintains a set of all currently logged-in users in a `TreeSet`. A user is represented by an object of the `MsgUser` class. This class implements `Comparable`: its `compareTo` method compares a user to another user by comparing their screen names, case blind. When a new user logs in, the server calls its `openDialog` method, which creates a new dialog window and attaches it to that user. A dialog window is an object of the `MsgWindow` class that handles GUI for the user: a text area for typing in messages and displaying received messages and a "buddy list" for all logged-in "buddies."

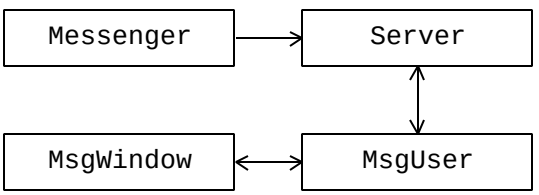


Figure 5-11. ***Java Messenger*** **classes**

As usual, we split the work evenly: I supply the `Messenger` and `MsgWindow` classes that deal with GUI, and you write the `Server` and `MsgUser` classes. Your `Server` class must use a `TreeMap` to associate screen names with passwords and a `TreeSet` to hold `MsgUser` objects for all currently logged-in users.

First, we have to agree on the interfaces: the constructors and public methods. The specs for your two classes are described in Table 5-1 and Table 5-2 below. Your `Server` class must supply a no-args constructor and three methods: `addUser`, `login`, and `logout`. The `addUser` and `login` methods must return correct error codes so that my `Messenger` class can display the appropriate error messages. `login` should pass a correct "buddy set" to the newly created user, which consists of all currently logged-in users excluding the new one. Your `MsgUser` class must provide a `quit` method, which I'll call when the dialog window is closed. That's pretty much all I need from your two classes.

You don't have to worry about my `Messenger` class at all, unless you are curious to see how it works. My `main` method creates a server (a `Server` object) and, to facilitate testing, registers four predefined users:

```
server.addUser("vindog1981", "no");
server.addUser("apscholar5", "no");
server.addUser("javayomama", "no");
server.addUser("lucytexan", "no");
```

You should be aware of my `MsgWindow`'s class constructor that takes two arguments, a user (who creates the dialog) and a set of that user's buddies (displayed in a combo box on the dialog window). My `MsgWindow` class also provides three methods that are useful to you: `addBuddy`, `removeBuddy`, and `showMessage`. Your `MsgUser`'s `addBuddy`, `removeBuddy` and `receiveMessage` methods simply check that your dialog window has been initialized, then pass the parameter to my corresponding method.

`Messenger.java` and `MsgWindow.java` are in the `Ch05\Messenger` 🖫 folder on your student disk.

public class Server	
Constructor:	
`public Server()`	Initializes the map of registered users and the set of logged-in users to empty.
Methods:	
`public int addUser (String name, String password)`	Registers a new user with a given screen name and password. Returns 0 if successful or a negative integer, the error code, if failed. Error codes: `-1` — invalid screen name (must be 4-10 chars) `-2` — invalid password (must be 2-10 chars) `-3` — the screen name is already taken
`public int login (String name, String password)`	Logs in a new user with a given screen name and password. Returns 0 if successful and a negative integer, the error code, if failed. Error codes: `-1` — user not found `-2` — invalid password `-3` — the user is already logged in This method creates a new `MsgUser` object and adds it to the "buddy lists" of all previously logged-in users (by calling their `addBuddy` method). It opens a dialog window for this user by calling its `openDialog` method and passing all previously logged-in users to it as a "buddy list." It then adds the new user to the set of logged-in users.
`public void logout (MsgUser u)`	Removes a given user from the set of logged-in users and from the "buddy lists" of all other logged-in users.

Table 5-1. `Server` constructor and public methods

public class MsgUser implements Comparable	
Constructor:	
`public MsgUser (Server server, String name, String password)`	Saves a reference to the server and initializes this user's and screen name and password fields.
Methods:	
`public String toString()`	Returns this user's screen name.
`public String getPassword()`	Returns this user's password.
`public boolean equals (Object other)`	Returns `true` if this user's name is equal to `other`'s (case blind), `false` otherwise.
`public int compareTo (Object other)`	Compares this user's screen name to `other`'s screen name, case blind.
`public void openDialog (Set buddies)`	Creates a dialog window passing `this` user and the `buddies` set to its constructor. Saves a reference to the new dialog window in the `myWindow` field.
`public void addBuddy (MsgUser u)`	If `myWindow` is initialized, adds `u` to this user's "buddy list" by calling `myWindow.addBuddy(u)`.
`public void removeBuddy (MsgUser u)`	If `myWindow` is initialized, removes `u` from this user's "buddy list" by calling `myWindow.removeBuddy(u)`.
`public void receiveMessage (String text)`	If `myWindow` is initialized, shows `text` by calling `myWindow.showMessage(text)`.
`public void quit()`	Disposes of this user's dialog window. Logs out this user by calling server's `logout` method. (This method is called from the `MsgWindow` class when the "close" button is clicked on the dialog window.)

Table 5-2. `MsgUser` constructor and public methods

5.9 Summary

A tree is a structure of connected nodes where each node, except one special root node, is connected to one parent node and may have one or more child nodes. Each node has a unique ascending path to the root. A tree is an inherently recursive structure because each node in a tree can be considered the root of its own tree, called a subtree. A binary tree is a tree where each node has no more than two children. These are referred to as the left and right children. In the linked representation, each node of a tree contains references to its child nodes. The nodes of a tree contain some data values.

The nodes of a tree are arranged in layers: all nodes at the same level are connected to the root by a path of the same length. The number of levels in a tree is called its height or depth. One important property of trees is that they can hold a large number of values in a relatively shallow structure. A full binary tree with h levels contains $2^h - 1$ values.

A binary search tree is a binary tree whose data elements have some relation of order defined for them and are organized so that the data value in each node is larger than all the values in the node's left subtree and smaller than all the values in its right subtree. The binary search tree combines the benefits of a sorted array for quick binary search with the advantages of a linked list for easy value insertion and deletion.

Due to the recursive structure of trees, it is convenient to use recursive methods when working with them. The base case in such methods is when the tree is empty and, sometimes, when the tree consists only of the root. The method is usually applied recursively to the root's left and/or right subtrees.

Tree traversals can be easily implemented with recursion. Preorder traversal first visits the root of the tree, then processes its left and right subtrees; postorder traversal first processes the left and right subtrees, then visits the root; inorder traversal first processes the left subtree, then visits the root, then processes the right subtree. Inorder traversal of a BST processes its nodes in ascending order.

The `java.util` package includes two classes, `TreeSet` and `TreeMap`, that implement BSTs. The `TreeSet` class implements the `java.util.Set` interface and provides methods for adding and removing an object from the tree and for finding out whether the tree contains a given object. The `TreeMap` class implements the `java.util.Map` interface. This class is used to associate "keys" with "values"

in a BST ordered by keys, and it can be used to quickly retrieve a value for a given key. A few commonly used `Set` (and `TreeSet`) and `Map` (and `TreeMap`) methods are summarized in Figure 5-9 and Figure 5-10, and in Appendix A.

Exercises

Sections 5.1-5.3

1. Define the following tree-related terms:

root
child
leaf
parent
ancestor
depth

2. What is the smallest number of levels required to store 100,000 nodes in a binary tree? ✓

3. What is the smallest and the largest possible number of leaves
(a) in a binary tree with 15 nodes?
(b) in a binary tree containing exactly six non-leaf nodes?

4.■ Prove using mathematical induction that a binary tree of depth h cannot have more than $2^h - 1$ nodes. ✓

5.■ Prove using mathematical induction that in a binary tree with N nodes

$$L \leq \frac{N+1}{2}$$

where L is the number of leaves.

6. Write a method

```
public boolean isLeaf(TreeNode node)
```

that returns `true` if node is a leaf, `false` otherwise. ✓

7. Write a method

```
public int sumTree(TreeNode root)
```

that returns the sum of the values stored in the tree defined by `root`, assuming that the nodes hold `Integer` objects.

8. What does the following method count? ✓

```
public int countSomething(TreeNode root)
{
  if (root == null)
    return 0;
  if (root.getLeft() == null && root.getRight() == null)
    return 1;
  else
    return countSomething(root.getLeft()) +
                  countSomething(root.getRight());
}
```

9. (a) Write a method

```
public int depth(TreeNode root)
```

that returns the depth of the binary tree. ✓

(b) Let us call the "bush ratio" the ratio of the number of nodes in a binary tree to the maximum possible number of nodes in a binary tree of the same depth. Using the `countNodes` method that returns the number of nodes in a tree and the `depth` method from Part (a), write a method

```
public double bushRatio(TreeNode root)
```

that calculates and returns the "bush ratio" for a given binary tree. The method should return 0 if the tree is empty.

10. (a) Write a method

```
public TreeNode copy(TreeNode root)
```

that creates a copy of a given binary tree and returns a reference to its root. Assume that there is enough memory to allocate all the nodes in the new tree. ✓

(b) Write a method

```
public TreeNode mirrorImage(TreeNode root)
```

that creates a mirror image of a given binary tree and returns a reference to its root.

11.■ Write a method

```
public int countPaths(TreeNode root)
```

that returns the total number of paths that lead from the root to any other node of the binary tree.

12.■ (a) Write a method

```
public boolean sameShape(TreeNode root1, TreeNode root2)
```

The method returns `true` if binary trees with the roots `root1` and `root2` have exactly the same shape and `false` otherwise.

(b) Using `sameShape` from Part (a), write a method

```
public boolean hasSameSubtree
       (TreeNode root1, TreeNode root2)
```

`root1` and `root2` refer to the roots of two binary trees. The method returns `true` if the second tree (rooted in `root2`) is empty or has the same shape as the subtree of some node in the first tree, `false` otherwise.

13.◆ (a) Write a method

```
public TreeNode buildFull(int depth)
```

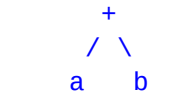
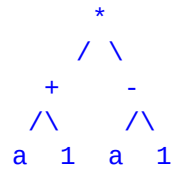
that builds a binary tree of the given depth with all levels completely filled with nodes. The method should set the values in all the nodes to `null` and return a reference to the root of the new tree.

(b) Write and test a method

```
public void fillTree(TreeNode root)
```

that appends new nodes (with `null` values) to the tree until all existing levels in the tree are completely filled.

14. An algebraic expression with parentheses and defined precedence of operators can be represented by a binary tree, called an ***expression tree***. For example:

Expression	Tree
$a + b$	
$(a + 1)(a - 1)$	

In an expression tree, leaves represent operands and other nodes represent operators.

(a) Draw an expression tree for

$$\frac{2}{\frac{1}{x}+\frac{1}{y}}$$ ✓

Continued

(b) Draw an expression tree for

```
yr % 4 == 0 && (yr % 100 != 0 || yr % 400 == 0)
```

(c) Write a class `ExpressionTree` extending `TreeNode`. Assume that the nodes contain `String`s: operands are strings that represent integers and operators are `"+"` or `"*"`. Add a method that determines whether a node contains an operand or an operator and another method that extracts an integer value from the operand string.

(d)■ Write a recursive method

```
public static int eval(ExpressionTree root)
```

that evaluates the expression represented by this tree.

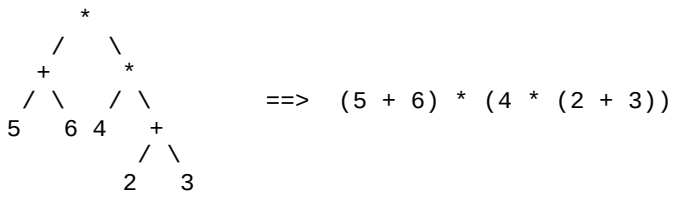
(e) The conventional algebraic notation is called ***infix*** notation. In infix notation, the operator is placed between the operands:

```
Infix:      x + y
```

Write a method

```
public static String toInfixNotation(ExpressionTree root)
```

that generates a fully parenthesized infix expression from a given expression tree. For example:



Continued

(f) There are two other ways to represent expressions which are useful in computer applications. They are called ***prefix*** and ***postfix*** notations. In prefix notation we place the operator before the operands; in postfix notation we place the operator after the operands:

```
Prefix:      + x y
Postfix:     x y +
```

As you can guess, prefix and postfix notations can be generated by traversing the expression tree in preorder and postorder, respectively. Prefix and postfix notations are convenient because they do not use parentheses and do not need to take into account the precedence of the operators. The order of operations can be uniquely reconstructed from the expression itself. Prefix notation is also called ***Polish*** notation after the Polish mathematician Lukasiewicz who invented it, and postfix notation is sometimes called ***reverse Polish notation*** (***RPN***).

Rewrite in postfix notation:

```
(x - 3) / (x*y - 2*x + 3*y)
```

(g) Prove that the operands appear in the same order in the infix, postfix, and prefix notations — only the position of the operators is different. (This is a good test for converting one notation into another manually.) ✓

(h) Write and test a method

```
public static int eval(String tokens[])
```

that evaluates a postfix expression. The `tokens` array contains operands — string representation of integers — and operators (e.g., `"+"` or `"*"`). ≤ Hint: go from left to right, save "unused" operands on a stack. ≥

(i) Write a method

```
public static ExpressionTree toExpressionTree(String tokens[])
```

that converts a postfix expression, represented by the `tokens` array, into an expression tree and returns the root of that tree.

Sections 5.4-5.9

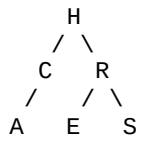
15. Mark true or false and explain:

(a) The smallest node in a binary search tree is always a leaf. _____

(b) If a binary search tree holds integers (with the usual order of comparison) and the root holds 0, then all the nodes of the left subtree hold negative numbers. _____ ✓

(c) The number of comparisons necessary to find a target node in a binary search tree never exceeds $\log_2 n + 1$, where n is the number of nodes. _____ ✓

16. (a) Swap two nodes in the following binary tree to obtain a binary search tree.



(b) What will be the sequence of nodes when the resulting binary search tree is traversed inorder? Preorder?

17. Suppose we start with an empty binary search tree and add nodes with values

```
475, 474, 749, 623, 292, 557, 681
```

(in that order). Draw the resulting tree. How can we arrange the same numbers in a balanced binary search tree (with three levels)? ✓

18. Draw the binary search tree created by inserting the letters

```
L O G A R I T H M
```

(in that order) into an empty tree. List the nodes of this tree when it is traversed inorder, preorder, and postorder.

19.■ Using mathematical induction, prove that inorder traversal of a BST visits the nodes in ascending order.

20.■ Write a non-recursive method

```
public TreeNode maxNode(TreeNode root)
```

that finds and returns the largest node in a BST. ✓

21.◆ Write a method

```
public TreeNode remove(TreeNode root)
```

that removes the root from a BST, repairs the tree and returns the new root. ⋜ Hint: find the smallest node `xMin` in the right subtree; unlink `xMin` from the tree, promoting its right child into its place; place `xMin` into the root. (The same works with the largest node in the left subtree.) ⋝

22.◆ (a) Write a method

```
public TreeNode buildNCT(TreeNode root)
```

that takes a binary tree and builds a new tree. The shape of the new tree is exactly the same as the shape of the original tree, but the values in its nodes are different: in the new tree the value in each node is an `Integer` object that represents the total number of nodes in the subtree rooted at that node. (For example, the value at the root is equal to the total number of nodes in the tree.) The method returns the root of the new tree.

(b) Suppose you have a BST and a companion "node count" tree (NCT) as described in Part (a). Write an efficient method

```
public Object median(TreeNode bstRoot, TreeNode nctRoot)
```

that finds the median of the values stored in the BST. The median here is the $((n + 1) / 2)$-th value in the tree, in ascending order, where n is the total number of nodes (when the nodes are counted starting from 1). ⋜ Hints: write a more general helper function that finds the k-th node in the tree for any given k. Use recursion or iterations building parallel paths in the BST and the NCT from the root to the node you are looking for. ⋝

23. A program maintains a list of doctors on call for the current month. One doctor is on call for each day of the month. Which of the following structures is most appropriate for this task? ✓

A. A `TreeSet` of doctor names
B. A `TreeMap` keyed by doctor names
C. A `TreeMap` keyed by the day of the month
D. An `ArrayList` holding the names of doctors

24.■ (a) Define a class `Movie` that represents a movie record from the `movies.txt` 🖫 file (in the `Ch02\movies` folder on your student disk, see Section 2.7), with fields for the release year, title, director's name, and a list (`LinkedList`) of actors. Make `Movie` objects `Comparable` by title.

(b) Write a program that reads the `movies.txt` file into an array of `Movie` objects and sorts them by title using the `Arrays.sort` method.

(c) Sort the movies array by director's last name using a `Comparator`.

(d) Find out the total number of different actors listed in `movies.txt` by inserting them into a `TreeSet`.

(e) List all the actors from `movies.txt` in alphabetical order by last name using an iterator for the `TreeSet` of actors.

25.■ A set cannot contain "equal" objects (duplicates). In the `TreeSet` implementation, which approach is used to determine whether two objects are equal?

A. `==` operator
B. `equals`
C. `compareTo` (or `compare` if a `TreeSet` was created with a comparator)
D. Both `equals` and `compareTo` (the program throws an exception if the two disagree)

Write a small program to test each hypothesis.