Diwa Ashwini Vittala
CSE 13S - Fall 2021

⁰❀ Assignment 7: Design ❀。

THE GREAT FIREWALL OF SANTA CRUZ

About the program:

This program uses hash tables, bloom filters, and binary trees to filter specified “bad words”.

Banhammer:

The main (executable) file. It takes a given input text and filters out any badspeak.

Bloom filter (bf.c):

The bloom filter is an array of bit vectors. It is used in banhammer to contain oldspeak words. Bloom filters use 3 salts (primary, secondary, and tertiary) to set bits at a given index and identify if a given oldspeak is likely to be in the filter.

Binary search trees (bst.c):

A BST is a tree made of nodes, each with their own left and right child until the bottom of the tree. I use two BSTs to carry the list of bad words (and bad words with newspeak) in banhammer. The number of branches traversed is calculated in bst_find() and bst_insert(). Since we are working with a tree, I used recursion in almost all of the BST functions to perform the same operation at each node.

Bit Vectors (bv.c):

A bit vector is an array made of 8 bit unsigned integers, and can be manipulated using bitwise operations. For example, a bit at a given index i in the vector array can be set (made 1) using the OR (|) and left shift (<<) operations: vector[i/8] | (1 << i%8). Since the vector is 8 bits long, we have to divide by 8 to find the index, and mod the index by 8 to find the right spot for the 1. Clearing a bit uses the AND (&), NOT (~) and left shift (<<) operation. To get a bit, we can use a right shift (>>) to move the bit at the index we want to the right, then use AND (&) to preserve just that one bit.

Hash Tables (ht.c):

Hash tables are arrays containing root nodes to BSTs. There is a salt in hash tables to find the index where we can likely find or insert a given oldspeak in ht_lookup() and ht_insert(). The number of lookups is also incremented in these two functions. The ht.c file also contains functions to find the average BST size and height, which will be used to print out statistics in banhammer.

Nodes (node.c):

Nodes each contain their own oldspeak and newspeak. They can have left and right child nodes. I use nodes to make up the binary search tree (BST).

About the executable:

banhammer.c

Inputs: The program takes in the following commands:

-h help, displays the program synopsis and usage

-s print program statistics
-t size of hash table (default: 2^16)
-f size of bloom filter (default: 2^20)
The program also asks for some text input. For example, I can input “hello, my name is Diwa” to check if I used badspeak, and the program will check if “hello”, “my”, “name”, “is”, and “Diwa” are bad words. If any of them are bad words, then a message about the crime committed is displayed.

Outputs:

Statistics will be printed if specified. If statistics are not specified (-s), then any crime that was committed will be printed along with the words that qualified the violation.

Other files involved:

Makefile, README.md, set.h, banhammer.c, bst.[c,h], bv.[c,h], bf.[c,h], ht.[c,h], node.[c,h], parser.[c,h], speck.[c,h], salts.h, messages.h

Pseudocode/information about each file:

banhammer.c:

Include the header files for all the necessary files
Message function:
    Prints the synopsis and usage
Lowercase function:
    While the index of the character string is not null:
        Make the character lowercase
        Increment the index
    Return the word (which is a character string)
Enumerate the banhammer options, including punishments
Main function:
    Create a set to contain command options
    Create a set to contain the crimes committed
    Set the default table and filter size
    While we are parsing through the command line options:
        In the case that “h” was chosen:
            Print the help message
            End the program
        If “s”:
            Add statistics printing to the set
        If “f”:
            Set the bloom filter size to the given input
        If “t”:

If the number given is less than 0 (negative) or too large:
Print an error message and end program
Otherwise, set the hash table size to the given input
By default:
Print the help message
End the program
If the size of either table is 0:
Print an error message and end program
Open the badspeak.txt and newspeak.txt files for reading
Create a bloom filter and hash table
Read in the list of badspeak words from badspeak.txt:
Add each word to the bloom filter and hash table (with null newspeak)
Read in the list of oldspeak and newspeak from newspeak.txt:
Add each pair of words to the bloom filter (just oldspeak) and hash table
Compile the regular expression so that we can find a-z, A-Z, 0-9, _, ‘, and - with words
If there's an error in compilation, delete the bloom filter and hash table,
Close the badspeak and newspeak text files.
Create two binary search trees to contain
(1) only bad words and
(2) contain badspeak with newspeak translations
Parse the input and filter out any bad words
Make sure the word is all lowercase
For each word read in, check to see it was added to the bloom filter
Look for the node containing the the oldspeak word
If the word does not have a newspeak translation, they are guilty of thoughtcrime:
Add thoughtcrime to the punishment set and the word to the bad words list
If the word has a newspeak translation, they need to be advised on rightspeak:
Add rightspeak to the punishment set and word to the bad words list with newspeak translations
If verbose (aka statistics printing) was chosen:
Print the average BST size and height, average branches traversed, hash table load (calculated as the hash table count / hash table size), and the bloom filter size (calculated as the bf count / bf size)
Otherwise:
If thoughtcrime and rightspeak counseling needed:
Print the mixspeak message and print both the bad words list
(only oldspeak first, then the oldspeak with newspeak translations)

If thoughtcrime but not rightspeak counseling:
Print the badspeak message and print both the bad words list
(only oldspeak first, then the oldspeak with newspeak translations)
If rightspeak counseling but not thoughtcrime:
Print the goodspeak message and print both the bad words list
(only oldspeak first, then the oldspeak with newspeak translations)
Clear any memory that was allocated (including for regular expressions)
Close all files

bst.h:
Provided by professor, declarations of all the functions/variables/structures/basic headers required for bst.c

bst.c:
Include all necessary header files
Branches is an external variable that counts the number of branches traversed in the BST
BST create (create the binary search tree):
Returns null node
BST delete (delete the binary search tree):
If root is valid:
If left node exists, delete it
If right node exists, delete it
Delete the root
BST height (calculates the height of the binary search tree):
If the root is valid:
If the height of the left side is greater than the height of the right side:
Return the height of the left side plus 1 for the root
Else:
Return the height of the right side plus 1 for the root
If the root is not valid, the height is 0
BST size (calculates the size of the binary search tree):
If the root is valid:
Return the size of the left side, the size of the right side, plus 1
Otherwise, size is 0
BST find (uses recursion to look through the tree based on the string and find oldspeak in the binary search tree):
If the root is valid:
If the oldspeak of the root is longer than the oldspeak:
Increment the branches since we are moving into a root
Find oldspeak in the left side

Else if it is shorter:
Increment the branches since we are moving into a root
Find oldspeak in the right side
Return the root

BST insert (uses recursion to insert a given oldspeak/newspeak combo into the tree):
If the root and oldspeak are valid:
If the oldspeak of the root is longer than the oldspeak:
Increment the branches since we are moving into a branch
Insert into the left root
If the oldspeak of the root is shorter than the oldspeak:
Increment the branches since we are moving into a branch
Insert into the right root
Return the root
Otherwise if oldspeak is null:
Return null
Otherwise if the root is null:
Return a new node with the given oldspeak and newspeak

BST print (uses post-traversal to print out the binary search tree):
If root is valid:
Print the root node
If the left root exists, print it
If the right root exists, print it

bv.h:

Provided by professor, declarations of all the functions/variables/structures/basic headers required for bv.c

bv.c:

Include all necessary header files

Structure for a Bit Vector (provided by the professor):
The length is an unsigned integer of 32 bits
The vector is an array made up by unsigned integers of 8 bits

Bit Vector Create (creates the bitvector):
Allocate some memory for the bitvector
If the bit vector is valid:
Set the length
Allocate memory to the vector array
Return the bit vector

Print bit vector:
Loop through the bit vector until length:

Print the bit at that index

Bit vector delete:

Free the array

Free the bit vector

Set the bit vector to null

Bit vector length:

Returns the length of the bit vector

Bit vector set bit:

If the index is within range:

Use bitwise operations to set the bit (1) at the given index using
OR and left shift (array at the index | (1 left shift to the index))

Return true to indicate bit was set

Otherwise:

Return false to indicate that the bit was not set

Bit vector clear bit:

If the index is within range:

Use bitwise operations to clear the bit (0) at the given index using
AND, NOT, and left shift (array at the index & ~(1 shifted left to index))

Return true to indicate bit was cleared

Otherwise:

Return false to indicate that the bit was not cleared

Bit vector get bit:

If the index is within range:

Return the bit vector after performing the bitwise operation
Right shift by index mod 8 (due to the unsigned 8 bit length),
AND by 1 to preserve at the given index

Otherwise:

Return false to indicate that the bit was not within the range

bf.h:

Provided by professor, declarations of all the functions/variables/structures/basic headers required for bf.c

bf.c:

Include all necessary header files

Structure for bloom filter (provided by the professor):

A primary salt array which contains two unsigned integer of size 64 bits

A secondary salt array which contains two unsigned integer of size 64 bits

A tertiary salt array which contains two unsigned integer of size 64 bits

A filter array made up by bit vectors

Bloom filter create:
    Allocate memory to the bloom filter
    If the bloom filter is valid:
        Set the low (index 0) and highs (index 1) of each of the three salts
        Set the filter to a newly created bit vector
        If the bit vector if not valid:
            Delete the filter, free the memory, and set the bloom filter to null
    Return bloom filter
Bloom filter delete:
    If the bloom filter and the bloom filter array are valid:
        Free the memory allocated to both of them
        Set the bloom filter to null
Bloom filter size:
    Return the bloom filter length
Bloom filter insert:
    Find the index using all three salts. This is the hash mod the size of the bloom filter
    Set the bit at the given three indices
Bloom filter probe:
    Find the index using all three salts. This is the hash mod the size of the bloom filter
    If the bloom filter is set at all three indices:
        Return true
    Otherwise:
        Return false
Bloom filter count:
    Set a count
    Loop through the bit vector and add to the count if the bit is set at an index
    Return the count
Bloom filter print:
    Prints the bit vector part of the bloom filter

ht.h:
Provided by professor, declarations of all the functions/variables/structures/basic headers required for ht.c

ht.c:
Include all necessary header files
Lookups counts the number of times that ht_lookups() and ht_insert() are called
Structure for hash table (provided by the professor):
    A salt array which contains two unsigned integer of size 64 bits
    A size that is an unsigned integer of size 32 bits

A trees array made up of nodes

Hash table create:

Allocate memory to the hash table

If the hash table is valid:

Set the low (index 0) and high (index 0) of salt

Set the size of the hash table

Allocate memory to the trees array with null nodes

Return the hash table

Hash table print:

If the hash table is valid:

Loop through the hash table and print the BST

Hash table delete:

If the hash table and the trees array are valid:

Loop through the tree and delete the tree elements

Free the hash table trees array

Free the hash table

Set the hash table to null

Hash table size:

Return the size of the hash table

Hash table lookup:

If the hash table and oldspeak are valid:

Increment the lookups

Set the index to the hashed salt mod size of the hash table

Return the node found after performing a bst find on the trees array at index

Otherwise, return a null node

Hash table insert:

If the hash table and the oldspeak are not null:

Increment the lookups

Set the index to the hashed salt mod size of the hash table

Return the node found after inserting oldspeak at the given trees index

Hash table count:

Set a variable to track the count

Loop through the tree and add the size of the tree to the count

Return the count

Hash table average BST size:

Sum the sizes of the BSTs by looping through the hash table

Divide the sum by the number of non-null BSTs in the hash table

Hash table average BST height:

Sum the heights of the BSTs by looping through the hash table

Divide the sum by the number of non-null BSTs in the hash table

node.h:
Provided by professor, declarations of all the functions/variables/structures/basic headers required for node.c

node.c:
Include all necessary header files
Node create:
    Allocate memory to the node
    If the oldspeak is not null:
        Use string duplicate to set the oldspeak
    If the newspeak is not null:
        Use string duplicate to set the newspeak
    Return the node
Node delete:
    If oldspeak is valid:
        Free the oldspeak
    If newspeak is valid:
        Free the newspeak
    Free the node
    Set the node to null
Node print:
    If oldspeak and newspeak are both valid:
        Print them both
    If the oldspeak is valid:
        Print the oldspeak

parser.h:
Provided by professor, declarations of all the functions/variables/structures/basic headers required for parser.c

parser.c:
Provided by professor
Contains functions to find the next word and clear words, which I use in banhammer for parsing the input using regular expressions.

speck.h:
Provided by professor, declarations of all the functions/variables/structures/basic headers required for speck.c

speck.c:
Provided by professor
Contains functions that allow hashing. I use the hash function to find the indices in hash tables and bloom filters.

messages.h:
Provided by professor, contains the messages that will be printed if there is some badspeak in the input for banhammer.

salts.h:
Provided by professor, contains definitions for the salt lows and highs for the primary, secondary, tertiary, and hash table salts

set.h:
Provided by professor, contains functions that can be used for set operations