```java
/**
 * HuffmanRunner class.
 *
 * @author  Jessica Li
 * @version 02/07/16
 */
public class HuffmanRunner
{
    /**
     * Main method, tests HuffmanTree.
     * Creates new HuffmanTree, encodes String and prints result, decodes encoded String and prints re
     *
     * @param args  main method parameter
     */
    public static void main( String[] args )
    {
        //HuffmanNode law = new HuffmanNode( "key", 2 );
        System.out.println();
        HuffmanTree hTree = new HuffmanTree( "Maya Huffman huffs at Huffman trees." );
        //hTree.printPQueue();
        System.out.println();
        String code = hTree.encode( "Maya Huffman huffs at Huffman trees." );
        System.out.println( code + "\n");
        System.out.println( hTree.decode( code ) );
    }
}

import java.util.Map;
import java.util.HashMap;
import java.util.PriorityQueue;
import java.util.Set;

/**
 * Huffman Tree class.
 * I chose to import HashMap rather than TreeMap because the stored values do not need to be sorted, a
 * HashMap also has faster runtime for put and peek.
 * Huffman Code works by taking in a sentence and putting each letter of the sentence into a map with
 * map are converted into HuffmanNodes and transferred into a priority queue, with the elements with t
 * To create the tree, it takes the first two elements out of the priority queue, connects them to a n
 * nodes, and reinserts the new node into its proper place in the priority queue. This continues unti
 * Encode traverses through the tree, recording the path of each letter with 0 and 1 for left and righ
 * Decode reads the number, going left for 0 and right for 1. Once it hits a leaf node, it records tha
 * until the entire number is read.
 *
 * @author  Jessica Li
 * @version 02/07/16
 */
public class HuffmanTree
{
    private HashMap<String, Integer> hmap;
    //pQueue is initialized in the method intoPQueue()
    private PriorityQueue<HuffmanNode> pQueue;
    //root is initialized in the method buildTree()
    private HuffmanNode root;
    private String sentence;

    /**
     * Constructor for HuffmanTree that takes in a sentence that will be used to create the tree and e
     * Initializes the map and sentence. Creates the map and priority queue, builds the tree.
     *
     * @param s     string for the original sentence
     */
    public HuffmanTree( String s )
    {
        hmap = new HashMap<String, Integer>();
        sentence = s;
        intoMap( sentence );
        intoPQueue();
        buildTree();
    }

    /**
     * Converts String into map of all values.
     * For loop traverses through the string sentence, adding each character to the map. If the map al
```

```
     * replaced with the same key but frequency is increased by 1. Otherwise, a new entry with that ch
     *
     * @param s     the original sentence
     */
    private void intoMap( String s )
    {
        for ( int i = 0; i < s.length(); i++ )
        {
            String letter = s.substring(i, i+1 );
            if ( hmap.containsKey( letter ) )
            {
                //oh my god bane of my code -- why can't I have ++ in the parameter? why do it outside
                //this marks where I started making progress on debugging. Before I discovered nothing
                int frequency = hmap.get( letter ) + 1;
                hmap.put( letter, frequency );
            }
            else
                hmap.put( letter, 1 );
        }
    }

    /**
     * Transfers elements from the map to a priority queue. Gets all the keys in map as a Set. Change
     * priority queue. Loop iterates for each element in map, adding new HuffmanNodes to the priority
     * the array. Type casts from Object to String.
     */
    private void intoPQueue()
    {
        Set k = hmap.keySet();
        Object[] keys = k.toArray();
        pQueue = new PriorityQueue<HuffmanNode>();

        //high priority, low frequency
        //find and place low frequency nodes in first
        //all in loop
        for ( int i = 0; i < hmap.size(); i++ )
        {
            pQueue.add( new HuffmanNode( (String) keys[i], hmap.get( (String) keys[i] ) ) );
        }
        //after the loop, the pQueue should be finished, with all the nodes in the proper order. It or
    }

    /**
     * Prints the priority queue in order. For testing purposes.
     * Loop traverses through priority queue. Removes the top HuffmanNode, prints the value and count.
     * queue is also decreasing by one.
     */
    public void printPQueue()
    {
        //System.out.println( pQueue.size() );
        for ( int i = 0; i < pQueue.size(); i++ )
        {
            HuffmanNode n = pQueue.poll();
            System.out.println( "V: " + n.getValue() + "     C: " + n.getCount() );
            i--;
        }
    }


    /**
     * Builds the Huffman Tree from the priority queue. While the size of the priority queue is greate
     * creates a new HuffmanNode with combined values and counts, and adds new Node to priority queue.
     * fully built. Sets root to the only remaining HuffmanNode in the priority queue, which is the to
     * because I want to be able to see a visual representation of the tree when I run the code.
     */
    private void buildTree()
    {
        while ( pQueue.size() > 1 )
        {
            HuffmanNode n1 = pQueue.poll();
            HuffmanNode n2 = pQueue.poll();
            HuffmanNode combinedNode = new HuffmanNode( n1.getValue() + n2.getValue(), n1.getCount() +
            pQueue.offer( combinedNode );
        }
```

useful

```java
        //peek or poll? both work?
```

?

```java
        //System.out.println( pQueue.peek() );
        root = pQueue.peek();
        System.out.println( root );
    }

    /**
     * Encodes String s. String code stores the sequence of 0's and 1's. For loop traverses through St
     * with root and a single character as parameters, adding the returned value to code. Once the loo
     *
     * @param s     String to be encoded
     * @return      encoded sequence of 0's and 1's
     *
     */
    public String encode( String s )
    {
        //System.out.println( root );
        String code = "";
        for ( int i = 0; i < s.length(); i++ )
        {
            code += encodeHelper( root, s.substring( i, i+1 ) );
        }
        return code;
    }

    /**
     * Helper method to encode() that returns the encoded value of a single character. Recursive.
     * Base case: if the HuffmanNode taken in is a leaf and its value equals the letter taken in, then
     * If the node's left child contains the letter in its value, returns 0 + recursive call with left
     * If the node's right child contains the letter in its value, returns 1 + recursive call with rig
     *
     * @param n     current HuffmanNode, used for traversing through the tree
     * @param l     letter/character to be encoded
     * @return      encoded String for a particular letter
     */
    private String encodeHelper( HuffmanNode n, String l )
    {
        //System.out.println(n);
        //base case for n is leaf
        if ( n.isLeaf() && n.getValue().equals( l ) )
        {
            //System.out.println("leaf");
            return "";
        }
```

Commenting a little cluttered, but good solution.

```java
        //System.out.println( n.getLeft().getValue() );
        //System.out.println( n.getRight().getValue() );
        if ( n.getLeft().getValue().contains( l ) )
        {
            //System.out.println( n.getLeft().getValue() );
            return "0" + encodeHelper( n.getLeft(), l );
        }
        //I don't need all the code in the line below, but for clarity's sake
        else if ( n.getRight().getValue().contains( l ) )
        {
            //System.out.println( n.getRight().getValue() );
            return "1" + encodeHelper( n.getRight(), l );
        }
        //if neither left nor right paths contain l -- but that's not possible right, it must be conta
        //System.out.println( "out of if" );
        return "";
    }

    /**
     * Decodes String code. Calls helper method decodeHelper with root and code as parameters.
     *
     * @param code  the code to be decoded
     * @return      the String for which code translates
     *
     */
    public String decode( String code )
    {
        return decodeHelper( root, code );
```

```java
    }

    /**
     * Helper method to decode() that takes in a HuffmanNode and code as parameters. Recursive.
     * Base case: If n, or the current node, is a leaf and the length of code is 0, returns the value
     * If n is a leaf and the length of code is not 0, returns the value of n + recursive call with ro
     * the next node that the code refers to.
     * If the first character in code equals 0, then sets n to n's left child, sets code to same Strin
     * recursive call with n, code as param. Otherwise (if first char equals 1), sets n to n's right c
     * character, and returns recursive call with n, code as param.
     *
     * @param n     the current HuffmanNode, used to traverse the tree
     * @param code  the code to be decoded
     * @return      the String that for which the code translates
     */
    public String decodeHelper( HuffmanNode n, String code )
    {
        if ( n.isLeaf() )
        {
            //System.out.println( "LEAF" );
            if ( code.length() == 0 )
            {
                return n.getValue();
            }
            else
            {
                return n.getValue() + decodeHelper( root, code );
            }
        }
        if ( code.substring(0,1).equals("0") )
        {
            n = n.getLeft();
            code = code.substring( 1 );
            return decodeHelper( n, code );
        }
        else
        {
            n = n.getRight();
            code = code.substring( 1 );
            return decodeHelper( n, code );
        }
    }

}

/**
 * HuffmanNode class.
 *
 * @author  Jessica Li
 * @version 02/07/16
 */
public class HuffmanNode implements Comparable<HuffmanNode>
{
    private String value;
    private int count;
    private HuffmanNode left;
    private HuffmanNode right;

    /**
     * Constructor for HuffmanNode, gives value and count. Sets left and right to null.
     *
     * @param v     string for the value
     * @param c     int for frequency
     */
    public HuffmanNode( String v, int c )
    {
        value = v;
        count = c;
        left = null;
        right = null;
    }

    /**
     * Constructor for HuffmanNode, sets value, count, left, and right.
```

```java
     *
     * @param v     string for the value
     * @param c     int for frequency
     * @param l     left node
     * @param r     right node
     */
    public HuffmanNode( String v, int c, HuffmanNode l, HuffmanNode r )
    {
        value = v;
        count = c;
        left = l;
        right = r;
    }

    /**
     * Accessor for value.
     *
     * @return value
     */
    public String getValue()
    {
        return value;
    }

    /**
     * Modifier for value.
     *
     * @param v     new value
     */
    public void setValue( String v )
    {
        value = v;
    }

    /**
     * Accessor for count.
     *
     * @return value
     */
    public int getCount()
    {
        return count;
    }

    /**
     * Modifier for count.
     *
     * @param v     new count
     */
    public void setCount( int c )
    {
        count = c;
    }

    /**
     * Accessor for left.
     *
     * @return value
     */
    public HuffmanNode getLeft()
    {
        return left;
    }

    /**
     * Modifier for left.
     *
     * @param v     new left
     */
    public void setLeft( HuffmanNode l )
    {
        left = l;
    }

```

```java
    /**
     * Accessor for right.
     *
     * @return value
     */
    public HuffmanNode getRight()
    {
        return right;
    }

    /**
     * Modifier for right.
     *
     * @param v     new right
     */
    public void setRight( HuffmanNode r )
    {
        right = r;
    }

    /**
     * Compares the frequency of each node, returning positive if greater, 0 if equal, and negative i
     * current node.
     *
     * @param node      the node being compared to
     */
    public int compareTo( HuffmanNode node )
    {
        return count - node.getCount();
    }

    /**
     * Returns true if node does not have any children - if left and right are both null.
     *
     * @return      true if left and right are both null, false otherwise
     */
    public boolean isLeaf()
    {
        return left == null && right == null;
    }

    /**
     * Returns the number of descendants of node, including the current node.
     * If both left and right nodes are null, returns 1. If neither nodes are null, returns 1 + the s
     * If only left is null, returns 1 + the size of right. If only right is null (the else case), re
     *
     * @return      the size of the tree, or the total number of nodes within the tree
     */
    public int size()
    {
        if ( left == null && right == null )
            return 1;
        else if ( left != null && right != null )
            return 1 + left.size() + right.size();
        else if ( left == null && right != null )
            return 1 + right.size();
        else
            return 1 + left.size();
    }


    /**
     * Recursive method that returns String representation of tree, separated by parentheses and comm
     *
     * @return a String representation of the tree.
     */
    public String toString()
    {
        if ( left == null && right == null )
            return value.toString();                    //does this need toString()???
        if ( left == null && right != null )
            return value.toString() + "( , " + right.toString() + ")";
        if ( left != null && right == null )
            return value.toString() + "(" + left.toString() + ", )";
```

Do you actually need all these methods for HuffmanNode?

```java
        else
            return value.toString() + "(" + left.toString() + ", " + right.toString() + ")";
    }

    /**
     * Returns the maximum path length to a descendent.
     * Two overall if cases - one for if left is null, one for if left is not null. Within these two
     * If both left and right are null, returns 1. If only left or right is null, returns 1 + the hei
     * If neither left not right is null, returns 1 + the maximum of the heights of both nodes.
     *
     * @return      the height of the tree, or the maximum path length to a descendent
     */
    public int height()
    {
        //if math.max or if statements
        //just return 1+

        if ( left == null )
        {
            //if both left and right are null
            if ( right == null )
                return 1;
            //if only left is null
            else
                return 1 + right.height();
        }
        else
        {
            //if only right is null
            if ( right == null )
                return 1 + left.height();
            //if neither left nor right is null
            else
                return 1 + Math.max( left.height(), right.height() );
        }

    }


    /**
     * Returns true if adding a node to tree would increase its height - or in other words, if the tr
     * If both left and right nodes are null, returns true. If only the left or right node is null, r
     * If neither left nor right is null, checks if left and right are full. If both are full, checks
     * If both have the same height, returns true. Otherwise, returns false.
     *
     * @return      true if tree is full (if adding a node to tree would increase its height), false
     */
    public boolean isFull()
    {
        //check to see if null
        //check to see if height is same
        if ( left == null && right == null )
            return true;
        else if ( left == null || right == null )
            return false;
        else
        {
            if ( left.isFull() && right.isFull() )
            {
                if ( left.height() == right.height() )
                    return true;
            }
            return false;
        }
    }

    /**
     * Returns true if tree has minimal height and any holes in the tree appear in the last level to
     * If both left and right are null, returns true. If left is full, right is complete, and the hei
     * returns false. If left is complete, right is full, and the height difference is one, returns t
     * If none of the if statements execute, returns false.
     *
     * @return      true if tree has no holes, false otherwise
     */
```

```java
    public boolean isComplete()
    {
        //check nulls
        //if right is null, return left.isLeaf()
        if ( left == null && right == null )
            return true;
        if ( left == null && right != null )
            return false;
        if ( left != null && right == null )
            return left.isLeaf();
        else
        {
            if ( left.isFull() && right.isComplete())
            {
                if ( left.height() == right.height() )
                    return true;
                return false;
            }
            else if ( left.isComplete() && right.isFull() )
            {
                if( left.height() == right.height() + 1 )
                    return true;
                return false;
            }
            return false;
        }
    }

    /**
     * Returns true if the difference of heights of subtrees at every node is no greater than one.
     * Two overall if cases - one for when left is null and when left is not null. Within each of the
     * when right is null or not null.
     * If both left and right are null, returns true. If only left or only right is null, checks the
     * is less than or equal to 1, returns isBalanced() of that node. Otherwise, returns false. If ne
     * difference between heights of the two nodes. If difference is less than or equal to 1, returns
     * Otherwise, returns false.
     *
     * @return      true if the difference of heights of subtrees at every node is no greater than on
     */
    public boolean isBalanced()
    {
        if ( left == null )
        {
            //if both left and right are null
            if ( right == null )
                return true;
            //if only left is null
            else
            {
                if ( right.height() <= 1 )
                    return right.isBalanced();
                else
                    return false;
            }
        }
        else
        {
            //if only right is null
            if ( right == null )
            {
                if ( left.height() <= 1 )
                    return left.isBalanced();
                else
                    return false;
            }
            //if neither left nor right is null
            else
            {
                if ( left.height() - right.height() <= 1 )
                    return left.isBalanced() && right.isBalanced();
                else
                    return false;
            }
        }
```

```
    }


}



```

Great job. Worked for all of my tests. Your design is generally clear, even if some of the methods are unnecessary to solve this problem.
A+