Pre-Lab Part1

Constructor for a BloomFilter

```
BloomFilter *bf_create(uint32_t size);
```
-Provided

Destructor for a BloomFilter

```
Void bf_delete(BloomFilter *bf) {
  bv_delete(bf->filter)
  free(bf)
}
```

Sets the bits for the key according to the hash function

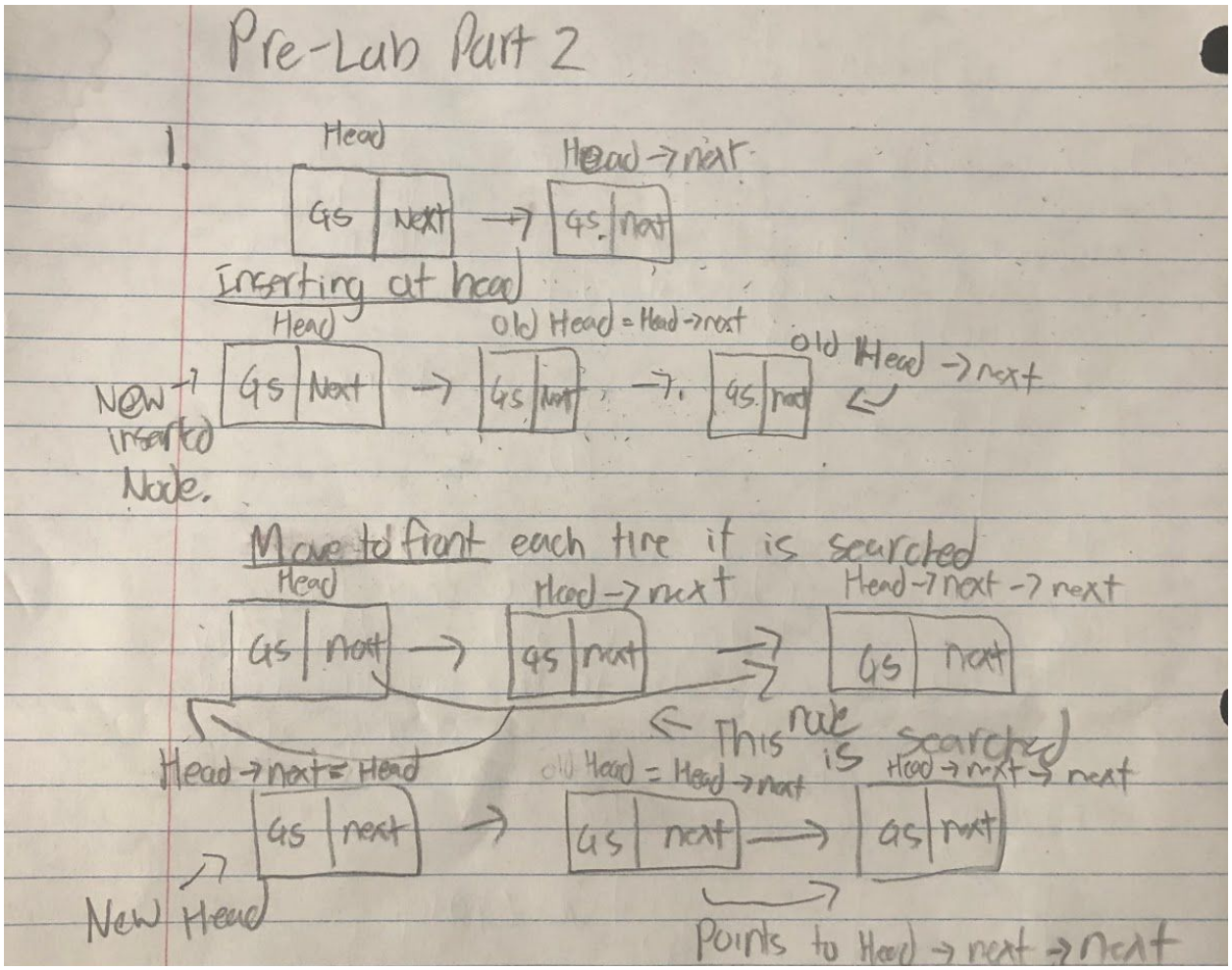
```
Void bf_insert(BloomFilter *bf, char *key) {
  Uint32_t index1 = Hash the key with primary salt % length of bf
  Uint32_t index2 = Hash the key with secondary salt % length of bf
  Uint32_t index3 = Hash the key with tertiary salt % length of bf
  Set bit (bf->filter, index1)
  Set bit (bf->filter, index2)
  Set bit (bf->filter, index3)
}
```

Probes a BloomFilter to check if a key has been inserted.

```
Bool bf_probe(BloomFilter *bf, char *key) {
  Uint32_t index1 = Hash the key with primary salt % length of bf
  Uint32_t index2 = Hash the key with secondary salt % length of bf
  Uint32_t index3 = Hash the key with tertiary salt % length of bf
  Uint8_t bit1 = bv_get_bit(bf->filter, index1)
  Uint8_t bit2 = bv_get_bit(bf->filter, index2)
  Uint8_t bit3 = bv_get_bit(bf->filter, index3)
  If (bit1 && bit2 && bit3) {
    Return true
  }
  Else {
    Return false
  }
}
```

Deletes elements from a BloomFilter
Void bf_delete(BloomFilter *bf, char *key) {
  Uint32_t index1 = Hash the key with primary salt % length of bf
  Uint32_t index2 = Hash the key with secondary salt % length of bf
  Uint32_t index3 = Hash the key with tertiary salt % length of bf
  clear bit (bf->filter, index1)
  clear bit (bf->filter, index2)
  clear bit (bf->filter, index3)
}

2. For creating a bloom filter with m bits and k hash functions, the insertion and search will take O(k) time. The space of the actual data structure will be O(m).



2.
This function creates a new linked list nodes

```
ListNode *ll_node_create(GoodSpeak *gs) {
        ListNode *newlistnode = (ListNode *) malloc the size of listnode +1
            newlistnode ->gs = gs_create(gs_oldspeak(gs), gs_newspeak(gs));
            newlistnode->next = NIL;
            Return newlistnode;
}
```

Deconstructors for the listnodes

```
Void ll_node_delete(ListNode *n) {
            free(n->next);
            gs_delete(n->gs)
            free(n->gs)
            free(n)
            return
}
```

Deletes the Linked list of listnodes

```
Void ll_delete(ListNode *head) {
            If (head == NULL) {
                        return
            }
            while(head != NULL) {
                        ListNode *temp = head->next
                        Listnode delete(head)
                        Head = temp
            }
            Listnode delete(temp)
}
```

Inserts a new ListNode to the head of the linked list.

```
ListNode *ll_insert(ListNode **heads, GoodSpeak *gs) {
            ListNode *new_head = ll_node_create(gs)
            new_head->next = pointer of head
            Pointer to the head = new_head
            Return pointer to the head
}
```

Searches for a specific key in the linked list. Returns the listnode if found. If move to front is true, The searched listnode is placed at the head.

```
ListNode *ll_lookup( ListNode **head, char *key) {
            If (*head != NULL) {
                        ListNode *temp = *head
```

```
                While(temp != NULL && strcmp(gs_oldspeak((temp->next)->gs), key)) {
                        If (movetofront = true) {
                                If (temp->next != NULL &&
strcmp(gs_oldspeak((temp->next)->gs), key)) {
                                        Listnode temp1 = temp->next
                                        temp->next = temp1->next
                                        ll_insert(head, temp1->gs)
                                        ll_delete(temp1)
                                }
                        }
                }
                Return temp
        }
        Return (ListNode *) NIL
}

Prints the listnode n
Void ll_node_print(ListNode *n) {
        If (gs_newspeak(n->gs) != NULL ) {
                Print the old speak and new speak
        }
        Else {
                Print the old speak
        }
}
Prints the linked list
Void ll_print(ListNode *h) {
        While(head != NULL) {
                ll_node_print(head)
                Head = head->next
        }
}
```

Objective of this Lab:

1. Creating GoodSpeak struct to store translations in
2. Creating a linked list and list nodes to store the GoodSpeak Structs into
3. Creating a BloomFilter to set the bits for any char* for fast and memory efficient checks to see if that char* has been inserted into the bit vector of the bloom filter
4. Using salts to hash keys
5. Hashing keys to save memory and if the index is occupied, using linked lists to place the list node into the hased index within the hashtable
6. Reading a file using regex and determining if they words within the inputfile has translations that are in the hashtable

7. Printing the appropriate message depending on if forbidden words have been used or words that has translations were in the input file.

High-level Description:

1. Storing the bad speak words and newspeak words as a link node into the hashtable and sets the bloom filter's bitvector using three different hash functions
2. Read the input file word by word and determine if that word is in the hashtable and the bloomfilter.
    a. If it is in both, determine if the word is a forbidden word or it has a translation
3. Prints the correct message depending on forbidden words have been used or words with newspeak translations has been used or both
    a. Messages: Encouragement message, Thoughtcrime message or both

Given in the Lab file

```
Hashtable *ht_create(uint32_t length) ;
```

Free the memory thats allocated for the hashtable

```
ht_delete(HashTable *ht) {
        ll_delete(*ht->heads)
        free(ht)
        return
}
```

Searches for the key within the hash table and returns the head once its found

```
ListNode *ht_lookup(HashTable *ht, char *key) {
        Hashindeex = hash(salt, key) % hash length
        If (ll_lookup(&ht->heads[hashindex], key != Null)) {
                Return ll_lookup(&ht->heads[hashindex], key)
        }
        Return NULL
}
```

Inserting a node into the hashtable using hashing for the index

```
Void ht_insert(HashTable *ht, GoodSpeak *gs) {
        Insertnode = ll_node_create(gs)
        Hashindex = hash(salt, oldspeak(gs)) % length of the hashtable
        If (heads[hashindex] == NULL) {
                Heads[hashindex] = insertnode
        }
        Else {
                ll_insert(heads[index], insertnode->gs)
        }
        ll_node_delete(insertnode)
```

```
}
The number of heads within the hastable
Ht_count (Hashtable *h) {
        Count = 0;
        For (i=0; i< h->lengthl i++) {
                If (h->heads[i] != NIL)
                        count++
        }
        Return count
}

Creates a GoodSpeak Struct to store the oldspeak and new speak into
GoodSpeak *gs_create(char *oldspeak) {
        Newgoodspeak = (GoodSpeak*) malloc(size of goodspeak +1)
        newgoodspeak->Oldspeak = malloc (strlen(oldspeak)+1)
        strcpy(newgoodspeak->oldspeak, oldspeak)

        If (newspeak != NULL) {
                newgoodspeak->newspeak = (char *)  malloc(strlen(newspeak)+1)
                strcpy(newgoodspeak->newspeak, newspeak)

        }
        Else {
                newgoodspeak->newspeak = NULL
        }
        Return newgoodspeak
}
Deletes the malloc spaces for the goodspeak struct
Void gs_delete(GoodSpeak *g) {
        free(g->oldspeak)
        free(g->newspeak)
        free(g)
}
Returns the oldspeak of the goodspeak struct
Void *gs_oldspeak(GoodSpeak *g) {
        Return g->oldspeak
}
Returns the newspeak of the goodspeak struct
Void *gs_newspeak(GoodSpeak *g) {
        Return g->newspeak
}
```

NewSpeak.c

```
Int main(int argc, char **argv) {
        While(getopt(argc, argv, “sh:f:mb”)) {
                Switch
                Case s sets the stats booleans to be true in order to print the stats of the
program.
                Case h sets the size of the hashtable
                Case f sets the size of the bloomfilter
                Case m sets the move_to_front to be true
                Case b sets the move_to_front to be false
        }
        Creates a bloomfilter with according size based on input
        Creates a hashtable with according size based on input

        Open newspeak file

        While(fscanf(new, “%s”, key11) != -1) {
                fscanf(new, “%s”, key22)
                If the end of the file is reached break
                bf_insert(bf, key11)
                Create a new goodspeak with the two keys
                Insert into the hashtable
                Delete the allocated memory of the goodspeak
        }


        Open badspeak file
        Key 1 = NULL
        While(fscanf(new, “%s”, key) != -1) {
                fscanf(new, “%s”, key1)
                If the end of the file is reached break
                bf_insert(bf, key)
                Create a new goodspeak with the two keys
                Insert into the hashtable
                Delete the allocated memory of the goodspeak
        }

        Regcomp the desired the string of characters
        while(1) {
                Inputkey = next_word(stdin, &regex)
                Change the input key to lower case
                If(the keys are set within the bitvector of the bloomfilter) {
```

Look up the key within the hashtable and if it exists, and the goodspeak is null, insert into to the forbidden linked list and if the word has a translation then insert into the good linked list after creating a node for it

Before inserting, check if the link node already exists within the linked lists

}

}

If stats flag was called,

Print stats, seeks, seek length , hash table load and bloomfilter load

Counts the number of bits set within the bloom filter to calculate for the bloomfilter load

Else {

If linked list good is empty, and forbidden linked list is not null, {

Print the thoughtcrime message

}

Else {

If forbidden is null print the encouragement message

Else {

Print the thoughtcrime and encouragement message.

}

}

}

}

Bv.c

This function creates a new bitvector

```
BitVector *bv_create(uint32_t bit_len) {
    BitVector *newBitVector = (BitVector*) allocate space of(sizeof(BitVector))
    newBitVector->vector = (uint32_t*) allocate space of (8*bit_len)
    newBitVector->length = bit_len
    Return BitVector
}
```

This function deletes the allocated space of the bitvector

```
void bv_delete(BitVector *v) {
    Frees the v
    Frees the v vector
}
```

```
Uint32_t bv_get_len(BitVector *v) {
```

```
        Return v length
}

Void bv_set_bit(BitVector *v, uint32_t i) {
        Int f=i/8
        Double bf = i%8
        Int bf1 = (int)bf
        V vector[f] |= 1 << bf1
}

Void bv_clr_bit(BitVector *v, uint32_t i) {
        Int f=i/8
        Double bf = i%8
        Int bf1 = (int)bf
        V vector[f] &= ~(1 << bf1)
}

Uint8_t bv_get_bit(BitVector *v, uint32_t i) {
        Int f=i/8
        Double bf = i%8
        Int bf1 = (int)bf
        Return V vector[f] >> bf1 &1
}

Void bv_set_all_bits(BitVector *v) {
        For uint32_t i = 0 til i < v length while i increments {
                bv_set_bits(v, i)
        }
}
```