Assignment 7
Huffman Coding

by Dr. Kerry Veenstra
edits by Jess Srinivas
CSE 13S, Winter 2024
Document version 1 (changes in Section 14)

Design Draft due **Wednesday, March 6th**, 2024
Assignment due **Friday, March 8th**,2024

1 Introduction

Jessie's partner, Summer has recently found themselves in a pickle. Due to the large influx of Tank pictures from Alissa, their computer's storage is running low. After looking around at what files they have on their computer, they realized that they had multiple large books on their computer. They also had a lot of images of their dog, Toby. Unfortunately, they need to keep the books around, but want more space on their computer for pictures of Tank as well. Summer would like you to help them compress many of the things that were on their computer. Your task will be to write two programs that can losslessly compress and decompress any files given to them.

One can use Huffman Coding to compress a data file (introduction[1], original paper[2]). The key idea is to determine which bytes ("symbols") of the input file are most common and switch their representations to use fewer bits. To compensate, the less common symbols will switch to representations that uses more bits. The overall result is usually that fewer total bits are needed to represent the entire file, meaning that the file has been compressed.

In this assignment, you will create two programs. The first is a data compressor, `huff`, which computes the Huffman code of an input file.

```
huff -i infile -o outfile
```

This data compressor:

- reads a stream of **bytes** from a binary input file using functions from `stdio.h`: `fopen()`, `fgetc()`, `fseek()`, and `fclose()`.
- writes a stream of **bits** to a binary output file using functions from `bitwriter.c`, which you will write.

The second program is a data decompressor, `dehuff`, which converts a Huffman Coded input file back into its original form.

```
dehuff -i infile -o outfile
```

This data decompressor:

- reads a stream of **bits** from its input file using functions from `bitreader.c`, which you will write.
- writes a stream of **bytes** to its output file using functions from `stdio.h`: `fopen()`, `fputc()`, and `fclose()`.

In addition to `bitreader.c` and `bitwriter.c`, you will be creating two other support modules: `node.c` for a binary tree and `pq.c` for a priority queue. Here is a checklist of the source code that you will be writing.

- ☐ A "bit writer" abstract data type in `bitwriter.c` (see Section 2 and `bitwriter.h`). We have provided you with unit tests for this module in `bwtest.c`.
- ☐ A "bit reader" abstract data type in `bitreader.c` (see Section 3 and `bitreader.h`). We have provided you with unit tests for this module in `brtest.c`.
- ☐ A binary tree abstract data type in `node.c` (see Section 4 and `node.h`). We have provided you with unit tests for this module in `nodetest.c`.
- ☐ A priority queue abstract data type in `pq.c` (see Section 5 and `pq.h`). We have provided you with unit tests for this module in `pqtest.c`.
- ☐ A Huffman Coding data compressor (see Section 6). We have provided you with system tests in `runtests.sh`.
- ☐ A Huffman Coding data decompressor (see Section 7). We have provided you with system tests in `runtests.sh`.

Your Makefile will build the four unit-test programs (`bwtest`, `brtest`, `nodetest`, `pqtest`) and the data compressor/decompressor programs (`huff`, and `dehuff`).

2 Bit Writer

Previously you have used the `fopen()`, `fclose()`, and `fputc()` functions to write a stream of *bytes* to a binary file. This approach works well when a binary file's format is defined as a sequence of bytes. However in this assignment, the `.huff` file format is defined as a sequence of *bits.* So to make creating a `.huff` file straightforward, you will write a set of *bit*-writing functions.

2.1 BitWriter Functions

These functions write a binary file one bit at a time. So in your `huff.c` program, you do *not* call the `fputc()` function directly. Instead you use these functions, which you will write:

- `bit_write_open()` calls `fopen()`
- `bit_write_close()` calls `fclose()`
- `bit_write_bit()` calls `fputc()` after it collects 8 bits

The functions just mentioned manage a byte buffer.

- `bit_write_open()` creates the buffer.
- `bit_write_close()` flushes the buffer and frees it.
- `bit_write_bit()` collects bits into the buffer and writes the full buffer by calling `fputc()`.

It is important to understand that when your `huff.c` program needs to write 8-bit, 16-bit, and 32-bit data values, do not call the function `fputc()` directly. This is because these values almost always will *not* be aligned on byte boundaries within the binary file. Instead, call these functions:

- `bit_write_uint8()` calls `bit_write_bit()` 8 times
- `bit_write_uint16()` calls `bit_write_bit()` 16 times
- `bit_write_uint32()` calls `bit_write_bit()` 32 times

So, essentially, all of the bit-writing functions send their data through `bit_write_bit()`.

The functions in this section use these data types.

```
1  /* bitwriter.h */
2  typedef struct BitWriter BitWriter;
```

```
/* bitwriter.c */
#include "bitwriter.h"
struct BitWriter {
    FILE *underlying_stream;
    uint8_t byte;
    uint8_t bit_position;   /*
};
```

Function descriptions are below.

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

Open **binary or text file**, `filename` for write using `fopen()` and return a pointer to a newly aloocated `BitWriter` struct. You must check all function return values and return NULL if any of them report a failure. The pseudocode is below.

```
def bit_write_open(filename):
    allocate a new BitWriter
    open the filename for writing as a binary file, storing the result in FILE *f
    store f in the BitWriter field underlying_stream
    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
```

`void bit_write_close(BitWriter **pbuf);`

Using values in the `BitWriter` pointed to by `*pbuf`, flush any data in the `byte` buffer, close `underlying_stream`, free the `BitWriter` object, and set the `*pbuf` pointer to `NULL`. You must check all function return values and report a fatal error if any of them report a failure.

```
def bit_write_close(BitWriter **pbuf):
    if *pbuf != NULL:
        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
```

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

This is the main writing function. It writes a single bit, `bit`, using values in the `BitWriter` pointed to by `buf`. This function collects 8 bits into the buffer `byte` before writing it using `fputc()`. You must check all function return values and report a fatal error if any of them report a failure.

```
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
```

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

Write the 8 bits of function parameter `x` by calling `bit_write_bit()` 8 times. Start with the LSB (least-significant, or rightmost, bit) of `x`.

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

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

Write the 16 bits of function parameter `x` by calling `bit_write_bit()` 16 times. Start with the LSB (least-significant, or rightmost, bit) of `x`.

```
def bit_write_unit16(buf, x):
    for i = 0 to 15:
        write bit i of x using bit_write_bit()
```

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

Write the 32 bits of function parameter `x` by calling `bit_write_bit()` 32 times. Start with the LSB (least-significant, or rightmost, bit) of `x`.

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

3 Bit Reader

Previously you have used the `fopen()`, `fclose()`, and `fgetc()` functions to read a stream of *bytes* from a binary file. This approach works when the binary file is defined as a sequence of bytes. However in this assignment, the `.huff` file format is defined as a sequence of *bits*. So to make reading a `.huff` file straightforward, you will write a set of *bit*-reading functions.

3.1 BitReader Functions

These functions read a binary file one bit at a time. So in your `dehuff.c` program, do *not* call the `fgetc()` function directly. Instead you use these functions, which you will write:

- `bit_read_open()` calls `fopen()`
- `bit_read_close()` calls `fclose()`

- `bit_read_bit()` calls `fgetc()`

The functions just mentioned manage a byte buffer.

- `bit_read_open()` creates the buffer.
- `bit_read_close()` frees the buffer.
- `bit_read_bit()` reads bytes from the input file into the buffer using `fgetc()` but returns them only one bit at a time.

It it important to understand that when your `dehuff.c` program needs to read 8-bit, 16-bit, and 32-bit data values, do not call the function `fgetc()` directly. This is because these values almost always will *not* be aligned on byte boundaries within the binary file. Instead, call these functions:

- `bit_read_uint8()` calls `bit_read_bit()` 8 times
- `bit_read_uint16()` calls `bit_read_bit()` 16 times
- `bit_read_uint32()` calls `bit_read_bit()` 32 times

So, essentially, all of the bit-reading functions get their data from `bit_read_bit()`.

The functions in this section use these data types:

```
1    /* bitreader.h */
2    typedef struct BitReader BitReader;
```

```
1    /* bitreader.c */
2    #include "bitreader.h"
3    struct BitReader {
4        FILE *underlying_stream;
5        uint8_t byte;
6        uint8_t bit_position;
7    };
```

Function descriptions are below.

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

Open binary `filename` using `fopen()` and return a pointer to a `BitReader`. On error, return `NULL`. The pseudocode is below. Notice in line 6 that the `byte` field is being set to an unexpected value of 8 rather than 0. This value forces `bit_read_bit()` to read the first byte of the file when it is first called.

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

`void bit_read_close(BitReader **pbuf);`

Using values in the `BitReader` pointed to by `*pbuf`, close `(*pbuf)->underlying_stream`, free the `BitReader` object, and set the `*pbuf` pointer to `NULL`. You must check all function return values and report a fatal error if any of them report a failure.

```
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
```

`uint8_t bit_read_bit(BitReader *buf);`

This is the main reading function. It reads a single bit using values in the `BitReader` pointed to by `buf`.

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

`uint8_t bit_read_uint8(BitReader *buf);`

Read 8 bits from `buf` by calling `bit_read_bit()` 8 times. Collect these bits into a `uint8_t` starting with the LSB (least-significant, or rightmost, bit).

```
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
```

`uint16_t bit_read_uint16(BitReader *buf);`

Read 16 bits from `buf` by calling `bit_read_bit()` 16 times. Collect these bits into a `uint16_t` starting with the LSB (least-significant, or rightmost, bit).

```
def bit_read_uint16(buf):
    uint16_t word = 0x0000
    for i in range(0, 16):
        read a bit b from the underlying_stream
        set bit i of word to the value of b
    return word;
```

`uint32_t bit_read_uint32(BitReader *buf);`

Read 32 bits from `buf` by calling `bit_read_bit()` 32 times. Collect these bits into a `uint32_t` starting with the LSB (least-significant, or rightmost, bit).

```
def bit_read_uint32(buf):
    uint32_t word = 0x00000000
    for i in range(0, 32):
        read a bit b from the underlying_stream
        set bit i of word to the value of b
    return word;
```

4 Node

Use `Node`s to make a binary tree. Each `Node` contains numerous fields that the Huffman Coding algorithm will use. This is a fairly simple module: you will implement functions to create and free nodes and to print trees. Your code can access the nodes' fields directly using the C element-selection-through-pointer (`->`) operator.

```
    /* node.h */
    typedef struct Node Node;

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

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

Create a `Node` and set its `symbol` and `weight` fields. Return a pointer to the new node. On error, return NULL.

```
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
```

`void node_free(Node **pnode);`

Free the children of `*pnode`, free `*pnode`, and set `*pnode` to `NULL`.

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

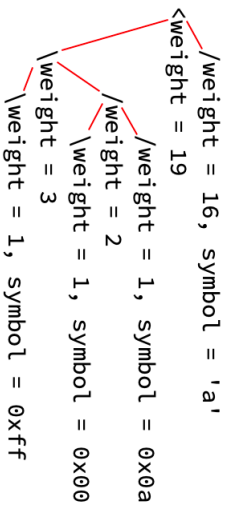
`void node_print_tree(Node *tree);`

This function is for diagnostics and debugging. You can print the tree in any way that you want. You may use the recursive tree-printing routine below or write your own function. The provided function prints on-screen a sideways view of the binary tree using text characters (Fig 1a).

View the tree by rotating the printed image 90° to the right (or rotate your head 90° to the left). Imagine lines connecting the nodes (shown as red annotations in Fig. 1b). As you can see, the `<` character indicates the root of the tree.

```
   /weight = 16, symbol = 'a'
<weight = 19
         /weight = 1, symbol = 0x0a
      /weight = 2
         \weight = 1, symbol = 0x00
   \weight = 3
      \weight = 1, symbol = 0xff
```

(a) Binary tree as printed.



(b) Binary tree rotated and with manual annotations to show the relationships between the nodes.

Figure 1: How to interpret the output of `node_print_tree()`.

```
void node_print_node(Node *tree, char ch, int indentation) {
    if (tree == NULL)
        return;
    node_print_node(tree->right, '/', indentation + 3);
    printf("%*cweight = %d", indentation + 1, ch, tree->weight);

    if (tree->left == NULL && tree->right == NULL) {
        if (' ' <= tree->symbol && tree->symbol <= '~') {
            printf(", symbol = '%c'", tree->symbol);
        } else {
            printf(", symbol = 0x%02x", tree->symbol);
        }
    }

    printf("\n");
    node_print_node(tree->left, '\\', indentation + 3);
}
```

```
void node_print_tree(Node *tree) {
    node_print_node(tree, '<', 2);
}
```

As a reminder about a `printf()` feature, notice that the format string of the first `printf()` has a `*` character where a numeric width value ought to be, as in `%*c` instead of `%2c`. The `*` indicates that the field's width is given by an integer parameter that follows the format string. (The field width is *not* the number of space characters that will be printed. It's the total number of characters, including the `ch`.) In this case, given `%*c`, the `*` means that after the format string is an integer width (`indentation + 1`) followed by a character (`ch`). We use this feature to programmatically control the indentation of `ch`.

5 Priority Queue

You will write a Priority Queue abstract data type. The Priority Queue will store pointers to trees. Since a Priority Queue orders its entries based on *priorities* (or weights), each of the queue entries needs to have a weight. Since each queue entry has a pointer to a tree, and each tree node has a weight, you can use the `weight` field of a tree's root node as the value of the queue entry that points to it.

You will implement the Priority Queue using a linked list. For a reason explained below, you will use two structs. One of the structs (`ListElement`) makes the linked list (that's why the struct has a `next` field). The other struct (`PriorityQueue`) represents the queue itself. We represent the queue using a second struct that points to a separate linked list with a `list` field. That way, we always can have a pointer to the queue, even when an empty queue is represented by a `NULL list` value.

```
/* pq.h */
typedef struct PriorityQueue PriorityQueue;
```

```c
/* pq.c */
typedef struct ListElement ListElement;

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

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

`PriorityQueue *pq_create(void);`

Allocate a `PriorityQueue` object and return a pointer to it. If there's an error, return NULL.

`void pq_free(PriorityQueue **q);`

Call `free()` on `*q`, and then set `*q = NULL`.

`bool pq_is_empty(PriorityQueue *q);`

We indicate an empty queue by storing `NULL` in the queue's `list` field. Return `true` if that's the case.

`bool pq_size_is_1(PriorityQueue *q);`

If the Priority Queue contains a single element, then return true. Otherwise return false.

`bool pq_less_than(ListElement *e1, ListElement *e2)`

The `pq_less_than()` function compares the `tree->weight` values of two `ListElement` objects, returning `true` if the weight of the first element is less than the weight of the second element. If the weights of the elements are equal, then compare their `tree->symbol` values, and return `true` if the symbol of the first element is less than the symbol of the second element.

This function is not used outside of `pq.c`, and so it is not declared in `pq.h`.

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

Insert a tree into the priority queue. Keep the tree with the lowest weight at the head (that is, next to be dequeued). There are three possibilities to consider:

- The queue currently is empty.
- The new element will become the new first element of the queue.
- The new element will be placed after an existing element.

`Node *dequeue(PriorityQueue *q);`

Remove the queue element with the lowest weight and return it. If the queue is empty, then report a fatal error.

`void pq_print(PriorityQueue *q);`

Here's a diagnostic function. It prints the trees of the queue `q`.

```
1   void pq_print(PriorityQueue *q) {
2       assert(q != NULL);
3       ListElement *e = q->list;
4       int position = 1;
5       while (e != NULL) {
6           if (position++ == 1) {
7               printf("================================================\n");
8           } else {
9               printf("------------------------------------------------\n");
10          }
11          node_print_tree(e->tree, '<', 2);
12          e = e->next;
13      }
14      printf("================================================\n");
15  }
```

6 Huffman Coding

Huffman Coding consists of five steps.

1. Read the file, and count the frequency of each symbol. Use that to create a histogram of the input file's bytes/symbols. See `fill_histogram()`.
2. Create a code tree from the histogram. See `create_tree()`.
3. Fill a 256-entry code table, one entry for each byte value. See `fill_code_table()`.
4. Rewind the input file using `fseek()` in preparation for the next step.
5. Create a Huffman Coded output file from the input file. See `huff_compress_file()`.

6.1 Functions

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

This function updates a histogram array of `uint32_t` values with the number of each of the unique byte values of the input file. It also returns the total size of the input file.

Parameter `fin` provides access to the input file using `fgetc()`. Parameter `histogram` points to a 256-element array of `uint32_t` values. Clear all elements of this array, and then read bytes from `fin` using `fgetc()`. For each byte read, increment the proper element of the histogram: `++histogram[byte]`. The return value of the function is the total size of the file. Determine this value by declaring a local variable `uint32_t filesize` and incrementing it for every byte read with `++filesize`.

Important Hack. To vastly simplify the rest of your `huff` program as well as the corresponding `dehuff` program, ensure that at least two values of the `histogram` array are non-zero. This hack forces the code tree that is generated later to have at least two leaves. (Dealing with an empty code tree or a single-node code tree is complicated. It's best to avoid needing to do this.) So put this code somewhere in your function after clearing the `histogram` array. You can increment any two, different bins of the histogram, but choosing `0x00` and `0xff` will match the choice made in the Huffman Code Tree visualizer.

```
++histogram[0x00];
++histogram[0xff];
```

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

This function creates and returns a pointer to a new Huffman Tree. It also returns the number of leaf nodes in the tree. (Here's a Huffman Tree visualization that will show you what the algorithm is doing: 190n.github.io/huffman-visualization.)

Here's how to create the tree:

1. **Create and fill a Priority Queue.**

   ```
   Go through the histogram, and create a node for every non-zero histogram entry.
   Write the number of nodes added to the Priority Queue to *num_leaves.
   Initialize each node with the symbol and weight of its histogram entry.
   Put each node in the priority queue.
   ```

2. **Run the Huffman Coding algorithm.**

   ```
   while Priority Queue has more than one entry
        Dequeue into left
        Dequeue into right
        Create a new node with symbol = 0 and weight = left->weight + right->weight
        node->left = left
        node->right = right
        Enqueue the new node
   ```

3. **Dequeue the queue's only entry and return it.**

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

This is a recursive function that traverses the tree and fills in the Code Table for each leaf node's symbol. Call it as `fill_code_table(code_table, code_tree, 0, 0)`.

The parameter `code_table` is a pointer to an array of 256 `Code` objects, each of which looks like this:

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

The `code` and `code_length` parameters give the Huffman Code for this node of the tree. The code starts empty (`code == 0` and `code_length == 0`), and then gets a bit added to it for every recursive call. When recursing to the left child, add a 0 (which means just passing `code_length + 1` in the recursive function call). When recursing to the right child, you need to set bit `code_length` of `code` before passing `code_length + 1` in the recursive function call.

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

    /* append a 0 to code and recurse *
    /* (don't neeed 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:
    /* Leaf node:  store the Huffman Code. */
    code_table[node->symbol].code = code;
    code_table[node->symbol].code_length = code_length;
```

`void huff_compress_file(outbuf, fin, filesize, num_leaves, code_tree, code_table)`

Write a Huffman Coded file. The parameters of the function are

- `BitWriter *outbuf` — Use this parameter with calls to `bit_write_bit()`, `bit_write_uint8()`, `bit_write_uint16()`, and `bit_write_uint32()` to write the output file.
- `FILE *fin` — Use this parameter with calls to `fgetc()` to read the input file. **Note: the code assumes that `fin` has been rewound using `fseek()` after the call to `fill_histogram()`. Remember that `fill_histogram()` already has read the entire input file, and so the `FILE *fin` needs to be rewound before this function can re-read the file.**
- `uint32_t filesize` — The size of the file, as returned by the call to `fill_histogram()`.
- `uint16_t num_leaves` — The number of leaves of the Code Tree, as returned by `create_tree()`.
- `Node *code_tree` – A pointer to the Code Tree, as returned by `create_tree()`.
- `Code *code_table` — A pointer to the Code Table, as prepared by `fill_code_table()`.

The pseudocode below gives the compression algorithm.

```
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
```

Below is a recursive routine that writes the code tree.

```
def huff_write_tree(outbuf, node):
    if node->left == NULL:
        /* node is a leaf */
        write bit 1 to outbuf
        write uint8 node->symbol to outbuf
    else:
        /* node is internal */
        huff_write_tree(outbuf, node->left)
        huff_write_tree(outbuf, node->right)
        write bit 0 to outbuf
```

7 Huffman Decoding

Huffman Decoding reads the code tree and then uses it to decompress the compressed file. The algorithm uses a stack pointers to Node elements (`Node *`). The stack stores allocated nodes as they are assembled into the code tree. Wherever you see `stack_push()`, that means to push a Node pointer into the stack. Where you see `stack_pop()`, that means to pop a Node pointer from the stack. You can implement the stack in any way that you like, but a `Node *stack[64]` array and a top-of-stack `int` will work.

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

```
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 is a leaf:
                break
        write uint8 node->symbol to fout
```

8 Command line options

Your programs should support these command-line options. `-i` and `-o` are required.

- `-i` : Sets the name of the input file. Requires a filename as an argument.
- `-o` : Sets the name of the output file. Requires a filename as an argument.
- `-h` : Prints a help message to `stdout`.

The reference programs also supports a verbose command-line option: `-v`. Your program does not need to support `-v`.

- `-v` : Print verbose information about the input file. Supported by `huff-ref` and `dehuff-ref` only.

9 Program Output and Error Handling

If any invalid options or files are specified, your program should report an error and exit cleanly. Your program should be able to compress both text and binary files, and if you follow the steps in this assignment, it should.

10 Testing your code

To get you started on testing your code, we have provided you four test programs:

- `bwtest.c` — Test your `bitwriter.c` functions.
- `brtest.c` — Test your `bitreader.c` functions.
- `nodetest.c` — Test your `node.c` functions.
- `pqtest.c` — Test your `pq.c` functions.

Although we are not promising that the tests will find all bugs, we *strongly* suggest that you test each module before using it in your program. Also, be aware that `pq.c` uses `bitwriter.c` and `node.c`, and so it would be best to test `bitwriter.c` and `node.c` to be sure that they are working before testing `pq.c`.

- You will receive a folder of test files. Your program should be able to successfully compress these files, the the decompressed files should match the originals.
- Your program should have no *memory leaks*. Make sure you `free()` before exiting. `valgrind` should pass cleanly with any combination of the specified command-line options, including on an error condition. Note that `valgrind` does report errors other than memory leaks, such as invalid reads/writes and use of uninitialized data. These errors are generally *worse* than leaks and you should fix them as well.
- Your program must pass the static analyzer, `scan-build`. You can run this by running `make clean` and then `scan-build --use-cc=clang make`. If scan-build reports a bug that you think is actually not a bug, explain in your `design.pdf` document why you think it is wrong.

11 Questions to answer in your Design PDF

Along with the usual design template, answer the following questions:

- Describe the goal of compression. (As a hint, why is it easy to compress the string "aaaaaaaaaaaaaaa")
- 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?
- Can your program accept any file as an input? Will compressing a file using Huffman coding make it smaller in every case?
- How big are the patterns found by Huffman Coding? What kind of files does this lend itself to?
- Take a picture on your phone. What resolution is the picture? How much space does it take up in your storage (in 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 the image you took is smaller?
- 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.

- Do you expect to be able to compress the image you just took a second time with your Huffman program?[1] Why or why not?
- Are there multiple ways to achieve the same smallest file size? Explain why this might be.
- When traversing the code tree, is it possible for a internal node to have a symbol?
- Why do we bother creating a histogram instead of randomly assigning a tree.
- 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 spaces and newlines?possible
- Using the example binary, calculate the compression ratio of a large text of your choosing [2]

12 Submission

Your submission must have these files. You must have run `clang-format` on the `.c` and `.h` files. While you must submit at least the provided source code for the four test programs, you may add additional tests to these files if you want.

- `bitreader.h` — *provided header file*
- `bitwriter.h` — *provided header file*
- `node.h` — *provided header file*
- `pq.h` — *provided header file*

- `brtest.c` — *provided unit-test file*
- `bwtest.c` — *provided unit-test file*
- `nodetest.c` — *provided unit-test file*
- `pqtest.c` — *provided unit-test file*

- `design.pdf` — Your Design
- `bitreader.c` — Your BitReader functions
- `bitwriter.c` — Your BitWriter functions
- `node.c` — Your Node functions
- `pq.c` — Your PriorityQueue functions
- `huff.c` — Your Huffman Coder
- `dehuff.c` — Your Huffman Decoder
- `Makefile` – Your Makefile
  - The compiler must be `clang`, and the compiler flags must include `-Werror -Wall -Wextra -Wconversion -Wdouble-promotion -Wstrict-prototypes -pedantic`
  - `make` and `make all` must build `brtest`, `bwtest`, `nodetest`, `pqtest`, `huff`, and `dehuff`.
  - `make huff` and `make dehuff` must build your compressor and decompressor programs.
  - `make brtest`, `make bwtest`, `make nodetest`, and `make pqtest`, must build the respective test programs.
  - `make format` must format all C and header files using `clang-format`.
  - `make clean` must delete all object files and compiled programs.

[1] We will define "Working" as compression that makes a file significantly smaller.
[2] At least 2^32 bytes

13 Supplemental Readings

- *The C Programming Language* by Kernighan & Ritchie
- *Managing Projects with GNU Make, 3rd ed.* by Robert Mecklenburg

14 Revisions

Version 1 Original.

References

[1] Elliot Lichtman. Data compression drives the internet. here's how it works. `https://www.quantamagazine.org/how-lossless-data-compression-works-20230531`.

[2] David A Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the I.R.E.*, 40(9):1098–1101, September 1952.