Michael Nguyen
mnguy181@ucsc.edu
2/18/2021

CSE13S Winter 2021
Assignment 6: The Great Firewall of Santa Cruz
Bloom Filters, Linked Lists and Hash Tables
Design Document

PRE-LAB QUESTIONS

Pre-lab Part 1:

1) Write down the pseudocode for inserting and deleting elements from a Bloom filter.

The description of the function void bf_insert() states that the function's purpose will be to take a string parameter oldspeak and insert it into the Bloom filter. In essence, I will be hashing oldspeak with each of the three salts for three indices, setting the corresponding bits in the underlying bit vector *filter. Thus, the pseudocode for inserting an element would be:

```
Void bf_insert(BloomFilter *bf, char *oldspeak):
        Hash oldspeak with each of the three salts (primary, secondary, tertiary) for three indices
        Store the three indices into temporary variables, prim_index, sec_index, tert_index
        Call setbit() on the underlying bit vector bf->filter to set corresponding bits of indices
```

I believe it would be very unconventional to delete elements from a Bloom filter. This is because there are cases where inserting an element into the Bloom filter can cause something known as a hash collision. A hash collision occurs when two elements in the Bloom filter share an index at a corresponding bit in the bit vector. To solve this issue where a hash collision has occurred when inserting, the assignment calls for the implementation for a hash table that contains an array of Linked Lists. In this case, we can store multiple nodes representing each oldspeak (and their translation if any) at the same index in the hash table. This is extremely helpful in checking if a string or word was truly considered oldspeak or improper. However, deleting an element from the Bloom filter (to my understanding) would generally call for clearing the corresponding bits in the Bloom filter for a certain element. If one of the bits had a case of collision, clearing a bit could potentially cause our program to lose another insertion that was made. For example, let's say that "Professor" (indices 1, 3, 5) and "TA" (indices 2, 4, 5) shared a corresponding bit at index 5. Clearing all of TA's corresponding bits in the bit vector would cause "Professor" to have one of its indices set to 0. If we wanted to search up in our Bloom Filter that "Professor" was oldspeak, the Bloom Filter would return to the program that it was not inserted yet (even when it was). Thus, deleting an element causes a false negative. In the general case that we wanted to delete an element (assuming that it was not a case of hash collision), then the pseudo code would be:

```
Void bf_delete(BloomFilter *bf, char *oldspeak):
        Hash oldspeak with each of the three salts (primary, secondary, tertiary) for three indices
        Store the three indices into temporary variables
        clrbit() at specified indices from temporary variables
```

Pre-lab Part 2:

1) Write down the pseudocode for each of the functions in the interface for the linked list ADT.
Note: Already provided struct definition in lab document. I put this here for better understanding in case I wanted to look back on the assignment in the future.

```
Struct LinkedList {
        U32 length
        Node *head // Head sentinel node
        Node *tail    // Tail sentinel node
        Bool mtf      // Move to Front
}
```

The following contains pseudocode for each of the functions for the LinkedList ADT. Please note that the following pseudocode was made with the help of TA Eugene during his lab section on 2/16/2021. Thus, credit goes to TA Eugene.

```
LinkedList *ll_create(bool mtf){
        Allocate memory for LinkedList *ll using malloc()
        Set ll->length to 0
        node_create() head and tail sentinel nodes, parameters as NULL
        Set ll->head->next to the tail node
        Set ll->tail->prev to head node to fully link
        Set ll->mtf to parameter mtf
        Return a pointer to the linked list
}

Void ll_delete(LinkedList **ll){
        For loop() going through each node, starting at head, ending at tail
                node_delete(Node i)
        Free the pointer to the Linked List *ll
        Set *ll to NULL
}

Uint32_t ll_length(LinkedList *ll){
        Return ll->length
}

Node *ll_lookup(LinkedList *ll, char *oldspeak){
        For loop() that goes through each node, n starting at head->next, ending when = tail
                Check if Node n->oldspeak is the same as parameter oldspeak
                        Check if the LinkedList bool mtf is set
                                n->prev->next will be  n->next
                                n->next->prev will be set to n->prev
                                n->next is head->next
                                n->prev is the head
```

```
                Head->next->prev is node n
                head->next is node n
            Return node n
    Return NULL

Void ll_insert(LinkedList *ll, char *oldspeak, char *newspeak){
    node_create() a Node that contains strings oldspeak and newspeak
    For loop() that iterates through the linked list’s nodes
        Check if the node’s oldspeak is the same as the parameter oldspeak
            Return (no insertion made since duplicate)
    Insert after head by setting Node’s next as original head->next
    Set the Node’s prev to the head
    Set the head’s next prev to the Node
    Set the head’s next to the Node
    Increment the length of the LinkedList
}

Void ll_print(LinkedList *ll){
    For loop that iterates through each node, starting at head->next, ending when = tail
        node_print() the node
}
```

Pre-lab Part 3:

1) Write down the regular expression you will use to match words with. It should match hyphenations and contractions as well.

The regular expression for this assignment will require a character set that contains characters from a-z, A-Z, 0-9, the underscore character as well as apostrophes and hyphens. Thus, even though I’m not entirely sure if this will be correct, but I think my regular expression would look something like:

#define WORD “[a-zA-Z0-9]*|([_’-])”

PURPOSE

The purpose of this programming assignment is to essentially read in a list of badspeak words and another list of oldspeak and newspeak pairs of words. With these words, we will in essence be creating a dictionary that parses and reads user input from stdin, comparing the user’s words to our dictionary. If the user uses badspeak words (words without a translation) in their input, they will be given a message stating that they will be sent off to joycamp. However, the user only uses oldspeak words (words with a translation to newspeak), then they will require counseling and will be given an encouraging goodspeak message. These messages will provide back the badspeak and oldspeak words that were read from stdin for the user.

To construct our “dictionary”, the program will utilize a hashtable and a Bloom filter. The Bloom Filter will contain an underlying bit vector ADT that sets bits for corresponding elements through the use of the hash() function provided by the SPECK cipher. Consider the Bloom Filter to serve essentially as an array of bits. The speck.c and speck.h files have been provided, and I will not be modifying these files. In addition, the hash table ADT will serve as an array of Linked

Lists that contain nodes. Each node will contain 2 strings: a word and its translation if it has any. When user input is parsed through and a word is read, the program will be checking if the word has been added to the Bloom filter first, and the Hash Table second. In essence, these two separate files serve as 2 checkpoints to ensure if a word was truly an oldspeak/badspeak word.

DESCRIPTION

Please do note that the following descriptions contain pseudocode for each part of the program. I will be starting from a low-level and end at the top level of the program. In this program, the order of levels from lowest to the highest level would be node ADT, linked list ADT, the hash table ADT, the bit vector ADT, the Bloom Filter ADT, and finally the banhammer.c file that contains the program's main() function. Files parser.c and speck.c have been provided as well as other header files.

NOTE: This is just a rough general outline of how I plan to approach the assignment (without having done any coding yet). The design document will be updated as I progress through the assignment when coding. *

Starting from the lowest level, a node will contain oldspeak and its newspeak translation if it exists. Nodes will be used to construct the program's linked lists, and because the program will implement a doubly linked list, each node will contain a pointer to the previous node and the next node in the linked list. Thus, the pseudocode for node.c which contains the implementation of the node ADT will be:

Struct definition already provided in the lab document, but included for reference.

```
Struct Node {
        Char *oldspeak
        Char *newspeak
        Node *next
        Node *prev
}

Node *node_create (char *oldspeak, char *newspeak)
        Allocate memory for a Node n using malloc()
        Allocate memory for oldspeak and newspeak strings
        Set the fields of Node n to parameters oldspeak and newspeak
        Set the pointers of the Node n to the next and prev nodes to NULL

Void node_delete(Node **n)
        Free oldspeak and newspeak strings
        Free the pointer to the node
        Set the pointer of the node n to NULL

Void node_print(Node *n)
        If the node's oldspeak and newspeak are not NULL
                print the node's oldspeak and newspeak (print statement supplied on lab doc)
```

Else

Print the node's oldspeak (print statement supplied on lab doc)

Going into the next level, the linked list will essentially serve as a list of connected nodes, initialized originally with two sentinel nodes, the head and the tail. Specifically, this ADT will be used in a hash table (the next level of the program) which contains an array of linked lists. The pseudocode for the Linked List ADT has already been written down in a previous section under Part 2 of the Prelab Questions.

A Hashtable will contain a salt that is passed to hash() whenever an oldspeak entry is inserted. For the Bloom Filter, hash collisions may occur, and the best solution for this will be the implementation of a Hashtable that serves as an array of linked lists, each list containing nodes that represent an oldspeak word and its translation. The mtf field will indicate whether or not the linked list should use the move-to-front technique. The move-to-front technique basically allows for commonly searched words to be at the front of a linked list, fastening the search time and time complexity of the program (in essence, O(1) lookup). The following pseudocode for the hashtable describes the different functions and includes the struct definition that was supplied in the lab document.

```
Struct HashTable{
    U64 salt[2]
    U32 size
    Bool mtf
    Linked List **lists
}
```

Function *ht_create(uin32_t size, bool mtf) has already been supplied in the lab document, and thus, I will not be including this in the pseudocode for the ADT's functions.

```
Void ht_delete(HashTable **ht)
    For loop() starting from index 0 to the size of the hashtable ht
        ll_delete() called on the array's index, freeing each node and the pointer
    Free the underlying array of linked lists ht->lists
    Free the pointer to the HashTable
    Set the Hashtable pointer to NULL

Uint32_t ht_size(HashTable *ht)
    Return the ht->size

Node *ht_lookup(HashTable *ht, char *oldspeak)
    hash() the parameter oldspeak which provides the index of the linked list
    ll_lookup() for the node that contains the oldspeak, given the index of linked list
    Store the returned node into a temp variable
    If the temp variable != NULL
        Return the node
    Else
```

```
        Return NULL

Void ht_insert(HashTable *ht, char *oldspeak, char *newspeak)
    hash() the parameter oldspeak which provides the index of the linked list
    If the linked list at the index is NULL
        ll_create() a new linked list
    ll_insert() the oldspeak and newspeak, given the index of the linked list

Void ht_print(HashTable *ht)
    For loop() starting from index 0, ending at size of hashtable, iterates through each index
        ll_print() the linked list at specified index
        Print a new line
```

Moving onto the next level of the program, we will need to implement a bit vector ADT which will be used for the program's Bloom Filter. In essence, a bit vector is represented by an array of bits, used to determine if something is true or false. The struct definition of the BitVector ADT is as follows:

```
Struct BitVector{
    U32 length
    U8 *vector
}
```

The following pseudocode provides insight into how I plan to approach each function for the bit vector ADT.

```
BitVector *bv_create(uint32_t length)
    Allocate memory for the BitVector using calloc()
    Check if calloc() returned a null
        Return NULL
    Set the BitVector's length to the parameter length
    For loop() that iterates until reaching the specified length
        Allocate memory for the BitVector's underlying array using calloc()
    Check if calloc() return a null
        Return NULL
    Return a pointer to the BitVector

Void bv_delete(BitVector **bv)
    For loop() that iterates through the array
        Free each index of the array
    Free the BitVector's vector
    Free the BitVector
    Set the pointer to the BitVector to NULL

Uint32_t bv_length(BitVector *bv)
```

Return the BitVector’s length

Void bv_set_bit(BitVector *bv, uint32_t i)
    Set a variable to the BitVector’s index i/8
    Mask is 1 left shifted by (i % 8)
    The BitVector’s index at i/8 is set to the variable OR mask

Void bv_clr_bit(BitVector *bv, uint32_t i)
    Set a variable to the BitVector’s index i/8
    Mask is NOT (1 left shifted by (i % 8))
    Set the specified index/8 for the BitVector to the variable AND mask

Uint8_t bv_get_bit(BitVector *bv, uint32_t i)
    Set a temp1 variable to the BitVector’s index i/8
    Mask is 1 left shifted by i % 8
    Set a temp2 variable to equal temp2 AND mask
    Return temp2 right shifted by i % 8

Void bv_print(BitVector *bv)
    For loop() that iterates through each index of the underlying array, stop when = length
        printf() the bit specified by bv_get_bit()
        Print a new line

In one of the higher levels of the program, the BloomFilter in essence serves as a test to see if an element is a member of a set. The Bloom Filter will contain 3 salts or keys. Utilizing the SPECK cipher already provided, in essence, each word will have 3 different hashes() and keys. The struct definition of the Bloom Filter is as follows:

Struct BloomFilter{
    U64 primary[2]
    U64 secondary[2]
    U64 tertiary[2]
    BitVector *filter
}

The following pseudocode contains how I would approach the BloomFilter ADT and its functions. Please do note that function *bf_create(uint32_t size) has been provided already.

Void bf_delete(BloomFilter **bf)
    Free the BloomFilter struct
    Set the pointer to the BloomFilter to NULL

Uint32_t bf_size(BloomFilter *bf)
    Set a temp variable to equal result of bv_length of the BloomFilter’s bitvector
    Return the temp variable

Void bf_insert(BloomFilter *bf, char *oldspeak)

    Set temp1 to equal hash() of primary and oldspeak
    Set temp2 to equal hash() of secondary and oldspeak
    Set temp3 to equal hash() of tertiary and oldspeak
    Set the corresponding bits in the underlying bitvector at the specified indices from temp variables

Bool bf_probe(BloomFilter *bf, char *oldspeak)

    Set temp1 to equal hash() of primary and oldspeak
    Set temp2 variable to equal hash() of secondary and oldspeak
    Set temp3 variable to equal hash() of tertiary and oldspeak
    Use get_bit() on underlying vector and specified indices
    Store the results into more temp variables
    If all of them are true
        Return true
    Return false

Void bf_print(BloomFilter *bf)

    For loop() that iterates through primary
        Print “Primary:\n”
        printf() at index
    For loop() that iterates through secondary
        Print “Secondary: \n”
        printf() at index
    For loop() that iterates through tertiary
        Print “Tertiary:\n”
        printf() at index
    Call bv_print() on the BloomFilter’s Bitvector

At the top level, banhammer.c showcases the program’s main() function and ties the entire program together, supporting several different command line options and reads from stdin for the user input as well as the .txt files for the oldspeak and newspeak pairs and badspeak words. The main() function will, in essence, test to see if our BloomFilter and Hashtable are working as intended, and print back a message describing whether or not the user should be sent to joycamp or be encouraged to use more newspeak words.

main()

    While loop() using getopt()
        Case h: set the hashtable to certain size (default 10000)
        Case f: set the size of the Bloom filter (default 2 ^20)
        Case m: enable mtf rule
    Initialize Bloom filter and hash table using bf_create() and ht_create()
    fopen() the badspeak.txt file
    While loop() that ends when fscanf reaches EOF
        fscanf() each badspeak word

```
        Insert into bloom filter and hashtable using ht_insert() and bf_insert()
fclose() the badspeak.txt file
fopen() the newspeak.txt file
While loop() that ends when fscanf reaches EOF
        fscanf() each pair of words
        Insert oldspeak words into Bloom Filter using bf_insert()
        Insert both oldspeak and newspeak into Hash table using ht_insert()
fclose() newspeak.txt file
Read user input with supplied parsing module
For loop() that iterates through each word
        Check if in bloom filter using bf_probe()
                If so, ht_lookup()
                If word does not have a newspeak translation
                        Thoughtcrime = true
                If word does have a newspeak translation
                        Rightspeak = true
        Else:
                Continue
If thoughtcrime and rightspeak
        Print corresponding message, print the errors, and vacation words
If thoughtcrime
        Print corresponding message, errors
If rightspeak
        Print corresponding message, print oldspeak words and newspeak translations
```