```java
/**
    Runner to test the HuffmanNode and HuffmanTree functions
    @author Zachary Keller
    @version 1
*/
public class HuffmanRunner
{
    public static void main(String[] args)
    {
        HuffmanTree tree = new HuffmanTree();
        String bits = tree.encode();
        System.out.println("Decoded: " + tree.decode(bits));

    }
}

import java.lang.Iterable;
import java.util.Iterator;
/**
    The Huffman Node class is a node of a Tree-Structure. The node has two
    pointers to a left and right subtree. This node also holds a value and letters. There are
    various accessors and modifiers in this class, as well as functions that return
    information about the entire tree.
    @author Zachary Keller
    @version 0

*/
public class HuffmanNode implements Comparable<HuffmanNode>
{
    /**
        The value held in this node of the tree
    */
    protected int value;

    /**
        The letters held in this node
    */
    protected String letters;

    /**
        A pointer to the left branch subtree
    */
    protected HuffmanNode left;

    /**
        A pointer to the right branch subtree
    */
    protected HuffmanNode right;

    /**
        Constructor- Creates a new Binary Tree with given values
        @param v The value of this particular node of the tree
        @param l The left branch
        @param r The right branch
    */
    public HuffmanNode(Integer v, HuffmanNode l,HuffmanNode r)
    {
        value = v;
        left = l;
        right = r;
    }

    /**
        Overrides the Comparable Class so it can be used in the Priority Queue
    */
    public int compareTo(HuffmanNode other)
    {
        return value - other.value;
    }

    /**
        Constructor- creates a new Binary Tree with a stored value,
        but without a left or right branch
        @param v The value stored by this node
```

I figured it out, but it's strange that you call encode without providing text.

Oh yeah? That's not really saying much about the actual purpose of the data structure.

How is it being used?

```
75      */
76      public HuffmanNode(int v)
77      {
78          this(v, null, null);
79      }
80
81      /**
82          Constructor for HuffmanNode that takes in a value and letters
83          @param v Value
84          @param s Letters
85      */
86      public HuffmanNode(int v, String s)
87      {
88          value = v;
89          letters = s;
90      }
91
92      /**
93          Constructor- Calls the first Constructor with all null values
94      */
95      public HuffmanNode()
96      {
97          this(null, null, null);
98      }
99
100     /**
101         Returns the letters that the node holds"
102         @return letters
103     */
104     public String letters()
105     {
106         return letters;
107     }
108
109     /**
110         Returns the left branch of the tree
111         @return The left branch of the tree
112     */
113     public HuffmanNode left()
114     {
115         return left;
116     }
117
118     /**
119         Returns the right branch of the tree
120         @return The right branch of the tree
121     */
122     public HuffmanNode right()
123     {
124         return right;
125     }
126
127     /**
128         Returns the value this node of the tree is holding
129         @return The value this node of the tree is holding
130     */
131     public int value()
132     {
133         return value;
134     }
135
136
137
138     /**
139         Sets the left branch of the tree to a new Binary Tree
140         @param The new Tree
141     */
142     public void setLeft(HuffmanNode t)
143     {
144         left = t;
145     }
146
147     /**
148         Sets the right branch of the tree to a new Binary Tree
```

```
149         @param The new Tree
150     */
151     public void setRight(HuffmanNode t)
152     {
153         right = t;
154     }
155 
156     /**
157         Sets or changes the value that this tree node is holding
158         @param The new value
159     */
160     public void setValue(int v)
161     {
162         value = v;
163     }
164 
165 
166     /**
167         Determines whether or not this tree is a leaf,
168         meaning that it has no subtrees
169         @return Whether or not it is a leaf
170     */
171     public boolean isLeaf()
172     {
173         return (left == null && right == null);
174     }
175 
176     /**
177         Returns a string representation of the binary tree
178         @return A string representation of the binary tree
179     */
180     public String toString()
181     {
182         if (isLeaf())
183         {
184             return  "" + letters + " : " +  value;
185         }
186         if (left == null)
187             return letters + " : " + value + "(" + "EMPTY, " + right.toString() + ")";
188         else if (right == null)
189             return  letters + " : " + value + "(" + left.toString() + ", EMPTY" + ")";
190         else
191             return  letters + " : " + value + "(" + left.toString() + ", " + right.toString() + ")";
192     }
193 
194 
195 
196 
197 
198 
199 
200 
201 }
202 
203 import java.util.PriorityQueue;
204 import java.util.HashMap;
205 /**
206     This HuffmanTree class begins by making a map of the occurences of each letter in
207     a given string. Then I use that map to create HuffmanNodes, one for each letter. Then
208     I put those nodes in a priority queue, and use that to make the tree that will eventually solve
209     the Huffman Code. To create the tree, I take the first two nodes in the queue, combine them,
210     make that the parent node of the two first nodes, and put that parent node back into the queue. On
211     the tree is finished, it can be used to find letters given only bits of 1s and 0s.
212     @author Zachary Keller
213     @version final
214 */
215 public class HuffmanTree
216 {
217     /**
218         The very top of my tree
219     */
220     private HuffmanNode root;
221 
222     /**
```

I know we both know what Huffman code, but you should still include a description.

```
223         The phrase being encrypted
224     */
225     private String phrase;
226 
227     /**
228         Constructor that takes in a string to encode
229         @param input The message being encoded
230     */
231     public HuffmanTree(String input)
232     {
233         phrase = input;
234         root = createTree(createNodes(createMap(input)));
235 
236 
237     }
238 
239     /**
240         Default Constructor that creates a default string to encode
241     */
242     public HuffmanTree()
243     {
244         phrase = "sam scherl scooted school";
245         root = createTree(createNodes(createMap(phrase)));
246         //System.out.println(root);
247     }
248 
249     /**
250         Creates and returns a HashMap with the occurences of every letter
251         @param input The message being encoded
252         @return The HashMap containing the letters and occurences
253     */
254     private HashMap<String, Integer> createMap(String input)
255     {
256         HashMap<String, Integer> occur = new HashMap<String, Integer>();
257         for (int i = 0; i < input.length(); i++)
258         {
259             if (occur.containsKey("" + input.charAt(i)))
260             {
261                 occur.put("" + input.charAt(i),occur.get("" + input.charAt(i)) +1);
262             }
263             else
264             {
265                 occur.put("" + input.charAt(i),1);
266             }
267         }
268         return occur;
269 
270     }
271 
272     /**
273         Creates HuffmanNodes from the HashMap and puts them into a Priority Queue
274         @param occur The HashMap with letters and occurences
275         @return A priority Queue with the HuffmanNodes in it
276     */
277     private PriorityQueue<HuffmanNode> createNodes(HashMap<String,Integer> occur)
278     {
279         String[] keys = occur.keySet().toArray(new String[0]);
280         PriorityQueue<HuffmanNode> q = new PriorityQueue<HuffmanNode>();
281         for (int i = 0; i < keys.length; i++)
282         {
283             q.add(new HuffmanNode(occur.get(keys[i]), keys[i] ));
284             //System.out.println("hello");
285         }
286         /*
287         for (HuffmanNode node : q)
288         {
289             System.out.println(node);
290         }
291         */
292         return q;
293     }
294 
295     /**
296         Uses the priority Queue with the HuffmanNodes in it to create the Huffman Tree.
```

Having a default constructor of this is a little strange. Shouldn't you enter the phrase in the runner?

```
297         It polls the first two nodes in the queue, combines them,
298         makes that the parent node of the two first nodes, and puts that parent
299         node back into the queue.
300         @param q The Priority Queue that will be used to make the Tree
301         @returns the root of the HuffmanTree created from the Priority Queue
302     */
303     private HuffmanNode createTree(PriorityQueue<HuffmanNode> q)
304     {
305         while (q.toArray().length > 1)
306         {
307             HuffmanNode first = q.poll();
308             HuffmanNode second = q.poll();
309             HuffmanNode parent = new HuffmanNode(first.value() + second.value(), first.letters() + se
310             //System.out.println("parent: " + parent);
311             parent.setLeft(first);
312             parent.setRight(second);
313             q.offer(parent);
314         }
315         //System.out.println(q.peek());
316         return q.poll();
317     }
318 
319     /**
320         Turns the input phrase into a string of 1s and 0s based off
321         of the binary tree
322         @return The string of bits
323 
324     */
325     public String encode()
326     {
327         String code = "";
328         for (int i = 0; i < phrase.length(); i++)
329         {
330             code += findCode("" + phrase.charAt(i), root);
331         }
332         return code;
333     }
334 
335     /**
336         Te recursive helper method to encode
337         @param letter a specific letter being converted to bits
338         @param curr the HuffmanNode that is where the code is focused
339         @return the bit representation of the letter
340     */
341     private String findCode(String letter, HuffmanNode curr)
342     {
343         if (curr.isLeaf())
344         {
345             return "";
346         }
347         if (curr.left().letters().contains(letter))
348         {
349             return "0" + findCode(letter, curr.left());
350         }
351         else
352         {
353             return "1" + findCode(letter,curr.right());
354         }
355 
356     }
357 
358     /**
359         Turns the string of bits back into letters
360         @param bits The string of 1s and 0s that are being turned back
361         into letters using the binary tree
362         @return the phrase
363     */
364     public String decode(String bits)
365     {
366         return decodeHelper(bits, root);
367     }
368 
369     /**
370         The recursive helper function that turns the string of bits back
```

```
371         into letters
372         @param bits The 1s and 0s that are being turned back into a phrase
373         @param curr The huffman node that it is currently on
374         @return the original phrase of letters
375     */
376     private String decodeHelper(String bits, HuffmanNode curr)
377     {
378         if (curr.isLeaf())
379         {
380             if (bits.length() > 0)
381             {
382                 //System.out.print(curr.letters());
383                 return curr.letters() + decodeHelper(bits, root);
384             }
385             else
386                 return curr.letters();
387         }
388         else if (("" + bits.charAt(0)).equals("0"))
389         {
390             return decodeHelper(bits.substring(1), curr.left());
391         }
392         else
393         {
394             return decodeHelper(bits.substring(1), curr.right());
395         }
396     }
397
398
399 }
400
401
402
```

Good work, per usual. All feedback above is related to writing precise comments. Your code works fine and the design is logical.
A+