Assignment 7 Design Document

In this document, I will be going over my thought process of how I will be going through the process of completing this assignment. Additionally, this document will also include pseudocode that I will be using as a guide during my coding process.

Bloom Filters

This ADT seems like it will be fairly easy to implement. I will be using my implementation of a BitVector that I created for a previous lab.

```
bf_create(size):
  # This code is given in the assignment PDF

bf_delete(bf):
  free_memory(bf.bitVector)
  free_memory(bf)
  set_to_null(bf)
  return

bf_size(bf):
  return bf.bitVector.size

bf_insert(bf, oldspeak):
  bf.bitVector.setBit(firstHash(oldspeak) % bf_size(bf))
  bf.bitVector.setBit(secondHash(oldspeak) % bf_size(bf))
  bf.bitVector.setBit(firstHash(oldspeak) % bf_size(bf))

bf_probe(bf, oldspeak):
  return bf.bitVector.getBit(firstHash(oldspeak) % bf_size(bf)) &&
    bf.bitVector.getBit(secondHash(oldspeak) % bf_size(bf)) &&
    bf.bitVector.getBit(thirdHash(oldspeak) % bf_size(bf))

bf_count(bf):
  count = 0
  for(i in range(bf_size(bf))):
    if(bf.bitVector.getBit(i)):
      count++
  return count

bf_print(bf):
  bv_print(bf.bitVector)
```

Hashing with the SPECK Cipher

One concern that I would like to bring up here is with the way that the hash is being used to generate indices to set. Here, the hash function return an int of type uint32_t. This provides quite a large range of possible indices to have to access in the bitVector. My concern is that if the bitVector is not sufficiently large, the hash function may end up generating indices that are out of the possible indices in the bitVector. I attempted to solve this by performing a modulus operation on the result of the hash function that limits it to being less that teh size of the underlying bitVector. I am not positive if this solution is the correct one, but we will see when I actually implement this code.

Bit Vectors

I did not have any issues with my previous bitVector implementation, and thus I will not be making any changes to it.

HashTable

I do not have any experience implementing a HashTable, and so this is uncharted territory for me. Thankfully, the implementation does not seem too difficult, although we will see.

```
ht_create(size, mtf):
  # This code is given in the Assignment PDF

ht_delete(ht):
  free_memory(ht.linkedList)
  free_memory(ht)
  set_to_null(ht)
  return

ht_lookup(ht, oldspeak):
  index = hash(ht.salt, oldspeak)
  ll = ht.lists[index]
  n = ll.head.next
  while(n != ll.tail):
    if(n.oldspeak == oldspeak):
      if(ht.mtf):
        move_to_front(n)
      return n
    n = n.next
  return null_pointer

ht_insert(ht, oldspeak, newspeak):
```

```
  index = hash(ht.salt, oldspeak)
  if(!ht.lists[index]):
    ht.list[index] = ll_create(ht.mtf)
  ll = ht.list[index]
  ll_insert(ll, oldspeak, newspeak)
  return

ht_count(ht):
  count = 0
  for(i in range(ht.size)):
    if(ht.lists[i]):
      count++
  return count

ht_print(ht):
  for(i in range(ht.size)):
    if(ht.lists[i]):
      ll_print(ht.list[i])
  return
```

Nodes

In order to talk about Linked Lists, we need to talk about what is being linked together in the first place, which are Nodes. Nodes are pretty basic, and I will going over them here.

```
node_create(oldspeak, newspeak):
  n = dynamically_allocated_memory(sizeof(Node))
  n.oldspeak = string_duplicate(oldspeak)
  n.newspeak = string_duplicate(newspeak)
  n.next = null_pointer
  n.prev = null_pointer

node__delete(n):
  free(n.oldspeak)
  free(n.newspeak)
  free(n)
  set_to_null(n)
  return

node_print(n):
  if(n.newspeak && n.oldspeak):
    print("" + n.oldspeak + "->" n.newspeak + "\n")
    return
  if(n.oldspeak):
```

```
    print("" + n.oldspeak + "\n")
  return
```

Linked Lists

At this point, I have a decent amount of experience working with Nodes, and I don't think that this will be particulary difficult to implement. The concept that I was new to up until recently is the idea of sentinel nodes.

```
ll_create(mtf):
  ll = dynamically_allocate_memory(sizeof(LinkedList))
  ll.length = 0
  ll.head = node_create(null_pointer, null_pointer)
  ll.tail = node_create(null_pointer, null_pointer)
  ll.mtf = mtf
  return ll

ll_delete(ll):
  n = ll.head.next
  while(n!= ll.tail):
    node_delete(n.prev)
    n = n.next
  node_delete(n)
  free_memory(ll)
  set_to_null(ll)
  return

ll_length(ll):
  return ll.length

ll_lookup(ll, oldspeak):
  n = ll.head.next
  while(n != ll.tail):
    if(n.oldspeak == oldspeak):
      if(ll.mtf):
        move_to_front(n)
      return n
    n = n.next
  return null_pointer

ll_insert(ll, oldspeak, newspeak):
  if(!ll_lookup(ll,oldspeak)):
    return
  n = node_create(oldspeak, newspeak)
  n.next = ll.head.next
```

```
  n.prev = ll.head
  ll.head.next.prev = n
  ll.head.next = n
  return

ll_print(ll):
  n = ll.head.next
  while(n!= ll.tail):
    node_print(n)
    n = n.next
  return
```

Main Function

This is the pseudocode for the main function that will actually run my program

```
int main():
  parse_command_line()
  bf = bf_create(MAX_SIZE)
  ht = ht_create(MAX_SIZE, mtf)
  word badspeak[]
  word rightspeak[]
  for(word w in newspeak.txt):
    ht_insert(ht, w, (w = next_word))
    w = next_word()
  for(word w in oldspeak.txt):
    bf.add(w)
    ht_insert(ht, w, null_pointer)
    w = next_word()
  for(word w in stdin if w.is_valid_word()):
    if(bf_probe(bf, w)):
      if(ht_lookup(ht, w)):
        if(ht_lookup(ht,w).newspeak):
          rightspeak.add(w)
          continue
        badspeak.add(w)
        continue
  if(badspeak.size > 0 && rightspeak.size > 0):
    print_mixspeak_message()
  if(badspeak.size > 0 && rightspeak.size == 0):
    print_badspeak_message()
  if(badspeak.size == 0 && rightspeak.size > 0):
    print_goodspeak_message()
  if(statistics):
    print_statistics()
```

```
return 0
```

All of this pseudocode is preliminary and will most likely have changes as I actually implement all of it.