Assignment 7: The Great Firewall of Santa Cruz - Design

Thadeus Ballmer

November 2021

1 Introduction

People are increasingly relegating themselves to a digital life created and controlled by multinational corporations of unprecedented size. With more people dissolving into the Metaverse, the possibility for a 1984, dystopian nightmare only rises. The trademark of any 1984-esque regime is utilizing technology to manipulate the actions, speech, and thoughts of the population. This assignment is a study on how written speech can be controlled. By studying such destructive technology before widespread implementation, the chance of a successful united resistance improves.

2 Bloom Filter

The brain of this beast is the Bloom Filter. The Bloom Filter is a bit index which helps identify if a word is probably *oldspeak*. How does this work? The bit index is initialized to 0's. Then, words which are deemed unsuitable are hashed and the bit at the hashed value is set to 1. To check if a word is potentially improper, just check if the bit at its hash value is 1. Since the hash functions used are *deterministic*, there is no chance of an indicated bad word slipping through the Bloom Filter. Hash collisions are possible which means that a deterministic data structure must be implemented.[1] In order to insure maximum specificity, three different hash functions will utilize 3 distinct salts provided in `salts.h`.

The function `bf_create` is the constructor for the Bloom Filter. It initializes the three salts and implements the bit vector ADT.

```
BloomFilter *bf_create(uint32_t size)
    primary = primary salt
    secondary = secondary salt
    tertiary = tertiary salt
    Dynamically allocate memory to the Bloom Filter
    Initialize Bit Vector
```

[1] Hash Tables explained in Section 4

The function `bf_delete` is the destructor for the Bloom Filter.

```
void bf_delete(BloomFilter **bf)
    free memory
    set pointer to null
```

The function `bf_size` returns the size of the Bloom Filter

```
uint32_t bf_size(BloomFilter *bf)
    return length of bit vector
```

The function `bf_insert` inserts *oldspeak* into the Bloom Filter.

```
void bf_insert(BloomFilter *bf, char *oldspeak)
    i = hash(primary, oldspeak)
    bv_set_bit(bf, i)
    i = hash(secondary, oldspeak)
    bv_set_bit(bf, i)
    i = hash(tertiary, oldspeak)
    bv_set_bit(bf, i)
```

The function `bf_probe` checks if all three hashes of the probing word are set in the Bloom Filter.

```
bool bf_probe(BloomFilter *bf, char *oldspeak)
    i = hash(primary, oldspeak)
    if (bv_get_bit(bf, i) equals 0)
        return false
    i = hash(secondary, oldspeak)
    if (bv_get_bit(bf, i) equals 0)
        return false
    i = hash(tertiary, oldspeak)
    if (bv_get_bit(bf, i) equals 0)
        retrun false
    return true
```

The function `bf_count` returns the number of set bits.
The function `bf_print` is a debugging function which prints out all bits in the Bloom Filter.

3 Bit Vectors

This is where the bit vector section of the Bloom Filter is implemented.
The function `bv_create` is the constructor of the bit vector.

```
BitVector *bv_create(uint32_t length)
    length = length
    Dynamically allocate memory to the bit vector of length length using calloc
    return pointer to vector
```

The function `bv_delete` is the destructor for the bit vector.
The function `bv_length` returns the length of the bit vector.
The function `bv_set_bit` sets the bit at index `i`.

```
bool bv_set_bit(BitVector *bv, uint32_t i)
    if (0 > i >= bv_length)
        return false
    set bit at index i
    return true
```

The function `bv_clear_bit` clears the bit at index `i`.

```
bool bv_clr_bit(BitVector *bv, uint32_t i)
    if (0 > i >= bv_length)
        return false
    clear bit at index i
    return true
```

The function `bv_get_bit` returns the value of the bit at index `i`.

```
bool bv_get_bit(BitVector *bv, uint32_t i)
    if (0 > i >= bv_length)
        return false
    return vector[i]
```

The function `bf_print` is a debugging function which prints the bits in the bit vector.

4 Hash Table

The hash table itself implements an array of binary search trees. This will resolve any hash collisions because the *oldspeak* values are stored deterministically.

The function `ht_create` is the constructor for the hash table.

```
HashTable *ht_create(uint32_t size)
    salt = hash salt
    size = size
    Dynamically allocate memory for trees of size size
    return pointer to trees
```

The function `ht_delete` is the destructor for the hash table.

```
void ht_delete(HashTable **ht)
    Loop through trees and delete each node
    free memory
    set pointer to null
```

The function `ht_size` returns the size of the hash table.

The function `ht_lookup` searches for a node with the *oldspeak* translation or returns a NULL pointer if given *oldspeak* is *badspeak*.

```
Node *ht_lookup(HashTable *ht, char *oldspeak)
    i = hash(oldspeak)
    while ()
        if (trees[i] node is less than oldspeak)
            move to the right node
        else if (trees[i] node is greater than oldspeak)
            move to the left node
        if (oldspeak node is reached)
            return node value
```

The function `ht_insert` inserts a oldspeak node with its newspeak translation into the hash table.

```
void ht_insert(HashTable *ht, char *oldspeak, char *newspeak)
    i = hash(oldspeak)
    while ()
        if (trees[i] left node is NULL)
            insert oldspeak Node
            break
        else if (trees[i] left node is NULL)
            insert oldspeak Node
            break
```

The function `ht_count` returns the number of *non-badspeak oldspeak* words in the hash table.

The function `ht_avg_bst_size` returns the average binary search tree size.

```
double ht_avg_bst_size(HashTable *ht)
    for (Loop through hash table size)
        sum += bst_size(trees[i])
    return sum / ht_count()
```

The function `ht_avg_bst_height` returns the average binary search tree height.

```
double ht_avg_bst_height(HashTable *ht)
    for (Loop through hash table size)
        sum += bst_height(trees[i])
    return sum / ht_count()
```

The function `ht_print` is a debugging function used to print out the conetents of the hash table.

5 Nodes

The binary search trees which populate the hash table are made up of nodes.

The function `node_create` is the constructor for a node.

```
Node *node_create(char *oldspeak, char *newspeak)
    use strcpy to copy oldspeak and newspeak into node
    dynamically allocate memory for Node pointer
```

The function `node_delete` is the destructor of a node.

```
void node_delete(Node **n)
    free memory of oldspeak and newspeak, set their pointers to NULL
    free memory for node and set its pointer to NULL
```

The function `node_print` is a debugging function which prints out the contents of a node.

6 Binary Search Trees

The hash table is populated with binary search trees constructed of nodes.

The function `bst_create` is the constructor of a binary search tree.

```
Node *bst_create(void)
    return NULL node
```

The function `bst_delete` is the destructor for a binary search tree.

```
void bst_delete(Node **root)
    if (root)
        bst_delete(root->left)
        bst_delete(root->right)
        node_delete(root)
```

The function `bst_height` returns the height of the binary search tree.

The function `bst_size` returns the size of the binary search tree.

The function `bst_find` searches for a node containing the given `oldspeak`.

```
Node *bst_find(Node *root, char *oldspeak)
    if (strcmp(root, oldspeak) == 0)
        return root
    else if (strcmp(root->oldspeak, oldspeak))
        bst_find(root->left, oldspeak)
```

```
    else if (strcmp(root->oldspeak, oldspeak) < 0)
        bst_find(root->right, oldspeak)
    return NULL
```

The function `bst_insert` inserts a node into the binary search tree

```
Node *bst_insert(Node *root, char *oldspeak, char *newspeak)
    n = node_create(oldspeak, newspeak)
    while()
        if (strcmp(root, oldspeak) == 0)
            return root
        else if (strcmp(root->oldspeak, oldspeak))
            // Continue to left side
        else if (strcmp(root->oldspeak, oldspeak) < 0)
            // Continue to right side

    Insert node at proper placement
```

The function `bst_print` is a debugging function which prints out each node. This problem is solved using post-order traversal.

7 Banhammer

`Banhammer` utilizes all the ADT's presented in its main function.

```
main()
    read through CLI options including: help, ht_size, bf_size, and stats.

    Initialize Bloom Filter and the hash table

    while (Scaning badspeak.txt doesn't return EOF)
        bf_insert(oldspeak)
        ht_insert(oldspeak, NULL)

    while (Scanning newspeak.txt doesn't return EOF)
        bf_insert(oldspeak)
        ht_insert(oldspeak, newspeak)

    Compile regex

    Initialize two priority queues to store each word crime

    while ((word = next_word(stdin)) != NULL)
```

```
    if (bf_probe(word))
        Node *p = ht_lookup(word)
        if (p and p->newspeak is NULL)
            enqueue(bad_list, p)
        else if (p and p->newspeak)
            enqueue(old_list, p)

if (stats)
    print stats
else if (pq_empty(bad_list) is false and pq_empty(old_list) is false)
    print (Corresponding message)
    pq_print(bad_list)
    pq_print(old_List)
else if (pq_empty(bad_list) is false and pq_empty(old_list) is true)
    print (Corresponding message)
    pq_print(bad_list)
else if (pq_empty(bad_list) is true and pq_empty(old_list) is false)
    print (Corresponding message)
    pq_print(old_list)

clear words
free regex
close files
delete all data structures
```