Assignment 6
Huffman Coding

Prof. Darrell Long
CSE 13S – Spring 2021

First `DESIGN.pdf` draft due: May 13th at 11:59 pm PST
Assignment due: May 23rd at 11:59 pm PST

1 Introduction

> *You're trying to take something that can be described in many, many sentences and pages of prose, but you can convert it into a couple lines of poetry and you still get the essence, so that's compression. The best code is poetry.*
>
> —Satya Nadella

When David Huffman was a graduate student in a class at MIT, the professor gave the class an unsolved problem: How to construct an optimal static encoding of information. The young Huffman came back a few days later with his solution, and that solution changed the world. Data compression is now used in all aspects of communication. David Huffman joined the faculty of MIT in 1953, and in 1967 he joined the faculty of University of California, Santa Cruz as one of its earliest members and helped to found its Computer Science Department, where he served as chairman from 1970 to 1973. He retired in 1994, and passed away in 1999.


David A. Huffman

The key idea is called *entropy*, originally defined by Claude Shannon in 1948. Entropy is a measure of the amount of information in a, say, set of symbols. If we define $I(x) = \log_2 \Pr[x]$ to be the information content of a symbol, then the entropy of the set $X = \{x_1, \ldots, x_n\}$ is

$$H(X) = \sum_{i=1}^{n} \Pr[x_i] I(x_i) = -\sum_{i=1}^{n} \Pr[x_i] \log_2 \Pr[x_i].$$

It should be easy to see that the optimal *static* encoding will assign the least number of *bits* to the most common symbol, and the greatest number of bits to the least common symbol.

2 The Encoder

Your first task for this assignment is to implement a Huffman encoder. This encoder will read in an input file, find the Huffman encoding of its contents, and use the encoding to compress the file. Your encoder program, named `encode`, must support any combination of the following command-line options:

- `-h`: Prints out a help message describing the purpose of the program and the command-line options it accepts, exiting the program afterwards. Refer to the reference program in the resources repo for an idea of what to print.
- `-i infile`: Specifies the input file to encode using Huffman coding. The default input should be set as `stdin`.
- `-o outfile`: Specifies the output file to write the compressed input to. The default output should be set as `stdout`.
- `-v`: Prints compression statistics to `stderr`. These statistics include the uncompressed file size, the compressed file size, and *space saving*. The formula for calculating space saving is:

  $$100 \times (1 - (\text{compressed size} / \text{uncompressed size})).$$

  Refer to the reference program in the resources repository for the exact output.

The algorithm to encode a file, or to compress it, is as follows:

1. Compute a histogram of the file. In other words, count the number of occurrences of each unique symbol in the file.
2. Construct the Huffman tree using the computed histogram. This will require a *priority queue*.
3. Construct a code table. Each index of the table represents a symbol and the value at that index the symbol's code. You will need to use a *stack of bits* and perform a traversal of the Huffman tree.
4. Emit an encoding of the Huffman tree to a file. This will be done through a *post-order traversal* of the Huffman tree. The encoding of the Huffman tree will be referred to as a *tree dump*.
5. Step through each symbol of the input file again. For each symbol, emit its code to the output file.

3 The Decoder

The second task for this assignment is to implement a Huffman decoder. This decoder will read in a compressed input file and decompress it, expanding it back to its original, uncompressed size. Your decoder program, named `decode`, must support any combination of the following command-line options.

- `-h`: Prints out a help message describing the purpose of the program and the command-line options it accepts, exiting the program afterwards. Refer to the reference program in the resources repo for an idea of what to print.
- `-i infile`: Specifies the input file to decode using Huffman coding. The default input should be set as `stdin`.
- `-o outfile`: Specifies the output file to write the decompressed input to. The default output should be set as `stdout`.

- `-v`: Prints decompression statistics to `stderr`. These statistics include the compressed file size, the decompressed file size, and *space saving*. The formula for calculating space saving is:

$$100 \times (1 - (\text{compressed size} / \text{decompressed size})).$$

  Refer to the reference program in the resources repository for the exact output.

The algorithm to decode a file, or to decompress it, is as follows:

1. Read the emitted (*dumped*) tree from the input file. A *stack of nodes* is needed in order to reconstruct the Huffman tree.
2. Read in the rest of the input file bit-by-bit, traversing down the Huffman tree one link at a time. Reading a 0 means walking down the left link, and reading a 1 means walking down the right link. Whenever a leaf node is reached, its symbol is emitted and you start traversing again from the root.

4 Nodes

The first ADT that we will cover is a *node*. Huffman trees are composed of nodes, with each node containing a pointer to its left child, a pointer to its right child, a symbol, and the frequency of that symbol. The node's frequency is only needed for the encoder.

```
1 typedef struct Node Node;
2
3 struct Node {
4     Node *left;          // Pointer to left child.
5     Node *right;         // Pointer to right child.
6     uint8_t symbol;      // Node's symbol.
7     uint64_t frequency; // Frequency of symbol.
8 };
```

Immediately, we notice that a symbol is a `uint8_t`, and not a `char`. This is because we want to interpret the input file as *raw bytes*, not as a string. The following subsections define the interface for a `Node` and will be supplied in `node.h`. The definition of a `Node` will be made transparent in order to simplify things.

4.1 `Node *node_create(uint8_t symbol, uint64_t frequency)`

The constructor for a node. Sets the node's symbol as `symbol` and its frequency as `frequency`.

4.2 `void node_delete(Node **n)`

The destructor for a node. Make sure to set the pointer to `NULL` after freeing the memory for a node.

4.3 `Node *node_join(Node *left, Node *right)`

Joins a left child node and right child node, returning a pointer to a created parent node. The parent node's left child will be `left` and its right child will be `right`. The parent node's symbol will be `'$'` and its frequency the *sum* of its *left* child's frequency and its *right* child's frequency.

4.4 `void node_print(Node *n)`

A debug function to verify that your nodes are created and joined correctly.

5 Priority Queues

As stated in the encoding algorithm, the encoder will make use of a *priority queue* of nodes. A priority queue functions like a regular queue, but assigns each of its elements a *priority*, such that elements with a high priority are dequeued before elements with a low priority. Assuming that elements are enqueued at the tail and dequeued from the head, this implies that the `enqueue()` operation does not simply add the element at the tail. Of course, the `dequeue()` operation could *search* for the highest priority element each time, but that is a *bad idea*.

How you implement your priority queue is up to you. There are a couple choices: 1) mimicking an *insertion sort* when enqueuing a node, finding the correct position for the node and shifting everything back, or 2) using a *min heap* to serve as the priority queue. Why a min heap? Because we want nodes with *lower* frequencies to be dequeued first. The lower the frequency of a node, the higher its priority. Your priority queue, no matter the implementation, *must* fulfill the interface that will be supplied to you in `pq.h`.

5.1 `PriorityQueue *pq_create(uint32_t capacity)`

The constructor for a priority queue. The priority queue's maximum capacity is specified by `capacity`.

5.2 `void pq_delete(PriorityQueue **q)`

The destructor for a priority queue. Make sure to set the pointer to `NULL` after freeing the memory for a priority queue.

5.3 `bool pq_empty(PriorityQueue *q)`

Returns `true` if the priority queue is empty and `false` otherwise.

5.4 `bool pq_full(PriorityQueue *q)`

Returns `true` if the priority queue is full and `false` otherwise.

5.5 `uint32_t pq_size(PriorityQueue *q)`

Returns the number of items currently in the priority queue.

5.6 `bool enqueue(PriorityQueue *q, Node *n)`

Enqueues a node into the priority queue. Returns `false` if the priority queue is full prior to enqueuing the node and `true` otherwise to indicate the successful enqueuing of the node.

5.7 `bool dequeue(PriorityQueue *q, Node **n)`

Dequeues a node from the priority queue, passing it back through the double pointer `n`. The node dequeued should have the *highest* priority over all the nodes in the priority queue. Returns `false` if the priority queue is empty prior to dequeuing a node and `true` otherwise to indicate the successful dequeuing of a node.

5.8 `void pq_print(PriorityQueue *q)`

A debug function to print a priority queue. This function will be significantly easier to implement if your `enqueue()` function always ensures a *total ordering* over all nodes in the priority queue. Enqueuing nodes in a insertion-sort-like fashion will provide such an ordering. Implementing your priority queue as a heap, however, will only provide a *partial ordering*, and thus will require more work in printing to assure you that your priority queue functions as expected (you will be displaying a *tree*).

6 Codes

After constructing a Huffman tree, you will need to maintain a stack of bits while traversing the tree in order to create a code for each symbol. We will create a new ADT, a `Code`, that represents a stack of bits.

```
1 typedef struct Code {
2     uint32_t top;
3     uint8_t bits[MAX_CODE_SIZE];
4 } Code;
```

The `struct` definition of a `Code` will be made transparent. This is done for the sole purpose of being able to pass a `struct` by value. The macro `MAX_CODE_SIZE` reflects the maximum number of bytes needed to store any valid code. The definition of this macro — and other macros — will be given in `defines.h`. You will need to combine your knowledge of bit vectors and stacks in order to implement this ADT. The interface, given in `code.h`, is defined in the the following subsections.

Macros defined in `defines.h`

```
// 4KB blocks.
#define BLOCK 4096

// ASCII + Extended ASCII.
#define ALPHABET 256

// 32-bit magic number.
#define MAGIC 0xDEADBEEF

// Bytes for a maximum, 256-bit code.
#define MAX_CODE_SIZE (ALPHABET / 8)

// Maximum Huffman tree dump size.
#define MAX_TREE_SIZE (3 * ALPHABET - 1)
```

6.1 `Code code_init(void)`

You will immediately notice that this "constructor" function is unlike any of the other constructor functions you have implemented in the past. You may also have noticed, if you glanced slightly ahead, that there is no corresponding destructor function. This is an engineering decision that was made when considering the constraints of the Huffman coding algorithm.

This function *will not* require any dynamic memory allocation. You will simply create a new `Code` on the stack, setting `top` to 0, and zeroing out the array of bits, `bits`. The initialized `Code` is then returned.

6.2 `uint32_t code_size(Code *c)`

Returns the size of the `Code`, which is exactly the number of bits pushed onto the `Code`.

6.3 `bool code_empty(Code *c)`

Returns `true` if the `Code` is empty and `false` otherwise.

6.4 `bool code_full(Code *c)`

Returns `true` if the `Code` is full and `false` otherwise. The maximum length of a code in bits is 256, which we have defined using the macro `ALPHABET`. Why 256? Because there are exactly 256 ASCII characters (including the extended ASCII).

6.5 `bool code_push_bit(Code *c, uint8_t bit)`

Pushes a bit onto the `Code`. The value of the bit to push is given by `bit`. Returns `false` if the `Code` is full prior to pushing a bit and `true` otherwise to indicate the successful pushing of a bit.

6.6 `bool code_pop_bit(Code *c, uint8_t *bit)`

Pops a bit off the `Code`. The value of the popped bit is passed back with the pointer `bit`. Returns `false` if the `Code` is empty prior to popping a bit and `true` otherwise to indicate the successful popping of a bit.

6.7 `void code_print(Code *c)`

A debug function to help you verify whether or not bits are pushed onto and popped off a `Code` correctly.

7 I/O

Now that we have covered all the essential ADTs necessary for the encoder, we will discuss I/O. Instead of the buffered I/O functions from `<stdio.h>` that you have become acquainted with in previous assignments, we will use low-level system calls (*syscalls*) such as `read()`, `write()`, `open()` and `close()`. The former two functions can be included with `<unistd.h>` and the latter two can be included with `<fcntl.h>`. Functions defined by the following I/O module will be used by both the encoder and decoder. The interface for the I/O module will be supplied in `io.h`.

7.1 `int read_bytes(int infile, uint8_t *buf, int nbytes)`

This will be a useful wrapper function to perform reads. As you may know, the `read()` syscall *does not* always guarantee that it will read all the bytes specified (as is the case with *pipes*). For example, a call could be issued to read a a block of bytes, but it might only read part of a block. So, we write a wrapper function to *loop calls* to `read()` until we have either read all the bytes that were specified (`nbytes`) into the byte buffer `buf`, or there are no more bytes to read. The number of bytes that were read from the input file descriptor, `infile`, is returned. You should use this function whenever you need to perform a read.

7.2 `int write_bytes(int outfile, uint8_t *buf, int nbytes)`

This functions is very much the same as `read_bytes()`, except that it is for looping calls to `write()`. As you may imagine, `write()` is not guaranteed to write out all the specified bytes (`nbytes`), and so we must loop until we have either written out all the bytes specified from the byte buffer `buf`, or no bytes were written. The number of bytes written out to the output file descriptor, `outfile`, is returned. You should use this function whenever you need to perform a write.

7.3 `bool read_bit(int infile, uint8_t *bit)`

You should all know by now that it is *not* possible to read a single bit from a file. What you *can* do, however, is read in a block of bytes into a buffer and dole out bits one at a time. Whenever all the bits in the buffer have been doled out, you can simply fill the buffer back up again with bytes from `infile`. This is exactly what you will do in this function. You will maintain a `static` buffer of bytes and an index into the buffer that tracks which bit to return through the pointer `bit`. The buffer will store `BLOCK` number of bytes, where `BLOCK` is yet another macro defined in `defines.h`. This function returns `false` if there are no more bits that can be read and `true` if there are still bits to read. It may help to treat the buffer as a *bit vector*.

7.4 `void write_code(int outfile, Code *c)`

The same bit-buffering logic used in `read_bit()` will be used in here as well. This function will also make use of a `static` buffer (we recommend this buffer to be static to the file, not just this function) and an index. Each bit in the code `c` will be buffered into the buffer. The bits will be buffered starting from the 0^{th} bit in `c`. When the buffer of `BLOCK` bytes is filled with bits, write the contents of the buffer to `outfile`.

7.5 `void flush_codes(int outfile)`

It is not always guaranteed that the buffered codes will align nicely with a block, which means that it is possible to have bits leftover in the buffer used by `write_code()` after the input file has been completely encoded. The sole purpose of this function is to write out any leftover, buffered bits. Make sure that any extra bits in the last byte are zeroed before flushing the codes.

8 Stacks

You will need to use a *stack* in your decoder to reconstruct a Huffman tree. The interface of the stack should be familiar from assignments 3 and 4. The difference is that the stack this time around will store *nodes*. The interface for the stack is defined in `stack.h`.

```
1 struct Stack {
2     uint32_t top;
3     uint32_t capacity;
4     Node **items;
5 };
```

8.1 `Stack *stack_create(uint32_t capacity)`

The constructor for a stack. The maximum number of nodes the stack can hold is specified by `capacity`.

8.2 `void stack_delete(Stack **s)`

The destructor for a stack. Remember to set the pointer to `NULL` after you free the memory allocated by the stack.

8.3 `bool stack_empty(Stack *s)`

Returns `true` if the stack is empty and `false` otherwise.

8.4 `bool stack_full(Stack *s)`

Returns `true` if the stack is full and `false` otherwise.

8.5 `uint32_t stack_size(Stack *s)`

Returns the number of nodes in the stack.

8.6 `bool stack_push(Stack *s, Node *n)`

Pushes a node onto the stack. Returns `false` if the stack is full prior to pushing the node and `true` otherwise to indicate the successful pushing of a node.

8.7 `bool stack_pop(Stack *s, Node **n)`

Pops a node off the stack, passing it back through the double pointer `n`. Returns `false` if the stack is empty prior to popping a node and `true` otherwise to indicate the succesuccessful popping of a node.

8.8 `void stack_print(Stack *s)`

A debug function to print the contents of a stack.

9 A Huffman Coding Module

A interface for a Huffman coding module that you will need to implement will be given in `huffman.h`. Do not worry if you do not initially understand the exact purpose of each function, as they will be clarified in §10 and §11. The interface is just given now as a reference for which functions are used in the aforementioned sections.

9.1 `Node *build_tree(uint64_t hist[static ALPHABET])`

Constructs a Huffman tree given a computed histogram. The histogram will have `ALPHABET` indices, one index for each possible symbol. Returns the root node of the constructed tree. The use of `static` array indices in parameter declaration is a C99 addition. In this case, it informs the compiler that the histogram `hist` should have *at least* `ALPHABET` number of indices.

9.2 `void build_codes(Node *root, Code table[static ALPHABET])`

Populates a code table, building the code for each symbols in the Huffman tree. The constructed codes are copied to the code table, `table`, which has `ALPHABET` indices, one index for each possible symbol.

9.3 `Node *rebuild_tree(uint16_t nbytes, uint8_t tree_dump[static nbytes])`

Reconstructs a Huffman tree given its post-order tree dump stored in the array `tree_dump`. The length in bytes of `tree_dump` is given by `nbytes`. Returns the root node of the reconstructed tree.

9.4 `void delete_tree(Node **root)`

The destructor for a Huffman tree. This will require a post-order traversal of the tree to free all the nodes. Remember to set the pointer to `NULL` after you are finished freeing all the allocated memory.

10 Specifics for the Encoder

For this section, the input file to compress will be referred to as `infile` and the compressed output file as `outfile`. Your encoder, after parsing command-line options with `getopt()`, must perform the following steps exactly to produce the correct Huffman encoding:

1. Read through `infile` to construct a histogram. Your histogram should be a simple array of 256 (`ALPHABET`) `uint64_t`s.
2. Increment the count of element 0 and element 255 by one in the histogram. This is so that at the very minimum, the histogram will have two elements present. Do this regardless of what you read in. While doing this may result in a slightly sub-optimal Huffman tree later on, it is a quick and clean solution to handling the case when a file has no bytes or contains only one unique symbol.
3. Construct the Huffman tree using a priority queue. This will be done using `build_tree()`.
   (a) Create a priority queue. For each symbol histogram where its frequency is greater than 0 (there should be at minimum two elements because of step 2), create a corresponding `Node` and insert it into the priority queue.
   (b) While there are two or more nodes in the priority queue, dequeue two nodes. The first dequeued node will be the left child node. The second dequeued node will be the right child node. Join these nodes together using `node_join()` and enqueue the joined parent node. The frequency of the parent node is the sum of its left child's frequency and its right child's frequency.
   (c) Eventually, there will only be one node left in the priority queue. This node is the *root* of the constructed Huffman tree.
4. Construct a code table by traversing the Huffman tree. This will be done using `build_codes()`. The code table is a simple array of 256 (`ALPHABET`) `Code`s.
   (a) Create a new `Code c` using `code_init()`. Starting at the root of the Huffman tree, perform a *post-order* traversal.
   (b) If the current node is a leaf, the current code `c` represents the path to the node, and thus is the code for the node's symbol. Save this code into code table.
   (c) Else, the current node must be an interior node. Push a 0 to `c` and recurse down the left link.
   (d) After you return from the left link, pop a bit from `c`, push a 1 to `c` and recurse down the right link. Remember to pop a bit from `c` when you return from the right link.
5. Construct a *header*. A header is defined by the following `struct` definition, which will be supplied to you in `header.h`:

```
1 typedef struct Header {
2     uint32_t magic;       // 32-bit magic number.
3     uint16_t permissions; // Input file permissions.
4     uint16_t tree_size;   // Emitted tree size in bytes.
5     uint64_t file_size;   // Input file size.
6 } Header;
```

The header's magic number field, `magic`, should be set to the macro `MAGIC`, as defined in `defines.h`. This magic number identifies a file as one which has been compressed using your encoder. It is crucial that you use this magic number and nothing else.

The header's `permissions` field stores the original permission bits of `infile`. You can get these permissions by using `fstat()`. You should also set the permissions of `outfile` to match the permissions of `infile` using `fchmod()`.

The header's `tree_size` field represents the number of bytes that make up the Huffman tree dump. This size is calculated as $(3 \times \text{unique symbols}) - 1$.

Finally, the header's `file_size` field is the size in bytes of the file to compress, `infile`. You get this size using `fstat()` as well.

6. Write the constructed header to `outfile`.
7. Perform a *post-order traversal* of the Huffman tree to write the tree to the `outfile`. This should write an `'L'` followed by the byte of the symbol for each leaf, and an `'I'` for interior nodes. You should not write a symbol for an interior node.
8. Starting at the beginning of `infile`, write the corresponding code for each symbol to `outfile` with `write_code()`. When finished with all the symbols, make sure to flush any remaining buffered codes with `flush_codes()`.
9. Close `infile` and `outfile`.

11 Specifics for the Decoder

For this section, the input file to decompress will be referred to as `infile` and the compressed output file as `outfile`. Your decoder, after parsing command-line options with `getopt()`, must perform the following steps exactly to decode the file:

1. Read in the header from `infile` and verify the magic number. If the magic number does not match `0xDEADBEEF` (defined as `MAGIC` in `defines.h`), then an invalid file was passed to your program. Display a helpful error message and quit.
2. The `permissions` field in the header indicates the permissions that `outfile` should be set to. Set the permissions using `fchmod()`.
3. The size of the dumped tree is given by the `tree_size` field in the header. Read the dumped tree from `infile` into an array that is `tree_size` bytes long. Then, reconstruct the Huffman tree using `rebuild_tree()`.
   (a) The array containing the dumped tree will be referred to as `tree_dump`. The length of this array will be `nbytes`. A stack of nodes will be needed to reconstruct the tree.
   (b) Iterate over the contents `tree_dump` from 0 to `nbytes`.
   (c) If the element of the array is an `'L'`, then the next element will be the symbol for the leaf node. Use that symbol to create a new node with `node_create()`. Push the created node onto the stack.

(d) If the element of the array is an `'I'`, then you have encountered an interior node. Pop the stack once to get the *right* child of the interior node, then pop again to get the *left* child of the interior node. Note: the pop order *is important*. Join the left and right nodes with `node_join()` and push the joined parent node on the stack.

(e) There will be one node left in the stack after you finish iterating over the contents `tree_dump`. This node is the root of the Huffman tree.

4. Read `infile` one bit at a time using `read_bit()`. You will be traversing down the tree one link at a time for each bit that is read.

   (a) Begin at the root of the Huffman tree. If a bit of value 0 is read, then walk down to the left child of the current node. Else, if a bit of value 1 is read, then walk down to the right child of the current node.

   (b) If you find yourself at a leaf node, then write the leaf node's symbol to `outfile`. Note: you may alternatively buffer these symbols and write out the buffer whenever it is filled (this will be more efficient). After writing the symbol, reset the current node back to the root of the tree.

   (c) Repeat until the number of decoded symbols matches the original file size, which is given by the `file_size` field in the header that was read from `infile`.

5. Close `infile` and `outfile`.

12 Deliverables

You will need to turn in:

1. `encode.c`: This file will contain your implementation of the Huffman encoder.
2. `decode.c`: This file will contain your implementation of the Huffman decoder.
3. `entropy.c`: This file will be provided in the resources repository, but should be included in your repo as well. You *must not* modify this file.
4. `defines.h`: This file will contain the macro definitions used throughout the assignment. You *may not* modify this file.
5. `header.h`: This will will contain the `struct` definition for a file header. You *may not* modify this file.
6. `node.h`: This file will contain the node ADT interface. This file will be provided. You *may not* modify this file.
7. `node.c`: This file will contain your implementation of the node ADT.
8. `pq.h`: This file will contain the priority queue ADT interface. This file will be provided. You *may not* modify this file.

9. `pq.c`: This file will contain your implementation of the priority queue ADT. You *must* define your priority queue `struct` in this file.

10. `code.h`: This file will contain the code ADT interface. This file will be provided. You *may not* modify this file.

11. `code.c`: This file will contain your implementation of the code ADT.

12. `io.h`: This file will contain the I/O module interface. This file will be provided. You *may not* modify this file.

13. `io.c`: This file will contain your implementation of the I/O module.

14. `stack.h`: This file will contain the stack ADT interface. This file will be provided. You *may not* modify this file.

15. `stack.c`: This file will contain your implementation of the stack ADT. You *must* define your stack `struct` in this file.

16. `huffman.h`: This file will contain the Huffman coding module interface. This file will be provided. You *may not* modify this file.

17. `huffman.c`: This file will contain your implementation of the Huffman coding module interface.

18. `Makefile`: This is a file that will allow the grader to type `make` to compile your programs.

    - `CC=clang` must be specified.
    - `CFLAGS=-Wall -Wextra -Werror -Wpedantic` must be included.
    - `make` should build the encoder, the decoder, *and* the supplied entropy-measure program, as should `make all`.
    - `make encode` should build *just* the encoder.
    - `make decode` should build *just* the decoder.
    - `make entropy` should build *just* the supplied entropy-measure program.
    - `make clean` must remove all files that are compiler generated.
    - `make format` should format all your source code, including the header files.

19. Your code must pass `scan-build` *cleanly*. If there are any bugs or errors that are false positives, document them and explain why they are false positives in your `README.md`.

20. `README.md`: This must be in *Markdown*. This must describe how to build and run your program.

21. `DESIGN.pdf`: This *must* be a PDF. The design document should answer the pre-lab questions, describe the purpose of your program, and communicate its overall design with enough detail such that a sufficiently knowledgeable programmer would be able to replicate your implementation. This does not mean copying your entire program in verbatim. You should instead describe how your program works with supporting pseudocode. **C** code is **not** considered pseudocode.

13 Submission

To submit your assignment through `git`, refer to the steps shown in `asgn0` Remember: *add, commit,* and *push*! Your assignment is turned in *only* after you have pushed and submitted the commit ID to Canvas. Your design document is turned in *only* after you have pushed and submitted the commit ID to Canvas. If you forget to push, you have not turned in your assignment and you will get a *zero*. "I forgot to push" is not a valid excuse. It is *highly* recommended to commit and push your changes *often*.

14 Supplemental Readings

> *The more that you read, the more things you will know. The more that you learn, the more places you'll go.*
>
> —Dr. Seuss

- *The C Programming Language* by Kernighan & Ritchie
  - Chapter 8
- *Introduction to Algorithms* by T. Cormen, C. Leiserson, R. Rivest, & C. Stein
  - Chapter 2 §2.1
  - Chapter 6 §6.5
  - Chapter 10 §10.1
  - Chapter 12 §12.1
  - Chapter 16 §16.3

15 Strategy

> *I will, in fact, claim that the difference between a bad programmer and a good one is whether he considers his code or his data structures more important. Bad programmers worry about the code. Good programmers worry about data structures and their relationships.*
>
> —Linus Torvalds

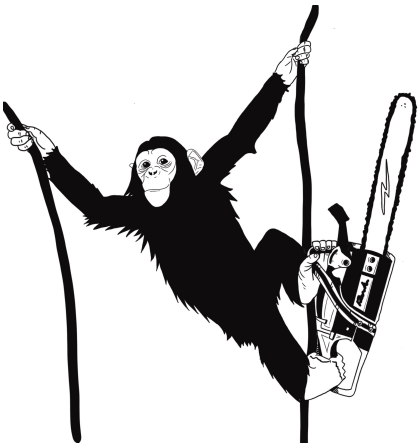
Let's talk strategy.

First, develop the data structures that you will need. Draw pictures, work them out, implement them, and test them. Test them again. A program called `printtree` will be supplied in the resources repository. It will help you determine whether or not you are constructing and dumping your Huffman trees correctly.

Do things in small, incremental steps. Make a histogram. Test it with a small input file. Create a node using a symbol in the histogram. Does that work? Good, now try putting a tree together. Do the same for the rest of the data structures. You'll thank yourself later down the road if you do this.

As with all assignments, a working encoder and decoder will be supplied in the resources repository. Test if you can decode what the reference encoder encodes. Test if the reference decoder and decode what your encoder encodes.

Build your toolkit. Build components. Test them.



```
Mafbalrrpyb py Z ph kpdn bpcpyb l rfydnx l zilpyhlj.
```