```java
import java.util.PriorityQueue;
import java.util.HashMap;
/**
Class that tests the methods of HuffmanNode
@author Yanni Angelides
@version 01/29/16
*/

public class HuffmanRunner
{
    public static void main(String [] args)
    {
        HuffmanNode node = new HuffmanNode("test");
        HuffmanNode node2 = new HuffmanNode("test");
        HuffmanNode node3 = new HuffmanNode("wrong");
        HuffmanNode node4 = new HuffmanNode(node, node3);
        System.out.println(node.compareTo(node2));
        System.out.println(node.compareTo(node3));
        HuffmanTree tree = new HuffmanTree("racecar racecar");
        String code = tree.encode("racecar race");
        System.out.println(code);
        System.out.println(tree.decode(code));
        HuffmanTree tree2 = new HuffmanTree("aaaabbc ddeeffgghijkklmnoopq rrsttuvwx xyyyyyyyyz!!!");
        String code2 = tree2.encode("yanni is the best coder ever!");
        System.out.println(code2);
        System.out.println(tree2.decode(code2));

    }
}

import java.util.PriorityQueue;
import java.util.HashMap;

/**
Creating a HumanTree using the HuffmanNode. Huffman Code works by first taking a String,
counting the frequency of each letter in that String and placing all that information in a
map that way you can have random access. Then you take the information in the map and put
it in a priorityQueue. Once the all the nodes are in a priorityQueue then the top two items
on the top Queue (characters with lowest frequency) are combined together and then placed
back into the Queue. The program repeats this process until there is only one node in the
Queue. This last node is then set as the root of the tree. As this process is happening the
nodes are automatically formed into a tree because when two nodes are combined the combination
is used as the parent node and the two other nodes are set to the left and right of that
parent, so in the end everything is connected.
@author Yanni Angelides
@version 01/27/16
*/

//I used a HashMap because the order that the characters are in doesn't really matter because
//they are going to be reordered in the Priority Queue any way. Also, because the program will
//have to look through the whole map to find characters and add to their count there is no point
//of wasting time sorting the tree to make the program faster.

public class HuffmanTree
{
    private HuffmanNode root;
    //first HuffmanNode of the Huffman Tree, start of the tree
    private HashMap<String, Integer> map;
    //map that stores the count values of all the different characters in the master String, stores ho
    private PriorityQueue<HuffmanNode> que;
    //sorts the values in the HashMap in an order from least to greatest so that they can be added to

    //Constructor
    public HuffmanTree(String str)
    {
        createTree(str);
    }

    /**
    Uses helper methods to create map of the values and count of the different characters in sentence,
    @param String sentence, master String that is going to be used to create the Huffman Tree
    */
    public void createTree(String sentence)
    {
```

Comments should go above the code they refer to.

Do you really need map and que to be class fields?

Why do you need a helper method?

```
75          this.makeMap(sentence);
76          this.makeQueue();
77          this.makeTree();
78          //Three methods above are helper methods used to create the tree
79      }
80
81      /**
82      method that initializes the string and iterates through the initial sentence to find all the chara
83      @param String str, master String that is passed into the createTree method
84      @return
85      */
86      public void makeMap(String str)
87      {
88          map = new HashMap<String, Integer>();
89          for(int i = 0; i < str.length(); i++)
90          {
91              String let = "" + str.charAt(i);
92              //converting the first character of the String into a String so it is easier to work with
93              if(map.containsKey(let))
94              {
95                  int save = map.get(let);
96                  map.remove(let);
97                  map.put(let, save + 1);
98                  //there is no set method for HashMaps so you have to remove the old HashMap node add c
99              }
100             else
101             {
102                 map.put(let, 1);
103             }
104         }
105     }
106
107     /**
108     Creates a Queue out of the different nodes in the HashMap
109     */
110     public void makeQueue()
111     {
112         que = new PriorityQueue<HuffmanNode>();
113         Object[] arr = map.keySet().toArray();
114         //keySet method for HashMap returns a Set of the Key values in the HashMap then the toArray me
115         for (int i = 0; i < arr.length; i++)
116         {
117             HuffmanNode node = new HuffmanNode(arr[i].toString(), map.get(arr[i]));
118             que.offer(node);
119         }
120     }
121
122     /**
123     Creates a type of BinaryTree out of HuffmanNodes with root as the start of the tree
124     */
125     public void makeTree()
126     {
127         while (que.size() > 1)
128         {
129             HuffmanNode node1 = que.poll();
130             HuffmanNode node2 = que.poll();
131             //takes two nodes out of the que and puts them in the tree
132             HuffmanNode combo = new HuffmanNode(node1, node2);
133             //combines the two nodes together using a specialized constructor from the HuffmanNode cla
134             que.offer(combo);
135             //puts the now combined node back into the Queue
136         }
137         root = que.poll();
138         //last object in the Queue is used as the start of the tree
139     }
140
141     /**
142     Creates a String of 1s and 0s that represents a the String parameter based upon the original Huffm
143     @param String str, String that is passed in that needs to be encoded
144     @return String of ones and zeros that is based upon the tree created with the master String that a
145     */
146     public String encode(String str)
147     {
148         String code = "";
```

put works the same as set, no?

You could just traverse over the keyset

```
149         for (int i = 0; i < str.length(); i++)
150         {
151             code += helper(""+str.charAt(i), root);
152             //calls helper method that actually encodes each individual letter of the String
153         }
154         return code;
155     }
156 
157     /**
158     Helper method that takes in two Strings, String a being a letter and String b being an combination
159     @param String a, one letter String, String b combination of letters that is being checked
160     @return boolean determining if String b contains String a
161     */
162     public boolean contains(String a, String b)
163     {
164         for(int i = 0; i < b.length(); i++)
165         {
166             if(a.equals(""+b.charAt(i)) == true)
167             {
168                 return true;
169             }
170         }
171         return false;
172     }
173 
174     /**
175     Helper method that takes in a String and iterates through the HuffmanTree to track a path to that
176     @param String str, that is being checked for, HuffmanNode n, that is being checked for that indivi
177     @return
178     */
179     public String helper(String str, HuffmanNode n)
180     {
181         if(n.getRight() == null && n.getLeft() == null)
182         //if the the iterator hits a "Leaf" or a node where both left and right values are null it mea
183         {
184             return "";
185         }
186         if(contains(str, n.getRight().getValue()))
187         //if the iterator sees that the letter it is looking for is contained in the right value of th
188         {
189             return "1" + helper(str, n.getRight());
190         }
191         else
192         //if the iterator sees that the letter it is looking for is contained in the left value of the
193         {
194             return "0" + helper(str, n.getLeft());
195         }
196     }
197 
198     /**
199     Takes in a String of ones and zeros and using the HuffmanTree creates a new string of actual lette
200     @param String str, of ones and zeros that code for a message
201     @return String, that the 1s and 0s encode for
202     */
203     public String decode(String str)
204     {
205         int i = 0;
206         String code = "";
207         while(i < str.length())
208         //makes sure the method iterates through all the 1s and 0s in the str parameter
209         {
210             HuffmanNode node = root;
211             //resets the node variable to the start of the tree
212             while(node.getRight() != null && node.getLeft() != null)
213             //will keep iterating through the tree until it hits a "leaf" node where both left and rig
214             {
215                 String let = "" + str.charAt(i);
216                 //variable that iterates through all the 1s and 0s
217                 if(let.equals("1"))
218                 {
219                     node = node.getRight();
220                     //Based on the encode method if there is a one in the String then the the code sho
221                 }
222                 else
```

Good

```
                {
                    node = node.getLeft();
                    //if let = 0 should iterate to the left
                }
                i++;
            }
            code += node.getValue();
            //when both left and right are null the iterator should be at the letter it is looking for
        }
        return code;
    }
}

import java.util.PriorityQueue;
import java.util.HashMap;
/**
Creating Huffman Code Program
@author Yanni Angelides
@version 01/27/16
*/

public class HuffmanNode implements Comparable
{
    protected HuffmanNode left;
    //daughter HuffmanNode that is contained in a memory space to the "left" of this one or the parent
    protected HuffmanNode right;
    //daughter HuffmanNode that is contained in a memory space to the "right" of this one or the paren
    protected String value;
    //String the the Huffman node stores, its actual value
    protected int count;
    //tracks the amount of times the value of the HuffmanNode appears in the master String

    /**
    Basic constructor that just sets the value of the HuffmanNode to a specified String
    @param String val, specified value for the value class field
    */
    public HuffmanNode(String val)
    {
        left = null;
        right = null;
        value = val;
        count = 1;
    }

    /**
    Combines the values and counts of two HuffmanNode and makes them into one single HuffmanNode
    @param HuffmanNode a, HuffmanNode b two HuffmanNodes that are being combined
    */
    public HuffmanNode(HuffmanNode a, HuffmanNode b)
    {
        left = a;
        right = b;
        value = a.getValue() + b.getValue();
        count = a.getCount() + b.getCount();
    }

    /**
    Constructor that sets both the value and the count to the specified parameter
    @parameter String val, String that value is set to, int c, int count is set to
    */
    public HuffmanNode(String val, int c)
    {
        left = null;
        right = null;
        value = val;
        count = c;
    }

    /**
    Gets the count of the specified HuffmanNode
    @return int, value of the count
    */
    public int getCount()
    {
```

```
297         return count;
298     }
299 
300     /**
301     Sets the value of the count
302     @param int num, count is set to
303     */
304     public void setCount(int num)
305     {
306         count = num;
307     }
308 
309     /**
310     Gets the value of the specified HuffmanNode
311     @return String, value of the HuffmanNode
312     */
313     public String getValue()
314     {
315         return value;
316     }
317 
318     /**
319     Sets the value to the specified String
320     @param String str, that value is set to
321     */
322     public void setValue(String str)
323     {
324         value = str;
325     }
326 
327     /**
328     gets the value of the HuffmanNode to the left
329     @return value of the HuffmanNode to the left
330     */
331     public HuffmanNode getLeft()
332     {
333         return left;
334     }
335 
336     /**
337     gets the value of the HuffmanNode to the right
338     @return value of the HuffmanNode to the right
339     */
340     public HuffmanNode getRight()
341     {
342         return right;
343     }
344 
345     /**
346     Sets the value of the HuffmanNode on the left
347     @param HuffmanNode node, that the left value is set to
348     */
349     public void setLeft(HuffmanNode node)
350     {
351         left = node;
352     }
353 
354     /**
355     Sets the value of the HuffmanNode on the right
356     @param HuffmanNode node, that the right value is set to
357     */
358     public void setRight(HuffmanNode node)
359     {
360         right = node;
361     }
362 
363     /**
364     Compares two HuffmanNodes based on their count value, if the counts are the same zero is returned
365     @param object node, that is being compared
366     */
367     public int compareTo(Object node)
368     {
369         return count - (((HuffmanNode)node).getCount());
370     }
```

```
371
372     /**
373     Creates a String representation of a HuffmanNode
374     @return String, representation of a HuffmanNode
375     */
376     public String toString()
377     {
378         String str = "Value: " + value + " Count: " + count;
379         return str;
380     }
381 }
```

Strange design in a few places, but overall good execution.
A