Huffman code

This problem was asked by Amazon

Now let's solve the problem asked by Amazon. That is, given a message, create a Huffman Code Tree where shorter code lengths correspond to more common letters. In this discussion the term frequency gets used often. For the sake of this program, frequency is used to represent the number of total times and is an `int`.

Basically, here is the algorithm to build a Huffman Tree (Borrowed from stated website):

1. Create a node for each character with its frequency and insert it to a list.
2. Remove from the list the two nodes with minimum frequencies. Then create a tree where the two nodes are "siblings" leafs and their parent is marked with the sum of their frequencies (the root has no character associated to it).
3. add the root node into the list. The list now is shorter by one element.
4. goto 3 until a single node remains in the list. This node will be the root to the Huffman tree.

https://distributedalgorithm.wordpress.com/2016/01/02/huffman-coding-the-optimal-prefix-code/

I have provided you with a fully implemented `HuffmanNode` class, which is very similar to the TreeNode class from the Morse Code lab. (Fully implemented implies it works!) The main differences are:

- `HuffmanNode` contains a `getFrequency()` method which returns an `int` representing the number of times the character appeared in the message.
- `HuffmanNode` does not contain the methods `setRight` or `setleft`. You are welcome to implement them if you wish. I did not need them in my solution.

Your task is to complete the `BuildOptimalHuffmanTree` class which will implement the algorithm shown above. There are two incomplete methods in the `BuildOptimalHuffmanTree` class. Your task is to complete both these methods.

To implement the above method, the first thing needed is a frequency count (how many times) of all 'letters' in the message. Tis is accomplished in the constructor/`getFrequuencyTable()`. These two must work together. It does not matter which method does the work, but invoking `getFrequuencyTable()` returns a `Map<String, Integer>`. The Map should contain the number of times each 'letter' (the key) appears in the String passed to the constructor.
Here is a tester for this method is:

```
public void testOptimalHuffmanTree01a()  {
   BuildOptimalHuffmanTree hc = new BuildOptimalHuffmanTree("abcdabcaba");

   Map<String, Integer> freq = hc.getFrequuencyTable();
   assertEquals(1, freq.get("d").intValue() );
   assertEquals(2, freq.get("c").intValue() );
   assertEquals(3, freq.get("b").intValue() );
   assertEquals(4, freq.get("a").intValue() );
}
```

Huffman code

This problem was asked by Amazon

The second method is `getOptimalTree()` which return a `HuffmanNode` that is the root of optimal Huffman Tree requested by Amazon. Now that we have the frequency of each 'letter' in the message, we can proceed with the algorithm previously shown.

This is what I did, but you are welcome to ….

1. I created a `List<HuffmanNode> nodes` and repeatedly added the 'letter' from `getFrequency()` with the minimum frequency. This way my List was sorted from minimum frequency to largest frequency. Remember to remove each element from the Map when it is added to the List. This is basically step 1 of the algorithm.
2. If you notice step 4 is to go until there is one element in the List, I started a while loop `while (nodes.length() > 1)`
3. Inside the while loop remove the first two elements from the List `nodes` and create a new `HuffmanNode` with the following values:
    a. The value is set to `null`
    b. The left child is set to the first (minimum frequency of the two) `HuffmanNode` removed from the List.
    c. The right child is set to the second (maximum frequency of the two) `HuffmanNode` removed from the List.
    I admit the assigning of the left and right child is arbitrary, but if you want to pass the tester, you will do it my way!
    d. The frequency is set to the sum of the frequency of the nodes removed from the List.
4. Now the hard part: Add this newly created `HuffmanNode` into the correct sorted location in the List `nodes`. **It is VERY important** this newly created `HuffmanNode` goes in front of any `HuffmanNode` currently in the List with equal frequency. This means that all `HuffmanNode nodes` before it (smaller index) have a smaller frequency and all `HuffmanNode nodes` after it (larger index) have a frequency greater than **OR equal.**
5. When the while loop terminates, it will contain ONE node, which is the root of the Huffman Tree ☺.

Here is a tester for this method is:

```
public void testOptimalHuffmanTree01b()  {
   BuildOptimalHuffmanTree hc = new BuildOptimalHuffmanTree("abcdabcaba");

   HuffmanNode hn = hc.getOptimalTree();
   String nL = hn.getLeft().getValue();
   assertEquals("a", nL );
   assertEquals("d", hn.getRight().getLeft().getLeft().getValue() );
   assertEquals("c", hn.getRight().getLeft().getRight().getValue() );
   assertEquals("b", hn.getRight().getRight().getValue() );
}
```