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

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

```
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;
   }
   c->bits[c->top] = bit;
   c->top++;
}

bool code_pop_bit(Code *c, uint8_t *bit) {
   if (c->top == 0) {
      return false;
   }
   c->top--;
   *bit = c->bits[c->top];
   c->bits[c->top] = NULL;
   return true;
}
```

D. IO

```
int read_bytes(int infile, uint8_t *buf, int nbytes) {
        while (1) {
                nbr = read(infile,buf, nbytes);
                if (nbr != nbytes) {
                        nbytes = nbytes-nbr
                }
}
```

}