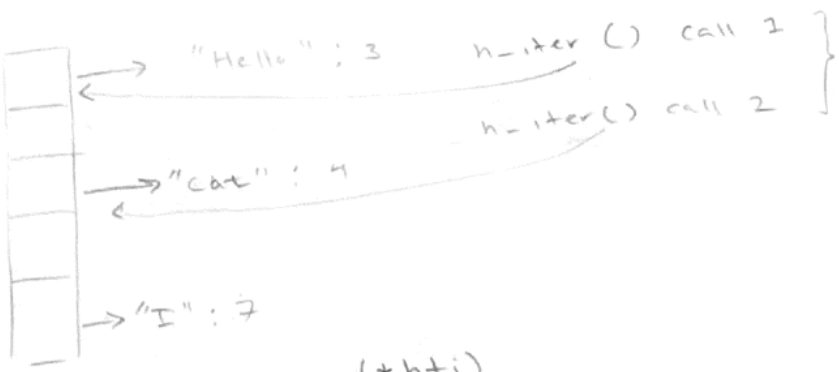
Assignment 7:

In some 3D space, which of known texts are closest to new value.

Hash table size is 1.3 → 1.5x the expected unique words

ht_size (*ht)
return (top); every add of unique word increments to top.



Node *ht_iter(*hti)
do { • do atleast once
slot ++; move to next slot
} while (ht[slot % ht_size] → frequency == 0 or slot > ht_size)
• If the slot goes past max exit % to Loop
• go to next non-empty instance
return ht[slot % ht_size()];

Ω ← lower bound ← best case Ω(1)

Θ ← Average case

O ← upper bound ← worst case O(n)

SPECK Cipher

- hash (salt 1, "cat") → 6
  hash (salt 2, "cat") → 100

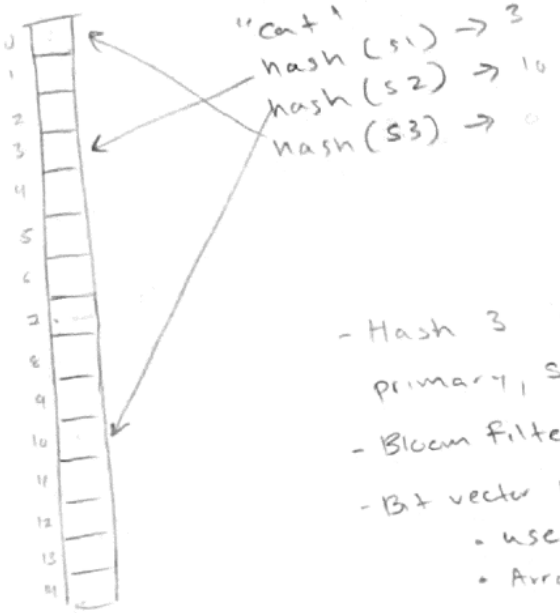
] different salts result in different values

salt is an array of 2 uint64_t s
- we use 128 bit salts

Hash table uses 1 salt

Bloom Filters
- A bit vector, reports if an entry has not been seen before. Can result in false positives because probabalistic data structure. Salt [2]

{ WENT OVER IN LECTURE REVIEW }



- Hash 3 times for Bloom Filter
  primary, secondary, tertiary
- Bloom filter is constructed from a bit vector
- Bit vector is a dynamically size of bits
  - use malloc, and realloc
  - Array of uint8s

bv-create (n bits)
Allocate $\lfloor n/8 \rfloor$ if 8 ∤ n (+1) bytes

bv-length
return bv → length

bv-clrbit
bv $[n/8] >> n \% 8 \,\&\, 0$

bv-setbit
bv $[n/8] >> n \% 8 \mid 1$

Distance functions

$\vec{u}$ = Anon.txt (infile)

$\vec{v}$ = known.txt refFile

$\vec{u}$ = <1, 0, 1>

$\vec{v}$ = <0, 2, 1>

Anon.txt "Hello World"

known.txt "Goodbye, goodbye, world"

<"hello", "goodbye", "world">

- use regex to make lowercase

u[unique], v[unique] // dimension is # of unique words in both infile and reffile (maybe calloc)

unique = ht_size(&ht)

? is there a ht for each file ?

use a for loop

```
in_wc = 0; // infile wordcount
ref_wc = 0; // reffile wordcount

for (i=0; i< unique; i++)
    word = ht_iter(ht) -> word; // node -> word
    if (word == NULL) break;
    u[i] = text_frequency(infile, word) // u_i = u_i / |u_i|
    u[i] /= in_wc
    v[i] = text_frequency(reffile, word) // v_i = v_i / |v_i|
    v[i] /= ref_wc

float dist = 0 // dist of 1 of the 3 dist methods
```

Manhattan

```
for (i < unique)
  dist += |u[i] - v[i]|

return dist;
```

Euclidean

```
for (i < unique)
  dist += sqrt((u[i] - v[i])^2)

return dist;
```

Cosine

```
for (i < unique)
  dist += u[i] . v[i]

return 1 - dist
```

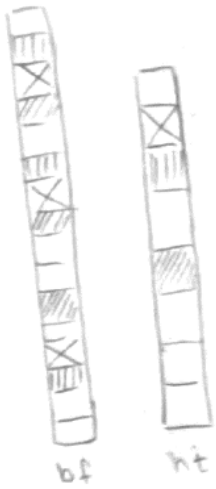
file (anonymous)

"Kim likes cats"

noise

"the and of ..."

Text {
  word_count
  bf
  ht
}



- "Kim" wc = 1
- "likes" wc = 2
- "cats" wc = 3

lib.db

2

Author A
texts/a.txt → "Alice likes cats"

Author B
texts/b.txt → "Bob likes dogs"

Manhattan dist

$\vec{u}, \vec{v} \quad \sum_{i=1}^{n} |\vec{u}_i - \vec{v}_i|$

Anon A

Anon ht

"kim": 1
"likes": 1
"cats": 1

A ht

"Alice": 1
"likes": 1
"cats": 1

```
for (key,value in anon_txt) {
  Anon_freq = text_freq
  A_freq = text_freq
```

Anon.txt

	Anon_Freq	A_freq	$\sum \lvert u_i - v_i \rvert$
"kim"	$\frac{1}{3}$	$\frac{0}{3}$	$\frac{1}{3}$
"likes"	$\frac{1}{3}$	$\frac{1}{3}$	0
"cats"	$\frac{1}{3}$	$\frac{1}{3}$	0
		(word_contains function)	
	$\frac{1}{3}$	$\frac{1}{3}$	
"Alice	$\frac{0}{2}$		
dist to A			$\frac{2}{3}$

$\in$ a.txt ? $\in$ anon.txt

Priority Queue

- getting the k^{th} nearest neighbors
  → have the lowest value distance (text_dist(...))

pq stores capacity elements ?what exactly is capacity based on?

- remove from end first.
- lowest distance has highest priority, min queue
- use heap or quick sort because pq can become very large

distance away from author's text compared to anonymous

```
bool enqueue(*q, *author, dist)
    if (pq_full(q)
        return False
    q[top] = (author, dist)   ?what is an author, dist pair?
                              • ? A struct?
                              • ? 2 element array?
                                [can be anything
                                 struct is probably best]
    top++  // move up top
    return True
```

create a new instance of AuthorDist
ad_create(author, dist)

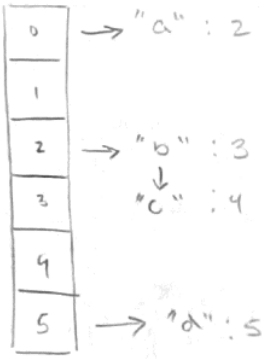
```
bool dequeue(*q, **author, *dist)
    if (pq_empty(q)): return False
    top--  // move down top
    *author, *dist = q[top]   // get values of author and dist from queue
    *author = q[top]->author
    *dist = q[top]->distance   // maybe have unpack function
    ad_delete(&q[top])  // AD node no longer needed, can be freed
    return True
```

```
struct AuthorDist {
    char author
    float distance
}
```

Hash Table

speck → takes in a word outputs a number
use mod to make it fall into the array

if collision add to linked list (can do binary search tree but seems like a lot of work)




```
Node *ht_insert(ht, *word)
    Msize = ht->Max_size// total spaces in hash table
     key; // stores key
     hash(ht->salt, &key) // function of speck takes
     key %=Msize // keep within bounds
     Node * link_node :=ht[key]
     while (link_node -!= 0 OR != NULL)
          if(link_node->next == NULL):
               link_node->next = create_node(word);
               link_node = link_node->next
               break;
     ht->size ++;
     return link_node;
```

```
Node *ht_lookup(*ht, *word)        <- Bloom Filter if False stop
    key; hash(ht->salt, &key);        if true check
     key % = ht->size;
     Node * look_node = ht[key]
     while (look_node != Null or 0)
          if (look_node ->word == word)
               return look_node;
          if (look_node ->next != Null)
               look_node = look_node ->next

     return NULL; // Nothing
```

```
ht_create (size)
    HashTable *ht = (HashTable *) malloc (sizeof (HashTable))
    ht -> size = size
    ht -> n_items = 0
    ht -> slots = (Node **) calloc(sizeof (Node), size)
    return ht;

ht_delete (**ht)
    free *ht -> slots
    free (*ht)
ht_size (*ht)    // USED WRONG IN OTHER AREAS
    return ht -> size;
```

Assignment 7 : Author Identification

purpose of the program is to take an anonymous txt file and find k authors and how different the styles are and maybe find a match

General Outline

1) get 2 files Anon.txt, Known.txt

2) Create 2 Text ADT, each with
   - Hashtable ht_create → ht
   - BloomFilter bv_create → bv
   - word_count ; A for loop

3) Find Distances with either Man, Euclid, or Cos
   - Fill the hts of both, exclude noise → also find the # of unique words in both files together u_wc
   - Create 2 arrays All both with frequency of a corresponding word. (Might need to make 3rd Array for words) and divide by their own individual wc from their ht
   - push onto priority_queue

4) Sort Priority Queue 0th index should be greatest distance, a min queue with heap or quick sort

5) Output take the First k off the queue and output it.

Texts

Struct Text
    Hash table ht
    BloomFilter bf
    int wordcount ← total number of words in text not unique

text_create (infile, noise)
    noise "a", "the", "of" ← Not indicative of authors dictation
        "him", "her", "them"
    ◦ create a text file of only noise words
    →text_create(file, NULL) // create noise text
    if current word being looked at in noise table do not add
    it to table

text_dist (text1, text2, metric) ← 1 of the 3 measurement methods

text_frequency (text, word)
    return (double) f/(# unique words)
        → f is frequency of word

text_contains (text, word)
    bf → if (NO) return false
    bf → if (YES) check ht

Priority Queue
    use Last Assignment

12/160 = 100
12+X

X = 0.6L
1.5L 3:5

0.6L : 1L Rum
0.4L : 0.5L GF
0L : 1.5L whisky

L(A) / L(A)+L(W) (Prof) = w_prof
1L : 3L

parsing words at a time

- word character should contain letters from a-z, A-Z and accept contractions, and hyphenations

msg = "Hello world"

[a-z]+ : ello world

[a-z A-Z]+ : Hello world

+ means 1 or more * means 0 or more ? 0 or 1

[ ] character set

Ex 0b10110101 "0b(01)+"

WORD "[a-z A-Z]+" ← using through words

while (nextword (file, WORD) != NULL)

- for apostaphe bob's => word "bob's"
  bobs' => word "bobs"

- for hyphen bo-be => "bo-be"
  bobe- => "bobe"

' and - must have a letter before and after it
if that is met can have any number of them