Programmieraufgabe 5 - Nachbesserung

```
Robert Wettstädt 535161
Sona Pecenakova 540607

Nachbesserung von der Aufgabe 2:
Wir haben uns entschieden für diese Aufgabe die Java Programmiersprache zu
verwenden, hauptsächlich wegen der größeren Komplexität und vielen String und
Character Verarbeitungen in dieser Aufgabe.
```

2. Schreiben Sie ein Programm, mit dem Sie ein Textfile Huffman-codieren ko□nnen. Welche Reduktion der Filegro□ße erreichen Sie damit? (Sie sollten Ihr Programm testen, indem Sie einen Text codieren, wieder dekodieren und u□berpru□fen, ob Sie damit wieder den Auggangstext erhalten.)

```
Code:
- Main.java - Main Klasse mit Implementierung von der Compress und Decompress
Methoden
- HuffmanMinHeap.java - Java Klasse die einen Huffman-Tree implementiert
- MinHeapNode.java - Java Klasse die den Node eines Baumes implementiert
```

Reduktion von Filegröße

```
Original size (input.txt) = 78 Bytes
Compressed size (compressedFile) = 39 Bytes
- Die Größe der Textdatei ist mit dem Huffman Coding um 50% reduziert
```

Test

Original File: input.txt

```
this is an exercise for algorithms course that we are taking in the university
```

Code Tabelle mit Häufigkeiten:

```
e (8) : 000
n (4) : 0010
h (4) : 0011
u (2) : 01000
c (2) : 01001
f (1) : 010100
m (1) : 010101
k (1) : 010110
v (1) : 0101110
j (0) : 010111100
z (0) : 010111101
b (0) : 0101111100
```

```
d (0) : 0101111101
p (0) : 0101111110
q (0) : 0101111111
i (8) : 011
y (1) : 100000
w (1) : 100001
x (1) : 100010
l (1) : 100011
g (2) : 10010
o (3) : 10011
a (5) : 1010
r (6) : 1011
s (6) : 1100
t (7) : 1101
  (13) : 111
```

Compressed File: compressedFile (binary file)

```
xxd -b compressedFile
0000000: 11010011 01111001 11011110 01111010 00101110 00100010  .y.z."
0000006: 00010110 10010111 10000011 10101001 00111011 11110101  ....;.
000000c: 00011100 10100111 01101111 01001101 01011100 11101001  ..oM\.
0000012: 10011010 00101111 00000111 11010011 10101101 11110000  ./....
0000018: 10001111 01010110 00111110 11010010 11001100 10100101  .V>...
000001e: 11011001 01111101 00110001 11010000 01001101 01110000  .}1.Mp
0000024: 10111100 01111011 00000000                             .{.
```

DecompressedFile: decompressedFile.txt

```
this is an exercise for algorithms course that we are taking in the university
```

- Als erstes wird die Originaldatei durchgelaufen und die Hauefigkeiten von den einzelnen Zeichen gespeichert

```
int[] freq = getFrequencies(originalFile);
```

- Danach wird ein MinHeap von den Zeichen und Hauefigkeiten gebaut und davon dann ein Huffman-Tree gebaut

```
HuffmanMinHeap minHeap = new HuffmanMinHeap(data, freq, size);
minHeap.buildMinHeap();
MinHeapNode root = minHeap.buildHuffmanTree();
```

- Als naechstes wird eine Code Tabelle erstellt, wo der Zeichen als Key gespeichert wird mit dem entsprechenden Code als Value

```
HashMap<Character, String> codeTable = new HashMap<Character, String>();
minHeap.getCodes(root, arr, top, codeTable);
```

- Diese wird dann fuer die Kompression und Dekompression von der Datei benutzt

```
File compressedFile = new File("compressedFile");
int finalZeros = compressFile(originalFile, compressedFile, codeTable);

File decompressedFile = new File("decompressedFile.txt");
decompressFile(compressedFile, decompressedFile, codeTable, finalZeros);
```

Quellcode

Main.java

```
import java.io.BufferedInputStream;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.FileWriter;
import java.io.IOException;
import java.io.InputStream;
import java.math.BigInteger;
import java.nio.file.Files;
import java.util.HashMap;
import java.util.Set;


public class Main {

    public static char[] data = {'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z', ' '};


    private static int[] getFrequencies(File file) throws IOException {
        int[] freq = new int[data.length];
        InputStream input = new BufferedInputStream(new FileInputStream(file));
        try {
            int c;
            while ((c = input.read()) != -1) {
                char ch = (char)c;
                for(int i = 0; i < data.length; i++){
                    if(data[i] == ch){
                        freq[i]++;
                    }
                }
            }
        } finally {
            input.close();
        }

        return freq;
    }

    public static int compressFile(File inputFile, File outputFile,
HashMap<Character, String> codeTable) throws IOException{
        InputStream inStream = new BufferedInputStream(new
FileInputStream(inputFile));
        try {
            String outputString = "";
            int c;
            while ((c = inStream.read()) != -1) {
                char ch = (char)c;
                String code = codeTable.get(ch);
                outputString += code;
            }

            //fill the last unfilled spots of the last byte with zeros
            int bitsLeftToFill = 8 - outputString.length()%8;
            if(bitsLeftToFill > 0){
                for(int i = 0; i < bitsLeftToFill; i++){
                    outputString += "0";
                }
            }

            //convert the string to bitearray
            BigInteger big = new BigInteger(outputString, 2);
            byte[] b = big.toByteArray();

            //drop the most significant bit - no need for sign
            if (b[0] == 0) {
                byte[] tmp = new byte[b.length - 1];
                System.arraycopy(b, 1, tmp, 0, tmp.length);
                b = tmp;
            }

            FileOutputStream fos = new FileOutputStream(outputFile);
            fos.write(b);
            fos.close();

            System.out.println("File compressed");
            return bitsLeftToFill;

        } finally {
            inStream.close();
        }

    }

    public static void decompressFile(File toDecompress, File decompressed,
HashMap<Character, String> codeTable, int finalZeros) throws IOException{
        //revert the codetables keys and values
        HashMap<String, Character> revCodeTable = revertCodeTable(codeTable);

        //read all bytes from file
        byte[] fileBytes = Files.readAllBytes(toDecompress.toPath());
        InputStream inStream = new BufferedInputStream(new
FileInputStream(toDecompress));
        try {
            String outputString = "";
            String readCode = "";
            String code = "";
            for(int i = 0; i < fileBytes.length; i++){
                byte by = fileBytes[i];
                String byteStr = Integer.toBinaryString(by & 0xFF);

                if(byteStr.length() < 8){
                    int bitsToComplete = 8 - byteStr.length();
                    while(bitsToComplete > 0){
                        //get the zeros on the most left bits
                        byteStr = "0" + byteStr;
                        bitsToComplete--;
                    }
                }

                code += byteStr;

            }

            //do not decompress the last zeros used to fill the byte
            if(finalZeros > 0){
                code = code.substring(0, code.length() - finalZeros);
            }

            for(int i = 0; i < code.length(); i++){
                char c = code.charAt(i);
                readCode += c;
                if(revCodeTable.containsKey(readCode)){
                    outputString += revCodeTable.get(readCode);
                    readCode = "";
                }
            }

            FileWriter out = new FileWriter(decompressed);
            out.write(outputString);
            out.close();

            System.out.println("File decompressed");
        } finally {
            inStream.close();
        }


    }

    //Revert the keys and values in the codetable
    public static HashMap<String, Character> revertCodeTable(HashMap<Character,
String> origCodeTable){
        HashMap<String, Character> revCodeTable = new HashMap<String, Character>();

        Set<Character> set = origCodeTable.keySet();
        Object[] keys = set.toArray();
        for(int i = 0; i < keys.length; i++){
            char key = (char)keys[i];
            String value = origCodeTable.get(key);
            revCodeTable.put(value, key);
        }

        return revCodeTable;
    }

    public static void main(String[] args) throws IOException{
        //get letter frequencies from file
        File originalFile = new File("input.txt");
        int[] freq = getFrequencies(originalFile);
        int size = freq.length;

        HuffmanMinHeap minHeap = new HuffmanMinHeap(data, freq, size);
        minHeap.buildMinHeap();

        MinHeapNode root = minHeap.buildHuffmanTree();

        //get codes
        int[] arr = new int[100];
        int top = 0;
        HashMap<Character, String> codeTable = new HashMap<Character, String>();
        minHeap.getCodes(root, arr, top, codeTable);

        //compress and decompress
        File compressedFile = new File("compressedFile");
        int finalZeros = compressFile(originalFile, compressedFile, codeTable);

        File decompressedFile = new File("decompressedFile.txt");
        decompressFile(compressedFile, decompressedFile, codeTable, finalZeros);


    }

}
```

HuffmanMinHeap.java

```
import java.util.HashMap;


public class HuffmanMinHeap {

    int heapSize;
    MinHeapNode[] nodeArray;

    public HuffmanMinHeap(char[] data, int[] freq, int size){
        heapSize = size;
        nodeArray = new MinHeapNode[heapSize];

        for(int i = 0; i < size; i++){
            nodeArray[i] = new MinHeapNode(data[i], freq[i]);
        }
    }

    private void minHeapify(int idx){
        int min = idx;
        int left = getLeft(idx);
        int right = getRight(idx);
        if(left < heapSize && nodeArray[left].freq < nodeArray[idx].freq){
            min = left;
        }

        if(right < heapSize && nodeArray[right].freq < nodeArray[min].freq){
            min = right;
        }

        if(min != idx){
            swapNodes(idx, min);
            minHeapify(min);
        }

    }

    private void swapNodes(int a, int b){
        MinHeapNode temp = nodeArray[a];
        nodeArray[a] = nodeArray[b];
        nodeArray[b] = temp;
    }

    private int getLeft(int idx){
        return 2 * idx + 1;
    }

    private int getRight(int idx){
        return 2 * idx + 2;
    }

    private int getParent(int idx){
        return (idx - 1)/2;
    }

    public void buildMinHeap(){
        for(int i = heapSize/2 - 1; i >= 0; i--){
            minHeapify(i);
        }
    }

    private boolean isGreaterOne(){
        return (heapSize > 1);
    }

    private MinHeapNode extractMin(){
        MinHeapNode min = nodeArray[0];
        nodeArray[0] = nodeArray[heapSize-1];
        heapSize--;
        minHeapify(0);

        return min;
    }

    private void heapIncreaseKey(int idx, MinHeapNode node){
        if(node.freq < nodeArray[idx].freq){
            System.out.println("New key less than actual key");
        }
        nodeArray[idx] = node;

        while(idx > 1 && nodeArray[getParent(idx)].freq > nodeArray[idx].freq){
            swapNodes(idx, getParent(idx));
            idx = getParent(idx);
        }
    }


    private void insertNode(MinHeapNode node){

        heapSize++;
        nodeArray[heapSize-1] = new MinHeapNode();

        heapIncreaseKey(heapSize-1, node);
    }

    public MinHeapNode buildHuffmanTree(){
        MinHeapNode left, right, internal;

        while(isGreaterOne()){
            left = extractMin();
            right = extractMin();

            internal = new MinHeapNode('$', left.freq + right.freq);
            internal.left = left;
            internal.right = right;
            insertNode(internal);
        }

        return extractMin();
    }

    private boolean isLeaf(MinHeapNode node){
        return (node.left == null && node.right == null);
    }


    public void getCodes(MinHeapNode root, int arr[], int top, HashMap<Character,
String> codeTable){

        if(root.left != null){
            arr[top] = 0;
            getCodes(root.left, arr, top + 1, codeTable);
        }

        if(root.right != null){
            arr[top] = 1;
            getCodes(root.right, arr, top + 1, codeTable);
        }

        if(isLeaf(root)){
            System.out.print(root.letter + ": ");
            printArr(arr, top);

            String code = "";
            for(int i = 0; i < top; i++){
                code += Integer.toString(arr[i]);
            }

            codeTable.put(root.letter, code);
        }

    }



    private void printArr(int arr[], int n){
        for(int i = 0; i < n; i++){
            System.out.print(arr[i]);
        }

        System.out.println();
    }


}
```

MinHeapNode.java

```
public class MinHeapNode {

    char letter;
    int freq;
    MinHeapNode left, right;

    public MinHeapNode(char c, int n){
        letter = c;
        freq = n;
    }

    public MinHeapNode(){

    }

}
```