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

Prof. Max Dunne
CSE 13S

1 Introduction

"When I use a word," Humpty Dumpty said, in rather a scornful tone, "it means just what I choose it to mean—neither more nor less." "The question is," said Alice, "whether you can make words mean so many different things." "The question is," said Humpty Dumpty, "which is to be master—that's all."

—The Red Queen shook her head. "You may call it 'nonsense' if you like," she said, "but I've heard nonsense, compared with which that would be as sensible as a dictionary!"

The Red Queen (of confused with the Queen of Hearts of "off with their heads" fame) has been ruling over her kingdom for many an eon, and has been particularly hung up on the sense of meaning and works and particularly nonsense. In order to ward off the potential devolution of looking glass speak into unintelligible gibberish being promoted by Humpty Dumpty and enforced by the Jabberwocky, the queen has decreed:

> All speech shall be in Hatter Speech, and no other shall be accepted, under pain of suspicion of alliance with the ne'er-do-well Dumpty and his nefarious accomplice Mr. Wocky. It is hereby decreed that all denizens of the looking glass speak only in the manner of the Hatter, and violations of this decree shall be delivered to harsh punishments in the royal dungeons.

2 Background

2.1 Bloom Filters

As a computer savvy programmer in the looking glass kingdom, you have been summoned by the queen to devise a program that will filter the internet for non-hatter speech. But how do you process so many words as they flow in to your little kingdom at 10 Gbits/second? The solution that comes to mind — a Bloom filter.

A Bloom filter is similar to a hash table, where the entries in the hash table are simply single bits. Consider `h(`*text*`) = k`, then if $B_k = 0$ then the entry is definitely missing; if $B_k = 1$ then the entry may be present. The latter is called a false positive, and the false positive rate depends on the size of the Bloom filter and the number of texts that hash to the same position (hash collisions).

You assign your programming minions to take every *potentially nonsense* word that they can find in an English dictionary and hash them into the Bloom filter. That is, for each word the corresponding bit in the filter is set to 1.

You will make a Bloom filter with *three* salts for *three* different hash functions. Why? To reduce the chance of a *false positive*.

```
#ifndef NIL
#define NIL (void *)0
#endif

#ifndef __BF_H__
#define __BF_H__
```

```c

#include "bv.h"
#include <inttypes.h>
#include <stdbool.h>

//
// Struct definition for a BloomFilter.
//
// primary:     Primary hash function salt.
// secondary:   Secondary hash function salt.
// tertiary:    Tertiary hash function salt.
// filter:      BitVector that determines membership of a key.
//
typedef struct BloomFilter {
  uint64_t primary  [2]; // Provide for three different hash functions
  uint64_t secondary[2];
  uint64_t tertiary [2];
  BitVector *filter;
} BloomFilter;

//
// Constructor for a BloomFilter.
//
// size:  The number of entries in the BloomFilter.
//
BloomFilter *bf_create(uint32_t size);

//
// Destructor for a BloomFilter.
//
// bf:  The BloomFilter.
//
void bf_delete(BloomFilter *bf);

//
// Inserts a new key into the BloomFilter.
// Indices to set bits are given by the  hash functions.
//
// bf:    The BloomFilter.
// key:   The key to insert into the BloomFilter.
//
void bf_insert(BloomFilter *bf, char *key);

//
// Probes a BloomFilter to check if a key has been inserted.
//
// bf:    The BloomFilter.
// key:   The key in which to check membership.
//
bool bf_probe(BloomFilter *bf, char *key);

```

```
#endif
```

bf.h

```
#include "bf.h"
#include <stdlib.h>

//
// Constructor for a BloomFilter.
//
// size:  The number of entries in the BloomFilter.
//
BloomFilter *bf_create(uint32_t size) {
  BloomFilter *bf = (BloomFilter *)malloc(sizeof(BloomFilter));
  if (bf) {
    bf->primary  [0] = 0xfc28ca6885711cf7;
    bf->primary  [1] = 0x2841af568222f773;
    bf->secondary[0] = 0x85ae998311115ae3;
    bf->secondary[1] = 0xb6fac2ae33a40089;
    bf->tertiary [0] = 0xd37b01df0ae8f8d0;
    bf->tertiary [1] = 0x911d454886ca7cf7;
    bf->filter = bv_create(size);
    return bf;
  }
  return (BloomFilter *)NIL;
}

// The rest of the functions you must implement yourselves.
```

bf.c

A stream of words is passed to the Bloom filter. If the Bloom filter rejects all words then the person responsible is *innocent* of *oldspeak*. But if any word has a corresponding bit set in the Bloom filter, it is likely that they are *guilty* of *oldspeak*. To make certain, we must consult the *hash table* and if the word is there as a *nonsense* word then they will be hauled off to the dungeon. If the word passed both Bloom filters and is not a forbidden word, the hash table will provide a translation that will replace nonsense words with hatter approved words. The advantage is that your kingdom can augment this list at any time. Simply put, there are three cases to consider:

1. Words that are approved will *not* appear in the Bloom filter.
2. Words that should be replaced, which will have a mapping from the old word, *oldspeak*, to the new approved word, *hatterspeak* (also referred to as a *translation*).
3. Words without translations to new approved words means that it's off to the *dungeon*.

You will use the *BitVector* data structure that you developed for *Assignment 4* to implement your Bloom filter.

Pre-lab Part 1

1. Write down the pseudocode for inserting and deleting elements from a Bloom filter.
2. Assuming that you are creating a bloom filter with m bits and k hash functions, discuss its time and space complexity.

2.2 Hash Tables

In general, rather than punish them, we would rather counsel them how to *Rightspeak*. You decide that the easiest way to make word replacements is through a translation table, with entries of the form:

```
typedef struct HatterSpeak {
  char *oldspeak;
  char *hatterspeak;
} HatterSpeak;
```

HatterSpeak struct

If a word is in your Bloom filter, then it is either a nonsense word or a word that needs to be translated by the hash table. You will use the hash table to locate that word. If it is found, the system helpfully provides the appropriate new improved word in its place.

A hash table is one that is indexed by a function, `hash()`, applied to the *key*. The key in this case will be the word in *oldspeak*. As we have said, this provides a mapping from *oldspeak* to *hatterspeak*. Words for which there is no translation result in *dungeon*. All other words must have a replacement *hatterspeak* word.

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 discussed in class), we will be using *linked lists* to resolve *oldspeak* hash collisions.

2.3 Hashing with the SPECK Cipher

First of all, you need a good hash function. We have discussed hash functions in class, and rather than risk having a poor one implemented, we will simply provide you one. The SPECK[1] 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.

```
#ifndef __SPECK_H__
#define __SPECK_H__

#include <inttypes.h>

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

#endif
```

speck.h

```
#include "speck.h"
#include <inttypes.h>
#include <stddef.h>
```

[1] 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.

```c
#include <string.h>

#define LCS(X, K) \
  (X << K) | (X >> (sizeof(uint64_t) * 8 - K)) // left circular shift
#define RCS(X, K) \
  (X >> K) | (X << (sizeof(uint64_t) * 8 - K)) // right circular shift

// Core SPECK operation
#define R(x, y, k) (x = RCS(x, 8), x += y, x ^= k, y = LCS(y, 3), y ^= x)


void speck_expand_key_and_encrypt(uint64_t pt[], uint64_t ct[], uint64_t K[]) {
  uint64_t B = K[1], A = K[0];
  ct[0] = pt[0];
  ct[1] = pt[1];

  for (size_t i = 0; i < 32; i += 1) {
    R(ct[1], ct[0], A);
    R(B, A, i);
  }
}

uint64_t keyed_hash(const char *s, uint32_t length, uint64_t key[]) {
  uint64_t accum = 0;

  union {
    char b[sizeof(uint64_t)]; // 16 bytes fit into the same space as
    uint64_t ll[2]; // 2 64 bit numbers.
  } in;

  uint64_t out[2]; // SPECK results in 128 bits of ciphertext
  uint32_t count;

  count = 0; // Reset buffer counter
  in.ll[0] = 0x0;
  in.ll[1] = 0x0; // Reset the input buffer (zero fill)

  for (size_t i = 0; i < length; i += 1) {
    in.b[count++] = s[i]; // Load the bytes

    if (count % (2 * sizeof(uint64_t)) == 0) {
      speck_expand_key_and_encrypt(in.ll, out, key); // Encrypt 16 bytes
      accum ^= out[0] ^ out[1]; // Add (XOR) them in for a 64 bit result
      count = 0; // Reset buffer counter
      in.ll[0] = 0x0;
      in.ll[1] = 0x0; // Reset the input buffer
    }
  }
```

```
52
53    // There may be some bytes left over, we should use them.
54    if (length % (2 * sizeof(uint64_t)) != 0) {
55      speck_expand_key_and_encrypt(in.ll, out, key);
56      accum ^= out[0] ^ out[1];
57    }
58
59    return accum;
60  }
61
62  //
63  // Wrapper function to get a 32-bit hash value by using SPECK's key hash.
64  // SPECK's key hash requires a key and a salt.
65  //
66  // ht:      The HashTable.
67  // key:     The key to hash.
68  //
69  uint32_t hash(uint64_t salt[], char *key) {
70    union {
71      uint64_t full;
72      uint32_t half[2];
73    } value;
74
75    value.full = keyed_hash(key, strlen(key), salt);
76
77    return value.half[0] ^ value.half[1];
78  }
```

speck.c

```
1
2  #ifndef __HASH_H__
3  #define __HASH_H__
4
5  #ifndef NIL
6  #define NIL (void *)0
7  #endif
8
9  #include "ll.h"
10 #include <inttypes.h>
11
12 //
13 // Struct definition for a HashTable.
14 //
15 // salt:        The salt of the HashTable (used for hashing).
16 // length:      The maximum number of entries in the HashTable.
17 // heads:       An array of linked list heads.
18 //
19 typedef struct HashTable {
20   uint64_t salt[2];
21   uint32_t length;
22   ListNode **heads;
```

```
} HashTable;

//
// Constructor for a HashTable.
//
// length:  Length of the HashTable.
// salt:    Salt for the HashTable.
//
HashTable *ht_create(uint32_t length);

//
// Destructor for a HashTable.
//
// ht:  The HashTable.
//
void ht_delete(HashTable *ht);

//
// Returns number of entries in hash table
//
// h:  The HashTable.
//
uint32_t ht_count(HashTable *h);

//
// Searches a HashTable for a key.
// Returns the ListNode if found and returns NULL otherwise.
// This should call the ll_lookup() function.
//
// ht:      The HashTable.
// key:     The key to search for.
//
ListNode *ht_lookup(HashTable *ht, char *key);

//
// First creates a new ListNode from HatterSpeak.
// The created ListNode is then inserted into a HashTable.
// This should call the ll_insert() function.
//
// ht:          The HashTable.
// gs:          The HatterSpeak to add to the HashTable.
//
void ht_insert(HashTable *ht, HatterSpeak *gs);

#endif
```

hash.h

```


//
// Constructor for a HashTable.
```

```c
//
// length:  Length of the HashTable.
// salt:    Salt for the HashTable.
//
HashTable *ht_create(uint32_t length) {
  HashTable *ht = (HashTable *)malloc(sizeof(HashTable));
  if (ht) {
    ht->salt[0] = 0x85ae998311115ae3;
    ht->salt[1] = 0xb6fac2ae33a40089;
    ht->length = length;
    ht->heads = (ListNode **)calloc(length, sizeof(ListNode *));
    return ht;
  }

  return (HashTable *)NIL;
}

// The rest of the functions you must implement yourselves.
```

create from hash.c

2.4 Linked Lists

A *linked list* will be used to resolve hash collisions. Each node of the linked list contains a `HatterSpeak struct` which contains *oldspeak* and its *hatterspeak* translation if it exists. The *key* to search with in the linked list is *oldspeak*.

```c
#ifndef __LL_H__
#define __LL_H__

#ifndef NIL
#define NIL (void *)0
#endif

#include <stdbool.h>

// If flag is set, ListNodes that are queried are moved to the front.
extern bool move_to_front;

typedef struct ListNode ListNode;

//
// Struct definition of a ListNode.
//
// gs:  HatterSpeak struct containing oldspeak and its hatterspeak
    translation.
//
struct ListNode {
  HatterSpeak *gs;
  ListNode *next;
};

```

```
//
// Constructor for a ListNode.
//
// gs:  HatterSpeak struct containing oldspeak and its hatterspeak
    translation.
//
ListNode *ll_node_create(HatterSpeak *gs);

//
// Destructor for a ListNode.
//
// n:  The ListNode to free.
//
void ll_node_delete(ListNode *n);

//
// Destructor for a linked list of ListNodes.
//
// head:   The head of the linked list.
//
void ll_delete(ListNode *head);

//
// Creates and inserts a ListNode into a linked list.
//
// head:  The head of the linked list to insert in.
// gs:    HatterSpeak struct.
//
ListNode *ll_insert(ListNode **head, HatterSpeak *gs);

//
// Searches for a specific key in a linked list.
// Returns the ListNode if found and NULL otherwise.
//
// head:   The head of the linked list to search in.
// key:    The key to search for.
ListNode *ll_lookup(ListNode **head, char *key);

#endif
```

ll.h

You will be implementing this in two forms, and comparing the performance of the methods:

- Inserting each new word at the front of the list and
- Inserting each new word at the front of the list, but *each* time it is searched for it is *moved to the front* of the list.

The move-to-front technique moves a node that was just searched for in the linked list to the front. This decreases look-up times for frequently-searched-for nodes. You will learn more about optimality in your future classes. You will keep track of the *average number of links followed* in order to find each word in a linked list. The easiest way to do this is to keep track in lookup and should include the original building of the hash table. Without lots of duplicates in the input this will only alter your statistics barely.

Pre-lab Part 2

1. Draw the pictures to show the how elements are being inserted in different ways in the Linked list.
2. Write down the pseudocode for the above functions in the Linked List data type.

2.5 Lexical Analysis with Regular Expressions

You will need a function to pick words from an input stream. The words should be just valid words, and these can include *contractions*. A valid word is any sequence of one or more characters, that are considered part of the word character set in regex. A word character set in regex contains characters from $a-z$, $A-Z$, $0-9$, including the underscore character. Since your program also accepts contractions like "don't" where you will have to account for apostrophes, and words like "pseudo-code" where you will have to account for hyphens.

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. To aid you with this, here is a simple 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.

```
#ifndef __PARSER_H__
#define __PARSER_H__

#include <regex.h>
#include <stdio.h>

//
// Returns the next word that matches the specified regular expression.
// Words are buffered and returned as they are read from the input file.
//
// infile:      The input file to read from.
// word_regex:  Pointer to a compiled regular expression for a word.
// returns:     The next word if it exists, a null pointer otherwise.
//
char *next_word(FILE *infile, regex_t *word_regex);

//
// Clears out the static word buffer.
//
void clear_words(void);

#endif
```

parser.h

```
#include "parser.h"
#include <regex.h>
#include <inttypes.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define BLOCK   4096

```

```c
static char *words[BLOCK] = { NULL };  // Stores a block of words maximum.

//
// Returns the next word that matches the specified regular expression.
// Words are buffered and returned as they are read from the input file.
//
// infile:      The input file to read from.
// word_regex:  Pointer to a compiled regular expression for a word.
// returns:     The next word if it exists, a null pointer otherwise.
//
char *next_word(FILE *infile, regex_t *word_regex) {
  static uint32_t index = 0;  // Track the word to return.
  static uint32_t count = 0;  // How many words have we stored?

  if (!index) {
    clear_words();

    regmatch_t match;
    uint32_t matches = 0;
    char buffer[BLOCK] = { 0 };

    while (!matches) {
      if (!fgets(buffer, BLOCK, infile)) {
        return NULL;
      }

      char *cursor = buffer;

      for (uint16_t i = 0; i < BLOCK; i += 1) {
        if (regexec(word_regex, cursor, 1, &match, 0)) {
          break;  // Couldn't find a match.
        }

        if (match.rm_so < 0) {
          break;  // No more matches.
        }

        uint32_t start  = (uint32_t)match.rm_so;
        uint32_t end    = (uint32_t)match.rm_eo;
        uint32_t length = end - start;

        words[i] = (char *)calloc(length + 1, sizeof(char));
        if (!words[i]) {
          perror("calloc");
          exit(1);
        }

        memcpy(words[i], cursor + start, length);
        cursor += end;
        matches += 1;
      }
```

```
61
62          count = matches;  // Words stored is number of matches.
63        }
64      }
65
66      char *word = words[index];
67      index = (index + 1) % count;
68      return word;
69  }
70
71  //
72  // Clears out the static word buffer.
73  //
74  void clear_words(void) {
75      for (uint16_t i = 0; i < BLOCK; i += 1) {
76          if (words[i]) {
77              free(words[i]);
78              words[i] = NULL;
79          }
80      }
81
82      return;
83  }
```

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. Note that you will need to transform all of your words from *mixed case* to *lowercase* before adding them to your Bloom filter and hash table.

3 Your Task

- Read in a list of *nonsense* words, setting the corresponding bit for each word in the Bloom filter. This list is the oldspeak.txt file we mentioned earlier in the assignment.
- Create a `HatterSpeak struct` for each forbidden word. The word should be stored in `oldspeak`, and `hatterspeak` should be set to `NULL`; forbidden words do not have translations.
- Read in a space-separated list of *oldspeak*, *hatterspeak* pairs. This list is the hatterspeak.txt file we mentioned earlier in the assignment.
- Create a `HatterSpeak struct` for each *oldspeak*, *hatterspeak* pair, placing them in `oldspeak` and `hatterspeak` respectively.
- The hash index for each *nonsense* word is determined by using `oldspeak` as the key.
- Read text from *standard input* (I/O redirection must be supported).
- Words that pass through the Bloom filter but have no translation are forbidden, which constitutes a *nontalk*.
- The use of nonsense words constitutes a *nontalk*. If only forbidden words were used, you will send them a *nontalk message*.

```
Dear Wonderlander,

You have chosen to use words that the queen has decreed oldspeak.
Due to your infraction you will be sent to the dungeon where you will
    be taught hatterspeak.

Your errors:

kalamazoo
gizbits
```

Example nontalk message.

- The use of only *oldspeak* words that have *hatterspeak* translations elicits a *hatterspeak message* to ensure better speech in the future.

```
Dear Wonderlander,

The decree for hatterspeak finds your message lacking. Some of the
    words that you used are not hatterspeak.
The list shows how to turn the oldspeak words into hatterspeak.

body -> plott
sarcastic -> sarky
blood -> krovvy
```

Example hatterspeak message.

- The use of nonsense words and words that have *hatterspeak* translations warrants a combination of a *nontalk message* and a *hatterspeak* message.

```
Dear Comrade,

You have chosen to use words that the queen has decreed oldspeak.
Due to your infraction you will be sent to the dungeon where you will
    be taught hatterspeak.

Your errors:

kalamazoo
gizbits

Appropriate hatterspeak translations.

body -> plott
sarcastic -> sarky
blood -> krovvy
```

Example nontalk/hatterspeak message.

- The list of proscribed words is available on CANVAS.
- The list of *oldspeak* words and their respective *hatterspeak* on CANVAS
- Your executable, `./hatterspeak`, will provide the following options:

- `./hatterspeak -s` will suppress the letter from the censor, and instead print the statistics that were computed as illustrated below
  - `Seeks:` number of seeks performed
  - `Average seek length:` links searched / total seeks
  - `Average Linked List Length:` average length of linked lists in hash table
  - `Hash table load:` percentage of loading for the hash table
  - `Bloom filter load:` percentage of loading for the bloom table
- `./hatterspeak -h size` specifies that the hash table will have `size` entries (the default will be 10000).
- `./hatterspeak -f size` specifies that the Bloom filter will have `size` entries (the default will be 2^{20}).
- `./hatterspeak -m` will use the *move-to-front rule*.
- `./hatterspeak -b` will not use the *move-to-front rule*.
- ANY combination of these flags except for `-m -b` *must be supported*.

4 Deliverables

You will need to turn in:

1. `hatterspeak.c`: This contains `main()` and *may* contain the other functions necessary to complete the assignment.
2. `speck.c` and `speck.h`: These files are available on CANVAS.
3. `hash.c` and `hash.h`: These contain your implementation of the hash table ADT.
4. `ll.c` and `ll.h`: These contain your implementation of the linked list ADT.
5. You may have other source and header files, but *do not make things overly complicated*. For example the `parser.c` and `parser.h` are available on CANVAS.
6. `Makefile`: This is a file that will allow the grader to type `make` to compile your program. Typing `make` must build your program.
   - `CFLAGS=-Wall -Wextra -Werror -Wpedantic -std=c99` must be included.
   - `CC=clang` must be specified.
   - `make clean` must remove all files that are compiler generated.
   - `make infer` runs `infer` on your program. Complaints generated by `infer` must either be fixed or explained in your `README`.
   - `make` should build your program, as should `make all`.
   - Your program executable *must* be named `hatterspeak`.
7. `README.md`: This must be in markdown. This must describe how to use your program and `Makefile`. This should also contain explanations for any complaints from `infer`.
8. `DESIGN.pdf`: This *must* be a PDF. The design document should contain answers to the pre-lab questions at the beginning and 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 pseudo-code. For this program, pay extra attention to how you build each necessary component.
9. `WRITEUP.pdf`: Your writeup should contain a discussion on the following topics:

- What happens when you vary the size of a hash table?
- What happens when you vary the Bloom filter size?
- Do you really need the *move to front* rule?

You *must* push the `DESIGN.pdf` before you push *any* code.

All of these files must be in the directory `asgn6`.

5 Submission

To submit your assignment, refer back to `assignment0` for the steps on how to submit your assignment through `git`. Remember: *add, commit,* and *push*!

Your assignment is turned in *only* after you have pushed and turned in the commit ID to Canvas. If you forget to push, you have not turned in your assignment and you will get a *zero*. "I forgot to push" is not a valid excuse. It is *highly* recommended to commit and push your changes *often*.