17 Binary Trees

Overview

This chapter introduces a standard data structure called a binary tree. You have seen that a node for a linked list contains data and also a reference to one other node (or to null, at the end of the sequence). A node for a binary tree contains data and also references to two other nodes (or to null).

- Sections 17.1-17.3 develop an application to keeping track of descendents of a particular person. This application uses a family tree to store the ancestor relationships and discusses various ways of traversing the family tree.
- Sections 17.4-17.6 implement the Mapping interface and iterators with a binary search tree, on the condition that data values be Comparable with each other.
- Sections 17.7-17.8 discuss how to keep binary search trees decently balanced -- red-black trees, AVL trees -- and prove useful properties of these trees.
- Section 17.9 discusses more advanced applications of binary tree nodes, e.g., 2-3-4 trees and B-trees.
- Section 17.10 introduces Data Flow Diagrams as a tool for software design and applies it to a company that maintains a database of orders from customers.

17.1 Analysis And Design Of The Genealogy Software

Problem statement

A client doing a genealogical study needs a program to track all of the blood descendents of one individual. She will begin by entering that one individual's name. Thereafter, she will enter the name of each person only after that person's parent has previously been entered. She wants the program to be able to do two things at any point during and after this data entry process, upon request: (a) list all children of a person who has already been entered, one child per line, and (b) display a list of all the people currently in the dataset, one person per line, with those further from the ancestor-of-all listed earlier.

You are part of a software development group that takes on this assignment. You begin by performing an analysis of the requirements for the software, then continue by developing a design for the software. The design is in two parts: The overall main logic and a description of the objects needed by the main logic, along with the services offered by those objects. Services offered by a class of objects are usually expressed as the public methods that can be called for those objects.

Analysis of requirements

First you find out what details are missing from the problem statement, details that you will need to solve the problem. Then you get clarification from the client:

Question In what order should the children of a given person be listed?
Client's Answer "I don't care." So you wait until you develop the algorithm and use whatever is easiest.

Question How will the client be able to tell who is the child of whom in the full listing?
Client's Answer "Oops, I didn't think of that. What do you suggest?"
Our Response Have each person's name indented three spaces further than its parent, and in such a way that you can visually find the parent of any person by going down the list to the first person who is "out-dented" further to the left than that person. The client accepts the suggestion.

Question What if the user asks for information about a person who is not already in the dataset?
Client's Answer Have the program say, "I'm sorry, that person is not in the family" and carry on.

Question How does the user indicate which of the possible choices is wanted?
Client's Answer Enter a single letter: L = list, A = add, S = search, and E = exit.

Question What if two people have the same name?
Client's Answer Assume that the user will never do this and act accordingly.

Main logic design

This family-tree software could proceed as shown in the accompanying design block. It gives the overall logic of the software expressed in ordinary English (though you may use whatever natural language you wish). However, selection between alternative actions and repetitions of actions are indicated by indenting the subordinate actions.

STRUCTURED NATURAL LANGUAGE DESIGN of the main logic

```
1.  Ask the user for the ancestor-of-all and store it in the dataset.
2.  Give directions on the four options available: list all, add one, search for one, exit.
3.  Repeat the following as long as the user does not choose the exit option...
        3a. If the user's choice is to add one person then...
                Get the parent and search the dataset for it.  If there...
                        Ask for the name of the child and add that person.
        3b. If the user's choice is to search then...
                Get the parent and, if there, list all of his or her children.
        3c. If the user's choice is to list everyone then...
                List all of the people of the dataset with proper indenting.
```

Object design

The master design makes it seem quite natural to have a Genealogy dataset object that allows the three capabilities corresponding to the three choices the user has. The IO class developed in Listing 10.2 offers simple input/output services we will use here: `IO.say` prints a message using `JOptionPane.showMessageDialog` and `IO.askLine` prompts for String input using `JOptionPane.showInputDialog`. This basic object design then leads to the coding in Listing 17.1 (see next page).

```
Genealogy (String lilith)
    // constructor to create a dataset of parent-child relations with lilith at the root
public void addChild (String parent)
    // if the given parent is in the dataset, ask for a child of that parent and then add it
public void listChildren (String parent)
    // list all children of the parent, unless the parent is not in the dataset
public void listEverybody()
    // List all values in the dataset, 1 per output line, with all children of any data
    // value X listed before X and in the order they were entered, and each indented
    // 3 spaces further than X.  Also, for each child of X, the first line below that child
    // that is indented less than the child is X itself.
```

Exercise 17.1* Discuss the drawbacks of allowing two people of the same name.
Exercise 17.2* How would the design change if the user were allowed an additional option, namely, to find the parent of a given person?

Listing 17.1 The main logic for the genealogy problem

```
class GenealogyApp
{
   /** Read in names of members of a family with their
    *  parent-child relationships.  Then answer questions about
    *  what people are the children of what other people.  */

   public static void main (String[] args)
   {  String s = IO.askLine ("Enter the ancestor of everyone:");
      Genealogy family = new Genealogy (s);
      IO.say ("Your choices: L = list everyone\n"
            + "\t A = add a child of a person\n"
            + "\t S = search for children of 1 person\n"
            + "\t E = exit the program ");

      String choice = IO.askLine ("Choose L, A, S, or E ");
      while ( ! choice.equals ("E"))
      {  if (choice.equals ("A"))
            family.addChild (IO.askLine
                       ("parent of the person to be added? "));
         else if (choice.equals ("S"))
            family.listChildren (IO.askLine
                       ("parent of children to be listed? "));
         else  // it must be "L"
            family.listEverybody();
         choice = IO.askLine ("Choose L, A, S, or E ");
      }
      System.exit (0);
   }  //======================
}
```

17.2 Implementing The Genealogy Software With Binary Trees

The abstraction we are using here is a non-empty collection of data values organized into a **general tree**. A general tree G has one data value known as its root value. And the tree G has a number of subtrees, also general trees, which contain the other data values of G. G could have 0 or 2 or 13 subtrees, as long as no two subtrees contain the same data value. The root value of each subtree of G is a child of the root value of G.

Implementing a general tree using a binary tree

You can implement a general tree in coding quite easily as follows: A TreeNode object stores the root data value of the general tree plus a linked list of its children, reading left-to-right for the most-recently-added down to the first-one-added. For a TreeNode `X`, `X.itsLeft` is the TreeNode containing the leftmost child of `X`. Also, `X.itsRight` is a reference to the next child of the parent of `X` to X's right. Concretely...

- `X.itsData` is the root value of X.
- `C1 = X.itsLeft` contains X's leftmost child, the last-added child of X.
- `C2 = C1.itsRight` contains X's second child from the left, added just before `C1.itsData`.
- `C3 = C2.itsRight` contains X's third child from the left, etc.

Figure 17.1 is an example of a Genealogy dataset, with single letters for data values. The abstract general tree is on the right; the concrete TreeNodes are on the left.

If b is the TreeNode whose root value is B, then `b.itsRight` is the TreeNode with root value H and `b.itsLeft` is the TreeNode with root value C. The TreeNode `b.itsLeft.itsRight` has root value D. The TreeNode class in Listing 17.2 (see next page) can be used to implement the Genealogy class. The coding in Listing 17.2 contains some methods not needed for the Genealogy software, just to give you some feel for how to work with tree structures. All but the last one allow you to get information about a tree but not modify it.

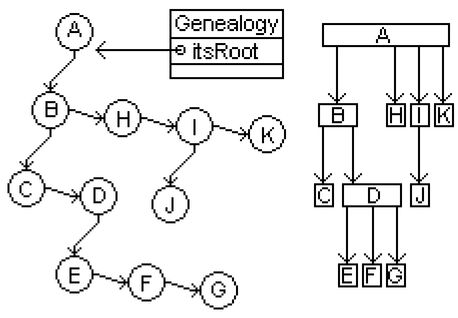


Figure 17.1 TreeNodes vs. general

We let **ET**, standing for Empty Tree, be one particular empty tree (i.e., having no data). This turns out to be advantageous at times; it is like a trailer node for linked lists. The following coding adds `newInfo` to the left of a `node`, thus making `newInfo` the most recently added child of the `node`'s data value. It puts the current linked list of the children of the `node` off to the right of the newly-added child, and it automatically puts `ET` to the left of the newly-added `node` to signal that this child has no children (as yet):

```
TreeNode newNode = new TreeNode (newInfo);
newNode.itsRight = node.itsLeft;
node.itsLeft = newNode;
```

The `size` method in the middle part of Listing 17.2 tells the number of data values in the TreeNode executor. It is zero if the executor is empty, otherwise it is 1 for the executor's root value plus the number of all data values to the left of the executor plus the number of all data values to the right of the executor. If you were to call `b.size()` for b being the node containing B in Figure 17.1, it would get 5 from its call of `c.size()` (c being the node containing C) and get 4 from its call of `h.size()` (h being the node containing H), and so it would return 10 (which is 1 + 5 + 4).

The `firstNode` method in Listing 17.2 returns the node that is furthest left in the executor's tree, assuming that the executor is a non-empty tree. This furthest-left node is (a) itself if the executor has an empty left subtree, otherwise it is (b) the furthest-left node of its left subtree. If you were to call `a.firstNode()` for a being the node containing A in Figure 17.1, it would call `b.firstNode()`, which would call `c.firstNode()`, which would return c.

The `deleteLastNode` method in Listing 17.2 deletes the furthest node to the right in the executor's tree, assuming that the executor has a non-empty node to its right. If the executor's right (call it R) has only the empty tree to its right, the executor replaces R by whatever is to R's left (which may or may not be empty). Otherwise the executor asks R to delete the furthest node to the right of R. If you were to call `c.deleteLastNode()` for c being the node containing C in Figure 17.1, it would see that `d.itsRight` is empty (d is the node containing D) and therefore set `c.itsRight` to be e (the node containing E).

Binary trees

The TreeNode class implements a binary tree structure. A **non-empty binary tree** is a structure consisting of one data value (its **root value**) plus two trees called its **left subtree** and its **right subtree**. It is represented in coding as a **node** object, called the **root node** of that tree. There are two kinds of nodes, internal nodes and external nodes. An **internal node** has a minimum of three attributes (instance variables): a data value stored there plus references to the root nodes of the left subtree and the right subtree. A **leaf node** is an internal node for which both subtrees are empty.

Listing 17.2 The TreeNode class used by the Genealogy class

```
public class TreeNode
{
   public static final TreeNode ET = new TreeNode (null);
   private Object itsData;  // the data value stored in this tree
   private TreeNode itsLeft  = ET; // left subtree of this tree
   private TreeNode itsRight = ET; // right subtree of this tree


   public TreeNode (Object given)
   {  itsData = given;
   }  //=======================

   public boolean isEmpty()
   {  return this.itsData == null;  // generally, only ET
   }  //=======================

   public Object getData()
   {  return itsData;
   }  //=======================

   public TreeNode getLeft()
   {  return itsLeft;
   }  //=======================

   public TreeNode getRight()
   {  return itsRight;
   }  //=======================


   /** Return the number of data values in the tree. */

   public int size()
   {  return this.isEmpty() ? 0
             :  1 + this.itsLeft.size() + this.itsRight.size();
   }  //=======================


   /** Return the leftmost node in the tree rooted at this.
    *  Throw an Exception if this is an empty tree.  */

   public TreeNode firstNode()
   {  return itsLeft.isEmpty()  ?  this : itsLeft.firstNode();
   }  //=======================


   /** Precondition: this is a non-empty node with a non-empty
       node to its right.  Postcondition: the rightmost node of
       the subtree rooted at this is removed. */

   public void deleteLastNode()
   {  if (itsRight.itsRight.isEmpty())
         itsRight = itsRight.itsLeft;
      else
         itsRight.deleteLastNode();
   }  //=======================
}
```

An **empty binary tree** does not contain any data or any subtrees; it is an **external node**. In figures, internal nodes are typically drawn as ellipses and external nodes are drawn as rectangles. Figure 17.2 shows all binary trees of 1 to 3 data values, though only the first three show the external nodes explicitly. We often draw only internal nodes; it is understood that each node drawn with less than two subtrees has an external node for each undrawn subtree.

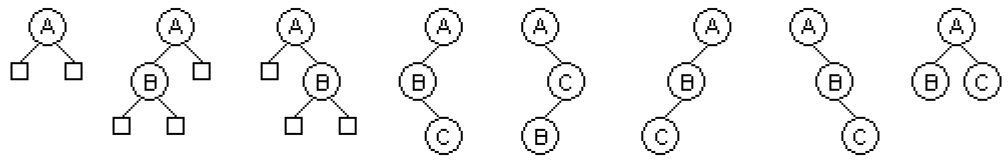


Figure 17.2 Eight binary trees, the first three with external nodes drawn

A **path** in a tree from one node to another is a sequence of nodes beginning with the first and ending with the second such that, for each time node Y follows node X in the sequence, node Y is the root of a subtree of node X. For instance, the fourth binary tree in Figure 17.2 has two paths of length 1 (from the A-node to the B-node and from the B-node to the C-node) and one path of length 2 (from the A-node via the B-node to the C-node).

A binary tree has a limitation on the way in which nodes can be connected, namely, you cannot have two different paths from one internal node to another internal node, and you cannot have a path from an internal node to itself. In particular, none of the nodes in the left subtree of a binary tree can be in the right subtree.

Another method that could be in the TreeNode class is the following, which prints all the data values in a given binary tree in order from left to right. The logic is (a) an empty tree has no data values to print, but (b) printing the data values in a non-empty tree requires that you first print all the data values in its left subtree in order from left to right, then print the root value, then print all the data values in its right subtree in order from left to right.

```
public void printInOrder()   // in the TreeNode class
{  if (this.isEmpty())
      return;  // a void method forbids returning with a value
   itsLeft.printInOrder();
   System.out.println (itsData.toString());
   itsRight.printInOrder();
}  //=======================
```

For the fourth tree in Figure 17.2, this method would print the data in the order B, C, A. For the eighth tree in Figure 17.2, this method would print the data in the order B, A, C.

Exercise 17.3 State the order in which the `printInOrder` method would print the data in the fifth, sixth, and seventh trees in Figure 17.2.
Exercise 17.4 Describe the action of `c.size()` on the tree in Figure 17.1, using the nomenclature of this section.
Exercise 17.5 Describe the action of `b.deleteLastNode()` on the tree in Figure 17.1, using the nomenclature of this section.
Exercise 17.6 Write a TreeNode method that returns the rightmost node in the tree.
Exercise 17.7 Write a TreeNode method that deletes the leftmost node in the tree.
Exercise 17.8 State the big-oh of the average execution time of the `printInOrder` method and of the `deleteLastNode` method.
Exercise 17.9* Draw all binary trees with four data values.
Exercise 17.10* Draw all binary trees with five data values.

17.3 Searching Through A Binary Tree

For the Genealogy software, you will sometimes need to search for the TreeNode that contains a particular person. This search process can potentially go through every node in the entire genealogy dataset, since there are no ordering properties to speed the search. If the search method finds the person somewhere in the dataset, the `addChild` method then asks the user for the child to be added for that person and puts the child in the dataset. Similarly, the `listChildren` method searches the dataset for a user-specified person and then lists all the children that person has, in last-in-first-out order.

One way of searching a binary tree is **breadth-first search**. This means that you first look at the data at the root, then at the data to the right of the root, then at the data to the left of the root, and only then at the data below them. And only after checking out all of the data two steps away from the root would you search through the data that is three steps away from the root. This kind of search can be controlled by a queue as shown in Listing 17.3. Note that, even though the root node of a tree used for a Genealogy dataset is empty on its right, this coding checks the tree on the right anyway. That allows this coding to be used for trees that have data on the right of the root node.

Listing 17.3 A non-recursive search method in TreeNode using a queue

```
/** The search process implemented with a queue. Return the
 *  TreeNode containing given, or ET if given is not there.
 *  Precondition:  The executor is not empty. */

public TreeNode search (Object given)
{  QueueADT people = new NodeQueue();                    //1
   people.enqueue (this);                                //2
   do                                                    //3
   {  TreeNode node = (TreeNode) people.dequeue();       //4
      if (node.itsData.equals (given))                   //5
         return node;                                    //6
      if ( ! node.itsRight.isEmpty())                    //7
         people.enqueue (node.itsRight);                 //8
      if ( ! node.itsLeft.isEmpty())                     //9
         people.enqueue (node.itsLeft);                  //10
   }while ( ! people.isEmpty())                          //11
   return ET;                                            //12
}  //=======================
```

QueueADT is described in Chapter Fourteen; all you really have to know for this chapter is that the `enqueue` method adds an Object value (its parameter) to the rear, the `dequeue` method removes an Object value from the front, and the `isEmpty` method checks to see whether you have a value to remove. **NodeQueue** is one particular implementation of the QueueADT interface.

Similarly, **StackADT** is described in Chapter Fourteen; all you really have to know for this chapter is that the `push` method adds an Object value (its parameter) to the top, the `pop` method removes an Object value from the top, and the `isEmpty` method checks to see whether you have a value to remove. **NodeStack** is one particular implementation of the StackADT interface.

This search method first puts this root node in the queue. Then it starts the loop, whose first action is to remove from the Queue `node = this`, which is at the first **level** of the binary tree. It tests `node.itsData.equals (given)`. If true, it returns this node. If false, it puts `this.itsLeft` in the queue (and also `this.itsRight` if it is called for some other tree that has a non-empty node to its right). Now the node or nodes on the second level of the binary tree are in the queue.

On the second time through the loop, the coding removes the `node` on the left of the root and processes it similarly. It puts `node.itsRight and node.itsLeft` on the Queue, etc. This is considered breadth-first whether you look at the right before the left or vice versa. In the earlier Figure 17.1, this search process would visit the data values in this order: A, then B (1 level away from the top), then H and C (2 levels away), then I and D (3 levels away), then K, J, and E (4 levels away), then F, then G. Note that levels are expressed in terms of the binary tree structure, not the data structure itself.

Depth-first search

Depth-first search means that, after looking at the data in a particular node, you look at all the data in its subtree before you look elsewhere. Concretely, first you check out the data in the root. If you find what you want there, you quit. Otherwise you search through the entire left subtree of the root, going down however many levels are necessary. Only if you do not find what you are looking for there do you go on to the right subtree of the root and search through all of the levels of that subtree.

A non-recursive depth-first search can be coded exactly the way it is done in Listing 17.3 except you replace "Queue" by "Stack" and therefore "enqueue" by "push" and "dequeue" by "pop". In the earlier Figure 17.1, this search process would visit the data values in this order: A, then B, then B's left subtree C, D, E, F, and G, then B's right subtree H, I, J, and K.

For a recursive depth-first search, you would first check out the data in the root (if any; if the root is empty, then of course the result of the search is an empty tree). If you do not find what you want in the root, call the search method recursively for the left subtree of the root and see what you get back from that call. If you get back anything but the empty tree, you have found the desired TreeNode value and you do not need to search further. Otherwise, search the right subtree of the root and return whatever it produces.

In Figure 17.3, the breadth-first coding in Listing 17.3 will look at the information in the following order (one level at a time): 54, 76, 28, 90, 64, 40, 17. But depth-first search would look at the information in the following order: 54, 28, 17, 40, 76, 64, 90.

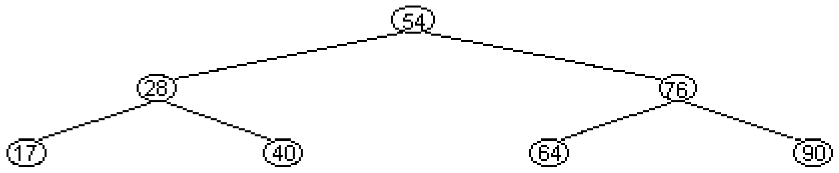


Figure 17.3 A binary tree with 7 nodes

Traversals

A recursive depth-first search is called **left-to-right preorder traversal**. "Traversal" means you visit each node in the tree (potentially). "Left-to-right" means that you visit the nodes in the left subtree before you visit any node in the right subtree; the alternative is right-to-left traversal. "Preorder" means that you visit the root node of each subtree before you visit any of the nodes in its subtrees. The alternatives to preorder are **postorder traversal** (visit the root node after you visit all the nodes of the subtrees) and **inorder traversal** (visit the root node in between visiting the nodes of the subtrees).

For instance, the `printInOrder` method coded at the end of the preceding section is a <u>left-to-right inorder traversal</u>. An example of <u>right-to-left inorder traversal</u> is the following `toString` method to return a String value containing a representation of the values in the entire binary tree. It is right-to-left because Java always evaluates the operands of the plus operator from left to right.

```
public String toString()  // in the TreeNode class
{  if (this.isEmpty())
      return "";
   else
      return " (" + itsRight.toString() + itsData.toString()
                  + itsLeft.toString() + ") ";
}  //=======================
```

For the binary tree in Figure 17.3, this `toString` method would return the String form of the right subtree followed by 54 followed by the String form of the left subtree. The String form of the right subtree is "((90) 76 (64))". The String form of the left subtree is "((40) 28 (17))". So the output from a call of `toString` would be "(((90) 76 (64)) 54 ((40) 28 (17)))".

Listing 17.4 (see next page) codes the Genealogy class, using the IO class from Listing 10.2. It calls on the `search` method in Listing 17.3 and a recursive `listAll` method in TreeNode that traverses the tree, listing all the data values with each value indented appropriately. This `listAll` method is left as an exercise. It is a <u>right-to-left postorder traversal</u>. In the Genealogy class, `itsRoot` is a reference to the root of a binary tree containing the family members.

Exercise 17.11 Consider a binary tree with the same structure as shown in Figure 17.3, but with different data values in the seven nodes. If the order of the data values produced by the breadth-first search method in Listing 17.3 is A,B,C,D,E,F,G, then what is the order produced by the depth-first search method described in this section?
Exercise 17.12 Consider a binary tree with the same structure as shown in Figure 17.3, but with different data values in the seven nodes. If the order of the data values produced by the recursive depth-first search method described in this section is A,B,C,D,E,F,G, then what is the order produced by the breadth-first search method in Listing 17.3?
Exercise 17.13 Is it possible to have a binary tree of seven nodes in which both of the algorithms mentioned in the preceding exercise produce exactly the same sequence of data values? If so, what does it look like?
Exercise 17.14 (harder) What is the relation between the number of parentheses in the output of the `toString` method at the end of this section and the levels of the data values?
Exercise 17.15** Write the `search` function that must be added to the TreeNode class in order to make the Genealogy methods work correctly. It does not assume that the dataset is ordered in any way. You are to code it recursively as described in the text, without a stack or queue. Be sure to have it stop the search as soon as it finds a match.
Exercise 17.16** Genealogy's `listEverybody` method calls a recursive tree traversal method `itsRoot.listAll("")` with one parameter, which is a String containing the right number of blanks to indent the executor node's data. Each data value should be indented 3 blanks for every generation it is removed from the root data. Code `listAll` so that it prints all the data values stored in the right subtree of the root node, then prints all the data values stored in the left subtree of the root node, then prints the data in the root node. For instance, for the data in Figure 17.1, it will print A at the bottom of the list; it will print K, I, H, and B in that order, each indented 3 blanks (since they are the children of A), with K first; and it will print J just before I and indented 6 blanks.

Listing 17.4 The Genealogy class

```
public class Genealogy
{
   private TreeNode itsRoot;   // the root node of the tree

   /** Create a dataset of parent-child relations with lilith as
    *  the root data value and no other values in the dataset. */

   Genealogy (String lilith)
   {  itsRoot = new TreeNode (lilith);                        //1
   }  //=======================

   /** If the given parent is in the dataset, ask for a child of
    *  that parent and then add that child to the dataset. */

   public void addChild (String parent)
   {  TreeNode node = itsRoot.search (parent);                //2
      if (node == TreeNode.ET)                                //3
         IO.say ("I'm sorry, that person is not in the family");
      else                                                    //5
         node.pushLeft (IO.askLine ("What's the child's name?"));
   }  //=======================

   /** List all children of the parent. */

   public void listChildren (String parent)
   {  TreeNode node = itsRoot.search (parent);                //7
      if (node == TreeNode.ET)                                //8
         IO.say ("I'm sorry, that person is not in the family");
      else if (node.getLeft() == TreeNode.ET)                 //10
         IO.say ("None are in the dataset");                  //11
      else                                                    //12
      {  TreeNode kid = node.getLeft();                       //13
         String s = kid.getData().toString();                 //14
         for (kid = kid.getRight();  kid != TreeNode.ET;      //15
                                     kid = kid.getRight())    //16
            s += ", " + kid.getData().toString();             //17
         IO.say (s);                                          //18
      }                                                       //19
   }  //=======================

   /** List all values in the dataset, one per output line,
    *  with all children of any data value X listed before X
    *  and in the order they were entered (right-to-left
    *  postorder), and each indented 3 spaces further than X. */

   public void listEverybody()
   {  itsRoot.listAll ("");  // left as a recursive exercise //20
   }  //=======================
}

// The following method is added to the TreeNode class

   public void pushLeft (Object newInfo)
   {  TreeNode newNode = new TreeNode (newInfo);              //21
      newNode.itsRight = this.itsLeft;                        //22
      this.itsLeft = newNode;                                 //23
   }  //=======================
```

17.4 Implementing The Mapping Interface With Binary Search Trees

The **Mapping** interface (Listing 16.4) simplifies Sun's standard library Map interface:

```
public Object put (Object id, Object value);  // add id/value
public boolean containsKey (Object id); // is this id in it?
public Object get (Object id); // return the value for this id
public Object remove (Object id); // remove the id/value
public boolean isEmpty();         // has it no id/value pairs?
public int size();                // how many id/value pairs?
public java.util.Iterator iterator(); // for listing all pairs
```

One way to implement the Mapping interface is to use a binary search tree, as long as all the keys are from a class that implements Comparable (and so contains the usual `compareTo` method). A nonempty tree contains one MapEntry, its root data, plus two subtrees, one on its left and one on its right. An empty tree contains no entry or subtree. A **MapEntry** object (described in Listing 16.5) has two final Object instance variables whose values can be obtained using `me.getKey()` and `me.getValue()`.

In a **binary search tree**, keys in a left subtree are less than the key for the root data, and keys in a right subtree are greater than or equal to the key for the root data. Mappings do not allow duplicate keys. So if `Lkey` is any key from a MapEntry data value in X's left subtree, and if `Rkey` is any key from a MapEntry data value in X's right subtree, then `Lkey.compareTo (((MapEntry) X.itsData).getKey()) < 0` and also `Rkey.compareTo (((MapEntry) X.itsData).getKey()) > 0`.

If the tree is well balanced, then about half of the entries will be in the left subtree and about half will be in the right subtree. A perfectly balanced tree with 31 entries has 15 entries in its left subtree and 15 entries in its right subtree (and so the root data would be the middle value as determined by `compareTo`). Each of those subtrees has 7 entries in each of its two subtrees, etc. The tree has a total of five levels; the fifth level has 16 subtrees each with two empty subtrees. Figure 17.4 shows such a tree.

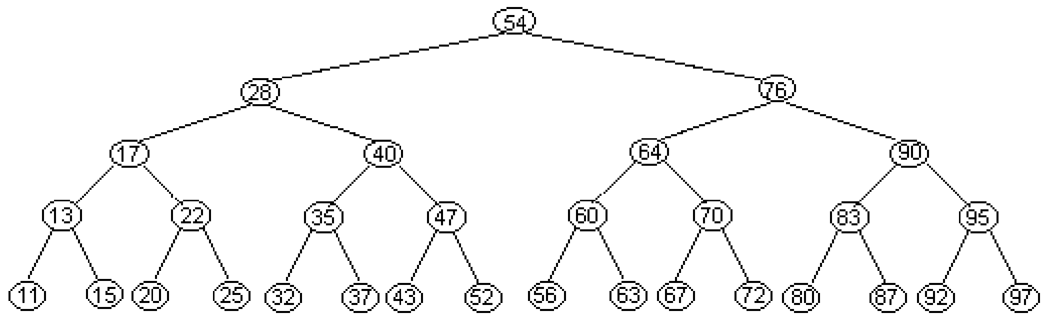


Figure 17.4 Example of a balanced binary search tree with 31 entries

Searching in a binary search tree

When you search for a particular key in this 31-node tree, you only need to look at at most five different entries before you find the entry you are looking for (or find that it is not in the data structure at all). Specifically, you compare the given `id` with the root data 54. That tells you whether to look in its left subtree or in its right subtree (unless it equals the root data). Either way, you are in a subtree with only 15 possibilities left instead of the original 31. When you compare the given `id` with the root of the subtree you are in, it tells you whether to look in its left subtree or in its right subtree. That reduces the number of possibilities to 7, again cutting them in half (unless you have already found it on the first two levels). This process continues for at most five levels.

There are only ten levels in a balanced binary tree with a thousand entries and only twenty levels in a balanced binary tree with a million entries. So you can find a value you are looking for by looking at only twenty different values among a million entries. That is just not possible with a linked list. Of course, you get the same speed using binary search in an ordered array (in Section 13.3). A balanced binary search tree has a close correspondence with an ordered array (this is why it is called a binary search tree): The root data is roughly the middle value in the array; the root data of the left subtree is roughly the value a quarter of the way up from the bottom of the array; etc.

Inserting in a binary search tree

The drawback to using an ordered array is that putting a new entry in the array or taking an existing one out is so slow. If you have a million entries, you may have to make almost a million assignments of variables to make room for a new one that is near the front of the array. And you may have to make almost a million assignments to move entries down when you remove an existing one that is near the front of the array.

With a binary search tree, you only have to make less than half-a-dozen assignments to put a new entry in the data structure or remove an existing entry, once you find where it goes. That is an enormous amount of time saved. And that is true even if you have a million million million entries.

Our **BintMap** implementation of Mapping is in Listing 17.5 (see next page) has just one instance variable, the TreeNode stored in `itsRoot`. Each TreeNode in the binary search tree stores a MapEntry value in `itsData` (though `itsData` is declared as type Object). To simplify coding comparisons of Comparable `id` values with the key value in `itsData`, we use the `compare` method defined in the bottom of Listing 17.5.

Since the Object class has a `hashCode` method, we could instead use each ID's int `hashCode` value (described in Section 16.7) to decide where a data value is stored. Then the IDs would not have to be Comparable; they would just have to come from a class of objects that overrides the Object `hashCode` method appropriately.

Each TreeNode X has references to its two subtrees, `X.itsLeft` and `X.itsRight`. The nodes at the roots of those two subtrees are the **left child** and **right child** of `X`. `X` is their **parent**. If a node has no data to its left, `itsLeft` is an empty tree (normally `ET`), and similarly for `itsRight`.

The lookUp method

The `size` and `isEmpty` methods for a BintMap simply return `itsRoot.size()` and `itsRoot.isEmpty()`, respectively (calling methods in Listing 17.2). The `containsKey` method returns `false` if the `id` parameter is null or if the root node has no data. Otherwise `containsKey` calls a recursive `lookUp` method in the TreeNode class to return the node that has `itsData.getKey()` equal to `id` (if any). The logic of the `lookUp` method follows the structure of a binary search tree:

- If the `id` is less than the root data, the search should be made in the left subtree, and the result that search produces is what the method returns.
- If the `id` is larger than the root data, the search should be made in the right subtree, and the result that search produces is what the method returns.
- The only case left is that the `id` equals the root data, in which case your search is successfully completed and you can return this current node.

For the `get` method, if the `id` is null or if the executor is empty of data or if `lookUp` returns null, `get` returns null, otherwise `get` returns the value part of the id/value pair in the node that `lookUp` returns.

Listing 17.5 The BintMap class, partially done

```
//Methods throw a RuntimeException for non-Comparable ids.

public class BintMap implements Mapping
{
   private TreeNode itsRoot;


   public BintMap()
   {  itsRoot = TreeNode.ET;
   }  //======================

   public int size()
   {  return itsRoot.size();
   }  //======================

   public boolean containsKey (Object id)
   {  return id != null && ! this.isEmpty()
                 && itsRoot.lookUp ((Comparable) id) != null;
   }  //======================

   public Object get (Object id)
   {  if (id == null || this.isEmpty())
         return null;
      TreeNode loc = itsRoot.lookUp ((Comparable) id);
      return (loc == null)  ?  null
              :  ((MapEntry) loc.getData()).getValue();
   }  //======================
}



// The following 2 methods are added to the TreeNode class

   /** Return null if id is nowhere in the subtree rooted at
    *  this node.  Otherwise return the TreeNode containing id.
    *  Precondition: id is not null, nor is this.itsData; also,
    *  this tree is a binary search tree. */

   public TreeNode lookUp (Comparable id)
   {  if (compare (id, itsData) < 0)
         return itsLeft.isEmpty() ? null : itsLeft.lookUp (id);
      if (compare (id, itsData) > 0)
         return itsRight.isEmpty() ? null : itsRight.lookUp (id);
      return this;  // this is the node with the id
   }  //======================

   private static int compare (Comparable id, Object data)
   {  return id.compareTo (((MapEntry) data).getKey());
   }  //======================
```

Example for `lookUp` If the root node in Figure 17.4 performs `lookUp` for the `id` 47, it asks the node to its left containing 28 to perform `lookUp`, which asks the node to its right containing 40 to perform `lookUp`, which asks the node to its right containing 47 to perform `lookUp`, which returns itself. But if the `id` had been 45, the node containing 47 would then have asked the node to its left containing 43 to perform `lookUp`, which would have seen `ET` to its right and therefore returned null.

The put method

The `put` method for the BintMap class is in Listing 17.6. After first checking that the `id` parameter is not null and the BintMap is not completely empty, `put` calls a recursive TreeNode method structured similarly to `lookUp`. Closely compare `putRecursive` with `lookUp`. Note the **indirect recursion**: `putRecursive` calls one of two other methods, each of which can call `putRecursive` again.

Listing 17.6 The put method for the BintMap class

```
   public Object put (Object id, Object value)
   {  if (id == null)                                     //1
         return null;                                     //2
      if (this.isEmpty())                                 //3
      {  itsRoot = new TreeNode (new MapEntry (id, value));  //4
         return null;                                     //5
      }                                                   //6
      return itsRoot.putRecursive ((Comparable) id, value);  //7
   }  //======================

// The following 3 methods are added to the TreeNode class

   /** Add the id/value pair to the non-empty binary search tree.
    *  Return the value the id previously had (null if none). */

   public Object putRecursive (Comparable id, Object val)
   {  if (compare (id, itsData) < 0)    // go left           //8
         return putInLeftSubtree (id, val);                  //9
      if (compare (id, itsData) > 0)    // go right          //10
         return putInRightSubtree (id, val);                 //11
      Object valueToReturn = ((MapEntry) itsData).getValue();//12
      itsData = new MapEntry (id, val);                      //13
      return valueToReturn;                                  //14
   }  //======================

   private Object putInLeftSubtree (Comparable id, Object val)
   {  if (itsLeft.isEmpty())                                 //15
      {  itsLeft = new TreeNode (new MapEntry (id, val));    //16
         return null;                                        //17
      }                                                      //18
      else                                                   //19
         return itsLeft.putRecursive (id, val);              //20
   }  //======================

   private Object putInRightSubtree (Comparable id, Object val)
   {  if (itsRight.isEmpty())                                //21
      {  itsRight = new TreeNode (new MapEntry (id, val));   //22
         return null;                                        //23
      }                                                      //24
      else                                                   //25
         return itsRight.putRecursive (id, val);             //26
   }  //======================
```

The executor of the `putRecursive` method puts the MapEntry in the left or right subtree depending on whether the `id` is smaller or larger than its data. But if it is equal, it simply replaces the corresponding value. Putting a value in the left subtree consists in calling the `putRecursive` method unless the left subtree is empty, in which case it is replaced by a one-node subtree containing the new MapEntry value. Putting a value in the right subtree is line-for-line analogous.

Example for `put` If the root node in Figure 17.4 performs `putRecursive` for the id 45, it asks the node to its left containing 28 to perform `putRecursive`, which asks the node to its right containing 40 to perform `putRecursive`, which asks the node to its right containing 47 to perform `putRecursive`, which asks the node to its left containing 43 to perform `putRecursive`, which sees that 45 goes to its right where it has an empty subtree, so it creates a new node to its right and stores the 45 in it.

The remove method

The hardest method to implement for a binary search tree is the `remove` method. This method requires that you search through the tree to find a data value whose key matches the one given by the parameter. Then you remove it without altering the relationships of the other data values to each other. You are to return null if the search fails.

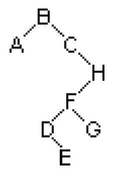
If the root node is empty or the `id` is null, you can simply return null. Otherwise you need to find the node that contains the data with that `id`; `lookUp` will do this for you. If `lookUp` returns null, the `id` is not in the BintMap. Otherwise, you can extract the data from that node, remove that data from the binary tree, and return the data's value. This logic is in the upper part of Listing 17.7.

Listing 17.7 The remove method for the BintMap class

```
   public Object remove (Object id)
   {  if (id == null || this.isEmpty())
         return null;
      TreeNode loc = itsRoot.lookUp ((Comparable) id);
      if (loc == null)
         return null;
      MapEntry data = (MapEntry) loc.getData();
      loc.removeData();
      return data.getValue();
   }  //======================


// The following method is added to the TreeNode class

   /** Precondition: The executor is a non-empty binary search
    *  tree.  Postcondition:  The data in the executor node is
    *  removed, leaving a binary search tree 1 node smaller.  */

   public void removeData()
   {  if (itsRight.isEmpty())
      {  itsData  = itsLeft.itsData;
         itsRight = itsLeft.itsRight;
         itsLeft  = itsLeft.itsLeft;
      }
      else if (itsRight.itsLeft.isEmpty())
      {  itsData  = itsRight.itsData;
         itsRight = itsRight.itsRight;
      }
      else
      {  TreeNode p = this.itsRight;
         while ( ! p.itsLeft.itsLeft.isEmpty())
            p = p.itsLeft;
         this.itsData = p.itsLeft.itsData;
         p.itsLeft = p.itsLeft.itsRight;
      }
   }  //======================
```

For the `removeData` method: Removing the data value in a linked list is easy – you simply copy the values from the next node into the current node, thereby replacing its data value by the next node's data and then bypassing the next node. You can also do this in a binary tree if the current node only has a left subtree, i.e., its right subtree is empty (this is in lines 9-12). What do you do if the right subtree is not empty? You obtain a replacement data value from the right subtree.

With which value in the current node's right subtree do you replace the current node's data? Only the data in the leftmost node, since that is the data value that comes immediately after the current node's data value. And since that leftmost node has an empty left subtree, you may replace it by its own right subtree without altering the ordering relationships of the data values. Study Listing 17.7 carefully to see this.



Example for `remove` In the small binary seach tree figure at right:
To remove data H, lines 10-12 copy F and its two subtrees into the H-node.
To remove data B, lines 15-16 copy C and its right subtree into the B-node.
To remove data C instead, lines 19-21 set p to the F-node; line 22 copies D into the C-node; and line 23 assigns the right subtree of the D-node to be the left subtree of the F-node (the original D-node is thereby deleted).

Creating a balanced tree from an array

If you have a filled array of one or more MapEntry values in ascending order (every component having a MapEntry value), you could create a nicely balanced binary tree from those values by calling the following method in the TreeNode class with the statement `make(item, 0, item.length - 1)`:

```
public static TreeNode make (Object[] item, int lo, int hi)
{  int middle = (lo + hi + 1) / 2;  // halfway from lo to hi
   TreeNode root = new TreeNode (item[middle]);
   if (lo < middle)
      root.itsLeft = make (item, lo, middle - 1);
   if (middle < hi)
      root.itsRight = make (item, middle + 1, hi);
   return root;
}  //======================
```

Exercise 17.17 Rewrite the `lookUp` method to only call `compare` at most one time for each pair of values, instead of wasting execution time calling it twice.
Exercise 17.18 Rewrite the `removeData` method so that it checks for an empty left subtree and makes the easy removal in that case. Does this improve overall execution time?
Exercise 17.19 (harder) Write a TreeNode method `public TreeNode copy()`: The non-empty executor returns a new binary search tree containing exactly the same entries as the executor's binary search tree and in the same relative positions. Use recursion.
Exercise 17.20 (harder) Write a TreeNode method `public int under (Comparable id)`: The executor tells how many of its MapEntries have a key less than that of the given key.
Exercise 17.21* Write a BintMap method `public Comparable firstKey()` that returns the smallest key. Return null if the executor is empty. Use recursion.
Exercise 17.22* Rewrite the `lookUp` method in Listing 17.5 without using recursion.
Exercise 17.23* Write a recursive TreeNode method `public TreeNode reverse()`: the executor returns a new binary tree with the same entries in the opposite order.
Exercise 17.24** Write a recursive BintMap method `public boolean equals (BintMap given)`: The executor tells whether the BintMap parameter has the same key/value pairs in the same order and the same positions in the tree as the executor.
Exercise 17.25** Write out the internal invariant for the BintMap class.

17.5 Implementing The Iterator Interface For A Binary Search Tree

A Mapping needs an Iterator object so a client can obtain the values in the Mapping one at a time. An obvious implementation is to keep track of the TreeNode that contains the information to be returned by a call of `next()`. This is like what the array implementation and the linked list implementation in Chapter Sixteen did:

In ArrayMap, `next()` returns `itsItem[itsPos]` if `itsPos < itsSize`.
In NodeMap, `next()` returns `itsPos.itsData` if `itsPos.itsData != null`.

For the BintMap class, we could have `next()` return `itsPos.itsData` where `itsPos` is a TreeNode, just as we did for the NodeMap class. This coding is in Listing 17.8 (see next page). Our Mapping iterators disallow `remove` to make this chapter simpler. When we need to move on to the next TreeNode, we do one of two things:

- If the right subtree of `itsPos` contains any data, then the next value is the first value in that right subtree. So we set `itsPos` to be the leftmost node in that right subtree.
- Otherwise, the next value after the one in `itsPos` is a value that is on the path from the root node down to `itsPos`. It will be in a node that contains `itsPos` in its left subtree, since the value in `itsPos` comes before it. If there are two or more nodes on that path that have `itsPos` in the node's left subtree, we want the one that is furthest down the path.

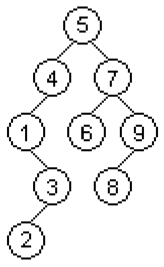


Figure 17.5 Binary search tree

The algorithm described above is <u>left-to-right inorder traversal</u> of the nodes, which is in ascending order using `compareTo`. An iterator is not required to produce the data values in this order, but it is preferable. Figure 17.5 shows a binary search tree with nine values in it. The effects of the first few accesses to the iterator are as follows:

- The constructor sets `itsPos` to refer to the leftmost node, which contains 1.
- A call of `next()` returns that 1 and then, when it sees that `itsPos.itsRight` is not the empty tree, sets `itsPos` to the leftmost node in that right subtree, which contains 2.
- Another call of `next()` returns that 2 and then, when it sees that `itsPos` has an empty right subtree, starts from the root to search down the tree for `itsPos`. It notes that `itsPos` is to the left of the node containing 5, and of the node containing 4, and of the node containing 3. Since the node containing 3 is lowest in the tree of all of those, it sets `itsPos` to that node containing 3.
- Another call of `next()` returns that 3 and then, when it sees that `itsPos` has an empty right subtree, starts from the root to search down the tree for `itsPos`. It notes that `itsPos` is to the left of the node containing 5 and also of the node containing 4. Since the latter is lower in the tree, it sets `itsPos` to that node containing 4.

Listing 17.8 uses a throw statement when someone calls a method that should not be called (`remove`), or calls a method under conditions when it should not be called (`next` when `hasNext()` is false). That is a bug in the coding that calls it. Notification of the bug is through the following kind of statement, which immediately terminates the method containing the statement:

```
throw new RuntimeException ("some explanatory message");
```

Listing 17.8 The MapIt class inside the BintMap class, providing iterators

```
   public java.util.Iterator iterator()    // member of BintMap
   {  return new MapIt (this.itsRoot);                        //1
   }  //======================


   private static class MapIt implements java.util.Iterator
   {
      private TreeNode itsPos;
      private TreeNode itsRoot;


      public MapIt (TreeNode given)
      {  itsRoot = given;                                     //2
         itsPos = given.isEmpty() ? null : given.firstNode();//3
      }  //======================


      public boolean hasNext()
      {  return itsPos != null;                               //4
      }  //======================


      public Object next()
      {  if ( ! hasNext())                                    //5
            throw new java.util.NoSuchElementException        //6
                       ("iterator has no next element!");     //7
         Object valueToReturn = itsPos.getData();             //8
         itsPos = itsPos.nextIn (itsRoot);                    //9
         return valueToReturn;                                //10
      }  //======================

      public void remove()
      {  throw new UnsupportedOperationException ("no remove!");
      }  //======================
   }


// The following method is added to the TreeNode class

   /** Return the next TreeNode after the executor that appears
    *  in left-right inorder traversal; return null if there is
    *  none. Precondition: The executor is not empty, . */

   public TreeNode nextIn (TreeNode ancestor)
   {  if ( ! this.itsRight.isEmpty())                         //12
         return this.itsRight.firstNode();                    //13
      Comparable id = (Comparable) ((MapEntry) itsData).getKey();
      TreeNode lastLeftTurn = null;                           //15
      while (ancestor != this)                                //16
      {  if (compare (id, ancestor.itsData) > 0)              //17
            ancestor = ancestor.itsRight;                     //18
         else                                                 //19
         {  lastLeftTurn = ancestor;                          //20
            ancestor = ancestor.itsLeft;                      //21
         }                                                    //22
      }                                                       //23
      return lastLeftTurn;                                    //24
   }  //======================
```

Using parent pointers

The nextIn method has to go through several nodes, starting from the root of the tree and moving down to the current position, in order to move on to the next position. It would be helpful if we could avoid the time it takes to execute that loop. One way is to have each node keep track of its parent. That is, we put another instance variable in the TreeNode class in addition to itsLeft, itsRight, and itsData:

```
   private TreeNode itsParent = null;  // add to TreeNode
```

Whenever you consider storing additional information in an object, you should consider two things: What do you gain and what do you lose? This is like decisions in a business venture: What is your added revenue and what are your added costs? You need to make sure you will be making a profit if you make a change.

What you gain by adding parent information is a faster implementation of the Iterator next method. None of the other methods in the BintMap class profit from having the parent information. What you lose is the space required to store the added information and the time required to update the information when needed.

Parent information changes whenever you add or remove a node in the binary tree. So for the put logic in Listing 17.6, set the parent of the root node to null and add the following statements directly after the two statements that mention new TreeNode (lines 16 and 22):

```
   this.itsLeft.itsParent = this;   // after line 16
   this.itsRight.itsParent = this;  // after line 22
```

For the remove logic in Listing 17.7, you need to change the record of the parent in the nodes directly below the one whose data you are changing. Specifically, you add these three lines right after deleting a data value whose right subtree is empty:

```
   if (this.itsData != null)    // after line 12
   {  this.itsLeft.itsParent = this;
      this.itsRight.itsParent = this;
   }
```

The only other change required in Listing 17.7 is that each node that is moved up into another node's position has to have its parent information corrected. So have the following statement after deleting the node to the right of this. Its new right child may be empty, but it does not matter what an empty node thinks its parent is:

```
   this.itsRight.itsParent = this;  // after line 16
```

Finally, put this statement after deleting a node to the left of p:

```
   p.itsLeft.itsParent = p;         // after line 23
```

The only advantage of having this parent information is that you may now make the nextIn method execute faster, by replacing everything after the first two lines by the following coding to handle the case when this.itsRight is empty:

```
   Node p = this;
   while (p.itsParent != null && p.itsParent.itsLeft != p)
      p = p.itsParent;
   return p.itsParent;
```

Implementing an Iterator with a threaded binary search tree

Adding a parent pointer as just described saves execution time, but even more time would be saved if we had an `itsNext` value stored in each node, instead of an `itsParent` value, where `itsNext` tells the node containing the very next value in sequence in the tree. Then we could replace the entire body of the `nextIn` method in Listing 17.8 by one statement:

```
   itsPos = itsPos.itsNext;  // new body of nextIn
```

On the other hand, we need to go to more trouble to adjust the `itsNext` values each time we add or remove a node in the tree. Specifically, we declare a new instance variable in the TreeNode class as follows (instead of having `itsParent`):

```
   private TreeNode itsNext = null;  // add to TreeNode
```

Then we assign it the correct value when we put a new data value in the tree. For the `put` logic in Listing 17.6, we have to make some adjustments around lines 16 and 22. The adjustment for line 16 is left as an exercise; the adjustment for line 22 is to add the following two statements after it:

```
   itsRight.itsNext = this.itsNext;  // after line 22
   this.itsNext = itsRight;
```

Some adjustment also has to be made for the `remove` logic in Listing 17.7, at lines 10, 16, and 20. Line 10 is left as an exercise. The adjustment for the other two is straightforward – just add the following statement at line 16 and also at line 19:

```
   this.itsNext = this.itsNext.itsNext;  // in removeData
```

<u>Historical Note</u> This kind of link is called a **thread** through the tree. Back in the days when RAM was expensive, it was standard to put the `itsNext` values and the `itsRight` values in the same storage space and just add a boolean value to each node to tell what kind of value was in that storage space (and similarly for an `itsPrevious` value sharing space with the `itsLeft` value). They would however only store something in `itsNext` when its right subtree was empty.

Exercise 17.26 Write statements that delete a TreeNode node `toDelete` from a binary tree when `toDelete` is not empty and `toDelete.itsRight` is empty and `toDelete.itsParent != null`. Assume you have the `itsParent` values.
Exercise 17.27 (harder) Write a private recursive TreeNode method so that you can replace everything in `nextIn` starting from `TreeNode lastLeftTurn` by the statement: `return ancestor.lastLeft (null, id);` and still do the same thing.
Exercise 17.28 (harder) Determine the amount of execution time saved by adding the parent information described in this section for a well-balanced tree.
Exercise 17.29* Essay: What are the advantages and disadvantages of having each MapIt iterator create a stack and put on it all of the TreeNodes at which the iterator made a left turn on its way to the current position `itsPos`? Compare this approach with using the `itsNext` instance variable.
Exercise 17.30** Revise the `put` logic at line 16 in Listing 17.6 to allow for the `itsNext` value.
Exercise 17.31** Revise the `remove` logic at lines 10 through 12 in Listing 17.7 to allow for the `itsNext` value.
Exercise 17.32** Implement the Iterator's `remove` method for Listing 17.8. Hint: You cannot simply call TreeNode's `removeData` method in Listing 17.7.

Part B Enrichment And Reinforcement

17.6 More Tree Traversals

If you want to put all the data values from a binary tree on a queue for later use, you have a choice of six standard depth-first traversal processes. You can process all the data in the left subtree either before or after all the data in the right subtree (left-to-right or right-to-left). Whichever of these two you choose, you can process the data in the root either before all subtree data (preorder traversal), or after all subtree data (postorder traversal), or in between the data of the two subtrees (inorder traversal).

Figure 17.6 shows three of these standard traversals. Start at the down-arrow at the top of one of the figures and follow the dotted line. Process each of the seven data values when you come to the little rectangle pointing to it. For the left-to-right preorder traversal, the order of processing is 17, 13, 11, 15, 22, 20, 25. For the left-to-right inorder traversal, the order of processing is 11, 13, 15, 17, 20, 22, 25. For the right-to-left postorder traversal, the order of processing is 25, 20, 22, 15, 11, 13, 17.

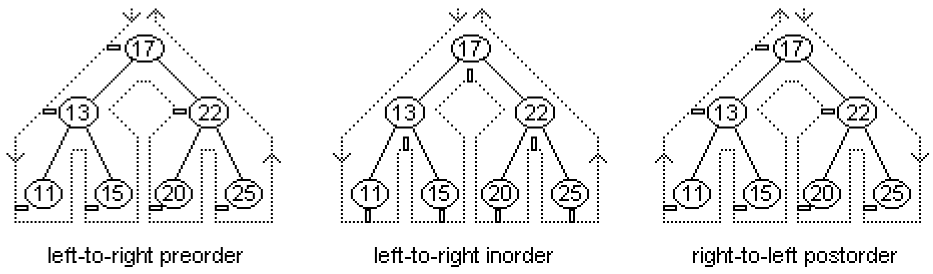


Figure 17.6 Three of the six standard traversals (start at the down-arrow)

The following right-to-left postorder traversal method, illustrated on the right side of Figure 17.6, could be in the TreeNode class, with the parameter being any QueueADT object to which you want to add all the data values:

```
public void postorderTraverseRL (QueueADT queue)// in TreeNode
{  if (isEmpty())
      return;
   itsRight.postorderTraverseRL (queue);
   itsLeft.postorderTraverseRL (queue);
   queue.enqueue (itsData);
}  //=======================
```

There are five more analogous methods, one for each kind of standard depth-first traversal. Note that the queue produced by a preorder traversal of a binary search tree can be used to produce an exact copy of that tree, with the same parent-child relations.

The most commonly-used traversal of a binary tree is left-to-right inorder traversal (as shown in the middle of Figure 17.6). This produces the data values in the **natural order** (i.e., using `compareTo`) if the tree is a binary search tree. An iterator could be made for BintMap using this traversal: When the iterator is constructed, it traverses the entire tree recursively and puts the data values on a queue; each time `next` is called, the next data value is removed from the queue.

The coding for this **QueueMapIt** iterator class for BintMap is in Listing 17.9 (see next page). Note that `next` does not need an explicit throw statement, since the queue will throw a RuntimeException if anyone tries to take a data value from it when it is empty.

Listing 17.9 The QueueMapIt class, providing iterators for BintMap

```
   public java.util.Iterator iterator()    // member of BintMap
   {  return new QueueMapIt (this.itsRoot);
   }  //======================


   private static class QueueMapIt implements java.util.Iterator
   {
      private QueueADT itsQueue = new NodeQueue();


      public QueueMapIt (TreeNode given)
      {  given.inorderTraverseLR (itsQueue);
      }  //======================

      public boolean hasNext()
      {  return ! itsQueue.isEmpty();
      }  //======================

      public Object next()
      {  return itsQueue.dequeue();
      }  //======================

      public void remove()
      {  throw new UnsupportedOperationException ("no remove!");
      }  //======================
   }



// The following method is added to the TreeNode class

   /** Add to the given queue all data values in the standard
    *  left-to-right inorder traversal. */

   public void inorderTraverseLR (QueueADT queue)
   {  if (isEmpty())
         return;
      itsLeft.inorderTraverseLR (queue);
      queue.enqueue (itsData);
      itsRight.inorderTraverseLR (queue);
   }  //=======================
```

This implementation of Iterators can be more efficient than the three kinds of implementations described in the preceding section. For one thing, you do not have to update parent pointers or threading information for each insertion or deletion of data. However, if users often progress only a little way through most iterations, you waste the time spent making a complete traversal to fill in the queue.

Yet another implementation of BintMap's iterator

An alternative for iterating in left-to-right inorder traversal order is to have each iterator keep track of all the nodes at which it made a left turn on the way down the tree to its current position. Those nodes, plus any nodes in the right subtrees of any of those "left-turn nodes", are all the nodes that it has not yet returned from the `next` method. If we keep the left-turn nodes on a stack, we can easily move back up the tree from one TreeNode to another. Listing 17.10 has the coding (see next page).

Listing 17.10 The StackMapIt class, providing iterators for BintMap

```
public java.util.Iterator iterator()    // member of BintMap
{  return new StackMapIt (this.itsRoot);
}  //======================


private static class StackMapIt implements java.util.Iterator
{
   private StackADT itsStack = new NodeStack();


   public StackMapIt (TreeNode given)
   {  pushAllLefties (given, itsStack);
   }  //======================

   public boolean hasNext()
   {  return ! itsStack.isEmpty();
   }  //======================

   public Object next()
   {  TreeNode pos = (TreeNode) itsStack.pop();
      pushAllLefties (pos.getRight(), itsStack);
      return pos.getData();
   }  //======================

   public void remove()
   {  throw new UnsupportedOperationException ("no remove!");
   }  //======================

   /** push all nodes from the "left edge" of this tree. */

   private void pushAllLefties (TreeNode node, StackADT stack)
   {  while ( ! node.isEmpty())
      {  stack.push (node);
         node = node.getLeft();
      }
   }  //======================
}
```

The key concept for this **StackMapIt** class is that, at any given time, the `next` method has yet to return the data values (a) in any node in the stack or (b) in any node in the right subtree of any node in the stack. When the iterator is created, it pushes onto the stack all the non-empty nodes down the "left edge" of the tree. When `next` is called, it pops the top node (which, of all the nodes on the stack, is the node furthest down the tree). It then pushes onto the stack all the non-empty nodes down the "left edge" of the popped node's right subtree. Finally, it returns the data in the popped node.

Exercise 17.33 Describe the effect of replacing "Left" by "Right" and vice versa throughout Listing 17.10.
Exercise 17.34 Write the method `public void preorderTraverseRL (QueueADT queue)` analogous to the method in the lower part of Listing 17.9.
Exercise 17.35* Write the other three methods analogous to `inorderTraverseLR`.
Exercise 17.36* Rewrite the entire Listing 17.9 to use a NodeStack (methods `push` and `pop` instead of `enqueue` and `dequeue`) instead of a NodeQueue: Construction of an iterator puts all of the data values on a stack so that the `next` method produces left-to-right inorder traversal and has only one statement.
Exercise 17.37* Revise the constructor in Listing 17.9 to obtain an iterator that goes through the tree in breadth-first order. Use a modification of the logic in Listing 17.3.

17.7 Red-Black And AVL Binary Search Trees

One problem with binary search trees is that they may become unbalanced as values are put into them and taken out of them. We define a **full tree** to be a tree where every node with less than two subtrees is on the lowest level (and thus is a leaf with no subtrees). So the full tree with 3 nodes has 2 levels; the full tree with 7 nodes has 3 levels; and the full tree with 15 nodes has 4 levels. In general, a full tree with 2^k - 1 nodes has k levels. Figure 17.4 had an example of the full tree with 31 nodes. Figure 17.7 shows full trees with 1 node, 3 nodes, 7 nodes, and 15 nodes.

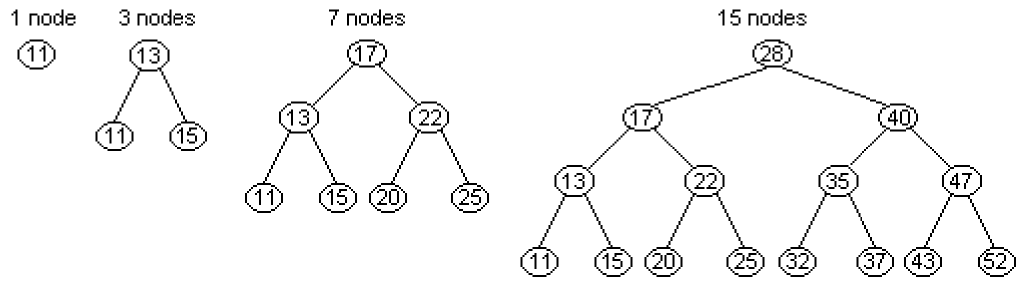


Figure 17.7 Four full trees

This book defines a tree to be **near-full** if it has the minimum possible number of levels for the number of data values in the tree. For instance, any tree with at least 4 data values has to have at least 3 levels; any tree with at least 8 data values has to have at least 4 levels; and any tree with at least 16 data values has to have at least 5 levels. In general, a tree with anywhere from 2^{k-1} data values up to and including 2^k-1 data values is near-full if it has only k levels.

In a near-full tree, the worst-case execution time to find a single value (execution time for `containsKey` or `get`) is big-oh of log(N). The reason is that the search process requires looking at only one value on each level of the binary tree. For example, since log2(32) is 5, if N is in the range from 16 to 31, log2(N) ranges from 4 to 5, and a near-full tree with N data values has 5 levels. So it takes at most 5 comparisons to find a key you are searching for. In general, you need at most log2(N+1) comparisons to find a data value in a near-full tree with N data values. By contrast, a badly-balanced tree could require up to N comparisons. Reminder: This book uses **log2(x)** to mean the number of times you have to halve x (i.e., execute `x /= 2.0`) to get 1 or less. In Java coding, log2(x) is the same as Math.ceil (Math.log(x) / Math.log(2)).

The **average search time** for a non-empty data structure is the total search time divided by the number of data values in the data structure. The total search time is the number of data values you have to look at before you find D, summed over all data values D in the data structure.

Example: The average search time for the first three full trees in Figure 17.7 is 1/1 for the first, 5/3 for the second, and 17/7 for the third. The 17 in the value 17/7 is the sum of 1 for the root node, 2 for each of the 2 second-level nodes, and 3 for each of the 4 third-level nodes: 1 + 2*2 + 3*4 = 17. A probabilistic analysis would show that, if you insert a random sequence of data values into a binary search tree, the average search time is 1.38 * log2(N) where N is the number of data values in the tree.

Balancing an existing binary search tree

If you have any Mapping object that contains mutually Comparable objects in ascending order, you can create a new BintMap object with the same data values in the same order but in a near-full tree. The BintMap constructor in Listing 17.11 (see next page) does this.

Listing 17.11 The balancing constructor in BintMap, with a method in TreeNode

```
   /** Precondition:  par has MapEntries in ascending order. */

   public BintMap (Mapping par)
   {  itsRoot = (par == null || par.isEmpty())               //1
            ? TreeNode.ET                                     //2
            : TreeNode.balanced (par.size(), par.iterator());//3
   }  //======================

   /** Return a near-full binary tree with the same ordering.
    *  Precondition:  it has at least numNodes >= 1 values.  */

   public static TreeNode balanced (int numNodes, Iterator it)
   {  if (numNodes == 1)                                      //4
         return new TreeNode (it.next());                     //5
      TreeNode leftSide = balanced (numNodes / 2, it);        //6
      TreeNode root = new TreeNode (it.next());               //7
      root.itsLeft = leftSide;                                //8
      int n = numNodes – 1 - numNodes / 2;                    //9
      root.itsRight = (n == 0) ?  ET  :  balanced (n, it);    //10
      return root;                                            //11
   }  //======================
```

If for instance the given Mapping object has 20 or 21 data values, the `balanced` method iterates through the first 10 data values and constructs a near-full BintMap named `leftSide`. Then it makes a tree with the next (eleventh) data value in its root and `leftSide` as its left subtree. Finally, it iterates through the remaining 9 or 10 data values and puts them in a near-full BintMap as the right subtree of the root.

AVL trees

It would be best if each call of `put` or `remove` for a binary search tree would check the tree to be sure that the change it makes does not make the tree too far out of balance. If it would go too far out of balance, the method should make a small adjustment (on the order of big-oh of 1). This can be done, but it requires a somewhat looser concept of balance: This book defines a **decently-balanced binary tree** to be one with not more than twice as many levels as a near-full tree of the same number of nodes.

Some computer scientists named Adelson-Velskii and Landis figured out a way to do this, so the result is called an AVL tree. They add one more instance variable to the TreeNode class. This number is the number of levels in the left subtree of a node minus the number of levels in the right subtree of that same node:

```
   private int leftMinusRight = 0;
```

The objective of the AVL algorithm is to keep this `leftMinusRight` number equal to 0, 1, or -1 for every node in the tree at all times. A binary search tree for which this property is true is called an **AVL tree**. So each time `put` or `remove` is called, it must see whether any node has had this value changed to an unacceptable value and, if so, make some adjustment to bring the tree back to having the AVL property.

When `put` is called for some AVL tree, it adds one node. It should be clear that the `leftMinusRight` number only changes for nodes that are on the path from the root down to the added node. If the addition of a node X loses the AVL property, consider the node P that is furthest down on the path from the root to X and has the wrong `leftMinusRight` number. That number will clearly be 2 if X is in the left subtree of P and will be -2 if X is in the right subtree of P.

To restore the AVL property, perform a **rotation** (which consists of moving a node from the side of P with more levels to the side with fewer levels). In this description and in Figure 17.8, C denotes the child of P and G denotes C's child on the side of C that has more levels (the "grandchild" G will be X or an ancestor of X):

1. Move node G to the other side of P (opposite from C) to be P's other child.
2. Swap the data in G, P, and C around to get them in increasing order left to right.
3. Attach the four subtrees of G, P, and C to those three nodes in the way that keeps their original order left to right.

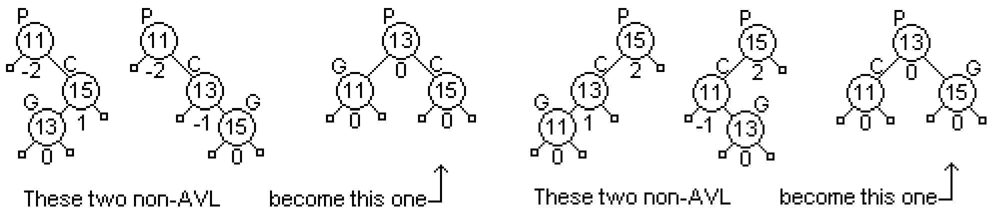


Figure 17.8 Rotations to correct an AVL tree, showing leftMinusRight values

The coding is left as a major programming problem. The next section proves logically that an AVL tree (also known as a **height-balanced tree**) is decently-balanced. The **height** of a binary tree is the number of nodes on the longest path from the root to a leaf. It is therefore the number of levels in the binary tree. But first we discuss another kind of tree, a red-black tree. We will also prove logically that a red-black tree is decently-balanced (so the worst-case execution time for a search is never more than twice as much as in a near-full tree with the same number of nodes).

Red-black trees

A **red-black tree** is a binary search tree in which each node is considered to be colored either red or black. We add a new instance variable to each node, as follows:

```
   private boolean isRed = true;
```

This says that each node is created red by default; we change its color to black when needed (by executing `isRed = false`). A red-black tree must satisfy three properties:

1. The root node is black, and external nodes are considered to be black.
2. No red node is the child of a red node.
3. The number of black nodes on each path from the root to some external node is the same as on any other path from the root to some external node.

The first data value we put into a red-black tree is black, since it goes at the root. Each additional data value we put in goes in a red node X. There are three possible cases:

1. If X is the child of a black node, the tree maintains the three red-black properties.
2. If X is the child of a red node whose parent P has a black node on the other side, then perform a rotation as described previously for AVL trees.
3. If X is the child of a red node whose parent P has a red node on the other side, then color both children of P black. If P is not the root node of the whole tree, then color P red and apply this same logic (steps 1 through 3) to P in place of X.

Figure 17.9 illustrates what happens if you add one data value at a time to an AVL tree or a red-black tree, each value larger than the one before. When the 15 is added, it makes the tree unacceptable, which causes a rotation: The node containing 15 becomes the left child of the root node. Then the data values are moved around: 15 where 13 was, 13 in the root, and 11 to the left of the root.

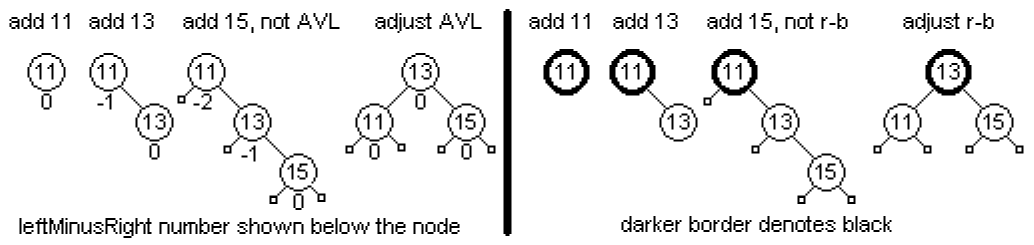


Figure 17.9 Adding ever larger values to an AVL tree and to a red-black tree

If you then added 17 to either of the balanced 3-node trees in Figure 17.9, it would be acceptable on the right of the 15-node in the AVL tree, but the red-black tree would require that you change the color on the 11-node and the 15-node to black.

Coding the red-black insertion algorithm

A partial coding of this algorithm for red-black trees is in Listing 17.12. The line numbers from the `put` logic in Listing 17.6 have been kept so that you can see that there are only two kinds of changes: (a) Lines 17, 20f, 23, and 26f return `ET` instead of null to signal the problem that the current node is red and has a red child; (b) Lines 20a-20e and lines 26a-26e call special private methods `fixLeftRedRed` and `fixRightRedRed` to correct the problem when the current node is a black node with a red child that itself has a red child. The `putRecursive` coding is the same as in Listing 17.6.

Listing 17.12 The put logic in the TreeNode class, revised for red-black trees

```
private Object putInLeftSubtree (Comparable id, Object val)
{  if (itsLeft.isEmpty())                              //15
   {  itsLeft = new TreeNode (new MapEntry (id, val));   //16
      return this.isRed ? ET : null;                   //17'
   }                                                   //18
   else                                                //19
   {  Object e = itsLeft.putRecursive (id, val);        //20'
      if ( ! this.isRed && e == ET)                    //20a
      {  fixLeftRedRed();                              //20b
         return null;                                  //20c
      }                                                //20d
      else                                             //20e
         return this.isRed && itsLeft.isRed ? ET : e;  //20f
   }                                                   //20g
}  //======================

private Object putInRightSubtree (Comparable id, Object val)
{  if (itsRight.isEmpty())                             //21
   {  itsRight = new TreeNode (new MapEntry (id, val));  //22
      return this.isRed ? ET : null;                   //23'
   }                                                   //24
   else                                                //25
   {  Object e = itsRight.putRecursive (id, val);       //26'
      if ( ! this.isRed && e == ET)                    //26a
      {  fixRightRedRed();                             //26b
         return null;                                  //26c
      }                                                //26d
      else                                             //26e
         return this.isRed && itsRight.isRed ? ET : e; //26f
   }                                                   //26g
}  //======================
```

These two private methods are left as exercises. For instance, `fixLeftRedRed()` fixes the situation where the problem is on the left of the executor. The overall logic is:

```
if (itsLeft.itsLeft.isRed && itsLeft.itsRight.isRed)
   // make it red but make its two children black
else if (itsLeft.itsLeft.isRed)
   // rotate that grandchild to itsRight and readjust
else  // itsLeft.itsRight is red
   // rotate that other grandchild to itsRight and readjust
```

This requires one other change: The `put` method at the top of Listing 17.6 must make the root black again if it became red. This is an exercise too. The `remove` method for red-black trees is left as a major programming project.

When the executor of `putInLeftSubtree` is a red node and the key is not already in the tree, then that executor acts as follows: If its left subtree is empty, it creates a new red node to its left for the data and returns ET, which signals a problem. If its left subtree is not empty, it asks the black node Z on its left to insert the data and then, if Z turns red as a consequence, returns ET to signal a problem. Red nodes do not fix problems.

When the executor of `putInLeftSubtree` is a black node and the key is not already in the tree, then that executor acts as follows: If its left subtree is empty, it creates a new red node to its left for the data and returns null, which signals "no problem". If its left subtree is not empty, it asks the node Z on its left to insert the data; if Z returns ET (signaling a problem) it fixes the problem and returns null, otherwise it returns what Z returned. But fixing the problem may include turning itself red, which might cause a problem for its parent.

Exercise 17.38 Calculate the average search time for the 15-node full tree.
Exercise 17.39 Draw or precisely describe the trees of 4 or 5 nodes that `makeBalanced` produces.
Exercise 17.40 Draw or precisely describe the trees of 6 or 7 nodes that `makeBalanced` produces.
Exercise 17.41 (harder) Revise the `put` method in the upper part of Listing 17.6 to make the root node black after each value is added to the tree.
Exercise 17.42 (harder) If you put five data values into an AVL tree, each larger than the one before, what does the tree look like? What does it look like if you then add one more data value larger than all the others?
Exercise 17.43 (harder) If you put five data values into a red-black tree, each larger than the one before, what does the tree look like? What does it look like if you then add one more data value larger than all the others?
Exercise 17.44* If you put seven data values into an AVL tree, each larger than the one before, what does the tree look like? What does it look like if you then add one more data value larger than all the others? Hint: Continue Exercise 17.42.
Exercise 17.45* If you put seven data values into a red-black tree, each larger than the one before, what does the tree look like? What does it look like if you then add one more data value larger than all the others? Hint: Continue Exercise 17.43.
Exercise 17.46* Write a recursive TreeNode method `public int height()`: The executor returns the height of its subtree; it returns 0 if empty.
Exercise 17.47* Write a recursive TreeNode method `public int howFar()`: The executor tells the shortest distance to an external node (it is 1 if either subtree is empty).
Exercise 17.48** Write the `fixLeftRedRed` and `fixRightRedRed` methods called by Listing 17.12 (they are identical except for swapping "Left" and "Right").
Exercise 17.49** Write the `fixLeftRedRed` and `fixRightRedRed` methods called by Listing 17.12, but instead of actually moving the grandchild node, create a new node on the other side of P from C and later discard the grandchild node. This takes fewer statements than the method described in the text. Is it more efficient?

17.8 Inductive Reasoning About Binary Trees

The definition of a binary tree is recursive, so most of the logic for binary trees is also recursive. Let us review the definition of a binary tree (parts 1 and 2 are the reason for having ET as the external node instead of using null to represent an external node):

1. There are two kinds of binary trees: empty ones and non-empty ones.
2. An empty binary tree has no data and no subtrees.
3. A non-empty binary tree has one data value and two subtrees (which are also binary trees), called its left and right subtrees.

Each binary tree corresponds in a fundamental way to its root node, which is external if the tree is empty and internal if the tree is not empty.

An example of recursive logic for binary trees is the proof that, **in every binary tree X, the number of external nodes is one more than the number of internal nodes**:

1. Basis Step If X is empty, then X has one external node (its root node) and zero internal nodes, so the difference is 1 as the assertion says.
2. Inductive Step Say X is some non-empty tree and say we have already proven the assertion for every tree with fewer nodes than X. Then each of X's two subtrees has 1 more external node than internal nodes, since each has fewer nodes than X. So together X's two subtrees have 2 more external nodes than internal nodes. Since the root node is internal, that brings the difference for X down to 1 more external node than internal nodes.
3. Conclusion The truth of the assertion for all binary trees follows by the Induction Principle from the Basis Step and the Inductive Step.

When some people first see this kind of logic, they think that it is circular logic, since we prove something about a non-empty tree X by assuming we have already proven it about each of its subtrees. But this is not circular logic, it is **inductive logic**, because each of the subtrees of X has fewer data values than X has. The Basic Step of the proof shows that the assertion is true for trees with 0 data values (empty trees). The Inductive Step of the proof shows that the assertion is true for any tree X with a positive number of data values as long as the assertion is true for all trees that have fewer data values than X has. **So for any n > 0, the Inductive Step deduces that the assertion is true for any n-node tree from the assertion being true for all trees with less than n nodes.**

How do you know that the assertion is true for any binary tree X? Because it could be proven for each number of data values 0, 1, 2, 3, 4, 5, etc. in that order:

- If X has 0 data values, the Basic Step shows the assertion is true for X.
- If X has 1 data value, then its subtrees both have 0 data values, so the assertion is true for each of its subtrees, so the Inductive Step shows the assertion is true for X.
- If X has 2 data values, then each of its subtrees has at most 1 data value, so the two preceding points show that the assertion is true for each of its subtrees, so the Inductive Step shows the assertion is true for X.
- If X has 3 data values, then each of its subtrees has at most 2 data values, so the three preceding points show that the assertion is true for each of its subtrees, so the Inductive Step shows the assertion is true for X.
- If X has 4 data values, then each of its subtrees has at most 3 data values, so the four preceding points show that the assertion is true for each of its subtrees, so the Inductive Step shows the assertion is true for X.
- If X has 5 data values, then each of its subtrees has at most 4 data values, so the five preceding points show that the assertion is true for each of its subtrees, so the Inductive Step shows the assertion is true for X.
- etc.

Minimum size of a red-black tree

We define the **black-height** of a red-black tree to be the number of black nodes on each path from the root to an external node. Figure 17.10 shows a red-black tree with a black-height of 3. We can prove that **every non-empty red-black tree X with black-height k has at least 2^k-1 internal nodes** using induction on k as follows:

1. Basis Step If X has black-height 1, it has at least 1 node, and 1 is at least 2^1-1.
2. Inductive Step Say X has black-height h+1 where h is 1 or more, and say we have already proven the assertion for each red-black tree with black-height h. We could then prove the assertion for X as follows: The tree has a black root node R with two children. A black child of R is the root of a red-black tree with black-height h, and a red child of R contains two subtrees that are red-black trees with black-height h. So the whole tree contains at least two separate subtrees each with at least 2^h-1 nodes, besides the root itself, and therefore has at least 2*(2^h-1)+1 nodes, which is 2^{h+1}-1.
3. Conclusion The truth of the assertion for all non-empty red-black trees follows by the Induction Principle from the Basis Step and the Inductive Step.

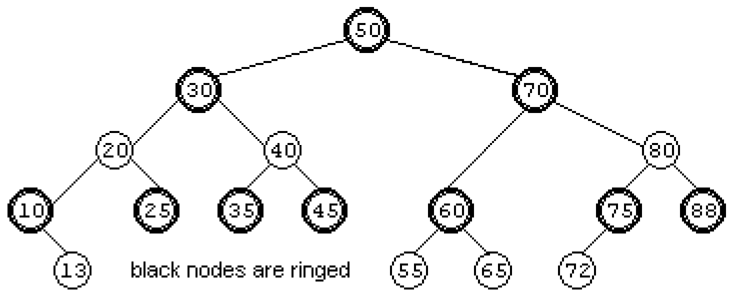


Figure 17.10 A red-black tree with black-height 3 and 5 total levels.

Worst-case execution time for a search of a red-black tree

Say you have a red-black tree with a positive number of nodes N. Let k denote the black-height of the tree. Then the preceding proof showed that N is 2^k-1 or more, so log2(N+1) is k when N is 2^k-1 and is at least k+1 otherwise. This tree with black-height k has at most 2*k levels, since you cannot have two red nodes in a row. And in the special case when N is exactly 2^k-1, the tree has only k levels. Now 2*log2(N+1)-1 is 2*k-1 in the special case (when the number of levels is k) and is at least 2*k+1 otherwise (when the number of levels cannot be more than 2*k). Conclusion: The number of levels, and therefore the worst-case time for a search in a red-black tree, is at most 2*log2(N+1) - 1. For instance, if N is 16 to 31, then log2(N+1) is 5 and so the worst-case search time is 2 * 5 - 1 = 9; the red-black tree cannot have 10 levels.

Worst-case execution time for a search of an AVL tree

We define **size(h)** to be the minimum possible number of data values in an AVL tree with h levels. An AVL tree with 1 level has 1 data value, so size(1) is 1. An AVL tree with 2 levels must have either 2 or 3 data values, so size(2) = 2 (the minimum).

If an AVL tree has h levels and h is more than 2, then one of its root's two subtrees must logically have h-1 levels, though the other subtree could have either h-1 or h-2 levels (not less, otherwise it would not have the AVL balance property). So the minimum possible number of data values in the subtree with level h-1 is size(h-1) and the minimum possible in the other subtree is size(h-2). Therefore, the minimum possible number of data values in the whole AVL tree is 1 + size(h-1) + size(h-2). Clearly, size(h-1) is larger than size(h-2). So it follows that size(h) is larger than 1 + 2 * size(h-2). This fact can be used to prove inductively that **size(2*k) >= 2^k for every positive k**:

1. Basic Step size(2) = 2 implies that size(2*1) >= 2^1, so the assertion is true for k==1.
2. Inductive Step To show that size(2*k) >= 2^k in situations where k > 1 and we know that the formula is true for k-1, i.e., that size(2*(k-1)) >= 2^{k-1}, we reason as follows: size(2*k) > 1 + 2 * size((2*k)-2) = 1 + 2 * size(2*(k-1)) > 2 * 2^{k-1} = 2^k.
3. Conclusion The truth of the assertion for all positive integers k follows by the Induction Principle from the Basis Step and the Inductive Step. For instance:

- The Basic Step showed that size(2*1) >= 2^1.
- size(h) > 2 * size(h-2) implies that size(4) >= 2 * size(2), i.e., size(2*2) >= 2^2.
- size(h) > 2 * size(h-2) implies that size(6) >= 2 * size(4), i.e., size(2*3) >= 2^3.
- size(h) > 2 * size(h-2) implies that size(8) >= 2 * size(6), i.e., size(2*4) >= 2^4, etc.

To find the longest path from the root to an external node in an AVL tree with N data values, define k to be log2(N+1), so that N < 2^k. Since we have proven that size(2*k) is at least 2^k, the AVL tree must have less than 2*k levels, thus less than 2*log(N+1) levels. So the worst-case search time in an AVL tree is 2*log2(N+1) - 1. For instance, if N is 16 to 31, then log2(N+1) is 5 and so the worst-case search time is 2 * 5 - 1 = 9; the tree cannot have 10 levels. This is the same result as for red-black trees.

Exercise 17.50 Calculate size(3) and size(4). Justify your answer.
Exercise 17.51 (harder) There is 1 binary tree with 1 data value; there are 2 binary trees with 2 data values and 5 binary trees with 3 data values. Calculate the number of binary trees with 4 data values and the number with 5 data values. Justify your answer.
Exercise 17.52 (harder) There are 4 red-black trees with a black-height of 1. Calculate the number of red-black trees with a black-height of 2. Justify your answer.
Exercise 17.53* Define rbsize(h) to be the minimum possible number of data values in a red-black tree with h levels, so rbsize(2) is 2 and rbsize(4) is 6. Find rbsize(6), rbsize(8), and rbsize(10). Find a formula for rbsize(2*n). Also calculate size(4), size(6), size(8), and size(10). Which kind of tree can be more unbalanced for a given number of levels, red-black or AVL?
Exercise 17.54* Prove by induction on the number of nodes that left-to-right inorder traversal of a binary search tree produces the data values in ascending order (each less than or equal to the next).
Exercise 17.55* Prove by induction on the number of nodes that size(2*k) >= 2^{k+1} - 2 for k >= 1.
Exercise 17.56* Prove by induction on k that a full tree with k levels has 2^k - 1 nodes.
Exercise 17.57** Prove by induction on h that when h>= 7, size(h) <= 1.66*size(h-1) and size(h) >= 1.60*size(h-1). Can you discover its relation to the golden mean 1.61803...?
Exercise 17.58** Explain why every height-balanced tree can be colored to make it a red-black tree that meets the conditions required, but not every red-black tree is height-balanced.
Exercise 17.59*** Write a method that calculates the number of binary trees with n data values for any given positive int n. Hint: See Exercise 17.51. Store intermediate results in an array (this use of an array, here and in the next exercise, is called dynamic programming).
Exercise 17.60*** Write a method that calculates the number of red-black trees with a black-height of n for any given positive int n. Hint: See Exercise 17.52. Store intermediate results in an array.

17.9 2-3-4 Trees And B-Trees

In a drawing of a red-black tree, draw a big circle around each internal black node so that the big circle includes its red subtrees (of which there are either 0 or 1 or 2). So each big circle has either 2 or 3 or 4 subtrees pointing down from it. Define a class of objects that represent these big circles in an implementation of Mapping: up to 3 data values and up to 4 subtrees. You could define such an object class as follows:

```
public class BigCircle
{  private final int MAX = 4; // maximum number of subtrees
   private MapEntry [] itsData = new MapEntry[MAX - 1];
   private BigCircle[] itsSubtree = new BigCircle[MAX];
}
```

We store the 1 or 2 or 3 data values of one big circle in increasing order in the MapEntry array, leaving the unused components equal to null. That means:

- If a black node X has 0 red subtrees, fill in only `itsData[0]` with the black node's data. Make `itsSubtree[0]` and `itsSubtree[1]` refer to the two subtrees in order left to right, except use null if the subtree is empty.
- If the black node X has a black left subtree and a red right subtree, fill in `itsData[0]` with X's data and `itsData[1]` with the red node's data. Make `itsSubtree[0]` the left subtree of X, `itsSubtree[1]` the left subtree of the red node, and `itsSubtree[2]` the right subtree of the red node.
- If the black node X has a red left subtree and a black right subtree, fill in `itsData[0]` with the red node's data and `itsData[1]` with X's data. Make `itsSubtree[0]` the left subtree of the red node, `itsSubtree[1]` the right subtree of the red node, and `itsSubtree[2]` the right subtree of X.
- If the black node X has two red subtrees, fill in `itsData[0]` with the left red node's data and `itsData[1]` with X's data and `itsData[2]` with the right red node's data. Make `itsSubtree` contain the four subtrees of the red nodes in the proper order.

Now we write coding to do exactly what the red-black tree does to implement a Mapping, except we use the array representations of BigCircles instead of the left-right representations of nodes. Each of the BigCircle objects contains either 2 or 3 or 4 subtrees, so this implementation of a Mapping is called a **2-3-4 tree**. Note that it is not a binary tree, because a BigCircle does not have just two subtrees and because it has more than one data value. Figure 17.11 contains an example of a 2-3-4 tree with three levels, the exact analog of the red-black tree in the earlier Figure 17.10.

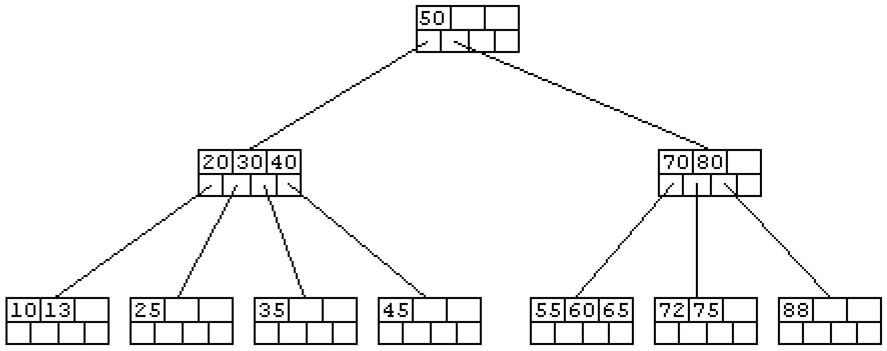


Figure 17.11 A 2-3-4 tree equivalent to the red-black tree of Figure 17.10

The get method for 2-3-4 trees

The `get` Mapping method is to search through the data structure for the MapEntry containing a given `id` and return the corresponding value. It is to return null if the `id` is not in the data structure. The coding of that `get` method for the BigCircle implementation could be as shown in Listing 17.13. The completion of this BigCircle class is left as a major programming problem.

Listing 17.13 The get method for the BigCircle class of 2-3-4 trees

```
public class BigCircle implements Mapping
{
   private final int MAX = 4;  // maximum number of subtrees
   private MapEntry[] itsData = new MapEntry[MAX - 1];
   private BigCircle[] itsSubtree = new BigCircle[MAX];

   /** Return the value for this id, or null if not there.
    *  Throw a RuntimeException if id is not Comparable. */

   public Object get (Object id)
   {  int k = 0;
      for ( ;  k < MAX - 1 && itsData[k] != null;  k++)
      {  int comp = ((Comparable) id).compareTo
                                (itsData[k].getKey());
         if (comp <= 0)
            return (comp == 0)  ?  itsData[k].getValue()
                   : (itsSubtree[k] == null)  ?  null // a leaf
                    : itsSubtree[k].get (id);
      }
      return (itsSubtree[k] == null)  ?  null         // a leaf
             : itsSubtree[k].get (id);
   }  //======================
}
```

Another data structure sometimes used for a Map is a **2-3 tree**, which is halfway between a binary search tree and a 2-3-4 tree: Each node has 1 data value and 2 subtrees or else 2 data values and 3 subtrees, with the same ordering properties as for 2-3-4 trees, and all leaves are on the same level.

B-trees

When an operating system reads information from a data file, it normally gets the information in chunks of a particular number of bytes, usually a power of two. For instance, an operating system may read in chunks of 4096 bytes. Even if you only want a few hundred bytes of information, each read operation gets 4096 bytes.

When you are reading information for a Mapping object from a file, it is most efficient if you can use most or all of the 4096 bytes you will get from each read operation. To this end, you could have BigCircle objects with lots more than 4 subtrees. Make them large enough that they will barely fit within 4096 bytes.

Say you know that each record (one object's information) is 190 bytes long. A reference to a subtree takes up 4 bytes of space, for a total of 194 bytes. 194 divided into 4096 is 21. So you could have as many as 21 data values in each BigCircle object, with the data values in increasing order of key values. You would thus have up to 22 subtrees. Then the `get` method for these really big circles could be the same as the one in Listing 17.13, except that MAX would be 22 instead of 4. Even better, you could use binary search to find the desired index.

This data structure is called a B-tree of order 22. In general, a **B-tree of order N** has up to N subtrees in each node for some N, always with one less data value than it has subtrees. A B-tree has two additional restrictions: All leaf nodes must be on the bottom level, and all nodes except the root must have at least N/2 subtrees (rounded up if N is odd). A 2-3-4 tree is just a B-tree of order 4.

When you add a new data value to a B-tree structure, you put it in the appropriate leaf node on the bottom level. There will only be one such place it can go in, according to the restrictions on ordering. However, if the node where it goes is full (already has N-1 data values), there is no room for the additional data value. In that case, you split the full leaf node into two nodes, with at least N/2 data values in the first node, the next (middle) data value moved up into the parent node, and the rest of them in the second node. Both of these nodes become children of the parent node. Of course, if that makes the parent node full, you have to split again, possibly going as far as splitting the root node.

A B-tree of order 22 that organizes a data file with 400 million records (more than the population of the United States) would have at most nine levels: It would have at least 1 data value in the root, therefore at least 2 nodes on the second level with at least 11 subtrees of each. The third level would then have at least 22 nodes, so the fourth level would have at least 22*11 nodes, etc. Each level multiples by 11, so the ninth level would have at least $22*11^6$ nodes, about 39 million, with at least 10 data values in each of those leaf nodes. Since you would always keep the root node in memory, you can find a data value in only 8 disk accesses, instead of the 29 that even a near-full binary tree would require.

Indexing

A popular alternative is to use B-trees to hold an index to a random-access data file instead of the data itself. Only the keys are stored in the B-tree; the full data record is only in a disk file. Each leaf node contains the key for one record plus the file position telling where the full data record is in the data file. All keys are in the leaf nodes. In each non-leaf node, `itsData[k]` is the first key of `itsSubtree[k+1]`.

Example If keys are 10 bytes (e.g., Social Security Numbers), and file pointer values are 4 bytes, then you only need 14 bytes per record. Instead of the B-tree of order 22 just described, you would have room for 4096 / 14 = 292 subtrees in each "big circle" node. So you could use a B-tree of order 292 for the index. So six levels would store a minimum of 292 * 146 * 146 * 146 > 900 million keys and file pointers on the sixth level. Even allowing for the extra disk access to retrieve the record itself, it would take only six disk accesses to find a data record in the random-access file, assuming you keep the root node in memory at all times.

Exercise 17.61 How many 296-byte records could you store in a B-tree node if it is to barely fit into a disk block of 8192 bytes? Show why.
Exercise 17.62 (harder) How many disk accesses (at most) would it take to find a single data value in a B-tree of order 100 used to store ten million records? Show why.
Exercise 17.63* How many disk accesses (at most) would it take to find a single data value in a B-tree of order 200 used to store 60,000 records? Show why.
Exercise 17.64* Rewrite the BigCircle `get` method in Listing 17.13 to first look at the middle one of three data values in the BigCircle. Does this execute faster?

17.10 Data Flow Diagrams

A small manufacturing company wants to hire you to create software to automate some of their operations. A scenario of part of what this company does could go as follows:

A customer places an order. You verify that the information about the customer (address, references, preferences, etc.) is already in your files. You check with your accounting department that the customer's credit is good. You record the order, manufacture the product wanted, and ship it to the customer along with an invoice. The customer sends you a check in payment, which you use to pay investors' dividends.

The preceding paragraph is called a **use case**; it is a scenario that describes a possible sequence of events using the system you are to design. If you are to model the entire business in software, then the business is the system and everything outside the system is its **environment**. Figure 17.12 is a Data Flow Diagram that graphically shows the elements mentioned in this particular use case.

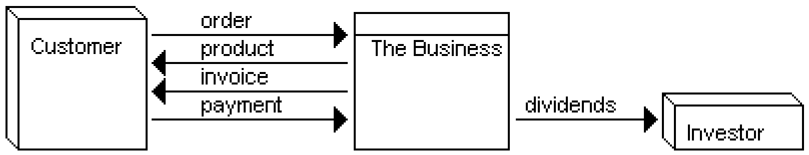


Figure 17.12 Data Flow Diagram for one use case

A **Data Flow Diagram** (**DFD**) has four basic kinds of elements. The first three in this list called the **nodes** of the diagram:

1. A **process**, which is part of the system to be modeled, shown as a rectangle divided horizontally. You put the name and/or description of the system part in the lower section. The middle node of Figure 17.12 is a process.
2. An **environment entity** that interacts with the system, shown as a 3D-box-like figure. You put the name of the entity on the front of the box. The nodes on the left and right of Figure 17.12 are environment entities.
3. A **data store**, which is part of the system to be modeled, shown as a rectangle divided vertically and with the right side containing its name: It stores a collection of information of a particular kind. No data stores appear in Figure 17.12.
4. A **data flow**, shown as an arrow between two nodes, of which one is almost always a process. A data flow indicates information flowing from one node to another. Its name is above or beside the arrow. Figure 17.12 has five data flows.

Two additional use cases for the manufacturing company are as follows:

You check the amount you have on hand of various parts you use to manufacture your product. You see that you are short on one kind of part. You place an order with a supplier. The supplier sends you boxes of parts and an invoice. You send a check to pay the invoice.

Investors contribute more capital to the operation, for which you issue stock. Every three months, you calculate your profits. Based on these calculations, you send a tax payment to the IRS and also dividend checks and a quarterly report to your investors.

When you look over these and other use cases, including entering a description of the customer (address, etc.) in your files, you might come up with the Data Flow Diagram in Figure 17.13 (see next page) to describe the major interactions of the company with its environment.

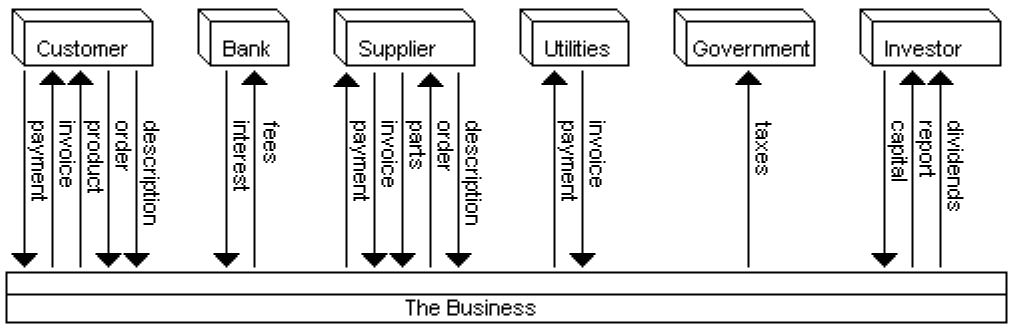


Figure 17.13 Data Flow Diagram for the entire business

The OrderHandling system

Your client wants you to implement in software only the part of the operation that handles orders from customers and orders to suppliers. So you can ignore some of the data flows described so far, such as dividends and payment of utility bills and taxes. Start with some examples of use cases for this OrderHandling system:

<u>Use Case 1</u> A person wants to order products from your company. You look up the name in your records and see that the customer is not recorded there. So you get the full description of the customer (name, address, references, etc.) and store it in your data store of customers. You then take the order and store it in your data store of orders. You check your inventory and see that the product is there. So you ship the product to the customer along with an invoice for payment, and at the same time notify the accounting department to expect payment since you have sent an invoice.

<u>Use Case 2</u> A person places an order. You look up the name in your records and retrieve the customer's information from your customer data store (since it was previously recorded). You then take the order and store it in your data store of orders. You check inventory and see that the product is not there. So you order it from the appropriate supplier (which in many cases will be a manufacturing department in the rest of the business, but treat it as a supplier for now).

<u>Use Case 3</u> A supplier ships parts to you along with an invoice. You store it in your inventory and notify the accounting department to send payment to the supplier.

<u>Use Case 4</u> A manufacturing department sends you finished product. You store it in your inventory.

<u>Use Case 5</u> You check your data store of orders not yet filled and find one for which you now have the required product. So you ship the product to the customer along with an invoice for payment, and at the same time notify the accounting department to expect payment since you have sent an invoice.

Further thought about the software and discussion with the client reveals that the customer will sometimes ask for the status of all of the customer's orders currently being processed, to verify its own records. In addition, the rest of the business operation may query the system from time to time for a list of all orders pending, of all orders filled in the past year, of inventory changes during the year, etc.

A very effective first step in the development of a software system is to describe its interactions with its environment, that is, with everything that is not the system to be developed. From this point of view, the rest of the business is part of the environment. These interactions usually take the form of input to the system from external entities and output from the system to external entities, both of which can be expressed as data flows.

Figure 17.14 is the only part of the overall operation of the company that concerns the OrderHandling system. Note that the system is split off from the rest of the business in this Data Flow Diagram. The names of the data flows have been made more precise: CustDescr is a description of one customer, SuppOrder is one order sent to a supplier, etc. The reporting functions have been added to the Data Flow Diagram. Minor "backflows" along certain arrows have been omitted, such as a confirmation message to the customer that the description or order has been received and a query for a report.

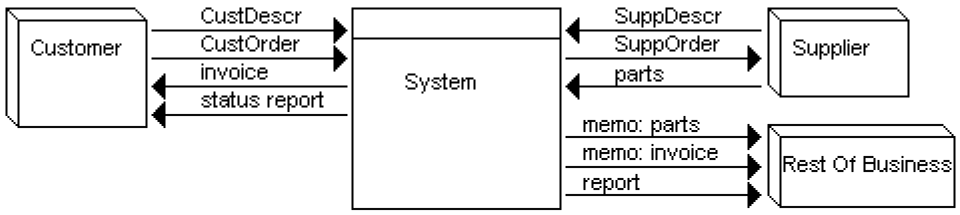


Figure 17.14 Data Flow Diagram for the OrderHandling software

Detailed Analysis

The next step in the analysis is to specify the exact form of the inputs and the outputs, the conditions under which they will occur, the sequence in which they occur, etc. This information fleshes out the DFD by saying what happens. Avoid saying how it happens; that is almost always a matter of design, so it comes later. Treat the system as a "black box" whose contents and workings are a mystery.

Analysis for this problem requires a full and precise description of the number and types of values entered, e.g., first and last name are strings of characters, the ZIP code is entered separately from the city and state, and the number of items ordered is a positive integer. It also requires a full and precise description of the form of the reports. This **Requirements Specification Document** should include examples and specifications for these things along with the Data Flow Diagram. Test data should also be developed at this point. We will not try to do it here for this problem.

Design

Now you start to divide up the proposed system into components. Design is a matter of deciding which components you want to have and how they will interact. For this OrderHandling software, some obvious choices are classes of CustomerOrder objects, SupplierOrder objects, Invoice objects, CustomerDescription objects, and SupplierDescription objects.

You also need a data structure for long-term storage of the list of CustomerDescription objects for all current customers, the list of CustomerOrder objects still pending, and the analogous lists of SupplierDescription and SupplierOrder objects. Reports are presumably simply long Strings of characters, and you already have a String object class.

The two main processes of the system are managing the customer relations and managing the supplier relations. These considerations lead to the Data Flow Diagram shown in Figure 17.15 (see next page). The abbreviations CustOrder, SuppOrder, CustDescr, and SuppDescr make the diagram more compact.

Number the process nodes 1, 2, 3, on up in whatever order you like, in the upper section of the node. Number data store nodes D1, D2, D3, on up also. We often append DS to the name to suggest a Data Store (also known as a Data Structure). Use these numbers for cross-referencing with more detailed Data Flow Diagrams and with narrative specifications.

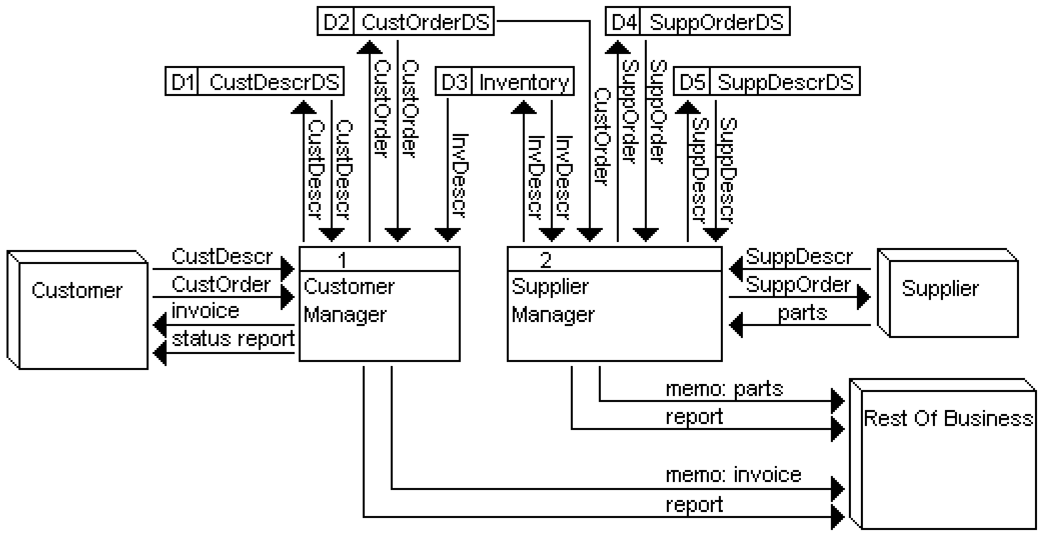


Figure 17.15 Data Flow Diagram for second level of OrderHandling

The middle and upper parts of Figure 17.15 are an **explosion** of the System node in the earlier Figure 17.14, since they are a more detailed replacement for that System node. You need to verify that all the data flows between the System node and its environment in Figure 17.14 are in the exploded view, and no others have been added. A close check shows that the ten data flows of the earlier figure are present in the later one, although "report" occurs twice in the later one (since the two separate processes have separate kinds of reports).

CRC cards

A useful device for a preliminary specification is a Class/Responsibilities/Collaborations card, a **CRC** card for short. You use an index card (no bigger than 4-by-6) on which you write the name of the class of objects at the top. Then you list on the left side of the card the major responsibilities of that class -- what messages it can be sent and what action it should take in response. You list on the right side of the card the collaborations of that class -- which other classes it sends messages to or otherwise uses.

The CustomerManager kind of object (process #1) has four major responsibilities:

1. Given a CustomerOrder object containing a CustomerID, see whether the CustomerDescription object is in the CustDescrDS. If not, create one from data the customer provides and store it in the CustDescrDS. Then store the CustomerOrder object in the CustOrderDS.
2. Search the CustOrderDS for orders not yet filled. For each one, check whether all products ordered are currently in the Inventory. If so, create an Invoice object, send it to the customer along with the product ordered, and send a memo of the Invoice to the accounts-receivable department.
3. Given a request from the customer for a status report, create and return it.
4. Given a request from the rest of the business for any one of several kinds of summary reports, create and return it.

This process #1 has therefore collaborations with three data stores (CustDescrDS, CustOrderDS, and Inventory), with three passive data flows (CustomerOrder, CustomerDescription, and InventoryDescription objects) and with the rest of the business. Now stubbed documentation for the CustomerManager class of objects can be developed as shown in Listing 17.14.

Listing 17.14 CustomerManager class of objects, stubbed documentation

```
public class CustomerManager           // stubbed
{
   /** Verify that the customer's description is on file.
    *  If not, obtain and file it.  Then file the order.  */

   public void storeOrder (CustomerOrder order)               { }


   /** Find all orders that are currently unfilled.
    *  Fill any such order for which all product is available. */

   public void fillAllPossibleOrders()                        { }


   /** Return a description of all orders from this customer
    *  that are currently unfilled. */

   public String getStatus (String customerID)   { return null; }


   /** Return a description of all orders currently unfilled,
    *  in order of customer ID. */

   public String getOrdersByID()                 { return null; }


   /** Return a description of all orders currently unfilled,
    *  in order of dates entered. */

   public String getOrdersByDate()               { return null; }


   // many other report functions would go here
}
```

You develop CRC cards by walking through the scenarios given by use cases and seeing which classes and operations a software system should have. A large number of people find that developing CRC cards is a far better way to develop an object design than data flow diagrams, since they have a concrete object (the 4-by-6 card) to help them anchor their thoughts. Other people prefer developing data flow diagrams. You need to attend a 2- or 3-hour workshop on the use of CRC cards in order to truly appreciate them. Reading about them would not adequately convey their advantages.

Exercise 17.65* Write out two more use cases for the OrderHandling software.

17.11 Review Of Chapter Seventeen

- A **general tree** has one or more data values and zero or more subtrees.
- A **binary tree** is either (a) empty, in which case it has no data and no subtrees, or (b) non-empty, in which case it has one data value and two subtrees. The data value is called its **root value**. The two subtrees are called its **left and right subtrees**.
- An **internal node** represents a non-empty binary tree and has a minimum of three attributes (instance variables): a data value stored there plus references to the root nodes of the left subtree and the right subtree. If both of the subtrees of an internal node are empty, it is called a **leaf node**. An **external node** has no data value and no subtrees; it represents the empty binary tree.
- A **path** in a tree from one node to another is a sequence of nodes beginning with the first and ending with the second such that, for each time node Y follows node X in the sequence, node Y is the root of a subtree of node X. A binary tree cannot have two different paths from one node to another, nor any path from a node to itself.
- **Breadth-first search** of a binary tree means that you first look at the data at the root, then at the data one step from the root, then at the data two steps from the root, etc. **Depth-first search** means that, after looking at the data in a particular node, you look at <u>all</u> the data in the subtree it is the root of before you look elsewhere.
- **Left-to-right preorder traversal** is a depth-first search that, after it looks at a node X, looks at all the nodes in the left subtree of X, then looks at all the nodes in the right subtree of X. "Traversal" means you visit each node in the tree (potentially). "Left-to-right" means that you visit all nodes in the left subtree before you visit any node in the right subtree; the alternative is right-to-left traversal. "Preorder" means that you visit the root node of each subtree <u>before</u> visiting any of the nodes in its subtrees. The alternatives to preorder traversal are **postorder traversal** (visit the root node <u>after</u> visiting all the nodes of the subtrees) and **inorder traversal** (visit the root node <u>in between</u> visiting the nodes of the subtrees).
- In a **binary search tree**, the Comparable keys have a special relationship: Every key in a left subtree is less than the key for the root and every key in a right subtree is greater than the key for the root. Some applications (not Mappings) allow several data values that are equal or the equivalent. In that case, a data value can be equal to another one in its right subtree.
- If a node X represents a non-empty binary tree, the nodes at the roots of the subtrees of X are called the **left child** and the **right child** of X. X is their **parent**.
- A tree is full if every node with less than two subtrees is on the lowest level. So the **full tree** with 3 nodes has 2 levels; the full tree with 7 nodes has 3 levels; and the full tree with 15 nodes has 4 levels. In general, a full tree with 2^k - 1 nodes has k levels. This book defines a tree to be **near-full** if it has the minimum possible number of levels for the number of data values in the tree. A binary tree can be considered to be **decently-balanced** if it does not have more than twice as many levels as a near-full tree of the same number of nodes.
- The **average search time** for a non-empty data structure is the total search time divided by the number of data values in the data structure. The total search time is the number of data values you have to look in order to find D, summed over all data values D in the data structure.
- An **AVL tree** is a binary search tree in which, for any node X, the number of levels in the right subtree of X differs from the number of levels in the left subtree of X by at most 1. An AVL tree is always decently-balanced.
- A **red-black tree** is a binary search tree in which the root node is black, no red node is the child of a red node, and the number of black nodes on each path from the root to some external node is the same as on any other path from the root to some external node. A red-black tree is always decently-balanced.
- A **B-tree of order N** has up to N subtrees in each node for some fixed N, always with one less data value than it has subtrees. A B-tree has two additional restrictions: All leaf nodes must be on the bottom level, and all nodes except the root and the leaves must have a minimum of 0.5*N subtrees. A **2-3-4 tree** is just a B-tree of order 4.

Answers to Selected Exercises

17.3 The fifth tree: A, B, C. The sixth tree: C, B, A. The seventh tree: A, B, C.

17.4 c.size() gets 0 from its call of c.itsLeft.size() and get 4 from its call of d.size(), so it returns 5.

17.5 b.deleteLastNode() would see that h.itsRight is not empty, so it would call h.deleteLastNode(), which would see that i.itsRight is not empty, so it would call i.deleteLastNode(), which would see that k.itsRight is empty, so it would set i.itsRight to be the empty tree.

17.6
```
public TreeNode lastNode()
{       return itsRight.isEmpty()  ?  this  :  itsRight.lastNode();
}
```

17.7 It is the same as deleteLastNode except interchange the words "Right" and "Left".

17.8 printInOrder executes in O(N) time, since it visits every node in the tree. deleteLastNode executes for the average binary tree in O(log(N)) time, since it navigates only one path down the tree.

17.11 A, C, G, F, B, E, D. A's left is C, C's left is G, A's right is B.

17.12 A, E, B, G, F, D, C. A's left is B, B's left is C, A's right is E.

17.13 The tree could have 7 levels with 1 node on each level. So no node has two children.

17.14 The level of a data value is equal to the difference between the number of left parentheses that come before it and the number of right parentheses that come before it. In other words, read the output from left to right, counting +1 for each left parenthesis and -1 for each right parenthesis, and when you come to a data value, the net count so far is the level of that data value.

17.17
```
public TreeNode lookUp (Comparable id)
{       int comp = compare (id, itsData);
        if (comp < 0)
                return itsLeft.isEmpty()  ?  null  :  itsLeft.lookUp (id);
        if (comp > 0)
                return itsRight.isEmpty()  ?  null  :  itsRight.lookUp (id);
        return this;
}
```

17.18 Insert the following just before the "else if" in line 14:
```
else if (itsLeft.isEmpty())
{       itsData = itsRight.itsData;
        itsLeft = itsRight.itsLeft;    // not the opposite order
        itsRight = itsRight.itsRight;
}
```
This executes faster on average only if there are a significant number of nodes in the tree with an empty left subtree and a right subtree of 2 or more levels. So it is slower if the tree is reasonably balanced, but faster if the tree is quite unbalanced.

17.19
```
public TreeNode copy()
{       TreeNode root = new TreeNode (itsData);
        root.itsLeft = itsLeft.isEmpty()  ?  ET  :  itsLeft.copy();
        root.itsRight = itsRight.isEmpty()  ?  ET  :  itsRight.copy();
        return root;
}
```

17.20
```
public int under (Comparable id)
{       return this.isEmpty()  ?  0
                  :  compare (id, itsData) <= 0  ?  itsLeft.under (id)
                          :  itsLeft.size() + 1 + itsRight.under (id);
}
```

17.26
```
if (toDelete.itsParent.itsLeft == toDelete)
        toDelete.itsParent.itsLeft = toDelete.itsLeft;
else
        toDelete.itsParent.itsRight = toDelete.itsLeft;
toDelete.itsLeft.itsParent = toDelete.itsParent;
```

17.27
```
private TreeNode lastLeft (TreeNode lastLeftTurn, Comparable id)
{       int comp = compare (id, itsData);
        return (comp == 0)  ?  lastLeftTurn
                  : (comp > 0)  ?  itsRight.lastLeft (lastLeftTurn, id)
                         : itsLeft.lastLeft (this, id);
}
```

17.28 In a well-balanced tree with N values, a node X with no right subtree should only be in one of the two bottom levels. The successor of X will tend to be roughly two levels above X, computed as follows: There is a 50% chance X's parent is its successor (i.e., X is to the left of its parent); a 25% chance that X's "grandparent" is its successor; a 12.5% chance that X's "great-grandparent" is its successor, etc. That averages out to X's successor being about two levels above X. X will also be log(N)-1 nodes below the root node on average. So the parent information allows next to find the successor by going up about 2 levels instead of down log(N)-1 levels, so we save about log(N)-3 comparisons. Since about half of the nodes will have no right subtree, that implies a total of (N/2) * (log(N)-3) comparisons saved in traversing N nodes in a well-balanced tree.

17.33 The iterator would traverse the tree in a right-to-left inorder traversal (i.e., backwards).

17.34
```
public void preorderTraverseRL (QueueADT queue)
{       if (isEmpty())
                return;
        queue.enqueue (itsData);
        itsRight.preorderTraverseRL (queue);
        itsLeft. preorderTraverseRL (queue);
}
```

17.38 The total search time is 17 for the top three levels (already calculated in the text for that full tree) plus 8 times 4 for the fourth level, a total of 49. So the average search time is 49/15.

17.39 Start with the full tree with 3 nodes. Add a left child of the left child of the root for the 4-node tree. Add also a left child of the right child of the root for the 5-node tree.

17.40 The 7-node tree is the full tree with 7 nodes shown in Figure 17.7. Omit the bottom-right node (which contains 25 in Figure 17.7) to get the 6-node tree.

17.41 Add a makeBlack method to TreeNode that has the one statement isRed = false; then replace the last line of the put method (line 7) by the following three statements:

```
Object valueToReturn = itsRoot.putRecursive ((Comparable) id, value);
itsRoot.makeBlack();
return valueToReturn;
```

17.42 To the full tree with values 11, 13, and 15 in Figure 17.9, add 17 to the right of the 15 to get the 4-node tree. Adding 19 forces a rotation to put 17 in place of 15, 15 to 17's left, and 19 to 17's right. That gives the 5-node tree. Adding 21 then forces a rotation of 17 into the root, 13 as 17's left child, 11 as 13's left child, 15 as 13's right child, 19 as 17's right child, and 21 as 19's right child.

17.43 To the full tree with values 11, 13, and 15 in Figure 17.9, add 17 to the right of the 15 and color 11 and 15 black to get the 4-node tree. Adding 19 forces a rotation to put 17 in place of 15, red 15 to 17's left, and red 19 to 17's right. That gives the 5-node tree. Adding 21 then turns the 19-node black, the 15-node black, and the 17-node red without forcing a rotation.

17.50 size(3) = 1 + size(2) + size(1) = 1 + 2 + 1 = 4. size(4) = 1 + size(3) + size(2) = 1 + 4 + 2 = 7.

17.51 4 data values could mean 3 on the left side of the root (5 cases), or 3 on its right side (5 cases), or 2 on the left side of the root (2 cases) or 2 on its right side (2 cases), a total of 14 cases.
5 data values could mean 4 on the left side of the root (14 cases) or on its right side (14 cases), or 3 on the left side of the root (5 cases) or on its right side (5 cases), or 2 on each side (2*2 = 4 cases), a total of 42 cases for the 5-node tree.

17.52 The root could have 2 black children, each rooting one of 4 red-black trees with a black-height of 1, which makes 4 * 4 = 16 cases. Or it could have one red child, on the left, which will have 16 possible subtrees, with 4 possible subtrees on the right, which makes 16 * 4 = 64 more cases. Or it could have one red child, on the right, for 64 more cases. Or it could have two red children, each with 64 possible subtrees, thus 64 * 64 = 4096 more cases. The total is 4240 cases.

17.61 296 + 4 = 300. 8192 / 300 = 27 with 92 left over, so you could store 27 records in each node.

17.62 The root could have only 1 record, but still the second level would have at least 2 nodes with at least 50 subtrees each, so the third level would have at least 100 nodes with at least 50 subtrees each, so the fourth level would have at least 5000 nodes with at least 50 subtrees each, so the fifth level would have at least 250,000 nodes with at least 50 subtrees each, which is over 12 million. Therefore four disk accesses would be enough, assuming you keep the root node in memory.