Bonifacio Miranda
CSE 13S

Summary/Purpose:

The purpose of this program is to create multiple double linked lists around a given hash function and for us to make the bloom filter. This program will censor a user's input by a list of bad words.

Banhammer.c

```
int main(int argc, char **argv) {

   // Check for invalid bf/ht size

   // Create bf and ht

   // Add badspeak words to bf and ht

   // Add newspeak translations to bf and ht

   // Create lists for storing transgressions found

   // Compile regex

   // Parse words from st
            // Make word lowercase
            // Ignore word if bf returns false
            // Otherwise, check if word is in ht
                // Record transgression if word is in ht

   // If stats enabled by user then print stats

   // Free memory for bf and ht

       // "If the citizen is accused of thoughtcrime
       // and requires counseling on proper Rightspeak ..."
            // Print mixspeak message
```

```
        // "If the citizen is accused soley if thoughtcrime ..."
            // Print badspeak message

        // "If the citizen only requires counseling ..."
            // Print newspeak message

        // Print the lists of transgressions found

    // Free memory for lists of transgressions found

    // Close files
}
```

Bf.c

```
// From assignment PDF
BloomFilter *bf_create(uint32_t size) {
    // Create new Bloom filter of given size
        // Grimm's Fairy Tales
        // The Adventures of Sherlock Holmes
        // The Strange Case of Dr. Jekyll and Mr. Hyde
    }
    return bf;
}

void bf_delete(BloomFilter **bf) {
    // Free Bloom filter and its contents
}

uint32_t bf_size(BloomFilter *bf) {
    // Return number of bits in filter
}

void bf_insert(BloomFilter *bf, char *oldspeak) {
    // Set 3 bits in Bloom filter at hash indices from oldspeak
}
```

```
bool bf_probe(BloomFilter *bf, char *oldspeak) {
   // Return if true if all 3 bits are 1
}

uint32_t bf_count(BloomFilter *bf) {
   // Count number of set bits
}

void bf_print(BloomFilter *bf) {
   // Display contents of Bloom filter
}
```

Bv.c

```
BitVector *bv_create(uint32_t length) {
   // Create new bit vector
           // Add an extra byte if number of bits is not divisible by
}

void bv_delete(BitVector **bv) {
   // Free bit vector and its contents
}

uint32_t bv_length(BitVector *bv) {
   // Return number of bits
}

void bv_set_bit(BitVector *bv, uint32_t i) {
   // Set bit at index i
}

void bv_clr_bit(BitVector *bv, uint32_t i) {
```

```
    // Clear bit at index i
}

uint8_t bv_get_bit(BitVector *bv, uint32_t i) {
    // Return bit at index i
}

void bv_print(BitVector *bv) {
    // Display contents of bit vector
}
```

Ht.c

```
// From assignment PDF
HashTable *ht_create(uint32_t size, bool mtf) {
    HashTable *ht = 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) {
    // Free hash table and its contents
}

Node *ht_lookup(HashTable *ht, char *oldspeak) {
    // Return node with given oldspeak if found, NULL otherwise
        // Don't lookup if linked list does not exist
}
```

```c
void ht_insert(HashTable *ht, char *oldspeak, char *newspeak) {
   // Insert given oldspeak and newspeak pair into hash table
}

uint32_t ht_count(HashTable *ht) {
   // Count the number of linked lists in hash table
}

void ht_print(HashTable *ht) {
   // Display contents of linked list
}
```

ll.c

```c
LinkedList *ll_create(bool mtf) {
   // Create a new linked list
}

void ll_delete(LinkedList **ll) {
   // Free linked list and its contents
}

uint32_t ll_length(LinkedList *ll) {
   // Return number of nodes not including sentinel nodes
}

Node *ll_lookup(LinkedList *ll, char *oldspeak) {
   // Return node if found, null otherwise
       // (Consider changing later)
           // Move-to-front technique inspired by Eugene's section
               // Bridge over n
               // Insert n at front
}

void ll_insert(LinkedList *ll, char *oldspeak, char *newspeak) {
   // Insert node with given oldspeak and newspeak
       // Do not insert if already exists or if node_create fails
   // Insert n at front
}

void ll_print(LinkedList *ll) {
   // Display contents of linked list ll
```

```
}
```

Node.c

```
static char *mystrdup(char *s) {
   // Create a duplicate of source string s
}

Node *node_create(char *oldspeak, char *newspeak) {
   // Create a new node with given oldspeak and newspeak
}

void node_delete(Node **n) {
   // Free node if not already null
}

void node_print(Node *n) {
   // Display contents of node n
}
```