Assignment 6: The Great Firewall of Santa Cruz: Bloom Filters, Linked Lists and Hash Tables Design Document

Shirin Rokni
shrokni

November 20, 2022

Citations

This PDF was generated using a LaTeXtemplate made by Jessie Srinivas (CSE13s tutor).
All pseudocode was provided by the assignment pdf, course staff (section leaders, tutors, TAs), or the professors lecture slides.

Purpose

This assignment creates a program called banhammer which is in charge of filtering language which is deemed invalid and giving the user feedback on what they should be using instead. In order to promote virtue and prevent vice, and to preserve social cohesion and discourage unrest, the program will filter through text so that the user cannot be corrupted through the usage of unfortunate, hurtful, offensive and far too descriptive language. It will read in words from stdin, identify any badspeak or old speak and output an appropriate punishment message. The badspeak and oldspeak (with the newspeak translation) that caused the punishment will be punished after the message. It statistics are enabled punishment messages are suppressed and only statistics will be printed.

1 Test Harness

1.1 Building

to building the program, type the following in your terminal

```
make
```

to remove any compiler-generated files, type the following in your terminal

```
make clean
```

1.2 banhammer command-line options

when running banhammer, you can use the following flag options

```
./banhammer [options]
```

these are what the options do

```
-t <ht_size>: Hash table size set to <hf_size>. (default: 1000)
-f <bf_size>: Bloom filter size set to <bf_size>. (default: 2^19)
-s          : Enables the printing of statistics.
-m          : Enables move-to-front rule.
-h          : Display program synopsis and usage.
```

---

2 Bit Vectors

2.1 bv.h

The header file for bv.c, it allows the main program (banhammer) to access the functions within the bv source code. It uses the standard library function stdint.h and this is so that the various functions can be defined using specfic integer types (like uint32_t).

2.2 bv.c

purpose: The function of bv.c is to serve as the array of bits necessary for a proper Bloom filter.

The standard library functions that his file uses are bv.h, stdio.h and stdlib.h. bv.h is used so that the source code can be properly accessed. The stdlib.h header is used to be able to use the malloc function and stdio.h is used in order to use the printf() functions which are vital for testing.

2.2.1 BitVector struct

purpose: A way to group several related variables (members) into one place. This struct contains the variable length which is a uint32_t variable and represents the length of the bit vector, and vector which is a uint64_t variable and represents the vector itself.

2.2.2 bv_create function

purpose: Acts as the constructor for a bit vector.

First, we want to construct the bit vector space, and this can be done with the malloc function. We thus create a BitVector type pointer bv with the allocated space of a BitVector.
Then, we define the length variable in the struct to the length as pointed by the bv pointer.
Next, we want to initialize each bit of the bit vector, and this can be done with using the calloc() function and making the size equal to the length divided by 64 and modulo 64 in order to make the length equivalent to the uint64_t type.
Additionally, we want to return NULL in the event that sufficient memory cannot be allocated. Finally, the function must return bv as it will return the bit vectors construct.

2.2.3 bv_delete function

purpose: Is the deconstructor for the bit vector.

First the bit vector will be freed using the free() function. Then, the memory associated with the vector will be freed using the same free() function. Finally, NULL will be set to the pointer after the memory allocated with the bit vector is freed.

2.2.4 bv_length function

purpose: Returns the length of the bit vector.

Since the value of the length variable will be changed in other functions, all this one need to do is return the length as pointed to by bv.

2.2.5 bv_set_bit function

purpose: Sets the ith bit in a bit vector.

First we want to determine which index that should be manipulated, and that can be done with [i / 64]. Then, we have to determine which bit in the index we choose to manipulate and that can be done with (1 left shift (i modulo 64)). However, since 1 will be treated as a uint32_t type variable, we need to type case it to be a uint64_t variable right before we use it. Finally, we can set the vector as pointed to by bv to the index or the chosen index.

2.2.6 bv_clr_bit function

purpose: Clearing the ith bit in the bit vector.

Essentially, this function uses the same structure as the bv_set_bit function since it is using the same logic except to clear the bits. The differences are that we want to flip the bits so that the function gets a 0 in the location that needs to be cleared. Additionally, instead of using the — operator, it will use the 'and' operator to successfully clear the bits. Thus, it will look like [i / 64] and= (one left shift i modulo 64)) set to the vector as pointed to by bv.

2.3 bv_get_bit function

purpose: Returns the ith bit in the bit vector.

This function will use the logic set by the bv_set_bit function but instead of left shifting, it will be right shifted. Similar to the previous functions, this one will also need to type cast 1 into a uint64 type variable in order to be used. This function also returns a 1 if there is a value at that point and a 0 otherwise.

2.3.1 bv_print

purpose: A way to debug and print the outputs of the various Bit Vector functions.

To test the functionality of get_bit, for example, I would iterate through a for loop until the iterator is less than the length as pointed to by bv. Then, I would use an if statement that checks if the output of bv_get_bit is equivalent to one, then it prints one, and otherwise to print 0. This way, I can see if bv_get_bit is printing what I want it to.

---

3 Bloom Filters

3.1 bf.h

purpose: The header file for bf.c, it allows the main program (banhammer) to access the functions within the bf source code. It uses the standard library functions stdbool.h and stdint.h. stdbool.h is necessary in order to have some functions that are boolean types and return a boolean. stdint.h is necessary in order to have some functions be able to return uint32_t types. It also calls bv.h inside of the local directory because it some of the functions in this file uses functions from them.

3.2 bf.c

purpose: The function of bf.c is to be able to quickly store and process various words. It will be represented as an array of bit vectors. It can take a list of proscribed words and add each word into the bloom filter. If any of the words that the user uses seem to be added to the Bloom filter, then further action will be taken place.

The standard library function that this file uses are stdio.h and stdlib.h. stdio.h is necessary in order to use NULL and in order to use the printf command. Additionally, the stdlib.h header is necessary in order for the functions to use malloc and calloc. However, the function also wants to use some files in the local repository. These files are bf.h, bv.h and cit.h. bf.h is necessary so that the source code can be properly accessed. bv.h is necessary because there are various functions that are being used from that file. Finally, city.h is used for their hash functions.

3.2.1 BloomFilter struct

purpose: A way to group several related variables (members) into one place.

This struct defines the BloomFilter ADT. The five salts will be stored in an array salt. The BloomFilter will also collect statistics. n_keys tracks the number of keys input into the BloomFilter. n_hits tracks the number of probes that return true. n_misses tracks the number of probes that return false, and n_bits_examined tracks the total number of bits examined, which for our BloomFilter will be from 1-5 for each probe.

3.2.2 bf_create function

purpose: The constructor for a Bloom filter. The code was also provided by Professor Miller.

It first defines a uint64_t type variable called 'default_salts' with five different values. Then, it creats a bf pointer using malloc. Next it initializes all of the statistics to 0 and sets the salts pointer to the array of values in default_salts. Finally, if the filter as pointer to by bf is NULL, then it frees the pointer and sets it to NULL.

3.2.3 bf_delete function

purpose: The destructor for a Bloom filter.

All this function needs to do is free memory allocated by the constructor with the free() function and then set the pointer to NULL in order to null out the pointer that was passed in.

3.2.4 bf_insert function

purpose: Takes oldspeak and inserts it into the Bloom filter.

First, the function hashes oldspeak with each of the five salts for five indices. This can be simply done with the hash() function from city. The arguments of such a function will be the salts as pointed to by bf and an incrementing index from 0 - 4, and the oldspeak character. Next, the function sets the bits at those indices in the underlying bit vector. This can be done with the bv_set_bit function with the first argument being the filter variable as pointed to by bf, and the second argument being the output of the hash function modulo the size of the pointer (can be done with bf_size). At the beginning of the function, the n_keys stat will be incremented.

3.2.5 bf_probe function

purpose: Probes the Bloom filter for oldspeak.

First, the function hashes oldspeak with each of the five salt for five indices. This can be simply done with the hash() function from city. The arguments of such a function will be the salts as pointed to by bf and an incrementing index from 0 - 4, and the oldspeak character. Next, the function sets the bits at those indices in the underlying bit vector. This can be done with bv_set_bit with the first argument being the filter variable as pointed to by bf and the second argument being the output of the hash function modulo the size of the pointer (can be done with bf_size). These outputs can all be compared to 1 if they are all checked with bv_get_bit, and if they are all equivalent to one, then the function may return true, otherwise it returns false.

This function is also the location of where a lot of the statistics will be incremented. While the five functions are being hashed, the value of n_bits_examined will be incremented by 1. In the area where all of the values are being checked, the n_hits stat can be incremented every time they equal 1, and the n_misses stat will be incremented otherwise. The n_bits_examined stat will also be incrementing with every for-loop because it is keeping track of all of the bits that are being hashed.

3.2.6 bf_count function

purpose: Returns the number of set bits in the Bloom filter.

The function first creates a uint32_t variable called counter that will be returned with the value of the number of set bits. Then, the function iterates through a for-loop until the bf_size of the bf pointer has been reached. Inside of the loop, it checks for if the bf_get_bit output with the arguments filter and the incrementer equal to 1 (if the bit exists in that location), and if that statement is true, then the counter variable is incremented.

3.2.7 bf_print function

purpose: Prints function for debugging

This function simply needs to use the print function from bv_print, so it calls bv_print with the argument of filter as pointed to by bf.

3.2.8 bf_stats function

purpose: It sets each passed pointers to the value of the original variable for the relevant statistic that the Bloom filter is tracking.

Each statistic that has been created from the struct has been incremented throughout the file. This function will just set each argument to the associated statistic and will thus be changed if they are incremented.

---

4 Node

4.1 node.h

purpose: The header file of node.c; it allows the main program (banhammer) to access the functions within the node source code. It uses no standard library functions.

4.2 node.c

purpose: The function of node.c is to be able to creat a node that will contain a pointer to the previous node and the next node in the linked list. This is because this function implements a doubly linked list.

This file constains the standard library functions stdlib.h, stdio.h and stdint.h. stdlib.h is a necessary header in order to use the malloc() functions. stdint.h is used for functions like printf. stdint.h is necessary in creating functions that return uint64_t type variables. This file also uses the node.h header which is located in its local repository so that the source code can be properly accessed.

4.2.1 mimic_strlen function

purpose: To implement my own implementation of the standard C function strlen.

This function first initializes a variable var to 0. Then, a do-while loop implements which will increment the var until that index of the passed character is not equal to 0. Finally, it returns that var variable.

4.2.2 my_strdup function

purpose: To implement my own implementation of the standard C function strdup.

This function first initializes a s_length size_t variable to the output of my_strlen and a character parameter s. Then, a variable 'copy' is made which will hold the copied string that is being passed in. Then, there is a for-loop that loops until the my_strlen of the passed in character. Each time the loop iterates, the index of the passed in character becomes the index of the copy variable. Finally, the function returns the copied variable.

4.2.3 node_create function

purpose: This function constructs the node.

First, the Node type pointer n is created with the calloc function. Then, the function checks for if the oldspeak character is NULL, and if so, it returns NULL. Otherwise, in the case that oldspeak isn't null, the oldspeak as pointed to from n is equal to the output of my_strdup with the argument of oldspeak. It also checks for if the newspeak character is NULL, and then equals the newspeak pointer from n to NULL. Otherwise, in the case that newspeak is not null, the newspeak as pointed to from n is equal to the output of my_strdup with the argument of newspeak. The function then returns n.

4.2.4 node_delete function

purpose: This function destructes the node.

This function needs to check for if the node exists in order to free it. Thus, it first checks for if oldspeak and newspeak as pointed to from n do not equal to NULL, and if they do not, then the function frees the memory allocated to them with the free() function. Then, it freeds the node n and the pointer to the node is set to NULL.

4.2.5 node_print function

purpose: This function prints the contents of a node.

If the node is null, then it prints null and returns. If the node n contains oldspeak and newspeak, then it prints both oldspeak and newspeak. If the node n contains only oldspeak, then it only prints oldspeak.

---

5 Linked Lists

5.1 ll.h

purpose: The header file of ll.c; it allows the main program (banhammer) to access the functions within the linked list source code. It uses the stdbool.h and stdint.h standard library functions. stdbool.h is necessary because some of the functions within ll.c file will use booleans. stdint.h is used so that the ll_length function can be declared with a uint32_t type. This function also calls node.h from the local directory because various functions will be used from there.

5.2 ll.c

purpose: The function of ll.c is to create a doubly linked list that can contain many nodes in them.

This file contains several standard library headers such as stdlib.h and stdio.h. stdlib.h is necessary to use functions like malloc and calloc, and stdio.h is a necessary header to use the printf() functions. Additionally, the ll.h and node.h header types are called from the local directory. This file uses some functions that are created in the node.c file and thus must be called referenced by node.h. The ll.h header is necessary so that the source code can be properly accessed.

5.2.1 LinkedList Struct

purpose: A way to group several related variables (members) into one place.

This struct defines the LinkedList ADT. A linked list, when constructed, will initially have two sentinel nodes (which are the head and tail nodes). The field mtf signifies the move-to-front technique which will be later expanded upon. The LinkedList interface also tracks basic statistics about the usage of linked lists. It will count the number of linked list look ups performed (seeks), and count the total number of links traversed during those lookups (links).

5.2.2 my_strcmp function

purpose: To implement my own implementation of the standard C function strcmp.

This function compares two strings character by character, so it will iterate through both of the characters, and if they do not equal one another at an index, then the function will return the difference between them. If there are no differences between the functions, then it returns 0.

5.2.3 ll_create function

purpose: The constructor for a linked list. The only parameter that this function takes is a boolean, mtf. If mtf is true, that means any node that is found in the linked list through a look-up is moved to the front of the linked list.

This function first initializes the LinkedList type ll with the malloc function. Then, it initializes the two sentinel nodes head and tail which are both pointed to by ll. The two variables are created with the node_create function. Then, the head and tail pointers must be initialized to the end and tail of the linked list, so head's next will become tail and tail's previous will become head. Then, length is initialized as 0 and mtf is initialized to the mtf as pointed to by ll.

5.2.4 ll_delete function

purpose: The destructor for the linked list.

It first sets a Node type variable 'current' to the head since we are deleting each node in the linked list.. Then, we make a while loop that checks for if the current node is not equal to NULL (because after the tail sentinel node there is a NULL value). Inside of the while-loop, it sets a temporary variable 'next' to NULL. 'next' will then equal current's next which will then be deleted with the node_delete function. Then, the current variable will be equal to the next variable. After the while-loop, the pointer to the linked list is freed and then set to NULL.

5.2.5 ll_length function

purpose: Returns the length of the linked list.

Since the length variable will be incremented in other functions, all this function needs to do is return the length variable as pointed to by ll.

5.2.6 ll_mtf function

purpose: To disconnect the node that will be moving to the front from it's adjacent nodes and then reconnect it to the first node after the sentinel head node.

First we create a Node type variable 'next' which will be equal to the current node's next node. Then, we create a Node type variable 'prev' which will be equal to the current node's previous node. Then, we set previous's next node to next and next's prev to prev. This removes the node from it's adjacent nodes by making them point to each other.

Now, we create a Node type variable 'first' which is equal to head node's next node. Then, we set head's next node to the current node, current's next node to the first node, first's previous node to the current node, and current's previous node to the heads node. This all reconnects the node to be the first non-sentinel node. Finally, the function returns the head node.

5.2.7 ll_lookup function

purpose: This function searches for a node containing oldspeak, and if that node is found, the pointer to the node is returned, otherwise a null pointer is returned.

A node is initialized as 'curr' which will act as the node that contains oldspeak. A for-loop is initialized which sets curr to the very first node after the head sentinel node of the linked lists and loops until curr is equal to the tail node (reaching the end of the linked list). An if statement compares oldspeak to the node with the my_strcmp function. If the if statement goes through, the the for-loop breaks and the function returns the pointer to the node. However, if both the mtf bool and the current node are not equal to NULL and if the current node is not equal to the tail sentinel node, then the ll_mtf function is called with curr as one of the parameters, and the node is moved to the front.

One thing to keep in mind, however, is that this function is where the seeks and links statistics can be incremented. The seeks variable will be incremented at the beginning of the function and the links variable will be incremented for each call of the for-loop.

5.2.8 ll_insert function

purpose: Inserts a new node containing the specified oldspeak and newspeak into the linked list.

First, a node type variable named 'first' is created which will represent the first node after the sentinel head node. Then, another node type variable named 'n' is created which will represent a created node that will be inserted with the function node_create. Then, the head's next node will be equaled to n. Next, n's next node will become the first node. First's previous node will become n and n's previous node will become the head node. This whole process has connecting n as the first node after the head node.

The length statistic will be incrementing at the beginning of this function since it will track how many nodes have been inserted in the linked list.

5.2.9 ll_print function

purpose: Prints out each node in the linked list except for the head and sentinel node.

A for-loop will loop from head's next node until the tail node using curr as the variable that will be looping. Inside of the for-loop, node_print will print the curr variable.

5.2.10 ll_stats function

purpose: Copies the number of linked list look ups to n_seeks and the number of links traversed during look ups to n_links.

Each statistic that has been created from the struct has been incremented throughout the file. This function will just set each argument to the associated statistic and will thus be changed if they are incremented.

---

6 HashTables

6.1 ht.h

purpose: The header file of ht.c; it allows the main program (banhammer) to access the functions within the ht.c source code. It uses the stdbool.h and stdint.h header files from the Standard Library. stdbool.h is necessary because some of the functions that this file uses needs to use booleans. stdint.h is necessary in order to have some functions return in the uint32_t type. It also includes the ll.h header from the local repository because it uses some functions that was created there.

6.2 ht.c

purpose: The function of ht.c is to create a hash table. A hash table is a data structure that maps keys to values and provides fast look-up times. It does so typically by taking a key k, hashing it with some hash function h(x), and placing the key's corresponding value in an underlying array at index h(k). This hash table in particular is a chained hash table, which indexes into an array of pointers to the heads of linked lists. Each linked list cell has the key for which it as allocated and the value which was inserted for that key. When you want to look up a particular element from its key, the key's hash is used to work out which linked list to follow, and then that particular list is traversed to find the element that you're after. If more than one key in the hash table has the same hash, then you'll have linked lists with more than one element.

This file contains several standard library headers such as stdlib.h and stdio.h. stdlib.h is necessary to use malloc and calloc functions, and stdio.h is necessary for the functions to be able to use printf(). This file also calls various header files that are located in the local repository like ht.h, ll.h and city.h. This is because all of these header files have functions that are being called in this one, and must thus be included.

6.2.1 HashTable struct

purpose: A way to group several variables (members) into one place.

This struct defines the HashTable ADT. A hash table contains a salt which is passed to hash() whenever a new oldspeak entry is being inserted. Inside the struct, is the initialization of several variables which will be used as statistics such as n_keys, n_hits, n_misses and n_examined.

6.2.2 ht_create function

purpose: The constructor for a hash table.

The size parameter denotes the number of indices, or linked lists, that the hash table contains. The mtf parameter indicates whether or not the linked lists should use the move-to- front technique. The salt for the hash table is initialized in the constructor as well. Statistics are also collected in the hash table. These statistics are similar to the Bloom filter. n_keys tracks the number of keys that are currently in the hash table. n_hits tracks the number of look ups that found the value that was being looked up. n_misses tracks the number of look ups that didn't find the value being looked up. n_examined tracks the number of linked list elements that were accessed for searches.

So, a HashTable pointer with the name ht is created with the malloc() function. Then, all of the statistics that are defined in the struct are initialized to various values and lists are also redefined with the calloc() function.

6.2.3 ht_delete

purpose: The deconstructor for the hash table.

First, there is an if statement that checks for if the ht pointer is not equal to NULL, and if it doesnt, then it checks for if the lists variable as pointed to by ht is not NULL. If that also is true, then a for-loop is initialized which loops until the size of the hash table. Within the for-loop, there is another check for if the lists at the particular index of the for-loop is not NULL, for which each of the linked lists are deleted using ll_delete. Then, outside of the for-loop, the lists as pointed to by ht are freed, and then set to NULL. Finally, the pointer that was passed in is freed and then set to NULL.

At the beginning of this function will be where the n_keys stats decrements by 1.

6.2.4 ht_size

purpose: Returns the hash table's size.

All this function needs to do is return the size variable as pointed to by ht.

6.2.5 ht_lookup

purpose: Searches for an entry in the hash table that contains oldspeak.

First an index is created which is equal to the output calling the hash function with the arguments being salt and oldspeak. Then, the index is equal to itself modulo the size of the hash table. Next, the function checks for if the hash table's index is null, and if so, to return null. Then, the function returns the ll_lookup with the arguments of the lists' index and oldspeak.

At the place of where the function returns null is where n_misses will increment by one. Right before the final return statement but outside of the if statement is where the function will increment n_hits by one. In order to get n_examined, the function will be using the ll_stats function from the Linked List file. First, it will create a links and seeks variable that will be passed in through ll_stats. It will also create a new links variable that will be representing the links after ll_lookup is called. The original values are passed in through ll_stats, then ll_lookup is called and ll_stats is called again with the new links variable. n_examined will equal the new links variable minus the old one.

6.2.6 ht_insert

purpose: Inserts the specified oldspeak and its corresponding newspeak translation into the hash table.

First an index is created which is equal to the output calling the hash function with the arguments being salt and oldspeak. Then, the index is equal to itself modulo the size of the hash table. Next, the function checks for if the hash table's list's index is null, and if so, to make that index equivalent to ll_create with the argument false. Then, outside of the if-statement, is a call to ll_insert with the arguments of the list with the index, oldspeak and newspeak.

At the beginning of this function will be where n_keys will be incremented as the number of keys in the hash table are increasing.

6.2.7 ht_count

purpose: Returns the number of non-null linked lists in the hash table.

A count variable is initialized which will act as the counter for the number of non-null linked lists. Then, a for-loop is initialized which iterates through the size of the hash table. Then, an if-statement checks for if the hash table's list at the index of the iterator is not equal to null, and if it is not, then the count variable increments. finally, the function returns the count variable.

6.2.8 ht_print

purpose: A debug function to print out the contents of the hash table.

A for-loop is initialized which iterates through the size of the hash table. Then, there is an if-statement which checks for if the hash table's list at the index of the iterator is not equal to null, and if it is not, then the ll_print function is called with the argument of the list's index.

6.2.9 ht_stats

purpose: It 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 during lookups. It uses the statistics tracked in the hash table to do so.

Each statistic that has been created from the struct has been incremented throughout the file. This function will just set each argument to the associated statistic and will thus be changed if they are incremented.

---

7 Parser

7.1 parser.h

purpose: The header file of parser.c; it allows the main program (banhammer) to access the functions within the parser.c source code. The standard library functions stdio.h and stdbool.h are utilized. stdio.h is a necessary header because one of the returns functions are boolean-types. stdio.h is a necessary header to use the NULL type and other such variable types.

7.2 parser.c

purpose: This function creates will parse out the words of a user, which will be passed in the form of an input stream. The words that they will use are valid words, which can contain contractions and hyphenations. A valid word is any sequence of one or more characters that are part of the word character set. The word character set also includes apostrophes and hyphens. This parsing module will lexically analyze the input stream.

This file uses several standard library functions such as stdint.h, stdlib.h, and ctype.h. stdint.h is necessary in order to use the NULL pointer. stdlib.h is necessary in order to use the malloc and calloc functions. ctype.h is the header that allows the usability of the isalnum() function which can check for if the input is alphanumeric. Finally, the function needs the parser.h header, so that the source code can be properly accessed by other files.

7.2.1 Parser struct

purpose: A way to group several variables (members) into one place.

The parser initializes various variables that will be used throughout the function. Such variables is the file type f, a character type which defines the max parser length, and the line_offset which represents the current offset the function is looking at.

7.2.2 parser_create function

purpose: The constructor for parser

First, the Parser type p will be defined with the malloc() function. Then, the line_offset is initialized to 0 and the file type is defined. Finally, the function calls fgets() to get the first line of the file. The function also returns the Parser type p.

7.2.3 parser_delete function

purpose: The deconstructor function.

All this function need to do is free the pointer that has been passed in and then set it to NULL.

7.2.4 my_strlen function

purpose: To mimic the standard C function strlen.

It first sets a size_t variable val to 0. It keeps on incrementing the val as long as the index of val into the passed in string isn't 0. It then returns val.

7.2.5 my_strncpy function

purpose: To mimic the standard C function strncpy

It first initializes a size_t variable i to 0. Then, it goes through a for loop from i to the size value that has been passed in through the arguments. In the for-loop, it sets the words's index i to the passed in character's index i. Additionally, the character is made lower-case with the function tolower(). Finally, the word character with the index is set to '0' as it will represent the 0 byte at the end of each word.

7.2.6 next_word

purpose: This function will take a pointer to a parser and a buffer for the returned word. It will find the next valid word in the buffer within the parser, and store it in the buffer passed in.

First, two variables are initialized: end and start. These two variables are intended to represent where each word starts and ends in each line of the file. Then, a for-loop is initialized which sets a character variable i to the specific characters in each word. If the line_offset variable equals 0, then this means that the file is either first being read or it is at the beginning of a new line. Then the first line is read. If, when this line is read, that the line equals null and there is a end of line equivalent character, then the function returns false. Otherwise, i is set to the current line offset.

While a newline character is read, then a new line is read with the fgets() function and the line_offset is set to 0. Additionally, i will be set to the new line with the new line offset.

If a invalid character is read (non alphabetical characters, non dashes, non single quotations, and spaces), then the function makes a new character variable j. This character will keep on iterating through the line until a valid character is read. If so, then the function will set j to i.

If a valid character is read then the start variable will be set to the current line offset. Similar to the previous if-statement, a variable j will be set to i and then iterated through the line until an invalid character is read. If so, then the function will set j to i. Additionally, the end variable will be set to the current line offset minus one (since that was the last valid word to be read). If a newline character is read, then the line offset is set to 0. At the end of the if-statement, the function breaks because a total valid word has been read. Below the for-loop, the my_strncpy function is created with the arguments word, current line plus the start variable, and the difference between end and start. If the difference between end and start is greater than 0, then the function returns true, otherwise it returns false.

---

8 Banhammer

8.1 banhammer.c

purpose: To have the user input text using stdin, and then banhammer will filter out their message, and if it detects improper usage, then it will raise a message informing the user on their mal-use.

The file contains several standard library functions, namely stdio.h, stdlib.h and unistd.h. stdio.h. is necessary so that the function will be able to use the printf function which is essential in printing the output messages. stdlib.h is vital in using malloc and calloc. unistd.h is used for the getopt function which will be used to determine the various flags the user will call.

First the main function is called with the arguments argc and arrgv. Then, several variables are initialized. Such variables are the mtf and stats booleans which will become true once the user triggers their flags. Then there is the hash_size and bloom_size default values. Then there are the thoughtcrime and rightspeak booleans which will trigger true once the specified words are read in through stdin. Finally, the opt variable is initialized to 0.

A while loop is initialized which will check for the command line options. If the '-h' flag is used in the command line, then a usage statement will be printed to the stderr file type and will return as 0. If the '-t' flag is used, then the hash_size variable will be set to the user's input and will then break. If the '-f' flag is used, then the bloom_size variable is set to the user's input and then break. If the '-m' flag is used, then the mtf boolean is set to true and will move all of the words to the front of the list and then breaks. If the '-s' flag is used, then the stats boolean is set to true and all of the stats will be printed and then breaked. If an incorrect command line option is used, then the default option is triggered which prints the usage statement and returns a nonzero value.

then, the function reads in a list of badspeak words with fgets(). What this means that a bloom filter and hash table need to be created in order to store the words that are being read from the text. Then, The badspeak file is read with the fopen() function and set to the variable badspeak_file. A character buffer is then created so that the words can be returned in an accessible area. Then, a parser type is created so that the words can be parsed through.

A while loop is initialized which calls the next_word command from parser with the arguments p and badspeak_words and will keep on going until the end of line character is read. In this while loop, each badspeak word is added to the bloom filter with bf_insert and is also added to the hash table with ht_insert.

Next, the function reads in a list of oldspeak and newspeak. This means that the fopen() function is used again to open the newspeak.txt file. Another parser type is created and some more character buffers are created for the oldspeak and newspeak words. Another while loop is initiated which will keep on going for as long as the next_word function returns true. Since the newspeak.txt file will have two words side by side, it can be interred that the first read word will be put into the oldspeak_words buffer and the second one will be put into the newspeak_words one. Thus, the oldspeak_words are put into the bloom filter and then the oldspeak_words and newspeak_words are put into the hash table.

Next, the function will read in words with stdin using the next_word function from the parsing module. First, a buffer called 'words' is initialized. Then, two linked lists are created using mtf as the arguments. The first Linked List type is called 'badspeak' and the next one is called 'oldspeak_newspeak'. Two Bloom Filters are created with similar names. Finally, the parser type is created which will read in the input and takes the argument stdin.

Now, the while-loop is initialized which will keep on going for as long as the next_word output remains true with the arguments input and words. Then, the function checks to see if the words as been added to Bloom Filter with the function bf_probe taking in the arguments bf and words. If it is true, then a node type hash is created equalling the ht_lookup output with the arguments ht and words. If the hash variable is not null, then it will check for two different situations. The first situation is for if the newspeak value is equal to null, in which the thoughtcrime boolean is set to true and the word is inserted into the bloom filter. If the word is not already in the badspeak buffer (meaning that it has not been passed through before), then it will add in the word to the badspeak buffer. The next case is if the newspeak value is not equal to null, in which case the rightspeak boolean is set to true and the word is inserted into the badspeak bloom filter. If the word has been been added to the oldspeak_newspeak buffer, then it will be added into the buffer with the ll_insert function.

Now will be the portion of the function which prints the message or the statistics. If the statistics flag has been set to true, then several statistics will be printed. Since all of the statistics have been collected with various files, all that needs to be done is pass in empty arguments into each statistic function and then print out the output. In order to get ht keys, ht hits, ht misses and ht probes, specific variables will be made to be passed into the ht_stats function and then are printed with the fprintf function (the first argument being stdout). Next, in order to get the bf keys, bf hits, bf misses and bf bits examined, specific variables will be initialized, and then passed into the bf_stats which will be printed using the same method as ht_stats. In order to get bits examined per miss, a numerator variable is bit masked to a double variable type and is set to the bf bits examined minus 5 multiplied by bf hits. Then the numerator is divided by the bf misses. If the denominator is 0, then the stat will print as 0, otherwise it would be set to the divided value. For the false positives statistic, a numerator variable is bit masked to a double variable type and is set to the ht misses. Then, if bf hits equal 0, then the stat will print 0, otherwise it will equal the numerator divided by ht misses. For the average seek length statistic, a denominator variable is bit masked to a double variable type and is set to the sum between the ht hits and the ht misses. If this value equals 0, then the statistic prints 0, otherwise it prints ht probes divided by the denominator. For the bloom filter load statistic, a counter variable is bit masked to the output of bf_count with bf as the argument. This value is divided by the returned output from bf_size with the argument bf.

If the statistic flag is not called, then the function will print various messages. If the thoughtcrime and rightspeak boolean are set to true, the mixspeak_message is printed as well as the badspeak and oldspeak_newspeak Linked List words. If only the thoughtcrime boolean is set to true, then the badspeak_message is printed as well as the badspeak Linked List. If only the rightspeak boolean is set to true, then only the goodspeak_message is printed as well as the oldspeak_newspeak Linked List words are printed.

Finally, all of the created files and modules are deleted in order to prevent any segmentation faults and the function returns 0.