```java
package project5;
// tree.java
// demonstrates binary tree
// to run this program: C>java TreeApp
import java.io.*;
import java.util.*;
//import java.awt.event.*;
import java.awt.Graphics;



////////////////////////////////////////////////////////////////
class Node
   {
   public int iData;              // data item (key)
   public double dData;           // data item
   public Node leftChild;         // this node's left child
   public Node rightChild;        // this node's right child

   public void displayNode()      // display ourself
      {
      System.out.print('{');
      System.out.print(iData);
      System.out.print(", ");
      System.out.print(dData);
      System.out.print("} ");
      }
   }  // end class Node
////////////////////////////////////////////////////////////////
class Tree
   {
   private Node root;             // first node of tree

   public void displayTree(Graphics g, Node localTree, int x, int y, int level ) {

       int adjustedX = (int)((double)(x/Math.pow(2.0, (double)level)));

       int nAdjustedX = -(adjustedX);

       g.drawOval(x - 3, y - 15, 20, 20);

       displayTree(g, localTree.leftChild, (x + nAdjustedX/2), y + 30, level + 1);
       if(localTree == null) {
       g.drawString(String.valueOf(localTree.iData), x, y);
       g.drawOval((x + nAdjustedX/2)-3, y + 15, 20, 20);
       g.drawLine((x + (nAdjustedX/2))+3, y + 15, x + 5, y + 5);
       }

       displayTree(g, localTree.rightChild, (x + adjustedX/2), y + 30, level + 1);
       if(localTree == null) {
       g.drawString(String.valueOf(localTree.iData), x, y);
       g.drawOval((x + adjustedX/2)-3, y + 15, 20, 20);
       g.drawLine((x + (adjustedX/2))+3, y + 15, x + 5, y + 5);
       }
}
// -------------------------------------------------------------
   public Tree()                  // constructor
      { root = null; }            // no nodes in tree yet
// -------------------------------------------------------------
   public Node find(int key)      // find node with given key
      {                           // (assumes non-empty tree)
      Node current = root;               // start at root
      while(current.iData != key)        // while no match,
         {
```

```
         if(key < current.iData)         // go left?
            current = current.leftChild;
         else                            // or go right?
            current = current.rightChild;
         if(current == null)             // if no child,
            return null;                 // didn't find it
         }
      return current;                    // found it
      }  // end find()
// -------------------------------------------------------------
   public void insert(int id, double dd)
      {
      Node newNode = new Node();    // make new node
      newNode.iData = id;           // insert data
      newNode.dData = dd;
      if(root==null)                // no node in root
         root = newNode;
      else                          // root occupied
         {
         Node current = root;       // start at root
         Node parent;
         while(true)                // (exits internally)
            {
            parent = current;
            if(id < current.iData)  // go left?
               {
               current = current.leftChild;
               if(current == null)  // if end of the line,
                  {                 // insert on left
                  parent.leftChild = newNode;
                  return;
                  }
               }  // end if go left
            else                    // or go right?
               {
               current = current.rightChild;
               if(current == null)  // if end of the line
                  {                 // insert on right
                  parent.rightChild = newNode;
                  return;
                  }
               }  // end else go right
            }  // end while
         }  // end else not root
      }  // end insert()
// -------------------------------------------------------------
   public boolean delete(int key) // delete node with given key
      {                           // (assumes non-empty list)
      Node current = root;
      Node parent = root;
      boolean isLeftChild = true;

      while(current.iData != key)        // search for node
         {
         parent = current;
         if(key < current.iData)         // go left?
            {
            isLeftChild = true;
            current = current.leftChild;
            }
         else                            // or go right?
            {
            isLeftChild = false;
            current = current.rightChild;
            }
         if(current == null)               // end of the line,
            return false;                  // didn't find it
         }  // end while
      // found node to delete

      // if no children, simply delete it
      if(current.leftChild==null &&
                                   current.rightChild==null)
         {
         if(current == root)               // if root,
            root = null;                   // tree is empty
         else if(isLeftChild)
            parent.leftChild = null;       // disconnect
         else                              // from parent
            parent.rightChild = null;
         }

      // if no right child, replace with left subtree
      else if(current.rightChild==null)
         if(current == root)
            root = current.leftChild;
         else if(isLeftChild)
            parent.leftChild = current.leftChild;
         else
            parent.rightChild = current.leftChild;

      // if no left child, replace with right subtree
      else if(current.leftChild==null)
         if(current == root)
            root = current.rightChild;
         else if(isLeftChild)
            parent.leftChild = current.rightChild;
         else
            parent.rightChild = current.rightChild;

      else  // two children, so replace with inorder successor
         {
         // get successor of node to delete (current)
         Node successor = getSuccessor(current);

         // connect parent of current to successor instead
         if(current == root)
            root = successor;
         else if(isLeftChild)
            parent.leftChild = successor;
         else
            parent.rightChild = successor;

         // connect successor to current's left child
         successor.leftChild = current.leftChild;
         }  // end else two children
      // (successor cannot have a left child)
      return true;                                // success
      }  // end delete()
// -------------------------------------------------------------
   // returns node with next-highest value after delNode
   // goes to right child, then right child's left descendents
   private Node getSuccessor(Node delNode)
      {
      Node successorParent = delNode;
      Node successor = delNode;
      Node current = delNode.rightChild;   // go to right child
      while(current != null)               // until no more
```

```
         {                                 // left children,
         successorParent = successor;
         successor = current;
         current = current.leftChild;      // go to left child
         }
                                           // if successor not
      if(successor != delNode.rightChild)  // right child,
         {                                 // make connections
         successorParent.leftChild = successor.rightChild;
         successor.rightChild = delNode.rightChild;
         }
      return successor;
      }

// -------------------------------------------------------------
   public Node getRoot(){
       return root;
   }

// -------------------------------------------------------------

   public void traverse(int traverseType)
      {
      switch(traverseType)
         {
         case 1: System.out.print("\nPreorder traversal: ");
                 preOrder(root);
                 break;
         case 2: System.out.print("\nInorder traversal:  ");
                 inOrder(root);
                 break;
         case 3: System.out.print("\nPostorder traversal: ");
                 postOrder(root);
                 break;
         }
      System.out.println();
      }
// -------------------------------------------------------------
   private void preOrder(Node localRoot)
      {
      if(localRoot != null)
         {
         System.out.print(localRoot.iData + " ");
         preOrder(localRoot.leftChild);
         preOrder(localRoot.rightChild);
         }
      }
// -------------------------------------------------------------
   private void inOrder(Node localRoot)
      {
      if(localRoot != null)
         {
         inOrder(localRoot.leftChild);
         System.out.print(localRoot.iData + " ");
         inOrder(localRoot.rightChild);
         }
      }
// -------------------------------------------------------------
   private void postOrder(Node localRoot)
      {
      if(localRoot != null)
         {
         postOrder(localRoot.leftChild);
         postOrder(localRoot.rightChild);
         System.out.print(localRoot.iData + " ");
         }
      }
// -------------------------------------------------------------
   public void displayTree()
      {
      Stack globalStack = new Stack();
      globalStack.push(root);
      int nBlanks = 32;
      boolean isRowEmpty = false;
      System.out.println(
      "......................................................");
      while(isRowEmpty==false)
         {
         Stack localStack = new Stack();
         isRowEmpty = true;

         for(int j=0; j<nBlanks; j++)
            System.out.print(' ');

         while(globalStack.isEmpty()==false)
            {
            Node temp = (Node)globalStack.pop();
            if(temp != null)
               {
               System.out.print(temp.iData);
               localStack.push(temp.leftChild);
               localStack.push(temp.rightChild);

               if(temp.leftChild != null ||
                                   temp.rightChild != null)
                  isRowEmpty = false;
               }
            else
               {
               System.out.print("--");
               localStack.push(null);
               localStack.push(null);
               }
            for(int j=0; j<nBlanks*2-2; j++)
               System.out.print(' ');
            }  // end while globalStack not empty
         System.out.println();
         nBlanks /= 2;
         while(localStack.isEmpty()==false)
            globalStack.push( localStack.pop() );
         }  // end while isRowEmpty is false
      System.out.println(
      "......................................................");
      }  // end displayTree()
// -------------------------------------------------------------
   }  // end class Tree
////////////////////////////////////////////////////////////////
class TreeApp
   {
   public static void main(String[] args) throws IOException
      {
      int value;
      Tree theTree = new Tree();

      theTree.insert(50, 1.5);
      theTree.insert(25, 1.2);
      theTree.insert(75, 1.7);
      theTree.insert(12, 1.5);
      theTree.insert(37, 1.2);
      theTree.insert(43, 1.7);
      theTree.insert(30, 1.5);
      theTree.insert(33, 1.2);
      theTree.insert(87, 1.7);
      theTree.insert(93, 1.5);
      theTree.insert(97, 1.5);

      while(true)
         {
         System.out.print("Enter first letter of show, ");
         System.out.print("insert, find, delete, or traverse: ");
         int choice = getChar();
         switch(choice)
            {
            case 's':
               theTree.displayTree();
               break;
            case 'i':
               System.out.print("Enter value to insert: ");
               value = getInt();
               theTree.insert(value, value + 0.9);
               break;
            case 'f':
               System.out.print("Enter value to find: ");
               value = getInt();
               Node found = theTree.find(value);
               if(found != null)
                  {
                  System.out.print("Found: ");
                  found.displayNode();
                  System.out.print("\n");
                  }
               else
                  System.out.print("Could not find ");
                  System.out.print(value + '\n');
               break;
            case 'd':
               System.out.print("Enter value to delete: ");
               value = getInt();
               boolean didDelete = theTree.delete(value);
               if(didDelete)
                  System.out.print("Deleted " + value + '\n');
               else
                  System.out.print("Could not delete ");
                  System.out.print(value + '\n');
               break;
            case 't':
               System.out.print("Enter type 1, 2 or 3: ");
               value = getInt();
               theTree.traverse(value);
               break;
            default:
               System.out.print("Invalid entry\n");
            }  // end switch
         }  // end while
      }  // end main()
// -------------------------------------------------------------
   public static String getString() throws IOException
      {
      InputStreamReader isr = new InputStreamReader(System.in);
      BufferedReader br = new BufferedReader(isr);
      String s = br.readLine();
      return s;
      }
// -------------------------------------------------------------
   public static char getChar() throws IOException
      {
      String s = getString();
      return s.charAt(0);
      }
//-------------------------------------------------------------
   public static int getInt() throws IOException
      {
      String s = getString();
      return Integer.parseInt(s);
      }
// -------------------------------------------------------------
   }  // end class TreeApp
////////////////////////////////////////////////////////////////
```