Assignment 6 Design Doc

bf.c

```
Add BloomFilter Struct
bf_create(size):
    take provided code
bf_delete(bf):
    free bf
    bf = NULL
bf_size(bf):
    return bv_length(bf->filter)
bf_insert(bf, oldspeak):
    loop through bf->salts:
        hash oldspeak with current hash
        bv_set_bit(bf->filter, hash result)
bf_probe(bf, oldspeak):
    loop through bf->salts:
        hash oldspeak with current hash
        if (!bv_get_bit(bf->filter, hash result)):
            return false
    return true
bf_count(bf):
    count = 0
    loop i from 0 to bv_length(bf->filter):
        if (bv_get_bit(bf->filter, i)):
            count += 1
    return count
bf_print(bf):
    bv_print(bf->filter)
bf_stats(bf, nk, nh, nm, ne):
    nk = bf->n_keys
    nh = bf->n_hits
    nm = bf->n_misses
    ne = bf->n_bits_examined
```

bv.c

```
Add BitVector struct
bv_create(length):
    make a BitVector bv and allocate memory size of a BitVector
    if (bv):
        bv->length = length
        for i from 0 to length/64+1:
            bv->vector[i] = 0
    return bv
bv_delete(bv):
    free(bv)
    bv = NULL
bv_length(bv):
    return bv->length
bv_set_bit(bv, i):
    bv->vector[i/64] bitwise or with 1 left shifted i%64 times
bv_clr_bit(bv, i):
    bv->vector[i/64] bitwise and with inverse of 1 left shifted i%64 times
bv_get_bit(bv, i):
    return (bv->vector[i/64] bitwise and with 1 left shifted i%64 times) right shifted i%64 times
bv_print(bv):
    for i from 0 to length:
        print bv_get_bit(bv, i)
```

ht.c

```
Add HashTable struct
ht_create(size, mtf):
    take provided code
ht_delete(ht):
    for i from 0 to ht_size(ht):
        free(ht->lists[i])
    free(ht->lists)
    free(ht)
    ht = NULL
ht_size(ht):
    return ht->size
ht_lookup(ht, oldspeak):
    n_links
    n_seeks
    ll_stats(&n_seeks, &n_links)
    ht_node = ll_lookup(ht->lists[hash(ht->salt, oldspeak)], oldspeak)
    if (!ht_node): ht_node = NULL
    new_links
    ll_stats((&n_seeks, &new_links)
    ht->n_examined += new_links - n_links
    return ht_node
ht_insert(ht, oldspeak, newspeak):
    ht_ll = ht->lists[hash(ht->salt, oldspeak)]
```

```
        if (!ht_ll): initiliaze ht_ll
        ll_insert(ht_ll, oldspeak, newspeak)
ht_count(ht):
        count = 0
        for i from 0 to ht->size:
                if (ht->lists[i]): count += 1
        return count
ht_print(ht):
        for i from 0 to ht->size:
                ll_print(ht->lists[i])
ht_stats(ht, nk, nh, nm, ne):
        hk = ht->n_keys
        nh = ht->n_hits
        nm = ht->n_misses
        ne = ht->n_examined
```

node.c

```
node_create(oldspeak, newspeak):
        make copies of oldspeak and newspeak, ospeak and nspeak
        create new Node n
        n->oldspeak = ospeak
        n->newspeak = nspeak
        return n
node_delete(n):
        free(n->oldspeak)
        free(n->newspeak)
        free(n)
        n = NULL
node_print(n):
        if (n->oldspeak and n->newspeak):
                print “oldspeak -> newspeak\n”
        else:
                print “oldspeak\n”
```

ll.c

```
Add LinkedList struct
ll_create(mtf):
        create LinkedList ll and allocate memory of size LinkedList
        if (ll != NULL):
                ll->mtf = mtf
                ll->length = 2
                ll->head = node_create(NULL, NULL)
                ll->tail = node_create(NULL, NULL)
                ll->head->next = ll->tail
                ll->tail->prev = ll->head
        return ll
ll_delete(ll):
        n = ll->head
        n_next = ll->head->next
        while (n_next):
                free(n)
                n = n_next
                n_next = n->next
        free(ll)
        ll = NULL
ll_length(ll):
        return ll->length
ll_lookup(ll, oldspeak):
        n = ll->head
        n_next = ll->head->next
        while (n_next):
                if (n->oldspeak == oldspeak):
                        if (ll->mtf):
                                n->prev->next = n->next
                                n->next->prev = n->prev
                                n->next = ll->head->next
                                ll->head->next = n
                                n->prev = ll->head
                        return n
                n = n_next
                n_next = n->next
        return NULL
ll_insert(ll, oldspeak, newspeak):
        if (ll_lookup(ll, oldspeak)): return
        Node *n = node_create(oldspeak, newspeak);
        n->next = ll->head->next
        n->prev = ll->head
        ll->head->next = n
ll_print(ll):
        n = ll->head
        n_next = ll->head->next
        while (n_next):
                node_print(n)
                n = n_next
                n_next = n->next
```

```
ll_stats(n_seeks, n_links):
    n_seeks = seeks
    n_links = links
```

parser.c

```
Add Parser struct
parser_create(f):
    create Parser p and allocate memory of size Parser
    if (p != NULL):
        p->f = f
        p->current_line = 0
        p->line_offset = 0
    return p
parser_delete(p)
    free(p)
    p = NULL
next_word(p, word):
    find next word
    word = copy of found word
    if no word found return false
    else return true
```

banhammer.c

```
ht_size = 10000
bf_size = 2**19
mtf = false
stats_print = false
take command line options and run associated code:
    -h: print out program usage
    -t: ht_size = atoll(optarg)
    -f: bf_size = atoll(optarg)
    -m: mtf = true
    -s stats_print = true
bf = bf_create(bf_size)
ht = ht_create(ht_size)
read all badspeak words from badspeak.txt:
    bf_insert(bf, word)
    ht_insert(ht, word, NULL)
read all oldspeak and newspeak pairs from newspeak.txt:
    bf_insert(bf, oldspeak)
    ht_insert(ht, oldspeak, newspeak)
```

```
LinkedList badspeak_words
LinkedList oldspeak_words_with_translation
read words from stdin using next_word()
if word is in bf:
    if word is in ht:
        if word has translation:
            add word and translation to oldspeak_words_with_translation
        else: add word to badspeak_words
if badspeak_words or oldspeak_words_with_translation are not empty:
    print letter reprimanding citizen
if (stats_print):
    print out statistics
```