Rahul Vaidun
rvaidun@ucsc.edu
May 8 2021

CSE 13S Spring 2021
Assignment 6: Huffman Coding
Design Document

I. Introduction

This assignment discusses how to encode and decode Huffman Codes. The assignment uses various data structures such as trees, stacks, and queues to make it easier to encode and decode.

II. Pseudocode

A. Nodes

```
node_create(uint8_t symbol, uint64_t frequency) {
        Node *n  = (Node *) malloc(sizeof(Node));
        Set n->symbol and n->frequency
        Return n
}
Void node_delete(**n) {
        free(n)
        n = NULL
        Returnl
}

node_join(Node *left, Node *right) {
        Node *n  = (Node *) malloc(sizeof(Node));
        Set symbol to $
        Set frequency to sum of left and right frequency
        Set left and right to left node and right node
        Return n
}

node_print(Node *n) {
        Print node symbol and frequency;
}
```

B. pq.c

Priority queue stores nodes in the queue. The nodes with least frequency have the most priority. When a node is dequeued, return the highest frequency node first. This is implemented with a min heap

```
struct PriorityQueue {
```

```
   uint32_t capacity;
   uint32_t size;
   Node **items;
};


int parent_index(i) { return ((i-1)/2)}
int left_index(i) {return 2 * i + 1;
int right_index(i) return 2* i + 2;

PriorityQueue *pq_create(uint32_t capacity) {
        PriorityQueue *pq = (PriorityQueue *) malloc(sizeof(PriorityQueue));
        Set size to 0 and capacity to capacity
        Malloc capacity*sizeof(Node) to items
        Return pq
}

void pq_delete(PriorityQueue **q) {
        free memory in q
        q = NULL;
        Return;

void pq_swap(PriorityQueue *q, uint32_t i, uint32_t j) {
        Node *temp = q->items[i];
        Set items[i] to items[j]
        Set items[j] to temp
}

pq_heap_up(PriorityQueue *q) {
        uint32_t i = size of q -1

        while(the index i has a parent element and parent frequency is greater than the
current frequency) {
        Swap(parent index of i and i)
        }

pq_heap_down(PriorityQueue *q) {
        uint32_t i = 0;
        While (left_index(i) <  size) {
                uint32_t smallest_child_index = left_index(i);
```

```
                if( there is right child and right child is smaller than left child) {
                        smallest_child_index = right_index(i);
                }
                If frequency of current index more than frequency of child {
                        swap(i, smallest_child)
                Else{
                        Break
                }
                i = smallest_child_index
        }
}

bool pq_empty(PriorityQueue *q) {
   return q->size == 0;
}

bool pq_full(PriorityQueue *q) {
   return q->size == q->capacity;
}

uint32_t pq_size(PriorityQueue *q) {
   return q->size;
}

bool enqueue(PriorityQueue *q, Node *n) {
        If pq_full return;
        Items[size] = n;
        Size++
        pq_heap_up()
        Return true;

bool dequeue(PriorityQueue *q, Node **n) {
        If pq_empty return false;
        *n = q->items[0];
        Items[0] = last item;
        Heap_down
        Return true;
}
```

C. Code

```
Code code_init(void) {
   Code c;
   return c;
}

uint32_t code_size(Code *c) {
   return c->top;
}

bool code_empty(Code *c) {
   return c->top == 0;
}

bool code_full(Code *c) {
   if (c->top == MAX_CODE_SIZE) {
      return true;
   }
   return false;
}

bool code_push_bit(Code *c, uint8_t bit) {
   if (c->top == MAX_CODE_SIZE) {
      return false;
   }
   If (bit) {
         set_bit()
Else {
         clear_bit
}

bool code_pop_bit(Code *c, uint8_t *bit) {
   if (c->top == 0) {
      return false;
   }
   Bit = get_bit(bits, --top);
}
```

D. IO

For this assignment we will be using the low level read and write system calls. The IO module makes it easier to read and write bytes by looping calls to read and write. We also

have additional io functions that make it easier to work with Huffman codes, One function is to write codes and the other function is to read one bit at a time.

```
int read_bytes(int infile, uint8_t *buf, int nbytes) {
        Total bytes read = 0
        While total bytes read is less than nbytes
                Total += Syscall read function(infile, buf, nbytes-total)

        Bytes_read += total
        Return total

int write_butes(int infile, uint8_t *buf, int nbytes) {
        Total bytes written = 0
        While total bytes written is less than nbytes
                Total += Syscall write function(infile, buf, nbytes-total)

        Bytes_written += total
        Return total

Void flush_code(int outfile) {
        Bytes = bit_index /8
        Create mask to preserve bits i need
        If (mask)
                Buf[bytes++] &= mask;
        write_bytes(outfile,buf,bytes)

Bool read_bit(infile bit) {
        If bit_index == 0 read block bytes with read_bytes
        Bit = get_bit(buf, bit_index)
        Increment bit index and mod by block *8
        If bit_index > end of buffer
                Return false else return true

Void write_code(int outfile, Code *c) {
        Loop through the code bit vector
                If bit then set bit else clear bit
                Increment bit_index
                If buffer full write the bytes and reset bit index
```

E. stack

```
stack_create(capacity) {
```

Malloc for stack
Set top to 0 and capacity to capacity, malloc capacity * size of Node
Return s

stack_delete() {
Free space in stack
}

Stack empty should return if top is equal to 0

Stack full should return if top == capacity

Stack_size should return value of top

stack_push(Node)
Set items of top to n and increment top

Stack pop(Node **n)
Decrement node and pop the node

F. Huffman Coding Module

This module has functions that can be used for encoding and decoding any file

build_tree(histogram[ALPHABET])
Node n, left, right, joined
Prioirityqueue pq
Loop through the alphabet and if hist of alphabet is not 0 create a node and set frequency to hist[i]
While the size of pq > 1
Dequeue left and right and the enqueue the joined ned

Dequeue the root node
Delete pq
Return node

build_codes(Node *root, Code table[ALPHABET]) {
if(root) {
If we are at a leaf node
Set table[root->symbol] to c
Else
Push 0 onto code
Recursive call with left
Pop bit

Push 1 onto code
Build code with right node
Code pop bit

rebuild_tree(nbytes, treedump) {
Node for right, left, and joined
Stack of size alphbet
Loop through nbytes
If its a leaf node
Increment the counter and create node with tree dump of i
Push onto stack
Else
Pop right and left and push the joined ned
Return the root node (Last node in the stack)

}

delete_tree(Node **root) {
If *root: Call recursively with left and right and node_delete(root)

G. Encoder

post_order_travers(Node *root, uint8_t *buf, uint32_t i) {
If root
recursively call with left and right
If leaf node
Buf[i++] = ‘L’ buf[i++] = root->symbol
Else:
Arr[i++] = ‘I’

Int main() {
Int br; bytes_read
Header h
Statbuf
Uint8_t treemdump[MAX_TREE_SIZE]
Uint8_t buf[BLOCK]
Unique_symbols = 0
Dump_index = 0
Infile = 0 stdin
Outfie = 1 stdout
Code table[ALPHABET]
Tempfiledesc
Uint64_t hist

Increment histogram of 0 and 255
Switch case for command line arguments using get opt
If the file is not seekable open a temp file and read to the temp file
Fstat the infile and save to statbuf
Change permission for outfile to match

While (br = read_bytes(infile, buf, BLOCK) > 0) {
    Hist[buf][i]++;
}

Loop through the alphabet
    If hist[i] > 0 increment unique symbols

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

H.magic = MAGIC
H.permissions = instatbuf.st_mode
H.tree_size = 3 * unique_symbols -1
H.file_size = instatbuf.st_size
Write the header using write bytes. Cast the header to a uint8
post_order_traverse(root,dump &dump_index

Lseek the file back to 0
while ((br = read_bytes(infile, buf, BLOCK)) > 0)
    For each byte (iterate to br)
        write_code(outfile, &table[buf[i]);

Flush_codes

If verbose print the stats

Unlink the tempfile if we made one
Free up any space used

H. Decoder

Bw - 0
Header h
Node root_node
Node node

```
Struct stat instatbuf
Uint8_t buf [BLOCK]
Uint8_t Bit
Infile = 0
Outfile = 1
Opt = 0
Verbose = false
Switch case to handle the command line arguments
Read_bytes(infile, (uint8_t *) &h, sizeof(Header));
If magic does not match print error and exit

Get stats of the infile
fchmod(outfile, h.permissions)
Dump[h.tree_size];
read_bytes(infile, dup, h.tree_size);
Node = root_node
While (bw <h.file_size && read_bit(infile &bit) {
        If bit is 1 node = right. else node = left
        If leaf node {
                Buf[buf_index++] = node->symbol
                Bw++
                Node = root_node

                If buffer is full write_buffer to file and reset buf_index

write_bytes(outfile, buf, buf_index)

If (verbose) {
        Print the stats
}
Free up any extra space so there are no memory leaks
```