Ruchit Patel

CSE13S, Spring 2021

Prof. Darrell Long

21 May 2021

Assignment 7: The Great Firewall of Santa Cruz

Brief Description

- In this assignment, a data filtering scheme is implemented.
- The data is parsed using regular expressions (regex), added using hash tables (collisions are handled using linked lists), and filtered using Bloom filter.
- Arguments:

    -h (prints the help message),

    -t (specifies the size of the hash table),

    -f (specifies the size of the bloom filter entries),

    -s (the program only prints the statistics),

    -m (specifies move-to-front rule to be used).

Descriptions/Implementation

Argument Parsing

- Arguments are parsed using getopt GNU utility.
- If a valid argument is encountered, an integer is added to an abstract data type BitVector.
- For example, if -a is encountered, inside the case for -a, an integer value in an enum is added to the BitVector.
- Later, when checking the arguments, the BitVector functions are used to check if the bit is set at the value in the enum (argument was parsed).

Pseudocode:

define flags

while (getopt returns valid)

switch to the argument case if valid

add the argument to the BitVector

if invalid print usage message and return

check the BitVector at specific indexes to see if the argument was parsed and take action accordingly

Abstract Data Structures (ADT)

[Note: the words string and character literal are used interchangeably and mean the same thing]

BloomFilter (BF)

- The BloomFilter ADT acts as a BitVector, however, it uses hash functions to set the bits in the BitVector.
- In this lab, three hash functions are used and thus three bits are set if any element is added. Thus, if those three bits are set then a given element is considered to be in the BF (false positives are possible).

- Structure/Members of a BloomFilter
  a. Primary salt : An array of 128 bits (for hashing).
  b. Secondary salt : An array of 128 bits (for hashing).
  c. Tertiary salt : An array of 128 bits (for hashing).
  d. Filter : A BitVector (BV).

- Methods associated with a BloomFilter (Pseudocode)
  a. BF pointer bf_create(integer size) [credits: based on the lab doc code]

     allocate memory for BF structure

     set the primary, secondary, and tertiary salts according to the provided values

     create a BitVector of size *size* using the bv_create function

     if mem for BV cannot be allocated then

     free the mem for the BF and set the BF pointer to null

     return the BF pointer

  b. bf_delete (pointer to BF pointer)

if BF pointer is not empty/null

free the BV mem using bv_delete

free the memory for BF structure

set BF pointer to null

c. integer bf_size (BF pointer)

if BF pointer is valid

return the size of underlying BV using bv_size function, else return zero

d. bf_insert (BF pointer, string oldspeak)

get and index by hashing the *string oldspeak* with the first salt of BF

set the bit in the underlying BV at that index using bv_set_bit

repeat the above two steps with second and third salts

return the pointer to the newly created node

e. boolean bf_probe (BF pointer, string oldspeak)

get three indexes by hashing the *string oldspeak* with first, second and third salts respectively

check if the bits in the underlying BV are set using bv_get_bit

return true if all three bits are set, else false

f. integer bf_count (BF pointer)

instead of loop through the underlying at the end, I keep a static variable that is updated every time a bit was not set (was zero) and is now going to be set (updated in bf_insert)

return the variable that has been tracking number of one bits

g. bf_print (BF pointer)

print the underlying BV using bv_print function

BitVector (BV)

- The BitVector data structure is an array of integers. However, each of those integer's bit has significance.

- Instead of using a whole integer to represent true (1) or false (0), each bit of the integer array is used to be space efficient.

- Structure/Members of a BitVector
    a. Length of vector array : Total length of the vector in bits.
    b. Array of bytes : Array that holds bytes.

- Methods associated with a BitVector (Pseudocode)
    a. BV pointer bv_create (length of vector)

    assign memory for BV structure
    set the length element to *length*
    assign memory for the array equal to size of integer * $\lfloor length/8 \rfloor + 1$(minimum elements for *length* bits) [credits: from the lab documentation]
    if cannot be assigned then return
    else initialize each integer to zero (either memset or loop)
    return pointer to newly created BV

    b. bv_delete (pointer to BV pointer)

    if BV is not empty/null
        free the memory for BV array
        free the memory for BV structure
        set BV pointer to null

    c. integer bv_length (BV pointer)

    return the *length* element of BV struct pointed by the pointer

    d. bv_set_bit (BV pointer, integer index) [credits: From the lab5 documentation]

    locate the byte for *index* (*index* / 8)
    locate the bit inside that byte (*index* % 8)
    set the bit using bitwise operation (byte ORed with 1 left shifted by bit location)

e. bv_clr_bit (BV pointer, integer index)

locate the byte for *index* (*index* / 8)

locate the bit inside that byte (*index* % 8)

clear the bit using bitwise operation (byte ANDed with NOT of 1 left shifted by bit location)

f. integer bv_get_bit (BV pointer, integer index)

locate the byte for *index* (*index* / 8)

locate the bit inside that byte (*index* % 8)

return the bit using bitwise operation (byte ANDed with 1 left shifted by bit location)

g. bv_print (BV pointer)

loop over the vector array inside the bit vector *length* times

each time call bv_get_bit and print accordingly

Hash Table (HT)

- The Hash Table ADT is used to store and lookup the strings that need to be filtered out.
- Hash collisions are resolved using a linked list (described later).

- Structure/Members of a Hash Table

a. Salt : An array of 128 bits.

b. Size : Size of the hash table.

c. mtf : Boolean for move-to-front.

d. Lists : An array of linked list pointers.

- Methods associated with a Hash Table (Pseudocode)

a. HT pointer ht_create(integer size, boolean mtf) [credits: from the lab doc implementation]

allocate memory for Hash Table structure

set the values in the salt array according to the provided values

set the size to *size*, mtf to *mtf*

allocate memory for the array of LinkedList pointers

if mem for array cannot be allocated then

free the mem for the HT and set the HT pointer to null

return the HT pointer

b. ht_delete (pointer to HT pointer)

if HT pointer is not empty/null

free the LinkedList mem using ll_delete

free the memory for HT structure

set HT pointer to null

c. Node pointer ht_lookup (HT pointer, string oldspeak)

find an index into the HashTable by hashing *oldspeak*

walk the LinkedList at that index using ll_lookup function

if a node is found containing the *string oldspeak*

return the pointer to that node, else return null

d. ht_insert (HT pointer, string oldspeak, string newspeak)

find an index into the HT by hashing *string oldspeak*

create a linked list (using ll_create) if there is none at the found index

insert using ll_insert with the parameters *string oldspeak, string newspeak,* and *pointer to linked list* (either looked up or the newly created one)

e. integer ht_count (HT pointer)

instead of loop through the underlying at the end, I keep a static variable that is updated every time HT entry was null and is now a LL is going to be added (updated in ht_insert)

return the variable that has been tracking number of linked lists

f. ht_print (HT pointer)

loop through each index of the LinkedList array

if the pointer is not null

print the linked list using ll_print function where argument would be current index

entry

Node

- The Node abstract data structure (ADT) is used to represent each entry in the LinkedList data structure.

- Structure/Members of a Node
    a. Oldspeak : A string literal.
    b. Newspeak : A string literal.
    c. Next pointer : Points to the next node.
    d. Prev pointer : Points to the previous node

- Methods associated with a Node (Pseudocode)
    a. Node pointer node_create(string oldspeak, string newspeak)

    allocate memory for Node structure
    if *string oldspeak* is not null
        allocate memory for *string oldspeak* of size length of *oldspeak* + 1(for \0)
        copy *string oldspeak* to oldspeak with help of strndup and strlen functions
    do the same above condition steps with *string newspeak*
    set the *next* and *prev* pointers to null
    return the pointer to the newly created Node

    b. node_delete (pointer to Node pointer)

    if Node pointer is not empty/null
        if *string oldspeak* of Node is not null then
            free the memory for the string and set the pointer to null
        do the same above condition steps with *string newspeak*
        free the memory for the Node and set the pointer to null

    c. node_print (Node pointer)

    print “oldspeak->newspeak” if newspeak not null
    else print “oldspeak”

LinkedList (LL)

- The LinkedList is made up of Node ADTs and is used to store the Nodes whose *oldspeak* string results in a hash collision.
- The nodes are inserted at the head of the linked list. Also, any looked up value is moved to the front of the list depending on the *mtf* value (moved if true, else not).

- Structure/Members of a LinkedList
    a. Length : Length of the linked list.
    b. Head : Points to the first node in LL.
    c. Tail : Points to the last node in LL.
    d. mtf : Boolean for move-to-front.

- Methods associated with a Node (Pseudocode)
    a. LL pointer ll_create(boolean mtf)

    allocate memory for LinkedList structure
    create the *head* sentinel node using node_create
        set the *oldspeak, newspeak,* and *prev* elements of *head* to null
    create the *tail* sentinel node using node_create
        set the *oldspeak, newspeak,* and *next* elements of *tail* to null
        set the *next* element of *head* to *tail*
    set the *head* to head, *tail* to tail, *mtf* to mtf, and *length* to 0
    return the pointer to newly created LL

    b. ll_delete (pointer to LL pointer)

    if pointer is not empty/null
        start at the end of the head of the LL
        move to the next node
        delete the previous node using node_delete
        do the above steps until current pointer is null
        delete the tail node using node_delete
        free the linked list and set the pointer to null

c. integer ll_length (LL pointer)

make a temporary integer to store the size of the LL

make a temporary Node pointer whose value is the same as *head*

start at the temporary Node and loop until it is equal to the *tail*

each time make the temp Node point to its *next* value

increment the temporary value of size variable

return the temporary size value

d. Node pointer ll_lookup (LL pointer, string oldspeak)

make a temporary node (temp) and assign it *head*'s value

loop over the list until the node is found *or* until the size of the LL

assign temp the value of *next* element (given by the Node)

compare the *oldspeak* of Node temp with the oldspeak argument (done with strncmp)

if *mtf* is true inside the LL, move the node to the front

assign *next* of *prev* of temp the value of temp's *next*

assign *prev* of *next* of temp the value of temp's *prev*

assign *next* of temp the value of node after *head*

assign *prev* of temp the value of *head*

assign *prev* of node after *head* the value of temp

assign *head's next* the value of temp

return the Node pointer to the temp

e. ll_insert (LL pointer, string oldspeak, string newspeak)

lookup the *oldspeak* argument using ll_lookup

if the node already exists, do nothing and return

else make a new node temp (using node_create) with the given arguments

add the newly made node to the LL

assign *next* of temp the value of node after *head*

assign *prev* of temp the value of *head*

assign *prev* of node after *head* the value of temp

assign *head's next* the value of temp

f. ll_print (LL pointer)

start at the node after the *head* of the LL

loop over *size* of the LL and each time print the node using node_print

Algorithms

Hashing (Credits: given in the lab folder)

- The hashing function is made using the SPECK algorithm.

Regular Expressions

- All the work related to regular expression makes use of the regex.h library.
- A valid word can contain lowercase or uppercase letters, digits, underscores, apostrophes, and hyphens.
- Thus, the regular expression is compiled using regcomp function and the pattern is "[a-zA-Z0-9_]+([-']?[a-zA-Z0-9_])*"

- Explanation:
    - [a-zA-Z0-9_] : word can start from this set
    - ([-']?[a-zA-Z0-9_])* : character group that must follow the valid beginning set
    - [-']? : the set can be followed by either at most one continuous - or '
    - [a-zA-Z0-9_] : which indeed must be followed by the valid set again (to avoid -- or '')
    - (group)* : do this zero or more times (to include single letters and multiple-hyphens'or'apostrophes-too)

- A regex parser (provided in the lab folder) is used to parse the words from a file stream according to the compiled regex.
- The function next_word returns the pointer to the next word matching the valid/compiled regex pattern. The function clear_word deallocates the memory used by next_word function.

- The valid words are then further used with the hash functions to set bits in the BloomFilter and are added to the HashTable.

High Level Program Flow (Credits: based on the lab document description)

initialize the BloomFilter and Hash Table
read in the badspeak words from a file using fscanf
for each returned word
    add the word to BF using bf_insert function
read in the badspeak and newspeak words from a file using fscanf
for each returned word
    add the first word to the BF using bf_insert function
    add the first and the next word to the HT using the ht_insert function
compile the regex using regcomp
using next_word, read in the words from stdin stream
make boolean variables to track whether the person has been accused of thoughtcrime or rightspeak
if statistics argument is not passed make buffers/LL that would be used to store node pointers to be displayed later [string buffers were too hacky]
for each word that is read in
    check if it has been added using bf_probe function and if it has been added
        check whether the word is in the HT using ht_lookup and if it is in the HT
            check if the word/node does not have a *newspeak string* and if it doesn't
                insert the *oldspeak* in the list of badspeak buffer/LL used by the person
    check if the word/node does have a *newspeak string* and if it does
                insert the *oldspeak* in the list of oldspeak words/LL used by the person
    if the word is not in the HT then do nothing
if the person is accused of thoughtcrime and rightspeak
    print the mixspeak message followed by badspeak words followed by oldspeak and the corresponding righspeak ones (ll_print)
else if the person is accused of thoughtcrime and not rightspeak
    print the thoughtcrime message followed by badspeak words (from the LL)
else the person is only guilty of rightspeak and thus
    print the goodspeak message followed by oldspeak and corresponding righspeak words (from the LL)
free the memory