Edison Chen

Assignment 7: The Great Firewall of Santa Cruz: Bloom Filters, Binary Trees, and Hash Tables

Program Description:

The purpose of this assignment is to filter out offensive speech through the use of a bloom filter and hash table as the one and only Beloved Leader of the Glorious People's Republic of Santa Cruz. The purpose of the bloom filter is to check if a list of prescribed words, *oldspeak*, are used by your citizens. The bloom filter is probabilistic and has the chance of producing a false positive, so the bloom filter will be implemented with three salts for three different hash functions to reduce the chances of a false positive. A hash table allows the leader to store not only translations from oldspeak to newspeak, but also a way to store oldspeak words without newspeak translations. The use of a hash table can distinguish citizens who are guilty of thoughtcrime, the use of oldspeak words that do not have newspeak translations, from the citizens that need counseling, the use of oldspeak words that have newspeak translations.

Files Included:

banhammer.c - contains main() and any other functions necessary to complete the assignment

messages.h - defines the mixspeak, badspeak, and goodspeak messages used in banhammer.c

salts.h - defines the primary, secondary, and tertiary salts to be used in Bloom filter implementation and also defines the salts used by the hash table

speck.h - defines the interface for the hash function using the SPECK cipher

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

ht.h - defines the interface for the hash table ADT

ht.c - contains the implementation of the hash table ADT

bst.h - defines the interface for the binary search tree ADT bst.c - contains the implementation of the binary search tree ADT

node.h - defines the interface of the node ADT

node.c - contains the implementation of the node ADT

bf.h - defines the interface for the Bloom filter ADT

bf.c - contains the implementation of the Bloom filter ADT

bv.h - defines the interface for the bit vector ADT

bv.c - contains the implementation of the bit vector ADT

parser.h - defines the interface for the regex parsing module

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

Makefile - file that builds the programs and formats all files to clang format

README.md - text file in markdown format that describes: the program, how to build the program, and how to run the program

DESIGN.pdf - pdf file that describes the program and its design

WRITEUP.pdf - pdf file with graphs that describes and compares the total number of lookups and average binary search tree branches traversed with varying hash table and bloom filter sizes.

Notes and Pseudocode:

bf.c:

- A bloom filter is represented as an array of m bits, or a bit vector
- False positive matches are possible, false negatives are not
- If any of the words used by the citizens are added to the bloom filter, further action must be taken
- The bloom filter will be implemented with three salts for three different hash functions to reduce the chances for a false positive

- The void bf_create(uint32_t size) function is the constructor for a Bloom filter and creates primary, secondary, and tertiary hash function salts

```
void bf_create(uint32_t size):
    Dynamically allocate memory for the BloomFilter struct using malloc()
    Set the lows and highs of the salts
    Return the created BloomFilter
```

- The void bf_delete(BloomFilter **bf) function is the destructor for the Bloom filter and should free any memory allocated by the constructor

```
void bf_delete(BloomFilter **bf):
    Free the memory of the pointer to bf
    Set the pointer to NULL
```

- The uint32_t bf_size(BloomFilter *bf) function returns the size of the Bloom filter, aka the number of bits that the Bloom filter can access

```
void bf_size(BloomFilter *bf):
    Return the length of the bit vector of BloomFilter
```

- The void bf_insert(BloomFilter *bf, char *oldspeak) function takes oldspeak and inserts it into the Bloom filter

```
void bf_insert(BloomFilter *bf, char *oldspeak):
    Hash oldspeak words with each of the three salts for three indices
    For each index, set the bits at the underlying bit vector
```

- The bool bf_probe(BloomFilter *bf, char *oldspeak) function probes the Bloom filter for oldspeak
- If all the bits are set, return true to show that oldspeak was added to the Bloom filter

```
bool bf_probe(BloomFilter *bf, char *oldspeak):
    Check each of the salts for the char containing oldspeak
    If each of the bits are set, aka not zero:
        return true
    return false
```

- The uint32_t bf_count(BloomFilter) function returns the number of set bits in the Bloom filter

```
uint32_t bf_count(BloomFilter):
    Have a for loop that checks every bit in the Bloom Filter
    If a bit in the filter is 1, aka set, then add one to the count
    Return count
```

- The void bf_print(BloomFilter *bf) function prints out the bits of a Bloom filter
- Utilizes the print function from bit vector

```
void bf_print(BloomFilter *bf):
    Use the bv_print() function with bf as an argument
```

bv.c:

- A bit vector is an ADT that represents a one dimensional array of bits
- Use the ceiling of n/8 uint8_ts instead of n
- Use bitwise operations to get, set, and clear a bit within a byte
- The BitVector *bv_create(uint32_t length) function is a constructor for a bit vector that holds length bits that are all initialized to 0, and if sufficient memory cannot be allocated, the function will return NULL

```
BitVector *bv_create(uint32_t length):
    Dynamically allocate memory for the BitVector using malloc()
    Initialize each bit of the vector to 0
    If vector cannot be initialized then return NULL
```

- The void bv_delete(BitVector **bv) function is a destructor for a bit vector

```
void bv_delete(BitVector **bv):
    Free the memory of the pointer to bv
    Set bv to NULL
```

- The uint32_t bv_length(BitVector *bv) function returns the length of a bit vector

```
uint32_t bv_length(BitVector *bv):
    Return a pointer to the length of bv defined in the Struct
```

- The bool bv_set_bit(BitVector *bv, uint32_t i) function sets the ith bit of a bit vector
- If i is out of range then return false, return true otherwise

```
bool bv_set_bit(BitVector *bv, uint32_t i):
    Set a variable to i/8 to get the ith bit in a byte
    If the variable is not within range of the bit vector:
        return false
    Use a bitwise OR of the ith bit in a bit vector with i % 8 to set the bit to 1
    return true
```

- The bool bv_clr_bit(BitVector *bv, uint32_t i) function clears the ith bit of a bit vector
- If i is out of range then return false, return true otherwise

```
bool bv_clr_bit(BitVector *bv, uint32_t i):
    Set a variable to i/8 to get the ith bit in a byte
    If the variable is not within range of the bit vector:
        return false
    Use a bitwise AND of the ith bit in a bit vector with i % 8 to set the bit to 0
    return true
```

- The bool bv_get_bit(BitVector *bv, uint32_t i) function returns the ith bit in the bit vector
- If i is out of range return false if the value of the bit is 0 and true if the bit is 1

```
bool bv_get_bit(BitVector *bv, uint32_t i):
    Set a variable to i/8 to get the ith bit in a byte
    Check the index variable of the bit vector
    Return true if the bit at bit vector index is 1
    Return false if the bit at bit vector index is 0
```

- The void bv_print(BitVector *bv) function prints out the bits of the bit vector

```
void bv_print(BitVector *bv):
    Go through all the bits in the bit vector using a for loop
    print the value of the bit at the end of each iteration
```

node.c:

- The Node *node_create(char *oldspeak, char *newspeak) function is a constructor for a node
- Allocate memory and copy over characters for both old and new speak
- Use strdup()

```
Node *node_create(char *oldspeak, char *newspeak):
    Dynamically allocate memory for the node
    Set the values of the oldspeak and newspeak pointer to oldspeak and newspeak argos
    Return the created node
```

- The void node_delete(Node **n) function is a destructor for a node
- Free the pointer to the node and also the memory in oldspeak and newspeak

```
void node_delete(Node **n):
    Free the memory of the pointer to n
    Free the memory of the pointers oldspeak and newspeak of n
    Set the pointer of n to NULL
```

- The void node_print(Node *n) function prints out the contents of the node

```
void node_print(Node *n):
    If the node n does not contain oldspeak and newspeak:
        Print the pointers oldspeak and newspeak of the node
    If the node n contains only oldspeak:
        Print the pointer of oldspeak of the node
```

bst.c:

- A binary search tree is either NULL or a node that points up to two subtrees
- The left subtree contains keys less than it in value and the right subtree contains keys greater than it in value
- The Node *bst_create(void) function is a constructor for a binary search tree

```
Node *bst_create(void):
    Create a node to present the binary search tree using the node create() function
```

- The void bst_delete(Node **root) function is a destructor for a binary search tree

```
void bst_delete(Node **root):
    Traverse the node of the tree at the left subtree
    Traverse the node of the tree at the right subtree
```

```
        Visit the root and free the memory of the node
        Set the node to NULL
```

- The uint32_t bst_height (Node *root) function returns the height of the tree rooted at root

```
uint32_t bst_height(Node *root):
        return the pointer to the height of the node
```

- The uint32_t bst_size(Node *root) function returns the size of the binary tree rooted at root

```
uint32_t bst_size(Node *root):
        Traverse starting from the root of the tree and return a count for the number of nodes in
the tree
```

- The Node *bst_find(Node *root, char *oldspeak) function searches for the node containing oldspeak and returns the pointer to the node found
- Else, return a NULL pointer

```
Node *bst_find(Node *root, char *oldspeak):
        Search for a node using a traversal of the tree at root
        If a node is found:
                Return a pointer to the node found
        Return a NULL pointer
```

- The Node *bst_insert(Node *root, char *oldspeak, char *newspeak) function inserts a new node containing specified oldspeak and newspeak into the binary tree

```
Node *bst_insert(Node *root, char *oldspeak, char *newspeak):
        Add a new node containing specified oldspeak and newspeak into the binary search tree
at root
```

Set the values of the pointers oldspeak and newspeak at the node to the oldspeak and newspeak args

- The function void bst_print(Node *root) prints out each node in the binary search tree using an inorder traversal

void bst_print(Node *root):

Use an inorder traversal of the tree rooted at root

Use node_print to print out each node of the tree

ht.c:

- A hash table is a data structure that maps keys to values
- The function HashTable *ht_create(uint32_t size) is a constructor for a hash table

HashTable *ht_create(uint32_t size):

Set the high and low of the salt indices

Dynamically allocate the memory of the hash table using size pointer

Dynamically allocate the memory of the array of nodes, trees

void ht_delete(HashTable **ht):

Free each of the binary search trees in ht

Free the hash table ht and set the pointer to the hash table to NULL

uint32_t ht_size(HashTable *ht):

Return the pointer to the size of the hash table ht

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

Search for a node in the hash table ht that contains the char oldspeak

Hash the oldspeak using the hash function provided in speck.h and use the return value as the index of where to look in the hash table

```
    Return the result of calling bst_find() with the index of trees in the ht and oldspeak as arguments
void ht_insert(HashTable *ht, char *oldspeak, char *newspeak):
    Hash the oldspeak similar to in ht_lookup and use that value as the index to insert in the node array, trees in the hash table
    Set the value of the node in the bst at the current tree to the result returned by bst_insert to insert and update the node
uint32_t ht_count(HashTable *ht):
    Make a variable count to track the number of non-NULL binary search trees
    Loop through the hash table until index is >= size of the hash table
    If the index of trees in the hash table does not equal NULL:
        Add 1 to count
    Return count
double ht_avg_bst_size(HashTable *ht):
    Make a variable size_sum to add all the sizes of the binary search trees in ht
    Loop through the hash table from 0 until the size of the hash table is reached
        For each iteration, call the bst_size function using the iteration count as the index for the trees in ht and add that value to size_sum
    Return the value of size_sum divided by the value returned from calling ht_count
double ht_avg_bst_height(HashTable *ht):
    Make a variable height_sum to add all the heights of the binary search trees in ht
    Loop through the hash table from 0 until the size of the hash table is reached
```

For each iteration, call the bst_height function using the iteration count as the index for trees in ht and add that value to height_sum

Return the value of height_sum divided by the value returned from calling ht_count

void ht_print(HashTable *ht):

Loop through the hash table from 0 to the size of ht

For each iteration, call bst_print on the array trees using the iteration count as the index

Credit:

- I watched Eugene’s section video on 11/23/21 for general guidance and explanation of the assignment
- I watched Eugene’s section video on 11/30/21 in which he explained functions in further detail and provided pseudocode which I used to implement my functions
- I went to Christan’s section on 12/3/21 in which he gave a brief overview of the functions with some pseudocode
- I used/referenced code from bv8.h in the code comments repository and code from lecture 18 and 28 slides provided by the professor to implement some functions in my program. Credit for the provided code is also commented in further detail in my program.