DESIGN

Pre-lab Part 1:

1)jhdj
Inserting to bloom filter:
Hash is modded by bitvector length so it fits in bitvector size.

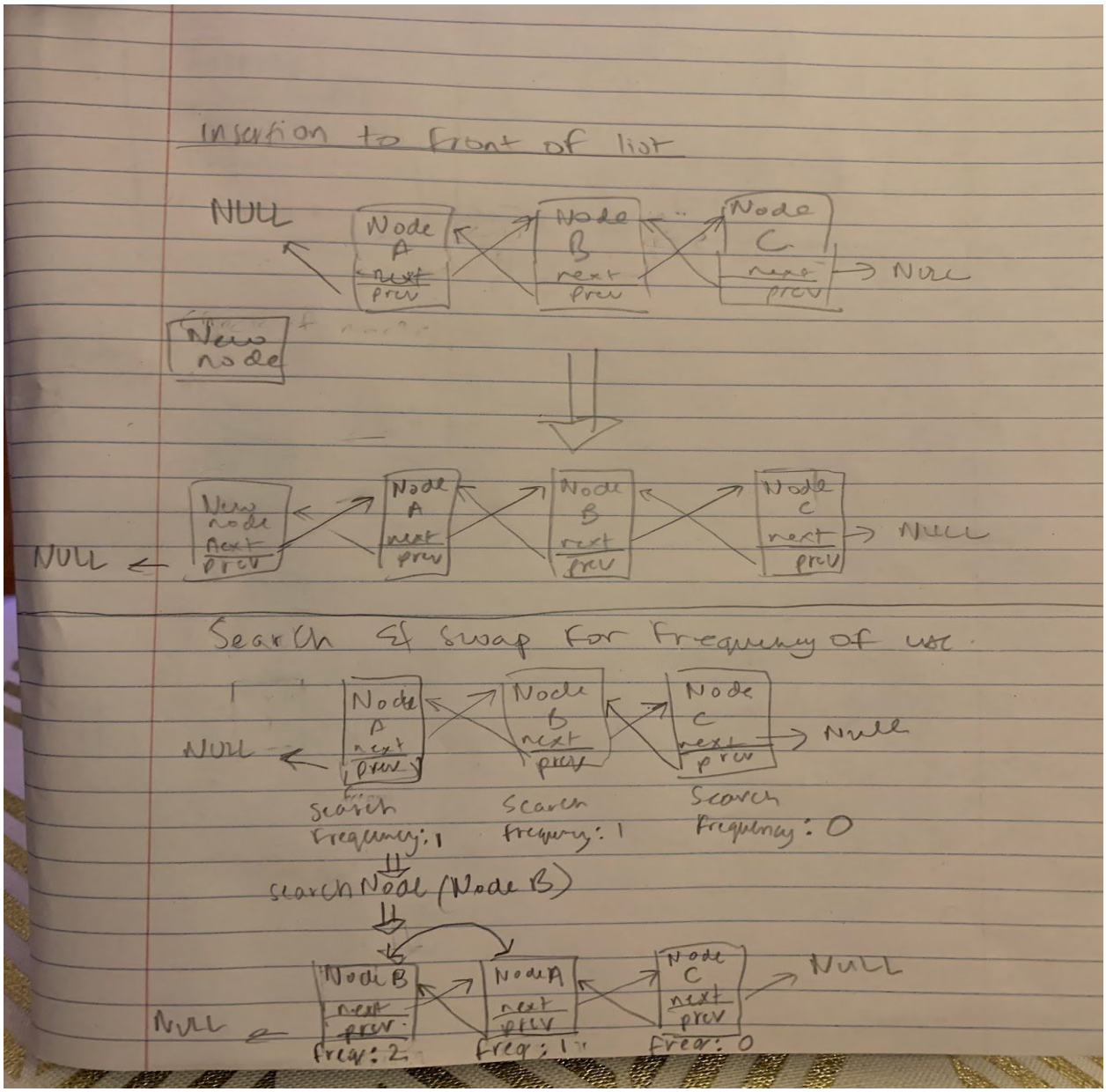
```
def bf_insert(input Bloomfilter b, key):
  bv_set_bit(b.filter, hash(primary_hash, key) mod (length_bit_vector))
  bv_set_bit(b.filter, hash(secondary_hash, key) mod (length_bit_vector))
  bv_set_bit(b.filter, hash(tertiary_hash, key) mod (length_bit_vector))
```

Removing from bloom filter:

```
def bf_delete(input Bloomfilter b, key):
  bv_clr_bit(b.filter, hash(primary_hash, key) mod (length_bit_vector))
  bv_clr_bit(b.filter, hash(secondary_hash, key) mod (length_bit_vector))
  bv_clr_bit(b.filter, hash(tertiary_hash, key) mod (length_bit_vector))
```

2)
Since bloom filters use bit vectors, they would be space-efficient because information about a big data set can be encoded in individual bits. The number of hash functions k would have the most impact on the space complexity since that would dictate the maximum number of distinct bit indices that could store a value for a single key. So the space used by the Bloom filter could be up to additional k-1 times the number of bits. As for time complexity, to search through all the elements(the bits), the time complexity would be linear since having more hash functions would directly increase time complexity.

Pre-Lab Part 2:

1) 

2) Create a linked list node:

Global variables: lookups(count seeks), links(counts node travelled)
def ll_node_create(input GoodSpeak pointer gs):
ListNode node = allocate memory for ListNode pointer object
Node.goodspeak.oldspeak_word = gs.oldspeak

```
  Node.goodspeak.newspeak_word = gs.newspeak
  Node.next set to point nowhere(NULL)
Delete Node:
ll_node_delete(input ListNode n):
  Free memory allocated to n.next
  N.next = Null
  Free memory allocated to goodspeak struct instance
  Goodspeak struct set to point nowhere(NULL)
  Free memory allocated to ListNode n
  ListNode n set to point nowhere(NULL)

Delete List:
def ll_delete(input ListNode head pointer):
  while(head doesn't point nowhere):
    ll_node_delete(input head)
    head set to point to next node in linked list.

Insert node
def ll_node_insert(input ListNode head double pointer, Goodspeak word set struct):
 If node(goodspeak) not in linked list:
  ListNode new_node = ll_node_create(Goodspeak word set struct)
  if(head single pointer doesn't point nowhere):
    new_node.next_node = *head
    return new_node
  Else:
    New_node.next_node points nowhere(NULL)
    return new_node

Lookup node
def ll_lookup(input ListNode head double pointer, key):
  lookups++
  if(head pointer and key both point somewhere):
    while(head != null AND key != head.goodspeak.oldspeak):
      head points to head.next_node
      links++
     return node
 if(move_to_front):
  prev_Node->next = ListNode->next
  ListNode = head;
```

```
    return NULL

  Node print
  def  ll_node_print(input ListNode node ptr):
   if(node != NULL):
     print n->gs->oldspeak

   Linked List print
  def ll_print(ListNode head ptr:
     while(head doesn't point to NULL):
       ll_node_print(head)
       Head points to head.next_node
```

HashTable pseudocode:

ht_create() provided by Darrell Long in asgn6.pdf file

```
Delete hashtable
def ht_delete(input hashtable ptr) {
  For list_head in h->heads:
    If list_head isn't null:
      ll_delete(list_head)
}

Lookup key
def ht_insert(input hashtable ptr, key):
  For list_head in h->heads:
    If list_head isn't NULL:
      If  ll_lookup(list_head, key) != NULL:
        return lookup(list_head,  key)

Count non-null hash indices
def  ht_count(input Hashtable ht):
  Count = 0
  For list_head in h->heads:
    If list_head != NULL:
       Count++
```

Return count

Hash table print
def ht_print(input Hashtable):
For list_head in h->heads:
If list_head != NULL:
Curr_node = list_head
-whlie(Curr_node != null):
Print hash_value + Curr_node.oldspeak
Curr_node = curr_node->next

Bloomfilter pseudocode;
Note: All bitvector(bv) functions are from Bitvector ADT written in asgn3

bf_create() provided by Darrell Long in asgn6.pdf file

Delete filter:
def bf_delete(input Bloomfilter ptr):
bv_delete(bf->filter) //Delete bitvector. From bitvector code in asgn3
free(ptr)
ptr = NULL

Insert in bloomfilter
def bf_insert(input Bloomfilter ptr bf, key):
bv_set_bit(bf->filter), hash(primary salt, key) % bitvector length))
bv_set_bit(bf->filter), hash(secondary salt, key) % bitvector length))
bv_set_bit(bf->filter), hash(tertiary salt, key) % bitvector length))

Bloomfilter length
def bf_length(input bloomfilter ptr b):
return bv_lenght(b->filter)

Check the filter for word
def bf_probe(input bloomfilter ptr b, key):

If bv_get_bit(bf->filter), hash(primary salt, key) % bitvector length)) == 1:
If bv_get_bit(bf->filter), hash(secondary salt, key) % bitvector length)) == 1:

```
    If bv_get_bit(bf->filter), hash(tertiary salt, key) % bitvector length)) == 1:
      return true:
Return false

Count number of set bits
def bf_count(input Bloomfilter ptr b)
  Set_count = 0
  For x in range(0,bf_length):
    if(bv_get_bit(b->filter, k) == 1:
       set_count ++
  Return set_count
```

Goodspeak pseudocode:

```
Create GoodSpeak:
def gs_create(input oldspeak, newspeak):
  Goodspeak gs = allocate memory
  if gs:
    If oldspeak != NULL:
      gs->oldspeak = allocate memory the size of oldspeak
      strcpy(gs-oldspeak, oldpeak)
    If newspeak != NULL:
      gs->newspeak = allocate memory the size of newspeak
      strcpy(gs-oldspeak, oldpeak)
    return Goodspeak ptr:

Delete goodspeak
gs_delete(input Goodspeak ptr):
  free(g->oldspeak)
  Set to null
   free(g->newspeak)
  Set to null
  free(ptr)
   Ser to null

Return oldspeak word:
def gs_oldspeak(input Goodspeak ptr g):
  return g->oldspeak
```

Return newspeak word:

```
def gs_newspeak(input Goodspeak ptr g):
  return g->newspeak
```

Main code:

Getopt code:

```
  if  flag -s:
    Run_censor = false
  if flag -h:
    Set hashtable size
  If flag -f:
    Ser bloomfilter size
  If flag -b:
   move_to_front = false
 If flag -m:
   Move_to_front = true
```

Regular expr code:

```
  Regex_t regex
  r = regcomp(regex, regular expression, style)
  If r:
    Raise error: “Regex compilation failed”
```

Copy string and switch to lower_case

```
def strLower(input dest,  src):
  if src != NULL:
    for character in src:
      dest.append(character)
    dest.append(null character)
```

Read badspeak.txt

```
Word = “ ”
while word != NULL
  Word = next_word(badspeak file, compiled regular expression)
   if(Word != NULL)
```

```
    Store_word = allocate space the size of Word:
     strLower(Store_word, Word)
     bf_insert(bf, Store_Word)
     Goodspeak gs = gs_create(Store_word, NULL)
     ht_insert(hastable pr, gs)

free(Store_word)
Set store_word to NULL


Read newspeak.txt
Word = " "
while word != NULL
  Word = next_word(badspeak file, compiled regular expression)
   if(Word != NULL)
     Store_old_word = allocate space the size of Word:
      strLower(Store_old_word, Word)
      bf_insert(bf, Store_Word)
  Word = next_word(newspeak file, compiled regular expression)
    if(Word != NULL)
     Store_new_word = allocate space the size of Word:
      strLower(Store_new_word, Word)

    Goodspeak gs = gs_create(Store_old_word, Store_new_word)
    ht_insert(hastable pr, gs)

free(Store_old_word)
Set store_old_word to NULL
free(Store_new_word)
Set store_new_word to NULL


Read message from Standard input:

Linked_list forbidden_nodes
Linked_list old_nodes

Num_old = 0
Num_forbidden = 0
```

```
Read = “”
While Read != NULL:
  Read = next_word(standard input, regular expression)
  Read_lowercase = allocate space size of Read
  strLower(Read_lowercase, Read)
  if(bf_probe(bf, Read_lowercase) == true:
    ListNode node_word = ht_lookup(ht, Read_lowercase)
   If node_word != NULL &&   node_word->gs->oldspeak == NULL:
     Num_forbidden++
     ll_insert(forbidden_nodes, node_word->gs)
   Else:
      Num_old++
     ll_insert(old_nodes, node_word->gs)
   free(Read_lowercase)
   Set Read_lowercase to NULL

if run_censor
  if  Num_old >0 && Num_forbidden == 0:
    Print  unique message for only oldspeak words used
    for node in old_nodes:
      ll_node_print(node) //prints oldspeak word
      Print “->”
      Print gs_newspeak(node)

   if  Num_old  == 0 && Num_forbidden > 0:
    Print  unique message for only forbidden words used
    for node in forbidden_nodes:
      ll_node_print(node)  //prints oldspeak word

   if  Num_old  > 0 && Num_forbidden > 0:
    Print  unique message for both forbidden and old words used
    for node in forbidden_nodes:
      ll_node_print(node)  //prints oldspeak word
    Print message “words to think about”
    for node in old_nodes:
      ll_node_print(node) //prints oldspeak word
      Print “->”
      Print gs_newspeak(node)
```

Else:
  Print “Seeks” + lookups
  Print “Average seek length” + links/lookups
  Print “Hash table load” + (100*(ht_count(ht ptr)/ht_size))
  Print “Bloom filter load” + (100*(bf_count(bf ptr)/bf_size))

bf_delete(bf)
ht_delete(ht)
ll_delete(forbidden_nodes)
ll_delete(old_nodes)
regfree(regex)