Isaac Flores
CruzID: **isgflore**
Prof. Miller
CSE13S
November 20, 2022

Final Design for ASGN 6: The Great Firewall of Santa Cruz: Bloom Filters, Linked Lists, and Hash Tables

Overview: The banhammer.c file uses bloom filters, doubly-linked lists, hash tables, and word parsers to check user input for old speak words. If the old speak word has a new speak translation, the user is told to use the new speak instead. If the oldspeak word has no newspeak translation, the user is sent to joycamp for "counseling".

General Idea:

banhammer.c: Creates a hash table and bloom filter with data of each oldspeak word with its newspeak translation and each badspeak word. Reads user input line for line and parsers each word. If a user uses oldspeak or badspeak those words are stored for future messages. Once the user has finished inputting text, each oldspeak and badspeak word is listed and the user is notified of their wrongdoings.

main(); The main function performs the task described above. It also allows the user to print out help, select the size of the hash table and bloom filter, enable the move-to-front option, and print out statistics.

print_stats(ht_k, ht_h, ht_m, ht_p, bf_k, bf_h, bf_m, bf_be, count, size): Prints out the statistics collected from running banhammer. The data is collected from the hash table and bloom filter file.

print_help(): Prints out help and usage information if prompted.

ht.c: A hash table is a table that connects a value with a key. In this case, the value is an old speak word and the key is the old speak word's hash function value. This will be used to know which words are old speak and which ones aren't. Values are stored in linked lists to prevent hash collision issues.

*ht_create(size, mtf): Creates and allocated memory for the hash table. Creates and allocates memory for the linked list that will be used in the hash table. Initializes the move-to-front option, salt, size of the hash table, number of keys, number of hits, number of misses, and number of elements examined. Returns a pointer to the new hash table.

ht_delete(*ht): Deletes and frees the memory for the linked lists in the hash table and the hash table itself.

ht_size(*ht): Returns the size of the hash table.

*ht_lookup(*ht, *oldspeak): Hashes the oldspeak to get the index of the location of its node. If the linked list it should be in exists and the oldspeak is contained in that linked list the node is returned or else NULL is returned. Also adds to the number of hits if a node with oldspeak is found, adds to the number of misses if a node with oldspeak isn't found, and adds to the number of list elements examined when looking up the oldspeak word.

ht_insert(*ht, *oldspeak, *newspeak): Inserts the oldspeak word with its newspeak translation into the hash table. The index of oldspeak is calculated by hashing the word. If the linked list the word will be inserted into does not exist yet, it creates the list and adds the word after.

ht_count(*ht): Returns the count of created linked lists in the hash table.

ht_print(*ht): Prints the contents of the hash table which helps with debugging and visualizing the hash table.

ht_stats(*ht, *nk, *nh, *nm, *ne): Sets nk to the number of keys in the hash table, nh to the number of hits, nm to the number of misses, and ne to the number of links examined while looking up an oldspeak word. Uses data collected in the function to collect this data.

node.c: A node is what every element of the linked list contains. This file is used to implement linked lists. Each node points to a previous node and a next node. Each node contains an oldspeak word and a newspeak word. Newspeak words aren't necessary and will equal NULL if not given.

*node_create(*oldspeak, *newspeak): Creates and allocates space for a node. It copies the oldspeak and newspeak words to a new variable and uses that to set the oldspeak and newspeak words. It returns a pointer to the new node.

node_delete(**n): Deletes and frees the memory for the oldspeak word, newspeak word, and the node.

node_print(*n): Prints the oldspeak and newspeak contains in the node if both exist. If newspeak is NULL, it only prints the oldspeak.

*duplicate(*s): Replicates strdup() by allocating the same amount of memory that s needs to a copy variable. It then copies the memory of s into the copy variable and returns a pointer to the copy.

str_length(*s): Returns the length of the string s. Will be used to duplicate strings.

ll.c: A linked list is a list that allocated memory for each new element during run time. This linked list is doubly-linked so each node of the list points to a previous node and a next node. It will start with two sentinel nodes that act as the ends of the list. This list also has a move-to-front option that moves a node to the front if it has been looked up. This linked list will be used to store values for the hash table and it will prevent hash collision issues.

*ll_create(mtf): Creates and allocates memory for the linked list and the head and tail sentinel nodes. The next node for the head is the tail node and the previous node for the tail is the head node. It will also initialize the move-to-front option if set. Returns a pointer to the linked list.

ll_delete(**ll): Runs through the linked list and deletes each node. Once all nodes have been freed, the linked list is freed.

ll_length(*ll): Returns the length of the linked list ignoring the head and tail sentinel nodes.

*ll_lookup(*ll, *oldspeak): Searches for a node that contains the oldspeak. If the node is found, a pointer to it is returned, or else a NULL pointer is returned. If the move-to-front option is enabled, the node that is found is moved right next to the head sentinel node if it isn't already at the front. Also adds 1 to the number of seeks at the beginning and 1 to the number of links for each node searched.

ll_insert(*ll, *oldspeak, *newspeak): Inserts a new node with the given oldspeak and newspeak words. Before inserting a lookup is performed to ensure a node with the given oldspeak does not already exist. If the node does not exist already, create a new node directly after the head sentinel node.

ll_print(*ll): Print each node in the linked list except for the head and tail sentinel nodes.

ll_stats(*n_seeks, *n_links): Set n_seeks to the number of linked list lookups. Sets n_links to the number of links traversed during lookups. This data is tracked globally across all linked lists made.

str_compare(*s, *w): Compares the strings s and w. Will be used to look up oldspeak words in the linked lists.

bv.c: Bit vectors are arrays of bits. Each element in the vector will be 64 bits long. The bit vector will be used to implement bloom filters.

*bv_create(length): Creates and allocates memory for the bit vector and the vector array. Sets the length of the bit vector to the given length. There will be length amount of accessible bits in the vector. If the vector can not allocate length amount of memory the function returns NULL. Else, it returns a pointer to the bit vector. Every bit in the vector is set to 0.

bv_delete(**bv): Deletes and frees the memory for the vector and the bit vector.

bv_length(*bv): Returns the length of the bit vector.

bv_set_bit(*bv, i): Sets the ith bit of the bit vector.

bv_clr_bit(*bv, i): Clears the ith bit of the bit vector.

bv_get_bit(*bv, i): Returns the ith bit of the bit vector.

bv_print(*bv): Prints each element of the bit vector array.

bf.c: A bloom filter is a field of bits that give information. The information this bloom filter will store is if a word is an oldspeak word. It does this by hashing an oldspeak word with 5 different salts.

*bf_create(size): Creates and allocates memory for the bloom filter. The bit vector for the bloom filter will be size bits long. Sets the five salts that will be used. Returns a pointer to the bloom filter.

bf_delete(**bf): Deletes and frees the memory for the bit vector and the bloom field.

bf_size(*bf): Returns the size of the bloom filter.

bf_insert(*bf, *oldspeak): Hashes the oldspeak word with 5 different salts. The hash value is modulo by the size of the bloom field. That value is used as the location of the bit that will be set

in the bit vector contained in the bloom field. If the bit is already set, nothing is done. If the bit isn't set, the bit is set and the number of keys is incremented by 1.

bf_probe(*bf, *oldspeak): Checks if the oldspeak word is probably in the bloom field. Hashes the oldspeak word with 5 different salts. The hash value is modulo by the size of the bloom field. It then checks if the bit at the location of the value is set. If it isn't set, the number of misses is incremented by 1 and the function returns false. If it is set the number of hits is incremented by 1. For both cases, the number of bits examined is incremented by 1. If all five locations are set the function returns true.

bf_count(*bf): Searches each accessible bit in the bloom filter. Checks if the bit is set and if it is a counter variable is increment by 1. Returns the count once every accessible bit has been checked.

bf_print(*bf): Print the bit vector contained in the bloom field.

bf_stats(*bf, *nk, *nh, *nm, *ne): Sets the nk to the number of keys, nh to the number of hits, nm to the number of misses, and ne to the number of bits examined.

parser.c: Reads a line from a file and checks each word in the line. This function will be used to check if the user uses any oldspeak or badspeak words.

*parser_create(*f): Creates and allocates memory for the parser. Opens the given file with reading permissions.

parser_delete(**p): Closes the given file. Deletes and frees the memory allocated for the parser.

next_word(*p, *word): Takes a parser and a word buffer. It will find the next valid word for the buffer in the parser and store it in the word buffer that was given. The word will be normalized meaning all uppercase letters will be converted to lowercase. A word is defined as being separated by a space. Numbers, dashes, and single quotes are allowed in words. The normalized word is stored in the word buffer that was given. If a word can be successfully parsed, the function returns true. If there are no more words to check, the function returns false and the word buffer is undefined.

copy_next_word(*p, *word, *current_char, *index): Copies the next available word of the current line to *word starting from the current character of the current line. Used to not repeat code.

next_valid_word(*p, *current_char): Leaves the current character at the start of where the next available word is. Used to not repeat code.

string_length(*s): Returns the length of the string s. Will be used to duplicate strings.

Makefile: Compiles all the code and allows different targets to run. Make and make all compiles banhammer and all the c files it needs. Make clean removes files that the compiler made except for the executable. Make clean removes files that the compiler made. Make format clang formats the c and header files.

Pseudocode:

banhammer.c:

main():

```
# take the options from the user
        # -h prints out program usage help
        # -t size sets the size of the hash table to user input, the default value is 10,000
        # -f size sets the size of the bloom filter to user input, the default value is 2^19
        # -m enables the move-to-front option, default is disabled
        # -s will print out statistics, the statistics are the data collected from the bloom filter and hash table. It also includes bits examined per bloom filter miss, the number of false positives, the bloom filter load, and the average seek length
# open the badspeak and newspeak files with reading permission
# create the bloom filter and hash table
# create a parser for the badspeak and newspeak files
# while there are still next words to read
        # add the read word to the hash table and bloom filter
# while there are still next words to read
        # get the second next word from the oldspeak file
        # add the 1st next word to the bloom filter and the 1st and 2nd next word to the hash table
# close the badspeak and newspeak files
# create two linked lists to hold the oldspeak and badspeak the user uses
# while there are still next words to read
        # if the next word is in the bloom filter
                # lookup the node the next word is stored inside the hash table
                # if the next word is in the hash table and doesn't have a newspeak translation
                        # add the word the badspeak linked list
```

if the next word is in the hash table and does have a newspeak translation
add the word to the oldspeak linked list with its newspeak translation
if statistics are wanted
collect the statistics from the bloom filter and hash table
print the statistics
else
if the user used both oldspeak and badspeak
print the oldspeak words with their newspeak translations and the badspeak words the user used with their respective message
if the user only used badspeak
print the badspeak words the user used with their respective message
if the user only used oldspeak
print the oldspeak words the user used with their respective message
delete the parsers used, the linked lists used, the hash table used, and the bloom filter used
return 0

print_stats(ht_k, ht_h, ht_m, ht_p, bf_k, bf_h, bf_m, bf_be, count, size):
print out the bits examined per miss which is the (bloom filter bits examined minus (number of hashes times bloom filter hits)) divided by the number of bloom filter misses
print out the number of false positives which is the hash table misses divided by the bloom filter hits
print out the bloom filter load which is the bloom filter count divided by the bloom filter size
print out the average seek length which is the hash table elements examined divided by the (hash table hits plus hash table misses)

print_help():
prints the help and usage information for banhammer

ht.c:

struct:
initialize the salt, size, number of keys, number of hits, number of misses, number of links examined, the move-to-front option, and the array of linked lists

*ht_create(size, mtf):
allocate memory for the hash table
if successfully allocated
set the move-to-front option to mtf
set the salt

    # set the number of keys, number of hits, number of misses, and number of links examined to 0
    # set the size to what the size
    # allocate memory for size amount of linked lists for the linked list array
    # if unsuccessfully allocated, free the hash table
return a pointer to the new hash table

ht_delete(*ht):
run through each created linked list and free it
free the hash table and set it to NULL

ht_size(*ht):
return the size of the hash table

*ht_lookup(*ht, *oldspeak):
get the index of the linked list by hashing the oldspeak word with the salt
set the index to itself modulo the size of the hash table
store how many links were traversed before looking up
if the linked list at the index location exists and oldspeak is in that linked list
    # store how many links were traversed before looking up
    # set a current node to the node returned by looking up the oldspeak word in the linked list at the index location
    # collect the updated links traversed stats
    # set the number of links examined to itself plus (the number of links that were traversed before looking up minus the updated links traversed stats)
    # add 1 to the number of hits
    # return the current node
else
    # store how many links were traversed before looking up
    # set the number of links examined to itself plus (the number of links that were traversed before looking up minus the updated links traversed stats)
    # add 1 to the number of misses
    # return NULL

ht_insert(*ht, *oldspeak, *newspeak):
get the index of the linked list by hashing the oldspeak word with the salt
set the index to itself modulo the size of the hash table
if the linked list at the index location does not exist yet
    # create the linked list at the index location
    # insert the oldspeak and newspeak word in the linked list created

        # add 1 to the number of keys
else if the oldspeak word is contained in the linked list at the index location
        # do nothing
else
        # insert the oldspeak and newspeak word in the preexisting linked list
        # add 1 to the number of keys

ht_count(*ht):
run through all linked lists in the array of linked lists
        # if the linked list does not equal to NULL, add 1 to a count variable
return the count variable

ht_print(*ht):
run through all linked lists in the array of linked lists
        # if the linked list does not equal to NULL, print the linked list

ht_stats(*ht, *nk, *nh, *nm, *ne):
set *nk to the number of keys, *nh to the number of hits, *nm to the number of misses, and *ne to the number of links examined

node.c:
struct:
initialize the oldspeak and newspeak values stored in the node
initialize the pointer to the next and previous node

*node_create(*oldspeak, *newspeak):
allocate memory for the node structure
if the node was created
        # the node's oldspeak value is set to a duplicate of the oldspeak word passed in
        # the node's newspeak value is set to a duplicate of the newspeak word passed in
        # the pointer to the next and previous node is set to NULL

node_delete(**n):
if the node exists
        # free the memory allocated for the oldspeak and newspeak words
        # free the memory allocated for the node and set the node pointer to NULL

node_print(*n):
if newspeak is NULL print out the oldspeak word

if newspeak isn't NULL print out the oldspeak and newspeak word

*duplicate(*s):
if s is NULL, return NULL
get the length of the word s and add 1 to it to account for the terminating character
allocate length amount of memory for a copy variable
copy every character from s over to the copy variable
return a pointer to the copy variable

str_length(*s):
run through each character of s and add 1 to the total number of characters
return the total number of characters

ll.c:
struct:
initialize the length of the list
initialize the node for the head and tail sentinels
initialize the move-to-front option

*ll_create(mtf):
allocates memory for the linked list
set the length to 0 and the move-to-front option to mtf
create the head and sentinel nodes
set the next node for the head sentinel to the tail and the previous node for the tail sentinel to the head

ll_delete(**ll):
run through each node in the linked list
    # delete the current node
free the linked list and set the pointer to the linked list to NULL

ll_length(*ll):
return the length of the list

*ll_lookup(*ll, *oldspeak):
add 1 to the number of seeks
run through each node of the list except for the head and tail sentinel nodes
    # add 1 to the links traversed
    # if the oldspeak word in the current node is the same as the one we're looking for and the move-to-front variable is set to true

```
            # if the current node is already at the front
                # return the current node
            # else
                # set the current node's previous node's next node to the current node's next node
                # set the current node's next node's previous node to the current node's previous node
                # set the current node's previous node to the head sentinel node
                # set the current node's next node to the head sentinel node's next node
                # set the head's next node's previous node to the current node
                # set the head sentinel node's next node to the current node
                # return the current node
        # else if the oldspeak word in the current node is the same as the one we're looking for and the move-to-front variable is set to false
            # return the current node
# return NULL

ll_insert(*ll, *oldspeak, *newspeak):
# if the oldspeak word is not already in the linked list
        # create a new node containing the oldspeak and newspeak words given
        # the new node's next node is what the head sentinel's next node was
        # the new node's previous node is the head sentinel
        # the new node's next node's previous node is the new node
        # the head sentinel's next node is the new node
        # increment the length of the linked list by 1

ll_print(*ll):
# run through the linked list
        # print the node

ll_stats(*n_seeks, *n_links):
# set n_seeks to the number of seeks and n_links to the number of links traversed

str_compare(*s, *w):
# for each character in both words until each word has reached the NULL terminating character
        # if the current character in s does not equal the current character in w, return false
# return true
```

bv.c:

struct:
initialize the length and the vector array

*bv_create(length):
allocates memory for the bit vector
sets the length to the given length
the size of the vector is equal to the given length divided by 64 rounded up 1 if there is a remainder
allocate size amount of 64-bit integers for the vector and set every bit to 0
if the given bits couldn't be allocated, print an error message and return NULL
else return the new bit vector

bv_delete(**bv):
free the memory allocated for the vector and the bit vector
set the pointer of the bit vector to NULL

bv_length(*bv):
return the length of the bit vector

bv_set_bit(*bv, i):
the element the ith bit is stored in is equal to i divided by 64
the position of the ith bit is i modulo 64
set a variable to 1 left shifted position times
the integer stored in the element location of the vector is equal to itself (bitwise or) the variable made in the last line

bv_clr_bit(*bv, i):
the element the ith bit is stored in is equal to i divided by 64
the position of the ith bit is i modulo 64
set a variable to 1 left shifted position times
the integer stored in the element location of the vector is equal to itself (bitwise and) the (bitwise not) of the variable made in the last line

bv_get_bit(*bv, i):
the element the ith bit is stored in is equal to i divided by 64
the position of the ith bit is i modulo 64
set a variable to 1 left shifted position times
set a get_bit variable to the integer stored in the element location of the vector (bitwise and) the variable made in the last line
return 1 if the get_bit variable does not equal to 0, else return 0

bv_print(*bv):
run through each element of the bit vector
	# print the current element of the bit vector

bf.c:
struct:
initialize the salts array
initialize the variable for the number of keys, number of hits, number of misses, and the number of bits examined
initialize the bit vector filter

*bf_create(size):
allocate memory for the bloom filter
if successfully allocated
	# set the number of keys, number of hits, number of misses, and the number of bits examined to 0
	# set that salts array
	# create the bit vector that will be used in the bloom filter
	# if the bit vector is NULL, free the bloom filter and set the bloom filter to NULL
return the pointer to the bloom filter

bf_delete(**bf):
free the memory for the bit vector filter
free the memory for the bloom filter and set the pointer to the bloom field to NULL

bf_size(*bf):
return the size of the bloom filter

bf_insert(*bf, *oldspeak):
for each salt
	# the position of the bit that will be set is the hash value of the current salt and the given oldspeak word modulo the size of the bloom filter
	# if the position of the bit that will be set isn't already set, set the bit and add 1 to the number of keys
	# else do nothing

bf_probe(*bf, *oldspeak):
for each salt

        # the position of the bit that will be probed is the hash value of the current salt and the given oldspeak word modulo the size of the bloom filter
        # check if the position of the bit that is being probed is set and add 1 to the number of bits examined
        # if the bit isn't set, add 1 to the number of misses and return false
        # else add 1 to the number of hits
return true

bf_count(*bf):
for each accessible bit in the bloom filter
        # check if the current bit is set and if it is, add 1 to the total count
return the total count

bf_print(*bf):
print the bit vector used inside the bloom filter

bf_stats(*bf, *nk, *nh, *nm, *ne):
set nk to the number of keys, nh to the number of hits, nm to the number of misses, and ne to the number of bits examined

parser.c

struct:
initialize the file
initialize the line buffer with a max length of 1001
initialize the line offset

*parser_create(*f):
allocate memory for the parser
set the file to be parsed to the given file
se the line offset to 0

parser_delete(**p):
free the memory allocated for the parser
set the pointer for the parser to NULL

next_word(*p, *word):
set the current character to the line offset
set an index variable to 0
create a current word buffer of size 1000

if the current character of the current line is equal to the ending character ‘\0’
	# set the line offset and current character to 0
	# get a line from the parser file and store the read line in the current line array.
	# if getting a line reached the end of the file
		# return false
copy the next word
if the line offset is equal to 0 and the current words length is 0
	# go the the next valid word
	# copy the next word
	# while the current words length is 0
		# set the line offset and current character to 0
		# get a line from the parser file and store the read line in the current line array.
		# if getting a line reached the end of the file
			# return false
		# go the the next valid word
		# copy the next word
go to the next valid word
set the line offset to the current character
get the length of the current word and add 1 to account for the ending character ‘\0’
copy length amount of characters from current word to the buffer for the returned word
return true

copy_next_word(*p, *word, *current_char, *index):
set index to 0
while the current character of the current line is alphanumeric, a hyphen, or a single quote
	# set the index character of *word to the current character of the current line
	# increment the current character and index by 1
set the index character of *word to the NULL ending character ‘\0’

next_valid_word(*p, *current_char):
while the current character of the current line is not alphanumeric, a hyphen, a single quote, and a NULL ending character ‘\0’
	# add 1 to the current character

string_length(*s):
run through each character of s and add 1 to the total number of characters
return the total number of characters

Makefile:
the compiler is clang and the CFLAGS are -Wall -Werror -Wextra -Wpedantic

```
# phony target all depends on banhammer

# target banhammer depends on banhammer.o, city.o, ht.o, ll.o, node.o, bf.o, bv.o & parser.o
	# compile banhammer with clang and with each dependency file
# target %.o depends on %.c
	# compile the given c file with clang and with the CFLAGS
# phony target clean
	# remove all .o files
# phony target spotless
	# remove all .o files and banhammer
# phony target format
	# clang format each .c file
```