Design Document
Despina Patronas
Assignment 6

Bloom Filter implementation:
ADT which, upon creation will generate 128 bit 3 salts (2 64 bit each) for hashing keys into bloomfilter
Largely depends on foundational Bit vector ADT to run successfully

Bloomfilter Defined Fields:
bf.filter = array of bitvector
3 salts (divided into 64 bit values held at index of array)

> **NOTE: Parsing and Speck files implementation will not be covered in this design PDF, it is simply used within the implementations specified below, it is beyond the scope of this course..**

//bloomfilter will use the length in order hash correctly restrict to the length of the bf (with MOD)

//return a newly created bloomfilter with fields defined

Bloom_filter_create(length)

Malloc memory for bloomfilter struct

Define the salts index each 64 bit ex:

Bf.firstsalt [0] = 0xfc28ca6885711cf7;
Bf.secondsalt [1] = 0x2841af568222f773;

Define bloomfilter field filter as a bitvector

//check if bloomfilter was allocated correctly

//return the bloomfilter after creation

End bloom filter create

Destructor to keep the memory clean once the program is done call this to remove BF ADT

Bloom filter destructor (bf)

Free filter field
free the alloc memory for bloomfilter

End destructor

To insert we need to use the hash function provided by speck
Hash same key with 3 different salts in order to reduce false positives

Bloomfilter_insert (bf, key)

Define bf_length as bitvectorlength(bf.filter)

Index1 = hash(bf.firstsalt, key) MOD bf_length

//repeat for 2 other salts

Bvsetbit (bf.filter, index1)

//repeat setting the bit for 2 other index

increment extern variable setbit *//keep track of the # bloomfilter set for stats*

end bloomfilter_insert

to check if a key exists, we must query the bloomfilter
returns a TRUE or FALSE watch out for false positives as bloomfilters are not perfect

bloomfilter_probe (bf, key)

Define bf_length as bitvectorlength(bf.filter)

Index1 = hash (bf.firstsalt, key) MOD bf_length

//repeat for 2 other indexes w/ respective salts
//to ensure the bloomfilters are set check the bit for each index (true or false)

Boolean first = bitvector_getbit (bf.filter, index1)

//repeat for 2 other indexes

//for bloomfilter if all the encrypted hashes have been set, PROBABLy was inserted
//there MAY still be false positives occurring depends on the hash and salt used to
//make key unique. Keep this in mind when consulting the bloomfilter for keys

If (first && second && third)

Return true

//this will be a guaranteed statement (unlike the true)

Otherwise

Return false

End bloomfilter_probe

End Bloomfilter

Bitvector ADT (pulled from my assignment implementation on gitlab)

<u>Pre-Lab Part 2</u>

1)

```
//Bit Vector Function Implementation

//Ceiling function for creating vector field
Ceiling(real n)
        //if truncated n is less than the decimal number n
        If (casted int type(n) < real n)
                //return the ceiling which is truncated n +1
                Return casted int type(n) +1
        //otherwise the two values are equal
        Otherwise
                //return the truncated n (floor of n)
                Return casted int type n
        Endif
End ceiling

//create the bitvector and initialize the fields
BitVector *create( pos int input_length)

        allocate new_ vec of type Bitvector on heap (size of 32 positive int bits)

        //check the creation is successful
        If (!new_vec)
                Return NULL
        Endif

        Set new_vec field length = input_length

        allocate new_ vec  field vector of type (positive int 8 bit) on heap (with size of 8 positive int bits* Ceiling(
                                                                                                   new_vector length /8 )
        //check the creation is successful
        if (!new_vec)
                return NULL
        endif
        return new_vec

end BitVector create
```

//delete the BitVector that is on the heap
Delete(BitVector *new_vec)

//delete allocated memory of the array in new_vec
Free(new_vec vector)

//delete allocated memory of the Bitvector new_vec
Free(new_vec)

End delete

//return the length in bits of the bitvector
Positive int getLength (BitVector *new_vec)

Return new_vec field length

End getLength

//set the bit at index in BitVector
Setbit(BitVector *new_vec, positive in index)

//accessing the byte element of the vector field
Declare Int Byte = index / 8

//access the bit element of vector field
Declare int bit = index MOD 8

// vector[byte] OR with 1 shifted left by the bit amount
Set new_vec vector[Byte] = new_vec vector[Byte] OR (1 << bit)

End Setbit

```
example:
     1010
  OR 0100  <- 0001 << 3 (index bit)
__________
     1110    index bit is set to 1
```

//Clears the bit at index in the BitVector from 0->1
ClearBit(BitVector *new_vec, positive in index)

//accessing the byte element of the vector field
Declare Int Byte = index / 8

//access the bit element of vector field
Declare int bit = index MOD 8

//& with inverse of (1 << bit)
Set new_vec vector [Byte] = new_vec vector[Byte] AND (NOT(1 << bit))

End ClearBit

```
example:
      1100
  AND 1011  <- the inverse of ( 0001) << 3 (index)
__________
    = 1000  the index bit is cleared
```

//Gets a bit from a BitVector.
Positive int GetBit(BitVector *new_vec, positive in index)

```
    //accessing the byte element of the vector field
    Declare Int Byte = index / 8

    //access the bit element of vector field
    Declare int bit = index MOD 8

    //vector[byte] >> (bit & 1)
    Return new_vec vector [Byte] >> (bit AND 1 )
```

```
example:
        0001 <- 0100 >> 3 (the index bit)
   AND  0001
   __________
       = 0001 (this would be 0 if we started with 1000)
```

End GetBit

//Sets all bits in a BitVector to 1
SetAll(BitVector *new_vec, positive in index)

```
    //accessing the byte element of the vector field
    Declare Int Byte = index / 8

    //access the bit element of vector field
    Declare int bit = index MOD 8

    //set all bits to one in the BitVector
    Iterate through index [0, new_vec length]
        SetBit(new_vec, index)
```

End SetAll

END BitVector

Linked List ADT implementation:

This is a singly Linked list which inserts in the front. Simple implementation
Linked list is working directly with the word structures to create valid linked list data
The Linked list will be the foundation of the HashTable (next ADT)

linked list Defined Fields:
data struct (hatterspeak data held within the linked list)
next (ptr to next node in list)

Ll_create (ptr struct HS)

Allocate memory for newnode //make the node in heap

//check if allocation was sucessful

Newnode.data= HS //set data field to hatterspeak structure

Newnode.next = null //set ptr field to null (End of List)

Return newnode

End ll_create

//free the fields within nodes and then the node itself

Ll_NODE_delete (ptr nodehead)

Free(node.HS.oldspeak)

//If there is a hatter word in struct

Free(node.HS.hatter)

Free(node)

End ll_NODE_delete

*//**free the entire linked list starting at the head***
//iterate through the whole linked list and call ll_NODE_delete

Ll_list_delete (ptr nodehead) //while head is a valid ptr address aka !NULL

Ptr currhead = head //as to not tamper with parameter

Ptr next = NULL //start by initializing this data

While (currhead) //loop through entire linked list

Next = currhead.next //pick up the next item in linked list

Ll_NODE_delete(currhead) // delete the current node which Is head

Currhead= next //head is the next item in linked list

End while

Head = NULL //ptr passed through is NULL

End ll_list_delete

Creates and Insert a node into an existing linked list at the ptr to ptr of the linked lists head Data to insert is the Hatterspeak word structure

Ll_insert (ptr ptr of head, ptr HS struct)

//check duplicates very important for hashtable to not be inserting multiple keys
//look into the existing linked list and see if the key were trying to insert already exists
//the key is oldspeak

if (ll_lookup (head, HS->oldspeak) == TRUE)

//handle it! Can either get rid of the old and replace it with the new return the head
//or u can ignore the new key altogether and just return the head
//if u chose to get rid of the old node u must free all the struct data passed in as parameter to function as it will be unused and cause memory leakage

End if //end checking dupes

//Otherwise make a new node and insert to the front of linked list

Allocate memory for newnode

Newnode.data = HS

Newnode.next = head //old head (old node head)

Head = newnode //new head

Return the newnode

End ll_insert

The goal here is to find the oldspeak word which is keyed into the lookup. Search begins are the head of the linked list passed through function

```
Ll_lookup ( head, key)

    Ptr Temp = head
    ptr res = NULL          //field to return the node
    prev = NULL             //field to handle movetofront

    Char stored = NULL      //field to hold the oldspeak word we will be accessing in node

    //traverse the linked list to see if key data matches node data

    While ( temp != NULL )

        Increment the links     //this is for the stats later on. Traversing each node counts +1

        Set stored = temp.HS.oldspeak   //pull the data from the current head

        if ( stored == key )            //string comparison

            //at this point we found a listnode and normally would just return that node
            //but since we have a move_to_front option we must, move that node
            //to the front of  the linked list and rearrange the chain SO CHECK OPTION

            if ( move_to_front == TRUE )    //affects what we do when we find the listnode

                prev.next = temp->next  //previous node is set to current head next

                temp.next = head        //the node we found next ptr is current head

                head = temp             //temp is the new head

            end if
            otherwise

                res = temp              //set the result to the node found
                temp = temp.next        //increment the linked list

            end otherwise

        end if //key was NOT Found keep looking
        temp = temp.next                //increment the linked list

    end while

    return res

end ll_lookup
```

At this point you may want print nodes and print entire linked lists create two helper functions to do so
Remember the struct may or may not hold hatterspeak data due to the txt files nature
Oldspeak has no counterpart ->(NULL)
Hatterspeak -> translation

Temp parameter is the current head passed into function (as to not tamper with the ptr parameter)

Ll_NODE_print (head)

Temp = head

Print (temp.HS.oldspeak)

If (temp.HS.hatterspeak)

Print that too

End ll_NODE_print

//iterate through entire linked list printing the nodes

Ll_print_list (head)

Temp = head

While (temp == TRUE)

Ll_NODE_print(temp)

Temp = temp.next

End ll_print_list

End Linked List Implementation

HS structure implementation

HS is the structure which holds the data of words from text files and determines whether they are oldspeak or 'hatterspeak'

HS Defined Fields:
char oldspeak
char hatterspeak

HS hs_create (os, hs)

Allocate hs memory on heap

Check allocation

Set hs.oldspeak = allocated memory of parameter os

Check allocation

If (hs parameter != NULL)

Set hs.hatterspeak = allocated memory of parameter hs

Check allocation

Otherwise set hs.hatterspeak = NULL

End if else

Return structure hs

End hs_create

Like all things defined memory on the heap, we must have a destructor

Hs_delete (hs)

Free(hs.oldspeak)

If (hs.hatterspeak != NULL) *// Should not try to delete a null variable*

Free(hs.hatterspeak)

Free(hs struct)

End hs_delete

HashTable ADT implementation
the hashtable is a linked list of listnode heads. In order to perform correctly linked list will need to be functioning as intended

The hashtable takes the concept of unique encryption to salt and hash ptr of heads into an array at hashed index

Hashtable defined fields:

(1) Salt sized 128 bits
Htlength
Array of linked list defined heads

//hashtable will use the length in order hash correctly restrict to the length of the ht (with MOD)
//returns a newly created hastable with fields defined

Ht_create (length)

Allocate ht memory on the heap

Check allocation

Define ht.salt[0] and ht.salt[1]

Ht.length = length

Calloc memory for ht.heads array

Check allocation

Return the ht

End ht_create

Ht_delete (ht)   ***//keep the heap clean of extraneous allocated memory***

Iterate from 0 to length
Check if ht.heads is != NULL

Free ht.heads
Free ht.heads   *//the ptr*
FREE ht   *// the initial contruction*

End ht_delete

Since the hashtable is storing data hashed into an array, we can retrieve that data similar to how bf retrieves bits in its bit vector array

Either return null or if data is found, node at which data was found
Returns in the form of a listnode (node which holds the data aka key)

Ht_lookup (ht, key)

//whenever we dip into a look up we increment seeks for statistics

Seeks++ *//hash index to retrieve head location*

Index = hash (ht.salt, jey) MOD ht.length *//since length is a field direct call*

If (ll_lookup (addr ht.heads [index], key) == TRUE) *//lookup using that head from index*

Ret = the return of ll_lookup

Otherwise return NULL

End ht_lookup

Ht_insert (ht, gs)

//pick up the index from the hash
//only interested in oldspeak words

Index = (hash (ht.salt, gs.oldspeak) MOD ht.length)

//now that we have the index for the location of head array,
// call ll insert to give the node and hs information

Ht.heads[index] = ll_insert (addr ht.heads[index], gs)

End ht_insert

//keep track of the amount of valid heads in array of heads at given index
//aka how populated is the hash table??

Ht_count (ht)

Initialize the count

Iterate index through 0 to ht->length

If we find a head[index] != NULL

Increment the heads count

End loop

Return count

End ht_count

End HashTable implementation

Main function implementation

Define regex to validate words parsing through stdin

Define options for getopt

Define externs or global variables for statistics in their respective header files or in main

//function to normalize all forms of file / input streams into lower case
// return nothing array is pass by reference

To_low (str[])

Char letter //hold the index of string temporarily to convert to lower case

Iterate from index 0 to string length (str)

Letter = str[index] //grab that index of str array

Str[index] = tolower(letter) //convert to lower from temp

End iteration

End to_lower

Reading form file input oldspeak.txt in this case
the goal here is to open file, scan words until end of file in the fashion they appear
convert to lower case
insert into bloomfilter (oldspeak words only)
create a HS structure to hold the words parsed from input
insert into a hashtable the struct HS

Pseudo:

Fopen ("oldspeak.txt", stdin)

//Check if opened ok

Char oldspeak [100] //place to store words into

While (fscanf ("oldspeak.txt", "s\n", oldspeak) != EOF) //terminate with NULL character

To_low(oldspeak)

Bf_insert(bf, oldspeak)

Hs = HS_create (oldspeak, NULL)// null because oldspeak has no hatterspeak translation

Ht_insert (ht, Hs)   //populate hash table with the new hs

//down the rabbit hole of checking dupes with ll_lookup, calling ll_insert, hashing on both hashtable and bitvector to set the bit for new word presence)

//it is critical at this point that each ADT is functioning not only independently but with each other

//also make sure there is a hashtable and bloomfilter (1) of each declared before this point

End while

Close ("oldspeak.txt")

/* Reading form file input hatterspeak.txt very similar except you will be looking for two strings next to each other per line of stdin infile stream.
All of the steps remain the same except in HS struct creation, you will be populating the hatterspeak instead of leaving it as NULL */

Parsing stdin data after text files have been loaded

*/*using a regex defined in global scope to filter input, a while loop will use parser implementation to feed words in word by word*
convert words to lower case
if it exists in bloom filter

if it does look it up from ht_lookup
set a node to the return of ht_lookup

check if theres a hatterspeak attached to the node
if there is create a struct HS for that nodes fields (both)
insert that node into a dedicated linked list for translatable user input

if hatterspeak is NULL for that found node

create struct HS for the found nodes fields (both)
insert that node into a 'forbidden' linked list of user input words
end if
end if
end if
end while

//close stdin, regfree(regex), clear_words() to avoid extraneous memory leakage from buffer / heap

//get opt will support printing of stats, printing of letter, move to front set/disable, adjusting hashtable size, bloomfilter size

Getopt Support:

Psenglish:

Within a while loop main argc, and argv, and defined OPTIONS

Using a switch case for OPTIONS

Case s: set bool stats = 1

Case m: set bool move_to_front = 1

Case h:

Get_Ht_size = optarg
//check bounds > 0
set htsize = atoi(get_ht_size)

//continues similarly for set / toggle options

End getopt support

//Move_to_front and NotMTF are mutually exclusive cannot select both

//create bloomfilter with default size if user has not specified in getopt
//same creating ht

The final part of this program is displays which are handled in order of getopt

Ex: *if (stats) Display stats*

Else display letter message

Seeks: # of Lookups

Avg seek length: $\dfrac{\text{number of nodes trave}}{\text{seeks}}$

Avg LL length: $\dfrac{\text{sum(LL length)}}{\text{length(ht)}}$

hash Load: $\dfrac{\text{Non Null Heads}}{\text{length(ht)}}$

Bloom load: $\dfrac{\text{\# of 1's}}{\text{length(bf)}}$

//data for stats provided by Professor Dunne

Finally must clean the heap of ALL allocated memory from:

- ***regex***
- ***Hashtable heads, hashtable creations***
- ***struct+ structs data within linked lists***
- ***Linked lists creations***
- ***bitvector creation, bitvector array***
- ***bloom filter creation, bloomfilter array of bit vector (filter)***

Note* credits to TA Oly and Eugene lab / office sessions on asgn 6

Oly: who provided insight on hash.c pseudo implementation/ visualization diagram.

Eugene: who provided a macro for regex and steps to parse user input with regex and stdin specifically

Eugene offered macro for stndup -> strcpy for hatter struct and also a rought pseudo for the hatter struct implementation.

Thank you Tas for all your support and guidance