Tu-An Nguyen
tunhnguy@ucsc.edu
02/28/2021

CSE 13S Winter 2021
Assignment 6: The Great Firewall of Santa Cruz
Design Document

Pre-lab

Part 1

1. Write down the pseudocode for inserting and deleting elements from a Bloom filter.

Pseudocode for inserting an element w to a Bloom filter b with k hash functions:

```
for i in [0, k):
      b[hash of w w/ salt i] = 1
```

Deleting elements from a Bloom filter is not done in practice, because it will create false negatives. The whole point of Bloom filters is to ensure there are no false negatives and only possible false positives. That being said, deleting elements from a Bloom filter is still possible. Here is the pseudocode for deleting an element w from a Bloom filter b with k hash functions:

```
for i in [0, k):
      if b[hash of w w/ salt i] == 1:
            b[hash of w w/ salt i] = 0
```

Part 2

1. Write down the pseudocode for each of the functions in the interface for the linked list ADT.

See pseudocode here.

Part 3

1. Write down the regular expression you will use to match words with. It should match hyphenations and concatenations as well.

```
[a-zA-Z0-9_’-]+
```

Purpose

The purpose of this assignment is to create a firewall to filter text, detect banned words (*badspeak*), and convert any old words (*oldspeak*) into new words (*newspeak*).

The program will read in multiple files--`badspeak.txt`, `oldspeak.txt`, and `newspeak.txt`--to create a lexicon of badspeak and oldspeak/newspeak translations.

Then, the program will read in text from a citizen. If a word from the text is possibly badspeak or oldspeak, a Bloom filter will be used to carry out a few different actions: send the citizen to *joycamp* by accusing them of *thoughtcrime* if they used badspeak, help the citizen practice *Rightspeak* by providing oldspeak/newspeak pairs, or refrain from any disciplinary action when a false positive was detected.

Pseudocode

banhammer.c

```
int main(int argc, char **argv):
    int opt
    int ht_size = 10000
    int bf_size = $2^{20}$
    bool mtf = false

    assign command line options

    BloomFilter *bf = bf_create(bf_size)
    HashTable *ht = ht_create(ht_size)

    char *badspeak
    while fscanf(badspeak.txt, “%s\n”, badspeak) != EOF:
        bf_insert(bf, badspeak)
        ht_insert(ht, oldspeak, newspeak)

    char *oldspeak, newspeak
    while fscanf(newspeak.txt, “%s %s\n”, oldspeak, newspeak) != EOF:
        bf_insert(bf, oldspeak)
        ht_insert(ht, oldspeak, newspeak)

    char *word
    LinkedList *bs_words
    LinkedList *os_words
    while fscanf(stdin, “%s ”, &word) != EOF:
        if bf_probe(bf, word):
            Node *n = ht_lookup(ht, word)

            if n:
                if n->newspeak == NULL:
                    add word to bs_words
                else:
                    add word to os_words

    if bs_words and os_words:
        send thoughtcrime message
        print out bs_words
        print out os_words
    else if bs_words and !os_words:
        send thoughtcrime message
        print out bs_words
    else if !bs_words and os_words:
        send rightspeak message
        print out os_words

    bf_delete(bf)
    ht_delete(ht)
```

bf.c

```
struct BloomFilter:
    uint64 primary[2]
    uint64 secondary[2]
    uint64 tertiary[2]
    BitVector *filter

BloomFilter *bf_create(uint32 size):
    BloomFilter *bf = allocate memory of size BloomFilter

    if bf:
        bf->primary[0] = 0x02d232593fbe42ff
        bf->primary[1] = 0x3775cfbf0794f152

        bf->secondary[0] = 0xc1706bc17ececc04
        bf->secondary[1] = 0xe9820aa4d2b8261a

        bf->tertiary[0] = 0xd37b01df0ae8f8d0
```

```
            bf->tertiary[1] = 0x911d454886ca7cf7

            bf->filter = bv_create(size)

            if !bf->filter:
                  free bf
                  bf = NULL

      return bf

void bf_delete(BloomFilter **bf):
      bv_delete(bf->filter)

      free bf
      *bf = NULL

uint32 bf_length(BloomFilter *bf):
      return bv_length(bf->filter)

void bf_insert(BloomFilter *bf, char *oldspeak):
      uint32 index_1 = hash(bf->primary, oldspeak) % bf_length(bf)
      uint32 index_2 = hash(bf->secondary, oldspeak) % bf_length(bf)
      uint32 index_3 = hash(bf->tertiary, oldspeak) % bf_length(bf)

      bv_set_bit(bf->filter, index_1)
      bv_set_bit(bf->filter, index_2)
      bv_set_bit(bf->filter, index_3)

bool bf_probe(BloomFilter *bf, char *oldspeak):
      uint32 index_1 = hash(bf->primary, oldspeak) % bf_length(bf)
      uint32 index_2 = hash(bf->secondary, oldspeak) % bf_length(bf)
      uint32 index_3 = hash(bf->tertiary, oldspeak) % bf_length(bf)

      return vector’s bit at index_1, index_2, and index_3 are set

void bf_print(BloomFilter *bf):
      bv_print(bf->filter)
```

bv.c

```
struct BitVector:
      uint32 length
      uint8 *vector

int bytes (int bits):
    if bits % 8 == 0:
        return bits / 8
    return bits / 8 + 1

BitVector *bv_create(uint32 length):
      BitVector *b = allocate memory of size BitVector

      b->length = length

      b->vector = allocate memory of length byte(length) and size uint8

      if !b->vector:
            free b
            b = NULL

      return b

void bv_delete(BitVector **bv):
      free *bv->vector
      *bv->vector = NULL

      free *bv
      *bv = NULL

uint32 bv_length(BitVector *bv):
      return bv->length

int byte_col (int c):
    int ans = bytes(c)

    // Account for zero indexing.
    if ans == 0:
        ans -= 1

   return ans

void bv_set_bit(BitVector *bv, uint32 i):
      uint8 index = i % 8
      uint8 mask = 1 << index
      bv->vector[byte_col(i)] |= mask

      return

void bv_clr_bit(BitVector *bv, uint32 i):
      uint8 index = i % 8
      uint8 mask = ~(1 << index)

      bv->vector[byte_col(i)] &= mask

      return

uint8 bv_get_bit(BitVector *bv, uint32 i):
      uint8 index = i % 8
      uint8 mask = 1 << index

      uint8 result = (uint8) (bv->vector[byte_col(i)]) & mask

      result = result >> index

      return result

void bv_print(BitVector *bv):
      for i in [0, bv_length(bv)):
            if bv_get_bit(bv, i) == 0:
                  print “0”
            else
                  print “1”

      print “\n”

      return
```

hash.c

```
struct HashTable:
      uint64 salt[2]
      uint32 size
      bool mtf
      LinkedList **lists

HashTable *ht_create(uint32_t size, bool mtf):
      HashTable *h = allocate memory of size HashTable
      if h:
            h->salt[0] = 0x85ae998311115ae3
            h->salt[1] = 0xb6fac2ae33a40089
            h->size = size
            h->mtf = mtf

            ht->lists = allocate memory with size elements and
sizeof(LinkedList **)

            if !ht->lists:
                  free ht
                  ht = NULL

      return ht

void ht_delete(HashTable **ht):
      ll_delete((*ht)->lists)

      free(*ht)
      (*ht) = NULL

      return

uint32 ht_size(HashTable *ht):
      return ht->size;

Node *ht_lookup(HashTable *ht, char *oldspeak):
      uint32 index = hash(ht->salt, oldspeak) % ht_size(ht)
      LinkedList *l = ht->lists[index]

      return ll_lookup(ll, oldspeak)

void ht_insert(HashTable *ht, char *oldspeak, char *newspeak):
      uint32 index = hash(ht->salt, oldspeak) % ht_size(ht)

      if ht->lists[index] == NULL:
            ht->lists[index] = ll_create(ht->mtf)

      ll_insert(ht->lists[index], oldspeak, newspeak)

void ht_print(HashTable *ht):
      for i in [0, ht_size(ht)):
            ll_print(ht->lists[i])
```

ll.c

```
struct LinkedList:
      uint32 length
      Node *head
      Node *tail
      bool mtf

LinkedList *ll_create(bool mtf):
      LinkedList *l = allocate memory of sized LinkedList

      l->length = 0
      l->mtf = mtf
      l->head = new node w/ NULL parameters
      l->tail = new node w/ NULL parameters

      l->head->next = tail
      l->tail->prev = head

      return l

void ll_delete(LinkedList **ll):
      Node *n = ll’s head

      while n has a next:
            node_delete(n)
            n = n’s next

      free allocated memory in *ll
      *ll = NULL

      return

uint32 ll_length(LinkedList *ll):
      uint32 count = 0
      Node *n = ll’s head’s next

      while n has a next:
            count += 1
            n = n’s next

      return count

Node *ll_lookup(LinkedList *ll, char *oldspeak):
      Node *n = ll->head->next

      while n has a next:
            if strcmp(n->oldspeak, oldspeak) == 0:
                  if ll->mtf:
                        n->next = ll->head->next
                        n->prev = ll->head

                        ll->head->next->prev = n
                        ll->head->next = n

                  return n
            n = n->next

      return NULL

void ll_insert(LinkedList *ll, char *oldspeak, char *newspeak):
      if ll_lookup(ll, oldspeak) != NULL:
            return

      Node *x = new node w/ oldspeak and newspeak parameters

      x->next = ll->head->next
      x->prev = ll->head

      ll->head->next->prev = x
      ll->head->next = x

      ll->length += 1

      return

void ll_print(LinkedList *ll):
      Node *n = ll->head->next

      while n has a next:
            node_print(n)
            n = n’s next

      return
```

node.c

```
struct Node:
      char *oldspeak
      char *newspeak
      Node *next
      Node *prev

char *str_copy(char *c):
      char *cp = allocate memory w/ strlen(c) elements of size char

      copy in char of c into cp

      return cp

Node *node_create(char *oldspeak, char *newspeak):
      Node *n = allocate memory of size Node

      n->oldspeak = str_copy(oldspeak)
      n->newspeak = str_copy(newspeak)
      n->next = NULL
      n->prev = NULL

      return n

void str_copy_delete(char **cp):
      free allocated memory in *cp
      *cp = NULL

      return

void node_delete(Node **n):
      str_copy_delete(&(*n)->oldspeak)
      str_copy_delete(&(*n)->newspeak)

      free allocated memory in *n
      *n = NULL

      return

void node_print(Node *n):
      if n has oldspeak and newspeak:
            print oldspeak + “ -> “ + newspeak;

      if n has oldspeak and not newspeak:
            print oldspeak
```