Design Document

Description

The purpose of this program is to quickly determine if a word is illegal or legal. This will be done using a Bloom filter. If the word is illegal, then the word should quickly be searched to see if it maps to a legal word or not. This will be done using a hash table, where the key is the hash of the illegal word and the value is a binary search tree mapping keys to values. The reason for the binary search tree in the hash table is in the case where two illegal words have the same hash, we can still distinguish them and find their mappings.

Structure

This section contains the pseudocode.

Bloom Filter

All illegal words will be hashed through the SPECK cipher using three different 128-bit keys, or salts, which will set 3 random bits in the bit vector. To check if a word is illegal, we simply compute its 3 hashes and check if those bits are set. If they are not all set we know for certain the word is legal; if they are all set we are pretty sure the word is illegal.

```
void bf_insert(BloomFilter *bf, char *oldspeak){
      first = hash(primary, oldspeak);
      second = hash(secondary, oldspeak);
      third = hash(tertiary, oldspeak);
      bv_set_bit(bf, first);
      bv_set_bit(bf, second);
      bv_set_bit(bf, third);
}
```

Bit Vector

A bit vector was used in Assignment 5 for the code data structure. Here it is used as the bits of the bloom filter.

```
bv_set_bit(BitVector *bv, uint32_t i){
      if (i > bv->length)
return false;
bv->vector[i / 8] |= (1 << i % 8);
return true;
}
void bv_print(BitVector *bv) {
    printf("Bit Vector: ");
    for (uint32_t i = 0; i < bv->length; i++) {
        if (bv_get_bit(bv, i))
            printf("1");
        else
            printf("0");
        if (i != bv->top - 1 && i % 8 == 7)
            printf(" ");
    }
    printf("\n");
}
```

Hash Table

A hash table is used to map illegal word hashes to legal word binary search trees. The illegal word is hashed, the hash value is looked up, and the binary search tree is used to find the illegal word's corresponding legal word.

```
Node *ht_lookup(HashTable *ht, char *oldspeak){
      index = hash(ht->salt, oldspeak);
return ht->trees[index];
}
void ht_insert(HashTable *ht, char *oldspeak, char *newspeak){
      index = hash(ht->salt, oldspeak);
      bst_insert(ht->trees[index], oldspeak, newspeak);
}
uint32_t ht_count(HashTable *ht){
      int ret=0;
      for(int i=0; i<ht->size; i++){
            if(ht->trees[i]) ret++;
}
return ret;
}
double ht_avg_bst_size(HashTable *ht){
      double total_size = 0;
      for(int i=0; i<ht->size; i++){
            if(ht->trees[i]) total_size += bst_size(ht->trees[i]);
      }
      return total_size / ht_count(ht);
}
double ht_avg_bst_height(HashTable *ht){
      double total_height = 0;
      for(int i=0; i<ht->size; i++){
            if(ht->trees[i]) total_height += bst_height(ht->trees[i]);
      }
      return total_height / ht_count(ht);
}
```

Binary Search Tree

A binary search tree is used to store oldspeak words and their newspeak translations (if they exist) with the same hash value.

```
uint32_t bst_height(Node *root){
      if(root == NULL) return 0;
      return 1 + max(bst_height(root->left), bst_height(root->right);
}
Node *bst_find(Node *root, char *oldspeak){
      if(root == NULL){
            Error: Not found
            return NULL;
}
if(root->oldspeak == oldspeak) return root;
if(root->oldspeak < oldspeak) return bst_find(root->right, oldspeak);
return bst_find(root->left, oldspeak);
}
Node *bst_insert(Node *root, char *oldspeak, char *newspeak){
      if(root = NULL){
            root = new node(oldspeak, newspeak);
      }
      if(root->oldspeak == oldspeak){
            Warning: duplicate node found
            return root;
      }
      if(root->oldspeak > oldspeak){
            return bst_insert(root->left, oldspeak, newspeak);
      }
      return bst_insert(root->right, oldspeak, newspeak);
}
```

Inputs and Outputs

The list of supported command-line options are:

➢ -h Prints out program synopsis and usage.
➢ -t <size> Size of the hash table (default: 2^{16}).

- -f <size> Size of the bloom filter (default: 2^{20}).
- -s Statistics are printed to stdout.
    - Average binary search tree size
    - Average binary search tree height
    - Average branches traversed
    - Hash table load
    - Bloom filter load