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
```

This function checks if the priority queue is empty.

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

This function checks if the priority queue is full.

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

This function returns the size of the priority queue.

```
uint32_t pq_size(PriorityQueue *q):
    return size
```

The enqueue function adds a node to the priority queue with insertion sort. If the queue was full prior to inserting the node, the function returns false.

```
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
        return true
```

The dequeue function removes a node from the beginning of the queue. If the queue was already empty, the function returns false.

```
    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
```

This is a function that prints the nodes in the priority queue.

```
    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 to work for nodes.

```
    struct Stack
        u32 top;
        u32 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:
This module will handle the reading from and writing to file procedures.

Read bytes loops through the file to read in bytes and fill in a buffer. While there are still bytes to read and the total read is less than the specified number of bytes to read, we continue reading. If the read syscall errors while reading, this means zero bytes are read. Otherwise the function continues to read and will return the total number of bytes read in the end.

```
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
```

Write bytes is very similar to read bytes, except it writes instead of reads. It uses the same logic and returns the total number of written bytes in the end.

```
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 b
    return total_write
```

Read bits reads in a buffer of bytes and uses bitwise operations to divide the bytes into single bits. The bit is returned through a pointer. The function returns true if there are no more bytes to read and false if there are more.

```
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
```

Write code uses bitwise operations as well. The function loops through the code parameter and stores and uses bitwise operations to store a 1 or a 0 in the buffer. When the buffer is filled with bits, the contests are written out.

```
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
```

Flush codes means writing out the leftover bytes in the buffer.

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

Huffman Coding Module:
The Huffman Coding module will take care of the major steps in encoding and decoding.

Build tree uses a priority queue and a constructed histogram. For each symbol in the histogram, the function creates a node for it and enqueues it. After the priority queue is completely created with all the symbols, the function dequeues two elements, joins them, then enqueues the node. When there is only one node left, that node is returned.

```
    Node *build_tree(uint64_t hist[static ALPHABET])
        pq = pq_create(capacity)
        for each symbol in hist:
            if hist[symbol] > 0:
                n = node_create('$', hist[symbol])
                enqueue(pq, n)
        while pq_size >= 2:
            dequeueleft_child
            dequeue right_child
            parent = node_join(left_child, right_child)
            enqueue parent
        return single node in pq
```

The build codes function does a post order traversal of the Huffman tree. If a node in the tree is a leaf node, the code for that node is added to the table. Else, it traverses down the left node, then the right node and adds the path to the code. The easiest way is to do this recursively so I created another function to do the post order traversal.

```
    void build_codes(Node *root, Code table[static ALPHABET]):
        create code
        call post order traversal

    post order traversal (node, code table, code):
        if left and right node:
            table[symbol] = c
        else
            push bit 0 to code
            post order traversal(left node)
            pop bit from code
            push bit 1 to code
            post order traversal(right node)
            pop bit from code
```

The rebuild tree function takes in a tree dump in post-order and reconstructs the Huffman tree using a stack. For each item in the tree dump, if it is an L, a leaf node is created and pushed to the stack. If the item in the tree dump is an I, the node is an interior node and two nodes are popped from the stack and joined together. The joined nodes are pushed into the stack again. In the end, there is one node left in the stack and that node is returned.

```
Node *rebuild_tree(uint16_t bytes, uint8_t tree_dump[static nbytes]):
    create stack
    initialize parent node
    for range nbytes
         if tree dump item == L:
             create leaf node
             push node to stack
         else if tree dump item == I:
             create left and right node
             pop right node from stack
             pop left node from stack
             join nodes and push to stack
    return single node in stack
```

The delete tree function traverses the tree recursively.

```
void delete_tree(Node **root):
    if left node
        delete_tree(left node)
    if right node
        delete_tree(right node)
node delete(root)
```

Encoding/Decoding

Based on the specification, the top level design for the encoder and decoder is as follows:
Encoder:

1. Parse command line options with getopt. Open files if input/output files are specified.
2. Read bytes into a buffer of BLOCK size. Loop through the buffer to find unique symbols and add the frequency to the histogram.
3. Add 1 to element 0 and 255 of histogram.
4. Call build tree.

5. Call build codes to construct a code table.
6. Construct header (initialize struct members).
7. Call write_bytes to write header to outfile.
8. Dump tree into outfile. using a recursive function that writes the bytes out.
9. Starting again from the beginning of the infile, read in a block of bytes and write codes for each byte.
10. Close infile, outfile, and free any allocated memory.

Decoder:

1. Parse command line options and open any files.
2. Read through the infile for the sizeof header to get the header. Verify the magic number and set file permissions from the header.
3. Call read bytes to read in the dumped tree.
4. Call rebuild_tree to reconstruct the Huffman tree.
5. Walk down the tree recursively to write out the symbols.
6. Close the files and free any memory.

Resources:

Eugene and Sahiti's sections.
man pages