Huffman Coding in Kotlin: A Complete Implementation

Olin DSA–style Project Report (Compressor, Visualizer, and CLI)

Kotlin Implementation

`HuffmanCompressor + HuffmanVisualizer (single-file)`

October 26, 2025

Abstract

This report documents a complete, single-file implementation of *Huffman coding* in Kotlin. The project provides two primary classes: a `HuffmanCompressor` for the core logic and a `HuffmanVisualizer` for generating ASCII-art trees and frequency analyses. We present the API, the core algorithms (frequency analysis, heap-based tree construction, code generation, encoding, and decoding), and a robust I/O strategy using a text-based header for binary data. Following Olin DSA conventions, we provide correctness arguments (prefix-free property) and a full asymptotic runtime analysis. The report concludes with reflections on Kotlin-specific implementation choices and a discussion of the interactive CLI.

Contents

1 Overview and Goals

Huffman coding is a fundamental lossless data compression algorithm. It constructs an optimal, prefix-free set of binary codes based on the frequencies of input symbols, minimizing the expected code length.

The goals for this project were:

- Implement a **full round-trip** Huffman encoder and decoder in Kotlin.

- Provide a **clean API** separating compression logic from visualization.
- Build an **interactive CLI visualizer** to inspect frequencies, codes, and the tree structure.
- Ensure **correctness** for edge cases (empty files, single-symbol files).
- Analyze the **asymptotic runtime** and memory usage.

2 Interface and Code Architecture

The project is built around three core classes, all contained in a single file for portability.

Listing 1: Public API surface

```
// Basic node structure for the huffman tree
data class Node(
    val symbol: Char? = null,
    val frequency: Int,
    val left: Node? = null,
    val right: Node? = null
) : Comparable<Node> {
    override fun compareTo(other: Node): Int
    fun isLeaf(): Boolean
}

class HuffmanCompressor {
    // count how many times each character appears
    fun analyzeFrequency(text: String): Map<Char, Int>

    // build the huffman tree using a min-heap
    fun buildHuffmanTree(frequencies: Map<Char, Int>): Node

    // traverse tree to generate binary codes
    fun generateCodes(root: Node): Map<Char, String>

    // encode text using the code table and write to file
    fun encode(text: String, codeTable: Map<Char, String>, outputFile: File)

    // decode a compressed file back to original text
    fun decode(inputFile: File): String

    // full compression pipeline
    fun compress(inputFile: File, outputFile: File)

    // full decompression pipeline
    fun decompress(inputFile: File): String
}

// visualization tool for understanding the huffman tree
class HuffmanVisualizer(private val compressor: HuffmanCompressor) {
    fun visualizeFromText(text: String)
    private fun printTree(node: Node?, prefix: String, isTail: Boolean)
    fun interactiveMode()
}

// main interactive menu
fun interactiveMain()
fun main()
```

Key design choices.

- **ADT:** The `Node` is a simple `data class`. Leaf nodes are identified by a non-null `symbol` and null children (via `isLeaf()`).
- **Heap:** We use `java.util.PriorityQueue<Node>` for the min-heap, relying on the `Node`'s `Comparable` implementation.
- **Header Format:** To decompress, the decoder needs the code tree. We store this as a human-readable text header:
  1. Line 1: Number of symbols k.
  2. Lines 2 to $k+1$: The code table, one entry per line, as `[char.code]:[bitstring]`. Using `char.code` (its integer value) ensures newlines and control characters are serialized safely.
  3. Line $k+2$: A sentinel string `"END_HEADER"`.
  4. Line $k+3$: The exact `bitLength` of the data, used to trim padding from the final byte.
- **Payload:** The header is followed immediately by the raw binary payload (the packed bitstream).

3 Algorithms & Data Structures

3.1 Frequency Analysis

Goal: Build a histogram of character frequencies.

Listing 2: Frequency analysis

```kotlin
fun analyzeFrequency(text: String): Map<Char, Int> {
    val freqMap = mutableMapOf<Char, Int>()
    for (char in text) {
        freqMap[char] = freqMap.getOrDefault(char, 0) + 1
    }
    // ... sanity check ...
    require(freqMap.values.sum() == text.length) { /* ... */ }
    return freqMap
}
```

Runtime: $\Theta(n)$ where n is the number of characters in the text.
Space: $O(k)$ where k is the number of distinct symbols (the alphabet size).

3.2 Huffman Tree Construction

Goal: Use a min-priority queue to implement the standard greedy Huffman algorithm.

Listing 3: Greedy tree construction with a min-heap

```kotlin
fun buildHuffmanTree(frequencies: Map<Char, Int>): Node {
    require(frequencies.isNotEmpty()) { "can't build tree from nothing" }

    val pq = PriorityQueue<Node>()
    // start with all symbols as leaf nodes
    frequencies.forEach { (char, freq) ->
        pq.offer(Node(symbol = char, frequency = freq))
    }

    // merge until we have a single root
    while (pq.size > 1) {
        val left = pq.poll()
```

```
        val right = pq.poll()
        val merged = Node(
            frequency = left.frequency + right.frequency,
            left = left,
            right = right
        )
        pq.offer(merged)
    }
    return pq.poll()
}
```

Runtime: The loop runs $k - 1$ times. Each iteration involves two `poll` and one `offer`, each costing $O(\log k)$. The initial heap build is $O(k \log k)$ (or $O(k)$ with 'heapify', but k pushes is equivalent here). Total time: $O(k \log k)$.
Space: $O(k)$ for the priority queue and the $2k - 1$ nodes in the final tree.

3.3 Code Generation

Goal: Perform a DFS traversal on the tree to build the bitstring codes.

Listing 4: DFS code generation

```
fun generateCodes(root: Node): Map<Char, String> {
    val codeTable = mutableMapOf<Char, String>()

    // recursive DFS to build codes
    fun traverse(node: Node, code: String) {
        if (node.isLeaf()) {
            // edge case: single symbol gets code "0"
            codeTable[node.symbol!!] = if (code.isEmpty()) "0" else code
        } else {
            node.left?.let { traverse(it, code + "0") }
            node.right?.let { traverse(it, code + "1") }
        }
    }

    traverse(root, "")
    // ... prefix-free verification ...
    return codeTable
}
```

Runtime: The traversal visits every node and edge in the tree exactly once. The tree has k leaves and $k - 1$ internal nodes, so $2k - 1$ nodes in total. Runtime is $\Theta(k)$.
Space: $O(k)$ to store the code table. The recursion depth is at most k (in a skewed tree).

3.4 Encoding

Goal: Write the text header, then map the input text to a bitstream and pack it into bytes.

Listing 5: Bitstring building and byte packing

```
fun encode(text: String, codeTable: Map<Char, String>, outputFile: File) {
    // convert text to bit string
    val bitString = StringBuilder()
    for (char in text) {
        bitString.append(codeTable[char] ?: error("no code for '$char'"))
    }

```

```
    // build header with code table... (omitted for brevity)
    val header = /* ... build header ... */

    // pack bits into bytes (pad last byte with 0s if needed)
    val bytes = mutableListOf<Byte>()
    for (i in bitString.indices step 8) {
        val chunk = bitString.substring(i, minOf(i + 8, bitString.length))
            .padEnd(8, '0')
        bytes.add(chunk.toInt(2).toByte())
    }

    // write header as text, then binary payload
    outputFile.writeText(header.toString(), Charsets.UTF_8)
    outputFile.appendBytes(bytes.toByteArray())
}
```

Runtime: Let B be the total number of bits in the encoded message. Building the ‘bitString‘ takes $\Theta(B)$ time (assuming n lookups in $O(n)$ total). Packing into bytes takes $\Theta(B/8)$ time. Total: $\Theta(n + B)$.
Space: $\Theta(B)$ for the intermediate ‘bitString‘. This is a memory bottleneck for large files and could be replaced by a streaming bit-writer.

3.5 Decoding

Goal: Read the file, safely parse the text header, unpack the binary payload, and decode.

Listing 6: Binary-safe header parse + greedy decode

```
fun decode(inputFile: File): String {
    val bytes = inputFile.readBytes()
    var i = 0

    // helper to read lines from byte array (avoids text/binary mixing issues)
    fun readLine(): String {
        val start = i
        while (i < bytes.size && bytes[i] != '\n'.code.toByte()) i++
        val s = String(bytes, start, i - start, Charsets.UTF_8)
        i++  // skip newline
        return s
    }

    // parse header (omitted, see Listing 2) ...
    val codeTable = mutableMapOf<String, Char>() // Note: inverse map
    // ...
    val bitLength = readLine().toInt()
    // ...
    val payload = bytes.copyOfRange(i, bytes.size)

    // convert bytes back to bit string
    val bitString = buildString(payload.size * 8) { /* ... */ }
        .substring(0, bitLength)  // trim to actual bit length

    // decode by matching prefixes
    val decoded = StringBuilder()
    var cur = ""
    for (bit in bitString) {
        cur += bit
        codeTable[cur]?.let { ch ->
```

```
            decoded.append(ch)
            cur = ""  // reset for next code
        }
    }
    return decoded.toString()
}
```

Runtime: $\Theta(B)$ to rebuild the ‘bitString‘ and $\Theta(B)$ to scan it (since each bit is appended and checked once). Total: $\Theta(B)$.
Space: $\Theta(B)$ for the ‘bitString‘ and $\Theta(B)$ for the output ‘StringBuilder‘.

4 Correctness (DSA-style)

Prefix property. The tree construction guarantees the prefix-free property. All symbols are at leaves. A code is a path from the root to a leaf. By definition, no path to a leaf can be a prefix of another path to a different leaf. The explicit check in `generateCodes` (omitted from listing) provides a redundant safeguard.

Greedy decoding. Because the code set is prefix-free, the first match found when scanning the bitstream is guaranteed to be the *only* valid match. There is no ambiguity. This allows for a simple, linear-time greedy scan (as seen in Listing 6). The loop invariant is: at the end of each iteration, the `cur` buffer is either empty or a valid prefix of one or more codes in the table.

Tree optimality. The standard Huffman exchange argument proves optimality. The greedy choice is to merge the two subtrees with the smallest frequencies. Any optimal tree can be transformed into the one generated by this algorithm without increasing the total weighted path length, proving the greedy choice is always part of an optimal solution.

5 Asymptotic Runtime and Space

Let n be the number of input characters, k the number of distinct symbols ($k \leq n$), and B the total number of bits in the compressed payload.

Phase	**Time**	**Space**
Frequency analysis	$\Theta(n)$	$O(k)$
Tree construction (heap)	$O(k \log k)$	$O(k)$
Code generation (DFS)	$\Theta(k)$	$O(k)$
Encoding (build+pack)	$\Theta(n + B)$	$O(B)$
Decoding (parse+scan)	$\Theta(B)$	$O(B)$

For typical text files, $k \ll n$. The dominant cost is linear in the input size n and compressed size B. The $O(k \log k)$ term is usually negligible. The $O(B)$ space for intermediate bitstrings is the main scalability limit.

6 Interactive Interface & Visualization

A key feature of this project is the `HuffmanVisualizer` class, which provides insight into the algorithm's state. The `visualizeFromText` method prints:

- A **frequency table** with scaled ASCII bars.
- The **generated code table**, sorted by code length.
- The full **ASCII tree structure**, implemented with a recursive printer.

Listing 7: Recursive ASCII tree printer

```
private fun printTree(node: Node?, prefix: String = "", isTail: Boolean = true) {
    if (node == null) return
    val connector = if (isTail) " " else " "
    val extension = if (isTail) "    " else "    "

    if (node.isLeaf()) {
        val display = when (node.symbol) {
            ' ' -> "SPACE"
            '\n' -> "NEWLINE"
            else -> "'${node.symbol}'"
        }
        println("$prefix$connector$display (${node.frequency})")
    } else {
        println("$prefix$connector[${node.frequency}]")
        node.left?.let { printTree(it, prefix + extension, false) }
        node.right?.let { printTree(it, prefix + extension, true) }
    }
}
```

The `interactiveMain()` function wraps all functionality in a simple menu-driven CLI, allowing users to visualize text, load from files, compress, and decompress.

7 Implementation Reflection (Kotlin)

Strengths. Kotlin was an excellent choice. `data class` makes the `Node` ADT trivial. Local functions (like `traverse` in Listing 4 and `readLine` in Listing 6) are perfect for helper routines that share parent scope. The `require(...)` function provides clean, expressive precondition checks. Java interop with `PriorityQueue` is seamless.

I/O Pitfalls & Fix. The most critical part of this implementation is mixing text and binary data. Reading the whole file as text will corrupt the binary payload. Writing the header as text and *appending* bytes (‘appendBytes‘) is correct, but reading is harder. The solution (Listing lst:decode) is to read the *entire file* as raw `ByteArray`, then parse the header portion manually with a byte-aware `readLine` helper, and finally slice the remaining `payload` bytes. This is robust and binary-safe.

Memory Considerations. As noted, building the entire `bitString` as a `StringBuilder` (or `String`) consumes $O(B)$ memory. For a 1GB file, this could require gigabytes of RAM for the intermediate string. A production-grade implementation would use a streaming `BitWriter` and `BitReader` to operate in small $O(1)$ or $O(\text{buffersize})$ memory.

Alphabet Choice. This implementation uses `Char` as the symbol. This works well for text files but is not general. A more robust implementation would treat `Byte` (0-255) as the symbol, allowing it to compress *any* file (images, executables, etc.). The logic of the algorithm remains identical.

8 Acknowledgments

General Acknowledgments I express my gratitude to various resources that contributed to this work. Specifically, I utilized ChatGPT to assist with LaTeX writing and formatting. Additionally, I consulted several reputable online sources for coding guidance, including Wikipedia,

GeeksforGeeks, MIT's resources on Huffman coding, and Berkeley EECS materials. ChatGPT also provided valuable support in developing the command-line interface (CLI) tool.

End of report.