Brady Chan
brmchan@ucsc.edu
11/29/2020

CSE 13s Fall2020
Down the Rabbit Hole and Through the Looking Glass:
Bloom Filters, Hashing and the Red Queen's Decrees
Design Document

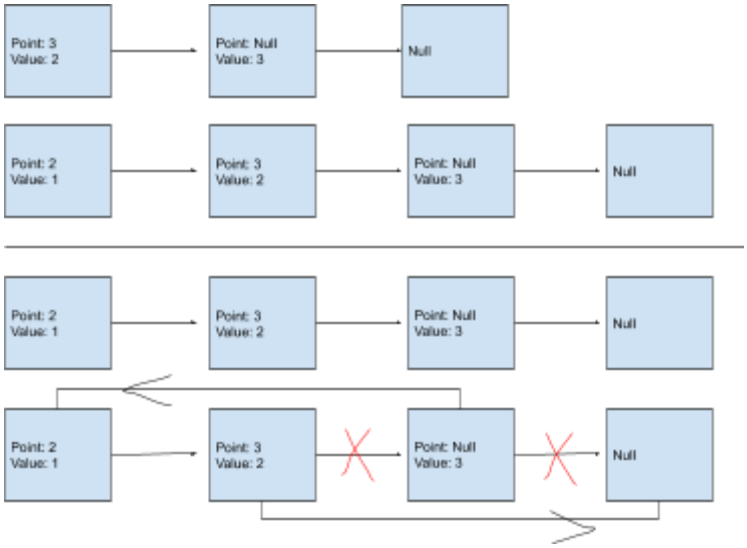
Pre-Lab Part 1:
1)

Void bf_insert(BloomFIlter *b, char *key):
Index = hash(b->primary, key) % length
bv_set_bit(b->filter, index)
Index = hash(b->secondary) % length
bv_set_bit(b->filter, index)
Index = hash(b->thirdary) % length
bv_set_bit(b->filter, index)

Void bf_element_delete(BloomFilter *b, char *key)
Index = hash (primary, key) % length
bv_set_bit(b->filter, index)
Index = hash(secondary, key) % length
bv_set_bit(b->filter, index)
Index = hash(thirdiary, key) % length
bv_set_bit(b->filter, index)

2) The memory and time complexity of the bloom filter will be m bits and O(k). This is because the size of the bloom filter is only equal to amount of bits used to store the word. And the time it takes is the amount of time the hash functions run which is k.

Pre-Lab Part 2
1)



2)

```
ListNode *ll_node_create(HatterSpeak *gs) {
        Malloc Listnode
        node->gs = gs
        node->next = NULL
}

Void ll_node_delete(ListNode *n) {
        hs_delete(n->gs)
        free(n)
}

Void ll_delete(ListNode *head) {
        If head -> next :
                ll_node_delete(head)
        Else
                Void ll_delete(head->next)
                ll_node_delete(head)
}
```

```
List Node *ll_insert(ListNode **head, HatterSpeak *gs) {
        If ll_lookup(head, gs):
                Delete gs
                Return head
        Else
                ListNode *newnode = ll_node_create(gs)
                newnode->next = *head
                Return newnode
}

ListNode *ll_lookup(ListNode **head, clear *key) {
        ListNode *prev = NULL
        ListNode *current = head
        While current:
                If current->gs->hs = key and move to front:
                        prev->next = current->next
                        current->next = head
                        Return head
                If current->gs->hs = key:
                        Return current
                Prev = current
                Current = current->next
        Return null
}
```

In this i call hs_delete to free the hatter speak allocated data.and used recursion in ll_delete to run through to the end of the list. I use recursion because it makes the most sense to run through a list where each node points to another. Ll_insert uses lookup to check if the node already exists, if it does then it returns the head of the list, if it doesn't find it, it creates and sets a new node and makes it the new head of the list. Lastly, lookup runs through the list by saving the previous node. If it does find the corresponding key, it either returns it or moves it to the front before returning it. If it doesn't find any at all it returns null.

```
Main:
        Int c;
        Uint32_t def_h_size = 10000
        Uint32_t def_b_filter = 2 ^ 20
        Bool hash, filter, move set to false
        Bool used = false;
        While getopt(argc, argv, "hfmb") {
                Switch
```

```
        Case ‘h’:
                Hash = true
        Case ‘f’:
                Filter atoi argv
        Case ‘m’:
                If used:
                        Return 0
                Used = true
                Move = true
        Case ‘b’:
                If used:
                        Return 0
                used = true
                Move = false
}
Create bloomfilter
Create hashtable
File fp = open(oldspeak)
While fscanf(fp, %s, eachword):
        bf_insert(eachword)
        Old = str_create(eachword)
        Hatterspeak *hs = eachword
        ht_insert(ht, hs);
Close fp
Fp = open(hatterspeak)
While fscanf(fp, %s, eachword):
        bf_insert(eachword)
        Old = str_create(eachword)
        Hatterspeak *hs = eachword
        fscanf(fp, %s, eachword)
        Hatter = str_create(eachword)
        hs->hatterspeak = hatter
        ht_insert(ht, hs);
ListNode *reallybad = NULL
ListNode *kindabad = NULL
Regex regex
if regcomp regex:
        Printf error
        Return 0
While myword = nextword(stind, regex):
```

```
                If bf_probe(bf, myword):
                        ListNode *node = ht_lookup(ht, myword)
                        If node:
                                If node->gs->hatterspeak == NULL:
                                        Hatterspeak *hs =
hs_create(str_create(node->gs->oldspeak)
                                        ll_insert(reallybad, hs);
                                Else
                                        Hatterspeak *hs =
hs_create(str_create(node->gs->oldspeak);
                                        hs->hatterspeak = str_create(node->gs->hatterspeak)
                                        ll_insert(kindabad, node)
ListNode *rhead = NULL
ListNode *khead = NULL
If stats:
        Printf tarv/seeks
        Printf seeks/h_size
        Printf ht_count/h_size * 100
        Printf count/b_size * 100
Else
        If kindabad && reallybad:
                Print letter
        Rhead = reallybad
        Khead = kindabad
        While reallybad:
                Print
                Reallybad = reallybad->next
        While kindabad
                Print
                Kindabad = kindabad->next
        Else if kindabad && !reallybad:
                Print
                Khead = kindabad->next
                While kindabad:
                        Print
                        Kindabad = kindabad->next
        Else:
                Print
                Rhead = reallybad
                While reallybad:
```

```
                Print
                Reallybad = reallybad->next
Reallybad = rhead
Kindabad = khead
Bf_delete
Ht_delete
If kindabad:
        LL_delete(kindabad)
If reallybad:
        LL_delete(reallybad)
regfree(&regex)
Clear_word
```

In the main file I first start with the getopt to take in all user command line inputs. Then I create the bloom filter object and hashtable to be able store the words that will be parsed from the files. Next I create a file object and take in the words from the text files and hash them and store them into the bloom filter and the hash table. After closing the file I did the same with the hatterspeak file however I must scanf again inside the while to take the second word. I also create a hatterspeak object inside and use str, a str to allocate space for string. After parsing both files I create a regex and compile it to search for input from the user to be compared to. I then use a while loop to continuously take in input from the user and check it against the bloom filter and hash table I created. If it is in there, I create a new node and store it inside a reallybad and kindabad list. Lastly, I either print the stats or the letters depending on the user input. I then free everything to avoid memory leaks.

Hash:

```
HashTable *ht_create(uint32_t length) {
        HashTable *ht = (HashTable *)malloc(sizeof(HashTable));
        if (ht) {
                ht->salt[0] = 0x86ae998311115ae;
                ht->salt[1] = 0xb6fac2ae33a40089;
                ht->length = length;
                ht->heads = (ListNode **)calloc(length, sizeof(ListNode *));
                Return ht;
        }
        Return (HashTable *)NIL;
}
Void ht_delete(HashTable *ht) {
        For o to ht->length:
```

```
                ll_delete(ht->heads[i])
        free(ht->heads)
        fre(ht)
        return
}
Uint32_t ht_count(HashTable *h) {
        Count = 0
        For i = 0 i < h->length:
                If h->haeds[i]:
                        Count++
        Return count
}
ListNode *ht_lookup(HashTable *ht, char *key) {
        Index = hash(ht->salt, key) % ht->length;
        Return ll_lookup(&ht->heads[index],key)
}
Void ht_insert(HashTable *ht, HatterSpeak *gs) {
        Index = hash(ht->salt, key) % ht->length
        ll_insert(&->heads[index],. gx)
}
```

With the hash table, the create is given. For ht_delete i run through all the indexes and completely delete the linked list using the ll_delete function. The count runs through each index of the hash table checking for head. Ht_lookup uses the hash function to find the correct index of the hash that it is saved at and uses the ll_lookup to pass it the correct linked list and search if there is a corresponding key. Lastly, ht_insert hash the key and adds a new linked list node to the existing linked list.

```
        BloomFilter *bf_create(uint32_t size):
                BloomFilter *bf malloc
                BitVector *vector = bv_create(size)
                bf->filter = vector;
                Return bf
        Void bf_delete(BloomFilter *b):
                bv_delete(bf->filter)
                free(bf)
                return
```

```
Void bf_insert(BloomFIlter *b, char *key):
        Index = hash(b->primary, key) % length
        bv_set_bit(b->filter, index)
        Index = hash(b->secondary) % length
        bv_set_bit(b->filter, index)
        Index = hash(b->thirdary) % length
        bv_set_bit(b->filter, index)

Bool bf_probe(BloomFilter *b, char * key):
        Index = hash(b->primary, key) % length
        Bool first = bv_get_bit(b->filter, index)
        Index = hash(b->secondary) % length
        Bool second = bv_get_bit(b->filter, index)
        Index = hash(b->thirdary) % length
        Bool third = bv_get_bit(b->filter, index)
        Return first & second & third
```

To create the bloom filter I malloc the size and create the bit vector to store the bits. In this I set or clear bit in the bit vector by hashing the key or word given and mod by length of the bloom filter so its within the index. Then after I get the index I set or get bit. Probe and insert use the same logic because the way they access the correct index is the same way, through hash. Lastly, bf_delete frees the bit vector used and the malloc.

str.c:

```
Char *str_create(char *word):
        Char malloc srtlen(word) + 1
        memcpu(out, word, strlen(word) + 1)
        Return out
Void str_delete:
        free(in)
        Return
```

In this helper function I allocate memory for the string to be used and referenced in the hatterspeak struct. After allocating the correct amount of memory, I copy over the letter and return the string. After its use I use str_delete to free up the memory.

Hs.c:

```
        HatterSpeak *hs_create(char *oldspeak):
                HatterSpeak *word = malloc size of HatterSpeak
                word->oldspeak = oldspeak
                word->hatterspeak = NULL
```

```
        Return word
Void hs_delete(Hatterspeak *hs):
        str_delete(hs->oldspeak)
        str_delete(hs->hatterspeak)
        Free hs
```

In this helper file I create an str to allocate memory and save the string over to it. This is very helpful so I can allocate memory for each word in the text files and don't keep overwriting.

Parser:
Uses a compiled regex and stdin to take in words from the user. The word_next continues to take in words from the user that match with the regex, and clear_words clear the buffer used to help keep track and free memory to avoid a memory leak.

Speck:
Speck hashes the word into a usable index for the hash table to save the words stored in the linked list. Hash is also used to index the words in the bloom filter. By passing it the salt and word it returns a value for the word to be stored at.

Development Process:

- I had an infuriating valgrind error that kept thinking there was a conditional jump at an if and I couldn't figure out why. However, after messing around with the bv.c I had from assignment 4 I figured out that since i used malloc instead of calloc when creating the bit vector it caused that weird conditional jump error in my hatterspeak file.
- Another difficult problem I encountered was trying to allocate string memory and constantly overwriting my saved char pointers. This causes the problem of all the linked list editing the same pointer. I overcame this by creating a new helper function to allocate and save the string into it.

All in all, this project had a steep understanding of how to use each separate data structure, however once you understand how each works and their purpose the only challenge was piecing it all together. I have a much better understanding of how each individual data structure works as well as a better understanding of the concept of object oriented programming. This is because each data structure required references to numerous other data structures and their functions. Without this object oriented design, it would have been impossible to correctly free up memory with errors and create different type of structs within each other.
Sources:

- Lab with Oly in design process and pseudo
- Piazza post about saving memory and Euguene's suggestion on how to copy a string.
- Bv.h and Bv.c in Lab 4