Assignment 6 Design

Brian Quach

March 2, 2022

1 Introduction

The program compresses and decompresses messages using Huffman encoding. Symbols are processed and are assigned frequency values which determines a the number of bits they are assigned to. Symbols with higher frequencies will have less bits while those with lower frequencies will require a higher number of bits. This compresses by lowering the total number of bits for a message when compared to the 8 bit ASCII format. The Huffman tree used to assign the bits to each symbol is dumped to the compressed file along with the compressed version of the message. The decompressor then reconstructs the same Huffman tree and then decompresses the message.

2 Necessary Files

1. encode.c: encodes a message using a Huffman tree and outputs the Huffman tree and encoded message to a file.
2. decode.c: Decodes a compressed message by reading a Huffman tree from the file and then decompressing the message using said tree.
3. defines.h: contains macro definitions used in the program.
4. header.h: Defines the header struct.
5. node.h: Declares node struct and relevant functions.
6. node.c: Defines the node struct and implements its relevant functions.
7. pq.h: Declares priority queue struct and relevant functions.
8. pq.c: Defines the priority queue struct and implements its relevant functions.
9. code.h: contains the declaration for the code table struct and its relevant functions.
10. code.c: Defines the code struct and relevant functions.
11. io.h: Declares functions and global variables for input output such as reading bits.
12. io.c: Implements functions and defines global variables for io.
13. stack.h: Declares the stack struct and its appropriate functions.

14. stack.c: Implements stack functions and defines the stack ADT.
15. huffman.h: Declares functions for using huffman trees.
16. huffman.c: Implements functions in huffman.h
17. Makefile: Builds the program, cleans any compiler made files, and formats any .c or .h files.
18. DESIGN.pdf: Describes the program as well as how the program was made with supporting pseudo code.
19. README.md: Describes how to build the program, how to run the program, valgrind and scan build errors, and command line arguments.

3 Node.c

Node.c implements the node interface where each node contains a symbol, a frequency, and two pointers to other nodes. The actual struct of the node is defined in node.h such that it can be accessed in other functions without having to create access and setter functions.

```
Node *node_create(uint8_t symbol, uint64_t frequency) {
    n = malloc a node onto the heap
    set the symbol of n to symbol
    set the frequency of n to frequency
    left child of n is null
    right child is also null
}

void node_delete(Node **n) {
    free the node using the pointer pointed to by n
    set that pointer to null
}

Node *node_join(Node *left, Node *right) {
    call node_create to create the parent
    where sympbol should be '$' and the frequency is
    frequency of left + right
    return the parent
}

void node_print(Node *n) {
    print the symbol and frequency of n
    print the symbol and frequency of n's children and label them
    accordingly
}
```

4 pq.c

pq.c defines a priority queue and implements the functions it requires. This ranges from construction, deconstruction, access of variables and so on.

```
struct PriorityQueue {
    capacity
    an array of pointers to nodes
    size
}

PriorityQueue *pq_create(uint32_t capacity) {
    malloc a priority queue onto the heap
    malloc an array of node pointers onto the array
    set the array of the priority queue to the allocated array
    set the capacity of the priority queue
    set the size to 0
    return the priority queue
}

void pq_delete(PriorityQueue **q) {
    for each node node_delete using q+i where i is the iterating value that ranges from 0 to siz
    free q
    set q to null
}

bool pq_empty(PriorityQueue *q) {
    if the first node in the array is null then its empty
}

bool pq_full(PriorityQueue *q) {
    if the node at capacity - 1 is not null then its full
}

uint32_t pq_size(PriorityQueue *q) {
    return the size characteristic of q
}

bool enqueue(PriorityQueue *q, Node *n) {
    if q is empty {
        the 0th index of the array is n
        add 1 to the size of q
        return true
    }

    if the q is full {
```

```
        return false
    } else {
        j = size - 1
        while the frequency at j in the array is less than the frequency of n then
        the array at j + 1 = the node at j
        j = j - 1

        the array at j + 1 = n
        add 1 to the size of q
    }
    return true
}

bool dequeue(PriorityQueue *q, Node **n) {
    if q is empty return false
    n = the node at size - 1 in the array
    subtract 1 from the size
    return true
}
```

5 Codes

Codes establishes the implementation of huffman codes which comprises of a stack of bits. The length is 256 bits long and is stored within a byte array of length 32.

```
code_init {
    make an array of length 256
    set top to 0
    zero out the array
}

code_size
    return top

code_empty
    true if top is 0 otherwise false

code_full
    if top is greater than or equal to 256 then true

code_set_bit( uint32_t i)
    if i is greater than 256 then quit
    byte = i divided by eight
    bit = i mod eight
```

```
    set the bits index byte to the array value or'd by 1 left shifted bit times

code_clr_bit(i)
    similar to set bit
    however return the array value and'd with the inverse of 1 left shifted bit times

code_get_bit(i) {
    similar to set bit
    however return array value and'd by 1 left shifted bit times and then right shifted bit time

code_push_bit(i) {
    set the bit at top to the one given
    increase top by 1
}

code_pop_bit
    get bit at top
    return the bit
    subtract one from top

code_print
    iterate through bit array
    print the get bit for every bit less than top
}
```

6 io

```
set up static variables such as read and write index and buffers

read_bytes {
    set bytes read to 0
    while the bytes read are less than than specified and the bytes read from infile is
    greater than 0 add the bytes read to the static bytes_read variable
    return the static bytes read
}

write_bytes {
    bytes written = 0
    while bytes written is less than specified and bytes being written is greater than 0
    add the number of bytes read to the static number written
    return the number written
}

read bit{
bytes = 0
```

```
if read index / 8 equals bytes then read in another block of bytes and set read index to 0
nbyte is byte pointed at by read index
nbit is the bit pointed at by read index
get the value of the bit from the byte by anding with 1 left shifted bit times and then right sh

add one to the read index
return the bit value
}

write code {
    if the write index byte is equal to block
    write out the block bytes using write_byte
    set the write index to 0

    for each bit in the code
    get the bit and then change the bit pointed at by write index to that bit
    this can be done by using the bit manipulations used in the codes section
    add one to the write index
}

flush codes {
    while write index is inside a byte
    set the value of the bits to zero
    essentially zero out the rest of the bits of the last byte

    write out the rest of the bytes in the buf
}
```

7 huffman

```
build tree {
make a priority queue
add every non zero frequency symbol in the histogram to the priority queue

while there are at least two nodes in the pq
    dequeue the top 2 with the first being left and the second being right
    join them with node_join
    enqueue the parent on the pq

dequeue the node which is the root of the tree and return it
}

build_tree {
create code c
if the node exists then
```

```
    if the node is a leaf then set the index of table that corresponds with the node's
    symbol to the code
else
    push a 0 onto the code
    build_tree with the left child
    pop a bit from code

    push a 1 onto the code
    build_tree with the right child
    pop a bit from code
}

dump tree {
    if the node exists then
        dump_tree of the left child
        dump tree of the right child

        if the node is a leaf then write L followed by the node's symbol to outfile
        else write an I
}

rebuild tree {
    create a stack
    read through the tree dump array
    if its an L then push the value of the next index onto the stack and increment
    the index by an extra one to skip the next symbol

    if its a I then pop 2 nodes with the first being right and the second being left
    push the parent of the two onto the stack with node_join

    the last node on the stack is the root
    return the root and delete the stack
}

delete tree {
    post order traversal deletion
    if the node exists delete the left and right child and the the node itself
}
```

8 Encode

```
usage function to print command line options and synopsis

main {
    getopt loop with switch case to parse through command line arguments
```

```
    infile is default stdin i.e 0
    outfile is default stdout i.e 1

    if the files are specified then use open the create the file descriptors

    create a histogram with ALPHABET elements
    set the first and last indexes to value 1

    read bytes from the infile and increase the frequency of the symbol in histogram
    when read

    calculate the number of unique symbols by counting every non zero index in hist

    build a tree using build_tree(hist)

    create a code table of alphabet elements
    use build_codes(root, table) to fill the code table

    create a header
    use fstat to read in desire fields and fill in header traits

    write the header to outfile

    export the tree to outfile using dump_tree(outfile, root)

    set the file descriptor back to the beginning of infile

    until the end of file read in 1 byte at a time
    write the code of the read byte to outfile using write_code

    flush the codes left in the write buffer

    if v is specified then get the file size of outfile and infile using fstat
    and perform the necessary calculations

    delete the tree
    free the histogram
    close infile and outfile
}
```

9 decode

```
usage function to print command line arguments and synopsis
```

```
main {
    getopt loop to parse command line arguments

    infile is default 0
    outfile is default 1
    if the files are specified then use open

    read in the header from infile
    if the magic number in header isn't MAGIC then exit the program

    create a uint8_t buffer of tree_size read from the header
    read in tree size bytes and populate the buffer

    create a tree with the read bytes using rebuild_tree

    while the number of decoded symbols is less then the original file size
    read a bit from the infile
    if the bit is a one then walk to the right on the tree and if it is a one then
    walk to the left
    when a leaf is reached write the leaf's symbol to outfile and then return to the root
    every time a symbol is written add one to the decoded variable

    if v is specified then print the decompression statistics similar to encode

    delete the tree
    close the infile and outfile
}
```

10 Explanations

Encode effectively works by first iterating once over the infile to create a tree which is roughly ordered based upon the frequencies of the symbols in infile. The tree is then traversed to give codes to each symbol node where symbols with lower probability will get longer codes due to their lower position in the tree. The infile is then read again where when a symbol is read, its code is written to outfile therefore creating a compressed version of the message.

Decode works by first reading in the tree dump and header from the compressed file. The tree is then reconstructed. Then by reading in a bit at a time from infile, the tree is traversed until a leaf node is reached and the symbol of the node is then written to outfile. Upon completion, the outfile should be a 1 to 1 copy of the original message / file.

11 Credit

1. Used the pseudo code given in the assignment pdf such as the build and dump functions

12 Errors

1. Scan build returns a warning for dereferencing a null pointer however this is expected and necessary in order to detect leaves when traversing trees.