Bv.c

(Must be in bv.c)
struct BitVector {
	uint32_t length ;
	uint8_t * vector ;
};

*BitVector *bv_create(uint32_t length)* {
	Use calloc or malloc to allocate memory for the Bitvector
	Use calloc or malloc again to allocate memory for the array size length
	(probably calloc to make each bit in array 0)
	Return pointer
}

*Void bv_delete(BitVector **bv)* {
	Free array that is size length
	Free btivector
	Set pointer to NULL
}

Uint32_t bv_length(bv) {
	Return the bit vector length
}

Bool bv_set_bit(bv, uint32_t i) {
	Set the i_th bit of the bit vector. If i is out of range of the bitvector return false;
}

*bool bv_clr_bit(BitVector *bv, uint32_t i)* {
	Same as bv_set_bit but now clears the i_th bit of the bitvector
}

*bool bv_get_bit(BitVector *bv, uint32_t i)* {

Same as clear and set bit, returns false if bit is 0 and true if bit is 1

}

Node.c

*Node *node_create(char *oldspeak, char *newspeak)* {

Allocate memory using calloc or malloc

We need to make a copy of oldspeak and newspeak using strdup() (#include string.h)

Set left and right to NULL

Return the pointer to the Node

}

*void node_delete(Node **n)* {

Free only Node n, not the next and previous nodes.

Because we allocated memory for oldspeak and newspeak, we much free those too

}

*void node_print(Node *n)* {

If node n has oldspeak AND newspeak use:

printf ("%s -> %s\n", n- > oldspeak , n- > newspeak) ;

If node n ONLY has oldspeak, use

printf ("%s\n", n- > oldspeak) ;

}

Bf.c (BloomFilter)

```
struct BloomFilter {
  uint64_t primary[2];   // Primary hash function salt.
  uint64_t secondary[2]; // Secondary hash function salt.
  uint64_t tertiary[2];  // Tertiary hash function salt.
  BitVector *filter;
};
```

(must go in bf.c)

*BloomFilter *bf_create(uint32_t size)* {
    Allocate memory for the Bloom Filter
    Set primary[0] to the lower primary salt from salts.h
    Set primary[1] to the higher primary salt from salts.h
    Do the same for secondary and tertiary salts
    Use bv create to make filter
}

*void bf_delete(BloomFilter **bf)* {
    Free filter using bv_delete
    Free bf
    Pointer Bf = NULL
}

*uint32_t bf_size(BloomFilter *bf)* {
    Use bv_length to get size of the bloom filter
}

*void bf_insert(BloomFilter *bf, char *oldspeak)* {
    Use bv_set_bit to insert, use hash function from speck.c together with each Salts, make sure it is in bounds of the bloom filter.
}

*bool bf_probe(BloomFilter *bf, char *oldspeak)* {
    The same inserting but this time using bv_get_bit
}

*uint32_t bf_count(BloomFilter *bf)* {
    Have a uint32_t variable to hold the count
    Iterate through from 0 to length of bloom filter
        Increment the variable
    Return the variable
}

HashTable

```c
struct HashTable {
  uint64_t salt[2];
  uint32_t size;
  Node **trees;
};
```

This `struct` definition *must* go in `ht.c`.

HashTable *ht_create(uint32_t size) {
	Allocate memory for HashTable using malloc
	Same as bf salt, salt[0] = the lower salt for Hashtable
	Salt[2] = higher salt for Hashtable
	ht->size = size;
	Trees: we allocate memory for a node using calloc and size as the number
}

void ht_delete(HashTable **ht) {
	We must iterate through the tree array to free the memory(if there are things to free)
	Then free the tree
	Free the pointer
	Set pointer to ht to NULL
}

uint32_t ht_size(HashTable *ht) {
	Just return the size of the hashtable
}

Node *ht_lookup(HashTable *ht, char *oldspeak) {
	Search for a node with the oldspeak, we use the hash function given in speck.c, and mod it by the ht_size to make sure it is in bounds of the Hash Table
	Then use that number we get as the index for the tree array, now we call bst_find
}

void ht_insert(HashTable *ht, char *oldspeak, char *newspeak) {
    Similar to ht_lookup but now use bst_insert
}

uint32_t ht_count(HashTable *ht) {
    Iterate through the tree array, if the node is not null, increment your variable to hold the count.
}

double ht_avg_bst_size(HashTable *ht) {
    Have a variable to hold the count
    Iterate through the tree from 0 to size of the tree;
        Get size of each tree using bst_size() and add that to the variable
    Return the variable divided by the count of hashtable
}

double ht_avg_bst_height(HashTable *ht) {
    The same way you calculated the average size of the binary tree, but now we use bst_height
    Return the variable that holds the could divided by the count of hashtable
}

Nodes

```
1 struct Node {
2     char *oldspeak;
3     char *newspeak;
4     Node *left;
5     Node *right;
6 };
```

Already in node.h

Node *node_create(char *oldspeak, char *newspeak) {
    Allocate memory for a node
    Use strdup() from string.h to duplicate oldspeak and newspeak and set that to the node's oldspeak newspeak
    Set left and right equal to NUL

}

void node_delete(Node **n) {
	Need to free memory from strdup
	Free the Node
	Set node node to NULL
}

`void node_print(Node *n)`

While helpful as debug function, you will use this function to produce correct program output. Thus, it is imperative that you print out the contents of a node in the following manner:

- If the node n contains oldspeak *and* newspeak, print out the node with this print statement:

```
1 printf("%s -> %s\n", n->oldspeak, n->newspeak);
```

- If the node n contains *only* oldspeak, meaning that newspeak is null, then print out the node with this print statement:

```
1 printf("%s\n", n->oldspeak);
```

Binary Search Trees

Node *bst_create(void) {
	Makes an empty tree, so just return NULL
}

void bst_delete(Node **root) {
	We use postorder traversal to delete the tree, since our tree is made of Nodes, use node_delete();
}

uint32_t bst_height(Node *root) {
	Returns the height of the tree, use max function from lecture 18,
	max(x, y) {
		return x > y ? x : y;

}
Make sure to add one to include the root of the tree
}

uint32_t bst_size(Node *root) {
Returns the size of the tree, recursively call the function plus one to include the root
}

Node *bst_find(Node *root, char *oldspeak) {
We look for a node that contains the oldspeak we are looking for, if found return that node.
If (root does not equal NULL && oldspeak is not NULL) {
while(node is not NULL && node's oldspeak is not the same as the oldspeak we are looking for)
Use strcmp to compare strings, if node's oldspeak is larger than the oldspeak we are looking for, go down the left of the node. Else go to the right
}
}

Node *bst_insert(Node *root, char *oldspeak, char *newspeak) {
Inserting a Node with its oldspeak and newspeak into the binary tree
If the root is null, return a new node by creating it.
If the root's oldspeak is larger than the oldspeak inserting, go down the left else go right
(call the bst_insert function again to do this recursively.)
Check if the oldspeak we are inserting is already is in the tree, if it is, return the root
}

Banhammer

Initialize the bloom filter and hash table
Read in the bad words from badspeak.txt using fscanf
Do the same for the oldspeak and newspeak translation

Do the regex, to check for the words
Make the words into lowercase
If the words are in the bloom filter, then we need to see if it is in the hashtable.
If it is in hashtable and has a newspeak translation, then make a binary search tree to insert it's oldspeak and newspeak translations

If the words are in the hashtable but does not have newspeak translation then,
Make another binary search tree to hold these badspeak words.