Mahyar Vahabi
12/05/2021

# Asgn 7 Design Document

*The Great Firewall of Santa Cruz*

**Purpose:** The purpose of this program is to filter and censsor words that are forbidden in writting and display. Just like in the novel *1984* by George Orwell, the authoritorian government has a department where they use a newspeak set of words and replace the oldspeak words. There are special workers, like the main character Winston, who are tasked to censor/filter and replace these words by hand, but why do that when you can make a program that does that for you.

**Bf:** A Bloom filter is a space-efficient probabilistic data structure, used to test whether an element is a member of a set. Elements can be added to the set but not removed from it.

**Bv:** A bit vector is an ADT that represents a one dimensional array of bits, the bits in which are used to see if something is true or false. This is similar to a previous ADT in assignment 5.

**Ht**: A hash table is a data structure that maps keys to values and provides fast, O(1), look-up times. It does so typically by taking a key k, hashing it with some hash function h(x), and placing the key's corresponding value in an underlying array at index h(k). This is the perfect way not only to store translations from oldspeak to newspeak, but also as a way to store all prohibited oldspeak words without newspeak translations.

**Node:** To resolve hash collisions, we ought to use a binary search tree; any trees are made up of nodes, so it is best to implement a code for node functions that consists of oldspeak and newspeak and pointers to left and right children.

**Bst:** This is a fast data structure that allows us to maintain a sorted list of keys/numbers.

**Pseudocode:**

**Bf.c:**

**// defined in the design doc**

```
struct BloomFilter {
    uint64_t primary [2]; // Primary hash function salt .
    uint64_t secondary [2]; // Secondary hash function salt .
    uint64_t tertiary [2]; // Tertiary hash function salt .
    BitVector * filter ;
 };
BloomFilter *bf_create(uint32_t  size) {
    Dynamically allocate BloomFilter *bf
    If bf exists {
        Set bv primary/secondary/tertiary [0/1] to SALT_PRIMARY/SECONDARY/TERTIARY_LO/HI
        Set bf.filter to bv_create(size)
        If there is no bf.filter {free it then set it to NULL}
    }
    Return bf
}
void bf_delete(BloomFilter **bf) {
    If bf exists { delete bf.filter using bv_delete, free bf, set bf to NULL}
    Return
}
uint32_t bf_size(BloomFilter *bf) { Return the bv_length of the filter }
```

```
void bf_insert(BloomFilter *bf, char *oldspeak) {
    Set bit (filter, hash(primary/secondary/tertiary, oldspeak) modulus the length of the filter)
    Return }
bool bf_probe(BloomFilter *bf, char *oldspeak) {
    Basically copy and paste bf_insert code, but put it inside the if parameter, and return true, else
return false }
uint32_t bf_count(BloomFilter *bf) {
    Set a variable for counter
    Loop ‘i’ through 0-length of the filter
        check if get bit(filter, i) {
            Increment counter }
    } return counter
}
void bf_print(BloomFilter *bf) {
    Print filter using bv_print and return }
```

**Bv.c:**

```
// struct was given in assignment doc
struct BitVector {
    uint32_t length;
    uint8_t *vector;
};
```

```
// divert bits to bytes
static inline size_t bytes(size_t bits) {
    return bits % 8 ? bits / 8 + 1 : bits / 8;
}
BitVector *bv_create(uint32_t length) {
    Dynamically allocate for BitVector *bv
    Set length to itself
    Allocate vector to bytes(length)
    If vector doesn't exsist { free bf and set it to NULL }
}
void bv_delete(BitVector **bv) {
    If bv and its vector exsists {
        Free the vector and set it to NULL
        Free bv and set it to NULL
    }
}
uint32_t bv_length(BitVector *bv) {
    Return the length }
bool bv_set_bit(BitVector *bv, uint32_t i) {
    If 'i' is less then the length of bv {
        Set bit in the bit vector by doing a bitwise operate: vector[i >> 3] |= (1 << (i & 7))
        Return true
    } else return false }
```

```
bool bv_clr_bit(BitVector *bv, uint32_t i) {
    If ‘i’ is less the length of bv {
        Clear bit from the vector using bitwise operate: vector[i >> 3] &= ~(1 << (i & 7))
        Return true;
    } else return false
}
bool bv_get_bit(BitVector *bv, uint32_t i) {
    If ‘i’ is less the length of bv {
        Get bit from vector and return bool for bitwise operate: vector[i >> 3] >> (i & 7)) & 1) == 1
    } return false
}
void bv_print(BitVector *bv) {
    Loop ‘i’ through 0-length of bv {
        Print bv_get_bit(bv, i)
    }
}
```

**Ht.c:**

```
// struct from assignment PDF
struct HashTable {
    uint64_t salt[2];
    uint32_t size;
    Node **trees;
};
```

```
HashTable *ht_create(uint32_t  size) {
    Dynamically allocate HashTable *ht
    If ht exists {
        Set ht.salt[0/1] to SALT_HASHTABLE_LO/HI
        Set size to itself
        Allocate trees to size of Node
        If trees doesn't exists {
            Free ht and set it to NULL }
    } return ht
}
void ht_delete(HashTable **ht) {
    If ht exists {
        Iterate through each element of trees[i] and delete using bst_delete
        Free trees and set it to NULL, free ht and set it to NULL
    }
}
uint32_t ht_size(HashTable *ht) {
    Return size }
Node *ht_lookup(HashTable *ht, char *oldspeak) {
    Increment looups
    Set i to hash(salt, oldspeak) modulus size
    If trees[i] is empty return NULL else return bst_find(trees[i], oldspeak)
}
```

```
void ht_insert(HashTable *ht, char *oldspeak, char *newspeak) {
    Set i to hash(salt, oldspeak) modulus size
    If trees[i] is empty {
        Initialize using bst_create
    Then use bst_insert(trees[i], oldspeak, newspeak) and return
}
uint32_t ht_count(HashTable *ht) {
    Set a variable for counter
    Iterate ‘i’ through 0-size {
        If trees[i] exists encrement counter
    } return counter
}
void ht_print(HashTable *ht) {
    Iterate “i” through 0-size {
        If trees[i] exists { bst_print(trees[i]) }
    }
}
```

**Node.c:**

```
Node *node_create(char *oldspeak, char *newspeak) {
    Dynamically allocate for Node *n
    If n exists {
        Set right and let child to NULL
        If oldspeak and newspeak exists {
            Dynamically allocate oldspeak/newspeak and set it to themselves; if empty set to NULL
        }
    Return n
}
void node_delete(Node **n) {
    If n exists {
        Free oldspeak set it to NULL, free newspeak set it to NULL, free n set it to NULL
    } return
}
void node_print(Node *n) {
    If oldspeak and newspeak exists {
        Print both
    } else print just the regular oldspeak
    Return
}
```

**Bst.c**

```
Node *bst_create() {
    Return null;
}
void bst_delete(Node **root) {
    If root exists {
        Recursivley delete left and right node; delete root
}
uint32_t bst_height(Node *root) {
    If root exists {
        Return the max value that is either right or left root
    }
}
uint32_t bst_size(Node *root) {
    Return the addition of bst size of (left and right root) and add a 1
}
Node *bst_find(Node *root, char *oldspeak)
    If root and oldspeak exists
        While string comparison of root oldspeak and char oldspeak is not equal
            If string comparison of root oldspeak is greater than char oldspeak
                Set root to root left
            Else set root to root right
    Return root
```

```
Node *bst_insert(Node *root, char *oldspeak, char *newspeak) {
    Set a node ‘x’ to root and ‘y’ to NULL
    if  root exists and oldspeak exists) {
        while x exists {
            Set y to x
            if  string compare x->oldspeak is greater than oldspeak {
                Set x to x->left
            } else {
                Set x to x->right;
            }
        }
        if string compare y->oldspeak is greater than oldspeak {
            Set y->left to node_create(oldspeak, newspeak);
        } else {
            Set y->right node_create(oldspeak, newspeak);
        }
    } else {
        Set y to node_create(oldspeak, newspeak);
    }
    return y;
}
```

**Banhammer.c:**

Include all header files and variables.

Initiate a command line option using get opt, switch cases and return.

Initialize your Bloom filter and hash table using user input or default values.

Open two FILES (“newspeak.txt”, “badspeak.txt”) and check if they are null

Read in a list of badspeak words with fscanf() and insert them in bf and ht.

Read in a list of oldspeak and newspeak pairs with fscanf() and insert both old and newspeak to bf and ht.

Set two nodes (mixed, bad) to bst_create.

For each word that is read in (usinng the parser),

turn the words into a lowercase;

If the word has most likely been added to the Bloom filter (bf_probe() == TRUE),

then set a node ‘n’ to ht_lookup for word;

then check if n exists;

then check if n->oldspeak and n->newspeak exists and set mixed to bst insert for mixed

Else if n->oldspeak exists

Set bad to bst insert for bad with a empty newspeak

If stats is called

Print: Average binary search tree size, Average binary search tree height, Average branches traversed, Hash table load, Bloom filter load.

**Parser.c and speck.c were already implemented on the resources**