# Assignment 7 Design

Nhan Nguyen

March 2022

## 1 Introduction

### 1.1 Description

The purpose of this program is to find the author with the most similarity in terms of text to the current text in the input. This is done by calculating the distance between words in each of the paragraphs compared

### 1.2 List of files

- **identify.c:** Provide the implementation for the text identifier
- **bf.h:** The interface for bf.c
- **bf.c:** The implementation for the Bloom Filter
- **bv.h:** The interface for bv.c
- **bv.c:** The implementation for the Bit Vector
- **ht.h:** The interface for ht.c
- **ht.c:** The implementation for the Hash Table
- **pq.h:** The interface for pq.c
- **pq.c:** The implementation for the Priority Queue
- **text.h:** The interface for text.c
- **text.c:** The implementation for the Text data structure
- **node.h:** The interface for node.c
- **node.c:** The implementation for the Node data structure
- **salts.h:** Defines the salt used in the Bloom Filter and the Hash function
- **speck.h:** The interface for speck.h

- **speck.c:** The implementation hash function using the SPECK cipher
- **salts.h:** Defines the salt used in the Bloom Filter and the Hash function
- **metric.h:** Defines the different distance calculating methods
- **parser.h:** The interface for parser.c
- **parser.c:** The implementation of the regex module
- **Makefile:** This file formats program into clang-format and also compiles it. This Makefile will be responsible for compiling encode.c decode.c with all the files specified above
- **README.md:** A text file in Markdown format that describes how to build the program, as well as describing the problems encountered while making the program
- **DESIGN.pdf:** This PDF file describes the idea behind the program with visualization and pseudocode
- **WRITEUP.pdf:** This PDF file describes the program's behavior after running a few iterations

# 2 node.c

This file contains the necessary functions to create a Node data structure

## 2.1 node_create()

This function creates a Node object by allocating a memory for the Node container, assign the word to the one in the parameter and assign the count to the default value of 1 **Pseudocode:**

```
node_create(word)
    allocate memory for the Node container
    if Node contain != NULL:
        Node's word = word
        Node's count = 1
    return Node
```

## 2.2 node_delete()

This is the destructor method for the Node. Free the memory in the following order: Node's word then the Node container itself **Pseudocode:**

```
node_delete(node pointer)
    free memory for the node pointer's word
    free the node pointer
```

node pointer = NULL

### 2.3 node_print()

This is just a debug function for the node, the choice of printing what is up to you, for me I print the node's word and count

## 3 pq.c

This file consists all the functions for a priority queue data structure, this file will also contain a new data structure called Pair which contains the name of the author and the distance of the words between the author's text and the subject's text

### 3.1 pq_create()

This function creates the priority queue and set the capacity to the one in the parameter. We will need to allocate the memory for 2 things, the priority queue itself and the Pair array. The head and tail pointer of the priority queue is also set to 0

**Pseudocode:**

```
pq_create(capacity)
    allocate memory for priority queue pointer
    if priority queue pointer != NULL:
        priority queue's front = priority queue's tail = 0
        priority queue's capacity = capacity
        allocate memory for the priority queue's Pair array
    return priority queue pointer
```

### 3.2 pq_delete()

This function is the destructor function for the priority queue. We will free the components in the following order: elements in the Pair array, the Pair array, and the priority queue itself. Also set the priority queue pointer to NULL

**Pseudocode:**

```
pq_delete(priority queue pointer)
    for i from 0 to priority queue pointer's top:
        free the node at position i
    free priority queue's pointer Node array
    free priority queue pointer
    priority queue pointer = NULL
```

### 3.3 pq_empty()

This function will check if the current priority queue is empty or not. This can be done by checking if ppriority queue pointer's tail is equal to 0 or not
**Pseudocode:**
pq_empty(priority queue pointer)
    return priority queue pointer's tail == 0

### 3.4 pq_full()

This function will check if the current priority queue is full or not. We can check this by seeing if priority queue pointer's tail is equal to priority queue pointer's capacity or not
**Pseudocode:**
pq_full(priority queue pointer)
    return priority queue pointer's tail == priority queue pointer's capacity

### 3.5 pq_size()

This function will return the current size of the priority queue. For this we can just return the priority queue pointer's tail to determine the size
**Pseudocode:**
pq_size(priority queue pointer)
    return priority queue pointer's tail

### 3.6 enqueue()

This function will push a Pair to the back of the queue (the queue's tail). Since this is a priority queue, this Node will also move to the necessary spot to maintain the heap. To do this we will utilized a while loop and swap the added node with it's parents (the ((i - 1) / 2)-th node) until either the parents is smaller than the node or the index is 0 (aka the root). In addition, we will not enqueue if the queue is already full
**Pseudocode:**
enqueue(priority queue pointer, author, dist)
    if pq_full(priority queue pointer) return false
    i = priority queue pointer's tail
    create a pair data structure using author and dist      priority queue pointer's Node array[i] = pair pointer
    while i != 0 and (priority queue pointer's Pair array[(i - 1) / 2]'s frequency > priority queue pointer's Pair array[i]'s frequency
        swap priority queue pointer's Pair array[(i - 1) / 2] and priority queue pointer's Pair array[i]

i = (i - 1) / 2
priority queue pointer's tail += 1
return true

## 3.7 dequeue()

This function will pop a Pair out of the queue from the front (the queue's top). Since this is a priority queue, there's will be a fix_heap() function to maintain the heap after the element is pop. To pop the element, simply replace the top element with the element from the tail then decrease the priority queue's pointer tail. Note that every time is operation is done, the heap need to be fix with fix_heap(), which is done recursively. In addition, we can't pop the element out if there's no element in the heap

**Pseudocode for dequeue():**

dequeue(priority queue pointer, author pointer, dist pointer)
if pq_empty(priority queue pointer) return false
author pointer = priority queue pointer's Pair array[priority queue pointer's top]'s author
dist pointer = priority queue pointer's Pair array[priority queue pointer's top]'s dist
swap(priority queue pointer's Pair array[priority queue pointer's top], priority queue pointer's Node array[priority queue pointer's tail - 1]
priority queue pointer's tail -= 1
fix_heap(priority queue pointer, 0)

**Pseudocode for fix_heap():**

fix_heap(priority queue pointer, start)
l = start * 2 + 1
r = start * 2 + 2
small = start
if l < priority queue pointer's tail and (priority queue pointer's Pair array[l]'s dist < priority queue pointer's Pair array[start]'s dist:
small = l
if r < priority queue pointer's tail and (priority queue pointer's Pair array[r]'s dist < priority queue pointer's Pair array[small]'s dist
small = r
if small != start
swap priority queue pointer's Pair array[small] and priority queue pointer's Pair array[start]
fix_heap(priority queue pointer, small)

## 3.8 pq_print()

This is the debugging function for the priority queue, the choice to print the priority queue is up to you. For me, I print all the pairs from the priority queue.

# 4 ht.c

This file contains the functions for the Hash Table data structure. In addition, there will be a Hash Table Iterator that will serve to iterate through the hash table

## 4.1 ht_create()

This function is the constructor for the hash table data structure and set the size to the one in the parameter. In addition, we will set the salt to the one defined in salts.h, and we will also allocate memory for the Node array.
**Pseudocode:**

```
ht_create(size)
    allocate memory for the Hash Table object
    if hash table pointer != NULL
        hash table pointer's salt[0] = SALT_HASHTABLE_LO
        hash table pointer's salt[1] = SALT_HASHTABLE_HI
        hash table pointer's size = size
        allocate memory for hash table pointer's slots
    return hash table pointer
```

## 4.2 ht_delete()

This is a destructor function for the hash table. We will need to free the following components in order: The nodes in the Node array, the Node array, the hash table
**Pseudocode:**

```
ht_delete(hash table pointer)
    for i from 0 to hash table pointer's top:
        free the node at position i
    free hash table's pointer Node array
    free hash table pointer
    hash table pointer = NULL
```

## 4.3 ht_size()

This function simply returns the size of the hash table. Simply returning the hash table pointer's size will do the trick
**Pseudocode:**

```
ht_size(hash table pointer)
    hash table pointer's size
```

## 4.4 ht_lookup()

This function will search for the node in that hash table that has the same word as the one in the parameter. If the node is in the hash table, the function will return the node otherwise a NULL will be returned. In order to reduce the number of operations, we will need to do 2 things: use the hash function hash() from speck.c in order to get a good idea where to look, and stop the function once we hit a NULL since we assume that we will definitely not find the node at that point.

**Pseudocode:**

```
ht_lookup(hash table pointer, word)
    k = hash(hash table pointer's salt, word) mod (hash table pointer's size)
    for i from 0 to hash table pointer's size:
        k = (k + i) mod (hash table pointer's size)
        if hash table's pointer Node array[k] == NULL: break
        if hash table's pointer Node array[k]'s word == word:
            return hash table's pointer Node array[k]
    return NULL
```

## 4.5 ht_insert()

This function with insert a node into the hash table. If the node is already in the hash table, we simply increment the count by 1. To make sure the Node is in the hash table or not we're going to use ht_lookup() to check. In addition when inserting the node, we will use the hash() from speck.c to get a good idea where to insert our node, we will look from that starting point and loop forward until we find a NULL which is an empty spot **Pseudocode:**

```
ht_insert(hash table pointer, word)
    if ht_lookup(hash table pointer, word) != NULL:
        inc = ht_lookup(hash table pointer, word)
        inc's count += 1
        return inc
    key = hash(hash table pointer's salt, word) mod (hash table pointer's size)
    for i from 0 to hash table pointer's size:
        key = (key + i) mod (hash table pointer's size)
        if hash table pointer's Node array[k] == NULL:
            hash table pointer's Node array[k] = node_create(word)
            return hash table pointer's Node array[k]
    return NULL
```

## 4.6 ht_print()

This function is debugging function for the hash table, the choice to print in this function is up to you, for me I print everything in the hash table's node array

## 4.7 hti_create()

This function create a hash table iterator, simply allocate memory for the hash table iterator container and assign the pointer to default value of zero and the hash table to iterate to the one in the parameter

**Pseudocode:**

hti_create(hash table pointer)
    allocate memory for the hash table iterator container
    if hash table iterator pointer != NULL:
        hash table iterator pointer's table = hash table pointer
        hash table iterator pointer's slot = 0
    return hash table iterator pointer

## 4.8 hti_delete()

This function is the destructor function for the hash table iterator. Simply free the memory for the hash table iterator container will do the trick. Note that you shouldn't free the hash table inside hash table iterator container since we might need to use the hash table later

**Pseudocode:**

hti_delete(hash table iterator pointer)
    free the memory for the hash table iterator pointer
    hash table iterator pointer = NULL

## 4.9 ht_iter()

This function iterates through the hash table until the iterator reaches a node. If the iterator is at the end of the hash table simply return the NULL.

**Pseudocode:**

ht_iter(hash table iterator pointer)
    while hash table iterator pointer's table's Node array[hash table iterator pointer's slot] == NULL and hash table iterator pointer's slot ¡ ht_size(hash table iterator pointer's table):
        hash table iterator pointer's slot += 1
    if hash table iterator pointer's slot == ht_size(hash table iterator pointer's table):
        return NULL
    hash table iterator pointer's slot += 1
    return while hash table iterator pointer's table's Node array[hash table iterator pointer's slot - 1]

# 5 bv.c

This file contains the functions for the Bit Vector data structure, this data structure will be necessary for the Bloom Filter

## 5.1 bv_create()

This function will construct a Bit Vector. We will need to allocate memory for the following components: the bit vector itself, and the 8 bit unsigned integers array. Additionally, the size of the bit vector is set to the one in the parameter
**Pseudocode:**

```
bv_create(length)
    allocate memory for the bit vector object
    if bit vector pointer != NULL:
        bit vector pointer's length = length
        allocate memory for the bit vector pointer's unsigned int array, size length
    return bit vector pointer
```

## 5.2 bv_delete()

This function is the destructor function for the bit vector. Free the following components in order: The unsigned int array then the bit vector itself. Remember to set the bit vector pointer to NULL
**Pseudocode:**

```
bv_delete(bit vector pointer)
    free the unsigned int array
    free the bit vector pointer
    bit vector pointer = NULL
```

## 5.3 bv_length()

This function returns the bit vector's length. Simply return the bit vector pointer's length will do the trick
**Pseudocode:**

```
bv_length(bit vector pointer)
    return bit vector pointer's length
```

## 5.4 bv_set_bit()

This function simply set the bit at the specified position in the parameter to 1. We will need to use bit-wise operations to solve this problem. We will also need to figure out what byte we are modifying and what bit in that byte we need to set to 1. In addition, we will not set the bit if the position is out of the bit

vector's bounds
**Pseudocode:**

```
bv_set_bit(bit vector pointer, i)
    if i >= bv_length(bit vector pointer) * 8: return false
    pos = i / 8
    val = i mod 8
    bit vector pointer's unsigned int array[pos] |= (1 << val)
    return true
```

## 5.5 bv_clr_bit()

This function will set the bit at the specified position in the parameter to 0. We will also use bit-wise operations like in bv_set_bit. We will also need to know which byte and which bit in the byte to change. We won't be setting the bit to 0 if the position is out of the vector's bounds
**Pseudocode:**

```
bv_clr_bit(bit vector pointer, i)
    if i >= bv_length(bit vector pointer) * 8: return false
    pos = i / 8
    val = i mod 8
    bit vector pointer's unsigned int array[pos] &= NOT(1 << val)
    return true
```

## 5.6 bv_get_bit()

This function will get the state of the bit at the position in the parameter. Similar to bv_set_bit() and bv_clr_bit(), this function will utilize the bit-wise operation. The function needs to know which byte and which bit in the byte to change, the function will also return false if the position is not in the vector's bounds
**Pseudocode:**

```
bv_get_bit(bit vector pointer, i)
    if i >= bv_length(bit vector pointer) * 8: return false
    pos = i / 8
    val = i mod 8
    return (bit vector pointer's unsigned int array[pos] & (1 << val)) > 0
```

## 5.7 bv_print()

This function is just a debugging function for the Bit vector. The choice to print in this function is up to you. For me I print the unsigned int array

# 6 bf.c

This function contains the function for the Bloom Filter, which will be used in the Text data structure.

## 6.1 bf_create()

This function will create a new Bloom Filter object. For the salts, we will need to set the salts to ones specified in salts.h. For the bit vector, we will create one with the size specified in the parameter using bv_create()
**Pseudocode:**

```
bf_create(size)
    allocate memory for the Bloom Filter object
    if bloom filter pointer != NULL:
        bloom filter pointer's primary[0] = SALT_PRIMARY_LO
        bloom filter pointer's primary[1] = SALT_PRIMARY_HI
        bloom filter pointer's secondary[0] = SALT_SECONDARY_LO
        bloom filter pointer's secondary[0] = SALT_SECONDARY_HI
        bloom filter pointer's tertiary[0] = SALT_TERTIARY_LO
        bloom filter pointer's tertiary[0] = SALT_TERTIARY_HI
        bloom filter pointer's bit vector = bv_create(size)
    return bloom filter pointer
```

## 6.2 bf_delete()

This function is the destructor function for the Bloom Filter. Simply freeing the memory in the following order: the bit vector, the bloom filter. Make sure to set the bloom filter pointer is set to NULL
**Pseudocode:**

```
bf_delete(bloom filter pointer)
    bv_delete(bloom filter pointer's bit vector)
    free the bloom filter
    bloom filter pointer = NULL
```

## 6.3 bf_size()

This function return the bloom filter's size (more specifically the bit vector's length), simply use bv_length() will do the trick
**Pseudocode:**

```
bf_size(bloom filter pointer)
    return bv_length(bloom filter pointer's bit vector)
```

### 6.4 bf_insert()

This function will insert the word in the parameter into the bit vector, we will hash the word into 3 different number using the 3 different salts in the bloom filter. We will utilize bv_set_bit() to accomplish this

**Pseudocode:**

bf_insert(bloom filter pointer, word)

key1 = hash(bloom filter pointer's primary salt, word) mod bf_size(bloom filter pointer)

key2 = hash(bloom filter pointer's secondary salt, word) mod bf_size(bloom filter pointer)

key3 = hash(bloom filter pointer's tertiary salt, word) mod bf_size(bloom filter pointer)

bv_set_bit(bloom filter pointer's bit vector, key1)

bv_set_bit(bloom filter pointer's bit vector, key2)

bv_set_bit(bloom filter pointer's bit vector, key3)

### 6.5 bf_probe()

This function checks if the node is possibly in the hash table or not. To check if the node is possibly in the hash table or not, simply hash the word using the 3 salts then get the bits from the positions. If all 3 bits are set to 1 then the node is possible in the hash table.

**Pseudocode:**

bf_probe(bloom filter pointer, word)

key1 = hash(bloom filter pointer's primary salt, word) mod bf_size(bloom filter pointer)

key2 = hash(bloom filter pointer's secondary salt, word) mod bf_size(bloom filter pointer)

key1 = hash(bloom filter pointer's tertiary salt, word) mod bf_size(bloom filter pointer)

return bv_get_bit(bloom filter pointer's bit vector, key1) and bv_get_bit(bloom filter pointer's bit vector, key2) and bv_get_bit(bloom filter pointer's bit vector, key3)

### 6.6 bf_print()

This function is the debugging function for the Bloom Filter. The choice to print in this function is up to you. For me, I just print the bit vector

## 7 text.c

This will contain the function of the main data structure we will be using to compare the texts between the subject and the authors.

## 7.1 text_create()

This function creates a Text object. Since this Text object consists of both the Bloom Filter and the Hash Table, we will create them using bv_create() and ht_create() respectively. In addition we will be using a regex module to parse out all the words and insert them into the hash table and bloom filter. A special thing to note about this insertion of words is that these words will also be filter by the Text noise simultaneously, we check if the word is in the noise Text by using text_contains() which will be discussed later in this section

**Pseudocode:**

```
text_create(infile, noise)
    allocate memory for the Text container
    if text pointer != NULL:
        regex re
        if compiling regex failed:
            return NULL
        text pointer's hash table = ht_create(524288) //default size of hash table
        text pointer's bloom filter = bf_create(2097152) // default size of bloom
filter
        text pointer's word_count = 0
        while (word = parsing text) != NULL:
        change the word to lowercase
        if (noise != NULL and !text_contains(noise, word)) or noise == NULL:
            text pointer's word_count += 1
            ht_insert(text pointer's hash table, word)
            bf_insert(text pointer's bloom filter, word)
    return text pointer
```

## 7.2 text_delete()

This is the destructor function for the Text data structure. We will need to delete both the hash table and the bloom filter before we free the Text container. **Pseudocode:**

```
text_delete(text pointer)
    ht_delete(text pointer's hash table)
    bf_delete(text pointer's bloom filter)
    free text pointer
    text pointer = NULL
```

## 7.3 text_dist()

This function calculates the distance between the 2 texts based on the type of metric. We will calculate the distance using 1 of the 3 metrics: euclidean, manhattan, and cosine. To get through all the text in the hash table, a hash table iterator is needed

**Pseudocode for distances():**

```
distances(d1, d2, metric)

    if metric == EUCLIDIAN: return (d1 - d2) * (d1 - d2)
    else if metric == MANHATTAN: return abs(d1 - d2)
    else d1 * d2
```

**Pseudocode for text_dist():**

```
text_dist(text1, text2, metric)
    hti1 = hti_create(text1 pointer's hash table)
    hti2 = hti_create(text2 pointer's hash table)
    f1 = 0.0
    f2 = 0.0
    total = 0.0
    while (sub = ht_iter(hti1)) != NULL:
        f1 = text_frequency(text1, sub pointer's word)
        f2 = text_frequency(text2, sub pointer's word)
        total += distances(f1, f2, metric)
    while (sub = ht_iter(hti2)) != NULL:
        if text_contains(text1, sub pointer's word) and text_contains(text2, sub
pointer's word): continue
        f1 = text_frequency(text1, sub pointer's word)
        f2 = text_frequency(text2, sub pointer's word)
        total += distances(f1, f2, metric)
    if metric == EUCLIDIAN: total = sqrt(total)
    if metric == COSINE: total = 1 - total
    return total
```

## 7.4 text_frequency()

This function will return the frequency (i.e count) of a certain word in the text's hash table. If the word is not in that hash table, the answer is 0. To reduce the number of operations, we will first check the bloom filter by using bf_probe. If the bf_probe() is false we know the word is not in the hash table, therefore we should output 0, otherwise we should check again using ht_lookup() to make sure the word is really in the hash table.

**Pseudocode:**

```
text_frequency(text, word)
    if bf_probe(text pointer's bloom filter, word):
        if (t = ht_lookup(text pointer's hash table, word)) == NULL: return 0.0
        else return (t pointer's count) / (text pointer's word_count)
    return 0.0
```

### 7.5 text_contains()

This function will return a true if the word is in the text (i.e in the hash table), and false otherwise. Since if text_frequency outputs a 0, the word is not present in the hash table. We can utilize this to determine whether the word is in the text

**Pseudocode:**

```
text_contains(text, word)
    text_frequency(text, word) != 0
```

### 7.6 text_print()

This function is the debugging function for the Text data structure. It's up to you to decide what to print in this function. For me, I choose to print the hash table

## 8 identify.c

This function contains the main() function that will utilized all the function in the previous files mentioned in order to perform an author identification. For the noise filter, since there's is an option to limit how many words is allowed. We will be storing them into a temporary file and read in the inputs from there. Other than that, the pseudocode below will describe the entire operation in detail of how to perform an author identification.

**Pseudocode:**

```
main()
    data = open file lib.db
    noise = open file noise.txt
    check if the files open properly
    matches = 5
    filter = 100
    metric = EUCLIDEAN
    read the commands
        if command is d:
            close the current file
            data = open file in the argument
            check if the file is open properly
        if command is n:
            close the current file
            noise = open file in the argument
            check if the file is open properly
        if command is k:
            matches = read unsigned int in the argument
        if command is l:
```

```
            filter = read unsigned int in the argument
        if command is e:
            metric = EUCLIDEAN
        if command is m:
            metric = MANHATTAN
        if command is c:
            metric = COSINE
        if command is h:
            display synopsis
    limitnoise = open a temporary file
    allocate memory for the string inserts
    for i from 0 to filter:
        inserts get input from noise file
        print inserts into the limitnoise
    reset the file pointer for limitnoise
    Text ban pointer = text_create(limitnoise, NULL) //this is the noise itself
    Text subject pointer = text_create(stdin, limitnoise)
    Priority Queue pq pointer = pq_create(cases)
    allocate memory for the string author
    allocate memory for the string path
    cases = 0
    cnt = 0
    read the number in the first line of data file
    read the string and put it in the path string //This is to prevent the fgets
from outputting the empty string
    for i from 0 to cases * 2:
        if i mod 2 == 1:
            read the string and put it in path
            delete the newline that fgets() left behind
            authors = open the path file
            check if the file is open properly
            Text source pointer = text_create(authors, ban)
            enqueue(pq, author, text_dist(subject, source, metric))
            text_delete(source pointer)
            cnt++
            close the path file
        else:
            read the string and put it in author
            delete the newline that fgets() left behind
    loops = minimum between matches and cnt
    print("Top k, metric metric, noise limit filter)
    for i from 1 to loops + 1:
        outputstring
        dist
        dequeue(pq, outputstring, dist)
        print("i) outputstring dist) //the float is set to 15 decimal places
```

pq_delete(pq pointer)
text_delete(ban pointer)
text_delete(subject pointer)
free the author string
free the path string
free the inserts string
close the data file
close the noise file
close the limitnoise file