Homework Turnin

Name:	Ameya Singh
Account:	ameyas (ameyas@uw.edu)
Student ID:	1868457
Section:	AQ
Course:	CSE 143 18au
Assignment:	a8b
Receipt ID:	aeb4ca62b249e889c7bc9823834dbfaa

Turnin script completed with output:

Turnin Successful!

The following file(s) were received:

HuffmanNode.java (1920 bytes)

```
 1. /*
 2.  * Author: Ameya Singh
 3.  * CSE 143 AQ
 4.  * TA: Soham P.
 5.  * Homework 8: Huffman Code
 6.  */
 7.
 8. /**
 9.  * HuffmanNode provides a node class used in creating instances of HuffmanTree.
10.  */
11. public class HuffmanNode implements Comparable<HuffmanNode> {
12.     /**
13.      * Nullable object holding integer representation of character.
14.      */
15.     public Integer character;
16.     /**
17.      * Nullable object holding integer number of occurrences of character.
18.      */
19.     public Integer numOccurrences;
20.     /**
21.      * Reference to left child node.
22.      */
23.     public HuffmanNode left;
24.     /**
25.      * Reference to right child node.
26.      */
27.     public HuffmanNode right;
28.
29.     /**
30.      * Constructs new HuffmanNode with no children.
31.      * @param character Character node represents.
32.      * @param numOccurrences Occurrences of represented character.
33.      */
34.     public HuffmanNode(Integer character, Integer numOccurrences) {
35.         this(character, numOccurrences, null, null);
36.     }
37.
38.     /**
39.      * Constructs a new HuffmanNode.
40.      * @param character Character node represents.
41.      * @param numOccurrences Occurrences of represented character.
42.      * @param left Left child node.
43.      * @param right Right child node.
44.      */
45.     public HuffmanNode(Integer character, Integer numOccurrences,
46.                        HuffmanNode left, HuffmanNode right) {
```

```
47.         this.character = character;
48.         this.numOccurrences = numOccurrences;
49.         this.left = left;
50.         this.right = right;
51.     }
52.
53.     /**
54.      * Compares number of occurrences.
55.      * @param o Other HuffmanNode.
56.      * @return Whether node has greater than less than or equal numOccurrences.
57.      */
58.     @Override
59.     public int compareTo(HuffmanNode o) {
60.         if (this.numOccurrences > o.numOccurrences) {
61.             return 1;
62.         } else if (this.numOccurrences < o.numOccurrences) {
63.             return -1;
64.         } else {
65.             return 0;
66.         }
67.     }
68. }
69.
70.
```

HuffmanTree2.java (5621 bytes)

```
1.  /*
2.   * Author: Ameya Singh
3.   * CSE 143 AQ
4.   * TA: Soham P.
5.   * Homework 8b: Huffman Code Bonus
6.   */
7.
8.  import java.io.*;
9.  import java.util.*;
10.
11. /**
12.  * Allows for compressing files using the Huffman Coding scheme.
13.  */
14. public class HuffmanTree2 {
15.     /**
16.      * Holds a reference to the root node of the HuffmanTree.
17.      */
18.     private HuffmanNode root;
19.
20.     /**
21.      * Constructs a new HuffmanTree using the passed array of the frequency of
22.      * characters.
23.      * @param count Integer array where count[i] is the number of occurrences of
24.      *              the character with integer value i.
25.      */
26.     public HuffmanTree2(int[] count) {
27.         Queue<HuffmanNode> priorityQueue = new PriorityQueue<HuffmanNode>();
28.
29.         for (int i = 0; i < count.length; i++) {
30.             if (count[i] > 0) {
31.                 priorityQueue.add(new HuffmanNode(i, count[i]));
32.             }
33.         }
34.         priorityQueue.add(new HuffmanNode(count.length, 1));
35.
36.         while (priorityQueue.size() > 1) {
37.             HuffmanNode node1 = priorityQueue.remove();
38.             HuffmanNode node2 = priorityQueue.remove();
39.             HuffmanNode newNode = new HuffmanNode(null,
40.                     node1.numOccurrences + node2.numOccurrences, node1, node2);
41.             priorityQueue.add(newNode);
42.         }
43.
44.         root = priorityQueue.remove();
45.     }
46.
47.     /**
48.      * Constructs a Huffman tree from the given input stream.
49.      * @param input BitInputSteam containing standard bit representation of
50.      *              tree.
51.      */
```

```
52.     public HuffmanTree2(BitInputStream input) {
53.         root = read(input);
54.     }
55.
56.     /**
57.      * Private helper to read tree from input stream.
58.      * @param input input stream containing tree.
59.      * @return new root node of tree.
60.      */
61.     private HuffmanNode read(BitInputStream input) {
62.         HuffmanNode node;
63.         if (input.readBit() == 0) {
64.             node = new HuffmanNode(null, null);
65.             node.left = read(input);
66.             node.right = read(input);
67.         } else {
68.             node = new HuffmanNode(read9(input), null);
69.         }
70.         return node;
71.     }
72.
73.     /**
74.      * Assigns codes for each character of the tree.
75.      * @param codes Array to fill with String for each character in the tree
76.      *              indicating its code.
77.      */
78.     public void assign(String[] codes) {
79.         assign(codes, root, "");
80.     }
81.
82.     /**
83.      * Private helper for assigning codes.
84.      * @param codes array to assign to.
85.      * @param node current node.
86.      * @param pathToNode current path.
87.      */
88.     private void assign(String[] codes , HuffmanNode node, String pathToNode) {
89.         if (node != null) {
90.             if (node.left == null && node.right == null) {
91.                 codes[node.character] = pathToNode;
92.             }
93.             assign(codes, node.left, pathToNode + "0");
94.             assign(codes, node.right, pathToNode + "1");
95.         }
96.     }
97.
98.     /**
99.      * Writes the current tree to the output stream using the standard bit
100.     * representation.
101.     * @param output BitOutputStream to which to write tree.
102.     */
103.    public void writeHeader(BitOutputStream output) {
104.        write(output, root);
105.    }
106.
107.    /**
108.     * Private helper to write bit representation of tree.
109.     * @param output output stream to write to.
110.     * @param node current node.
111.     */
112.    private void write(BitOutputStream output, HuffmanNode node) {
113.        if (node != null) {
114.            if (node.left != null && node.right != null) {
115.                output.writeBit(0);
116.            } else {
117.                output.writeBit(1);
118.                write9(output, node.character);
119.            }
120.            write(output, node.left);
121.            write(output, node.right);
122.        }
123.    }
124.
125.    /**
126.     * Reads bits from passed input stream and writes corresponding characters
127.     * from the HuffmanTree to the given output stream. Will stop reading when
128.     * a character matching the passed end of file parameter is reached.
129.     * @param input BitInputStream containing encoded characters to decode.
130.     * @param output PrintStream to which decoded characters will be written.
131.     * @param eof Character representing position at which to stop reading file.
132.     */
```

```
133.     public void decode(BitInputStream input, PrintStream output, int eof) {
134.         int currentBit = input.readBit();
135.         HuffmanNode currNode = root;
136.         boolean reachedEOF = false;
137.
138.         while (!reachedEOF) {
139.             if (currNode.left == null && currNode.right == null) {
140.                 if (currNode.character == eof) {
141.                     reachedEOF = true;
142.                 } else {
143.                     output.write(currNode.character);
144.                     currNode = root;
145.                 }
146.             }
147.
148.             if (currentBit == 0) {
149.                 currNode = currNode.left;
150.             } else {
151.                 currNode = currNode.right;
152.             }
153.
154.             currentBit = input.readBit();
155.         }
156.     }
157.
158.     // pre : an integer n has been encoded using write9 or its equivalent
159.     // post: reads 9 bits to reconstruct the original integer
160.     private int read9(BitInputStream input) {
161.         int multiplier = 1;
162.         int sum = 0;
163.         for (int i = 0; i < 9; i++) {
164.             sum += multiplier * input.readBit();
165.             multiplier = multiplier * 2;
166.         }
167.         return sum;
168.     }
169.
170.     // pre : 0 <= n < 512
171.     // post: writes a 9-bit representation of n to the given output stream
172.     private void write9(BitOutputStream output, int n) {
173.         for (int i = 0; i < 9; i++) {
174.             output.writeBit(n % 2);
175.             n = n / 2;
176.         }
177.     }
178. }
179.
```