**Alex Asch**
**ASGN7 Design**
The Great Firewall of UCSC, filters input by parsing the words in the input, using a bloom filter and hash table together. This hashes unallowed words, adds those to the bloom filter and hash table, and uses binary search trees to resolve hash collisions. Then this reads through user input and checks the input banned words.
**Deliverables**

- banhammer.c
    - This file contains the implementation of the main function, which will parse user input.
- messages.h
    - Defines the mixspeak badspeak and goodspeak messages used.
    - This document is provided in the resources repository by Professor Long.
- salts.h
    - Defines the primary secondary and tertiary salts to be used in the Bloom filter.
    - Also defines the salt of the hash table.
    - This document is provided in the resources repository by Professor Long.
- speck.h
    - Defines the interface for the has function using the SPECK cipher.
    - This document is provided in the resources repository by Professor Long.
- speck.c
    - Contains the implementations of the has function using the SPECK cipher.
    - This document is provided in the resources repository by Professor Long.
- ht.h
    - Defines the interface for the hash table ADT.
    - This document is provided in the resources repository by Professor Long.
- ht.c
    - Contains the implementation of the has table ADT.
- bst.h
    - Defines the interface for the binary search tree ADT.
    - This document is provided in the resources repository by Professor Long.
- bst.c
    - Contains the implementation of the binary search tree ADT.
- node.h
    - Defines the interface for the node ADT.
    - This document is provided in the resources repository by Professor Long.
- node.c
    - Contains the implementation of the node ADT.
- bf.c
    - Contains the implementation of the bloom filter ADT.

- bv.h
    - Defines the interface for the bit vector ADT.
    - This document is provided in the resources repository by Professor Long.
- bv.c
    - Contains the implementations of the bit vector ADT.
- parser.h
    - Defines the interface for the regex parsing module.
    - This document is provided in the resources repository by Professor Long.
- parser.c
    - Contains the implementation of the regex parsing module
    - This code included in the
- Makefile
    - The file that compiles the program using make.
- README.md
    - A document in markdown syntax describing program usage
- Design.pdf
    - The document being viewed currently, describes the design of the implementation of the program
- WRITEUP.pdf
    - Analysis and graphs of the program as Bloom filter and hash table size is changed.

**General Notes**

- We need to implement both a Bloom filter and Hash table because of hash collisions.
- A Bloom filter is a bit vector with hashing functions in inserts.
- A Hash table is an array of binary search trees.
- Binary search trees are collections of linked nodes, which are inserted in an ordered manner.

**Pseudo code**

- **Bloom filter**
    - bf_create
        - Allocate memory for the bloom filter.
        - Set the salts to the salts specified in the header file.
        - Create a bit vector of the given length.
    - delete
        - Delete the bitvector using bitvec delete.
        - Then free the Bloom filter.
        - Set the pointer to null.
    - bf_size
        - Return the length of the bit vector in the bloom filter

- bf_insert
    - Hash the given oldspeak using all the salts, then take the numbers and set the bit at each number.
- bf_probe
    - Hash the given oldspeak using all the salts, and find the bloom filter for all of those bits at the hashed number.
    - If all three of those bits are true then return true, if it is not return false.
- bf_count
    - Iterate through the entire bit vector, and add one to a counting variable each time you find a set bit, use bv_get_bit and add.
    - Return the total.
- bf_print
    - Print the bit vector.

- **Bit Vector**
    - bv_create
        - Take one argument for the length of the vector.
        - Allocate memory for the struct
        - Make the length of the struct equal to the given length.
        - If the length is divisible by 8.
            - Make an array of 8 bit integers of length/8.
        - If the length is not divisible by 8.
            - Make an array of 8 bit integers of length/8+1.
        - Return the bit vector.
    - bv_delete
        - Free the array of integers.
        - Free the struct itself.
        - Set the pointer to NULL.
    - bv_length
        - Return the length of the given bit vector.
    - bv_set bit
        - If the given bit to set is greater than the length of the bit vector.
            - Return false.
        - Set the i/8th bit in the array to itself bitwise or 1 left shifted by i modulo 8, where i is the given bit to set.
        - Return true.
    - bv_clr_bit
        - If the given bit to set is greater than the length of the bit vector.
            - Return false.
        - Set the i/8th bit in the array to itself bitwise and all integers of all 1, except for the i modulo 8th bit, which is zero, where i is the bit to be set.

        - Return true.
    - bv_get_bit
        - If the bit to get is out of range, return false.
        - If it is not out of range, return the result of the ith index of the bit vector bit shifted to the right by i modulo 8 and 1. This compares 1 and the given bit to get.
    - bv_print
        - Loop for the entire length, and print the result of get bit.
- **Nodes**
    - node_create
        - Allocate memory for the node.
        - If newspeak is given, duplicate the newspeak string into the node newspeak.
        - Duplicate the oldspeak string into the node oldspeak.
        - Return the node.
    - node_delete
        - Free the strings of oldspeak and newspeak.
        - Free the node itself.
        - Set the node pointer to null.
    - node_print
        - If oldspeak and newspeak exist, print both.
        - If only oldspeak exists, print only oldspeak.
- **BST**
    - bst_create
        - Return Null.
    - bst_delete
        - If the node exists.
        - Go to the left node, call bst_delete on it.
        - Go to the right node, call bst_delete on it.
        - Delete the given node.
        - Set the node pointer to NULL.
    - bst_height.
        - If the given node does not exist, return 0.
        - If the node exists, return 1 + the greater of the heights of the left and right node recursively.
    - bst_size
        - If the current node exists, return 1 plus the size of the left plus the size of the right, recursively.
        - If the given node does not exist, return 0.
    - bst_find

        - If the node exists, compare lexicographically the oldspeak to find and the current node.
            - If the current node is greater than the given node we want to find, go search the left subtree, and return the result of searching in the left subtree recursively. Also add one to the branches tracking variable.
            - If the current node is less than the given node we want to search the right subtree, do this by recursively searching on the right node. Also add one to the branches tracking variable.
            - If the node is equal to the current node, we found the node, return the node.
        - If the root does not exist, return the root (null) this means we didn't find the node.
    - bst_insert
        - If the current node exists, compare lexicographically the node to the oldspeak to be inserted.
            - If the insertion node is less than the current node, go to the left node of the current node recursively, and set the left node of the current node to the result. Also increment the branches tracking variable, and return the current node.
            - If the insertion node is greater than the current node, go to the right of the current node recursively, set the right node of the current node to the result recursively. Also increment the branches tracking variable, and return the current node.
            - If the current node is equal to the node to be inserted, we have found a duplicate, so return the current node.
        - If the current node does not exist, create a node and return it, this is inserting the new node.
- **Hash table**
    - ht_create
        - Allocate memory for the ht.
        - Set the size to the given size.
        - Allocate an array of nodes of length size.
        - Set the salts to the salts given in speck.h.
        - return the hash table.
    - ht_delete
        - Go through the array of nodes calling bst_delete on each node.
        - Free the array of nodes.
        - Free the hash table itself.
        - Set the given pointer to off.

- ht_size
    - Return the size of the hash table.
- ht_lookup
    - Add one to the lookup variable.
    - Hash the given oldspeak.
    - If there is a binary tree at the index hashed.
        - Search the binary tree using bst find for the oldspeak and return the result.
    - If there is not, return null.
- ht_insert
    - Add one to the lookup variable.
    - Hash the given oldspeak.
    - If there is a binary tree at the hashed oldspeak, insert the oldspeak and newspeak into the binary tree using bst_insert.
    - If there is not a binary tree there, create a node and set it to the hashed index.
- ht_count
    - Iterate over the array of the hash table, if there is a tree there, add one to the counter variable.
    - Return the counter variable.
- ht_avg_bst_size
    - Iterate over the entire array of trees, and add the size of each tree to the counter variable.
    - Return the total divided by the count using the previous function.
- ht_avg_bst_height
    - Iterate over the entire array of trees, and add the height of each tree to the counter variable.
    - Return the total divided by the count using the previous function.
- ht_print
    - Loop through the array and print each tree.

- **Banhammer.c**
    - Go through command line options, and set arguments as appropriate.
    - Open the badspeak file, loop calls to fscanf to read each word from the badspeak individually.
    - Add that word to both the bloom filter and the hash table.
    - Open the newspeak file
        - Read the oldspeak newspeak pairs, from the file, add the oldspeak to the bloom filter, and the pair to the hash table.
    - Compile the regular expression used. (additional notes on this later).

    - Create two new bst to hold the found badspeak and oldspeak, and booleans to track if they are found
    - Use the given parser to read through the user input using the regular expression.
    - For each word, first convert the entire word to lowercase characters.
    - Then prove the bloom filter for that word, if it exists, search the hash table for that word. If it is in the hash table, add it to the respective badspeak/oldspeak trees, evaluated by if there is a newspeak translation.
    - If stats are specified in the command line, print the stats of the trees.
    - If the stats is not specified, print the mixspeak message if oldspeak and badspeak are found, the goodspeak_message if only oldspeak is found, and the badspeak message if only badspeak is found.
    - Then print the badspeak tree, and the oldspeak tree using bst_print.
    - Close all appropriate files.
- **Regex notes**
    - Regex needs to parse a-zA-Z0-9 and underscore, and should only read ‘ or - if it is not at the ends of a word, and is no more than one in a row.
    - Either get a string of letters until ` or - is found, then get more letters and repeat this, until there is end of line or there are duplicate ` or -. Or read a certain number of a-z, A-Z, and 0-9, this is for cases of three or less characters.
- **Parser Notes**
    - This code provided by the professor in the resources repo
    - This code contains the next word function, which reads in a block of words from an input file, then using the provided regex matches the word input.
        - This function tracks the number of words read, and returns words one by one.
    - This also comes with clear words, which clears any leftover words in the buffer and frees the buffer.
- **Speck notes**
    - This file contains the hashing algorithms to be used in the assignment.
    - The hash is the SIMON and SPECK lightweight block cipher, by Ray Beaulieu, Stefan Treatman-Clark, Douglas Shors, Bryan Weeks, Jason Smith and Louis Wingers.
    - This function has a hashing function that takes a salt and a key, and hashes the given key.