Assignment 7 Design Document

Nicholas Chu

March 2022

1 Introduction

The purpose of this assignment is to utilize hashing in order to analyze linguistic styles, and when given an anonymous piece of text able with a high degree of accuracy to identify its author given a large database of texts with known authors.

1.1 Nodes

Nodes will be used to store a word and their count, which will be used to fill up the hash table. The struct looks like:

```
struct Node{
    char *word;
    uint32_t count;
};
```

1.2 Node *node_create(char *word)

Constructs a node and copies the word given by the parameter and then returns it.

```
struct Node *n = (Node *)malloc(sizeof(Node))
n->word = strdup(word)
n->count = 1
return n
```

strdup should copy the pointer to word and dynamically allocate memory for it, and saving into the Node.

1.3 void node_delete(Node **n)

Destructor for a node, should free word, the node itself, and set its pointer to NULL.

```
free(n->word)
free(n)
n = NULL
```

1.4 void node_print(Node *n)

Should print the node, is used for debugging.

```
printf("word: %s\n", n->word)
printf("count: %" PRIu32 "\n", n->count)
```

2 Hash Tables

The program will need to use hash tables in order to store unique words from a sample text. Its struct looks like:

```
struct HashTable {
    uint64_t salt[2];
    uint32_t size;
    Node ** slots; //Array of node pointers
};
```

Salt is for the hash function which will be used with a SPECK cipher (used in a later function to create an array for key to hash), size is the number of indices that the hash table will have, and slots is an array that stores nodes.

2.1 Salt

Salt is an array of 2 unit64t's which acts almost like a seed, and will create a unique index for a given word given a specific salt, each hash table will use one salt and creates a different hash, which will later be used to create three separate hashes for the Bloom filter. The function uint32_t hash(uint64_t salt[], char *key) in the provided file speck.c returns the the index for a key given a 128-bit salt. Salts for this program are provided in the file salts.h

2.2 HashTable *ht_create(uint32_t size)

The constructor for the hash table, it should be responsible for constructing an instance of a hash table, setting its size and allocating the proper memory for slots based on the size specified in the parameter. The salt should be set to the salt defined in salt.h, SALT_HASHTABLE_LO and SALT_HASHTABLE_HIGH.

```
stuct HashTable *h = (Hashtable *)(malloc(sizeof(HashTable)))
h->size = size;
h->slots = (Node **)calloc(size, sizeof(Node))
h->salt = [SALT_HASHTABLE_LO, SALT_HASHTABLE_HI]
return h
```

2.3 void ht_delete(HashTable **ht)

Destructor for the hash table, should free the slots in the hash table then the hash table itself and finally set the pointer to the hash table to zero.

```
free(*ht->slots)
free(*ht)
*ht = NULL
```

2.4 uint32_t ht_size(HashTable *ht)

Should simply just return the hash table's size. This is the number of total slots for the hash table.

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

Searches the hash table, specifically the slots of nodes, for a node containing a specific word. It should return the pointer to the place the node was found or NULL if nothing was found. This is done first by getting the hash index of the word (which acts as the key), then linearly probing the hash table until an node is found that has the word is found.

```
count = 0
//hash(ht->salt, word) generates the index for word
index = hash(ht->salt, word) % ht->size
while count < ht->size:
    temp word = ht->slots[index]
    if word and temp word are the same:
        return index
    count += 1
    index = (index += 1) % ht->size
return NULL
```

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

Inserts a word into the hash table by creating a new node with the word and setting its frequency to 1. If a node containing the word is already in the hash table it increments the value of frequency. Once done it should return the node that was inserted or if nothing was able to be inserted then return NULL. You do this by first calling ht_lookup and trying to see if an existing node with the same word already exists, if a node already exists use the returned index and increment the frequency of that node and return it. If nothing was found then go through the hash table until finding a NULL spot and create a new node there, set it, and return it.

```
Node *n = ht_lookup(ht, word)
if (n):
    n->frequency += 1
    return n
count = 0
index = word % ht->size
while count < ht_size:
    if ht->items[index] == NULL:
        n = node_create(word)
        ht->items[index] = n
```

```
        return n
    index = (index + 1) % ht->size
    count += 1
return NULL
```

2.7 void ht_print(HashTable *ht)

Prints the contents of the hast table. Should print the hash table.

3 Hash Table Iteration

This functions are for iterating over the entries of the hash table. The struct will need to store the table itself and the slot(think like the index).

```
struct HashTableIterator{
    HashTable *table;
    uint32_t slot;
};
```

3.1 HashTableIterator *hti_create(HashTable *ht)

Creates the iterator which should iterate over the hash table specified in the parameter. It should allocate the proper memory set the slot to 0.

```
HashTableIterator hti = (HashTableIterator*)malloc(sizeof(HashTableIterator))
hti->table = ht
hti->slot = 0
return hti
```

3.2 void hti_delete(HashTableIterator **hti)

A destructor function that frees the contents of the hash table iterator. This should free the table in hti, hti itself, and finally setting its pointer to NULL.

```
free(hti->table)
free(hti)
hti = NULL
```

3.3 Node *ht_iter(HashTableIterator *hti)

Returns the pointer to the next entry in the hash table that has a valid entry. It should return NULL after the whole table has been iterated over. This should simply just be a while loop that iterates over the table until it finds a slot that is not null and returns it.

```
n = ht->table[slot]
while n is NULL and slot < ht_size(ht):
    hti->slot += 1
    n = ht->table[slot]
```

```
if n is not null:
    return n
return NULL
```

4 Bit Vectors

A bit vector is simply just an array of bits, with the bits used to determine is something is true or false. This will be used inside the bloom filter. The struct looks like:

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

Since it would be inefficient to use a uint8_t to represent a bit, the functions will need to use bitwise operations to set and access bits within the bytes of the vector.

4.1 BitVector *bv_create(uint32_t length)

This is a constructor function for a bit vector, it should create a pointer to a new bit vector of 0's and then allocate the proper memory based off length, set length, and return the pointer to the new bit vector. If the memory allocation failed then it should just return NULL. Remember that lenght is the number of bits for the vector.

```
BitVector bv = (*BitVector)malloc(sizeof(HashTableIterator))
bv->length = length
if memset(bv->vector, 0, (length + (8 - 1)) / 8) and bv:
    return bv
return NULL
```

4.2 void bv_print(BitVector *bv)

This should just print the contents of the bit vector for debugging purposes.

4.3 void bv_delete(BitVector **bv)

This a destructor for the the bit vector, it should free the memory set in its vector, the bit vector itself, then set the pointer to the bit vector to NULL.

```
free(bv->vector)
free(bv)
bv = NULL
```

4.4 uint32_t bv_length(BitVector *bv)

Should just return the length assigned to the bit vector.

4.5 bool bv_set_bit(BitVector *bv, uint32_t i)

The function should set the i'th bit to 1 and return true when successful or false when i is out of range. It should first check if i is within range (i/8 should be less than length), then preform a bitwise operation to set the bit and return true after doing so. Length is divided by 8 since it is measuring in bits, not bytes.

```
if i is less than bv->length/8:
    bv->vector[i/8] |= (0x1 << i % 8)
    return true
return false
```

4.6 bool bv_clr_bit(BitVector *bv, uint32_t i)

The function sets the i'th bit to zero using a similar process to set bit. It checks if i is within the range of the vector then if it was using bitwise operation set the bit at index i to 0 and return true, else return false.

```
if i/8 is less than bv->length:
    bv->vector[i/8] &= ~(0x1 << i % 8)
    return true
return false
```

4.7 bool bv_get_bit(BitVector *bv, uint32_t i)

Gets the i'th bit from the vector. It should return true if that bit is set to 1 and false when set to 0 or out of range.

```
if i is in range:
    //checks if bit it 1
    if bv->vector[i/8] & (0x1 << i % 8):
        return true
return false
```

5 Bloom Filters

A Bloom filter is supposed to represent a bit vector. It is for the purpose of deterministically checking the contents of a hash table and if a specific key exists within it. It does this by using thhree seperate hashfunctions and thus three salts The struct should look like:

```
struct BloomFilter {
    uint64_t primary [2]; // Primary hash function salt.
    uint64_t secondary [2]; // Secondary hash function salt.
    uint64_t tertiary [2]; // Tertiary hash function salt.
    BitVector *filter;
};
```

They should be filled with their respective salts provided within salt.h.

5.1 BloomFilter *bf_create(uint32_t size)

This is a constructor for a Bloom filter and should set their respective salts to the ones found in salts.h. It should also create a bit vector using the provided size.

```
BloomFilter bf = (BloomFilter *)malloc(sizeof(BloomFilter))
bf->primary = [SALT_PRIMARY_LO, SALT_PRIMARY_HI]
bf->secondary = [SALT_SECONDARY_LO, SALT_SECONDARY_HI]
bf->tertiary = [SALT_TERTIARY_LO, SALT_TERTIARY_HI]
bf->filter = bv_create(size)
return bf
```

5.2 void bf_delete(BloomFilter **bf)

This is a destructor function which free the memory allocated from the constructor. It should free the bit vector and the bloom filter itself, and finally set its pointer to NULL.

```
bv_delete(*bf->filter)
free(*bf)
*bf = NULL
```

5.3 uint32_t bf_size(BloomFilter *bf)

Returns the size of the bloom filter, the length of the filter (the number of bits in the bit vector).

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

Inserts a word into the bloom filter by hasing the word with each of the salts using hash(salt, word) to get the indexes for the three filters. You would use these three indexes

```
primary bit index = hash(bf->salt->primary, word)
secondary bit index = hash(bf->salt->secondary, word)
tertiary bit index = hash(bf->salt->tertiary, word)
bv_set_bit(bf, primary bit index)
bv_set_bit(bf, secondary bit index)
bv_set_bit(bf, tertiary bit index)
```

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

Goes though the bloom filter and checks if the word is hashed at their hashes given their three respective hashes then it should return true. It will return false when not.

```
primary index = hash(bf->salt->primary, word)
secondary index = hash(bf->salt->secondary, word)
tertiary index = hash(bf->salt->tertiary, word)
if bv_get_bit(bv, primary index),
   bv_get_bit(bv, secondary index), and
   bv_get_bit(bv, tertiary index):
    return true
return false
```

5.6 void bf_print(BloomFilter *bf)

Debug function that should print the bloom filter.

6 Lexical Analysis with Regular Expressions

For this assignment when scanning in words with regular expressions through using the regex.h and using the expression [a-zA-Z'-]+ should be sufficient for detecting words, contractions, and hyphenations in the input text. A breakdown of this is regular expression is shown as:

```
[       //Any expression of:
a-z     //lower case letters,
A-Z     //upper case letters,
'       //apostrophes,
-       //and hyphens.
] +     //with 1 or more instances (in cases when connected by ' or -)
```

The provided file parser.c contians the provided functions next_word() and regcomp(). The function regcomp() compiles a regular expression which will be used in next_word(). next_word() takes the next word of an input file from the file stream of the specified file and compiles it into a regular expression. It will take two arguments, one which is the infile and the other is a pointer to a compiled regular expression.

7 Texts

This abstract data type is used for representing the distribution of words in a file. It will need to use a hash table and a bloom filter in order to do so.

```
struct Text{
    HashTable *ht;
    BloomFilter *bf;
    uint32_t word_count;
};
```

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

This is a constructor for text. It takes an input file, which it will scan words from, and another Text struct, noise, which will be used to filter out frequently used words when constructing the hash tables in order to avoid polluting them. Thus when using the Text struct it is important that you first create an instance of the text struct using the noise file and setting its noise to null. This noise text will be used to hold the contents of frequently used words and compared to when creating other Texts to not use specific words. The file noise.txt contains frequently used words to avoid. An example of this kind of implementation would look like:

```
Text *noise = textcreate(noise.txt, NULL)
Text *t = textcreate(infile, noise)
```

The function should create and allocate the memory for the hash table and bloom filter. Their sizes should be set to 2^21 and 2^19 respectively. It it is unable properly allocate the memory for Text then it should just return NULL. Then if everything was properly created, then it should go through the infile and filling up the HashTable and BloomFilter.

```
if !(Text *t = (Text *)malloc(sizeof(Text))):
    return NULL
if !(t->bf = bf_create(2^21)):
    return NULL
if !(t->ht = ht_create(2^19)):
    return NULL
t->word_count = 0

for word from infile:
    if word not in noise:
        add word to t->ht and t->bf

return n
```

7.2 void text_delete(Text ** text)

Deletes text, should free the hash table and bloom filter of the Text and then the Text itself.

```
ht_delete(*text->ht)
bf_delete(*text->bf)
free(*text)
```

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

This function is used to get the distance between the two Texts based on the specified metric. These metrics specify the method that their distance will include Manhattan, Euclidean, and Cosine distance and are found in metric.h. It will do this by iterating over each text and finding the specified nodes and their frequencies, then using the frequencies of each node to calculate their distance and normalized frequency based off the specified metric.

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

Returns the normalized frequency of a word in the text, and 0 if it is not in the list. This is done first by checking the bloom filter to first see if the word is there. If it is not in the bloom filter then return 0, otherwise iterate through the hash table and look for a node in the hash table that has a matching word. It should get the frequency of the node and return the normalized frequency.

```
if word in ht->bf:
    HashTableIterator *hti = hti_create(*ht)
    while word is not matched:
        n = ht_iter(ht)
        if n->word matches word:
            hti_delete(hti)
```

```
            return normailized n frequency
return 0
```

7.5 double text_contains(Test *text, char *word)

Returns true or false when when a word in in the text. This is done by using a bloom filter and the hash table. If the bloom filter returns false then it should just return 0, else it should look through the hash table.

```
if word in ht->bf:
    HashTableIterator *hti = hti_create(*ht)
    while word is not matched:
        n = ht_iter(ht)
        if n->word matches word:
            hti_delete(hti)
            return true
return false
```

7.6 void text_print(Text *text)

Prints contents of the Text, meant for debugging.

8 Priority Queue

This is a priority queue that for storing the author and the distance of the author's text and the anonymous text, ranking them by smallest distance. This can be done through using 2 structs. This implementation is very similar to Assignment 6's priority queue.

```
stuct pqNode{
    char *author;
    double dist;
};

struct PriorityQueue{
    uint32_t head;
    uint32_t tail;
    uint32_t capcity;
    pqNode **queue;
};
```

The author and distance will be stored in nodes, which are stored in queue.

8.1 pqNode pqNode_create(char *author, double dist)

A constructor for the priority queue nodes, not to be confused with the hash table nodes which contain word and count, with the author's name and the distance. It should create a new instance and set its respective parameters to the contents of this new instance and return it.

```
pqNode pqN = (pqNode *)(malloc(sizeof(pqNode)))
pqN->author = author
pqN->dist = dist
return pq
```

8.2 pqNode pqNode_delete(pqNode **pqN)

A destructor function for a priority queue node. Should just free its contents and set its pointer to NULL.

8.3 void swap(pqNode *x, pqNode *y)

Just swaps the pointers for two specified nodes.

8.4 uint32_t min_child(PriorityQueue *q, uint32_t first, uint32_t last)

Finds and returns the index of the child node with the smallest distance of the node at the index given by first.

```
left = 2 * first
right = left + 1
if right < last dist and right's dist is lager:
    return right
else:
return left
```

8.5 void fix_heap (PriorityQueue *q, uint32_t first, uint32_t last)

Fixes the heap, used for organizing nodes so they follow the conventions of the heap data structure from the given index first.

```
found = false
parent = first
child = min_child(q, parent, last)
while parent < last // 2 and found is false:
    if node distance at index parent is larger than the child's:
        swap the nodes at parent and child
        child = min_child(q, parent, last)
    else:
        found = true
```

8.6 void build_heap(PriorityQueue *q, uint32_t first, uint32_t last)

This function build a heap using the queue from a specified range of indexes given by first and last.

```
for (parent = last/2, parent > first-1, i-=1):
    fix_heap(q, parent, last)
```

8.7 PriorityQueue *pq_create(uint32_t capacity)

Creates the priority queue. It should set the capacity to the capacity specified by the parameter and create a queue (an array) and allocate the proper memory for them. The function also sets the head and tail to 1 in order to simulate 1 indexing.

```
PriorityQueue *q = (PriorityQueue *)(malloc(sizeof(PriorityQueue *)))
 q->head = 1
 q->tail = 1
 q->capacity = capacity
 q->queue = (pqNode **)calloc(capacity, sizeof(pqNode))
 return q
```

8.8 void pq_delete(PriorityQueue **q)

Deletes a priority queue, it should free the memory allocated by the struct and it queue, then set its pointer to NULL.

8.9 bool pq_empty(PriorityQueue *q)

It returns true or false if the priority queue is empty.

```
if q->head == q->tail:
   return true
return false
```

8.10 bool pq_full(PriorityQueue *q)

Returns true or false if the priority queue is full.

```
if q->head == q->capacity:
    return true
return false
```

8.11 uint32_t pq_size(PriorityQueue *q)

Returns the size of the priority queue.

```
return pq->head
```

8.12 bool enqueue(PriorityQueue *q, Node *n)

This function takes a node n and enqueues it into the priority queue q. To do this you need to add the node to the array and then build it into a heap. In the code below it first adds the node from the parameter to the queue's array then uses an insertion sort to sort the array from nodes with the greatest frequency to the lowest.

```
if q is full:
    return false
set top of queue to node n
build_heap(q, q->tail, q->head)
q.head += 1
return true
```

8.13 bool dequeue(PriorityQueue *q, Node **n)

This dequeues a node from the priority q and saves it to n and returns true when it is able to dequeue and node or false when when the queue is empty. You set the node to top of the queue, swap it with the tail, and finally rebuild the heap. This will takes the the node that the tail points to and save it to n**, then increment the tail.

```
if q is empty:
    return false
n = q->queue[q->head - 2]
swap head and tail nodes
build_heap(q, q->tail, q->head-1)
q.head -=1
return true
```

8.14 void pq_print(PriorityQueue *q)

This is a debug function that will print the specified priority queue. Should print the head, tail, and array of the priority queue.

9 Identify

This is the main program which takes in an anonymous text and prints the top specified number of authors. The smaller the distance, the higher they are ranked. The program should:

- First create a Text using noise.txt and NULL which will be used as the noise text. Letters in the noise word should also be all lower cased.
- Open up specified database, if nothing was specified open lib.db. Let n be the number of author text pair which is located at the top of the file.
- Using n, create a priority queue using n as its size.
- Go through the database the lines of the data base and in the process:
  - Get the two lines, the top line is that author's name (which will be later used for the node) and path to their identified work.
  - Open the path and the file, making sure to create a new Text, and computing the distance.
  - Create a node with the author's name and distance, and enqueue it to the priority queue

- The priority queue should now be sorted from least to greatest distance. Now start popping off the nodes from the priority queue and printing out the number of specified authors.