Govind Pillai
gopillai@ucsc.edu
5/6/2021

CSE 13s Spring 2021
Assignment 5: Hamming Codes
Design Document

PURPOSE:

The purpose of this lab is to use encoder and decoder functions to properly produce a hamming code and then decode it in order to find out the error in given data. In order to do this, XOR, AND, OR, and NOT operators are used on bit vectors and bit matrices. This lab uses (8,4) hamming codes which contain 4 data bits and 4 parity bits. Given how to calculate the value of each parity bit, it is possible to find the error in the given message.

DEFINITIONS:

Hamming Codes (in the context of this lab): Error-correcting codes used to locate and correct errors created by bit flips

Generator Matrix:

1	0	0	0	0	1	1	1
0	1	0	0	1	0	1	1
0	0	1	0	1	1	0	1
0	0	0	1	1	1	1	0

Bit Vector (ADT): a one dimensional array which carries bits (1 or 0)

Bit Matrix (ADT): takes in parameters rows and columns in order to create a properly sized bit vector which will act as a matrix using the equation:

Row * (# of cols) + c

to get the index of an element in the bit vector.

Parity-Checker Matrix:

0	1	1	1	1	0	0	0
1	0	1	1	0	1	0	0
1	1	0	1	0	0	1	0
1	1	1	0	0	0	0	1

Pre-Lab Questions:

1. Lookup table:

0	0
1	4
2	5
3	HAM_ERR
4	6
5	HAM_ERR
6	HAM_ERR
7	3
8	7
9	HAM_ERR
10	HAM_ERR
11	2
12	HAM_ERR
13	1
14	0
15	HAM_ERR

2a. (1100 0111) (0 1 1 1)
(1 0 1 1)
(1 1 0 1)
(1 1 1 0)
(1 0 0 0)
(0 1 0 0)
(0 0 1 0)

(1 2 3 3) = (1 0 1 1)
1000 0111
$1110\ 0001_2$
b. (0001 1011) (0 1 1 1)
(1 0 1 1)
(1 1 0 1)
(1 1 1 0)
(1 0 0 0)
(0 1 0 0)
(0 0 1 0)
(2 1 2 1) = (0 1 0 1)
HAM_ERR

GRAPH (based on above diagram**):**

	1	2	3	4	5	6
1	0	3	1	0	0	0
2	0	0	1	0	0	0
3	0	0	0	1	0	0
4	0	0	0	0	2	3
5	0	0	0	0	0	2
6	4	0	0	0	0	0

PSEUDOCODE:

bv.c:

```
BitVector *bv_create(uint32_t length) {
        Malloc space for bit vector pointer;
        Set v→ length to length;
        Malloc space for the vector;
}
void bv_delete(BitVector **v) {
        Free space from bit vector pointer;
        Free space from the vector;
}

uint32_t bv_length(BitVector *v) {
        Return v→ length;
}
void bv_set_bit(BitVector *v, uint32_t i) {
        Set offset to i % 8;
        Left shift 1 by offset and set to val;
        Set element at index i/8 to the or of val;
}
void bv_clr_bit(BitVector *v, uint32_t i) {
        Set offset to i%8;
        Left shift 1 by offset and set to val;
        Set element at index i/8 to the and of val;
}
uint8_t bv_get_bit(BitVector *v, uint32_t i) {
        Set offset to i%8;
        Left shift 1 offset and set to val;
        And vector v and val and set to newVal;
        Right shift newVal by offset;
}
void bv_xor_bit(BitVector *v, uint32_t i, uint8_t bit) {
        Set offset to i%8;
        Left shift bit by offset and set it to val;
        Set element at index i/8 to the xor of val;
}
void bv_print(BitVector *v) {
        Iterate through and print each bit of vector;
}
```

bm.c:

```
BitMatrix *bm_create(uint32_t rows, uint32_t cols){
        Malloc space for bit matrix pointer;
        Set m→ rows to rows;
        Set m→ cols to cols;
```

```
        Set m→ vector to vector with length (rows*cols)
}
void bm_delete(BitMatrix **m){
        Free space from bit matrix pointer;
        Delete vector using bm_delete function;
}
uint32_t bm_rows(BitMatrix *m) {
        Return m→ rows;
}


uint32_t bm_cols(BitMatrix *m) {
        Return m→ cols;
}
void bm_set_bit(BitMatrix *m, uint32_t r, uint32_t c) {
        Call to bv_set_bit with parameters m→ vector and row*(# of cols) +col
}
void bm_clr_bit(BitMatrix *m, uint32_t r, uint32_t c) {
        Call to bv_clr_bit with parameters m→ vector and row*(# of cols) + col
}
uint8_t bm_get_bit(BitMatrix *m, uint32_t r, uint32_t c) {
        Call to bv_get_bit with parameters m→ vector and row*(# of cols) + col
}
BitMatrix *bm_from_data(uint8_t byte, uint32_t length) {
        Create and return a matrix with dimensions 1xlength;
        Iterate through the matrix and set each bit from byte;
}
uint8_t bm_to_data(BitMatrix *m) {
        Unsure for Design draft 1;
}
BitMatrix *bm_multiply(BitMatrix *A, BitMatrix *B) {
        For i in A:
                For j in B:
                        For c in common dimension:
                                Use and and xor to for matrix multiplication;
        Return bit matrix;
}
void bm_print(BitMatrix *m) {
        Call to bv_print with parameter m→ vector;
}
```