Text file compression and decompression using Huffman Encoding.

Introduction

In simple terms, each symbol in a text file uses at least 1 byte of memory. Using Huffman coding, we can analyse a text file to determine which symbols are more common and which are less common. We can then fit many of these smaller codes into a single byte.

Data Structures

Binary Tree

My implementation makes use of a binary tree style structure to create, store, encode and decode symbols to and from their Huffman code representation. My implementation is simple and based on Node objects.

Heap

The algorithm uses a heap when constructing tree, to sort and hold Nodes before processing. It is a minimum heap. Symbols with a lower frequency are processed first.

HashMap

A Hash Map style structure Is used to store symbols and their created Huffman codes e.g. ‘a’=>“011”.

Bit Shifting

The program can encode binary representations of multiple symbols in a single byte using bit shifting. This is a bit level manipulation of the value of a byte – we shift the value of the byte to match our code and then write it to a file.

Execution Sequence

Compression (summary)

1. Count all the symbols in our file, recording their frequency.
2. Create an object representing each symbol and its frequency, place into minHeap.
3. While there are two symbols left:
    a. Poll two symbols from the queue.
    b. Create a new node, this node being the parent of the two nodes. Assign it’s frequency to be the sum of its children.
    c. Place this node back in the queue.
4. The final node in the queue is the root node of our tree.
5. ‘Walk’ along the tree, recording each step. A ‘left’ step is 0, a ‘right’ step 1. When we reach a leaf, the concatenation of all steps is its Huffman code. So if reach leaf node ‘a’ with LLL, its Huffman code is ‘000’.
6. Do this for all Nodes, place all symbols and codes in HashMap.

7. Calculate a string representation of the tree. Encode the string representation using bit shifting. Add to metadata buffer.
8. Capture all symbols of the tree in an arbitrary order. Encode them in bytes. Add to metadata buffer.
9. Write metadata header to file.
10. Read the input file character by character.
    a. For each character, get its Huffman code.
    b. Use bitshifting to write the code to a byte.
    c. When we have shifting a multiple of 8 bits, write this to the ByteBuffer.
    d. Do this for every symbol.
11. When we reach 4096 bytes, or EOF, write the buffer to file.
12. Repeat until the whole file is written.
13. Close the file.