Liam Murray
Professor Miller
CSE13s
November 11, 2022

The Great Firewall of Santa Cruz Design Document

General Idea:

The general idea behind the Great Firewall of Santa Cruz is to implement a filtering system that compares the words of Santa Cruz citizens to a database of banned words, and reprimands them for using words that are not allowed, defined as “oldspeak”, or the far more deplorable “Badspeak”. This will be done via a series of Abstract Data type implementations, mainly a hash table, bloom filter, and linked list to store all of the oldspeak-newspeak translations, and to parse through the spoken words of the Citizens of the Great People's Republic of Santa Cruz.

An overview of the implementation:

First, a bloom filter will be used with a hash function from CityHash. A list of banned words will be hashed five times, with five different “salts”. This will result in 5 different numeric hash values, which will in turn be used to set those bit addresses in the bloom filter. Then, when words are being parsed, one can hash each word, and see if the 5 numbers produced are set in the bloom filter, which indicates that the word is probably on the banned word list. Words that are considered “Badspeak” (the highest offense) are only put into the bloom filter, whereas words that are “oldspeak” (and have newspeak translations), are put into a chained hash table, along with their corresponding newspeak translation. The chained hash table works with a doubly linked list acting as the chain apparatus.

From there, the parsing is as follows:

```
If a word is (probably) in the filter:
    Check if the word is actually in the filter.
    If it isn't:
        False positive, move on
    If it is, check the Hash table:
        If it has a translation, it is oldspeak:
```

Return the translation.

If it has no translation, it's badspeak:

Reprimand as such.

Deliverables:

Hash Table: used to store data in an effective, efficient way, to reduce search and processing time. This specific hash table will use a doubly linked list, in order to create a chained hash table to avoid collisions.

Pseudo:

```
//create a hash table. Taken from assignment doc
Structure definition: Hash table
    Define salt
    Define size
    Define number of keys
    Define number of bits
    Define number of hits
    Define number of misses
    Define number of elements examined
    Define whether table is move to front or not
Building function: makes the hash table
HashTable ht_create(num_elements , boolean move to front (mtf))
    HashTable ht = allocate(size of (HashTable Element));
    if (hash table is empty) {
        Ht set ;
        Set ht salt = 0x9846e4f157fe8840;
        Set ht n_hits = 0;
        Set ht misses = 0;
        Set Ht n_examined = 0;
        Set ht n_keys = 0;
        Set ht size = size;
        Set ht lists = array allocate(size , size of(LinkedList *));
    if (!ht->lists) {
        free(ht);
        ht = NULL:
    return ht;
```

Delete function: Deletes Hash table

```
Void ht_delete(**ht)
    Counter = ht size
    while(counter > 0, counter - 1 each pass)
        If ht[counter] is empty
            Pass
        While(ht[counter][next address] != NULL) //while there are still nodes on the list
            free(ht[counter][address])
            Address = next address
    //once all the list nodes are free
    free(*ht)
    **ht = NULL
```

Hash table size

```
Uint64_t ht_size(*ht) //returns hash tables size
    Return ht-> size
```

Hash Table lookup

Searches for a node containing oldspeak, and returns a pointer to the newspeak translation associated with it. If the node is not found, return a null pointer.

```
Node *ht_lookup(Hashtable *ht, char* oldspeak)
    Hash_address = ht-> hash(oldspeak)
    If ht[hash_address] = NULL,
        Return null *
    If ht[hash_address] != NULL
        //search the linked list that is there
        *node = ll_search(ll, oldspeak)
        If node != null
            Return node -> newspeak
```

Hashtable insert

Inserts Oldspeak and Newspeak into hash Table

```
Void ht_insert(Hash Table *ht, char * Oldspeak, char *newspeak)
    Hash_addr = hash(Oldspeak)
    If hashtable[hash_addr] = NULL
        ll_create()
        ll_insert(ll, oldspeak, newspeak )
        Return
    If hashtable[hash_addr] != NULL;
        ll_insert(ll, oldspeak, newspeak)
        Return
```

Ht_count

//returns the non-null linked lists in the hash table

```
Uint32_t ht_count(Hashtable *ht)
    Counter = ht->size
    Nodes = 0
    While (counter > 0, counter -1 each pass)
        If ht[counter] != NULL
            Nodes += 1
    Return Nodes
```

Ht_print

Prints out the contents of a hash table.

```
Void ht_print(Hashtable *ht)
    For (counter = 0, until counter == ht size, counter + 1 each time)
        If ht[counter] = NULL
```

```
            printf(“Node {counter} is empty”)
            continue
        printf(“Node {counter}: ”)
        ll_print(ll)
    return;
```

Hash Table Stats

Sets pointer values to stat values in the hash table

```
Void ht_stats(HashTable *ht, uint32_t *nk, uint32_t *nh, uint32_t *nm, uint32_t *ne)
    Nk = ht-> keys
    Nh = ht-> hits
    Nm = ht-> misses
    Ne = ht-> examined
    Return
```

Bloom Filter

The Bloom Filter is used to determine whether or not a word is *probably* in a set. It used 5 salts to hash the given words, and sets a bit corresponding to each hash value. Then, if the bits corresponding to each hash are set, one can determine that the word is likely part of the filter. This will let us determine whether or not to search the hash table for a word.

Bloom filter Create

Creates a bloom filter of size “size”.

```
    First, Define the salts
    Static uint64_t default_salts [] =
        Salt1 ,
        Salt2 ,
        Salt3,
        Salt4 ,
        Salt5
BloomFilter *bf_create(uint32_t size)
    BloomFilter *bf = BloomFilter * allocate(sizeof(BloomFilter));
    //if allocation is successful
    if (bf) {
        //set keys and hits to 0
        bf->n_keys = bf ->n_hits = 0;
        //set misses and bits examined to 0
        bf->n_misses = bf->n_bits_examined = 0;
        //set bf salts to the default salts, based on the number of hashes specified
        for (int i = 0; i < N_HASHES; i++) {
            //set bf salts to default salts
            bf ->salts[i] = default_salts[i];
    //set the filter to a bit vector of specified size
    bf->filter = bv_create(size);
    //If unsuccessful void bloom filter and return null pointer
    if (bf->filter == NULL) {
        free(bf);
        bf = NULL
    return bf;
```

Bloom Filter Delete

Deletes the Bloom filter specified

```
Void bf_delete(Bloomfilter **bf)
    bv_delete(bf->bitvector)
    free(bf)
    *Bf = NULL
    return
```

Bloomfilter Size

Returns the size of the Bloomfilter

```
uint32_t bf_size(Bloomfilter *bf)
    Return bv_length(bf->filter)
```

Bloom Filter Insert

Insert a value into the bloom filter bit vector

```
Void bf_insert(Bloomfilter *bf, char* oldspeak)
    For(i = 0, while i < N_HASHES, i + 1 each pass)
        Hash_index = hash(salt[i], oldspeak)
        Bv_set_bit(bf-filter, hash_index)
    Return
```

Bloom filter probe

Probes Bloom filter to see if a word was added. Return true if all 5 hashed indexes where

```
// potentially helpful pseudo for linked list search

If oldspeak has a translation
            Return Newspeak translation
        If oldspeak doesn't have a translation
            Return badspeak
    Next_address = ht[node][next address] (the next address stored in the Hash Table Node)
    While next_address != Null (go until the end of the linked list)
        If node[next_address] == Oldspeak
            If oldspeak has a translation
                Return Newspeak translation
            If oldspeak doesn't have a translation
                Return “Badspeak”
        Next_address = node[next address]
    //No address was found
    Return * NULL
```