Dmitrius Agoston
Prof. Darrell Long
5 December 2021

CSE 13S Fall 2021
Assignment 7: The Great Firewall of Santa Cruz
Design Document

Description of Program:
The purpose of this program is to first create a Bloom filter with a corresponding hash table. This Bloom filter and hash table is responsible for taking in words that are designated as badspeak. The program will then read in a list of words that are formatted in pairs for oldspeak and the corresponding newspeak. Though only the oldspeak will be added to the Bloom filter while both new and oldspeak will be added to the hash table. The program can then take in an input and filter out the desired words from the given input. While parsing through the input if a word found is also in the Bloom filter there will be three choices of actions to take from this point. One being there is no translation and the user of the input is “guilty”. Second being there is a translation and they are deemed needing “counselling”. Third is the possibility of a false positive. If either of the first two cases are found true there are two separate messages to be given for each corresponding case. In both cases the user will be notified of the words caught by the Bloom filter.

Layout/Pseudocode:
- bf.c
  This file is responsible for the creation and use of the Bloom filter to be used by the program. The Bloom filter is a probabilistic data structure that will be able to determine if a word is likely belonging to a set. The way the Bloom filter will be represented is through the use of a bit vector to give a pseudo random distribution to the given words. Since the Bloom filter has the potential for false positives, there will be three salts being used in order to decrease the likelihood of triggering a false positive. These salts essentially act as three different hash functions which would probably result in all three function’s results not being equal to each other. In particular the hash function being used is the SPECK cipher.

  Create typedef structure for Bloom filter with primary, secondary, tertiary, and BitVector as variables

  Define bf_create with size
      Set primary, secondary, and tertiary to corresponding salts

Create a new BitVector with the given size to be used.

Define bf_delete with bf
Delete the bit vector named filter
Free memory allocated by bf constructor
Set the bf to NULL

Define bf_size with bf
Return the size of the bf which is the length of the bit vector

Define bf_insert with bf and oldspeak
Hash primary, secondary and tertiary
Mod the hashes by the size of bf
Set the bits at the indices of the hash in the bit vector

Define bf_probe with with bf and oldspeak
Get all the hashes by hashing primary, secondary and tertiary
Mod the hashes by size of bf
Check if all the indices of the hasehs with the salts of oldspeak are set
Return true
Return false

Define bf_count with bf
Create a variable for count
Loop as many time as the size of bf
Get the bit at the current index and if it is set
Increment the count
Return the count

Define bf_print with bf
Print bits of bf

- bv.c
  This file is used for the creation and implementation of a bit vector, which will be necessary for the functionality of the bloom filter. The bit vector is an ADT that acts as a one dimensional array of bits that determines if something is true or false (1 or 0). Though the indices in the vector will be treated more like bytes in order to achieve more efficiency in the process. Through this decision, this file will also have the capabilities of certain bitwise operations in order to account for the necessary functions for this specific structure.

Create structure BitVector with variables for length and vector

Define bv_create with length
    Create new bit vector and allocate memory
    Check if bit vector was not created successfully
        Set length to given length
        Allocate memory for vector
        Check if allocated incorrectly
            Set bit vector to NULL, free memory, and return
    Return new bit vector

Define bv_delete with bv
    Free the vector in bit vector
    Free the memory allocated for bv by the constructor
    Set bv to NULL

Define bv_length with bv
    Return length of bv

Define bv_set_bit with bv and i
    Check if i is not out of range of bv
        Set ith index of bv and return true
    Return false

Define bv_clr_bit with bv and i
    Check if i is not out of range of bv
        clear ith index of bv and return true
    Return false

Define bv_get_bit with bv and i
    Check if i is not out of range of bv
        if ith index of bv is set, return true
    Return false

Define bv_print with bv
    Print all bits of bv

- ht.c

This file is responsible for the creation of a hash table in the program. This hash table is essentially a structure that maps keys to their corresponding values. With O(1) time complexity when searching for the values, a hash table is an efficient choice for the purposes of this program. The hash table that is created will be capable of determining if the prohibited words found have an acceptable translation or not. This leaves the possibility of having hash collisions that will have to be resolved through the implementation of a binary search tree, which will be discussed later in this design document.

Create structure HashTable with variables salt, size, and trees

Define ht_create with size
- Create new hash table and allocate memory
- Check if hash table exists
  - Set the size to the given size
  - Set salt to corresponding salt
  - Allocate space for trees
  - If done improperly free memory and set to NULL

Define ht_delete with ht
- Free the memory allocated for the binary search trees along with the nodes
- Free the memory allocated for ht
- Set ht to NULL

Define ht_size with ht
- Return the size of ht

Define ht_lookup with ht and oldspeak
- Search for node that contains oldspeak by hashing and modding by size
- If node is found with bst search
  - Return pointer of node
- Return NULL pointer

Define ht_insert with ht. oldspeak, and newspeak
- Insert the given oldspeak into the hash and mod by size
- Use the hash of the oldspeak for where to insert into the tree

Define ht_count with ht
- Return number of non NULL binary search trees in hash table

Loop through trees one by one and check if not NULL
If not NULL increment count and return count when done

Define ht_avg_bst_size with ht
Return sum of of sizes of bsts divided by number of non NULL bsts
This is done through looping through the non null trees and summing the size

Define ht_avg_bst_height with ht
Return sum of of heights of bsts divided by number of non NULL bsts
This is done through looping through the non null trees and summing the height

Define ht_print
Print all contents of hash table

- node.c
  This file's purpose is involved in the creation and use of nodes in this program. In order to create a binary search tree which is necessary for other functions of the program, nodes must be created first. This is due to the fact that often binary search trees are made up of and implemented through the use of nodes. In particular for this program the nodes will be constructed with oldspeak and newspeak translations, if applicable. The oldspeak word will also work as the key when searching through the binary tree.

  Create structure Node with variables oldspeak, newspeak, right, and left

  Define node create with oldspeak and newspeak
  Allocate memory for node
  Set oldspeak to oldspeak
  Set newspeak to newspeak
  Return node

  Define node print with node
  Print contents of node for oldspeak and newspeak or just oldspeak

- bst.c
  This file is responsible for the creation and use of binary search trees throughout the program. The structure of the tree itself is recursive leaving only two options when searching the tree. That being either NULL or pointing to potentially two

other subtrees contained within the original tree. In terms of how the binary search tree will be organized is through lexicographical order. Each left subtree of a currently viewed node will contain an oldspeak word that is lexicographically less than the current value.

Define bst_create
    Create empty binary search tree

Define bst_delete with root
    Go through tree with postorder traversal
    Free memory for each node and set to NULL

Define bst_height with root
    Return height of tree at root
    Recursively check for left and right node height to see which is bigger
    Return 1 + the bigger height
    If the root is not found return 0

Define bst_size with root
    Return number of nodes at root
    Recursively check if left and right node exist
    If either or both exist add to a sum
    Increment sum by 1
    Return sum
    If node is not found return zero

Define bst_find with root and oldspeak
    Increment branches
    Compare the roots oldspeak lexicographically
    If root is bigger
        Recursively call on self with left node
    If root is smaller
        Recursively call on self with right node
    Return pointer to node containing oldspeak

Define bst_insert with root, oldspeak, and newspeak
    Increment branches
    Check if root exists
        If current root is bigger than oldspeak
            Recursively call itself with left and set left to result

If current root is smaller than oldspeak
Recursively call itself with right and set right to result
Return root
Create new node

Define bst print with root
Print tree starting at root with inorder traversal

- speck.c
  This file contains the implementation of the hash function using the SPECK cipher.

- parser.c
  This file contains the implementation of the regex parsing module.

- banhammer.c
  This file contains the main function along with the implementations of the functions necessary for finding any potential violations of prohibited words. Along with that, doling out the necessary punishment, if one is even required at all. In particular this file will first create the Bloom filter with the desired values given as well as the corresponding hash table. It will then parse through the input given that potentially has a violation embedded somewhere in it that violates the previously defined Bloom filter. If a prohibited word is found, the user will be notified of the violation and either be punished if no translation for this word is found or be sent to counseling if a translation is given.

  Parse through command line to get and set desired setting
  Parse through given input to create Bloom filter
  Create Bloom filter and hash table
  Populate the bloom filter with badspeak.txt
  Populate the hash table with newspeak.txt
  Parse through input being checked for prohibited words
  If prohibited word is found and no translation is found
  Add to bst full of words without translation
  If prohibited word is found and translation is found
  Add to bst full of words with translation
  If stats are desired print and terminate
  Print corresponding punishment with words used to receive punishment

Credit:
When creating this code I largely used asgn7.pdf as my main source of reference for creating this program. I also used some code from the slides of lecture 18 and is cited where used in the program.