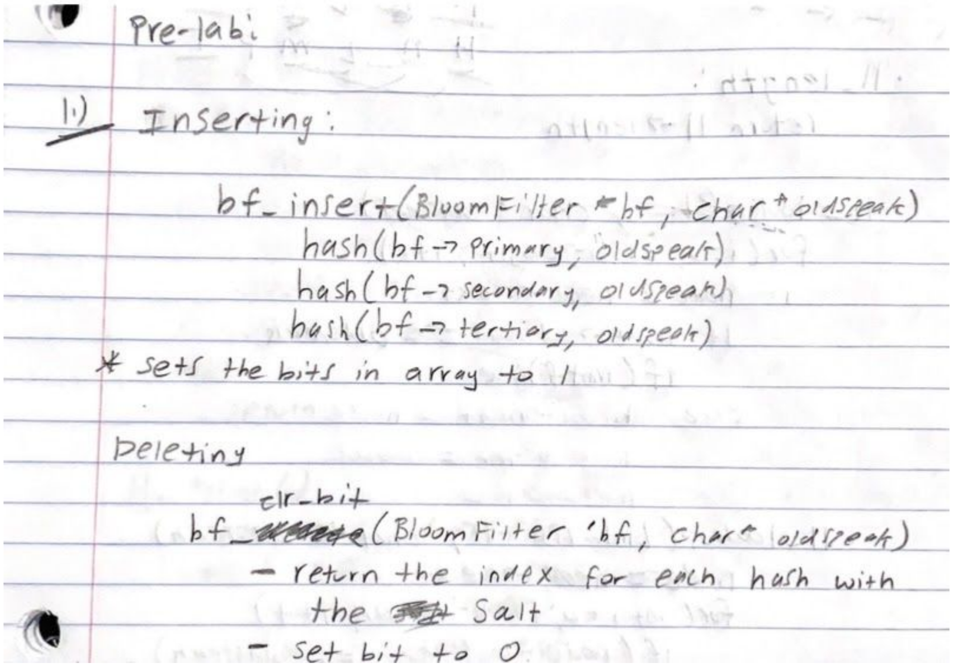
Version 1.0

Section 0: Pre-Lab



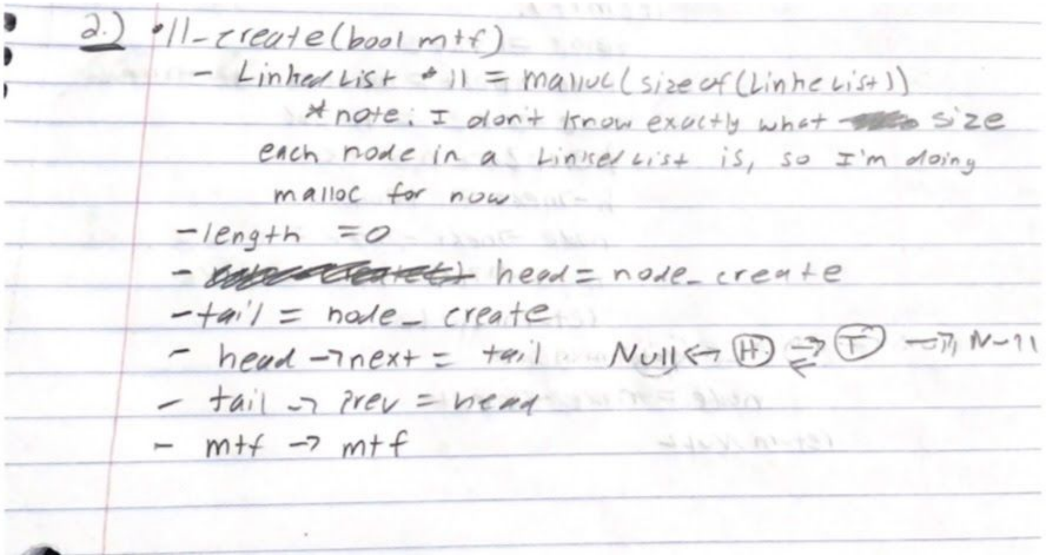
Pre-lab:

1.) Inserting:

```
        bf_insert(BloomFilter *bf, char* oldspeak)
            hash(bf -> primary, oldspeak)
            hash(bf -> secondary, oldspeak)
            hash(bf -> tertiary, oldspeak)
* sets the bits in array to 1
```

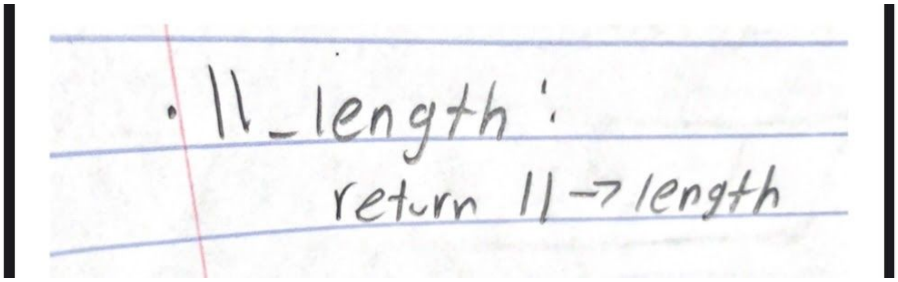
Deleting

```
          clr_bit
    bf_~~delete~~(BloomFilter *bf, char* oldspeak)
        - return the index for each hash with
          the ~~[illegible]~~ Salt
        - set bit to 0.
```



2.) • ll_create(bool mtf)

```
    - LinkedList *ll = malloc(sizeof(LinkedList))
          * note: I don't know exactly what ~~[illegible]~~ size
        each node in a LinkedList is, so I'm doing
        malloc for now
    - length = 0
    - ~~[illegible]~~ head = node_create
    - tail = node_create
    - head -> next = tail      NULL <- (H) ⇄ (T) -> NULL
    - tail -> prev = head
    - mtf -> mtf
```



• ll_length:

```
    return ll -> length
```

```
• ll_lookup(LinkedList *ll, char * oldspeak)
      node = head->next
      for( int i = 0; i < ll->length; i++)
          if( node->oldspeak == oldspeak):
              if(mtf):
                  temp = node
                  node->next = head->next->next
                  node->prev = head
                  temp2 = h->next
                  h->next = temp
                  node->next->prev = node
                  node->prev->next = node
                  return node
          else return node
        node = node->next
    return NULL
```

```
• ll_insert
  If( ll_lookup != Null)
        node = head->next
        node->prev = head
        node->next = head->next
        head->next->prev = n
        head->next = n
  else
      return

  ll_print(ll)
      node = head->next
      for( i = 0 ; i < ll->length - 1 ; i++ )
          node_print(node)
          node = node->next
```

3.) ~~[a-[A-F~~

```
[a-zA-Z0-9\-_"'] +
```

Version 2.0

Section 1: Overview

- This program utilizes multiple ADT’s, such as Linkedlists, a Hashtable, Bit Vectors, and a Bloom Filter to gather user data and filter it such that specific words are checked for being allowed but an alternative must be used, or banned altogether.
- In order to implement this, we first create The Node ADT which stores its linear neighbors and the values of oldspeak and newspeak, data representing the words that are allowed/substituted or banned.
- We then construct a LinkedList ADT which is a chain of nodes, pointing to one another noted as a “doubly linked list”. This will be used in each hash table index for avoiding key collisions.
- The Hashtable itself consists of indexes that are each a linked list or NULL until a data value is hashed to that index, then a linked list is created in that place with its first value the key. It's essentially an array of linked list pointers.
- The BitVector ADT is what our bloom filter’s filter will be, essentially a linear array of 1’s and 0’s that are accessed by the hash function and will determine if a word is definitely in the hashtable. False positives will be natural, however false negatives will never occur.
- This particular bloom filter will have a function from speck.c called “hash” hash to a certain index, 3 times for a word.
- The main file will implement all the main functions, remember to close all files and delete all memory allocations. Reads words from newspeak and oldspeak.txt and stores them in bf. If a word read from stdin appears to be in the bf, then check the ht. If not in ht then do nothing.

Section 2: Main Structure and Layout Overview

- node.c
    - First make a function for duplicating a string, essentially allocating enough bytes to store a word that will be stored in either a newspeak or oldspeak.

  - A function to create a node, assigning it a newspeak and oldspeak words. Must also set the next and previous nodes to null. A node will either have an oldspeak and a newspeak or just an oldspeak.
  - A function for deleting a node. Free only the newspeak or oldspeak if they are NOT null.
  - A helper function to print the node
- ll.c
  - A function to create a linked list. The initial length must be 0, until nodes are added. The head and tail nodes will be null.
  - A function to delete a linked list, free each node in the list by creating 2 temporary nodes to keep track of which node you are on and what's next.
  - A function to return a lists length.
  - A function to lookup a node in a list and return the node that matches the inputted oldspeak word. If the node with the word is found, return the node, else return NUll. Use strcmp() to make comparing the inputted oldspeak and node's oldspeak easier.
  - A function to insert a node after the head of a list. This function stores any word inputted into the linked list as a node, with a newspeak or a "null speak" translation.
  - A helper function to print a linked list, great for seeing whether nodes inserted actually move to the front. Was great for troubleshooting.
- hash.c
  - The create function to construct the hashtable with specified elements. It should contain a list of pointers to linked lists.
  - A function to delete a hash table, which should call upon the linkedlist delete function for each ll in the ht->lists array. Then standard frees.
  - A function to return the size of the hash table, which is the size of ht->lists, but it's also specified upon its creation so easy stuff.
  - A function to look up an old/newspeak pair node in the table. This function will first check to see if the

index is null, if so then return null. If not, then return a ll_lookup.
    - A function to insert a node into the hashtable. This will find the correct index by calling upon the hash function in speck.c to find the linked list at the array index and perform a ll_insert.
    - A helping function to print a hashtable.
- bv.c
    - I will define a macro BITS_PER_UNIT to be 8.
    - Also, I will include a helper function called bytes which converts a number of bits to bytes. Helpful for accessing indices and allocating enough spots in the bv vector.
    - A function to delete the bv ADT by freeing the pointer to bv->vector first then the bv. Set to null.
    - Functions to set/clr/get bits from the vector. These are all similar to what we did in the previous labs. Simple binary manipulation to get a bit from a byte in the array.
    - A function to return the length of the bv.
    - A helper function to print the bv.
- bf.c
    - For the bloomfilter adt, the code is already supplied for the creation. Set the 3 salt arrays and allocate memory for the bloomfilter's filter called bf->filter.
    - The function for deleting the bloom filter will use bv_delete and get set to null.
    - Bf_length will return the length of the bv by using the bv function bv_length.
    - Bf_insert will find the 3 filter indexes by using the hash() function 3 times with each salt array. Use bv_setbit to set the indices to 1.
    - Bf_probe will use the bv function get_bit in a similar fashion to bf_insert. Find the 3 indices and use get_bit to return the bit value. Useful for checking if a word is in a filer. This part will guarantee a word is not in the filter, but also might return false positives.
    - A helper function to print the bloom filter's filter.

- banhammer.c
    - Define all the necessary macros: OPTIONS, WORD, HT_SIZE, BF_SIZE, BUFF_SIZE
    - The regex function I will be using doesn’t account for case sensitivity, so I created a function to return a word in all lower case. If i experiment some more I’m sure I could get the right regex pattern, just not familiar enough with it.
    - The main part of the function:
        - Initialize sizes and conditions for bf and ht. Parse command line for arguments.
        - Create the filer and table. Return if either fails.
        - Parse the badspeak and newspeak.txt files for the words to be added to the filter and hashtable.
        - Compile the regex pattern and create 2 linkedlists that will hold all the badspeak and oldpseak words used from stdin.
        - For each word from stdin, as long as it matches the regex pattern, check the bloom filter for the indices of the word to be set. If they are then check the hashtable for the word.
        - Print the correct statement depending on the words used from the user.
        - Free the linkedlists created for badspeak and oldspeak words, then delete the hashtable and bf ADT’s. Finally free any memory associated with regcomp.