Isaac Flores
CruzID: **isgflore**
Prof. Miller
CSE13S
December 4, 2022

Final Design for ASGN 7: A (Huffman Coding) Tree Grows in Santa Cruz

Overview: Huffman coding is a type of data compression. It encodes a file by storing the characters used in a tree that is traversed by a code that is similar to a bit vector. When decoding, the code tells the program which tree nodes to traverse and when it reaches a leaf node it will print the nodes character.

General Idea:

huffman.c: A general-purpose file that conducts procedures needed to build, rebuild, dump, and delete a Huffman tree. Also creates a code for each symbol in a Huffman tree. Functions will be used to encode and decode files.

<u>*build_tree(hist[ALPHABET]):</u> Builds a Huffman tree using a histogram that contains the frequency of each valid symbol in the file to be encoded. Returns the root node of the tree.

<u>build_codes(*root, table[ALPHABET]):</u> Creates a code stack that tracks how to reach each symbol in the Huffman tree. Populates the code table passed in with the code stack data for each respective symbol.

<u>dump_tree(outfile, *root):</u> Does a post-order traversal of a Huffman tree starting at the root. The data is written to outfile. Each leaf node found should be written with an 'L' and then the symbol for it. Each interior node should be labeled with an 'I'.

<u>*rebuild_tree(nbytes, tree_dump[nbytes])</u>: Rebuilds a Huffman tree with the post-order traversal data dumped into the tree dump array. The number of bytes dumped is given by nbytes. Returns the root node of the rebuilt Huffman tree.

<u>delete_tree(**root):</u> Deletes a Huffman tree. Conducts a post-order traversal to free each node starting from the root node passed in. This ensures that each child is deleted before its parent is deleted.

encode.c: Encode will read an input file and create the Huffman encoding of its data. It will use encoding to compress the file into a smaller size. It takes the option -h which prints help and usage information, -i which chooses the input file, -o which chooses the output file, and -v which prints statistics to standard error.

main(); Scans user input to print out help and usage information for -h, compression statistics for -v, as well as setting the infile for -i and the outfile for -o. First, the file computes a histogram of each character used in the infile. It then constructs a Huffman tree using the histogram using a priority queue. It then constructs a code table of how to reach each character in the Huffman tree. It then tree dumps the encoding of a Huffman tree to the outfile using post-order traversal. Then each character in the infile will be scanned and the code to reach the character in the Huffman tree is put into the outfile.

print_help(): Prints a synopsis and usage of the program if prompted by the user.

decode.c: Decode will read a compressed input file and decode or decompress it. It will be returned to its original state before being compressed. It takes the option -h which prints help and usage information, -i which chooses the input file, -o which chooses the output file, and -v which prints statistics to standard error.

main(); Scans user input to print out help and usage information for -h, compression statistics for -v, as well as setting the infile for -i and the outfile for -o. First, the file reads the dumped tree data from the input file. This will be done with a stack that will reconstruct the Huffman tree. Then the rest of the infile will be read which holds the code for the encoded file. The bits will be followed one at a time starting from the root until a leaf node is read. Once a leaf node is reached, its symbol will be put into the outfile. Then it will restart from the root until no more code data can be read.

print_help(): Prints a synopsis and usage of the program if prompted by the user.

pq.c: Priority queue acts like a normal queue but certain nodes hold higher priority to being dequeued than other nodes. This will help to allow characters with lower frequencies to be added to the Huffman tree first.

insert_sort(**list, size): Insert sorts a list of nodes with a size of *size*. Checks each index of the list and if the left frequency is greater than the right frequency the nodes get swapped. Then every node to the left of the swapped node is checked and shifted over if the frequencies are greater than the current node. Used to sort the priority queue.

*pq_create(capacity): Allocates memory for the priority queue and the list of nodes it will need. The maximum capacity for the priority queue will be the given capacity. Returns the priority queue.

pq_delete(**q): Deletes and frees memory for the priority queue and the list of nodes inside of the priority queue. Sets its pointer to NULL afterward.

pq_empty(*q): Returns true if the priority queue is empty or else it returns false.

pq_full(*q): Returns true if the priority queue is full or else it returns false.

pq_size(*q): Returns the number of nodes that are currently in the priority queue.

enqueue(*q, *n): Adds the given node to the priority queue. If the priority queue is already full it will return false. If the node can be added it will return true. Once a node is enqueued the function will insertion sort to sort the frequencies so that characters with the lowest frequencies are at the end of the priority queue.

dequeue(*q, **n): Removes the highest priority node from the priority queue. Returns false if the priority queue was empty before removing a node. Returns true otherwise. The node to be removed will be passed into the double node pointer passed in.

pq_print(*q): A debug function that prints out the contents of the priority queue.

node.c: A node that will be used to create priority queues and stacks to perform Huffman coding. Each node contains a pointer to its left and right child as well as its symbol and frequency of the symbol.

*node_create(symbol, frequency): Creates and allocates space for a node. It sets the node's symbol to the given symbol and its frequency to the given frequency. It returns a pointer to the new node.

node_delete(**n): Deletes and frees memory allocated for the node. Sets the node's pointer to NULL afterward.

*node_join(*left, *right): Joins two nodes under a parent with a symbol of '$' and a frequency of the sum of both the left and right node's frequencies. The parent node's left child will be the left node given and the right child will be the right node given. Returns a pointer to the parent node.

node_cmp(*n, *m): Returns if node n's frequency is greater than node m's frequency

node_print(*n): Prints the symbol and frequency of the given node. Used for debugging.

node_print_sym(*n): Prints the symbol of the given node. Used for debugging.

code.c: A stack of bits that will help to know which nodes to traverse in the Huffman tree. Acts like a map so that we know which characters go in the order they were in originally.

code_init(void): Initializes the top of the code stack to 0 and sets every bit to 0. Returns the code.

code_size(*c): Returns the size of the code stack which is equal to the top of the code stack.

code_empty(*c): Returns if the code stack is empty which means that the top of the code stack is equal to 0.

code_full(*c): Returns if the code stack is full which means that the top of the stack equals to the APLHABET macro made which is 256.

code_set_bit(*c, i): Sets the ith bit in the code stack. If the ith position is greater than or equal to ALPHABET, it returns false. Else, it sets the ith position in the code stack and returns true.

code_clr_bit(*c, i): Clears the ith bit in the code stack. If the ith position is greater than or equal to ALPHABET, it returns false. Else, it clears the ith position in the code stack and returns true.

code_get_bit(*c, i): Returns if the ith bit is set or not. If the ith position is greater than or equal to ALPHABET, it returns false. Else, if the bit at the ith location is set the function returns true, and if not it returns false.

code_push_bit(*c, bit): Pushes the given bit to the code stack. If the stack is full before pushing, the function returns false. Else, the function clears the top of the code stack if the given bit is 0 and sets the top of the code stack if the given bit was 1. Increments the top by 1 and returns true after doing this.

code_pop_bit(*c, *bit): Pops the top of the code stack. If the stack is empty before popping, the function returns false. Else, the function gets the top of the code stack and sets the bit pointer passed in to the bit that was at the top of the stack. Clears the top bit after doing this and then returns true.

code_print(*c): A debug function that prints each of the bytes in the code stack.

io.c: IO handles input and output procedures needed to read data from a file and to write data to a file. Will be used in the encode and decode files. Keeps track of bytes read and bytes written.

read_bytes(infile, *buf, nbytes): Reads in bytes from the infile. The amount of bytes to be read is specified by nbytes. The function will read bytes until nbytes have been read or there are no more bytes to be read. The bytes read will be passed into the *buf buffer that was passed in. The number of total bytes read will be returned.

write_bytes(outfile, *buf, nbytes): Writes bytes from the *buf buffer into the outfile. The amount of bytes to be written is specified by nbytes. The function will write bytes until nbytes have been written or there are no more bytes to be written. The number of total bytes written will be returned.

read_bit(infile, *bit): Reads in a BLOCK or 4096 bytes from infile and reads them bit by bit. A counter of the index left off on which is stored in *bit and a buffer which holds the read bytes is kept since we can not read single bits. The function returns false if there are no more bits to be read and returns true if there are still more bits to be read.

write_code(outfile, *c): Writes a bit from the code stack *c into a buffer which will hold BLOCK bytes. The index of the bit left off on will be tracked as well. Once the buffer is filled the contents will be written into the outfile.

flush_code(outfile): Used to fix any errors that write_code will produce since files won't always have BLOCK amount of bytes in them. The function writes leftover buffered bits to outfile and ensures that any unneeded bits are zeroed out before doing so.

stack.c: A stack of nodes will be used to reconstruct the Huffman tree when decoding. A stack holds items and makes sure that the first item in is the first item out.

*stack_create(capacity): Creates and allocates memory for the stack and the items inside the stack which consists of nodes. The number of nodes inside the stack is specified by the capacity passed in. Sets the top of the stack to 0 and the capacity to the capacity passed in. Returns the stack.

stack_delete(**s): Deletes and free memory allocated for the stack and the node items inside the stack. Sets its pointer to NULL afterward.

stack_empty(*s): Returns true if the stack is empty which means the top of the stack is equal to 0. Returns false otherwise.

stack_full(*s): Returns if the stack is full which means that the top of the stack is equal to the capacity of the stack. Returns false otherwise.

stack_size(*s): Returns the size of the stack which is equal to the top of the stack.

stack_push(*s, *n): Pushes a node onto the stack. Returns false if the stack is full before pushing. Else, it pushes the node passed in to the top of the stack and increments the top by 1. Returns true after.

stack_pop(*s, **n): Pops the node at the top of the stack. Returns false if the stack is empty before popping. Else, it pops the top node and sets it to the node passed in. It then sets the top node to 0 which is how we signal that it is empty. Returns true after.

stack_print(*s): A debug function that prints each node in the stack if it exists.

Makefile: Compiles all the code and allows different targets to run. Make and make all compiles encode and decode as well as all the c files it needs. Make encode only compiles encode and make decode only compiles decode. Make clean removes files that the compiler made except for the executable. Make spotless removes all files that the compiler made. Make format clang formats the c files.

Pseudocode:

huffman.c:

*build_tree(hist[ALPHABET]):

```
# create a priority queue
# add each index in the histogram with a frequency that isn't 0 to a priority queue as a node
# while the priority queue's size is greater than 1
    # dequeue a left node from the priority queue
    # dequeue a right node from the priority queue
    # make a parent node by joining the left and right nodes
    # enqueue the new parent node
# dequeue the last node in the priority queue which is the root
# delete the priority queue
# return the root node
```

```
build_codes(*root, table[ALPHABET]):
# declare a static code variable outside of this function
# if the static code is empty
    # initialize the code
# if the root node isn't NULL
    # if the current node is a leaf node
        # set the table at the current node's symbol's index to the static code
    #  else
        # push a 0 to the static code
        # build a code with the current root node's left child and the given code table
        # pop a bit from the static code
        # push a 1 to the static code
        # build a code with the current root node's right child and the given code table
        # pop a bit from the static code

dump_tree(outfile, *root):
# if the root node isn't NULL
    # dump the tree starting at the current root node's left child
    # dump the tree starting at the current root node's right child
    # if the current root node is a leaf node
        # write an 'L' into the outfile
        # write the current root node's symbol to the outfile
    # else
        # write an 'I' into the outfile

*rebuild_tree(nbytes, tree[nbytes]):
# create a stack
# iterate through tree from 0 to nbytes in increments of 1
    # if the current element is an 'L'
        # create a node with the symbol being the ith + 1 element of tree
        # push the new node to the stack
        # increment i by 1
    # else if the current element is an 'I'
        # pop the stack and set it to a right node
        # pop the stack and set it to a left node
        # join the left and right nodes and it to a parent node
        # push the parent node to the stack
# pop the last node in the stack and set it to a root node
# delete the stack
# return the root node
```

delete_tree(**root):

```
# if the root node is not NULL
        # delete the tree starting from the current root node's left child
        # delete the tree starting from the current root node's left child
        # delete the current root node
```

encode.c:

main():

```
# take the options from the user
        # -h enable help and usage information for the program. Exits the program after.
        # -i sets the infile to the given location. The default is stdin.
        # -o sets the outfile to the given location. The default is stdout.
        # -v enables the printing of compression statistics. The default is off.
# open the input file with reading permissions
# create a histogram of ALPHABET size or 256
# while reading in BLOCK bytes from infile doesn't return 0
        # iterate through the number of bytes actually read
                # increment the respective index in the histogram for each time a symbol occured
# if the first or second indices of the histogram are 0
        # set their values to 1
# build a Huffman tree with the computed histogram and get the returned root node
# create a code table of APLHABET size with each index being a code
# build a code table starting from the root of the Huffman tree
# create a header stucture
# set the header's magic number to MAGIC or 0xBEEFBBAD
# get the stats from the infile
# get the permissions of the infile and set it to the header's permissions
# get the size of the infile and set it to the header's file size
# collect the number of histogram values that are greater than 0
# set the header's tree size to (3 * unique histogram values) - 1
# set the outfile file descriptor to stdout's file descriptor if that is what the user wants, else create
and open the given location of the outfile with write permissions
# match the permissions of the infile with the outfile
# cast the header data to an array of bytes
# write the header data to the outfile
# dump the Huffman tree to the outfile
# reset the infile so that it can be read from the beginning
# while reading in BLOCK bytes from infile into a buffer doesn't return 0
```

```
    # iterate through the number of bytes actually read
        # write the respective symbols code from the code table to the outfile
# flush any remaining code
# if the -v option was enabled
    # print the uncompressed file size, the compressed file size, and the space saving which is equal to 100 * (1 - (compressed size/uncompressed size))
# delete the Huffman tree and free the histogram
# close the infile and outfile
```

<u>print_help():</u>

```
# print any needed usage and help information for encode to the standard error
```

decode.c:

<u>main():</u>

```
# take the options from the user
    # -h enable help and usage information for the program. Exits the program after.
    # -i sets the infile to the given location. The default is stdin.
    # -o sets the outfile to the given location. The default is stdout.
    # -v enables the printing of decompression statistics. The default is off.
# if the user wants the infile to be standard input
    # set an infile temporary location to standard inputs file descriptor
    # set the main infile a created and opened temporary file with read and write permissions
    # create a temporary buffer that holds 1 byte
    # create another write buffer that holds BLOCK bytes
    # set the write buffer's index to 0
    # while reading 1 byte from the standard input to the temporary buffer does not return 0
        # set the write buffer's current index to the first index of the temporary buffer
        # increment the write buffer's index by 1
        # if the write buffer's index equals to BLOCK
            # write BLOCK bytes from the write buffer to the infile
            # reset the write buffer index to 0
    # if the write buffer's index is not 0
        # write write buffer index bytes from the write buffer to the infile
    # reset the infile so that it can be read from the beginning
    # set the external bytes read to 0
# else
    # open the given location of the infile with reading permissions
# read in the header data from the infile to a byte array
# get the magic number from infile by setting a magic number variable to the first four elements of the byte array. Left shift each byte respectively so that they each fill a byte of space.
# if the magic number that eas read does not match the defined MAGIC number
    # print an error message and exit
# get the permissions for infile by setting a permissions variable to the next two elements of the byte array. Left shift each byte respectively so that they each fill a byte of space.
# if the user wants the output to be to standard output
    # set the outfile to the file descriptor for standard output
# else
    # create and open the given outfile location with writing permissions
# set the permissions for the outfile to the permissions variable read
# get the dumped tree size from infile by setting a tree size variable to the next two elements of the byte array. Left shift each byte respectively so that they each fill a byte of space.
# get the original file size from infile by setting a file size variable to the last eight elements of the byte array. Left shift each byte respectively so that they each fill a byte of space.
# create a byte array with the size being the read dumped tree size
# read the dumped tree data from the infile into the previously made array
# rebuild the Huffman tree with the size being the read tree size and get the returned root
# set a current node to the returned root node
# create a byte array that is BLOCK long
# set the index of the byte array and current size of the outfile to 0
# while there are still bits to read from the infile
    # if the read bit is 0
        # set the current node to the current node’s left child
    # if the read bit is 1
        # set the current node to the current node’s right child
    # if the current node is a leaf node
        # put it’s symbol to the byte array at the location of the bytes arrays current index
        # increment the byte arrays current index and current file size by 1
        # reset the current node to the root
    # if the bytes arrays current index is equal to BLOCK
        # write BLOCK bytes from the byte array to the outfile
        # reset the byte arrays current index to 0
    # if the current file size equals to the read file size
        # break
# if the byte arrays current index doesn’t equal 0
    # write current index bytes from the byte array to the outfile
# if the -v option was enabled
    # print the compressed file size, the decompressed file size, and the space saving which is equal to 100 * (1 - (compressed size/decompressed size))
# free the Huffman tree array
# delete the Huffman tree
# close the infile and outfile
```

<u>print_help():</u>

```
# print any needed usage and help information for decode to the standard error
```

pq.c:

<u>struct:</u>

```
# initialize a load, the capacity, and the queue that consists of nodes
```

<u>insert_sort(**list, size):</u>

```
# for i from 0 to size-1 in increments of 1
    # for j from i+1 to size in increments of 1
        # set a temporary variable to j
        # while the node to the left of the temporary location is greater than the node at
the temporary location
            # swap the both nodes
            # decrement the temporary variable by 1
            # if temp equals to 0
                # break
```

<u>*pq_create(capacity):</u>

```
# allocate memory for the priority queue
# if the priority queue was successfully allocated
    # set the load to 0
    # set the capacity to the given capacity
    # allocate memory for the queue of nodes
# if the queue of nodes wasn't successfully allocated
    # free the priority queue and set its pointer to NULL
# return the priority queue
```

<u>pq_delete(**q):</u>

```
# if the priority queue exists
    # free the queue of nodes
    # free the priority queue and set its pointer to NULL
```

<u>pq_empty(*q):</u>

```
# return if the priority queue's load is equal to 0
```

pq_full(*q):
```
# return if the priority queue’s load is equal to the priority queue’s capacity
```

pq_size(*q):
```
# return if the priority queue’s load
```

enqueue(*q, *n):
```
# if the priority queue is full
    # return false
# set the node at the load index of the queue to the node n passed in
# increment the load by 1
# insertion sort the queue with the size being the load
# return true
```

dequeue(*q, **n):
```
# if the priority queue is empty
    # return false
# set the node pointer passed in to the node at the 1st index of the queue
# decrement the load by 1
# shift each node in the queue to the left by 1
# return true
```

pq_print(*q):
```
# for i from 0 to the load by increments of 1
    # print the node at the ith index of the queue
```

node.c:

struct:
```
# initialize the left and right child nodes
# initialize the symbol and frequency of the node
```

*node_create(symbol, frequency):
```
# allocate memory for the node
# if successfully allocate
    # set the nodes symbol to the given symbol
    # set the nodes frequency to the given frequency
    # set the left and right nodes to NULL
# return a pointer to the new node
```

node_delete(**n):
```
# if the node exists
    # free the memory for the node and set it to NULL
```

node_cmp(*n, *m):
```
# return if node n’s frequency is greater than node m’s frequency
```

*node_join(*left, *right):
```
# create a parent node with the symbol of ‘$’ and a frequency of the left node’s frequency plus the right node’s frequency
# set the parent node’s left child to the left node given and the right child to the right node given
# return a pointer to the parent node
```

node_print(*n):
```
# print the symbol and frequency of the given node
```

node_print_sym(*n):
```
# print the symbol of the given node
```

code.c:

struct:
```
# initializes the top of the code stack and the byte array which is MAX_CODE_SIZE long
```

code_init(void):
```
# create a new code and set its top to 0.
# for each byte in the byte array
    # set its initial value to 0.
# return the new code
```

code_size(*c):
```
# return the top of the code stack
```

code_empty(*c):
```
# if the top of the code stack is not equal to 0
    # return false
# return true
```

code_full(*c):
```
# return if the top of the code stack is equal to APLHABET or 256
```

```
code_set_bit(*c, i):
# if i is greater than or equal to APLHABET
    # return false
# set the location of the bit to i divided by 8
# set the position of the bit to i modulo 8
# create a bit variable that is equal to 1 left-shifted position times
# set the code stack's location byte to itself bitwise or the bit variable created
# return true

code_clr_bit(*c, i):
# if i is greater than or equal to APLHABET
    # return false
# set the location of the bit to i divided by 8
# set the position of the bit to i modulo 8
# create a bit variable that is equal to 1 left-shifted position times
# set the code stack's location byte to itself bitwise and the inverse of the bit variable created
# return true

code_get_bit(*c, i):
# if i is greater than or equal to APLHABET
    # return false
# set the location of the bit to i divided by 8
# set the position of the bit to i modulo 8
# create a bit variable that is equal to 1 left-shifted position times
# create a variable that will hold the bit value of the ith bit. It will equal to the code stack's
location byte bitwise and the bit variable created.
# return true if the variable that holds the bit value of the ith bit is not equal to 0 and false if it is
equal to 0.

code_push_bit(*c, bit):
# if the code stack is full
    # return false
# if the bit passed in is 1
    # set the top bit
#else
    # clear the top bit
# increment the top by 1
# return true
```

```
code_pop_bit(*c, *bit):
# if the code stack is empty
    # return false
# if the top is not equal to 0
    # decrement top by 1
# if the bit at the top position is 1
    # set the bit pointer passed in to 1
# else
    # set the bit pointer passed in to 0
# clear the top bit
# return true

code_print(*c):
# for each byte in the code stack
    # print its value
```

io.c:

```
global variables:
# initialize variables for the total amount of bytes read and written to 0
# initialize a bit buffer and a write buffer of BLOCK size with each index being a byte
# initialize a bit index and write index to 0
# initialize a size variable

read_bytes(infile, *buf, nbytes):
# while the total read bytes is less than nbytes
    # read nbytes from infile and pass it into buf. Set a variable to the number of bytes actually read.
    # if the number of bytes actually read was 0
        # break
    # add the number of bytes actually read to the total bytes read
# add the number of total bytes read to the global bytes read variable
# return the number of total bytes read

write_bytes(outfile, *buf, nbytes):
# while the total written bytes is less than nbytes
    # write nbytes from *buf into outfile. Set a variable to the number of bytes actually written.
    # if the number of bytes actually written was 0
        # break
        # add the number of bytes actually written to the total bytes written
# add the number of total bytes written to the global bytes written variable
# return the number of total bytes written

read_bit(infile, *bit):
# if the global bit index equals 0
        # read BLOCK bytes from infile into the global bit buffer and set the global size to the number of bytes actually read. If the number of bytes actually read is 0.
                # set the global bit index to 0
                # return false
# if size does not equal BLOCK and the global bit index equals to size times 8
        # set the bit index to 0
        # return false
# if the bit at the (global bit index % 8) bit of the (global bit index / 8) bytes of the global bit buffer does not equal 0
        # set the bit pointer passed in to 1
# else
        # set the bit pointer passed in to 1
# if the global bit index equals to BLOCK * 8 - 1
        # set the global bit index to 0
# else
        # increment the global bit index by 1
# return true

write_code(outfile, *c):
# set a size variable to the size of the code c
# set a current index variable to 0
# while the current index is less than the size of the code
        # get the current index bit of code c and if its a 1
                # set the (global write index % 8) bit of the (global write index / 8) byte in the global write buffer
        # else
                # clear the (global write index % 8) bit of the (global write index / 8) byte in the global write buffer
        # increment the global write index and current index by 1
        # if the global write index equals to BLOCK * 8
                # write BLOCK bytes from the global write buffer to outfile
                # set the global write index to 0

flush_code(outfile):
```

```
# for i equals the global write index to BLOCK * 8 in increments of 1
    # clear the (global write index % 8) bit of the (global write index / 8) byte in the global write buffer
# set the global write index to itself divided by 8 if itself divided by 8 has no remainder, else set it to itself divided by 8 plus 1
# write global write index bytes from the global write buffer into outfile
```

stack.c:

struct:

```
# initialize the top, capacity, and items.
```

*stack_create(capacity):

```
# allocate memory for the stack
# if successfully allocated
    # set the stack's top to 0
    # set the stack's capacity to the given capacity
    # allocate capacity amount of nodes into the items
# if items weren't successfully allocated
    # free the stack and set it to NULL
# return a pointer to the stack
```

stack_delete(**s):

```
# if the stack exists
    # free the node items
    # free the stack and set its pointer to NULL
```

stack_empty(*s):

```
# if the stack's top does not equal 0
    # return false
# return true
```

stack_full(*s):

```
# if the stack's top is equal to the stack's capacity
    # return true
# return false
```

stack_size(*s):

```
# return the top of the stack
```

```
stack_push(*s, *n):
# if the stack is full
    # return false
# set the top of the stack to n
# increment the top by 1
# return true

stack_pop(*s, **n):
# if the stack is empty
    # return false
# decrement the top of the stack by 1
# set the node pointer passed in to the top of the stack
# set the top of the stack to 0
# return true

stack_print(*s):
# for each node in items
    # if the current node isn’t equal to 0
        # print the current node
```

Makefile:

```
# the compiler is clang and the CFLAGS are -Wall -Werror -Wextra -Wpedantic
# phony targets are all, clean, spotless, and format

# target all depends on encode and decode
# target encode depends on *.o
    # compile encode with clang and with each dependency file
# target decode depends on *.o
    # compile decode with clang and with each dependency file
# target %.o depends on %.c
    # compile the given c file with clang and with the CFLAGS
# target clean
    # remove all .o files
# target spotless
    # remove all .o files and the encode and decode executables
# target format
    # clang format each .c file
```