## **Introduction**

This project implements the **Huffman Encoding Algorithm**, a widely-used technique for lossless data compression. It utilizes the principles of frequency-based binary tree construction to assign variable-length codes to characters. Frequently occurring characters are assigned shorter codes, while less frequent characters are assigned longer codes, ensuring an efficient encoding scheme.

The repository contains modularized C# code that handles the command-line interface, encoding, decoding, and various helper utilities to facilitate the entire compression and decompression workflow.

## **How to Use**

1. Clone the repository and navigate to the project directory.
2. Build the project using your preferred C# development environment (e.g., Visual Studio).
3. Use the command-line interface to encode or decode files:
   * Encoding: dotnet run -- encode <input-file-path> -o <output-file-path>
   * Decoding: dotnet run -- decode <input-file-path> -o <output-file-path>
   * Help: dotnet run -- help

## **Project Structure**

### **internals**

This directory contains the core components of the project, further divided into three submodules:

#### **a. cli (Command-Line Interface)**

* **AppParams.cs**: Defines and manages application parameters, including input file paths, output locations, and flags for encoding or decoding.
* **CLI.cs**: Handles user interactions through the command line, parsing arguments, and invoking the appropriate encoding or decoding workflows.

#### **b. encoding**

* **EncodingHandler.cs**: Implements the logic for generating Huffman codes based on character frequencies. It constructs the Huffman tree and encodes the input data.
* **DecodingHandler.cs**: Decodes compressed files by reconstructing the Huffman tree and converting the encoded bitstream back into readable data.

#### **c. helpers**

A collection of utility classes to support core functionalities:

* **BinaryTreeNode.cs**: Represents nodes in the Huffman tree, holding character data and associated frequencies.
* **BitOperations.cs**: Provides methods for handling bit-level operations, such as shifting and masking, which are essential for efficient compression.
* **BitReader.cs**: Reads encoded bitstreams from files, ensuring accurate data interpretation during decoding.
* **BitWriter.cs**: Writes bit-level data to files, facilitating the generation of compressed outputs.
* **Logger.cs**: Offers logging functionalities for debugging and tracking program execution.

## **Code Documentation**

### **AppParams Class**

The AppParams class serves as a centralized container for application-wide parameters and configurations, enabling seamless management of command-line arguments and flags. It is defined as internal and is accessible within the CLI namespace.

#### **Attributes**

* **sourceLocation** *(string?)*: The file path of the source file to be encoded or decoded.
* **outputLocation** *(string?)*: The target file path for the output file. Defaults to a location in the ./output folder if not specified.
* **isEncoding** *(bool)*: Indicates the mode of operation. true for encoding, false for decoding.
* **isDebug** *(bool)*: Enables debug mode if set to true.

### **CLI Class**

The CLI class manages the parsing and validation of command-line arguments, the primary interaction point for users. It uses robust error-checking mechanisms to ensure proper usage and provides feedback through custom error codes.

#### **Error Codes**

The ErrorCodes enum defines a range of possible errors, ensuring precise error identification during CLI interactions. Notable error codes:

* TOO\_FEW\_ARGUMENTS: Insufficient arguments passed to the CLI.
* UNKNOWN\_MODE: The operation mode (encode or decode) is invalid.
* SOURCE\_FILE\_DOES\_NOT\_EXIST: The source file specified does not exist.
* UNSUPPORTED\_FILE\_EXTENSION: The file extension is unsupported (.txt for encoding, .huffman for decoding).

#### **Methods**

1. **ProcessCommandArguments(string[] args)**:
   * Parses and validates the command-line arguments.
   * Updates AppParams based on valid inputs.
   * Returns an appropriate ErrorCodes value in case of errors.
2. **Example Valid Commands**:
   * encode input.txt -o output.huffman: Encodes the file and outputs it to output.huffman.
   * decode input.huffman --debug: Decodes the file with debug mode enabled.
3. **Flags**:
   * -o <location>: Specifies the output file path.
   * --debug: Enables debug mode.
4. **PrintError(ErrorCodes errorCodes)**:
   * Prints user-friendly error messages corresponding to the ErrorCodes.
5. **PrintHelp()**:
   * Displays usage instructions, including valid flags and examples.
6. **ProcessOutputLocation()** *(Private)*:
   * Validates and determines the output file location based on the sourceLocation and isEncoding flag.
   * Automatically appends the appropriate file extension (.huffman or .txt).

### **EncodingHandler Class**

The EncodingHandler class provides the core implementation of the Huffman encoding process. It processes input text or file streams, generates Huffman codes, and produces compressed binary data using the Huffman coding algorithm.

#### **Methods**

#### **HuffmanEncode(FileStream source)**

Encodes the content of a file stream using the Huffman encoding algorithm.

* **Parameters**:
  + source *(FileStream)*: A file stream containing the data to be encoded.
* **Returns**:
  + A tuple containing:
    - List<bool>? bits: The encoded binary data, or null if an error occurred.
    - ErrorCodes: Status code indicating success or the type of error.
* **Errors**:
  + Returns ErrorCodes.EMPTY\_FILE if the source file is empty.

#### **HuffmanEncode(string source)**

Encodes a string into its Huffman-compressed binary representation.

* **Parameters**:
  1. source *(string)*: The input string to encode.
* **Returns**:
  1. A tuple containing:
     + List<bool>? bits: The encoded binary data, or null if an error occurred.
     + ErrorCodes: Status code indicating success or the type of error.
* **Steps**:
  1. Generates the Huffman table from the input string using GetHuffmanTable.
  2. Encodes the string using the Huffman table.
  3. Merges the Huffman table and binary data into a single binary representation.

#### **MergeTableAndBinData(Dictionary<char, List<bool>> table, List<bool> binData)**

Combines the Huffman table and the encoded binary data into a single binary representation.

* **Parameters**:
  + table *(Dictionary<char, List<bool>>)*: The Huffman encoding table.
  + binData *(List<bool>)*: The encoded binary data for the input text.
* **Returns**:
  + List<bool>: The combined binary representation of the table and encoded data.
* **Details**:
  + Prepends the table size, table entries, and encoded data size to the binary data for decoding purposes.

#### **EncodeString(string source, Dictionary<char, List<bool>> table)**

Encodes the input string using the provided Huffman table.

* **Parameters**:
  + source *(string)*: The input text to encode.
  + table *(Dictionary<char, List<bool>>)*: The Huffman encoding table.
* **Returns**:
  + List<bool>: The encoded binary data.

#### **GetHuffmanTable(string source)**

Generates the Huffman table for the given input string.

* **Parameters**:
  1. source *(string)*: The input text for which to generate the Huffman table.
* **Returns**:
  1. A tuple containing:
     + Dictionary<char, List<bool>>?: The Huffman table mapping characters to their binary codes.
     + ErrorCodes: Status code indicating success or the type of error.
* **Steps**:
  1. Counts character occurrences in the input string using CountCharOccurancesInText.
  2. Builds the Huffman binary tree using GenerateBinaryTree.
  3. Extracts Huffman codes from the tree using GenerateHuffmanTable.

#### **CountCharOccurancesInText(string input)**

Counts the frequency of each character in the input string.

* **Parameters**:
  + input *(string)*: The input text to analyze.
* **Returns**:
  + List<KeyValuePair<char, int>>: A list of characters and their frequencies, sorted by frequency in descending order.

#### **GenerateBinaryTree(List<KeyValuePair<char, int>> list)**

Constructs a Huffman binary tree based on character frequencies.

* **Parameters**:
  + list *(List<KeyValuePair<char, int>>)*: A list of characters and their frequencies.
* **Returns**:
  + BinaryTreeNode<char>: The root node of the Huffman binary tree.
* **Details**:
  + Uses a priority queue to iteratively combine nodes with the lowest frequencies into a binary tree.

#### 

#### **GenerateHuffmanTable(BinaryTreeNode<char> head)**

Generates a Huffman encoding table by traversing the binary tree.

* **Parameters**:
  + head *(BinaryTreeNode<char>)*: The root node of the Huffman binary tree.
* **Returns**:
  + Dictionary<char, List<bool>>: A dictionary mapping each character to its corresponding Huffman code.

#### **WalkTree(BinaryTreeNode<char> node, List<bool> bits, Dictionary<char, List<bool>> dict)**

Recursively traverses the Huffman binary tree to generate binary codes for each character.

* **Parameters**:
  + node *(BinaryTreeNode<char>)*: The current tree node.
  + bits *(List<bool>)*: The binary path (encoding) to the current node.
  + dict *(Dictionary<char, List<bool>>)*: The Huffman table being constructed.

### **Overall Flow**

1. **Character Analysis**: Counts character occurrences in the input text.
2. **Tree Construction**: Builds a binary tree where each leaf node represents a character.
3. **Table Generation**: Extracts Huffman codes by traversing the tree.
4. **Encoding**: Encodes the input text using the Huffman table.
5. **Table & Data Merge**: Combines the table and binary data for output.

### **DecodingHandler Class**

The DecodingHandler class provides functionality for decoding binary data compressed using the Huffman encoding algorithm. It reconstructs the original text by reading the Huffman table and decoding the binary data.

#### **Methods**

### **HuffmanDecode(List<bool> bits)**

Decodes a Huffman-encoded binary representation into the original text.

* **Parameters**:
  + bits *(List<bool>)*: The input binary data containing the Huffman table and encoded text.
* **Returns**:
  + A tuple containing:
    - string: The decoded original string, or an empty string if an error occurred.
    - ErrorCodes: Status code indicating success or the type of error.
* **Implementation**:
  + **Extract Huffman Table**:
    - Reads the size of the Huffman table from the first 16 bits.
    - Iterates through the table entries to map encodings (as binary strings) to characters:
      * Each entry consists of:
        + A character (16 bits).
        + The size of its encoding (8 bits).
        + The encoding itself (variable-length bits based on the size).
  + **Extract Encoded Data Size**:
    - Reads the size of the encoded data from the next 32 bits.
  + **Decode Encoded Data**:
    - Iteratively reads bits and matches them against the Huffman table.
    - When a match is found, the corresponding character is appended to the result string, and the current bits are cleared.
  + **Error Handling**:
    - Returns ErrorCodes.FAILED\_TO\_DECODE\_FILE in case of an exception.
* **Errors**:
  + Returns ErrorCodes.FAILED\_TO\_DECODE\_FILE if the decoding fails due to malformed input or any unexpected issues.

#### **Example Flow**

1. **Input**:
   * bits: A binary list containing:
     + Huffman table size, table entries, encoded data size, and the encoded binary data.
2. **Process**:
   * Extract the Huffman table.
   * Decode the binary data by mapping it to characters using the table.
3. **Output**:
   * Original decoded string.

### **Overall Decoding Process**

1. The method first reconstructs the Huffman table from the encoded data.
2. It then decodes the binary data using this table, mapping the sequences back to their original characters.
3. The final output is the original string or an error code indicating the decoding process's success or failure.

### **BitReader Class Documentation**

The BitReader class provides functionality for reading binary data from a file and converting it into a sequence of bits.

### **Methods**

### **ReadFromFile(FileStream source)**

Reads a binary file and converts its content into a list of boolean values (bits). Each byte in the file is processed into 8 bits, with the most significant bit (MSB) first.

#### **Parameters**

* **source** *(FileStream)*: The file stream to read binary data from.

#### **Returns**

* A tuple containing:
  + **List<bool>**: A list of boolean values representing the bits of the file's binary data, or null if an error occurred.
  + **ErrorCodes**: An enumeration value indicating the success or type of error encountered.

#### **Implementation Details**

1. **Bit Conversion**:
   * Each byte in the file is read sequentially.
   * Each bit of the byte is extracted by performing a bitwise AND operation with a bitmask (1 << i) for bit positions i ranging from 7 (MSB) to 0 (LSB).
   * The result of the operation determines whether the bit is true (1) or false (0), which is then added to the bit list.
2. **Error Handling**:
   * Uses a try-catch block to handle exceptions that might occur during file reading (e.g., file corruption or access issues).
   * If an error occurs, logs the exception using the Logger utility and returns ErrorCodes.FAILED\_TO\_READ\_FROM\_FILE.

#### **Errors**

* **ErrorCodes.FAILED\_TO\_READ\_FROM\_FILE**:
  + Returned if an exception occurs while reading from the file.
  + The method will log the exception for debugging purposes.
* **ErrorCodes.NO\_ERROR**:
  + Returned if the file is successfully read and converted to a bit list.

### **BitWriter Class Documentation**

The BitWriter class provides functionality to write a sequence of boolean values (bits) into a binary file.

### **Methods**

### **WriteToFile(List<bool> data, string fileName, FileMode fm = FileMode.Create, FileAccess fa = FileAccess.Write)**

Writes a list of boolean values (representing bits) into a binary file. The bits are packed into bytes, with each byte consisting of 8 bits, and then written sequentially to the file.

#### **Parameters**

* **data** *(List<bool>)*: The list of boolean values representing the bits to be written to the file.
* **fileName** *(string)*: The full path of the file where the data will be written.
* **fm** *(FileMode, optional)*: Specifies how the file should be opened. Default is FileMode.Create.
  + Common values: FileMode.Create, FileMode.Append.
* **fa** *(FileAccess, optional)*: Specifies the file access permissions. Default is FileAccess.Write.

#### **Returns**

* **ErrorCodes**: An enumeration value indicating the success or type of error encountered:
  + **ErrorCodes.NO\_ERROR**: If the data was successfully written to the file.
  + **ErrorCodes.FAILED\_TO\_WRITE\_TO\_FILE**: If an exception occurred during the write operation.

#### **Implementation Details**

1. **Directory Handling**:
   * Ensures the directory for the specified file exists.
   * If the directory does not exist, it is created using Directory.CreateDirectory.
2. **Bit Packing**:
   * Bits are grouped into bytes (8 bits per byte).
   * A byte variable, currentByte, is used to accumulate bits using bitwise operations:
     + currentByte |= (1 << (7 - index)) sets the corresponding bit position in the byte if the current bit is true.
   * When 8 bits are collected, the byte is written to the file, and the process continues.
3. **Partial Byte Handling**:
   * If there are leftover bits after the last full byte, the remaining bits are written as a single byte.
4. **Error Handling**:
   * Uses a try-catch block to handle exceptions, such as file access or directory creation failures.
   * Logs exceptions using the Logger utility.

#### **Errors**

* **ErrorCodes.FAILED\_TO\_WRITE\_TO\_FILE**:
  + Returned if an exception occurs during file write operations.
  + Logs the exception using Logger.
* **ErrorCodes.NO\_ERROR**:
  + Returned if all bits are successfully written to the specified file.