Assignment Seven Design.pdf

By James Yim

Description of Program:

This program will be able to take in a sample of text, analyze it, and then identify the top ‘k’ likely authors that could have authored the text.

Files to be included in Assignment 7

- bf.h
    - Defines the interface for the Bloom Filter ADT
- bf.c
    - Contains the implementation of the Bloom Filter
- bv.h
    - Defines the interface for the bit vector ADT
- bv.c
    - Contains the implementation of the bit vector
- ht.h
    - Defines the interface of the hash table
- ht.c
    - Contains the implementation of the hash table
- identify.c
    - Contains main() and implementation of the author identification program
- metric.h
    - Defines the distance metrics and their respective names in an array
- node.h
    - Defines the interface for the node ADT
- node.c
    - Contains the implementation of the node ADT
- parser.h
    - Defines the interface for the regex module
- parser.c
    - Contains the implementation for the regex module
-

- pq.h
  - Defines the interface for the priority queue
- pq.c
  - Contains the implementation for the priority queue
- salts.h
  - Defines the salts that will be used in the bloom filter. It also defines the salt used by the hash table
- speck.h
  - Defines the interface for the hash function using the SPECK cipher
- speck.c
  - Contains the implementation of the hash function using the SPECK cipher
- text.h
  - Defines the interface for the text ADT
- text.c
  - Contains the implementation for the text ADT
- Makefile
  - This will help compile and run the program with the necessary files.
- README.md
  - This describes how to use the program. It will explain command line options that the program accepts. It will also note bugs and errors that are in the program.
- DESIGN.pdf
  - This pdf will show the process of how to create this program. It will include pseudocode and it will credit sources where information was learned.
- WRITEUP.pdf
  - This will discuss what was observed about the program's behavior. We will talk about different sizes of text, and the different metrics.

Pseudocode and Design

Hash Table

Creating the Hash Table

```
HashTable *ht_create(size of hash table)
{
    allocate space for the hash container
    if (the hash container was not successfully allocated)
    {
        free the container
        send error message that container allocation failed
    }
    allocate space for the array of Nodes
    if (the array of Nodes failed to allocate)
    {
        free the array
        free the container
        send error message that the array failed to allocate
    }
    set the size of the hash table to the input size
    set the value for salts
    return the hash table
}
```

Deleting the Hash Table

```
void ht_delete(HashTable)
{
    free the array of Nodes in the hash table
    free the hash table container
    set to NULL
}
```

Getting the size of the Hash Table

```
uint32_t ht_size (HashTable)
{
    return the number of slots that it can index up to (set in ht_create)
}
```

Looking up an entry

```
Node *ht_lookup(Hashtable, Word)
{
    for (every slot in the hash table)
    {
        if (the slot contains the node with the word)
        {
            return the pointer to the node
        }
    }
    if (the word was not found)
    {
        return NULL;
    }
}
```

Inserting into the Hash Table

```
Node *ht_insert(HashTable, word)
{
    if (we lookup the word using ht_lookup and it returns a pointer)
    {
        use the pointer and increment the count by one
        return the pointer returned by ht_lookup
    }
    else
    {
        return NULL;
    }
}
```

Creating a Hash Table iterator

```
struct HashTableIterator
{
    HashTable *table (this is the hash table to iterate through)
    uint32_t slot (this is the current slot the iterator is on)
}

HashTableIterator *hti_create(HashTable *ht)
{
    allocate space for the HashTableIterator
    set slot to 0
    return the HashTableIterator
}
```

Deleting the HashTableIterator

```
void hti_delete(HashTableIterator)
{
    free the HashTableIterator
}
```

Iterating the hash table

```
Node *ht_iter(HashTableIterator)
{
    for (size of the hash table size)
    {
        if (we iterate through the slots and we find a filled slot)
        {
            return pointer to that slot
        }
    }
    (if we finish looping through the entire hash table)
    return NULL
}
```

Nodes

Creating Nodes

```
Node node_create(word)
{
        allocate memory for node container
        if (node was successfully allocated)
        {
                set the word variable in node to input word
                set the count variable in node to 1 (we set to one because when we create we already put the word in so it starts at 1)
        }
        return the node
}
```

Deleting Nodes

```
node_delete(Node)
{
        set the node to NULL;
        free the node
}
```

Bitvector

Creating Bitvectors

```
bv_create(length)
{
        allocate memory for the bitvector container. use calloc cause we need 0's
        if (bitvector container was successfully created)
        {
                set bitvector length to input length
                allocate memory for vector in bitvector.
                if (vector in bitvector was successfully allocated)
                {
                        return the bit vector
                }
                free (bit vector) //This is if it was unsuccessfully allocated
        }
        return NULL
}
```

Deleting bit vector

```
bv_delete(bit vector)
{
    if (bit vector exists)
    {
        free the vector
        free the bit vector container
    }
}
```

Getting bit vector length

```
bv_length(bit vector)
{
    return the length of bv set in create
}
```

Setting bit

```
bv_set_bit(bitVector, index)
{
    if (check if the index is in range of the bit vector)
    {
        if out of range return false
    }

    perform a bitwise operation to set the bit at the index to 1
    return true
}
```

Clearing bit

```
bv_clr_bit(bitVector, index)
{
    if (check if the index is in range of the bit vector)
    {
        if out of range return false
    }

    perform a bitwise operation to set the bit at the index to 0
    return true
}
```

Getting bit

```
bv_get_bit(bitVector, index)
{
    if (check if the index is in range of the bit vector)
    {
        if out of range return false
    }

    perform a bitwise operation to set the bit at the index
    if (bitwise operation equals 1)
    {
        return true
    }
    return false
}
```

Text

Creating a Text

```
text_create(infile, noiseText)
{
    allocate the text container
    check if text container was successfully allocated and if not return null
    allocate the text’s hash table and check if it was successfully created
    allocate the text’s bloom filter and check if it was successfully created

    initalize regex and check if it was successful
    while (there is still words in the file)
    {
        make word lowercase
        if (the noiseText is null)
        {
            insert the word into the bloom filter and the hash table
            increment word count of text
        }
        else (there is a word text)
        {
            check if the current word is in the noise file. if it isnt
            {
                insert the word into the bloom filter and the hash table
                increment the word count of text
            }
        }
    }
    free the regex
    return text
}
```

Deleting Text

```
text_delete(text)
{
    if (text exists)
    {
        delete the hash table
        delete the bloom filter
        free the text
    }
}
```

Getting Text Distance

```
text_dist(first text, second text, metric)
{
    loop through each text’s hash table
    {
        if (word exists in both texts’ hash tables)
        {
            get the frequency of the word in each text
            if (metric is manhattan)
            {
                distance_between the two texts += the absolute val of text1_word
freq - text2_word freq
            }
            if (metric is euclidean)
            {
                distance_between += (text1_word - text2_word);
                distance_between += distance_between * distance_between;

            }
            if (metric is cosine)
            {
                distance_between += (text1_word - text2_word)
                *and then at the end of loop we do 1 - distance_between*
            }
        }
    }
    return the distance between
}
```

Getting Text Frequency

```
text_frequency(Text, word)
{
    if there is a node with the word in the hash table
    {
        return current_node’s count / total number of words in text as a double
    }
}
```

Seeing if the text contains a word

```
text_contains(text, word)
{
    use bloom filter to initially check. if bloom filter says its probably in
    {
        loop through the hash table for the word
        {
            if its in return true else return false
        }
    }
    return false if bloom filter says its not in
}
```

Notes about code

The identify.c was not finished in this current version
We use bloom filter in order to cut down the amount of time it takes to search

Credit

Euegene's section helped with how to get started
The discord helped with questions I had pertaining to segmentation faults