```
 1 package project5;
 2 // tree.java
 3 // demonstrates binary tree
 4 // to run this program: C>java TreeApp
 5 import java.io.*;
 6 import java.util.*;
 7 //import java.awt.event.*;
 8 import java.awt.Graphics;
 9
10
11
12 ////////////////////////////////////////////////////////////////
13 class Node
14    {
15    public int iData;              // data item (key)
16    public double dData;           // data item
17    public Node leftChild;         // this node's left child
18    public Node rightChild;        // this node's right child
19
20    public void displayNode()      // display ourself
21       {
22       System.out.print('{');
23       System.out.print(iData);
24       System.out.print(", ");
25       System.out.print(dData);
26       System.out.print("} ");
27       }
28    }  // end class Node
29 ////////////////////////////////////////////////////////////////
30 class Tree
31    {
32    private Node root;             // first node of tree
33
34    public void displayTree(Graphics g, Node localTree, int x, int y, int level ) {
35
36        int adjustedX = (int)((double)(x/Math.pow(2.0, (double)level)));
37
38        int nAdjustedX = -(adjustedX);
39
40        g.drawOval(x - 3, y - 15, 20, 20);
41
42        displayTree(g, localTree.leftChild, (x + nAdjustedX/2), y + 30, level + 1);
43        if(localTree == null) {
44        g.drawString(String.valueOf(localTree.iData), x, y);
45        g.drawOval((x + nAdjustedX/2)-3, y + 15, 20, 20);
46        g.drawLine((x + (nAdjustedX/2))+3, y + 15, x + 5, y + 5);
47        }
48
49        displayTree(g, localTree.rightChild, (x + adjustedX/2), y + 30, level + 1);
50        if(localTree == null) {
51        g.drawString(String.valueOf(localTree.iData), x, y);
52        g.drawOval((x + adjustedX/2)-3, y + 15, 20, 20);
53        g.drawLine((x + (adjustedX/2))+3, y + 15, x + 5, y + 5);
54        }
55 }
56 // -------------------------------------------------------------
57    public Tree()                  // constructor
58       { root = null; }            // no nodes in tree yet
59 // -------------------------------------------------------------
60    public Node find(int key)      // find node with given key
61       {                           // (assumes non-empty tree)
62       Node current = root;               // start at root
63       while(current.iData != key)        // while no match,
64          {
```

```
 65          if(key < current.iData)         // go left?
 66             current = current.leftChild;
 67          else                            // or go right?
 68             current = current.rightChild;
 69          if(current == null)             // if no child,
 70             return null;                 // didn't find it
 71          }
 72       return current;                    // found it
 73       }  // end find()
 74 // --------------------------------------------------------------
 75    public void insert(int id, double dd)
 76       {
 77       Node newNode = new Node();    // make new node
 78       newNode.iData = id;           // insert data
 79       newNode.dData = dd;
 80       if(root==null)                // no node in root
 81          root = newNode;
 82       else                          // root occupied
 83          {
 84          Node current = root;       // start at root
 85          Node parent;
 86          while(true)                // (exits internally)
 87             {
 88             parent = current;
 89             if(id < current.iData)  // go left?
 90                {
 91                current = current.leftChild;
 92                if(current == null)  // if end of the line,
 93                   {                 // insert on left
 94                   parent.leftChild = newNode;
 95                   return;
 96                   }
 97                }  // end if go left
 98             else                    // or go right?
 99                {
100                current = current.rightChild;
101                if(current == null)  // if end of the line
102                   {                 // insert on right
103                   parent.rightChild = newNode;
104                   return;
105                   }
106                }  // end else go right
107             }  // end while
108          }  // end else not root
109       }  // end insert()
110 // --------------------------------------------------------------
111    public boolean delete(int key) // delete node with given key
112       {                           // (assumes non-empty list)
113       Node current = root;
114       Node parent = root;
115       boolean isLeftChild = true;
116
117       while(current.iData != key)        // search for node
118          {
119          parent = current;
120          if(key < current.iData)         // go left?
121             {
122             isLeftChild = true;
123             current = current.leftChild;
124             }
125          else                            // or go right?
126             {
127             isLeftChild = false;
128             current = current.rightChild;
```

```
129            }
130         if(current == null)             // end of the line,
131            return false;                // didn't find it
132         }  // end while
133       // found node to delete
134
135       // if no children, simply delete it
136       if(current.leftChild==null &&
137                                  current.rightChild==null)
138          {
139          if(current == root)             // if root,
140             root = null;                 // tree is empty
141          else if(isLeftChild)
142             parent.leftChild = null;     // disconnect
143          else                            // from parent
144             parent.rightChild = null;
145          }
146
147       // if no right child, replace with left subtree
148       else if(current.rightChild==null)
149          if(current == root)
150             root = current.leftChild;
151          else if(isLeftChild)
152             parent.leftChild = current.leftChild;
153          else
154             parent.rightChild = current.leftChild;
155
156       // if no left child, replace with right subtree
157       else if(current.leftChild==null)
158          if(current == root)
159             root = current.rightChild;
160          else if(isLeftChild)
161             parent.leftChild = current.rightChild;
162          else
163             parent.rightChild = current.rightChild;
164
165       else  // two children, so replace with inorder successor
166          {
167          // get successor of node to delete (current)
168          Node successor = getSuccessor(current);
169
170          // connect parent of current to successor instead
171          if(current == root)
172             root = successor;
173          else if(isLeftChild)
174             parent.leftChild = successor;
175          else
176             parent.rightChild = successor;
177
178          // connect successor to current's left child
179          successor.leftChild = current.leftChild;
180          }  // end else two children
181       // (successor cannot have a left child)
182       return true;                                  // success
183       }  // end delete()
184 // -----------------------------------------------------------
185    // returns node with next-highest value after delNode
186    // goes to right child, then right child's left descendents
187    private Node getSuccessor(Node delNode)
188       {
189       Node successorParent = delNode;
190       Node successor = delNode;
191       Node current = delNode.rightChild;   // go to right child
192       while(current != null)               // until no more
```

```
193          {                                  // left children,
194          successorParent = successor;
195          successor = current;
196          current = current.leftChild;       // go to left child
197          }
198                                             // if successor not
199       if(successor != delNode.rightChild)   // right child,
200          {                                  // make connections
201          successorParent.leftChild = successor.rightChild;
202          successor.rightChild = delNode.rightChild;
203          }
204       return successor;
205       }
206
207 // -----------------------------------------------------------
208    public Node getRoot(){
209        return root;
210    }
211
212 // -----------------------------------------------------------
213
214    public void traverse(int traverseType)
215       {
216       switch(traverseType)
217          {
218          case 1: System.out.print("\nPreorder traversal: ");
219                  preOrder(root);
220                  break;
221          case 2: System.out.print("\nInorder traversal:  ");
222                  inOrder(root);
223                  break;
224          case 3: System.out.print("\nPostorder traversal: ");
225                  postOrder(root);
226                  break;
227          }
228       System.out.println();
229       }
230 // -----------------------------------------------------------
231    private void preOrder(Node localRoot)
232       {
233       if(localRoot != null)
234          {
235          System.out.print(localRoot.iData + " ");
236          preOrder(localRoot.leftChild);
237          preOrder(localRoot.rightChild);
238          }
239       }
240 // -----------------------------------------------------------
241    private void inOrder(Node localRoot)
242       {
243       if(localRoot != null)
244          {
245          inOrder(localRoot.leftChild);
246          System.out.print(localRoot.iData + " ");
247          inOrder(localRoot.rightChild);
248          }
249       }
250 // -----------------------------------------------------------
251    private void postOrder(Node localRoot)
252       {
253       if(localRoot != null)
254          {
255          postOrder(localRoot.leftChild);
256          postOrder(localRoot.rightChild);
```

```
257          System.out.print(localRoot.iData + " ");
258          }
259       }
260 // --------------------------------------------------------------
261    public void displayTree()
262       {
263       Stack globalStack = new Stack();
264       globalStack.push(root);
265       int nBlanks = 32;
266       boolean isRowEmpty = false;
267       System.out.println(
268       "......................................................");
269       while(isRowEmpty==false)
270          {
271          Stack localStack = new Stack();
272          isRowEmpty = true;
273
274          for(int j=0; j<nBlanks; j++)
275             System.out.print(' ');
276
277          while(globalStack.isEmpty()==false)
278             {
279             Node temp = (Node)globalStack.pop();
280             if(temp != null)
281                {
282                System.out.print(temp.iData);
283                localStack.push(temp.leftChild);
284                localStack.push(temp.rightChild);
285
286                if(temp.leftChild != null ||
287                                  temp.rightChild != null)
288                   isRowEmpty = false;
289                }
290             else
291                {
292                System.out.print("--");
293                localStack.push(null);
294                localStack.push(null);
295                }
296             for(int j=0; j<nBlanks*2-2; j++)
297                System.out.print(' ');
298             }  // end while globalStack not empty
299          System.out.println();
300          nBlanks /= 2;
301          while(localStack.isEmpty()==false)
302             globalStack.push( localStack.pop() );
303          }  // end while isRowEmpty is false
304       System.out.println(
305       "......................................................");
306       }  // end displayTree()
307 // --------------------------------------------------------------
308    }  // end class Tree
309 ////////////////////////////////////////////////////////////////
310 class TreeApp
311    {
312    public static void main(String[] args) throws IOException
313       {
314       int value;
315       Tree theTree = new Tree();
316
317       theTree.insert(50, 1.5);
318       theTree.insert(25, 1.2);
319       theTree.insert(75, 1.7);
320       theTree.insert(12, 1.5);
```

```
321       theTree.insert(37, 1.2);
322       theTree.insert(43, 1.7);
323       theTree.insert(30, 1.5);
324       theTree.insert(33, 1.2);
325       theTree.insert(87, 1.7);
326       theTree.insert(93, 1.5);
327       theTree.insert(97, 1.5);
328
329       while(true)
330          {
331          System.out.print("Enter first letter of show, ");
332          System.out.print("insert, find, delete, or traverse: ");
333          int choice = getChar();
334          switch(choice)
335             {
336             case 's':
337                theTree.displayTree();
338                break;
339             case 'i':
340                System.out.print("Enter value to insert: ");
341                value = getInt();
342                theTree.insert(value, value + 0.9);
343                break;
344             case 'f':
345                System.out.print("Enter value to find: ");
346                value = getInt();
347                Node found = theTree.find(value);
348                if(found != null)
349                   {
350                   System.out.print("Found: ");
351                   found.displayNode();
352                   System.out.print("\n");
353                   }
354                else
355                   System.out.print("Could not find ");
356                   System.out.print(value + '\n');
357                break;
358             case 'd':
359                System.out.print("Enter value to delete: ");
360                value = getInt();
361                boolean didDelete = theTree.delete(value);
362                if(didDelete)
363                   System.out.print("Deleted " + value + '\n');
364                else
365                   System.out.print("Could not delete ");
366                   System.out.print(value + '\n');
367                break;
368             case 't':
369                System.out.print("Enter type 1, 2 or 3: ");
370                value = getInt();
371                theTree.traverse(value);
372                break;
373             default:
374                System.out.print("Invalid entry\n");
375             }  // end switch
376          }  // end while
377       }  // end main()
378 // --------------------------------------------------------------
379    public static String getString() throws IOException
380       {
381       InputStreamReader isr = new InputStreamReader(System.in);
382       BufferedReader br = new BufferedReader(isr);
383       String s = br.readLine();
384       return s;
```

```
385       }
386 // --------------------------------------------------------------
387    public static char getChar() throws IOException
388       {
389       String s = getString();
390       return s.charAt(0);
391       }
392 //--------------------------------------------------------------
393    public static int getInt() throws IOException
394       {
395       String s = getString();
396       return Integer.parseInt(s);
397       }
398 // --------------------------------------------------------------
399    }  // end class TreeApp
400 ////////////////////////////////////////////////////////////////
```