Sean Brandon
spbrando@ucsc.edu
11/16/2020

CSE13s Fall 2020
Assignment 6: Down the Rabbit Hole and Through the Looking Glass: Bloom Filters, Hashing, and the Red Queen's Decrees
Design Document

Introduction:

In this project I use a Bloom filter and hash table to store outlawed words (a hash will be used to store these words in the Bloom filter and hash table) given to me in .txt files. I check the user input against these stored words to determine the severity of their mistakes in their language. First checking the bloom filter (because it is faster) then if I get a match in the Bloom filter I check in the hash table (because sometimes the bloom filter gives false positives). If they used a word that is in the hashtable it has been outlawed. However, depending on the word it might have a translation and only need a warning. If there is no translation though, they will be sent away to the dungeon.

Pre-lab Questions:

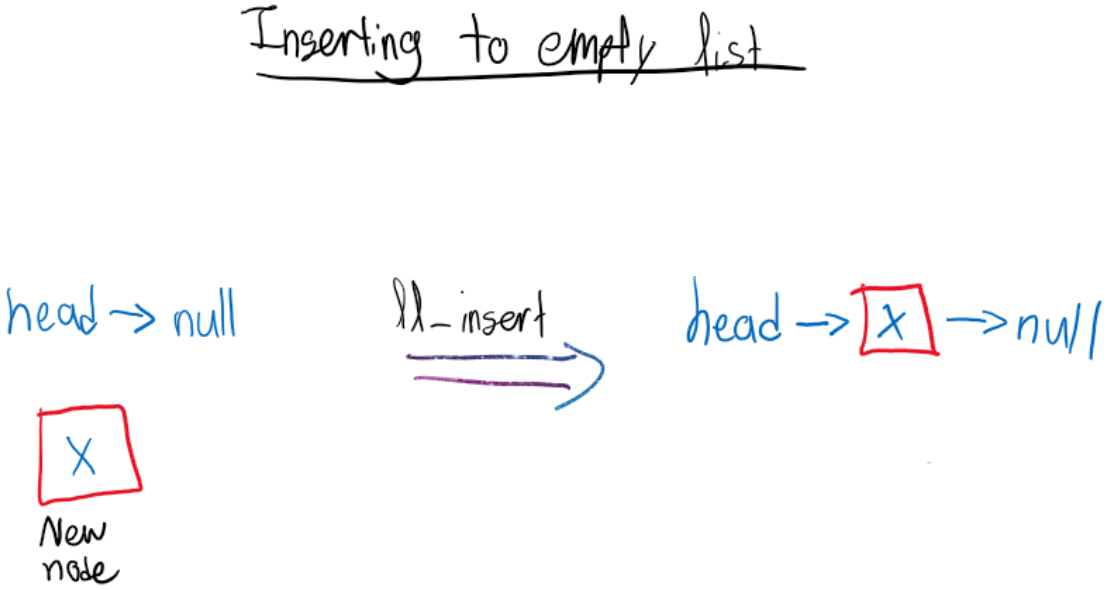
Part 1:

1. Pseudo-code below.
2. The smaller number of bits in a bloom filter, the less space (linearly) it will take up obviously. However it will be more prone to collisions (false positives). The more hash functions done for a bloom filter for a single value the longer it will take (in linear time) to set the all bits, but it will also have less collisions (false positives).

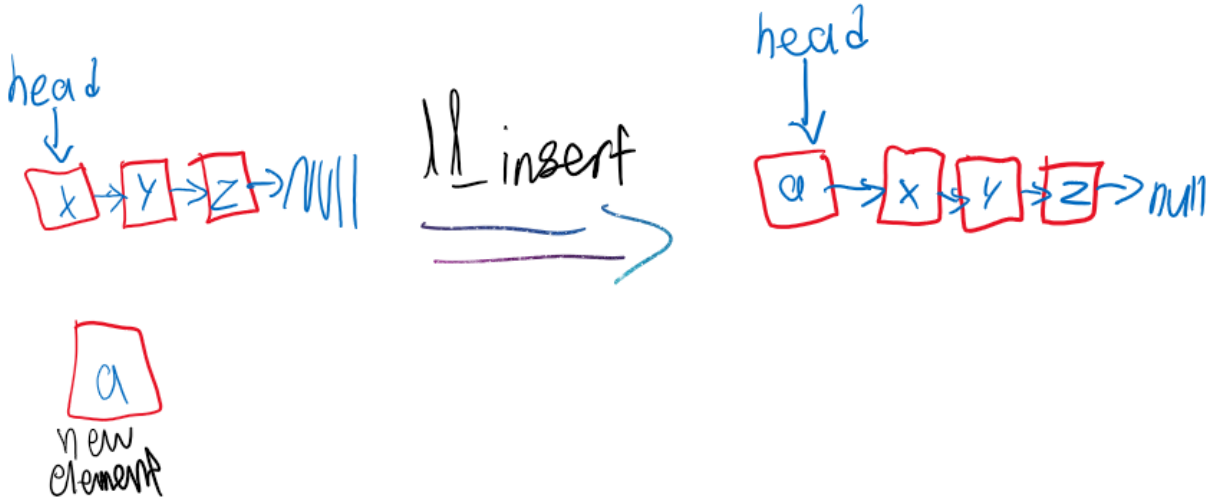
Part 2:

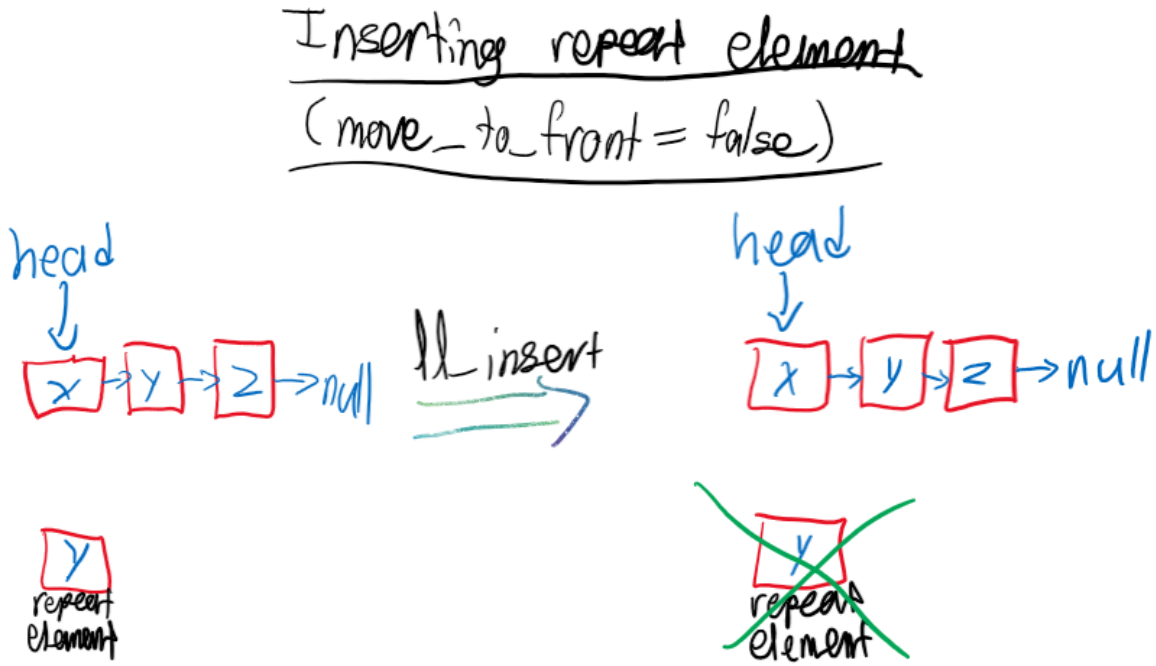
1. Pictures below

a.

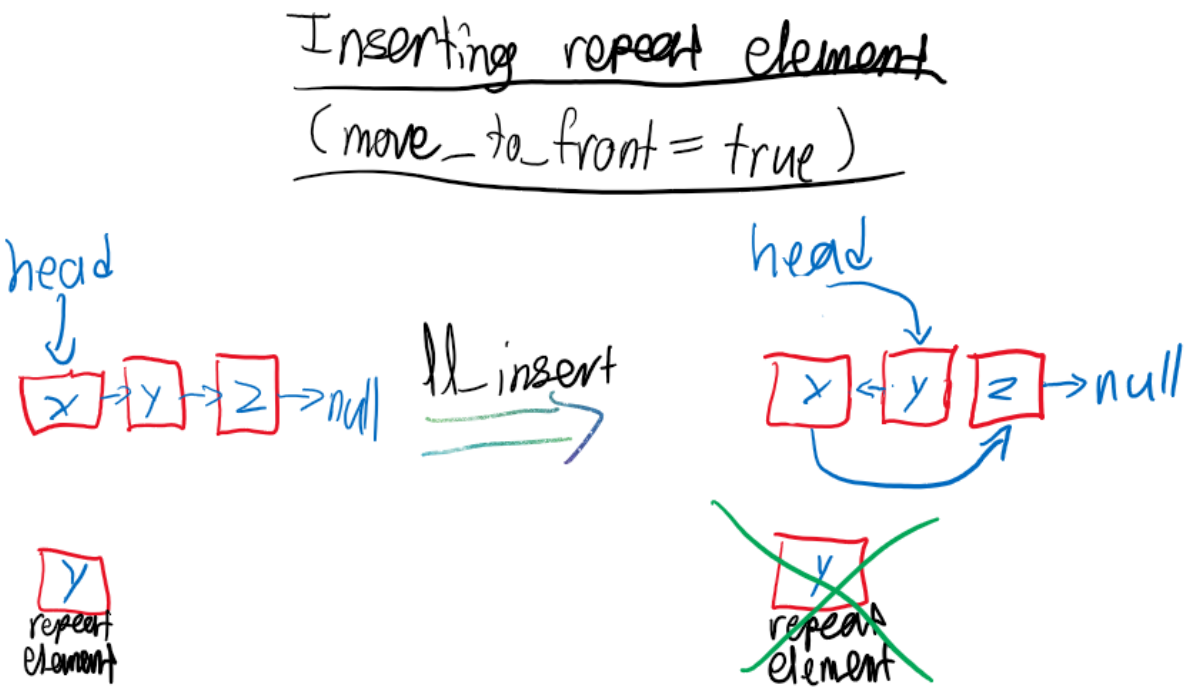

Inserting to empty list
head -> null
ll_insert
head -> X -> null
X
New node

b.


Inserting into filled list
head
X -> Y -> Z -> NULL
ll_insert
head
a -> X -> Y -> Z -> null
a
new element



c.



d.

2. Pseudo-code shown below.

Pseudo-code:

Notes in red

Citations in red and highlighted yellow

bv.c

```
bv_create( bit_len) {
        Malloc memory for bitvector
        Check that malloc worked
        Calloc memory for bitvector array inside of bitvector
        Return bitvector
}

bv_delete( *v) {
        free(memory in v)
        free(v)
}

bv_get_len( *v) {
        Make sure v isnt null
        return v -> length
}

bv_set_bit( *v, i) {
        Sets the bit at index i in v to true/prime/1
        Byte_position |= byte mask regarding bit position
}

bv_clr_bit( *v, i) {
        Make sure v is not null
        Sets the bit at index i in v to false/composite/0
        Byte_position &= byte mask regarding bit position
}
bv_get_bit ( *v, i) {
        Make sure v is not null
        Returns the bit at index i in v
        Return (Byte_position |= mask of byte) shifted right back into the 1s place so its
        either a 0 or 1 returned
}

bv_set_all_bits( *v) {
        Make sure v is not null
        Sets all bits in v to true/prime/1
        Loops through all bytes in bitvector and “or” masks them with 0xff
}
```

bv_create is the constructor function for our bitvector. It allocates memory to for itself and also the int array that is used as our bitvector. It allocates that array based on how many bits we'll need by giving just enough byte size chunks to fit all our data. Returns bitvector pointer.
bv_delete is the destructor function for our bitvector. It frees all the memory that has been allocated in bv_create. Returns void.
bv_get_len just returns the length of our bitvector (how many bits we use). Returns uint32_t.
bv_set_bit sets the bit at the given index to 1 (AKA true). It does this using a bitmask on the correct byte chunk. Returns void.
bv_clr_bit clears the bit at a given index to 0 (AKA false). It does this using a bitmask on the correct byte chunk. Returns void.
bv_get_bit returns value of the bit (either 1 or 0) at the given index position. It does this using a bitmask on the correct byte chunk. Returns uint8_t.
set_all_bits sets all bits in a bit vector to 0. It does this by using a bitmask with value of 0xff on each byte chunk. Returns void.

I use this file and its functions to work with my Bloom filter. In fact, a Bloom filter is basically just a bit vector anyways so these functions are very important for working with, creating, and deleting Bloom filters and their data points.

bf.c

bf create given in lab pdf

```
bf_create(size) {
        bf = malloc(BloomFilter)
        If b(f) {
                Set both elements of primary, secondary and tertiary arrays to the salt numbers
                  given in the lab pdf
                bf->filter = bv_create(size)
                return bf
        }
        return null
}

bf_delete(bf) {
        Make sure bf isn't null
        Free the bit vector “filter” using the bv_delete method then free the bloom filter
        bv_delete(bf->filter)
        Free bf
        return
}

bf_insert(bf, key) {
        Make sure bf isn't null
        Modulo the hash value by size to make sure the index isn’t beyond the bloom filters size
        Done 3 times for each salt to avoid collisions
```

```
        Use bv_set_bit to set the the primary, secondary, and tertiary hash indexes as 1
        return
}

bf_probe(bf, key) {
        Make sure bf isn't null
        Modulo the hash value by size to make sure the index isn’t beyond the bloom filters size
        Use bv_get_bit to determine if the primary, secondary, and tertiary hash indexes are all 1
        If (primary, secondary, tertiary hash indexes are all true) {
                Return true
        }
        else {
                Return false
        }
}
```

This following function was not given in the original hash.h file, I added it to make finding statistics easier.

```
bf_set_count (bf) {
        For loop through every bit in bit vector of bf {
                If (bv_get_bit at that index returns true {
                        Increment count
                }
        }
        Return count
}
```

bf_create was given in the lab pdf but basically it just creates a bloom filter to be used by setting the primary, secondary, tertiary salts to the numbers provided and then it creates a bit vector to store the elements of the filter. Returns BloomFilter.

bf_delete frees the memory of the bit vector and then frees the bloom filter that was created by bf_create. Returns void.

bf_insert takes a string of chars, hashes that string and modulus that hash by the size of the bloom filter to make sure it goes into the bloom filter, then inserts a 1 (AKA a true) into that position to indicate that that hashed word is in the filter. It does this 3 times with 3 differently salted hashes to reduce the number of possible collisions. Returns void.

bf_probe determines whether a string of characters is in the bloom filter by hashing the string with the same salts that were used in bf_insert and checks if those indices are marked as true. Because of collisions, all 3 indices must be marked as true to return true, otherwise it is definitely false. Use bv_get_bit. Returns bool.

I use this file and its functions to keep track of the words that are “bad”. This a quick way to check or “filter” out good words that the user may have used because while a bloom filter cannot give an absolutely correct answer about whether or not a value may be in it (because of false positives with collisions), it CAN give an absolute answer about if a word is NOT in it. I use bf.c to maintain, create, delete, and check the information in my bloom filter.

bf_set_count is a function that I created to make finding statistics easier. Returns the number of bits set to 1 in a bloom filter.

ll.c

bool move_to_front extern bool set in the getopt loop. If true, move queried node to front of LL

```
ll_node_create( gs) {
        malloc ListNode
        Check malloc
        ListNode->gs = gs
        ListNode->next = null
        Return ListNode
}

ll_node_delete( *n) {
        Make sure n isnt null
        hs_delete(n->gs)
        Free the rest of the node
        return
}

ll_delete( *head) {
        Make sure head isn't null
        For each node in linked list starting at head {
                next = current node -> next
                ll_node_delete(current node)
                current node = next
        }
        return
}

ll_insert( **head, *gs) {
        node = ll_lookup(head, gs->oldspeak)        Making sure there won't be duplicates
        if(node != null) {
                hs_delete(gs)
                Return head
        }
        If it won't be a duplicate…
        Create node with ll_node_create and make it point to the current head
        Return node
}

ll_lookup( **head, *key) {
```

```
        Make sure head isn't null
        Prev = null     Make sure to track previous node incase move_to_front is true
        For every node in the linked list {
                If current node's old speak == key {
                        If move_to_front AND prev != null {
                                Move found node to head before returning it if move_to_front is on
                                AND if current node is not already the head
                                Prev node now points to what current node is pointing to
                                current node now points to head
                                Head now equals current node
                        }
                        Return the found node
                        Return current node
                }
                Move through list
                Prev node = current node
                current node = current node's next node
        }
        Return null if key could not be found
        Return null
}
```

ll_node_create is the constructor function for a linked list node. It allocates memory, points the node to null (because it's the first and last node in the linked list at the moment), and saves the given hatterspeak struct to the node. Returns a ListNode.

ll_node_delete is the destructor for ListNode. This uses hs_delete to free the hatterspeak struct then frees the rest of the node. Returns void.

ll_delete is a function that deletes and frees the entirety of a linked list by using ll_node_delete in a for loop. Returns void.

ll_insert creates and inserts a list node into a linked list at the head. If the node is already present though, it just frees the hatterspeak struct that was input. Returns head.

ll_lookup searches for a specific key in a linked list. If move_to_front is on and we find the key in a node we need to move that node to the head and connect the previous node to the found nodes next node to avoid breaking the link (which is why we keep track of previous while searching). Returns the list node if found and NULL otherwise.

ll.c is used to create, delete, search for, and move nodes in a linked list. I use these nodes and linked lists later in my hash table. My linked lists are singly linked and store HatterSpeak structs.

hash.c

Ht_create function given in Lab pdf

```
ht_create(length) {
        Malloc HashTable
        Check that HashTable is not null
```

```
        Assign HashTable salt array numbers
        HashTable length = length
        HashTable heads = calloc array of list nodes
        Return HashTable
}

ht_delete(*ht) {
        Make sure ht isn't null
        For i in length of ht {          Loop through hashtable indices
                If ht->heads[i] isn't null {      If the index has a linked list in it
                        Ll_delete that head          ll_delete to free that whole linked list
                }
        }
        Free the rest of the hash table
        Free ht->heads
        Free ht
        return
}

ht_count(h) {
        Make sure h is not null
        Count = 0
        For i in h length {      Loop through hash table indices
                If h[i].heads != null {   If not null, then there is a linked list/node at this index
                        count++
                }
        }
        Return count
}

ht_lookup(ht, key) {
        Make sure ht is not null
        Index = hash(salt, key) % length of hash table      find the correct linked list index in ht
        Return ll_lookup(ht heads[index], key      find out if the key is in that linked list
}

ht_insert(ht, gs) {
        Make sure ht is not null
        Index = hash (salt, gs->oldspeak) % ht length      find the correct linked list index in ht
        Ht heads[index] = ll_insert(&ht heads, gs)    insert linked list into correct index in ht
}
```

This following function was not given in the original hash.h file, I added it to make finding statistics easier.

```
ht_node_count(ht) {
        For loop through each index in the hash table {
                If there is a linked list at that index {
                        For loop through each node in that linked list {
                                Increment count
                                Move to the next node
                        }
                }
        }
        Return count
}
```

ht_create is our constructor for a hash table. This is basically just an array of linked list heads and various indices. Returns HashTable.

ht_delete is our destructor for a hash table. This uses ll_delete to free each node in the hash table and then frees the actual hash table. Returns void.

ht_count returns the number of entries in the hash table. H is the hash table.

ht_lookup searches a HashTable for a key. Returns the ListNode if found and returns NULL otherwise.

ht_insert First creates a new ListNode from HatterSpeak. The created ListNode is then inserted into a HashTable. Returns void.

ht_node_count is a function that I created to make finding statistics easier. Returns the number of nodes in a hash table.

hash.c is my file for hash tables. The functions in hash.c allow me to efficiently keep track of the values in my hash table. If a user input word gets passed my bloom filter then the next stage is to check if it exists in one of the HatterSpeak structs that are stored in list nodes which is stored in a hash table. If the word is in my hash table then it is definitely a bad word. Whether or not it is forbidden or not depends on if it has a translation, but that will be discussed later. Hash.c allows me to create and delete hash tables and manage the information stored inside of them.

hatspeak.c

Received help from a TA in a section for this function. I believe it was in Maxwell L's section

```
hs_create(oldspeak, hatterspeak) {
        g = malloc HatterSpeak
        Check that malloc worked
        Copy oldspeak input into g->oldspeak
        g->oldspeak = char malloc ((strlen of oldspeak) +1)
        Check malloc
        strcpy(g->oldspeak, oldspeak)

        If a hatterspeak translation is available
        if(hattespeak) {
```

```
            Copy hatterspeakinput into g->hatterspeak
            g->hatterspeak = char malloc ((strlen of hatterspeak) +1)
            Check malloc
            strcpy(g->hatterspeak, hatterspeak)
        }
        Else {
            g->hatterspeak = NULL
        }

        Return g
}

hs_delete(g) {
        Make sure g is not null
        free(g oldspeak)
        Hatterspeak translation may be null so check before freeing
        If (g hatterspeak != null) {
            free(g hatterspeak)
        }
        return
}
```

hs_create constructor for a HatterSpeak struct. Returns HatterSpeak
hs_delete destructor for a HatterSpeak struct. Returns void.

I use this file to create and delete my HatterSpeak structs which are contained in ListNodes. Not all oldspeak words will have hatterspeak translations. In that case the bad word is also a forbidden word. HatterSpeak struct allows me to store that information and hatspeak.c functions allow me to create and delete HatterSpeak structs.

Speck.c

Speck.c is a file that was given to us in its entirety in the assignment pdf and is used multiple times throughout this project. The function used in this file is hash. What hash does is it takes in an input value and a salt array with 2 numbers in it (salt numbers were given in the assignment pdf as well) and uses an add-rotate-xor cipher and the salt numbers to return a numeric value to be used as a key in a hash table. I used the hash function when assigning hash table indices to list nodes and when I needed to lookup where a list node would be in a hash table.

parser.c

parser.h allows me to pick individual valid words from an input stream. I decide what is valid by creating a regular expression using regex. The function used in parser.c is next_word which uses this regular expression to find words that match its parameters and returns individual words from the input stream. I use this when taking in the user input before checking if its a bad word or not. Also, clear_words function is used at the end of my program to free any memory used by the module.

hatterspeak.c

includes
Externs move_to_front, seeks, nodes_trav

main {
    Set default values
    Getopt while loop{
        Switch case statements for getopt flags
    }

    bf = bf_create(bf_size)
    ht = ht_create(ht_size)

    Fill hash table and Bloom filter with hatterspeak.txt values
    while (fscanf (hatterspeak.txt, %s %s, word1, word2) != EOF){
        bf_insert(word1)
        ht_insert(hs_create(word1, word2))
    }

    Fill hash table and Bloom filter with oldspeak.txt values
    while (fscanf (oldspeak.txt, %s, word1) != EOF){
        bf_insert(word1)
        ht_insert(hs_create(word1, NULL))
    }

    I received help creating my regular expression in Oly's TA section
    Set regular expression filter (uppercase letters, lowercase letters, " ' ", and "-")

    forbidden = linked list = NULL
    non_forbiden linked list = NULL

    Loop through each word user input
    While (my_word = next word(stdin, regular expression)) {
        Set my_word to all lowercase letters
        I found the function "tolower" here:
        https://www.geeksforgeeks.org/tolower-function-in-c/#:~:text=If%20the%20character%20passed%20is,to%20be%20converted%20to%20lowercase
        for(i in range(my_word)) {
            my_word[i] = tolower(my_word[i])
        }
        Check if the word is in the Bloom filter
        If (bf_probe(bf, my_word)) {
            Make sure it wasn't a false positive by checking in the hash table

```
                If (ht_lookup(ht, my_word != NULL)) {
                        Check if its translatable or forbidden
                        If (ht_lookup(ht, my_word)->hatterspeak != NULL) {
                                Insert found hatterspeak struct into non_forbiden linked list
                                Non_forbiden = ll_insert(hs_create using values found in
                                                                                ht_lookup)
                        }
                        Else {
                                Insert found hatterseak struct into forbidden linked list
                                Forbidden = ll_insert(hs_create using values found in
                                                                                ht_lookup)
                        }
                }
        }
}

If (stats flag is set) { Print only stats if stats flag is set
        printf(Seeks: seeks)
        printf(Average seek length: nodes_trav / seeks)
        printf(Average linked list length: ht_node_count(ht) / ht_size)
        printf(Hash table load: 100 * (ht_count(ht) / ht_size))
        printf(Bloom filter load: 100 * (bf_set_count(bf) / bf_size))
}
Else { If stats flag isn't set print the correct message from the censor
        If (forbidden) { If there are any forbidden words, always print dungeon message
                printf( the "you're going to the dungeon message")
                For every node in the forbidden list, print the forbidden word
                for(current node = forbidden; current node != null; ) {
                        printf(current node -> oldspeak)
                        Current node = current node -> next
                }
        }

        If (non_forbiden != null) { If there are any translatable bad words used
                If (forbidden == null) { If ONLY translatable bad words were used
                        Print warning message
                        printf( warning message )
                }
                Else { If both forbidden bad words AND translatable bad words were used
                        Print continued dungeon message
                        printf( appropriate hatterspeak translations: )
                }
                For every word in the non_forbided list, print the bad word and its
                translation
```

```
                    for(current node = forbidden; current node != null; ) {
                        printf(current node -> oldspeak  ---> current nod -> hatterspeak)
                        Current node = current node -> next
                    }
                }
            }
            Free everything
            ll_delete(forbidden)
            ll_delete(non_forbidden)
            regfree(word_regex)
            clear_words()
            bf_delete(bf)
            ht_delete(ht)

            Return 0
}
```

Design Process:

I already had a working version of bv.c and bv.h from a previous assignment so I just used the files from that previous assignment in this one. That was step one. The next step was writing my bloom filter functions. This wasn't very difficult having realized that a bloom filter was basically just a bit vector but with hashes and multiple indices per key. Once that was done I proceeded to test my bv.c and bf.c functions and once I was confident that there were no issues in either I moved onto implementing a linked list.

For my ll.c file I had to figure out how to implement the necessary functions to make a linked list work. This wasn't too tricky though since I've had experience with linked lists in the past but I will admit that the pointers and double pointers caused some confusion. About halfway through writing my ll.c file I decided to make my hatspeak.c file to hold my HatterSpeak struct, constructor and destructor. I wish I had done this first though because once I had the HatterSpeak struct written and fully understood it, finishing ll.c became much easier. Once I finished ll.c and hatspeak.c I decided to stop there and test that both were working correctly. When I was confident that they were, I moved on to implementing my hash table functions in hash.c.

Starting hash.c was the most difficult part because I wasn't totally sure what a hash table in this project was supposed to look like. I attended a TA section to receive some clarification and once I realized it was basically just an array with certain indices containing a linked list depending on the hash, it became easier to write each necessary function. Once I was finished with this I took time to test and make sure that there were no issues with my code. It was at this point that I decided that I had the foundational code of this project completed and went on to write the getopt function and file input functionality so that I could fill my bloom filter and hash table with actual file values from the assignment.

The getopt part of my main function was easy, standard stuff. Filling my hash table and Boom filter was also no problem and passed all the tests I gave it the first time around. From there I went on to trying to figure out how to take in user input and put it through a regular expression check.

This was a little difficult because it was the first time in C that I had done anything like this. After asking the professor a few questions in lecture about it though and after going to a TA section it all became plain and simple to me how to do this. After making sure it worked I moved onto actually using the input to check for bad and forbidden words.

This is really the culmination of all the other code that had been written up to this point. I wrote all my checks and tested that they worked. Luckily since I had tested everything before, this all worked just fine. I decided to store all the bad words in two separate linked lists to keep track of them, non_forbidded (words with a hatterspeak translation) forbidden (words without a hatterspeak translation). After making sure it all worked I went on to write the outputs.

This is where I ran into a bit of trouble. The standard output was no problem whatsoever, just printing some paragraphs and all the words in my forbidden and/or non_forbidden linked lists. However, I hadn't implemented anything to keep track of the stats yet. In the end it was just a few pretty simple counters but I wasn't totally familiar with how to use extern so I had to get comfortable with that. I also, at this point, decided to write two of my own functions to help me find certain stats. These functions are bf_set_bits, which finds the number of bits set to true in a bloom filter, and ht_node_count, which finds the number of nodes in a hash table. Once those were written and I had figured out how to use extern implementing stats was no problem.

My final hurdle to clear was the move_to_front rule. I had saved this until last at the advice of TA's in sections who said it would just be a simple if statement tacked onto ll_lookup, which it was. However unfortunately for me I made a small mistake and forgot to set head = current node once I had rewired the linked list. This caused a segmentation fault in my program which took me a good 2 hours to figure out. This issue did however force me to run through every single step my program took which I think is beneficial.

Finally, I wrote the last part of my program. The freeing and deletion of all data structures and of all data.

If I could change something about this lab for the future I would recommend changing the file names and how the lab is explained. It is clearly overly complex to have the executable be named hatterspeak, the main file to be called hatterspeak.c, to have a struct name HatterSpeak, and for one of the variables in that struct to be named hatterspeak. Change the executable and the main file to censor, and the variable to translation. This caused unnecessary confusion for me when trying to figure out what I was supposed to be doing. Also, it seems silly to name a file hash.c when a different file (speck.c) contains a necessary hash function. hash.c should be renamed to ht.c, in the same way that the file for Bloom filter is named bf.c.