

# Assignment 8

## Huffman Coding

by Dr. Kerry Veenstra  
edits by Jess Srinivas  
CSE 13S, Fall 2023  
Document version 2 (changes in Section 14)

**Report Draft** Due **Friday, December 1<sup>st</sup>**, 2023, at 11:59 pm  
**Final Report** Due **Wednesday, December 6<sup>th</sup>**, 2023, at 11:59 pm  
**Final Code** Due **Monday, December 11<sup>th</sup>**, 2023, at 11:59 pm

## 1 Introduction

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 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;
```

---

```

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

```

Function descriptions are below.

#### **BitWriter \*bit\_write\_open(const char \*filename);**

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

```

1  def bit_write_open(filename):
2      allocate a new BitWriter
3      open the filename for writing as a binary file, storing the result in FILE *f
4      store f in the BitWriter field underlying_stream
5      clear the byte and bit_positions fields of the BitWriter to 0
6      if any step above causes an error:
7          return NULL
8      else:
9          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.

```

1  def bit_write_close(BitWriter **pbuf):
2      if *pbuf != NULL:
3          if (*pbuf)->bit_position > 0:
4              /* (*pbuf)->byte contains at least one bit that has not yet been written */
5              write the byte to the underlying_stream using fputc()
6          close the underlying_stream
7          free the BitWriter
8          *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.

---

```

1 def bit_write_bit(buf, bit):
2     if bit_position > 7:
3         write the byte to the underlying_stream using fputc()
4         clear the byte and bit_position fields of the BitWriter to 0
5         set the bit at bit_position of the byte to the value of bit
6         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*.

```

1 def bit_write_uint8(buf, x):
2     for i = 0 to 7:
3         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*.

```

1 def bit_write_uint16(buf, x):
2     for i = 0 to 15:
3         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*.

```

1 def bit_write_uint32(buf, x):
2     for i = 0 to 31:
3         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 is 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.

```
1  def bit_read_close(BitReader **pbuf):
2      if *pbuf != NULL:
3          close the underlying_stream
4          free *pbuf
5          *pbuf = NULL
6          if any step above causes an error:
7              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.

```
1  def bit_read_bit(buf):
2      if bit_position > 7:
3          read a byte from the underlying_stream using fgetc()
4          bit_position = 0
5          get the bit numbered bit_position from byte
6          bit_position += 1;
7          if any step above causes an error:
8              report fatal error
9      else:
10         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).

```
1  def bit_read_uint8(buf):
2      uint8_t byte = 0x00
3      for i in range(0, 8):
4          read a bit b from the underlying_stream
5          set bit i of byte to the value of b
6      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).

---

```

1  def bit_read_uint16(buf):
2      uint16_t word = 0x0000
3      for i in range(0, 16):
4          read a bit b from the underlying_stream
5          set bit i of word to the value of b
6      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).

```

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

```

## 4 Node

Use Nodes 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.

```

1  /* node.h */
2  typedef struct Node Node;

3  struct Node {
4      uint8_t symbol;
5      uint32_t weight;
6      uint64_t code;
7      uint8_t code_length;
8      Node *left;
9      Node *right;
10 };

```

**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.

---

```

1 def node_create(symbol, weight):
2     allocate a new Node
3     set the symbol and weight fields of Node to function parameters symbol and weight
4     if any step above causes an error:
5         return NULL
6     else:
7         return a pointer to the new Node

```

---

**void node\_free(Node \*\*pnode);**

Free \*pnode and set it to NULL.

```

1 def node_free(Node **pnode):
2     if *pnode != NULL:
3         free(*pnode)
4         *pnode = NULL

```

---

**void node\_print\_tree(Node \*tree, char ch, int indentation);**

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.

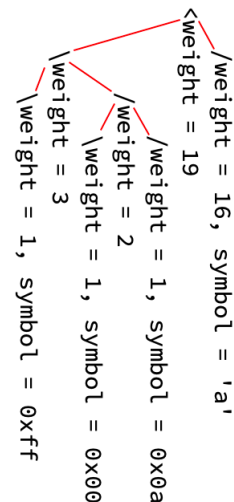
---

```

    /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().



```

1 void node_print_node(Node *tree, char ch, int indentation) {
2     if (tree == NULL)
3         return;
4     node_print_node(tree->right, '/', indentation + 3);
5     printf("%*cweight = %.0f", indentation + 1, ch, tree->weight);

6     if (tree->left == NULL && tree->right == NULL) {
7         if (' ' <= tree->symbol && tree->symbol <= '~') {
8             printf(", symbol = '%c'", tree->symbol);
9         } else {
10            printf(", symbol = 0x%02x", tree->symbol);
11        }
12    }

13    printf("\n");
14    node_print_node(tree->left, '\\', indentation + 3);
15 }

```

```

1 void node_print_tree(Node *tree) {
2     node_print_node(tree, '<', 2);
3 }

```

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.

```

1 /* pq.h */
2 typedef struct PriorityQueue PriorityQueue;

```

---

```

1      /* pq.c */
2      typedef struct ListElement ListElement;

3      struct ListElement {
4          Node *tree;
5          ListElement *next;
6      };

7      struct PriorityQueue {
8          ListElement *list;
9      };

```

### **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.

```

1  def enqueue(node, tree):
2      Allocate ListElement new_element
3      set the tree field to the value of the tree function parameter

4      if the queue is empty:
5          point the queue to new_element
6      elif pq_less_than(new_element, q->list):
7          /* New element E1 goes before all existing elements of the list */
8          /*                                     */
9          /* Q -----> E2 --> E2 ...           */
10         /*      ^                               */
11         /*      E1                             */

12         insert new element E1 as the first element of the list
13     else:
14         /* Either the new element E3 goes at the end of the list: */
15         /*                                     */
16         /* Q --> E1 --> E2                       */
17         /*                                     ^       */
18         /*                                     E3          */

19         /* Or the new element E3 goes before an existing element E4: */
20         /*                                     */
21         /* Q --> E1 --> E2 -----> E4 ...       */
22         /*                                     ^       */
23         /*                                     E3          */

24         /* In both cases, we are looking for existing element E2,    */
25         /* and then we put the new element E3 after it. */

26         find existing element E2 (either E2->next is NULL or E3 < E2->next)
27         insert new element after E2

```

#### **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 `inbuf` provides access to the input file using `read_uint8()`. Parameter `histogram` points to a 256-element array of `uint32_t` values. Clear all elements of this array, and then read bytes from `inbuf` using `read_uint8()`. 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](https://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. Put each node in the priority queue. Be sure that each node is initialized with the symbol and weight of its histogram entry.
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 a 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 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:
    /* 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.

```
1 def huff_compress_file(outbuf, fin, filesize, num_leaves, code_tree, code_table)
2     write uint8_t 'H' to outbuf
3     write uint8_t 'C' to outbuf
4     write uint32_t filesize to outbuf
5     write uint16_t num_leaves to outbuf
6     huff_write_tree(outbuf, code_tree)
7     while true:
8         b = fgetc(fin)
9         if b == EOF:
10             break
11         code = code_table[b].code
12         code_length = code_table[b].code_length
13         for i in range(0, code_length):
14             write bit (code & 1) to outbuf
15             code >>= 1
```

Below is a recursive routine that writes the code tree.

```
1 def huff_write_tree(outbuf, node):
2     if node->left == NULL:
3         /* node is a leaf */
4         write bit 1 to outbuf
5         write uint8 node->symbol to outbuf
6     else:
7         /* node is internal */
8         huff_write_tree(node->left)
9         huff_write_tree(node->right)
10        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)`**

```
1  def dehuff_decompress_file(fout, inbuf):
2      read uint8_t type1 from inbuf
3      read uint8_t type2 from inbuf
4      read uint32_t filesize from inbuf
5      read uint16_t num_leaves from inbuf
6      assert(type1 == 'H')
7      assert(type2 == 'C')
8      num_nodes = 2 * num_leaves - 1
9      Node *node
10     for i in range(0, num_nodes):
11         read one bit from inbuf
12         if bit == 1:
13             read uint8_t symbol from inbuf
14             node = node_create(symbol, 0)
15         else:
16             node = node_create(0, 0)
17             node->right = stack_pop()
18             node->left = stack_pop()
19         stack_push(node)
20     Node *code_tree = stack_pop()
21     for i in range(0, filesize):
22         node = code_tree
23         while true:
24             read one bit from inbuf
25             if bit == 0:
26                 node = node->left
27             else:
28                 node = node->right
29             if node is a leaf:
30                 break
31     write uint8 node->symbol to fout
```

## 8 Using Simple Shell Scripts

We've provided you with the source C code for four unit tests. Your Makefile should compile the unit tests. Also, in the `resources/asgn8/files` directory we've provided you with the data files for a number of system tests. You can run all of the tests using this shell script.

```
$ ./runtests.sh
```

Since the `runtests.sh` script is a text file. You can edit it to see what it does.

---

## 9 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.

## 10 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.

## 11 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 `report.pdf` document why you think it is wrong.

## 12 Submission

This submission is unusual because there are three deadlines. Your report is graded by humans, but your code is graded by a script. So your final code can be turned in after your final report. Consequently, there will be **three** places on Canvas to submit Commit IDs.

- For your **draft report**, you must submit a commit ID on canvas **before Friday, December 1<sup>st</sup> at 11:59 Pacific Time**. You must have a PDF called `report.pdf`. **As usual, draft reports don't have late days.**



- 
- For your **final report**, you must submit a commit ID on canvas **before Wednesday, December 6<sup>th</sup> at 11:59 Pacific Time**. You must have a PDF called `report.pdf`. We understand that your code (which you may submit later) may deviate from your “final” report. Don’t worry. We won’t be comparing the final code to the “final” report. We will grade whatever report is submitted here. Since our readers are undergraduates like you, who need to study for final exams too, **final reports don’t have late days**.
  - For your **final code**, you must submit a commit ID on canvas **before Monday, December 11<sup>th</sup> at 11:59 pm Pacific Time**. We understand that your “final” report (which you may submit earlier than the code) may deviate from your final code. Don’t worry. We won’t be comparing the final code to the “final” report. The **final final** deadline for code (**after three late days**) is Thursday, December 14, 2023, at 11:59 pm. (The same day as the final exam, but in the evening.)

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*
- `report.pdf` — Your Report
- `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.

## 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.

**Version 2** Corrected typos or added clarifications to `bit_write_bit()`, `bit_read_uint32()`, `node_free()`, `pq_less_than()`, `huff_compress_file()`, and `dehuff_decompress_file()`.

## 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.