

```

1  /**
2   * HuffmanRunner class.
3   *
4   * @author Jessica Li
5   * @version 02/07/16
6   */
7  public class HuffmanRunner
8  {
9      /**
10     * Main method, tests HuffmanTree.
11     * Creates new HuffmanTree, encodes String and prints result, decodes encoded String and prints result.
12     *
13     * @param args    main method parameter
14     */
15     public static void main( String[] args )
16     {
17         //HuffmanNode law = new HuffmanNode( "key", 2 );
18         System.out.println();
19         HuffmanTree hTree = new HuffmanTree( "Maya Huffman huffs at Huffman trees." );
20         //hTree.printPQueue();
21         System.out.println();
22         String code = hTree.encode( "Maya Huffman huffs at Huffman trees." );
23         System.out.println( code + "\n" );
24         System.out.println( hTree.decode( code ) );
25     }
26 }
27
28 import java.util.Map;
29 import java.util.HashMap;
30 import java.util.PriorityQueue;
31 import java.util.Set;
32
33 /**
34 * Huffman Tree class.
35 * I chose to import HashMap rather than TreeMap because the stored values do not need to be sorted, and
36 * HashMap also has faster runtime for put and peek.
37 * Huffman Code works by taking in a sentence and putting each letter of the sentence into a map with
38 * map are converted into HuffmanNodes and transferred into a priority queue, with the elements with
39 * To create the tree, it takes the first two elements out of the priority queue, connects them to a
40 * nodes, and reinserts the new node into its proper place in the priority queue. This continues until
41 * Encode traverses through the tree, recording the path of each letter with 0 and 1 for left and right
42 * Decode reads the number, going left for 0 and right for 1. Once it hits a leaf node, it records the
43 * until the entire number is read.
44 *
45 * @author Jessica Li
46 * @version 02/07/16
47 */
48 public class HuffmanTree
49 {
50     private HashMap<String, Integer> hmap;
51     //pQueue is initialized in the method intoPQueue()
52     private PriorityQueue<HuffmanNode> pQueue;
53     //root is initialized in the method buildTree()
54     private HuffmanNode root;
55     private String sentence;
56
57     /**
58     * Constructor for HuffmanTree that takes in a sentence that will be used to create the tree and
59     * Initializes the map and sentence. Creates the map and priority queue, builds the tree.
60     *
61     * @param s        string for the original sentence
62     */
63     public HuffmanTree( String s )
64     {
65         hmap = new HashMap<String, Integer>();
66         sentence = s;
67         intoMap( sentence );
68         intoPQueue();
69         buildTree();
70     }
71
72     /**
73     * Converts String into map of all values.
74     * For loop traverses through the string sentence, adding each character to the map. If the map al

```

```

75     * replaced with the same key but frequency is increased by 1. Otherwise, a new entry with that ch
76     *
77     * @param s      the original sentence
78     */
79 private void intoMap( String s )
80 {
81     for ( int i = 0; i < s.length(); i++ )
82     {
83         String letter = s.substring(i, i+1 );
84         if ( hmap.containsKey( letter ) )
85         {
86             //oh my god bane of my code -- why can't I have ++ in the parameter? why do it outside
87             //this marks where I started making progress on debugging. Before I discovered nothing
88             int frequency = hmap.get( letter ) + 1;
89             hmap.put( letter, frequency );
90         }
91         else
92             hmap.put( letter, 1 );
93     }
94 }
95
96 /**
97  * Transfers elements from the map to a priority queue. Gets all the keys in map as a Set. Change
98  * priority queue. Loop iterates for each element in map, adding new HuffmanNodes to the priority
99  * the array. Type casts from Object to String.
100 */
101 private void intoPQueue()
102 {
103     Set k = hmap.keySet();
104     Object[] keys = k.toArray();
105     pQueue = new PriorityQueue<HuffmanNode>();
106
107     //high priority, low frequency
108     //find and place low frequency nodes in first
109     //all in loop
110     for ( int i = 0; i < hmap.size(); i++ )
111     {
112         pQueue.add( new HuffmanNode( (String) keys[i], hmap.get( (String) keys[i] ) ) );
113     }
114     //after the loop, the pQueue should be finished, with all the nodes in the proper order. It or
115 }
116
117 /**
118  * Prints the priority queue in order. For testing purposes.
119  * Loop traverses through priority queue. Removes the top HuffmanNode, prints the value and count.
120  * queue is also decreasing by one.
121  */
122 public void printPQueue()
123 {
124     //System.out.println( pQueue.size() );
125     for ( int i = 0; i < pQueue.size(); i++ )
126     {
127         HuffmanNode n = pQueue.poll();
128         System.out.println( "V: " + n.getValue() + "      C: " + n.getCount() );
129         i--;
130     }
131 }
132
133
134 /**
135  * Builds the Huffman Tree from the priority queue. While the size of the priority queue is greater
136  * creates a new HuffmanNode with combined values and counts, and adds new Node to priority queue.
137  * fully built. Sets root to the only remaining HuffmanNode in the priority queue, which is the to
138  * because I want to be able to see a visual representation of the tree when I run the code.
139  */
140 private void buildTree()
141 {
142     while ( pQueue.size() > 1 )
143     {
144         HuffmanNode n1 = pQueue.poll();
145         HuffmanNode n2 = pQueue.poll();
146         HuffmanNode combinedNode = new HuffmanNode( n1.getValue() + n2.getValue(), n1.getCount() +
147         pQueue.offer( combinedNode );
148     }

```

```

149     //peek or poll? both work? ?
150     //System.out.println( pQueue.peek() );
151     root = pQueue.peek();
152     System.out.println( root );
153 }
154
155
156 /**
157  * Encodes String s. String code stores the sequence of 0's and 1's. For loop traverses through St
158  * with root and a single character as parameters, adding the returned value to code. Once the loc
159  *
160  * @param s      String to be encoded
161  * @return       encoded sequence of 0's and 1's
162  *
163  */
164 public String encode( String s )
165 {
166     //System.out.println( root );
167     String code = "";
168     for ( int i = 0; i < s.length(); i++ )
169     {
170         code += encodeHelper( root, s.substring( i, i+1 ) );
171     }
172     return code;
173 }
174
175 /**
176  * Helper method to encode() that returns the encoded value of a single character. Recursive.
177  * Base case: if the HuffmanNode taken in is a leaf and its value equals the letter taken in, then
178  * If the node's left child contains the letter in its value, returns 0 + recursive call with left
179  * If the node's right child contains the letter in its value, returns 1 + recursive call with rig
180  *
181  * @param n      current HuffmanNode, used for traversing through the tree
182  * @param l      letter/character to be encoded
183  * @return       encoded String for a particular letter
184  */
185 private String encodeHelper( HuffmanNode n, String l )
186 {
187     //System.out.println(n);
188     //base case for n is leaf
189     if ( n.isLeaf() && n.getValue().equals( l ) )
190     {
191         //System.out.println("leaf");
192         return "";
193     }
194
195     //System.out.println( n.getLeft().getValue() );
196     //System.out.println( n.getRight().getValue() );
197     if ( n.getLeft().getValue().contains( l ) )
198     {
199         //System.out.println( n.getLeft().getValue() );
200         return "0" + encodeHelper( n.getLeft(), l );
201     }
202     //I don't need all the code in the line below, but for clarity's sake
203     else if ( n.getRight().getValue().contains( l ) )
204     {
205         //System.out.println( n.getRight().getValue() );
206         return "1" + encodeHelper( n.getRight(), l );
207     }
208     //if neither left nor right paths contain l -- but that's not possible right, it must be conta
209     //System.out.println( "out of if" );
210     return "";
211 }
212
213 /**
214  * Decodes String code. Calls helper method decodeHelper with root and code as parameters.
215  *
216  * @param code   the code to be decoded
217  * @return       the String for which code translates
218  *
219  */
220 public String decode( String code )
221 {
222     return decodeHelper( root, code );

```

Commenting a
little cluttered,
but good
solution.

```

223     }
224
225     /**
226     * Helper method to decode() that takes in a HuffmanNode and code as parameters. Recursive.
227     * Base case: If n, or the current node, is a leaf and the length of code is 0, returns the value
228     * If n is a leaf and the length of code is not 0, returns the value of n + recursive call with root
229     * the next node that the code refers to.
230     * If the first character in code equals 0, then sets n to n's left child, sets code to same String
231     * recursive call with n, code as param. Otherwise (if first char equals 1), sets n to n's right child
232     * character, and returns recursive call with n, code as param.
233     *
234     * @param n      the current HuffmanNode, used to traverse the tree
235     * @param code    the code to be decoded
236     * @return       the String that for which the code translates
237     */
238     public String decodeHelper( HuffmanNode n, String code )
239     {
240         if ( n.isLeaf() )
241         {
242             //System.out.println( "LEAF" );
243             if ( code.length() == 0 )
244             {
245                 return n.getValue();
246             }
247             else
248             {
249                 return n.getValue() + decodeHelper( root, code );
250             }
251         }
252         if ( code.substring(0,1).equals("0") )
253         {
254             n = n.getLeft();
255             code = code.substring( 1 );
256             return decodeHelper( n, code );
257         }
258         else
259         {
260             n = n.getRight();
261             code = code.substring( 1 );
262             return decodeHelper( n, code );
263         }
264     }
265 }
266
267 /**
268 * HuffmanNode class.
269 *
270 *
271 * @author Jessica Li
272 * @version 02/07/16
273 */
274 public class HuffmanNode implements Comparable<HuffmanNode>
275 {
276     private String value;
277     private int count;
278     private HuffmanNode left;
279     private HuffmanNode right;
280
281     /**
282     * Constructor for HuffmanNode, gives value and count. Sets left and right to null.
283     *
284     * @param v      string for the value
285     * @param c      int for frequency
286     */
287     public HuffmanNode( String v, int c )
288     {
289         value = v;
290         count = c;
291         left = null;
292         right = null;
293     }
294
295     /**
296     * Constructor for HuffmanNode, sets value, count, left, and right.

```

```

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

```

```

371  /**
372   * Accessor for right.
373   *
374   * @return value
375   */
376  public HuffmanNode getRight()
377  {
378      return right;
379  }
380
381  /**
382   * Modifier for right.
383   *
384   * @param v      new right
385   */
386  public void setRight( HuffmanNode r )
387  {
388      right = r;
389  }
390
391  /**
392   * Compares the frequency of each node, returning positive if greater, 0 if equal, and negative if
393   * current node.
394   *
395   * @param node    the node being compared to
396   */
397  public int compareTo( HuffmanNode node )
398  {
399      return count - node.getCount();
400  }
401
402  /**
403   * Returns true if node does not have any children - if left and right are both null.
404   *
405   * @return        true if left and right are both null, false otherwise
406   */
407  public boolean isLeaf()
408  {
409      return left == null && right == null;
410  }
411
412  /**
413   * Returns the number of descendants of node, including the current node.
414   * If both left and right nodes are null, returns 1. If neither nodes are null, returns 1 + the s
415   * If only left is null, returns 1 + the size of right. If only right is null (the else case), re
416   *
417   * @return        the size of the tree, or the total number of nodes within the tree
418   */
419  public int size()
420  {
421      if ( left == null && right == null )
422          return 1;
423      else if ( left != null && right != null )
424          return 1 + left.size() + right.size();
425      else if ( left == null && right != null )
426          return 1 + right.size();
427      else
428          return 1 + left.size();
429  }
430
431  /**
432   * Recursive method that returns String representation of tree, separated by parentheses and comm
433   *
434   * @return a String representation of the tree.
435   */
436  public String toString()
437  {
438      {
439          if ( left == null && right == null )
440              return value.toString();
441          if ( left == null && right != null )
442              return value.toString() + "( , " + right.toString() + ")";
443          if ( left != null && right == null )
444              return value.toString() + "(" + left.toString() + ", )";

```

Do you actually need
all these methods for
HuffmanNode?

//does this need toString()???

```

445         else
446             return value.toString() + "(" + left.toString() + ", " + right.toString() + ")";
447     }
448
449     /**
450     * Returns the maximum path length to a descendent.
451     * Two overall if cases - one for if left is null, one for if left is not null. Within these two
452     * If both left and right are null, returns 1. If only left or right is null, returns 1 + the height
453     * If neither left nor right is null, returns 1 + the maximum of the heights of both nodes.
454     *
455     * @return the height of the tree, or the maximum path length to a descendent
456     */
457     public int height()
458     {
459         //if math.max or if statements
460         //just return 1+
461
462         if ( left == null )
463         {
464             //if both left and right are null
465             if ( right == null )
466                 return 1;
467             //if only left is null
468             else
469                 return 1 + right.height();
470         }
471         else
472         {
473             //if only right is null
474             if ( right == null )
475                 return 1 + left.height();
476             //if neither left nor right is null
477             else
478                 return 1 + Math.max( left.height(), right.height() );
479         }
480     }
481 }
482
483
484     /**
485     * Returns true if adding a node to tree would increase its height - or in other words, if the tree
486     * If both left and right nodes are null, returns true. If only the left or right node is null, returns false.
487     * If neither left nor right is null, checks if left and right are full. If both are full, checks
488     * If both have the same height, returns true. Otherwise, returns false.
489     *
490     * @return true if tree is full (if adding a node to tree would increase its height), false otherwise
491     */
492     public boolean isFull()
493     {
494         //check to see if null
495         //check to see if height is same
496         if ( left == null && right == null )
497             return true;
498         else if ( left == null || right == null )
499             return false;
500         else
501         {
502             if ( left.isFull() && right.isFull() )
503             {
504                 if ( left.height() == right.height() )
505                     return true;
506             }
507             return false;
508         }
509     }
510
511     /**
512     * Returns true if tree has minimal height and any holes in the tree appear in the last level to
513     * If both left and right are null, returns true. If left is full, right is complete, and the height
514     * returns false. If left is complete, right is full, and the height difference is one, returns true.
515     * If none of the if statements execute, returns false.
516     *
517     * @return true if tree has no holes, false otherwise
518     */

```

```

519 public boolean isComplete()
520 {
521     //check nulls
522     //if right is null, return left.isLeaf()
523     if ( left == null && right == null )
524         return true;
525     if ( left == null && right != null )
526         return false;
527     if ( left != null && right == null )
528         return left.isLeaf();
529     else
530     {
531         if ( left.isFull() && right.isComplete() )
532         {
533             if ( left.height() == right.height() )
534                 return true;
535             return false;
536         }
537         else if ( left.isComplete() && right.isFull() )
538         {
539             if( left.height() == right.height() + 1 )
540                 return true;
541             return false;
542         }
543         return false;
544     }
545 }
546
547 /**
548  * Returns true if the difference of heights of subtrees at every node is no greater than one.
549  * Two overall if cases - one for when left is null and when left is not null. Within each of the
550  * when right is null or not null.
551  * If both left and right are null, returns true. If only left or only right is null, checks the
552  * is less than or equal to 1, returns isBalanced() of that node. Otherwise, returns false. If ne
553  * difference between heights of the two nodes. If difference is less than or equal to 1, returns
554  * Otherwise, returns false.
555  *
556  * @return true if the difference of heights of subtrees at every node is no greater than one
557  */
558 public boolean isBalanced()
559 {
560     if ( left == null )
561     {
562         //if both left and right are null
563         if ( right == null )
564             return true;
565         //if only left is null
566         else
567         {
568             if ( right.height() <= 1 )
569                 return right.isBalanced();
570             else
571                 return false;
572         }
573     }
574     else
575     {
576         //if only right is null
577         if ( right == null )
578         {
579             if ( left.height() <= 1 )
580                 return left.isBalanced();
581             else
582                 return false;
583         }
584         //if neither left nor right is null
585         else
586         {
587             if ( left.height() - right.height() <= 1 )
588                 return left.isBalanced() && right.isBalanced();
589             else
590                 return false;
591         }
592     }

```



```
593     }
594
595
596 }
597
598
599
```

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+