

```

1  /**
2  *   Brendan Raimann
3  *   2/6/16
4  *   Version 1.0
5  *   HuffmanRunner Class - Used for running the HuffmanTree and HuffmanNode classes
6  */
7
8  public class HuffmanRunner
9  {
10     /** Main method */
11     public static void main (String[] args)
12     {
13         String s = "This is a test.";
14         HuffmanTree tree = new HuffmanTree(s);
15         System.out.println(tree.encode());
16         // returns 1001111011110100111101001000001011101011011100
17         System.out.println(tree.decode("1001111011110100111101001000001011101011011100"));
18     }
19 }
20
21 /**
22 *   Brendan Raimann
23 *   2/6/16
24 *   Version 1.0
25 *   HuffmanTree Class - Used for building a binary tree to compact data
26 */
27
28 import java.util.PriorityQueue;
29 import java.util.HashMap;
30
31 public class HuffmanTree
32 {
33     /** Pointer to the root node */
34     private HuffmanNode root;
35
36     /** Contains the string for encoding */
37     private String str;
38
39     /** Constructor that builds the binary tree
40     *   @param s The string of letters to be used to build a Huffman Tree
41     */
42     public HuffmanTree(String s)
43     {
44         str = s;
45         root = buildTree(buildQueue(createMap(s)));
46     }
47
48     /**
49     *   Takes in a string and turns it into a HashMap
50     *   @param s A string to be turned into a HashMap
51     */
52     private HashMap<String,Integer> createMap(String s)
53     {
54         HashMap<String, Integer> map = new HashMap<String, Integer>();
55         for (int i = 0; i < s.length(); i++)
56         {
57             if (map.containsKey(s.substring(i,i+1)))
58                 map.put(s.substring(i,i+1),map.get(s.substring(i,i+1))+1);
59             else
60                 map.put(s.substring(i,i+1),1);
61         }
62         return map;
63     }
64
65     /**
66     *   Builds a PriorityQueue of HuffmanNode's using a HashMap
67     *   @param map A HashMap<String, Integer> That stores a letter and its number of occurrences
68     *   @return A PriorityQueue<HuffmanNode> that stores nodes with letters and their frequencies
69     */
70     private PriorityQueue<HuffmanNode> buildQueue(HashMap<String,Integer> map)
71     {
72         PriorityQueue<HuffmanNode> queue = new PriorityQueue<HuffmanNode>();
73
74

```

I think you could test a bit more rigorously than this.

You should elaborate.

Why is this a private variable?

```

75     for (String s: map.keySet())
76     {
77         queue.add(new HuffmanNode(s, map.get(s)));
78     }
79     return queue;
80 }
81
82 /**
83  * Builds a binary tree using a PriorityQueue<HuffmanNode>
84  * @param queue A PriorityQueue that stores nodes for the tree
85  * @return Returns the root node
86  */
87 private HuffmanNode buildTree(PriorityQueue<HuffmanNode> queue)
88 {
89     HuffmanNode node;
90     HuffmanNode head1;
91     HuffmanNode head2;
92     while (queue.size() > 1)
93     {
94         head1 = queue.remove();
95         head2 = queue.remove();
96         node = new HuffmanNode(head1.getString() + head2.getString(), head1.getFreq() + head2.getFreq());
97         node.setLeft(head1);
98         node.setRight(head2);
99         queue.add(node);
100    }
101    return queue.remove();
102 }
103
104 /**
105  * Used to encode the String in the class field to binary digits
106  * @return Returns a series of binary digits in a String
107  */
108 public String encode(String str)
109 {
110     String output = "";
111     String temp = "";
112     HuffmanNode index;
113     for (int i = 0; i < str.length(); i++)
114     {
115         index = root;
116         while (index.isLeaf() == false)
117         {
118             if (index.getLeft().getString().indexOf(str.charAt(i)) >= 0)
119             {
120                 index = index.getLeft();
121                 temp += "0";
122             }
123             else
124             {
125                 index = index.getRight();
126                 temp += "1";
127             }
128         }
129         output += temp;
130         temp = "";
131     }
132     return output;
133 }
134
135 /**
136  * Used to decode
137  * @param s Binary digits in a String
138  * @return Returns the result of decoding the binary digits into symbols using the tree
139  */
140 public String decode(String s)
141 {
142     String output = "";
143     HuffmanNode index = root;
144     for (int i = 0; i < s.length(); i++)
145     {
146         if (index.isLeaf() == true)

```

Comment your algorithms!

This should take in a parameter - otherwise a given Huffman implementation can only ever encode the same string.

```

149         {
150             output += index.getString();
151             index = root;
152             i--;
153         }
154         else
155         {
156             if (s.charAt(i) == '0')
157                 index = index.getLeft();
158             else
159                 index = index.getRight();
160         }
161     }
162     output += index.getString();
163     return output;
164 }
165 }
166
167 /**
168  *   Brendan Raimann
169  *   2/6/16
170  *   Version 1.0
171  *   HuffmanNode Class - Used as nodes for a binary tree
172  */
173
174 public class HuffmanNode implements Comparable<HuffmanNode>
175 {
176     /** The frequency of the String */
177     private int freq;
178     /** The stored String for the node */
179     private String key;
180     /** Pointer to the left node */
181     private HuffmanNode left;
182     /** Pointer to the right node */
183     private HuffmanNode right;
184
185     /**
186     *   Constructor that builds the node
187     *   @param s The String to be stored in the node
188     *   @param n The number of occurrences for the String
189     */
190     public HuffmanNode(String s, int n)
191     {
192         key = s;
193         freq = n;
194         left = null;
195         right = null;
196     }
197
198     /**
199     *   Returns the String of the node
200     *   @return Returns the stored String
201     */
202     public String getString()
203     {
204         return key;
205     }
206
207     /**
208     *   Returns the frequency of the String
209     *   @return Returns the number of occurrences for the String
210     */
211     public int getFreq()
212     {
213         return freq;
214     }
215
216     /**
217     *   Returns the left node pointer
218     *   @return Returns the pointer to the left node
219     */
220     public HuffmanNode getLeft()
221     {
222         return left;

```

```

223     }
224
225     /**
226     * Returns the right node pointer
227     * @return Returns the pointer to the right node
228     */
229     public HuffmanNode getRight()
230     {
231         return right;
232     }
233
234
235     /**
236     * Sets the left node
237     * @param node The node for the left pointer
238     */
239     public void setLeft(HuffmanNode node)
240     {
241         left = node;
242     }
243
244     /**
245     * Sets the right node
246     * @param node The node for the right pointer
247     */
248     public void setRight(HuffmanNode node)
249     {
250         right = node;
251     }
252
253     /**
254     * Returns whether or not the node has left and right pointers
255     * @return Returns true if the left and right pointers are null
256     */
257     public boolean isLeaf()
258     {
259         if (left == null && right == null)
260             return true;
261         return false;
262     }
263
264     /**
265     * Allows for comparing nodes
266     * @param node Another node for comparison
267     * @return Returns the difference in frequency between the nodes
268     */
269     public int compareTo(HuffmanNode node)
270     {
271         return freq - node.freq;
272     }
273
274     /**
275     * Returns a String representation of the node and its children
276     * @return A String representation of the node
277     */
278     public String toString()
279     {
280         String value = "[" + key + ", " + freq + "]";
281         if (isLeaf() == true)
282             return value;
283         else
284         {
285             if (left != null && right == null)
286                 return value + "(" + left.toString() + ",";
287             if (left == null && right != null)
288                 return value + "(" + right.toString() + ",";
289             return value + "(" + left.toString() + "," + right.toString() + ")";
290         }
291     }
292 }
293 }

```

Overall, good, though this was hard to test because your encode does not take in a parameter. The program seems to work properly though. Good job using javadoc, though make sure you have comments that actually explain what your algorithm as you go.

A/A+