Assignment 7
The Great Firewall of Santa Cruz:
Bloom Filters, Linked Lists, Binary Trees and Hash Tables

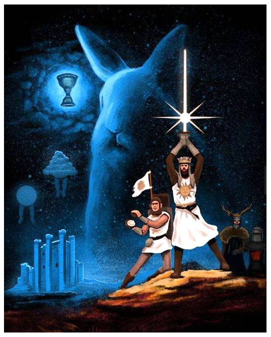
Prof. Darrell Long
CSE 13S – Fall 2021

First `DESIGN.pdf` draft due: November 25th at 11:59 pm PST
Assignment due: December 3rd at 11:59 pm PST

1 Introduction

Of all tyrannies, a tyranny sincerely exercised for the good of its victims may be the most oppressive. It would be better to live under robber barons than under omnipotent moral busybodies. The robber baron's cruelty may sometimes sleep, his cupidity may at some point be satiated; but those who torment us for our own good will torment us without end for they do so with the approval of their own conscience.

—C. S. Lewis



You have been selected through thoroughly democratic processes (and the machinations of your friend and hero Ernst Blofeld) to be the Dear and Beloved Leader of the Glorious People's Republic of Santa Cruz following the failure of the short-lived anarcho-syndicalist commune, where each person in turn acted as a form of executive officer for the week but all the decisions of that officer have to be ratified at a special bi-weekly meeting by a simple majority in the case of purely internal affairs, but by a two-thirds majority in the case of more major decisions. Where it was understood that strange women lying in ponds distributing swords is no basis for a system of government. Supreme executive power derives from a mandate from the masses, not from some farcical aquatic ceremony. It was a very silly place, and so with Herr Blofeld's assistance you are now the leader of your people.

In order to promote virtue and prevent vice, and to preserve social cohesion and discourage unrest, you have decided that the Internet content must be filtered so that your beloved children are not corrupted through the use of unfortunate, hurtful, offensive, and far too descriptive language.

You are the giver of all that your creatures love: A full belly twice a day, and clean straw to roll on. You bear such imposing titles as *Eternal General Secretary*, *Supreme Commander of the People's Surfing Commission, Father of all animals, Terror of mankind, Protector of the sheep-fold, Ducklings' friend,* and *Friend of the featherless.* You know, above all, that all citizens of your little republic are equal but some are more equal than others. And so, your happy little bund sets itself to solving the problem of offensive speech. What is offensive speech? You can't define it, but as Justice Potter Stewart once said, "I know it when I see it."

2 Bloom Filters

The Ministry of Peace concerns itself with war, the Ministry of Truth with lies, the Ministry of Love with torture and the Ministry of Plenty with starvation. These contradictions are not accidental, nor do they result from from ordinary hypocrisy: they are deliberate exercises in ***doublethink****.*

—George Orwell, 1984

The Internet is very large, very fast, and full of *badthink*. The masses spend their days sending each other cat videos and communicating through their beloved social media platforms: Facebook, Instagram, Snapchat, Twitter, Discord and worst of all, TikTok. Some of these will ban users for *badthink* that is contrary to their political views, but that is not enough for you. To your dismay, you find that a portion of the masses frequently use words you deem improper: *oldspeak*. You decide, as the newly elected Dear and Beloved Leader of the Glorious People's Republic of Santa Cruz (GPRSC), that a more neutral *newspeak* is required to keep your citizens of the GPRSC content, pure, and from thinking too much. But how do you process and store so many words as they flow in and out of the GPRSC at 10 Gbits/second? The solution comes to your brilliant and pure mind—a *Bloom filter*.

A Bloom filter is a space-efficient probabilistic data structure, conceived by Burton H. Bloom in 1970, and is used to test whether an element is a member of a set. False-positive matches are possible, but false negatives are not—in other words, a query for set membership returns either "possibly in the set" or "definitely not in the set." Elements can be added to the set but not removed from it; the more elements added, the higher the probability of false positives.

A Bloom filter can be represented as an array of m bits, or a **bit vector**. A Bloom filter should utilize k different hash functions. Using these hash functions, a set element added to the Bloom filter is mapped to at most k of the m bit indices, generating a uniform pseudo-random distribution. Typically, k is a small constant which depends on the desired false error rate ϵ, while m is proportional to k and the number of elements to be added.

Assume you are adding a word w to your Bloom filter and are using $k = 3$ hash functions, $f(x)$, $g(x)$, and $h(x)$. To add w to the Bloom filter, you simply set the bits at indices $f(w)$, $g(w)$, and $h(w)$. To check if some word w' has been added to the same Bloom filter, you check if the bits at indices $f(w')$, $g(w')$, and $h(w')$ are set. If they are all set, then w' has *most likely* been added to the Bloom filter. If *any one* of those bits was cleared, then w' has definitely *not* been added to the Bloom filter. The fact that the Bloom filter can only tell if some word has *most likely* been added to the Bloom filter means that *false positives* can occur. The larger the Bloom filter, the lower the chances of getting false positives.

So what do Bloom filters mean for you as the Dear and Beloved Leader? It means you can take a list of proscribed words, *oldspeak* and add each word into your Bloom filter. If any of the words that your citizens use seem to be added to the Bloom filter, then this is very ungood and further action must be taken to discern whether or not the citizen did transgress. You decide to implement a Bloom filter with *three* salts for *three* different hash functions. Why? To reduce the chance of a *false positive.*

You can think of a "salt" as an initialization vector or a key. Using different salts with the same hash function results in a different, unique hash. Since you are equipping your Bloom filter with three different salts, you are effectively getting three different hash functions: $f(x)$, $g(x)$, and $h(x)$. Hashing a word w, with extremely high probability, should result in $f(w) \neq g(w) \neq h(w)$. These salts are to be used for the SPECK cipher, which requires a 128-bit key, so we have used MD5 [1] "message-digest" to reduce three books down to 128 bits each. These salts are provided for you in `salts.h`. Do not change them. You will use the SPECK cipher as a hash function, which will be discussed in §3.

The following `struct` defines the `BloomFilter` ADT. The three salts will be stored in the `primary`, `secondary`, and `tertiary` fields. Each salt is 128 bits in size. To hold these 128 bits, we use an array of two `uint64_t`s.

```
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;
};
```

This `struct` definition *must* go in `bf.c`.

`BloomFilter *bf_create(uint32_t size)`

The constructor for a Bloom filter. The primary, secondary, and tertiary salts that should be used are provided in `salts.h`. Note that you will also have to implement the bit vector ADT for your Bloom filter, as it will serve as the array of bits necessary for a proper Bloom filter. Bit vectors will be discussed in §4.

`void bf_delete(BloomFilter **bf)`

The destructor for a Bloom filter. As with all other destructors, it should free any memory allocated by the constructor and null out the pointer that was passed in.

`uint32_t bf_size(BloomFilter *bf)`

Returns the size of the Bloom filter. In other words, the number of bits that the Bloom filter can access. Hint: this is the length of the underlying bit vector.

`void bf_insert(BloomFilter *bf, char *oldspeak)`

Takes `oldspeak` and inserts it into the Bloom filter. This entails hashing `oldspeak` with each of the three salts for three indices, and setting the bits at those indices in the underlying bit vector.

[1] Rivest, R.. "The MD5 Message-Digest Algorithm." RFC 1321 (1992): 1-21.

`bool bf_probe(BloomFilter *bf, char *oldspeak)`

Probes the Bloom filter for `oldspeak`. Like with `bf_insert()`, `oldspeak` is hashed with each of the three salts for three indices. If all the bits at those indices are set, return `true` to signify that `oldspeak` was most likely added to the Bloom filter. Else, return `false`.

`uint32_t bf_count(BloomFilter *bf)`

Returns the number of set bits in the Bloom filter.

`void bf_print(BloomFilter *bf)`

A debug function to print out the bits of a Bloom filter. This will ideally utilize the debug print function you implement for your bit vector.

3 Hashing with the SPECK Cipher

War is peace. Freedom is slavery. Ignorance is strength.

—George Orwell, 1984

You will need a good hash function to use in your Bloom filter and hash table (discussed in §5). We will have discussed hash functions in lecture, and rather than risk having a poor one implemented, we will simply provide you one. The SPECK[2] block cipher is provided for use as a hash function.

SPECK is a family of lightweight block ciphers publicly released by the National Security Agency (NSA) in June 2013. SPECK has been optimized for performance in software implementations, while its sister algorithm, SIMON, has been optimized for hardware implementations. SPECK is an add-rotate-xor (ARX) cipher. The reason a cipher is used for this is because encryption generates random output given some input; exactly what we want for a hash.

Encryption is the process of taking some file you wish to protect, usually called plaintext, and transforming its data such that only authorized parties can access it. This transformed data is referred to as ciphertext. Decryption is the inverse operation of encryption, taking the ciphertext and transforming the encrypted data back to its original state as found in the original plaintext. Encryption algorithms that utilize the same key for both encryption and decryption, like SPECK, are symmetric-key algorithms, and algorithms that don't, such as RSA, are asymmetric-key algorithms.

You will be given two files, `speck.h` and `speck.c`. The former will provide the interface to using the SPECK hash function which has been named `hash()`, and the latter contains the implementation. The hash function `hash()` takes two parameters: a 128-bit salt passed in the form of an array of two `uint64_t`s, and a key to hash. The function will return a `uint32_t` which is exactly the index the key is mapped to.

```c
uint32_t hash(uint64_t salt[], char *key);
```

[2] Ray Beaulieu, Stefan Treatman-Clark, Douglas Shors, Bryan Weeks, Jason Smith, and Louis Wingers, "The SIMON and SPECK lightweight block ciphers." In Proceedings of the 52nd ACM/EDAC/IEEE Design Automation Conference (DAC), pp. 1–6. IEEE, 2015.

4 Bit Vectors

I was responsible for everything so I accept responsibility and blame but show me, comrade, one document proving that I was personally responsible for the deaths.

—Pol Pot

A bit vector is an ADT that represents a one dimensional array of bits, the bits in which are used to denote if something is true or false (1 or 0). This is an efficient ADT since, in order to represent the truth or falsity of a bit vector of n items, we can use $\lceil \frac{n}{8} \rceil$ `uint8_t`s instead of n, and being able to access 8 indices with a single integer access is extremely cost efficient. Since we cannot directly access a bit, we must use bitwise operations to get, set, and clear a bit within a byte. Much of the bit vector implementation can be derived from your implementation of the `Code` ADT used in assignment 6. If there were any issues with getting, setting, or clearing bits in that assignment, make sure you address them here.

```c
struct BitVector {
  uint32_t length;
  uint8_t *vector;
};
```

This `struct` definition *must* go in `bv.c`.

`BitVector *bv_create(uint32_t length)`

The constructor for a bit vector that holds `length` bits. In the even that sufficient memory cannot be allocated, the function must return `NULL`. Else, it must return a `BitVector *)`, or a pointer to an allocated `BitVector`. Each bit of the bit vector should be initialized to 0.

`void bv_delete(BitVector **bv)`

The destructor for a bit vector. Remember to set the pointer to `NULL` after the memory associated with the bit vector is freed.

`uint32_t bv_length(BitVector *bv)`

Returns the length of a bit vector.

`bool bv_set_bit(BitVector *bv, uint32_t i)`

Sets the i^{th} bit in a bit vector. If `i` is out of range, return `false`. Otherwise, return `true` to indicate success.

`bool bv_clr_bit(BitVector *bv, uint32_t i)`

Clears the i^{th} bit in the bit vector. If `i` is out of range, return `false`. Otherwise, return `true` to indicate success.

`bool bv_get_bit(BitVector *bv, uint32_t i)`

Returns the i^{th} bit in the bit vector. If `i` is out of range, return `false`. Otherwise, return `false` if the value of bit `i` is `0` and return `true` if the value of bit `i` is `1`.

`void bv_print(BitVector *bv)`

A debug function to print the bits of a bit vector. That is, iterate over each of the bits of the bit vector. Print out either 0 or 1 depending on whether each bit is set. You should write this immediately after the constructor.

5 Hash Tables

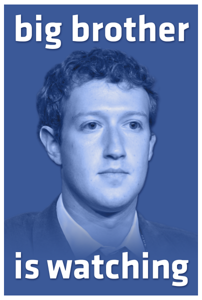
He tried his best to educate the children not to be traitors.

—Mea Son (widow of Pol Pot)

Armed with a Bloom filter, you now exercise the power to catch and punish those who practice wrongthink and continue to use oldspeak. It comes to mind however, that a Bloom filter is probabilistic and that it is better to exercise mercy and counsel the oldspeakers so that they may atone and use *newspeak*. To remedy this, another solution pops into your brilliant and pure mind—a *hash table*.

A hash table is a data structure that maps keys to values and provides fast, O(1), 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 is the perfect way not only to store translations from oldspeak to newspeak, but also as a way to store all prohibited oldspeak words without newspeak translations. We will refer to oldspeak without newspeak translations as *badspeak*. So what happens when two *oldspeak* words have the same hash value? This is called a *hash collision*, and must be resolved. Rather than doing *open addressing* (as will be discussed in lecture), we will be using *binary search trees* to resolve *oldspeak* hash collisions.

A hash function maps data of arbitrary size to fixed-size values. Its values are called hash values, hash codes, digests, or simply hashes, which index a fixed-size table called a hash table. Hash functions and their associated hash tables are used in data storage and retrieval applications to access data (on average) at a nearly constant time per retrieval. They require a storage space that is only fractionally greater than the total space needed for the data. Hashing avoids the non-linear access time of ordered and unordered lists and some trees and the often exponential storage requirements of direct access of large state spaces.



Hash functions rely on statistical properties of key and function interaction: worst-case behavior is an exhaustive search but with a vanishingly small probability, and average-case behavior can be nearly constant (with minimal collisions).

Below is the `struct` definition for a hash table. Similar to a Bloom filter, a hash table contains a salt which is passed to `hash()` whenever a new oldspeak entry is being inserted. As mentioned in §5, we will be using binary search trees to resolve oldspeak hash collisions, which is why a hash table contains an array of trees.

```c
struct HashTable {
  uint64_t salt[2];
  uint32_t size;
  Node **trees;
};
```

This `struct` definition *must* go in `ht.c`.

`HashTable *ht_create(uint32_t size)`

The constructor for a hash table. The `size` parameter denotes the number of indices, or binary search trees, that the hash table can index up to. The salt for the hash table is provided in `salts.h`.

`void ht_delete(HashTable **ht)`

The destructor for a hash table. Each of the binary search trees `trees`, the underlying array of binary search tree root nodes, is freed. The pointer that was passed in should be set to `NULL`.

`uint32_t ht_size(HashTable *ht)`

Returns the hash table's size.

`Node *ht_lookup(HashTable *ht, char *oldspeak)`

Searches for an entry, a node, in the hash table that contains `oldspeak`. A node stores oldspeak and its newspeak translation. The index of the binary search tree to perform a look-up on is calculated by hashing the `oldspeak`. If the node is found, the pointer to the node is returned. Else, a `NULL` pointer is returned.

`void ht_insert(HashTable *ht, char *oldspeak, char *newspeak)`

Inserts the specified `oldspeak` and its corresponding `newspeak` translation into the hash table. The index of the binary search tree to insert into is calculated by hashing the `oldspeak`.

`uint32_t ht_count(HashTable *ht)`

Returns the number of non-`NULL` binary search trees in the hash table.

`double ht_avg_bst_size(HashTable *ht)`

Returns the average binary search tree size. This is computed as the sum of the sizes over all the binary search trees divided by the number of non-`NULL` binary search trees in the hash table. You will need to use `bst_size()`, presented in §7, to compute this.

`double ht_avg_bst_height(HashTable *ht)`

Returns the average binary search tree size. This is computed as the sum of the heights over all the binary search trees divided by the number of non-`NULL` binary search trees in the hash table. You will need to use `bst_height()`, presented in §7, to compute this.

`void ht_print(HashTable *ht)`

A debug function to print out the contents of a hash table. Write this immediately after the constructor.

6 Nodes

As mentioned in §5, binary search trees will be used to resolve hash collisions. As discussed in lecture, binary search trees, and trees in general, are made up of nodes. For this assignment, each node contains *oldspeak* and its *newspeak* translation if it exists. The *key* to search with in a binary search tree is *oldspeak*. Each node, in typical binary search tree fashion, will contain pointers to its left and right children. The node `struct` is defined for you in `node.h` as follows:

```c
struct Node {
    char *oldspeak;
    char *newspeak;
    Node *left;
    Node *right;
};
```

If the `newspeak` field is `NULL`, then the oldspeak contained in this node is badspeak, since there is no newspeak translation. The rest of the interface for the node ADT is provided in `node.h`. The node ADT is made transparent in order to simplify the implementation of the binary search tree ADT.

`Node *node_create(char *oldspeak, char *newspeak)`

The constructor for a node. You will want to make a *copy* of the `oldspeak` and its `newspeak` translation that are passed in. What this means is *allocating memory* and copying over the characters for both `oldspeak` and `newspeak`. You may find `strdup()` useful.

`void node_delete(Node **n)`

The destructor for a node. Only the node `n` is freed. The previous and next nodes that `n` points to *are not* deleted. Since you have allocated memory for `oldspeak` and `newspeak`, remember to free the memory allocated to both of those as well. The pointer to the node should be set to `NULL`.

`void node_print(Node *n)`

While helpful as debug function, you will use this function to produce correct program output. Thus, it is imperative that you print out the contents of a node in the following manner:

- If the node `n` contains oldspeak *and* newspeak, print out the node with this print statement:

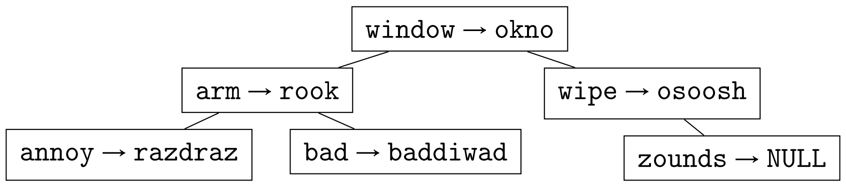
  ```c
  printf("%s -> %s\n", n->oldspeak, n->newspeak);
  ```

- If the node `n` contains *only* oldspeak, meaning that newspeak is null, then print out the node with this print statement:

  ```c
  printf("%s\n", n->oldspeak);
  ```

7 Binary Search Trees

The definition of a binary search tree is a recursive one, as presented in lecture. It is either NULL, or a node that points to up to two subtrees. For any non-NULL node, the left subtree contains keys that are less than it in value, and the right subtree contains keys that are greater than it in value. Since our nodes will contain oldspeak, each node's left subtree contains oldspeak that is lexicographically less than it, and each node's right subtree contains oldspeak that is lexicographically greater than it. You will want to make use of the `strcmp()` function. The diagram below showcases an example binary search tree that might appear for this assignment.



The interface for the binary search tree ADT is provided in `bst.h` and explained in the following sections. You may not modify this header file.

`Node *bst_create(void)`

Constructor for a binary search tree that constructs an *empty* tree. That means that the tree is NULL.

`void bst_delete(Node **root)`

Destructor for a binary search tree rooted at `root`. This should walk the tree using a *postorder* traversal and delete each node of the tree.

`uint32_t bst_height(Node *root)`

Returns the height of the binary search tree rooted at `root`.

`uint32_t bst_size(Node *root)`

Returns the size of the binary search tree rooted at `root`. The size of a tree is equivalent to the number of nodes in the tree.

`Node *bst_find(Node *root, char *oldspeak)`

Searches for a node containing `oldspeak` in the binary search tree rooted at `root`. If a node is found, the pointer to the node is returned. Else, a NULL pointer is returned.

`Node *bst_insert(Node *root, char *oldspeak, char *newspeak)`

Inserts a new node containing the specified `oldspeak` and `newspeak` into the binary search tree rooted at `root`. Duplicates *should not* be inserted.

`void bst_print(Node *root)`

Prints out each node in the binary search tree through an *inorder* traversal. This will require the use of `node_print()`.

8 Lexical Analysis with Regular Expressions

Ideas are more powerful than guns. We would not let our enemies have guns, why should we let them have ideas.

—Joseph Stalin

Back to regulating your citizens of the GPRSC. You will need a function to parse out the words that they speak, which will be passed to you in the form of an input stream. The words that they will use are valid words, which can include *contractions* and *hyphenations*. A valid word is any sequence of one or more characters that are part of your regular expression word character set. Your word character set should contain characters from a–z, A–Z, 0–9, and the underscore character. Since you also accept contractions like "don't" and "y'all've" and hyphenations like "pseudo-code" and "move-to-front", your word character set should include apostrophes and hyphens as well.

You will need to write your own *regular expression* for a word, utilizing the `regex.h` library to lexically analyze the input stream for words. You will be given a parsing module that lexically analyzes the input stream using your regular expression. You are not required to use the module itself, but it is *mandatory* that you parse through an input stream for words using at least one regular expression. The interface for the parsing module will be in `parser.h` and its implementation will be in `parser.c`.

The function `next_word()` requires two inputs, the input stream `infile`, and a pointer to a compiled regular expression, `word_regex`. Notice the word *compiled*: you must first compile your regular expression using `regcomp()` before passing it to the function. Make sure you remember to call the function `clear_words()` to free any memory used by the module when you're done reading in words. Here is a small program that prints out words input to `stdin` using the parsing module. In the program, the regular expression for a word matches one or more lowercase and uppercase letters. The regular expression you will have to write for your assignment will be more complex than the one displayed here, as it is just an example.

Example program using the parsing module.

```c
#include "parser.h"
#include <regex.h>
#include <stdio.h>

#define WORD "[a-zA-Z]+"

int main(void) {
    regex_t re;
    if (regcomp(&re, WORD, REG_EXTENDED)) {
        fprintf(stderr, "Failed to compile regex.\n");
        return 1;
    }

    char *word = NULL;
    while ((word = next_word(stdin, &re)) != NULL) {
        printf("Word: %s\n", word);
    }

    clear_words();
    regfree(&re);
    return 0;
}
```

9 Your Task

The people will believe what the media tells them they believe.

—George Orwell

- Initialize your Bloom filter and hash table.
- Read in a list of *badspeak* words with `fscanf()`. Again, badspeak is simply oldspeak without a newspeak translation. Badspeak is strictly forbidden. Each badspeak word should be added to the Bloom filter and the hash table. The list of proscribed words will be in `badspeak.txt`, which can be found in the `resources` repository.
- Read in a list of *oldspeak* and *newspeak* pairs with `fscanf()`. Only the oldspeak should be added to the Bloom filter. The oldspeak *and* newspeak are added to the hash table. The list of oldspeak and newspeak pairs will be in `newspeak.txt`, which can also be found in the `resources` repository.
- Now that the lexicon of badspeak and oldspeak/newspeak translations has been populated, you can start to filter out words. Read words in from `stdin` using the supplied parsing module.
- For each word that is read in, check to see if it has been added to the Bloom filter. If it has not been added to the Bloom filter, then no action is needed since the word isn't a proscribed word.
- If the word has most likely been added to the Bloom filter, meaning `bf_probe()` returned `true`, then further action needs to be taken.

1. If the hash table contains the word and the word *does not* have a newspeak translation, then the citizen who used this word is guilty of `thoughtcrime`. Insert this badspeak word into a list of badspeak words that the citizen used in order to notify them of their errors later. What data structure could be used to store these words?
2. If the hash table contains the word, and the word *does* have a newspeak translation, then the citizen requires counseling on proper *Rightspeak*. Insert this oldspeak word into a list of oldspeak words with newspeak translations in order to notify the citizen of the revisions needed to be made in order to practice Rightspeak.
3. If the hash table does not contain the word, then all is good since the Bloom filter issued a false positive. No disciplinary action needs to be taken.

- If the citizen is accused of thoughtcrime *and* requires counseling on proper *Rightspeak*, then they are given a reprimanding *mixspeak message* notifying them of their transgressions and promptly sent off to *joycamp*. The message should contain the list of badspeak words that were used followed by the list of oldspeak words that were used with their proper newspeak translations.

```
Dear beloved citizen of the GPRSC,

We have some good news, and we have some bad news.
The good news is that there is bad news. The bad news is that you will
be sent to joycamp and subjected to a week-long destitute existence.
This is the penalty for using degenerate words, as well as using
oldspeak in place of newspeak. We hope you can correct your behavior.

Your transgressions, followed by the words you must think on:

kalamazoo
antidisestablishmentarianism
write -> papertalk
sad -> happy
read -> papertalk
music -> noise
liberty -> badfree
```

- If the citizen is accused solely of thoughtcrime, then they are issued a thoughtcrime message and also sent off to *joycamp*. The *badspeak message* should contain the list of badspeak words that were used.

```
Dear beloved citizen of the GPRSC,

You have been caught using degenerate words that may cause
distress among the moral and upstanding citizens of the GPSRC.
As such, you will be sent to joycamp. It is there where you will
sit and reflect on the consequences of your choice in language.

Your transgressions:

kalamazoo
antidisestablishmentarianism
```

- If the citizen only requires counseling, then they are issued an encouraging *goodspeak message*. They will read it, correct their *wrongthink*, and enjoy the rest of their stay in the GPRSC. The message should contain the list of oldspeak words that were used with their proper newspeak translations.

```
Dear beloved citizen of the GPRSC,

We recognize your efforts in conforming to the language standards
of the GPSRC. Alas, you have been caught uttering questionable words
and thinking unpleasant thoughts. You must correct your wrongspeak
and badthink at once. Failure to do so will result in your deliverance
to joycamp.

Words that you must think on:

write -> papertalk
sad -> happy
read -> papertalk
music -> noise
liberty -> badfree
```

- Each of the messages are defined for you in `messages.h`. You may not modify this file.

- The list of the command-line options your program must support is listed below. *Any* combination of the command-line options must be supported.
  - `-h` prints out the program usage. Refer to the reference program in the resources repository for what to print.
  - `-t size` specifies that the hash table will have `size` entries (the default will be 2^{16}).
  - `-f size` specifies that the Bloom filter will have `size` entries (the default will be 2^{20}).
  - `-s` will enable the printing of statistics to `stdout`. The statistics to calculate are:
    * Average binary search tree size
    * Average binary search tree height
    * Average branches traversed
    * Hash table load
    * Bloom filter load

  The latter three statistics are computed as follows:

$$\text{Average branches traversed} = \frac{\texttt{branches}}{\texttt{lookups}}$$

$$\text{Hash table load} = 100 \times \frac{\texttt{ht\_count()}}{\texttt{ht\_size()}}$$

$$\text{Bloom filter load} = 100 \times \frac{\texttt{bf\_count()}}{\texttt{bf\_size()}}$$

  The hash table load and Bloom filter load should be printed with up to 6 digits of precision. Enabling the printing of statistics should *suppress all messages* the program may otherwise print. The number of lookups is defined as the number of times `ht_lookup()` and `ht_insert()` is called. The number of branches is defined as the count of links traversed during calls to `bst_find()` and `bst_insert()`. The global variable `lookups` should be defined in `ht.c` and the global variable `branches` should be defined in `bst.c`.

10 Deliverables

I would rather have questions that can't be answered than answers that can't be questioned.

—Richard P. Feynman

You will need to turn in the following source code and header files:

1. `banhammer.c`: This contains `main()` and *may* contain the other functions necessary to complete the assignment.
2. `messages.h`: Defines the mixspeak, badspeak, and goodspeak messages that are used in `banhammer.c`. Do not modify this.
3. `salts.h`: Defines the primary, secondary, and tertiary salts to be used in your Bloom filter implementation. Also defines the salt used by the hash table in your hash table implementation.
4. `speck.h`: Defines the interface for the hash function using the SPECK cipher. Do not modify this.
5. `speck.c`: Contains the implementation of the hash function using the SPECK cipher. Do not modify this.
6. `ht.h`: Defines the interface for the hash table ADT. Do not modify this.
7. `ht.c`: Contains the implementation of the hash table ADT.
8. `bst.h`: Defines the interface for the binary search tree ADT. Do not modify this.
9. `bst.c`: Contains the implementation of the binary search tree ADT.
10. `node.h`: Defines the interface for the node ADT. Do not modify this.
11. `node.c`: Contains the implementation of the node ADT.
12. `bf.h`: Defines the interface for the Bloom filter ADT. Do not modify this.
13. `bf.c`: Contains the implementation of the Bloom filter ADT.
14. `bv.h`: Defines the interface for the bit vector ADT. Do not modify this.
15. `bv.c`: Contains the implementation of the bit vector ADT.
16. `parser.h`: Defines the interface for the regex parsing module. Do not modify this.
17. `parser.c`: Contains the implementation of the regex parsing module.

You may have other source and header files, but *do not try to be overly clever.* You will also need to turn in the following:

1. `Makefile`: This is a file that will allow the grader to type `make` to compile your program.
   - `CC = clang` must be specified.
   - `CFLAGS = -Wall -Wextra -Werror -Wpedantic` must be included.
   - `make` should build the `banhammer` executable, as should `make all` and `make banhammer`.
   - `make clean` must remove all files that are compiler generated.
   - `make format` should format all your source code, including the header files.
2. Your code must pass `scan-build` *cleanly.* If there are any bugs or errors that are false positives, document them and explain why they are false positives in your `README.md`.
3. `README.md`: This must be in *Markdown.* This must describe how to use your program and `Makefile`. This includes listing and explaining the command-line options that your program accepts. Any false positives reported by `scan-build` should go here as well.
4. `DESIGN.pdf`: This *must* be a PDF. The design document should describe your design for your program with enough detail that a sufficiently knowledgeable programmer would be able to replicate your implementation. This does not mean copying your entire program in verbatim. You should instead describe how your program works with supporting pseudocode. For this program, pay extra attention to how you build each necessary component.
5. `WRITEUP.pdf`: This document *must* be a PDF. The writeup must include *at least* the following:
   - Graphs comparing the total number of lookups and average binary search tree branches traversed as you vary the hash table and Bloom filter size.
     - How do the heights of the binary search trees change?
     - What are some factors that can change the height of a binary search tree?
     - How does changing the Bloom filter size affect the number of lookups performed in the hash table?
   - Analysis of the graphs you produce.

11 Submission

Refer back assignment 0 for the instructions on how to properly submit your assignment through `git`. Remember: *add*, *commit*, and *push*!

Your assignment is turned in *only* after you have pushed and submitted the commit ID you want graded on Canvas. “I forgot to push” and “I forgot to submit my commit ID” are not valid excuses. It is *highly* recommended to commit and push your changes *often.*

12 Supplemental Readings

It is good that Pol Pot is dead. I feel no sorrow. . . I hope he suffers after death.

—Ta Mok

- *The C Programming Language* by Kernighan & Ritchie
  - Chapter 5 §5.7
  - Chapter 7
- *Introduction to Algorithms* by T. Cormen, C. Leiserson, R. Rivest, & C. Stein
  - Chapter 11 (Hash tables and hash functions)
  - Chapter 12 (Binary Search Trees)
- *Introduction to the Theory of Computation* by M. Sipser
  - Chapter 1 §1.3 (Regular expressions)

The scientific notes with which your letters are so richly filled have given me a thousand pleasures. I have studied them with attention and I admire the ease with which you penetrate all branches of arithmetic, and the wisdom with which you generalize and perfect. —C.F. Gauß to Sophie Germain