Merilyn Kuo
CSE 13S
Spring 2021

Assignment 7
Design Document

Purpose:

Program Implementation:

Bit Vector ADT:

A bit vector can be represented as an array of ones and zeros. We will be using this in the bloom filter. The structure definition of the Bit Vector has been provided in the Assignment PDF. The structure members are the length in bits and the vector array of bytes.

```
struct BitVector {
    uint32_t length;
    uint8_t *vector;
}
```

This bit vector ADT design is from my assignment 5.

The constructor for a bit vector allocates memory for the bit vector and initializes the members of the bit vector. If memory could not be allocated, then any allocated memory must be freed and the pointers nulled.

```
BitVector *bv_create(uint32_t length):
    BitVector *v = (BitVector *) calloc(sizeof(BitVector))
    if memory allocated:
        v->length = length
        v->vector = (uint8_t *) calloc(sizeof(uint8_t))
        if memory not allocated:
            free(v)
            v = null
    return v
```

The destructor function will free any allocated memory and set the vector pointer to null.

```
void bv_delete (BitVector **v):
    if v and v->vector:
        free(v->vector)
        free(v)
        *v = null
```

Return the length of the bit vector.

```
uint32_t bv_length(BitVector *v):
    return v->length
```

Sets the ith bit in the bit vector. The location of the byte is i / 8 and the position of the bit in that byte is i % 8.

```
void bv_set_bit(BitVector *v, uint32_t i):
    byte = i / 8
    bit = i % 8
    v->vector[byte] or (1 << bit)
```

Clears the ith bit.

```
void bv_clr_bit(BitVector *v, uint32_t i):
    byte = i / 8
    bit = i % 8
    v->vector[byte] and ~(1 << bit)
```

Returns the ith bit.

```
uint8 bv_get_bit(BitVector *v, uint32_t i):
    byte = i / 8
    bit = i % 8
    return (v->vector[byte] and (1 << bit)) >> bit
```

XORs the ith bit with the value specified.

```
void bv_xor_bit(BitVector *v, uint32_t i, uint8_t bit):
     byte = i / 8
     bit = i % 8
     v->vector[byte] xor (1 << bit)
```

Prints the bit vector.

```
void bv_print(BitVector *v):
    for i in range length
        print bv_get_bit(v, i)
```

Node ADT:

This function mimics what strdup() does and will be used in the Node ADT.

```
char *str_dup(char *og_str):
    while c in og_str != ‘\0’:
        str_len += 1
        c += 1
    static char *dup_str = (char *) malloc(str_len)
    for c in str_len:
        dup_str[c] = og_str[c]
    return dup_str
```

The node structure given to us in the assignment pdf is as follows:

```
struct Node {
    char *oldspeak;
    char *newspeak;
    Node *next;
    Node *prev;
};
```

The nodes will make up a linked list and each contains oldspeak and newspeak. If the newspeak is null, then the oldspeak is badspeak. The node also has pointers to the previous and following nodes to be able to implement a doubly linked list.

The node constructor creates a copy of the oldspeak and newspeak parameters passed in.

```
Node *node_create(char *oldspeak, char *newspeak):
    initialize node, *n
    if memory for n allocated:
        n->oldspeak = str_dup(oldspeak)
        n->newspeak = str_dup(newspeak)
```

The node destructor frees the node and the memory allocated for the strings.

```
void node_delete(Node **n):
    if *n exists:
        free (*n)->oldspeak
        free (*n)->newspeak
        free *n
        *n = null
```

To print the node, if the node contains oldspeak and newspeak, we print the oldspeak and newspeak. If the node contains only oldspeak, we only print the oldspeak. The exact print statements are provided in the assignment pdf.

```
void node_print(Node *n):
    if n->oldspeak and n->newspeak:
        print n->oldspeak and n->newspeak
    else if n->newspeak == null:
        print n->oldspeak
```

Linked List ADT:

Using linked lists in the hash table will prevent hash collisions. The structure definition of a linked list provided in the assignment pdf is as follows:

```
struct LinkedList {
    uint32_t length;
    Node *head;
    Node *tail;
    bool mtf;
};
```

It contains the length of the list, a head and tail sentinel node, and a boolean called mtf or move to front. Move to front is a technique that optimizes the lookup time when searching for a node that is frequently visited.

The constructor for the linked list allocates memory for the linked list and initializes the struct members. Two nodes for the head and tail will be created and their previous and next pointers set up accordingly.

```
LinkedList *ll_create(bool mtf):
    calloc memory for linked list, ll
    if allocated:
        ll->head = node_create()
        ll->tail = node_create()
        ll->head->next = ll->tail
        ll->tail->prev = ll->head
        ll->mtf = mtf
    return ll
```

The destructor for the linked list should loop through the list and free each node. The pointer to the linked list should be nulled.

```
void ll_delete(LinkedList **ll):
    if ll exists:
        for each node in the linked list:
            node_delete()
        *ll = NULL
```

This function returns the length of the linked list. The length is how many nodes are in the list not including sentinel nodes.

```
uint32_t ll_length(LinkedeList **ll):
    return ll->length
```

This function finds the node containing the same oldspeak as the parameter that is passed in. If found, the pointer to the node is returned. If mtf was true, the node is moved as such. The node's previous node will be set to the head and the node's next node will be set to the node currently next to the head. The move to front procedure is shown in Figure 1. If the node was not found a null pointer is returned.

```
Node *ll_lookup(LinkedList *ll, char *oldspeak):
    for each node in ll:
        if node->oldspeak == oldspeak:
            if node->mtf:
                node->prev->next = node->next
                node->next->prev = node->prev
                node->next = head->next
                head->next->prev = node
                head->next = node
                node->prev = head
            return *node
    *node = null
    return *node
```

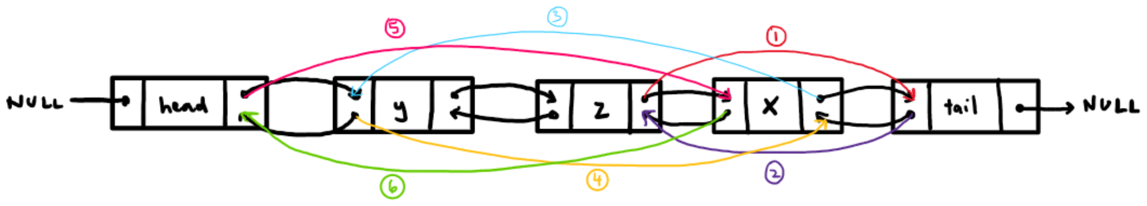

Figure 1. Move to front procedure for node x.

The insert function inserts a new node containing the oldspeak and newspeak at the head of the linked list if the node does not already exist. Figure 2 shows the procedure for inserting a node.

```
Node *ll_insert(LinkedList *ll, char *oldspeak, char *newspeak):
    if ll_lookup == NULL:
        *new = node_create(oldspeak, newspeak)
        new->next = ll->head->next
        new->prev = ll->head
        ll->head->next->prev = new
        ll->head->next = new
    return
```

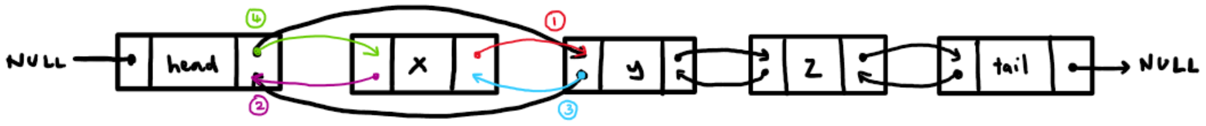

Figure 2. Inserting node x.

The function to print linked lists should loop through the list starting at the node next to the head and print out every node until it reaches the tail.

```
void ll_print(LinkedList *ll):
    for each node in ll from head->next to tail
        node_print
    print length
```

Hash Table ADT:

The hash table structure definition and constructor has been provided to us in the assignment pdf.

Destructor

```
void ht_delete(HashTable **ht):
    if *ht and *ht->lists
        for each item in ht
            ll_delete(item)
        *ht = null
    return


uint32_t ht_size(HashTable *ht):
    return ht->size


void ht_insert(HashTable *ht, char *oldspeak, char *newspeak):
    index = hash(salt, oldspeak) % ht_size()
    if !(ht->lists[index])
        create linked list
    insert linked list at index


Node *ht_lookup(HashTable *ht, char *oldspeak):
    index = hash(salt, oldspeak) % ht_size
    if no list exists at index
        return Null
    else return ll_lookup(ll at index, oldspeak)


uint32_t ht_count(HashTable *ht):
    int count = 0
    for each linked list in ht
        if linked list exists
        count ++
    return count


void ht_print(HashTable (ht):
    for each ll in ht:
        if ll exists
            print ll
        else print new line
```

Bloom Filter ADT:

The structure definition and bloom filter constructor has been provided in the assignment PDF.

Destructor:

```
uint32_t ht_size(HashTable *ht):
    return ht->size


uint32_t bf_size(BloomFilter *bf):
    return bv_length(filter)


void bf_insert(BloomFilter *bf, char *oldspeak):
    index1 = hash(primary, oldspeak) % bf_size
    index2 = hash(secondary, oldspeak) % bf_size
    index3 = hash(tertiary, oldspeak) % bf_size
    set bit at index1, index2, and index3


bool bf_probe(BlookFilter *bf, char *oldspeak):
    index1 = hash(primary, oldspeak) % bf_size
    index2 = hash(secondary, oldspeak) % bf_size
    index3 = hash(tertiary, oldspeak) % bf_size
    return get bit at index1, index2, index3 == 1


uint32_t bf_count(BloomFilter *bf):
    for each i in bloom filter
        if get bit == 1
            count ++
    return count
void bf_print(BloomFilter *bf):
    bv_print(filter)
```

Regular Expression: [a-zA-Z0-9'\-]+

Resources: