Bill Zhang
jzhan411@ucsc.edu
05/27/2021

CSE13S Spring 2021
Assignment 7: The Great Firewall of Santa Cruz
Design Document

This program will simulate a firewall filtering bad words out and reminding the user of which new words to use in place of old words.

BloomFilter ADT

Struct BoomFilter

Int 64 primary[2]
Int 64 secondary[2]
Int64 tertiary[2]
Uint 32 setted = 0
//3 different salts for hash function
BitVector *filter

BloomFilter *bf_create(uint32 size)

//Code is given from assignment document

```
1  BloomFilter *bf_create(uint32_t size) {
2    BloomFilter *bf = (BloomFilter *) malloc(sizeof(BloomFilter));
3    if (bf) {
4      // Grimm's Fairy Tales
5      bf->primary[0] = 0x5adf08ae86d36f21;
6      bf->primary[1] = 0xa267bbd3116f3957;
7      // The Adventures of Sherlock Holmes
8      bf->secondary[0] = 0x419d292ea2ffd49e;
9      bf->secondary[1] = 0x09601433057d5786;
10     // The Strange Case of Dr. Jekyll and Mr. Hyde
11     bf->tertiary[0] = 0x50d8bb08de3818df;
12     bf->tertiary[1] = 0x4deaae187c16ae1d;
13     bf->filter = bv_create(size);
14     if (!bf->filter) {
15       free(bf);
16       bf = NULL;
17     }
18   }
19   return bf;
20 }
```

Void bf_delete(BloomFilter **Bf)

Free *bf
Set *bf to null

Bv_delete (*bf -filter)

Returns the size of the Bloom filter.
uint32_t bf_size(BloomFilter *bf)
Return bv_size(bf->filter)

*Takes oldspeak and inserts it into the Bloom filter. This entails hashing oldspeak with each of the three salts for three indices, and setting the bits at those indices in the underlying bit vecto*r.
void bf_insert(BloomFilter *bf, char *oldspeak)
Insert old speak into Filter
Hash old speak with 3 salts
Set in bits in indices in bit vector

Probes the Bloom filter for oldspeak. Like with bf_insert(), oldspeak is hashed with each of the three salts for three indices. If all the bits at those indices are set, return true to signify that oldspeak was most likely added to the Bloom filter. Else, return false.
bool bf_probe(BloomFilter *bf, char *oldspeak)
Hash old speak with 3 salts
Check if bits in filter are set
Return true if they are, false if not

Return number of set bits in Bloom Filter
uint32_t bf_count(BloomFilter *bf)
Return setted

A debug function to print out a Bloom filter.
void bf_print(BloomFilter *bf)

Bitvector
A bit vector is an ADT that represents a one dimensional array of bits, the bits in which are used to denote if something is true or false (1 or 0)

Structure bitvector
Length (length in bits)
Array *vector Array of bytes containing bits

BitVector *bv_create(uint32_t length)

The constructor for a bit vector. In the event that sufficient memory cannot be allocated, the function
must return NULL.
Allocate memory for bv
If bv
    Set length
    allocate memory for vector
    Set each element of vector 0
    If didnt set vector
        Free bv
    Set v = null

void bv_delete(BitVector **v)
The destructor for a bit vector. Remember to set the pointer to NULL
after the memory associated with the bit vector is freed.
    If v and if v->vector
        Free both
        *v = null

uint32_t bv_length(BitVector *v)
Returns the length of a bit vector.
    Return v->length

void bv_set_bit(BitVector *v, uint32_t i)
    Set ith bit in vit vector.
    Use bitwise math

void bv_clr_bit(BitVector *v, uint32_t i)
    Clears ith bit in bitvector
    Use bitwise math

uint8_t bv_get_bit(BitVector *v, uint32_t i)
    Returns ith bit
    Use bitwise math

void bv_xor_bit(BitVector *v, uint32_t i, uint8_t bit)
    Xors the ith bit in bit vector with value of specified bit

void bv_print(BitVector *v)
    A debug function to print a bit vector.

Struct Hash table

Uint64 salt[2] // contains salt

Uint32 size

Bool mtf

LinkedList ** lists

HashTable *ht_create(uint32_t size, bool mtf)

//Given by assignment document

```c
HashTable *ht_create(uint32_t size, bool mtf) {
  HashTable *ht = (HashTable *) malloc(sizeof(HashTable));
  if (ht) {
    // Leviathan
    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;
    }
  }
  return ht;
}
```

void ht_delete(HashTable **ht)

Free each linked lists in lists

Free(*ht)

*ht = null

uint32_t ht_size(HashTable *ht)

Return ht->size

Searches for an entry, a node, in the hash table that contains oldspeak. A node stores oldspeak and its newspeak translation. The index of the linked list to perform a look-up on is calculated by hashing the oldspeak. If the node is found, the pointer to the node is returned. Else, a NULL pointer is returned.

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

Inserts the specified oldspeak and its corresponding newspeak translation into the hash table. The index of the linked list to insert into is calculated by hashing the oldspeak. If the linked list that should be inserted into hasn't been initialized yet, create it first before inserting the oldspeak and newspeak
void ht_insert(HashTable *ht, char *oldspeak, char *newspeak)

Returns the number of non-NULL linked lists in the hash table.
uint32_t ht_count(HashTable *ht)

A debug function to print out the contents of a hash table.
void ht_print(HashTable *ht)

Node adt
    Char *oldspeak
    Char *newspeak
    Node *next
    Node*prev

Node *node_create(char *oldspeak, char *newspeak)
    Allocate memory for oldspeak and new speak
    Copy over characters

void node_delete(Node **n)
    Only free this node and not next prev
    Free *n
    *n = null
    Free 2 chars
void node_print(Node *n)

- If the node `n` contains oldspeak *and* newspeak, print out the node with this print statement:

```
printf("%s -> %s\n", n->oldspeak, n->newspeak);
```

- If the node `n` contains *only* oldspeak, meaning that newspeak is null, then print out the node with this print statement:

```
printf("%s\n", n->oldspeak);
```