Assignment 7 - The Great Firewall of Santa Cruz

Teresa Joseph

CSE 13S - Professor Long

Fall 2021 - RD November 28/ FD December 5

Purpose

The purpose of this assignment is to implement a method of going through text and finding "oldspeak" terms (and their "newspeak" words if possible) and printing out a corresponding message with the terms used. It will make use of bloom filters, hash tables, and binary trees to do this, their functions being written out in bf.c, ht.c, and bst.c respectively. A file that implements nodes for the binary search tree will be in node.c and a file that creates the bit vector operations needed for the bloom filter will be in bv.c. The main test harness is banhammer.c, which will take on command line options and filter through any badspeak or oldspeak/newspeak to print corresponding messages.

Pseudocode

bv.c

```
    bv_create
        allocate bit vector pointer memory with malloc
        if bit vector is NULL after allocation:
            return NULL (means allocation failed)
        set bit vector length to given length
        allocate bit vector vector memory with malloc
        for i from 0 to length:
            set index i of vector array to 0
        return bit vector
    bv_delete
        free bit vector pointer
        set pointer to NULL
    bv_length
        return bit vector struct member length
    bv_set_bit
```

if i is greater than or equal to bit vector length:

return false

set byte index to i divided by 8

set bit index to i modulo 8

create a mask of 1 left shifted by the bit index

OR vector array at byte index with mask

return true to indicate success

bv_clr_bit

if i is greater than or equal to bit vector length:

return false

set byte index to i divided by 8

set bit index to i modulo 8

create a mask of 1 left shifted by the bit index

AND vector array at byte index with inverse of mask

return true to indicate success

bv_get_bit

if i is greater than or equal to bit vector length:

return false

set byte index to i divided by 8

set bit index to i modulo 8

create a mask of 1 left shifted by the bit index

if AND of vector array at byte index with mask is 1:

return true since bit is 1

otherwise:

return false since bit is 0

bv_print

for i from 0 to length:

if vector array at i is 1, print 1

if vector array at i is 0, print 0

node.c

node_create

allocate node memory with malloc
set left and right of node to NULL
call strdup() on given oldspeak
if given newspeak is null:
    set node's newspeak to NULL as well
else:
    call strdup() on newspeak and set it to node's newspeak
return node

node_delete
    free pointer to oldspeak and newspeak if both are not NULL
    free node and set its pointer to NULL

node_print
    if both oldspeak and newspeak are not NULL:
        print both in format shown in assignment document
    if oldspeak is not NULL and newspeak is NULL:
        print oldspeak in format shown in assignment document

bst.c

bst_create
    return NULL

bst_height
    if root is not NULL:
        recursively find the leftmost of the root
        recursively find the rightmost of the root
        find the greatest value of the two (may add helper function for this)
        add 1 (root height) to the found max value and return this
    else:
        return 0 (the root is NULL and there is no height)

bst_size
    set an integer to be the size counter = 0
    if root is not NULL:
        increment the counter by 1

```
        recursively call with the left of the root
        recursively call with the right of the root
    return counter
bst_find
    if root is not NULL:
        compare root's oldspeak to given oldspeak with strcmp()
            if 0 is returned, return the root (matches at current node)
            increment branches variable (for printing statistics) by 1
            if  > 0 returned, recursively call with left of root and oldspeak
            if < 0 returned, recursively call with right of root and oldspeak
        if this portion is reached, return NULL (oldspeak does not exist in bst)
bst_insert
    if root is NULL:
        create new node with given oldspeak and newspeak
        return this node (will be root of new bst)
    else:
        if root's oldspeak matches given oldspeak:
            return root (matches root of tree and no action is needed)
        compare root's oldspeak to given oldspeak with strcmp()
            increment branches variable (for printing statistics) by 1
            if  > 0 returned, recursively call with left of root and old/newspeak
            if < 0 returned, recursively call with right of root and old/newspeak
        return pointer current node
bst_print
    if the root is not NULL:
        recursively call with the left of the root
        call node_print() on the root
        recursively call with the right of the root
bst_delete
    if the root pointer is NULL:
        return nothing (have reached the end)
```

else:

recursively call with the address of the left of the root

recursively call with the address of the right of the root

call node_delete() on the root

set the root pointer to NULL

bf.c

bf_create

allocate enough memory (of given size) for bloom filter

if bloom filter pointer is not NULL:

set primary[] indices to the lower and upper values in salts.h

set secondary[] indices to the lower and upper values in salts.h

set tertiary[] indices to the lower and upper values in salts.h

call bv_create() to set bit vector filter

return bloom filter pointer

bf_delete

free bit vector memory with bv_delete()

free bloom filter pointer

set pointer to NULL

bf_size

return length of bit vector

bf_insert

hash given oldspeak with primary salt

hash given oldspeak with secondary salt

hash given oldspeak with tertiary salt

for each of the returned hash values v:

set bit vector at index v with bv_set_bit()

bf_probe

hash given oldspeak with primary salt

hash given oldspeak with secondary salt

hash given oldspeak with tertiary salt

for each of the returned hash values v:

if index v of bit vector == 1:

continue

if index v of bit vector == 0:

return false immediately

if this point is reached, return true

bf_count

create counter and set to 0

iterating through all indices of bloom filter:

if value at index is 1:

increment counter by 1

else, continue

return counter

bf_print

for i from 0 to end of bloom filter (to size):

if bloom filter at i is 1, print 1

if floom filter at i is 0, print 0

if i is a multiple of 16, continue printing on new line (simply for formatting purposes/to be read easily)

ht.c

ht_create

allocate enough memory (of given size) for hash table

if hash table pointer is not NULL:

set salts array indices to the lower and upper values in salts.h

set hash table size to given size

allocate enough memory for node pointer trees (of size Node * indices)

make all roots of each tree NULL to begin with

return hash table pointer

ht_delete

for each index of the hash table:

free corresponding tree with bst_delete if not already NULL

free hash table tree pointer

free hash table pointer

set pointer to NULL

ht_size

return hash table size (from struct)

ht_lookup

hash given oldspeak and save its return value

use return value as index of hash table

increment lookups variable (for printing statistics) by 1

call bst_find() on the bst at the index to see if oldspeak is there

if it is, return pointer

if it is not, return NULL

ht_insert

hash oldspeak and save its return value

use return value as index of hash table

increment lookups variable (for printing statistics) by 1

insert oldspeak and newspeak at tree index using bst_insert()

ht_count

create counter and set it = 0

iterating through all indices of the hash table:

check if bst exists at index (if bst_height() > 0)

if so, increment the counter

else, continue

return counter

ht_avg_bst_size

create bst counter

create ht counter and set it = ht_count()

iterating through all indices of the hash table:

call bst_size() and increment bst counter by it

return bst counter divided by ht counter

ht_avg_bst_height

create bst counter

create ht counter and set it = ht_count()

iterating through all indices of the hash table:

call bst_height() and increment bst counter by it

return bst counter divided by ht counter

ht_print

for all indices of hash table:

print binary search tree at index with call to bst_print()

banhammer.c

main():

initialize bloom filter and hash table with bf_create and ht_create

parse through command line options with getopt()

-h prints help message

-s prints statistics

-t size specifies hash table size (default: 2^16)

-f size specifies bloom filter size (default: 2^20)

open badspeak.txt file

print error and exit if unable to open

while true:

call fscanf() to read badspeak words

break if scanned value is less than or equal to 0

else:

add each word to bloom filter with bf_insert()

add each word and NULL newspeak to hash table with ht_insert()

close badspeak.txt file

open newspeak.txt file

print error and exit if unable to open

while true:

call fscanf() to read oldspeak and newspeak words

break if scanned value is less than or equal to 0

else:

add each oldspeak to bloom filter with bf_insert()

add both oldspeak and newspeak to hash table with ht_insert()

close newspeak.txt file

makes call to create_report(bf, ht, statistics flag) to print message

free memory:

call bf_delete on bloom filter pointer

call ht_delete on hash table pointer

create_report()

initialize regex variable

call regcomp() on variable, my WORD pattern, and REG_EXTENDED

if compilation failed, exit

open a file to write badspeak terms to

print error and exit if unable to open

open another file to write old/newspeak terms to

print error and exit if unable to open

initialize badspeak counter and rightspeak counter to 0

initialize a character to NULL (to store read words in)

while result of next_word() isn't NULL (meaning not end of read in text):

call bf_probe() on word

if result is false:

continue looping (meaning word is not in bloom filter)

else:

create a node and store result of ht_lookup() of the word

if node is NULL (meaning word is not in the hash table):

continue looping

[at this point, the word is definitely a forbidden word]

if the node's newspeak is NULL, node contains badspeak:

increment badspeak counter by 1

print the oldspeak to the badspeak file

else, node contains rightspeak:

increment rightspeak counter by 1

print both oldspeak and newspeak to rightspeak file

call clear_words() and regfree on the regex variable

close the badspeak file and rightspeak file

if the boolean flag for printing statistics was enabled:

print ht_avg_bst_size() result

print ht_avg_bst_height() result

use extern variables branches/lookups and print as branch traversal value

calculate hash table load by ht_count() / ht_size() and print result

calculate bloom filter load by bf_count() / bf_size() and print result

return here (will not print message)

if badspeak count is not 0 and rightspeak count is not 0:

print mixspeak message

print contents of badspeak file with print_file(file name)

print contents of rightspeak file with print_file(file name)

if badspeak count is not 0 but rightspeak count is 0:

print badspeak message

print contents of badspeak file with print_file(file name)

if badspeak count is 0 but rightspeak count is not 0:

print goodspeak message

print contents of rightspeak file with print_file(file name)

print_file()

call fopen() on file to read

print error and exit if unable to open

create buffer array of max length 1024

while true:

call fgets() on file with buffer

if fgets() returns NULL:

break because EOF reached

print read character to buffer

close file

Overall Description

Bit Vector Implementation

As with all of my previous assignments, the create function starts with allocating enough memory for a bit vector pointer using malloc() to the size of the structure BitVector. Though this will not happen, I had it return NULL if the allocation failed. I also set the vector length to the given length in the parameter and set each index of the vector array to 0.

bv_delete() simply frees the given pointer and sets it to NULL.

bv_length() returns the length, which comes from its structure.

Setting, clearing, and getting bits at specified indices required logical operations that I used in previous assignments, especially in assignment five. In all three functions, I had them return false if the given index was out of range right away. This would save time that it would have otherwise spent going through the other portions of each function. Next, I created a byte index that divided the given value by 8 and a bit index that mods the value by 8. This ensures that I access the correct bit of the byte when indexing the vector array. In each function, I also created a "mask" that was simply 1 left shifted by the bit index. This will be used for each logical operation. I used OR when setting the bit, AND with the mask's inverse when clearing the bit, and AND when getting the bit. When setting the bit, this ensures that the bit becomes 1 if originally 0 and remains 1 if originally 1. When clearing, it does the same thing but becomes/remains as 0. When getting the bit, ANDing will only return 1 if the bit is already 1, in which case I would return true. Else, the biit must be 0 and I would have to return false.

When printing the bit vector, I simply printed 1's and 0's accordingly.

Node Implementation

When creating a node, I again used malloc() to allocate enough memory. While the node pointer is not NULL, I set the left and right nodes to NULL, as described in the assignment document. I also called strdup() to copy the given oldspeak to the node's oldspeak. I originally planned on doing the same for the newspeak, but then I realized I need to check if it is NULL or not first. I added this check and only called strdup() on the newspeak if it was not NULL.

node_delete() frees the pointers to the oldspeak and the newspeak, as calling strdup() uses memory. I then freed the node pointer and set it to NULL.

For printing purposes, I again followed the format described in the assignment document. I checked the two cases (oldspeak with NULL newspeak and oldspeak with non-NULL newspeak) and printed their respective formats.

Binary Search Tree Implementation

bst_create() simply returns NULL. I thought I had to make a node at first and set that equal to NULL, but rereading the document and watching a lab section made me realize that was unnecessary. This NULL represents the beginning of a new tree.

bst_height() makes use of a helper max() function that I wrote. It finds the larger value between two integers and returns it. Using this, I recursively called the leftmost node and the rightmost node and found the greatest height between the two. I added one to this value and returned this as my final height. I needed to add 1 to account for the height taken up by the root node. This function is also written such that if the given root is NULL, it returns a height of 0 (meaning there is no tree).

bst_size() has a counter that increments itself by 1 for every recursive call. Each call goes to a node that is not NULL, so the number of recursive calls is also the number of non-NULL nodes in a tree. I returned counter in the end.

bst_find() first checks to see if the given oldspeak is at the current root, using strcmp() to do so. If it is, then we have already found the node that we needed to find. If the oldspeak is not at this node, then I check the left and/or right of the node as needed. This entails a recursive call, so it checks if the oldspeak is at the current root once again. If I parse through the entire tree and the oldspeak is not found anywhere, then that means it does not exist in the tree. For this reason, I return NULL. For printing statistics in banhammer.c, I also included a variable that keeps track of the number of recursive calls. This will be used and described later.

bst_insert() has two conditions like the previous function: the root can be NULL or non-NULL. If it is NULL, then I need to create a new node to act as the root of a new binary search tree. As such, I create a new node with the given oldspeak and newspeak and return it. Otherwise, we can assume that a tree exists. Once again, I recursively called the function to find its lexicographical position, checking the left and/or right of the node as needed. Once the correct position is found, I insert the node. The number of recursive calls were kept track of here as well for banhammer.c purposes.

Printing a tree in inorder traversal requires recursively calling the left of the root, printing the node, then recursively calling the right of the node. This ensures that the nodes will be printed in the correct order. We learned about this in class, so I made use of the resources provided then.

Finally, deleting a tree in postorder traversal requires recursively calling the left, then the right, calling node_delete(), and setting the pointer to NULL. Again, we learned about this class.

Bloom Filter Implementation

Allocating memory with malloc() was done once again in bf_create. Given the created BloomFilter pointer, I set up its primary, secondary, and tertiary arrays using the values provided in salts.h. I was confused on which indices should contain which values, as there are three indices but only two salt values per array, but I learned with help from a TA that this does not really matter (granted, there would be some miniscule differences between my work and the example file in the resources repository). I opted to set the low values in the zeroth index and the high values in the first index for each array as this made the most logical sense to me. For setting up the bloom filter's bit vector, I simply made a call to bv_create().

Deletion only required freeing the vector memory with bv_delete() and freeing the bloom filter pointer and setting it to NULL, as I have been doing for the other files.

bf_size() simply needs the size of the bit vector, which would be a call to bv_length().

Inserting into the bloom filter requires hashing the oldspeak with all three of the salts. Each value would represent one of the three bits of the bit vector that needed to be set, or made 1. I used bv_set_bit() here, as it did exactly what I needed to do.

bf_probe() is similar to bf_insert() in the sense that it also hashing the oldspeak with each salt. However, here, it checks the value at the bit of each return. If the index of the bit vector for any of the three returned values was 0, I had it return false immediately. This means that the oldspeak is not in the given bloom filter. However, if all of the indices had values of 1, then it means that the oldspeak is in the bloom filter, in which case I would return true.

bf_count() looks at all of the indices of the bit vector. Using a counter initialized to 0, it increments by 1 for every index that has a value of 1. If it encounters an index with a value of 0, then it simply continues as this does not represent a set bit.

I originally wrote bf_print() to be just like bv_print, printing 1's and 0's as seen in the bit vector. However, when I needed to test this function as I was debugging, I realized it would be easier to read if I printed onto a new line after every 16th number. To do this, I took the mod of the index by 16 and printed a new line if it resulted in 0. This was only for formatting/readability purposes.

Hash Table Implementation

I once again allocated memory with malloc() to the size of HashTable. Going through the structure members, I set the salt array to the provided values in salts.h, just like bf_create. I also allocated enough memory for each node of the trees array. I learned from a TA's section that ht->trees was an array of pointers to binary search trees, so I had each node at each index set to NULL.

Deletion first requires freeing the memory in the trees array. Iterating through all the indices, I called bst_deleted() if the node there was not already NULL. If it was already NULL, I simply skipped over it. Then, I freed the hash table pointed and set it to NULL.

ht_size() is merely the size provided as a member of the hash table structure. I returned this value.

Like bf_probe, I hashed the oldspeak and used its return value as the index of my table in ht_lookup(). I called bst_find() then to see if the oldspeak was there. If it wasn't, then there was nothing to return besides NULL. If it was, then I had it return the pointer there. This is really important for detecting false positives in banhammer.c.

ht_insert() is a lot like bf_insert(). I once again hashed the oldspeak and used its return value as an index. I called bst_insert() to insert the oldspeak and newspeak (even if it's NULL) there.

ht_count() is almost exactly like bf_count(). Once again using a counter, I checked to see if each index of the hash table had a binary search tree. Though there was an easier method of doing this, it made more sense to me to check its height. I wrote bst_height() in a way such that it would return 0 if the root was NULL/if there was no tree, so I checked this condition here. If calling bst_height() at an index returned 0, I would continue looping. Otherwise, I would increment the counter and finally return it once done looping.

ht_avg_bst_size() and ht_avg_bst_height() are exactly the same except for one tiny portion of one line. Both make use of the formulas described in the assignment document, where I need to find the cumulative size or height of all non-NULL binary search trees and divide it by the total number of trees in the hash table. The latter is what ht_count() calculates, so I made a call to it. For size and height, I called bst_size() and bst_height respectively as I iterated through all indices of the table.

Again iterating through the hash table, I called bst_print() at each index. Since each index of the table contains a binary tree, I figured a call like this would be sufficient.

Banhammer Implementation

I originally planned on writing everything I needed to do for banhammer.c in its main function, but I realized that it got really long and hard to keep track of where I was. To address this issue, I decided to split what I needed to do into helper functions. From this came create_report() and print_file(), which I will describe after I describe my main.

First off, I defined variables that I would need to use, and then I parsed through common line options with getopt() as I have been doing for several assignments now. I used optarg for -t and -f to use user values, set my statistics boolean flag to true if enabled, and had my help message print itself if called. Past the command line options, I initialized my bloom filter and hash table right away, as described in the document. I used fscanf() in a while loop to read the oldspeak in badspeak.txt next. I figured this would be the fastest and easiest way to do this. I am also most comfortable reading files like this as

I have done this for other assignments too. I broke out of the while loop when fscanf() returned a value less than or equal to 0, indicating that it reached the end of the file. Before breaking, I would insert each read oldspeak into the bloom filter as well as the hash table with a NULL newspeak. This is because badspeak terms do not have a newspeak translation. Closing this file, I used the same file pointer (to use less memory) to open newspeak.txt. I went through the same process for this file, using fscanf() in a while loop. However, when inserting into the hash table, I added its newspeak translation instead of NULL. Once I was done reading, I closed the file and moved onto my helper function create_report(), which took three parameters: my bloom filter, hash table, and statistics boolean flag.

In create_report(), I implemented my regex parsing module and printed my message reports as required. I checked my WORD pattern with regcomp() to ensure that compilation would not fail. If it did, it would print an error to let me know and exit. Next, I opened two files to write any badspeak and rightspeak (oldspeak with nespeak translations) that I would read in from stdin. I figured this would be the most straightforward way to keep track of which words I encounter as I go through stdin. With these files opened for writing, I went into another while loop, calling next_word() and ensuring that it was not NULL in the condition. In the loop, I checked if the first read-in word was in the bloom filter. If it was not, then it definitely is not a forbidden word and I would continue looping. If it was in the bloom filter, I called ht_lookup to see if it was in the hash table. If the value that this function returned was NULL, then it would indicate that the bloom filter resulted in a false positive and the word is not truly a forbidden word. In this case, I would continue looping and reading in the next word. If it returned a non-NULL node, however, that would mean that the word is definitely a forbidden word. In this case, I would proceed to see if the word was a badspeak or rightspeak word. This would allow me to determine which file the word should be written to. Here, I checked if the newspeak was NULL, which would mean that it is a badspeak term and should be written to my badspeak file. I also increment a counter I had made to keep track of the number of badspeak words here. If the newspeak was not NULL, then it would have to be a rightspeak word that must be written to my rightspeak file. Again, I incremented another counter here to keep track of the number of rightspeak words used. However, before printing, I would need to check the statistics flag. If it was enabled, then I would need to suppress the message that I would have otherwise printed out. If the option was given in the main function, then I calculated the necessary statistics and printed them out as shown in the example binary in the resources repository. If it was not enabled, then I would proceed to print messages. This is where the counters I made come in. If both counters had values greater than 0, it meant that the user used both badspeak and rightspeak. I would print the mixspeak message from messages.h if so along with all the content from both the badspeak file and rightspeak file, which involves calls to my other helper function, print_file(). This will be described below. If the badspeak counter was not 0 but the rightspeak counter was 0, then the user used only badspeak. In this case, I printed the badspeak message with the content from the badspeak file. On the other hand, if the rightspeak counter was not 0 but the badspeak counter was, then the user only used rightspeak. This would require printing the goodspeak message with the contents of the rightspeak file. Again, all instances of printing content from files were calls to print_file().

This function only took one one parameter, that being a char pointer to the file name. When I used this function, I simply provided the name of the files that contained my badspeak and rightspeak respectively. In this function, I opened the file to be read with fopen() and exited if I was unable to open it. This never happened, but I thought it would be consistent of me to include this check. Then, I proceeded to read each line of the file with fgets(), as I have done for a previous assignment. This required making a buffer of a size that I assumed to be 1024. If fgets() returned NULL, then that would mean that I reached the end of the file and have nothing else to read. I would break out of the loop and close the file in this case. While still reading, I would print the line's contents that were read and stored in the buffer. This was perfect since printing out the message report required printing each file's words, formatted as one oldspeak or one set of oldspeak and newspeak, line by line. Fgets() was very helpful for this reason.

Notes

- The variables *lookups* and *branches* are defined in ht.c and bst.c respectively. They are identified as extern variables in banhammer.c to be used when printing statistics.
- Default hash table size is 2^16, which is represented as 1 << 16 in my code.

- Default bloom filter size is 2^20, which is represented as 1 << 20 in my code.

Other (notes taken as I watched lab sections and planned out what I needed to do)

HASH TABLE

ht_trees = array of BST roots
(all start at NULL)

ht_insert
hash oldspeak
root= bst_insert() ← update ptr in HT

[
bv → bf
node → bst → ht
ht & bf → banhammer
]

PARSER

① compile regex
② read from stdin, parse w/ ①
③ free memory
(clear_words, regfree)

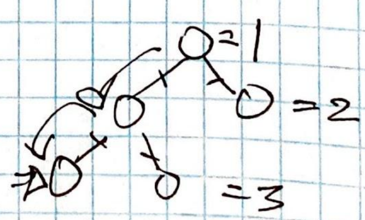
[a-z A-Z 0-9 ~~-'~~]+

[a-z]+
word is 1 or more letters

word-word
[a-z]+-[a-z]+
⇓
more inclusive with
([a-z]+-)+[a-z]+

node — nodes
bv — bitvector
bst — binary search tree
ht — hash table
bf — bloom filter

[
only rightspeak = wrongthink, good message
only badspeak = thoughtcrime, bad message
both = mixspeak message
]



4 links/branches
2 lookups