Assignment 6
Huffman Coding

by Dr. Kerry Veenstra
edits by Ben Grant
CSE 13S, Spring 2023
Document version 2 (changes in Section 12)

Due Thursday June 8th, 2023, at 11:59 pm
Draft Due Friday June 2nd, 2023, at 11:59 pm

1 Introduction

One can use Huffman Coding to compress a data file. The key idea is to determine which bytes ("symbols") of the input file are most common and switch their representation to use fewer than 8 bits. To compensate, the less common symbols will switch to a representation that uses more than 8 bits. The overall result is that the fewer total bits are needed to represent the entire file, meaning that the file has been compressed. Fortunately determining the Huffman Coding for input data is straightforward.

In this assignment, you will write a data compressor, `huff`, that computes the Huffman code of an input file. You will be provided with a program, `dehuff`, that decompresses a Huffman Coded file. You'll also be provided with several unit-test programs: `bwtest.c`, `nodetest.c`, and `pqtest.c`.

As part of this exercise, you will

- ☐ Create a "bit writer" abstract data type (Section 2).
- ☐ Create a binary tree abstract data type (Section 3).
- ☐ Create a priority queue abstract data type (Section 4).
- ☐ Compress a data file using Huffman Coding (Section 5).

2 Bit Writer

In Assignment 5 you wrote Unbuffered File I/O functions to read and write `uint8_t`, `uint16_t`, and `uint32_t` data types. In this assignment, you will reuse the `read_open()`, `read_uint8()`, and `read_close()` functions for reading files. However you will write additional functions for writing files bit-by-bit. Your new functions, which are defined below, will call `write_open()`, `write_uint8()`, and `write_close()`.

2.1 Reusing the Unbuffered File I/O Library

Although we use the same I/O interface as the last assignment, we provide you with an already-debugged library of these functions (files `io-aarch64.a` and `io-x86_64.a`) to ensure that everyone starts on equal footing. (Note: a previous version of this document said you could reuse your own implementation from assignment 5. This is no longer the case; you must use the static library we provide as it makes grading easier.) A `.a` file is a *static library*, essentially a collection of object files (although in this case there is only one) that can be added to your program as a single unit. In your Makefile, do *not* use `io.o`. Instead, list one of these `.a` files where you normally would put `io.o`, like this:

```
EXEC1=huff
OBJS1=huff.o bitwriter.o node.o pq.o
LIBS1=io-$(shell uname -m).a
$(EXEC1): $(OBJS1) $(LIBS1)
        $(CC) -o $@ $^
```

Make sure to use `io-$(shell uname -m).a` as the filename to link against. This makes sure that you always use the appropriate library for whichever architecture your program is being compiled on (i.e. even if you use ARM, a grader who uses x86 can still compile your program as it switches to the x86 library), as the command `uname -m` prints out the architecture name.

2.2 BitWriter Functions

These functions provide a bit-wise interface to the Unbuffered File I/O write functions from the last assignment. So in your Huffman Coding algorithm, do *not* call the `write_open()`, `write_close()`, and `write_uint8()` functions directly. Instead use these functions, which you will write:

- `bit_write_open()` calls `write_open()`
- `bit_write_close()` calls `write_close()`
- `bit_write_bit()` calls `write_uint8()`

In addition, 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 `write_uint8()`.

Finally, **when your Huffman Coding algorithm needs to write 8-bit, 16-bit, and 32-bit data values, do not call the original functions `write_uint8`, `write_uint16`, and `write_uint32` directly**. 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

The functions in this section use these data types.

```
/* put in bitwriter.h */
typedef struct BitWriter BitWriter;

/* put in bitwriter.c */
#include "io.h"
struct BitWriter {
    Buffer *underlying_stream;
    uint8_t byte;
    uint8_t bit_position;
};
```

Function descriptions are below.

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

Open `filename` using `write_open()` and return a pointer to a `BitWriter struct`. On error, return `NULL`. The pseudocode is below.

```
Allocate a BitWriter object, BitWriter *buf.
Create Buffer *underlying_stream using write_open().
buf->underlying_stream = underlying_stream
Return a pointer to the new BitWriter object, buf.
If unable to perform any of the prior steps, report an error and end the program.
```

`void bit_write_close(BitWriter **pbuf);`

Using values in the `BitWriter struct` 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`.

```
if bit_position > 0
    write byte to underlying_stream using write_uint8()
write_close() underlying_stream
free *pbuf
set *pbuf to NULL
```

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

Write a single bit, `x`, using values in the `BitWriter` struct pointed to by `buf`. This function collects 8 bits into the buffer `byte` before writing the entire buffer using `write_uint8()`.

```
if bit_position > 7
    write byte to underlying_stream using write_uint8()
    clear byte to 0x00
    clear bit_position to 0
if x & 1 then byte |= (x & 1) << bit_position
++bit_position;
```

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

Write 8 bits of the `uint8_t x` by calling `bit_write_bit()` 8 times. Start with the LSB (least-significant, or rightmost, bit).

```
for i = 0 to 7
    write bit i of x using bit_write_bit()
```

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

Write 16 bits of the `uint16_t x` by calling `bit_write_bit()` 16 times. Start with the LSB.

```
for i = 0 to 15
    write bit i of x using bit_write_bit()
```

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

Write 32 bits of the `uint32_t x` by calling `bit_write_bit()` 32 times. Start with the LSB.

```
for i = 0 to 31
    write bit i of x using bit_write_bit()
```

3 Node

Use `Node`s to make a binary tree. Each `Node` contains numerous fields that the Huffman Coding algorithm will use. 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.

```
/* put everything in node.h */
typedef struct Node Node;

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

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

Create a `Node` and set its `symbol` and `weight` fields. Return a pointer to the new node.

`void node_free(Node **node);`

Free the `*node` and set it to `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.

To print the tree rooted at `Node *tree`, call the function as `node_print_tree(tree, '<', 2)`.

Recursive Tree Printing Routine

```
void node_print_tree(Node *tree, char ch, int indentation) {
    if (tree == NULL)
        return;
    node_print_tree(tree->right, '/', indentation + 3);
    printf("%*cweight = %.0f", 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_tree(tree->left, '\\', indentation + 3);
}
```

Note: Here's a comment about a feature of `printf()` that we haven't yet talked about. 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. In this case, the `*c` characters mean that the sequence of parameters following the format string contains an integer width (`indentation + 1`) followed by a character (`ch`). We use this this feature to programmatically control the indentation of `ch`.

4 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. Each queue entry has a pointer to a tree, and each tree node has a weight. So 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.

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

/* put in 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. (By using `calloc()` you don't need to manually initialize the `list` data field.)

`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);`

The Huffman Coding algorithm fills the Priority Queue and then runs a loop until the queue contains a single value. This is how we determine that the queue contains a single value. (Although we *could* have created a function that measures the length of the queue, its runtime would be much longer than just checking for a single element.)

`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).

Follow these steps:

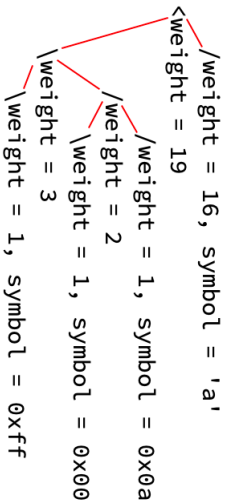
1. Allocate a new `ListElement *e`, and set `e->tree = tree`.
2. The enqueuing operation performed depends on the queue's current state.
   - **Empty queue.** Set `q->list = e`.
   - **Weight of the new tree is less than the weight of the head.** Insert the new element at the beginning of the queue. Set `e->next = q->list` and then `q->list = e`. (Draw a picture to see how this works.)
   - **New element goes after one of the existing elements. Find it.** Starting at `q->list`, follow the linked list until you find the last queue entry (the queue entry whose `next` field is `NULL`) or until you find a queue entry whose next field points to a tree whose weight is greater than that of `tree`. Then insert `e` after the queue element that you found. (Draw a picture to figure this out.)

`bool dequeue(PriorityQueue *q, Node **tree);`

If the queue is empty, return `false`. Otherwise, remove the queue element with the lowest weight, set `e` to point to it, set parameter `*tree = e->tree`, call `free(e)`, and return `true`.

```
   /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 pq_print(PriorityQueue *q);`

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

```
void pq_print(PriorityQueue *q) {
    assert(q != NULL);
    ListElement *e = q->list;
    int position = 1;
    while (e != NULL) {
        if (position++ == 1) {
            printf("================================================\n");
        } else {
            printf("------------------------------------------------\n");
        }
        node_print_tree(e->tree, '<', 2);
        e = e->next;
    }
    printf("================================================\n");
}
```

`bool pq_less_than(Node *n1, Node *n2)`

The `enqueue()` function compares the weights of two `Node` objects, with the order of the nodes in the priority queue determined by the weights. Since it's possible for two nodes to have the same weights, the operation of `enqueue` depends on its implementation. In what order should equally weighted nodes be placed? Does the result depend on the order in which the nodes were added to the queue?

How your `huff` program treats equally weighted nodes will *not* affect its ability to create a valid Huffman Coded file, since a subsequent run of `dehuff` should regenerate the original `huff` program input. However, writing a program that operates deterministically ensures that your program's binay result can be compared with other implementations and ensures that its results will be repeatable for test verification.

Hence, it's prudent to include a tie-breaker in the comparison function, essentially eliminating the possibility that two nodes' weights will compare equal. Since all of the nodes in the queue have unique `symbol`s, using the `symbol` value to break the tie is a natural choice.

Using this function instead of C less-than (`<`) operator is optional since your program will create a correct Huffman Coded file regardless. But including a tie-breaker in the comparision makes deterministic program operation more likely.

```
bool pq_less_than(Node *n1, Node *n2) {
    if (n1->weight < n2->weight) return true;
    if (n1->weight > n2->weight) return false;
    return n1->symbol < n2->symbol;
}
```

5 Huffman Coding

Huffman Coding consists of five steps.

1. 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. **Close and Reopen the input file in preparation for Step 5**
5. Create a Huffman Coded output file from the input file (see `huff_compress_file()`).

5.1 Functions

`uint64_t fill_histogram(Buffer *inbuf, double *histogram)`

This function updates a histogram array 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 `double`s. 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 `uint64_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(double *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. 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. */
    fill_code_table(code_table, node->left, code, code_length + 1);
    code |= 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()`

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.
- `Buffer *inbuf` — Use this parameter with calls to `read_uint8()` to read the input file. **Note: the code assumes that `inbuf` has been closed and reopened after the call to `fill_histogram()`. Remember that `fill_histogram()` already has read the entire input file, and so the buffer needs to be closed and reopened 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.

The pseudocode represents data being written to the output file as rows of a table. The left column indicates the number of bits to write, using `bit_write_bit()`, `bit_write_uint8()`, `bit_write_uint16()`, or `bit_write_uint32()`. The right column gives the value to write.

huff_compress_file(outbuf, inbuf, filesize, num_leaves, code_tree, code_table)

8	'H'
8	'C'
32	filesize
16	num_leaves

```
huff_write_tree(outbuf, code_tree)
for every byte b from inbuf
    code = code_table[b].code
    code_length = code_table[b].code_length
    for i = 0 to code_length - 1
        /* write the rightmost bit of code */
```

1	code & 1

```
        /* prepare to write the next bit */
        code >>= 1
```

Below is a recursive routine that writes the code tree.

huff_write_tree(outbuf, node)

```
if node is an internal node
    huff_write_tree(node->left)
    huff_write_tree(node->right)
```

1	0

```
else
    // node is a leaf
```

1	1
8	node->symbol

6 Using Simple Shell Scripts

We've provided you with a number of test files in the `files` directory. You can run `huff` on all of them manually using commands like this:

```
$ ./huff -i files/zero.txt -o files/zero.huff
$ ./huff -i files/one.txt -o files/one.huff
$ ./huff -i files/two.txt -o files/two.huff
$ ./huff -i files/test1.txt -o files/test1.huff
$ ./huff -i files/test2.txt -o files/test2.huff
$ ./huff -i files/color-chooser-orig.txt -o files/color-chooser-orig.huff
```

You also can run the reverse program, `dehuff-arm` or `dehuff-x86`, on the `.huff` files to make `.dehuff` files that you can compare to the original `.txt` files. For example, on an Apple laptop you would run these commands:

```
$ ./dehuff-arm -i files/zero.huff -o files/zero.dehuff
$ ./dehuff-arm -i files/one.huff -o files/one.dehuff
$ ./dehuff-arm -i files/two.huff -o files/two.dehuff
$ ./dehuff-arm -i files/test1.huff -o files/test1.dehuff
$ ./dehuff-arm -i files/test2.huff -o files/test2.dehuff
$ ./dehuff-arm -i files/color-chooser-orig.huff -o files/color-chooser-orig.dehuff
```

And finally, the comparisons are done like this:

```
$ diff files/zero.txt files/zero.dehuff
$ diff files/one.txt files/one.dehuff
$ diff files/two.txt files/two.dehuff
$ diff files/test1.txt files/test1.dehuff
$ diff files/test2.txt files/test2.dehuff
$ diff files/color-chooserig.txt files/color-chooser-orig.dehuff
```

If there are and differences, then `diff` will report them.

You can do all of this, however it is easier to use a "shell script." We've provided a script called `runtests.sh` that looks in the `files` directory and essentially runs all of the commands above. If you add a file to the files directory whose name ends with `.txt`, then `runtests.sh` will add that file to the tests that it runs.

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

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

A final note: the test file `color-chooser-orig.txt` isn't really a text file. We just renamed the large BMP file from the last assignment.

7 Command line options

Your program 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`.

8 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.

9 Testing your code

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

- `bwtest.c` — Test your `bitwriter.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.

10 Submission

For the **report draft**, you must submit a commit ID on canvas *before Friday June 2nd at 11:59 Pacific Time*. You must have a PDF called `report.pdf`.

For the entire project, you must submit a commit ID on canvas *before Thursday June 8th at 11:59 pm Pacific Time*.

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 three test programs, you may add additional tests to these files if you want.

- `report.pdf` — Your Report
- `bitwriter.c` — Your BitWriter functions
- `bitwriter.h` — *provided*

- `huff.c` — Your Huffman Coder
- `io-aarch64.a` — *provided*
- `io-x86_64.a` — *provided*
- `io.h` — *provided*
- `node.c` — Your Node functions
- `node.h` — *provided*
- `pq.c` — Your PriorityQueue functions
- `pq.h` — *provided*
- `pqtest.c` — *provided*
- `nodetest.c` — *provided*
- `bwtest.c` — *provided*
- `Makefile` – Your Makefile
  - The compiler must be `clang`, and the compiler flags must include `-Wall -Wextra -Werror -pedantic -Wstrict-prototypes`.
  - `make` and `make all` must build `huff`, `pqtest`, `nodetest`, and `bwtest`.
  - `make huff` must build your program
  - `make pqtest`, `make nodetest`, and `make bwtest` 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 (not including the `.a` static libraries).

11 Supplemental Readings

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

12 Revisions

Version 1 Original.

Version 2 Added mention of `uname -m` to select appropriate IO library and correct architecture names. Corrected commands in Section 6. `enqueue()` no longer returns a `bool`. Added requirement to submit test programs and support compiling them in `Makefile`. Added instructions to run `scan-build` and instructions in case it reports a false positive.

References

[1] Y. S. Abu-Mostafa and R. J. McEliece. Maximal Codeword Lengths in Huffman Codes. In *Telecommunications and Data Acquisition Progress Report 42-110*, pages 188–193. Jet Propulsion Laboratory, Pasadena, California, August 15 1992.

Appendix A: Longest Expected Codeword

Your `huff` program internally stores code words in 64-bit `uint64_t` variables. This appendix confirms that the longest possible Huffman codewords that result from compressing practically sized files will fit in 64 bits.

We start with a theorem by Abu-Mostafa et al.[1] which shows how to compute the length K of the longest possible Huffman codeword given the probability p of the least likely input symbol. One starts with this inequality:

$$\frac{1}{F_{K+3}} < p \leq \frac{1}{F_{K+2}} \tag{2}$$

where F_n is the n-th Fibonnaci number from the sequence below (the equation numbers in this Appendix match those of the reference).

$$F_0 = 0, F_1 = 1, F_n = F_{n-1} + F_{n-2} \text{ for } n \geq 2 \tag{1}$$

The theorem shows that given the probability p of the input's least likely symbol, then the value of K that satisfies (2) is the number of bits in the longest Huffman codeword.

We know that the probability p of the least likely input symbol depends on the size of the source file that is being converted. That is, assume that a file of size s has one symbol that appears only once. Then the probability of that symbol is

$$p = \frac{1}{s}$$

Substituting into (2) we get

$$\frac{1}{F_{K+3}} < \frac{1}{s} \leq \frac{1}{F_{K+2}}$$

whose expressions can be inverted (and inequalities reversed).

$$F_{K+3} > s \geq F_{K+2}$$

Summarizing, given a longest Huffman codeword of K bits, the size of an input file s cannot exceed the value of Fibonacci number $K + 3$.

$$s < F_{K+3}$$

Since `huff` stores codewords in fields of type `uint64_t`, $K = 64$ and we can compute the maximum allowed size of an input file using the 67^{th} Fibonacci number[1].

$$\begin{aligned} s &< F_{64+3} \\ s &< F_{67} \\ s &< 44\,945\,570\,212\,853 \\ s &\lessapprox 4.5 \times 10^{13} \end{aligned}$$

Or a maximum file size of about 45 terabytes (45 TB). Are files of this size likely or possible?

First, 45 TB exceeds the capacity of laptop storage devices (look at the capacity of your laptop). So a single file of that size is not likely on your laptop. However, if one has a desktop computer and is willing to pay, one can get an internal storage card with 64-TB capacity[2]. So individual storage devices support sizes beyong our 45-TB limit. And beyond that, external disk arrays can store an order of magnitude more[3]. So considering storage capacity limits only, a file exceeding 45 TB in size is possible but unlikely.

Operating systems limit file sizes, too. Older operating systems enforced limits of 4 GB (FAT32), but modern operating systems practically eliminate file-size limits (Apple's APFS: $2^{63} \approx 9.2 \times 10^{18}$. Microsoft's

[1] Enter "fibonacci(67)" at wolframalpha.com.
[2] OWC 64-TB Accelsior 8M2 PCIe 4.0 Storage Card.
[3] Rocstor Enteroc F1632 Dual Controller 256-TB 16-Bay SAS Array (16×16-TB HDDs).

NTFS: $2^{64} - 1 \approx 1.8 \times 10^{19}$). So it appears that it is *possible* to exceed the 45-TB file size limit that results from this assignment's 64-bit Huffman codeword limit.

Going beyond this assignment, how long would the longest Huffman codeword be when compressing the largest APFS or NTFS file?

$$\begin{array}{rcl} F_{89+3} & \approx & 7.5 \times 10^{18} \\ 2^{63} & \approx & 9.2 \times 10^{18} \\ F_{90+3} & \approx & 1.2 \times 10^{19} \\ 2^{64} & \approx & 1.8 \times 10^{19} \\ F_{91+3} & \approx & 2.0 \times 10^{19} \end{array}$$

Meaning that in some future where we want to compress maximally sized APFS and NTFS files, we'd need to prepare for a codeword with $K = 91$ bits.

But for this assignment, 64 bits will suffice.

Appendix B: Reading a Huffman Coded File

This Appendix documents the procedure used by `dehuff` to read your Huffman Coded files. You don't write `dehuff`; this Appendix is here for the person who does.

`void huff_decompress_file()`

Once again, you aren't going to write a program to read a Huffman Coded file, but *someone* is, and this Appendix is for them. The pseudocode below represents data being read from the file as rows of a table. The left column indicates the number of bits to read, using `bit_read_bit()`, `bit_read_uint8()`, `bit_read_uint16()`, or `bit_read_uint32()`. The right column gives the variable that receives the data and its type.

huff_decompress_file(outbuf, inbuf)

8	uint8_t type1
8	uint8_t type2
32	uint32_t filesize
16	uint16_t num_leaves

```
Node *node
assert(type1 == 'H')
assert(type2 == 'C')
uint16_t num_nodes = 2 * num_leaves - 1
for (int i = 0; i < num_nodes; ++i) {
```

1	uint8_t bit

```
    if bit == 1
```

8	uint8_t symbol

```
        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 (int i = 0; i < filesize; ++i) {
    node = code_tree
    do {
```

1	uint8_t bit

```
        if bit == 0
            node = node->left
        else
            node = node->right
    } while node is not a leaf
    write_uint8(outbuf, node->symbol)
}
```