DESIGN

Yifan Zou

February 2022

1 Introduction

This assignment is going to identify the most likely authors for an anonymous sample of text given a large database of texts with known authors.

This assignment includes following source code files:

- bf.c -Bloom filter ADT
- bv.c -Bit vector ADT
- ht.c -hash table ADT and hash table iterator ADT
- identify.c -c main() and author identification program
- node.c -node ADT
- parser.c -regex parsing module
- pq.c -priority queue ADT
- text.c -text ADT

2 Main Program

Running the command-line option to specify the variables using in the program.

```
-d database     Database of authors and texts [default: lib.db]
-k matches      Set the number of top matches to display [default: 5]
-n noisefile    Set file of words to ignore [default: noise.txt]
-l limit        Set noise word limit [default: 100]
-e              Set distance metric as Euclidean distance [default]
-m              Set distance metric as Manhattan distance
-c              Set distance metric as cosine distance
-v              Enable verbose printing of program run
-h              Display program help and usage
```

After running the command-line,

- Create a Noise Text as the noise word filter.
- Create an anonymous text from the text passed in to stdin, using noise filter.
- Open database and get the number n (capacity of the pq) to create pq.
- Loop through the database and read the author name and text path pairs, create a new text for each known user.
  - Compute the distance between author's text and anonymous text.
  - Enqueue the author's name and distance into the pq, free the text.
- After read all the text from the database, dequeue from the pq, and get the top k likely matches.
- Free all the memory by create text, and pq.

3 Source Files

3.1 Nodes

Each node contains an author name and distance between the anonymous text, which would be used in priority queue for sorting.

3.1.1 Node *node_create(char *word)

```
create memory for node
if success:
    duplicate the word and set count to 1
return the node pointer
```

3.1.2 void node_delete(Node **n)

```
if node exist:
    free the memory of the node
    set the pointer to null
```

3.1.3 void node_print(Node *n)

```
print the word and count of the node for debug
```

3.2 Bit Vectors

Bit vector represents an array of bits, which would be used in Bloom Filter to denote if one item exists or not. Length is the numbers of bits hold in the vector.

3.2.1 BitVector *bv_create(uint32_t length)

```
create memory for a bitvector
if succeed:
    init the length
    create memory for vector in the bv, size should be the length
    (use calloc so all the bits would be set to 0)
    if fail to create memory:
        free memory
        set the pointer to null
return the pointer to bv
```

3.2.2 void bt_delete(BitVector **bv)

```
if bv exist:
    if the vector in the bv exist:
        free the vector
    free bv
    set the pointer to null
```

3.2.3 uint32_t bv_length(BitVector *bv)

```
return the length in bv
```

3.2.4 bool bv_set_bit(BitVector *bv, uint32_t i)

```
if i(index wanna be set) < length:
    set the bit to 1
    (use bv->vector[i / 8] |= (0x1 << (i % 8)))
    return true to announce create successfully
return false (the index is in valid)
```

3.2.5 bool bv_clr_bit(BitVector *bv, uint32_t i)

```
if i(index wanna be set) < length:
    set the bit to 0
    (use bv->vector[i / 8] &= ~(0x1 << (i % 8)))
    return true to announce create successfully
return false (the index is in valid)
```

3.2.6 bool bv_get_bit(BitVector *bv, uint32_t i)

```
// If and only if i in within the range and the bit is 1, return true.
if i < length:
    if i == 1:
    (using (bv->vector[i / 8] >> (i % 8)) & 1 == 1 to verify)
    return true
return false
```

3.2.7 void bv_print(BitVector *bv)

```
print all the bits in bv to debug
```

3.3 Bloom Filters

Bloom Filter would be used to verify if a word is inserted into the hashtable. A bloom filter contains three hash function salts, and a BitVector called filter. Using salts, we would get three index for the inserted word. If and only if the three indexes are labeled as true, we can say the word might exists, and then we could use hashtable.

3.3.1 BloomFilter *bf_create(uint32_t size)

```
create memory for the Bloom Filter
if success:
    init the three salts
    create memory for filter
return bf
```

3.3.2 void bf_delete(BloomFilter **bf)

```
if bf exist:
    if bf->filter exist:
        deleter the bv
    free bf
    set the pointer to null
```

3.3.3 uint32_t bf_size(BloomFilter *bf)

```
return bv_length(bf->filter)
```

3.3.4 void bf_insert(BloomFilter *bf, char *word)

```
get three index of the word using three salts
use bv_set_bit to insert three indexes into filters.
```

3.3.5 bool bf_probe(BloomFilter *bf, char *word)

```
get three index of the word using three salts
use bv_get_bit to get the bits in filter
if all 3 indexes == true:
    return true
else:
    return false
```

3.3.6 void bf_print(BloomFilter *bf)

```
use bv_print to debug
```

3.4 Priority Queue

Priority Queue is using to sort the distance between known author and anonymous text, the smaller the distance between the texts, the higher the priority would be. In my code, I added an struct called Priority Queue Entry as the items in the Priority Queue.

3.4.1 struct PQEntry

An entry packages author name and distace between two authors together, just like a node.

```
struct PQEntry {    // From Eugene's section.
    char *author;
    double dist;
};
```

3.4.2 PQEntry *entry_create(char *author, double dist)

```
create memory for an entry
if success:
    duplicate the author and dist
return pointer of the entry
```

3.4.3 void entry_delete(PQEntry **e)

```
if entry exists:
    free the memory
    set the pointer to null
```

3.4.4 void entry_print(PQEntry *e)

```
print the author and dist to debug
```

3.4.5 struct PriorityQueue

size is the number of entries in the pq, capacity is the maximum number that could contain.

```
struct PriorityQueue {
    uint32_t size;
    uint32_t capacity;
    PQEntry **E;
};
```

3.4.6 PriorityQueue *pq_create(uint32_t capacity)

```
create memory for pq
if success:
    Initialize size and capacity
    create memory for entry, set all the pointers to entry as null
return the pointer of pq
```

3.4.7 void pq_delete(PriorityQueue **q)

```
create memory for pq
if success:
    Initialize size and capacity
    create memory for entry, set all the pointers to entry as null
return the pointer of pq
```

3.4.8 bool pq_empty(PriorityQueue *q)

```
check if the size of pq is 0
```

3.4.9 bool pq_full(PriorityQueue *q)

```
check if the size of pq == capacity
```

3.4.10 uint32_t pq_size(PriorityQueue *q)

```
return the size of the pq
```

3.4.11 bool enqueue(PiorityQueue *q, char *author, double dist)

```
check if the q exist:
    if there is enough space for the new entry:
        loop through the q to find a good place for insert:
            if entry->dist >= q[index-1]->dist: (good place)
                insert the entry into the q
                size += 1
            else: (bad place)
                shift the previous entry to the current place
                go to the previous index
```

3.4.12 bool dequeue(PriorityQueue *q, char **author, double *dist)

```
if q exist:
    if the q is not empty:
        output the author and dist
        delete the dequeued entry
        size -= 1
```

```
    if there is still entry remained in the q:
        shift all the entry forward

    return true
return false (there is no entry in the pq)
```

3.4.13 void pq_print(PriorityQueue *q)

```
use entry_print to print all the items in the q
```

3.5 Hash Tables

Hashtable is an array of node pointers. Hashtable contains salt, which is used to get the hashed index. A node pointer would be inserted into the hashed index, if there is already a node pointer exist, then go down until find a null pointers. If the pointer already exists, then just add the frequency by 1.

3.5.1 HashTable *ht_create(uint32_t size)

```
create memory for hashtable
if success:
    Initialize salt and size
    create memory for slots inside the hashtable
    set all the pointer to null
return the pointer of ht
```

3.5.2 void ht_delete(HashTable **ht)

```
if hashtable exists:
    check through the hashtable
    if there is remained slot:
        free the slot using node_delete
    free the slots
    zero ht->size
    free ht
    set pointer to null
```

3.5.3 uint32_t ht$_(HashTable * ht)$

```
return ht->size
```

3.5.4 Node *ht_lookup(HashTable *ht, char *word)

```
use salt get the index
init the count (to check if check all the items in ht)
check through the hashtable:
    if slot exist && slot->word == word:    // Found the slot.
```

```
            return node
        index += 1
        count += 1
    return null
```

3.5.5 Node *ht_insert(HashTable *ht, char *word)

```
    get the index using salt
    init count = 0
    loop through the ht to find the word or suitable slot for new word:
        if word exists: // Found the word.
            slot->count += 1
            return the pointer of the slot
        else word not exist:    // Found suitable place.
            create a new node
            set node count to 1
            insert the node to ht
            return the pointer
        check next slot
        index += 1
        count += 1
    return null (the ht is full)
```

3.5.6 void ht_print(HashTable *ht)

```
    use node_print to print all the slots in ht
```

Hash table iterator is used to iter all the slots in the hashtable.

3.5.7 HashTableIterator *hti$_c$reate($HashTable * ht$)

```
    create memory for hti
    if success:
        init table to a hash table
        set the slot to 0
    return pointer
```

3.5.8 void hti_delete(HashTableIterator **hti)

```
    if hti exists:
        free hti
        set pointer to null
```

3.5.9 Node *ht_iter(HashTableIterator *hti)

```
    loop through the ht to find a valid entry:
        if valid:
```

```
            return pointer of the slot
        else:
            check the next slot
    return null if loop over the ht
```

3.6 Parser

I added a lower_case_word function before returning the word to change all the parsed word into lower case.

3.6.1 char *lower_case_word(char *word)

```
    index = 0
    loop through the char *:
        use tolower() to change char[i]
        i += 1
    return word
```

3.7 Texts

Text is used to turn a word file into statistics version, which contains a hash table, bloom filter, and total word count of the text.

3.7.1 Text *text_create(FILE *infile, Text *noise)

```
    create memory for text
    if success:
        create hash table and bloom filter
        set word count to 0
        if there is no noise text:  // Then purpose is to make a noise filter.
            while wordcount < noiselimit:   // only put specific number of noise word
                use parse to get all the word
                insert the word into ht and bf
                word_count += 1
        else there is a noise filter:
            use parser to get all the lower case words:
                use bf_probe to check if the word in the noise filter:
                    ht_lookup to double check:
                        if not in the noise filter:
                            insert into ht and bf
                            word_count += 1
        return pointer of text
    return null
```

3.7.2 void text_delete(Text **text)

```
    if text exists:
```

```
    delete ht and bf
    zero out the word count
    free text
    set pointer to null
```

3.7.3 double text_dist(Text *text1, Text *text2, Metric metric)

```
init dist to 0
if (MANHATTAN):
    init sum = 0
    create hti1 and hti2
    iter over hti1:
        get freq1 and freq2 from each node:
            sum += abs(freq1 - freq2)
    iter over hti2:
        if node we get exists in text1:
            pass
        else:
            sum += freq2
    dist = sum
else if (EUCLIDEAN):
    init sum = 0
    create hti1 and hti2
    iter over hti1:
        get freq1 and freq2 from each node:
            sum += (freq1 - freq2)^2
    iter over hti2:
        if node we get exists in text1:
            pass
        else:
            sum += freq2^2
        dist = sum^(1/2)
else if (COSINE):
    init sum = 0
    create hti1 // Need only one hti as we just need to compute the words exist in
    both texts.
    iter through text1 to calculate the dist:
        sum += freq1 * freq2
    dist = 1 - sum
```

3.7.4 double text_frequency(Text *text, char* word)

```
if word exist:
    return node->count/word_count
return 0
```

3.7.5 bool text_contains(Text *text, char* word)

```
if word contains in both ht and bf:
    return true
return false
```

3.7.6 void text_print(Text *text)

```
use hti to iter all the item in text's hash table
```