Assignment 5: Huffman Coding - Design Document DRAFT

Thadeus Ballmer

October 2021

$$W = \alpha \frac{r}{\overline{r}} \frac{\log \overline{T}}{\log T}$$

1 Introduction

More data is transferred now than ever before, but this comes at an enormous cost. To curb this cost, data compression algorithms can be used to encode data. Encoded data requires less memory to store than does data in plain text. This is why data compression is of crucial importance in this age. Assignment 5 implements the Huffman method of data compression.

2 Encoder

The first half of operating on encoded data in a useful way is encoding said data. From a large point of view, the encoder

1. Constructs a histogram from the symbols in the file.
2. Constructs a Huffman tree from the histogram—filtered through a priority queue
3. Constructs a code table based on the Huffman tree. A stack of bits will be used for this implementation.
4. Tree dump: encodes the Huffman tree to a file using the code table
5. Code dump: write the code for each symbol in the tree to the output file.

The Huffman encoding compression program to implement, `encode`, takes the following CLI options:

- `-h`: Prints out a help menu.
- `-i infile`: Specifies the input file.
- `-o outfile`: Specifies the output file.
- `-v`: Prints the uncompressed file size, the compressed file size, and space saving defined as $100 * (1 - (\text{compressed size}/\text{uncompressed size}))$.

3 Decoder

Encoding data would be of no use if the data couldn't be converted back into plain text. The second half of this assignment requires the implementation of `decode`, a program which will read the encoded information and output its original form. The decoder functions as follows:

1. Read the dumped tree from the input using a stack of nodes
2. Traverse the tree using the rest of the input bits.

`decode` takes the following command line options:

- `-h`: Prints out a help menu.
- `-i infile`: Specifies the input file.
- `-o outfile`: Specifies the output file.
- `-v`: Prints compressed file size, decompressed file size, and space saving

4 Nodes

One of many ADT's required to implement this double program is Node. The Huffman tree is a binary tree comprised of nodes.

```
struct Node {
    Node *left;          // Pointer to left child.
    Node *right;         // Pointer to right child;
    uint8_t symbol;      // Node's symbol.
    uint64_t frequency; // Frequency of symbol.
};
```

4.1 Constructor

```
Node *node_create(uint8_t symbol, uint64_t frequency)
    Dynamically allocate Node to n
    symbol = symbol
    frequency = frequency
    return n
```

4.2 Destructor

```
void node_delete(Node **n)
    free(n)
    n = Null
```

4.3 Accessors

node_print is a debugging function which prints a node's values.

```
void node_print(Node *n)
    print(Frequency, and symbol)
```

4.4 Manipulators

node_join joins two nodes together and creates a parent node.

```
Node *node_join(Node *left, Node *right)
    p = node_create('\$', left_frequency + right_frequency)
    p_left = left
    p_right = right
    return p
```

5 Priority Queues

Priority queues are used to construct the nodes in the Huffman tree in the correct order. Priority queues are like queues but each element has a priority weight which affects the order in which elements are dequeued.

```
struct PriorityQueue {
    uint32_t capacity;
    uint32_t top = 0;
    Node **queue;
}
```

5.1 Constructor

```
PriorityQueue *pq_create(uint32_t capacity)
    Dynamically allocate memory to pq
    capacity = capacity
    Dynamically allocate memory to queue of size capacity
    return pq
```

5.2 Destructor

```
void pq_delete(PriorityQueue **q)
    for (Loop through queue capacity)
        free(queue[i])
        queue[i] = Null
    free(q)
    q = Null
```

5.3 Accessors

pq_empty returns whether the queue is empty.

```
bool pq_empty(PriorityQueue *q)
    if (top == 0)
       return true
    return false
```

pq_full returns whether the queue is full.

```
bool pq_full(PriorityQueue *q)
    if (top)
        return true
    return false
```

pq_size returns the number of elements in the queue.

```
uint32_t pq_size(PriorityQueue *q)
    return top
```

pq_print is a debugging function which prints out the contents of the queue.

```
void pq_print(PriorityQueue *q)
    copy(debug_queue, q)
    insert_sort(debug_queue)
    print(debug_queue)
```

5.4 Manipulators

The queue itself functions as a min heap. So after a node in queued, the heap must be fixed.

```
bool enqueue(PriorityQueue *q, Node *n)
    if (top == capacity)
        return false
    queue[top] = n
    top += 1
    fix_heap()
    return true
```

The same logic for a min heap applies to when a node is dequeued.

```
bool dequeue(PriorityQueue *q, Node **n)
    if (pq_empty())
        return false
    swap(queue[0], queue[top - 1])
    n = queue[top - 1]
    queue[top - 1] == Null
    return true
```

6 Codes

A stack of bits is used to create a code for each symbol. While traversing the Huffman tree, the Code ADT remembers the path taken for each node—0 for left and 1 for right.

```
typedef struct {
    uint32_t top;
    uint8_t bits[MAX_CODE_SIZE]
}
```

MAX_CODE_SIZE is a macro defined in defines.h alongside other macros:

```
// 4KB blocks.
#define BLOCK 4096

// ASCII + Extended ASCII.
#define ALPHABET 256

// 32-bit magic number.
#define MAGIC 0xBEEFD00D

// Bytes for a maximum, 256-bit code.
```

```
#define MAX_CODE_SIZE (ALPHABET / 8)

// Maximum Huffman tree dump size.
#define MAX_TREE_SIZE (3 * ALPHABET - 1)
```

6.1 Constructor

The Code constructor does not dynamically allocate any memory, thus not requiring any destructor.

```
Code code_init(void)
    Code c
    c.top = 0
    for (Loop through bits)
        bits[i] = 0
    return c
```

6.2 Accessors

code_size returns the number of length of the code.

```
uint32_t code_size(Code *c)
    return top
```

code_empty returns whether the code is empty.

```
bool code_empty(Code *c)
    if (top == 0)
        return true
    return false
```

code_full returns whether the code is full.

```
bool code_full(Code *c)
    if (top == ALPHABET)
        return true
    return false
```

code_get_bit returns the value of the bit at the requested index.

```
bool code_get_bit(Code *c, uint32_t i)
    if (i <= ALPHABET)
        return false
    uint8_t get_bit = bits[i / 8]
    if (get_bit &= (1 << (i mod 8)))
        return true
    return false
```

code_print is a debugging function which prints the code.

```
void code_print(Code *c)
    for (Loop through bits)
        print(bits[i])
```

6.3 Manipulators

code_set_bit sets the value of the bit at given index to 1.

```
bool code_set_bit(Code *c, uint32_t i)
    if (i >= ALPHABET)
        return false
    bits[i / 8] |= 1 << (i mod 8)
    return true
```

code_clr_bit sets the value of the bit at the given index to 0.

```
bool code_clr_bit(Code *c, uint32_t i)
    if (i >= ALPHABET)
        return false
    bits[i / 8] &= ~(1 << (i mod 8))
    return true
```

code_push_bit pushes a bit onto the code.

```
bool code_push_bit(Code *c, uint8_t bit)
    if (code_full())
        return false
    if (bit)
        code_set_bit(c, top)
    top += 1
    return true
```

code_pop pops a bit from the code.

```
bool code_pop_bit(Code *c, uint8_t *bit)
    if (code_empty())
        return false
    bit = code_get_bit(c, top)
    code_clr_bit(c, top)
    top -= 1
    return true
```

7 I/O

Any function from <stdio.h> is prohibited in assignment 5. Instead of relying on fgets or fscan, low level system-calls like `read(), write(), open(), and close()` must be used to manage input and output.

```
int read_bytes(int infile, uint8_t *buf, int nbytes)
    while (read_value = read(infile, buf, nbytes) > 0)
        bytes += read_value
        if (bytes == nbytes)
            return bytes
    return bytes
```

read_bytes reads a given number of bytes from the infile into a buffer.

```
int write_bytes(int outfile, uint8_t *buf, int nbytes)
    while(write_bytes = write(outfile, buf, nbytes) > 0)
        bytes += write_bytes
        if (bytes == nbytes)
            return bytes
    return bytes
```

write_bytes writes a given number of bytes from the buffer to outfile.

```
bool read_bit(int infile, uint8_t *bit)
    if (buf_index == 0)
        buf_size = read_bytes(infile, buf, BLOCK)
        if (buf_size == 0)
            return false
        if (buf_size < BLOCK)
            end = buf_size * 8
    retrieves the bit at buf_index
    buf_index += 1
    if (buf_index == BLOCK * 8)
        buf_index = 0
    return buf_index != end
```

read_bit reads a single bit from the infile. It keeps track of its location in the infile and reloads bits into its buffer when needed.

```
void write_code(int outfile, Code *c)
    for (i = 0; i < code_size(c); i += 1)
        bit = code_get_bit(c, i)
        if (bit == 1)
            sets the bit to the proper index
```

```
        write_bit_index += 1
        if (write_bit_index == BLOCK * 8)
            flush_codes(outfile)
```

write_code adds a given code to a buffer filled with codes and writes the buffer to outfile when filled.

```
void flush_codes(int outfile)
    if (write_bit_index > 0)
        bytes = write_bytes(outfile, code_buf, ceiling(write_bit_index / 8))
        reset the buf values to 0
        write_bit_index = 0
```

flush_code is the function which writes the code buffer to outfile.

8 Stacks

A stack is used to reconstruct the Huffman tree.

```
struct Stack {
    uint32_t top;
    uint32_t capacity;
    Node **items;
};
```

8.1 Constructor

```
Stack *stack_create(uint32_t capacity)
    Dynamically allocate memory to Stack s
    top = 0
    capacity = capacity
    Dynamically allocate memory to capacity items each with size Node
    return s
```

8.2 Destructor

```
void stack_delete(Stack **s)
    free(items)
    items = Null
    free(s)
    s = Null
```

8.3 Accessors

stack_empty returns whether the stack is empty.

```
bool stack_empty(Stack *s)
    if (top == 0)
        return true
    return false
```

stack_full returns whether is stack is full.

```
bool stack_full(Stack *s)
    if (top == capacity)
        return true
    return false
```

stack_size returns the number of items in stack.

```
uint32_t stack_size(Stack *s)
    return top
```

8.4 Manipulators

stack_push pushes a given node onto the stack.

```
bool stack_push(Stack *s, Node *n)
    if (stack_full())
        return false
    items[top] = n
    top += 1
    return true
```

stack_pop pops a node from the stack.

```
bool stack_pop(Stack *s, Node **n)
    if (stack_empty())
        return false
    n = items[top - 1]
    top -= 1
    return true
```

stack_print is a debugging function which prints the contents of stack.

```
void stack_print(Stack *s)
    for(Loop through items)
        print(items[i])
```

9 Huffman Coding Module

The Huffman Coding Module connects the ADT's together to accomplish the task of the encoder and decoder

```
Node *build_tree(uint64_t hist[static ALPHABET])
    pq_create(ALPHABET)
    // This pseudocode is provided in the assignment document
    while len(q) > 1
        left = dequeue(q)
        right = dequeue(q)
        parent = join(left, right)
        enqueue(q, parent)
    root = dequeue(q)
    return root
```

build_tree constructs a Huffman Tree from the given histogram.

```
void build_codes(Node *root, Code table[static ALPHABET])
// Pseudocode is provided in the assignment document
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(c, 1)
        build(node.right, table)
        pop_bit(c)
```

build_codes populates a code table from the given Huffman Tree.

```
void dump_tree(int outfile, Node *root)
    // Pseudocode is provided in the assignment document
    if root:
        dump_tree(outfile, root.left)
        dump_tree(outfile, root.right)

        if not root.left and not root.right:
            // Leaf Node
            write('L')
            write(node.symbol)
```

```
        else:
            // Interior node
            write('I')
```

dump_tree writes the Huffman tree to outfile.

```
Node *rebuild_tree(uint16_t nbytes, uint8_t tree_dump[static nbytes])
    s = stack_create(nbytes)
    for(Loop through tree_dump)
        if (treedump[i] == 'L')
            n = node_create(treedump[i + 1])
            stack_push(s, n)
        if (treedump[i] == 'I')
            stack_pop(s, right)
            stack_pop(s, left)
            parent = node_join(left, right)
            stack_push(s, parent)
    stack_pop(s, root)
    return root
```

rebuild_tree reconstructs the Huffman tree from the tree_dump.

```
void delete_tree(Node **root)
    for(Loop through tree)
        free(tree index)
        set pointer to null
    free(root)
    root = null
```

delete_tree deletes the nodes in tree through a post-order traversal.

10 Encode Implementation

```
main()
    cycle through CLI options
    if (help or error)
        print(Help manual)
        exit

    if (file size == 0)
        exit

    do {
```

```
        size = read_bytes(infile, buf, BLOCK)
        for (cycle through buf)
            hist[buf[i]] += 1
    } while (size != 0)

    hist[0] += 1
    hist[1] += 1

    root = build_tree(hist)
    build_codes(root, table)

    Construct header

    dump_tree(outfile, root)

    do {
        size = read_bytes(infile, buf, BLOCK)
        for(Loop through buf)
            write_codes(outfile, table[buf[i]])
    } while (size != 0)
    flush_codes(outfile)

    if (stats)
        print(Stats)

    delete_tree(root)
    close(infile)
    close(outfile)
```

11 Decode Implementation

```
main()
    cycle through CLI options

    if (error or help)
        print(Help manual)
        exit

    if (file size == 0)
        exit

    read header
```

```
if (magic number doesnt match)
    print(Error message)
    exit

set permissions of outfile to permissions given in header

read through huffman tree
root = rebuild_tree(tree_size, tree_dump)

while(File isnt completely decoded)
    traverse through Huffman Tree and print symbols

if (stats)
    print(Stats)

delete_tree(root)
close(infile)
close(outfile)
```