Eduardo Garcia
egarc127@ucsc.edu
11/28/2021

CSE 13S Fall 2021
Assignment 6: The Great Firewall of Santa Cruz:
Bloom Filters, Binary Trees and Hash Tables

Design Document

Introduction

This program will make sure that the citizens of the People's Republic of Santa Cruz will follow the new form of speaking, *newspeak*. This will be done through banning words labeled as *badthink*, and updating words from a list of *oldspeak* to the fresh and reviving *newspeak*. This will be carried out using the Bloom filter, Binary Tree, and Hash table ADTs.

Bloom filter ADT(bf.c)

We will be using a bit vector as well as 3 provided salts from the **`salts.h`** provided file.

Pseudocode

`BloomFilter *bf_create(uint32_t size)`

```
    Allocate space for bloom filter
    Use provided salts
    Create a bitVector with size size
```

`void bf_delete(BloomFilter **bf)`

```
    Call bv_delete
    Free bf and all space allocated to it
```

`uint32_t bf_size(BloomFilter *bf)`

```
    Return bv_length()
```

`void bf_insert(BloomFilter *bf, char *oldspeak)`

```
    Set bf.bv[hash(salt1, oldspeak) % size]
    Set bf.bv[hash(salt2, oldspeak) % size]
    Set bf.bv[hash(salt1, oldspeak) % size]
```

`bool bf_probe(BloomFilter *bf, char *oldspeak)`

```
If bv_get_bit(bf.bv[hash(salt1, oldspeak) % size]) and bv_get_bit
(bf.bv[hash(salt1, oldspeak) % size]) and bv_get_bit (bf.bv[hash(salt1, oldspeak)
% size])
    Return true
Return false
```

`uint32_t bf_count(BloomFilter *bf)`

```
Returns number of set bits
```

`void bf_print(BloomFilter *bf)`

```
Call bv_print(bf.bv)
```

Hashing with the SPECK Cipher(vi speck.c)

This file will implement our `hash()` that will be crucial to our program

Pseudocode

`uint32_t hash(uint64_t salt[], char *key);`

```
This will take our key, and perform the salt salt and return a number
```

Bitvectors(bv.c)

This is what the Bloom Filter will use. included are way to set, get, clear, specific bits from each our vector of bytes(much of this has already been written in assignment 5, and will be copied over)

Pseudocode

`BitVector *bv_create(uint32_t length)`

```
Bv.length = length
Vector = Allocate an array of length uint32_ts, all bits set to 0
```

`void bv_delete(BitVector **bv)`

```
free(*bv.vector)
free(*bv)
**bv = Null
```

uint32_t bv_length(BitVector *bv)

```
    Return bv.length
```

bool bv_set_bit(BitVector *bv, uint32_t i)

```
    If i > bv.length
        Return false
    bv.vector[i/8] |= 1<<(i%8)
    Return true
```

bool bv_clr_bit(BitVector *bv, uint32_t i)

```
    If i > bv.length
        Return false
    bv.vector[i/8] &= NOT(1<<(i%8))
    Return true
```

bool bv_get_bit(BitVector *bv, uint32_t i)

```
    If i > bv.length
        Return false
    x = 0
    x = 1 << (i % 8)
    x &= (bv.vector[i / 8])
    if x > 0
        Return false
    Return true
```

void bv_print(BitVector *bv)

```
    For (byte in vector)
        For(bit in byte)
            If bv_get_bit()
                print("1")
            Else
                print("0")
        Print newline
```

Hash Table ADT(ht.c)

The hash table will be used to store corresponding *oldspeak* words to their *newspeak* counterparts. *Oldspeak* words without a *newspeak* equivalent are considered *badspeak*.

Pseudocode

`HashTable *ht_create(uint32_t size)`

```
    Allocate space for hash table
    Allocate space for array of pointer to nodes of array size size
```

`void ht_delete(HashTable **ht)`

```
    free(*ht.trees)
    free(*ht)
    *ht = null
```

`uint32_t ht_size(HashTable *ht)`

```
    Return ht.size
```

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

```
    if(ht.trees[hash(salt_table, oldspeak)])
        Return true
    Return false
```

`void ht_insert(HashTable *ht, char *oldspeak, char *newspeak)`

```
    For the returned has value, perform an addition using bst_insert
```

`uint32_t ht_count(HashTable *ht)`

```
    Count = 0
    For hash in ht.trees
        If ht.trees[hash]
            Count +=1
    Return count
```

`double ht_avg_bst_size(HashTable *ht)`

```
    Count = 0
    Num = ht_count()
    For hash in ht.trees
        If ht.trees[hash]
```

```
                Count +=bst_size()
        Return count / num
double ht_avg_bst_height(HashTable *ht)
        Count = 0
        Num = ht_count()
        For hash in ht.trees
                If ht.trees[hash]
                        Count +=bst_height()
        Return count / num

void ht_print(HashTable *ht)

        For hash in ht.trees
                If ht.trees[hash]
                        Print hash root node
```

Node ADT (node.c)

Nodes will be used for solving hash collisions, along with binary search trees. If an *oldspeak* word is *badspeak,* its value will be Null, otherwise it will be the corresponding *newspeak*.

Pseudocode

```
Node *node_create(char *oldspeak, char *newspeak)
        Allocate space for the node
        If there is newspeak translation:
                New  = strdup() the newspeak
                N.newspeak = new
        Old = strdup() the oldspeak
        N.oldspeak = old
        N.left = null
        N.right = null
        Return n
void node_delete(Node **n)
        Temp = *n
        free(temp.newspeak)
        free(temp.oldspeak)
        free(temp)
        Temp = null
void node_print(Node *n)
        If there is a newspeak term:
```

```
        Print(n.oldspeak -> n.newspeak)
    Else
        print(n.oldspeak)
```

Binary Search Tree ADT(bst.c)

Along with node.c, this will help resolve hash collisions. This will be done in a standard order. This means that for any given node, its left node will be lexicographically less than it, and its right node will be lexicographically more than it. This will be done with strcmp(). Also note that a lot of this code was already given via the lecture slides, and adjusted in order to account for `char *` instead of `int`.

Pseudocode

`Node *bst_create(void)`

```
    Return null
```

`void bst_delete(Node **root)`

```
    If root:
        bst_del(root.left)
        bst_del(root.right)
        node_del(root)
```

`uint32_t bst_height(Node *root)`

```
    If root:
        Return 1+ max(bst_height(root.left), bst_height(root.right)
    Return 0(if the root does not exist, it is worth 0)
```

`uint32_t bst_size(Node *root)`

```
    If root:
        Return 1 + bst_size(root.left) + bst_size(root.right)
    Return 0
```

`Node *bst_find(Node *root, char *oldspeak)`

```
    If root:
        if(root.oldspeak > oldspeak)
            Return bst_find(root.left, oldspeak)
        Else if (root.oldspeak < oldspeak)
            Return bst_find(root.right, oldspeak)
        Return root(returns root)
    Return null(nothing was found)
```

`Node *bst_insert(Node *root, char *oldspeak, char *newspeak)`

```
    If root:
        if(root.oldspeak > oldspeak)
            Return bst_insert(root.left, oldspeak)
        Else if (root.oldspeak < oldspeak)
            Return bst_insert(root.right, oldspeak)
        Return root(returns root)
    Return node_create(oldspeak, newspeak) (make the new node)
```

`void bst_print(Node *root)`

```
    If (root)
        bst_delete(root.left)
        bst_delete(root.right)
        node_delete(root)
```

Lexical Analysis with Regular Expressions(in banhammer.c)

Since we need to make sure we correctly parse words through stdin and make sure that only the correct format, we will have to write a regex. Our character should contain A-Z, a-z, 0-9, and _. We also must allow for contractions and hyphenations.

Regex ([A-Za-z0-9_]+['-]{1})*[A-Za-z0-9_]+ and explanation

[A-Za-z0-9_]+

One or more characters with A-Z, a-z, 0-9, and _(for second half, assures that words with non-alphanumeric characters are read in regardless)

['-]{1}

Exclusively Only 1 Hyphen or - for contractions and hyphenations

([A-Za-z0-9_]+['-]{1})*

O or more Words containing characters and hyphen XOR apostrophe

Main program(banhammer.c)

The following is the main overhead view of our entire program. The messages are taken from `messages.h`, and help the user know if they are using correct language, *oldspeak*, *mixspeak*(both oldspeak AND badthink), or *badthink*.

Pseudocode

```
Arrest = Make a bst for *badthink* words
Correct = Make a bst for *oldspeak* words
Parse command line arguments
Ht = ht_create(size of table)
Bf = bf_create(size of filter)
Scan in words from badspeak.txt
    ht_insert(ht, oldspeak word, NULL)
    bf_insert(bf, oldspeak word)
Scan in words from newspeak.txt
    ht_insert(ht, oldspeak word, NULL)
    bf_insert(bf, oldspeak word, newspeak word)
Read in words from stdin
    Only parse words that are in our regex
    If word has probably been added in bloomfiler:
        Temp = ht_lookup(ht, word)
        if(temp):
            if(temp->newspeak):
                bst_insert(correct, word, temp->newspeak)
            Else:
                bst_insert(correct, word, NULL)
If(arrest and correct have elements):
    Print mixspeak message
    Print arrest words
    Print correct words
Else if (arrest has elements)
    Print badspeak message
    Print arrest words
Else if(correct has elements)
    Print goodspeak message
    Print correct words
If( stats option was chosen):
    Print our state message
Free all allocated space and close files
```

Conclusion

We have learned how to use and implement the Bloom Filter ADT, Hash Table ADT, and Binary Search Tree ADT. The citizens of People's Republic of Santa Cruz will now be safe! And we have secured our position as an outstanding citizen and worker.