```
/**
*   Brendan Raimann
*   2/6/16
*   Version 1.0
*   HuffmanRunner Class - Used for running the HuffmanTree and HuffmanNode classes
*/

public class HuffmanRunner
{
    /** Main method */
    public static void main (String[] args)
    {
        String s = "This is a test.";
        HuffmanTree tree = new HuffmanTree(s);
        System.out.println(tree.encode());
        //  returns 1001111011110100111101001000001011101011011100
        System.out.println(tree.decode("1001111011110100111101001000001011101011011100"));

    }
}
```

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

```
/**
*   Brendan Raimann
*   2/6/16
*   Version 1.0
*   HuffmanTree Class - Used for building a binary tree to compact data
*/
```

You should elaborate.

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

public class HuffmanTree
{
    /** Pointer to the root node */
    private HuffmanNode root;

    /** Contains the string for encoding */
    private String str;
```

Why is this a private variable?

```
    /** Constructor that builds the binary tree
    *   @param s The string of letters to be used to build a Huffman Tree
    */
    public HuffmanTree(String s)
    {
        str = s;
        root = buildTree(buildQueue(createMap(s)));
    }

    /**
    *   Takes in a string and turns it into a HashMap
    *   @param s A string to be turned into a HashMap
    */
    private HashMap<String,Integer> createMap(String s)
    {
        HashMap<String, Integer> map = new HashMap<String, Integer>();
        for (int i = 0; i < s.length(); i++)
        {
            if (map.containsKey(s.substring(i,i+1)))
                map.put(s.substring(i,i+1),map.get(s.substring(i,i+1))+1);
            else
                map.put(s.substring(i,i+1),1);
        }
        return map;
    }

    /**
    *   Builds a PriorityQueue of HuffmanNode's using a HashMap
    *   @param map A HashMap<String, Integer> That stores a letter and its number of occurrences
    *   @return A PriorityQueue<HuffmanNode> that stores nodes with letters and their frequencies
    */
    private PriorityQueue<HuffmanNode> buildQueue(HashMap<String,Integer> map)
    {
        PriorityQueue<HuffmanNode> queue = new PriorityQueue<HuffmanNode>();

```

```java
        for (String s: map.keySet())
        {
            queue.add(new HuffmanNode(s, map.get(s)));
        }
        return queue;
    }

    /**
    *   Builds a binary tree using a PriorityQueue<HuffmanNode>
    *   @param queue A PriorityQueue that stores nodes for the tree
    *   @return Returns the root node
    */
    private HuffmanNode buildTree(PriorityQueue<HuffmanNode> queue)
    {
        HuffmanNode node;
        HuffmanNode head1;
        HuffmanNode head2;
        while (queue.size() > 1)
        {
            head1 = queue.remove();
            head2 = queue.remove();
            node = new HuffmanNode(head1.getString() + head2.getString(), head1.getFreq() + head2.getF
            node.setLeft(head1);
            node.setRight(head2);
            queue.add(node);
        }
        return queue.remove();

    }

    /**
    *   Used to encode the String in the class field to binary digits
    *   @return Returns a series of binary digits in a String
    */
    public String encode()
    {
        String output = "";
        String temp = "";
        HuffmanNode index;
        for (int i = 0; i < str.length(); i++)
        {
            index = root;
            while (index.isLeaf() == false)
            {
                if (index.getLeft().getString().indexOf(str.substring(i,i+1)) >= 0)
                {
                    index = index.getLeft();
                    temp += "0";
                }
                else
                {
                    index = index.getRight();
                    temp += "1";
                }

            }
            output += temp;
            temp = "";
        }
        return output;
    }

    /**
    *   Used to decode
    *   @param s Binary digits in a String
    *   @return Returns the result of decoding the binary digits into symbols using the tree
    */
    public String decode(String s)
    {
        String output = "";
        HuffmanNode index = root;
        for (int i = 0; i < s.length(); i++)
        {
            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.

```
            {
                output += index.getString();
                index = root;
                i--;
            }
            else
            {
                if (s.charAt(i) == '0')
                    index = index.getLeft();
                else
                    index = index.getRight();
            }
        }
        output += index.getString();
        return output;
    }
}

/**
*   Brendan Raimann
*   2/6/16
*   Version 1.0
*   HuffmanNode Class - Used as nodes for a binary tree
*/

public class HuffmanNode implements Comparable<HuffmanNode>
{
    /** The frequency of the String */
    private int freq;
    /** The stored String for the node */
    private String key;
    /** Pointer to the left node */
    private HuffmanNode left;
    /** Pointer to the right node */
    private HuffmanNode right;

    /**
    *   Constructor that builds the node
    *   @param s The String to be stored in the node
    *   @param n The number of occurrences for the String
    */
    public HuffmanNode(String s, int n)
    {
        key = s;
        freq = n;
        left = null;
        right = null;
    }

    /**
    *   Returns the String of the node
    *   @return Returns the stored String
    */
    public String getString()
    {
        return key;
    }

    /**
    *   Returns the frequency of the String
    *   @return Returns the number of occurrences for the String
    */
    public int getFreq()
    {
        return freq;
    }

    /**
    *   Returns the left node pointer
    *   @return Returns the pointer to the left node
    */
    public HuffmanNode getLeft()
    {
        return left;
```

```java
    }

    /**
    *   Returns the right node pointer
    *   @return Returns the pointer to the right node
    */
    public HuffmanNode getRight()
    {
        return right;
    }


    /**
    *   Sets the left node
    *   @param node The node for the left pointer
    */
    public void setLeft(HuffmanNode node)
    {
        left = node;
    }

    /**
    *   Sets the right node
    *   @param node The node for the right pointer
    */
    public void setRight(HuffmanNode node)
    {
        right = node;
    }

    /**
    *   Returns whether or not the node has left and right pointers
    *   @return Returns true of the left and right pointers are null
    */
    public boolean isLeaf()
    {
        if (left == null && right == null)
            return true;
        return false;
    }

    /**
    *   Allows for comparing nodes
    *   @param node Another node for comparison
    *   @return Returns the difference in frequency between the nodes
    */
    public int compareTo(HuffmanNode node)
    {
        return freq - node.freq;
    }

    /**
    *   Returns a String representation of the node and its children
    *   @returns A String representation of the node
    */
    public String toString()
    {
        String value = "[ " + key + ", " + freq + "]";
        if (isLeaf() == true)
            return value;
        else
        {
            if (left != null && right == null)
                return value + "(" + left.toString() + ",)";
            if (left == null && right != null)
                return value + "(," + right.toString() + ")";
            return value + "(" + left.toString() + "," + right.toString() + ")";
        }
    }

}
```

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+