Assignment 7: *Design*

Sriramya Prayaga

March 14, 2022

Abstract

In this document, I will be describing my design process for Assignment 7. Assignment 7 is essentially simulating an author identification program –that is, it essentially compares a given text to texts from various, well-known authors, and determines which author was most likely to have written it.

1 Pseudocode/Structure

Because there are many moving parts to create the author identification algorithm, there are many files required for this assignment.

1.1 Hash Table

The first file includes functions that are necessary for a Hash table implementation. There are 6 functions in this file: ht_create(), ht_delete(), ht_size(), ht_lookup(), ht_insert(), and ht_print(). There is also the hash table structure definition in this file. The structure consists of a uint64_t two-item array called salt, a uint32_t size value, and an array of Node pointers called slots. The pseudocode for these functions is below:

```
HashTable *ht_create(uint32_t size) {
    dynamically allocate memory for a HashTable of size size called ht
    if (memory was not allocated) {
        return a NULL pointer
    }
    set ht->size = size
    set ht->salt[0] to SALT_HASHTABLE_LO
    set ht->salt[1] to SALT_HASHTABLE_HI
    dynamically allocate memory for an array of node pointers
    set ht->slots = pointer to array of node pointers
    return ht
}

void ht_delete(HashTable **ht) {
    free the nodes in the *ht-> slot
    free *ht->salt
```

```
    free(*ht)

}

uint32_t ht_size(HashTable *ht) {
 return ht->size
}

Node *ht_lookup(HashTable *ht, char *word) {
    for (all nodes in the ht) {
        if (node->word = word) {
             return node
        }
    }
    return NULL pointer
}

Node *ht_insert(HashTable *ht, char *word) {
  if (ht_lookup) {
     set node->count += 1
  } else {
     set Node *node_new = node_create()
     set node_new->count += 1
  }
  return node_new
}

void ht_print(HashTable *ht) {
   print (all values in ht)
}
```

1.2 Hash Table Iterator

As we'll also need to iterate over the Hash Table, there will also be a hash table iterator created for this purpose. There are three functions, and a structure meant for this purpose. The structure consists of a pointer to a hash table, and a uint32_t slot value. The functions are: hti_create(), hti_delete(), and ht_iter(). The pseudocode for these functions is below:

```
HashTableIterator *hti_create(HashTable *ht) {
 dynamically allocate memory for a HashTableIterator called hti
 set hti->table = ht
 set hti->slot = 0
}
```

```
void hti_delete(HashTableIterator **hti) {
 free *hti
}

Node *ht_iter(HashTableIterator *hti) {
    if (ht->size = hti->slot) {
        set ht->slot += 1
        return (ht->slots[hti->slot])
    }
}
```

1.3 Node

The next structure needed is the nodes structure —which represents a character, and its word count. These structures contain a pointer to a character, and a uint32_t count value. The three functions used are node_create(), node_delete(), and node_print(), and the pseudocode is below.

```
Node *node_create(char *word) {
         allocate memory for a Node node

         if node pointer exists {
           set node word to word
           set node count to 0
         }
         return node
}

void node_delete(Node **n) {
    free n
    set n to NULL
}

void node_print(Node *n) {
    print(n word and n count)
}
```

1.4 Bloom Filters

The next file includes a bloom filter structure. It includes 3 uint64_t salt values, and a pointer to a bit vector called filter. The functions used are: bf_create(), bf_delete(), bf_size(), bf_insert(), bf_probe(), and bf_print(). This file is used to check if a key in a hash table has been seen before. The pseudocode for these functions is as shown below:

```
BloomFilter *bf_create(uint32_t size) {
```

```
  dynamically allocate memory for a BloomFilter of size size called bf
  set bf->primary = {SALT_PRIMARY_LO, SALT_PRIMARY_HI}
  set bf->secondary = {SALT_SECONDARY_LO, SALT_SECONDARY_HI}
  set bf->tertiary = {SALT_TERTIARY_LO, SALT_TERTIARY_HI}
  set bf->filter = value of bv_create(size)
}

void bf_delete(BloomFilter **bf) {
 free *bf->filter
 free *bf
}

uint32_t bf_size(BloomFilter *bf) {
  return bf->filter->size
}

void bf_insert(BloomFilter *bf, char *word) {
  set one = hash(bf->primary, word)
  set two = hash(bf->secondary, word)
  set three = hash(bf->tertiary, word)
  call bv_set_bit(bf->filter, one)
  call bv_set_bit(bf->filter, two)
  call bv_set_bit(bf->filter, three)
}

bool bf_probe(BloomFilter *bf, char *word) {
    set one = hash(bf->primary, word)
    set two = hash(bf->secondary, word)
    set three = hash(bf->tertiary, word)

    if (bv_get_bit(one), bv_get_bit(two), bv_get_bit(three) are all set) {
       return true
    } else {
       return false
    }
}

void bf_print(BloomFilter *bf) {
 print(bits in bloom filter)
}
```

1.5 Bit Vectors

The next file implements bit vectors, which are needed when trying to manipulate the bloom filters ADT discussed above, The file contains a uint32_t length value, and a pointer to a uint8_t vector value. Below

is the pseudocode for these functions.

```
BitVector *bv_create(uint32_t length) {
   dynamically allocate memory for a BitVector bv of length value length
   set top to 0
   make values in bits[] equal 0
   return bv
}

void bv_delete(BitVector **bv) {
   free bv
}

uint32_t bv_length(BitVector *bv) {
   return bv->length
}

bool bv_set_bit(BitVector *bv, uint32_t i) {
    if i > bv_length {
       return false
    }
    take a BitVector with all bits but bit at index i set to 0
    (done through left-shifting the value)
    OR that Bitvector value with bv
    return true
}

bool bv_clr_bit(BitVector *bv, uint32_t i) {
    if i > bv_length {
       return false
    }
    take a BitVector with all bits but the bit at index i set to 1
    (done through left-shifting the value)
    AND that value with bv
    return true
}

bool bv_get_bit(BitVector *bv, uint32_t i) {
    if i > bv_length {
       return false
    }
    take a BitVector with all bits but bit at index i set to 0
    (through left-shifting the value)
    AND that BitVector value with bv
    by right-shifting that value and OR it with 1
```

```
    return true
}

void bv_print(BitVector *bv) {
    print(value of pushed bits, and popped bits)
}
```

1.6 Texts

This file contains a structure, and six functions which support the text ADT. This files contains functionality that handles the parsing of a text file, and contains the calculations of the distances between the texts. The structure consists of a pointer to a hash table, a pointer to a bloom filter, and a uint32_t word count value. The functions in the file are: text_create(), text_delete, text_dist(), text_frequency(), text_contains(), and text_print(). The pseudocode for this file is included below.

```
create regular expression for parsing words

Text *text_create(FILE *infile, Text *noise) {
    dynamically allocate memory for a Text called text
    if text is NULL {
      return NULL pointer
    }
    set text->ht = ht_create(2^19)
    set text->*bf = bf_create(2^21)
    convert all letters to lower case letters
    compile regex
    for all words in infile parse words using next_word() and regex{
      if noise is not NULL {
       if word is not in bloom filter {
         add word to hash table with ht_insert()
         add word to bloom filter with bf_insert()
         increment word count
       } else if word is not in hash table {
         add word to hash table with ht_insert()
         add word to bloom filter with bf_insert()
 increment word count
       }
      } else {
         if word_count is less than noise limit {
            add word to hash table with ht_insert()
            add word to bloom filter with bf_insert()
            increment word count
         }
         else {
            break loop
```

```
        }
      }
    }
    free regex
    return text
}

void text_delete(Text **text) {
   free *text->bf
   free *text->ht
   free *text
}

double text_dist(Text *text1, Text *text2, Metric metric) {
    calculate distance from text1 to text2 by iterating over text1 with ht_iter()
    calculate distances from words in text2 but not in text1 by
    iterating over text2 with ht_iter()

    switch if metric is Manhattan {
      compute Manhattan distance/word_count
    } else if Metric is Euclidean{
      compute Euclidean distance/word_count
    } else if metric is cosine {
      return cosine distance/word_count
    }
}

double text_frequency(Text *text, char *word) {
  if (word is not in text) {
     return 0;
  }
  return word_count
}

bool text_contains(Text *text, char *word) {
  if word is in text {
    return true
  } else {
    return false
  }
}

void text_print(Text *text) {
print (the text, text_dist(), text_frequency(), and text_contains())
}
```

1.7 Priority Queue

Another data structure needed is a priority queue. It will be implemented using insertion, and contain a head, tail and size value. There are eight functions in this file: pq_create(), pq_delete(), pq_full(), pq_size(), enqueue(), dequeue(), and pq_print(). I have also created an additional structure called authors which can hold an author's name, and a distance. The pseudocode for this file is as below.

```
PriorityQueue *pq_create(uint32_t capacity) {
   allocate memory to create a priority queue called pq that can hold
   capacity number of nodes
   if pq was created properly {
       set pq head to 0
       set pq tail to 0
       set pq size to 0
       set pq capacity to capacity
       dynamically allocate memory for an array of author pointers
       set pq items to dynamically allocated array
       return pq
   } else {
       return a NULL pointer
   }
}

void pq_delete(PriorityQueue **q) {
    if *q exists {
      free *q->items
      set *q->items to NULL
      free(*q)
      set (*q) = NULL
    }
   }

bool pq_empty(PriorityQueue *q) {
   if pq has no nodes {
       return true
    } else {
       return false
   }
}

bool pq_full(PriorityQueue *q) {
    if pq has capacity number of items {
      return true
```

```
    } else {
      return false
   }
}

uint32_t pq_size(PriorityQueue *q) {
    returns pq size
}

bool enqueue(PriorityQueue *q, char *author, double dist) {
  set authors->author_name =  author
  set authors->dist= dist
  set j = pq_size(q) +q->tail
  set k = pq->size
  if n is NULL {
     return false
  }
  if pq is full {
     return false
  }
  if pq is empty {
     set q->items[q->tail] = temp
  } else {
     while (author's dist < q->items[j-1]->dist) {
        set q->items[j] = q->items[j - 1]
        decrease j,k by 1
     }
  }
  increase q->size by 1
  set q->head = q->head + 1
  return true
}

bool dequeue(PriorityQueue *q, char **author, double *dist) {
   if q is empty {
      return false
   }
   set author = to q->items[q->tail->author_name]
   set dist = to q->items[q->tail->author_dist]
   increment q-> tail by 1
   decrease q->size by 1
   return true

}
```

```
void pq_print(PriorityQueue *q) {
   print(all elements in priority queue)
}
```

1.8 Identify

This file contains the main program, which, utilizing the files and functions above, prints out the authors whose diction most closely resembles the given text. The pseudocode for this file is below:

```
set default variables
parse command-line options using getopt()
call text_create()
call fopen() to open the input file
set n = to first element read from infile with fprintf()
call pq_create(n)
for all values in infile {
    set author = fgets() value of the name of the author
    set path = fgets() value of the path of the text file
    set author_text = fopen() to open author text
    if (file could not be opened) {
        read the next line
    }
    call text_create() to get an instance of a text
    filter out noise words
    call text_contains() to see if the text is already there
    set dist =  text_dist(author_text, input_text)
    call enqueue(author, dist)
}
for (k authors in the priority queue) {
 call dequeue(author k)
 print (the  matches to stdout)
}
free memory allocated
close infile, and author file
}
```

2 Summary

For this assignment, several files and functions are necessary for implementing author identification. We will implement all of the functions described above in bf.c, bv.c, ht.c, identify.c, node.c, pq.c, and text.c. We were also given a parser.c file to help with parsing the text, a salt.h header file which contains salt definitions, speck.c/h to define a hash function, and a metric.h to define distance metrics. We were also given header files for all of the formerly stated c files as well.