Pre-Lab

```
1. Inserting an element in bloom filter
        Hash oldspeak 3 times with the three salts - returns indices
        Set the bits at those indices in the bit vector
   Deleting an element from bloom filter
        Hash oldspeak 3 times with the three salts - returns indices
        clear the bits at those indices in the bit vector (making them = 0)
```

2.

Linked List

```
LinkedList *ll_create(bool mtf)
    LinkedList *ll = malloc(sizeof(LinkedList))
    ll->length = 0
`   ll->head = node_create(NULL, NULL)
    ll->tail = node_create(NULL, NULL)
    ll->head->next = ll->tail
    ll->tail->prev = ll->head
    ll->mtf = mtf
    Return ll

void ll_delete(LinkedList **ll)
    Go through each node and free it
    Free the list
    Set pointer to NULL

uint32_t ll_length(LinkedList *ll)
    Going to return length
    Return ll->length

Node *ll_lookup(LinkedList *ll, char *oldspeak)
    Go through each node in ll until it hits tail node
    For (Node *n = ll->head->next; n! = ll->tail; n = n->next)
        Compare n->oldspeak and oldspeak
        See if that particular node is in that function
        If you find the node then return the pointer to that node
            If mtf is true then perform mtf

            Return the node
    Else:
        Return NULL
```

```
void ll_insert(LinkedList *ll, char *oldspeak, char *newspeak)
        Inserts a node after the head sentinel node into the ll
        Node *n = node_create(oldspeak, newspeak)
        n->prev = ll->head
        n->next = ll->head->next
        ll->head->next->prev = n (node to insert)
        ll->head->next = n (node to insert)
        ll->length += 1
```

```
void ll_print(LinkedList *ll)
        Iterating over linked list and printing it (for loop)
        Printing out each node using node_print
```

Pre-lab

3. [a-z-A-Z0-9_]+((‘|-)[a-zA-Z0-9_]+)*

Bloom Filter

```
BloomFilter *bf_create(uint32_t size)
        Allocate memory with sizeof length with calloc
        If it fails return Null
        Else return a pointer to the bitvector
```

```
void bf_delete(BloomFilter **bf)
        Free memory for bloom filter
        Free *bf
        Set pointer to NULL
```

```
void bf_size(BloomFilter *bf)
        Returns length of bit vector
```

```
void bf_insert(BloomFilter *bf, char *oldspeak)
        Hash oldspeak 3 times with the three salts - returns indices
        Set the bits at those indices in the bit vector
```

```
bool bf_probe(BloomFilter *bf, char *oldspeak)
        Hash oldspeak 3 times with the three salts to get three indices
        Check those indices to see if set in vector
        AND them together
        If 1 then return True
        Else return False

void bf_print(BloomFilter *bf)
        Print out the vector of the bloom filter
```

Bit Vectors

```
BitVector *bv_create(uint32_t length)
        Allocate memory for the number of bits in length
        Use calloc() to set everything to 0
        If it fails return NULL
        Otherwise returns a pointer to a bit vector

void bv_delete(BitVector **bv)
        Free the vector then free the pointer
        Set the pointer to NULL

uint32_t bv_length(BitVector *bv)
        Returns length of bitvector
        Return -> length

void bv_set_bit(BitVector *bv, uint32_t i)
        Divide the index by 8 to get location in vector
        Create a mask by shifting a bit over to location
        Set the bit by OR it with the mask
        Anything OR 1 = 1

void bv_clr_bit(BitVector *bv, uint32_t i)
        Divide the index by 8 to get location in vector
        Create a mask by shifting a bit over to location and inverting it (0)
        clear the bit by AND it with the mask
        Anything AND 0 = 0

uint8_t bv_get_bit(BitVector *bv, uint32_t i)
        Access the bit and create the mask same as set_bit
        Return the result of inverting mask AND bit

void bv_print(BitVector *bv)
        Loop through all bits and print each one
```

Hash Table

```
HashTable *ht_create(uint32_t size, bool mtf)
```

```
    (provided in lab doc)

void ht_delete(HashTable **ht)
    Free all of the linked lists
    Free the pointer
    Set the pointer to NULL

uint32_t ht_size(HashTable *ht)
    Returns the size of the hash table
    ht->size

Node *ht_lookup(HashTable *ht, char *oldspeak)
    Hash the oldspeak input to get an index
    Go to the index in the hash table
    If the node is found the return return pointer
        Else return NULL


void ht_insert(HashTable *ht, char *oldspeak, char *newspeak)
    Hash oldspeak input to get index for hash table
    Create a linked list of none is created yet
    Insert oldspeak followed by newspeak into the linked list in the hash table at the index


void ht_print(HashTable *ht)
    Loop through hash table and print out contents from ll
```

Node

```
Node *node_create(char *oldspeak, char *newspeak)
    Node *n = (Node*) calloc(sizeof(Node))
    n->oldspeak = oldspeak
    n->newspeak = newspeak
    n->next = NULL
    n->prev=NULL

void node_delete(Node **n)
    Free the pointer
    Set the pointer to NULL

void node_print(Node *n)
    If it only has bad speak print only bad speak
    If it has both badspeak and newspeak print both
```

Moving to front (used in linked list)

```
    Move all 6 pointers between head, tail, and specific element
    Like swapping pointers to elements in linked list
```

Banhammer (contains main)

```
Initialize bloom filter and hash table by calling create()

Loop through words in badspeak.txt using fscanf
    Insert the word into the bloom filter
    Insert the word into the hash table


Loop through the pair of oldspeak and newspeak pairs from newspeak.txt using fscanf
    Insert oldspeak into bloom filter
    Insert the pair into the hash table

Read the words in from stdin using parsing module

Have 2 linked lists, one for words than cannot be corrected (bad words)
And one that has a bad word and a translation

Loop through each word:
    Probe the words to see if added to bloom filter
    If it's not in the bloom filter then continue
    If it is in the bloom filter:
        See if it is in the hash table, if not continue
            If it is then add it to the appropriate linked list.
            Depending on if the word has a translation or not return appropriate text
```

Purpose of Program

The purpose of this program is to filter out bad words from text files.

The output of the program is a string that says which bad words were used in the text file input. The output will depend whether a bad word is in the document or a bad word that has a translation is in the document.

How each file connects

This program has a lot of moving parts that work together in order for the main function to work.

- Node.c is used for creating nodes in the linked lists and printing nodes within the linked list
- ll.c is used for the hash table in order to keep track of what nodes are added into the hash table
- bv.c is a bit vector for keeping track of what has been added to the bloom filter.
- The bloom filter works as a more efficient way than the hash table for checking if an element has NOT been added. The bloom filter cannot report false negatives.
- Once a word gets past the bloom filter is it double checked by the hash table to make sure it was genuinely added.
- The bloom filter is used to speed up the process because looking through the linked lists within the hash table is highly inefficient.
- Banhammer.c is the main function which reads in the words from badspeak and newspeak and inserts them into the hash and bloom filter. It then reads each word from an input file and checks if the word is in both the bloom and hash table. It then inserts the bad word that was used into a linked list to keep track of bad words used. It then prints out the bad words used and if they have a translation.