Specification

Assignment 7. The Great Firewall of Santa Cruz, Bloom Filters, Linked Lists, Hash Tables

Files needed:

Makefile: a make file to easily run the program by prompting make into the Linux terminal on our Ubuntu virtual machine.

README.md: describes how the program works and runs in addition to how to use the makefile

banhammer.c: this will be treated as the main file for this assignment. Everything will be combined and brought together for the totality of this assignment here

messages.h: Defines the mixspeak, badspeak, and goodspeak messages that are used in banhammer.c

speck.h: Defines the interface for the hash function using the SPECK cipher. Do not modify this.

speck.c: Contains the implementation of the hash function using the SPECK cipher. Do not modify this.

ht.h: Defines the interface for the hash table ADT. Do not modify this.

ht.c: Contains the implementation of the hash table ADT.

ll.h: Defines the interface for the linked list ADT. Do not modify this.

ll.c: Contains the implementation of the linked list ADT.

node.h: Defines the interface for the node ADT. Do not modify this.

node.c: Contains the implementation of the node ADT.

bf.h: Defines the interface for the Bloom filter ADT. Do not modify this.

bf.c: Contains the implementation of the Bloom filter ADT.

bv.h: Defines the interface for the bit vector ADT. Do not modify this.

bv.c: Contains the implementation of the bit vector ADT.

parser.h: Defines the interface for the regex parsing module. Do not modify this.

parser.c: Contains the implementation of the regex parsing module.

WRITEUP.pdf: this document will analyze the statistics of this program and we will show thought and some analysis About the overall program. We will use the -s command line option to have some numbers to work with

Program Description:

We are a member of the Glorious People's Republic of Santa Cruz (GPRSC), and the citizens of Santa Cruz are convoluting their minds with badspeak. Badspeak are forbidden words that we must filter out in a firewall to get rid of. We also have what's called oldspeak and newspeak and these are words that are not forbidden but words that are outdated so we need to update them into our newer versions. To do this we must understand what a firewall is and how it is built using hash tables, linked lists and bloom filters. And two great examples to understand the assignment in better context is the Chinese government and how they have a firewall to filter out websites that are unwanted. For example most social media platforms are banned in China including those of *Facebook.com, Instagram.com, Twitter.com, etc.* So similar to that we are trying to filter out unwanted words that we refer to as badspeak so we can control what is being displayed on standard output. Another way to understand it in real life is to think about a bar (credit to Eugene's section). In the bar the bloom filter is equivalent to that of the bouncer and the hash table is equivalent to the bartender. You must first bypass the bouncer to get access to the hash values. This assignment introduces a wide scope of new ADT's but overall sounds pretty cool. I mean after all this assignment was credited to the *Washington Post*. It is also the

last of its kind for CSE13S and I'm excited to begin my last hurdle of this class. I cannot wait to get started!

Pseudocode:
Banhammer.c

```
main()
//Getopt stuff

Create hash table
Create bloom filter

Open and read badspeak.txt (1 word)
Open and read newspeak.txt (2 words)

Make linked list to hold bad words
Make linked list to hold old words with translations

Compile a regular expression using regcomp

Loop through each word
   Make them lower case
   If word is not in bf or ht
      Its not an issue
   Else
      If its in badspeak.txt
         Shame on you
      Else
         Just don't say that next time

If you only have bad words
   Go to joycamp
If you have only old words
   Fix them and don't get them wrong next time
If you have both
   Give them warning about joy camp do better next time

Close files
Free memory
```

Banhammer.c is the main module where all the magic happens. As expected we will go through the get opt stuff to establish all the command line arguments in this file. Then we will simply follow all the instructions of the pseudocode above and this will determine whether our citizens will have to go to joycamp, fix their errors and thoughts, or be warned about joycamp and think about their errors and thoughts.

Bloomfilter.c

```
//Struct for Bloom filters
  struct  BloomFilter {
    uint64_t  primary [2];    //  Primary  hash  function  salt.
    uint64_t  secondary [2]; //  Secondary  hash  function  salt.
    uint64_t  tertiary [2];   //  Tertiary  hash  function  salt.
    BitVector *filter;
 };

Bf_create{
  Allocate memory with malloc
  Salt 1 - Grimms fairy tale
  Salt 2 - The adventures of Sherlock Holmes
  Salt 3 - The strange case of Dr. Jekyll and Mr. Hyde
}

bf_Delete{
  //Deletes bloom filter by freeing memory
}

Bf_size{
  //Returns the size of the bloom filter
}
Bf_insert{
  //Takes oldspeak and hashes them into the bloom filter using the
3     salts meaning it will occupy a total of 3 bits
}

Bf_probe{
  //Checks to see if the 3 salts are hashed into the 3 indices. If
they are the oldspeak is likely in there but if it's not we know it
is definitely not in there
}

Bf_count{
  //returns the number of set bits in the bloom filter. Easily can
be tracked with the help of bv
}

Bf_print{
  //prints the current bloom filter
}
```

A bloom filter is a brand new ADT that is introduced to us in this lab. This ADT is used simply for its performance to see if a specific element is in a set. As we spoke about earlier the bloom filter is equivalent to the bouncer at a bar. If the person is not 21 years of age or above we know automatically that the person is not qualified to enter. Similarly we will be checking through the bloom filter to see if all the indices of all 3 salts have been marked in order to believe that perhaps the oldspeak has been

included in the bloom filter. In bloom filters you cannot guarantee that something is in, but you can assume it is probably there if the 3 bits that the 3 salts have marked are indeed marked. However if the three indices are not marked you will know for certain that the element or the badspeak is in fact not in the bloom filter. To go more in depth about the maybe, there are times where the indices might overlap in what is called hashing collision but it tends to happen occasionally. Therefore it is impossible to say for sure whether it is in the bloom filter.

Speck.c

The SPECK block cipher was a provided module that is a functional hash function. SPECK as introduced in the assignment document was a cipher released by the NSA (National Security agency) and it was used specifically for software. The SPECK is an add-rotate-xor cipher. The reason we want to use the SPECK is because when we hash something we need to give it some unique output, and the SPECK takes in some random input and performs its job and gives you a random output. If you know the decryption pattern it will be easy to cipher but the point of this encryption technique is to generate some random output perfect for our hash table. All this talk about encryption and decryption leads to a topic known as cryptography and it is a very cool branch of computer science/ mathematics which will apply to fields such as cybersecurity. Fun fact that the Enigma Machine is responsible for ciphering messages from Nazi germany and mathematician Alan Turing is a big figure in cryptography today.

Bv.c

```
struct  BitVector {
      uint32_t  length; //  Length  in bits.
      uint8_t *vector; // Array of bytes.
};
BitVector *bv_create(uint32_t length)
 Create ADT by allocating memory
 Need to calloc the vector bc they need to all be zeros if not replac
}

Void bv_delete(BitVector **v){
  Delete all bit vectors
}

Uint32_t bv_length(bitVector *v){
Return length
}

Void bv_set_bit(BitVector *v, uint32_t i)
 Remember how we calloced. We set specific bit to 1’s
}

void_bv_clr_bit(BitVector *v, uint32_t i){
 Clears if there is a 1 in the ith bit in bit vector
}
```

```
Void bv_xor_bit(BitVector *v, uint32_t i, uint8_t bit){
 Xors using a modulus 2 of a specified bit to check for parity
}

Void bv_print(Bitvector *v){
 Will print bit vectors. Debugging purposes
}
```

The bit vector assignment is important because the linked list, hash tables, and bloom filters rely on checking if a particular bit has been set. A Lot of the ideas from this adt were already pre introduced in assignment 5 and we are basically using the same concept to apply them to a different set of ADT's. We once again have its main functions set, clear, and get bit and I am certain that these functions will be utilized quite a bit in the implementations of the hash tables, bloom filters, and linked lists.

Ht.c

```
//struct definition for the hash table
 struct  HashTable {
    uint64_t  salt [2];
    uint32_t  size;
    bool  mtf;
    LinkedList  **lists;
 };

Ht_create{
  Allocate memory
  More salts [0][1] - Leviathan
  Initialize all variables
}

Ht_delete{
  Free memory of hash table
}

Ht_size{
  Returns the size of the hash table
}


Ht_lookup{
  Look up whether a certain oldspeak value is somewhere in the ht.
}

Ht_insert{
  Inserts oldspeak and newspeak into a linked list and will be hashed
```

```
  And be put into the hash table
}

Ht_count{
  Will return the value of non-NULL linked lists inside the
  Hash table
}

Ht_print{
  Prints out the hash table
}
```

The hash table is the second new ADT that is introduced in this assignment. The hash tables will be using the SPECK hash function to properly hash values into random outputs and the hash table will also utilize a linked list to avoid hash collision. The hash table will once again take in a salt and utilize a mtf or move-to-front technique which is one of the command line options for this assignment. The meaning of mtf is if two arbitrary oldspeak words have the same hash value then we will add the hashed word at the front in between the head, tail, and the previous value. As explicitly stated it will be moved to the front if prompted into the command line but should be defaulted to come after in all cases.

ll.c

```
//Linked lists
//will need Nodes

//struct definition for Node:
  struct  Node {
    char *oldspeak;
    char *newspeak;
    Node *next;
    Node *prev;
 };

Node_create{
  Create a node that will be included in the hash table
}

Node_delete{
  Frees memory of the node
}

Node_print{
  Prints out the node which is a word that will be in the linked list
  Waiting to be hashed into the hash table
}
```

```
//linked list struct definition:
  struct  LinkedList {
    uint32_t  length;
    Node *head; // Head  sentinel  node.
    Node *tail; // Tail  sentinel  node.
    bool  mtf;
 };

Ll_create{
  Allocated memory for linked list
  Initialize all variables
}

ll_Delete{
  Frees memory of the linked list ADT
}

Ll_length{
  Returns the length of the linked lists which is the same number
as   nodes in the linked list
}

Ll_lookup{
  Lookup a specific index in the linked list to see if it's there.
  It will return the oldspeak word and if not found will return
  Null pointer
}

Ll_insert{
  Inserts an oldspeak or newspeak word into the linked list.
  Does this by creating a node and putting it in the correct
  index
}

Ll_print{
  Prints the linked list
}
```

At last we are at the final brand new ADT that we are introduced to. A linked list is a sequence of nodes that resemble a sequence. The type of linked list we are implementing I believe is a doubly linked list, because we want to point to the next node as well as its previous node. In a doubly linked list it also contains what is called one data field which just refers to the node. So in a sense the linked list is almost like two ADT's in one because we have to worry about both the Node part of it and the linked list itself. All the functions present should be pretty straightforward and it is important and well linked with the previous two ADT's disclosed in this DESIGN document. Those ADT's were the bloom filter, and the hash table. In a sense the linked lists are what's needed for the hash table and the hash table is what's needed for the bloom filter to actually work. They are all working together in this lab and it is crucial to make sure each works independently before testing the next ADT.

Parser.c

Last of all we have the parser module which is going to check every single word that is read in using scanf() to see if it is a valid word. I believe for this assignment any letters, numbers, contractions, and underscores are valid. We have to make sure these conditions are checked using what are called regular expressions for regex. Regex is primarily used for pattern matching and in this case it will check to see if a word is valid. If a word is valid we will check to see if it is a badspeak, or oldspeak and if oldspeak it will advise you to send out the proper newspeak. But this module is primarily used to check if a word is valid. We are parsing words.

Some thoughts:

It feels weird writing my last design document for CSE 13S. I just wanted to reflect on the whole process and what I have learned from writing multiple DESIGN.pdf. Overall the Design documents have been nothing but helpful to help us map out our ideas and write out what we think the assignment is supposed to be. Though my pseudocode has been the one thing I did not nail on the first try, I do want to point out how much clearer my understanding of each assignment has been thanks to these design documents. Not only are they useful to me but when I decide to put these assignments on Github, there will be an explanation of all the functions and parts of the program before they begin looking at the source code itself. Oftentimes looking at code alone is not the easiest to understand although I do judiciously comment on my code, but having a design document, a README.md, and a writeup we can show the pre-process, the post-process, and how to instruct the user to the programs functionality. Though writing these 7-8 page Design Documents in addition to the writeups and README have been grueling at times, I will say I have become a more organized and better computer scientist. I will also say that although these assignments are hard as hell, I have come to appreciate what I have learned through my nearly 10 weeks into the quarter. Overall I am happy with the result and how I faced many trials and tribulations to make me understand things better. I feel so many times that the assignments are too easy and that I want some degree of challenge and I got that up front in this class. And although transitioning from Python to C sucked, I believe this class has really helped me to figure out how to go from an object oriented language to a procedural oriented language. I also think Python is too easy to pick up sometimes and that you don't truly understand the pain of coding until you have to begin worrying about memory allocation, having variable types, and a whole lot more less tools than that of Python. But I am glad I got to learn so much from a class and look forward to all my future classes in Computer Science. Although I won't get the best grade, this was indeed my favorite CSE class I have taken so far in college. Though hard, it was all worth it in the end. Simon Lee is now signing off and never doing a design document for this class again >o<