I. My implementation and design

My work on this project was divided to 5 parts: compressing single file, decompressing single file, compressing folder, decompressing folder. Here I will describe each stage and my design on each stage. Problems will be described alongside design.

a) Compress single file

I use File class to refer to the given files.

One of the biggest problems during this project was choosing a right I/O tool. I tried several ones, but all of them could write only one byte or array of bytes. I tried using bitwise operations to deal with this problem, but in the end it proved to be inefficient and troublesome. After this I found DataInputStream and DataOutputStream. This is an I/O for writing primitive data types. For compressing I use DataInputStream. I read all the bytes of given file to a byte array. Then I created an array of integers with length of 256, which I called 'weights'. 256 is the biggest unsigned value that can be represented by byte. In other words, there are only 256 unique bytes. Then I loop through my array of bytes, and increase the integer in weights array, index of which is equal to unsigned value of current byte. `weights[Byte.toUnsignedInt(currentByte)]++;`

Then each pair of unique byte and its weight is stored in priority queue, unless its weight is equal to zero (which would mean that this byte never appears in given file After this I added to the priority queue a unique element with weight equal to zero. I called this element PSEUDO_EOF. The reason to have its weight equal to zero is to give it a unique Huffman encoding. Later this encoding will be used as End of File. My Huffman tree node has following data fields: parent, leftChild, rightChild, symbol and weight(frequency). I use one of the priority queues and Class called HuffmanTree, which stores a tree root, to build a proper Huffman Tree according to Huffman's algorithm.

After which I loop from bottom Node of Tree, where my byte symbol is stored, till the root of the tree, using Node’s parent data field, and store the Huffman code in String, after which symbol byte and string representation of Huffman encoding is stored in a hashtable, where byte symbol is a key, and Huffman string is a value. Having this hashtable I encode the size of my hashtable and write this size in my compressed file, then I write there every unique byte and its weight. Amount of this bytes and weights is obviously equal to the size of hashtable, which I wrote previously. Then I do a double loop. In the outer one I iterate through the bytes of original file, use my hashtable to get the string representing Huffman code, which represents this byte and in the inner loop using bitwise operations and mask (0x01) I write the Huffman code to the compressed file.

```
for(int i = 0;i< data.length;++i) {
        char[]code = encode.get(data[i]).toCharArray();
        for(int j = 0; j < code.length;++j) {
               if(code[j]=='1') {
                      c|=0x01;
               }
               if(bitCount==7) {
                      bitCount = 0;

                      out.writeByte(c);
                      c=0;
               }else {
                      c<<=1;
```

```
                    bitCount++;
                    }
            }
    }
```

After doing so, I use the inner loop above to write Huffman code for PSEUDO_EOF, and after this compressing is done.

2) Decompressing.

Using the size, unique byte symbols and weights I stored during compression, I exploit the same methods I used for compression to recreate the Huffman tree and get back my encoding. I use the same Hashtable, but this time I have a String as a key, and byte symbol as a value.

Then I read the compressed file byte by byte and use the following masks and bitwise AND to read the element of Huffman code and write the corresponding byte to the decompressed file.

3)Compressing folder.

Compressing Folder was relatively easy. In the very beginning I open one DataInputStream to write all the data there. I reference the folder as File and go recursively through it to get the access to every file within this folder and folders inside this folder. Once I enter the directory I will write the integer, which represents DIRECTORY_START, once I leave the directory I will write the integer, which represents DIRECTORY_END, and once I enter the file I will write the Integer, which represents FILE_START. I need all those delimiters because I was not able to find a better way to distinguish different files in one compressed document. The problem with this way is that it decreases compress rate for a lot of small files, or for files which were somehow compressed before (videos, pictures, etc.) And I also need to write down the name of the file, which also consumes memory. Besides that, I would use the same methods, that I used for compressing single file, with the only difference of having one common DataOutputStream for all the files. In the end I will write an Integer to represent a HUFFMAN_END

4) Decompressing a folder.

For decompressing a folder I will have a loop, each iteration of which looks for the one of the delimiters (DIRECTORY_START, DIRECTORY_END, FILE_START, HUFFMAN_END) and an object of File class, which stores current location of the decompressing process.

If the delimiter is DIRECTORY_START or DIRECTORY_END I will change the cur location in my File object., if delimiter is HUFFMAN_END I will break the loop, and if delimiter is FILE_START I will use exactly the same algorithms I used to decompress a single file.

5) GUI part was probably the easiest part. The only problem was that FileChooser cannot choose folders, as it could in swing, so I needed to add one more button with DirectoryChooser to compress folders.

TEST:

My computer (Very Old one): Intel(R) Core i3 CPU M350 2.27 Ghz

RAM: 3.00 GB

Folder	Compress time(ms)	Decompress time(ms)	Original size	Size after compress	rate
Test1	63155	93515	8,78Mb	6.52Mb	25,7%
Test2	46287	126260	14,2Mb	9.05Mb	36,2%
Test3	4	4	0	148b	-
Test4	647462	2132803	242Mb	161Mb	33%