Aditya Agrawal
aagraw14@ucsc.edu

CSE13s: Spring 2021
Assignment 7: The Great Firewall of Santa Cruz
Design Document

This program will implement a bloom filter to filter out 'badthink' in place of the accepted 'newspeak'. This will be done using multiple ADTs including BloomFilter, hash tables, bit vectors, and linked lists as well as a parser module that will lexically analyze the input stream.

banhammer.c:
This file will contain the main() function and follow the procedure outlined in the assignment pdf.

```
int main(){
        loop calls to getopt for command-line options
        initialize bloom filter and hash table
        read in badspeak from badspeak.txt
        read in oldspeak and newspeak from newspeak.txt
        read in words from stdin using regex and parsing module
                WORD = "[a-zA-Z0-9_]+([- | '][a-zA-Z0-9_]+)*"
        determine if mixspeak, badspeak only, or oldspeak only
        print message and occurrences of transgressions
        delete and close files
}
```

ht.c:
This file will contain the hash table ADT and all supporting functions.

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

HashTable *ht_create(uint32_t size, bool mtf);

```c
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;
}
```

```
void ht_delete(HashTable **ht)
        if(ht exists)
                for list in lists:
                        ll_delete(list)
                free(ht)

uint32_t ht_size(HashTable *ht)
        return ht->size

Node *ht_lookup(HashTable *ht, char *oldspeak)
        index = hash(salt, oldspeak) % ht_size
        if( ht->lists[index] does not exist)
                return NULL
        return ll_lookup(ht->lists[index], oldspeak)

void ht_insert(HashTable *ht, char *oldspeak, char *newspeak)
        index = hash(salt, oldspeak) % ht_size
        if( ht->lists[index] does not exist)
                ht->lists[index] = ll_create
        ll_insert(ht->lists[index], oldspeak)

uint32_t ht_count(HashTable *ht)
        for 0 to ht_size
                if(ht->lists[i] exists)
                        count += 1
```

return count

ll.c:

This file will implement the linked list ADT and all supporting functions.

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

extern uint64_t seeks; // Number of seeks performed.
extern uint64_t links; // Number of links traversed.

Node *ll_lookup(LinkedList *ll, char *oldspeak)
    for nodes in ll
        if(oldspeak == node->oldspeak)
            if(mtf)
                move to front
            return node
    return NULL

void ll_insert(LinkedList *ll, char *oldspeak, char *newspeak)
    if(!ll_lookup)
        create node
        place at head(instructions in linked list slides)

node.c

This file will implement the node ADT which is to be utilized by linked lists, similar to other assignment.

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

typedef struct Node Node;

struct Node {
  char *oldspeak;

```
    char *newspeak;
    Node *next;
    Node *prev;
};
```

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

void node_delete(Node **n);

void node_print(Node *n);

bf.c

This file will contain the BloomFilter ADT and all supporting functions for it.

typedef struct BloomFilter BloomFilter;

```
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 *bf_create(uint32_t size);

```
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;
}
```

```
uint32_t bf_size(BloomFilter *bf)
        return bv_length(bf->filter)

void bf_insert(BloomFilter *bf, char *oldspeak)
        hash(primary salt, oldspeak)
        bv_set_bit(hash)
        hash(secondary salt, oldspeak)
        bv_set_bit(hash)
        hash(tertiary salt, oldspeak)
        bv_set_bit(hash)

bool bf_probe(BloomFilter *bf, char *oldspeak)
        primary = bv_get_bit(hash(primary))
        secondary = bv_get_bit(hash(secondary))
        tertiary = bv_get_bit(hash(tertiary))
        if(primary and secondary and tertiary)
                return true
        else
                return false
```

```
uint32_t bf_count(BloomFilter *bf)
        for 0 to bf_size
                if( bv_get_bit(i))
                        count += 1
void bf_print(BloomFilter *bf);
```

bv.c

This file will implement the bitvector ADT and all the supporting functions like from previous assignments.

parser.c

This file will contain the parinsing module and make use of a regular expression.