asgn7: The Great Firewall of Santa Cruz - Bloom Filters, Binary Search Trees and Hash Tables

Student name: *Serjo Barron*

Course: *CSE13S* – Professor: *Prof. Long*
Due date: *11/28/21*

1 Program

This program will simulate a scenario in which you are the leader of a glorious nation known as the Glorious People's Republic of Santa Cruz (GPRSC for short). As the leader of this glorious nation you want the inhabited internet to be free of badthink, yet due to the masses that task seems quite daunting and impossible to keep track of.
As such, in order to keep track of the communication across you nation and uphold the standards of properthink, we will make use of 2 abstract data structures, a bloom filter and binary search trees. Both data structures will allow to store information of badspeak and newspeak words, which will allow to keep information of what words are proper and improper to decide whether a citizen is in need of guidance.
With a way of storing data of oldspeak and newspeak pairs, you can then individually look up each word to determine whether a word said by a citizen is not using a proper newspeak translation, is badspeak, or a combination of the two. Based on whether any of these words when looked up in the bloom filter and binary search trees match will decide whether a citizen is given a reprimanding message detailing potential crimes.

2 Required Files

1. banhammer.c
    - This file will include the main() function of the program and will be the main point of user interaction.
2. messages.h
    - This header file will include convenient strings which are the reprimanding messages of the GPRSC.
3. salts.h
    - This header file includes defined macros for the upper and lower 64-bits of the salts used in the bloom filter and hash table.
4. speck.h

- This header file includes the function prototype of the Hashing ADT.

5. speck.c
    - This file will include the implementation of the Hashing ADT. (given implemented by Prof. Long)
6. ht.h
    - This header file includes the function prototypes of the Hash Table ADT.
7. ht.c
    - This file will include the implementation of the Hash Table ADT.
8. bst.h
    - This header file includes the function prototypes of the Binary Search Tree ADT.
9. bst.c
    - This file includes the implementation of the Binary Search Tree ADT.
10. node.h
    - This header file includes the function prototypes of the Node ADT.
11. node.c
    - This file includes the implementation of the Node ADT.
12. bf.h
    - This header file includes the function prototypes of the Bloom Filter ADT.
13. bf.c
    - This file includes the implementation of the Bloom Filter ADT.
14. bv.h
    - This header file includes the function prototypes of the Bit Vector ADT.
15. bv.c
    - This file includes the implementation of the Bit Vector ADT.
16. parser.h
    - This header file includes the function prototypes of the Parsing ADT.
17. parser.c
    - This file includes the implementation of the Parsing ADT. (given implemented by Prof. Long)

18. Makefile

    - This file will link all the required ADT's to keygen, encrypt, and decrypt, and will produce their corresponding executables/binaries.

19. README.md

    - A file in markdown syntax which contains a program description and the instructions on how to use and run the program.

20. DESIGN.pdf

    - The design process of the program (this document).

3 Psuedocode and Function Descriptions

3.1 Bloom Filter

Constructor for a bloom filter. The primary, secondary, and tertiary low and high 64-bits are defined macros in salt.h.

```
bf_create(size):
        bf = malloc memory for filter
        // set the salts to their upper and lower 64-bits
        primary[0] = SALT_PRIMARY_LO
        primary[1] = SALT_PRIMARY_HI
        secondary[0] = SALT_SECONDARY_LO
        secondary[1] = SALT_SECONDARY_HI
        tertiary[0] = SALT_TERTIARY_LO
        tertiary[1] SALT_TERTIARY_HI
        // make bit vector
        filter = bv_create(size)
        if mem allocation for filter fails:
                free(bf)
                set bf ptr to NULL
        return bf
```

Destructor for a bloom filter.

```
bf_delete(**bf):
        if bf and filter exists:
                bv_delete(filter)
                free(bf)
                set bf ptr to NULL
```

This function will return the size of a bloom filter (the bit length of the bit vector "filter").

```
bf_size(*bf):
        return bv_length(bf->filter)
```

This function will will insert oldspeak into the bloom filter. The indices of which bit to set is calculated by hashing the oldspeak with the three salts modulus the size of the filter. The 3 corresponding bits are then set using bv_ set_ bit().

```
bf_insert(*bf,*oldspeak):
        bv_set_bit(bf->filter,hash(primary,oldspeak) % bf_size(bf))
        bv_set_bit(bf->filter,hash(secondary,oldspeak) % bf_size(bf))
        bv_set_bit(bf->filter,hash(tertiary,oldspeak) % bf_size(bf))
```

This function will probe (check the bits) of the bloom filter by once again hashing the three salts with oldspeak modulus the size of the filter. If the 3 bits are set (1) then return true, otherwise return false.

```
bf_probe(*bf,*oldspeak):
        first = bv_get_bit(bf->filter,hash(primary,oldspeak) % bf_size(bf))
        second = bv_get_bit(bf->filter,hash(secondary,oldspeak) % bf_size(bf))
        third = bv_get_bit(bf->filter,hash(tertiary,oldspeak) % bf_size(bf))
        // returns true if first, second, and third bits are set
        // false otherwise
        return first and second and third
```

This function will return the number of set (1) bits in the filter.

```
bf_count(*bf):
        set_bits = 0
        for i in range(0,bf_size(bf)):
                if bv_get_bit(bf->filter,i) == true:
                        set_bits += 1
        return set_bits
```

A function for debugging purposes that will print out the bits of a bloom filter.

```
bf_print(*bf):
        bv_print(bf->filter)
```

3.2 Bit Vectors

A helper function (for bv_ create) to calculate the ceiling of two numbers x and y. The computed ceiling of the numbers will be returned.

```
ceiling(x,y):
        c = (x + y - 1) / y
        return c
```

Constructor for a bit vector that will be able to hold length specified amount of bits. And since this is an array, each element will be a byte, while length is the number of bits. Hence, the size of the vector (in bytes) will be calculated as the ceiling of $\frac{length}{8}$.

```
bv_create(length):
        bv = malloc memory for bit vector
        if vector exists:
                set specified length
                vector = calloc memory for ceiling(length,8) bits
                if mem allocation for vector fails:
                        free(bv)
                        set bv ptr to NULL
```

Destructor for a bit vector.

```
bv_delete(**bv):
        if bv and vector exists:
                free(vector)
                free(bv)
                set bv ptr to NULL
```

This function returns the length of a bit vector.

```
bv_length(*bv):
        return bv->length
```

This function will set the i^{th} bit of a bit vector to 1. The position of the byte in the vector to set a bit in will be calculated as $\frac{i}{8}$.

```
bv_set_bit(*bv,i):
        shift = 0x1 left-shift by i % 8 bits
        bv->vector[i / 8] bitwise-or with shift
```

This function will clear the i^{th} bit of a bit vector to 0. The position of the byte in the vector to clear a bit in will be calculated as $\frac{i}{8}$.

```
bv_clr_bit(*bv,i):
        shift = 0x1 left-shift by i % 8 bits
        bv->vector[i / 8] bitwise-and not(shift)
```

This function will retrieve (get) the i^{th} bit of a bit vector. The function will return true if and only if the bit retrieved is 1, and will return false otherwise.

```
bv_get_bit(*bv,i):
        shift = bv->vector[i / 8] right-shift by i % 8 bits
        if shift bitwise-and 0x1 == 1:
                return true
        return false
```

A function for debugging purposes that will print the bit vector (0's and 1's).

```
bv_print(*bv):
        for i in range(0,bv->length):
                if bv_get_bit(bv,i) == true:
                        print a 1
                else:
                        print a 0
        print newline
```

3.3 Hash Tables

The constructor for a hash table. The size defines the number of BST (binary search trees) that the hash table can have. This function will also set the low and high 64-bits of the salt using macros defined in salts.h.

```
ht_create(size):
        ht = malloc memory for hash table
        if hash table exists:
                salt[0] = SALT_HASHTABLE_LO
                salt[1] = SALT_HASHTABLE_HI
                set specified size
                trees = calloc memory for size amount of trees
                if calloc fails for trees:
                        free(ht)
                        set ht ptr to NULL
        return ht
```

The destructor for a hash table. This will require iterating over the hash table (size), and deleting every BST that exists.

```
ht_delete(**ht):
        for i in range(0,size):
                if tree exists in trees:
                        bst_delete(ht->trees[i])
        free(ht->trees)
        free(ht)
        set ht ptr to NULL
```

This function returns the hash tables size.

```
ht_size(*ht):
        return ht->size
```

This function will search for an entry (root) in the hashtable that contains oldspeak. The specific index of the entry will be calculated by taking hashing salt and oldspeak modulus the size of the hash table. If the node containing oldspeak is found, return that node (using bst_ find()), otherwise return a NULL pointer.

```
ht_lookup(*ht,*oldspeak):
        // hash oldspeak to find index of BST
        i = hash(ht->salt,oldspeak) % ht_size(ht)
        t = ht->trees[i]
        if t exists:
                return bst_find(t,oldspeak)
        return NULL ptr
```

This function will insert oldspeak and newspeak into the hash table. The specific index is once again calculated by hashing salt and oldspeak modulus the size of the hash table. If the BST exists, then insert the specified oldspeak and newspeak using bst_ insert(). Otherwise, a BST does not exist prior to inserting, so create one and then insert.

```
ht_insert(*ht,*oldspeak,*newspeak):
        // hash oldspeak to find index of BST
        i = hash(ht->salt,oldspeak) % ht_size(ht)
        t = ht->trees[i]
        if t exists:
                bst_insert(t,oldspeak,newspeak)
        else:
                t = bst_create()
                t = bst_insert(t,oldspeak,newspeak)
```

This function will return the number of non-NULL BST's (binary search trees) in the hash table. This will be done by iterating over the ht_ size().

```
ht_count(*ht):
        num_bst = 0
        for i in range(0,ht_size(ht)):
                if ht->trees[i] != NULL:
                        num_bst += 1
        return num_bst
```

```
ht_avg_bst_size(*ht):
        size_sum = 0
        for i in range(0,ht_size(ht)):
                size_sum += bst_size(ht->trees[i])
        // use type conversion to double to
        // calculate avg_size
        avg_size = size_sum / ht_count(ht)
        return avg_size
```

```
ht_avg_bst_height(*ht):
        height_sum = 0
        for i in range(0,ht_size(ht)):
                height_sum += bst_height(ht->trees[i])
        // use type conversion to double to
        // calculate avg_height
        avg_height = height_sum / ht_count(ht)
        return avg_height
```

A function for debugging purposes that will iterate over the hash table and print existing BST's.

```
ht_print(*ht):
        for i in range(0,ht_size(ht)):
                if ht->trees[i] != NULL:
                        bst_print(ht->trees[i])
```

3.4 Nodes

The constructor for a node. A node will be created by taking the copy of oldspeak and newspeak, and have its right and left children initialized as NULL.

```
node_create(*oldspeak,*newspeak):
        n = malloc mem for node
        n->oldspeak=NULL
        n->newspeak=NULL
        // use strdup() to copy old/newspeak
        // only if they are not NULL
        if oldspeak:
                n->oldspeak = strdup(oldspeak)
        if newspeak:
                n->newspeak = strdup(newspeak)
        // initialzie right & left child as NULL
        n->left = NULL
        n->right = NULL
        return n
```

The destructor for a node.

```
node_delete(**n):
        if node exists:
                if n->oldspeak exists:
                        free(n->oldspeak)
                if n->newspeak exists:
```

```
            free(n->newspeak)
        free(n)
        set n ptr to NULL
```

A function for debugging purposes that will print oldspeak and/or newspeak depending on what is not NULL.

```
node_print(*n):
        if oldspeak and newspeak is not NULL:
            print oldspeak and newspeak
        if only oldspeak is not NULL:
            print oldspeak
```

3.5 Binary Search Trees

Helper function to find the max between to numbers *a* and *b*. Implementation based off of Prof. Long (lecture 18).

```
max(a,b):
        // if a > b return a
        // otherwise return b
        return a > b ? a : b
```

Constructor for the binary search tree. The constructed tree will initially be empty (NULL).

```
bst_create():
        root = NULL
        return root
```

Destructor for a binary search tree starting at the root. From the root, this function will walk the tree using a postorder traversal to delete (free the memory) of each node of the tree.

```
bst_delete(**root):
        if root exists:
            bst_delete(root->left) // traverse left
            bst_delete(root->right) // traverse right
            node_delete(root)
```

This function returns the height of the binary search tree.

```
bst_height(*root):
        if root exists:
            height = 1 + max(bst_height(root->left),bst_height(root->right))
            return height
        return 0
```

This function returns the size of a binary search tree, where size is the number of nodes in the tree.

```
bst_size(*root):
        if root exists:
                return 1 + bst_size(root->left) + bst_size(root->right)
        return 0
```

This function searches the binary tree for a node containing oldspeak, beginning from the root of the tree. If a root is found containing the same oldspeak as specified, then the root node is returned. Otherwise a null pointer is returned.

```
bst_find(*root,*oldspeak):
        // compare strings using strcmp()
        if root exists:
                if root->oldspeak == oldspeak:
                        return root // found the match
                if root->oldspeak > oldspeak:
                        return bst_find(root->left,oldspeak) // recurse left
                else if root->oldspeak < oldspeak:
                        return bst_find(root->right,oldspeak) // recurse right
        else:
                return NULL ptr
```

This function will insert a new node containing oldspeak and newspeak into the binary tree. There are 3 cases to account for when inserting a node. The first case is we create a new node as a root (which is the last return statement in the pseudocode below). The second case is that we recursively insert the key in the left subtree, and the third is the same but for the right subtree. It should be noted however that if a node containing oldspeak already exists in the subtree, then return a null pointer.

```
bst_insert(*root,*oldspeak,*newspeak):
        // compare strings using strcmp()
        if root exists:
                if root->oldspeak == oldspeak:
                        return root // found duplicate, dont insert
                if root->oldspeak > oldspeak:
                        // recurse left of BST
                        root->left = bst_insert(root->left,oldspeak,newspeak)
                else:
                        // recurse right of BST
                        root->right = bst_insert(root->right,oldspeak,newspeak)
                return root
        // else make a new root
        return node_create(oldspeak,newspeak)
```

This function performs an inorder traversal to print the nodes in a lexicographically order.

```
bst_print(*root):
        if root exists:
                bst_print(root->left)
                node_print(root)
                bst_print(root->right)
```

3.6 Parser/Regex

This program will make use of a provided parsing module (from the assignment documentation) to be able to read words from a file stream while using a regex (regular expression). The regex for this particular assignment will be defined as shown below.

```
([a-zA-Z0-9_]+)|(([a-zA-Z0-9_]+[-'])+[a-zA-Z0-9_]+)
```

The first portion of the regular expression "([a-zA-Z0-9_]+)" will be able to handle recognizing any word containing a character between the brackets. The second half of the expression "(([a-zA-Z0-9_]+[-'])+[a-zA-Z0-9_]+)" will be able to handle hyphenated words such as "one-sided" and contracted words like "i'm", where the word being connected by the hyphen or single quote follow the same constraints as the former expression.
Though as seen there is an or '|' symbol separating the two expressions, which allows to handle both cases as described above.

3.7 SPECK (Hashing)

The program will make use of a SPECK block cipher provided in the corresponding assignment documentation as a means of efficiently hashing. Hashing will be done by taking two parameters, the first being a 128-bit salt and a word to hash (the key). The salts are predefined macros in the salts.h header and are made up of an array of 2 unsigned 64-bit integers.

4 Banhammer

This file (banhammer.c) will include the defined main function of the program, and will be responsible for the CLI (command line options) and reading in words from stdin typed by the user and looking up those words in the bloomfilter and hashtable to decide whether a citizen is in need of reprimanding. This is done by first reading in the contents of "newspeak.txt" and "badspeak.txt" which contain a large list of oldspeak and newspeak words.
Afterwards, the program will then as mentioned read words typed from the user from stdin. This will be done until the end of stdin is specified (which is done by "ctrl + d"). Until then, the entered words will be categorized in two BST's, bad and new words. However, the word will only be put into either of the categorizes if and only if the word exists in the bloomfilter and hashtable.

```
while getopt loop:
        case 'h':
                print header = true
        case 't':
                take user specified hashtable size
        case 'f':
                take user specified bloomfilter size
        case 's':
                print stats = true
        default:
                print header = true

if specified:
        print synopsis message
        end program

ht = ht_create(hashtable size)
bf = bf_create(bloomfilter size)
regex_t re // set regex

if regcomp(re,WORD,REG_EXTENDED):
        // failed to compile regex
        ht_delete(ht)
        bf_delete(bf)
        print error message
        exit program

open "badspeak.txt" for reading
word = NULL

while word = next_word(badspeak,re) is not NULL:
        bf_insert(bf,word)
        ht_insert(ht,word,NULL)
clear_words()
close "badspeak.txt"

open "newspeak.txt" for reading

while word = next_word(newspeak,re) is not NULL:
        bf_insert(bf,word)
        tmp_word = strdup(word) // store first word
        word = next_word(newspeak,re) // word after
        ht_insert(ht,tmp_word,word)
        free(tmp_word)
clear_words()
close "newspeak.txt"
```

```
bad_words = bst_create() // BST's to store user specified words
new_word = bst_create()  // if they are bad or new words

// read words from stdin and catergorize them into bad_words or
// new_words depending on whether that word exists in the bloomfilter
while word = next_word(stdin,re) is not NULL:
        if bf_probe(bf,word) == true:
                if w = ht_lookup(ht,word) is not NULL
                        if w->oldspeak and w->newspeak: // old and newspeak exist
                                new_words = bst_insert() oldspeak and newspeak
                        else: // else only oldspeak exists
                                bad_words = bst_insert() only oldspeak

// bools to check whether bad or new words exist
// is true if the size of either BST is 1 or more
bad = bst_size(bad_words) > 0
new = bst_size(new_words) > 0

if not print stats: // only print reprimanding messages
        if bad and new:
                print mixspeak message
                bst_print(bad_words)
                bst_print(new_words)
        if bad and not new:
                print badspeak message
                bst_print(bad_words)
        if not bad and new:
                print goodspeak message
                bst_print(new_words)
else: // if stats is toggled, only print stats
        // the following stats are retrieved via function calls
        avg_sizeof_bst = ht_avg_bst_size(ht)
        avg_heightof_bst = ht_avg_bst_height(ht)

        // the following 3 calculations require type conversion
        // of each number used in the computation (to double)
        avg_traversed = branches / lookups
        ht_load = 100.0 * ht_count(ht) / ht_size(ht)
        bf_load = 100.0 * bf_count(bf) / bf_size(bf)

        print stats

bf_delete(bf)
ht_delete(ht)
bst_delete(bad_words)
```

```
bst_delete(new_words)
regfree(re) // free regex
end program
```

5 Error/Specific Case Handling

For the actual main function of this program (banhammer.c), the regex (regular expression) must be able to handle a multitude of inputs and scenarios. That being, it must handle all uppercase and lowercase characters. And it must also handle special characters such as underscores, apostrophes, and hyphens.
Since all the words included in newspeak.txt and badspeak.txt are all lowercase, if a word from stdin (standard in) is used, regardless of its case it can match with a word from newspeak or badspeak. An example of this is 'A' and 'a', which are to be considered as the same message. To handle this all characters inputting by the user will be converted to lowercase. Another thing to consider is in the event of failing to compile regex, in which an error message will be prompted to the user and the program will end prematurely.

6 Credit

1. The speck.c file which includes the functions needed for hashing and parser.c including functions needed for regex have been supplied in the corresponding resources repository, and have been specified to have been provided in the corresponding assignment 7 documentation by Prof. Long.

2. Function designs pertaining to the BST ADT have been inspired and based off of example code in the Trees and BST lecture (lecture 18) by Prof. Long.