Assignment 7 – Huffman Coding

Nickolas Tran

CSE 13S – Winter 24

Purpose

This program implements Huffman coding for lossless data compression and decompression. Huffman coding is a widely used algorithm for encoding data more efficiently by assigning shorter codes to more frequent symbols and longer codes to less frequent symbols. The program consists of functions for constructing a Huffman tree from the input data, encoding the data using the generated Huffman codes, compressing the data into a binary format, and decompressing the compressed data back to its original form. It employs various data structures such as nodes for constructing the Huffman tree, a priority queue for efficiently merging nodes based on their frequencies, and a code table for storing the Huffman codes assigned to each symbol.

Questions

- Describe the goal of compression. (As a hint, why is it easy to compress the string "aaaaaaaaaaaaaaa")
  To reduce the size of data, typically in digital formats like images, audio, video, or text, while preserving its essential information and minimizing loss of quality. Compression is employed to save storage space, reduce transmission time, and optimize resources such as bandwidth.

- What is the difference between lossy and lossless compression? What type of compression is Huffman coding? What about JPEG? Can you lossily compress a text file?
  Lossy: Compressed data resembles the original data and non reversible.
  Lossless: Compressed data does not resembles the original data and reversible.
  Huffman coding is lossless compression.
  JPEG is lossy compression.

- Can your program accept any file as an input? Will compressing a file using Huffman coding make it smaller in every case?
  My program can accept any file. Huffman coding will not make a file smaller every time.

- How big are the patterns found by Huffman coding? What kind of files does this lend itself to?
  The patterns depends on the frequency of occurrence of characters in the input data. The files that it lend itself to are text files, source code files, structured data files, and encoded files.

- Take a picture on your phone. What resolution is the picture? How much space does it take up in your storage (in bytes)?
  The resolution is 3024x4032. It takes up 2097152 bytes.

- If each pixel takes 3 bytes, how many bytes would you expect the picture you took to take up? Why do you think that image you took is smaller.
  It would take 36578304 bytes.

- What is the compression ratio of the picture you just took? To get this, divide the actual size of the image by the expected size from the question above. You should not get a number above 1.
  0.05733322135

- Do you expect to be able to compress the image you just took a second time with your Huffman program? Why or why not?
  Compressing a file that has already been compressed using Huffman coding is unlikely to result in significant further reduction in file size.
- Are there multiple ways to achieve the same smallest file size? Explain why this might be.
  There are multiple ways through data being compressed and the flexibility of the compression algorithms.
- When traversing the code tree, is it possible for an internal node to have a symbol?
  It is not possible, internal nodes do not have associated symbols because they represent clusters of symbols rather than individual symbols themselves.
- Why do we bother creating a histogram instead of randomly assigning a tree.
  It achieves optimal compression efficiency by prioritizing more frequent symbols with shorter codes whereas randomly assigning a tree would not exploit the statistical properties of the data, resulting in sub optimal compression.
- Relate this Huffman coding to Morse code. Why does Morse code not work as a way of writing text files as binary. What if we created more symbols for things like space and newlines?
  It wouldn't be as efficient or effective as Huffman coding for compression purposes. Introducing more symbols for space and newlines might reduce issues, but it wouldn't change the fixed-length nature of Morse code encoding.
- Using the example binary, calculate the compression ratio of a large text of your choosing.
  I used `pride.txt` and got a compression ratio of 0.57

Testing

Integration Testing: Test the integration of all functions together to ensure that they work correctly when combined. This involves testing the compression and decompression process.
Input Validation Testing: Test with various input file types: binary files, text files, empty files, files with large content, files with small content, etc. Test with inputs that have varying frequencies of symbols to ensure the algorithm handles different distributions well.
Error Handling Testing: Test error handling mechanisms for cases such as invalid input file paths, insufficient memory allocation, inability to open or write to files, etc.
Performance Testing: Test the performance of compression and decompression on large input files to ensure they complete in a reasonable amount of time.
Usability Testing: Evaluate the usability of the compression and decompression tools by performing tests with users from different backgrounds. Collect feedback on the user interface, ease of use, and overall experience.

How to Use the Program

The user uses the program to compile and run the code

```
make
make clean
```

The user then runs the huff program to compress the file of choice

```
./huff -i infile -o outfile
```

The user then runs the dehuff program to decompress the file of choice

```
./dehuff -i infile -o outfile
```

Program Design

0.1 Main Data Structures

- Node Structure: Represents a node in the Huffman tree. Contains fields for the symbol, weight, Huffman code, code length, and pointers to left and right child nodes. Used for constructing the Huffman tree during compression and for traversing the tree during decompression.
- Priority Queue (PQ): Implemented using a linked list. Used for efficiently storing and accessing nodes based on their weights during the construction of the Huffman tree.
- Code Table: An array of structures to store the Huffman codes for each symbol. Each structure contains fields for the symbol, its corresponding Huffman code, and the code length. Used for encoding symbols during compression and decoding Huffman codes during decompression.
- BitReader and BitWriter: Structures representing bit-level input and output streams. Used for reading from and writing to files at the bit level during compression and decompression.

0.2 Main Algorithms

- Huffman Tree Construction (create_tree): Builds a Huffman tree from the histogram of symbol frequencies. Uses a priority queue to efficiently merge nodes with the lowest frequencies into subtrees until a single tree is formed.
- Huffman Coding (fill_code_table): Traverses the Huffman tree to assign Huffman codes to each symbol. Recursively traverses the tree, assigning a binary code to each symbol based on its position in the tree.
- Compression (huff_compress_file): Reads input data from a file and compresses it using Huffman coding. Utilizes the Huffman code table to encode symbols and writes the compressed data to an output buffer using bit-level I/O.
- Decompression (dehuff_decompress_file): Reads compressed data from a file and decompresses it using the Huffman tree. Uses the Huffman tree to decode Huffman codes and reconstruct the original data, writing it to an output file.
- Histogram Calculation (fill_histogram): Reads input data from a file and calculates the frequency of each symbol. Constructs a histogram representing the frequency distribution of symbols in the input data. These main data structures and algorithms provide a clear organization of the program's functionality, allowing for easier maintenance and understanding of the codebase.

Pseudocode

for bitreader.c

```
#include "bitreader.h"

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

{insert following functions in function descriptions}
```

for bitwriter.c

```
#include "bitwriter.h"

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

{insert following functions in function descriptions}
```

for node.c

```
#include "node.h"

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

{insert following functions in function descriptions}
```

for pq.c

```
#include "pq.h"

typedef struct ListElement ListElement;

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

struct PriorityQueue {
       ListElement *list;
};

{insert following functions in function descriptions}
```

for huff.c

```
#include <stdio.h>
#include <stdint.h>
#include <stdbool.h>

#include "node.h"
#include "pq.h"
#include "bitwriter.h"

void fill_histogram(FILE *fin, uint32_t *histogram) {
    int c;
    while ((c = fgetc(fin)) != EOF) {
        histogram[c]++;
    }
}

Node *create_tree(uint32_t *histogram, uint16_t *num_leaves) {
    PriorityQueue *pq = pq_create();
    *num_leaves = 0;

    for (int i = 0; i < SYMBOL_COUNT; i++) {
        if (histogram[i] > 0) {
            *num_leaves += 1;
            Node *leaf = node_create((uint8_t)i, histogram[i]);
            enqueue(pq, leaf);
        }
    }

    while (!pq_size_is_1(pq)) {
        Node *left = dequeue(pq);
        Node *right = dequeue(pq);
        Node *combined = node_create(0, left->weight + right->weight);
        combined->left = left;
        combined->right = right;
        enqueue(pq, combined);
    }

    Node *root = dequeue(pq);
    pq_free(&pq);
    return root;
}

void fill_code_table(Code *code_table, Node *node, uint64_t code, uint8_t code_length) {
    if (node->left == NULL && node->right == NULL) {
        code_table[node->symbol].code = code;
        code_table[node->symbol].code_length = code_length;
    } else {
        fill_code_table(code_table, node->left, code << 1, code_length + 1);
        fill_code_table(code_table, node->right, (code << 1) | 1, code_length + 1);
    }
}

void huff_compress_file(const char *out_filename, const char *in_filename) {
    FILE *fin = fopen(in_filename, "rb");
    FILE *fout = fopen(out_filename, "wb");

    if (fin == NULL || fout == NULL) {
        fprintf(stderr, "Error: Unable to open input or output file\n");
```

```
        return;
    }

    uint32_t histogram[SYMBOL_COUNT] = {0};
    uint16_t num_leaves = 0;
    fill_histogram(fin, histogram);
    Node *root = create_tree(histogram, &num_leaves);
    Code code_table[SYMBOL_COUNT];
    fill_code_table(code_table, root, 0, 0);

    for (int i = 0; i < SYMBOL_COUNT; i++) {
        bit_write_uint32(fout, histogram[i]);
    }

    fseek(fin, 0, SEEK_SET);
    BitWriter *bit_writer = bit_write_open(fout);
    compress_file(bit_writer, fin, num_leaves, code_table);
    bit_write_close(&bit_writer);
    node_free(&root);
    fclose(fin);
    fclose(fout);
}

int main(int argc, char *argv[]) {
    if (argc != 3) {
        fprintf(stderr, "Usage: %s <input_file> <output_file>\n", argv[0]);
        return 1;
    }

    huff_compress_file(argv[2], argv[1]);
    return 0;
}
```

for dehuff.c

```
#include <stdio.h>
#include <stdint.h>
#include <stdbool.h>

#include "node.h"
#include "bitreader.h"


void huff_decompress_file(const char *in_filename, const char *out_filename) {
    FILE *fin = fopen(in_filename, "rb");
    FILE *fout = fopen(out_filename, "wb");

    if (fin == NULL || fout == NULL) {
        fprintf(stderr, "Error: Unable to open input or output file\n");
        return;
    }

    uint32_t histogram[SYMBOL_COUNT];
    Code code_table[SYMBOL_COUNT];

    for (int i = 0; i < SYMBOL_COUNT; i++) {
        histogram[i] = bit_read_uint32(fin);
    }

    Node *root = create_tree(histogram);
```

```
    fill_code_table(code_table, root, 0, 0);
    BitReader *bit_reader = bit_read_open(fin);
    huff_decompress(bit_reader, fout, root);
    bit_read_close(&bit_reader);
    node_free(&root);
    fclose(fin);
    fclose(fout);
}

int main(int argc, char *argv[]) {
    if (argc != 3) {
        fprintf(stderr, "Usage: %s <input_file> <output_file>\n", argv[0]);
        return 1;
    }
    huff_decompress_file(argv[1], argv[2]);
    return 0;
}
```

Function Descriptions

For function BitWriter *bit_write_open(const char *filename):

- Input: `filename`
- Output: `NULL` or `buf`
- Purpose: Opens binary or text file. Returns NULL if any of them fail
- Pseudocode:

```
def bit_write_open(filename):
    allocate a new BitWriter
    open the filename for writing as a binary file, storing the result in FILE *f
    clear the byte and bit_positions fields of the BitWriter to 0
    if any step above causes an error:
        return NULL
    else:
        return a pointer to the new BitWriter
```

For function void bit_write_close(BitWriter **pbuf):

- Input: `pbuf`
- Output: None
- Purpose: Uses values from the pointer, flushes out data in the byte, then closes the file, and then sets the pointer to NULL
- Pseudocode:

```
def bit_write_close(BitWriter **pbuf):
    if (*pbuf)->bit_position > 0:
        /* (*pbuf)->byte contains at least one bit that has not yet been written */
        write the byte to the underlying_stream using fputc()
    close the underlying_stream
    free the BitWriter
    *pbuf = NULL
```

For function void bit_write_bit(BitWriter *buf, uint8_t bit):

- Input: `buf`
- Output: Writes the provided bit
- Purpose: Writes a single bit. Collects 8 bits into the buffer before using fputc()
- Pseudocode:

```
def bit_write_bit(buf, bit):
    if bit_position > 7:
        write the byte to the underlying_stream using fputc()
        clear the byte and bit_position fields of the BitWriter to 0
    set the bit at bit_position of the byte to the value of bit
    bit_position += 1
```

For function void bit_write_uint8(BitWriter *buf, uint8_t byte):

- Input: `buf`
- Output: Writes the 8 bits of the provided byte
- Purpose: Writes the 8 bits by calling the function 8 times
- Pseudocode:

```
def bit_write_uint8(buf, x):
    for i = 0 to 7:
        write bit i of x using bit_write_bit()
```

For function void bit_write_uint16(BitWriter *buf, uint16_t x):

- Input: `buf`
- Output: Writes the 16 bits of the provided byte
- Purpose: Writes the 16 bits by calling the function 16 times
- Pseudocode:

```
def bit_write_uint16(buf, x):
    for i = 0 to 15:
        write bit i of x using bit_write_bit()
```

For function void bit_write_uint32(BitWriter *buf, uint32_t x):

- Input: `buf`
- Output: Writes the 32 bits of the provided byte
- Purpose: Writes the 32 bits by calling the function 32 times
- Pseudocode:

```
def bit_write_uint32(buf, x):
    for i = 0 to 31:
        write bit i of x using bit_write_bit()
```

For function BitReader *bit_read_open(const char *filename):

- Input: `filename`
- Output: Returns a pointer or `NULL`
- Purpose: Opens binary file name and returns a pointer
- Pseudocode:

```
def bit_read_open(filename):
    allocate a new BitReader
    open the filename for reading as a binary file, storing the result in FILE *f
    store f in the BitReader field underlying_stream
    clear the byte feild of the BitReader to 0
    set the bit_position field of the BitReader to 8
    if any step above causes an error:
        return NULL
    else:
        return a pointer to the new BitReader
```

For function void bit_read_close(BitReader **pbuf):

- Input: `pbuf`
- Output: The structure pointer by "pbuf" is closed and memory is de-allocated
- Purpose: Uses values from the pointer, frees the object, then sets the pointer to NULL
- Pseudocode:

```
def bit_read_close(BitReader **pbuf):
    if *pbuf != NULL:
        close the underlying_stream
        free *pbuf
        *pbuf = NULL
        if any step above causes an error:
            report fatal error
```

For function uint8_t bit_read_bit(BitReader *buf):

- Input: `buf`
- Output: Returns the next bit read
- Purpose: Main reading function. Reads a single bit pointed by `buf`
- Pseudocode:

```
def bit_read_bit(buf):
    if bit_position > 7:
        read a byte from the underlying_stream us9ing fgetc()
        bit_position = 0
    get the bit numbered bit_position from byte
    bit_position += 1;
    if any step above causes error:
        report fatal error
    else:
        return the bit
```

For function uint8_t bit_read_uint8(BitReader *buf):

- Input: `buf`
- Output: Returns the unsigned integer read from the file associated with `buf`
- Purpose: Reads 8 bits of `buf` by calling the function 8 times
- Pseudocode:

```
def bit_read_uint8(buf):
    uint8_t byte = 0x00
    for i in range(0, 8):
        read a bit b from the underlying_stream
        set bit i of byte to the value of b
    return byte
```

For function uint16_t bit_read_uint16(BitReader *buf):

- Input: `buf`
- Output: Returns the unsigned integer read from the file associated with `buf`
- Purpose: Reads 16 bits of `buf` by calling the function 16 times
- Pseudocode:

```
def bit_read_uint16(buf):
    uint8_t byte = 0x0000
    for i in range(0, 16):
        read a bit b from the underlying_stream
        set bit i of byte to the value of b
    return byte
```

For function uint32_t bit_read_uint32(BitReader *buf):

- Input: `buf`
- Output: Returns the unsigned integer read from the file associated with `buf`
- Purpose: Reads 32 bits of `buf` by calling the function 32 times
- Pseudocode:

```
def bit_read_uint8(buf):
    uint8_t byte = 0x00000000
    for i in range(0, 32):
        read a bit b from the underlying_stream
        set bit i of byte to the value of b
    return byte
```

For function Node *node_create(uint8_t symbol, uint32_t weight):

- Input: `symbol` and `weight`
- Output: Returns the pointer to the new structure
- Purpose: Creates a `Node` and sets the inputs and returns a pointer to the new node
- Pseudocode:

```
def node_create(symbol, weight):
    allocate a new Node
    set the symbol and weight fields of Node to function parameters symbol and weight
    if any step above causes an error:
        return NULL
    else:
        return a pointer to the new Node
```

For function void node_free(Node **node):

- Input: `node`
- Output: None
- Purpose: Frees the children of `*pnode`, free `*pnode` and sets `*pnode` to `NULL`
- Pseudocode:

```
def node_free(Node **pnode):
    if *pnode != NULL:
        node_free(&(*pnode)->left)
        node_free(&(*pnode)->right)
        free(*pnode)
        *pnode = NULL
```

For function void node_print_tree(Node *tree):

- Input: `tree`
- Output: Prints the tree
- Purpose: Prints the tree for debugging
- Pseudocode:

```
node_print_tree(tree):
    1. If the provided tree pointer is NULL, return.
    2. Print the symbol and weight of the current node.
    3. Print the code and code length of the current node.
    4. Print the memory addresses of the left and right children.
    5. Recursively print the left subtree.
    6. Recursively print the right subtree.
```

For function PriorityQueue *pq_create(void):

- Input: None
- Output: A pointer to a new `PriorityQueue` structure or `NULL` if fails
- Purpose: Allocates a `PriorityQueue` object and returns a pointer to it
- Pseudocode:

```
def PriorityQueue *pq_create(void) {
    PriorityQueue *q = (PriorityQueue *)malloc(sizeof(PriorityQueue));
    if (q == NULL) {
        return NULL;
    }
    q->front = NULL;
    q->rear = NULL;
    q->size = 0;
    return q;
}
```

For function void pq_free(PriorityQueue **q):

- Input: A pointer to a pointer to `PriorityQueue` structure
- Output: None
- Purpose: Call `free()` on `*q` and then sets it to `NULL`
- Pseudocode:

```
def pq_free(PriorityQueue **q) {
    if (q == NULL || *q == NULL) {
        return;
    }
    PQNode *current = (*q)->front;
    while (current != NULL) {
        PQNode *next = current->next;
        node_free(&(current->tree));
        free(current);
        current = next;
    }
    // 4. Free the PriorityQueue structure (*q).
    free(*q);
    // 5. Set *q to NULL.
    *q = NULL;
}
```

For function bool pq_is_empty(PriorityQueue *q):

- Input: A pointer to `PriorityQueue` structure
- Output: `true` if queue is empty, `false` otherwise
- Purpose: Indicates an empty queue
- Pseudocode:

```
def pq_is_empty(PriorityQueue *q) {
    if (q == NULL || q->front == NULL) {
        return true;
    }
    return false;
}
```

For function bool pq_size_is_1(PriorityQueue *q):

- Input: A pointer to `PriorityQueue` structure
- Output: `true` if queue has only one element, `false` otherwise
- Purpose: Indicates if the `PriorityQueue` has a single element
- Pseudocode:

```
def pq_size_is_1(PriorityQueue *q) {
    if (q == NULL || q->front == NULL || q->front != q->rear) {
        return false;
    }
    return true;
}
```

For function void enqueue(PriorityQueue *q, Node *tree):

- Input: A pointer to `PriorityQueue` structure and a pointer to `Node` structure
- Output: None
- Purpose: Inserts a tree into the `PriorityQueue`
- Pseudocode:

```
def enqueue(PriorityQueue *q, Node *tree) {
    PQNode *newNode = (PQNode *)malloc(sizeof(PQNode));
    if (newNode == NULL) {
        return;
    }
    newNode->tree = tree;
    newNode->next = NULL;

    if (q->front == NULL) {
        // a. Set q->front and q->rear to newNode.
        q->front = newNode;
        q->rear = newNode;
    } else {
        q->rear->next = newNode;
        q->rear = newNode;
    }
    q->size++;
}
```

For function Node *dequeue(PriorityQueue *q):

- Input: A pointer to `PriorityQueue` structure
- Output: A pointer to dequeued `Node` structure,or `NULL` if queue is empty
- Purpose: Removes the queue element with the lowest weight and returns it
- Pseudocode:

```
def Node *dequeue(PriorityQueue *q) {
    if (q->front == NULL) {
        return NULL;
    }
    PQNode *temp = q->front;
    Node *tree = temp->tree;
    q->front = temp->next;
    free(temp);
    q->size--;
    return tree;
}
```

For function void pq_print(PriorityQueue *q):

- Input: A pointer to `PriorityQueue` structure
- Output: None
- Purpose: Prints the `PriorityQueue`
- Pseudocode:

```
def pq_print(PriorityQueue *q) {
    // 1. If the queue is empty (q->front is NULL), print "Priority Queue is empty" and return
         .
    if (q->front == NULL) {
        printf("Priority Queue is empty\n");
        return;
    }
    // 2. Create a temporary pointer, current, and initialize it with q->front.
    PQNode *current = q->front;
    // 3. Print "Priority Queue (size %d): " followed by the queue size, q->size.
    printf("Priority Queue (size %d): ", q->size);
    // 4. Iterate through the linked list:
    while (current != NULL) {
        // a. Print the weight of the tree associated with the current node.
        printf("%d ", current->tree->weight);
        // b. Move to the next node.
        current = current->next;
    }
    // 5. Print a newline character.
    printf("\n");
}
END
```

For function uint32_t fill_histogram(FILE *fin, uint32_t *histogram):

- Input: `FILE *fin` and `uint32_t *histogram`
- Output: `uint32_t`
- Purpose: Updates an array of `uint32_t` with the number of each unique byte
- Pseudocode:

```
1. Initialize an array to store the histogram of symbol frequencies. Each element should be
    initialized to 0.
2. Initialize a variable to keep track of the total number of symbols processed and set it to
    0.
3. Read characters from the input file stream until end-of-file (EOF) is reached:
   3.1 Read a character from the input file.
   3.2 Increment the corresponding element in the histogram array based on the character read.
   3.3 Increment the total number of symbols processed.
4. Return the total number of symbols processed.
```

For function Node *create_tree(unt32_t *histogram, uint16_t *num_leaves):

- Input: `histogram` and `num_leaves`
- Output: Returns a pointer to the root Node of the constructed Huffman tree
- Purpose: Creates and fills a priority queue, then runs the huffman code, then dequeues the queue's only entry and returns it
- Pseudocode:

```
def Node *create_tree(uint32_t *histogram, uint16_t *num_leaves) {
    PriorityQueue *pq = pq_create();
    *num_leaves = 0;

    for (uint8_t symbol = 0; symbol < 256; symbol++) {
        if (histogram[symbol] > 0) {
            Node *leaf = node_create(symbol, histogram[symbol]);
            enqueue(pq, leaf);
            (*num_leaves)++;
        }
    }
    while (!pq_size_is_1(pq)) {
        Node *left = dequeue(pq);
        Node *right = dequeue(pq);
        Node *internal_node = node_create_internal(left, right);
        enqueue(pq, internal_node);
    }
    Node *root = dequeue(pq);
    pq_free(&pq);
    return root;
}
```

For function fill_code_table(Code *code_table, Node *node, uint64_t code, uint8_t code_length):

- Input: `code_table` and `node` and `code` and `code_length`
- Output: None
- Purpose: To traverse the tree and fill in the table for each node symbol.
- Pseudocode:

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

if node is internal:
    /* Recursive calls left and right. */

    /* append a 0 to code and recurse *
    /* (don't need to append a 0; it's already there)
    fill_code_table(code_table, node->left, code, code_length + 1);

    /* append a 1 to code and recurse */
    code |= (uint64_t) 1 << code_length;
    fill_code_table(code_table, node->right, code, code_length + 1)
else:
    /* Lead node: store the Huffman Code */
    code_table[node->symbol].code = code;
    code_table[node->symbol].code_length = code_length;
```

For function void huff_compress_file(outbuf, fin, filesize, num_leaves, code_tree, code_table):

- Input: `code_table` and `code_tree` and `num_leaves` and `fin` and `filesize` and `outbuf`
- Output: None
- Purpose: To produce a compressed version of the input file using Huffman coding
- Pseudocode:

```
def huff_compress_file(outbuf, fin, filesize, num_leaves, code_tree, code_table)
    write uint8_t 'H' to outbuf
    write uint8_t 'C' to outbuf
    write uint32_t filesize to outbuf
    write uint16_t num_leaves to outbuf
    huff_write_tree(outbuf, code_tree)
    while true:
        b = fgetc(fin)
        if b == EOF:
            break
        code = code_table[b].code
        code_length = code_table[b].code_length
        for i in range(0, code_length):
            write bit (code & 1) to outbuf
            code >>= 1
```

For function void dehuff_decompress_file(FILE *fout, BitReader *inbuf):

- Input: `FILE *fout` and `BitReader *inbuf`
- Output: None
- Purpose: Decompresses the data read from the `inbuf` and writes the decompressed data to the output `FILE *fout`
- Pseudocode:

```
def dehuff_decompress_file(fout, inbuf):
    read uint8_t type1 from inbuf
    read uint8_t type2 from inbuf
    read uint32_t filesize from inbuf
    read uint16_t num_leaves from inbuf
    assert(type1 == 'H')
    assert(type2 == 'C')
    num_nodes = 2 * num_leaves - 1
    Node *node
    for i in range(0, num_nodes):
        read one bit from inbuf
        if bit == 1:
            read uint8_t symbol from inbuf
            node = node_create(symbol, 0)
        else:
            node = node_create(0, 0)
            node->right = stack_pop()
            node->left = stack_pop()
        stack_push(node)
    Node *code_tree = stack_pop()
    for i in range(0, filesize):
        node = code_tree
        while true:
            read one bit from inbuf
            if bit == 0:
                node = node->left
            else:
                node = node->right
            if node if a leaf:
                break
        write uint8 node->symbol to fout
```