asgn 6: DESIGN - Huffman Coding

Eric Wang

June 2, 2023

Purpose:

We can use Huffman Coding to compress a file. From the lectures, the key takeaway from Huffman coding is to look at the most common bytes so that we can compress it for it to use fewer than 8 bits. And less common bytes will use more than 8 bits to compensate. The definition of compression is to use fewer total bits to represent the same file. For the assignment, we are to write a data compressor using Huffman Coding, the decompressor is already given to us, we only have to write the compressor part. Much like the previous assignments, the main function consists of command line options, where -i specifies the name of the input file, -o specifies the name of the output file, and -h prints the help message.

How to use the Program:

Download the required .c and .h files: bitwiter.c, bitwriter.h, io.h, io(aarch64/x86_64), huff.c, Makefile, node.c, node.h pq.c pq.h, has well as the required bmp test files.
Once you have the required files, run make in your unix terminal and get the binary file.
Next type in this syntax './huff -i (input file name - the original file) -o (output file name - new outfile)"
To decompress file, use the dehuff binary provided using the outfile as the -i and a new outfile.
To view the syntax again, use -h to display the help message.
(Sample results in the results section)

Pseudocodes:
bitwriter.c

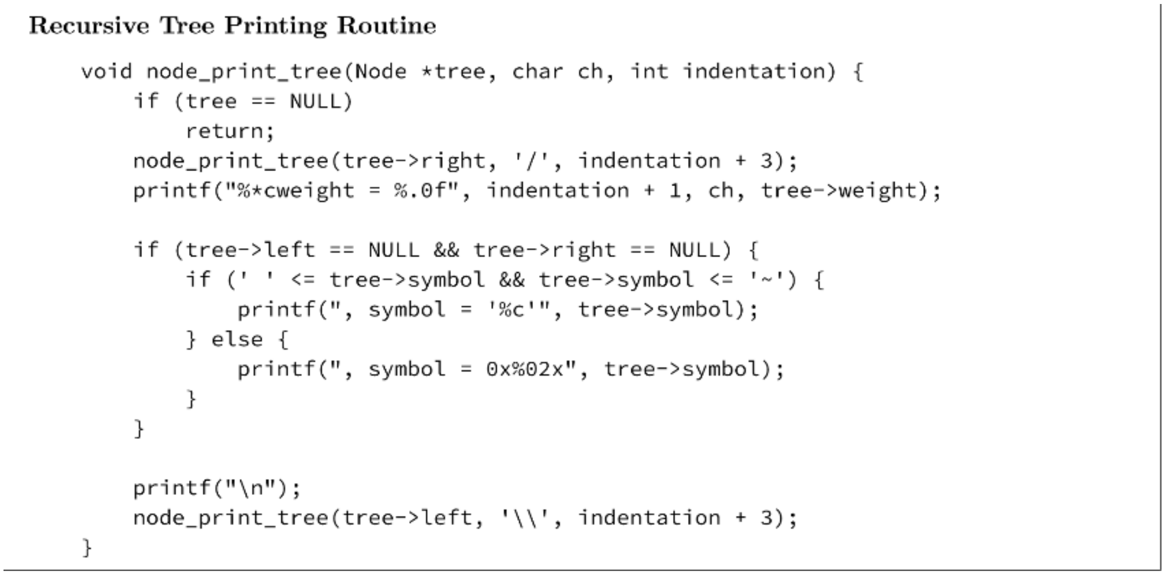
```
Def *bit_write_open(filename)
    Allocate a BitWriter object
    Create buffer using write_open
    buf->underlying_stream = underlying_stream
    Return a pointer to buf
    Report error if unable to perform prior steps

Def bit_writer_close(**pbuf)
    If bit_position > 0
        Write byte to underlying_stream
    write_close()
    Free *pbuf
    Set *pbuf to NULL
```

```
Def bit_write_bit(buf, x)
    If bit_position > 7
        Write byte to underlying_stream using write_uint8()
        Clear byte to 0x00
        Clear bit_position to 0
    If x & 1 then byte |= (x & 1) << bit_position
    ++bit_position

Def bit_write_uint8(buf, x)
    For i = 0 to 7
        Write bit i of x using bit_write_bit()

Def bit_write_uint16(buf, x)
    For i = 0 to 15
        Write bit i of x using bit_write_bit()

Def bit_write_uint32(buf, x)
    For i = 0 to 31
        Write bit i of x using bit_write_bit()
```

node.c

```
Def node_create(symbol, weight)
    Create node and set symbol and weight, return node pointer

Def node_free(**node)
    Free node and set to null
Def node_print_tree(tree, ch, indentation)
```

(below are the code for recursive tree printing routine from Dr Veenstra's lab doc)



Recursive Tree Printing Routine

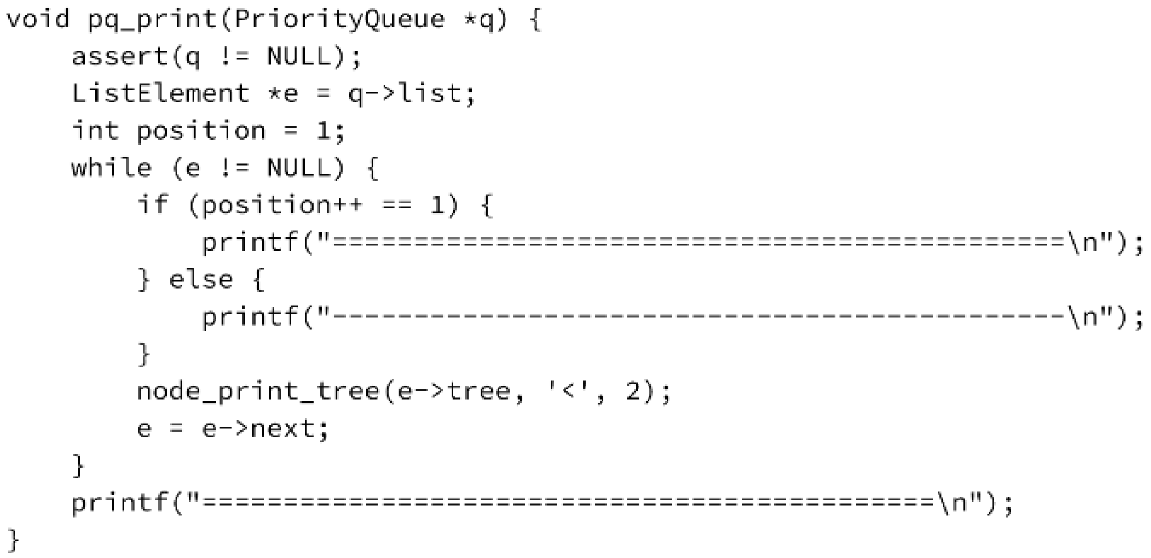
```c
void node_print_tree(Node *tree, char ch, int indentation) {
    if (tree == NULL)
        return;
    node_print_tree(tree->right, '/', indentation + 3);
    printf("%*cweight = %.0f", indentation + 1, ch, tree->weight);

    if (tree->left == NULL && tree->right == NULL) {
        if (' ' <= tree->symbol && tree->symbol <= '~') {
            printf(", symbol = '%c'", tree->symbol);
        } else {
            printf(", symbol = 0x%02x", tree->symbol);
        }
    }

    printf("\n");
    node_print_tree(tree->left, '\\', indentation + 3);
}
```

pq.c

```
Def pq_create(void)
    Allocate a new priorityqueue and return a pointer using calloc()

Def pq_free(**q)
    Free the priorityqueue and set it to NULL

Def pq_is_empty(*q)
    Return true if the priorityqueue is empty.

Def pq_size_is_1(*q)
    Return false if q is empty
    Return true of the next element of queue’s list is NULL

Def pq_less_than(*n1, *n2)
    If weight of n1 is less than weight of n2, return true
    If weight of n1 is greater than weight of n2, return false
    Return n1’s symbol is less than n2’s symbol

Def enqueue(*q, *tree)
    Allocate new listelement e, and set e->tree = tree
    If queue empty
        Set q->list = e
    If weight of the new tree is less than the weight of the head
        Set e->next to e->list
        q->list = e
    If new element goes after one existing element
        Starting q->list, go through the list until the next field is NULL
        Insert e after the queue element

Def dequeue(*q, **tree)
    If queue is empty return false otherwise set *tree to e->tree
    Call free on e, return true
```

Def pq_print(*q)

    (code below is from dr veenstra's lab document)



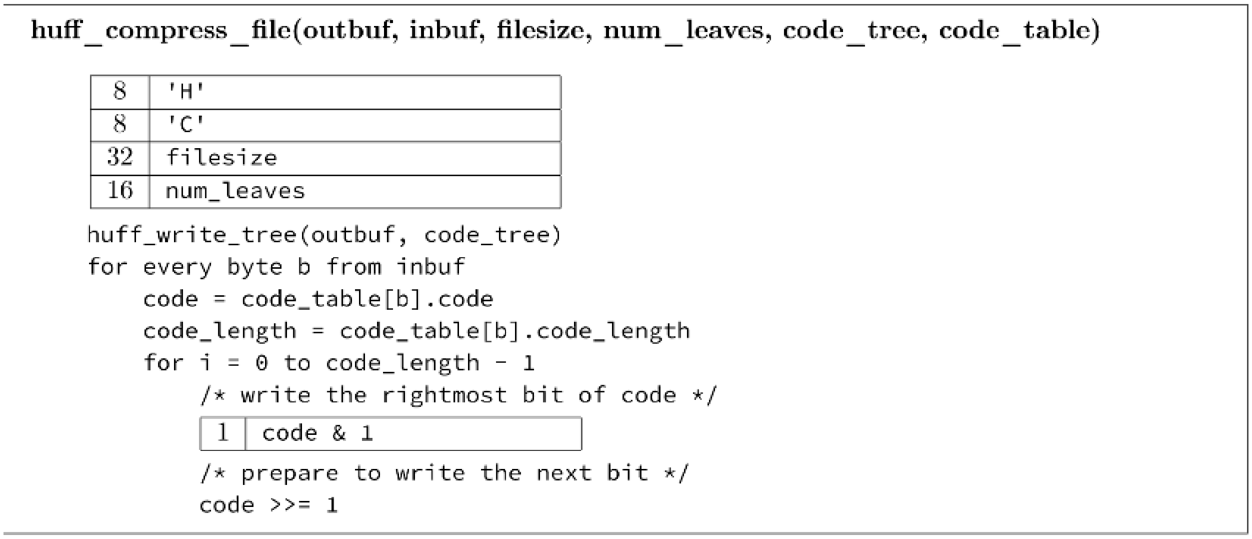
```
void pq_print(PriorityQueue *q) {
    assert(q != NULL);
    ListElement *e = q->list;
    int position = 1;
    while (e != NULL) {
        if (position++ == 1) {
            printf("==============================================\n");
        } else {
            printf("----------------------------------------------\n");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

huff.c

```
Def fill_histogram(inbuf, histogram)
    Create uint64 variable filesize and set it to 0
    For i to 256
        Set histogram to 0

    Create uint8 variable x
    While read_uint8 is true
        Increment histogram
        Increment filesize

    Increment histogram by 0x00
    Increment histogram by 0xff
    Return filesize

Def create_tree(histogram, num_leaves)
    Create and fill a priority queue
    Run huffman coding algorithm
    Dequeue the queue's only entry and return

Def fill_code_table(code_table, node, code, code_length)
    If node is internal
        fill_code_table(code_table, node->left, code, code_length + 1);
        Code |= 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;
```

Def huff_compress_file()

(below are the pseudocode in dr veenstra’s lab document)



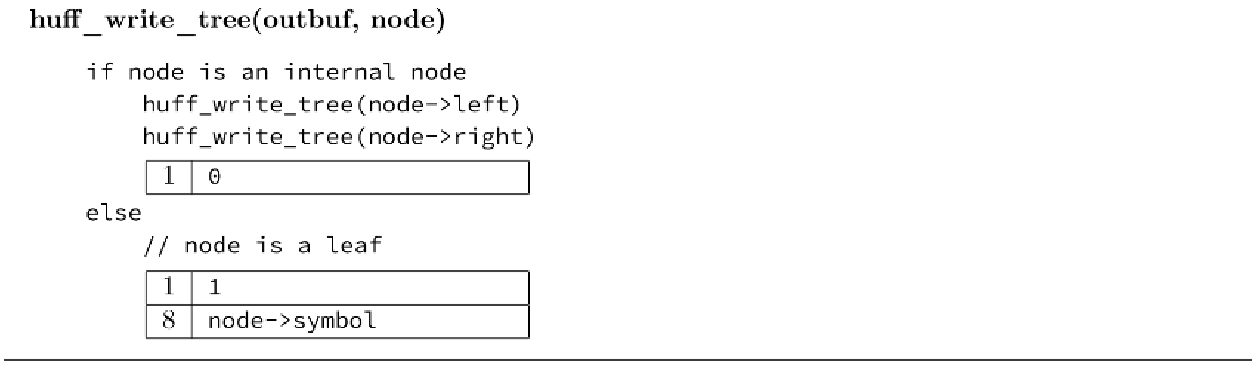
huff_compress_file(outbuf, inbuf, filesize, num_leaves, code_tree, code_table)

8	'H'
8	'C'
32	filesize
16	num_leaves

```
huff_write_tree(outbuf, code_tree)
for every byte b from inbuf
    code = code_table[b].code
    code_length = code_table[b].code_length
    for i = 0 to code_length - 1
        /* write the rightmost bit of code */
        | 1 | code & 1 |
        /* prepare to write the next bit */
        code >>= 1
```

Def huff_write_tree(outbuf, node)

(below are the pseudocode in dr veenstra’s lab document)



huff_write_tree(outbuf, node)

```
if node is an internal node
    huff_write_tree(node->left)
    huff_write_tree(node->right)
    | 1 | 0 |
else
    // node is a leaf
    | 1 | 1           |
    | 8 | node->symbol |
```

Def main(argc, **argv)

(you know how to write a getopt by now)

Define the options “i:o:h”

```
    Set opt to 0
    Create 3 boolean values for each option, set them to false
    Create 2 char values, in and out, set them equal to null

    Write the getopt
    Case i
        Set in to optarg
```

```
                Set the test for case i as true
        Case o
                Set out to optarg
                Set the test for case o as true
        Case h
                Set the test for case h as true

        Read_open a buffer
        Create histogram variable and set the size to 256
        Fill the histogram to a variable filesize

        Set num_leaves to 0
        Create tree

        Set code table to 256
        Fill code table

        Read close the buffer
        Reopen the buffer
        Write open a bitwriter
        Huff compress file
        Read close buffer
        Close bitwriter
        Free all nodes
(end of huff.c)
```

Algorithms:

We use the huffman coding algorithm for creating a tree, pseudocode below:

```
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
```

Results:

(running ./huff -i test1.txt -o test1.huff)
test1.txt = 12 bytes
test1.huff = 30 bytes

(running ./huff -i test2.txt -o test2.huff)
test2.txt = 69 bytes
test2.huff = 76 bytes

Error Handling:

We have to think about how if a user gives invalid options or if no files are specified, we have to handle the error accordingly. With that in mind, I created if statements in the main functions that if any of the test boolean values is false, print out the error message and the help message again. As we can see from the pseudocode in the create functions, I have to account for if the variable creates is NULL or not. For each of them, I created if statements that if they're false, return an error message and return NULL because an error is present.

Credit:

Dr. Kerry Veenstra's asgn6 document