**Assignment 7**
**The Great Firewall of Santa Cruz**
**Bloom Filters, Linked Lists, Binary Trees and Hash Table**

**Bloom Filters:**

Assume you are adding a word w to your bloom filter, and are using k = 3 hash functions, f(x), g(x), and h(x). To add w to the Bloom filter, set the bits at indices of the 3 functions. To check if w prime has been added to the same Bloom filter, check if bits at indices 3 functions primed are set. If they are all set, then w prime has most likely been added to the bloom filter.

The larger the bloom filter, the lower the chances of getting false positives

False positives = “possibly in the set”
False negatives = “definitely not in the set”
Salt = “initialization vector or a key”
SPECK cipher = 128 bits

Equipping Bloom filter with three different salts, you are effectively getting three different hash functions

Hashing a word w, with extremely high probability, should result in 3 functions not being the same as each other

In the struct:

{
Primary[2] (primary hash function unit)
Secondary[2] (secondary hash function unit)
Tertiary[2] (tertiary has function unit)
BitVector *filter
}

---

**Hashing with the SPECK Cipher:**

To avoid the risk of having a poorly implemented hash function, one is given to us.

SPECK has been optimized for performance in software implementations

SPECK is an add-rotate-xor (ARX) cipher. The reason a cipher is used for this is because encryption generates random output given some input, exactly what we want for a hash.
---

**Bit Vectors:**

ADT that represents a one dimensional array of bits, the bits in which are used to denote if something is true or false (1 or 0). Since we cannot directly access a bit, we must use bitwise operations to get, set, and clear a bit within a byte (check out Code ADT from assignment 5).

---

**Hash Tables:**

Hash table is a data structure that maps keys to values and provides fast, O(1), look up times.

Takes a key *k*, hashing it with some hash function h(x), and placing the key's corresponding value in an underlying array at index h(k)

Hash function maps data of arbitrary size to fixed size values. Its values are called hash values, hash codes, digests, or simply hases, which index a fixed-size table called a hash table.

---

**Nodes:**

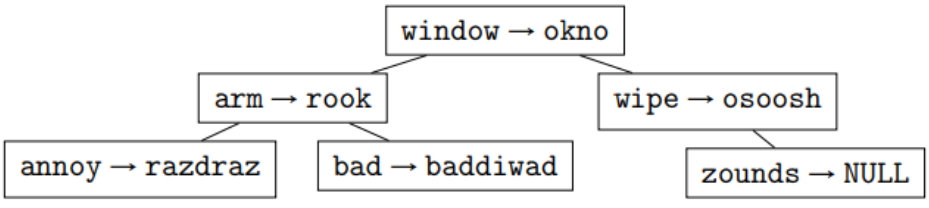
Binary trees will be used to resolve hash collisions. Binary search trees, and trees in general, are made up of nodes. For this assignment, each node contains oldspeak and its newspeak translation if it exists.

Oldspeak = the key to search with in a binary search tree

If newspeak field is NULL, then the oldspeak contained in this node is badspeak, since there is no newspeak translation

---

**Binary Search Trees:**




Example diagram showcases an example binary search tree

---

**Main file:**

User_input = stdin

Create_bloom_filter()
Create_hash_table()
Scan words from a list of bad speak (badspeak should be added to bloom filter and hash table)
Scan words from olspeak and newspeak

filter_words(user_input)

if (read word is not in bloom filter)
    Do nothing
Else: (decent amount of work here)
    Add it

---

-h prints out the program usage
-t size specifies that the hash table will have size entries (default is 2^16)
-f size specifies that the bloom filter will have size entries (the default is 2^20)
-s will enable the printing of statistics to stdout. The statistics to calculate are:
    Average binary search tree size
    Avg binary search tree height
    Avg branches traversed
    Hash table load
    Bloom filter load

Avg branches traverse = branches / lookups
Hash table load = 1-- * (ht_count / ht_size)
Bloom filter load = 100 * (bf_count / bf_size)

Printing this should be 6 digits of precision

Num of look ups = num of calls for ht_lookup and ht_insert

---

**Files to submit:**

1. Banhammer.c
2. Messages.h
3. Salts.h
4. Speck.h
5. Speck.c
6. Ht.h
7. Ht.c
8. Bst.h
9. Bst.c
10. Node.h
11. Node.c
12. Bf.h
13. Bf.c
14. Bv.h
15. Bv.c
16. Paser.h
17. parser.c