Jaden Liu

Zliu259@ucsc.edu

11/13/2020

CSE13s Fall 2020

Assignment6:

Down the Rabbit Hole and Through the Looking Glass:

Bloom Filters, Hashing, and the Red Queen's Decrees

Design document

For this assignment, we need to find whether it is hatterspeak and translate them or just put them to dungeons.

The flow of the work should be like this:

1. Read the command from the command line
2. Add each word in old speak to the bloom filter and hash table
3. Add each key into bloom filter and hash table, each key has a translation.
4. Determine whether the given text have the forbidden words. If so, then punish them to dungeons or give the translation(correction).

Pre-lab Part 1

1. Write down the pseudocode for inserting and deleting elements from a Bloom filter.
2. Assuming that you are creating a bloom filter with m bits and k hash functions, discuss its time and space complexity.

1. Bloom filter cannot delete an element, because the there might be a collision that two keys stands on the same bit. Therefore, it can't be deleted. But it can be add like this.

```
Void bf_delete(BloomFilter *bf){
free(bf);
bf=NULL;
}

Void bf_insert(BloomFilter *bf, char *key){
uint32_t Val_hash1 = hash(primary,key);
bv_set_bit(bf, val_hash1);
```

```
uint32_t Val_hash2 = hash(secondary,key);
bv_set_bit(bf, val_hash2);

uint32_t Val_hash3 = hash(tertiary,key);
bv_set_bit(bf, val_hash3);
}
```

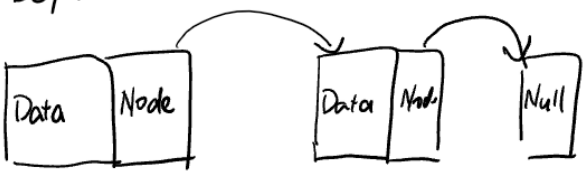
2. Time complexity: O(k)
   Space complexity: O(m)

Pre-lab Part 2

1. Draw the pictures to show the how elements are being inserted in different ways in the Linked list.
2. Write down the pseudocode for the above functions in the Linked List data type.

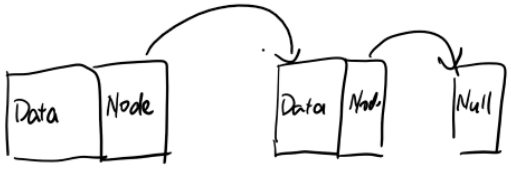
1.

① Default



Inseating:

② Move to front

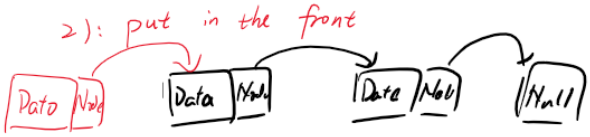


Inseating:

But when search:

1): Delete the original one

2): put in the front



2.

```
ListNode * ll_node_create ( HatterSpeak *gs) {
ListNode head_node=(ListNode *)malloc(sizeof(ListNode));
If head_node == NULL{
```

```
        Printf(“Fail malloc head node”);
        }
 Head_node->next=NULL;
 Head_node->gs=gs;
Return head_node;
}

void ll_node_delete ( ListNode *n){
free(n);
n=NULL;
}

void ll_delete ( ListNode * head ) {
ListNode *i, *j;
i=head;
while(i->next != NULL){
        j = i->next;
        i = j->next;
        free(j);
        j = NULL;
}
free(head);
head=NULL
}

ListNode * ll_insert ( ListNode ** head , HatterSpeak *gs) {
ListNode insert_node=(ListNode *)malloc(sizeof(ListNode));
If head_node == NULL{
        Printf(“Fail malloc insert node”);
        }
Insert_node->next = head;
Head = insert_node;
}

ListNode * ll_lookup ( ListNode ** head , char * key ) {
ListNode **p=head;
ListNode **prev;
While(p->gs->oldspeak != key){
Prev=p;
P = p->next;
}
If(move_to_front){
Return p;
}
```

```
Prev->next=p->next;
p->next=head;
}
```

LL.h(Cited from asgn6.pdf)

```c
#ifndef __LL_H__
#define __LL_H__

#ifndef NIL
#define NIL (void *)0
#endif

#include <stdbool.h>

// If flag is set, ListNodes that are queried are moved to the front.
extern bool move_to_front;

typedef struct ListNode ListNode;

//
// Struct definition of a ListNode.
//
// gs:  HatterSpeak struct containing oldspeak and its hatterspeak translation.
//
struct ListNode {
  HatterSpeak *gs;
  ListNode *next;
};


//
// Constructor for a ListNode.
//
// gs:  HatterSpeak struct containing oldspeak and its hatterspeak translation.
//
ListNode *ll_node_create(HatterSpeak *gs);

//
// Destructor for a ListNode.
//
// n:  The ListNode to free.
//
void ll_node_delete(ListNode *n);

//
// Destructor for a linked list of ListNodes.
//
// head:   The head of the linked list.
//
void ll_delete(ListNode *head);

//
// Creates and inserts a ListNode into a linked list.
//
// head:  The head of the linked list to insert in.
// gs:    HatterSpeak struct.
//
ListNode *ll_insert(ListNode **head, HatterSpeak *gs);

//
// Searches for a specific key in a linked list.
// Returns the ListNode if found and NULL otherwise.
//
// head:   The head of the linked list to search in.
// key:    The key to search for.
ListNode *ll_lookup(ListNode **head, char *key);

#endif
```

Parser.c Parser.h(cited from asgn6.pdf)

```c
#ifndef __PARSER_H__
#define __PARSER_H__

#include <regex.h>
#include <stdio.h>

//
// Returns the next word that matches the specified regular expression.
// Words are buffered and returned as they are read from the input file.
//
// infile:      The input file to read from.
// word_regex:  Pointer to a compiled regular expression for a word.
// returns:     The next word if it exists, a null pointer otherwise.
//
char *next_word(FILE *infile, regex_t *word_regex);

//
// Clears out the static word buffer.
//
void clear_words(void);

#endif
```

parser.h

```c
#include "parser.h"
#include <regex.h>
#include <inttypes.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define BLOCK   4096

```

```c
static char *words[BLOCK] = { NULL };  // Stores a block of words maximum.

//
// Returns the next word that matches the specified regular expression.
// Words are buffered and returned as they are read from the input file.
//
// infile:      The input file to read from.
// word_regex:  Pointer to a compiled regular expression for a word.
// returns:     The next word if it exists, a null pointer otherwise.
//
char *next_word(FILE *infile, regex_t *word_regex) {
  static uint32_t index = 0;  // Track the word to return.
  static uint32_t count = 0;  // How many words have we stored?

  if (!index) {
    clear_words();

    regmatch_t match;
    uint32_t matches = 0;
    char buffer[BLOCK] = { 0 };

    while (!matches) {
      if (!fgets(buffer, BLOCK, infile)) {
        return NULL;
      }

      char *cursor = buffer;

      for (uint16_t i = 0; i < BLOCK; i += 1) {
        if (regexec(word_regex, cursor, 1, &match, 0)) {
          break;  // Couldn't find a match.
        }

        if (match.rm_so < 0) {
          break;  // No more matches.
        }

        uint32_t start  = (uint32_t)match.rm_so;
        uint32_t end    = (uint32_t)match.rm_eo;
        uint32_t length = end - start;

        words[i] = (char *)calloc(length + 1, sizeof(char));
        if (!words[i]) {
          perror("calloc");
          exit(1);
        }

        memcpy(words[i], cursor + start, length);
        cursor += end;
        matches += 1;
      }
```

```
61
62        count = matches;  // Words stored is number of matches.
63      }
64    }
65
66    char *word = words[index];
67    index = (index + 1) % count;
68    return word;
69  }
70
71  //
72  // Clears out the static word buffer.
73  //
74  void clear_words(void) {
75    for (uint16_t i = 0; i < BLOCK; i += 1) {
76      if (words[i]) {
77        free(words[i]);
78        words[i] = NULL;
79      }
80    }
81
82    return;
83  }
```

Hash.h(cited from asgn6.pdf)

```
1
2  #ifndef __HASH_H__
3  #define __HASH_H__
4
5  #ifndef NIL
6  #define NIL (void *)0
7  #endif
8
9  #include "ll.h"
10 #include <inttypes.h>
11
12 //
13 // Struct definition for a HashTable.
14 //
15 // salt:       The salt of the HashTable (used for hashing).
16 // length:     The maximum number of entries in the HashTable.
17 // heads:      An array of linked list heads.
18 //
19 typedef struct HashTable {
20   uint64_t salt[2];
21   uint32_t length;
22   ListNode **heads;
```

```
} HashTable;

//
// Constructor for a HashTable.
//
// length:  Length of the HashTable.
// salt:    Salt for the HashTable.
//
HashTable *ht_create(uint32_t length);

//
// Destructor for a HashTable.
//
// ht:  The HashTable.
//
void ht_delete(HashTable *ht);

//
// Returns number of entries in hash table
//
// h:  The HashTable.
//
uint32_t ht_count(HashTable *h);

//
// Searches a HashTable for a key.
// Returns the ListNode if found and returns NULL otherwise.
// This should call the ll_lookup() function.
//
// ht:      The HashTable.
// key:     The key to search for.
//
ListNode *ht_lookup(HashTable *ht, char *key);

//
// First creates a new ListNode from HatterSpeak.
// The created ListNode is then inserted into a HashTable.
// This should call the ll_insert() function.
//
// ht:          The HashTable.
// gs:          The HatterSpeak to add to the HashTable.
//
void ht_insert(HashTable *ht, HatterSpeak *gs);

#endif
```

Bf.h

```
7
8  #include "bv.h"
9  #include <inttypes.h>
10 #include <stdbool.h>
11
12 //
13 // Struct definition for a BloomFilter.
14 //
15 // primary:     Primary hash function salt.
16 // secondary:   Secondary hash function salt.
17 // tertiary:    Tertiary hash function salt.
18 // filter:      BitVector that determines membership of a key.
19 //
20 typedef struct BloomFilter {
21   uint64_t primary  [2]; // Provide for three different hash functions
22   uint64_t secondary[2];
23   uint64_t tertiary [2];
24   BitVector *filter;
25 } BloomFilter;
26
27 //
28 // Constructor for a BloomFilter.
29 //
30 // size:  The number of entries in the BloomFilter.
31 //
32 BloomFilter *bf_create(uint32_t size);
33
34 //
35 // Destructor for a BloomFilter.
36 //
37 // bf:  The BloomFilter.
38 //
39 void bf_delete(BloomFilter *bf);
40
41 //
42 // Inserts a new key into the BloomFilter.
43 // Indices to set bits are given by the  hash functions.
44 //
45 // bf:    The BloomFilter.
46 // key:   The key to insert into the BloomFilter.
47 //
48 void bf_insert(BloomFilter *bf, char *key);
49
50 //
51 // Probes a BloomFilter to check if a key has been inserted.
52 //
53 // bf:    The BloomFilter.
54 // key:   The key in which to check membership.
55 //
56 bool bf_probe(BloomFilter *bf, char *key);
57
```