Kyven Wang
kwang126@ucsc.edu
4 June 2021

CSE13S Spring 2021
Assignment 7: The Great Firewall of Santa Cruz: Bloom Filters, Linked Lists and Hash Tables
Design Document

Description:

This assignment is a language filter that takes in input and detects any badspeak (strictly forbidden word) and oldspeak (has a replacement newspeak word). If only badspeak is detected, a disciplinary message is printed. If only oldspeak is detected, a warning message is printed. If both are detected, a message serving to notify the user of disciplinary action and warn the user is printed. This filter utilizes a Bloom filter, hash table, linked list, bit vector, SPECK cipher, and regex parser to operate.

Sources:

- asgn7-v2.pdf (Professor Darrell Long)

Given Code: (taken from asgn7-v2.pdf unless specified otherwise)

BloomFilter ADT:

```
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;
};
```

BloomFilter Constructor:

```
BloomFilter *bf_create(uint32_t size) {
        BloomFilter *bf = (BloomFilter *)  malloc(sizeof(BloomFilter));
        if (bf) {
                // Grimm's Fairy Tales
                bf->primary [0] = 0x5adf08ae86d36f21;
                bf->primary [1] = 0xa267bbd3116f3957;
                // The Adventures of Sherlock Holmes
                bf->secondary [0] = 0x419d292ea2ffd49e;
                bf->secondary [1] = 0x09601433057d5786;
                // The Strange Case of Dr. Jekyll and Mr. Hyde
                bf->tertiary [0] = 0x50d8bb08de3818df;
```

```
                bf->tertiary [1] = 0x4deaae187c16ae1d;
                bf->filter = bv_create(size);
                if (!bf->filter) {
                        free(bf);
                        bf = NULL;
                }
        }
        return bf;
}
```

SPECK Cipher hash() Function Header (for usage, implementation is provided):

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

BitVector ADT:

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

HashTable ADT:

```
struct HashTable {
        uint64_t salt[2];
        uint32_t size;
        bool mtf;
        LinkedList **lists;
};
```

HashTable Constructor:

```
HashTable *ht_create(uint32_t size , bool mtf) {
        HashTable *ht = (HashTable *)  malloc(sizeof(HashTable));
        if (ht) {
                // Leviathan
                ht->salt [0] = 0x9846e4f157fe8840;
                ht->salt [1] = 0xc5f318d7e055afb8;
                ht->size = size;
                ht->mtf = mtf;
                ht->lists = (LinkedList **) calloc(size, sizeof(LinkedList *));
                if (!ht->lists) {
                        free(ht);
                        ht = NULL;
```

```
            }
        }
        return ht;
}
```

Node ADT:

```
struct Node {
        char *oldspeak;
        char *newspeak;
        Node *next;
        Node *prev;
};
```

node_print() Oldspeak to Newspeak Print Statement:

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

node_print() Only Oldspeak Print Statement:

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

LinkedList ADT:

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

Files: (taken from asgn7-v2.pdf)

- **banhammer.c** *contains main() and may contain other functions necessary*
- **messages.h** *defines mixspeak, badspeak, and goodspeak messages used in banhammer.c*
- **speck.h** *defines interface for hash function using the SPECK cipher*
- **speck.c** *contains implementation of hash function using the SPECK cipher*
- **ht.h** *defines interface for the hash table ADT*
- **ht.c** *contains implementation of hash table ADT*
- **ll.h** *defines interface for the linked list ADT*
- **ll.c** *contains implementation of linked list ADT*
- **node.h** *defines interface for the node ADT*

- **node.c** *contains implementation of node ADT*
- **bf.h** *defines interface for the Bloom filter ADT*
- **bf.c** *contains implementation of Bloom filter ADT*
- **bv.h** *defines interface for the bit vector ADT*
- **bv.c** *contains implementation of bit vector ADT*
- **parser.h** *defines interface for the regex parsing module*
- **parser.c** *contains implementation of regex parsing module*
- **Makefile** *allows the grader to type make to compile the program. Typing make as well as make all builds all programs.*
    - *CFLAGS=-Wall -Wextra -Werror -Wpedantic included.*
    - *CC=clang specified.*
    - *make clean removes all files that are compiler generated.*
    - *make builds the banhammer executable, as does make all.*
    - *make format formats all source code, including the header files.*
- **README.md** *in Markdown. Describes how to use the program and Makefile.*
- **DESIGN.pdf** *design document*
- **WRITEUP.pdf** *a reflection containing graphs comparing the total number of seeks and average seek length as you vary the hash table and Bloom filter size, as well as an analysis of these graphs*

Top Level Pseudocode:

bf.c:

```
void bf_delete(BloomFilter **bf) {
        if bf exists {
                bv_delete(bf.filter);
                free bf;
                bf = NULL;
        }

        return;
}
```

```
uint32_t bf_size(BloomFilter *bf) {
        return bv_length(bf.filter);
}

void bf_insert(BloomFilter *bf, char *oldspeak) {
        bv_set_bit(bf.filter, hash(primary, oldspeak));
        bv_set_bit(bf.filter, hash(secondary, oldspeak));
        bv_set_bit(bf.filter, hash(tertiary, oldspeak));

        return;
}

bool bf_probe(BloomFilter *bf, char *oldspeak) {
        int bit0 = bv_get_bit(bf.filter, hash(primary, oldspeak));
        int bit1 = bv_get_bit(bf.filter, hash(secondary, oldspeak));
        int bit2 = bv_get_bit(bf.filter, hash(tertiary, oldspeak));

        if any of them are 1 {
                return true;
        }

        return false;
}

uint32_t bf_count(BloomFilter *bf) {
        int count = 0;

        for (int i = 0; i < bf.filter's length; i += 1) {
                counting set bits. bits are 1 if set and 0 otherwise, so can just add
                up the result of getting all bits and the result is the count.
                count += bv_get_bit(bf.filter, i);
        }

        return count;
}


void bf_print(BloomFilter *bf) {
        bv_print(bf.filter);
        return;
```

```
}
```

bv.c:
use bv.c implemented in asgn5 since it worked perfectly fine

ht.c:

```
void ht_delete(HashTable **ht) {
        if ht exists {
                for (int i = 0; i < size; i += 1) {
                        if ht.lists[i] exists {
                                ll_delete(ht.lists[i]);
                        }
                }

                free ht;
                ht = NULL;
        }

        return;
}

uint32_t ht_size(HashTable *ht) {
        return ht.size;
}

Node *ht_lookup(HashTable *ht, char *oldspeak) {
        return ll_lookup(ht.lists[hash(salt, oldspeak)], oldspeak);
}

void ht_insert(HashTable *ht, char *oldspeak, char *newspeak) {
        int index = hash(salt, oldspeak);

        if ht.lists[index] doesn't exist {
                ht.lists[index] = ll_create(ht.mtf);
        }

        ll_insert(ht.lists[index], oldspeak, newspeak);

        return;
}
```

```
uint32_t ht_count(HashTable *ht) {
	int count = 0;

	for (int i = 0; i < ht.size; i += 1) {
		if ht.lists[i] exists {
			count += 1;
		}
	}

	return count;
}

void ht_print(HashTable *ht) {
	for (int i = 0; i < ht.size; i += 1) {
		ll_print(ht.lists[i]);
	}

	return;
}
```

node.c:

```
static char *copy_string(char *orig) {
	int size = 0;
	char *copy;

	while orig[index] isn't '\0' {
		size += 1;
	}

	copy = allocate memory for size bytes;

	less than or equal to so that the null character is included
	for (int i = 0; i <= size; i += 1) {
		copy[i] = orig[i];
	}

	return copy;
}
```

```
Node *node_create(char *oldspeak, char *newspeak) {
	Node n = allocate memory for something size of a Node;

	if n exists {
		if oldspeak exists {
			n.oldspeak = copy_string(oldspeak);
		}

		if newspeak exists {
			n.newspeak = copy_string(newspeak);
		}

		n.next = NULL;
		n.prev = NULL;
	}

	return n;
}

void node_delete(Node **n) {
	if n exists {
		if n.oldspeak exists {
			free n.oldspeak;
			n.oldspeak = NULL;
		}

		if n.newspeak exists {
			free n.newspeak;
			n.newspeak = NULL;
		}

		free n;
		n = NULL;
	}

	return;
}

void node_print(Node *n) {
	if n.oldspeak and n.newspeak exist {
```

```
            printf("%s -> %s\n", n->oldspeak, n->newspeak);
        } else if oldspeak exists {
            printf("%s\n", n->oldspeak);
        } else {
            print something else that indicates that there's no old or newspeak;
        }

        return;
}
```

ll.c:

```
LinkedList *ll_create(bool mtf) {
        LinkedList ll = allocate memory for size LinkedList;

        if ll exists {
            ll.length = 0;
            ll.head = node_create();
            ll.tail = node_create();
            ll.head.next = ll.tail;
            ll.tail.prev = ll.head;
            ll.mtf = mtf;
        }

        return ll;
}

void ll_delete(LinkedList **ll) {
        if ll exists {
            Node current = ll.head;
            Node next = ll.head.next;

            while current is not NULL {
                node_delete(current);
                current = next;
                next = current.next;
            }

            free ll;
            ll = NULL;
        }
```

```
        return;
}

uint32_t ll_length(LinkedList *ll) {
        return ll.length;
}

Node *ll_lookup(LinkedList *ll, char *oldspeak) {
        Node thing = NULL;

        go through linked list {
                if current node's oldspeak equals the given oldspeak {
                        thing = current node;

                        if ll.mtf is true {
                                move current node to front, adjust each node's next
                                and prev accordingly;
                        }
                }
        }

        return thing;
}

void ll_insert(LinkedList *ll, char *oldspeak, char *newspeak) {
        go through linked list {
                if current node's oldspeak equals given oldspeak {
                        return;
                }
        }

        Node new = node_create(oldspeak, newspeak);
        ll.head.next.prev = new;
        ll.head.next = new;

        return;
}
void ll_print(LinkedList *ll) {
        Node current = ll.head;
```

```
        while current is not null {
                node_print(current);
                current = current.next;
        }

        return;
}
```

banhammer.c:
DEFINE OPTIONS: ‘h’ (print help msg), ‘t’ (specify hash table size), ‘f’ (specify bloom filter size), ‘m’ (enable move-to-front), ‘s’ (allow printing of stats)

```
int main(int argc, char **argv) {
        bool stats = false;
        bool mtf = false;
        int ht_size = 10000;
        int bf_size = 1048576

        while (getopt(argc, argv, OPTIONS) is parsing args) {
                switch (parsed arg) {
                case ‘h’:
                        print help msg;
                        end program;
                        break;

                case ‘t’:
                        ht_size = arg;
                        break;

                case ‘f’:
                        bf_size = arg;
                        break;

                case ‘m’:
                        mtf = true;
                        break;

                case ‘s’:
                        stats = true;
```

```
                break;

        default:
                break;
        {
}

BloomFilter bf = bf_create(bf_size);
HashTable ht = ht_create(ht_size, mtf);
open badspeak.txt;

while there are still words to read in using fscanf() {
        bf_insert(bf, word);
        ht_insert(ht, word, NULL);
}

close badspeak.txt;

open newspeak.txt;

while there are still words to read in using fscanf() {
        bf_insert(bf, oldspeak);
        ht_insert(ht, oldspeak, newspeak);
}

close newspeak.txt;

LinkedList *badspeak = ll_create(mtf);
LinkedList *oldspeak = ll_create(mtf);

while next_word(stdin, [regex]) is not NULL {
        if bf_probe(bf, word) returns true {
                if ht_lookup(ht, word) is not NULL {
                        if the node has newspeak {
                                ll_insert(oldspeak, node.oldspeak,
                                                node,newspeak);
                        } else if the node has only oldspeak {
                                ll_insert(badspeak, node.oldspeak, NULL);
                        }
                }
```

```
            }
        }

        if stats is false {
            if both lists are populated {
                print mixspeak message;
                ll_print(badspeak);
                ll_print(oldspeak);
            } else if only badspeak is populated {
                print badspeak message;
                ll_print(badspeak);
            } else if only oldspeak is populated {
                print goodspeak message;
                ll_print(oldspeak);
            }
        } else if stats is true {
            print stats;
        }

        delete both linked lists;
    }
```

Changes:

- very slightly changed implementation of bf_probe in bf.c so that it doesn't loop through an array
- changed copy_string to return a char pointer
- very slightly changed loop logic when deleting LinkedList nodes so that it checks if current node is not NULL rather than current's next node
- slightly changed ll_print so that it includes the sentinel nodes
- added banhammer.c
- added some details to program description