Design Document

Assignment 7

Purpose

Create a program that serves as a censorship firewall. The program will read words with regex from standard input and print a message with a list of the user's transgressions. The program will also print statistics if specified. The program will determine if each word is allowed by searching for it in a database using a Bloom filter and a Hash table.

Pseudocode

My pseudocode will illustrate the **layout/structure** and will provide a **description/explanation** of how each part of the program works.

banhammer.c

```
Parse command-line arguments

 Phase 1:
 Construct Bloom filter and hash table
 from badspeak.txt and newspeak.txt

 Initialize Bloom filter and hash table

 Read badspeak words from badspeak.txt with fscanf()
     Convert word to lowercase
     Convert badspeak word to lowercase
     Add badspeak word to Bloom filter
     Add badspeak word to hash table

 Close badspeak.txt because it will not be needed again

 Read translations from newspeak.txt with fscanf()
 Convert oldspeak to lowercase
 Convert newspeak to lowercase
 Add oldspeak to Bloom filter
 Add translation to hash table
 Close newspeak.txt because it will not be needed again

 Use linked lists to keep track of user's transgressions

 Phase 2:
   Apply firewall to input from `stdin`

   Compile regex

   Read words from `stdin` using parsing module
       Convert word to lowercase
       If word is in bf
           If word is in ht
               Add it to corresponding linked list

   Phase 3:
   Print output to `stdout`

       Print statistics if enabled

       Print appropriate message

       Print transgressions

   Free memory
```

node.c

```
Node *node_create(char *oldspeak, char *newspeak) {
   Allocate memory for a new node `n`
   Set n->oldspeak
   Set n->newspeak
   Set the `next` and `previous` fields to null for now
   Return the new node `n`
}

void node_delete(Node **n) {
       Free memory allocated for oldspeak
       Free memory allocated for newspeak
       Free the node
       Set pointer to null

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

ll.c

```
Number of seeks
and number of links traversed
are initially zero

LinkedList *ll_create(bool mtf) {
   Allocate memory for a new linked list
   Return null if malloc failed
   Initialize length of linked list to zero
   Create head and tail sentinel nodes
   Link the head and tail together
   Set move-to-front status boolean
   Return the newly created linked list
}

void ll_delete(LinkedList **ll) {
   Free the memory allocated for the linked list
   if it exists and is not already null
       Free every node in the linked list
           First, save the head node
           Shift the head over
           Delete the old head node that you saved
           Repeat until the head becomes null,
           meaning we have reached the far end of the linked list
       Free the rest of the linked list
       Set the pointer to the linked list to null
}

uint32_t ll_length(LinkedList *ll) {
   Return the linked list's `length` property,
   which is the number of nodes in `ll` (not including the sentinel nodes)
}

Node *ll_lookup(LinkedList *ll, char *oldspeak) {
   Immediately count this seek
   Return null if list does not exist
   Start at the node after the head
   Traverse the linked list until
       either the end of the linked list is reached
       or the matching key (oldspeak) is found
           Move found node `n` to front if `mtf` is enabled
               Bridge over `n`
               Re-insert `n` after head of linked list
       Traverse to next node
       (And count this traversal/seek)
   }
}

void ll_insert(LinkedList *ll, char *oldspeak, char *newspeak) {
   Do not insert if `ll` already contains a node with matching `oldspeak`
   Create a new node with given `oldspeak` and `newspeak`
       Exit if malloc failed

   Link new node n into `ll`
}
```

```
static void print_word(Node *n, char *word) {
   Get width `w`
   If word is not null,
       Get word with quotes `q`
       Print word with quotes
       Else word is null so print "Null"
}

void ll_print(LinkedList *ll) {
       Print "Null" if `ll` is null
       Print top row of contents
       Print bottom row of contents
}
```

ht.c

```
HashTable *ht_create(uint32_t size, bool mtf) {
   (Given by PDF)
}

void ht_delete(HashTable **ht) {
   Free memory if `ht` and `*ht` exist
   and are not already null
       Free memory allocated
       for the linked lists in the hash table
           Delete every linked list
       Free the rest of the hash table
       Null out the pointer that was passed in
}

uint32_t ht_size(HashTable *ht) {
   Return the size (number of slots for linked lists)
}

Node *ht_lookup(HashTable *ht, char *oldspeak) {
   Hash to get the right index
   Perform lookup of `oldspeak` at that index
}

void ht_insert(HashTable *ht, char *oldspeak, char *newspeak) {
   Hash the key `oldspeak` to get index `i`
       Initialize the list at `i` if it is null
       Perform insertion on the linked list at `i`
}

uint32_t ht_count(HashTable *ht) {
   count number of linked lists
}

void ht_print(HashTable *ht) {
   print contents
}
```

bf.c

```
BloomFilter *bf_create(uint32_t size) {
   (Given by PDF)
}

void bf_delete(BloomFilter **bf) {
   Free memory if `bf` and `*bf` exist
   and are not already null
       Free filter if it exists and is not already null
       Null out the pointer that was passed in
}

uint32_t bf_size(BloomFilter *bf) {
   Return the length of the filter (the underlying bit vector)
}

void bf_insert(BloomFilter *bf, char *oldspeak) {
   Set bits in filter at each index given by each hash
}

bool bf_probe(BloomFilter *bf, char *oldspeak) {
   Return true if there is a 1 at all indices given by hashing oldspeak
}

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

void bf_print(BloomFilter *bf) {
      Print bits in bloom filter

}
```