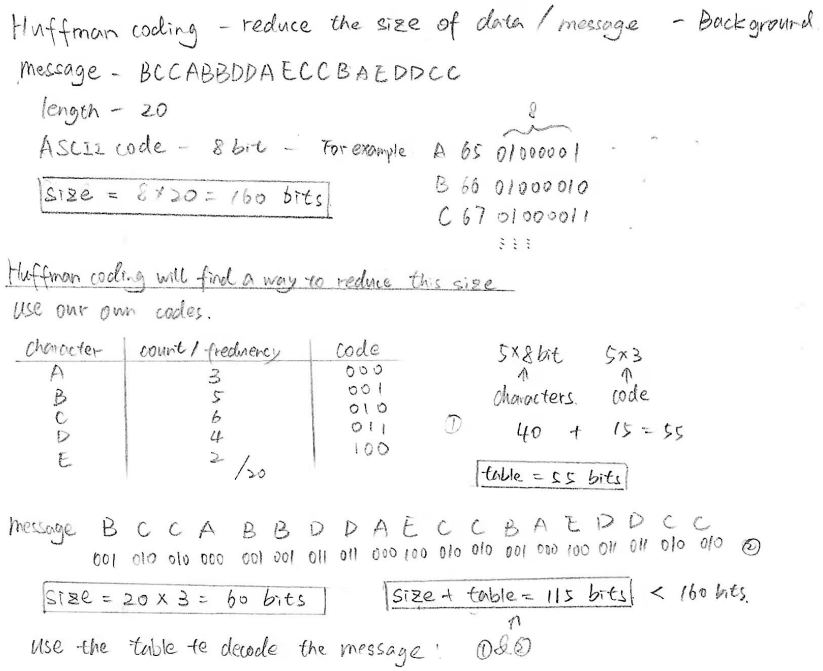
Assignment 6 DESIGN.pdf

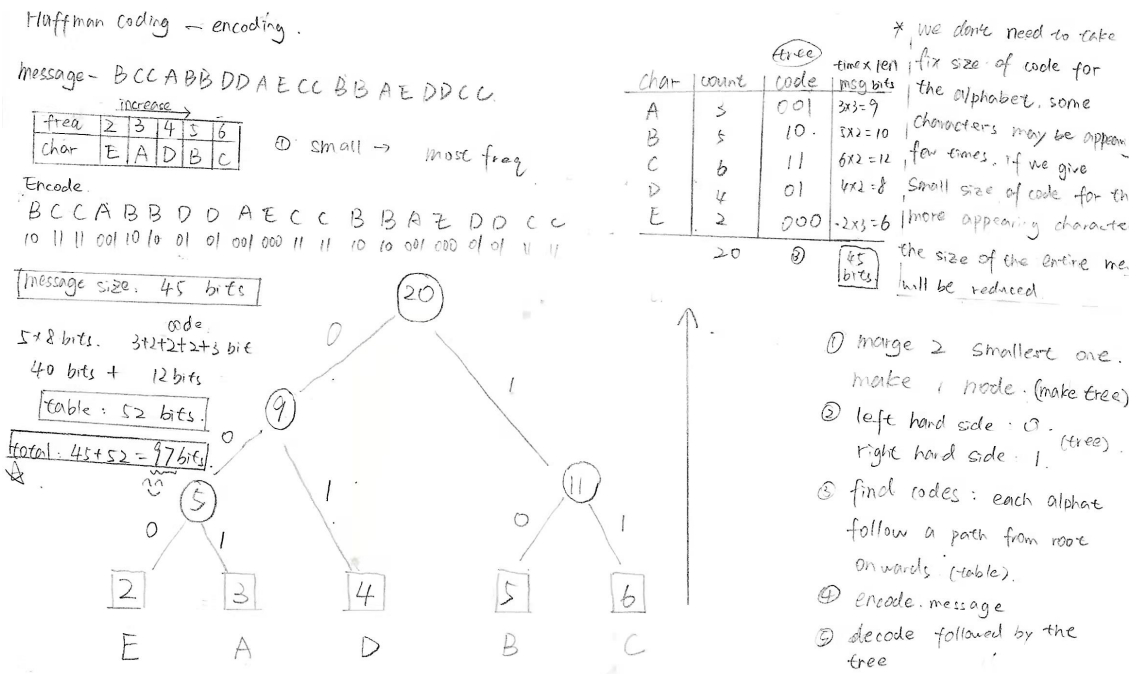
Ning Jiang

Description of Program: For this assignment, we mainly need to do huffman coding which is a way to reduce the size of data/message. The logical steps are:

1. Merge 2 smallest one, make 1 node. (make tree)
2. Left hand side is 0 and the right side is 1. (tree)
3. Find codes: each alphabet follow a path from root onwards. (table)
4. Encode message
5. Decode followed by the tree.

Basic rules and background:





Files to be included in directory “asgn6”:

1. encode.c: This file will contain your implementation of the Huffman encoder.
2. decode.c: This file will contain your implementation of the Huffman decoder.
3. defines.h: This file will contain the macro definitions used throughout the assignment. You may not modify this file.
4. header.h: This will will contain the struct definition for a file header. You may not modify this file.
5. node.h: This file will contain the node ADT interface. This file will be provided. You may not modify this file.
6. node.c: This file will contain your implementation of the node ADT.
7. pq.h: This file will contain the priority queue ADT interface. This file will be provided. You may not modify this file.
8. pq.c: This file will contain your implementation of the priority queue ADT. You must define your priority queue struct in this file.

9. code.h: This file will contain the code ADT interface. This file will be provided. You may not
modify this file.
10. code.c: This file will contain your implementation of the code ADT.
11. io.h: This file will contain the I/O module interface. This file will be provided. You may not modify
this file.
12. io.c: This file will contain your implementation of the I/O module.
13. stack.h: This file will contain the stack ADT interface. This file will be provided. You may not
modify this file.
14. stack.c: This file will contain your implementation of the stack ADT. You must define your stack
struct in this file.
15. huffman.h: This file will contain the Huffman coding module interface. This file will be provided.
You may not modify this file.
16. huffman.c: This file will contain your implementation of the Huffman coding module interface.
16. Makefile
17. README.md
18. DESIGN.pdf

2. Structure & explanation / Pseudocode:
Explanations for some of the equations are cited from asgn6.pdf.
encode.c
Command line options:
-h: Prints out a help message
-i infile
-o outfile
-v: Prints compression statistics to stderr.
Algorithm
1. Compute a histogram of the file. In other words, count the number of occurrences of each unique
symbol in the file.
2. Construct the Huffman tree using the computed histogram. This will require a priority queue.
3. Construct a code table. Each index of the table represents a symbol and the value at that index the
symbol's code. You will need to use a stack of bits and perform a traversal of the Huffman tree.
4. Emit an encoding of the Huffman tree to a file. This will be

done through a post-order traversal of
the Huffman tree. The encoding of the Huffman tree will be referred to as a tree dump.
5. Step through each symbol of the input file again. For each symbol, emit its code to the output file.

decode.c
Command line options:
-h: Prints out a help message
-i infile
-o outfile
-v: Prints decompression statistics to stderr.

Algorithm
1. Read the emitted (dumped) tree from the input file. A stack of nodes is needed in order to recon-
struct the Huffman tree.
2. Read in the rest of the input file bit-by-bit, traversing down the Huffman tree one link at a time.
Reading a 0 means walking down the left link, and reading a 1 means walking down the right link.
Whenever a leaf node is reached, its symbol is emitted and you start traversing again from the root.

node.c
crate a node to know the node in the message to make a tree. Each node(except leaf) contain a pointer to its left child, a pointer to its right child, a symbol, and the frequency of that symbol. We will make function to create, delete, join(two nodes to a parent), print the node.

pq.c
Priority queue is used to store nodes to make a tree. Priority queue is actually a circular queue which will use insertion sort after enqueue a Node. we will make functions to create, delete, check whether the queue is empty, check whether the queue is full, count the size, enqueue, dequeue and print the priority queue.

My pq:
LSB MSB
Front _ _ _ _ ... _ _ _ rare
Pop push

```
Enqueue:
If queue is fill, return false
Else, rare = rare + 1
Node[rare] = n
Do the insertion sort and return true

Dequeue:
If queue is empty, return false
Else: Node[front] = n, front = front + 1 and return true
```

Code.c

a stack of bits while traversing the tree in order to create a code for each symbol. We willcreate functions for initing the code, calculating the size, checking whether the code is empty, checking whether the coed is full, setting bit at index i in code, clearing bit i in code, getting the bit, pushing a bit in the code, poping a bit from the code, print the code.

Stack:

```
_ _ _ ......_ _ _
               Top (push and pop from there)
```

Io.c

read_byte: The number of bytes that were read from the input file descriptor, infile, is returned.

```
While the byte we read is > 0:
   Totalbyte = totalbyte + byte
   If totalbyte == nbytes
      Break
Return totalbytes
```

Write_bytes: The number of bytes written out to the output file descriptor, outfile, is returned.

```
While the byte we write is > 0:
   Totalbyte = totalbyte + byte
   If totalbyte == nbytes
      Break
Return totalbytes
```

Read_bits: You should all know by now that it is not possible to read a single bit from a file.

```
If (index % 8 = 0):
   Read the bytes in the file into a buffer
```

```
If bytes has been read is 0:
   Return false
Read one bit from the buffer.
Index + 1
Return true
```

Write_code:Each bit in the code c will be buffered into the buffer. The bits will be buffered starting from the 0th bit in c. When the buffer of BLOCK bytes is filled with bits, write the contents of the buffer to outfile.

```
For i in range(0, code size):
   If c at the index i is 0:
      Write 0 bit into the buffer
   Else
      Write 1 bit into the buffer
Index = index + 1

If index of bit i = block:
   B = Write the buffer into the outfile
Return
```

flush_codes: The sole purpose of this function is to write out any leftover, buffered bits. Make sure that any extra bits in the last byte are zeroed before flushing the codes.

```
Zerod the extra bits
Write the b of bytes into the outfile
```

Stack.c

This is also a stack for decoder. Like code.c

Stack:

```
_ _ _ ......_ _ _
               Top (push and pop from there)
```

We need to createm delete, check whether the stack is empty, check whether the stack is full, calculate the stack size which is the top, push and pop.

Hufmman.c

build_tree: Constructs a Huffman tree given a computed histogram.

```
def construct(q):
    while len(q) > 1:
        left = dequeue(q)
        right = dequeue(q)
        parent = join(left, right)
        enqueue(q, parent)
    root = dequeue(q)
    return root
```

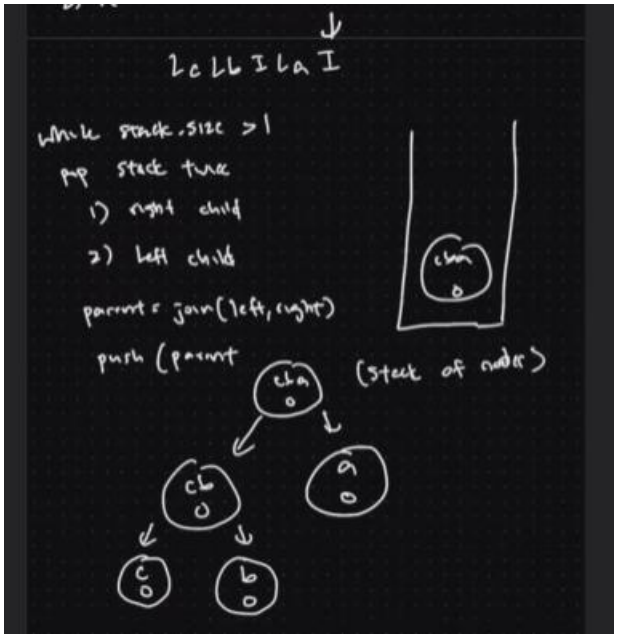
build_codes:Populates a code table, building the code for each symbols in the Huffman tree.

```
Code c = code_init()

def build(node, table):
    if node is not None:
        if not node.left and not node.right:
            table[node.symbol] = c
        else:
            push_bit(c, 0)
            build(node.left, table)
            pop_bit(c)

            push_bit(c, 1)
            build(node.right, table)
            pop_bit(c)
```

dump_tree: Conducts a post-order traversal of the Huffman tree rooted at root, writing it to outfile.

```
def dump(outfile, root):
    if root:
        dump(outfile, root.left)
        dump(outfile, root.right)

        if not root.left and not root.right:
            # Leaf node.
            write('L')
            write(node.symbol)
        else:
            # Interior node.
            write('I')
```

rebuild_tree: Reconstructs a Huffman tree given its post-order tree dump stored in the array tree_dump.



// From Eugene's section

delete_tree: The destructor for a Huffman tree.

Encode.c:

This encoder will read in an input file, find the Huffman encoding of its contents, and use the encoding to compress the file.command-line options:

-h: Prints out a help message describing the purpose of the program and the command-line op-
tions it accepts, exiting the program afterwards. Refer to the reference program in the resources
repo for an idea of what to print.
• -i infile: Specifies the input file to encode using Huffman coding. The default input should be
set as stdin.
• -o outfile: Specifies the output file to write the compressed input to. The default output should
be set as stdout.
• -v: Prints compression statistics to stderr. These statistics include the uncompressed file size,
the compressed file size, and space saving.

The algorithm to encode a file, or to compress it, is as follows:
1. Compute a histogram of the file. In other words, count the number of occurrences of each unique
symbol in the file.

2. Construct the Huffman tree using the computed histogram. This will require a priority queue.
3. Construct a code table. Each index of the table represents a symbol and the value at that index the
symbol's code. You will need to use a stack of bits and perform a traversal of the Huffman tree.
4. Emit an encoding of the Huffman tree to a file. This will be done through a post-order traversal of
the Huffman tree. The encoding of the Huffman tree will be referred to as a tree dump.
5. Step through each symbol of the input file again. For each symbol, emit its code to the output file.

Decode.c
This decoder will read in a compressed input file and decompress it, expanding it back to its original, uncompressed size.
-h: Prints out a help message describing the purpose of the program and the command-line op-
tions it accepts, exiting the program afterwards. Refer to the reference program in the resources
repo for an idea of what to print.
• -i infile: Specifies the input file to decode using Huffman coding. The default input should be
set as stdin.
• -o outfile: Specifies the output file to write the decompressed input to. The default output
should be set as stdout.
• -v: Prints decompression statistics to stderr. These statistics include the compressed file size,
the decompressed file size, and space saving.

The algorithm to decode a file, or to decompress it, is as follows:
1. Read the emitted (dumped) tree from the input file. A stack of nodes is needed in order to recon-
struct the Huffman tree.
2. Read in the rest of the input file bit-by-bit, traversing down the Huffman tree one link at a time.
Reading a 0 means walking down the left link, and reading a 1 means walking down the right link.
Whenever a leaf node is reached, its symbol is emitted and you start traversing again from the root.

Error Handing:

I use a test file to test each part of my code:

Here is my test file.

Test Node

```
 Node *e = node_create(69, 2);
 Node *a = node_create(65, 3);
 Node *d = node_create(68, 4);
 Node *b = node_create(66, 5);
 Node *c = node_create(67, 6);

 Node *ea = node_join(e, a);
 Node *ead = node_join(ea, d);
 Node *bc = node_join(b, c);
 Node *eadbc = node_join(ead, bc);
 node_print(eadbc); node.c checked

 node_delete(&n);
 //how to check whether n is already deleted.
 printf("Node's symbol is %c\n", n->symbol);
 printf("Node's freaquency is %lu\n", n->frequency);
```

```
struct PriorityQueue *q = pq_create(5);
Node *e = node_create(69, 60);
Node *a = node_create(65, 23);
Node *d = node_create(68, 42);
Node *b = node_create(66, 18);
Node *c = node_create(67, 1);
enqueue(q, e);
enqueue(q, a);
enqueue(q, d);
enqueue(q, b);
enqueue(q, c);
pq_print(q);
```

```
Node *i;
if(dequeue(q, &i)){
        printf("could dnqueue\n");
        pq_print(q);
}
else{
        printf("dnqueue false\n");
}
```

Test Code

```
Code c = code_init();
code_push_bit(&c, 1);
code_push_bit(&c, 0);
code_push_bit(&c, 0);
code_push_bit(&c, 0);
code_push_bit(&c, 1);
code_push_bit(&c, 0);
code_push_bit(&c, 1);
code_push_bit(&c, 0);
code_push_bit(&c, 0);
code_push_bit(&c, 1);
uint32_t s = code_size(&c);
printf("%d\n", s);
code_print(&c);

uint8_t bit = 0;
code_pop_bit(&c, &bit);
printf("poped bit is %hhu", bit);
uint32_t s1 = code_size(&c);
printf("size after pop is %d\n", s1);
code_print(&c);
```

Test stack

```
Stack *s = stack_create(5);
Node *e = node_create(69, 2);
Node *a = node_create(65, 3);
Node *d = node_create(68, 4);
Node *b = node_create(66, 5);
printf("%d\n", e == NULL);
stack_push(s, e);
stack_push(s, a);
stack_push(s, d);
stack_push(s, b);
//stack_print(s);

Node *n;
stack_pop(s, &n);
stack_print(s);
uint32_t size = stack_size(s);
printf("%d\n", size);
stack_delete(&s);
```

Test io.c

```
uint8_t buf[256];
int input = open("test.txt", O_RDONLY);
int output = open("test1.c", O_WRONLY|O_CREAT|O_TRUNC, 0600);
int read = read_bytes(input, buf, 256);
printf("%d\n", read);
printf("%s", buf);

write_bytes(output, buf, read);
close(output);

uint8_t bit = 0;
while(read_bit(input, &bit)){
        printf("%hhu", bit); // seg fault in line 57
}
printf("\n");
```

```
Code c = code_init();
code_push_bit(&c, 0);
code_push_bit(&c, 1);
code_push_bit(&c, 0);
code_push_bit(&c, 0);
code_push_bit(&c, 0);
code_push_bit(&c, 0);
code_push_bit(&c, 0);
code_push_bit(&c, 1);
code_push_bit(&c, 0);
code_push_bit(&c, 0);
code_push_bit(&c, 1);
code_push_bit(&c, 1);
code_push_bit(&c, 0);
code_push_bit(&c, 0);
code_push_bit(&c, 0);
code_push_bit(&c, 1);
code_push_bit(&c, 1);
code_push_bit(&c, 1);
write_code(output, &c);
flush_codes(output);
```

Test the huffman coding by using encode and decode step by step. Something I haven't solved yet: I couldn't use stdin and stdout yet. But as soon as you gave the input and output to my encode.c and decode.c. It could encode and decode successflly.