ASGN 8 - Huffman

Isabella Shapland

CSE 13S – Fall 2023

Purpose

The purpose of this program is to code and encode text files using the huffman codes. Huffman code checks the most frequent letter in a text file, then it compresses the text by mapping binary numbers (0 or 1, depending on the path you take to traverse the binary tree) to each letter in a sentence (using a binary tree based on the character frequency.) The program has two different parts, huff and dehuff. One encodes a text or binary file and the other decodes. This assignment probably took me more than 30 hours spread over a week and a half but here it is!

How to Use the Program

When you run the program, it must use the flags -i, -o and optionally -h. The program takes text files as input, using the compiler flag -i and stating the file path after. You must also include an output file, after the -o flag, where you include the output file path after. The last flag is the -h flag, which if included will print the usage statement. There is one more flag, which controls

```
Usage: huff -i infile -o outfile
       huff -h

Usage: dehuff -i infile -o outfile
       dehuff -h
```

It ignores any other files or flags in the command line and will not run the program. The MAKEFILE: The makefile runs multiple different files, first being the program itself, huff/dehuff. Calling make huff will run huff, including the many other files that make up the program. bitreader.c, for creating the bitreader which reads bits one at a time, 8 bits at a time, 16 or 32, and (not included in this specific make call but i explain it here anyways) bitwriter.c for writing bits 1, 8, 16, or 32 at a time, pq.c for making the priority queue that helps make the binary tree, node.c that creates and prints the tree, and test files for each c file. The other way that the makefile runs is with the dehuff program dehuff.c, and running make dehuff will compile that file as well as the specified bitwriter, node, and pq files. There are a few other execs in the makefile, namely the given test files. Running Make pqtest will make pq.c/.h, node.c/.h, and pqtest(main). Make nodetest will run node.c/.h and pq.c/.h files. Make bwtest and make brtest makes bitwriter.c/.h, bitreader.c/.h files respectively. Running make all will compile all of the files, running make clean will remove all object (binary) files, and make format will clang format all files. Headers included are pq.h and node.h, bitwriter.h and bitreader.h. Here are the compiler flags and lflags:

```
    CFLAGS = -Werror -Wall -Wextra -Wconversion -Wdouble-promotion
    -Wstrict-prototypes -pedantic
    LFLAGS = -lm
```

Program Design

Are you ready? There's a lot.

Data Structures

Our data structures starts with the bitreader, which is a DT that has three fields: the underlying_stream which is the file to be read from, uint8_t (unsigned 8 bit integer) byte, which is the current byte that bits are being read from, and uint8_t bit_position which is the bit (of the byte) that we are focusing on for reading. The reader has several functions which will be detailed in the functions section, but overall they open and close the bitreader and the associated file, and read a bit, 8 bits, 16 bits, or 32 bits.

Bitwriter is along the same vein as the reader with the same fields. It contains the underlying_stream which is the file to be written to, uint8_t (unsigned 8 bit integer) byte, which is the current byte that bits are being written to, and uint8_t bit_position which is the bit (of the byte) that we are focusing on for writing. The reader has several functions which will be detailed in the functions section, but overall they open and close the bitwriter and the associated file, and write a bit, 8 bits, 16 bits, or 32 bits.

The next one is the priority queue, which is a linked list. It includes another structure (DT) called List Element. Each list element has a pointer (to the next element) and a pointer to the tree (the list element with the character and frequency, see node for more information.) Priority queues list based on weight, which is a total of the frequencies of all the letters in a given tree.

Node is the final datatype. It has fields (8 bit) symbol and code length, (32 bit) weight, (64 bit) code and two node pointers left and right. There are three functions that handle nodes, first of which being node create, node free and print tree.

there is also a secret DT called code, its used in the fill code table function within the huff main. It has fields 64 bit int doce and 8 bit int code length.

Algorithms

There are two main functions: HUFF:

```
define histogram, bool help message (Code struct is also defined)
earlier in the file, before all the functions
int option = 0
input file = stdin
outputfile = NULL

while getopt
    switch(option)
        case i:
            if optarg !=NULL
            input = fopen(optarg 'r)
                if input == NULL
                    error
                break
            else
                error
        case h
            h = true
            break
        case o
        if optarg != NULL
            output = optarg
            break
        else
            error
        break
    case ?
        error
if input ===stdin
    h = true
if output == NULL
    h = true
```

```
if h
    print help message
allocate code table: 256 x size of code
bw = bit_write_open(output)
check if bw is null
    error
filesize = fill_histogram
num_leaves= 0
node tree = create tree (histogram, &num_leaves)
if tree==NULL
    fclose(input)
    free code table
    bit_write_close(&bw)
fill_code_table(code_table, tree, 0,0)
if fseek((input, 0, seek_set)!=0
    fclose(input)
    free code table
    node free tree
    bit_write_close(&bw)

huff_compress_file(bw, input, filesize, num_leaves, tree, code_table)
fclose(input)
bit write close
node_free(&tree)
free(code_table)
return 0
```

DEHUFF:

```
define bool h as false
int option = 0
input file = stdin
outputfile = NULL

while getopt
    switch(option)
        case i:
            if optarg !=NULL
            input = fopen(optarg 'r)
            if input == NULL
                error
            break
        else
            error
    case h
        h = true
        break
    case o
        if optarg != NULL
            output = optarg
            break
        else
            error
        break
    case ?
        error
if input ===stdin
    h = true
if output == NULL
    h = true
```

```
if h
   print help message
BitReader *br = bit read open(input)
if br is null
   error
file outfile = fopen output 'wb'
decompress the file (outfile, br)
bit read close (&br)
fclose(outfile)
return 0
```

Function Descriptions

0.0.1 bitreader.c:

bit_read_open()

- takes a const char *filename (pointer) as input
- returns bitreader pointer

```
allocate memory for bitreader
open the filename for reading (fopen)
if the file or the bitreader == null
    return null
set the file bitreader underlying stream
bitreader byte = 0
bireader bit position == 0;
return bitreader
```

bit_read_close()

- takes the bitreader as input
- returns nothing, closes the bitreader

```
if the bitreader != null
    result = fclose underlying stream
    free the bitreader
    null the pointer to bitreader
    if result !=0
        file closing issue
        exit
```

bit_read_bit()

- takes bit reader pointer buf
- returns the integer that the bit is (0 or 1)
- reads a bit from the byte, then shifts back so the int can be processed as a 0 or 1

```
if bit position is > 7
    int byte = fgetc(underlying stream)
    if byte is the eof
        exit
    bitreader->byte = (uint8_t)byte
    bit positon =0
```

```
        *this part is reading a new byte*
    mask = 1 << bit position
    mask &= buf->byte
    mask >> bitposition (sets back to 0 or 1)
    return mask
```

bit_read_uint32()

- takes buf
- returns nothing
- this function reads a 32 bit int by calling the bit_read_bit function 32 times.

```
word = 0x00000000;
for loop from 0 to 32 exclusive
    b = bit read bit
    set ith bit of word to the bir b
return word
```

bit_read_uint16()

- takes buf
- returns nothing
- this function reads a 16 bit int by calling the bit_read_bit function 16 times.

```
word = 0x000000;
for loop from 0 to 16 exclusive
    b = bit read bit
    set ith bit of word to the bit b
return word
```

bit_read_uint8()

- takes buf
- returns nothing
- this function reads a 8 bit int by calling the bit_read_bit function 8 times.

```
word = 0x00000000;
for loop from 0 to 8 exclusive
    b = bit read bit
    set ith bit of byte to the bit b
return byte
```

BITWRITER.C:

bit_write_open()

- takes a const char *filename (pointer) as input
- returns bitwriter pointer

```
    allocate memory for bitwriter
    open the filename for writing (fopen)
    if the file or the bitwriter == null
        return null
    set the file bitwriter underlying stream
    bitwriter byte = 0
    bitwriter bit position == 0;
    return bitwriter
```

bit_write_close()

- takes a pointer to (the pointer of) the bitwriter as input
- returns nothing, closes the bitwriter

```
if the bitwriter != null
    if the bit position isnt 0
        put the last byte
        return null if the byte cant be put
    if fclose underlying stream is NULL
        return null
    free the bitwriter pointer
    null the pointer
```

bit_write_bit()

- takes bit writer pointer buf, uint8_t bit (just a 0 or 1)
- returns nothing
- writes a the given from to the byte, and writes the byte to the file if all bits are full

```
if bit position is > 7
    int result = fputc(byte, underlying stream)
    if result is the eof
        error exit
    byte = 0
    bit positon =0
    *this part is reading a new byte*
bit = bit << bit position
set bit position of buf->byte to bit
bitposition+=1
```

bit_write_uint32()

- takes buf, byte
- returns nothing
- this function writes a 32 bit int by calling the bit_write_bit function 32 times.

```
for loop from 0 to 32 exclusive
    32 bit int mask = 1 shifted i times
    set bit i of mask to that i bit of byte
    shift mask back 1 times to make it 0 or 1
    bit_write_bit(mask)
return word
```

bit_write_uint16()

- takes buf, byte
- returns nothing
- this function writes a 16 bit int by calling the bit_write_bit function 16 times.

```
for loop from 0 to 16 exclusive
    16 bit int mask = 1 shifted i times
    set bit i of mask to that i bit of byte
    shift mask back 1 times to make it 0 or 1
    bit_write_bit(mask)
return word
```

bit_write_uint8()

- takes buf, byte
- returns nothing
- this function writes a 8 bit int by calling the bit_write_bit function 8 times.

```
for loop from 0 to 32 exclusive
    8 bit int mask = 1 shifted i times
    set bit i of mask to that i bit of byte
    shift mask back 1 times to make it 0 or 1
    bit_write_bit(mask)
return word
```

NODE.C: **node_create()**

- takes 8 bit int symbol and 32 bit int weight
- returns node pointer
- allocates memory for and creates a new node

```
allocate space for a node nptr
if nptr is null return NULL
nptr code = 0;
nptr codelength =0
node->symbol = symbol
node->weight = weight
return nptr
```

node_free()

- takes a double pointer to node (**node)
- returns nothing
- frees up all of the allocations and makes the pointer null

```
if the single pointer *node isnt already null
    (call node free recursively on left and right children)
    node_free(&(*node)->left)
    node_free(&(*node)->right)
    then free the *node itself
    set *node to null
```

node_print_node()

- takes node pointer tree, char ch, and int identation
- returns nothing
- prints a tree!

```
if the tree is null //base case
    return
call node_print_node on the right child, /, indentation +3
print weight = (weight of node)
if both sides of the tree are null
    if ' ' is less than tree symbol & tree symbol less than '~'
        print the char and the tree symbol
        //this part is just testing if its a printable char
    else
        print it in hex

print newline
call node_print_node on the left child \\ indentation +3

the second part to this function is another function,
node_print_tree, which just calls the fucntion with
parameters *tree < 2
```

PRIORITY QUEUE: Its important to note that priority queue has two datastructures of its own, one of which being a ListElement, which has a list element pointer and node *tree as fields.

The other struct is priority queue which has a field ListElement *list which is a pointer to the first element of the list.

pq_create()

- takes nothing
- returns priority queue pointer
- creates and allocates memory for a priority queue!

```
allocate memory for pq
if pq is null
    return null
return pq pointer
```

pq_free()

- takes pq pointer **q
- returns nothing
- frees the memory that was allocated for pq

```
if *q isnt null
    free *q
    *q =null
```

pq_is_empty()

- takes pq pointer
- returns boolean
- if it is empty, return true if it isnt return false

```
if pq->list == null
    return true
return false
```

pq_size_is_1()

- takes q pointer
- returns boolean
- if it is size is 1, return true if it isnt return false

```
if q->next == null && q->list
    return true
return false
```

pq_less_than()

- takes listelement pointer e1 and listelement pointer e2
- returns boolean
- if the first elements weight is greater return true if not return false

```
if e1->weight < e2->weight
    return true
else if e1->weight == e2->weight
    if e1->symbol < e2->symbol
        return true
    return false
else
    return false
```

enqueue()

- takes pq pointer q and node pointer tree
- returns nothing
- Puts a tree into the right place in the queue depending on the weight (or symbol, see is less than()) function

```
allocate space for listelement new
if new == NULL
    return
set the tree field of new to the tree parameter of the function
if the queue is empty
    q->list = new
else if size is 1 and pq_less_than(new, q->list)
    new->next = q->list
    q->list = new //first element
else
    e2 = q->list
    e3 = e2->next
    while(e3!= NULL)
        if new is less than e3
            e2->next = new //the next element is new
            new->next = e3 //put e3 after new
            return
        else
        e2 = e3 //move to the next element (for insertion)
        e3 = e3->next //next element (for comparison)
    if it reaches the end without inserting put new at the end
    new->next = NULL //and put NULL at after new
```

dequeue() As I am currently writing this, there are issues with the return values of this code. It passes the pqtest file provided but "returns NULL" when it shouldn't be (?) in the pipeline.

- takes pq pointer q
- returns node pointer
- takes the lowest element out of the list

```
if(q is NULL or q->list is null)
    return NULL
else
    list element temp = q->list
    temp tree is q->list tree
    node returnval = q->list->tree
    free the temp list element
    return returnval
```

pq_print()

- takes pq pointer q
- returns nothing
- prints the tree for diagnostic purposes

```
make sure q isnt null
list element *e = q->list
int position = 1
while e is not null
    if position++ == 1
        print a long == line
    else print a long -- line
    call node_print_tree with parameters e->tree, '<', 2
    e = e->next
print a long == line
```

HUFF.C: Huff has a data stucture too, called Code. The fields of this stucture are 64 bit int code and 8 bit code length. **fill histogram()**

- takes file pointer and uint32t *histogram
- returns uint32_t
- updates a histogram array with the value of the number of each unique byte in the array

```
int ch = 0
uint32_t filesize =0
++histogram[0x00]
++histogram[0xff]
while (result = fgetc) != EOF
    ++histogram[result]
    filesize++
return filesize
```

create_tree()

- takes histogram pointer, uint16_t *num_leaves

- returns node pointer
- creates a binary tree that is sorted by byte frequency

```
pq = pq_create()
for loop from i to 256 in histogram
    if histogram[i] > 0
        node = node_create(h, h->symbol)
        numleaves++
        enqueue node in pq
while priority queue size is not 1 && not empty
    dequeue into left
    dequeue into right
    node = nodecreate(0, left->weight+right->weight)
    node->left = left
    node-> right = right
    enqueue node
return dequeue
```

fill code table()

- takes code *codetable, node *node, 64 bit code, 8 bit code length
- returns node pointer
- traverses the tree and creates the code table (which is correlates to letters based on frequency)

```
if node->left == NULL
    fill_code_table(code table, node->left, code, code_length+1
if node->right == NULL
    code |= (uint64) 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
```

huff compress file()

- takes bitwriter *outbuf, buffer *inbuf (for reading) filesize num leaves code tree and code table
- returns nothing
- writes the huffman coded file

```
write h and c to outbuf
write filesize t outbuf
write num leaves to outbuf
write_tree(code_tree)
while 1
    b = fgetc(inbuf)
    if b reacher eof break
    code = codetable[b]->code
    codelength = code_table[b]->codelength
    for loop from 0 to code_length
    write bit (code & 1) to outbuf
    code >> =1

huff_write_tree()
```

```
here is the function that writes the code tree:
it takes parameters node and outbuf
    if node->left is NULL
        write bit 1 to outbuf
        write 8 bit int node-> symbol to outbuf
    else ( the node is internal)
        huff_write_tree(node->left)
        huff_write_tree(node->right)
        write bit 0 to outbuf
```

DEHUFFMAN DECODING:

dehuff_decompress_file()

- takes bitwriter *outbuf, buffer *inbuf (for reading) filesize num leaves code tree and code table
- returns nothing
- decompresses the huffman coded file for human reading

```
read 8 bit int type1 from inbuf
read 8 bit int type2 from inbuf
read uint32_t filesize from inbuf
assert type1 = h
assert type2 = c
num_nodes = 2*num_leaves - 1
Node *node
for loop from 0 to num_nodes
    read one bit from inbut
    if that bit == 1
        read 8 bit symbol
        node = node_create(symbol, 0)
    else
        node_create(0,0)
        node->right = stack_pop()
        node->left = stack_pop()
    stack_push(node)
node *code_tree = stack_pop()
for loop from 0 to filesize
    node currnode = code_tree
    while true:
        read one bit from inbuf
        if bit == 0
            currnode is left node
        else
            currnode is the right node
    write 8 bit int node->symbol to fout
free code tree
```

0.1 Error handling and testing

If any step of any of the functions returns NULL or EOF then the program handles it cleanly. It will print a descriptive error to stderr. I test my files with the given test files and alter my makefile as necessary to run them. There are also test files given, pqtest, nodetest, bwtest and brtest, and my program passes all of them. There is also a file to run all of the tests, including the ones examining the difference between my output and the reference files, and my program passes them too.

Results

12/07/23 My program has been getting flagged for enqueue-dequeue in the pipeline which really slowed me down. At first it got flagged for returning NULL when it shouldn't be. I fixed this by fixing a bug in my enqueue where I needed to correctly put new_element before q-¿list. I now have a error in the pipeline saying that I returned the incorrect node weight. My pq.c still passes pqtest. I'm not sure yet where I am going wrong. My code also has trouble passing valgrind, in huff it has problems passing due to a conditional jump or move being from an uninitialized value, and my dehuff currently has a segmentation fault. All the other files pass the given tests mentioned in the previous section.

12/08/23 I fixed all of my bugs! The issue was that I was only putting elements in the front of the queue if the size of the queue was 1. It would put elements in the second spot if the new element was smaller than all of the other elements in the queue. After fixing that issue, I was able to move on to huff.c At first I had the most ridiculously long valgrind error message, and it was all because I was only allocating space for one code in the code table. After fixing that issue, I passed valgrind, my huff read all the files correctly and I was finally able to move on to dehuff. Dehuff was segmentation-faulting, and after HOURS of looking at the code, I realized I was incrementing the top of stack int after adding the element to the stack. I would also decrement it after popping so the push-code ordering was incorrect. After fixing that and adding the free(code_table) at the very bottom of the dehuff decompress file, (it was one of the loops before, accidentally,) it passed everything! Thank god.

Thank you graders for all of your hard work, this class was fun.

References