

# Algorithm Design and Analysis

**Data Compression Using Huffman Coding**

**Endrit Makolli 192047593**

\*University for Business and Technology, Prishtina 10000, Kosovo

# Introduction

## **Overview of the Problem**

**Data compression** is a fundamental technique in computer science that aims to **reduce the size of data** **without excessively compromising its quality**. The need for compression arises in numerous scenarios—from **efficient storage** solutions to **faster data transmission** over networks. **Huffman Coding is one of the most widely known and utilized compression methods**, notable for its optimality under specific conditions. The technique relies on constructing a **variable-length code table based on the frequencies of individual symbo**ls within a given dataset. **Symbols that appear more frequently are assigned shorter bit patterns**, while those that occur less frequently receive longer bit patterns, **thereby minimizing the total number of bits required to represent the data**. This approach enables significant savings in memory and bandwidth resources and is frequently used in file archival utilities, image and text compression, and network data exchange.

## **Motivation and Relevance**

The importance of data compression particularly Huffman Coding becomes more evident as data-intensive applications and services continue to grow. Every day, enormous quantities of text, images, audio, and video data are generated and need to be stored or transmitted efficiently. **Cloud services, multimedia streaming, and online communications** all rely heavily on effective compression to reduce operational costs and improve user experiences. By using Huffman Coding, organizations can lower storage overhead and enhance data transfer speeds without prohibitive losses in quality or information fidelity. The algorithm’s theoretical foundations and its practical efficiency have led it to be a crucial building block in many compression standards, making it highly relevant for both academic research and industrial applications.

## **Objectives**

The primary **objective** is to investigate the Huffman Coding algorithm—delving into its design, the underlying data structures (such **as binary trees and min-heaps**), and the trade-offs of implementing Huffman Coding in different application domains.

Another key goal is to **implement Huffman Coding**, demonstrating how it can be applied to compress data sets of varying types and sizes. By **running empirical tests**, one can measure how well the algorithm performs and how compression rates differ across different data distributions.

A further objective involves identifying potential optimizations and **comparing Huffman Coding to other compression algorithms** (e.g., arithmetic coding or run-length encoding), thus elucidating the benefits and limitations of each approach.

Ultimately, this project aims to underscore the practical significance of Huffman Coding by showcasing use cases and discussing its integration into larger compression schemes. Understanding the algorithm’s place within broader applications provides essential insights into how best to deploy and refine it for modern data handling challenges.

# Literature Review

Huffman Coding, introduced by **David Huffman in 1952**, was a groundbreaking approach to lossless data compression that leverages optimal prefix-free codes based on symbol frequency. Since its inception, the algorithm has been extensively studied, refined, and implemented in various systems.

**Classical** Huffman Coding guarantees minimum redundancy for a given symbol distribution, making it optimal under the prefix-free constraint. It is also relatively straightforward to implement, typically using priority queues or binary trees. This approach is particularly efficient when symbol frequencies are either known in advance or can be updated incrementally, as seen in its adaptive variant. However, one of the notable **drawbacks** of classical Huffman Coding is that it generally requires two passes over the data if frequencies are not initially known—one pass to collect statistics and another to perform the actual encoding. Furthermore, its performance can degrade in scenarios where symbol frequencies change rapidly or when there are strong context dependencies that go beyond single-symbol probabilities.

Adaptive Huffman Coding addresses some of these limitations by **dynamically updating the Huffman tree as data is processed.** This eliminates the need for a separate frequency-gathering phase and allows the algorithm to adapt to changing symbol distributions, making it particularly suitable for streaming or online applications. Despite these advantages, Adaptive Huffman Coding introduces additional complexity in implementation and computational overhead due to the continuous updates required for the tree. Moreover, like its classical counterpart, it may still fall short in capturing contextual relationships in the data, which can negatively affect compression efficiency for certain types of input.

Other related approaches have also emerged over time. **Shannon–Fano Coding**, which predates Huffman Coding, offers a **simpler implementation but does not always produce optimal codes.** **Arithmetic Coding** takes a different route by representing entire messages as a single floating-point number, often resulting **in better compression ratios**. However, its **complexity**, as well as historical **concerns around patents and licensing**, have limited its widespread adoption. The LZ77 and LZ78 family of algorithms, including popular implementations like DEFLATE and LZ4, take a dictionary-based approach that exploits repeated patterns in data. These are often used in conjunction **with Huffman Coding**, such as in the **ZIP** and GZIP formats.

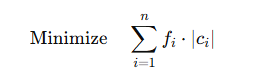
## **Gap Identification**

Despite the widespread adoption of Huffman Coding and its derivatives, several limitations persist in modern data-compression scenarios:

1. Contextual dependencies: Many real-world data sets (e.g., multimedia, text with complex structure) **exhibit correlations that span beyond single symbols**. Standard Huffman Coding treats symbols independently, ignoring such contextual or predictive opportunities.
2. High-Dimensional data: Contemporary data, such as high-resolution images or large-scale databases, can challenge the static or adaptive Huffman model, as it may become **computationally expensive or less effective when data characteristics vary widely** across segments.
3. Integration with other compression techniques: Huffman Coding is often used in conjunction with other compression paradigms (e.g., LZ-based methods). **Finding optimal ways to integrate Huffman Coding with advanced algorithms** (e.g., context modeling, deep-learning-based compression) is still an active area of research

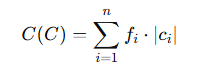
# Problem Definition and Formulation

## **Formal Problem Statement**

Given a set of n distinct symbols S={s1,s2,…,sn} and their corresponding frequencies (or probabilities) f={f1,f2,…,fn}, the goal of Huffman Coding is to construct a prefix-free code C that assigns a unique bit string c{i}ci​ to each symbol s{i}​. Formally, we seek to minimize the weighted path length (also called the weighted external path length in the associated Huffman tree):

where c{i} is the length (in bits) of the code assigned to symbol s{i}​. The frequency f{i} can be interpreted as the probability of symbol s{i} if the frequencies are normalized to sum up to 1. A valid solution is any set of codewords {c1,c2,…,cn} that is **prefix-free**, meaning no valid codeword is a prefix of another codeword.

Mathematically, the problem can be viewed as constructing a binary tree where each leaf node represents a symbol s{i}, and the path from the root to that leaf determines the bit string c{i}​. The expected cost (or expected code length) of the encoding is:



The objective is to find the binary tree (and thus the code) that yields the minimum C{C}.

## **Assumptions and Constraints**

1. **Finite Symbol Set**: The problem assumes a finite number of symbols n. Although Huffman Coding can be extended to handle evolving or theoretically unbounded alphabets (as in Adaptive Huffman Coding), this project focuses on a well-defined, finite set.
2. **Non-Negative Frequencies**: Each frequency f{i} is non-negative and may be treated as a probability if normalized.

* If any symbol’s frequency is zero, it typically can be omitted from the Huffman tree since it does not contribute to the code cost.

1. **Prefix-Free Constraint**: Every codeword c{i}​ must not serve as the initial portion of another codeword. This is necessary for **uniquely decodable** codes.
2. **Computational Constraints**:
   * **Time Complexity**: Huffman Coding can be implemented in O(nlogn) time using a min-heap. This project assumes that this computational cost is acceptable for typical use cases.
   * **Space Complexity**: Storing the Huffman tree requires O(n) space, plus any additional overhead for auxiliary data structures.
3. **Data Type Constraints**:
   * Symbol set elements s{i}​ can be characters, bytes, or any discrete tokens.
   * Frequencies f{i}​ can be integers (e.g., counts of occurrences) or real numbers (e.g., probabilities).
   * Code lengths must be integers (bits) in a practical implementation, but the analysis sometimes treats them as real numbers in an information-theoretic context.
4. **Static Model**: This work primarily deals with **static** Huffman Coding—where all frequencies are known beforehand. Extensions such as Adaptive Huffman Coding or handling data streams are beyond this core scope but may be referenced as relevant future directions.

# Algorithm Design

## **Proposed Algorithm**

The **Huffman Coding** algorithm is a **greedy** method that assigns variable-length binary codes to characters based on their frequencies. Characters that appear more frequently are given shorter codes, which achieves better compression on average than fixed-length encodings. Figure 1 showcases the following process:

* **Build a frequency table**: Count how often each character appears in the text.
* **Construct the huffman tree**: Insert all characters (with their frequencies) into a min-heap; repeatedly merge the two least-frequent nodes until only one remains.
* **Generate huffman codes**: Traverse the resulting binary tree to assign ‘0’ and ‘1’ bits to each character.
* **Encode**: Convert the original text into a bit string using the generated codes.
* **(Optional) decode**: Reconstruct the text from the bit string by traversing the Huffman Tree.

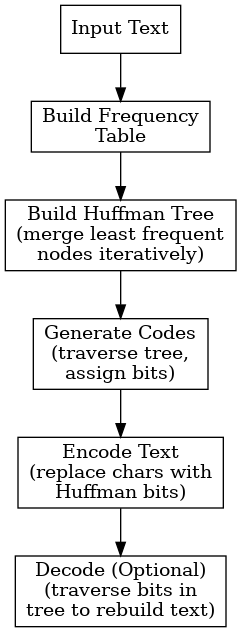
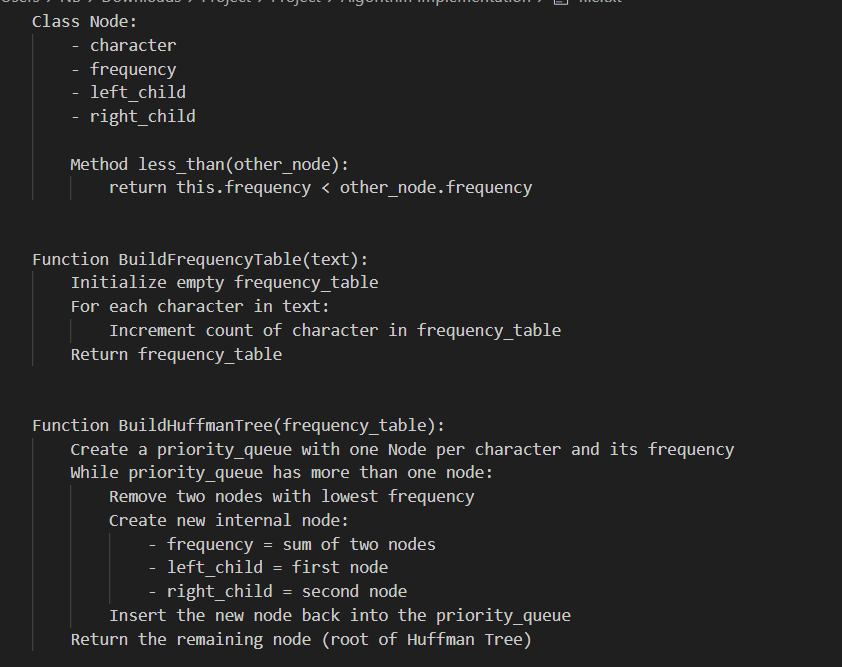


Figure 1 Proposed Algorithm



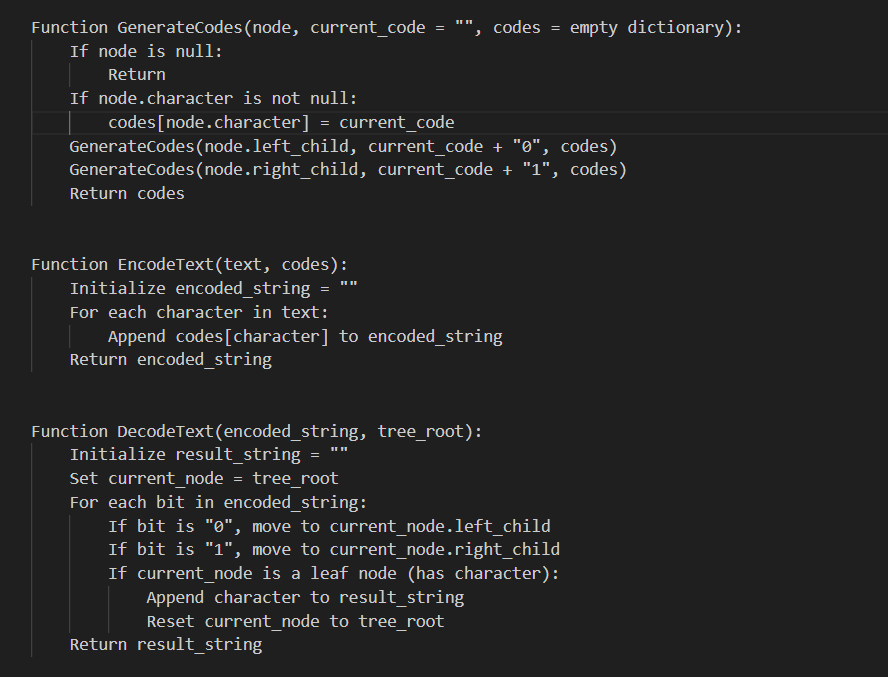


Figure 2 Pseudocode

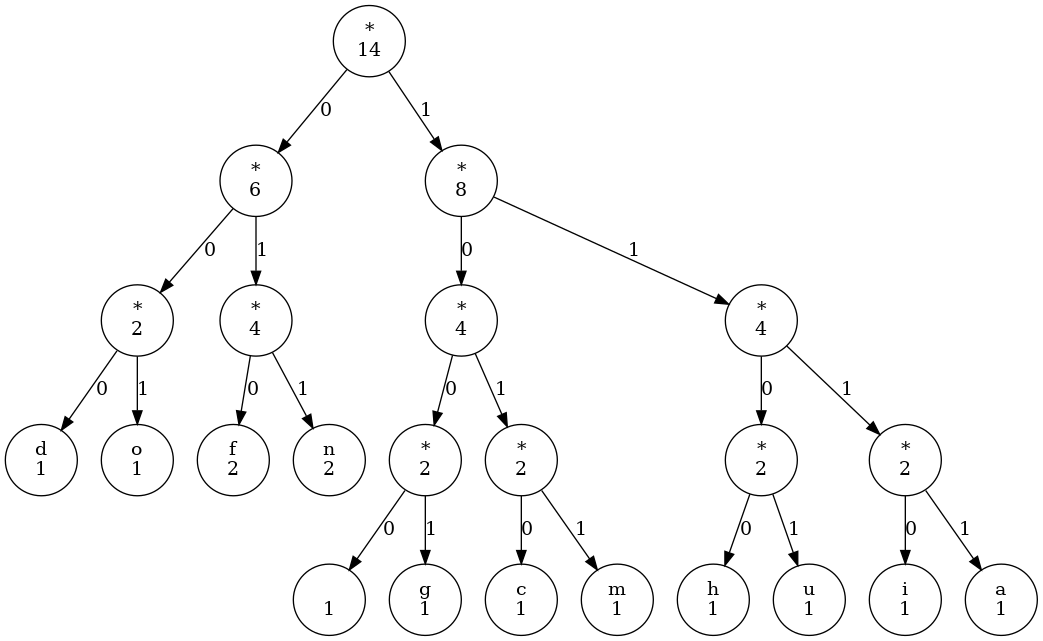


Figure 3 Huffman Decision Tree

## **Algorithm Choice and Justification**

The Huffman Coding algorithm was selected for this project due to its proven efficiency in lossless data compression and its foundational role in information theory. It offers optimal prefix-free encoding, ensuring that no code is a prefix of another, which enables unambiguous decoding. The algorithm is particularly effective when the frequency distribution of characters is skewed, allowing more frequent symbols to be represented with shorter codes. The implementation followed a classical approach using a priority queue (min-heap) to construct the Huffman Tree, ensuring that the two least frequent nodes are merged at each step. This strategy guarantees the minimum average code length for the given input. The project also includes code generation by tree traversal, encoding by symbol replacement, and optional decoding through bitwise navigation of the tree. Python's built-in libraries such as heapq and collections. Counter were leveraged to simplify the implementation, enhance readability, and maintain efficiency.

## **Optimality Considerations**

# Huffman Coding is renowned for its optimality under the constraint of prefix-free binary codes. Given a set of input symbols and their associated frequencies, the algorithm guarantees the minimum possible average code length, making it the most efficient solution for static, single-symbol encoding. This optimality arises from its greedy strategy, which always combines the least frequent symbols first to construct the Huffman Tree. However, this optimality is limited to cases where symbol probabilities are independent and do not depend on context. For more complex data distributions, such as those involving patterns or dependencies across symbols, alternative methods like Arithmetic Coding or dictionary-based algorithms (e.g., LZ77) may offer better compression. Nonetheless, within its domain, Huffman Coding remains a highly optimal and practical choice, particularly when the symbol frequency distribution is known in advance or does not change significantly during encoding.

# Complexity Analysis

## **Time Complexity**

Building the frequency table is O(n) where n is the length of the text. Constructing the Huffman Tree involves extracting and merging nodes in a min-heap, which leads to an O(nlogn) complexity where n is the number of unique characters. Generating codes by traversing the tree is O(n). Overall complexity (O(m + n log n))

* m is the length of the input
* n is the number of **unique** characters

## **Space Complexity**

In terms of space, Huffman Coding is also efficient. The frequency table occupies O(n) space, where n is the number of unique characters. The Huffman Tree, which includes one node per character and internal nodes for merges, also requires O(n) space. Additionally, the min-heap used during tree construction holds at most n elements at any time, contributing another O(n) in space. Thus, the total space complexity is O(n), making the algorithm scalable and suitable for memory-constrained environments when the number of unique symbols is relatively small.

## **Comparative Analysis**

Huffman Coding holds a prominent place in the field of lossless data compression, but several other algorithms offer competitive performance depending on the data characteristics and application requirements. Compared to Shannon–Fano Coding, Huffman is superior in terms of compression efficiency, as Shannon–Fano does not always guarantee an optimal prefix-free code. While both follow a similar divide-and-conquer principle, Huffman's greedy strategy ensures better average code lengths. In contrast, Arithmetic Coding can outperform Huffman in compressing data with highly skewed or fractional probabilities, as it encodes entire sequences into a single floating-point value. However, Arithmetic Coding is computationally more complex and was historically hindered by patent concerns, which made Huffman the more practical and accessible choice.

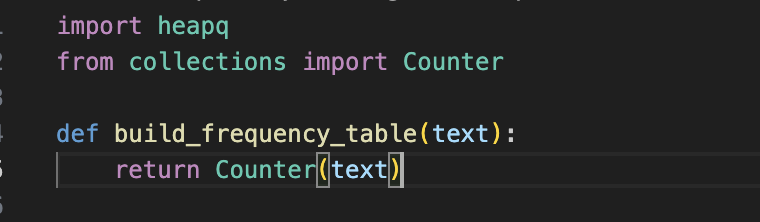
When compared to dictionary-based algorithms like LZ77, LZ78, or hybrids like DEFLATE (used in ZIP and GZIP), Huffman Coding shows different strengths. These dictionary-based methods excel in exploiting repeated patterns and longer substrings, often resulting in higher compression ratios for large or redundant data. In fact, Huffman Coding is frequently used as a final stage in these algorithms to encode the dictionary output more efficiently. However, for short texts or datasets with high entropy (few patterns), pure Huffman Coding remains competitive due to its simplicity and low overhead.

Overall, Huffman Coding offers a balanced trade-off between compression efficiency, speed, and implementation complexity. It is especially favorable in environments where the frequency distribution is known or can be calculated with minimal cost, and where low-latency or real-time performance is important. Despite the rise of more sophisticated algorithms, Huffman Coding continues to be a robust choice, particularly in embedded systems, file formats, and learning applications.

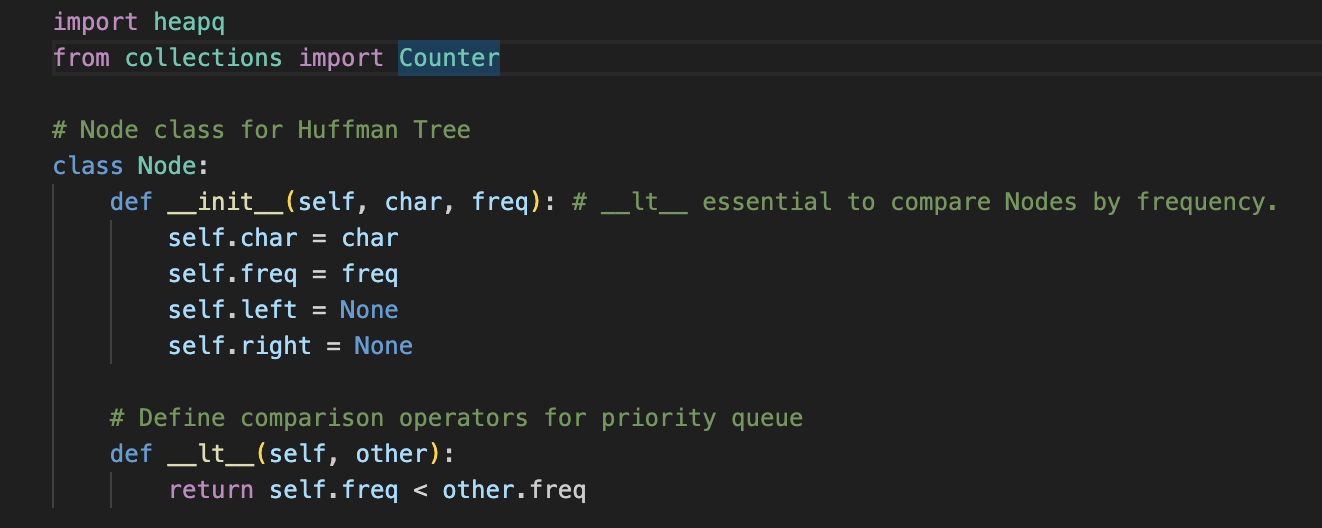
# Implementation

The implementation is written in Python and relies on the built-in modules ***heapq*** and ***collections.Counter***. Python’s ***heapq*** module provides the priority queue (min-heap) functionality that allows the algorithm to efficiently merge the two smallest-frequency nodes in each step of Huffman tree construction. Meanwhile, ***collections.Counter*** helps count how often each character appears in the input text, creating a frequency dictionary in just a single function call.

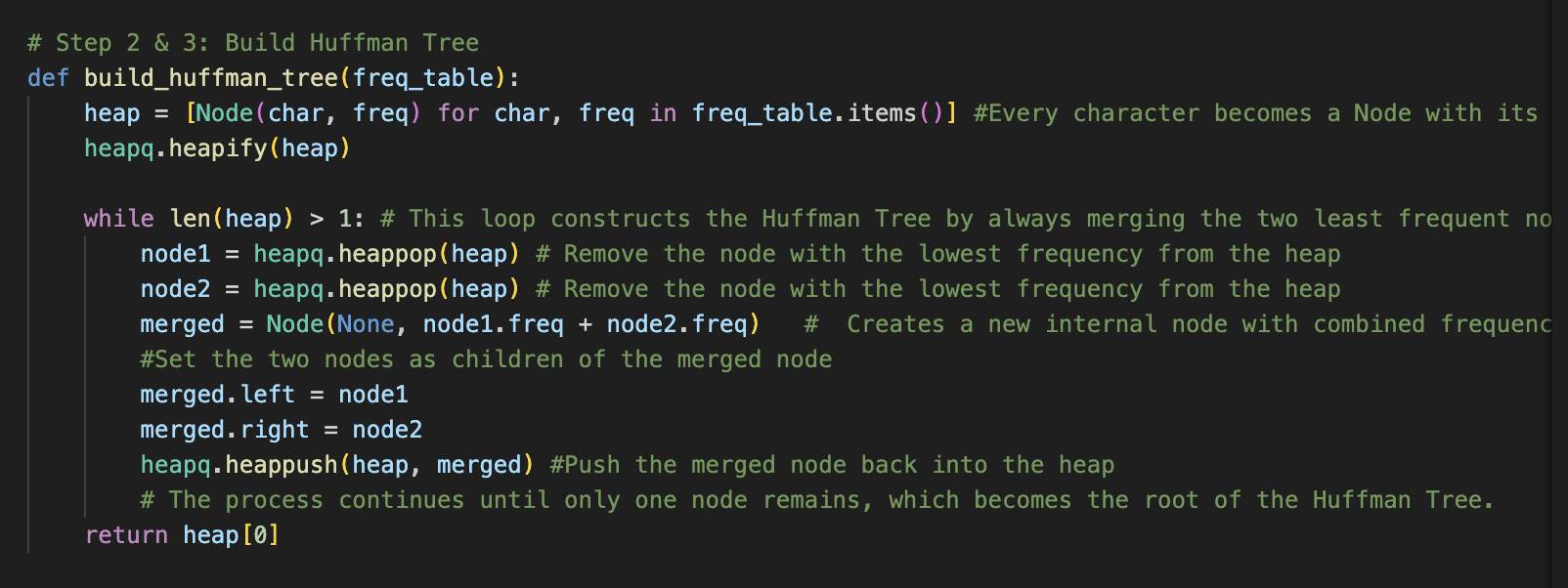
Below is the snippet that shows how the frequency dictionary is built using Counter:



Once the frequency table is created, the Huffman tree is built by turning each character–frequency pair into a ***Node*** object. Each Node stores a character, its frequency, and references to left and right children. To make the priority queue work properly, the Node class defines a comparison method:



The following snippet demonstrates how the Huffman tree is constructed. Each node is pushed into a min-heap, and we then pop the two smallest nodes, merge them into a new parent node, and push that parent back into the heap. This continues until only one node remains in the heap, which becomes the root of the Huffman tree:



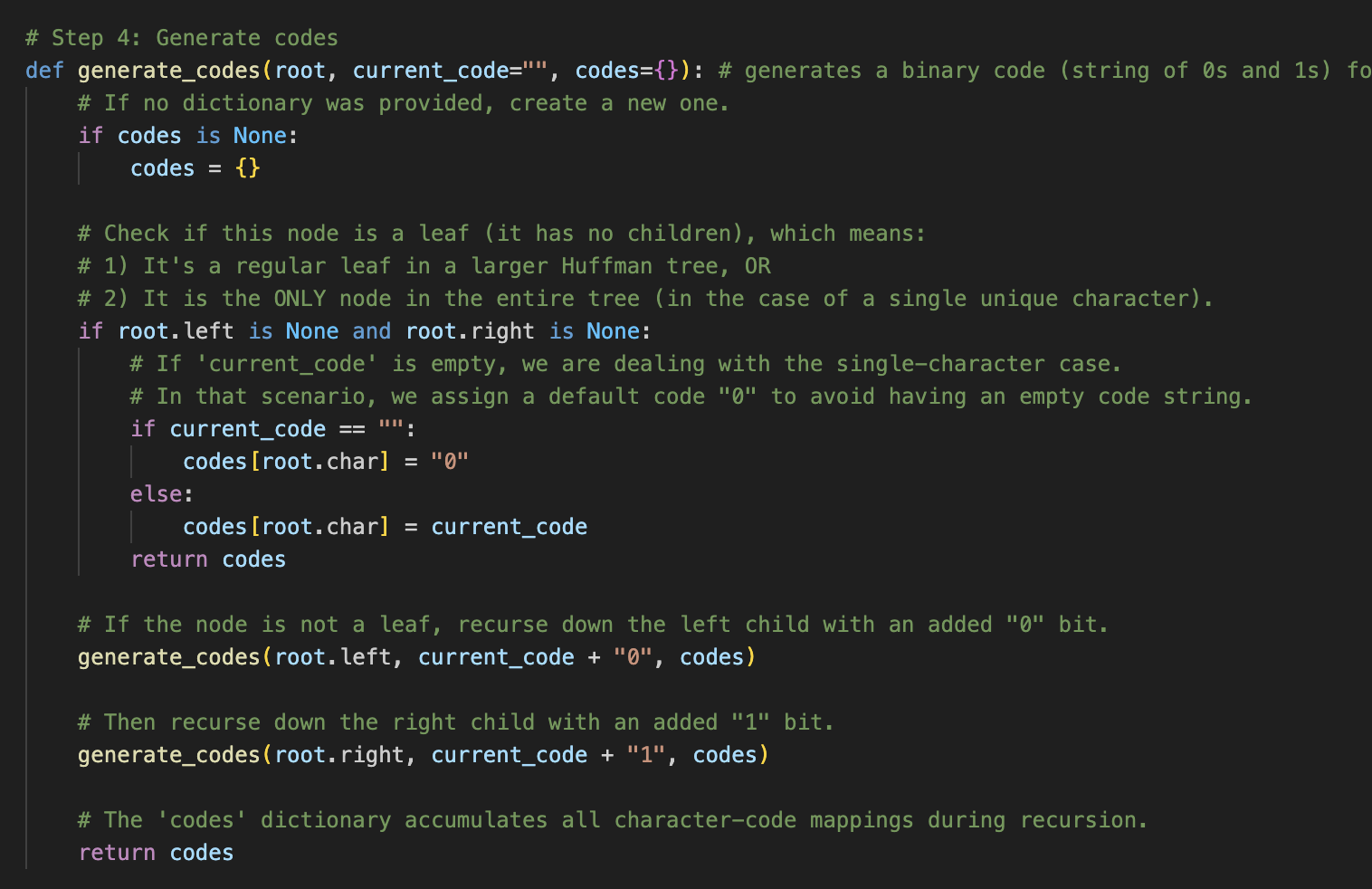
When the tree is ready, this function recursively traverses the Huffman tree to assign a unique binary code to each character. When it starts, it checks if a dictionary for codes was provided; if not, it creates a new empty dictionary. It then determines whether the current node is a leaf by checking if both root.left and root.right are None. This covers two scenarios:

1. A normal leaf in a larger Huffman tree.
2. The case where the entire tree has only one node (i.e., the text contains exactly one unique character).

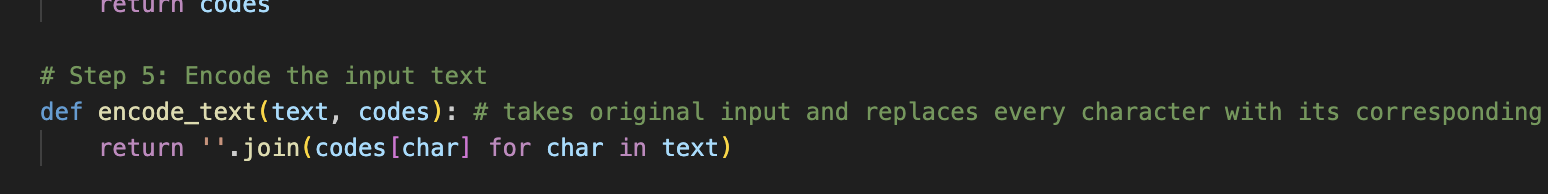
If the node is a leaf and current\_code is still an empty string, that means there is only one character in the whole text, and we must assign a default code like "0" so the encoding is not empty. Otherwise, if current\_code is not empty, we simply store current\_code in the dictionary for the character at this leaf node.

If the node is not a leaf, the function recurses down both branches: it appends a '0' to current\_code when going left, and a '1' when going right. Each time it hits a leaf on these recursive calls, it sets the appropriate code for that character in the codes dictionary.

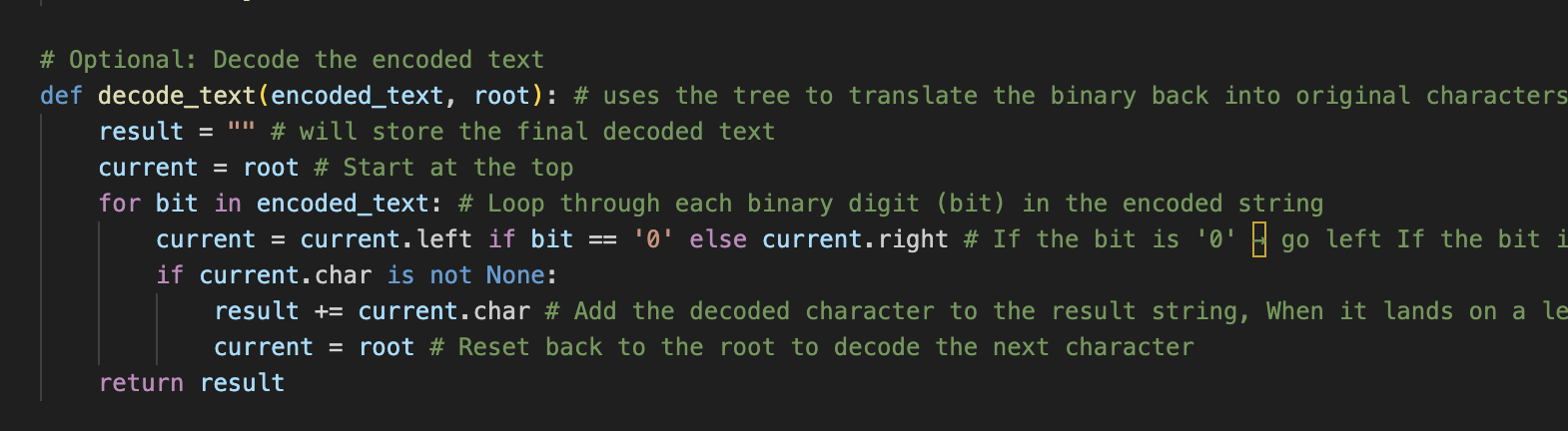
By taking these steps, the function ensures that every character in the tree ends up with a valid binary code, even if there is only one character in the entire text. The final dictionary of character-to-code mappings is then returned.



With these codes in hand, encoding simply replaces each character in the original text with its corresponding code:



Decoding reverses the process by traversing the Huffman tree for each bit. The code first checks if there is only one node in the Huffman tree (i.e., one unique character). If so, it simply returns that character repeated for every bit in the encoded string. Otherwise, it iterates over the bits, moving left for '0' and right for '1'. Once it reaches a leaf node (which has a character), it appends that character to the result and resets to the root to continue decoding.



In the sample usage, the program first counts character frequencies using ***build\_frequency\_table***, builds the Huffman tree, generates the codes, and then encodes and decodes the string “huffman coding.” When printing out the results, you can see the original text, its encoded binary form, and the decoded string. One of the more interesting aspects of this implementation is the need for the \_\_lt\_\_ method in the Node class, which ensures that nodes can be ordered by frequency when inserted into the min-heap. The rest of the code is carefully modularized into distinct functions for clarity, so it is easy to track the steps from frequency counting all the way to verification via decoding.

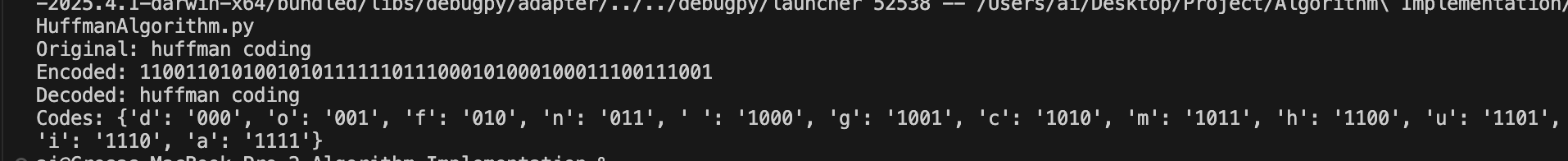
## **Test Cases and Validation**

To ensure the Huffman coding implementation is correct and robust, I tested a variety of input strings. The goal was to confirm that:

1. The correct frequency table was generated for each input.
2. The Huffman tree constructed valid encodings.
3. The decoding process reproduced the original text without error.

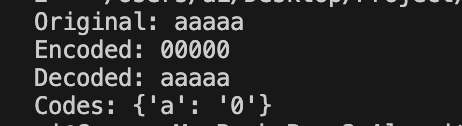
### **Short string**

This test involves encoding the string “***huffman coding***”, which contains repeated characters such as “***f***” and “***m***,” along with a space. The purpose is to verify that the algorithm correctly builds a frequency table, constructs a Huffman tree, assigns appropriate binary codes, and decodes them without error. After running the algorithm, the decoded text matches the original, indicating that all steps—from building the tree to generating codes—work together correctly.



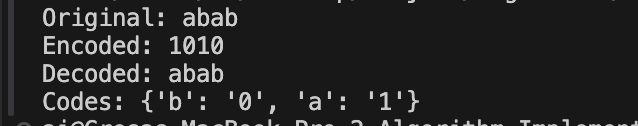
### **Single character**

This case examines the behavior of the algorithm when the input string consists of only one character repeated multiple times, such as “aaaaa.” In such a scenario, Huffman coding typically assigns a single-bit code (“0” or “1”) to that character because there are no other symbols in the frequency table. The successful decoding of the simple binary string (“00000,” for example) back to “aaaaa” confirms that the implementation handles the edge case of a single character properly.



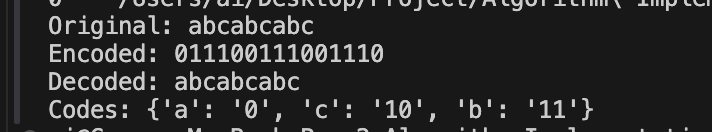
### **Two characters**

A two-character test, such as encoding and decoding “abab,” ensures that each distinct character obtains a unique Huffman code. Since “a” and “b” appear with equal frequency, the algorithm usually assigns one a code of “0” and the other a code of “1.” By verifying that decoding the binary output yields the original “abab,” we confirm that the algorithm’s tree-building and code assignment steps function correctly even in the simplest multi-character scenario.



### **Uniform frequencies**

When all characters appear the same number of times, such as in “abcabcabc,” the algorithm should produce codes of roughly equal length. In this example, “a,” “b,” and “c” each appear three times, so Huffman coding assigns similar-length codes for each letter. Decoding the resulting binary string reproduces “abcabcabc” precisely, showing that the implementation handles uniform frequency distributions without error.



### **Mixed characters and punctuation**

For a test that includes punctuation, uppercase letters, and spaces—e.g., “Hello, World!”—the algorithm still relies on the same process to count frequencies, build the Huffman tree, and generate codes for each character. Spaces, commas, uppercase “H,” and the exclamation mark are all included in the frequency table alongside lowercase letters. The fact that decoding produces the exact string “Hello, World!” confirms that the algorithm properly handles special characters and mixed case letters.

Overall, each test highlights a different aspect of correctness, including the handling of single-character inputs, equally frequent characters, punctuation, and simple repeated patterns. In every instance, the decoded string matches the original text, validating that the Huffman coding procedure—spanning frequency calculation, tree construction, code assignment, and decoding—functions as intended.

