1 CompressionInterface.java

```
1  import java.util.Scanner;
2  import java.util.ArrayList;
3  import java.util.HashMap;
4  import java.io.*;
5  import java.nio.file.*;
6
7  /**
8  * The CompressionInterface class gives a user interface to the Huffman coding algorithm for compression.
9  * @author Kate Belson
10 */
11 public class CompressionInterface {
12
13     /**
14     * Calculates all the characters used in the file and the frequencies of these characters.
15     * @author Kate Belson
16     * @param fileReader is the readable verion of the file used.
17     * @param characterFrequency is the HashMap of the characters and their frequencies.
18     * @return the completed version of the characterFrequency HashMap.
19    */
20     public static HashMap<Character, Integer> getCharacterFrequency(Scanner fileReader, HashMap<Character,
           Integer> characterFrequency) {
21         while (fileReader.hasNextLine()) {
22             String data = fileReader.nextLine();
23             for (int i=0; i<data.length(); i++) {
24                 boolean added = false;
25                 for (Character j : characterFrequency.keySet()) {
26                     if (data.charAt(i) == j) {
27                         characterFrequency.put(j, characterFrequency.get(j) + 1);
28                         added = true;
29                     }
30                 }
31                 if (added == false) {
32                     characterFrequency.put(data.charAt(i), 1);
33                 }
34             }
35         }
36           return characterFrequency;
37     }
38
39     /**
40    * A function that gets the binary string in bytes in a string.
41     * @author Kate Belson [ref Chico Camargo]
42     * @param data is the data to return.
43    */
44     static byte[] GetBinary(String s) {
45         while (s.length() % 8 != 0) {
46             s = s + '0';
47         }
48
49         byte[] data = new byte[s.length() / 8];
50
51         for (int i = 0; i < s.length(); i++) {
```

```
            char c = s.charAt(i);
            if (c == '1') {
                data[i >> 3] |= 0x80 >> (i & 0x7);
            }
        }
        return data;
    }

    /**
    * A function that gets the binary string in bytes in a string.
     * @author Kate Belson [ref Chico Camargo]
     * @param bytes is the string in bytes.
    */
    static String GetString(byte[] bytes) {
        StringBuilder sb = new StringBuilder(bytes.length * Byte.SIZE);
        for (int i = 0; i < Byte.SIZE * bytes.length; i++)
            sb.append((bytes[i / Byte.SIZE] << i % Byte.SIZE & 0x80) == 0 ? '0' : '1');
        return sb.toString();
    }

    /**
    * A function to save to a file.
     * @author Kate Belson [ref Chico Camargo]
     * @param binaryString is the string to save to the file.
     * @param outputName is the name of the outpt destination file.
    */
    public static void saveToFile(String binaryString, String outputName) {
        byte[] converted = GetBinary(binaryString);

        // Save bit array to file
        try {
            OutputStream outputStream = new FileOutputStream(outputName);
            outputStream.write(converted);
            outputStream.close();

        } catch (IOException e) {
            System.out.println("Error writing to file!");
        }
    }

    /**
    * A function to load the compressed file.
     * @author Kate Belson [ref Chico Camargo]
     * @param outputName is the name of the ouput file.
     * @return the string from the file.
    */
    public static String loadFromFile(String outputName) {
        // Load bit array from file
       try {
            byte[] allBytes = Files.readAllBytes(Paths.get(outputName));

            return GetString(allBytes);

        } catch (IOException ex) {
            return "An error has accoured.";
```

```
        }
    }

    /**
    * A function to compare the file sizes.
     * @author Kate Belson
     * @param original is the name of the original file.
     * @param compressed is the name of the compressed file.
    */
    public static void compareSizes (String original, String compressed) {
        File originalFile = new File(original);
        File compressedFile = new File(compressed);
        double originalSize = originalFile.length();
        double compressedSize = compressedFile.length();
        System.out.println("The original file's size is " + originalSize);
        System.out.println("The compressed file's size is " + compressedSize);
        double percentageReduction = ((originalSize - compressedSize) / originalSize) * 100;
        System.out.println("This is a reduction of " + percentageReduction + "%");
    }

    /**
    * The main function from which the program is run.
     * @author Kate Belson
     * @param args is the arguments to run.
    */
    public static void main( String[] args ) {
        HashMap<Character, Integer> characterFrequency = new HashMap<Character, Integer>();
        Scanner in = new Scanner(System.in);
        System.out.println("Enter a file name, ending in .txt, to compress.");
        System.out.println("Please include the whole file path: ");
        String fileName = in.nextLine();
        try {
            File file = new File(fileName);
            Scanner fileReader = new Scanner(file);
            characterFrequency = getCharacterFrequency(fileReader, characterFrequency);
            fileReader.close();
        } catch (FileNotFoundException e) {
            System.out.println("An error occurred with reading the file.");
            System.out.println("Please restart the program.");
        }

        ArrayList<Node> tree = new ArrayList<Node>();

        for (Character i : characterFrequency.keySet()) {
            Node node = new Node(characterFrequency.get(i), i, null, null);
            tree.add(node);
        }

        int start = 0;
        int end = tree.size() - 1;
        ArrayList<Node> sortedTree = QuickSort.quickSort(tree, start, end);

        Node root = null;

        while (sortedTree.size() > 1) {
```

```
162
163             Node right = sortedTree.get(0);
164             sortedTree.remove(0);
165
166             Node left = sortedTree.get(0);
167             sortedTree.remove(0);
168
169             Node f = new Node(right.getFrequency() + left.getFrequency(), '-', left, right);
170
171             root = f;
172
173             sortedTree.add(f);
174             sortedTree = QuickSort.quickSort(sortedTree, start, sortedTree.size() - 1);
175         }
176
177         System.out.println("Compressing file...");
178         Huffman huffmanCode = new Huffman(root);
179         String binary = "";
180         try {
181             File file = new File(fileName);
182             Scanner fileReader = new Scanner(file);
183             while (fileReader.hasNextLine()) {
184                 String data = fileReader.nextLine();
185                 for (int l=0; l<data.length(); l++) {
186                     for (Character i : huffmanCode.relevantCodes.keySet()) {
187                         if (i == data.charAt(l)) {
188                             binary = binary + huffmanCode.relevantCodes.get(i);
189                         }
190                     }
191                 }
192             }
193             fileReader.close();
194         } catch (FileNotFoundException e) {
195             System.out.println("An error occurred with reading the file.");
196             System.out.println("Please restart the program.");
197         }
198         System.out.println("Enter a file name for the compressed file.");
199         System.out.println("To put it in a certain folder, include the file path, or it will be in the src
                folder.");
200         System.out.println("It must end with .bin: ");
201         String outputName = in.nextLine();
202         saveToFile(binary, outputName);
203         System.out.println("File compressed.");
204         String fileString = loadFromFile(outputName);
205         compareSizes(fileName, outputName);
206         huffmanCode.decodeHuffman(sortedTree, fileString, in);
207         in.close();
208
209     }
210 }
```

2 Node.java

```
1 /**
```

```
* The Node class creates a node for a given tree.
* @author Kate Belson
*/

public class Node {

    private int frequency;

    private char character;

    private Node left;

    private Node right;

    /**
    * The constructor for the Node class.
    * @author Kate Belson
    * @param frequency is the frequency of the node.
    * @param character is the character represented.
     * @param left is the node to the left.
     * @param right is the node to the right.
    */
   public Node (int frequency, char character, Node left, Node right) {
        setFrequency(frequency);
        setCharacter(character);
        setLeft(left);
        setRight(right);
    }

    //setter methods

   /**
    * Sets the frequency variable.
    * @author Kate Belson
    * @param frequency is the frequency of the node.
    */
   public void setFrequency(int frequency) {
      this.frequency = frequency;
   }

    /**
    * Sets the character variable.
    * @author Kate Belson
    * @param character is the character represented by the node.
    */
   public void setCharacter(char character) {
      this.character = character;
   }

    /**
    * Sets the left variable.
    * @author Kate Belson
    * @param left is the node to the left of this node.
    */
   public void setLeft(Node left) {
```

```
57        this.left = left;
58    }
59
60     /**
61     * Sets the right variable.
62     * @author Kate Belson
63     * @param right is the node to the right of this node.
64     */
65    public void setRight(Node right) {
66        this.right = right;
67    }
68
69     //getter methods
70
71     /**
72     * Returns the frequency of the node.
73     * @author Kate Belson
74     * @return the frequency of the node.
75     */
76    public int getFrequency() {
77        return this.frequency;
78    }
79
80     /**
81     * Returns the character represented by the node.
82     * @author Kate Belson
83     * @return the character represented by the node.
84     */
85    public char getCharacter() {
86        return this.character;
87    }
88
89     /**
90     * Returns the node to the left of this node.
91     * @author Kate Belson
92     * @return the node to the left of this node.
93     */
94    public Node getLeft() {
95        return this.left;
96    }
97
98     /**
99    * Returns the node to the right of this node.
100   * @author Kate Belson
101   * @return the node to the right of this node.
102   */
103   public Node getRight() {
104       return this.right;
105   }
106
107 }
```

3 Huffman.java

```java
import java.util.ArrayList;
import java.util.HashMap;
import java.nio.file.*;
import java.io.*;
import java.util.Scanner;

/**
* The Huffman class for Huffman encoding and decoding.
* @author Kate Belson
*/
public class Huffman {

    String binaryCode;

    String decompressedCode;

    HashMap<Character, String> relevantCodes;

    /**
    * The constructor for the Huffman class.
    * @author Kate Belson
    * @param root is the root of the tree.
    */
    public Huffman(Node root) {
        this.binaryCode = "";
        this.decompressedCode = "";
        this.relevantCodes = new HashMap<Character, String>();
        String binaryString = "";
        encodeHuffman(root, binaryString);
    }

    /**
    * Implements Huffman Coding to compress the file.
    * @author Kate Belson
    * @param root is the root node of the tree.
    * @param binaryString is the string of binary digits for that character's coding.
   */
    public void encodeHuffman(Node root, String binaryString) {
        if (root.getLeft() == null && root.getRight() == null) {
            relevantCodes.put(root.getCharacter(), binaryString);
            binaryString = "";
            return;
          }
          if (root.getLeft() != null) {
              encodeHuffman(root.getLeft(), binaryString + "0");
          }
          if (root.getRight() != null) {
              encodeHuffman(root.getRight(), binaryString + "1");
          }
    }

    /**
    * Implements Huffman Decoding to decompress the file.
    * @author Kate Belson
    * @param nodes is the tree structure for the type of encoding.
```

```
56      * @param codedString is the string of bits for the file.
57      * @param in is the scanner used to take user input.
58     */
59      public void decodeHuffman(ArrayList<Node> nodes, String codedString, Scanner in) {
60          System.out.println("Enter a file name, ending in .txt, to put the decompressed file in.");
61          System.out.println("To put it in a certain folder, include the file path, or it will be in the src
                folder.");
62          System.out.println("It must end in .txt: ");
63          String fileName = in.nextLine();
64          try {
65              File file = new File(fileName);
66              if (file.createNewFile()) {
67                System.out.println("File created: " + file.getName());
68              } else {
69                System.out.println("File already exists.");
70              }
71          } catch (IOException e) {
72              System.out.println("An error occurred.");
73          }
74          Node root = nodes.get(0);
75          nextBranch(root, codedString, root, fileName);
76      }
77
78      /**
79      * Helps to implement the Huffman Decoding to decompress the file.
80      * @author Kate Belson
81      * @param root is the root node of the tree.
82      * @param codedString is the string of binary digits for that character's coding.
83      * @param topNode is the constant root node of the tree.
84      * @param fileName is the name of the destination file.
85     */
86      public void nextBranch(Node root, String codedString, Node topNode, String fileName) {
87          while (codedString != "") {
88              while (root.getLeft() != null && root.getRight() != null) {
89                  if (root.getLeft() != null && codedString.charAt(0) == '0') {
90                      nextBranch(root.getLeft(), codedString.substring(1), topNode, fileName);
91                      return;
92                  }
93                  else if (root.getRight() != null && codedString.charAt(0) == '1') {
94                      nextBranch(root.getRight(), codedString.substring(1), topNode, fileName);
95                      return;
96                  }
97              }
98              try {
99                  FileWriter writer = new FileWriter(fileName, true);
100                 PrintWriter printWriter = new PrintWriter(writer);
101                 printWriter.print(root.getCharacter());
102                 printWriter.close();
103                 writer.close();
104             } catch (IOException w) {
105                 System.out.println("An error occurred.");
106             }
107             nextBranch(topNode, codedString, topNode, fileName);
108             return;
109         }
```

```
    }

}
```

4 QuickSort.java

```
import java.util.ArrayList;

/**
* The QuickSort class contain the algorithm for implementing a QuickSort on the given ArrayList.
* @author Kate Belson
*/
public class QuickSort {

    /**
    * Used to split and sort the ArrayList from smallest to largest values frequencies of the nodes.
    * @author Kate Belson
    * @param tree is the ArrayList of nodes.
    * @param start is the start of the ArrayList.
    * @param end is the end of the ArrayList.
    * @return the new start value of the ArrayList.
   */
    public static int partition(ArrayList<Node> tree, int start, int end){
        Node pivot = tree.get(end);

        for(int i=start; i<end; i++){
            if(tree.get(i).getFrequency()<pivot.getFrequency()){
                Node temp = tree.get(start);
                tree.set(start, tree.get(i));
                Node n1 = temp;
                tree.set(i, n1);
                start = start + 1;
            }
        }

        Node temp = tree.get(start);
        Node n2 = tree.get(start);
        Node n3 = tree.get(end);
        n2 = pivot;
        tree.set(start, n2);
        n3 = temp;
        tree.set(end, n3);

        return start;
    }

    /**
    * Implements a QuickSOrt algorithm.
    * @author Kate Belson
    * @param tree is the ArrayList of nodes.
    * @param start is the start of the ArrayList.
    * @param end is the end of the ArrayList.
    * @return the sorted ArrayList tree.
   */
```

```
49      public static ArrayList<Node> quickSort(ArrayList<Node> tree, int start, int end) {
50          int partition = partition(tree, start, end);
51
52          if(partition-1>start) {
53              quickSort(tree, start, (partition - 1));
54          }
55          if(partition+1<end) {
56              quickSort(tree, (partition + 1), end);
57          }
58
59          return tree;
60      }
61  }
```