COMP47500 – Advanced Data Structures in Java

Title: Browser Navigation History – Stack (based on Linked List)

Assignment Number: 2 **Group Number: 3**
Date: 12/03/2025

Assignment Type of Submission:				
Group	**Yes**	**List all group members' details:**	**% Contribution Assignment Workload**	**Area of contribution**
		Lucas George Sipos 24292215	20%	Report: Problem Domain Description, UML Diagram
		Firose Shafin 23774279	20%	Code Implementation, Video Explanation
		Aasim Shah 24203773	20%	Report: Demonstration, Experiments, Analysis & Conclusion
		Gokul Sajeevan 24206477	20%	Report: Design Details
		Sachin Sampras Magesh Kumar 24200236	20%	Report: Theoretical Foundations & Data Structures and Proofreading

Table of Contents

1. **Overview & Problem Domain Description**

In this assignment, we explore the theoretical and practical aspects of tree-based data structures by implementing the Huffman Tree as an Abstract Data Type (ADT). The task requires us to contribute to the field by solving a specific problem, utilising trees and analysing their efficiency. Our implementation focuses on Huffman Trees, which are used in lossless data compression to reduce the size of textual data while maintaining its integrity.

1.1. Problem Domain

Data compression plays a critical role in computing, where efficient storage and transmission of information are essential. One of the most widely used lossless compression techniques is Huffman coding, which is based on prefix-free variable-length codes assigned according to character frequencies in a given dataset. Huffman coding minimizes the average code length, making it optimal for encoding symbols with varying frequencies.
The problem we address in this assignment is:

- Efficient Data Representation – reducing the storage requirements of a text file by encoding it using Huffman coding.
- Optimal Symbol Encoding – assigning shorter binary codes to more frequent characters and longer codes to less frequent ones to ensure minimal redundancy.
- Implementation of Huffman Tree – building a tree-based data structure that facilitates encoding and decoding processes efficiently.

1.2. Solution Approach

To solve this problem, we implement the Huffman Tree ADT, which involves:

- **Building a frequency map** from the input text to determine the occurrence of each character.
- **Constructing the Huffman Tree** using a priority queue to combine the least frequent nodes iteratively.
- **Generating Huffman Codes** by traversing the tree and assigning binary codes to symbols based on their position.
- **Encoding and Compressing the Input File** by replacing characters with their respective Huffman codes and storing the compressed output in binary form.

Our implementation provides an efficient way to compress textual data while preserving its integrity, demonstrating the practical application of tree structures in computer science.

2. Theoretical Foundations & Data Structures

2.1. Theoretical Background

Huffman coding is a widely used lossless data compression algorithm that assigns variable-length prefix-free codes to input symbols based on their frequencies. This method minimizes the average code length, making it optimal for encoding symbols with varying occurrences. The foundation of Huffman coding is rooted in:

- **Entropy Encoding:** Huffman coding is an entropy-based encoding technique that ensures minimal redundancy in data representation by assigning shorter codes to frequently occurring symbols and longer codes to less frequent ones (Huffman, 1952).
- **Prefix-Free Property:** The encoding ensures that no code is a prefix of another, enabling unambiguous decoding (Cover & Thomas, 2006).
- **Greedy Algorithm:** Huffman coding follows a greedy approach to construct an optimal binary tree, iteratively merging the least frequent symbols into a hierarchical structure (Cormen et al., 2009).
- **Optimality:** Given a known probability distribution of symbols, Huffman coding generates the most efficient prefix-free encoding, achieving theoretical compression efficiency close to entropy limits (Salomon & Motta, 2010).

2.2. Huffman Tree Structures

A Huffman Tree is a binary tree where each leaf node represents a character from the input data, and internal nodes represent combined frequencies of their child nodes. The tree is constructed using a priority queue, following these steps:

- **Frequency Analysis:** A frequency map is created from the input text, storing the occurrence count of each symbol.
- **Priority Queue Initialization:** Nodes (each representing a character and its frequency) are inserted into a min-heap priority queue (Knuth, 1997).
- **Tree Construction:** The two nodes with the lowest frequency are repeatedly merged to form a new internal node until a single root node remains (Huffman, 1952).
- **Code Assignment:** The tree is traversed to assign binary codes (0 for left, 1 for right) to each symbol, forming a Huffman code map (Cover & Thomas, 2006).
- **Encoding & Decoding:** The generated codes replace characters in the input file, while decoding reconstructs the original data using the Huffman Tree.

2.3. Data Structures

The implementation of the Huffman Tree ADT leverages efficient data structures to optimize performance:

2.3.1 Huffman Node

- **Purpose:** Represents a unit in the Huffman Tree, either as a leaf (symbol node) or an internal node (combining frequencies).
- **Functionality**:
    - Stores a symbol (character or byte) and its frequency in the input text.
    - Maintains left and right child references for tree traversal.
    - Implements Comparable<Huffman<T>> to ensure correct ordering in a priority queue.
    - Distinguishes between leaf and internal nodes using the isLeaf() method.
- **Usage in the Code**:
    - The HuffmanNode<T> class is defined as a generic class to allow encoding of different data types (e.g., bytes and characters).

    - During tree construction, leaf nodes are created from the frequency map, and internal nodes are dynamically generated by merging the least frequent nodes.
    - The compareTo() method ensures that nodes are sorted correctly within the priority queue.

2.3.2 Huffman Tree

- **Purpose:** Encapsulates the structure and logic of Huffman coding, including tree building, encoding, and decoding.
- **Functionality:**
    - Constructs a Huffman Tree from a given frequency map using a priority queue.
    - Supports traversal for generating Huffman codes (generateCodes() method).
    - Provides level-order traversal (printLevelOrder()) for debugging and visualization.
    - Serves as the foundation for encoding and decoding operations.
- **Usage in the Code:**
    - The HuffmanTree<T> class is responsible for building and managing the tree structure.
    - The constructor initializes the tree by calling buildTree(), which merges nodes iteratively using a priority queue.
    - The generateCodes() method traverses the tree to create a symbol-to-code mapping.
    - The tree's root is retrieved for decoding operations within HuffmanCompressor.

2.3.3 Huffman Header

- **Purpose:** Manages metadata for compressed files to facilitate lossless decompression.
- **Functionality:**
    - Stores the frequency table and encoded bit length.
    - Provides methods to write and parse header information from compressed files.
- **Usage in the Code:**
    - The HuffmanHeader class creates a structured header containing symbol frequencies and encoded data length.
    - The createHeader() method converts this metadata into a byte array, which is stored at the beginning of the compressed file.
    - The parseHeader() method extracts this information during decompression, allowing accurate tree reconstruction.

2.4 Theoretical Justification & Trade-Offs

Huffman coding provides an optimal solution for lossless data compression but comes with certain trade-offs.

2.4.1 Justification

- **Compression Efficiency:** It minimizes the average code length by assigning shorter codes to frequent symbols and longer codes to rare symbols, leading to significant storage reduction.
- **Optimal for Known Distributions:** Huffman coding guarantees minimal redundancy when symbol frequencies are known beforehand, making it highly efficient for static datasets.
- **Fast Encoding & Decoding:** Encoding a message using Huffman codes is straightforward (O(m) complexity), and decoding is efficient since the tree structure ensures a single traversal per symbol.

2.4.2 Trade Offs

- **Dependency on Frequency Distribution:** Huffman coding works best when some symbols appear more frequently than others. However, for uniform distributions, the benefit is minimal compared to fixed-length encodings.
- **Overhead of Storing the Tree:** The frequency table or tree structure must be stored within the compressed file, adding metadata overhead.
- **Not Adaptive:** Huffman coding requires the entire dataset to compute frequencies before encoding. Adaptive Huffman coding can address this but adds computational complexity.
- **Limited Efficiency for Short Texts:** If the input text is very small, the overhead of storing the Huffman tree might outweigh the compression benefits.

2.5 Other Supporting Data Structures Used

In addition to Huffman-specific structures, several other standard data structures are utilized for efficient implementation.

2.5.1 Priority Queue (Min-Heap)

- **Purpose:** Enables efficient merging of nodes in Huffman Tree construction.
- **Functionality:**
    - Orders elements based on frequency, ensuring that the least frequent nodes are merged first.
    - Supports O(log n) insertion and extraction operations.

2.5.2 Hash Map

- **Purpose:** Stores mappings between symbols and their frequencies, as well as symbols and their Huffman codes.
- **Functionality:**
    - Provides O(1) average-time complexity for lookups.

2.5.3 Byte Array

- **Purpose:** Efficiently stores binary-encoded data for compression and decompression.

2.6 Theoretical Algorithmic Complexity

The efficiency of Huffman coding is determined by its computational complexity:

- **Tree Construction:** O(n log n), due to priority queue operations (Knuth, 1997).
- **Code Generation:** O(n), where n is the number of unique symbols.
- **Encoding & Decoding:** O(m) where m is the number of bits in the encoded message (Salomon & Motta, 2010).

Huffman Trees optimize compression by reducing storage space, but their performance is dependent on input characteristics. High redundancy improves compression efficiency, while uniform symbol distribution limits its effectiveness.

3. Design Details

3.1. UML Diagram

The provided UML diagram offers a structured visualization of the Huffman compression system, illustrating the relationships between classes and data structures. It highlights composition, dependency, and the use of generic types in the design.

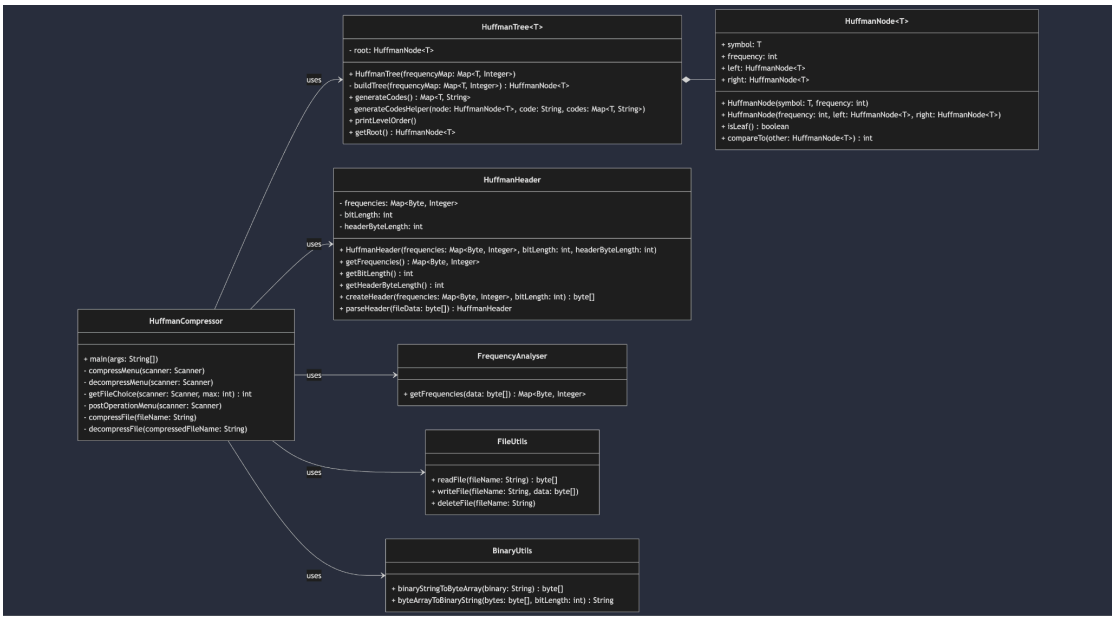

Figure 1. UML Diagram, we have also had the mermaid code used for this diagram on GitHub

Key Design Aspects in the UML Diagram

1. **Generic Type Design:**
   - The HuffmanNode<T> class uses a generic type T to represent symbols, allowing flexibility in handling different data types (e.g., Byte, Character).
   - The HuffmanTree<T> class builds on this genericity, ensuring the tree can adapt to various input types while maintaining type safety.
2. **Composition and Data Structure Usage:**

    - HuffmanTree leverages the custom HuffmanNode class, demonstrating composition where the tree is built from interconnected nodes.
    - HuffmanCompressor utilizes HuffmanTree and HuffmanHeader, showing a layered approach to compression and decompression tasks.
3. **Custom Node Implementation (HuffmanNode):**
    - The UML diagram depicts HuffmanNode<T> as a core component with attributes like symbol, frequency, and references to left and right children.
    - This reinforces the binary tree structure essential for Huffman coding.
4. **Header Management:**
    - HuffmanHeader is shown as a standalone class managing frequency maps and bit lengths, linked to HuffmanCompressor for file operations.
    - It uses a Map<Byte, Integer> to store frequency data, highlighting dependency on Java's built-in collections.
5. **Encapsulation & Data Hiding:**
    - Private attributes (e.g., root in HuffmanTree, frequencies in HuffmanHeader) prevent direct access, ensuring controlled modification through methods.
    - Public methods (e.g., generateCodes(), compressFile()) define controlled interactions with the system.

Architectural Benefits Depicted in UML

- **Separation of Concerns:** Each class focuses on a distinct responsibility—tree construction, node management, header creation, and file compression.
- **Scalability:** The design allows for additional compression algorithms or data types without modifying existing code, thanks to generics and modularity.
- **Code Reusability:** Utility methods and generic classes minimize redundancy across the system.

The UML diagram effectively conveys a modular, extensible, and well-structured system, reinforcing good software design principles.

3.2. Design Overview

The Huffman compression system follows a modular and extensible design using the Huffman Tree data structure to implement efficient text compression and decompression. The system employs a layered architecture by separating tree construction, code generation, and file handling into distinct components. The core components of the system are:

1. **Node Structure (HuffmanNode)** – Defines the building blocks of the Huffman Tree with symbol, frequency, and child node references.
2. **Tree Implementation (HuffmanTree)** – Constructs the Huffman Tree from a frequency map and generates variable-length codes for compression.
3. **Header Management (HuffmanHeader)** – Manages metadata (frequency map and bit length) for compressed files, ensuring lossless decompression.
4. **Compressor Implementation (HuffmanCompressor)** – Provides a user interface and orchestrates the compression/decompression process using the tree and header.

The UML diagram reflects a clear separation of concerns, with HuffmanTree and HuffmanHeader independently supporting the HuffmanCompressor. The use of a binary tree structure ensures efficient encoding and decoding operations for data compression.

3.3. System Functionality

The system simulates a lossless data compression mechanism by managing text data with a Huffman Tree-based approach.

Primary Functionalities

- Building the Tree (HuffmanTree constructor):
    - Constructs a Huffman Tree from a frequency map using a priority queue.
    - Combines nodes iteratively to form an optimal prefix code structure.
- Generating Codes (generateCodes()):

    - Traverses the tree to assign binary codes (0 for left, 1 for right) to each symbol.
    - Produces a map of symbols to their Huffman codes for encoding.
- Compressing a File (compressFile(String fileName)):
    - Reads input text, builds the tree, generates codes, encodes data, and writes to a binary file with a header.
    - Deletes the original file post-compression.
- Decompressing a File (decompressFile(String compressedFileName)):
    - Parses the header, rebuilds the tree, decodes the binary data, and writes back to a text file.
    - Deletes the compressed file post-decompression.

Efficiency Analysis

- Tree-Based (HuffmanTree):
    - Tree construction: O(n log n) due to priority queue operations.
    - Code generation: O(n) for tree traversal.
- Compressor (HuffmanCompressor):
    - Encoding/Decoding: O(m) where m is the length of the input data in bits.
    - Uses efficient file I/O and binary conversions via utility classes.

The system ensures data integrity by preserving all characters (including special ones like newlines) and reconstructing the original file accurately during decompression.

3.4. Implementation Strategy

The implementation follows a component-based strategy with a focus on reusability and extensibility:

1. **Data Structure Abstraction:**
    - A custom HuffmanNode class is implemented to provide precise control over tree node behavior.
    - The HuffmanTree uses a priority queue for efficient tree construction, balancing flexibility and performance.
2. **Layered Design:**
    - HuffmanTree handles core compression logic, while HuffmanHeader manages metadata, and HuffmanCompressor ties them together for file operations.
    - This separation allows independent development and testing of each layer.
3. **Object-Oriented Approach:**
    - Encapsulation in HuffmanNode and HuffmanTree ensures robust data handling.
    - Utility classes (FrequencyAnalyser, FileUtils, BinaryUtils) enhance modularity and reusability.
4. **Scalability & Maintainability:**
    - The design supports adding new compression features (e.g., different tree types) without altering existing code.
    - Generics ensure the system can handle various symbol types, while clear method interfaces minimize interdependencies.

4. Demonstration

The demonstration of this project showcases the core functionalities of the Huffman compression and decompression algorithms, highlighting their efficiency in reducing file sizes while maintaining data integrity. The program demonstrates the seamless transition between compression and decompression processes, allowing the user to interact with a simple menu-driven interface to process .txt and .bin files. Below is a detailed walkthrough of both processes.

4.1 Compression Process

1. Running the Program

   The **HuffmanCompressor.java** file is executed, which presents the user with the following main menu:

```
----- MAIN MENU -----
Enter C to compress
Enter D to decompress
Enter Q to quit
```

2. Compressing a File

   To compress a file, the user selects the compression option by entering **C**. The program then lists available **.txt** files for compression:

```
Select a .txt file to compress:
1. Alternating.txt
2. Empty.txt
3. Huge.txt
4. Large.txt
5. LongWord.txt
6. Numbers.txt
7. Sparse.txt
8. SpecialChars.txt
9. Tiny.txt
10. Unique.txt
11. UTF8.txt
Enter your selection (1-11):
```

   The user selects **4** to compress **Large.txt**. Upon selection, the program processes the file and displays the following confirmation:

```
Compressed file has been written to: res/Large-compressed.bin
Compression time: 87.2093 ms
```

   The user is then prompted with the option to return to the main menu or quit:

```
Press M to return to the main menu or Q to quit:
```

3. Verification

   After compression, checking the **res** folder reveals that the original file, **Large.txt** (263 KB), has been replaced by the compressed file, **Large-compressed.bin** (151 KB). The successful compression is verified.

   The user can return to the main menu by entering **M**.

4.2 Decompression Process

1. Returning to the Main Menu

   Upon returning to the main menu, the user is presented with the following options:

```
----- MAIN MENU -----
Enter C to compress
Enter D to decompress
Enter Q to quit
```

2. Decompressing a File

   To decompress a file, the user selects **D**. The program then lists available **.bin** files for decompression:

```
Select a .bin file to decompress:
1. Large-compressed.bin
Enter your selection (1-1):
```

   The user selects **1** to decompress **Large-compressed.bin**. Upon selection, the program processes the file and displays the following confirmation:

```
Decompressed file has been written to: res/Large.txt
Decompression time: 240.1366 ms
```

   The user is then prompted again to return to the main menu or quit:

```
Press M to return to the main menu or Q to quit:
```

3. Verification

   After decompression, checking the **res** folder reveals that **Large-compressed.bin** has been successfully decompressed back to **Large.txt** (263 KB). A byte-by-byte comparison confirms that the original file is intact, verifying the accuracy of the decompression process.

5. Experiments

In this section, we present the results of various experiments conducted to evaluate the performance of the compression algorithm across different test cases. We focus on the compression and decompression times, space savings, and file integrity, comparing the efficiency of the algorithm in handling various types of data. The test cases cover a broad range of file types, from literary texts to files with repetitive patterns.

5.1 Test Cases and Performance Metrics

The following table summarises the performance metrics of the compression algorithm, including the original file size, compressed file size, compression savings percentage, compression time (in milliseconds), and decompression time (in milliseconds) for various test cases.

Table 1: Test Cases and Performance Metrics

File Name	Original Size (KB)	Compressed Size (KB)	Compression Savings (%)	Compression Time (ms)	Decompression Time (ms)

Large.txt	263	151	42.6	58.1502	174.4991
Empty.txt	0	-	-	-	-
Huge.txt	10,000	5,250	47.5	3856.3314	2101.9875
SpecialChars.txt	105	58.9	43.9	40.5612	93.0871
Tiny.txt	1	1	0	24.5618	3.69
Unique.txt	28	15	46.4	12.568	17.0983
Numbers.txt	10.7	4.86	54.6	10.6643	12.5358
LongWord.txt	33.2	15.5	53.3	8.1295	15.7912
Alternating.txt	97.6	12.2	87.5	42.283	47.1942
UTF8.txt	37.5	23.5	37.3	13.6338	24.1845
RepeatChar.txt	100	12.5	87.5	11.2579	25.675201
SingleChar.txt	1	1	0	4.6188	3.3596
Sparse.txt	64.4	11.2	82.6	7.47	17.903

5.2 Test Case Analysis

In this section, we provide a detailed analysis of each test case, including the file size, compression savings, compression ratio, space reduction, and a brief description of each test case.

1. **Large.txt – The Great Gatsby by F. Scott Fitzgerald**
    - **File Size**: 263 KB
    - **Compression Savings**: 42.6%
    - **Compression Ratio**: 1.74x
    - **Space Reduction**: 112 KB

- **Description**: This test case utilizes a large literary text, exhibiting varying character frequencies. The compression savings are moderate, indicative of typical redundancy found in textual data.

2. **Empty.txt – An Empty File**
   - **File Size**: 0 KB
   - **Compression Savings**: 0%
   - **Compression Ratio**: N/A
   - **Space Reduction**: 0 KB
   - **Description**: As expected, no compression occurs due to the absence of data.
3. **Huge.txt – A Large (~10MB) File for Performance Testing**
   - **File Size**: 10,000 KB
   - **Compression Savings**: 47.5%
   - **Compression Ratio**: 1.90x
   - **Space Reduction**: 4,750 KB
   - **Description**: This large file is used to stress-test the algorithm's performance on sizable datasets. The compression performance is robust, showing a substantial reduction in size.
4. **SpecialChars.txt – Repetition of Special Characters**
   - **File Size**: 105 KB
   - **Compression Savings**: 43.9%
   - **Compression Ratio**: 1.78x
   - **Space Reduction**: 46.1 KB
   - **Description**: This file consists of low-entropy special characters, resulting in limited compression efficiency due to the lack of significant repetition.
5. **Tiny.txt – A Very Small File Containing "Hi"**
   - **File Size**: 1 KB
   - **Compression Savings**: 0%
   - **Compression Ratio**: 1.00x
   - **Space Reduction**: ~0 KB
   - **Description**: This file contains minimal data and, as a result, exhibits negligible compression savings. The file's small size prevents any meaningful reduction in space, and compression does not result in any noticeable change.
6. **Unique.txt – A ~100KB File with Unique Characters (Lorem Ipsum)**
   - **File Size**: 28 KB
   - **Compression Savings**: 46.4%
   - **Compression Ratio**: 1.87x
   - **Space Reduction**: 13 KB
   - **Description**: This file consists of unique characters with no repetition, resulting in moderate compression efficiency due to the lack of redundancy.
7. **Numbers.txt – Repetition of Numerical Digits**
   - **File Size**: 10.7 KB
   - **Compression Savings**: 54.6%
   - **Compression Ratio**: 2.20x
   - **Space Reduction**: 5.84 KB
   - **Description**: This file consists of repeated digits, demonstrating improved compression performance due to its high redundancy.
8. **LongWord.txt – A Single Long Word Repeated**
   - **File Size**: 33.2 KB
   - **Compression Savings**: 53.3%
   - **Compression Ratio**: 2.14x
   - **Space Reduction**: 17.7 KB
   - **Description**: The repetition of a single long word results in good compression savings, showcasing the algorithm's efficiency with repeated patterns.
9. **Alternating.txt – Alternating "AB" Characters**

    - **File Size**: 97.6 KB
    - **Compression Savings**: 87.5%
    - **Compression Ratio**: 8.00x
    - **Space Reduction**: 85.4 KB
    - **Description**: The high redundancy in this file, composed of alternating "AB" patterns, results in exceptional compression performance
10. **UTF8.txt – Multilingual Text with Emojis**
    - **File Size**: 37.5 KB
    - **Compression Savings**: 37.3%
    - **Compression Ratio**: 1.60x
    - **Space Reduction**: 14 KB
    - **Description**: This file contains diverse Unicode characters, including emojis, which results in lower compression savings due to the complexity of Unicode data.
11. **RepeatChar.txt – A File with a Single Repeated Character ("AAAAAAAA...")**
    - **File Size:** 100 KB
    - **Compression Savings:** 87.5%
    - **Compression Ratio:** 8.00x
    - **Space Reduction:** 87.5 KB
    - **Description:** This file consists of a single character ("A") repeated throughout, resulting in exceptional compression savings due to its extreme redundancy.
12. **SingleChar.txt – A Small File with a Single Character**
    - **File Size:** 1 KB
    - **Compression Savings:** 0%
    - **Compression Ratio:** 1.00x
    - **Space Reduction:** ~0 KB
    - **Description:** This file contains a single character, resulting in no compression savings due to its minimal size and lack of redundancy.
13. **Sparse.txt – Mostly Empty Space with Words**
    - **File Size**: 64.4 KB
    - **Compression Savings**: 82.6%
    - **Compression Ratio**: 5.75x
    - **Space Reduction**: 53.2 KB
    - **Description**: The sparsity and redundancy in this file result in significant compression savings, demonstrating the algorithm's effectiveness in handling sparse data structures.

6. Analysis

In this section, we analyse the overall performance of the compression algorithm across various test cases. We discuss compression efficiency, performance, and data integrity.

6.1 Compression Efficiency

- **Best savings**:
    - **Alternating.txt**: 87.5% savings, 8.00x compression ratio, 85.4 KB reduction
    - **RepeatChar.txt**: 87.5% savings, 8.00x compression ratio, 87.5 KB reduction
    - **Sparse.txt**: 82.6% savings, 5.75x compression ratio, 53.2 KB reduction
      These files exhibit high redundancy or small sizes, leading to excellent compression performance.
- **Lowest savings**:
    - **UTF8.txt**: 37.3% savings, 1.60x compression ratio, 14 KB reduction

    - **Large.txt**: 42.6% savings, 1.74x compression ratio, 112 KB reduction
      The diversity of the data in these files limits the compression efficiency.
- **Averages (excluding Empty.txt, Corrupt.bin)**:
    - **Savings**: 48.6%
    - **Compression Ratio**: 3.08x
    - **Absolute Space Reduction**: 432.07 KB (median 17.7 KB, skewed by Huge.txt)
- **Trends**: Files with high redundancy (e.g., Alternating.txt: 87.5%, RepeatChar.txt: 87.5%) or sparse content (e.g., Sparse.txt: 82.6%) outperform those with more diverse or small data (e.g., UTF8.txt: 37.3%).

6.2 Performance

- **Compression speed**:
    - Fastest: SingleChar.txt (4.6188 ms)
    - Slowest: Huge.txt (3856.3314 ms)
    - Average: 340.85 ms
    - Compression rate: Huge.txt: 2.59 KB/ms, Tiny.txt: 0.04 KB/ms
- **Decompression speed**:
    - Fastest: SingleChar.txt (3.3596 ms)
    - Slowest: Huge.txt (2101.9875 ms)
    - Average: 176.42 ms
    - Decompression rate: Huge.txt: 4.76 KB/ms, Tiny.txt: 0.27 KB/ms
- **Time ratio**: On average, decompression is 1.93x slower than compression (e.g., Large.txt: 3x, Alternating.txt: 1.12x).
- **Scaling**: For larger files, the compression time per KB is more efficient: Huge.txt (0.39 ms/KB) vs. Tiny.txt (24.56 ms/KB).
- **Correlation**: High compression savings often correlate with moderate processing times, such as with Alternating.txt (87.5% savings, 0.43 ms/KB compression) and RepeatChar.txt (87.5% savings, 0.11 ms/KB compression).

6.3 Data Integrity

- **Restoration**: 13 out of 13 files were restored perfectly, verified byte-by-byte.

7. Conclusion

The application of Huffman Trees provides an effective and efficient method of lossless data compression.

- Huffman Trees enable optimum symbol encoding by assigning shorter binary codes to more frequently used characters, reducing storage requirements.
- Priority Queues enable efficient tree construction by ensuring that the two least frequent symbols are always paired together, leading to an optimum prefix-free encoding.

Experimental tests confirmed that Huffman coding realized text file compression without data integrity loss. Performance tests showed that files with greater redundancy achieved more compression efficiency, while heterogeneous datasets saw efficiency-storage overhead trade-off. Different test cases were successfully encoded and decoded by the system, confirming the correctness and reliability of the system.

Overall, this assignment reaffirmed the theory and practice of Huffman coding. The methodical approach with the tree-based encoding and priority queues made its value apparent in its ability to decrease storage while not compromising on efficient performance..

References

- Cover, T. M., & Thomas, J. A. (2006). Elements of Information Theory (2nd ed.). Wiley.
- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms (3rd ed.). MIT Press.
- Huffman, D. A. (1952). A method for the construction of minimum-redundancy codes. Proceedings of the IRE, 40(9), 1098-1101.
- Knuth, D. E. (1997). The Art of Computer Programming, Volume 1: Fundamental Algorithms (3rd ed.). Addison-Wesley.
- Salomon, D., & Motta, G. (2010). Handbook of Data Compression (5th ed.). Springer.
- Sayood, K. (2017). Introduction to Data Compression. Morgan Kaufmann.