CSE 13S Fall 2021
James Gu
jjgu@ucsc.edu
28 November 2021

Assignment 7: The Great Firewall of Santa Cruz

Description of Program:

You have been selected to be the Dear and Beloved Leader of the Glorious People's Republic of Santa Cruz. As the new leader your goal is to censor and filter Internet content as to avoid corrupting young children's minds. This project involves censoring "badthink" and replacing "oldspeak" with newspeak. This will be done using a Bloom filter, hash tables, and binary trees. These are very prevalent computer science concepts that we will be practicing in this assignment.

Files:

banhammer.c:
This contains main() and may contain the other functions necessary to complete the assignment.

messages.h:
Defines the mixspeak, badspeak, and goodspeak messages that are used in banhammer.c. Do not modify this.

salts.h:
Defines the primary, secondary, and tertiary salts to be used in your Bloom filter implementation. Also defines the salt used by the hash table in your hash table implementation.

speck.h:
Defines the interface for the hash function using the SPECK cipher. Do not modify this.

speck.c:
Contains the implementation of the hash function using the SPECK cipher. Do not modify this.

ht.h:
Defines the interface for the hash table ADT. Do not modify this.

ht.c:
Contains the implementation of the hash table ADT.

bst.h:
Defines the interface for the binary search tree ADT. Do not modify this.

bst.c:
Contains the implementation of the binary search tree ADT.

node.h:
Defines the interface for the node ADT. Do not modify this.

node.c:
Contains the implementation of the node ADT.

bf.h:
Defines the interface for the Bloom filter ADT. Do not modify this.

bf.c:
Contains the implementation of the Bloom filter ADT.

bv.h:
Defines the interface for the bit vector ADT. Do not modify this.

bv.c:
Contains the implementation of the bit vector ADT.

parser.h:
Defines the interface for the regex parsing module. Do not modify this.

parser.c:
Contains the implementation of the regex parsing module.

Makefile:
This is a file that will allow the grader to type make to compile your program.
• CC = clang must be specified.
• CFLAGS = -Wall -Wextra -Werror -Wpedantic must be included.
• make should build the banhammer executable, as should make all and make banhammer.
• make clean must remove all files that are compiler generated.
• make format should format all your source code, including the header files.
Your code must pass scan-build cleanly. If there are any bugs or errors that are false positives, document them and explain why they are false positives in your README.md.

README.md:
This must be in Markdown. This must describe how to use your program and Makefile. This includes listing and explaining the command-line options that your program accepts. Any false positives reported by scan-build should go here as well.

DESIGN.pdf:
This must be a PDF. The design document should describe your design for your program with enough detail that a sufficiently knowledgeable programmer would be able to replicate your implementation. This does not mean copying your entire program in verbatim. You should

instead describe how your program works with supporting pseudocode. For this program, pay extra attention to how you build each necessary component.

WRITEUP.pdf:
This document must be a PDF. The writeup must include at least the following:
• Graphs comparing the total number of lookups and average binary search tree branches traversed as you vary the hash table and Bloom filter size.
– How do the heights of the binary search trees change?
– What are some factors that can change the height of a binary search tree?
– How does changing the Bloom filter size affect the number of lookups performed in the hash table?
• Analysis of the graphs you produce.

Pseudocode:

bf.c:
Initialize count;
Struct BloomFilter contains:
    primary[2] of type uint64_t
    secondary[2] of type uint64_t
    tertiary[2] of type uint64_t
    filter of type BitVector;

BloomFilter *bf_create(uint32_t size):
    Initialize BitVector using size and bv_create()
    Initialize primary, secondary, and tertiary with salt.h's defines
    Set count to 0

Void bf_delete(BloomFilter **bf):
    Free *bf
    bv_delete(*bf->filter)
    Set *bf to NULL

Uint32_t bf_size(BloomFiler *bf):
    Return bv_length(bf->filter)

Void bf_insert(BloomFilter *bf, char *oldspeak):
    Set bit at hash(primary, oldspeak) to 1
    Set bit at hash(secondary, oldspeak) to 1
    Set bit at hash(tertiary, oldspeak) to 1
    Add 1 to count

Bool bf_probe(BloomFilter *bf, char *oldspeak):
    If bit at hash(primary, oldspeak), hash(secondary, oldspeak), and hash(tertiary, oldspeak) is 1 return true else return false

Uint32_t bf_count(BloomFilter *bf):
    Return count

Void bf_print(BloomFilter *bf):
    Print out the bloom filter

bv.c:
Struct BitVector contains:
    length of type uint32_t
    vector of type uint8_t

BitVector *bv_create(uint32_t length):
    initialize bv
    Set bv->length to length
    If not enough memory:
        Return NULL
    Else:
        Initialize vector with all 0's
        Return bv

Void bv_delete(BitVector **bv):
    free(*bv->vector)
    free(*bv)
    Set *bv to NULL

Uint32_t bv_length(BitVector *bv):
    Return bv->length

Bool bv_set_bit(BitVector *bv, uint32_t i):
    If i is greater than length or less than 0:
        Return false
    Else:
        Set bv->vector[i] to 1;
        Return true

Bool bv_clr_bit(BitVector *bv, uint32_t i):
    If i is greater than length or less than 0:
        Return false
    Else:
        Set bv->vector[i] to 0;
        Return true

Bool bv_get_bit(BitVector *bv, uint32_t i):

If i is greater than length or less than 0:
Return false
Return bv->vector[i]

Void bv_print(BitVector *bv):
Print out the bit vector

Ht.c:
Initilize count
Struct HashTable contains:
Salt[2] of type uint64_t
Size of type uint32_t
**trees of type Node

HashTable *ht_create(uint32_t size):
Initialize ht
Set ht->size to size
Set ht->salt to the defines in salt.h
Set count to 0

Void ht_delete(HashTable **ht):
Free(*ht)
Set *ht to NULL

Uint32_t ht_size(HashTable *ht):
Return ht->size

Node *ht_lookup(HashTable *ht, char *oldspeak):
Search for node in bst
If found:
Return node pointer
Else:
Return NULL

Void ht_insert(HashTable *ht, char *oldspeak, char *newspeak):
Add “oldspeak:newspeak” to the ht->tree
Add 1 to count

Uint32_t ht_count(hashTable *ht):
Return count

Double ht_avg_bst_size(HashTable *ht):
Return bst_size()/ht_count()

Double ht_avg_bst_height(HashTable *ht):
    Return bst_height()/ht_count()

Void ht_print(HashTable *ht):
    Print out the HashTable ht

node.c:
Struct Node contains:
    oldspeak of type char
    newspeak of type char
    left of type Node
    right of type Node

Node *node_create(char *oldspeak, char *newspeak):
    Initialize n
    Copy oldspeak and put it in n->oldspeak
    Copy newspeak and put it in n->newspeak

Void node_delete(Node **n):
    free (*n)
    free(*n->oldspeak)
    free(*n->newspeak)
    Set *n to NULL

Void node_print(Node *n):
    If n contains both oldspeak and newspeak print out “oldspeak -> newspeak”
    If n contains only oldspeak print out “oldspeak”