

# Huffman Coding in Haskell

Rafał Włodarczyk, Michał Waluś

June 9, 2025

## Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Initial Requirements</b>	<b>2</b>
2.1	Parameters . . . . .	2
<b>3</b>	<b>Huffman Coding Algorithm</b>	<b>3</b>
3.1	Prefix Codes . . . . .	3
3.2	Frequency Map . . . . .	3
3.3	Priority Queue . . . . .	4
3.4	Huffman Tree construction . . . . .	4
3.5	Encoding and Decoding . . . . .	5
3.6	Bytestream Operations . . . . .	5
3.7	The Main Function . . . . .	5
<b>4</b>	<b>CI</b>	<b>5</b>
<b>5</b>	<b>Showcase</b>	<b>6</b>
<b>6</b>	<b>Conclusion</b>	<b>6</b>
<b>7</b>	<b>References</b>	<b>6</b>

# 1 Introduction

In 1952 David A. Huffman created an algorithm used for lossless data compression, building upon the advancements of Claude Shannon’s work on information theory. It has been proven to be optimal for a symbol-by-symbol coding with a known input probability distribution. The algorithm assigns variable-length codes to input characters, the shorter codes being assigned to more frequently occurring characters.

## 2 Initial Requirements

The program has been designed to fulfill the following requirements:

Write a program that uses Huffman coding (classic or dynamic) to compress files. This project will allow you to practice working with binary data and using tree-like data structures. To handle binary data, you can use the “bytestring” package.

We decided that our take on the implementation will provide a simple command line interface.

**Compress:** Compresses the input file using Huffman encoding.

```
1 $ huffman [input-file] -o [output-file]
```

Outputs the compression ratio and sizes.

**Decompress:** Decompresses the input file.

```
1 $ huffman [input-file] -d -o [output-file]
```

### 2.1 Parameters

- **[input-file]**: The file to be processed (required).
- **-o [output-file]**: Specifies the output file (required).
- **-d**: Enables decompression mode (default is compression).

### 3 Huffman Coding Algorithm

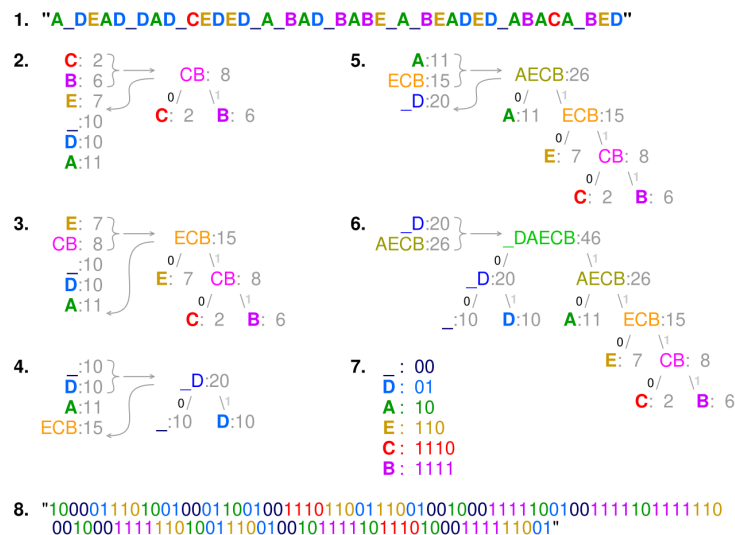


Figure 1: Huffman Coding Algorithm Schema

### 3.1 Prefix Codes

A set of codes  $\{C_1, C_2, \dots, C_n\}$  is a prefix code if:

$$\forall i, j \in \{1, 2, \dots, n\}, i \neq j : C_i \not\subseteq C_j \quad (1)$$

In any stream of bits, defined as  $S = b_1b_2 \dots b_m, b \in \{0,1\}$ , a prefix code can be decoded unambiguously. This feature is crucial for the Huffman coding algorithm, as it allows for deterministical decoding of the compressed data in linear time.

### 3.2 Frequency Map

A frequency map assigns each character in the input data the number of its occurrences. For example it can be easily constructed from a **String** intrinsic.

```

1 type CharMap = [(Char, Int)]
2
3 add :: CharMap -> Char -> CharMap
4 add [] c = [(c, 1)]
5 add ((x, y):xs) c
6   | x == c = (x, y + 1):xs
7   | otherwise = (x, y):add xs c
8

```

```

9 mapChars :: String -> CharMap
10 mapChars = mapCharsHelp []
11   where
12     mapCharsHelp cm [] = cm
13     mapCharsHelp cm (x:xs) = mapCharsHelp (add cm x) xs

```

Listing 1: Constructing a frequency map from a string.

### 3.3 Priority Queue

In order to build the Huffman tree in linear time, we must preserve the order of the characters based on their number of occurrences. A priority queue is a data structure which allows for efficient insertion and extraction of the minimum element.

```

1 type LeafQueue = [Tree Char Int]
2
3 createLQ :: String -> LeafQueue
4 createLQ = charMapToQueue . mapChars
5
6 insertLQ :: LeafQueue -> Tree Char Int -> LeafQueue
7 insertLQ [] tree = [tree]
8 insertLQ (t:lq) tree
9   | get tree < get t = tree:t:lq
10  | otherwise = t:insertLQ lq tree
11
12 mergeLQ :: LeafQueue -> Tree Char Int
13 mergeLQ [] = error "Cannot merge an empty queue"
14 mergeLQ [t] = t
15 mergeLQ (t1:t2:ts) = mergeLQ $ insertLQ ts $ Node t1 t2 $
16   get t1 + get t2
17
18 charMapToQueue :: CharMap -> LeafQueue
19 charMapToQueue = charMapToQueueHelp []
20   where
21     charMapToQueueHelp lq [] = lq
22     charMapToQueueHelp lq ((c, i):cn) =
23       charMapToQueueHelp (insertLQ lq $ Leaf c i) cn

```

Listing 2: Priority queue implementation using a binary tree.

### 3.4 Huffman Tree construction

We can now construct the Huffman tree (a binary tree with leaves containing characters) from the frequency map each time extracting the two least frequent characters and merging them into a new node.

```

1 data Tree a b = Leaf a b | Node (Tree a b) (Tree a b) b
2   deriving (Eq)

```

```

3  makeCode :: Tree Char Int -> Map Char String
4  makeCode t = makeCodeHelp t ""
5      where
6          makeCodeHelp (Leaf x _) s = singleton x s
7          makeCodeHelp (Node t1 t2 _) s = union (
8              makeCodeHelp t1 (s ++ "0")) (makeCodeHelp t2
9              (s ++ "1"))
10 makeCode :: Tree Char Int -> Map Char String
11 makeCode t = makeCodeHelp t ""
12     where
13         makeCodeHelp (Leaf x _) s = singleton x s
14         makeCodeHelp (Node t1 t2 _) s = union (makeCodeHelp
15             t1 (s ++ "0")) (makeCodeHelp t2 (s ++ "1"))

```

Listing 3: Huffman tree construction.

### 3.5 Encoding and Decoding

The encoding process is straightforward. we traverse the Huffman tree and replace each character with its corresponding code, we decided to use a **Map** intrinsic to store the codes for each character. We can join the coding map with the input data to produce a compressed binary stream.

### 3.6 Bytestream Operations

The package **bytestring** provides an efficient way to handle binary data in Haskell.

### 3.7 The Main Function

The main function uses the IO monad to read the input file, perform processing and write the output file.

## 4 CI

The project uses Stack as a build tool, which allows for simple dependency management and building.

```
1 $ stack build
```

Afterwards the executable can be run with the following command, or can be copied to any of the PATH directories for easier access.

```
1 $ stack exec huffman -- [input-file] -o [output-file]
```

- 5 Showcase
- 6 Conclusion
- 7 References