Homework Turnin

Name:	Akshit K Patel
Email:	akshit@uw.edu
Student ID:	1561387
Section:	DC
Course:	CSE 143 16au
Assignment:	a8
Receipt ID:	c5631ec3f7c8057ac90c197db39da54c

Turnin Successful!

The following file(s) were received:

HuffmanCode.java (10214 bytes)

```java
/**
 * @author Akshit Patel
 * @Date 1/2/2016
 * CSE 143D DC
 * TA: Melissa Medsker
 * HW #8 Huffman Coding
 */
import java.io.PrintStream;
import java.util.*;

/**
 * HuffmanCode class represents a huffman code for a particular message. It
 * compresses the message and provides useful method to save the compression and
 * also method to decompress into original message.
 *
 */
public class HuffmanCode {

    /**
     * Reference to the huffman code tree of the given message.
     */
    private HuffmanNode huffmanRoot;

    /**
     * Constructs a new HuffmanCode object from a given array of ASCII
     * characters as indices and the data value its frequency.
     *
     * @param frequencies array of integers representing the frequency or the
     *        count of occurrence of a particular ASCII character, where the
     *        ASCII character is the index of the given array. The frequencies
     *        need to be positive i.e. >= 0
     */
    public HuffmanCode(int[] frequencies) {
        // priority queue to prioritize characters according to frequency.
        Queue<HuffmanNode> sort = new PriorityQueue<HuffmanNode>();
        // add the characters with frequency greater than 0 to priority queue.
        for (int i = 0; i < frequencies.length; i++) {
            if (frequencies[i] > 0) {
                sort.add(new HuffmanNode((char) i, frequencies[i]));
            }
        }
        // make the huffman code by combining elements.
        while (sort.size() > 1) {
            HuffmanNode firstData = sort.remove();
            HuffmanNode secondData = sort.remove();
            // add the combination of first and second element to the queue.
            // new parent has no data value other than combined frequency.
            sort.add(new HuffmanNode('\0',
                    firstData.frequency + secondData.frequency, firstData,
                    secondData));
        }
        // get the last remaining element which is the huffman code.
        this.huffmanRoot = sort.remove();
    }

    /**
     * Constructs a new HuffmanCode object given a scanner to read from a
     * previously constructed code from .code file.
     *
     * @param input scanner representing a previously constructed .code file.
     *        Scanner should not be null and must have data encoded in legal,
     *        existing and valid huffman standard format
     */
    public HuffmanCode(Scanner input) {
        // scan until no elements left.
        while (input.hasNextLine()) {
            // get the ASCII character.
            int n = Integer.parseInt(input.nextLine());
            // get the huffman code path directions for the character.
            String code = input.nextLine();
            // construct the tree.
            this.huffmanRoot = huffmanTree(this.huffmanRoot, (char) n, code);
        }
    }

    /**
     * Constructs the huffman code tree for a HuffmanCode object when given with
     * the directions to store the given ASCII character. The frequency of all
     * nodes are set to default value of zero. Direction refers to the string
     * sequence of 1s and 0s representing the right and left node of tree
     * respectively.
     *
     * @param current HuffmanNode representing the node of huffman tree. Used to
     *        construct the tree without changing the overall root of the tree
     * @param letter character representing ASCII value to input into the
     *        huffman tree. The default value of the parent node is set to \0
     * @param code String representing the path to store the character in the
     *        huffman tree. String cannot be null. String has to be combinations
     *        of 1s or 0s or both
     * @return HuffmanNode of the huffman tree made for this HuffmanCode object
     */
    private static HuffmanNode huffmanTree(HuffmanNode current, char letter,
                                           String code) {
        // if no more path details then we have a value to store as leaf.
        if (code.isEmpty()) {
            return new HuffmanNode(letter, 0);
        }
        // construct a new parent if current is null.
        if (current == null) {
            current = new HuffmanNode('\0', 0);
        }
        // if path has 0 go left branch else go right for 1.
        if (code.charAt(0) == '0') {
            current.left = huffmanTree(current.left, letter, code.substring(1));
        } else {
            current.right =
                       huffmanTree(current.right, letter, code.substring(1));
        }
        return current;
    }

    /**
     * Store the current huffman codes for the HuffmanCode object in standard
     * format to a given output stream.
     *
     * @param output PrintStream representing the .code file to store the
     *        huffman code in standard format.
     */
    public void save(PrintStream output) {
        this.save(output, this.huffmanRoot, "");
    }

    /**
     * Stores the huffman code tree of the current HuffmanCode object in
     * standard format to a given output stream.
```

```java
     *
     * @param output PrintStream representing the .code file to store the
     *        huffman code in standard format.
     * @param current HuffmanNode representing the node of huffman tree. Used to
     *        traverse through to get the character path and the character
     *        itself
     * @param code String representation of the path of the character in the
     *        huffman tree. Initially an empty string. String should not be null
     */
    private void save(PrintStream output, HuffmanNode current, String code) {
        if (current != null) {
            // if a leaf then we have the character and the path to store.
            if (current.left == null && current.right == null) {
                // store character as int.
                output.println((int) current.data);
                output.println(code);
            } else {
                // go left and add 0 to path or go right and add 1 to path.
                this.save(output, current.left, code + "0");
                this.save(output, current.right, code + "1");
            }
        }
    }

    /**
     * Decompresses the current HuffmanCode object if given a input stream of
     * the compressed message and an output stream to store the original message
     *
     * @param input BitInputStream representing the compressed .short file.
     *        Should not be null. Used with HuffmanCode object to decompress the
     *        compressed message.
     * @param output PrintStream representing a .new file to store the decoded
     *        message.
     */
    public void translate(BitInputStream input, PrintStream output) {
        HuffmanNode current = this.huffmanRoot;
        // decompress until no bits left.
        while (input.hasNextBit() && current != null) {
            // if current bit is 0 then go left else go right for 1.
            if (input.nextBit() == 0) {
                current = current.left;
            } else {
                current = current.right;
            }
            // if leaf then we have a character to output.
            if (current.left == null && current.right == null) {
                output.write(current.data);
                // change the reference back to root to look for next bit.
                current = this.huffmanRoot;
            }
        }
    }

    /**
     * HuffmanNode class stores a single node of a binary tree representing an
     * int and a character. It also provides a method to compare the int data of
     * this node to other given node.
     */
    private static class HuffmanNode implements Comparable<HuffmanNode> {
        /**
         * character to store in the node.
         */
        public final char data;
        /**
         * int to store in the node.
         */
        public final int frequency;
        /**
         * represents the left node of the binary tree
         */
        public HuffmanNode left;
        /**
         * represents the right node of the binary tree
         */
        public HuffmanNode right;

        /**
         * constructs a leaf node with the given character and int data for the
         * node.
         *
         * @param data character representing the ASCII character to be stored
         *        in the node.
         * @param frequency int representing the frequency of the ASCII
         *        character to be stored in the node.
         */
        public HuffmanNode(char data, int frequency) {
            this(data, frequency, null, null);
        }

        /**
         * Constructs a branch node with given character and int data with left
         * subtree and right subtree.
         *
         * @param data character representing the ASCII character to be stored
         *        in the node.
         * @param frequency int representing the frequency of the ASCII
         *        character to be stored in the node.
         * @param left HuffmanNode representing the left binary subtree.
         * @param right HuffmanNode representing the right binary subtree.
         */
        public HuffmanNode(char data,
                           int frequency,
                           HuffmanNode left,
                           HuffmanNode right) {
            this.data = data;
            this.frequency = frequency;
            this.left = left;
            this.right = right;
        }

        /**
         * Compares the frequency of this HuffmanNode to an other given
         * HuffmanNode.
         *
         * @return a negative integer, zero, or a positive integer as this
         *         HuffmanNode frequency is less than, equal to, or greater than
         *         the given HuffmanNode frequency
         */
        @Override
        public int compareTo(HuffmanNode other) {
            return Integer.compare(this.frequency, other.frequency);
        }
    }
}

```

secretmessage.short (906 bytes)

(binary file)

secretmessage.code (889 bytes)

```
115
0000
99
00010
13
000110
10
000111
101
001
98
0100000
44
0100001
103
010001
112
01001
104
0101
114
0110
111
0111
```

```
97
1000
109
100100
36
1001010000
56
1001010001
50
100101001
40
100101010
69
100101011
118
1001011
102
100110
107
100111
116
1010
110
10110
119
101110
120
101111000
41
101111001
73
1011110100
47
10111101010
70
10111101011
82
10111101100
83
10111101101
39
1011110111
85
1011111000
37
1011111001
45
101111101
65
101111110
49
1011111110
66
10111111110
68
10111111111
108
11000
46
1100100
121
1100101
117
110011
58
1101000000
89
1101000001
87
11010000100
63
11010000101
57
11010000110
80
11010000111
72
110100010
71
```

```
11010001100
75
11010001101
76
11010001110
77
11010001111
48
110100100
53
1101001010
78
1101001011
84
11010011
100
110101
105
11011
32
111
```