```java
/*
   Ivan Fransazov D block Datastructures Project 8 Huffman Coding
   Goal: create a HuffmanTree class that builds a HuffmanTree given an array of information,
   use that tree to create a .code file, to reconstruct a HuffmanTree from input from a .code file,
   and to decode an encoded message using the rebuilt tree and output that into a file.
*/
import java.util.PriorityQueue;
import java.util.Queue;
import java.io.PrintStream;
import java.util.Scanner;
import java.util.Stack;

public class HuffmanTree
{
   private Queue<HuffmanNode> HuffmanNodes; // priority queue of the all HuffmanNodes

   // creates a binary HuffmanTree from a passed in array where the index is the ASCII value
   // and the value at that index is its occurrences
   public HuffmanTree(int[] counts)
   {
      HuffmanNodes = new PriorityQueue<HuffmanNode>();

      createQueue(counts); // populate queue with solo HuffmanNodes
      createTree(); // merge them together to create HuffmanTree
   }

   // given an array as mentioned in the constructor, create single HuffmanNodes with the ASCII
   // values and frequency, then populate HuffmanNodes with the "solo" HuffmanNodes
   private void createQueue(int[] counts)
   {
      for (int i = 0; i < counts.length; i++)
      {
         if (counts[i] == 0) // if it doesn't occur (frequency is 0)
         {
            continue;
         }
         // otherwise, add a single HuffmanNode to HuffmanNodes
         // frequency is value at index i, value is the index i
         HuffmanNodes.add(new HuffmanNode(counts[i], (char) i));
      }
      //End Of File (EOF) node
      // occurs once (frequency = 1) and has ASCII value one larger than the array
      HuffmanNodes.add(new HuffmanNode(1, (char) counts.length));
   }

   // combines all the single HuffmanNodes into a HuffmanTree
   // starts by removing the front two nodes (smallest frequency), combining them, and creating
   // a node w/ ASCII value of 0 (null) and adds that mini tree back into the queue
   // continues with that process until there is one HuffmanNode left (overallRoot)
   private void createTree()
   {
      while (HuffmanNodes.size() > 1) // while there are multiple nodes left...
      {
         HuffmanNode one = HuffmanNodes.remove(); // remove the first node (lowest freq)
         HuffmanNode two = HuffmanNodes.remove(); // remove the next front node (next lowest freq)

         int totalFrequency = one.frequency + two.frequency;
         // create a new node that connects the previous removed nodes, and has a value of NULL
         HuffmanNode newRoot = new HuffmanNode(totalFrequency, (char) 0, one, two);
         HuffmanNodes.add(newRoot); // add it back into the queue
      }
   }
```

```
63
64      // given a PrintStream object, writes the code file into the determined file
65      public void write(PrintStream output)
66      {
67         BitOutputStream bitOutput = new BitOutputStream(output, true);
68         write(HuffmanNodes.peek(), bitOutput, output, "");
69      }
70      // given a HuffmanNode, a BitOutputStream, PrintStream, and a String for
        accumulation of bits
71      // traverses the tree until it reaches a leaf, then writes its char value as an int
72      // and the code to reach that node (in 0s and 1s)
73      private void write(HuffmanNode root, BitOutputStream bitOutput, PrintStream output,
        String bits)
74      {
75         if (root.left == null && root.right == null) // if root is a leaf...
76         {
77            // output the char value as an int, and move down a line
78            output.print((int) root.value + "\n");
79
80            // for each character in the String bits...
81            for (char c : bits.toCharArray())
82            {
83               // '0' - '0' = 0 and '1' - '0' = 1
84               // bc of the ASCII table ordering
85               bitOutput.writeBit(c - '0'); // write it as an int
86            }
87
88            output.print("\n");
89            return;
90         }
91
92         bits += "0"; // add a 0, and traverse the left side
93         write(root.left, bitOutput, output, bits);
94         // remove the previous addition, after jumping out of the left tree
95         bits = bits.substring(0, bits.length() - 1);
96
97         bits += "1"; // add a 1, and traverse right side
98         write(root.right, bitOutput, output, bits);
99         // remove the previous addition, after jumping out of the right tree
100        bits = bits.substring(0, bits.length() - 1);
101     }
102
103     //////////////
104     //decryption//
105     //////////////
106
107     // builds a HuffmanTree given a scanner of a .code file
108     public HuffmanTree(Scanner input)
109     {
110        HuffmanNodes = new PriorityQueue<HuffmanNode>();
111        // create dummy root w/ negative frequency and NULL value
112        HuffmanNode overallRoot = new HuffmanNode(-1, (char) 0);
113
114        HuffmanNodes.add(overallRoot);
115
116        // build tree
117        overallRoot = buildTree(overallRoot, input);
118     }
119
120     // given a HuffmanNode, and a Scanner on the .code file,
121     // reads the input and builds the HuffmanTree
122     private HuffmanNode buildTree(HuffmanNode root, Scanner input)
123     {
124        while (input.hasNextLine()) // while there is input left...
125        {
126           // get the ASCII value as an int (I.E charInt = 97)
127           int charInt = Integer.parseInt(input.nextLine());
128           // get the path to the leaf as a String (I.E charCode = "01")
129           String charCode = input.nextLine();
```

```
130             // convert the charCode from a String int a stack of 0s and 1s
131             Stack<Integer> bitsStack = stringToStack(charCode);
132
133             // find farthest EXISTING node in the current tree
134             HuffmanNode farthestNode = findFarthestNode(root, bitsStack);
135
136             // add the rest of the path from that node
137             addPath(farthestNode, charInt, bitsStack);
138          }
139          return root;
140       }
141
142       // given a String of 0s and 1s, returns a Stack that mimics the order
143       // I.E: "011" --> (top) 0, 1, 1 (bottom)
144       private Stack<Integer> stringToStack(String bits)
145       {
146          Stack<Integer> bitsStack = new Stack<Integer>();
147          // start at the end of the String bc Stack adds from the top
148          for (int i = bits.length() - 1; i >= 0; i--)
149          {
150             int bit = bits.charAt(i) - '0'; // get the bit as an int
151             bitsStack.push(bit);
152          }
153          return bitsStack;
154       }
155
156       // given a HuffmanNode, a charInt (ASCII value), and a Stack of the path to leaf
157       // adds the rest of the path to the leaf on the tree
158       private void addPath(HuffmanNode root, int charInt, Stack<Integer> bitsStack)
159       {
160          int nextMove = bitsStack.peek();
161          // finds out if next node on path is left/right (true means next move is left)
162          boolean isLeft = (nextMove == 0);
163
164          if (bitsStack.size() == 1) // if the next node will be the leaf w/ ASCII value
165          {
166             if (isLeft) // if it is a left move...
167             {
168                // add the node w/ charInt to the left of root
169                root.left = new HuffmanNode(-1, (char) charInt);
170             }
171             else
172             {
173                // otherwise, add the node w/ charInt to the right of root
174                root.right = new HuffmanNode(-1, (char) charInt);
175             }
176             return;
177          }
178
179          // otherwise, add the next dummy node on the path (ASCII value = 0)
180          nextNode(root, charInt, bitsStack, isLeft);
181       }
182
183       // given a HuffmanNode, the charInt, the Stack of the code path,
184       // and the left or right path boolean, adds the next dummy node to continue the path
185       private void nextNode(HuffmanNode root, int charInt, Stack<Integer> bitsStack,
          boolean isLeft)
186       {
187          bitsStack.pop(); // removes next step, it's about to finish moving
188          if (isLeft) // if the next move is left...
189          {
190             root.left = new HuffmanNode(-1, (char) 0); // add the dummy node to the left
191             root = root.left; // move root
192          }
193          else // otherwise...
194          {
195             root.right = new HuffmanNode(-1, (char) 0); // add the dummy node to the right
196             root = root.right; // move root
197          }
```

```
198         addPath(root, charInt, bitsStack);
199      }
200
201      // given a HuffmanNode, and the Stack of integers that represent the code path,
202      // return the farthest existing HuffmanNode along the Stack code path
203      private HuffmanNode findFarthestNode(HuffmanNode root, Stack<Integer> bitsStack)
204      {
205         int nextPath = bitsStack.peek(); // gets the next int
206
207         // if the int is 0 and the left path is null, or the int is 1 and the right path
             is null
208         // return the root, that is the farthest existing node
209         if ((nextPath == 0 && root.left == null) || nextPath == 1 && root.right == null)
210         {
211            return root;
212         }
213
214         bitsStack.pop(); // pop from the top to update the stack before moving the root
215
216         // if the int was 0, traverses the left tree, if the int was 1, traverses the
             right tree
217         return (nextPath == 0 ? findFarthestNode(root.left, bitsStack) :
218                     findFarthestNode(root.right, bitsStack));
219      }
220
221      // given a BitInputStream object, a PrintStream object, and the integer for the EOF
         value
222      // reads each bit from the encoded file and traverse the tree accordingly, once it
         has found a
223      // leaf, outputs that character and starts over until the EOF char has been found
         (not printed)
224      public void decode(BitInputStream input, PrintStream output, int eof)
225      {
226         while (true)
227         {
228            // gets the next char to print
229            char outputChar = findChar(HuffmanNodes.peek(), input);
230
231            if (outputChar == 256) // if that char is the EOF char...
232            {
233               break; // stop decoding
234            }
235
236            output.write(outputChar); // otherwise, print that char and start over
237         }
238      }
239
240      // given a HuffmanNode, and the BitInputStream of the encoded file,
241      // traverses the HuffmanTree using the bits until it reaches a leaf,
242      // it returns the char value of the leaf
243      private char findChar(HuffmanNode root, BitInputStream input)
244      {
245         if (root.left == null && root.right == null) // if it's a leaf
246         {
247            return root.value; // return the char value
248         }
249
250         // otherwise, if the next bit is 0, find the char in the left tree,
251         // else find the char in the right tree
252         return (input.readBit() == 0 ? findChar(root.left, input) : findChar(root.right,
             input));
253      }
254
255      //////////////////////
256      //HuffmanNode Class//
257      //////////////////////
258
259      private class HuffmanNode implements Comparable<HuffmanNode>
260      {
```

```
261         public int frequency;     // number of occurrences
262         public char value;        // value of the node
263         public HuffmanNode left;  // left Node
264         public HuffmanNode right; // right Node
265 
266         // given a frequency and value
267         // constructs a single HuffmanNode with null left and right children
268         public HuffmanNode(int frequency, char value)
269         {
270            this (frequency, value, null, null);
271         }
272 
273         // given a frequency, a value, a left node, and a right node,
274         // constructs a single HuffmanNode
275         public HuffmanNode(int frequency, char value, HuffmanNode left, HuffmanNode right)
276         {
277            this.frequency = frequency;
278            this.value = value;
279            this.left = left;
280            this.right = right;
281         }
282 
283         // orders HuffmanNodes from smallest --> largest based on frequency
284         public int compareTo(HuffmanNode other)
285         {
286            return this.frequency - other.frequency;
287         }
288 
289         // returns a String with information about the frequency and value
290         public String toString()
291         {
292            return (value == 0 ? "freq: " + frequency + ", value: NULL" : "freq: " +
               frequency + ", value: " + value);
293         }
294      }
295      // fin :)
296   }
```