Cole Marquard
cdmarqua@ucsc.edu
11/29/2020

CSE13s Fall 2020

Assignment 6: Down the Rabbit Hole and Through the Looking Glass: Bloom Filters, Hashing, and the Red Queen's Decrees

Design Document

The queen of the kingdom is looking to stop the devolution of speak into unintelligible gibberish. The queen has hired us to make an implementation of a bloom filter and hash table in order to help the kingdom fight nonsense words. A Bloom filter is similar to a hash table, where the entries in the hash table are simply single bits. This project will use such a bloom filter, as well as a hash table with linked lists in order to both censor words that are not fit for the kingdom and translate words that are outdated and need to be improved to the new kingdom lexicon.

The inputs to our program are:

1. "-h x" as a command line argument will set the hash table size to x, default 10000
2. "-f x" as a command line argument will set the bloom filter size to x, default 2^{20}
3. "-m" as a command line argument will use the move to front rule where words that are searched, get moved to the front of the linked list for future searches
4. "-b" as a command line argument will not use the move to front rule described in 3
5. "-s" as a command line argument will suppress the letter from the censor and instead print statistics such as, seeks, avg seek length, avg linked list length, hash table load, and bloom filter load
6. User input should be used by typing "< inputfile" after ./hatterspeak ex: ./hatterspeak -f 1000 -h 1000 < input.in

Top Level:

The top level design of the code is given by the following pseudocode

Main:

Read Command line arguments

Load all words and their translations into a hash table

Load all nonsense words into the bloom filter

WordArr = User Input

//goes through each word of user input and checks the bloom filter and then the hashtable and then depending on if it passes the bloom filter or is in the hashtable, it could be an ok word and not be saved. A nonsense word without a translation and is save in an array, or a oldspeak word with a translation in which the word and its translation are saved in an array of nodes which store both the oldspeak words and their translations.

For Word in WordArr:

If BloomFilter_probe(word):

badwordsarr.appned(word)

Else if Hashtable_Lookup(word) != NULL

TranslationsNodeArr.append(word)

Else:

badwordsarr.appned(word) //was no translation so it's bad

If printStatistics:

printStatistics()

Else:

If len(**badwordsarr**) > 0 and len(**TranslationsNodeArr**)>0;

Print Both message

Else If len(**badwordsarr**) > 0:

Print nonsense Message

Else If len(**TranslationsNodeArr**) > 0:

Print Translation Message

Read Command line arguments explanation:

This is performed by iterating through argv either by looping or by utilizing getopt and then setting boolean values for things such as show statistics and move to front rule, for values such as the hash table size you have to get the number following the argument itself and save it to a variable.

Load all words and their translations into a hash table psudocode:

```
Ht = hash_table_create(hash_table_size)
File = open(hatterspeak.txt) //open in read mode
For x in range(len(File)):
    Words = getword(file) //gets the word/translation pair
    Hs = make hatterspeak struct
    Hs.oldspeak = words[0]
    Hs.hatterspeak = words[1]
    ht_insert(ht,Hs) //inserting Hs as a node in ht for each
                      word/translation explained later more
```

Load all nonsense words into the bloom filter psudocode:

```
Bloom = create_bloom_filter(bloom_filter_size) //initialize the bloom filter
                                                Obj
File2 = open(oldspeak.txt) //open in read mode
For x in range(len(File)):
    Word = getword(file) //gets a single word bc no translations
    bf_insert(bloom,word) //inserting word into bloom, explained later in
                           greater depth
```

Getword() explanation:

getword() is a function that utilizes regex in order to pick out wanted characters from text files and user input. It loops until it reads an end of file, space or period and then returns the characters it looped over which would make up a word to be used in our bloom filter or hashtable.

bf_insert(bloom,key) pseudocode:

```
//bloom is our bloom filter object and key is the word being inserted
```

//loops through all the salts which are like the seed for a random variable, has multiple salts so there are multiple different indexes for each word

For salt in bloom.salts:

Index = hash(salt,word) % bloom.filter.length //mod to keep it within the indexes available

bv_set_bit(bloom.filter,index) //seting the index of the bit vector to 1

//is repeated for each salt in our case we have 3

ht_insert(ht,Hs) pseudocode:

//ht is the hash table and Hs is the hatterspeak data struct/data with words and their translation

Index = hash(salt,Hs**.**oldspeak) % ht.length //mod to keep it within the indexes available

ll_insert(ht.heads,gs,index) //code and explanation is shown down below for **ll_insert**

bf_probe(bloom,key) pseudocode:

//bloom is our bloom filter object and key is the word being checked

//loops through all the salts which are like the seed for a random variable, has multiple salts so there are multiple different indexes for each word

For salt in bloom.salts:

Index = hash(salt,word) % bloom.filter.length //mod to keep it within the indexes available

if(bv_get_bit(bloom.filter,index) ==0)://getting the index bit of the bit vector

Return 0 //if any of the bit indexes are 0, then its not in the bloom filter yet

Return 1;

//is repeated for each salt in our case we have 3 and if it passes all of them returns 1

Hashtable_Lookup(ht, key, move_to_front_rule) pseudo code:

//ht is the hash table, key is the word being looked up, move to front rule is if its set to 1/true the node looked up will be moved to the head of the linked list so it can be found faster in future iterations**.**

Index = hash(ht->salt,key) % ht.length % ht.length //mod to keep it within the indexes available

Node = **ll_lookup**(ht->heads,key,index,move_to_front_rule) //further explanation of //ll_lookup is shown later in this document

Return node ///returns the node if found else it would return NULL

Prelab Part1

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

```
Bloom_insert(salts,key,bitvector):
    For salt in salts:
        Index =hash(key, salt) //for each salt given, get the index corresponding to the
        //salt and key combo given
        bv_set_bit(index,bitvector) //sets the bit at index of the bitvector
        //repeat for each salt with the same key
    Return;
```

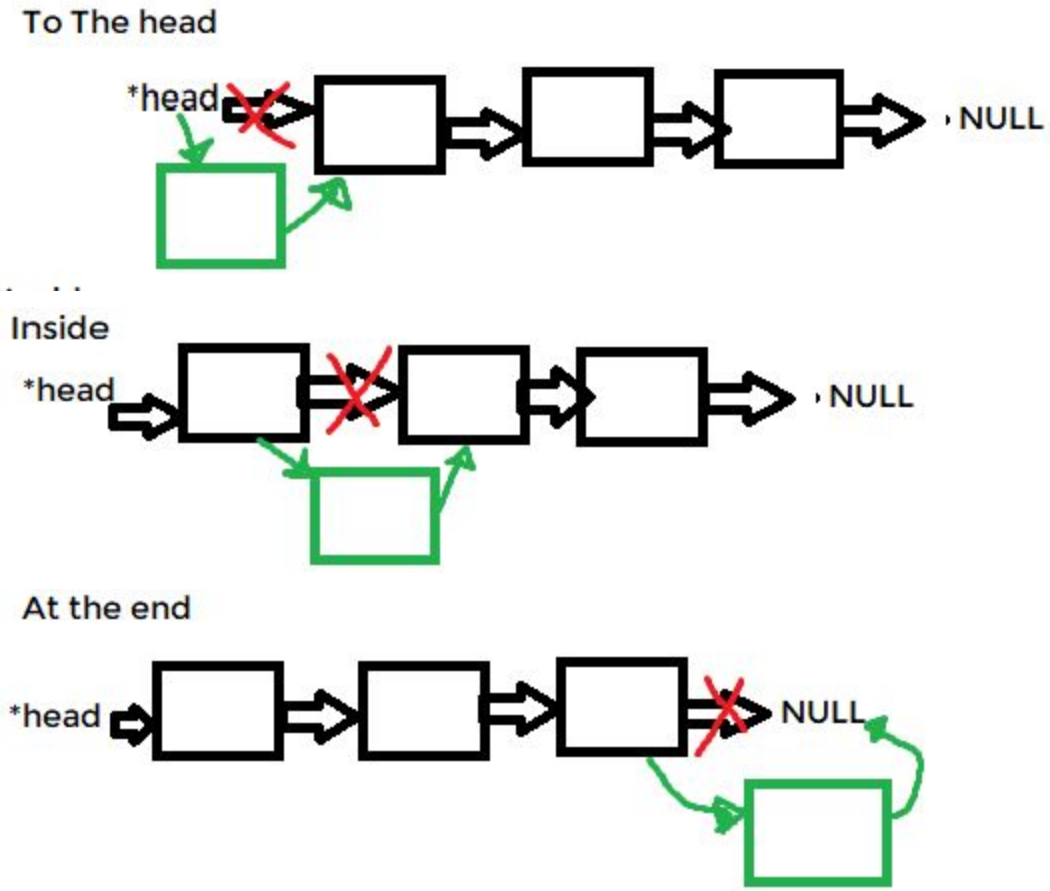
2. Assuming that you are creating a bloom filter with m bits and k hash functions, discuss its time and space complexity.

Its space complexity would be m as its the number of bits and the time complexity for inserts and probes would be k as that is the number of hash functions and inserts/lookups that would need to be performed

Pre-lab Part 2

1. Draw the pictures to show the how elements are being inserted in different ways in the Linked list.

In these picture the arrows represent the “next” pointer to the squares which represent the nodes of the linked list. The green is the node being inserted to the various linked lists in the examples and the crossed out arrows are the pointers which get changed to be the new node which then points to the following node



2. Write down the pseudocode for the above functions in the Linked List data type.

ll_node_create(datastruct hatterspeak): //data struct could basically be swapped for any data //struct you want to used to represent the data oldspeak and hatterspeak

Node = allocatemem(sizeofListnode) //not a thing in python but in c use malloc() to //allocate memory

Node.hatterspeak = hatterspeak //sets the data of the node to the inputted data //hatterspeak

Node.next = NULL //when you create a node, it will start out pointing to nothing

```
ll_node_delete(listnode n):
        free(n.gs) //sets the data portion free
        free(n)//sets the node itself free

ll_delete(listnode head):
        checking = head //used to traverse and delete nodes
        ToDelete = head //used as a temp so that we can transverse without getting lost

        while(ToDelete.next != NULL): //while next pointer is not null for ToDelete
                checking = checking.next //gets the next value ready/iterates to next value
                ll_node_delete(ToDelete)//frees the current node
                ToDelete = checking //iterates ToDelete to be the next node

        ll_node_delete(ToDelete)//after the loop we will be on the last node to delete so it is
        //deleted here

ll_insert(ListNode  head , HatterSpeak gs, int index):

        Node = ll_node_create(gs) //create a node with the give data (hatterspeak gs here)

        if(head[index] ==Null): //if there are no nodes currently in this space
                Head[index] = node //set the head to be the node as its the first node there
        Checking = head[index] //used to traverse the linked list

        while(checking.next!=null):
                checking = checking.next //traversing the linked list to the end node

        Checking.next = node //since we went till the current.next was NULL we can now set
        //current.next to the node we created to the end of this linked list

        Return head //just incase you want the head for some reason or another
```

```
ListNode ll_lookup(ListNode  head , char key, int index, bool move_to_front):
        //i use bool move_to_front as to avoid icky, loser, unneeded global variables
        checking = head[index] //again used to traverse the linked list
        Previous = NULL // NULL for now but will be used to swap if move_to_front is set

        If(checking ==NULL):
                Return NULL // if checking is null then there are no nodes here at all

        if(checking.gs.oldspeak == key):
                Return checking; //was found at the head of the linked list

        while(checking->next != NULL): //used to traverse the linked list
                Previous = checking //saving the previous node before iterating
                Checking = checking.next //used to iterate to the next node in the L list
                if(checking.gs.oldspeak == key)://If the current node has the correct key
                        if(move_to_front); //if searched words are moved to the front of LL
                                Previous.next = checking.next //last node now points to the
                                //node after the current one as to not make a gap

                                Checking.next = head[index] //sets the next node to be
                                //what the head currently is

                                Head[index] = checking //the head is now the one that got
                                //searched aka at the front of the LL/Linked list

                                Return checking /returns the node that matches the key

        if(checking.gs.oldspeak == key): //finished while loop, therefor is on the last node
        //and is checking to see if it has the same oldspeak word as the key

                if(move_to_front); //if searched words are moved to the front of LL
                                //find more detailed explanation above in while loop
                                Previous.next = checking.next
                                Checking.next = head[index]
                                Head[index] = checking

                                Return checking /returns the node that matches the key

        Else: //if the oldspeak word at the last index is not the same as the key
                Return NULL
```

Design Process

Over the course of this lab i modified my design multiple times

At first I didnt know how to use regex for real and learned through trial and error how it can be used and it is a very useful tool for text and finding specific characters

At first I was getting alot of segmentation faults while traversing the linked lists but I learned how to check when to stop traversing and avoid going to null pointers in general as they are very annoying and hard to find going back and looking at code

Before starting this project I had no clue what a hash table was and how I could implement it, for example some of my problems were seg faults with inserting into the bloom filter and hash table because I was not using modulus to keep the index within the range of either the bloom filter or the hash table, like some values I got were negative which need to be modded to get it in the positive values.

Overall this project was slightly challenging but I overall enjoyed the challenges of implementing its features and learning a lot about hash tables and bloom filters. If I could change anything about this lab it would be printing and getting the statistics as it really just felt like busy work as I would expect the almost everyone who can code a hash table could find the avg linked list len and from my perspective the statistics hold no real value and just take unneeded time to implement a feature that teaches us nothing as for things such as search len with move_to_front doesn't change things a whole lot unless there are ALOT of repeated words which is not very common outside of words like a and i .