Deeraj Gurram

dgurram@ucsc.edu

5 December 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, command line input handling, and the actual code execution to run the program.
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. These functions consist of speck_expand_key_and_encrypt, keyed_hash, and hash.
6. ht.h: Defines the interface for the hash table ADT.
7. ht.c: Contains the functions that create and manipulate the HashTable ADT, which include ht_create, ht_delete, ht_size, ht_lookup, ht_insert, ht_count, ht_avg_bst_size, ht_avg_bst_height, and ht_print.
    a. ht_create allocates memory to a HashTable and initializes the struct members
    b. ht_delete frees the allocated memory of the struct members and the HashTable
    c. ht_size returns the size of the HashTable
    d. ht_lookup searches the HashTable for a node that matches the argument oldspeak
    e. ht_insert Inserts a new node into the correct position of the binary search tree and the HashTable
    f. ht_count returns the number of non-NULL bst's in the HashTable
    g. ht_avg_bst_size returns the average bst size across the entire HashTable
    h. ht_avg_bst_height returns the average bst height across the entire HashTable

    i. ht_print prints the values in the HashTable

8. bst.h: Defines the interface for the binary search tree ADT.
9. bst.c: Contains the functions that create and manipulate the binary search tree ADT. These functions include bst_create, bst_height, bst_size, bst_find, bst_insert, bst_print, and bst_delete.
    a. bst_create creates a NULL bst
    b. bst_height returns the height of the bst
    c. bst_size returns the size of the bst
    d. bst_find searches the bst for a node containing argument oldspeak
    e. bst_insert inserts a new node into the bst at the correct position
    f. bst_print uses inorder traversal to print the values of every node in the bst
    g. bst_delete uses postorder traversal to delete every node in the bst
10. node.h: Defines the interface for the node ADT.
11. node.c: Contains the functions that create and manipulate the node ADT. These functions consist of node_create, node_delete, and node_print.
    a. node_create allocates memory to a Node and initializes the struct members
    b. node_delete frees the allocated memory of the struct members and the Node
    c. node_print prints the values of the struct members
12. bf.h: Defines the interface for the Bloom filter ADT.
13. bf.c: Contains the functions that create and manipulate the Bloom filter ADT. These functions consist of bf_create, bf_delete, bf_size, bf_insert, bf_probe, bf_count, and bf_print.
    a. bf_create allocates memory to a BloomFilter and initializes the struct members
    b. bf_delete frees the allocated memory of the struct members and the BloomFilter
    c. bf_size returns the length of the struct member filter
    d. bf_insert inserts the argument oldspeak into the BloomFilter

    e. bf_probe probes the BloomFilter the the argument oldspeak and returns true if found
    f. bf_count returns the number of set bits in the BloomFiter
    g. bf_print prints the values of the struct members for the BloomFilter
14. bv.h: Defines the interface for the bit vector ADT.
15. bv.c: Contains the functions that create and manipulate the bit vector ADT. These functions consist of bv_create, bv_delete, bv_length, bv_set_bit, bv_clr_bit, bv_get_bit, and bv_print.
    a. bv_create allocates memory to a BitVector and initializes the struct members
    b. bv_delete frees the allocated memory of the struct members and the BitVector
    c. bv_length returns the BitVector's length
    d. bv_set_bit Sets a bit in the vector to 1
    e. bv_clr_bit clears a bit in the vector to 0
    f. bv_get_bit gets the value of the bit at vector position i
    g. bv_print prints the length of the BitVector and the value of every bit
16. parser.h: Defines the interface for the regex parsing module.
17. parser.c: Contains the implementation of the regex parsing module. The functions consist of next_word and clear_words.
    a. next_word returns the next word that matches the specified regex
    b. clear_words clears out the static word buffer
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.

Pseudocode:

bf.c

Struct BloomFilter

Define the struct members

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

Set bf->filter equal to the function bv_create(argument size)

Return the created BloomFilter

void bf_delete(BloomFilter **bf)

Check if bf exists

Check if the BitVector filter exists

Call bv_delete on filter

Free the allocated memory for the BloomFilter

Set the pointer to bf equal to NULL

uint32_t bf_size(BloomFilter *bf)

Return the bv_length of *filter

void bf_insert(BloomFilter *bf, char *oldspeak)

Call bv_set_bit on the struct member filter, and hash the oldspeak to the primary salt

Call bv_set_bit on the struct member filter, and hash the oldspeak to the secondary salt

Call bv_set_bit on the struct member filter, and hash the oldspeak to the tertiary salt

bool bf_probe(BloomFilter *bf, char *oldspeak)

Return true if the bit value of bv_get_bit(filter, oldspeak hashed to the primary salt) is true and the bit value of bv_get_bit(filter, oldspeak hashed to the secondary salt) is true and the bit value of bv_get_bit(filter, oldspeak hashed to the tertiary salt) is true

uint32_t bf_count(BloomFilter *bf)

Initialize a counter integer

Create a for loop where i starts at 0 and ends when it reaches the length of the struct member filter

Increment count by 1 if the bit value of bv_get_bit(filter, i) is 1

Return the count

void bf_print(BloomFIlter *bf)

Print the size of the BloomFIlter

Loop through the length of the filter and print the bit value of the current iteration

bv.c

Struct BitVector

Define the BitVector struct members

BitVector *bv_create(uint32_t length)

Create a BitVector bv and allocate it memory of size BitVector

Check if bv exists

Set variable bytes equal to the length divided by 8 + the length modded by 8

Set the struct member vector equal to the allocated memory of size uint8_t

Set the struct member length equal to argument length

Return the created BitVector

Else

Return (BitVector *) 0

void bv_delete(BitVector **bv)

Check if the struct member vector exists

Free the allocated memory of the vector

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 i is less than 0 or greater than bv's length

Return false

Else check if bv and bv's vector exist

Set the i'th bit in bv to 1 using bit shifting

Return true

bool bv_clr_bit(BitVector *bv, uint32_t i)

Check if i is less than 0 or greater than bv's length

Return false

Else check if bv and bv's vector exist

Set the i'th bit in bv to 0 using bit shifting

Return true

bool bv_get_bit(BitVector *bv, uint32_t i)

Check if i is less than 0 or greater than bv's length or the value of the i'th bit in the vector is 0

Return false

Else check if the value of the i'th bit in the vector is 1

Return true

Return true

void bv_print(BitVector *bv)

Print the BitVector's length

Create a for loop to go through the bits in bv, the iterator should be less than the length of bv * 8

Print out the value of the i'th bit of the vector

ht.c

Struct HashTable

Define the HashTable struct members

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

Allocate the struct member trees memory of size Node

Return the created HashTable

void ht_delete(HashTable **ht);

Check if ht exists

Loop through the size of ht

Check if the current value of the struct member trees isn't NULL

Call bst_delete on the current value of the struct member trees

Check if ht's trees exists

Free the allocated memory of the struct member trees

Free the allocated memory of ht

Set *ht equal to NULL

uint32_t ht_size(HashTable *ht);

Return the size of ht

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

Create a 32 bit integer index and set it equal to the oldspeak hashed with the HashTable's salt

Increment global variable lookups by 1

Return the function bst_find(the index value of struct member trees, oldspeak)

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

Create a 32 bit integer index and set it equal to the oldspeak hashed with the HashTable's salt

Increment global variable lookups by 1

Set the index value of the struct member trees to function bst_insert(the index value of trees, oldspeak, newspeak)

uint32_t ht_count(HashTable *ht);

Initialize a integer counter to 0

Create a for loop where the iterator is less than ht's size

Check if the current value of ht's trees isn't NULL

Increment the counter by 1

Return the counter

double ht_avg_bst_size(HashTable *ht);

Initialize a double called avg_size to 0

Create a for loop where the iterator is less than ht's size

Increment avg_size by the function bst_size of the current value of the struct member trees

Return avg_size / ht_count(ht)

double ht_avg_bst_height(HashTable *ht);

Initialize a double called avg_height to 0

Create a for loop where the iterator is less than ht's size

Increment avg_height by the function bst_height of the current value of the struct member trees

Return avg_height / ht_count(ht)

void ht_print(HashTable *ht);

Print the size of the HashTable

node.c

struct Node

Define the Node struct members

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

Create a Node n and allocate memory of size Node to it

Check if n exists

Set the struct member left to NULL

Set the struct member right to NULL

Check if argument oldspeak doesn't equal NULL

Set struct member oldspeak equal to strdup(argument oldspeak)

Else

Set struct member oldspeak equal to NULL

Check if argument newspeak doesn't equal NULL

Set struct member newspeak equal to strdup(argument newspeak)

Else

Set struct member newspeak equal to NULL

Return the created Node

void node_delete(Node **n);

Check if Node n exists

Check if struct member oldspeak doesn't equal NULL

Free the allocated memory of oldspeak

Check if struct member newspeak doesn't equal NULL

Free the allocated memory of newspeak

Free the allocated memory of n

Set the pointer to n equal to NULL

void node_print(Node *n);

Check if struct member oldspeak doesn't equal NULL and struct member newspeak doesn't equal NULL

Print the struct member oldspeak -> struct member newspeak

Check if struct member oldspeak doesn't equal NULL and struct member newspeak equals NULL

Print out the struct member oldspeak

bst.c

Define the 64 bit global integer branches

Node *bst_create(void)

Return NULL

void bst_height(Node **root)

Check if root exists

Return 1 + the max between bst_height(the left child) or the bst_height(the right child)

Return 0

uint32_t bst_size(Node *root);

Check if the root equals NULL

Return 0

Return the bst_size(the left child) + bst_size(the right child) + 1

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

Check if root exists

Check if the strcmp(root's oldspeak, argument oldspeak) is greater than 0

Return bst_find(the left root, argument oldspeak)

Else check if the strcmp(root's oldspeak, argument oldspeak) is less than 0

Return bst_find(the right root, argument oldspeak)

Return root

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

Check if root exists

Check if the strcmp(root's oldspeak, argument oldspeak) is greater than 0

Increment global variable branches by 1

Return bst_insert(the left root, argument oldspeak)

Else check if the strcmp(root's oldspeak, argument oldspeak) is less than 0

Increment global variable branches by 1

Return bst_insert(the right root, argument oldspeak)

Return root

Return node_create(oldspeak, newspeak)

void bst_print(Node *root);

Perform inorder traversal of the bst

Call bst_print(root's left child)

Call node_print(root)

Call bst_print(root's right child)

void bst_delete(Node **root)

Check if root exists

Call bst_delete(root's left child)

Call bst_delete(root's right child)

Call node_delete(root)

Credit:

1. Used some of the deliverables descriptions from Dr. Long on the asgn7.pdf file.
2. Got bst_find, bst_insert, bst_height, bst_print, and max from Dr. Long's Lecture 18 slides
3. Followed the inorder traversal example from Lecture 18 page 22 for bst_print
4. Got bst_create and bst_delete from tutor Eric Hernandez
5. Got node_print from the asgn7 pdf by Dr. Long
6. Got the idea to do the if/else statement for n->oldspeak from tutor Eric Hernandez in node_create
7. Used the bitwise operators and shifts code for bv_set_bit, bv_clr_bit, and bv_get_bit from the example provided by Dr. Long in the CSE 13S Code Comments bv8.h file
8. Also got bv_create and bv_delete from Dr. Long's bv8.h file
9. Used tutor Eric Hernandez's pseudocode for all of ht.c
10. Got everything in bf.c except bf_create from tutor Eric Hernandez
11. Used the while (word[i]) from https://www.educative.io/edpresso/what-is-the-tolower-function-in-c for the while loop in banhammer.c
12. Got the bulk of the banhammer.c code from tutor Eric Hernandez
13. Used Dr. Long's banhammer.c steps/description from the asgn7.pdf
14. Got the idea to use left shifting from miles on Discord to set the default values for the command line options
15. Used the parsing module provided in the asgn7.pdf by Dr. Long in banhammer.c
16. Got the regex pattern from tutor Eric Hernandez for banhammer.c