Design Document

The Great Firewall of Santa Cruz

Rahul Vaidun

May 24, 2021

The purpose of this code is to creae bloom filters to make it easier to censor words that are not allowed

Bloom Filter

The structure for the bloom filter is defined below:

```
struct BloomFilter {
uint64_t primary[2]; // Primary hash function salt.
uint64_t secondary[2]; // Secondary hash function salt.
uint64_t tertiary[2]; // Tertiary hash function salt.
BitVector *filter;
};
```

bf_create

bf_create creates a new Bloom Filter. It takes the size of the filter as a parameter. and the taken from the assignment PDF

```
BloomFilter *bf_create(uint32_t size) {
    BloomFilter *bf = (BloomFilter *) malloc(sizeof(BloomFilter));
    if (bf) {
        // Grimm 's Fairy
        bf->primary[0] = 0x5adf08ae86d36f21
        bf->primary[1] = 0xa267bbd3116f3957
        // The Adventures of Sherlock Holmes
        bf->secondary[0] = 0x419d292ea2ffd49e;
        bf->secondary[1] = 0x09601433057d5786;
        // The Strange Case of Dr. Jekyll and Mr. Hyde
        bf->tertiary[0] = 0x50d8bb08de3818df;
        bf->tertiary[1] = 0x4deaae187c16ae1d;
        bf->filter = bv_create(size);
        if (!bf->filter) {
            free(bf);
            bf = NULL;
        }
    }
    return bf;
}
```

bf_delete(BloomFilter **bf)

Function should delete a bloom filter given a double pointer. This will be done by first deleting the `filter` bit vector and then set it to NULL

```
bf_delete(BloomFilter **bf) {
    free(bf->vector)
```

```
    bf->vector = NULL
    free(bf);
    bf = NULL;
}
```

bf_size(BloomFilter *bf)

Returns the size of the bloom filter. The size of the bloom filter.

```
bf_size(BloomFilter *bf) {
    return bv_size(bf->filter)
}
```

void bf_insert(BloomFilter *bf, char *oldspeak)

Inserts a new method into the bloom filter. This can be done by hashing old speak with all 3 salts and setting the bits at the indices

```
void bf_insert(BloomFilter *bf, char *oldspeak) {
    set_bit(bf->filter, hash(primary, oldspeak))
    set_bit(bf->filter, hash(seconary, oldspeak))
    set_bit(bf->filter, hash(tertiary, oldspeak))
}
```

bool bf_probe(BloomFilter *bf, char *oldspeak)

Similar to bf_insert but indicat if all the indices of hashed bits are set

```
bool bf_probe(BloomFilter *bf, char *oldspeak) {
    int bit1 = get_bit(bf->filter, hash(primary, oldspeak))
    int bit2 = get_bit(bf->filter, hash(seconary, oldspeak))
    int bit3 = get_bit(bf->filter, hash(tertiary, oldspeak))
    if (bit1 && bit2 && bit3) {
        return true
    }
    return false
}
```

uint32_t bf_count(BloomFilter *bf)

Return all set bets in the bloom filter

```
uint32_t bf_count(BloomFilter *bf) {
    count = 0
    for (int i = 0; i < bf->size; i++) {
        if get_bit(filter, i) {
            count++;
        }
    }
    return count
}
```

void bf_print(BloomFilter *bf)

```
void bf_print(BloomFilter *bf) {
    bv_print(filter);
```

```
    printf("There are %d bits set\n", bf->count);
}
```

Bit Vector

The bit vector will be very similar to the bit vector in asgn5. The structure is shown below

```
struct BitVector {
    uint32_t length;
    uint8_t *vector;
};
```

BitVector bv_create(uint32_t length)

Creates a new bit vector given a length.

```
// Code directly taken from assignment 5
BitVector *bv_create(uint32_t length) {
    BitVector *bv = (BitVector *) malloc(sizeof(BitVector));
    if (bv) {
        bv->length = length;
        if (length % 8 == 0) {
            bv->vector = (uint8_t *) calloc(bv->length / 8, sizeof(uint8_t));
        } else {
            bv->vector = (uint8_t *) calloc((bv->length / 8) + 1, sizeof(uint8_t));
        }
    }
    return bv;
}
```

bv_delete(BitVector **bv)

Delete a Bit Vector

```
// Code taken directly from assignment 5
void bv_delete(BitVector **v) {
    if (*v && (*v)->vector) {
        free((*v)->vector);
        free(*v);
        *v = NULL;
    }
    return;
}
```

uint32_t bv_length(BitVector **bv)

Returns length of bit vector

```
// Code taken directly from assignment 5
uint32_t bv_length(BitVector *v) {
    return v->length;
}
```

Set, Get and Clear Bit

Functions to get, set and clear a bit in the Bit Vector

```
// Code taken directly from assignment 5
// set a bit in the bit vector
void bv_set_bit(BitVector *v, uint32_t i) {
    uint32_t bytepos = i / 8;
    uint32_t bitpos = i % 8;
    v->vector[bytepos] |= (1 << bitpos);
    return;
}

// clear a bit from the bit vector
void bv_clr_bit(BitVector *v, uint32_t i) {
    uint32_t bytepos = i / 8;
    uint32_t bitpos = i % 8;
    v->vector[bytepos] &= ~(1 << bitpos);
    return;
}

// get a bit from the bit vector
uint8_t bv_get_bit(BitVector *v, uint32_t i) {
    uint32_t bytepos = i / 8;
    uint32_t bitpos = i % 8;
    return (v->vector[bytepos] >> bitpos) & 1;
}
```

void bv_print(BitVector *bv)

Debugger fucntion to print the bit vector

```
// Code taken directly from assignment 5
void bv_print(BitVector *v) {
    for (uint32_t i = 0; i < v->length; i++) {
        printf("bv[%d]=%d \n", i, v->vector[i]);
    }
    printf("\n");
    return;
}
```

Hash table

The hash table structure is similar to the bloom filter structure. The hash table will use linked lists to resolve the oldspeak hash colliisions. The structure is shown below

```
struct HashTable {
    uint64_t salt[2];
    uint32_t size;
    bool mtf;
    LinkedList **lists;
}
```

HashTable *ht_create(uint32_t size, bool mtf)

Constructs a new Hash Table. Code is shown below and taen directly from the assignment PDF

```
HashTable *ht_create(uint32_t size, bool mtf) {
    HashTable *ht = (HashTable *) malloc(sizeof(HashTable));
```

```
    if (ht) {
        ht->salt[0] = 0x9846e4f157fe8840;
        ht->salt[1] = 0xc5f318d7e055afb8;
        ht->size = size;
        ht->mtf = mtf;
        ht->lists = (LinkedList **) calloc(size, sizeof(LinkedList *));
        if (!ht->lists) {
            free(ht);
            ht = NULL;
        }
    }
}
```

void ht_delete(HashTable **ht)

Destructor function for the hash table

```
void ht_delete(HashTable **ht) {
    free(ht->lists)
    ht->lists = NULL
    free(ht);
    ht = NULL
}
```

ht_size(HashTable *ht)

Return size of the Hash Table

```
ht_size(HashTable *ht) {
    return ht->size;
}
```

Node *ht_lookup(HashTable *ht, char *oldspeak)

Search for a node in the hash table

```
Node *ht_lookup(HashTable *ht, char *oldspeak) {
    index = hash(ht->salt, oldspeak)
    return lookup(ht->lists[index], oldspeak)
}
```

void ht_insert(HashTable *ht, char *oldspeak, char *newspeak)

Insert oldspeak along with the newspeak translation into the hash table.

```
void ht_insert(HashTable *ht, char *oldspeak, char *newspeak) {
    index = hash(ht->salt, oldspeak);
    if (Linked list at index does not exist) {
        ht->lists[index] = ll_create(mtf);
    }
    insert(ht->lists[index], oldspeak, newspeak)
}
```

uint32_t ht_count(HashTable *ht)

Returns the non-null linked lists in the hash table

```
uint32_t ht_count(HashTable *ht) {
    count = 0
    for (int i = 0; i < size of hashtable; i++) {
        if (ht->lists[i]) {
            count++
        }
    }
    return count
}
```

void ht_print(HashTable *ht)

Debugger function to print a hash table

```
void ht_print(HashTable *ht) {
    for (int i = 0; i < size of hashtable; i++) {
        if (ht->lists[i]) {
            print(linked list at i)
        }
    return
}
```

Linked List

Linked lists are going to be used to resolve hash collisions. Each node of the linked list will have the *oldspeak* and *newspeak* translation. The key to search in the linked list is *oldspeak*. This linked list will be a doubly linked list so each node will have pointers to the previous and the next node. The structure for a node is shown below.

```
struct Node {
    char *oldspeak;
    char *newspeak;
    Node *next;
    Node *prev;
};
```

Node *node_create(char *oldspeak, char *newspeak)

Constructor for the node. The constructor makes copies of oldspeak and newspeak and allocates memory for these functions.

```
Node *node_create(char *oldspeak, char *newspeak) {
    Node *n = malloc(Node)
    if (n) {
        n->oldspeak = malloc(strlen(oldspeak))
        strcpy(n->oldspeak,oldspeak)
        n->newspeak = malloc(strlen(newspeak))
        strcpy(n->newspeak,newspeak)
        n->next = NULL
        n->prev = NULL
    }
}
```

`void node_delete(Node **n)`

Frees up any space that was designated for a node. This means freeing the `oldspeak` `newspeak` and finally the node itself.

`void node_print(Node *n)`

Prints a node. Used in debugging as well as for program output

```
void node_print(Node *n) {
    if (n->oldspeak && n->newspeak) {
        printf("%s -> %s\n", n->oldspeak , n->newspeak);
    } else {
        printf("%s\n", n->oldspeak);
    }
}
```

Linked Lists

The linked list structure is shown below.

```
struct LinkedList {
    uint32_t length;
    Node *head; // Head sentinel node.
    Node *tail; // Tail sentinel node.
    bool mtf;
};
```

`LinkedList *ll_create(bool mtf)`

The `mtf` parameter indicates wheather nodes that are looked up should be moved to the front if they are found.

```
LinkedList *ll_create(bool mtf) {
    LinkedList ll = malloc(LinkedList)
    if (ll) {
        n->head = malloc(Node)
        n->tail = malloc(Node)
        n->head->prev = NULL
        n->head->next = tail
        n->tail->prev = head
        n->tail->next = NULL

    }
}
```

`void ll_delete(LinkedList **ll)`

free up any space allocated for the linked list including all the nodes

`uint32_t ll_length(LinkedList *ll)`

returns length of linked list. To do this just return `ll->length`

`Node *ll_lookup(LinkedList *ll, char *oldspeak)`

Searches for a node containing the oldspeak. If the node is found return a pointer to the node. Else return null. if mtf is enabled move the node to the front

```
// Code was inspired from the lecture slides on Linked Lists by Professor Darrell Long
Node *ll_lookup(LinkedList *ll, char *oldspeak) {
    for (Node *curr = ll->head; curr != NULL; curr = curr->next) {
        if(cur->ll == oldspeak) {
            return curr
        }
    }
    return NULL
}
```

`void ll_insert(LinkedList *ll, char *oldspeak, char *newspeak)`

Insert a new node into the linked list

```
// Code was inspired from the lecture slides on Linked Lists by Professor Darrell Long
void ll_insert(LinkedList *ll, char *oldspeak, char *newspeak) {
    if (lookup(ll,oldspeak) then return
    create a new node, n, with oldspeak and newspeak
    n->next = ll->head
    head = n
    return head;
}
```

`void ll_print(LinkedList *ll)`

print out each node. Loop throgugh the nodes in the linked list and `node_print` each node