```java
public class HuffmanRunner
{
    public static void main(String[] args)
    {
        HuffmanNode test = new HuffmanNode("b", 5);
        test.setValue("c");
        test.setFrequency(6);
        System.out.println(test.value());
        System.out.println(test.frequency());
        HuffmanNode test2 = new HuffmanNode("d", 10);
        System.out.println(test.compareTo(test2));
        HuffmanTree tree = new HuffmanTree("The Bluebook bi
        System.out.println(tree.encode("The"));
        System.out.println(tree.decode("1110001111110"));
        System.out.println(tree);
    }
}

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


 /*
 @author Madeline Temares
 I used HashMap because it is more efficient. TreeMap can be sorted, but we do not need
 that function because that wouldn't help. We need to search through all of them to see if
 the character already exists in it so there is no way that it could be sorted that would
 make it worth it.

 I explained how it works broken up into all of the methods below. In short, it calls init,
 which calls the 3 helper methods. First it makes a map with each of the characters and their
 frequencies. Then it makes a priorityqueue with those keys and values. And then it takes
 that priorityqueue and does the trickiest part. It takes the lowest frequencies --
 top of the priorityqueue ~~i believe~~ -- and it combines them into a huffmanNode with the 2
 individual ones as the left and right -- it then puts that combined node into the
 priorityqueue in place of the other 2 in the right spot. It goes on and on until all of the
 nodes are connected and every character is a leaf at the bottom of the tree.

 */

public class HuffmanTree
{
    private HuffmanNode root; //null in constructor but then set to the top of the tree once the prior
    private String sentence;
    private HashMap<Character,Integer> map;
    private PriorityQueue<HuffmanNode> queue;

    /*
    Constructor - takes in a string - the opening sentence that the tree will be made with
    It calls the helper method init which calls all the other methods to make the tree
    @param s String that the tree is made with
    */

    public HuffmanTree(String s)
    {
        sentence = s;
        root = null;
        init(); //calls first helper method
    }

    /*
    Helper method that calls the 3 other helper methods
    @return void
    */
    private void init()
    {
        makeMap();
        makeQueue();
        makeTree();
    }

    /*
    Initializes to a map. It puts each character with its frequency into a map. Its a loop
    that if the character already exits, it gets that character and puts a frequency of one
```

Name?

So this tests if encode and decode are consistent, but how do you know the tree is correct? I tested against a tree with a few letters that had a high frequency and did get back appropriate encode lengths.

Using HashMap is fine, but if it were sorted, couldn't you do binary search?

Why the helper method? What is it helping with?

Initializes what to a map?

```
    more. If its not there, it just adds the character with a frequency of one
    @return void
    */
    private void makeMap()
    {
        map = new HashMap<Character, Integer>();
        for (int i = 0; i<sentence.length(); i++)
        {
            char c = sentence.charAt(i);
            if (map.containsKey(c))
            {
                int x = map.get(c);
                map.put(c, x+1);
            }
            else
            {
                map.put(c, 1);
            }
        }
        this.map = map;
    }


    /*
    Adds every character from the map into a priority queue of huffmanNodes
    The priority queue sorts them according to their natural sorting, which is based
    on the compareTo method
    @return void
    */
    private void makeQueue()
    {
        queue = new PriorityQueue<HuffmanNode>();
        for (char c: map.keySet())
        {
            HuffmanNode node = new HuffmanNode(String.valueOf(c), map.get(c));
            queue.add(node);
        }
    }

    /*
    Takes the priority queue and changes everything in it into linked HuffmanNodes - it links them
    by setting the left and right of each node
    @return void
    */
    private void makeTree()
    {
        HuffmanNode n;
        while (queue.size() > 1)
        {
            n = new HuffmanNode(queue.poll(), queue.poll());
            queue.add(n);
            root = n;

        }
    }



    /*
    Precondition: letters in encoding message are in the tree
    Encodes a message with the use of the HuffmanTree
    For every letter that you want to encode, it calls the encodeHelp helper method with the
    root and that character
    @param String s The sentence or word that you are encoding using the tree
    @return String String value of 1s and 0s encoded
    */
    public String encode(String s)
    {
        String output = "";
        for (int i = 0; i<s.length(); i++)
        {
            char c = s.charAt(i);
            output += encodeHelp(root, c);
```

What is their natural sorting?

```
149         }
150
151         return output;
152     }
153
154
155     /*
156     Helper method for encode
157     It is recursive - it takes in where you are in the tree which is how it goes to the left or the ri
158     @param HuffmanNode Where you start in the tree
159     @return char Character you are encoding
160     */
161     public String encodeHelp(HuffmanNode r, char c)
162     {
163         HuffmanNode h = r;
164         if (h.isLeaf())
165         {
166             return "";
167         }
168         HuffmanNode left = h.left();
169         HuffmanNode right = h.right();
170         for (int x = 0; x<left.value().length(); x++)
171         {
172             char z = left.value().charAt(x);
173             if (c == z)
174             {
175
176                 return "0" + encodeHelp(left, c);
177             }
178         }
179
180         for (int y = 0; y<right.value().length(); y++)
181         {
182             char zz = right.value().charAt(y);
183             if (c == zz)
184             {
185
186                 return "1" + encodeHelp(right, c);
187             }
188         }
189         return "";
190
191     }
192
193     /*
194     Decodes the message you are given in 1s and 0s according to the HuffmanTree
195     For every 1, it goes to the right and for every 0, it goes to the left
196     @param String s String representation of 1s and 0s
197     @return String String representation of encoded message
198     */
199     public String decode(String s)
200     {
201         String output = "";
202         HuffmanNode h = root;
203         HuffmanNode left;
204         HuffmanNode right;
205         for (int i =0; i <s.length(); i++)
206         {
207             char c = s.charAt(i);
208             if (c == '0')
209             {
210                 if (h.left().isLeaf())
211                 {
212                     output+= h.left().value();
213                     h = root;
214                 }
215                 else
216                 {
217                     h = h.left();
218                 }
219             }
220             else if (c== '1')
221             {
222                 if (h.right().isLeaf())
```

A lot of the header comments you write could be dispersed into the code, so they directly proceed the piece of the algorithm they describe.

Also, explain how it goes back to the top once it decodes a letter.

```
                {
                    output+= h.right().value();
                    h = root;
                }
                else
                {
                    h = h.right();
                }
            }
        }
        return output;
    }


    /*
    toString method - makes use of the node toString method - call it on the root
    @return String String representation of tree
    */
    public String toString()
    {
        return root.toString();
    }
}

/*
@author Maddie Temares

HuffmanNode is the node for each element in the HuffmanTree. It holds the frequency and value
of each letter and then each set of combined characters when the tree is being formed
upwards. A HuffmanNode has a left and right value, similar to binary tree, because the values
that the huffmanNode holds has subtrees. For example, a huffmannode with value "sf" would have
subtrees of "s" and "f".

*/

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


public class HuffmanNode implements Comparable
{
    protected int frequency;
    protected String value;
    protected HuffmanNode left;
    protected HuffmanNode right;

    /*
    Constructor - sets left and right to null - these would be leaf nodes
    @param v Value
    @param f Frequency
    */
    public HuffmanNode(String v, int f)
    {
        value = v;
        frequency = f;
        left = null;
        right = null;
    }

    /*
    Constructor that makes new node with 2 existing nodes as its left and right
    @param l Left node
    @param r Right node
    */
    public HuffmanNode(HuffmanNode l, HuffmanNode r)
    {
        left = l;
        right = r;
        value = l.value() + r.value();
        frequency = l.frequency() + r.frequency();
    }

    /*
    Accessor for frequency
```

```
297     @return int Frequency
298     */
299     public int frequency()
300     {
301         return frequency;
302     }
303 
304     /*
305     Modifier for frequency
306     @param f Frequency to be set
307     @return void
308     */
309     public void setFrequency(int f)
310     {
311         frequency = f;
312     }
313 
314     /*
315     Accessor for value
316     @return String Frequency
317     */
318     public String value()
319     {
320         return value;
321     }
322 
323     /*
324     Modifier for value
325     @param v Value to be set
326     @return void
327     */
328     public void setValue(String v)
329     {
330         value = v;
331     }
332 
333     /*
334     Accessor for left
335     @return HuffmanNode left
336     */
337     public HuffmanNode left()
338     {
339         return left;
340     }
341 
342     /*
343     Modifier for left
344     @param l Left huffmannode to be set
345     @return void
346     */
347     public void setLeft(HuffmanNode l)
348     {
349         left = l;
350     }
351 
352     /*
353     Accessor for right
354     @return HuffmanNode right
355     */
356     public HuffmanNode right()
357     {
358         return right;
359     }
360 
361     /*
362     Modifier for right
363     @param l Right huffmannode to be set
364     @return void
365     */
366     public void setRight(HuffmanNode r)
367     {
368         right = r;
369     }
370 
```

```
371     /*
372     CompareTo method - here so that the priorityqueue knows how to sort the huffmannodes
373     It compares the huffmannodes with their frequencies
374     @param o Object it is being compared to
375     @return int Positive if frequency is greater and negative if frequency is less
376     */
377     public int compareTo(Object o)
378     {
379         return frequency - ((HuffmanNode)o).frequency();
380     }
381
382     /*
383     Checks to see if the huffmanNode is a leaf - if there are no nodes to the left and right
384     It only needs to check if the left is null because the way the nodes are made, they will either
385     have both left and right values or both be null - it cannot be one or the other
386     @return boolean Whether or not huffmannode is a leaf
387     */
388     public boolean isLeaf()
389     {
390         if (null == left())
391         {
392             return true;
393         }
394         return false;
395     }
396
397
398     /*
399     toString method
400     @return String String rep of node
401     */
402     public String toString()
403     {
404         if (isLeaf())
405         {
406             return value();
407         }
408         else
409         {
410             return value()+ "\n" + left.toString() + right.toString();
411         }
412     }
413 }
414
```

Works well. See my notes above regarding commenting and your runner class. Grade: A/A+