Assignment 6 Design

Nhan Nguyen

February 2022

1 Introduction

1.1 Description

The purpose of this program is to perform a traditional Huffman Code which will be used to perform a lossless data compression

1.2 List of files

- **decode.c:** Provide the implementation of the Huffman decoder
- **encode.c:** Provide the implementation of the Huffman encoder
- **defines.h:** Provide the macros necessary for the programs
- **header.h:** Provide a header for the Huffman encoder
- **node.h:** The interface for node.c
- **node.c:** The implementation of the Nodes necessary for the Huffman Tree
- **pq.h:** The interface for pq.c
- **pq.c:** The implementation of a Priority Queue used to build a Huffman Tree
- **code.h:** The interface for code.c
- **code.c:** The implementation for Code, an abstract data structure build specifically to contain bits
- **io.h:** The interface for io.c
- **io.c:** The implementation for the I/O specifically used for this program as opposed to the regular I/O from stdio
- **stack.h:** The interface for stack.c
- **stack.c:** The implementation of a Stack used to rebuild the Huffman Tree

- **huffman.h:** The interface for huffman.h
- **huffman.c:** The implementation for the building, deleting, traversing the Huffman Tree
- **Makefile:** This file formats program into clang-format and also compiles it. This Makefile will be responsible for compiling encode.c decode.c with all the files specified above
- **README.md:** A text file in Markdown format that describes how to build the program, as well as describing the problems encountered while making the program
- **DESIGN.pdf:** This PDF file describes the idea behind the program with visualization and pseudocode

2 node.c

This file will contain the necessary functions to form a Node that is used for the Huffman tree

2.1 node_create()

This function creates a Node object, a node will have 4 components: the left node connected to it, the right node connected to it, the node's symbol, and the node's frequency. node_create() will create a new Node pointer by allocating memory for the Node, set both the left and right nodes to NULL and the symbol and frequency to that of ones specified in the parameter
Pseudocode:

```
node_create(symbol, frequency)
    allocate memory for node pointer
    if node pointer != NULL:
        node's left node = NULL
        node's right node = NULL
        node's symbol = symbol
        node's frequency = frequency
    return node pointer
```

2.2 node_delete()

This function will be the destructor function of Node since it can delete a Node. To delete the node simply use free() for the node's pointer and set the node pointer to NULL after that
Pseudocode:

```
node_delete(node pointer)
    free node pointer n
```

node pointer n = NULL

2.3 node_join()

This function creates a new node and joins the two nodes together from the parameter. To join the nodes, simply create a new pointer by allocate memory to it, set the left and right nodes to the ones specified in the parameter, node's symbol to the '$' symbol, and the node's frequency being the sum of the frequency of the nodes left and right

Pseudocode:

```
node_join(left, right)
    allocate memory for node pointer
    if node pointer != NULL:
        node pointer's left node = left
        node pointer's right node = right
        node pointer's symbol = '$'
        node pointer's frequency = left's frequency + right's frequency
    return node pointer
```

2.4 node_print()

This function just prints out the contents inside a node. This is a debug function so the choice to print the content is entirely up to you. For testing if the nodes are connected properly, it's better to print everything about the Node while printing the frequency only will be good if the Nodes are inside a data structure like Stack and Priority Queue.

Pseudocode:

```
node_print(node pointer)
    if node pointer == NULL:
        print "Node is not created"
        return
    print node's left
    print node's right
    print node's symbol
    print node's frequency
```

3 stack.c

This file contains the implementation for a data structure called a Stack, this data structure will be useful when rebuilding the Huffman Tree.

3.1 stack_create()

This function creates the Stack object and set the size to the size specified in the parameter. You will need to allocate memory twice in this function because you need to make one for the Stack object and one for the Node array inside of the Stack

Pseudocode:

```
stack_create(capacity)
    allocate memory for stack pointer
    if stack pointer != NULL:
        stack pointer's capacity = capacity
        stack pointer's top = 0
        allocate memory for the stack pointer's Node array
    return stack pointer
```

3.2 stack_delete()

This function is the destructor function of the Stack object, to delete a stack we need to delete the following in order by using free(): The elements in the stack, the stack's Node array, and the stack array. The stack pointer will also need to be set to NULL

Pseudocode:

```
stack_delete(stack pointer)
    for i from 0 to stack pointer's top:
        node_delete(stack pointer's Node array[i])
    free stack's pointer Node array
    free stack pointer
    stack pointer = NULL
```

3.3 stack_empty()

This function will check if the stack is empty or not. A simple way to detect this is by looking at the stack pointer's top since if the stack pointer's top is 0, then the stack is definitely empty

Pseudocode:

```
stack_empty(stack pointer)
    return stack pointer's top == 0
```

3.4 stack_full()

This function will check if the stack is full or not. A simple way to detect this is by once again looking at the stack pointer's top since if the stack pointer's top is equal to the stack pointer's capacity, the stack is full

Pseudocode:

```
stack_full(stack pointer)
    return stack pointer's top == stack pointer's capacity
```

3.5 stack_size()

This function will return the stack's current size. This is simply done by return the stack pointer's top
Pseudocode:
stack_size(stack pointer)
    return stack pointer's top

3.6 stack_push()

This function will push another Node specified in the parameter into the stack. Since the data structure we're using is a stack, the Node is inserted in the back (in the stack's case on top) of the stack. The stack pointer's top is increase by 1 after the operation. In addition, we can't insert the Node to the stack is the stack is full
Pseudocode:
stack_push(stack pointer, node pointer)
    if stack_full(stack pointer) return false
    stack pointer's Node array[stack pointer's top] = node pointer
    stack pointer's top += 1
    return true

3.7 stack_pop()

This function will pop the last node (the top node) out of the stack and stores it in the node pointer in the parameter. Simply decreasing the stack pointer's top down by 1 and storing the node to the node pointer will do the trick. In addition, we can't pop if there's no elements in the stack
Pseudocode:
stack_pop(stack pointer, node pointer)
    if stack_empty(stack pointer) return false
    stack pointer's top -= 1
    node pointer = stack pointer's Node array[stack pointer's top]
    return true

3.8 stack_print()

This function is a debugging function that will print out the current element's in the stack. Simply use a for loop and node_print() will do the trick.
Pseudocode:
stack_print(stack pointer)
    for i in stack pointer's size:
        node_print(stack pointer's Node array[i])

4 pq.c

This file contains the implementation for a data structure called a Priority Queue, this data structure will be useful when building the Huffman Tree.

4.1 pq_create()

This function creates the priority queue object and set the size to the size specified in the parameter. You will need to allocate memory twice in this function using malloc(): one for the priority queue object and the Node array inside the priority queue object

Pseudocode:

```
pq_create(capacity)
    allocate memory for priority queue pointer
    if priority queue pointer != NULL:
        priority queue's front = priority queue's tail = 0
        priority queue's capacity = capacity
        allocate memory for the priority queue's Node array
    return priority queue pointer
```

4.2 pq_delete()

This function is the destructor function for the priority queue. Similar to stack_delete(), you will need to delete (free) the components in the following order: elements in the Node array, the Node array, and the priority queue itself. Also set the priority queue pointer to NULL

Pseudocode:

```
pq_delete(priority queue pointer)
    for i from 0 to priority queue pointer's top:
        node_delete(priority queue pointer's Node array[i])
    free priority queue's pointer Node array
    free priority queue pointer
    priority queue pointer = NULL
```

4.3 pq_empty()

This function will check if the current priority queue is empty or not. This can be done by checking if priority queue pointer's front is equal to priority queue pointer's tail or not

Pseudocode:

```
pq_empty(priority queue pointer)
    return priority queue pointer's front == priority queue pointer's tail
```

4.4 pq_full()

This function will check if the current priority queue is full or not. We can check this by seeing if priority queue pointer's tail is equal to priority queue pointer's capacity or not

Pseudocode:

pq_full(priority queue pointer)
    return priority queue pointer's tail == priority queue pointer's capacity

4.5 pq_size()

This function will return the current size of the priority queue. For this we can just return the priority queue pointer's tail to determine the size

Pseudocode:

pq_size(priority queue pointer)
    return priority queue pointer's tail

4.6 enqueue()

This function will push a Node to the back of the queue (the queue's tail). Since this is a priority queue, this Node will also move to the necessary spot to maintain the heap. To do this we will utilized a while loop and swap the added node with it's parents (the ((i - 1) / 2)-th node) until either the parents is smaller than the node or the index is 0 (aka the root). In addition, we will not enqueue if the queue is already full

Pseudocode:

enqueue(priority queue pointer, node pointer)
    if pq_full(priority queue pointer) return false
    i = priority queue pointer's tail
    priority queue pointer's Node array[i] = node pointer
    while i != 0 and (priority queue pointer's Node array[(i - 1) / 2]'s frequency > priority queue pointer's Node array[i]'s frequency
        swap priority queue pointer's Node array[(i - 1) / 2] and priority queue pointer's Node array[i]
        i = (i - 1) / 2
    priority queue pointer's tail += 1
    return true

4.7 dequeue()

This function will pop a Node out of the queue from the front (the queue's top). Since this is a priority queue, there's will be a fix_heap() function to maintain the heap after the element is pop. To pop the element, simply replace the top element with the element from the tail then decrease the priority queue's pointer

tail. Note that every time is operation is done, the heap need to be fix with fix_heap(), which is done recursively. In addition, we can't pop the element out if there's no element in the heap

Pseudocode for dequeue():

dequeue(priority queue pointer, node pointer)
    if pq_empty(priority queue pointer) return false
    node pointer = priority queue pointer's Node array[priority queue pointer's top]
    priority queue pointer's Node array[priority queue pointer's top] = priority queue pointer's Node array[priority queue pointer's tail - 1]
    priority queue pointer's tail -= 1
    fix_heap(priority queue pointer, 0)

Pseudocode for fix_heap():

fix_heap(priority queue pointer, start)
    l = start * 2 + 1
    r = start * 2 + 2
    small = start
    if l < priority queue pointer's tail and (priority queue pointer's Node array[l]'s frequency < priority queue pointer's Node array[start]'s frequency:
        small = l
    if r < priority queue pointer's tail and (priority queue pointer's Node array[r]'s frequency < priority queue pointer's Node array[small]'s frequency:
        small = r
    if small != start
    swap priority queue pointer's Node array[small] and priority queue pointer's Node array[start]
    fix_heap(priority queue pointer, small)

4.8 pq_print(priority queue pointer)

This function is a debugging function to see what are the nodes in the priority queue currently. A simple for loop would do the trick

Pseudocode:

    for i from priority queue pointer's front to priority queue pointer's tail:
        node_print(priority queue pointer's Node array[i])

5 code.c

This file contains all the functions that supports the abtract data structure made for this assignment, Code

5.1 code_init()

This function creates a new Code object. To create this new Code object we just need to make the Code's top = 0 and every element in the array size MAX_CODE_SIZE (32) inside the code into 0. There will be no memory allocated in this function

Pseudocode:

```
Code c
c's top = 0
for i from 0 to MAX_CODE_SIZE:
    c's bits[i] = 0
return c
```

5.2 code_size()

This function will return the current size of the code. This is done by returning the Code's top.

Pseudocode:

```
code_size(code pointer)
    return code pointer's top
```

5.3 code_empty()

This function checks if the current code is empty or not. We can do this by checking if Code's top is equal to 0

Pseudocode:

```
code_empty(code pointer)
    return code pointer's top == 0
```

5.4 code_full()

This function checks if the current code is full or not. We can do this by checking if code's top is equal to the MAX_CODE_SIZE * 8 (256)

Pseudocode:

```
code_full(code pointer)
    return code pointer's top == 256
```

5.5 code_set_bit()

This function set the i-th bit specified in the parameter to 1. This can be done by using some bitwise operation: OR (|), left bit rotation (<<). In addition, we need to figure out what index in the Code's bit array and also what bit in that index we're setting (from 0th bit to the 7th bit). In addition, we can't set

the bit to 1 if i is larger than the code's size
Pseudocode:
code_set_bit(code pointer, i)
    if i > code_size(code pointer) return false
    position = i / 8
    value = i mod 8
    code pointer's bits[position] |= (1 << val)
    return true

5.6 code_clr_bit()

This function set the i-th bit specified in the parameter to 0. This can be done by using some bitwise operation: AND (&), left bit rotation (<<), and NOT. In addition, we need to figure out what index in the Code's bit array and also what bit in that index we're setting (from 0th bit to the 7th bit). In addition, we can't set the bit to 0 if i is larger than the code's size
Pseudocode:
code_clr_bit(code pointer, i)
    if i > code_size(code pointer) return false
    position = i / 8
    value = i mod 8
    code pointer's bits[position] &= NOT(1 << val)
    return true

5.7 code_get_bit()

This function check if the i-th bit specified in the parameter is 0 or 1. This can be done by using some bitwise operation: AND (&), left bit rotation (<<). In addition, we need to figure out what index in the Code's bit array and also what bit in that index we're setting (from 0th bit to the 7th bit). In addition, we can't set the bit to 0 if i is larger than the code's size
Pseudocode:
code_get_bit(code pointer, i)
    if i > code_size(code pointer) return false
    position = i / 8
    value = i mod 8
    return (code pointer's bits[position] & (1 << val)) ¿ 0

5.8 code_push_bit()

This function will push a bit specified in the parameter to the end of the Code (at code's top). This function will utilize code_set_bit() and code_clr_bit(). In addition, we can't push a bit to the code if the code is full
Pseudocode:

```
code_push_bit(code pointer, bit)
    if code_full(code pointer) return false
    if bit == 1:
        code_set_bit(code pointer, code pointer's top)
    else:
        code_clr_bit(code pointer, code pointer's top)
    code pointer's top += 1
    return true
```

5.9 code_pop_bit()

This function will push a bit specified in the parameter to the end of the Code (at code's top). This function will utilize code_clr_bit() and code_get_bit(). In addition, we can't push a bit to the code if the code is empty

Pseudocode:

```
code_push_bit(code pointer, unsigned int pointer)
    if code_empty(code pointer) return false
    unsigned int pointer = code_get_bit(code pointer, code pointer's top - 1)
    code_clr_bit(code pointer, code pointer's top - 1)
    code pointer's top -= 1
    return true
```

5.10 code_print()

This function is a debugging function that prints the current code specified in the parameter. A simple for loop through from 0 to MAX_CODE_SIZE will do the trick.

Pseudocode:

```
code_print(code pointer)
    for i from 0 to MAX_CODE_SIZE:
        print code pointer's bit array[i]
```

6 io.c

This file contains the implementation of the custom-made IO necessary to for encoding and decoding the message. In addition, the custom-made IO will also help when implementing functions in huffman.c

6.1 read_bytes()

This function make use of the low-level read function read() to read the bytes specified in the parameter and puts it into an array of bytes. Since the read()

does not guarantee to read all the bytes specified in one call, a while loop is needed to ensure that the read() function reads enough bytes. The loop is stopped once there are no more bytes to read or the number of bytes read reached the number bytes needed to be read in the parameter

Pseudocode:

```
read_bytes(infile, buf, nbytes)
    assign = nbytes
    cur = 0
    readb = 0
    while true:
        readb = read(infile, buf, assign)
        if readb == 0: break
        cur += readb
        assign -= cur
        if assign == 0: break
    return cur
```

6.2 write_bytes()

This function make use of the low-level print function write() to write the bytes specified in the parameter and puts it into an array of bytes. Since the write() does not guarantee to write all the bytes specified in one call, a while loop will be used to ensure the write() function writes the necessary number of bytes.

Pseudocode:

```
write_bytes(outfile, buf, nbytes)
    assign = nbytes
    cur = 0
    writeb = 0
    while true:
        writeb = write(outfile, buf, assign)
        if writeb == 0: break
        cur += writeb
        assign -= cur
        if assign == 0: break
    return cur
```

6.3 read_bit()

This file reads each bit from the file as opposed to reading the bytes like read_bytes(). It's important to note that we can't directly read the bits from a file, therefore a good way is to store the bytes read into a buffer and a have a pointer that points to the first bit in that buffer. Every time read bit is called we will increment the pointer by 1. It's important to note if the current buffer is full or is empty when we first call read_bit(), we will need to fill this buffer up

with bytes thanks to the read_bytes() we implemented earlier. To assign the bit to the unsigned int pointer we will be using the same bitwise operations used in code_get_bit()

Pseudocode:

```
curpos = 0
buffer[BLOCK] # BLOCK = 4096
bytes_read = 0
read_bit(infile, unsigned int pointer)
    if curpos mod (BLOCK * 8) == 0:
        for i from 0 to BLOCK: buffer[i] = 0
        bytes_read += read_bytes(infile, buffer, BLOCK)
    if curpos == (bytes_read * 8): return false
    pos = (curpos mod (BLOCK * 8)) / 8
    val = (curpos mod (BLOCK * 8)) mod 8
    unsigned in pointer = (buffer[pos] & (1 << val)) ¿ 0
    curpos++
    return true
```

6.4 write_code()

This function will write the Code specified in the parameter out into the buffer (outside of the function). This is accomplish by having a loop from 0 to code pointer's top, code_get_bit(), and write_bytes(). Node that the buffer mentioned also needs another pointer that at first points to 0. To set/clear the bit, we need to use the same bitwise operation we use for code_set_bit(), and code_clr_bit() respectively

Pseudocode:

```
curposc = 0
bufferc[BLOCK] # BLOCK = 4096
bytes_written = 0
write_code(infile, code pointer)
    pos = 0
    val = 0
    for i from 0 to code pointer's top
        pos = (curpos mod (BLOCK * 8)) / 8
        val = (curpos mod (BLOCK * 8)) mod 8
        if code_get_bit(code pointer, i): bufferc[pos] |= (1 << val)
        else: bufferc[pos] &= (255  (1 << val))
        curposc += 1
        if curposc mod (BLOCK * 8) == 0:
            bytes_written += write_bytes(outfile, bufferc, BLOCK)
            for j from 0 to BLOCK: bufferc[j] = 0
```

6.5 flush_codes()

This function will simply print out all the bits remaining in the bufferc that we initialized while calling write_code(). One important detail to note is that we need to know which byte we are at and perform a for loop from 0 to that byte to not accidentally print out the bytes where they're 0
Pseudocode:

```
flush_codes(outfile)
    pos = ((curposc - 1) mod (BLOCK * 8)) / 8
    writebytes(outfile, bufferc, pos + 1)
```

7 huffman.c

This file has all the functions necessary to create, rebuild, dump, delete a Huffman Tree. Which will be crucial during the encoding, decoding process

7.1 build_tree()

This function builds a Huffman tree and return the Node pointer of the root. To build this Huffman tree, it's necessary that we have a priority queue to sort the characters in terms of their frequency. **Pseudocode:**

```
build_tree(hist)
    pq = pq_create(256)
    for i from 0 to ALPHABET: # ALPHABET = 256
        if hist[i] ¿ 0:
            sub = node_create(i, hist[i])
            enqueue(pq pointer, sub pointer)
    while pq_size(pq) >= 2:
        n1 = node_create('0', 0)
        n2 = node_create('0', 0)
        dequeue(pq pointer, n1 pointer)
        dequeue(pq pointer, n2 pointer)
        enqueue(pq pointer, node_join(n1, n2))
    ans = node_create('0', 0)
    dequeue(pq, ans pointer)
    return ans
```

7.2 build_codes()

This function builds a code table where each character in the Huffman tree will have it's on code representing the path from the root to the Node. To build this Code table, we're going to need a post-order traversal implemented by using recursion to find a path to each leaf of the Huffman tree

Pseudocode for build_codes():

```
build_codes(root, table)
    if root != NULL
        c = code_init()
        travel(root, code pointer, table)
```

Pseudocode for travel():

```
travel(node pointer, code pointer, table)
    if node pointer's left == NULL and node pointer's right == NULL:
        for i from 0 to code pointer's top:
            if code_get_bit(code pointer, i):
                code_push_bit(table[node pointer's symbol], 1)
            else:
                code_push_bit(table[node pointer's symbol], 0)
            return
    popped = 0
    code_push_bit(code pointer, 0)
    travel(node pointer's left, code pointer, table)
    code_pop_bit(code pointer, popped)
    code_push_bit(code pointer, 1)
    travel(node pointer's right, code pointer, table)
    code_pop_bit(code pointer, popped)
```

7.3 dump_tree()

This function will print out a sequence of character that describes the Huffman tree, if the Node we're at is a leaf, we print out 'L' and the symbol of that Node, otherwise we just print 'I'. To print this sequence we will once again perform a post-order traversal.

Pseudocode:

```
dump_tree(outfile, node pointer)
    if node pointer != NULL:
        dump_tree(outfile, node pointer's left)
        dump_tree(outfile, node pointer's right)
        if node pointer's left == NULL or node pointer's right == NULL:
            print[2] = { 'L', node pointer's symbol }
            write_bytes(outfile, print, 2)
        else:
            print[1] = {'I'}
            write_bytes(outfile, print, 1)
```

7.4 rebuild_tree()

This function will rebuild a Huffman tree from the sequence of characters generated by dump_tree(). Since dump_tree() already produced a specific sequence of Nodes due to the post-order traversal, using a stack to rebuild the tree will be the optimal choice.

Pseudocode:

```
rebuild_tree(nbytes, tree)
    s = stack_create(nbytes)
    for i from 0 to nbytes:
        if tree[i] == 'L':
            stack_push(s, node_create(tree[i + 1], 0))
            i += 1
        else if tree[i] == 'I':
            n1 = node_create('0', 0)
            n2 = node_create('0', 0)
            stack_pop(s pointer, n1 pointer)
            stack_pop(s pointer, n2 pointer)
            stack_push(s, node_join(n2 pointer, n1 pointer)
    ans = node_create('0', 0)
    stack_pop(s pointer, ans pointer)
    return ans
```

7.5 delete_tree()

This function will delete the entire tree. It's crucial to delete the tree in a post-order traversal to avoid going to a Node that you have deleted and get a segmentation fault as a result

Pseudocode:

```
delete_tree(node pointer)
    if node pointer != NULL:
        if(node pointer's left != NULL) delete_tree(node pointer's left)
        if(node pointer's right != NULL) delete_tree(node pointer's right)
        node_delete(node pointer)
```

8 encode.c

This file contains the main program and will be responsible for encoding a file into a compressed version by utilizing the Huffman tree. We will encode the message by following these steps in order:

1) Read in the proper commands

2) Create a buffer size 4096, code table size 256, and histogram size 256

3) Read from the input file and store them into a temporary file (note that you should read 1 byte at a time in this case since stdin can't read multiple

blocks at a time), you can fill up the histogram in this step also to save a loop

4) build the tree from the histogram

5) build the code table from the tree

6) setup the header, fill in the necessary information, and print it to the output file

7) dump the tree to the output file

8) read in the data from the input again then print the appropriate code out from the code table. Make sure to flush out all the code remaining in the buffer

9) if printing statistics is enabled, you will need to print out the necessary statistics

Pseudocode:

```
main()
    stats = false
    input file = stdin's file no (0)
    output file = stdout's file no (1)
    create temp file
    make histogram size 256
    make code table size 256
    read the commands
        if command is i:
            close the current input file
            input = file descriptor of a new file using open()
            check if file is open properly
        if command is o:
            close the current output file
            output = file descriptor of a new file using open()
            check if file is open properly
        if command is v:
            stats = true
        if command is h:
            show synopsis
    make inarr buffer size 4096
    while true:
        read 1 byte from the file
        if can't read anymore: break
        write byte to temporary file
        histogram[the read byte]++
    build_tree()
    build_codes()
```

initialize header and set the header's content accordingly
write the header out into the output file
dump tree into the output file
while true:
bytes_read = attempt to read 4096 bytes from the temporary file and put in inarr array
for i from 0 to bytes_read:
write the code from code table[inarr[i]]
clear the buffer inarr
flush out the rest of the buffer

9 decode.c

This file will simply decode the encoded message from encode.c back into the original message. To accomplish this we will need to accomplish the following steps:

1) read in the commands
2) read in the header from the input file, if the header's magic number is not match we end the file.
3) read the dump tree thanks to the tree size from the header
4) rebuild the tree from the dump tree data
5) continue to read the remaining bits until the number of characters is equal to the original message's size, for each bit, traverse the tree accordingly until a leaf is reach. In that case reset the node back to the root and continue to traversal
6) if statistics is enabled, print the statistics

Pseudocode:
cnt = 0
traverse(outfile, root, dir):
if root's right == NULL and root's left == NULL:
out = root'symbol
write out into outfile
cnt++
root = mroot
if dir == 0:
return root's left
if dir == 1:
return root's right
main()
stats = false

```
input file = stdin file number
output file = stdout file number
read the commands
    if command is i:
        close the current input file
        input = file descriptor of a new file using open()
        check if file is open properly
    if command is o:
        close the current output file
        output = file descriptor of a new file using open()
        check if file is open properly
    if command is v:
        stats = true
    if command is h:
        show synopsis
read in the header
if header's magic number is not equal to magic number: exit the program
read in the dump tree
root = rebuild the tree
cnt = 0
currroot = root
while cnt ¡ header's file size
    bit = read_bit()
    currroot = traverse(output file, currroot, bit)
if stats == true:
    print stats
```