Improved Huffman coding

Task description

- Implement fully working Improved Huffman Coding algorithm (with constant time updates to the tree).
- Restrict your alphabet to 6 letters {a, b, c, d, e, f} and construct the sequence of 1000 symbols (e.g. abacefddcabbdedacffaeebdfcabeafbdcab …) using the following probability distribution:

  Pb(a)=0.05, Pb(b)=0.1, Pb(c)=0.15, Pb(d)=0.18, Pb(e)=0.22, Pb(f)=0.3
- First, compute the compression ratio when the above distribution is fixed.

  Secondly, compute the compression ratio when the distribution changes after each 100 symbols so that Pb(a) → Pb(c) and vice versa whereas Pb(d) → Pb(f) and v. versa.
- Your adaptive coding scheme is supposed to derive the true probability by counting the frequencies of the last 150 symbols.

Huffman coding

Coding based on counting the alphabet letter frequencies

Letters with corresponding frequencies are nodes with letter and its frequency as well as with left and right children nodes

These nodes are put to the Min Heap

Two nodes with smallest frequencies are aggregated (and removed) into new one

- New node is created, taking the sum of its children's' frequencies as its new frequency (and added to the nodes list)
  - *This node is then compared to other nodes as done above, and so on*
- We repeat these steps until there is only one element left in the node list, which is the root of our Huffman tree

Huffman coding is efficient if probabilities of symbols do not change though the time (and not efficient if they are changing)

Improved Huffman coding

Tries to be “adaptive” Huffman coding

Tries to minimize the negative effect of “normal” Huffman coding

Which one? → static symbol probabilities requirement

How? → recompute the Huffman tree after some symbols encoded

In such a way, if the letter probabilities change, these changes will be taken into consideration when building a new tree for the next portion of the letters

Compression ratio should be better than in normal Huffman coding (in theory)

Code example

```
class HuffmanNode {

    char symbol;
    int freq;

    HuffmanNode left;
    HuffmanNode right;

}
```

```
public static HuffmanNode generateTree(PriorityQueue<HuffmanNode> pq) {

    HuffmanNode root = null;

    while (pq.size() > 1) {

        HuffmanNode x = pq.poll();
        HuffmanNode y = pq.poll();
        HuffmanNode z = new HuffmanNode();


        z.freq = x.freq + y.freq;
        z.symbol = Character.MIN_VALUE;     // like a NULL value

        z.left = x;
        z.right = y;

        root = z;
        pq.add(z);
    }

    return root;
}
```

```
static final int howManyLastLettersToRead = 150;
static final int afterWhatLetterBeginAdaptive = 150;    // (the value is the position in the original message)
static final int adaptSecUpdFreq = 150;                 // after how many new letters read we will rebuild the tree
                                                        // (the less it is, the more often "updates to the tree" will occur)
```

```java
public static String encode(HashMap<Character, String> letterToCodeword, String originalText) {

    int messLen = originalText.length();
    int adaptLen = messLen - afterWhatLetterBeginAdaptive;
    String encoded = "";
    char[] text = originalText.toCharArray();

    for (int i = 0; i < afterWhatLetterBeginAdaptive; i++) {
        String currentCodeword = letterToCodeword.get(text[i]);
        encoded = encoded + currentCodeword;
    }

    // Now, we begin with "adaptive" part
    int lenLeft = adaptLen;

    while (lenLeft > 0) {

        int newStartIndex = messLen - lenLeft;
        int newEndIndex;
        int currentSecLen;

        if (lenLeft >= adaptSecUpdFreq) {
            currentSecLen = adaptSecUpdFreq;
            newEndIndex = newStartIndex + adaptSecUpdFreq - 1;
        } else {
            currentSecLen = lenLeft;
            newEndIndex = messLen - 1;
        }
```

```
        String last150Letters;
        int lettersEncodedSoFar = messLen - lenLeft;

        if (lettersEncodedSoFar >= howManyLastLettersToRead)
            last150Letters = originalText.substring(newStartIndex - howManyLastLettersToRead, newStartIndex);
        else            // lettersEncodedSoFar < howManyLastLettersToRead
            last150Letters = originalText.substring(newStartIndex - lettersEncodedSoFar, newStartIndex);

        letterToCodeword = updateCodewords(last150Letters);

        // After updating the letterToCodeword, continue encoding string with NEW codewords
        for (int i = newStartIndex ; i <=  newEndIndex; i++) {
            String currentCodeword = letterToCodeword.get(text[i]);
            encoded = encoded + currentCodeword;
        }

        lenLeft = lenLeft - adaptSecUpdFreq;
    }

    return encoded;
}
```

Results

4 different strings were tested with different probability distributions on both Normal and Improved Huffman coding

(1) random string of 1000 letters length with STATIC probabilities for the letters

a = 0.05 b = 0.1 c = 0.15 d = 0.18, e = 0.22, f = 0.3

(2) The same as (1), but with SWAPPING probabilities for the letters

After each 100 letters, swap ***a*** and ***c*** as well as ***d*** and ***f***

(3) The same as (2), but length of the string 10 times more, and distribution changes after each 1000 letters (not each 100 letters as before)

(4) The same as (3), but swapping changed:

Swap ***a*** and ***f***, ***b*** and ***e***, ***c*** and ***d***

Average of 10 tests for each case

String number		(1)	(2)	(3)	(4)
Entropy		2.41	2.46	2.46	2.58
Compression ratio	Usual Huffman	1.22	1.2 +- 0.01	1.19	1.06
	Improved Huffman	1.21	1.2 +- 0.01	1.20	1.18
Weighted average codeword length	Usual Huffman	2.45	2.5	2.5	2.84
	Improved Huffman	2.47	2.5 + 0.02	2.48	2.79

Conclusion

For Adaptive Huffman coding to work well, the following recommendations should be followed:

- Sample for the tree reconstruction should be big enough
  - *If it is small, then we can sometimes calculate inappropriate probabilities, which was the case with the string (2)*
- The tree that we have build at some point of adaptive encoding should be useful for the following letters in the string
  - *If the probability distribution changes all the time without a break, then the previous tree built will not be that much helpful for the following symbols*