Deeraj Gurram

dgurram@ucsc.edu

28 November 2021

# CSE 13S Fall 2021

# Assignment 7 - The Great Firewall of Santa Cruz

# Design Document

## Description of the Program Files:

This assignment consists of various program files that perform different tasks. The first program file is bf.c which contains the BloomFilter ADT and implements three different salts and hashing functions to ensure that any oldspeak found is actually oldspeak. The second program file is hash.c which implements the hashing functions. The third file is bv.c which will be used to get, set, and clear bits in various other files and functions. The fourth file is ht.c and implements a hash table to store words that are oldspeak along with their newspeak translations, and map them to specific keys in the table. The fifth program file is node.c and will contain the oldspeak that is found as the key, with the newspeak translation acting as the child of the tree. The sixth file is called bst.c and will contain recursive functions to properly order the new insertions of oldspeak into the hash table, and take care of any errors that might occur. The seventh file is called parser.c and contains the implementation to look through an input stream and find all the occurrences of oldspeak. The eighth and final program file is called banhammer.c and contains the command line input handling which takes care of size specifications in certain functions, the printing of statistics, the printing of a help message, and also contains the program implementation to execute all the program files.

## **Deliverables:**

1. banhammer.c: This contains the main() function and command line input handling
2. messages.h: Defines the mixspeak, badspeak, and goodspeak messages that are used in banhammer.c.
3. salts.h: Defines the primary, secondary, and tertiary salts to be used in the Bloom filter implementation. Also defines the salt used by the hash table in the hash table implementation.
4. speck.h: Defines the interface for the hash function using the SPECK cipher.
5. speck.c: Contains the implementation of the hash function using the SPECK cipher.
6. ht.h: Defines the interface for the hash table ADT.
7. ht.c: Contains the implementation of the hash table ADT.
8. bst.h: Defines the interface for the binary search tree ADT.
9. bst.c: Contains the implementation of the binary search tree ADT.
10. node.h: Defines the interface for the node ADT.
11. node.c: Contains the implementation of the node ADT.
12. bf.h: Defines the interface for the Bloom filter ADT.
13. bf.c: Contains the implementation of the Bloom filter ADT.
14. bv.h: Defines the interface for the bit vector ADT.
15. bv.c: Contains the implementation of the bit vector ADT.
16. parser.h: Defines the interface for the regex parsing module.
17. parser.c: Contains the implementation of the regex parsing module.
18. Makefile: Clang formats the program files, builds the banhammer executable, removes files that are compiler generated, and formats all source code including the header files

19. README.md: Gives a brief description of the program and Makefile, and lists the command line options that the program accepts. Also notes any errors or bugs found in the code and gives credit to sources of code or help.
20. WRITEUP.pdf: This file will include an explanation on what I learned from the binary search tree traversals, an analysis on which conditions the searches performed well under, graphs explaining the performance of searches when inputs are varied, and an examination of each graph that is produced.
21. DESIGN.pdf: Provides a description of the assignment and program files, the deliverables, pseudocode for each of the .c files, and lists sources of credit for parts of the code. Also describes the design and design process for the program.

## <u>Pseudocode:</u>

### <u>bf.c</u>

```
BloomFilter *bf_create(uint32_t size)
    Create a BloomFilter bf and allocate it memory of size BloomFilter
    Set primary[0] equal to the SALT_PRIMARY_LO defined in salts.h
    Set primary[1] equal to the SALT_PRIMARY_HI defined in salts.h
    Set secondary[0] equal to the SALT_SECONDARY_LO defined in salts.h
    Set secondary[1] equal to the SALT_SECONDARY_HI defined in salts.h
    Set tertiary[0] equal to the SALT_TERTIARY_LO defined in salts.h
    Set tertiary[1] equal to the SALT_TERTIARY_HI defined in salts.h
    Create a BitVector and set it equal to *filter

void bf_delete(BloomFilter **bf)
    Call bv_delete on *filter

uint32_t bf_size(BloomFilter *bf)
    Return the bv_length of *filter

void bf_insert(BloomFilter *bf, char *oldspeak)
    Insert argument oldspeak into the next available array in the three hashing
    functions
    Call bv_set_bit on the BitVector *filter and the index the oldspeak is inserted into

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

```
    Hash each salt with the oldspeak
    Check if the bit at each index is set
        Return true
    Else
        Return false
```

**uint32_t bf_count(BloomFilter *bf)**

```
    Use a for loop to go through the primary array where the incrementer is less than the
    length of primary
        Check if a bit is set
            Increment a counter j
    Use a for loop to go through the secondary array where the incrementer is less than the
    length of secondary
        Check if a bit is set
            Increment a counter j
    Use a for loop to go through the primary array where the incrementer is less than the
    length of tertiary
        Check if a bit is set
            Increment a counter j
    Return the counter j
```

**void bf_print(BloomFilter *bf)**

```
    Go in a loop and print the values of each salts array
    Call bv_print on the filter
```

## bv.c

**Struct BitVector**

```
    Define the BitVector struct with a length of size uint32_t and *vector of size uint8_t
```

**BitVector *bv_create(uint32_t length)**

```
    Create a BitVector bv and allocate it memory of size BitVector
    If the BitVector can’t be allocated memory
        Return NULL
    Else
        Return a BitVector *
        Loop through the BitVector and call bv_clr_bit on each *vector
    Set the struct member length equal to argument length
```

**void bv_delete(BitVector **bv)**

```
    Check if *bv exists
        Free *bv
        Set *bv to NULL
```

**uint32_t bv_length(BitVector *bv)**

```
    Return the length of bv
```

**bool bv_set_bit(BitVector *bv, uint32_t i)**

```
    Check if bv and i exist
        Set the i’th bit in bv to 1 using bit shifting
        Return true
    Else
```

```
        Return false
```

**bool bv_clr_bit(BitVector *bv, uint32_t i)**

```
    Check if bv and i exist
        Set the i’th bit in bv to 0 using bit shifting
        Return true
    Else
        Return false
```

**bool bv_get_bit(BitVector *bv, uint32_t i)**

```
    Check if bv and i exist
        Store the i’th bit in bv using bit shifting inside of a variable
        Check if the variable is equal to 1
            Return true
        Else
            Return false
    Return false
```

**void bv_print(BitVector *bv)**

```
    Create a for loop to go through the bits in bv, the iterator should be less than the length of bv
        Use bv_get_bit to see if the current bit is equal to 0 or 1
        Print out the value of the bit
```

## ht.c

**Struct HashTable**

```
    Define the HashTable struct with members array salt of size 2, integer size, and Node
    **trees
```

**HashTable *ht_create(uint32_t size);**

```
    Create a HashTable called ht and allocate it memory of size HashTable
    Set salt[0] equal to the SALT_HASHTABLE_LO defined in salts.h
    Set salt[1] equal to the SALT_HASHTABLE_HI defined in salts.h
    Set the struct member size equal to argument size
    Create a Node called trees
```

**void ht_delete(HashTable **ht);**

```
    Check if ht and ht's trees exists
        Free the memory allocated to the Node *trees
        Set *trees equal to NULL
```

**uint32_t ht_size(HashTable *ht);**

```
    Return the size of ht
```

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

```
    Hash the oldspeak to calculate the index to look at
    Search ht for the key that matches the calculated index
    If the node is found
        Return a pointer to the node
    Else
```

```
        Return a NULL pointer
```

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

```
    Hash the oldspeak to calculate the index to insert into
    Insert the oldspeak into the hashtable
    Set argument newspeak as the child of the specified node
```

**uint32_t ht_count(HashTable *ht);**

```
    Create a for loop where the iterator is less than ht's size
        Check if the current bst is non-NULL
            Increment a counter j
    Return the counter j
```

**double ht_avg_bst_size(HashTable *ht);**

```
    Create a for loop where the iterator is less than ht's size
        Increment a variable k by the bst_size of the current tree
        Check if the current bst is non-NULL
            Increment a counter j
    Return k/j
```

**double ht_avg_bst_height(HashTable *ht);**

```
    Create a for loop where the iterator is less than ht's size
        Increment a variable k by the bst_height of the current tree
        Check if the current bst is non-NULL
            Increment a counter j
    Return k/j
```

**void ht_print(HashTable *ht);**

```
    Loop through ht, where the iterator is less than ht’s size
        Print the value of the current Node in ht
```

## node.c

**struct Node**

```
    Define the Node struct with members oldspeak, newspeak, Node left, and Node right
```

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

```
    Create a Node n and allocate memory of size Node to it
    Copy the oldspeak into a variable called oldspeak_cp
    Copy the newspeak into a variable called newspeak_cp
    Set the struct member oldspeak equal to oldspeak_cp
    Set the struct member newspeak equal to newspeak_cp
```

**void node_delete(Node **n);**

```
    Free the memory of oldspeak
    Free the memory of newspeak
    Free the memory of n
    Set *n equal to NULL
```

**void node_print(Node *n);**

```
    Check if n contains oldspeak and newspeak
        Print the struct member oldspeak and the struct member newspeak
    If n only contains oldspeak
        Print out the struct member oldspeak
```

## bst.c

**Node *bst_create(void)**

```
    Call node_create with arguments NULL and NULL
```

**void bst_delete(Node **root)**

```
    Perform postorder traversal
    Free the current node using node_delete(*root)
```

**uint32_t bst_height(Node *root);**

```
    Perform postorder traversal on the tree starting at root *root
        Increment a counter j when a new level of the tree is reached
    Return the counter j
```

**uint32_t bst_size(Node *root);**

```
    Perform traversal on the tree
        Increment a counter j everytime a new node is reached
    Return j
```

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

```
    Perform postorder traversal on the tree starting at root *root
        Check if the left child is equal to argument oldspeak
            Return the pointer to the node
        Else
        Return a NULL pointer
```

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

```
Check if bst_find with arguments root and oldspeak returns a NULL pointer
    Insert a new Node into the bst with the root at *root
    The left child should equal oldspeak
    The right child should equal newspeak
```

**void bst_print(Node *root);**

```
Perform inorder traversal of the bst
    Print out each node using node_print when the traversal is performed
```

## **Credit:**

1. Used the deliverables descriptions from Dr. Long on the asgn7.pdf file.