Samantha Lane
Data Structures - Lab 3 Analysis
July 27, 2025

Data Compression Effectiveness
The Huffman encoding implemented successfully compresses data by assigning shorter binary codes to characters that appear more frequently, and longer codes to less frequent ones. This variable-length coding reduces the average number of bits per character compared to fixed-length encodings like ASCII, which always uses 8 bits per character. In the given frequency table, letters like 'L', 'S', and 'N', which show up a lot, will get shorter codes, decreasing the overall size of the encoded data. The amount of compression depends on the input data; data using more high-frequency characters will have more compression, but data with a more equal frequency distribution (less common letters used as often as more common letters in the given data) will have less compression.

Effect of Tie-breaking Schemes
The way ties are broken when building the Huffman tree has a large impact on the final codes and compression. The current tie-breaking prioritizes frequency first, then single-character nodes over multi-character ones, and finally alphabetical order. This strikes a balance between frequency and order, producing a deterministic tree. If we had given alphabetical order higher priority before considering frequency, the tree shape and codes would change, likely leading to less efficient compression, since the codes would not reflect character frequency as well.

Data Structures Used
The main data structure is a binary tree, implemented through a Node class holding character sets, their frequencies, and pointers to child nodes. During tree building, I used a list to store nodes, repeatedly sorting it based on the tie-breaking rules. Dictionaries store both the frequency table and the final mapping of characters to codes, which makes lookup during encoding and decoding very fast (O(1)).

Justification and Appropriateness
A binary tree fits Huffman encoding perfectly because it naturally enforces prefix-free codes and allows efficient traversal. Using a list for node management during construction is simple and practical for letters, despite its less efficient sorting compared to a priority queue. Dictionaries give fast access to frequencies and codes, which is important for handling large texts quickly.

Design Choices
- Processing input line-by-line to keep formatting intact and improve output readability.
- Converting all lines with valid input, but including an error message for lines with invalid characters (especially for encoded input).
- Including Huffman tree in preorder traversal, Huffman code, and result in the output file to simplify debugging and allow the user to see all steps of the conversion process.

Efficiency Analysis
Building the Huffman tree involves sorting the list of nodes repeatedly, which results in a time complexity roughly $O(n^2 \log n)$, where n is the number of unique characters. Since n is typically small (26 letters), this is manageable. Encoding and decoding each run in O(m) time, where m is the length of the input, as they process characters or bits one at a time. Memory-wise, the tree and dictionaries take up O(n) space. In worst cases, when character frequencies are uniform, the compression benefit may be limited.

Lessons Learned
Working on this project deepened my understanding of Huffman coding and the role of binary trees in data compression.. Building the tree according to character frequencies showed me how important the tree shape is for compression efficiency. The tie-breaking rules made me realize how subtle changes in tree construction can affect the final codes and thus the compression ratio. Traversing the tree to generate codes and decode bitstrings helped solidify my understanding of how recursion and tree traversal algorithms work in practice. Overall, implementing Huffman encoding reinforced key concepts from the course about binary trees, recursive algorithms, and the practical application of data structures in programming.

Potential Improvements
If I were to do this again, I would:
- Use a priority queue (min-heap) to improve the tree-building time to O(n log n).
- Support a wider range of characters, including punctuation and whitespace.
- Explore adaptive Huffman coding for streaming data or unknown frequencies.
- Add more robust error handling and input sanitization.
- Improve output formatting to make debugging and verification easier.

Enhancements Beyond Requirements
I preserved the lines of input so the output matches the original formatting. The output file includes the Huffman tree, the character codes, and the final encoded or decoded text, which helps users understand and verify each step. I also added error handling to validate the frequency table format and input text characters to make the program more robust.