Pre Lab questions

Pre-lab Part 1

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

void bf_insert(BloomFilter *bf, char *oldspeak)

- Takes oldspeak and inserts it into the Bloom filter. This entails hashing oldspeak with each of the three salts for three indices, and setting the bits at those indices in the underlying bit vector.

- using speck.c
  - hash1 = hash(bf->primary, oldspeak)
  - hash2 = hash(bf->secondary, oldspeak)
  - hash3 = hash(bf->tertiary, oldspeak)
- call bv_set_bit(bf->filter,hash1)
- call bv_set_bit(bf->filter,hash2)
- call bv_set_bit(bf->filter,hash3)

function for deleting elements (BloomFilter *bf, char *oldspeak)

- we should never delete from the bloom filter in our code bc we can get false negatives that way. This is really bad because a false negative will cause functions that use bloom filter to think something isn't in the hashmap when it is.

- using speck.c
  - hash1 = hash(bf->primary, oldspeak)
  - hash2 = hash(bf->secondary, oldspeak)
  - hash3 = hash(bf->tertiary, oldspeak)
- call bv_clr_bit(bf->filter,hash1)
- call bv_clr_bit(bf->filter,hash2)
- call bv_clr_bit(bf->filter,hash3)

Pre-lab Part 2

1. Write down the pseudocode for each of the functions in the interface for the linked list ADT.

LinkedList *ll_create(bool mtf)

- `The constructor for a linked list`

- dynamically allocate space for the new linked list container
- set length to 0
- create sentinel nodes
- connect sentinel nodes
- set "move to front" variable

void ll_delete(LinkedList **ll)

- `frees the allocated memory of the linked list`

- make sure sentinel nodes are defined
- iterates through all the nodes between head and tail, including tail
  - deletes the previous node to cur
- free the tail node bc there is no NULL->prev
- free the linked list and set it to NULL

void move_to_front(LinkedList *ll, Node *current)

- `moves a node to the front of the linked list`

- take current out of the linked list
  - Node prev = current->prev
  - Node next = current->next
  - prev->next = next
  - next->prev = prev
- stick current to the front
  - fixes current
    - current->next = head->next
    - current->prev = head
  - fixes the head node
    - head -> next = current
  - fixed the node that used to be directly behind head
  - but now it's behind current
    - Node new_next = current->next
    - new_next->prev = current
    - we keep new_next->next the same

Node *ll_lookup(LinkedList *ll, char *oldspeak)

- Searches for a node containing oldspeak.

- for(Node current = head->next; current!=tail; current = current->next)
  - if oldspeak in current == oldspeak
    - if(ll->mtf == true)
      - call move_to_front(ll,current)
    - return current
- return NULL

void ll_insert(LinkedList *ll, char *oldspeak, char *newspeak)

- Inserts a new node containing the specified oldspeak and newspeak into the linked list.

- new = node_create(oldspeak, newspeak)
- head = ll->head
- fix new
  - new->prev = head
  - new->next = head->next
- head->next->prev = new
  - re links the node that used to be right behind the head node
- head->next = new
  - fixes the head node

void ll_print(LinkedList *ll)

- prints the linked list

- iterates through all the nodes between head and tail and prints them

Pre-lab Part 3

1. Write down the regular expression you will use to match words with. It should match hyphenations and contractions as well. The regular expression refers to the pattern you will be using. For example, a regular expression to match strings consisting of only lowercase characters would be: `"[a-z]+"`.

[A-Za-z0-9]+(('|-)[A-Za-z0-9]+)*

-matches any letter/number with as many ‘ or - matched with any letter/number after it as possible

Marian Zlateva
mzlateva@ucsc.edu
2/26/2021

CSE13s Fall 2020
The Great Firewall of Santa Cruz:
Bloom Filters, Linked Lists and Hash Tables

Description:

In this lab, I created a program that reads from stdin, detects bad words the user imputed from newspeak.txt and badspeak.txt, and notifies them of their errors. If some or all of their words have good versions of them, the program will notify the user of the good versions.

Flags for both generator.c and decoder.c:

-m:	will enable the move-to-front rule.
-h:	Specifies that the hash table will have size entries (the default will be 10000).
-f	Specifies that the Bloom filter will have size entries (the default will be 2^{20}).

Files:

- banhammer.c :
  - contains main()
- ll.h :
  - Defines the interface for the linked list ADT.
- ll.c :
  - Contains the implementation of the linked list ADT.
- node.h :
  - Defines the interface for the node ADT.
- node.c :
  - Contains the implementation of the node ADT.
- bf.h :
  - Defines the interface for the Bloom filter ADT.
- bf.c :
  - Contains the implementation of the Bloom filter ADT.
- bv.h :
  - Defines the interface for the bit vector ADT.
- bv.c :
  - Contains the implementation of the bit vector ADT.
- parser.h :
  - Defines the interface for the regex parsing module.
- parser.c :
  - Contains the implementation of the regex parsing module.
- Makefile :
  - This is a file that will allow the grader to type make to compile your program.

Functions:

banhammer.c

int main()

- ➔ parses the commandline options
  - ➞ h
    - ➘ size specifies that the hash table will have size entries. (the default will be 10000).
  - ➞ f
    - ➘ size specifies that the Bloom filter will have size entries. (the default will be 2^20).
  - ➞ m
    - ➘ enables the move-to-front rule
- ➔ creates the hash table to store the badspeak words
- ➔ creates the bloom filter
- ➔ opens the badspeak.txt file
- ➔ opens the newspeak.txt file
- ➔ read in badspeak words from badspeak.txt
- ➔ read in badspeak and newspeak words from newspeak.txt
- ➔ close files once we are done reading from them
- ➔ create linked lists to hold bad words and fixable words
- ➔ compile regex
- ➔ use regex to parse stdin
  - ➞ make word into lowercase so that it can match the words in the files
  - ➞ if word is not in bloomfilter, word is not in hashtable so go to next word
  - ➞ if word is not in hashtable, go to next word
  - ➞ if word has no translation, insert it into bad_words
  - ➞ if word has a translation, insert it into fixable_words
- ➔ print output if needed
- ➔ free memory from the heap

void lowercase(char *string)

- ➢ makes an uppercase string to lowercase

- ➔ iterate through all characters in the string
- ➔ convert each character to it's lowercase version

ll.c

LinkedList *ll_create(bool mtf)

- ➢ the constructor for a linked list

- ➔ dynamically allocate space for the new linked list container
- ➔ set length to 0
- ➔ create sentinel nodes
- ➔ connect sentinel nodes
- ➔ set "move to front" variable

void ll_delete(LinkedList **ll)

- ➢ frees the allocated memory of the linked list

- ➔ makes sure ll and *ll are not NULL
- ➔ make sure sentinel nodes are defined
- ➔ iterates through all the nodes between head and tail, including tail
  - ➔ deletes the previous node to cur
- ➔ free the tail node bc there is no NULL->prev
- ➔ free the linked list and set it to NULL

uint32_t ll_length(LinkedList *ll)

- ➢ gets the length of the linked list, which is equivalent to the number of nodes in the linked list, not including the head and tail sentinel nodes.

- ➔ return ll->length

void move_to_front(LinkedList *ll, Node *cur)

- ➢ moves a node to the head of a linked list

- ➔ unlink cur from linked list
- ➔ stick cur to front
- ➔ fix the head node so that it is connected to cur
- ➔ fix the node that used to be directly behind head but now it's behind cur
- ➔ we don't change new_next->next

Node *ll_lookup(LinkedList *ll, char *oldspeak)

- ➢ searches for a node containing oldspeak

- ➔ iterates through all the nodes between head and tail
- ➔ if a node matching to oldspeak is found, return it
- ➔ if "move to front" is enabled, move the found node to the front

➔ if no matching node was found, return NULL

void ll_insert(LinkedList *ll, char *oldspeak, char *newspeak)

➢ searches for a node containing oldspeak

➔ if the node is already present in the array list, don't do anything
➔ creates the new node
➔ fix new node
➔ relinks the node that used to be right behind the head node
➔ links the new node to the head node
➔ increments the size of the linked list

void ll_print(LinkedList *ll)

➢ prints the linked list

➔ iterates through all the nodes between head and tail and prints them

node.c

bool string_cpy(char **pointer, char *string_to_cpy)

- ➢ copies a string to another string

- ➔ makes sure the memory adress of the string is not NULL
- ➔ check if string needs to be copied
- ➔ copies the string
  - ➞ allocates space for the new string
  - ➞ if allocating space failed, return false to show that it was not successful
  - ➞ copy the string character by character into *pointer
- ➔ copying the string was successful so return true

Node *node_create(char *oldspeak, char *newspeak)

- ➢ the constructor for a node

- ➔ set n->oldspeak and n->newspeak to the passed in parameters using string_cpy
- ➔ n->next = NULL
- ➔ n->prev = NULL

void node_delete(Node **n)

- ➢ free's the memory allocated in the heap for a node to delete it

- ➔ free((*n)->oldspeak)
- ➔ free((*n)->newspeak)
- ➔ free(*n)
- ➔ set *n to NULL

void node_print(Node *n)

- ➢ prints a node

- ➔ if the node has a newspeak, print both oldspeak and newspeak
- ➔ if the node doesn't have a newspeak, only print oldspeak

bf.c

BloomFilter *bf_create(uint32_t size)

- ➢ the constructor for a bloom filter

- ➔ if size is 0, then you can't allocate space
- ➔ dynamically allocate space for the new bloom filter container
- ➔ set primary salt
- ➔ set secondary salt
- ➔ set tertiary salt
- ➔ dynamically allocate space for the bit vector inside the bloom filter

void bf_delete(BloomFilter **bf)

- ➢ frees the memory of the bloom filter

- ➔ bv_delete(&((*bf)->filter))
- ➔ free(*bf)
- ➔ set bf to NULL

uint32_t bf_length(BloomFilter *bf)

- ➢ returns the bloom filter's size

- ➔ return bv_length(bf->filter)

uint32_t bf_get_hash_index(BloomFilter *bf, uint64_t salt[], char *oldspeak)

- ➢ gets the index of the hash value of an oldspeak string

- ➔ return hash(salt, oldspeak) % bf_length(bf);

void bf_insert(BloomFilter *bf, char *oldspeak)

- ➢ Takes oldspeak and inserts it into the Bloom filter.

- ➔ bv_set_bit(bf->filter, bf_get_hash_index(bf, bf->primary, oldspeak))
- ➔ bv_set_bit(bf->filter, bf_get_hash_index(bf, bf->secondary, oldspeak))
- ➔ bv_set_bit(bf->filter, bf_get_hash_index(bf, bf->tertiary, oldspeak))

bool bf_probe(BloomFilter *bf, char *oldspeak)

- ➢ checks if an oldspeak could have been inserted in the bloom filter

➔ checks if all the hash indexes of the oldspeak are true

void bf_print(BloomFilter *bf)

- ➢ prints the bloom filter

➔ call bv_print(bf->filter)

bv.c

uint32_t bytes(uint32_t bits)

- ➢ gets the amount of bytes needed for an amount of bits

- ➔ if bits can perfectly fit into a number of bytes, then return bits/8
- ➔ if bits doesn't perfectly fit into a number of bytes, allocate 1 extra byte so that the remaining bits can have space

BitVector *bv_create(uint32_t length)

- ➢ constructor for a bit vector

- ➔ if size is 0, then you can't allocate space
- ➔ dynamically allocate space for the new bit vector container
- ➔ set length of the bit vector
- ➔ dynamically allocate space for the bit vector

void bv_delete(BitVector **bv)

- ➢ free's the memory allocated towards a bit vector

- ➔ free((*bv)->vector)
- ➔ free(*bv)
- ➔ set *bv to NULL

uint32_t bv_length(BitVector *bv)

- ➢ gets the length of a bit vector

- ➔ return bv->length

void bv_set_bit(BitVector *bv, uint32_t i)

- ➢ sets a bit to 1 in a bit vector

- ➔ make sure i is inside scope of the array, no need to check if i<=0 bc i is unsigned
- ➔ get the index of the byte that the bit we want lays in
- ➔ get the index of the bit we want in that byte
- ➔ make a byte with a 1 shifted i spots in (put a 1 at spot i)
- ➔ set the byte in the ADT to the union of 'byte' and 'mask'

void bv_clr_bit(BitVector *bv, uint32_t i)

- ➢ sets a bit to 0 in a bit vector

- ➔ get the index of the byte that the bit we want lays in
- ➔ get the byte
- ➔ get the index of the bit we want in that byte
- ➔ make a byte with all 1s except one 0 at spot i
- ➔ multiply the 'mask' and 'byte' bytes together so that all bits that aren't in spot i stay the same while spot i turns to a 0

uint8_t bv_get_bit(BitVector *bv, uint32_t i)

- ➢ gets a bit in a bit vector

- ➔ get the index of the byte that the bit we want lays in
- ➔ get the byte
- ➔ get the index of the bit we want in that byte
- ➔ multiply the 'mask' and 'byte' bytes together so that all bits that aren't in spot i must be 0s
- ➔ move the bit to the least significant bit spot in the byte

void bv_print(BitVector *bv)

- ➢ prints a bit vector

- ➔ iterates through the bit vector and prints each byte with bv_get_git()

Inputs

badspeak.txt

- remove any words that come from this

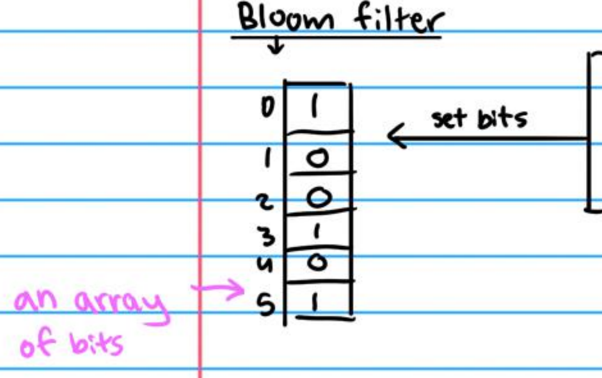
newspeak.txt

- contains <u>bad word</u> <u>new word</u>
- replace bad word with new word

Examples

badspeak.txt	newspeak.txt	
meme	asgn4	trashfire
java	windows	garbage
	↑ old speak	↑ newspeak translation

Bloom filter



bf-insert("meme")
- hash(bf→primary, "meme") = 0
- hash(bf→secondary, "meme") = 3
- hash(bf→tertiary, "meme") = 5 (↑ salts)

- hash(bf→primary, "java") = 1
- hash(bf→secondary, "java") = 2
- hash(bf→tertiary, "java") = 5

(hash collision)

bf_probe("meme")
- hash(bf→primary, "meme") = 0
- hash(bf→secondary, "meme") = 3
- hash(bf→tertiary, "meme") = 5

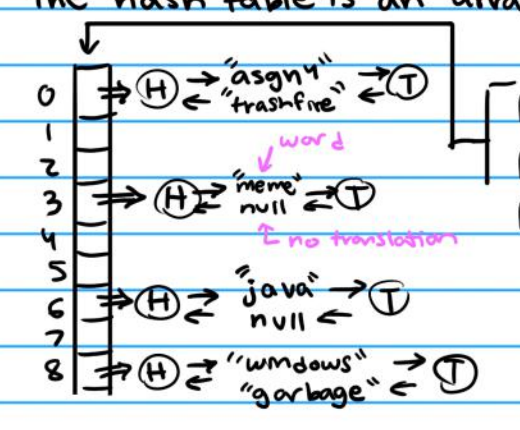
- b/c index 0,3,5 are all 1s, "meme" is most likely added in the bloom filter

<u>other ex</u> if(probe(apple) ⇒ 0,3,4)
- even though 0 & 3 are set, b/c 4 isn't set, apple is def not added in the bloom filter

Bit vector : array of bits

Hash table

the hash table is an array of linked lists



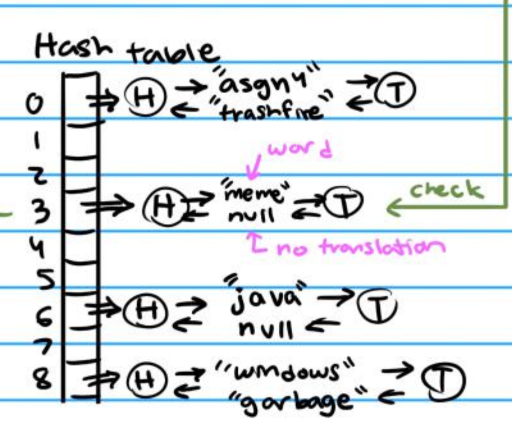
- hash(ht→salt, "meme") = 3
- hash(ht→salt, "java") = 6
- hash(ht→salt, "asgn4") = 0

Bloom filter

user says "i used java for asgn4 on windows"

bf_probe(..., "i")
- hash1 ⇒ 2
- hash2 ⇒ 6
- hash3 ⇒ 9
- 2,6,9 are set to 1 so return true

ht_lookup("i") ⇒ hash→3

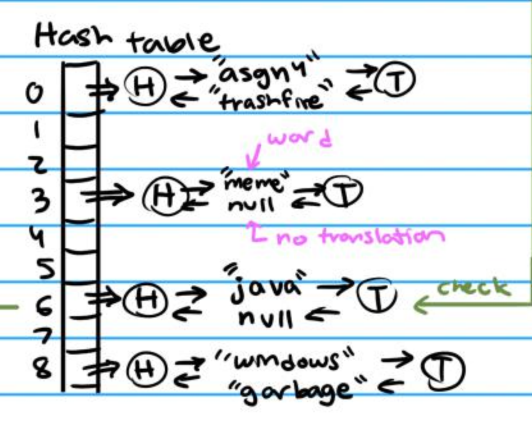


can not find "i", so "i" is not a bad word

---

bf_probe(..., "java")
- hash1 ⇒ *
- hash2 ⇒ *
- hash3 ⇒ *
- *,*,* are set to 1 so return true

ht_lookup("java") ⇒ hash→6



Linkedlist *forbidden
Linkedlist *torevise

add java to forbidden

forbidden → H ⇄ "java" null ⇄ T

---

bf_probe(..., "asgn4")
⋮
found "asgn4"
add asgn4 to torevise

torevise → H ⇄ "asgn4" trashfire ⇄ T

Print : Hey comrad,
You used some bad words.
- java
Think of these improvements in joycamp!
- windows → garbage
- asgn4 → trashfire

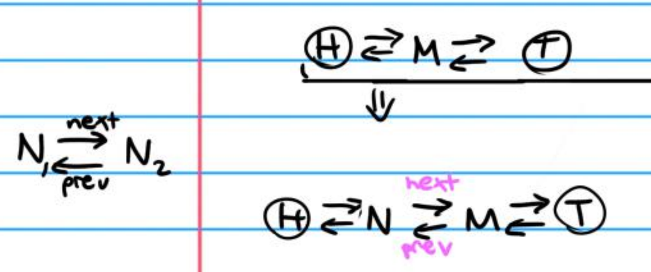
Arraylist

```
ll_create(bool mtf){        // mtf: move to front
    LinkedList *ll = malloc(sizeof(linked list));
    ll→length = 0

    ll→head = node_creat()
    ll→tail = node_create()
    ll→head→next = ll→tail
    ll→tail→prev = ll→head
    ll→mtf = mtf

    return
}
```

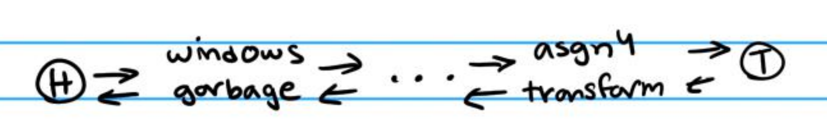
Adding a node

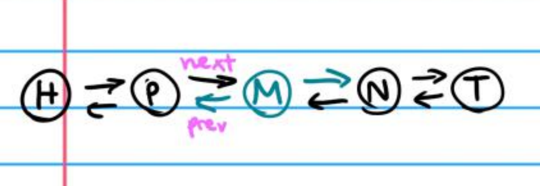


- n→prev = head
- n→next = head→next
- head→next→prev = n   sets M prev to N
- head→next = n

Move to front

- used to make words that are used most often go in the front





- Move M to front
  - Node p = m→prev
  - Node n = m→next
  - p→next = n
  - n→prev = p

  (takes M out of list)

  - fixes m: m→next = head→next
  - m→prev = head
  - fixes head: head→next = m
  - fixes node now behind m: m→next→prev = m

  (sticks node to front)

<u>Other</u>

X^*

00000 is accepted by (0^*)
11111111 is accepted by (1^*)

010110 is accepted by $(0|1)^*$

---

regular expressions

regx = "ab" → matches "ab"
doesn't match "abc"