DESIGN FOR ASGN6
Bloom Filters, Hashing, and the Red Queen's Decrees

TASK:
Read in a list of nonsense words, setting the corresponding bit for each word in the bloom filter. (List will be oldspeak.txt)

Create a HatterSpeak struct for each forbidden word. The word should be stored in oldspeak, and hatterspeak should be set to NULL; forbidden words do not have translations.

Read in a space-separated list of oldspeak, hatterspeak pairs.
(List will be hatterspeak.txt)

Create a Hatterspeak struct for each oldspeak, hatterspeak pair, placing them in oldspeak and hatterspeak respectively

The hash index for each nonsense word is determined by using oldspeak as the key

Read text from standard input (I/O redirection must be supported)

Words that pass through the bloom filter but have no translation are forbidden, which constitutes a nontalk

The use of nonsese words constitutes a nontalk. If only forbidden words were used, you will send them a nontalk message.

PRE-LAB Part 1:
1.

```
bf_insert(BloomFilter, char)
  index1 = hash(salt1, char) % length of filter
  index2 = hash(salt2, char) % length of filter
  index3 = hash(salt3, char) % length of filter

  bv_set_bit(filter, index1)
  bv_set_bit(filter, index2)
  bv_set_bit(filter, index3)

bf_delete(BloomFilter, char)
  bv_delete(filter) // use bitvector delete as has its own properties to be freed
  free(bf)
```

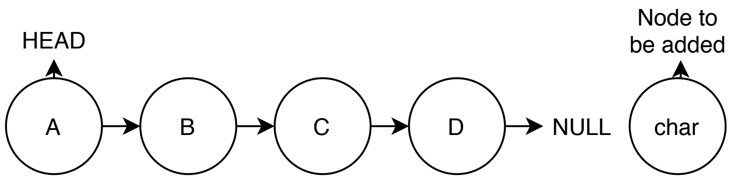
2.

Inserting and probing functions will have a time complexity of O(k) with k being the number of salts/hash functions.

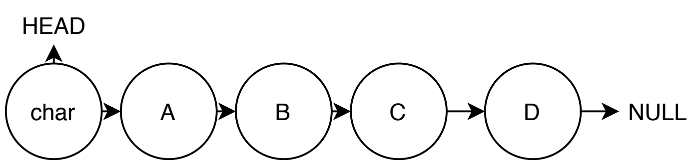
The space complexity will simply be O(m) with how many bits it reserves.

PRE-LAB Part 2:
1.

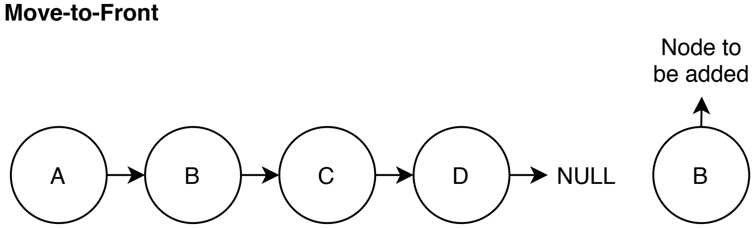
No Move-to-Front



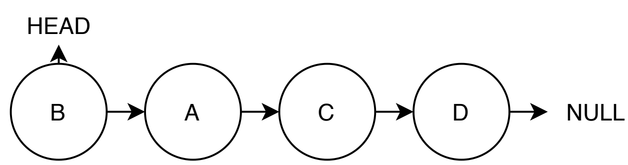

```
ll_lookup(ListNode, char)
  if not in list, make it the new head of the linked list
  by rewiring the nodes
  if in list, free it and dont insert
```



Move-to-Front




```
ll_lookup(ListNode, char)
  if not in list, make it the new head of the linked list
  by rewiring the nodes
  if in list, free it and dont insert; move it to front of list
  to avoid further, long lookups
```



PRE-LAB Part 2:
2.

ll_node_create(HatterSpeak *gs)
  malloc memory for listnode
  assign its hatterspeak pointer to gs
  set its next pointer to null
  return node

ll_node_delete(ListNode *n)
  free nodes's oldspeak
  free node's hatterspeak if it exists
  free node's hatterspeak
  free node

ll_delete(ListNode *n)
  iterate through each node of the linked list until
  it points to NULL, signifying the end of the list
  ll_node_delete(each node)
  set current node to next node

ll_insert(ListNode **head, HatterSpeak *gs)
  if ll_lookup finds node to be inserted
    free allocated space for node and
    return the current head

  Otherwise, create the new node with ll_node_create
  set it as the head of the linked list
  return the node

ll_lookup(ListNode **head, char *key)
  increment seeks and nodes traveled
  iterate through linked list
  increment nodes traveled
  if char and the current node's oldspeak are the same.
    if move to front enabled, make node the head of ll
      return current node

  Otherwise, return NULL;

ll_print_old(ListNode *head)
  while iterating through ll,
    print the oldspeak of each node as long
    as it is not "~" (placeholder node)

ll_print_translate(ListNode *head)
   while iterating through ll,
     print both oldspeak and its translation hatterspeak if
     oldspeak is not "~" (placeholder node)

ll_length(ListNode *head)
  while iterating through ll,
    keep track of number of nodes iterated,
    then return it

ll_void(ListNode *head)
  return; (void function to avoid dead store errors)

Pseudocode

Hash Table

ht_create(length)
  malloc memory for ht
  assign salts, length, and heads (which you malloc)
  return ht

ht_delete(HashTable *ht)
  iterate through hash table
  for each head in ht->heads,
  call ll_delete
  free ht->heads and ht

ht_count(HashTable *ht)
  iterate through hash table
  for each non-null head,
    increment n_heads
  return n_heads

ht_lookup(HashTable *ht, key)
  index = hash(salt, key) % ht->length

  return ll_lookup on ht->heads[index], key

ht_insert(HashTable *ht, HatterSpeak *gs)
  index = hash(salt, gs->oldspeak) % ht->length

  if the first head is null, create a node and
  increment seeks
  else, ll_insert the hatterspeak at the specific indexed head

ht_total_length(HashTable *ht)
  iterate through hash table
    call ll_length on each head node in ht->heads
    add length to total length
  return total_length

Bloom Filter

bf_create(size)
  malloc memory for bf
  assign three salts
  return bf

bf_probe(BloomFilter *bf, char *key)
  index1 = hash(salt1, char) % length of filter
  index2 = hash(salt2, char) % length of filter
  index3 = hash(salt3, char) % length of filter

  if all bits at index are 1,
    return true

  otherwise, return false

bf_count(BloomFilter *bf)
  iterate through bloom filter
  for each bit set, increment set_bits
  return set_bits

Pseudocode

```c
#include "speck.h"
#include <inttypes.h>
#include <stddef.h>
#include <string.h>

#define LCS(X, K)
        \
  (X << K) | (X >> (sizeof(uint64_t) * 8 - K)) // left circular shift
#define RCS(X, K)
        \
  (X >> K) | (X << (sizeof(uint64_t) * 8 - K)) // right circular shift

// Core SPECK operation
#define R(x, y, k) (x = RCS(x, 8), x += y, x ^= k, y = LCS(y, 3), y ^= x)


void speck_expand_key_and_encrypt(uint64_t pt[], uint64_t ct[], uint64_t K
    []) {
  uint64_t B = K[1], A = K[0];
  ct[0] = pt[0];
  ct[1] = pt[1];

  for (size_t i = 0; i < 32; i += 1) {
    R(ct[1], ct[0], A);
    R(B, A, i);
  }
}

uint64_t keyed_hash(const char *s, uint32_t length, uint64_t key[]) {
  uint64_t accum = 0;

  union {
    char b[sizeof(uint64_t)]; // 16 bytes fit into the same space as
    uint64_t ll[2]; // 2 64 bit numbers.
  } in;

  uint64_t out[2]; // SPECK results in 128 bits of ciphertext
  uint32_t count;

  count = 0; // Reset buffer counter
  in.ll[0] = 0x0;
  in.ll[1] = 0x0; // Reset the input buffer (zero fill)

  for (size_t i = 0; i < length; i += 1) {
    in.b[count++] = s[i]; // Load the bytes

    if (count % (2 * sizeof(uint64_t)) == 0) {
      speck_expand_key_and_encrypt(in.ll, out, key); // Encrypt 16 bytes
      accum ^= out[0] ^ out[1]; // Add (XOR) them in for a 64 bit result
      count = 0; // Reset buffer counter
      in.ll[0] = 0x0;
      in.ll[1] = 0x0; // Reset the input buffer
    }
  }
 // There may be some bytes left over, we should use them.
 if (length % (2 * sizeof(uint64_t)) != 0) {
   speck_expand_key_and_encrypt(in.ll, out, key);
   accum ^= out[0] ^ out[1];
 }

 return accum;
}


//
// Wrapper function to get a 32-bit hash value by using SPECK's key hash.
// SPECK's key hash requires a key and a salt.
//
// ht:      The HashTable.
// key:     The key to hash.
//
uint32_t hash(uint64_t salt[], char *key) {
 union {
   uint64_t full;
   uint32_t half[2];
 } value;

 value.full = keyed_hash(key, strlen(key), salt);

 return value.half[0] ^ value.half[1];
}
```

speck.h/c was provided to us by the lab manual.
It serves to avoid hash collisions in our hash table
using a specific hash function when generating the index
it will set an item in the hash table.

Pseudocode

parser.h/c was provided to us by the lab manual. It helps us to lexically analyze an input stream of words using regex.

```c
#include "parser.h"
#include <regex.h>
#include <inttypes.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define BLOCK   4096

static char *words[BLOCK] = { NULL };  // Stores a block of words maximum.

//
// Returns the next word that matches the specified regular expression.
// Words are buffered and returned as they are read from the input file.
//
// infile:      The input file to read from.
// word_regex:  Pointer to a compiled regular expression for a word.
// returns:     The next word if it exists, a null pointer otherwise.
//
char *next_word(FILE *infile, regex_t *word_regex) {
  static uint32_t index = 0;  // Track the word to return.
  static uint32_t count = 0;  // How many words have we stored?

  if (!index) {
    clear_words();

    regmatch_t match;
    uint32_t matches = 0;
    char buffer[BLOCK] = { 0 };

    while (!matches) {
      if (!fgets(buffer, BLOCK, infile)) {
        return NULL;
      }

      char *cursor = buffer;

      for (uint16_t i = 0; i < BLOCK; i += 1) {
        if (regexec(word_regex, cursor, 1, &match, 0)) {
          break;  // Couldn't find a match.
        }

        if (match.rm_so < 0) {
          break;  // No more matches.
        }

        uint32_t start  = (uint32_t)match.rm_so;
        uint32_t end    = (uint32_t)match.rm_eo;
        uint32_t length = end - start;

        words[i] = (char *)calloc(length + 1, sizeof(char));
        if (!words[i]) {
          perror("calloc");
          exit(1);
        }

        memcpy(words[i], cursor + start, length);
        cursor += end;
        matches += 1;
      }

      count = matches;  // Words stored is number of matches.
    }
  }

  char *word = words[index];
  index = (index + 1) % count;
  return word;
}

//
// Clears out the static word buffer.
//
void clear_words(void) {
  for (uint16_t i = 0; i < BLOCK; i += 1) {
    if (words[i]) {
      free(words[i]);
      words[i] = NULL;
    }
  }

  return;
}
```

Pseudocode

```
int main(command line arguments)
  while getopt parses through arguments
    if -s, print statistics
    if -h, set following number to size of hash table (default 10000)
    if -f, set following number to size of bloom filter (default 2^20)
    if -m, set move_to_front to true
    if -b, set move_to_front to false

  if both m & b, exit code and tell user to use only 1

  create bloomfilters for oldspeak, hatterspeaks, and hash table for
  translation pairs

  open oldspeak.txt and insert each oldspeak into the bloom filter and
  its respective hatterspeak into the hash table until EOF, then close file

  open hatterspeak.txt and insert each pair as a hatterspeak into the
  hash table, each key into the bloom filter, and each hatterspeak into
  the hatterspeak bloom filter. then close file

  check if regcomp

  create bad and revised linked lists to store words in

  while next_word is parsing through stdin,
    make word lowercase

    check if it is in bloom filter, continue
    check if it is a false positive using ht lookup, continue
    check if there is a hatterspeak translation, add it to revised ll
    check if there isn't a hatterspeak translation, add it to bad ll

clear_words and regfree

if stats, print seeks, avg seek length, avg ll length, hash table load, and bloom filter load
  delete bad and revised ll, both bloom filters, and hash table

if words in revised and bad, print letter with oldspeak and translatable words
if words in revised only, print letter with translatable words
if words in bad only, print letter with oldspeak words
  delete bad and revised ll, both bloom filters, and hash table

return 0;
```

Design Process:

The BloomFilter was by far the easiest ADT to implement as it really consisted of using our previous BitVector ADT, Linked lists were a bit more difficult as I was stuck on how we should rewire the node pointers when using the move to front rule. I realized, after drawing it out, that I should be using the next node when moving to front and rewire the pointers.

Figuring out on how to deal with false positives was fairly easy as I just double checked if the word was in the hash table. If it wasn't, that means my bf_probe returned a false positive and I can simply continue with the while loop.

I had a lot of memory leaks with make infer which resulted in memory not being reachable after a closing bracket. I realized that most of these were due to redundant checks in multiple of my ADTs to check if there were malloc/calloc failures.

I also had to a ll_void function because I was getting dead stores when inserting items into my bad and revisded as I wasn't using the nodes ll_insert returned.

Sources:

Speck and parser were provided from Asgn 6 lab manual.