Roleen Ferrer
Professor Maxwell Dunne
CSE 13S - Computer Systems and C Programming

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

In this lab, I will be implementing Abstract Data Types such as Bloom Filters, Hash Tables, and Linked Lists in order to translate and identify words in "HatterSpeak". I will be using I/O redirection in order to parse through a stream of words through standard input.

In order to deduce which word is considered "OldSpeak", I will pass a text file into a bloom filter. A bloom filter is a very efficient ADT, which utilizes bit vectors to set bits from a given index of words. If a given word is set at its bit vector index, the word is considered OldSpeak, and the person must be punished accordingly. There are three scenarios given a stream of words:

1) Guilty of NonTalk
    a) A person guilty of NonTalk only used OldSpeak words with no HatterSpeak translations. A person guilty of NonTalk will be sent to the dungeon.
2) HatterSpeak Translation
    a) A person who uses OldSpeak words but has a HatterSpeak translation will not receive a harsher punishment than someone guilty of NonTalk. The person will then be told of their errors by translation to HatterSpeak.
3) Both OldSpeak and HatterSpeak
    a) A person who uses OldSpeak words with no translations and words with translations will be guilty of both. The person will be told of their OldSpeak errors and proper HatterSpeak translations for OldSpeak words.

In order to parse through the stream of words, I will be using a HashTable coupled with a ListNode ADT in order to parse through the words and see which word is an OldSpeak word. The ListNode will hold a HatterSpeak struct inside of it, which will hold an OldSpeak (key) word and a HatterSpeak word, if there exists. The HashTable is then indexed using a hash

function applied to the key. Using a LinkedList will also help resolve hash collisions, thus improving the accuracy of finding an OldSpeak word.

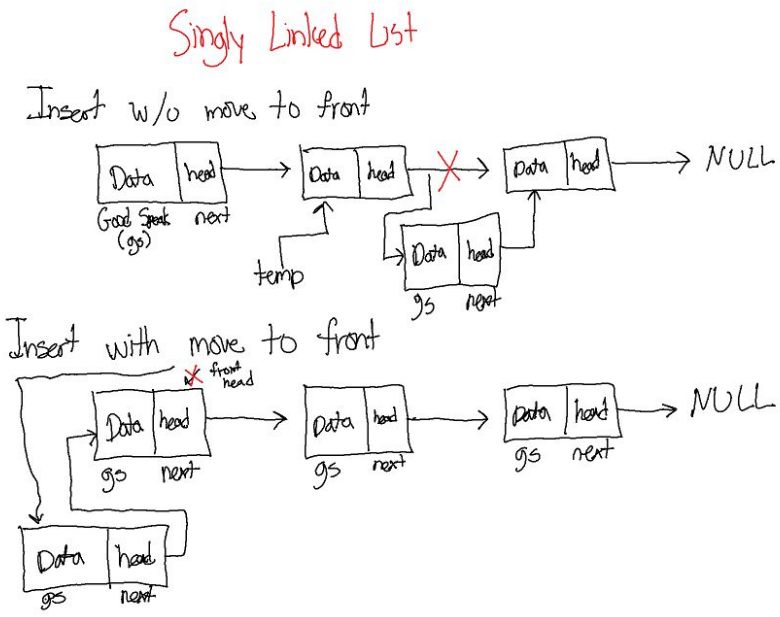
## Pre-Lab Questions

### Part I

1) See Bloom Filter implementation
2) Using a Bloom Filter with m bits and k hash functions, the Bloom Filter should have an average time complexity of O(k), since you only need to check or add an element with one pass through using the given hash function. This works the same way with the setting or clearing the bits.

### Part II

1)




2) See Linked List implementation

## Implementation

### Main (hatterspeak.c)

This source fill handles all of the commands that are input by the user, parses through files in order to obtain an OldSpeak list of words, parses through a stream of words using a parser, and printing statistics or given messages at the end of parsing through the words.

1. Parse through a group of commands in order to obtain certain information regarding how each ADT will be implemented. These commands are as follows:
    a. -s will print statistics of each ADT instead of a message at the end parse through. These statistics include:
        i. Seeks: number of seeks performed
        ii. Average seek length: The links searched / total seeks
        iii. Average linked list length: The average length of non-zero linked lists in the hash table
        iv. Hash table load: The percentage of loading for the hash table
        v. Bloom filter load: The percentage of loading for the bloom table
    b. -h will specify the hash table size of entries (default is 10000)
    c. -f will specify the bloom filter size of entries (default is 2^20)
    d. -m will use the move-to-front rule (for Linked Lists)
    e. -b will not use the move-to-front rule (for Linked Lists)
2. Create BloomFilter and HashTable structs, calling bf_create() and ht_create() to create these ADT's
3. Parse through the files
    a. Create word buffers that will hold a stored word
    b. Use a while loop to scan through the file until EOF
        i. while(fscanf(oldspeak_file, "%s\n", word) != EOF)
        ii. Use this loop to insert the word into the BloomFilter and HashTable by using bf_insert() and ht_insert()
    c. Close the files when finished
4. Use a Regular Expression to parse through a stream of words
    a. Using regcomp, compile a Regular Expression (regex) to use
        i. [a-zA-Z0-9_]+((‘|-)[a-zA-Z0-9_]*)*
5. Create ListNodes (linked lists) that will store the words from the parser
6. Create a while loop that will parse through each word

    a. This while loop should be calling next_word() function from parser.c
    b. Check if the word is in the Bloom Filter
        i. If the word is in the Bloom Filter, it is most likely an OldSpeak word
    c. Create a HatterSpeak struct that will check the OldSpeak and HatterSpeak translation word
        i. If the OldSpeak word has a HatterSpeak translation, store it into the translation ListNode
        ii. If the OldSpeak word does not have a HatterSpeak translation, store it into the forbidden ListNode
7. Print out the statistics or message
    a. This will be determined by the -s command the user input in the beginning
8. Free used Abstract Data Types

**Bloom Filter (bf.c)**

This source file includes Bloom Filter ADT and other functions such as inserting a word into the BloomFilter and checking membership.

1. BloomFilter *bf_create(uint32_size)
    a. This function will create a Bloom Filter ADT. It contains 3 salts that Bloom Filter will use coupled with a Hash Function, which will then set 3 bits in the Bit Vector.
2. void bf_delete(BloomFilter *bf)
    a. This function will delete the Bloom Filter, clearing up memory
    b. Use bv_delete() (from the Bit Vector source file) to free the filter
        i. bv_delete(bf->filter)
    c. Then, free the Bloom Filter itself
3. void bf_insert(BloomFilter *bf, char *key)
    a. This function will insert a certain key (OldSpeak word) into the Bloom Filter
    b. Store hashes using the hash() function from speck.c.
        i. hash(bf->primary, key) % bf->filter ->length
    c. Create a conditional that will check the bits that are not set
        i. This is used for statistics when printing the Bloom Filter load
    d. Set the bits in the Bloom Filter using bv_set_bit()

        i. bv_set_bit(bf->filter, hash)
4. bool bf_probe(BloomFilter *bf, char *key)
    a. This function will check if a key is inserted into the Bloom Filter. Will return True if it is and False otherwise
    b. Get the hashes of the word that were set in the Bloom Filter
        i. hash(bf->primary, key) % bf->filter ->length
    c. Check if each bit is set
        i. If all 3 are set, the word is most likely inside the Bloom Filter

**Hash Table (hash.c)**

This source file includes the HashTable ADT and contains functions that will return information about the Hash Table, including Hash Tables and item membership.

1. HashTable *ht_create(uint32_t length)
    a. This function will create the HashTable ADT, which contains a salt that will be used by the hash function. The HashTable will also store the heads of a ListNode by using a double pointer.
2. void ht_delete(HashTable *ht)
    a. This function will delete the HashTable ADT
    b. Create a for loop that will iterate through the heads that are stored in the HashTable
        i. for (uint32_t i = 0; i < ht->length; i++)
        ii. Check if the current head is not NULL
        iii. If it isn't NULL, delete the head by calling ll_delete()
            1. ll_delete(ht->heads[i])
    c. Free the heads element
    d. Free the HashTable
3. uint32_t ht_count(HashTable *h)
    a. This function will return the number of entries that are in the HashTable
    b. Create a for loop that will count the heads that are not NULL
        i. for (uint32_t i = 0; i < h->length; i++)
    c. Return the count
4. ListNode *ht_lookup(HashTable *ht, char *key)

a. This function will search the HashTable for a key. It will return the ListNode if found
b. Store the index of the HashTable by calling the hash() function and the HashTable's key
    i. uint32_t index = hash(ht->salt, key) % ht-> length
    ii. Return the LinkedList of the given index using ll_lookup() function
        1. return ll_lookup(&ht->heads[index], key)

5. void ht_insert(HashTable *ht, HatterSpeak *gs)
    a. This function will create a new ListNode from a HatterSpeak struct. This will then be inserted into the HashTable
    b. Store the index of the HashTable by calling the hash() function and the HashTable's key
    c. Create a ListNode by calling the ll_insert() function and store it into ht->heads[index]
        i. ht->heads[index] = ll_insert(&ht->heads[index], gs)

**Linked List (ll.c)**

This source file contains the LinkedList implementation and functions that will manipulate the LinkedList through deletion, insertion, or lookups.

1. ListNode *ll_node_create(HatterSpeak *gs)
    a. This function will create a ListNode struct
    b. node->gs will store the the HatterSpeak struct
    c. node->next will store the head
2. void ll_node_delete(ListNode *n)
    a. This function will delete a ListNode
    b. Free memory for the GoodSpeak struct
        i. free (n->gs->oldspeak)
        ii. free(n->gs->hatterspeak)
        iii. free(n->gs)
    c. Free the ListNode itself
3. void ll_delete(ListNode *head)

a. This function will delete a Linked List of ListNodes
b. Create a ListNode that will store the next head
c. Create a while loop that will run until head equal to NULL
    i. Set the ListNode to the current head's next head
    ii. Delete the ListNode by calling ll_node_delete()
    iii. Set the head to next

4. ListNode *ll_insert (ListNode ** head, HatterSpeak *gs)
    a. This function will create and insert a ListNode into a Linked List
    b. Check if there exists a LinkNode that is in the Linked List
        i. If there is, free the GoodSpeak struct and return the head
    c. Add 1 to the sum for statistics (Average Linked List Length)
    d. Create a new ListNode
        i. Set the head of the new ListNode to the head
        ii. Set the current head to the new ListNode
        iii. Return the head
5. ListNode *ll_lookup(ListNode **head, char *key)
    a. This function will search for a specific key in a Linked List. It will return the ListNode or NULL otherwise
    b. Create 2 ListNodes, one that is NULL and will store a head and a current one that will store the current head
    c. Create a while loop that will run until current ListNode is NULL
        i. Add 1 to traversed for statistics (Average Seek Length)
        ii. Create a conditional that will check compare if the OldSpeak key is equal to the given key
            1. Create another conditional if move_to_front is enabled and the previous ListNode is not NULL
                a. If this condition is met
                    i. Set previous->next to current->next
                    ii. Set current->next = *head
                b. Set the head to the current ListNode
                c. Return the head

iii. Set the previous to current

iv. Set current to current->next

d. Return NULL if nothing is found