Assignment 8 – Huffman Coding

Sebastian Avila

CSE 13S – Fall 2023

Purpose

The purpose of this program is to implement a data compressor and a data decompressor. The data will be compressed using Huffman Coding.

How to Use the Program

To use this program you will need to have several files in the same directory. The files are `huff.c`, `dehuff.c`, `bitreader.h`, `bitreader.c`, `bitwriter.h`, `bitwriter.c`, `node.h`, `node.c`, `pq.h`, `pq.c`, `brtest.c`, `bwtest.c`, `nodetest.c`, `pqtest.c`, and `Makefile`. With these files in the same directory, use command `make clean` to remove any preexisting object files. Then, use command `make format` to format all the header and source files. Then use command `make` to compile the program. You will also need an input file that you want to be compressed in the directory or in a sub-folder. The possible command line arguments for both `huff` and `dehuff` are as follows:

- `-i` : Sets the input file. It requires a file name as an argument. It is required to run the program.
- `-o` : Sets the output file. It requires a file name as an argument. It is required to run the program.
- `-h` : Prints a help message to `stdout`.

To run the program use the command `./huff -i infile -o huffout`. This will Huffman encode the file `infile` and return the encoded file as `huffout`. Then use the command `./dehuff -i huffout -o dehuffout`. This will decode the file `huffout` and give the decoded file `dehuffout`. The files `infile` and `dehuffout` will have the same contents.
The program uses several optional compiler flags:

- `-Wall`: This flag enables all warning messages.
- `-Werror`: This flag turns all warnings into errors.
- `-Wextra`: This flag enables extra warning flags that are not enabled by `-Wall`.
- `-Wconversion`: This flag warns for any implicit conversion that may change a value.
- `-Wdouble-promotion`: This flag warns when a float is implicitly promoted to a double.
- `-Wstrict-prototypes`: This flag warns if a function is declared or defined without specifying the argument types.
- `-pedantic`: This flag issues all the warnings demanded by strict ISO C and ISO C++.

Program Design

huff.c

The program begins by checking the command line arguments. The command line arguments and their meanings can be seen in the How To Use the Program section of this report. It then checks if either of the `-i` or `-o` command line arguments were not used. If either one is not used, the respective error message is printed and the usage message is printed. Then the program terminates. Otherwise, the program creates an array of 256 unsigned 32 bit integers to be used as the histogram. It then uses a function to fill this histogram and return the size of the file. It then uses another function to create a Huffman tree and return the number of leaves. It then creates an array 256 `Code` objects to be used as the code table. It then fills the code table using a function. A `BitWriter` object is the created. Then the program uses a function to compress the given input file and write it to the output file. The function makes use of the `BitWriter` object, the file size, the number of leaves, the Huffman tree, and the code table. The program then closes both the input file and the output file. It the frees the memory used for the `BitWriter` object and the Huffman tree. It ends by returning 0.

dehuff.c

This program operates in a similar manner to `huff.c` at first. It begins by checking the command line arguments. The command line arguments and their meanings can be seen in the How To Use the Program section of this report. It then checks if either of the `-i` or `-o` command line arguments were not used. If either one is not used, the respective error message is printed and the usage message is printed. Then the program terminates. Otherwise, the program creates an array of 256 unsigned 32 bit integers to be used as the histogram. It then creates a `BitReader` object. It then uses a function to decode the input file and write the output file. This function uses the `BitReader` object. The program then closes both the input file and the output file and frees the memory used for the `BitReader` object. It the returns 0.

Data Structures

All data structures were given in the assignment instructions[1].

BitWriter

This program defines a type `BitWriter` that is a struct also named `BitWriter` with three variables: a file pointer and two unsigned 8 bit integers.

- `underlying_stream` : This is a file pointer that points to the file in which the bits will be written.
- `byte` : This unsigned 8 bit integer is a buffer that holds the byte that is to be written.
- `bit_position` : This unsigned 8 bit integer represents the bit position where the bit will be written.

```
typedef struct BitWriter BitWriter;

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

BitReader

This program defines a type `BitReader` that is a struct also named `BitReader` with three variables: a file pointer and two unsigned 8 bit integers.

- `underlying_stream` : This is a file pointer that points to the file in which the bits will be written.

- `byte` : This unsigned 8 bit integer is a buffer that holds the byte that is to be written.
- `bit_position` : This unsigned 8 bit integer represents the bit position where the bit will be written.

```
typdef struct BitReader BitReader;

struct BitReader {
    FILE *underlying_stream;
    uint8_t byte;
    uint8_t bit_position;
};
```

Node

This program defines a type `Node` that is a struct, also named `Node`, with 6 variables: two unsigned 8 bit integers, an unsigned 16 bit integer, an unsigned 64 bit integer, and two pointers to `Nodes`.

- `symbol` : This 8 bit integer represents the character that the `Node` contains.
- `weight` : This 32 bit integer represents the weight of the node.
- `code` :
- `code_length` :
- `left` : This is a pointer to a `Node` object.
- `right` : This is a pointer to a `Node` object.

```
typedef struct Node Node;

struct Node {
    uint8_t symbol;
    uint32_t weight;
    uint64_t code;
    uint8_t code_length;
    Node *left;
    Node *right;
}
```

ListElement

The program defines a type `ListElement` that is a struct, also named `ListElement`, with 2 variables: a pointer to a `Node` object and a pointer to a `ListElement` object.

- `tree` : This is a `Node`
- `next` : This is the next `ListElement` in the list

```
typedef struct ListElement ListElement;

struct ListElement {
    Node *tree;
    ListElement *next;
};
```

PriorityQueue

The program defines a type `PriorityQueue` that is a struct, also named `PriorityQueue`, with 1 variable: a pointer to a `ListElement` object, `list`.

```
typedef struct PriorityQueue PriorityQueue;


     struct PriorityQueue {
        ListElement *list;
};
```

Code

The program defines a type `Code` that is a struct with 2 variables: an unsigned 64 bit integer `code` and an unsigned 8 bit integer `code_length`.

```
typedef struct Code {
       uint64_t code;
       uint8_t code_length;
   } Code;
```

Algorithms

Huffman Coding

The assignment instructions provide psuedocode for the Huffman Coding algorithm[1]. This psuedocode implies the use of a priority queue.

```
Huffman Coding
    while Priority Queue has more than one entry
            Dequeue into left
            Dequeue into right
            Create a new node with a weight = left->weight + right->weight
            node->left = left
            node->right = right
            Enqueue the new node
```

Function Descriptions

bitwriter

- `BitWriter *bit_write_open(const char *filename)` : This function takes in a constant character string. It returns a pointer to a `BitWriter`. Its purpose is to open binary `filename` for write. Psuedocode for this function was given in the assignment instructions[1].

  ```
  BitWriter *bit_write_open(const char *filename)
      BitWriter bw = (BitWriter*) malloc(sizeof(BitWriter))
      bw underlying_stream = fopen(filename, wb)
      bw byte = 0
      bw bit_position = 0
      if bw == NULL or bw underlying_stream = NULL
          return NULL
      return bw
  ```

- `void bit_write_close(BitWriter **pbuf)` : This function takes in a pointer to a pointer to a `BitWriter`. It does not return anything. Its purpose is to flush any data in the byte buffer, close `underlying_stream`, free the `BitWriter` object, and set the `*pbuf` pointer to `NULL`. Psuedocode for this function was given in the assignment instructions[1].

```
void bit_write_close(BitWriter **pbuf)
    if *pbuf != NULL then
        if *pbuf bit_position > 0 then
             if fputc(*pbuf byte, *pbuf underlying_stream) = EOF then
                print error message
                exit(1)
        if fclose(*pbuf underlying_stream) = EOF then
            print error message
            exit(1)
        free(pbuf)
        *pbuf = NULL
```

- `void bit_write_bit(BitWriter *buf, uint8_t bit)` : This function takes in a pointer to a `BitWriter` (`*buf`) and an 8 bit integer `bit`. It does not return anything. Its purpose is to write a single bit, `bit`, using values in the `BitWriter` pointed to by `buf`. Psuedocode for this function was given in the assignment instructions[1].

```
void bit_write_bit(BitWriter *buf, uint8_t bit)
    if bit_position > 7 then
        if fputc(buf byte, buf underlying_stream) = EOF then
            print error message
            exit(1)
        buf bit_position = 0
        buf byte = 0
    buf byte = buf byte OR (bit << buf bit_position)
    buf bit_position += 1
```

- `void bit_write_uint8(BitWriter *buf, uint8_t x)` : This function takes in a pointer to a `BitWriter` (`*buf`) and an 8 bit integer `x`. It does not return anything. Its purpose is to write the 8 bits of `x`. Psuedocode for this function was given in the assignment instructions[1].

```
void bit_write_uint8(BitWriter *buf, uint8_t x)
    for i, 0 to i < 8
        bit_write_bit(buf, x >> i AND 1)
```

- `void bit_write_uint16(BitWriter *buf, uint16_t x)` : This function takes in a pointer to a `BitWriter` (`*buf`) and an 16 bit integer `x`. It does not return anything. Its purpose is to write the 16 bits of `x`. Psuedocode for this function was given in the assignment instructions[1].

```
void bit_write_uint16(BitWriter *buf, uint16_t x)
    for i, 0 to i < 16
        bit_write_bit(buf, x >> i AND 1)
```

- `void bit_write_uint32(BitWriter *buf, uint32_t x)` : This function takes in a pointer to a `BitWriter` (`*buf`) and an 32 bit integer `x`. It does not return anything. Its purpose is to write the 32 bits of `x`. Psuedocode for this function was given in the assignment instructions[1].

```
void bit_write_uint32(BitWriter *buf, uint32_t x)
    for i, 0 to i < 32
        bit_write_bit(buf, x >> i AND 1)
```

bitreader

- `BitReader *bit_read_open(const char *filename)` : This function takes in a constant character string `filename`. It returns a pointer to a `BitReader` object. Its purpose is to open the binary `filename` for reading. Psuedocode for this function was given in the assignment instructions[1].

```
BitReader *bit_read_open(const char *filename)
    BitReader br = (BitReader*) malloc(sizeof(BitReader))
    if br == NULL then
        return NULL
    br underlying_stream = fopen(filename, rb)
    if br underlying_stream == NULL then
        return NULL
    br byte = 0
    br bit_position = 8
    return br
```

- `BitReader *bit_read_close(const char *filename)` : This function takes in a pointer to a pointer to a `BitReader`. It does not return anything. Its purpose is to close `underlying_stream`, free the `BitReader` object, and set the `*pbuf` pointer to `NULL`. Psuedocode for this function was given in the assignment instructions[1].

```
void bit_read_close(BitReader **pbuf)
    if *pbuf != NULL then
        if fclose(*pbuf underlying_stream) = EOF then
            print error message
            exit(1)
        free(pbuf)
        *pbuf = NULL
```

- `uint8_t bit_read_bit(BitReader *buf)` : This function takes in a pointer to a `BitReader` (`*buf`). It returns an 8 bit unsigned integer. Its purpose is to read a single bit using values in the `BitReader` pointed to by `buf`. Psuedocode for this function was given in the assignment instructions[1].

```
uint8_t bit_read_bit(BitReader *buf)
    if bit_position > 7 then
        b = fgetc(*buf byte, *buf underlying_stream)
        if b = EOF then
            print error message
            exit(1)
        buf bit_position = 0
    bit = 1 AND (byte >> buf bit_position)
    buf bit_position += 1
```

- `uint8_t bit_read_uint8(BitReader *buf` : This function takes in a pointer to a `BitReader` (`*buf`). It returns an unsigned 8 bit integer. Its purpose is to read 8 bits from `buf`. Psuedocode for this function was given in the assignment instructions[1].

```
uint8_t bit_read_uint8(BitReader *buf)
    uint8_t byte = 0x00
    for i, 0 to i < 8
        byte = byte OR (bit_read_bit(buf) << i)
    return byte
```

- `void bit_read_uint16(BitReader *buf)` : This function takes in a pointer to a `BitReader` (`*buf`). It returns an unsigned 16 bit integer. Its purpose is to read 16 bits from `buf`. Psuedocode for this function was given in the assignment instructions[1].

```
uint8_t bit_read_uint16(BitReader *buf)
    uint8_t byte = 0x00
    for i, 0 to i < 16
        byte = byte OR (bit_read_bit(buf) << i)
    return byte
```

- `void bit_read_uint32(BitReader *buf, uint32_t x)` : This function takes in a pointer to a `BitReader` (`*buf`). It returns an unsigned 32 bit integer. Its purpose is to read 32 bits from `buf`. Psuedocode for this function was given in the assignment instructions[1].

```
uint8_t bit_read_uint32(BitReader *buf)
    uint32_t byte = 0x00
    for i, 0 to i < 32
        byte = byte OR (bit_read_bit(buf) << i)
    return byte
```

node

- `Node *node_create(uint8_t symbol, uint32_t weight)` : This function takes in an unsigned 8 bit integer and an unsigned 32 bit integer. It returns a pointer to a `Node` object or `NULL` if an error occurs. Its purpose is to create and return a `Node` object. Psuedocode for this function was given in the assignment instructions[1].

```
Node *node_create(uint8_t symbol, uint32_t weight)
    Node node = (Node*) malloc(sizeof(Node))
    if node == NULL then
        reeturn NULL
    node symbol = symbol
    node weight = weight
    return node
```

- `void node_free(Node **pnode)` : This function takes in a pointer to a pointer to a `Node` object: `pnode`. It does not return anything. Its purpose is to free `pnode` and set it to `NULL`. Psuedocode for this function was given in the assignment instructions[1].

```
void node_free(Node **pnode)
    if *pnode != NULL then
        free(pnode)
        *pdone = NULL
```

- `void node_print_tree(Node *tree, char ch, int indentation)` : This function takes in a pointer to a `Node` object `tree`, a character `ch`, and an integer `indentation`. It does not return anything. Its purpose is to print the tree for diagnostic and debugging purposes. This function was given in the assignment instructions[1].

```
void node_print_tree(Node *tree, char ch, int indentation)
    if tree == NULL then
        return
    node_print_node(tree right, '/', indentation + 3)
    print (weight = (tree weight))

    if tree left == NULL && tree right == NULL then
        if ' ' <= tree symbol <= '~' then
            print (symbol = (tree symbol))
        else
            print (symbol = (tree symbol as hex))
```

```
    print newline
    node_print_node(tree left, '\\', indentation + 3)
```

pq

- `PriorityQueue *pq_create(void)` : This function does not accept any parameters. It returns a pointer to a `PriorityQueue` object. Its purpose is to allocate a `PriorityQueue` object. If an error occurs, it returns `NULL`.

```
PriorityQueue *pq_create(void)
    PriorityQueue pq = (PriorityQueue*) malloc(sizeof(PriorityQueue))
    if pq == NULL then
        return NULL
    pq list = NULL
    return pq
```

- `void pq_free(PriorityQueue **q)` : This function takes in a pointer to a pointer to a `PriorityQueue` object. It does not return anything. Its purpose is to free the `PriorityQueue` object and set the pointer to `NULL`.

```
void pq_free(PriorityQueue **q)
    if *q != NULL then
        free(*q)
        *q = NULL
```

- `bool pq_is_empty(PriorityQueue *q)` : This function takes in a pointer to a `PriorityQueue` object: `q`. It returns either true or false. Its purpose is to check whether `q` is empty (true) or not (false).

```
bool pq_is_empty(PriorityQueue *q)
    if q list = NULL then
        return true
    else
        return false
```

- `bool pq_size_is_1(PriorityQueue *q)` : This function takes a pointer to a `PriorityQueue` object. It returns true or false. Its purpose is to check whether `*q` contains a single element or not. If it does it returns true. Otherwise, it returns false.

```
bool pq_size_is_1(PriorityQueue *q)
    if q list next = NULL then
        return true
    else
        return false
```

- `bool pq_less_than(ListElement *e1, ListElement *e2)` : This function takes in two pointers to `ListElement` objects: `e1` and `e2`. It returns true or false. Its purpose is to compare the `tree->weight` values of the two `ListElement` objects. It returns true if the weight of the first element is less than the weight of the second element. If the weights of the elements are equal, it compares the `tree-symbol` values and returns true if the symbol of the first element is less than the second. Otherwise, it returns false.

```
bool pq_less_than(ListElement *e1, ListElement *e2)
    if e1 tree weight < e2 tree weight then
        return true
    else if tree weight == e2 tree weight then
```

```
        if e1 tree symbol < e2 tree symbol
            return true
    return false
```

- `void enqueue(PriorityQueue *q, Node *tree)` : This function takes in a pointer to a `PriorityQueue` object and a pointer to a `Node` object. These are `q` and `tree`, respectfully. It does not return anything. Its purpose is to insert a tree into the priority queue. Psuedocode for this function was given in the assignment instructions[1].

```
void enqueue(PriorityQueue *q, Node *tree)
    ListElement new_element = (ListElement*) malloc(sizeof(ListElement))
    new_element tree = tree

    if pq_is_empty(q) then
        q list = new_element
    else if pq_less_than(new_element, q list) then
        new_element next = q list
        q list = new_element
    else
        ListElement current = q list
        while true
            if current next == NULL then
                current next = new element
            else if pq_less_than(new_element, current next) then
                new_element next = current next
                current next = new element
            current = current next
```

- `Node *dequeue(PriorityQueue *q)` : This function takes a pointer to a `PriorityQueue` object `q`. It returns a `Node` object. Its purpose is to remove the queue element with the lowest weight and return it.

```
Node *dequeue(PriorityQueue *q)
    if pq_is_empty(q) then
        print error message
        exit(1)
    Node node = q list
    q list = q list next
    return node
```

- `void pq_print(PriorityQueue *q)` : This function takes in a pointer to a `PriorityQueue` object: `q`. It does not return anything. Its purpose is to print the trees of the queue `q`. This function was given in the assignment instructions[1].

```
void pq_print(PriorityQueue *q)
    assert(q != NULL)
    ListElement *e = q list
    position = 1
    while e != NULL
        if position++ = 1 then
            print "============================================\n")
        else
            printf("--------------------------------------------\n");
        node_print_tree(e->tree, '<', 2);
        e = e next
    print "============================================\n"
```

Huffman Coding

- `uint32_t fill_histogram(FILE *fin, uint32_t *histogram)` : This function takes in a file pointer `fin` and an array of unsigned 32 bit integers `histogram`. It returns an unsigned 32 bit integers. Its purpose is to update the `histogram` array with the number of each of the unique byte values of the input file. It also returns the total size of the input file.

```
uint32_t fill_histogram(FILE *fin, uint32_t *histogram)
    uint32_t filesize
    for i, 0 to i < 256
        histogram[i] = 0;
    ++histogram[0x00];
    ++histogram[0xff];
    while (byte = fgetc(fin)) != EOF
        ++histogram[byte]
        ++filesize
    return filesize
```

- `Node *create_tree(uint32_t *histogram, uint16_t *num_leaves)` : This function takes in an array of unsigned 32 bit integers `histogram` and a pointer to an unsigned 16 bit integer `num_leaves`. It returns a `Node` object. Its purpose is to create and return a pointer to a new Huffman Tree. It also returns the number of leaf nodes in the tree by placing the value in `num_leaves`.

```
Node *create_tree(uint32_t *histogram, uint16_t *num_leaves)
    q = pq_create()
    for i, 0 to i < 256
        if histogram[i] != 0 then
            nd = node_create(i, histogram[i])
            enqueue(q, nd)
            (*num_leaves)++

    while !(pq_size_is_1(q))
        left = dequeue(q)
        right = dequeue(q)
        node = create_node(0, left weight + right weight)
        node left = left
        node right = right
        enqueue(q, node)

    return dequeue(q)
```

- `void fill_code_table(Code *code_table, Node *node, uint64_t code, uint8_t code_length)` This function takes in 4 parameters: a pointer to a `Code` object `code_table`, a pointer to a `Node` object `node`, an unsigned 64 bit integer `code`, and an unsigned 8 bit integer `code_length`. It does not return anything. Its purpose is to traverse a tree and fill in the Code Table for each leaf node's symbol. Psuedocode for this function was given in the assignment instructions[1].

```
void fill_code_table(Code *code_table, Node *node, uint64_t code, uint8_t code_length)
    if node symbol = 0 then
        fill_code_table(code_table, node->left, code, code_length + 1)
        code |= (uint64_t) 1 << code_length
        fill_code_table(code_table, node->right, code, code_length + 1)
    else
        code_table[node->symbol].code = code
        code_table[node->symbol].code_length = code_length
```

- `void huff_write_tree(BitWriter *outbuff, Node *node)` : This function takes in a pointer to a `BitWriter` object and a pointer to a `Node` object. It does not return anything. Its purpose is to write a Huffman Tree. Psuedocode for this function was given in the assignment instructions[1].

```
void huff_write_tree(BitWriter *outbuff, Node *node)
    if node left == NULL then
        bit_write_bit(outbuf, 1)
        bit_write_uint8(outbuf, node symbol)
    else
        huff_write_tree(outbuf, node->left)
        huff_write_tree(outbuf, node->right)
        bit_write_bit(outbuf, 0)
```

- `void huff_compress_file(outbuf, fin, filesize, num_leaves, code_tree, code_table)` : This function takes in 6 parameters:
  - `BitWriter *outbuf` : a pointer to a `BitWriter object`
  - `FILE *fin` : a file pointer
  - `uint32_t filesize` : an unsigned 32 bit integer
  - `uint16_t num_leaves` : an unsigned 16 bit integer
  - `Node *code_tree` : a pointer to a `Node` object
  - `Code *code_table` : a pointer to a `Code` object

  It does not return anything. Its purpose is to write a Huffman Coded file. Psuedocode for this function was given in the assignment instructions[1].

```
void huff_compress_file(outbuf, fin, filesize, num_leaves, code_tree, code_table)
    bit_write_uint8(outbuf, 'H')
    bit_write_uint8(outbuf, 'C')
    bit_write_uint32(outbuf, filesize)
    bit_write_uint16(outbuf, num_leaves)
    while true
        b = fgetc(fin)
        if b == EOF then
            break
        code = code_table[b].code
        code_length = code_table[b].code_length
        for i, 0 to i < code_length
            bit_write_bit(outbuf, (code AND 1))
            code >>= 1
    huff_write_tree(outbuf, code_tree)
```

Huffman Decoding

- `void dehuff_decompress_file(FILE *fout, BitReader *inbuf)` : This function takes in a file pointer `fout` and a pointer to a `BitReader` object `inbuf`. It does not return anything. Its purpose is to read the code tree and then use it to decompress the compressed file. Psuedocode for this function was given in the assignment instructions[1].

```
void dehuff_decompress_file(FILE *fout, BitReader *inbuf)
    Node *stack[64]
    top_of_stack = 0
    type1 = bit_read_uint8(inbuf)
    type2 = bit_read_uint8(inbuf)
    filesize = bit_read_uint32(inbuf)
    num_leaves = bit_read_uint16(inbuf)
    num_nodes = 2 * num_leaves - 1
    Node *node
    for i, 0 to i < num_nodes
```

```
        bit = bit_read_bit(inbuf)
        if bit == 1 then
            symbol = bit_read_uint8(inbuf)
            node = node_create(symbol, 0)
        else
            node = node_create(0,0)
            node right = stack[top_of_stack]
            top_of_stack--
            node left = stack[top_of_stack]
            top_of_stack--
        stack[top_of_stack]
        top_of_stack++
    Node *code_tree = stack[top_of_stack]
    top_of_stack--
    for i, 0 to i < filesize
        bit = bit_read_bit(inbuf)
        if bit == 0 then
            node = node left
        else
            node = node right
        if node left == NULL and node right == NULL then
            break
        fprintf(fout, node symbol)
```

Psuedocode

huff.c

```
main
    fin = NULL
    fout = NULL

    int opt
    opterr = 0
    while (opt = getopt) != -1
        switch opt
            case 'h':
                print usageg message
                return 0
                break
            case 'i':
                fin = fopen(optarg, r)
                if fin = NULL then
                    print error
                    return 1
                break
            case 'o':
                fout = fopen(optarg, w)
                if fout = NULL then
                    print error
                    return 1
                break
            default:
                print error and usage message
                return 1

    if fin == NULL
        print error and usage message
```

```
        return 1
    if fout == NULL
        print error and usage message
        return 1

    uint32_t histogram[256] = {0}

    filesize = fill_histogram(fin, histogram)

    num_leaves = 0
    tree = create_tree(histogram, num_leaves)

    Code code_table[256]
    fill_code_table(code_table, tree, 0, 0)

    BitWriter outbuf
    huff_compress_file(outbuf, fin, filesize, num_leaves, tree, code_table)

    fclose(fin)
    fclose(fout)
    node_free(tree)
    bit_write_close(outbuf)
    return 0
```

dehuff.c

```
main
    fin = NULL
    fout = NULL

    int opt
    opterr = 0
    while (opt = getopt) != -1
        switch opt
            case 'h':
                print usageg message
                return 0
                break
            case 'i':
                fin = fopen(optarg, r)
                if fin = NULL then
                    print error
                    return 1
                break
            case 'o':
                fout = fopen(optarg, w)
                if fout = NULL then
                    print error
                    return 1
                break
            default:
                print error and usage message
                return 1

    if fin == NULL
        print error and usage message
        return 1
    if fout == NULL
```

```
    print error and usage message
    return 1

BitReader inbuf
dehuff_decompress_file(fout, inbuf)

fclose(fin)
flcose(fout)
bit_read_close(inbuf)
return 0
```

Error Handling

- Invalid command line arguments : If an invalid command line argument is given, the usage message will be printed and the program will be terminated.
- Missing arguments : If the program does not receive the `-i` and `-o` arguments, it will print an error message and the usage message and then terminate.
- Invalid input file: If the program cannot open the input file, it will print an error message and the usage message and terminate the program.
- Errors while reading program : If the program reads an `EOF` before expected, it will print and error message and terminate the program.
- Invalid out file : If the output file cannot be opened, an error message will be printed and the program will terminate.

Testing

These programs will be tested using the provided test files: `brtest.c`, `bwtest.c`, `nodetest.c`, and `pqtest.c`[1]. It will also be tested using `valgrind` to ensure that there are no memory leaks. The program will also be compiled with `scan-build`. The `diff` command will also be used to compare these programs' output to the output of the reference binary.

Results

References

[1] Dr. Keery Veenestra and TAs. Assignment 8: Huffman coding. `https://git.ucsc.edu/cse13s/fall-2023-section-01/resources`, Fall 2023.