Merilyn Kuo
CSE 13S
Spring 2021

Assignment 6: Huffman Coding
Design Document

Purpose:

The goal of this assignment is to learn about Huffman coding and how to compress data bits. Through creating our own Huffman code encoder and decoder, we will be learning to use priority queues, trees, and system calls as well as enforcing our past knowledge of bitwise operations, stacks, queues, parsing command line options, and input and output files.

Program Implementation:

The implementation of this entire program is quite long and made up of several parts. The most logical way of approaching it is to start off with the individual parts to build understanding of the program as a whole.

Nodes ADT:
Since we will be using trees for encoding and decoding, the first ADT will be for the nodes of the tree. The structure definition of the node includes a pointer to the left child, a pointer to the right child, the symbol of the node, and the frequency of the symbol. The structure is defined in node.h as well as the necessary function headers. This file has been provided in the repository and the pseudocode for the functions is as follows:

The constructor for a node will initialize the node symbol and frequency.

```
Node *node_create(uint8_t symbol, uint64_t frequency):
    Node *n = (Node *) calloc(1, sizeof(Node));
    if (memory for n allocated):
        n->symbol = symbol
        n->frequency = frequency
    return n
```

The destructor for a node will free memory and set the node pointer to null.

```
void node_delete(Node **n):
```

```
    if *n exists:
        free(*n)
        *n = null
```

The node join function joins the left child node and right child node with a symbol “$” and sums the frequencies of two children to store as the frequency for the new parent node. The parent node is returned.

```
Node *node_join(Node *left, Node *right):
    Node *parent = node_create(left, right)
    parent->left = left
    parent->right = right
    parent->symbol = $
    parent->frequency = left->frequency + right->frequency
```

Print node will print the members of the node structure.

```
Void node_print(Node *n):
    print(n->symbol)
    print(n->frequency)
    node_print(n->left)
    node_print(n->right)
```

Priority Queue ADT:
Part of the encoding of the Huffman coding requires us to use a priority queue of nodes to construct a tree. Therefore, we can create an ADT for the priority queue implementation. The pseudocode is as follows.

Since a priority queue is basically a queue with order, we can use the same structure definition of a regular queue. The members of the structure are the index of the start of the queue, the index of the end of the queue, the max size of the queue, the number of elements in the queue, and an array of nodes.

```
struct PriorityQueue:
    uint32_t head
    uint32_t tail
    uint32_t capacity
    uint32_t size
    Node *nodes
```

The priority queue constructor will initialize the structure members and allocate memory.

```
PriorityQueue *pq_create(uint32_t capacity):
    PriorityQueue *pq = (PriorityQueue *) malloc(sizeof(PriorityQueue))
    if (memory for pq allocated):
```

```
        pq->head = 0
        pq->tail = 0
        pq->capacity = capacity
        pq->nodes = (Node *) calloc(capacity, sizeof(Node))
        if (pq->nodes memory not allocated):
            free(pq)
            pq = null
    return pq
```

The priority queue destructor frees any allocated memory and nulls pointers.

```
void pq_delete(PriorityQueue **q):
    if (*pq and (*pq)->nodes):
        free((*pq)->items)
        free(*pq)
        *pq = null
    return


bool pq_empty(PriorityQueue *q):
    return size == 0


bool pq_full(PriorityQueue *q):
    return size == capacity


uint32_t pq_size(PriorityQueue *q):
    return size


bool enqueue(PriorityQueue *q, Node *n):
    if queue_full:
        return false
    uint32_t temp = q->tail
    while (temp != head):
        if q[(temp+capacity-1)%capacity]->frequency > n->frequency
            q[temp] = q[(temp+capacity-1)%capacity]
            q[(temp+capacity-1)%capacity] = n
            temp -= 1
        else
            q[temp] = n
            q->tail = (q->tail + 1) % n
            break



bool dequeue(PriorityQueue *q, Node **n):
```

```
    if (queue_empty):
        return false
    *n = q->nodes[q->head]
    q->size -= 1
    q->head +=1
    if (q->head == q->capacity):
        q->head = 0
    return true


void pq_print(PriorityQueue *q):
    for i = q->head; i != q->tail; i += 1:
        node_print(q->nodes[i % q->capacity])
```

Code ADT:
The code ADT will be used to store the array of bits that will represent the code for each symbol for traversing the tree. The pseudocode for the interface below uses the structure definition, macros, and function headers given to us in the assignment PDF.

The code ADT is just an array with the added element of storing the top index so it is stack-like. No memory will need to be allocated for it since we are just using an array. Instead, for the construtor, we just need to create the code and initialize the top.

```
Code code_init(void):
    Code c;
    c.top = 0
    return c
```

This function just returns the size of the code, the number of bits in the code.

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

This function returns whether the code is empty or not.

```
bool code_empty(Code *c):
    return c.top == 0
```

This function returns whether the code is full or not.

```
bool code_full(Code *c):
    return c.top == MAX_CODE_SIZE
```

This function adds a bit onto the code. The function returns true if it was successful or false if the code was full.

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

The pop function removes a bit off the code and passes the popped bit back through a pointer. It returns true if the popping was successful or false if the code was empty.

```
bool code_pop_bit(Code *c, uint8_t *bit):
    if code_empty:
        return false
    top -= 1
    *bit = bits[top]
    return true
```

Stack ADT:
The stack will be used in the decoder when reconstructing the Huffman tree. The pseudocode for this stack ADT is modified from my assignment 4 stack ADT.

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

Stack *stack_create(uint32_t capacity):
    Stack *s = (Stack *) malloc(sizeof(Stack))
    if s memory allocated:
        s->top = 0
        s->capacity = capacity
        s->items = (Node *) calloc(capacity, sizeof(Node))
        if items not memory allocated:
            free s from memory
            s = null
    return s

void stack_delete(Stack **s):
    if (s exists and s->items exists):
        free s->items from memory
        free s from memories
        *s = null
    return
```

```
bool stack_empty(Stack *s):
    return top == 0

bool stack_full(Stack *s):
    return top == capacity

uint32_t stack_size(Stack *s):
    return top

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

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

I/O Module:

```
int read_bytes(int infile, uint8_t *buf, int nbytes):
    total_read = 0
    while (total_read < nbytes and (b = read(infile, buf+total_read,
                                    nbytes-total_read)) > 0):
        total_read += b
    if b < 0:
        return b
    return total_read

int write_bytes(int outfile, uint8_t *buf, int nbytes):
    total_write = 0
    while (total_write < nbytes and (b = write(outfile, buf+total_write,
                                     nbytes-total_write)) > 0):
        total_write += b
    if b < 0
        return total_read
    return total_write

bool read_bit(int infile, uint8_t *bit):
    static uint8_t buffer[BLOCK]
    static uint64_t i = 0
    static uint32_t loaded = 0
    if loaded == 0:
```

```
        loaded = read_bytes(infile, bytes, BLOCK)
        i = 0
    if i < loaded * 8:
        *bit = (buffer[i / 8] >> (i % 8)) & 1
        i ++
    if i == loaded * 8:
        loaded = read_bytes(infile, bytes, BLOCK)
        i = 0
        if loaded <= 0:
            return false
    return true

static int buffer_index
void write_code(int outfile, Code *c):
    for i = 0 to c.top
        if (c->bits[i/8] >> (i%8)) & 1 == 1:
            buffer[i/8] |= (1 << (i%8))
        else:
            buffer[i/8] &= ~(1 << (i%8))
        buffer_index += 1
        if buffer_index == 8 * BLOCK:
            write_bytes(outfile, buffer, BLOCK)
            buffer_index = 0

void flush_codes(int outfile):
    if buffer_index > 0:
        write_bytes(outfile, buffer, BLOCK - buffer_index)
```

Huffman Coding Module: