```
1
2   // BinaryTree class; stores a binary tree.
3   //
4   // CONSTRUCTION: with (a) no parameters or (b) an object to
5   //    be placed in the root of a one−element tree.
6   //
7   // ******************PUBLIC OPERATIONS*********************
8   // Various tree traversals, size, height, isEmpty, makeEmpty.
9   // Also, the following tricky method:
10  // void merge( Object root, BinaryTree t1, BinaryTree t2 )
11  //                        −−> Construct a new tree
12  // ******************ERRORS********************************
13  // Error message printed for illegal merges.
14
15
16  public class BinaryTree<T>
17  {
18
19      private Node root;   // A pointer to the root of the Binary Tree.
20
21      /*
22       *    Construct an empty Tree.
23       *
24       */
25
26      public BinaryTree( )
27      {
28          root = null;
29      }
30
31      /*
32       *    Construct a Binary Tree consisting of a single node.
33       */
34
35      public BinaryTree( T rootItem )
36      {
37          root = new Node(rootItem, null, null);
38      }
39
40
41      /*
42       *    makeEmpty() − delete all Nodes from the Tree.
43       */
44
45      public void makeEmpty( )
46      {
47          root = null;
48      }
49
50      /*
51       *    isEmpty() − return true if there are no Nodes
52       *                in the tree, else return false.
53       */
54
55      public boolean isEmpty( )
56      {
57          if(root==null){
58              return true;
59          }
60          return false;
61      }
62
63      /**
64       *   Merge routine for BinaryTree class.
65       *   Forms a new tree from rootItem, t1 and t2.
66       *   Does not allow t1 and t2 to be the same.
67       *   Correctly handles other aliasing conditions.
68       */
69
```

```
70      public void merge(T rootItem, BinaryTree<T> t1, BinaryTree<T> t2 )
71      {
72
73          if( t1.root == t2.root && t1.root != null )
74          {
75              System.err.println( "leftTree==rightTree; merge aborted" );
76              return;
77          }
78
79              // Allocate new node
80
81          root = new Node( rootItem, t1.root, t2.root );
82
83          if (t1.root != null)          // The roots of the subtree point
84              t1.root.parent = root;    // back to the parent node.
85          if (t2.root != null)
86              t2.root.parent = root;
87
88              // Ensure that every node is in only one tree
89
90          if( this != t1 )
91              t1.root = null;
92          if( this != t2 )
93              t2.root = null;
94      }
95
96
97
98      /*
99       *    printTree() − display the binary tree.
100      */
101
102     public void printTree()
103     {
104         printBT(root, 0);   // Print the binary tree starting at the root.
105     }
106
107     /*
108      *    printBT() − A recursive routine to print the Binary tree
109      *                with given root parameter at the given level.
110      */
111
112     private void printBT(Node root, int level)
113     {
114
115         if (root == null)    // Empty tree, nothing to print.
116             return;
117
118         printBT(root.rchild, level + 1);     // Print the right subtree.
119
120         for (int i = 0; i < level; i++)              // Print the root node
121             System.out.print("   ");                 // indented by an amount
122         System.out.println(""+root.data.toString()); // corresponding to its
123                                                      // level.
124
125         printBT(root.lchild, level + 1 );    // Print the left subtree.
126     }
127
128     /*
129      *   size() − Return the number of Nodes in the tree.
130      */
131
132     public int size()
133     {
134         return rsize(root);  // Call recursive function rsize().
135     }
136
137     /*
138      *   rsize() − A recursive function to return the
```

```
139          *             number of nodes in the tree with the
140          *             given root parameter.
141          */
142
143         private int rsize(Node root)
144         {
145             if (root == null)                  // Base case − empty tree.
146                 return 0;
147             else                               // Inductive case.
148                 return 1 + rsize(root.lchild)  //   1 + size of left subtree
149                          + rsize(root.rchild); //     + size of right subtree
150         }
151
152         /*
153          *   height() − return the height of the tree.  The height of an empty
154          *              tree is −1, otherwise the height is 1 plus
155          *              the larger of the height of the root's left or right
156          *              subtree.
157          */
158
159         public int height()
160         {
161             return rheight(root);
162         }
163
164
165         /*
166          *   rheight() − a recursive function to return the
167          *               height of the tree starting at the
168          *               parameter root.
169          */
170
171         private int rheight(Node root)
172         {
173             int rlength=0;
174             int llength=0;
175             int larger=0;
176             if(root==null){
177                 return 0;
178             }
179             if(root.lchild==null&&root.rchild==null){
180                 return 0;
181             }
182             if(rheight(root.lchild)>rheight(root.rchild)){
183                 return rheight(root.lchild)+1;
184             }
185             else{
186                 return rheight(root.rchild)+1;
187             }
188         }
189
190
191         /*
192          *   numLeaf() − return the number of leaf nodes
193          *               in the tree.
194          */
195
196         public int numLeaf()
197         {
198             return rnumLeaf(root);
199         }
200
201         /*
202          *   rnumLeaf() − return the number of leaf nodes
203          *                in the tree with given root
204          *                parameter.
205          */
206
207         private int rnumLeaf(Node root)
```

```
208     {
209         if(root==null){
210             return 0;
211         }
212         if(root.lchild==null&&root.rchild==null){
213             return 1;
214         }
215         else{
216             return rnumLeaf(root.lchild)+rnumLeaf(root.rchild);
217         }
218     }
219
220
221
222
223     /*  main() − Used to construct a few binary trees to test out
224      *           the implementation.
225      */
226
227     static public void main( String [ ] args )
228     {
229         BinaryTree<String> sh=new BinaryTree<String>("H");
230         BinaryTree<String> w1=new BinaryTree<String>();
231         BinaryTree<String> w2=new BinaryTree<String>();
232         BinaryTree<String> sg=new BinaryTree<String>("G");
233         BinaryTree<String> w3=new BinaryTree<String>();
234         w2.merge("F",w1,sh);
235         w3.merge("D",w2,sg);
236         System.out.println();
237
238         BinaryTree<Integer> t1 = new BinaryTree<Integer>( 1 );  // Tree with root node
    1
239         BinaryTree<Integer> t3 = new BinaryTree<Integer>( 3 );  // Tree with root node
    3
240         BinaryTree<Integer> t5 = new BinaryTree<Integer>( 5 );  // Tree with root node
    5
241         BinaryTree<Integer> t7 = new BinaryTree<Integer>( 7 );  // Tree with root node
    7
242         BinaryTree<Integer> t2 = new BinaryTree<Integer>( );    // Empty tree
243         BinaryTree<Integer> t4 = new BinaryTree<Integer>( );    // Empty tree
244         BinaryTree<Integer> t6 = new BinaryTree<Integer>( );    // Empty tree
245         BinaryTree<Integer> t9 = new BinaryTree<Integer>( );    // Empty tree
246
247         BinaryTree<String> s1 = new BinaryTree<String>();  // Tree with root node 1
248         BinaryTree<String> s2 = new BinaryTree<String>();
249         BinaryTree<String> s3 = new BinaryTree<String>();
250         BinaryTree<String> s4 = new BinaryTree<String>();
251         BinaryTree<String> se=new BinaryTree<String>("E");
252         BinaryTree<String> w4=new BinaryTree<String>();
253         s1.merge("B",s4,w3);
254         s3.merge("C",w4,se);
255         s2.merge("A",s1,s3);
256
257         System.out.println("The tree from the first page has...");
258         System.out.println(s2.numLeaf()+" leaf Nodes");
259         System.out.println(s2.size()+" Nodes");
260         System.out.println(s2.height()+" length");
261         s2.printTree();
262
263
264
265         t2.merge( 2, t1, t3 );   // Merge trees to create new trees.
266         t6.merge( 6, t5, t7 );
267         t4.merge( 4, t2, t6 );
268         // This next merge should fail
269         t9.merge( 9, t4, t4 );
270
271
272         System.out.println( "t4 should be perfect 1−7; t2 empty" );
```

```
273             System.out.println( "----------------" );
274             System.out.println( "t4" );
275             System.out.println( "----------------" );
276             System.out.println( "t2" );
277             System.out.println( "----------------" );
278             System.out.println( "t4 size: " + t4.size() );
279             System.out.println( "t4 height: " + t4.height() );
280             System.out.printf( "t4 has %d leafNodes.\n\n", t4.numLeaf() );
281
282             t4.printTree();   // Print the tree.
283
284             System.out.println( "\n----------------" );
285             System.out.println( "s4 size: " + s4.size() );
286             System.out.println( "s4 height: " + s4.height() );
287             System.out.printf( "s4 has %d leafNodes.\n\n", s4.numLeaf() );
288             System.out.println( "----------------\n" );
289
290
291         }
292
293
294         /*
295          *   Node inner class.  A node of a binary tree.
296          */
297
298         private class Node
299         {
300             private T data;         // The data stored in the node.
301             private Node lchild;    // Pointer to the left child Node.
302             private Node rchild;    // Pointer to the right child Node.
303             private Node parent;    // Pointer to the parent Node.
304
305             /*
306              *    Construct a null Node with all fields null.
307              */
308
309             public Node()
310             {
311                 this.data = null;
312                 this.lchild = null;
313                 this.rchild = null;
314                 this.parent = null;
315             }
316
317             /*
318              *   Construct a Node with specified data and
319              *   left and right child pointers.
320              */
321
322             public Node(T data, Node lchild, Node rchild)
323             {
324                 this.data = data;
325                 this.lchild = lchild;
326                 this.rchild = rchild;
327                 this.parent = null;
328             }
329         }
330 }
```