Mika Peer Shalem
Asgn7: Final Design Doc
12/04/2022

Asgn 7: Design Doc

General Overview

The program replicates the process of Huffman coding. It is the most optimal way to compress data. It gives each character in a text a symbol, where characters with higher frequencies have smaller symbols. Thus, the least number of bits is going to the most frequent character/symbol. The assignment includes both an encoder (which compresses the data), and a decoder (which displays the original message). It implements a stack module, a priority queue, a code to assign bits for each symbol, an I/O module that handles files, and a Huffman file that creates the tree.

encode.c

Goal: Read in an input file, find the Huffman encoding of its contents, and use the encoding to compress the file.

Create a simple function that prints the error message to stderr when called.

Set the default value of the variable *input_file* to stdin
Set the default value of the variable *output_file* to stdout
Set the default value of the variable stats to 0/false
Set flags to check if different files were given

While getopt argument is not -1 (getopt goes through all the given options and returns -1 when it runs out of options):

- In case the argument is equal to h:
  - Displays the help message using the print error function
  - Exit with 0
- In case the argument is equal to i:
  - Set *input file* to the user input
  - Set the flag to true
- In case the argument is equal to o:
  - Set *output file* to the user input
  - Set the flag to true
- In case the argument is equal to v:
  - Enables printing statistics to stderr (set the variable stats to 1/true)
- Default case:
  - Displays the help message using the function

Exit with a nonzero number

// Handle files

If the user gave a different input file (using the -i option):

Open that file

Else:

Create a temporary file

Copy everything from stdin to the temporary file

Go back to the beginning of the file

If the user gave a different output file (using the -o option):

Open that file

Print error messages if the files could not be open

// encoding algorithm

Creates a histogram that holds a character and its frequency in the text (based on the infile).

- Create an empty array using the defined value ALPHABET (number of ASCII characters)
- Set all values of the array to 0
- Read the input file byte by byte, and increase the frequency of each character that was read

Make sure that at least two symbols have non-zero counts:

- If the first index of the histogram is 0:
    - Set it to 1
- If the second index of the histogram is 0:
    - Set it to 1

Creates a Huffman tree using the histogram (priority queue)

- Call the function build tree with the histogram

Creates a code table (using build_codes())

- Create an empty array using the defined value ALPHABET (number of ASCII characters)
- Make sure all values in the array are intialized by traversing it
- Call build codes

// Header

Creates a header and send it to outfile

- Set the magic member to the defined value MAGIC
- Creates a fstat structure to get the file size and get the permission
- Iterate through the histogram, and whenever finding a character with a frequency greater than 0, increase the tree size
- After traversing the histogram, set the tree size to the tree size * 3 + 1
- Bit cast the header to an array where every element is the size of a byte
- Use the write_bytes function that will be described below to send it to the output file

Find the constructed tree using the dump_tree function that will be described below

Go back to the beginning of the input file

Go through all characters in the input file:

Write the code of each character

Use the flush_codes function described below

// Statistics

If the statistics flag is true/1:

Prints the uncompressed file size, the compressed file size, and space-saving

The assignment pdf describes space saving as:

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

Close infile and outfile

decode.c

Goal: read in a compressed input file and decompress it, expanding it back to its original, uncompressed size.

Create a simple function that prints the error message to stderr when called.

Set the default value of the variable *input_file* to stdin
Set the default value of the variable *output_file* to stdout
Set the default value of the variable stats to 0/false
Set flags to check if a different files were given

While getopt argument is not -1 (getopt goes through all the given options and returns -1 when it runs out of options):

- If the argument is equal to h:
  - Displays help message using the function
  - Exit with 0
- If the argument is equal to i:
  - Set *input file* to the user input
  - Set the flag to true
- If the argument is equal to o:
  - Set *output file* to the user input
  - Set the flag to true
- If the argument is equal to v:
  - Enables printing statistics to stderr (set the stats variable to 1/true)
- Else
  - Displays help message using the function
  - Exit with a nonzero number

// Handle files

If the user gave a different input file (using the -i option):

- Open that file

If the user gave a different output file (using the -o option):

- Open that file

Print error messages if the files could not be open

// Header

Read from the input file the encoded bytes of the header

Cast it into a header

Make sure that the magic member of the header is equal to the magic variable defined in defines.h. If the magic number don't match, then display an error message and quit.
Set the permission of the output file based on the header member.

// Decoding algorithm
Reads the dumped tree from the input file and reconstructs it using a stack of nodes

- Create a new array with the size of the tree size member of the header
- Set every element in the array to 0
- Read from the input file the constructed tree
- Rebuild the tree using the function described below

Reads bits from the input and deciphering the original message
Set the current node to the root of the tree
While the number of decoded symbols are less than the file size:
    If at leaf:
        Increase the number of decoded symbols
        Write the symbol of the current node to the outfile
        Set the current node to the root
    Read a bit from the input file
    If the bit is equal to 0:
        Set the current node to the left child of the current node
    If it's equal to 1:
        Set the current node to the right child of the current node

// Statistics
Prints the uncompressed file size, the compressed file size, and space-saving

The assignment pdf describes space saving as: $100 \times (1 - (\text{compressed size} / \text{uncompressed size}))$.

Close infile and outfile

node.c

Goal: each node containing a pointer to its left child, a pointer to its right child, a symbol, and the frequency of that symbol.

Node *node_create(uint8_t symbol, uint64_t frequency)

Goal: the constructor for a node

Allocate memory for a node using the size of the node structure and malloc since its values do not matter

If the allocation of memory was successful:

- Set the symbol member of the new node to the symbol argument
- Set the frequency member of the new node to frequency argument
- Set the value of the left and right members to NULL

void node_delete(Node **n)

Goal: the destructor for a node

If the pointer to n exists:

- Free the node n
- Set the pointer to n to NULL

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

Goal: joining a left child node and a right child node, returning a pointer to the created parent node

If the left and right node exist:
Call node_create with the following values:

- Symbol is $
- Frequency is the frequency of left + frequency of right
- Left child is the argument left
- Right child is the argument right

Else:

- Return NULL

void node_print(Node *n)

Goal: a debug function that prints all characteristics of the node

Use the C function iscntrl() to check if it's printable (returns 0 is it's not control / is printable)
Prints the symbol and frequency

void node_print_sym(Node *n)
Goal: prints the symbol associated with a node

Use the C function iscntrl() to check if it's printable (returns 0 is it's not control / is printable)
Prints the symbol of the node

bool node_cmp(Node *n, Node *m)
Goal: compares the frequency of two nodes, returns true if n>m, false otherwise

If the frequency of node n is greater than the frequency of node m:
    Return true
Return false

pq.c

Goal: a queue where each element has a priority, so elements with high priority will dequeue faster

PriorityQueue Structure
Create a structure with the members capacity & size.
Have an array of items that represent all nodes in the queue.

PriorityQueue *pq_create(uint32_t capacity)
Goal: constructor for a priority queue.

Allocate memory for the priority queue using malloc and the size of a PriorityQueue structure.
Malloc either returns NULL if the memory cannot be allocated or a pointer to the allocated PriorityQueue
If the pointer is not NULL:
    Set the capacity member to the capacity argument
    Set size to 0
    Allocates memory for all possible items using the capacity member and the size of a node. (If the memory wasn't allocated, free the queue)
Return pointer to PriorityQueue

void pq_delete(PriorityQueue **q)
Goal: destructor for a priority queue.

If the queue exists:
    Go through every node in the queue and delete it
    Free the items array
    Free it and set its pointer to NULL

bool pq_empty(PriorityQueue *q)
Goal: returns true if the priority queue is empty, false otherwise

If the size member of the queue is 0:
    Return true
Return false

bool pq_full(PriorityQueue *q)
Goal: returns true if the priority queue is full, false otherwise

If the size member of the queue is equal to the capacity:
    Return true
Return false

uint32_t pq_size(PriorityQueue *q)
Goal: returns the number of items in the priority queue

Returns the size member

bool enqueue(PriorityQueue *q, Node *n)
Goal: inserts a node into the priority queue, return true to indicate success and false otherwise

Using an insertion sort:
Whenever a new item is inserted into the queue, sort the items array.
In my implementation, I will put the item with the highest frequency at the end of the queue.

If the queue and node exist, and the queue is not full:
    If the queue is empty:
        Set the top location to the argument node
        Increase the size by 1
        Return true
    Else:

Set the current location of the queue to the argument node
Go through all elements of the queue (using index i):
If queue[i] < queue[i+1]:
Swap them
Go through all elements to the left of the chosen element:
(variable j starts with i and decreases until it reaches 0):
If queue[j] > queue[j-1]:
Swap them
Else:
Stop the inner loop
Increase the size by 1
Return true
Return false

Ex. 9, 5, 1, 6, 14, 8
Compare 9 and 5:
Since 9>5, swap them
No elements to the left
Current array: 5, 9, 1, 14, 8

Compare 9 and 1:
Since 9>1, swap them
Compare 1 and 5.
Since 1<5, swap them
No elements to the left
Current array: 1, 5, 9, 14, 8

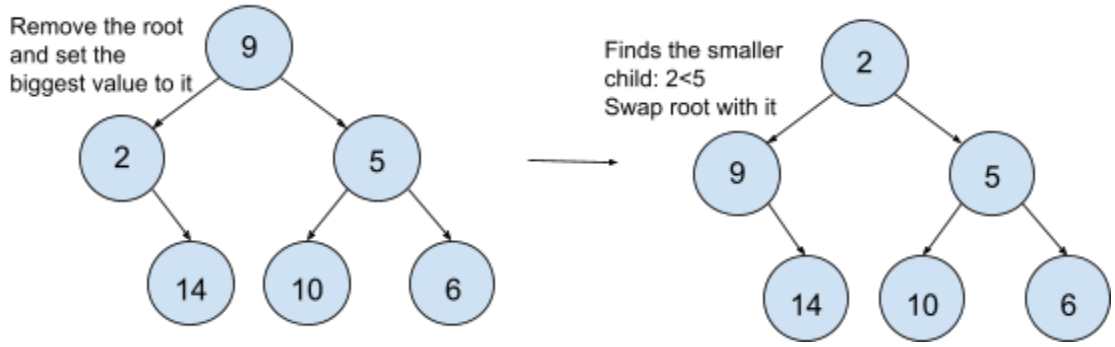
Compare 9 and 14:
Since 9<14, do nothing
Current array: 1, 5, 9, 14, 8

Compare 14 and 10:
Since 14>8, swap them
Compare 8 and 9.
Since 8<9, swap them
Compare 8 and 5
Since 8>5, do nothing
Current array: 1, 5, 8, 9, 14

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

Goal: dequeues a node from the priority queue, return true to indicate success and false otherwise

If the size of the queue is greater than 0:
    Set n to the node at the biggest location of the priority queue
    Set size to size-1
    Return true
Return false



void pq_print(PriorityQueue *q)
Goal: a debug function to print priority queue

Prints the size and capacity of the queue
Go through all nodes in the priority queue:
    Print the node

code.c

Goal: a stack of bits that create a unique code for each symbol
Structure (given on pdf):

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

Code code_init(void)
Goal: construct a new Code on the stack

Create a Code
Set the top member of code to 0
Go through all elements in the bit array and set them to 0

Return code

uint32_t code_size(Code *c)
Goal: returns the size of Code

The top of a stack keeps track of the number of elements in the stack. When a new element is pushed into the stack, the top moves up to point at it. Thus, the top member represents the size of the stack

Return the top member of the stack

bool code_empty(Code *c)
Goal: returns true of the code is empty, false otherwise

If the top member of Code is 0: // no items were added to the stack
    Return true
Else:
    Return false

bool code_full(Code *c)
Goal: return true if Code is full, false otherwise

Since Code gives a unique value to each character, it is full once it converts all ASCII characters. The number of possible ASCII characters to use in the program is defined in the macro ALPHABET

If the top member of Code is equal to the macro ALPHABET-1:
    Return true
Else:
    Return false

bool code_set_bit(Code *c, uint32_t i)
Goal: sets the bit at index i, return true to indicate success and false otherwise

If c is not NULL: // ensuring the Code exists
    Locate the byte by doing i / 8
    // Checks that the index is in range:
    If byte is greater than the macro MAX_CODE_SIZE:

Return false

Locate the bit by doing i % 8

// Checks that the index is in range:

If bit is greater than ALPHABET:

Return false

Access the location of the vector by doing masking and shifting: since we want to set the value, the mask is 1. Use OR (|) because if the current value is 0, it will switch to 1 (0|1=1, 1|1=1) Use shifting to get the right bit inside of the byte.

Return true

Return false

bool code_clr_bit(Code *c, uint32_t i)

Goal: clears the bit at index i, return true to indicate success and false otherwise

If c is not NULL: // ensuring the Code exists

Locate the byte by doing i / 8

// Checks that the index is in range:

If byte is greater than the macro MAX_CODE_SIZE:

Return false

Locate the bit by doing i % 8

// Checks that the index is in range:

If bit is greater than ALPHABET:

Return false

Use AND (&) because 0&1=0 and 0&0=0, so the chosen bit will become 0. Shift the bit 1, bit amount of places, and then get its counter value (0). Use shifting to get the right bit inside of the byte.

Return true

Return false

bool code_get_bit(Code *c, uint32_t i)

Goal: gets the bits at index i, return true to indicate success and false otherwise

If c is not NULL: // ensuring the Code exists

Locate the byte by doing i / 8

// Checks that the index is in range:

If byte is greater than the macro MAX_CODE_SIZE:

Return false

Locate the bit by doing i % 8

// Checks that the index is in range:

If bit is greater than ALPHABET:

Return false

Use shifting to get the right bit. Use &1UL because if the value is 1, it will just return 1. Else, it will return 0.

Return true

Return false

bool code_push_bit(Code *c, uint8_t bit)

Goal: push a bit onto Code. Since we are using a stack, the bit will be added to the top

Check if Code is full using the function we created
If it's not full:

If the bit is 1:

Use the set bit function to set the top value to 1

If the bit is 0:

Use the clear bit function to set the top value to 0

Increase the value of the top member by 1

Return true

Return false

bool code_pop_bit(Code *c, uint8_t *bit)

Goal: pops a bit off Code

Check if Code is empty using the function we created
If it's not empty:

Use code_get_bit that we created to get the value at the top of the stack

Decrease the value of the top member by 1

Set the pointer argument bit to the value we got with code_get_bit

Return true

Return false

void code_print(Code *c)
Goal: a debug function that tests the function for Code
Iterate through all bits of c:
    Print the value of each bit using code_get_bit

io.c

Goal: will be used by the encoder and decoder to perform actions on files

int read_bytes(int infile, uint8_t *buf, int nbytes)
Goal: read all specified bytes from a file to the buffer buf.

The function read() returns 0 when it's the end of the file, so can use that as the condition for the loop. It advances the file pointer every time it's being called.

While it's not the end of infile OR we reached nbytes:
    Call read with infile, the buf, and 1 byte
    Increase the variable bytes_read by 1
Return the variable bytes_read

Ex. The file has 4 bytes: b1, b2, b3, b4
The function would call read() 4 times, each time saving a byte at the corresponding index in the buf array. The index to put the value in will be the variable bytes_read since it increases with every function call.
Since each slot of the array has the size of uint8_t, it can hold exactly one byte.
At the end, buf = {b1, b2, b3, b4}.

int write_bytes(int outfile, uint8_t *buf, int nbytes)
Goal: write all specified bytes from the buffer buf into outfile.

The function write() returns 0 when it's the end of the file, so can use that as the condition for the loop. It advances the file pointer every time it's being called.

While it's not the end of buf OR we reached nbytes:
    Call read with outfile, the buf, and 1 byte
    Increase the variable bytes_written by 1
Return the variable bytes_written

bool read_bit(int infile, uint8_t *bit)

Goal: read a block of bytes into a static buffer variable, and return one bit at a time to the bit argument.

Create a static buffer array with the size of a block.
Create a static index that starts with 0. It needs to be a static/global variable so we can know at which index we left off after the function call ends.

If the index is equal to the block size: (all bits in the buffer have been doled out)
    Use our read_bytes function to fill out the block array (get a new block of characters)
    If the return value of the read_bytes is 0:
        Return false
    Reset the bit and byte index to 0 (start to read the block from 0 again)
Set the bit pointer argument to the bit of block array at the specific index using bit masking and shifting (byte and bit variable used here)
If the current bit is greater than 7:
    Set the bit varaible to 0
    Increase the byte variable by 1
Return true

Used in write code:
Create a static buffer array with the size of a block.
Create a static index that starts with 0. It needs to be a static/global variable so we can know at which index we left off after the function call ends.

void write_code(int outfile, Code *c)

Goal: write the contents of the Code to the outfile, one bit at a time.

While the code index is less than the size of the code:
    Use the function code_get_bit, call the function with the code index.
    If the bit is 1:
        Set the current location of the buffer to 1 using bit masking and shifting of the bit index
    If the bit is 0:
        Clears the current location of the buffer using the bit index
    Increase both the bit and current the index
If the index bit is greater than block size times 8:
    Use the write_bytes function to write the contents of the buffer to outfile.
    Reset the index

void flush_codes(int outfile)

Goal: write out any bits that are left over in the buffer after calling the write_code function.

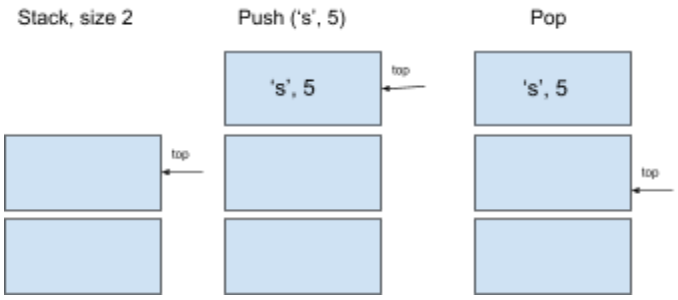
If the index is less than the size of the block and greater than 0:
    If the current index bit is not divisible by 8:
        Clears the current bit of the index
        Increase the index bit by 1
Use the write_bytes function to write the contents of the buffer to outfile, using the index bit divided by 8 as the size.
Reset the index

stack.c

Goal: reconstruct a Huffman tree using a stack of nodes

The stack will be made as a list, where each node contains a value and can point to the previous and next node.

Visual Representation:



Stack structure (given on assignment pdf):

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

Top - signals the location of the top of the stack
Capacity - number of nodes that can be added to the stack
Items - an array of nodes that holds all the nodes we add to the stack

Stack *stack_create(uint32_t capacity)

Goal: The constructor for the stack. Set the member capacity to the argument.

Allocate memory for a stack using the size of the stack structure and malloc since its values do not matter

If the allocation of memory was successful:

    Set the capacity member of the new stack to the capacity argument

Set the top member to 0

Allocate space for the item list. If the memory wasn't allocated, free the stack pointer

Return the new stack

void stack_delete(Stack **s)
Goal: the destructor for the stack

If the stack argument s exists:
Free all nodes in the items array
Free the items array and set the pointer to NULL
Free it and set the pointer to NULL

bool stack_empty(Stack *s)
Goal: returns true if the stack is empty, false otherwise

Same concept as the code_empty function, the top of the stack increases as more elements are added to the stack. So, if it's zero the stack is empty.
If the top is equal to 0:
Return true
Return false

bool stack_full(Stack *s)
Goal: returns true of stack is full, false otherwise

The capacity member has the number of nodes/elements that can be added to the stack, so the top member needs to be less than it
If the top member is equal to the capacity member-1:
Return true
Return false

uint32_t stack_size(Stack *s)
Goal: returns the number of nodes in the stack

Return the top member

bool stack_push(Stack *s, Node *n)
Goal: pushes a node onto the stack, return true to indicate success and false otherwise

Check if the stack is full using the function we created
If it's not full:
    Set the the items array at index top to n
    Increase the value of the top member by 1
    Return true
Return false

bool stack_pop(Stack *s, Node **n)
Goal: pops a node off the stack, return true to indicate success and false otherwise

Cannot delete the node at the top of the stack because then the pointer will point at a non existing node. Instead, just decrease the top member.

Check if the stack is empty using the function we created
If it's not empty:
    Get the value at the top of the stack by taking the value of the items array at the top.
    Set the pointer argument bit to the node we got
    Decrease the value of the top member by 1
    Return true
Return false

void stack_print(Stack *s)
Goal: a debug function that prints all characteristics of the stack
Print the top and capacity members
Iterate through the items array:
    Print each node using the node_print function

huffman.c

Goal: an interface for the provided Huffman coding module

Node *build_tree(uint64_t hist[static ALPHABET])
Goal: creates a tree given a histogram (using the priority queue)

Pseudo code is also given on the assignment pdf:

```
def construct(q):
    while len(q) > 1:
        left = dequeue(q)
        right = dequeue(q)
        parent = join(left, right)
        enqueue(q, parent)
    root = dequeue(q)
    return root
```

Step 1: populate the histogram
Iterate through all characters in the histogram:
    If the value at the index is greater than or equal to 1:
        Create a node with the index, frequency, and NULL left and right children
        Enqueue each node in the priority queue

Step 2: build a tree based on the priority queue
While there are more than two nodes in the queue:
    Dequeue two nodes from the queue and join them
    Add the joined node and the two dequeued nodes to the tree array and sort it
Return the root of the tree

Ex. we have the priority queue: 1, 1, 2, 2, 4, 4
Dequeue two nodes and and join them:
Tree: 1, 1, i2
Queue: i2, 2, 2, 4, 4
Dequeue the next two nodes from the queue:
Tree: 1, 1, i2, 2, i4
Queue: 2, 4, 4, i4
Dequeue the next two nodes from the queue:
Tree: 1, 1, i2, 2, 2, 4, i4, i6
Queue: 4, i4, i6
Dequeue the next two nodes from the queue:
Tree: 1, 1, i2, 2, 2, i4, 4, i6, i8
Queue: i6, i8
Dequeue the next two nodes from the queue:
Tree: 1, 1, i2, 2, 2, i4, 4, 4, i6, i8, i14
Queue: i14
Since we only have one node left, we stop

void build_codes(Node *root, Code table[static ALPHABET])
Goal: creates a code table based on the Huffman tree we built

During class, we got this code which can be useful for the traversal:

```
postorder_print(Node *root) {
    if (root) {
        postorder_print(root->left)
        postorder_print(root->right)
        printf("%d\n", root>key);
    }
}
```

We also got this pseudo code on the assignment pdf:

```
Code c = code_init()

def build(node, table):
    if node is not None:
        if not node.left and not node.right:
            table[node.symbol] = c
        else:
            push_bit(c, 0)
            build(node.left, table)
            pop_bit(c)

            push_bit(c, 1)
            build(node.right, table)
            pop_bit(c)
```

Create a static code on the stack outside of the function

- Use static so the values will stay during the recursion
- Outside of the function so the scope will be the entire file

If the root exists:

If both of the root's children exist:

Push the bit 0 to the code

Call the build_code function again with the left child fo the root root

Pop a bit

Push the bit 1 to the code

Call the build_code function again with the right child of the root

Pop a bit

Else:

Set the table at that index to the static code variable

void dump_tree(int outfile, Node *root)

Goal: create a string representation of the tree and write it to outfile.

Pseudo code is also given on the assignment pdf:

```
def dump(outfile, root):
    if root:
        dump(outfile, root.left)
        dump(outfile, root.right)

        if not root.left and not root.right:
            # Leaf node.
            write('L')
            write(node.symbol)
        else:
            # Interior node.
            write('I')
```

If the root exists:

Call the dump_tree function with the left child of the root

Call the dump_tree function with the right child of the root

If both the left and right children of the root don't exist:

Add the letter L and the symbol of the node

Else (we are at an interior node):

Add the letter I

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

Goal: reconstructs a Huffman tree given its tree dump

Create a new stack.

Go through the array tree_dump until we only have one node left at the tree_dump:

If the current value of the tree argument is the letter L:

Create a node with the value of the next location of the tree

Push the node to the stack

Increase the index by 1

Else if the letter is I:

Pop two nodes from the stack and join them

Push the joined node to the stack

Pop the stack again and return the node

Ex.

First iteration:

Tree dump: LpLmLdLuIIILiLsII

Tree: p, m, d, u, I

Stack: p, m, I

Second iteration:

Tree dump: LpLmLdLuIIILiLsII

Tree: p, m, d, u, I1, I2

Stack: p, I

Third iteration:
Tree dump: LpLmLdLuIIILiLsII
Tree: p, m, d, u, I1, I2, I3
Stack: I3
Fourth iteration:
Tree dump: LpLmLdLuIIILiLsII
Tree: p, m, d, u, I1, I2, I3, i, s, I4
Stack: I3, I4
Fifth iteration:
Tree dump: LpLmLdLuIIILiLsII
Tree: p, m, d, u, I1, I2, I3, i, s, I4, I5
Stack: I5

void delete_tree(Node **root)
Goal: the destructor for a Huffman tree.

If the root exists:
    Call the function on the left child of the root
    Call the function on the right child of the root
    Free the root node
Free the tree itself and set its pointer to NULL

Makefile

Goal: compiles the program
Set the compiler to clang
Set the C flags to the regular flags used to compile the program -Wall -Werror -Wextra -Wpedantic
Set EXECBIN to the names of the prgrams Makefile will build (encode and decode)
Define the target ‘all’ with the dependency of the file holding the main function (encode and decode)

Define the target ‘decode’ with the dependencies of the file decode
Define the target ‘encode’ with the dependencies of the file encode

Have a variable that will hold all c files
Have a variable that will hold all object files (Use the pattern match placeholder % to convert the C file to an object file).

Define the target clean with the command to remove all compiler-generated files except the executables

Define the target spotless with the command to remove all files that are generated and the executable files. This will call ‘make clean’ to remove everything except executables, and then will remove the executable

Define the target format with the command to format the source files