Assignment 7: The Great Firewall of Santa Cruz

By: Derfel Terciano

Background

Imagine you were a great ruler of a country called Santa Cruz. As a great ruler, you would want your citizens to say certain things while banning them from talking about things that are considered to have a negative impact on society. In order to do that, you would need to filter out any negative talk that your citizens are saying whether it would be through the internet, talking to people on the phone or through texting people.

Assignment 7 deals with just that by filtering out all speech any citizen says. So how specifically are we going to filter out speech? We would need to filter out speech by using a combination of bloom filters, hash tables, and binary search trees. For further context, **badspeak** is any word that does not have a newspeak translation. This means that badspeak are words that citizens are not allowed to use. **Newspeak** are any words that are used to replace **oldspeak** words. Oldspeak in this case are words that were used before you became the ruler of Santa Cruz.

So let's talk about each data structure and how we would use them together to create our speech filter.

Bloom Filters

In order to determine what words to filter out in the first place, we would need to insert a list of badspeak words into some sort of data structure that can tell us whether or not a specific word if considered badspeak. This is what the bloom filter is for.

A Bloom Filter is a probabilistic data structure that can tell whether an item is not in the data structure OR it *maybe* is in it. As you might notice, the Bloom Filter never gives us a definite *yes* rather it tells us maybe. This is due to the fact that a word may share the same bit in a bit vector. In order for the word to be implemented in the bit vector, we would need to hash the word multiple times (in this

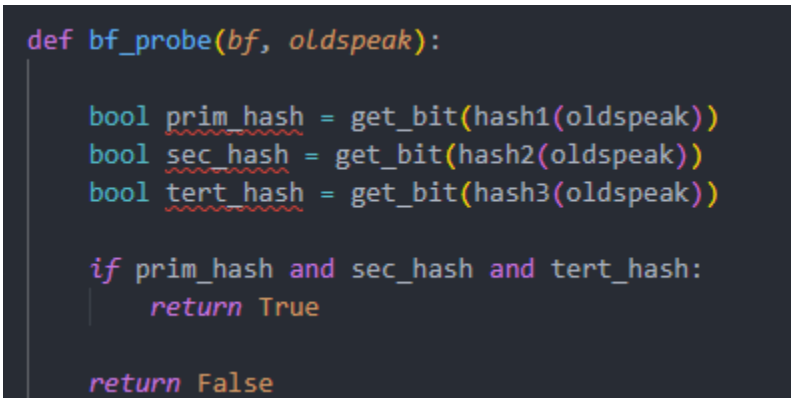
case, 3 times) in order to increase our chances of the word existing in the bit vector. So let's look at the implementation of the bitvector.

- BloomFilter *bf_create(uint32_t size)
    - This is the constructor for bloom filter.
    - We just need to create a bit vector of size *size* and set each salt to the corresponding hex values in salts.h
    - No pseudocode is needed here.
- void bf_delete(BloomFilter **bf)
    - This is the destructor of the Bloom Filter.
    - We would need to call bv_delete() in order to delete the bit vector. Also, we would need to free the bloom filter and set it to NULL.
    - No pseudocode is needed here
- uint32_t bf_size(BloomFilter *bf)
    - This just returns the size of the bit vector.
    - No pseudocode is needed here since we will just be returning bv_length().
- void bf_insert(BloomFilter *bf, char *oldspeak)
    - This function inserts oldpseak by hashing it 3 different times and setting the bit at each hash to 1.
    - The following pseudocode I made in python shows us how we would implement this function:
    - ```
      def bf_insert(bf, oldspeak):

          bv_set(hash1(oldspeak))
          bv_set(hash2(oldspeak))
          bv_set(hash3(oldspeak))

          return None
      ```
- bool bf_probe(BloomFilter *bf, char *oldspeak)
    - This function checks to see whether or not the word probably exists in the bit vector.
      ```

- In order to do this, all we have to do is hash the word three times and get the bit at each hash.
- This can be done with the following pseudo code that I made in python:



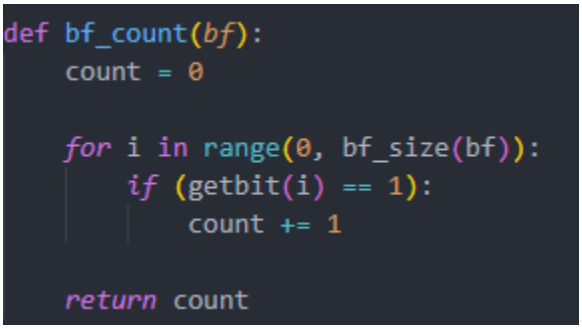
```
def bf_probe(bf, oldspeak):

    bool prim_hash = get_bit(hash1(oldspeak))
    bool sec_hash = get_bit(hash2(oldspeak))
    bool tert_hash = get_bit(hash3(oldspeak))

    if prim_hash and sec_hash and tert_hash:
        return True

    return False
```

- uint32_t bf_count(BloomFilter *bf)
  - This function tells us how many bits were set to 1 in the bit vector.
  - This can be done with the following pseudocode I made in python



```
def bf_count(bf):
    count = 0

    for i in range(0, bf_size(bf)):
        if (getbit(i) == 1):
            count += 1

    return count
```

Bit Vectors

In order for the bloom filter to work, we must use a bit vector since the bloom filter relies on bits to see whether or not a word exists. I will not be providing any pseudocode for this ADT since the code comments repo already provides the code on how to create the bit vectors and its functions.

- BitVector *bv_create(uint32_t length)
  - This constructs a bit vector array with a length of *length*.
- void bv_delete(BitVector **bv)
  - This deletes the bit vector and its bit vector array.
- uint32_t bv_length(BitVector *bv)

    - This returns the length of the bit vector.
    - All we need to do in this case is return bv->length.
- bool bv_set_bit(BitVector *bv, uint32_t i)
    - This function sets the bit to 1 at index *i*.
    - Refer to the code comments repo on how to implement this.
- bool bv_clr_bit(BitVector *bv, uint32_t i)
    - This function does the same as set_bit except we set the bit to 0 at index *i*.
- bool bv_get_bit(BitVector *bv, uint32_t i)
    - This function returns the bit at index *i*.
    - This means that either return a 1 or a 0.

Hash Tables

In order to determine whether or not the Bloom Filter probed the word correctly, we would need to use the hash table in order to confirm that it is true. The problem with hash tables that can be a pain are hash collisions. A hash collision is when 2 elements share the same hash value. Instead of using open addressing in order to prevent hash collisions, we will use a binary search tree to prevent those collisions.

- HashTable *ht_create(uint32_t size)
    - This is the constructor for the ADT. In order to construct this ADT, we will need to set the salt for the hash function, set the size to *size*, and then dynamically create an array of tree nodes.
- void ht_delete(HashTable **ht)
    - This deletes the underlying binary search tree and deletes the hash table itself.
- uint32_t ht_size(HashTable *ht)
    - This returns the hash table size.
    - This just returns ht->size
- Node *ht_lookup(HashTable *ht, char *oldspeak)

    - This hashes oldspeak, and uses the hash value to look up the corresponding newspeak value.
    - Remember, newspeak could either be a word or NULL.
    - The following pseudo code that I made shows us how it could be implemented:
    -
```
def ht_lookup(ht, oldspeak):
    bst_find(ht[hash(oldspeak)], oldspeak)
    return None
```

- void ht_insert(HashTable *ht, char *oldspeak, char *newspeak)
    - This function is like ht_lookup() however, we would be calling bst_insert() instead.
- uint32_t ht_count(HashTable *ht)
    - This iterates through the table and counts how many non-null trees there are.
- double ht_avg_bst_size(HashTable *ht)
    - This returns the bst_size() of each tree in the hashtable divided by the number of non-null trees in the table.
    - This is shown as: (sum of bst_size() of each element in array)/ht_count()
- double ht_avg_bst_height(HashTable *ht)
    - This returns the sum of heights of each tree in the array divided by the number of non-null trees in the array.
    - This can be shown as: (sum of bst_height() of each tree in array)/ht_count()

Binary Search Trees

In order to deal with hash collisions, we will be using binary search trees to deal with scenarios when two items are hashed into the same index. NOTE: I will not be using pseudo code for the BST since I will be using the code that was given to use in lecture 18 in order to implement the BST.

- Node *bst_create(void)
    - This is the constructor for the binary search tree.
    - All this has to do is return a NULL since we want an empty tree.
- void bst_delete(Node **root)
    - This is the deconstructor for the tree.

    - In order to delete the tree, all we have to do is walk down the tree in a postorder traversal order and call node_delete()
- uint32_t bst_height(Node *root)
    - This calculates the height of a tree.
    - Refer to lecture 18 for calculating the tree height
- uint32_t bst_size(Node *root)
    - This calculates how many nodes are in the tree.
    - This could be done by traversing down the left and the right children of the node and then adding 1 to the total.
- Node *bst_find(Node *root, char *oldspeak)
    - This finds the node with the corresponding key. The key in this case is the oldspeak word.
    - This could be implemented from the lecture 18 slides
- Node *bst_insert(Node *root, char *oldspeak, char *newspeak)
    - This inserts a node to the binary search tree. Words that are lexicographically less than the node's oldspeak are put to the right of the node. Else, it goes on the right.
    - This can be implemented using the code from the lecture 18 slides.
- void bst_print(Node *root)
    - This prints the binary search tree using *inorder* traversal.

Nodes

The binary search tree will need to use nodes in order to make a tree

- Node *node_create(char *oldspeak, char *newspeak)
    - This is the constructor for the Node ADT.
    - To implement this, all we need to do is strdup() the oldspeak and newspeak parameters.
        - If either the oldspeak or newspeak parameters are null, explicitly set the corresponding fields to NULL.
- void node_delete(Node **n)

    - In order to delete the node all you need to do is delete the node itself and to set the node to NULL.
- void node_print(Node *n)
    - This prints only the oldspeak and newspeak fields of the node.
    - **NOTE**: DO NOT TRAVERSE THE LEFT OR RIGHT CHILDS.
    - The specific print statements to use are given to us in the assignment doc.

Miscellaneous C Files that were provided

In addition to the ADTs, we were given the speck.c and parser.c files. The speck.c file contains the hash implementation that will hash the oldspeak word and return an index that can be used for the bloom filter or the hash table.

The provided parser file takes in a regex word and parsers text files using the provided regex word.

Banhammer

This is the .c file that will hold the main() for our program. No pseudocode would be needed here since the Task section (bullet point 9) already tells us the specifics of how to implement the program. The specifics are so detailed that it is basically the pseudocode for the program. Therefore, I will not be showing any pseudocode for banhammer.c

Credits

- For bv.c I used Prof. Long's bv8.h bit vector code in order to help me implement bit vectors.
- I attended Eugene's 11/30/21 and 11/23/21 sections. In those sections, he gave advice on regex and gave us some pseudo code for the assignment.
- On 11/24/21 I attended Eric's tutoring section. In that section he gave us pseudo code to help us with the assignment.
- For implementing the binary search tree, I used lecture 18 to help implement some functions of the ADT.
- Ultimately, all code that was used is cited in the source files.