Lynelle Goh
ljgoh
Assignment 6

General Idea

Our program will read in a list of words that are considered badspeak. Then we will read in oldspeak and newspeak pairs. We will parse the words with our own module and check to see if it has been added to the Bloom filter that we must create ourselves. If it is likely that the word has been added, then we need to insert this badspeak word into a list accordingly. We will print out different warnings for what transgressions are found.

Pseudocode

- ht.c (Contains the implementation of the hash table ADT.)
  - void ht_delete(HashTable **ht)
    - Free all linked lists and underlying array of linked lists. Place this at the end of the file. Set the pointer that was passed to NULL.
  - uint32_t ht_size(HashTable *ht)
    - Return the hash table's size using a for-loop and a counter to count the amount of times it goes through the hash table.
  - Node *ht_lookup(HashTable *ht, char *oldspeak)
    - Create a for-loop to search for an entry (a node) in the hash table that contains oldspeak. The node should contain oldspeak and its newspeak translation.
  - void ht_insert(HashTable *ht, char *oldspeak, char *newspeak)
    - Create an if-statement to see if a linked list has been created.
      - If not, initialize the linked list.
      - Else, Insert into the hash table the specified oldspeak and its corresponding newspeak translation. The index of the linked list to insert into is calculated by hashing the old speak.
  - uint32_t ht_count(HashTable *ht)
    - Return the number of non-NULL linked lists in the has table. This should be done by creating a while-loop that contains an if-statement.
      - The if-statement will check if the linked lists in the hash table is NULL.
        - If it is, it will add 1 to the counter variable.
  - void ht_print(HashTable *ht)
    - Write this immediately after the constructor. Print out the contents of a hash table, such as linked lists and if they're NULL or not, what values are inside, and other stuff that comes to mind.

- ➢ void ht_stats() (in full: ht_stats(HashTable *ht, uint32_t *nk, uint32_t *nh, uint32_t *nm, uint32_t *ne))
    - ■ Set *nk to the number of keys in the hash table
        - ● Create a for-loop that goes through the hash table.
            - ◆ Use a counter variable to count the number of keys.
    - ■ Set *nh to the number of hits.
        - ● Create a for-loop that goes through the hash table.
            - ◆ Use a counter variable to count the number of hits.
    - ■ Set *nm to the number of misses.
        - ● Create a for-loop that goes through the hash table.
            - ◆ Use a counter variable to count the number of misses.
    - ■ Set *ne to the number of links examined during lookups.
        - ● Create a for-loop that goes through the hash table.
            - ◆ Use a counter variable to count the number of links examined during lookups.

❖ banhammer.c (Contains main() and may contain the other (global) functions necessary to complete the assignment.)

- ➢ Initialize the Bloom filter and hash table.
- ➢ Read in a list of badspeak words with fgets().
- ➢ Then read in a list of oldspeak and newspeak with fgets(). Only the oldspeak should be added to the Bloom filter. The oldspeak and newspeak are added to the hash table.
- ➢ Read words in from stdin using our parsing module.
- ➢ Create an if-statement to check if a word has already been added to the Bloom filter.
    - ■ If not, no action is needed.
    - ■ If bf_probe() returns true…
        - ● If the hash table contains the word and the word does not have a newspeak translation, then the citizen who used this word is guilty of thoughtcrime. Insert this badspeak word into a list of badspeak words that the citizen used in order to notify them of their errors later.
        - ● If the hash table contains the word, and the word does have a newspeak translation, then the citizen requires counseling on proper Rightspeak. Insert this oldspeak word into a list of oldspeak words with newspeak translations in order to notify the citizen of the revisions needed to be made in order to practice Rightspeak.

- If the hash table does not contain the word, then all is good since the Bloom filter issued a false positive. No disciplinary action needs to be taken.
    - If the citizen is accused of thoughtcrime and requires counseling on proper Rightspeak, then they are given a reprimanding mixspeak message notifying them of their transgressions and promptly sent off to joycamp. The message should contain the list of badspeak words that were used followed by the list of oldspeak words that were used with their proper newspeak translations. Refer to the assignment document for what needs to be printed out.
    - If the citizen is accused solely of thoughtcrime, then they are issued a thoughtcrime message and also sent off to joycamp. The badspeak message should contain the list of badspeak words that were used. Refer to the assignment document for what needs to be printed out.
    - If the citizen only requires counseling, then they are issued an encouraging goodspeak message. They will read it, correct their wrongthink, and enjoy the rest of their stay in the GPRSC. The message should contain the list of oldspeak words that were used with their proper newspeak translations. Refer to the assignment document for what needs to be printed out.

- Use a boolean variable to see if an oldspeak word has already been printed out so that it doesn't get printed out again.
- Enabling the printing of statistics should suppress all messages the program may otherwise print.

➢ Our program will also need to support a list of command-line options:

- -h prints out the program usage. Refer to the reference program given to you for what to print.
- -t size specifies that the hash table will have size entries (the default will be 10000).
- -f size specifies that the Bloom filter will have size entries (the default will be 219).

        - -m will enable the move-to-front rule. By default, the move-to-front rule is disabled.
        - -s will enable the printing of statistics to stdout. The statistics to print include all statistics collected in the bloom filter and hash table as well as the number of bits examined per Bloom filter miss, the number of false positives, the Bloom filter load, and the average seek length.

- ll.c (Contains the implementation of the linked list ADT.)
    - LinkedList *ll_create(bool mtf)
        - Check if mtf is true.
            - Any node that is found in the linked list through a look-up is moved to the front of the linked list. Our linked lists will be initialized with exactly two sentinel nodes. They will serve as the heal and tails of the linked list.
            - If the linked list is not initialized and has no no nodes & is empty.
                - The inserted node becomes the head of the linked list.
            - Create an if-else-statement for when the linked list contains at least one node, which means the inserted node becomes the new head of the linked list and must point to the old head of the linked list. The old head of the linked list must also point back to the new head to preserve the properties of a doubly linked list.
            - Inserting a node means adding it at the head.
    - void ll_delete(LinkedList **ll)
        - Create a for-loop that goes through each node in the linked list and free it using node_delete().
        - The pointer to the linked list should be set to NULL.
    - uint32_t ll_length(LinkedList *ll)
        - Create a for-loop that goes through the linked list and returns the number of nodes in the linked list, but make sure we start it after the head node and end it before reaching the tail node.
            - Use a counter variable.
    - Node *ll_lookup(LinkedList *ll, char *oldspeak)
        - Create a while-loop that looks through the nodes. Search for a node containing oldspeak.
            - If it is found, the pointer to the node is returned.

- ◆ If a node was found, and the move-to-front option was specified when constructing the linked list, then the found node is moved to the front of the linked list. The move-to-front technique decreases look-up times for nodes that are frequently searched for.
  - ➢ Only accesses done for lookup are counted in the statistics of number of seeks and number of links examined.
- Else, a NULL pointer is returned.

➢ void ll_insert(LinkedList *ll, char *oldspeak, char *newspeak)

- ■ Inserts a new node containing the specified oldspeak and newspeak into the linked list. We must look up to make sure the linked list does not already contain the node we're holding. This will be done with a while-loop that goes through the linked list.
  - And an if-statement that checks if the variable holding the oldspeak matches with any nodes. If something matches, nothing will be done.
  - Else, the new node is inserted at the head of the linked list (comes after the head).

➢ void ll_print(LinkedList *ll)

- ■ Print out each node in the linked list except for the head and tail nodes. This can be done using a for-loop that starts after the head and ends before the tail.

➢ void ll_stats(uint32_t * n_seeks, uint32_t * n_links)

- ■ Set a variable, n_seeks, to the number of linked list lookups to, and the number of links traversed during lookups to n_links. These statistics are tracked across all linked lists, not maintained separately for each list.

❖ node.c (Contains the implementation of the node ADT.)

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

- ■ Make a copy of the oldspeak and its newspeak translation that are passed in. This means that we need to allocate memory and copy over the character for both oldspeak and newspeak.

➢ void node_delete(Node **n)

- ■ The node n is freed. Free the allocated memory to both newspeak and oldspeak. The pointer to the node should be set to NULL.

➢ void node_print(Node *n)

        - Print out nodes depending on guidelines.

- bf.c (Contains the implementation of the Bloom filter ADT.)
    - BloomFilter *bf_create(uint32_t size)
        - Following the code given to us, we need to create a static array (called default_salts) that contains the defaults. (A static function in C is a function that has a scope that is limited to its object file. This means that the static function is only visible in its object file. A function can be declared as static function by placing the static keyword before the function name.)
        - Create the function.
            - Allocate memory for the Bloom filter.
            - Create an if-statement to check if bf has anything.
                - n_keys and n_hits will equal to 0.
                - n_misses n_bits_examined will equal to 0.
                - Create a for-loop that goes through only if i is less than N_HASHES.
                    - salts[i] will equal one of the default_salts[i].
                - filter will equal bv_create(size).
                - If filter equals NULL…
                    - Free bf and set it to NULL.
    - void bf_delete(BloomFilter **bf)
        - Free any memory allocated by the constructor and null out the pointer that was passed in. Do this at the end of the file.
    - uint32_t bf_size(BloomFilter *bf)
        - Returns the size. This means that we need to find the length of the underlying bit vector.
    - void bf_insert(BloomFilter *bf, char *oldspeak)
        - Takes oldspeak and inserts it into the Bloom filter. Hash the oldspeak with each of the five salts for five indices and set the bits at those indices in the underlying bit vector.
    - bool bf_probe(BloomFilter *bf, char *oldspeak)
        - Probes the Bloom filter for oldspeak. Return true to signify that oldspeak was most likely added to the Bloom filter. Return false if not.
    - uint32_t bf_count(BloomFilter *bf)
        - Return the number of set bits in the Bloom filter.

- ➢ void bf_print(BloomFilter *bf)
  - A debug function to print out the contents of the Bloom filter.
- ➢ void bf_stats() (full function: bf_stats(BloomFilter *bf, uint32_t *nk, uint32_t *nh, uint32_t *nm, uint32_t *ne))
  - Sets *nk to the number of keys, *nh to the number of hits, *nm to the number of misses, and *ne to the number of bits examined.

❖ bv.c (Contains the implementation of the bit vector ADT.)

- ➢ BitVector *bv_create(uint32_t length)
  - Create an if-statement that check if there is sufficient memory.
    - If not, the function will return NULL.
    - Else, it must return a BitVector *, or a pointer to an allocated BitVector. Each bit of the bit vector should be initialized to 0.
- ➢ void bv_delete(BitVector **bv)
  - Set the pointer to NULL after the memory associated with the bit vector is free. Which means that we need to free the bit vector.
- ➢ uint32_t bv_length(BitVector *bv)
  - Returns the length of a bit vector.
- ➢ void bv_set_bit(BitVector *bv, uint32_t i)
  - Sets the i^{th} bit in a bit vector. The location of the byte is calculated as i / 64. The position of the bit in that byte is calculated as i % 64.
- ➢ void bv_clr_bit(BitVector *bv, uint32_t i)
  - Set the i^{th} bit in the bit vector,
- ➢ uint8_t bv_get_bit(BitVector *bv, uint32_t i)
  - Returns the i^{th} bit in the vit vector.
- ➢ void bv_print(BitVector *bv)
  - This is a debug function to print out a bit vector. Write this out immediately after the constructor.

❖ parser.c (Contains the implementation of the parsing module.)

- ➢ Parser *parser_create(FILE *f)
  - This is the constructor for the parser.
- ➢ void parser_delete(Parser **p)
  - The destructor for the parser. The pointer to the parser will be set to NULL.
- ➢ bool next_word(Parser *p, char *word)

- The function will find the next valid word in the buffer within the parser and store it in the buffer passed in. There is a limit to the length of the line, but there's no limit to the length of the word.
    - If the word was successfully parsed, the function shall return true. It will also copy the word into *word (include a 0 byte terminator, as for all C strings).
    - Else, the function returns false if there are no more words left and the contents of *word are undefined.
- Our function should first convert all uppercase letters to lowercase. Standard C Library function tolower() before copying to *word. use the Standard C Library function isalnum() to determine whether a character is alphanumeric, which leaves only a dash and single quote to compare against for determining whether a character is part of a word.