Huffman Coding
Assignment 7
Dalal Arafeh

Purpose:

The purpose of Huffman coding, central to our huff and dehuff programs, is to achieve lossless data compression. Huff compresses data. It counts the frequency of each byte in the input file and constructs a binary tree where each leaf represents a byte. This ensures shorter paths for more frequent bytes. Huff generates huffman code for each byte based on the constructed tree, encoding the original data. Dehuff decompresses the data encoded with Huffman coding. It reads the Huffman tree structure from the compressed file and converts the Huffman codes back into their original bytes, restoring the data to its uncompressed form.

Questions:

- **Describe the goal of compression. (As a hint, why is it easy to compress the string "aaaaaaaaaaaaaaaa")**

  Compression reduces data size for better storage. Compression algorithms can represent repeated data. For example it would indicate that 'a' occurred 16 times rather than storing each 'a' individually.

- **What is the difference between lossy and lossless compression? What type of compression is huffman coding? What about JPEG? Can you lossily compress a text file?**

  Lossless compression: Keeps all the original data so we can retrieve the original when decompressing (Huffman coding).
  Lossy compression: Not all the original data is kept since it reduces the size of the file and some data would be lost (JPEG).
  We could lossily compress a text file for removing some data but it is more common to use lossless methods.

- **Can your program accept any file as an input? Will compressing a file using Huffman coding make it smaller in every case?**

  The program can accept any files as an input. Compressing a file using Huffman coding will not make it smaller in every case. In some cases, the encoded file may be larger than the original file because Huffman coding relies on the frequency of symbols in the file and if there's no repetition or pattern, there's limited compression.

- **How big are the patterns found by Huffman Coding? What kind of files does this lend itself to?**

  The size of patterns found by Huffman Coding varies based on the frequency of symbols (bytes) in the data. It works best with files containing frequent repetitions or patterns, such as text files.

- **Take a picture on your phone. What resolution is the picture? How much space does it take up in your storage (in bytes)?**

The image resolution is 12MP and took up 1.4MB.

- **If each pixel takes 3 bytes, how many bytes would you expect the picture you took to take up? Why do you think that the image you took is smaller?**

  12MP = 3024 x 4032 pixels

  12,192,768 pixels

  12,192,768 pixels x 3 bytes/pixel = 36,587,304 bytes

  36,587,304 bytes

  The image I took was smaller because JPEG uses lossy compression and some of the data would be lost.

- **What is the compression ratio of the picture you just took? To get this, divide the actual size of the image by the expected size from the question above. You should not get a number above 1.**

  Actual size = 1.4 MB = 1,400,000 bytes

  Expected size = 36,587,304 bytes

  Compression ratio = 1,400,000 bytes / 36,587,304 bytes

  = 0.0383

  = 3.83% of the expected size

- **Do you expect to be able to compress the image you just took a second time with your Huffman program?1 Why or why not?**

  I do not think I would be able to compress the image if I used the Huffman program because Huffman coding is most effective on uncompressed data and the image is already compressed the first time.

- **Are there multiple ways to achieve the same smallest file size? Explain why this might be.**

  Yes, there are multiple ways to achieve the same smallest file size. There are different algorithms and methods for compressing data. If there are multiple characters with the same frequency in the data, the algorithm can create different codes for them, leading to multiple ways to achieve the smallest file size.

- **When traversing the code tree, is it possible for an internal node to have a symbol?**

  Only leaf nodes have symbols. Internal nodes only combine symbols and frequencies, acting as branching points in the tree structure.

- **Why do we bother creating a histogram instead of randomly assigning a tree?**

  Creating a histogram instead of randomly assigning a tree helps us understand how often each symbol appears in the data. Huffman coding assigns shorter codes to more frequent symbols meeting its purpose of compression.

- **Relate this Huffman coding to Morse code. Why does Morse code not work as a way of writing text files as binary? What if we created more symbols for things like spaces and newlines?**

Morse code always uses the same number of dots and dashes for each letter, no matter how often it's used. This makes it inefficient for compression because it doesn't take advantage of the fact that some letters are more common than others in typical text. Huffman coding assigns shorter codes to more frequent letters, making it better for compression.

- **Using the example binary, calculate the compression ratio of a large text of your choosing**

  Compression Ratio = Actual Size / Expected Size

  = 10,000,000 bytes / 5,000,000 bytes

  = 2

Testing:

To ensure there are no memory leaks:

valgrind --leak-check=full --track-origins=yes ./dehuff

valgrind --leak-check=full --track-origins=yes ./huff

I would also run this command:

./runtests.sh

To check for each one of my c files (bitwriter.c, bitreader.c, node.c, pq.c)

```
dalalarafeh@dalalarafehmac:~/harafeh/asgn7$ valgrind --leak-check=full --track-origins=yes ./huff
==4785== Memcheck, a memory error detector
==4785== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==4785== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==4785== Command: ./huff
==4785==
huff:  -i option is required
Usage: huff -i infile -o outfile
        huff -v -i infile -o outfile
        huff -h
==4785==
==4785== HEAP SUMMARY:
==4785==     in use at exit: 0 bytes in 0 blocks
==4785==   total heap usage: 1 allocs, 1 frees, 1,024 bytes allocated
==4785==
==4785== All heap blocks were freed -- no leaks are possible
==4785==
==4785== For lists of detected and suppressed errors, rerun with: -s
==4785== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
dalalarafeh@dalalarafehmac:~/harafeh/asgn7$ valgrind --leak-check=full --track-origins=yes ./dehuff
==4786== Memcheck, a memory error detector
==4786== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==4786== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==4786== Command: ./dehuff
==4786==
dehuff:  -i option is required
Usage: dehuff -i infile -o outfile
       dehuff -v -i infile -o outfile
       dhuff -h
==4786==
==4786== HEAP SUMMARY:
==4786==     in use at exit: 0 bytes in 0 blocks
==4786==   total heap usage: 1 allocs, 1 frees, 1,024 bytes allocated
==4786==
==4786== All heap blocks were freed -- no leaks are possible
==4786==
==4786== For lists of detected and suppressed errors, rerun with: -s
==4786== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

```
dalalarafeh@dalalarafehmac:~/harafeh/asgn7$ ./runtests.sh
Test bitwriter.c:
bwtest, as it is, reports no errors

Test bitreader.c
brtest, as it is, reports no errors

Test node.c:
Use "nodetest -v" to print trace information.
nodetest, as it is, reports no errors

Test pq.c:
Use "pqtest -v" to print trace information.
pqtest, as it is, reports no errors
```

How to use the program:

- Makefile
  - Includes all the header and function files
  - Create an objective file for each so they could be used when running the main file.
  - make format to fix the formatting of the file to match with clang
  - make clean to remove any previous object files
  - Run make to compile all the files and create an executable file

- The command line options:
    - (-i): read from input file
    - (-o): write the output in a file
    - (-h): print help message
- To run the program:
    ./huff -i inputfile -o outputfile
    ./dehuff -i inputfile -o outputfile

Program Design:

- **Huff:** reads an input file to count how often each byte appears. Then, it creates a Huffman tree based on these frequencies. It generates Huffman codes for each byte using that tree. It compresses the original data using these codes and saves it in a binary file along with the tree structure. This allows for later decompression of the data.
- **Dehuff:** The decompressor reads the binary file created by huff, reconstructing the Huffman tree and decodes the compressed data using this tree and decodes the Huffman codes back into their original byte sequences
- **Bitreader:** reading bits from a binary file and helps the decompressor interpret the stored binary data correctly
- **Bitwriter:** write bits to a binary file and makes it easier to write compressed data bit by bit
- **Priority Queue:** important for building the Huffman tree, giving priority to bytes with lower frequencies
- **Binary Tree:** build and navigate binary trees, important for converting byte frequencies into Huffman codes.

Data Structure:

- **Node:** stores a byte symbol, its frequency, the assigned Huffman code, the length of this code, and pointers to its child nodes
- **ListElement:** linked list, helps organize nodes in a specific order by including a pointer to a node representing a tree and another pointer to the next element in the list
- **PriorityQueue:** contains a pointer to the first ListElement in a linked list, forming the queue. It organizes nodes based on their frequencies for building the Huffman tree.
- **Bitwriter:** a pointer to the output file stream, a buffer to collect bits until they make a full byte, and a counter to keep track of how much of the buffer is filled
- **Bitreader:** a file stream pointer, a buffer for incoming bits, and a counter for tracking the current bit position.

Function Description:

Bitreader.c:

#include "bitreader.h"
#include <stdio.h>
#include <stdlib.h>

Define the BitReader structure
struct BitReader {

FILE *underlying_stream;
uint8_t byte;
uint8_t bit_position;
};

- Opens a file for reading binary data as bits and initializes a BitReader structure
BitReader *bit_read_open(const char *filename) {
allocate memory for the BitReader structure
open file for reading as binary file
If error opening file
free memory
return NULL
assign the file stream to BitReader
initialize current byte to 0
bit position to 8
}

- Closes the BitReader and releases the memory
void bit_read_close(BitReader **pbuf) {
if BitReader is not NULL
close underlying file stream
free BitReader
set pointer to NULL
}

- Reads 32 bits from the BitReader as an unsigned integer
uint32_t bit_read_uint32(BitReader *buf) {
initialize data to 0
iterate over 0 to 32
read single bit
set bit in data
}

- Reads 16 bits from the BitReader as an unsigned integer
uint16_t bit_read_uint16(BitReader *buf) {
initialize data to 0
iterate over 0 to 16
read single bit
set bit in data
}

- Reads 8 bits from the BitReader as an unsigned integer
uint8_t bit_read_uint8(BitReader *buf) {
initialize data to 0

```
        iterate over 0 to 8
                read single bit
                set bit in data
}
```

- Reads a single bit from the BitReader, gets the next byte from the file if needed

```
uint8_t bit_read_bit(BitReader *buf) {
        if the current byte is read
                read next byte
                if we reached the end of the file
                        return 1
                assign next byte to current byte
                reset bit position
        read next bit
        increment to next bit position
}
```

Bitwriter.c:

```
#include "bitwriter.h"
#include <stdio.h>
#include <stdlib.h>

define the BitWriter structure
struct BitWriter {
        FILE *underlying_stream;
        uint8_t byte;
        uint8_t bit_position;
};
```

- Opens a file for writing binary data as bits and initializes a BitWriter structure. If memory allocation or file opening fails, it returns NULL.

```
BitWriter *bit_write_open(const char *filename) {
        allocate memory for the BitWriter structure
        open file in write
        if allocation failed or opening file failed
                return NULL
        file stream to BitWriter
        current byte to 0
        bit position to 0
}
```

- Closes the BitWriter and releases the memory. If there are remaining bits in the current byte, they are written to the file before closing

```
void bit_write_close(BitWriter **pbuf) {
```

if BitWriter is not NULL
    If there are any remaining bits
        Write bits to file
Close underlying file stream
Free allocated memory
Set pointer to NULL
}

- Writes a single bit to the file using the BitWriter,
  void bit_write_bit(BitWriter *buf, uint8_t bi) {
      If current byte full
          Write byte to file
          Error
          Reset byte to 0
          Reset bit position to 0
      buf->byte = buf->byte OR ((bi AND 1) left-shifted by buf->bit_position)
      Move to next position
  }

- Writes a 16-bit unsigned integer to the file using the BitWriter, writing each bit from the least to the most significant
  void bit_write_uint16(BitWriter *buf, uint16_t x) {
      Iterate over 0 to 16
          Set buf's bit based on the least significant bit of x.
          Shift bits to the right
  }

- Writes a 32-bit unsigned integer to the file using the BitWriter, writing each bit from the least to the most significant
  void bit_write_uint32(BitWriter *buf, uint32_t x) {
      Iterate 32 times
          Set buf's bit based on the least significant bit of x.
          Shift bits to the right
  }

- Writes a 32-bit unsigned integer to the file using the BitWriter, writing each bit from the least to the most significant
  void bit_write_uint8(BitWriter *buf, uint8_t byte) {
      Iterate 8 times
          Set buf's bit based on the least significant bit of byte.
          Shift bits to the right
  }

Node.c

```
#include "node.h"
#include <stdio.h>
#include <stdlib.h>
```

- Creates a new Node with the specified symbol and weight, initializes code and code length to 0 and child pointers to NULL. If memory allocation fails, it returns NULL.
  Node *node_create(uint8_t symbol, uint32_t weight) {

```
    Allocate memory for new node
    Assign symbol to new_node's symbol attribute
    Assign weight to new_node's weight attribute
    Set new_node's code attribute to 0
    Set new_node's code length attribute to 0
    Set new_node's left attribute to NULL
    Set new_node's right attribute to NULL

    If memory allocation failed
        Return NULL
    Return new_node
}
```

- Recursively frees memory allocated for a Node and its children. If Node is not NULL, free memory for the left and right children recursively before freeing memory for the Node itself and setting the pointer to NULL.
  void node_free(Node **pnode) {

```
    If Node is not NULL
        Free memory occupied by the left child node of *pnode
        Free memory occupied by the right child node of *pnode
        Free memory
        Set pointer to NULL
}
```

- Recursively prints a Node and its children. Prints the weight of the node, if it's a leaf node, prints its symbol. If the symbol is printable ASCII, it prints the character. Otherwise, it prints the symbol in hexadecimal format.
  void node_print_node(Node *tree, char ch, int indentation) {

```
    If tree is NULL
        Return
    print information of the right child node of the tree
    Print the weight of the tree node with specified indentation

    if tree has no left and right children
        if the symbol of tree node is within printable ASCII range
            Print the symbol of the tree node
        Else
```

Print the symbol of the tree node in hexadecimal format
Print information of the left child node of tree

- Prints the entire tree starting from the root node with indentation
  void node_print_tree(Node *tree) {
  Print information of the tree node with specified indentation and character
  }

Pq.c

#include "pq.h"
#include "node.h"
#include <stdio.h>

Define ListElement structure
typedef struct ListElement ListElement;
struct ListElement {
Node *tree;
ListElement *next;
};

Define the structure for the priority queue
struct PriorityQueue {
ListElement *list;
};

- Creates a new priority queue. It initializes the list pointer to NULL and returns the created priority queue. If memory allocation fails, it returns NULL.
  PriorityQueue *pq_create(void) {
  Allocate memory
  If memory allocation failed
  Return NULL
  Set q's list attribute to NULL
  Return q
  }

- Free memory allocated for the priority queue and its elements. It iterates through all elements in the list, freeing memory for the associated node.
  void pq_free(PriorityQueue **q) {
  If priority queue is NULL
  Return NULL

  Pointer e pointing to the list of *q
  while e is not NULL
  Free memory occupied by the tree attribute of e

Move to next e
Free memory
Set pointer to NULL
}

- Checks if the priority queue is empty by checking if the queue itself or its list is NULL. If either is NULL, it returns true. Otherwise, it returns false.
bool pq_is_empty(PriorityQueue *q) {
If priority queue or list is NULL
Return true
Return false
}

- Checks if the size of the priority queue is 1 by verifying if there is exactly one element in the list. If the list is not NULL and its next pointer is NULL, it returns true. Otherwise, it returns false.
bool pq_size_is_1(PriorityQueue *q) {
Return true if q's list is not NULL and has only one element.
}

- Compares two list elements based on their tree's weights and symbols. It returns true or false based on conditions.
bool pq_less_than(ListElement *e1, ListElement *e2) {
If the weight of e1's tree is less than the weight of e2's tree
Return true
Else if the weight of e1's tree equals the weight of e2's tree
Return true if the symbol of e1's tree is less than the symbol of e2's tree
Else
Return false
}

- Adds a node to the priority queue by creating a new list element and setting its tree pointer to the given tree. It inserts the new element at the front of the queue If the queue is empty or the new element should be at the front. Otherwise, it finds and inserts the new element in the correct position.
void enqueue(PriorityQueue *q, Node *tree) {
Allocate memory
If memory allocation failed
Return
Assign tree to the tree attribute of newe
If the priority queue is empty or newe has lower priority than the first element of the queue
New element ->next = queue ->list;
Queue ->list = new element ;
Else

A pointer prev pointing to the list of q, front of queue
while the next element of prev is not NULL and the next element has higher priority than new element
prev = prev->next;
New element ->next = prev->next;
prev->next = new element;
}

- Removes and returns the front element from the priority queue. If the queue is empty, return an error message and NULL. Otherwise, get the front element and update the front of the queue after getting the tree from it.

Node *dequeue(PriorityQueue *q) {
If queue is empty
Print Error: Empty Queue
Return NULL
A pointer front pointing to the list of q
A pointer tree pointing to the tree attribute of the front element
Update the front of the queue
Free memory for front
Return tree
}

- Prints the elements of the priority queue. It starts from the front of the queue, iterates through all elements, and prints each tree associated with the element. It prints separators between elements

void pq_print(PriorityQueue *q) {
(q != NULL) using assertion
A pointer e pointing to the list of q
Initialize position to 1
while e is not NULL
If the current position is equal to 1, first element
Print indentation
Else
Print indentation
node_print_tree(e->tree);
e = e->next;
Print indentation
}

Huff.c:
#include"bitreader.h"
#include "bitwriter.h"
#include "node.h"
#include "pq.h"
#include <getopt.h>

#include <stdio.h>

Define a structure for storing Huffman codes
typedef struct Code {
    uint64_t code;
    uint8_t code_length;
} Code;

- Calculates byte frequencies from a file. It initializes the histogram, counts the occurrences of each byte while traversing the file, and returns the file size.
  uint32_t fill_histogram(FILE *fin, uint32_t *histogram) {
      Loop through 256 elements, incrementing i each iteration
          Set each element of the histogram array to 0

      Increment the element at index 0x00 in the histogram array
      Increment the element at index 0xFF in the histogram array
      uint32_t filesize = 0;
      int byte;
      Loop until end of file while reading each byte
          Increment element in the histogram array corresponding to the current byte
          Increment file size by 1
      Set the file position indicator to the beginning of the file.
      Return filesize
  }

- Builds a Huffman tree from a histogram by creating nodes for each byte. It combines the nodes in the queue until only one is left and returns the tree.
  Node *create_tree(uint32_t *histogram, uint16_t *num_leaves) {
      If histogram is NULL
          Return NULL
      Create a priority queue pq
      Iterate from 0 to 255
          If frequency of i in histogram > 0
              Create a new node with symbol i and its frequency from the histogram
              Add n_node to the priority queue pq
      until the size of the priority queue is not 1
          Remove the top element from the priority queue and assign it to left
          Remove the top element from the priority queue and assign it to right
          Create a new node with a combined weight of left and right nodes
          Assign the left node to the left child of n
          Assign the right node to the right child of n
          Add the newly created node n to the priority queue pq
          Increment number of leaves
      Remove the top element from the priority queue and assign it to tree

Free priority queue
Increment number of leaves
Return tree
}

- Fills a code table with Huffman codes by traversing a Huffman tree recursively. For each leaf node, it assigns the corresponding Huffman code and its length in the code table.
void fill_code_table(Code *code_table, Node *node, uint64_t code, uint8_t code_length) {
If node is NULL
Return NULL
If the node has no left and right children
code_table[node->symbol].code = code;
code_table[node->symbol].code_length = code_length;
Return
Recursively fill the code table for the left child node
Set the bit at the code length position in the code
Recursively fill the code table for the right child node
}

- Writes a Huffman tree to an output buffer. The tree is traversed recursively, writing the left subtree first and then the right subtree.
void huff_write_tree(BitWriter *outbuf, Node *node) {
If the left child of the node is NULL
Write a bit with value 1 to the output buffer
Write an 8-bit unsigned integer to the output buffer
Else
Write the Huffman tree structure to the output buffer for the left child node
Write the Huffman tree structure to the output buffer for the right child node
Write a bit with value 0 to the output buffer
}

- Compresses a file using Huffman coding. It writes the Huffman tree to the output buffer and reads the input file byte by byte, gets the corresponding Huffman code from the code table and writes it bit by bit to the output buffer.
void huff_compress_file(BitWriter *outbuf, FILE *fin, uint32_t filesize, uint16_t num_leaves, Node *code_tree, Code *code_table) {
bit_write_uint8(outbuf, 'H');
bit_write_uint8(outbuf, 'C');
Write a 32-bit unsigned integer (filesize) to the output buffer
Write a 16-bit unsigned integer (num_leaves) to the output buffer
Write the Huffman tree to the output buffer for the code_tree
fseek(fin, 0, SEEK_SET);
int file;
Read each byte from the file until the end of the file is reached

Get the code corresponding to the byte from the code table
Get the code length corresponding to the byte from the code table
Iterate through each bit of the code
Write the bit to the output buffer
Shift the code to the right
}

- Prints help message options

  void print_options(void) {
  fprintf(stdout, help message
  huff -i infile -o outfile
  huff -v -i infile -o outfile
  huff -h)
  }

- Reads input and output file paths from the command line arguments and opens the input file for reading and the output file for writing bits. It builds a Huffman tree from the input file and generates a code table. It compresses the input file using Huffman coding and writes the compressed data to the output file. It closes both the input and output files.

  int main(int argc, char **argv) {
  int option;
  FILE *inputf = NULL;
  BitWriter *outputvar;

  If more argc less than three
  huff: -i option is required
  Print options
  Return 1
  Else if argc less than or equal to 4
  huff: -o option is required
  Print options
  Return 1
  Until all command line options are processed
  switch (option)
  Case h
  Print options
  Return 0
  Case i
  Open input file
  If opening file failed
  huff: error opening input file
  Print options
  break
  Case o

Open output file for writing bits
If opening file failed
huff: error opening output file
Close output file
Return 1
break
Default
Return 1
Break
uint32_t histogram[256]
Calculate the size of the file and fill the histogram array
uint16_t num_leaves = 0
Create a Huffman tree based on the histogram and update the number of leaves
Allocate memory for the code table, space for 256 Code structures
Fill the code table with codes generated from the Huffman tree
Compress the input file using Huffman coding and write the compressed data to the output file
Free memory for table
Free memory for huffman tree
Close input file
Set input file pointer to NULL
Close output file
Return 0
}

Dehuff.c:

#include "bitreader.h"
#include "node.h"
#include "pq.h"
#include <getopt.h>
#include <stdio.h>
#include <stdlib.h>
#define STACK_SIZE 64

- Pushes a node onto a stack.
  void stack_push(Node **stack, int *front, Node *node) {
  If stack is not full
  Add the node to the stack, incrementing the top of the stack pointer
  Else
  Error: stack push failed
  }
- Pops a node from a stack
  Node *stack_pop(Node **stack, int *front) {

If stack is not empty
Return the top element from the stack, then decrement the top of the stack pointer.
Else
Error: stack pop failed
Return NULL
}

- Decompresses a file that was compressed using Huffman coding. It reads the necessary information from the compressed file's header and constructs the Huffman tree based on the encoded tree structure. It decodes the compressed data using the Huffman tree to write the original bytes to the output file.

void dehuff_decompress_file(FILE *fout, BitReader *inbuf) {
uint8_t type1 = bit_read_uint8(inbuf);
uint8_t type2 = bit_read_uint8(inbuf);
uint32_t filesize = bit_read_uint32(inbuf);
uint16_t num_leaves = bit_read_uint16(inbuf);
assert(type1 == 'H');
assert(type2 == 'C');
Calculate the total number of nodes in the Huffman tree
Node *stack[STACK_SIZE];
int front = -1;
Loop through all nodes in the Huffman tree
Read a single bit from the input buffer and assign it to bit
Node *node;
if (bit == 1)
Read symbol
Create a new node with the specified symbol and a weight of 0
Else
Create a new node with both symbol and weight set to 0
Assign the top element popped from the stack to the right child of the
node
Assign the top element popped from the stack to the left child of the node
Push the node onto the stack, updating the top of the stack pointer.
If stack is empty
Error: Empty
Return
Pop the top element from the stack and assign it to code_tree
Loop through each byte in the file
Node *node = code_tree;
While (1)
Read a single bit from the input buffer and assign it to bit
If (bit == 0)
Move to the left child of the current node
Else

Move to the right child of the current node
If the current node is a leaf node
Break;
Write the symbol of the leaf node to the output file
Free memory
}

- Prints help message options
void print_options(void) {
fprintf(stdout, help message
dehuff -i infile -o outfile
dehuff -v -i infile -o outfile
dehuff -h)
}

- Reads command-line options to determine the input and output files for decompression. It opens the input file for reading bits and uses Huffman coding for decompression. It closes input and output files.
int main(int argc, char **argv) {
int option;
FILE *outf = NULL;
BitWriter *inputvar;
If more argc less than three
dehuff: -i option is required
Print options
Return 1
Else if argc less than or equal to 4
dehuff: -o option is required
Print options
Return 1
Until all command line options are processed
switch (option)
Case h
Print options
Return 0
Case i
Store input
break
Case o
Open output file for writing
If opening file failed
dehuff: error opening output file
Print options
Return 1

```
                break
        Default
                Break
    Open the file specified by inputvar for reading binary data, and initialize a bit reader
    Decompress the data from the input buffer and write the decompressed data to the output
    file
    Close input file
    Close output file
    Return 0
}
```