Assignment 7 – Huffman Coding

Debi Majumdar

CSE 13S – Winter 24

Purpose

Audience for this section: Pretend that you are working in industry, and write this paragraph for your boss. You are answering the basic question, "What does this thing do?". This section can be short. A single paragraph is okay.

Do not just copy the assignment PDF to complete this section, use your own words.

The primary objective of this program is to perform file compression and decompression using Huffman encoding and decoding techniques. Huffman compression operates by identifying frequently occurring characters and representing them with shorter bit sequences. In some cases, this can reduce the size of a single byte from 8 bits to as few as 2 bits.

Decompressing a Huffman-compressed file requires knowledge of the codes assigned to each character, which are stored in a code table within the compressed file. As the code table must also be written to the compressed file, the effectiveness of Huffman compression may diminish when applied to relatively small files.

Goal of Compression

The primary goal of compression is to reduce the size of data or files, making them more efficient to store, transmit, or manipulate. Compression achieves this goal by encoding data in a more compact form, often by identifying and eliminating redundancy or repetitive patterns.

In the case of the string "aaaaaaaaaaaaaaaa", compression is easy because the string consists entirely of the same character, 'a'. Instead of storing each 'a' individually, compression can represent the string more efficiently by storing the character 'a' once and indicating how many times it repeats, significantly reducing the amount of storage space required. This process is known as run-length encoding and is a simple form of data compression.

Types of Compression

Lossy and lossless compression are two different approaches to reducing the size of data, each with its own trade-offs:

0.0.1 Lossless Compression

Lossless compression preserves all original data when compressing and decompressing. This means that the original data can be perfectly reconstructed from the compressed data. Techniques like Huffman coding and run-length encoding are examples of lossless compression. Huffman coding, specifically, falls under this category because it retains all the original information when encoding and decoding. Lossless compression is commonly used for text files, executable programs, and data where accuracy is crucial, such as medical images or scientific data.

0.0.2 Lossy Compression

Lossy compression sacrifices some of the original data during compression to achieve higher levels of compression. While lossy compression can achieve greater compression ratios compared to lossless compression, the reconstructed data may not be identical to the original. Lossy compression algorithms are often used for multimedia data like images, audio, and video, where minor losses in quality are acceptable to achieve significant reductions in file size. For example, the JPEG (Joint Photographic Experts Group) compression standard used for images employs lossy compression. JPEG achieves compression by discarding certain image details that are less perceptible to the human eye, resulting in smaller file sizes but with some loss of image quality.

0.0.3 Lossy Compression of Text Files

As for text files, while lossy compression is not typically applied directly to them due to the risk of losing critical information, text files can be indirectly subject to lossy compression if they are embedded within multimedia files (e.g., text in images or PDFs). In such cases, lossy compression algorithms may be applied to the multimedia file as a whole, affecting the quality of the embedded text along with other multimedia elements. However, when considering text files in isolation, lossless compression methods like Huffman coding are more appropriate for preserving the integrity of the text data.

Input File Handling

The program can accept any file as an input. Since Huffman coding operates on binary data, it can compress both text and binary files. Huffman coding works by analyzing the frequency of symbols in the input data and assigning shorter codes to more frequently occurring symbols, resulting in efficient compression regardless of the file type. Therefore, whether the input file contains text, images, audio, or any other type of data, the Huffman coding algorithm can be applied to compress it.

However, it's important to note that the effectiveness of Huffman coding may vary depending on the characteristics of the input data. For example, files with highly repetitive patterns or a limited set of symbols may achieve higher compression ratios compared to files with more diverse or random data. Additionally, since Huffman coding operates at the byte level, it may not be as effective for highly compressed or already encrypted files, as they may lack discernible patterns that can be exploited for compression.

In summary, while Huffman coding can accept any file as input, the compression performance may vary depending on the nature and characteristics of the input data.

Patterns and Compression Ratios

The size of the patterns found by Huffman Coding depends on the frequency of symbols in the input data. Huffman Coding identifies frequently occurring symbols and assigns shorter codes to them, while less frequent symbols are assigned longer codes. Therefore, the size of the patterns can vary, but they tend to be shorter for more common symbols and longer for less common symbols.

This compression technique lends itself well to files with repetitive patterns or a limited set of symbols, such as text documents, code files, or images with large areas of uniform color.

0.0.4 Analysis of Picture

- **Resolution of the Picture:** The resolution of the picture is typically represented by the number of pixels in width and height. For example, a common resolution for phone pictures is 1920x1080 pixels (Full HD).
- **File Size:** The file size of the picture depends on various factors such as the image format (e.g., JPEG, PNG), compression level, and image content. For instance, a JPEG image with a resolution of 1920x1080 pixels may have a file size ranging from a few hundred kilobytes to several megabytes.
- **Expected Size of the Picture:** If each pixel in the picture takes 3 bytes (one byte each for the red, green, and blue color channels in a typical RGB image), then the expected size of the picture can be calculated by multiplying the resolution (in pixels) by 3 (bytes per pixel). For a 1920x1080 pixel image, the expected size would be $1920 \times 1080 \times 3$ bytes.
- **Compression Ratio:** The compression ratio of the picture can be calculated by dividing the actual size of the image by the expected size. A compression ratio of 1 indicates no compression, while a

ratio below 1 indicates compression. For example, if the actual size of the compressed image is half the expected size, the compression ratio would be 0.5.

- **Second Compression with Huffman Coding:** After compressing the image using Huffman coding, it may still be possible to achieve further compression, but the extent of compression would depend on the characteristics of the image and the efficiency of the initial compression. Since Huffman coding exploits patterns and redundancies in the data, the potential for further compression may be limited if the initial compression already captured most of the available patterns.

Multiple Compression Strategies

There can be multiple ways to achieve the same smallest file size when using compression techniques like Huffman coding. This phenomenon occurs because certain input data may contain redundancies or patterns that can be represented in different ways while still achieving optimal compression.

For example, consider a text document containing the following text:

```
aaaaaabbbbbcccccdddddeeeee
```

In this case, both "a" and "b" occur frequently, while "c", "d", and "e" occur less frequently. Huffman coding will assign shorter codes to "a" and "b" and longer codes to "c", "d", and "e". However, there could be multiple valid Huffman trees that achieve the same compression ratio for this input data.

The flexibility in constructing Huffman trees arises because there can be different arrangements of nodes with the same frequency that result in equally optimal compression. Additionally, different implementations of the compression algorithm may make different choices when constructing the Huffman tree, leading to potentially different but equally efficient compression results.

While there may be multiple ways to achieve the same smallest file size, the key is to ensure that the compression algorithm effectively captures the redundancies and patterns in the data, regardless of the specific tree structure used.

Internal Nodes in Code Tree

When traversing the code tree in Huffman coding, it is not possible for an internal node to have a symbol. In the Huffman coding algorithm, internal nodes represent intermediate combinations of symbols and do not correspond to actual characters or symbols in the input data.

Each internal node in the Huffman tree represents the sum of the frequencies (or weights) of its child nodes. The leaves of the tree, which are the nodes without any children, correspond to the individual symbols in the input data. Therefore, only the leaf nodes in the tree will have symbols associated with them.

During traversal, the algorithm moves from the root of the tree towards the leaves, following branches labeled with "0" or "1" depending on whether it chooses the left or right child. Internal nodes are only encountered as intermediate steps in the traversal process and do not have associated symbols.

Importance of Histogram

We create a histogram instead of randomly assigning a tree in Huffman coding because the histogram provides valuable information about the frequency of occurrence of each symbol in the input data. This frequency information allows us to construct an optimal binary tree that minimizes the average encoding length of symbols.

Randomly assigning a tree would not take into account the frequency distribution of symbols, leading to suboptimal compression results. In Huffman coding, the goal is to assign shorter codes to more frequently occurring symbols and longer codes to less frequent symbols. This approach reduces the overall length of the encoded data, resulting in more efficient compression.

By analyzing the histogram and constructing the Huffman tree based on symbol frequencies, we can ensure that the most common symbols are encoded with the shortest codes, maximizing the compression ratio while preserving the integrity of the original data.

Relation to Morse Code

Huffman coding and Morse code share similarities in their encoding principles, as both aim to represent symbols (characters or letters) with variable-length codes, where more frequently occurring symbols are assigned shorter codes. However, there are significant differences between the two encoding methods.

Morse code is a fixed-length encoding scheme where each symbol (letter or character) is represented by a sequence of dots and dashes (short and long signals). Each symbol in Morse code has a unique representation, but the length of the encoding for each symbol is fixed. In contrast, Huffman coding is a variable-length encoding scheme where the length of the encoding for each symbol depends on its frequency of occurrence in the input data.

Morse code is not suitable for directly encoding text files as binary because it lacks the efficiency provided by variable-length encoding. In Morse code, each symbol is represented by a fixed sequence of signals, which may not be optimal for representing the frequency distribution of symbols in text files. This would result in inefficient use of space, as less frequently occurring symbols would still be encoded with fixed-length sequences, leading to suboptimal compression.

However, if we were to create more symbols for common elements like spaces, newline characters, or other frequently occurring patterns in text files, it could potentially improve the efficiency of Morse code as a binary encoding method. By introducing additional symbols and assigning shorter codes to common patterns, we could better represent the structure and characteristics of text data, resulting in more efficient compression compared to traditional Morse code. Nonetheless, even with these modifications, Morse code may still not achieve the compression efficiency of Huffman coding due to its fixed-length nature.

Testing

I've ran Valgrind to ensure that my program (huff and dehuff) don't have any memory leaks. I've tested multiple combinations of command line options for huff and dehuff. I've also used scan-build to ensure that my Makefile compiles everything correctly.

0.0.5 Error Handling

Errors I expect and how I plan to handle them:

- **bit read open():** If the file cannot be opened, a fatal error will be reported, allocated memory will be freed, opened files will be closed, and the program will exit with an exit code of 1.
- **bit write close():** If there's an error while writing using `fputc()` or closing the file with `fclose()`, a fatal error will be reported.
- **bit write bit():** If there's an error while writing using `fputc()`, a fatal error will be reported.
- **bit read open():** If the file cannot be opened, a fatal error will be reported, allocated memory will be freed, opened files will be closed, and the program will exit with an exit code of 1.
- **bit read close():** If there's an error while closing the file with `fclose()`, a fatal error will be reported.
- **bit read bit():** If there's an error while reading using `fgetc()`, a fatal error will be reported.
- **dequeue():** If trying to remove a Node from an empty PriorityQueue, a fatal error will be reported.
- In `huff.c` and `dehuff.c`: Fatal errors will be reported if an input and/or output file is not found or not specified. Additionally, any opened files will be closed, any allocated memory will be freed, and the program will exit with an exit code of 1.

How to Use the Program

Audience: Write this section for the user of your program. You are answering the basic question, "How do I use this thing?". Don't copy the assignment exactly; explain this in your own words. This section will be longer for a more complicated program and shorter for a less complicated program. You should show how to compile and run your program. You should also describe any optional flags or inputs that your program uses, and what they do.

To show "code font" text within a paragraph, you can use `\lstinline{}`, which will look like this: `text`.

For a code block, use `\begin{lstlisting}` and `\end{lstlisting}`, which will look like this:

```
Here is some code in lstlisting.
```

And if you want a box around the code text, then use `\begin{lstlisting}[frame=single]` and `\end{ lstlisting}`

which will look like this:

```
Here is some framed code (lstlisting) text.
```

Want to make a footnote? Here's how.[1]

Do you need to cite a reference? You do that by putting the reference in the file `bibtex.bib`, and then you cite your reference like this[1][**?**][**?**].

There are two programs here: huff and dehuff. Here is how to compile their executables and run them.

```
$ make clean
$ make
$ make format
$ ./huff -i [file to compress] -o [file to store compressed file result]
$ ./dehuff -i [file to decompress] -o [file to store decompressed file result]
```

- **$ make clean:** Prepares the compiling process by ensuring that everything will be recompiled.
- **$ make:** Compiles all executables and links headers and .o/.c files together. It only recompiles anything that has recently changed since the last **$ make** command.
- **$ make format:** Can be used if you use clang-format and have a configuration file alongside your code. It will add a target to run it with.

Note:

There are 2 mandatory flags for `huff` and `dehuff` and 1 optional flag:

- `-i` is mandatory and needs to be followed by an argument. This argument will tell `huff/dehuff` what file to read input from.
- `-o` is mandatory and needs to be followed by an argument. This argument will tell `huff/dehuff` what file to read output from.
- `-h` is optional and will print a help message to stdout.

Program Design

Audience: Write this section for someone who will maintain your program. In industry you maintain your own programs, and so your audience could be future you! List the main data structures and the main algorithms. You are answering the basic question, "How is this thing organized so that I can have a chance of fixing it?". This section will be longer for a more complicated program and shorter for a less complicated program.

The functions mentioned here were provided in the assignment pdf as well as their implementations by Jess Srinivas and Professor Veenstra. Here's an overview:

[1]This is my footnote.

0.0.6 Data Structures

The main data structures used in our program are:

BitWriter

The BitWriter struct, defined in `bitwriter.c`, comprises the following members:

```
struct BitWriter {
    FILE *underlying_stream;
    uint8_t byte;
    uint8_t bit_position;
};
```

The `underlying_stream` represents where the BitWriter writes bits to, `byte` denotes the current byte being written, and `bit_position` helps keep track of the current bit being written.

BitReader

The BitReader struct, defined in `bitreader.c`, consists of the following members:

```
struct BitReader {
    FILE *underlying_stream;
    uint8_t byte;
    uint8_t bit_position;
};
```

Similar to BitWriter, `underlying_stream` denotes where the BitReader reads bits from, `byte` represents the current byte being read, and `bit_position` tracks the current bit being read.

Node

The Node struct, defined in `node.h`, includes the following members:

```
struct Node {
    uint8_t symbol;
    uint32_t weight;
    uint64_t code;
    uint8_t code_length;
    Node *left;
    Node *right;
};
```

Here, `symbol` represents the character represented by the Node, `weight` denotes the character's frequency in the input file, `code` is the binary code assigned to the Node, `code_length` is the length of the Node's code, and `left` and `right` are pointers to any children Nodes.

ListElement

The ListElement struct, used in implementing a priority queue ADT, is defined in `pq.c`:

```
struct ListElement {
    Node *tree;
    ListElement *next;
};
```

It represents an element in the linked list, with `tree` pointing to a Node and `next` pointing to the next ListElement.

PriorityQueue

The PriorityQueue struct, also defined in `pq.c`, consists of a single member:

```
struct PriorityQueue {
    ListElement *list;
};
```

Here, `list` is a pointer to the first ListElement in the priority queue.

Code

The Code struct, defined in `code.c`, tracks the codes and code lengths of each node for creating the code table:

```
typedef struct Code {
    uint64_t code;
    uint8_t code_length;
} Code;
```

It includes an unsigned 64-bit integer called `code` and an unsigned 8-bit integer called `code_length`.

Pseudocode

Give the reader a top down description of your code! How will you break it down? What features will your code have? How will you implement each function.

Listing 1: My pseudocode for huff.c:

```
Include necessary header files

Defne function fill_histogram:
Input: File pointer to input file (fin), Array of 256 unsigned 32-bit integers (histogram)
Output: Unsigned 32-bit integer (filesize)

1. Initialize all elements of the histogram array to 0.
2. Initialize filesize to 0.
3. Read bytes from the input file until end of file:
   a. Read a byte (b) from the input file.
   b. If b is EOF, exit the loop.
   c. Increment the histogram at index b by 1.
   d. Increment the filesize by 1.
4. Increment histogram[0x00] and histogram[0xff] by 1 to ensure at least two non-zero values in the
     histogram array.
5. Rewind the input file.
6. Return the filesize.

Defne function create_tree:
Input: Array of 256 unsigned 32-bit integers (histogram), Pointer to unsigned 16-bit integer (
    num_leaves)
Output: Pointer to Node (huffman_tree)

1. Create an empty priority queue (priority_queue).
2. Iterate over each index i from 0 to 255:
   a. If histogram[i] is greater than 0:
      i. Create a new Node (new_node) with symbol i and weight histogram[i].
      ii. Enqueue new_node into the priority_queue.
      iii. Increment num_leaves by 1.
```

```
3. While priority_queue is not empty and its size is greater than 1:
   a. Dequeue two nodes (left and right) from the priority_queue.
   b. Create a new Node (new_node) with symbol 0 and weight equal to the sum of left's weight and
       right's weight.
   c. Set new_node's left child to left and right child to right.
   d. Enqueue new_node into the priority_queue.
4. Dequeue a node from the priority_queue and assign it to huffman_tree.
5. Free memory allocated for priority_queue.
6. Return huffman_tree.

Defne function fill_code_table:
Input: Array of Code structures (code_table), Pointer to Node (node), Unsigned 64-bit integer (code
    ), Unsigned 8-bit integer (code_length)
Output: None

1. If node's left child is NULL:
   a. Assign code to code_table[node->symbol].code.
   b. Assign code_length to code_table[node->symbol].code_length.
2. Else:
   a. Recursively call fill_code_table with node's left child, code, and code_length + 1.
   b. Update code by bitwise OR operation with (1 << code_length).
   c. Recursively call fill_code_table with node's right child, updated code, and code_length + 1.

Defne function huff_write_tree:
Input: Pointer to BitWriter (outbuf), Pointer to Node (node)
Output: None

1. If node's left child is NULL:
   a. Write 1 bit to outbuf.
   b. Write node's symbol as an 8-bit unsigned integer to outbuf.
2. Else:
   a. Recursively call huff_write_tree with outbuf and node's left child.
   b. Recursively call huff_write_tree with outbuf and node's right child.
   c. Write 0 bit to outbuf.

Defne function huff_compress_file:
Input: Pointer to BitWriter (outbuf), File pointer to input file (fin), Unsigned 32-bit integer (
    filesize), Unsigned 16-bit integer (num_leaves), Pointer to Node (code_tree), Array of Code
    structures (code_table)
Output: None

1. Write 'H' and 'C' as 8-bit unsigned integers to outbuf.
2. Write filesize as a 32-bit unsigned integer to outbuf.
3. Write num_leaves as a 16-bit unsigned integer to outbuf.
4. Call huff_write_tree with outbuf and code_tree.
5. While true:
   a. Read a byte (b) from the input file.
   b. If b is EOF, exit the loop.
   c. Get the code and code length for b from code_table.
   d. Write code_length bits of code to outbuf.
6. End of function.

Defne function main:
Input: Command line arguments (argc and argv)
Output: Integer indicating success or failure (0 for success, non-zero for failure)

1. Initialize input_filename and output_filename to NULL.
2. Iterate over command line arguments:
   a. If argument is "-i" and next argument exists:
      i. Assign the next argument to input_filename.
      ii. Increment loop index.
   b. If argument is "-o" and next argument exists:
      i. Assign the next argument to output_filename.
      ii. Increment loop index.
   c. If argument is "-h":
      i. Print usage message.
      ii. Return success (0).
3. If input_filename or output_filename is NULL:
   a. Print error message.
   b. Return failure (1).
4. Open input file for reading in binary mode.
5. If input file is NULL:
   a. Print error message.
   b. Return failure (1).
6. Fill histogram from input file and get filesize.
7. Increment histogram[0x00] and histogram[0xff] by 1.
8. Create huffman_tree from histogram.
9. Fill code_table using huffman_tree.
10. Open input file again for reading.
11. If input file is NULL:
    a. Print error message.
    b. Return failure (1).
12. Open output file for writing using BitWriter.
13. If output buffer is NULL:
    a. Print error message.
    b. Close input file.
    c. Return failure (1).
14. Compress file using huff_compress_file function.
15. Close input and output files.
16. Free memory allocated for huffman_tree.
17. Return success (0).
```

Listing 2: My pseudocode for dehuff.c:

```
Include necessary header files

Define Function: main
Input: Command line arguments (argc and argv)
Output: Integer indicating success or failure (0 for success, non-zero for failure)

1. Initialize input_filename and output_filename to NULL.
2. Iterate over command line arguments:
   a. If argument is "-i" and next argument exists:
      i. Assign the next argument to input_filename.
      ii. Increment loop index.
   b. If argument is "-o" and next argument exists:
      i. Assign the next argument to output_filename.
      ii. Increment loop index.
   c. If argument is "-h":
      i. Print usage message.
      ii. Return success (0).
3. If input_filename or output_filename is NULL:
   a. Print error message.
   b. Return failure (1).
4. Open input file for reading using BitReader.
5. If input file is NULL:
   a. Print error message.
   b. Return failure (1).
```

```
6. Open output file for writing in binary mode.
7. If output file is NULL:
   a. Print error message.
   b. Close input file.
   c. Return failure (1).
8. Decompress the input file using dehuff_decompress_file function.
9. Close input and output files.
10. Return success (0).

Define Function: dehuff_decompress_file
Input: File pointer to output file (fout), BitReader pointer to input buffer (inbuf)
Output: None

1. Read header information from input buffer:
   a. Read type1 as an 8-bit unsigned integer.
   b. Read type2 as an 8-bit unsigned integer.
   c. Read filesize as a 32-bit unsigned integer.
   d. Read num_leaves as a 16-bit unsigned integer.
2. Check if type1 and type2 are equal to 'H' and 'C' respectively.
3. Calculate the number of nodes in the Huffman tree as 2 * num_leaves - 1.
4. Initialize a stack of Node pointers with size 64 and top index as -1.
5. Reconstruct the Huffman tree from serialized representation:
   a. Loop over each node index until all nodes are processed:
      i. While the top of the stack is greater than or equal to 0:
         - Read a bit from the input buffer.
         - If the bit is 1:
            * Read a symbol as an 8-bit unsigned integer.
            * Create a new leaf node with the symbol and weight 0.
         - Else:
            * Create a new internal node with symbol 0 and weight 0.
            * Pop two nodes from the stack and set them as the right and left children of the new
               node.
         - Push the new node onto the stack.
6. Retrieve the root of the Huffman tree from the top of the stack.
7. Decompress the data using the Huffman tree:
   a. Loop over each byte until the total number of bytes decompressed equals filesize:
      i. Initialize a current node pointer to the root of the Huffman tree.
      ii. While the current node is not a leaf node:
          - Read a bit from the input buffer.
          - If the bit is 0, move to the left child of the current node.
          - Else, move to the right child of the current node.
      iii. Write the symbol of the current node to the output file.
8. Free memory allocated for the Huffman tree.
```

Function Descriptions

The function descriptions for the functions provided in the assignment by the Professor and tutors are mentioned here:

BitWriter Functions:

0.0.7 `BitWriter *bit_write_open(const char *filename)`

- Opens a binary file specified by `filename` for writing using `fopen()` and returns a pointer to a newly created `BitWriter` structure.
- Initializes the underlying stream and the internal byte buffer.
- Returns `NULL` on failure, and it is essential to check all function return values.

0.0.8 `void bit_write_close(BitWriter **pbuf)`

- Trashes any remaining data in the byte buffer, closes the underlying stream, frees the `BitWriter` object, and sets the pointer to `NULL`.
- It is crucial to check all function return values and report fatal errors if any occur.

0.0.9 `void bit_write_bit(BitWriter *buf, uint8_t bit)`

- Writes a single bit (`bit`) to the binary file using values in the `BitWriter` structure.
- Collects 8 bits into the buffer before writing it using `fputc()`.
- Checks all function return values and reports fatal errors if any occur.

0.0.10 `void bit_write_uint8(BitWriter *buf, uint8_t x)`

- Writes the 8 bits of the function parameter x by calling `bit_write_bit()` 8 times.
- Ensures correct bit alignment within the binary file.

0.0.11 `void bit_write_uint16(BitWriter *buf, uint16_t x)`

- Writes the 16 bits of the function parameter x by calling `bit_write_bit()` 16 times.
- Ensures correct bit alignment within the binary file.

0.0.12 `void bit_write_uint32(BitWriter *buf, uint32_t x)`

- Writes the 32 bits of the function parameter x by calling `bit_write_bit()` 32 times.
- Ensures correct bit alignment within the binary file.

BitReader Functions:

0.0.13 `BitReader *bit_read_open(const char *filename)`

- Opens a binary file specified by `filename` for reading using `fopen()` and returns a pointer to a newly created `BitReader` structure.
- Initializes the underlying stream and the internal byte buffer.
- Returns `NULL` on failure, and it is essential to check all function return values.

0.0.14 `void bit_read_close(BitReader **pbuf)`

- Closes the underlying stream and frees the `BitReader` object, setting the pointer to `NULL`.
- Checks all function return values and reports fatal errors if any occur.

0.0.15 `uint8_t bit_read_bit(BitReader *buf)`

- Reads a single bit from the binary file using values in the `BitReader` structure.
- Manages the byte buffer and ensures correct bit extraction.

0.0.16 `uint8_t bit_read_uint8(BitReader *buf)`

- Reads 8 bits from the binary file by calling `bit_read_bit()` 8 times.
- Collects these bits into a `uint8_t` starting with the least significant bit.

0.0.17 `uint16_t bit_read_uint16(BitReader *buf)`

- Reads 16 bits from the binary file by calling `bit_read_bit()` 16 times.
- Collects these bits into a `uint16_t` starting with the least significant bit.

0.0.18 `uint32_t bit_read_uint32(BitReader *buf)`

- Reads 32 bits from the binary file by calling `bit_read_bit()` 32 times.
- Collects these bits into a `uint32_t` starting with the least significant bit.

Node Functions:

0.0.19 `Node *node_create(uint8_t symbol, uint32_t weight)`

- Creates a new Node and sets its symbol and weight fields.
- Returns a pointer to the new Node on success and `NULL` on failure.

0.0.20 `void node_free(Node **pnode)`

- Frees the memory occupied by the Node pointed to by `*pnode` and sets it to `NULL`.

0.0.21 `void node_print_tree(Node *tree, char ch, int indentation)`

- Diagnostics and debugging function for printing the tree structure.
- Prints a sideways view of the binary tree using text characters.

Priority Queue Functions:

0.0.22 `PriorityQueue *pq_create(void);`

- Allocates a `PriorityQueue` object and returns a pointer to it.
- Returns `NULL` on failure.

0.0.23 `void pq_free(PriorityQueue **q);`

- Frees the memory occupied by the `PriorityQueue` pointed to by `*q` and sets it to `NULL`.

0.0.24 `bool pq_is_empty(PriorityQueue *q);`

- Returns `true` if the priority queue is empty (list field is `NULL`), otherwise returns `false`.

0.0.25 `bool pq_size_is_1(PriorityQueue *q);`

- Returns `true` if the priority queue contains a single element, otherwise returns `false`.

0.0.26 `void enqueue(PriorityQueue *q, Node *tree);`

- Inserts a tree into the priority queue, keeping the tree with the lowest weight at the head.
- Handles various cases, including an empty queue or inserting before/after existing elements.

0.0.27 `Node *dequeue(PriorityQueue *q);`

- Removes the queue element with the lowest weight and returns it.
- Reports a fatal error if the queue is empty.

0.0.28 `void pq_print(PriorityQueue *q);`

- Diagnostic function for printing the trees of the queue.

Huffman Coding Functions:

0.0.29 `uint32_t fill_histogram(FILE *fin, uint32_t *histogram);`

- Updates a histogram array with the number of occurrences of each unique byte value in the input file.
- Returns the total size of the input file.

0.0.30 `Node *create_tree(uint32_t *histogram, uint16_t *num_leaves);`

- Creates a Huffman tree from the histogram and returns a pointer to the root node.
- Updates `num_leaves` with the number of leaf nodes in the tree.

0.0.31 `void fill_code_table(Code *code_table, Node *node, uint64_t code, uint8_t code_length);`

- Recursively fills a code table for each leaf node's symbol in the Huffman tree.
- The code table is an array of `Code` objects, each containing a code and code length.

0.0.32 `void huff_compress_file(BitWriter *outbuf, FILE *fin, uint32_t filesize, uint16_t num_leaves, Node *code_tree, Code *code_table);`

- Writes a Huffman-coded file using the provided `BitWriter`, input file, file size, Huffman tree, and code table.

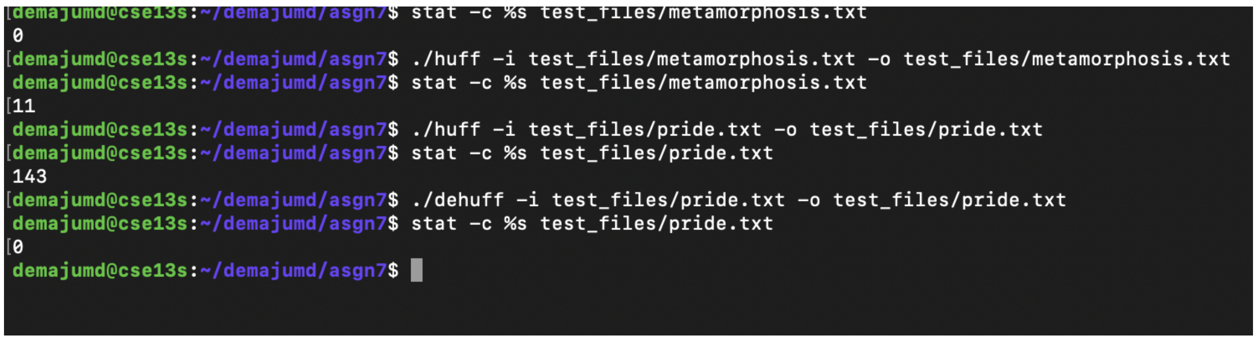
Huffman Decoding Function:

0.0.33 `void dehuff_decompress_file(FILE *fout, BitReader *inbuf);`

- Reads a Huffman-coded file using the provided `BitReader` and writes the decompressed output to the specified file.

Results

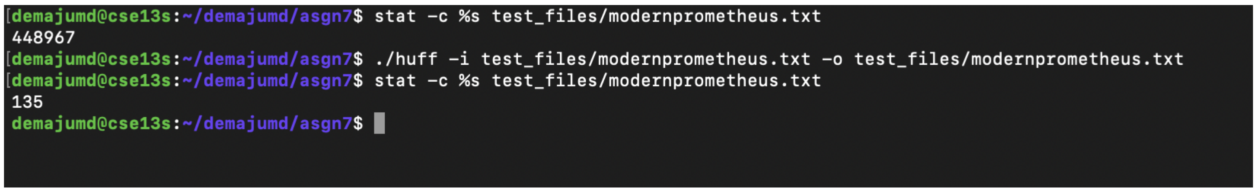
I compressed and decompressed the metamorphpsis.txt too many times. So when testing for pride.txt this is what I got.



```
[demajumd@cse13s:~/demajumd/asgn7$ stat -c %s test_files/metamorphosis.txt
0
[demajumd@cse13s:~/demajumd/asgn7$ ./huff -i test_files/metamorphosis.txt -o test_files/metamorphosis.txt
demajumd@cse13s:~/demajumd/asgn7$ stat -c %s test_files/metamorphosis.txt
[11
demajumd@cse13s:~/demajumd/asgn7$ ./huff -i test_files/pride.txt -o test_files/pride.txt
[demajumd@cse13s:~/demajumd/asgn7$ stat -c %s test_files/pride.txt
143
[demajumd@cse13s:~/demajumd/asgn7$ ./dehuff -i test_files/pride.txt -o test_files/pride.txt
demajumd@cse13s:~/demajumd/asgn7$ stat -c %s test_files/pride.txt
[0
demajumd@cse13s:~/demajumd/asgn7$ 
```

Figure 1: Testing some test_files

When testing for compression ratio for modernprometheus.txt I got the original value as = 448967 and compressed as = 135. So the ratio being around 3325:1 for this first compression.



```
[demajumd@cse13s:~/demajumd/asgn7$ stat -c %s test_files/modernprometheus.txt
448967
[demajumd@cse13s:~/demajumd/asgn7$ ./huff -i test_files/modernprometheus.txt -o test_files/modernprometheus.txt
[demajumd@cse13s:~/demajumd/asgn7$ stat -c %s test_files/modernprometheus.txt
135
demajumd@cse13s:~/demajumd/asgn7$
```

Figure 2: Calculating Compression Ratio

References

[1] Wikipedia contributors. C (programming language) — Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/C_(programming_language), 2023. [Online; accessed 20-April-2023].