The Great Firewall of Santa Cruz

Design Document

In this lab, The Great Firewall of Santa Cruz, I create a firewall that filters words known as badspeak words and oldspeak words. The badspeak words do not have a newspeak translation while the oldspeak words do. These words are specified in the badspeak.txt and newspeak.txt file. If a user inputs any of the words in these files, they will be notified of the type of the word they have used.

To achieve these results, I create a bloom filter which sets three bits, that were given from hashing the word with three salts, for each word. I also create a hash table which inserts words into a linked list at the index that is specified from hashing the word with a salt. The program goes through this process for every word in the badspeak.txt and newspeak.txt files until every word is in the hash table and its bits are set in the bloom filter.

To check if a word inputted by the user is a badspeak or oldspeak word, first, I check if the 3 bits for a word are set in the bloom filter. If they are set, then the word is most likely in the hash table. Therefore, I check the hash table. If the word is in the hash table, then that word is either an oldspeak or badspeak. If it is not in the hash table, then the bloom filter gave a false positive.

Pre Lab Part 1

1. Bloom Filter
    a. Inserting elements
        i. Hashing a word with the three salts will give three bit locations. The way to insert the word into the bloom filter is to set those three bit locations in the bit vector.

    hash(primary, “oldspeak”) = bit location 1

    hash(secondary, “oldspeak”) = bit location 5

    hash(tertiary, “oldspeak”) = bit location 2

    For example, to set the bits:

        Mask = 1 << 1

        Result = Bit vector | mask

Mask = 1 << 5

Result = Bit vector | mask

Mask = 1 << 2

Result = Bit vector | mask

b. Deleting elements
    i. To delete an element from the bloom filter, you would clear the bits that were set for that element and that don't overlap with other elements. In other words, one has to take into account that some of the bits are set for more than one element before deleting the bits that were set for the specified element.

Pre Lab Part 2

1. LL Very high level pseudocode (Refer to Ll header)

Pre Lab Part 3

1. Current understanding of RegEx: We're supposed to write an expression that allows us to identify if a string is a word or not.
   ^[A-zA-Z]$

Node

This file includes the Node ADT

node_create(oldspeak, newspeak)

Allocate memory for the node structure

If oldspeak and newspeak borth are not NULL (oldspeak)

Allocate memory for the length of newspeak + null char

Allocate memory for the length of oldspeak + null char

Copy the words into this newly allocated memory

If oldspeak is not NULL but newspeak is NULL (badspeak)

Copy oldspeak into new memory location

Allocate memory for length of oldspeak + null char

Copy the word into this newly allocated memory

Pointer to prev should be NULL

Pointer to next should be NULL

node_delete(n)

Free oldspeak

Free newspeak if is not NULL

Fee pointer n

N = NULL

node_print(n)

If the node includes both oldspeak and newspeak

Print (oldspeak -> newspeak)

If the node only include olspeak

Prints(oldspeak)

Ll

This file includes the Linked List ADT

ll_create(mtf)

Allocate memory for the linked list structure

If that was successful

Length = 0 because the sentinel nodes don't count in the length of the list

Head = node_create(NULL, NULL)

Tail = node_create(NULL, NULL)

Pointer to Mtf = mtf

Head and tail originally point at each other

head 's next = tail

Tail's prev = head

Head's prev = NULL

Tail's next = NULL

ll_delete(ll)

Loop through linked

Call free_node() to free each node in the list

Then free the linked list

ll_length(ll)

Loop through the linked starting at head's next and ending at tail's prev

length +=1

ll_lookup(ll, oldspeak)

Loop through the linked list

If a node n with the oldspeak exists

If mtf == true

N's prev's next = n's next

N's next's prev = n's prev

N's next = ll->head->next

Head's next's prev = n

Head's next= n

Return the node

Else

Return NULL

ll_insert(ll, oldspeak, newspeak)

If ll_lookup(ll, oldspeak) returns NULL that means oldspeak hasn't been isnerted yet and should be inserted

insert the node n at the head of the linked list

N's next = head's next

N's prev = head

Head's next's prev = n

Head's next = n

length += 1

ll_print(ll)

Loop through the linked starting at head's next and ending at tail's prev

Call node_print(n) to print out the elements in the node

Hash

This file includes the Hash Table ADT

ht_create(uize, mtf)

Allocate memory for the hash table structure

Salt required for hashing

Salt[0] = 0x85ae998311115ae3

Salt[1] = 0xb6fac2ae33a40089

Pointer to mtf = mtf

Allocate memory for the linked lists

ht_delete(ht)

Loop through the indices of the hash table

If there is a linked at the current index

Free the linked list

Free the hash table

Ht = NULL

ht_size(ht)

Return size of hash table

ht_lookup(ht, oldspeak)

hash(salt, oldspeak) = index

Index = Index % size

ll_lookup (list at index, oldspeak) search for oldspeak in the linked list at the index

ht_insert(ht, oldspeak, newspeak)

hash(salt, oldspeak) = index

Index = Index % size

if linked list has not been created at this index

    ht[index] = ll_create(mtf) create a linked list

    ll_insert(ht[index] ,oldspeak, newspeak) insert the contents

Else

    ll_insert(ht[index] ,oldspeak, newspeak) insert the contents

ht_print(ht)

Loop through hash table

    If there is a linked list at a hash table index

        ll_print(ht[index])

Bv

This file includes the Bit Vector ADT

Bytes_function (bits)

This function takes in the number of bits and returns the number of bytes(there are 8 bits in a byte)

If bits % 8 == 0

    Return bits/8

Else

    Return (bits/8)+1

bv_create(length)

Allocate memory for the bit vector structure

    If allocation was successful

        Allocate memory for the bit vector

            If allocation was successful

Pointer to length = length

bv_delete(bv)

Free the bit vector

bv_length(bv)

Return length of bit vector

bv_set_bit(bv, i)

Creates a mask where there is a 1 at the specified bit and “oring” the mask with the byte will set that bit to 1

Mask = 1 << i % 8

Vector = vector | mask

bv_clr_bit(bv, i)

Creates a mask where there is a 0 at the specified bit and “anding” the mask with the byte will turn the specified bit into 0

Mask = ~(1 << i % 8)

Vector = vector & mask

bv_get_bit(bv, i)

Creates a mask where there is a 1 at the specified bit and “anding” the mask with the byte will give us 1 if the bit is 1 and 0 if it is 0.

The result is then shifted back by index because the byte needs to be either 0 or 1

Mask = 1 << i % 8

Result = vector & mask

Result = result >> i % 8

Return result

bv_print(bv)

Loop through the but vector

print(0 + value of bit) which will be 0 or 1

Bf

This file includes the Bloom Filter ADT

bf_create(size)

Allocate memory for the bloom filter structure

3 salts required for hashing

primary[0] = 0x02d232593fbe42ff

primary[1] = 0x3775cfbf0794f152

secondary[0] = 0xc1706bc17ececc04

secondary[1] = 0xe9820aa4d2b8261a

tertiary[0] = 0xd37b01df0ae8f8d0;

tertiary[1] = 0x911d454886ca7cf7;

bv_crreate (size) will create the bit vector for the bloom filter

bf_delete(bf)

Free bloom filter

bv_delete(filter)

bf_length(bf)

Return the length of the filter

bv_length(filter)

bf_insert(bf, oldspeak)

Hash the oldspeak word with the three salts which will give 3 indices

first = (hash(primary, oldspeak)) % length of bf

second = (hash(econdary, oldspeak)) % length of bf

third = (hash(tertiary, oldspeak)) % length of bf

To insert the word into the filter the 3 bits should be set

bv_set_bit(filter, first)

bv_set_bit(filter, second)

bv_set_bit(ilter, third)

bf_probe(bf, oldspeak)

Get the 3 bits to be able to check of the word might've been added to the filter

first = (hash(primary, oldspeak)) % length of bf

second = (hash(econdary, oldspeak)) % length of bf

third = (hash(tertiary, oldspeak)) % length of bf

If bv_get_bit() returns 1 for all the 3 bits

    Return true because it means the word might be in the bf

Else

    Return false

bf_print(bf)

Print the bit vector which is the bloom filter

Banhammer

This file includes the implementation of The Great Firewall of Santa Cruz.

Mtf = false

Default ht_size = 10000

Default bf_size = 1048576

Parse through command line options while EOF hasn't been reached

    Option h: allows user to choose size of hash table

    Option f: allows user to choose size of bit vector

    Option m: allows user to enable the move-to-front technique

ht_create() to create hash table ht

bf_create() to create bloom filter bf

ll_create() to create a linked list to keep track of all the badspeak words used by user

ll_create() to create a linked list to keep track of all the oldspeak words used by user

Initialize an array badspeak_input[1024] to scan in all the words from the badspeak.txt file
Scan the badspeak.txt while EOF hasn't been reached
    bf_insert(bf, badspeak_input) will set the bits that resulted from hashing
    ht_insert(ht, badspeak_input, NULL) will insert the word into the hash table

Initialize an array oldspeak_input[1024] to scan in the oldspeak words from the newspeak.txt file
Initialize an array newspeak_input[1024] to scan in the newspeak words from the newspeak.txt file
Scan the newspeak.txt while EOF hasn't been reached
    bf_insert(bf, oldspeak_input) will set the bits that resulted from hashing
    ht_insert(ht, oldspeak_input, newspeak_input) will insert the oldspeak word into the hash table and its newspeak translation

Read in the word inputted by the user from stdin using regex which will match uppercase and lowercase letters, numbers, underscores, hyphens, and apostrophes.
Scan stdin while there are words to scan and not NULL
    Loop through each "word" to convert any uppercase letters to lowercase letters because badspeak.txt and newspeak.txt solely consists of lowercase

bf_probe(bf, word) to check if word is in the bf
if (bf_probe(bf, word)) if word is in the bloom filter
    ht_lookup(ht, word) one should check if it is in the bloom filter
else   otherwise word should be skipped
    Continue
If the word is both in the bloom filter and the hash table
    Check if it is an oldspeak or badspeak
    If it does not have a newspeak translation it is a badspeak word
        Ll_insert(used_badspeak,word,NULL) to add the word to the linked list with all the badspeak words spoken by user
    If it does have a newspeak translation it is an oldspeak word

Ll_insert(used_oldspeak, word, newspeak) to add the word to the linked list with all the oldspeak word spoken by user

If user spoke both badspeak and oldspeak words

Print both linked lists

If user spoke only badspeak

Print used_badspeak linked list

If user spoke only oldspeak words

Print used_oldspeak linked list