# The Great Firewall of Santa Cruz - CSE13S Assignment 7

Nicholas Reis

November 24, 2021

---

## Program Description

In this assignment, I wrote a program that filters out words that are deemed to be old and either replace them with a new, better word or if there is no replacement, the user is scolded for their thoughtcrimes. To accomplish this, I implemented a bloom filter ADT as an initial check for whether a word is marked as bad, a hash table ADT as a definitive check for this (including binary search trees in the case of a hash collision), and a bit vector ADT for setting, clearing, and getting bit values. I also used regular expressions to determine if a user input is a word.

---

## Required Files

- banhammer.c
- messages.h
- salts.h
- speck.h
- speck.c
- ht.h
- ht.c
- bst.h
- bst.c
- node.h
- node.c
- bf.h
- bf.c
- bv.h
- bv.c
- parser.h
- parser.c
- Makefile
- README.md
- DESIGN.pdf
- WRITEUP.pdf

---

## Banhammer

banhammer.c is the test harness file that contains the code that will allow the program to run. It reads in badspeak words from badspeak.txt and oldspeak-newspeak translations from newspeak.txt before then

taking in user input and correcting their use of bad words. There are options available to alter the size of hash tables and bloom filters, along with printing out a help/usage message and program statistics.

**Pseudocode**

```
inlcude necessary header files
define OPTIONS "ht:f:s"
define HT_SIZE 2^16
define BF_SIZE 2^20
define WORD (regex expression)

void print_usage():
    print out usage information
    end program

void print_mixspeak():
    print out mixspeak message

void print_thoughtcrime():
    print out thoughtcrime only message

void print_wrongthink():
    print out wrongthink only message

void print_stats(bst_size, bst_height, branches_traversed, ht_load, bf_load):
    print out statistics

int main():
    create booleans for help, stats, thoughtcrime, and wrongthink
    hash_size = HT_SIZE
    bf_size = BF_SIZE
    badspeak_file = open badspeak.txt
    newspeak_file = open newspeak.txt
    string word = NULL

    create and compile regex

    while loop that uses getopt to access the options:
    switch:
    case 'h' and default:
        set help boolean to true
    case 's':
        set stats boolean to true
    case 't':
        hash_size = optarg
    case 'f':
        bf_size = optarg

    if (help):
        print_usage()

    initialize ht, bf with hash_size and bf_size

    initialize bst's for badspeak and oldspeak
```

```
while (not EOF):
    read in word from badspeak_file
    insert word in ht and bf

while (not EOF):
    read in word from newspeak_file
    insert words into ht and bf

while (there are still words to be read from stdin):
    for (i; i < length of word; i++):
        lowercase each letter
    if (word ends with apostrophe):
        set last character to NULL

    if (word is in bf):
        node n = ht_lookup(ht, word)
        if (node is not NULL):
            if (node's newspeak is NULL):
                insert word into badspeak bst
                thoughtcrime = true
            else:
                insert word into oldspeak bst
                wrongthink = true

if (stats):
    print_stats()
else:
    if (thoughtcrime and wrongthink):
        print_mixspeak()
        bst_print() badspeak and oldspeak
    else if (thoughtcrime):
        print_thoughtcrime()
        bst_print(badspeak)
    else if (wrongthink):
        print_wrongthink()
        bst_print(oldspeak)

fclose() files
clear words parsing module
free regex, ht, bf, badspeak, oldspeak
```

---

1. I define some variables and options at the top of the file, along with my regex expression that accepts specific words from the input.
2. I wrote a few helper functions that simply print out messages when I call them in specific areas of the program.
3. At the beginning of main, I declare most of the variables I need for the program, including booleans and FILE *'s
4. After accepting user options, the program initializes the hash table, bloom filter, and binary search trees that are necessary.
5. It then reads in badspeak words and oldspeak-newspeak translations from specific files and inserts them

into the hash table and bloom filter.

6. The next while loops reads in words (defined by regex) from stdin, sets them to lowercase, and then checks if they are in the bf. If they are, the program then check that they are in the ht. If they are, it checks if the word has a newspeak translation, and then adds the word to the correct bst for later use.
7. If stats are enabled, it is all that will be printed. Otherwise, the program checks if mixspeak, thoughtcrime, or badspeak was committed, and then prints out the correct message and bst(s) accordingly.
8. The final part of the program is freeing the allocated memory utilized.

---

## Bloom Filter

Within bf.c, I implemented code to create an ADT called BloomFilter. It provide functions that allow me to use a bloom filter that can check if a word is possibly already in the filter or definitely not in the filter. These functions include: creating a bloom filter, deleting a bloom filter, returning the size of a bloom filter, inserting a word into the bloom filter, checking if a word is in the bloom filter, returning the number of set bits, and a debug function.

### Pseudocode

```
inlcude necessary header files

struct BloomFilter:
    uint64_t primary[2]
    uint64_t secondary[2]
    uint64_t tertiary[2]
    BitVector *filter

BloomFilter *bf_create(uint32_t size):
    malloc() memory for a BloomFilter bf
    if (bf created successfully):
        bf primary[0] = SALT_PRIMARY_LO
        bf primary[1] = SALT_PRIMARY_HI
        bf secondary[0] = SALT_SECONDARY_LO
        bf secondary[1] = SALT_SECONDARY_HI
        bf tertiary[0] = SALT_TERTIARY_LO
        bf tertiary[1] = SALT_TERTIARY_HI
        bf filter = bv_create(size)
        return bf

void bf_delete(BloomFilter *bf):
    if (bf pointer exists):
        bv_delete(bf filter)
        free(bf)
        set the pointer to the bf to NULL
    return

uint32_t bf_size(BloomFilter *bf):
    return bv_length(bf filter)

void bf_insert(BloomFilter *bf, char *oldspeak):
    bit = hash(bf primary, oldspeak)
```

```
    bv_set_bit(bf filter, bit)
    bit = hash(bf secondary, oldspeak)
    bv_set_bit(bf filter, bit)
    bit = hash(bf tertiary, oldspeak)
    bv_set_bit(bf filter, bit)
    return

bool bf_probe(BloomFilter *bf, char *oldspeak):
    bit = hash(bf primary, oldspeak)
    if (!bv_get_bit(bf filter, bit)):
        return false
    bit = hash(bf secondary, oldspeak)
    if (!bv_get_bit(bf filter, bit)):
        return false
    bit = hash(bf tertiary, oldspeak)
    if (!bv_get_bit(bf filter, bit)):
        return false
    return true

uint32_t bf_count(BloomFilter *bf):
    count = 0
    for (i; i < bf_size(bf); i++):
        if (bv_get_bit(bf filter, i):
            count++
    return count

void bf_print(BloomFilter *bf):
    print out all of the salts
    bv_print(bf filter)
    return
```

---

1. bf_create() allocates memory for a BloomFilter, sets all the salts to the values in salts.h, and creates a BitVector with a given size.
2. bf_delete() frees the memory allocated to the BV and BF, and then sets the pointer to the BF to NULL.
3. bf_size() just returns bv_length() of the BloomFilter's BitVector.
4. bf_insert() hashes the oldspeak with each salt and sets each of the three bits in the BV.
5. bf_probe() also hashes the oldspeak with each salt, but instead gets each bit and returns false if any of them are not already set.
6. bf_count() iterates through each bit in the BV and counts the amount of set bits in the BF.
7. bf_print() prints out every salt and every bit in the BF's bit vector.

---

## Bit Vector

Within bv.c, I implemented code to create an ADT called BitVector. It provide functions that allow me to use a bit vector for use in the bloom filter. These functions include: creating a bit vector, deleting a bit vector, returning the length of a bit vector, setting a bit, clearing a bit, getting a bit, and a debug function.

**Pseudocode**

```
inlcude necessary header files

struct BitVector:
    uint32_t length
    uint8_t *vector

BitVector *bv_create(uint32_t length):
    malloc() memory for a BitVector bv
    if (bv created successfully):
        bv length = length
        calloc() memory for the bv's vector
        return bv
    else:
        return NULL

void bv_delete(BitVector *bv):
    if (bv pointer exists):
        free(bv vector)
        free(bv)
        set the pointer to the bv to NULL
    return

uint32_t bv_length(BitVector *bv):
    return bv length

bool bv_set_bit(BitVector *bv, uint32_t i):
    if (i >= length):
        return false
    set the bit in bv vector by or'ing with a left shifted 1
    return true

bool bv_clr_bit(BitVector *bv, uint32_t i):
    if (i >= length):
        return false
    set the bit in bv vector by and'ing with an inversed, left shifted 1
    return true

bool bv_get_bit(BitVector *bv, uint32_t i):
    if (i >= length OR bit i is 0):
        return false
    return true

void bv_print(BitVector *bv):
    for (i; i < bv length; i++):
        print out bit i
```

---

1. bv_create() allocates memory for a BitVector, sets the length to the given input, sets all the bits to 0, and returns the BV. If the memory is not allocated, return NULL.
2. bv_delete() frees the memory allocated to the BV and then sets the pointer to the BV to NULL.
3. bv_length() just returns the length member of the given BitVector.

4. bv_set_bit() uses a 0x1 that is left shifted i%8 times and then or'ed with a byte in the bit vector to set the bit at a specific location. It returns false if the bit is out of range and true otherwise.
5. bv_clr_bit() uses a 0x1 that is left shifted i%8 times, nor'ed, and then and'ed with a byte in the bit vector to clear the bit at a specific location. It returns false if the bit is out of range and true otherwise.
6. bv_get_bit() uses uses similar logic to bv_set_bit() to check if a bit is a 0 before returning false, and true if it is a 1. It also returns false if the bit is out of range.
7. bv_print() prints out every bit in the bit vector.

---

## Hash Table

Within ht.c, I implemented code to create an ADT called HashTable. It provide functions that allow me to use a hash table for storing each translation from oldspeak to newspeak, along with any badspeak words. These functions include: creating a hash table, deleting a hash table, returning the size of a hash table, looking up an entry, inserting a node, returning the count of items in a hash table, returning the average size and height of each binary search tree, and a debug function.

### Pseudocode

```
inlcude necessary header files

struct HashTable:
    uint64_t salt[2]
    uint32_t size
    Node **trees

HashTable *ht_create(uint32_t size):
    malloc() memory for a HashTable ht
    if (ht created successfully):
        ht size = size
        calloc() memory for the trees
        for (i; i<size; i++):
            ht trees[i] = bst_create()
        ht salt[0] = SALT_HASHTABLE_LO
        ht salt[1] = SALT_HASHTABLE_HI
    return ht

void ht_delete(HashTable **ht):
    if (ht pointer exists):
        for (i; i<ht size; i++):
            bst_delete(ht trees[i])
        free(array of trees)
        free(ht)
        set the pointer to the ht to NULL
    return

uint32_t ht_size(HashTable *ht):
    return ht size

Node *ht_lookup(HashTable *ht, char *oldspeak):
    increment lookups
    index = hash(oldspeak)
```

```
    return bst_find(ht trees[index], oldspeak)

void ht_insert(HashTable *ht, char *oldspeak, char *newspeak):
    increment lookups
    index = hash(ht salt, oldspeak)
    bst_insert(ht trees[index], oldspeak, newspeak)
    return

uint32_t ht_count(HashTable *ht):
    count = 0
    for (i; i < ht size; i++):
        if (ht[i] is not NULL):
            count++
    return count

double ht_avg_bst_size(HashTable *ht):
    total_size = 0
    for (i; i < ht size; i++)
        if (h[i] is not NULL):
            total_size = total_size + bst_size(ht[i])
    return total_size / ht_count(ht)

double ht_avg_bst_height(HashTable *ht):
    total_height = 0
    for (i; i < ht size; i++)
        if (h[i] is not NULL):
            total_height = total_height + bst_height(ht[i])
    return total_height / ht_count(ht)

void ht_print(HashTable *ht):
    if (ht exists):
        for (i; i < ht size; i++):
            print out each binary search tree
    return
```

---

1. ht_create() allocates memory for a HashTable, sets the size to the given input, creates empty BSTs in each index of the HT, and sets the salt before returning the created HT.
2. ht_delete() frees the memory allocated to the array of BSTs and the HT and then sets the pointer to the HT to NULL.
3. ht_size() just returns the size member of the given HashTable.
4. ht_lookup() gets the index of the BST from the given oldspeak by hashing it, and then calls bst_find() to find the node in the specific BST that has the oldspeak translation in it.
5. ht_insert() also hashes the oldspeak to get the index of the BST, and then calls bst_insert() to insert the oldspeak/newspeak pair into that BST.
6. ht_count() iterates through the HT and increments a variable count each time there is a BST that is not NULL (meaning there is at least one value in it).
7. ht_avg_bst_size() iterates through the HT and calls bst_size() each time it finds a non-NULL BST, and then divides the found total size by the number of non-NULL BSTs.
8. ht_avg_bst_height() does the same thing but with bst_height() calls.

9. ht_print() prints out each BST in the ht.

---

## Node

Within node.c, I implemented code to create an ADT called Node. It provides functions that allow me to create nodes that will be used in binary search trees to solve hash collisions. These functions include: creating a node, deleting a node, and a debug function for the nodes.

### Pseudocode

```
include necessary header files

Node *node_create(oldspeak, newspeak):
    use malloc to dynamically allocate memory for a Node *n
    if (n was created):
        set n's oldspeak to a copy of oldspeak
        if (newspeak is not NULL):
            set n's newspeak to a copy of newspeak
        else:
            set n's newspeak to NULL
        set n's left to NULL
        set n's right to NULL
    return the node

void node_delete(Node **n):
    free() the memory allocated to oldspeak
    free() the memory allocated to newspeak
    free() the memory allocated to the node
    set the pointer to the node to NULL
    return

void node_print(node *n):
    if (n's oldspeak and newspeak exist):
        print out oldspeak and newspeak
    else if (oldspeak exists):
        print out oldspeak
    return
```

1. node_create() allocates memory for the created node, its oldspeak, and its newspeak (if the newspeak argument is not NULL). It copies the argument strings of oldspeak and newspeak into the node's members of the same name. It then sets the node's left and right children to NULL before returning the node.
2. node_delete() frees the memory for oldspeak and newspeak, as strdup() allocates memory for them, so they must be freed here. It also frees memory allocated to the node and sets the node's pointer to NULL.
3. node_print() is a debug function that either prints out both the node's newspeak and oldspeak, or just the oldspeak if there is no newspeak translation.

---

## Binary Search Tree

Within bst.c, I implemented code to create an ADT for binary search trees. These are important to resolve the issue of a hash collision in my hash table, as in the case of a hash collision, a node for the word will be added to the tree so that each word can still be stored in a reachable location. The functions in this file include: creating a BST, deleting a BST, getting the height of a BST, getting the size of a BST, finding a node in the tree, inserting a node in the tree, and a debug function.

### Pseudocode

```
include necessary header files

Node *bst_create(void):
    return NULL

void bst_delete(Node **root):
    if (root exists):
        recursive call root's left child
        recursive call root's right child
        node_delete(root)
    return

uint32_t bst_height(Node *root):
    if (root exists):
        return 1 + max(recursive call left, recursive call right)

uint32_t bst_size(Node *root):
    if (root exists):
        return 1 + recursive call left + recursive call right

Node *bst_find(Node *root, char *oldspeak):
    if (root exists):
        if (root's oldspeak is greater than oldspeak):
            increment branches
            recursive call root's left child
        else if (root's oldspeak is less than oldspeak):
            increment branches
            recursive call root's right child
    return root

Node *bst_insert(Node *root, char *oldspeak, char *newspeak):
    if (root exists):
        if (root's oldspeak is greater than oldspeak):
            increment branches
            root's left = recursive call left child
        else if (root's oldspeak is less than oldspeak):
            increment branches
            root's right = recursive call right child
        return root
    return node_create(oldspeak, newspeak)

void bst_print (Node *root):
    if (root exists):
        recursive call root's left child
        node_print(root)
```

```
    recursive call root's right child
return
```

---

1. bst_create() returns NULL, as the created tree is simply a NULL pointer.
2. bst_delete() does a postorder deletion of each node, meaning it deletes leaf nodes first until it reaches the root node last.
3. bst_height() returns the height of the BST by finding the longest path down to the bottom of the tree and returning that value.
4. bst_size() uses a similar method except it just counts each node in the entire tree.
5. bst_find() goes down the left or right side of the current node depending on whether the current node's oldspeak string is larger or smaller (lexicographically) than the oldspeak argument. It then returns the node if it is found.
6. bst_insert() does a similar process, except it creates a node in the correct location in the tree.
7. bst_print() performs an inorder traversal to print out each node in the tree using node_print().

---