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

```
Bool bf_probe(*bf, char* oldspeak)
    Probe1 = bv_get_bit(bf->vector, hash(salt1, oldspeak))
    Probe2 = bv_get_bit(bf->vector, hash(salt2, oldspeak))
```

```
Probe3 = bv_get_bit(bf->vector, hash(salt3, oldspeak))

Probe4 = bv_get_bit(bf->vector, hash(salt4, oldspeak))

Probe5 = bv_get_bit(bf->vector, hash(salt5, oldspeak))

If Probe1 + probe 2 + … + probe5 = 5:

        Return true

Return False
```

Bloom Filter count

Returns number of set bits in bloom filter

```
Counter = 0
Uint32_t bf_count(Bloomfilter *bf)
    for (i = 0, while i < bf_size, i +1 each pass)
        If bv_get_bit(bf, i) == 1
            Counter += 1
    Return counter
```

Bloom filter print

Prints out the bloom filter

```
Void bf_print(bloomfilter *bf)
    for(i = 0, while i < N_HASHES, i += 1 each pass)
        print("Salt [i+1]: ")
        print(bf->salt[i])
    bv_print(bf->filter)
```

Bloom Filter Stats

Sets variables outside of the bloom filter to the bloom filter stats values.

```
Void bf_stats(BloomFilter *bf, uint32_t *nk, uint32_t *nh, uint32_t *nm, uint32_t *ne)
    Nk = bf-> keys
    Nh = bf-> hits
    Nm = bf->misses
    Ne = bf-> examined
```

Bit Vector

The use of the bit vector is to store information in a simple, elegant way. In this case, we will be using it to store data about which words have been added to our bloom filter.

Bit Vector Creator

Creates a new bit vector object.

```
BitVector *bv_create(uint64_t length)
        Array_bytes  = (64 / length) + 1
        BitVector *bv = Calloc (array_bytes, size of Uint64_t)
        If bv = NULL;
                Return null pointer
```

Bit Vector Delete

Deletes a bit vector

```
Void bv_delete(Bit vector **bv)
        free(bv-> vector)
        Return
```

Bit Vector length

Returns the length of the bit vector

```
Uint32_t bv_length(Bit Vector)
        Return BitVector->length
```

Bit Vector Set Bit

Sets the ith bit in the bit vector.

```
Void bv_set_bit(bitvector, uint32_t i)
        Bit_vector_byte = i / 64
        Location_byte = i % 64
        Bitwise_number = 2 ^ (location_byte - 1)
        bv->vector[bit_vector_byte] = bv->vector[bit_vector_byte] OR bitwise_number
        Return
```

Bit Vector Clear Bit

Clear the ith bit in the bit vector.

```
Void bv_clear_bit(bitvector, uint32_t i)
        Bit_vector_byte = i / 64
```

Location_byte = i % 64

Bitwise_number = 2 ^ (location_byte - 1)

Bitwise_number = NOT bitwise_number

bv->vector[bit_vector_byte] = bv->vector[bit_vector_byte] OR bitwise_number

Return

Bit Vector Probe Bit

Returns the ith bit in the bit vector.

```
Uint8_t bv_get_bit(bitvector, uint32_t i)
	Bit_vector_byte = i / 64
	Location_byte = i % 64
	Bitwise_number = 2 ^ (location_byte - 1)
	bit_test = bv->vector[bit_vector_byte] AND bitwise_number
	If bit_test = 0
		Return 0;
	If bit_test != 0;
		Return 1;
```

Print Bit Vector

Prints Out the Bit Vector

```
Void bv_print(BitVector *bv)
	For(i = 0), while != bv_length(bv), i += 1 each pass )
		print(bv_get_bit(bv, i))
	print(newline)
	return
```

Linked List

The linked list ABS will be used to store data in each section of the hash table, to prevent hash collisions. It works by inserting and removing nodes based on the data entered into the list.

Struct definition for a linked list, taken from the assignment pdf

```
	struct LinkedList {
		uint32_t length;
		Node *head; // Head sentinel node.
		Node *tail; // Tail sentinel node.
```

```
        bool mtf;
    };
```

Linked List Create

Creates a new linked list.

```
LinkedList *ll_create(bool mtf)
    LinkedList *list
    list->mtf = mtf
    list-> length = 0
    list->head = node_create(NULL, NULL)
    list->tail = node_create(NULL, NULL)
    head->next = pointer to tail
    tail->previous = pointer to head
    return list
```

Linked List Delete

Deletes a Linked List.

```
Void ll_delete(LinkedList *list)
    Start = list->head
    for(i = list->length, while i != 0, i - 1 each pass)
        Next = start->next
        node_delete(start)
        Start = next
    *list = NULL
    Return
```

Linked List length

Returns the Length of the Linked List.

```
Void ll_delete(LinkedList *list)
    Start = list->head
    for(i = list->length, while i != 0, i - 1 each pass)
        Next = start->next
        node_delete(start)
        Start = next
```

```
    *list = NULL
    Return
```

Linked List Length

Prints the Length of the Linked List

```
Uint32_t ll_length(Linked List *list)
    Start = list->head
    Counter = 0
    While(Next != list-> Tail)
        Next = start->next
        Start = next
        Counter += 1
    Return Counter
```

Node Lookup

Looks for and returns a pointer to the node with a specific Oldspeak Translation

```
Node *ll_lookup(Linked List *ll, char * oldspeak)
    Start = list->head
    for(i = list->length, while i != 0, i - 1 each pass)
        Next = start->next
        If start->oldspeak = oldspeak
            break
        Start = next
        Counter += 1
    If Start != NULL
        If MTF = true
            start->next->previous = start->previous
            start->previous->next= start->next
            Temp = Head->next
            Head->next->previous = start
            Head->next = start
            start->next = temp
            start->previous = head
```

```
		Return Start
	Return *NULL
```

Node Insert

Inserts A node into the Linked List

```
Void ll_insert(Linked List *ll, oldspeak, newspeak)
	If ll_lookup(ll, oldspeak) == NULL;
		Return;
	New = Node_create(oldspeak, Newspeak)
	Temp = Head->next
	Head->next->previous = new
	Head->next = new
	new->next = temp
	new->previous = head
	Return;
```

Linked list stats

Sets variables to linked list stats

```
void ll_stats(uint32_t * n_seeks, uint32_t * n_links)
	N_seeks = seeks
	N_links = links
	Return;
```