Wendi Tan
wtan37@ucsc.edu

Assignment 6

Pre-lab:

Part1:

1). bf_delete(**bf){
```
    free((*bf)->filter)
    free(*bf)
    *bf = NULL
```
}

2). bf_insert(*bf, *oldspeak){
```
    index1 = hash(salt1, oldspeak)
    index2 = hash(salt2, oldspeak)
    index3 = hash(salt3, oldspeak)
    bv_set_bit(bf->filter, index1)
    bv_set_bit(bf->filter, index2)
    bv_set_bit(bf->filter, index3)
```
}

Part2:

1). ll_create(bool mtf){
```
    LinkList *ll = malloc(sizeof(LinkList))
    ll->length = 0
    ll->head = node_create(NULL, NULL)
    ll->tail = node_create(NULL, NULL)
    ll->head->next = ll->tail
    ll->tail->prev = ll->head
    ll->mtf = mtf
```
}

2). ll_delete(*ll){
```
    if(*ll is not NULL){
        Node *index = NULL;
        while( loop ends when index == ll->tail){
            index = ll->head->next
            node_delte(ll->head)
            ll->head = index
        }
        free(index)
        free(ll)
```

```
            ll=NULL
        }
    }

3). ll_length(*ll){
        Length = 0
        Node *index = ll->head->next
        while(loop ends when index == ll->tail){
            length ++
            index = index->next
        }
        free(index)
        Length = ll->length
}

4). ll_lookup(*ll, char *oldspeak){
        Node *index = ll->head->next
        While(index->oldspeak != oldspeak){
            index = index->next
            if(index == ll->tail){
                Return NULL
            }
        }
        if( ll->mtf ){                    // move-to-front operation; *n = index->prev
            n->next = index->next
            index->next->prev = index->prev
            index->next = ll->head->next
            index->prev = ll->head
            ll->head->next->prev = index
            ll->head->next = index
        }
        return index
}

5). ll_insert(*ll, *oldspeak, *newspeak){
        Node *n = ll_lookup()
        if(n is a NULL node){
            n = node_create(oldspeak, newspeak)
            n->prev = ll->head
            n->next = ll->head->next
            ll->head->next->prev = n
            ll->head->next = n
        }
}
```

6). ll_print(*ll){

```
        *index = node_create()
        index = ll->head->next
        while(index == ll->tail){
                node_print(index)
                index = index->next
        }
        node_delete(index)
}
```

Part3:

“([a-zA-Z0-9])+(('|-|_)[a-zA-Z0-9]+)*”

Design

This program implements Bloom Filter and Hash Table to search words from intext message that are forbidden, known as badspeak, and oldspeak which need to be translated into the corresponding newspeak. Then messaged will show notifying users what words are not allowed to use and what need to be translated. All badspeak and oldspeak-newspeak pairs are inserted into bloom filter and hash table beforehand.

1). Bloom Filter & Bit Vector:

Bit Vector:
An array with 8-bits elements. The length of a bv is the number of bits allocated in the array, each bit is represented as 1 or 0.

```
bv_create(length){
        BV *b = calloc(1, sizeof(BitVector))
        b->length = length
        b->vector = calloc(length/8+1, sizeof(uint8_t))
}

bv_delete(**bv){
    free(*bv->vector)
    free(*bv)
    *bv = NULL
}

bv_set_bit(*bv, i){
    byte = bv->vector[ I / 8 ]
    mask = 1 << i % 8
    bv->vector [ i / 8 ] = byte | mask
}

bv_clr_bit(*bv i){
    byte = bv->vector[ i / 8 ]
    mask = ~(1 << i % 8)
    bv->vector [ i / 8 ] = byte & mask
}

bv_get_bit(*bv, i){
    byte = bv->vector[ i / 8 ]
    mask = 1 << i % 8
    result = byte & mask
```

```
        result = result >> (i % 8)
}
```

Bloom Filter:

It is an array constructed by a Bit Vector and three salts.

```
bf_delete(**bf){
        bv_delete(*bf->filter)
        free(*bf->filter)
        free(*bf)
        *bf = NULL
}

bf_length(*bf){
        length = bv_length(bf->filter)
}

bf_insert(*bf, *oldspeak){
        index1 = hash(salt1) % bf_length(bf)
        index2 = hash(salt2) % bf_length(bf)
        index3 = hash(salt3) % bf_length(bf)
        bv_set_bit(bf->filter, index1)
        bv_set_bit(bf->filter, index2)
        bv_set_bit(bf->filter, index3)
}
//  bf_insert() uses hash() function to generate indices with three different salts for each
oldspeak. And then we set each index in the bloom filter as 1.

bf_probe(*bf, *oldspeak){
        index1 = hash(salt1) % bf_length(bf)
        index2 = hash(salt2) % bf_length(bf)
        index3 = hash(salt3) % bf_length(bf)
        if( bv_set_bit(bf->filter, index1) && bv_set_bit(bf->filter, index2) &&
bv_set_bit(bf->filter, index3)){
                return true;
        }
        return false;
}

// bf_probe() uses hash() function to generate indices with three different salts for each
oldspeak. Since this function tells us false-positive, if three indices are all 1, the oldspeak is likely
to be there in the bloom filter. But if there shows at least one 0, it's definitely impossible the
word could be there in bf.
```

2). Node & Linked List & Hash Table

Node:
One node has properties: char *oldspeak, char *newspeak, Node *next, and Node *prev. Nodes construct Linked Lists.

Linked List:
The initial ll contains a head and a tail, each with node of NULL and pointing to each other. When inserting a node, we always put the node at the front of other nodes, at back of the head.

Hash Table:
The hash table contains an array and a salt, each index in the array points to a linked list. Unlike the bloom filter, which has three salts for hashing in order to reduce the change of false positive, the hash table only has one salt for hashing because we need to insert node only once. ht_lookup() basically generates an index with hash and call ll_lookup() to check whether the node is in the linked list of arr[index]. ht_insert() generates an index for the oldspeak and call ll_insert() to insert the node.

3). Banhammer (main function)
p.s. green part of the pseudo-code below is the structure professor Long provided.

```
int main(){

// loop through getopt
// initialize bf and ht
// use fscanf() to scan badspeak.txt and insert each word in both bf and ht.
// do the same thing as above, scan newspeak.txt this time instead.

regex_t re;
   if (regcomp(&re, WORD, REG_EXTENDED)) {
      fprintf(stderr, "Failed to compile regex.\n");
      return 1;
   }
char *word = NULL;
//create an array to store badspeak that are used by the user: badspeak[]
//create an array to store oldspeak which have a translation that are used by the user: oldspeak[]

while ((word = next_word(stdin, &re)) != NULL){
      // check if the word is most possibly in bf: (needs to be true)
```

```
            //check if the word is in ht: (needs to be true)
                //if n->newspeak is NULL:
                        //insert word into badspeak[];
                //else (n->newspeak is not NULL):
                        //insert both of the node's oldspeak and newspeak into oldspeak[]
}

// print arrays badspeak[] and oldspeak[], and print warning messages to remind the users.
//free memory
return 0;
}
```