THE GREAT FIREWALL OF SANTA CRUZ

Summary: This program is like a firewall that checks the user's input file for any oldspeak and badspeak words. Oldspeak words are words that are outdated and have newer translations while badspeak words are words that are banned from this society. If we find any of these words, we will address the correct punishment towards that user whether it be counseling or being sent to joy camp.

Bloom filter function:

Bf_create: constructor of a bloomfilter
- Initialize the needed variables
    - N keys = n bits = n misses = n bits examined = 0
    - Set the salts to the default salt array
    - Create a bit vector for filter
- Return the bloom filter

Bf_delete: Frees everything in bloomfilter and then frees bloom filter

Bf_size: Returns the size of the bloom filter which we can get by getting the length of the bit vector

*Bf_insert(bloom filter *bf, char *oldspeak):*
- Inserts the char input into bloom filter

*Bool bf_probe(BloomFilter *bf, char *oldspeak):*
- Return True if the imputed char oldspeak was added to bloom filter
- Return False otherwise

*Uint32_t bf_count(BloomFilter *bf):*
- Return num of set bits in bf

Void bf_print(bf): print

Void bf_stat(): Print all the variables in the bloom filter
- List of printed vals:bf, nk(num of keys), nh(nums of hits), um(num of misses), ne(nm of bits)

Bit vector functions:

Struct bitvector: bit vector contains a length and an array each containing 64 bits

**bv_create(uint32_t length)*: Creates bit vector array. Allocate “length” amounts of space for the overall array.

Bv_delete: free the vector and the bit vector

Bv_length: return the length

Bv_set_bit: Sets the ith least significant bit to 1
- Create a uint64 value with all bits being zeros, but the last one being 1
- Shift the bit i % 64 times to the left
- Use bit wise operator “or” between our created uint64 value and the uint64 value from our bit vector

Bv_clear_bit: Set the ith least significant bit to 0.
- Opposite of above
- Create a uint64 value with all bits being 1, but last one being 0
- Shift the bit i % 64 times to the left
- Use bitwise operator “and” between created uint64 value and the uint64 value from our bit vector

Uint8_t bv get bit:
- Create another uint64 and set that value to the uint64 that the bit is in
- Shift the array, then remove the rest of the front of array so the only thing left is the bit

Bv_print:
- Print the bit vector

HASH tables:

Struct hashtable:
- Uint64_t salt;
- Uint32_t size;
- Uint32_t n_keys
- Uint32_t n_hits;
- Uint32_t n_misses;
- Uint32_t n_examined;
- Bool mtf;
- LinkedList **lists;

*Hashtable *ht_create(uint32_t size, bool mtf):*

- Constructor for the hashtable
- Initialize/set the variables stated above
    - Set mtf to the imputed boolean
    - Set salt to the value that is listed on the document
    - Set n_hits = n_misses = n_examined = n_keys = 0
    - Set the size equal to the imputed size value
    - Allocate memory for the linked list
- Return the hashtable

ht_delete(ht): Destructor

- Frees the hashtable and the values inside it

Uint32_t ht_size(ht): Returns hash table size

*Node *ht_lookup(HashTable *ht, char *oldspeak):*

- Returns the node that contains the character, oldspeak
- Increment the seeks counter and update links examined in ll_stats

*Void ht_insert(HashTable *ht, char *oldspeak, char *newspeak)*

- Insert oldspeak and the corresponding newspeak translation into hastable
- Index is determined by hashing oldspeak.
- Create a linked list if the linked list hasn't been initialized yet.

Uint32_t ht_count(HashTable *ht)

- Returns number of non-Null linked list contained in the hashtable

Void ht_print(Hashtable *ht):

- Prints hashtable

Void ht_stats:

- Assign the stats to the imputed pointers.

Node:

Struct Node variables:

- Char *oldspeak;
- Char *newspeak;
- Node *next
- Node *prev

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

- Create ur own strdup() function in order to create a copy of the oldspeak and newspeak translation that was inputted.
- Must allocazte memory and then copy.

Void node_delete(Node **n):

- Destructor for node
- Free the node and any allocated memory

Void node_print(Node *n)

- Print out the continents in node

- Format: “oldspeak -> newspeak”

Linked list:

Struct linked list variables:
- Uint32_t length;
- Node *head;
- Node *tail
- Bool mtf

LinkedList *ll_create(bool mtf)
- Creates a doubly linked list
- When creating the linked list you will start off with a head and a tail(2 nodes).
- Whenever you insert a node, you have to input it after head
- So head-> next = inserted node

Void ll_delete(LinkedList **ll)
- Destroy linked using by freeing each node using node_delete
- Set the pointer of the linked list to NULL

Uint32_t ll_length(LinkedList *ll)
- Return length of linked list(num of nodes not including head and tail)

Void ll_insert(LinkedList *ll, char *oldspeak, char *newspeak)
- Insert a new node containing the oldspeak and newspeak into the linked list, by updating the next’s and prev’s values of the nodes that are between where the node is inserted.
- Check if a duplicate node is contained. If there is a duplicate node, the node will not be inserted
- Insert node after the sentinel head node

Void ll_print(Linkedlist *ll)
- Print linked list

Void ll_stats(uint32_32 * n_seeks, uint32_t *n_links)
- Copies LinkedList lookups and the number of links traversed into their respective variables

Parser

Struct parser vars:
- File *f
- Char current_line[MAX_PARSER_LINE_LENGTH +1];
- Uint32_t line_offset

Void *parser_create(FILE *f)
- Constructor for parser

Void parser_delete(Parser **p)
- Destructor for the parser. Free any allocated memory

Bool next_word(Parser *p, char *word)

- Return true if the next word was successfully parsed. Return false if there are no more words left.
- Every time it reaches the end of the line, it calls fgets() to get the next line
- If it is not a valid character, increment the offset
- If it is, store it in a temp until it reaches a nonvalid character. Then I will copy it to *word

Strfuncs:

- Uint64_t Slen(char * input): returns the length of the string
- char * sdup(char * input): returns a duplicate character
- bool scmp(char * str1, char *str2): Compares the two strings to see if they are the same
- void scpy(char * res, char * object): Copies object into res.

Banhammer:

- Options:
    - H: prints out program usage
    - T: hashtable size(default = 10000)
    - F: size of bloom filter(default = 2^19)
    - M: enable the mtf rule(default = false)
    - S: enables printing of statistics to stdout
- The rest of the file:
    - Store the badspeak words in the hashtable and bloom filter
    - Store the oldspeak in bloom filter.
    - Store the oldspeak and the newspeak word in the hashtable.
    - Take in an input with stdin and check if each word is a badspeak or oldspeak word
        - If it is store it in the respective linkedlist
    - After checking, we will see which laws they broke depending on the length of the badspeak and oldspeak linkedlist.
    - Print out the statement to tell them the punishment.
    - Return 0