Assignment 7 - The Great Firewall of Santa Cruz

Sean Carlyle

November 26th 2021

1 Introduction

This is my Assignment 7, The Great Firewall of Santa Cruz Design document. This program uses bloom filters, hash tables, and binary trees to filter through text and find words that have been classified as *oldspeak* or *badspeak*. Then it send the user a message either of what *oldspeak* words they used that should become in *newspeak* or what words they have been saying that have been totally banned as *badspeak*. The command-line options that the program supports are as follows: -h to print out program usage, -t size to specify hash table size (default 2^{16}), -f size to specify bloom filter size (default 2^{20}), and -s to print out statistics to stdout such as binary tree size, height, number of traversed and more.

2 Pseudocode with Explanation

2.1 banhammer.c

Banhammer.c is the file that holds the main function that combines all of our other helper functions to parse through text and report which words are oldspeak and badspeak. It does this by checking if words in text are in the bloom filter (filled by badspeak.txt/newspeak.txt), and then whether they are in the hash table(filled bybadspeak.txt/newspeak.txt). If they are we know they are badspeak/oldspeak and are added to a list of words the user used. Then the users are shown what wrong words they used.

2.1.1 main

```
Initialize bloom filter and hash table
Add badspeak words to bloom filter and hash table from badspeak.txt using fscanf()
Add oldspeak words to bloom filter, but add oldspeak and newspeak to hash table
Parse text with parsing module
Check if each word is in the bloom filter
If a word is probably in the bloom filter (since bloom filter can return false positives)
    If word is in hash table with no newspeak translation, it is badspeak and should be put in a list of badspeak words used
```

If word is in hash table with newspeak translation, put it in a list of oldspeak words with their newspeak counterparts

If not in the hash table, bloom filter returned a false positive so do nothing

After parsing through the whole text, show the user what badspeak words or oldspeak words along with their newspeak words they used. Then print the appropriate message for each case. The cases being: there is no badspeak, there is no oldspeak-¿newspeak, or there is a mix of both.
Print out statistics if needed
Free everything

2.2 ht.c

ht.c is the file that holds the hash table ADT. This ADT in our program is used after bloom filters to check whether a word is badspeak/oldspeak because sometimes the bloom filters will report a false positive. The hash table is an array of trees since we use binary trees to solve the case of a hash collision.

2.2.1 HashTable *ht_create(uint32_t size)

Dynamically allocate memory for hash table
Allocate memory for the array of trees
Set size to size
Set salt to the given salts in salt.h
return ht

2.2.2 void ht_delete(HashTable **ht)

Delete every BST in the ht
Free dynamically allocated array of trees
Free allocated memory for hash table
Set the hash table pointer to NULL

2.2.3 uint32_t ht_size(HashTable *ht)

Return size of hashtable

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

Increment lookups
Call bst find on oldspeak in the index given by hashed oldspeak mod by the size of the ht. This will return a node pointer.
Return node pointer

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

Increment lookups
Insert the oldspeak and newspeak translation in the index given by hashing the oldspeak. This will return a node *.
Set the BST at the hashed index to the node *.

2.2.6 uint32_t ht_count(HashTable *ht)

Count non NULL binary trees in a for loop and return the count

2.2.7 double ht_avg_bst_size(HashTable *ht)

Use bst_size on each tree and sum their sizes
Divide the sum by the non-NULL trees in hash table using ht count
Return the calculated number

2.2.8 double ht_avg_bst_height(HashTable *ht)

Use bst_height on each tree and sum their heights
Divide the sum by the non-NULL trees in hash table using ht count
Return the calculated number

2.2.9 void ht_print(HashTable *ht)

Print out each tree in hash table using bst print in a loop

2.3 bst.c

bst.c is the binary tree ADT. We use binary trees to solve hash collisions where two words have the same hash value. The interesting thing about the BST ADT is that it is actually just a node *. Because nodes can point to left and right nodes, we can form a BST without another struct.

2.3.1 Node *bst_create(void)

Return a NULL *.

2.3.2 void bst_delete(Node **root)

```
If root exists
    bst_delete the right node
    bst_delete the left node
    node_delete the current root
```

2.3.3 uint32_t bst_height(Node *root)

```
If root exists
    Returns 1 + max of the bst_height of right and left nodes
Return 0
```

2.3.4 uint32_t bst_size(Node *root)

```
If root exists
    Return 1 + the size of left and right node
Return 0
```

2.3.5 Node *bst_find(Node *root, char *oldspeak)

```
If root exists
    If root oldspeak is greater than oldspeak
        If left node doesn't exist return null
        Return bst find on left root and oldspeak
    Else if root oldspeak is less than oldspeak
        If right node doesn't exist return null
        Return bst find on right root and oldspeak
Return root
```

2.3.6 Node *bst_insert(Node *root, char *oldspeak, char *newspeak)

```
If root exists
    If root oldspeak is oldspeak return root
    If root oldspeak is greater than oldspeak
        Left root is equal to bst insert on left root and oldspeak
    Else
        Right root is equal to bst insert on right root and oldspeak
Return new node with oldspeak, newspeak
```

2.3.7 void bst_print(Node *root)

```
If root exists
    Bst print the left node
    Node print the root
    Bst print the right node
```

2.4 node.c

node.c is the file that holds our node ADT. The nodes in this program hold pointer to oldspeak, pointer to newspeak, and a pointer to a left and right child node. We need the node ADT because binary trees are nodes.

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

Allocate memory for the node
If newspeak doesn't exist make node newspeak not exist
If they both exists strdup them both into node newspeak/oldspeak
Set two NULL pointers for right and left children nodes

2.4.2 void node_delete(Node **n)

Free the memory for the two strings
Free the node
Set the pointer to NULL

2.4.3 void node_print(Node *n)

Print newspeak and oldspeak in each node
Print out only oldspeak if node doesn't contain newspeak

2.5 bf.c

bf.c holds the bloom filter ADT which is used in our program to see whether a word is oldspeak/badspeak. It can return false positives but is a good way to quickly check if a word is supposed to be looked at in the hash table. Our bloom filter has three different arrays holding three different salts and a pointer to the bit vector of the bloom filter.

2.5.1 BloomFilter *bf_create(uint32_t size)

Allocate memory for bloom filter
Create bitvector using bv_create
Set the salts to each array
Return the bf

2.5.2 void bf_delete(BloomFilter **bf)

Free the bit vector
Free the bloom filter
Set the bloom filter pointer to NULL

2.5.3 uint32_t bf_size(BloomFilter *bf)

Return the length of the bit vector in the bloom filter

2.5.4 void bf_insert(BloomFilter *bf, char *oldspeak)

Hash the oldspeak with each salt and set the bit at those indices to 1

2.5.5 bool bf_probe(BloomFilter *bf, char *oldspeak)

Checks if the three indices of a hashed oldspeak are all 1
Return true if they are
Otherwise false

2.5.6 uint32_t bf_count(BloomFilter *bf)

Return number of 1's in the bit vector

2.5.7 void bf_print(BloomFilter *bf)

Print out the bit vector

2.6 bv.c

bv.c holds our bit vector ADT and is represented as a array of bits. Therefore it only needs to hold the size of the bit vector and an array of bytes. We use the bit vector in the bloom filter ADT to show which words might be oldspeak/newspeak.

2.6.1 BitVector *bv_create(uint32_t length)

Use malloc to allocate memory for the bitvector
Set bv length to legnth
Allocate memory for the bytes with calloc
return bv

2.6.2 void bv_delete(BitVector **bv)

Free the bytes
Free the bitvector
Set the pointer to NULL

2.6.3 uint32_t bv_length(BitVector *bv)

Return length of bit vector

2.6.4 bool bv_set_bit(BitVector *bv, uint32_t i)

Perform shifting on a byte to bit that needs to be set
Or it with the byte at the ith index / 8
If out of range return false

2.6.5 bool bv_clr_bit(BitVector *bv, uint32_t i)

Perform shifting on a byte to bit that needs to be cleared
Invert the bitmask
And it with the byte at the ith index / 8
If out of range return false

2.6.6 bool bv_get_bit(BitVector *bv, uint32_t i)

Perform shifting on a byte to bit that needs to be grabbed
And it with the byte at the ith index / 8
If out of range return false
Return false if bit is 0, true if bit is 1

2.6.7 void bv_print(BitVector *bv)

Print out each bit by getting the bit at each position and printing them out one by one

2.7 parser.c

Parser.c is the file that helps us parse through text using REGEX.

2.8 speck.c

speck.c is the file that contains a function that will allow us to hash a word with the SPECK cipher.

3 Structure of program

Overall the structure of our program is as follows: The main function uses bloom filters, and hash tables along with speck.c and parser.c to read in words and see whether they are oldspeak/newspeak. The bloom filter ADT uses the bit vector ADT stored in bv.c. The hashtable ADT uses bst.c to get the array of trees it requires and bst.c uses node.c since each binary tree is a node. This is how all of the files are used together to run banhammer.c